DOCUMENTATION

ASSIGNMENT 1

STUDENT NAME: Bălan Ionela-Loredana
GROUP: 30223

CONTENTS

1. Assignment Objective

1.1. Main Objective

The main objective of the project is to develop a polynomial calculator in Java for performing addition, subtraction, multiplication, division, derivation and integration operations.

1.2. Sub-objectives

Sub-objectives	Description	Chapter
Analyze the problem and identify requirements.	Identifying use-cases, functional and non-functional requirements	2
Design the polynomial calculator.	Division into sub-systems/packages, classes, sub-classes, routines.	3
Implement the polynomial calculator.	Each class and important method implementation will be described.	4
Test the polynomial calculator.	Test scenarios for implementation methods will be presented.	5

2. Problem Analysis, Modeling, Scenarios, Use Cases

2.1. Requirements

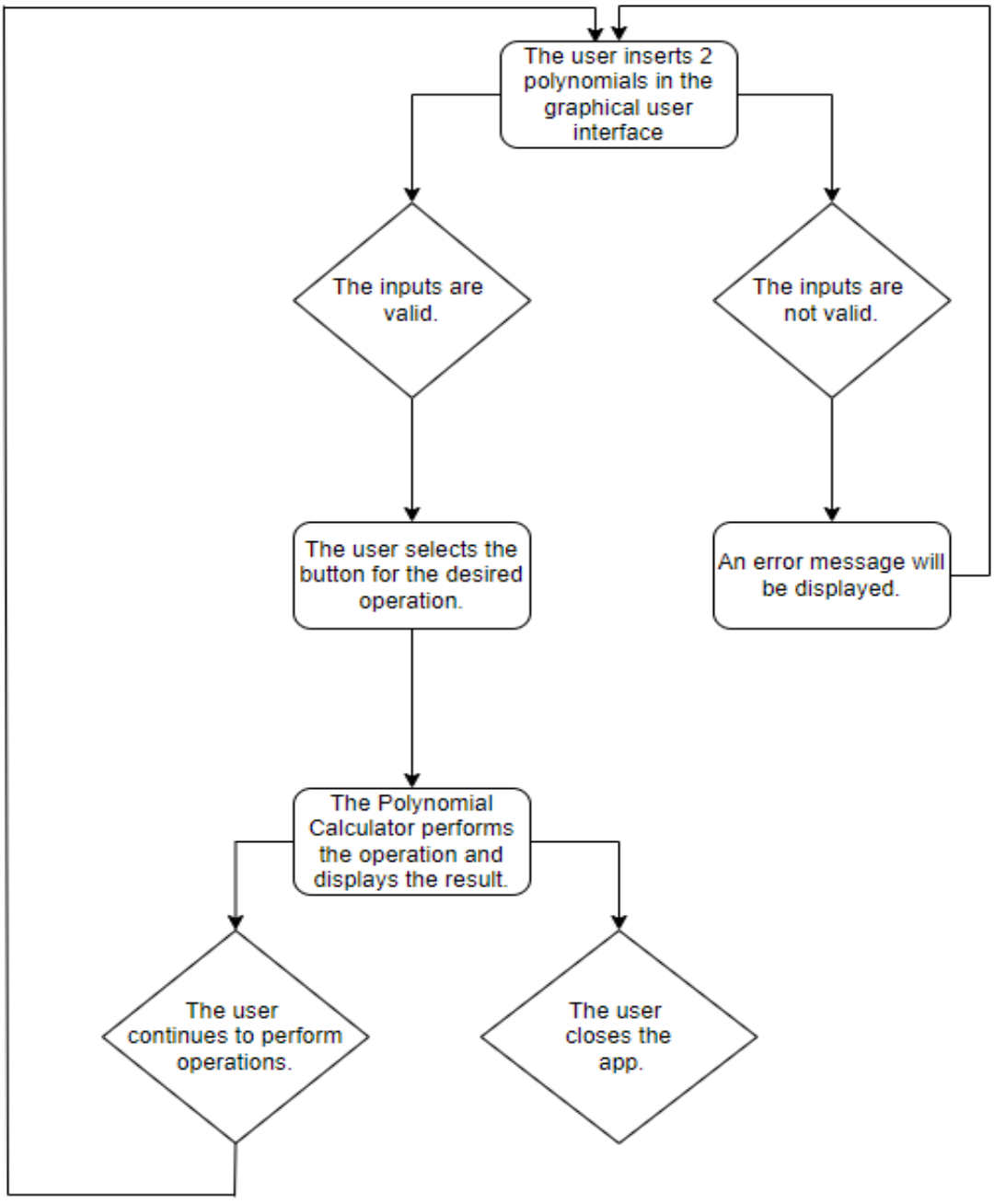
2.1.1. Functional requirements:

- The polynomial calculator should allow users to insert polynomials
- The polynomial calculator should allow users to delete the inserted polynomials
- The polynomial calculator should allow users to select the mathematical operation.
- The polynomial calculator should generate errors in case of invalid polynomial input.
- The polynomial calculator should allow users to perform successive operations.

2.1.2. Non-functional requirements:

- The polynomial calculator should be intuitive and easy to use by the user.
- The calculation of the result should be fast.
- The interface should provide error-handling messages for polynomial input.

2.2. ***Use-cases flowchart***



2.3. Example scenarios

1. Use Case: Add polynomials

Primary Actor: user

Main Success Scenario:

1. The user inserts the two polynomials in the graphical user interface.
2. The user selects the “addition” operation
3. The polynomial calculator performs the addition of the two polynomials and displays the result.

Alternative Sequence: Incorrect polynomials

- The user inserts incorrect polynomials.
- The scenario returns to the first step.

2. Use Case: Divide polynomials

Primary Actor: User

Main Success Scenario:

1. The user inserts the two polynomials in the graphical user interface.
2. The user selects the "division" operation.
3. The polynomial calculator computes the division of the polynomials.
4. The calculator displays the quotient and the remainder, showing the result of the input polynomial.

Alternative Sequence: Second Polynomial is 0

- The user tries to divide by 0
- The user is warned about trying to perform division by 0 with an error message
- The scenario returns to the first step

2.4. Modeling

Polynomial Class - This class represents the abstractization of a polynomial and serves as a fundamental entity in the system. Attributes of a polynomial include its constituent monomials.

Monomial Class - Monomials are the building blocks of polynomials, each representing a term with a coefficient and a degree. The Monomial class include methods for manipulating monomials, such as comparing.

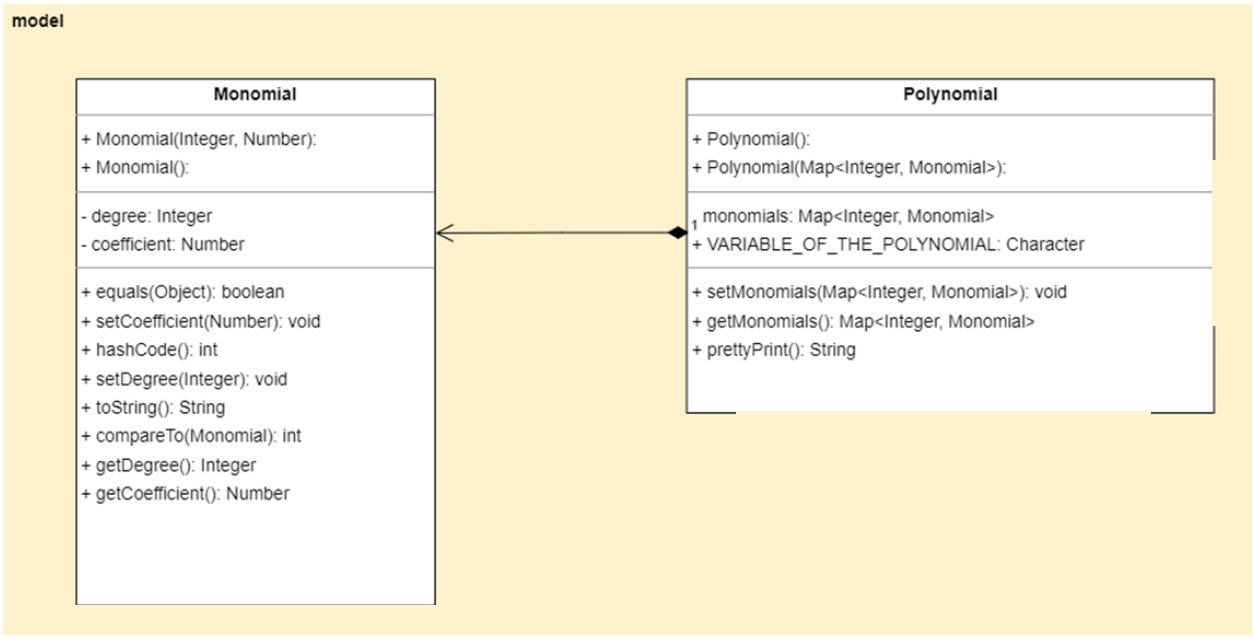
Operations Class - This class contains methods for performing addition, subtraction, multiplication, division, differentiation, and integration. Each operation is modeled as a function

