# DOCUMENTAȚIE

## TEMA *NUMARUL_1*

## CUPRINS