that operates on polynomials and involve iterating through the monomials, combining or modifying them according to mathematical rules.

StringConversionToPolynomials – This class is used to convert the String input from the graphical interface into a polynomial.

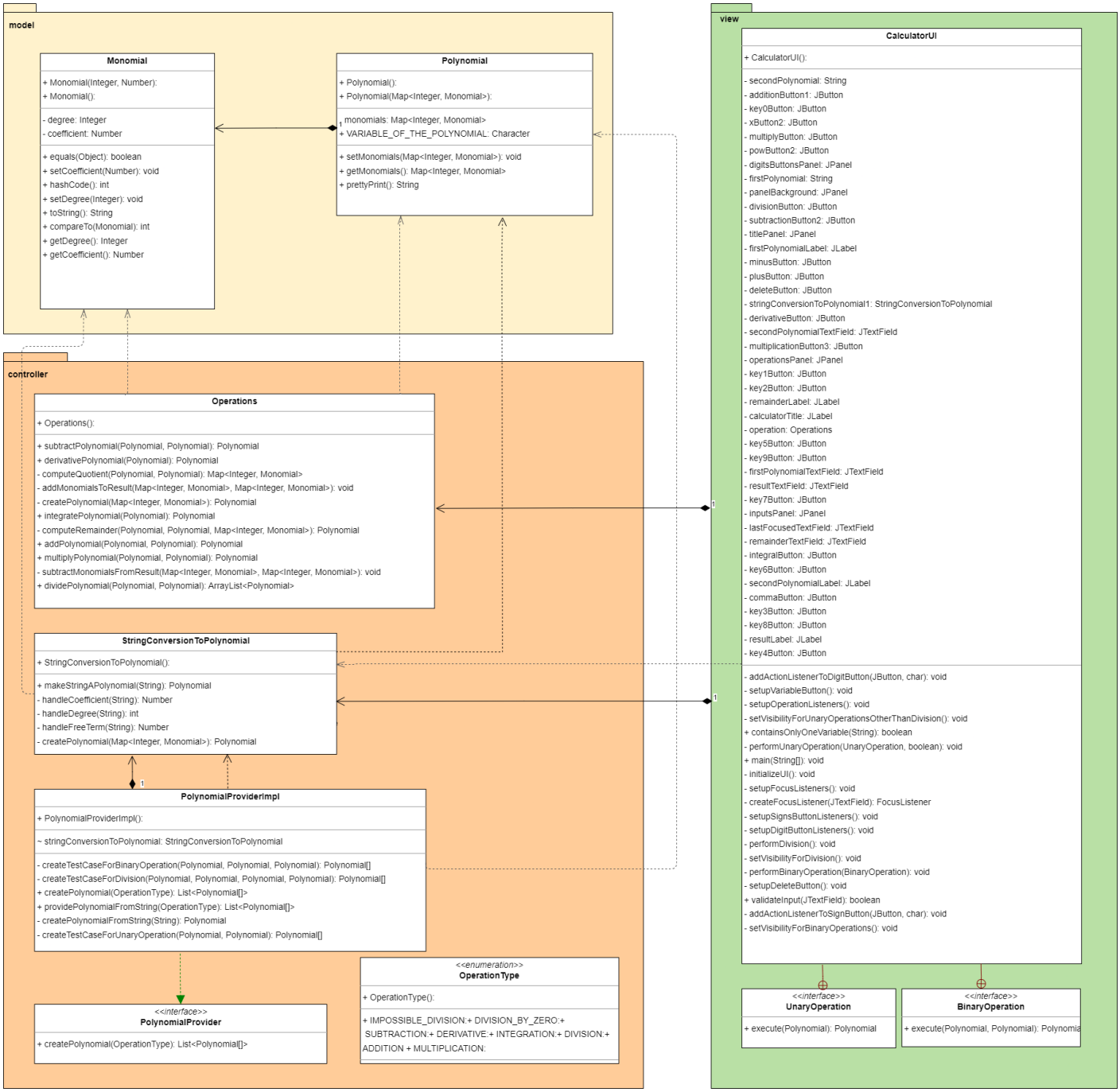
Relationships - The Polynomial class has a relationship with the Monomial class, as polynomials are composed of monomials. Operations on polynomials involve interactions between their constituent monomials, reflecting the relationship between terms within a polynomial expression.

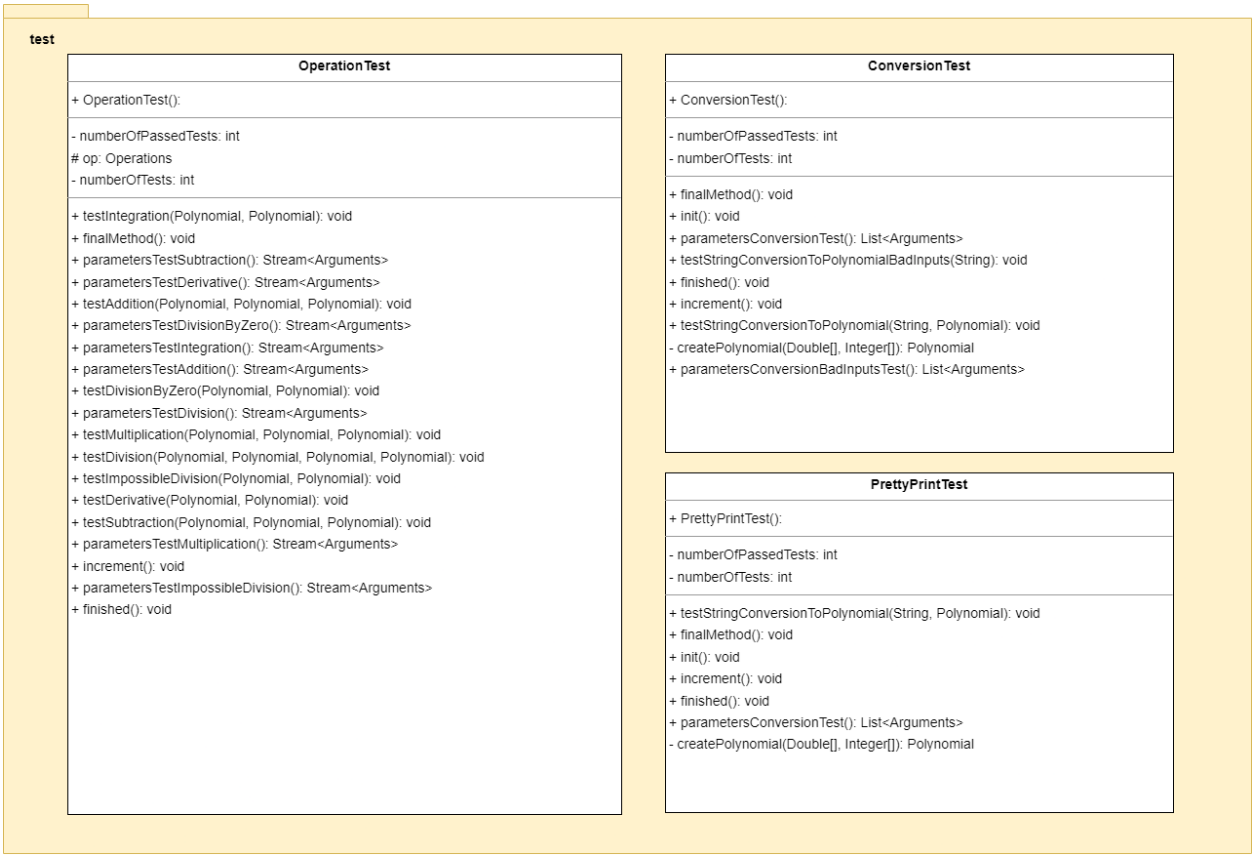
3.



Design

UML package and class diagrams





Used data structures

Lists

```
List<Polynomial[]> listTestCases = new ArrayList<>();
```

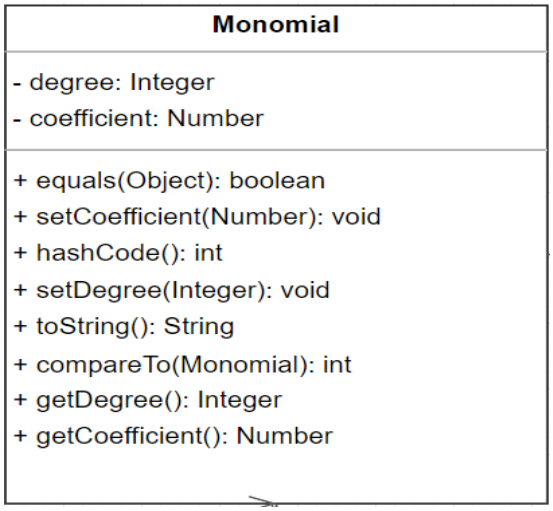
Maps, TreeMaps

```
Map<Integer, Monomial> monomialMap = new TreeMap<>(Collections.reverseOrder());

Map<Integer, Monomial> monomialMapCTest = new LinkedHashMap<>();
```

4. Implementation

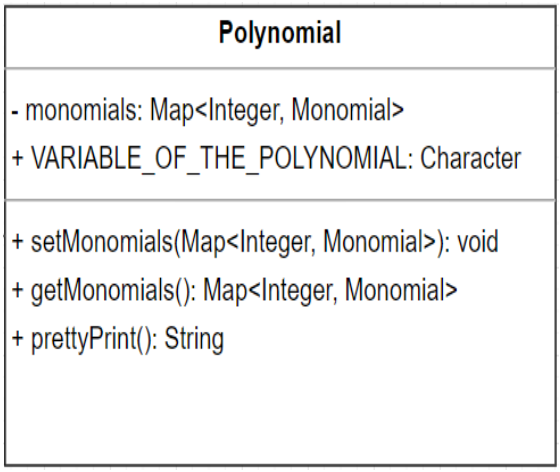
1. Class “Monomial”

- is a model class with degree and coefficient fields used to describe a monomial. It has basic methods like setters, getters, etc.



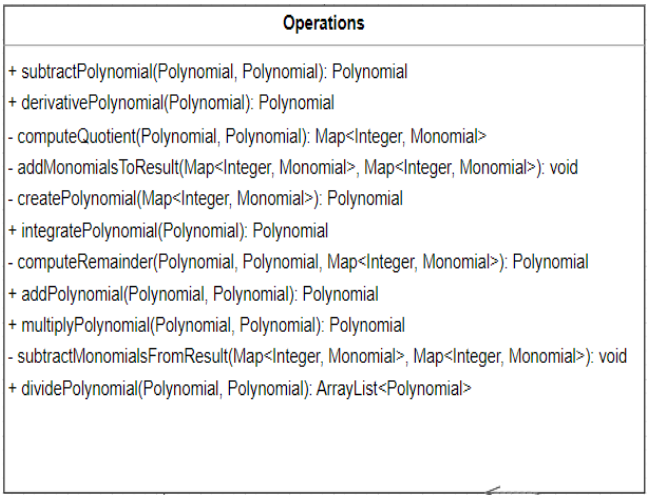
2. Class ”Polynomial”

- is a model class that contains a map of monomials used to describe a Polynomial and a constant field that represents the variable of the polynomial. It has basic methods and a method used to print the Polynomial as a string in a customized way.



3. Class “Operations”

- this class contains the operations methods and their helper methods. It also has a method that is used to create polynomials using a map of monomials as input.



```
private Polynomial createPolynomial(Map<Integer, Monomial> monomials) {
        Polynomial result = new Polynomial();
        result.setMonomials(monomials);
        return result;
    }
```

Addition – this method takes two polynomials as input. It begins by creating an empty map of monomials. Then, it adds the monomials of the first polynomial to the result empty map. Next, it iterates over the monomials of the second polynomial. For each monomial, it checks if there is already a monomial with the same degree in the result map. If found, it adds the coefficients of these monomials together. If no monomial with the same degree exists, it adds the new monomial to the result map as it is. Finally, it creates a new polynomial using the resulting map of monomials by calling the createPolynomial method, and returns the new polynomial as the result of the addition operation.

```
private void addMonomialsToResult(Map<Integer,Monomial> monomials, Map<Integer,Monomial>
resultMap) {
     for (Monomial monomial : monomials.values()) {
        int degree = monomial.getDegree();
        Number coefficient = monomial.getCoefficient();
        Monomial existingMonomial=resultMap.getOrDefault(degree,new Monomial(degree, 0));
        double existingCoefficient = existingMonomial.getCoefficient().doubleValue();
        double newCoefficient = coefficient.doubleValue();
        double sum = existingCoefficient + newCoefficient;
            existingMonomial.setCoefficient(sum);
            if (sum != 0.0) {
                resultMap.put(degree, existingMonomial);
            } else {
                resultMap.remove(degree);
            }
        }
    }
```

```
public Polynomial addPolynomial(Polynomial p1, Polynomial p2) {
        Map<Integer, Monomial> mapRes = new
TreeMap<>(Collections.reverseOrder());
        addMonomialsToResult(p1.getMonomials(), mapRes);
        addMonomialsToResult(p2.getMonomials(), mapRes);
        if (mapRes.isEmpty()) {
            mapRes.put(0, new Monomial(0, 0.0));
        }
        return createPolynomial(mapRes);
    }
```

Subtraction – similarly to the addition method this method subtracts the monomials with common degree or adds the unique degree monomials to the resulting map, it then creates the polynomial using the createPolynomial method.

Derivative – this method iterates through the map of monomials in the polynomial and multiplies their coefficient with their degree and decrement the degree. The resulting monomial is added to the result map used to create the result of the polynomials operation.

```
public Polynomial derivativePolynomial(Polynomial p1) {
        Map<Integer, Monomial> mapRes = new TreeMap<>(Collections.reverseOrder());
        for (Monomial monomial : p1.getMonomials().values()) {
            int degree = monomial.getDegree();
            Number coefficient = monomial.getCoefficient();
            double coef = coefficient.doubleValue();
            Monomial derivatedMonomial = new Monomial();
            derivatedMonomial.setCoefficient(coef * degree);
            if (degree >= 1) {
                derivatedMonomial.setDegree(degree - 1);
            } else {
                derivatedMonomial.setDegree(0);
            }
            if (derivatedMonomial.getCoefficient().doubleValue() != 0.0) {
                mapRes.put(degree, derivatedMonomial);
            }
        }
        if (mapRes.isEmpty()) {
            mapRes.put(0, new Monomial(0, 0.0));
        }
        return createPolynomial(mapRes);
    }
```

Integration - this method iterates through the map of monomials in the polynomial and divides their coefficient with their degree and increment the degree. The resulting monomial is added to the result map used to create the result polynomial.

```
public Polynomial integratePolynomial(Polynomial p1) {
    Map<Integer, Monomial> mapRes = new TreeMap<>(Collections.reverseOrder());
    for (Monomial monomial : p1.getMonomials().values()) {
        int degree = monomial.getDegree();
        Number coefficient = monomial.getCoefficient();
        DecimalFormat df = new DecimalFormat("#.####");
        Monomial integratedMonomial = new Monomial();
        double result = (coefficient).doubleValue() / (degree + 1);
        integratedMonomial.setCoefficient(Double.parseDouble(df.format(result)));
        integratedMonomial.setDegree(degree + 1);
        if (integratedMonomial.getCoefficient().doubleValue() != 0.0) {
            mapRes.put(degree, integratedMonomial);            }
    }
    return createPolynomial(mapRes);
}
```

Multiplication – this method iterates through both maps of monomials of the given polynomials, multiplies the coefficients of each monomial in the first polynomial's map with the coefficients of each monomial of second
polynomial's map and adds their degrees, it also make sure at a moment if there were already monomials with the resulting degree in the result map by that moment, if found, their coefficients are added.

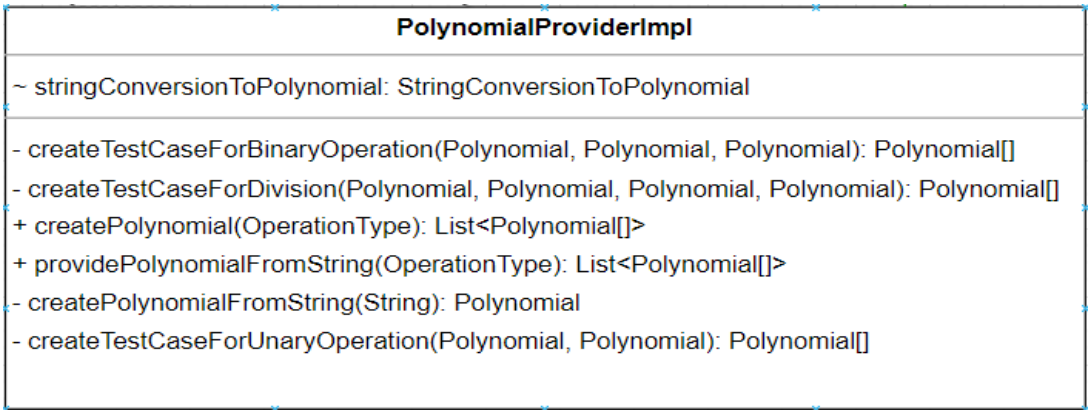
```
public Polynomial multiplyPolynomial(Polynomial p1, Polynomial p2) {
    Map<Integer, Monomial> mapRes = new TreeMap<>(Collections.reverseOrder());
    for (Monomial monomialP1 : p1.getMonomials().values()) {
        int degreeP1 = monomialP1.getDegree();
        Number coefficientP1 = monomialP1.getCoefficient();
        for (Monomial monomialP2 : p2.getMonomials().values()) {
            int degreeP2 = monomialP2.getDegree();
            Number coefficientP2 = monomialP2.getCoefficient();
            Monomial resMon = new Monomial();
            Number coef = (coefficientP1).doubleValue() * (coefficientP2).doubleValue();
            resMon.setCoefficient(coef);
            resMon.setDegree(degreeP1 + degreeP2);
            if (mapRes.containsKey(degreeP1 + degreeP2)) {
                mapRes.get(degreeP2 + degreeP1).setCoefficient(mapRes.get(degreeP2 + degreeP1)
                                    .getCoefficient().doubleValue() + coef.doubleValue());
            } else {
                if (coef.doubleValue() != 0.0) {
                    mapRes.put(degreeP1 + degreeP2, resMon);
                } else {
                    mapRes.put(0, new Monomial(0, 0.0));
                }
            }
        }
    }
    return createPolynomial(mapRes);
}
```

Division - this method computes the division of two polynomials, resulting in both the quotient and remainder. It iteratively calculates the quotient by determining the maximum degree and coefficients of the polynomials, then performs polynomial division until the dividend's degree is less than the divisor's. Additionally, it utilizes polynomial multiplication to subtract the product of p2 and each monomial of the quotient from the dividend, yielding the remainder. Finally, it returns a list containing both the quotient and remainder polynomials.

```
private Polynomial computeRemainder(Polynomial p1, Polynomial p2, Map<Integer,
Monomial> quotientMap) {
    Polynomial remainder = p1;
    Map<Integer, Monomial> remainderMap = new TreeMap<>(Collections.reverseOrder());
    for (Monomial monomial : quotientMap.values()) {
        Polynomial product = multiplyPolynomial(p2,
createPolynomial(Collections.singletonMap(monomial.getDegree(), monomial)));
        remainder = subtractPolynomial(remainder, product);
    }
    for (Monomial monomial : remainder.getMonomials().values()) {
        if (monomial.getCoefficient().doubleValue() != 0.0) {
            remainderMap.put(monomial.getDegree(), monomial);
        }
    }
    return createPolynomial(remainderMap);
}
```

```
private Map<Integer, Monomial> computeQuotient(Polynomial p1, Polynomial p2) {
    Map<Integer, Monomial> quotientMap = new TreeMap<>(Collections.reverseOrder());
    Monomial firstMonomialP1 = p1.getMonomials().isEmpty() ? null :
p1.getMonomials().values().iterator().next();
    int maxDegreeP1 = firstMonomialP1 != null ? firstMonomialP1.getDegree() : 0;
    double coef1 = firstMonomialP1 != null ? firstMonomialP1.getCoefficient().doubleValue() : 0.0;
    Monomial firstMonomialP2 = p2.getMonomials().isEmpty() ? null :
p2.getMonomials().values().iterator().next();
    int maxDegreeP2 = firstMonomialP2 != null ? firstMonomialP2.getDegree() : 0;
    double coef2 = firstMonomialP2 != null ? firstMonomialP2.getCoefficient().doubleValue() : 0.0;
    if (coef2 == 0 || maxDegreeP1 < maxDegreeP2) {
        throw new IllegalArgumentException();
    }
    while (coef1 != 0.0 && maxDegreeP1 >= maxDegreeP2) {
        double quotientCoefficient = coef1 / coef2;
        int degreeDifference = maxDegreeP1 - maxDegreeP2;
        if (quotientCoefficient != 0.0) {
            Monomial quotientMonomial = new Monomial(degreeDifference, quotientCoefficient);
            quotientMap.put(degreeDifference, quotientMonomial);
        }
        Map<Integer, Monomial> intermediate = new TreeMap<>();
        if (quotientCoefficient != 0.0) {
            intermediate.put(degreeDifference, new Monomial(degreeDifference, quotientCoefficient));
        }
        p1 = subtractPolynomial(p1, multiplyPolynomial(p2, createPolynomial(intermediate)));
        p1.getMonomials().remove(maxDegreeP1);
        Monomial firstMonomialInP1 = p1.getMonomials().isEmpty() ? null :
```

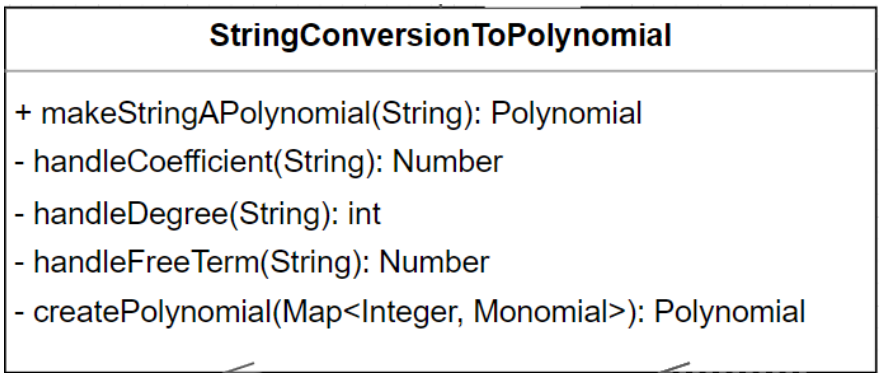
4. **Class "PolynomialProviderImpl"** is a class used to provide polynomials for the Operations test class. It implements a interface PolynomialProvider with a single method createPolynomial(OperationType operationType) that returns a list of arrays of polynomials used as arguments in the parametrized tests. It takes an OperationType object as input from the enumeration class OperationType, and based on it, it creates array of polynomials with one or two polynomials for the inputs and one expected polynomial for the result. The polynomials are created using the method makeStringAPolynomial from StringConversionToPolynomials class.



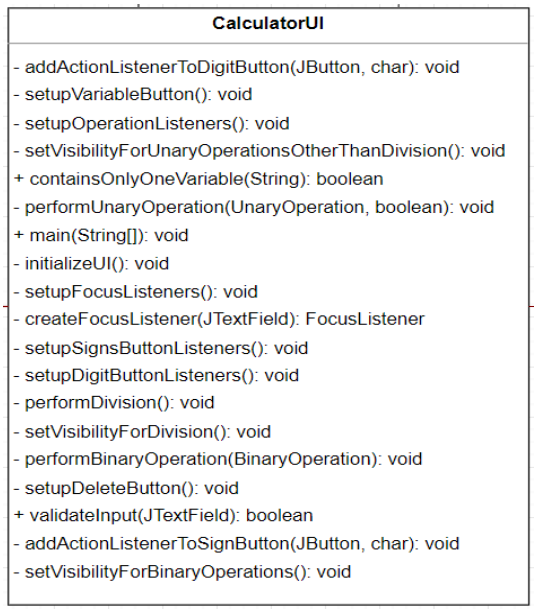

```
package controller.helper;
import..
public interface PolynomialProvider {
    List<Polynomial[]> createPolynomial(OperationType operationType);
}
```

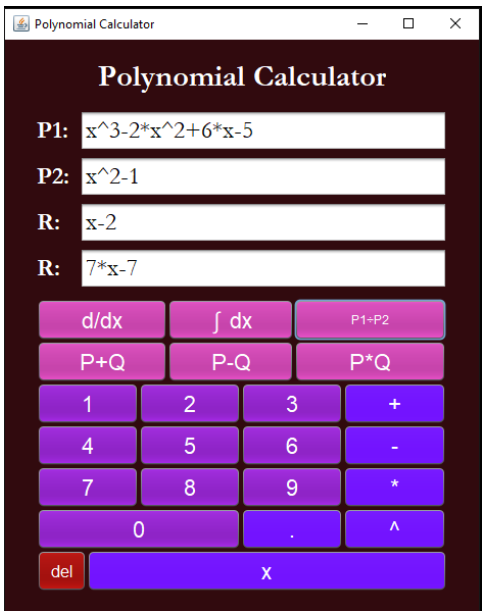
5. **Class "StringConversionToPolynomial"** is a class that have methods used to convert a string into a polynomial. It has a method, makeStringAPolynomial, that takes a String as input and based on a pattern that uses a regular expression to match any possible input. It then converts the matched strings into monomials used to create a Polynomial with the polynomials setter method.

```
Pattern pattern = Pattern.compile("([-+]?(?:\\d*\\.?\\d+)?)\\*?" +
Polynomial.VARIABLE_OF_THE_POLYNOMIAL + "\\^?(\\d*)?|([-+]?\\d*\\.?\\d+)?");
```



6. **Class "CalculatorUI"** - this class represents a Polynomial Calculator User Interface (UI) implemented in Java Swing. It provides functionalities to perform arithmetic operations on polynomials, including addition, subtraction, multiplication, division, differentiation, and integration. The UI includes text fields for inputting two polynomials, buttons for entering digits, mathematical symbols, and variable 'x'. It also provides buttons for various operations such as addition, subtraction, multiplication, division, differentiation, and integration. Input validation is performed ensure that the entered polynomials are correctly formatted.The class utilizes the Operations class for performing polynomial operations and the helper class (StringConversionToPolynomial) for converting string representations of polynomials into polynomial objects.The UI dynamically updates based on user interactions, showing the result of the operations in a text field.



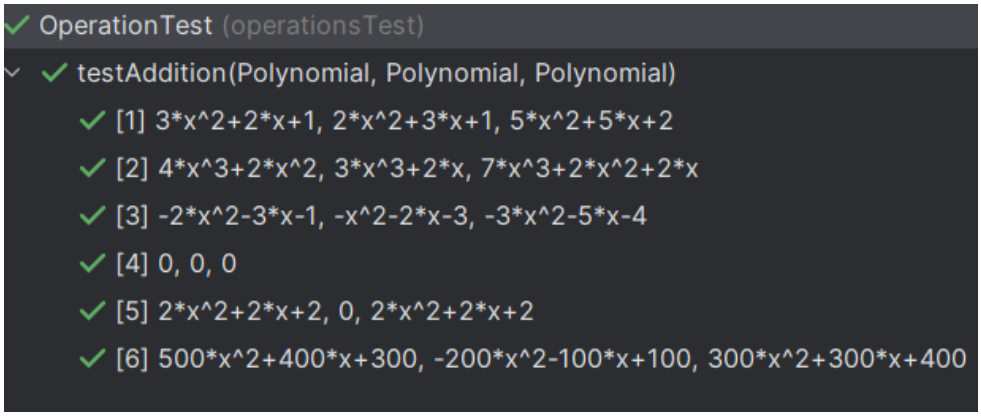


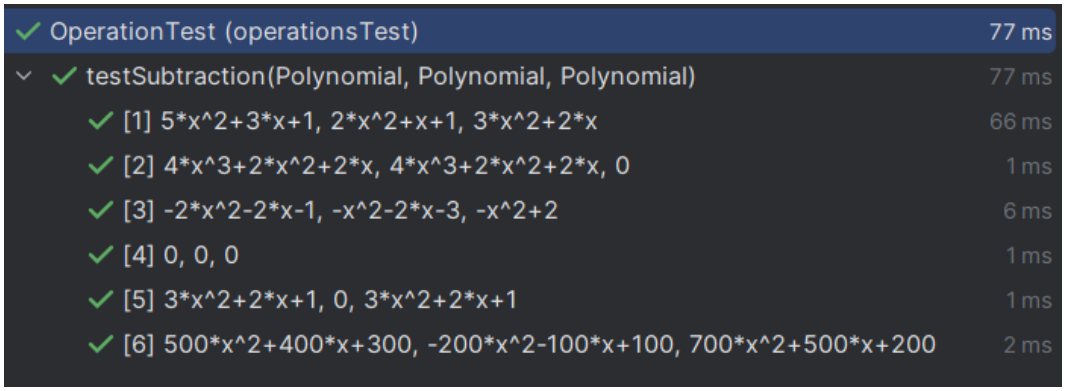
5. Results

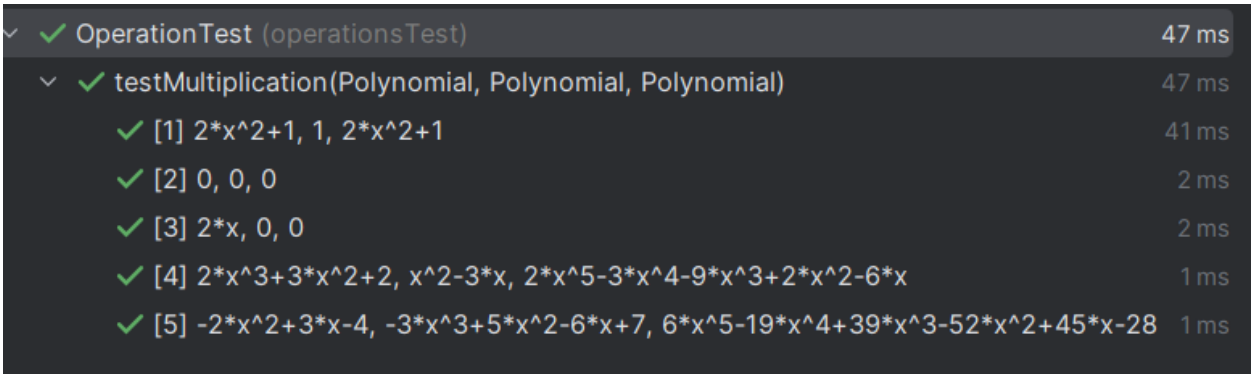
For the testing I have used parametrized JUnit tests.

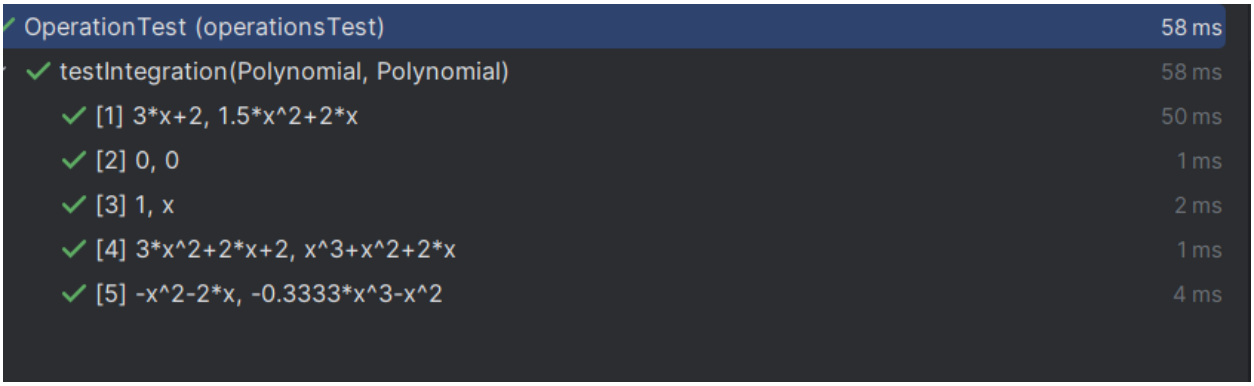
```
public List<Polynomial[]> providePolynomialFromString(OperationType operationType) {
    List<Polynomial[]> listTestCases = new ArrayList<>();
    if (operationType == OperationType.ADDITION) {
        listTestCases.add(createTestCaseForBinaryOperation(createPolynomialFromString( p: "3*x^2+2*x+1"), createPolynomialFromString( p: "2*x^2+3*x+1"),
                                                           createPolynomialFromString( p: "5*x^2+5*x+2")));
        listTestCases.add(createTestCaseForBinaryOperation(createPolynomialFromString( p: "4*x^3+2*x^2"), createPolynomialFromString( p: "3*x^3+2*x"),
                                                           createPolynomialFromString( p: "7*x^3+2*x^2+2*x")));
        listTestCases.add(createTestCaseForBinaryOperation(createPolynomialFromString( p: "-2*x^2-3*x-1"), createPolynomialFromString( p: "-x^2-2*x-3"),
                                                           createPolynomialFromString( p: "-3*x^2-5*x-4")));
        listTestCases.add(createTestCaseForBinaryOperation(createPolynomialFromString( p: "0"), createPolynomialFromString( p: "0"),
                                                           createPolynomialFromString( p: "0")));
        listTestCases.add(createTestCaseForBinaryOperation(createPolynomialFromString( p: "2*x^2+2*x+2"), createPolynomialFromString( p: "0"),
                                                           createPolynomialFromString( p: "2*x^2+2*x+2")));
        listTestCases.add(createTestCaseForBinaryOperation(createPolynomialFromString( p: "500*x^2+400*x+300"), createPolynomialFromString( p: "-200*x^2-100*x+100"),
                                                           createPolynomialFromString( p: "300*x^2+300*x+400")));

    }
```

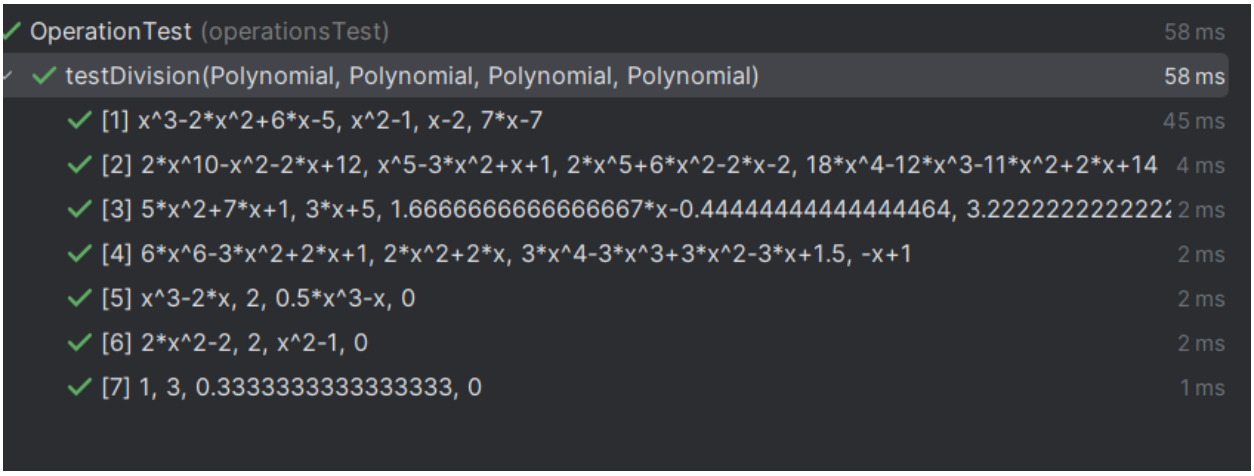
```
    @ParameterizedTest
    @MethodSource("parametersTestAddition")
    public void testAddition(Polynomial polynomial1, Polynomial polynomial2, Polynomial expectedResult) {
        Polynomial result = op.addPolynomial(polynomial1, polynomial2);
        for (Monomial expectedMonomial : expectedResult.getMonomials().values()) {
            assertTrue(result.getMonomials().containsValue(expectedMonomial));
        }
        numberOfPassedTests++;
    }

    1 usage
    public static Stream<Arguments> parametersTestAddition() {
        PolynomialProviderImpl polynomialProvider = new PolynomialProviderImpl();
        List<Polynomial[]> polynomials = polynomialProvider.createPolynomial(OperationType.ADDITION);
        return polynomials.stream().map(arr -> Arguments.of(arr[0], arr[1], arr[2]));
    }
```

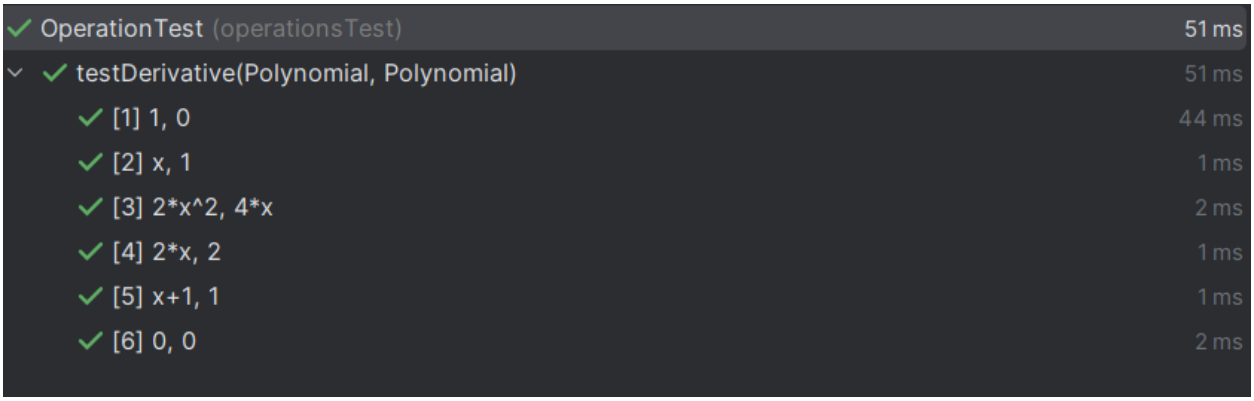





OperationTest (operationsTest) 47 ms
testMultiplication(Polynomial, Polynomial, Polynomial) 47 ms
[1] 2*x^2+1, 1, 2*x^2+1 41 ms
[2] 0, 0, 0 2 ms
[3] 2*x, 0, 0 2 ms
[4] 2*x^3+3*x^2+2, x^2-3*x, 2*x^5-3*x^4-9*x^3+2*x^2-6*x 1 ms
[5] -2*x^2+3*x-4, -3*x^3+5*x^2-6*x+7, 6*x^5-19*x^4+39*x^3-52*x^2+45*x-28 1 ms

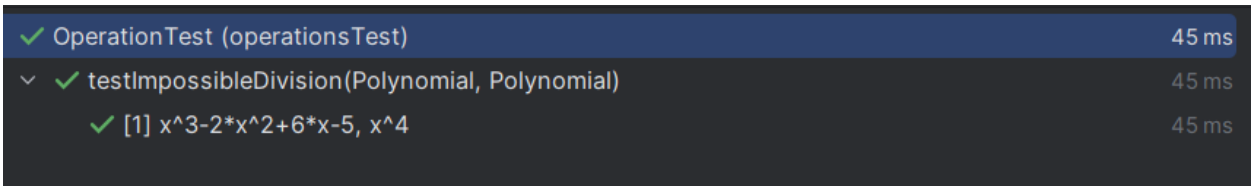

OperationTest (operationsTest) 58 ms
testIntegration(Polynomial, Polynomial) 58 ms
[1] 3*x+2, 1.5*x^2+2*x 50 ms
[2] 0, 0 1 ms
[3] 1, x 2 ms
[4] 3*x^2+2*x+2, x^3+x^2+2*x 1 ms
[5] -x^2-2*x, -0.3333*x^3-x^2 4 ms


OperationTest (operationsTest) 58 ms
testDivision(Polynomial, Polynomial, Polynomial, Polynomial) 58 ms
[1] x^3-2*x^2+6*x-5, x^2-1, x-2, 7*x-7 45 ms
[2] 2*x^10-x^2-2*x+12, x^5-3*x^2+x+1, 2*x^5+6*x^2-2*x-2, 18*x^4-12*x^3-11*x^2+2*x+14 4 ms
[3] 5*x^2+7*x+1, 3*x+5, 1.6666666666666667*x-0.44444444444444464, 3.2222222222222 2 ms
[4] 6*x^6-3*x^2+2*x+1, 2*x^2+2*x, 3*x^4-3*x^3+3*x^2-3*x+1.5, -x+1 2 ms
[5] x^3-2*x, 2, 0.5*x^3-x, 0 2 ms
[6] 2*x^2-2, 2, x^2-1, 0 2 ms
[7] 1, 3, 0.3333333333333333, 0 1 ms

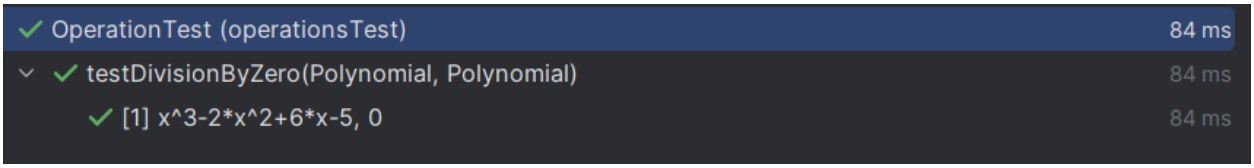

OperationTest (operationsTest) 51 ms
testDerivative(Polynomial, Polynomial) 51 ms
[1] 1, 0 44 ms
[2] x, 1 1 ms
[3] 2*x^2, 4*x 2 ms
[4] 2*x, 2 1 ms
[5] x+1, 1 1 ms
[6] 0, 0 2 ms

For the scenarios where division cannot be performed(e.g. null divisor or divisor's degree greater than dividend's degree) the tests expect an exception as result.

```java
@ParameterizedTest
@MethodSource("parametersTestImpossibleDivision")
public void testImpossibleDivision(Polynomial dividend, Polynomial divisor) {
    assertThrows(IllegalArgumentException.class, () -> op.dividePolynomial(dividend, divisor));
    numberOfPassedTests++;
}

1 usage
public static Stream<Arguments> parametersTestImpossibleDivision() {
    PolynomialProviderImpl polynomialProvider = new PolynomialProviderImpl();
    List<Polynomial[]> polynomials = polynomialProvider.createPolynomial(OperationType.IMPOSSIBLE_DIVISION);
        return polynomials.stream().map(arr -> Arguments.of(arr[0], arr[1]));
    }
```




```java
@ParameterizedTest
@MethodSource("parametersTestDivisionByZero")
public void testDivisionByZero(Polynomial dividend, Polynomial divisor){
    assertThrows(IllegalArgumentException.class, () -> op.dividePolynomial(dividend, divisor));
    numberOfPassedTests++;
}

1 usage
public static Stream<Arguments> parametersTestDivisionByZero() {
    PolynomialProviderImpl polynomialProvider = new PolynomialProviderImpl();
    List<Polynomial[]> polynomials = polynomialProvider.createPolynomial(OperationType.DIVISION_BY_ZERO);
    return polynomials.stream().map(arr -> Arguments.of(arr[0], arr[1]));
}
```



For the conversion class the test compares the string obtained by calling the makeStringPolynomial method with a string that represents the expected result. The polynomials are created using a helper method that take arrays of double and int coefficients and degrees as input.

```java
private static Polynomial createPolynomial(Double[] coefficients, Integer[] degrees) {
    if (coefficients.length != degrees.length) {
        throw new IllegalArgumentException("Coefficients and degrees arrays must have the same length.");
    }
    Map<Integer, Monomial> monomialsMap = new TreeMap<>();
    for (int i = 0; i < coefficients.length; i++) {
        monomialsMap.put(degrees[i], new Monomial(degrees[i], coefficients[i]));
    }
    return new Polynomial(monomialsMap);
}
```

```java
public static List<Arguments> parametersConversionTest() {
    List<Arguments> arguments = new ArrayList<>();
    arguments.add(Arguments.of( ...arguments: "1", createPolynomial(new Double[]{1.0}, new Integer[]{0})));
    arguments.add(Arguments.of( ...arguments: "2", createPolynomial(new Double[]{2.0}, new Integer[]{0})));
    arguments.add(Arguments.of( ...arguments: "-3", createPolynomial(new Double[]{-3.0}, new Integer[]{0})));
    arguments.add(Arguments.of( ...arguments: "100", createPolynomial(new Double[]{100.0}, new Integer[]{0})));
    arguments.add(Arguments.of( ...arguments: "x", createPolynomial(new Double[]{1.0}, new Integer[]{1})));
    arguments.add(Arguments.of( ...arguments: "-x", createPolynomial(new Double[]{-1.0}, new Integer[]{1})));
    arguments.add(Arguments.of( ...arguments: "-2*x", createPolynomial(new Double[]{-2.0}, new Integer[]{1})));
    arguments.add(Arguments.of( ...arguments: "x^2", createPolynomial(new Double[]{1.0}, new Integer[]{2})));
    arguments.add(Arguments.of( ...arguments: "-3*x^2", createPolynomial(new Double[]{-3.0}, new Integer[]{2})));
    arguments.add(Arguments.of( ...arguments: "300*x^2", createPolynomial(new Double[]{300.0}, new Integer[]{2})));
    arguments.add(Arguments.of( ...arguments: "x^2-1", createPolynomial(new Double[]{1.0, -1.0}, new Integer[]{2, 0})));
    arguments.add(Arguments.of( ...arguments: "2*x^2", createPolynomial(new Double[]{2.0}, new Integer[]{2})));
    arguments.add(Arguments.of( ...arguments: "2*x^3+x^2+3*x+5", createPolynomial(new Double[]{2.0, 1.0, 3.0, 5.0}, new Integer[]{3, 2, 1, 0})));
    arguments.add(Arguments.of( ...arguments: "2*x^2-x^4+x+6", createPolynomial(new Double[]{2.0, -1.0, 1.0, 6.0}, new Integer[]{2, 4, 1, 0})));
    return arguments;
}
```

```java
@ParameterizedTest
@MethodSource("parametersConversionTest")
public void testStringConversionToPolynomial(String stringPolynomial, Polynomial convertedPolynomial) {
    StringConversionToPolynomial stringConversionToPolynomial = new StringConversionToPolynomial();
    Polynomial result = stringConversionToPolynomial.makeStringAPolynomial(stringPolynomial);
    Map<Integer, Monomial> mapRes = result.getMonomials();
    assertEquals(mapRes.size(), convertedPolynomial.getMonomials().size());
    for (Monomial monomial : mapRes.values()) {
        assertTrue(convertedPolynomial.getMonomials().containsValue(monomial));
    }
    numberOfPassedTests++;
}
```

The prettyPrint method is a customized version of the toString method from the Polynomial class used to make the result from the interface readable and user-friendly.

```java
public static List<Arguments> parametersConversionTest() {
    List<Arguments> arguments = new ArrayList<>();
    arguments.add(Arguments.of( ...arguments: "0", createPolynomial(new Double[]{0.0}, new Integer[]{0})));
    arguments.add(Arguments.of( ...arguments: "5", createPolynomial(new Double[]{5.0}, new Integer[]{0})));
    arguments.add(Arguments.of( ...arguments: "0.08", createPolynomial(new Double[]{0.08}, new Integer[]{0})));
    arguments.add(Arguments.of( ...arguments: "10.08", createPolynomial(new Double[]{10.08}, new Integer[]{0})));
    arguments.add(Arguments.of( ...arguments: "100.01", createPolynomial(new Double[]{100.01}, new Integer[]{0})));
    arguments.add(Arguments.of( ...arguments: "1", createPolynomial(new Double[]{1.0}, new Integer[]{0})));
    arguments.add(Arguments.of( ...arguments: "2", createPolynomial(new Double[]{2.0}, new Integer[]{0})));
    arguments.add(Arguments.of( ...arguments: "-3", createPolynomial(new Double[]{-3.0}, new Integer[]{0})));
    arguments.add(Arguments.of( ...arguments: "100", createPolynomial(new Double[]{100.0}, new Integer[]{0})));
    arguments.add(Arguments.of( ...arguments: "x", createPolynomial(new Double[]{1.0}, new Integer[]{1})));
    arguments.add(Arguments.of( ...arguments: "-x", createPolynomial(new Double[]{-1.0}, new Integer[]{1})));
    arguments.add(Arguments.of( ...arguments: "-2*x", createPolynomial(new Double[]{-2.0}, new Integer[]{1})));
    arguments.add(Arguments.of( ...arguments: "x^2", createPolynomial(new Double[]{1.0}, new Integer[]{2})));
    arguments.add(Arguments.of( ...arguments: "-3*x^2", createPolynomial(new Double[]{-3.0}, new Integer[]{2})));
    arguments.add(Arguments.of( ...arguments: "-3*x", createPolynomial(new Double[]{-3.0}, new Integer[]{1})));
    arguments.add(Arguments.of( ...arguments: "2", createPolynomial(new Double[]{2.0}, new Integer[]{0})));
    arguments.add(Arguments.of( ...arguments: "300*x^2", createPolynomial(new Double[]{300.0}, new Integer[]{2})));
    arguments.add(Arguments.of( ...arguments: "x^2-1", createPolynomial(new Double[]{1.0, -1.0}, new Integer[]{2,0})));
    arguments.add(Arguments.of( ...arguments: "2*x^2", createPolynomial(new Double[]{2.0}, new Integer[]{2})));
    arguments.add(Arguments.of( ...arguments: "2*x^3+x^2+3*x+5", createPolynomial(new Double[]{2.0, 1.0, 3.0, 5.0}, new Integer[]{3, 2, 1, 0})));
    arguments.add(Arguments.of( ...arguments: "2*x^4-x^2+x+6", createPolynomial(new Double[]{2.0, -1.0, 1.0, 6.0}, new Integer[]{4, 2, 1, 0})));
    return arguments;
}
```

```java
private static Polynomial createPolynomial(Double[] coefficients, Integer[] degrees) {
    if (coefficients.length != degrees.length) {
        throw new IllegalArgumentException("Coefficients and degrees arrays must have the same length.");
    }
    Map<Integer, Monomial> monomialsMap = new TreeMap<>(Collections.reverseOrder());
    for (int i = 0; i < coefficients.length; i++) {
        monomialsMap.put(degrees[i], new Monomial(degrees[i], coefficients[i]));
    }
    return new Polynomial(monomialsMap);
}

@ParameterizedTest
@MethodSource("parametersConversionTest")
public void testStringConversionToPolynomial(String stringPolynomial, Polynomial convertedPolynomial){
    System.out.println(convertedPolynomial.prettyPrint());
   assertEquals(stringPolynomial, convertedPolynomial.prettyPrint());
}
```

```
✓ PrettyPrintTest (prettyPrintTest)                                179
  ✓ prettyPrintTest(String, Polynomial)                            179
      ✓ [1] 0, 0                                                   88
      ✓ [2] 5, 5                                                   4
      ✓ [3] 0.08, 0.08                                             4
      ✓ [4] 10.08, 10.08                                           8
      ✓ [5] 100.01, 100.01                                         5
      ✓ [6] 1, 1                                                   3
      ✓ [7] 2, 2                                                   3
      ✓ [8] -3, -3                                                 5
      ✓ [9] 100, 100                                               4
      ✓ [10] x, x                                                  2
      ✓ [11] -x, -x                                                4
      ✓ [12] -2*x, -2*x                                            2
      ✓ [13] x^2, x^2                                              3
      ✓ [14] -3*x^2, -3*x^2                                        15
      ✓ [15] -3*x, -3*x                                            4
      ✓ [16] 2, 2                                                  2
      ✓ [17] 300*x^2, 300*x^2                                      7
      ✓ [18] x^2-1, x^2-1                                          3
      ✓ [19] 2*x^2, 2*x^2                                          3
      ✓ [20] 2*x^3+x^2+3*x+5, 2*x^3+x^2+3*x+5                      7
      ✓ [21] 2*x^4-x^2+x+6, 2*x^4-x^2+x+6                          3
```

6. Conclusion

In conclusion, the polynomial calculator project has been successfully developed to provide users a reliable tool for performing various mathematical operations on polynomial expressions, through the implementation of the Polynomial and Monomial classes, along with the Operations class and other helper classes capable of handling addition, subtraction, multiplication, division, differentiation, and integration.

I have learned to apply patterns with regular expressions, implement the MVC software architecture, utilize parameterized JUnit tests, and handle polynomial operation data effectively.

Future development and upgrades might include a picker for the polynomial variable, a calculator history, step by step solving guides.

7. Bibliography

1. *https://stackoverflow.com/questions/4617615/how-to-set-nimbus-look-and-feel-in-mainWhat are Java classes? - www.tutorialspoint.com*
2. *https://www.arhohuttunen.com/junit-5-migration/*
3. *https://regex101.com/*
4. *https://dsrl.eu/courses/pt/materials/PT_2024_A1_S1.pdf*
5. *https://dsrl.eu/courses/pt/materials/PT_2024_A1_S3.pdf*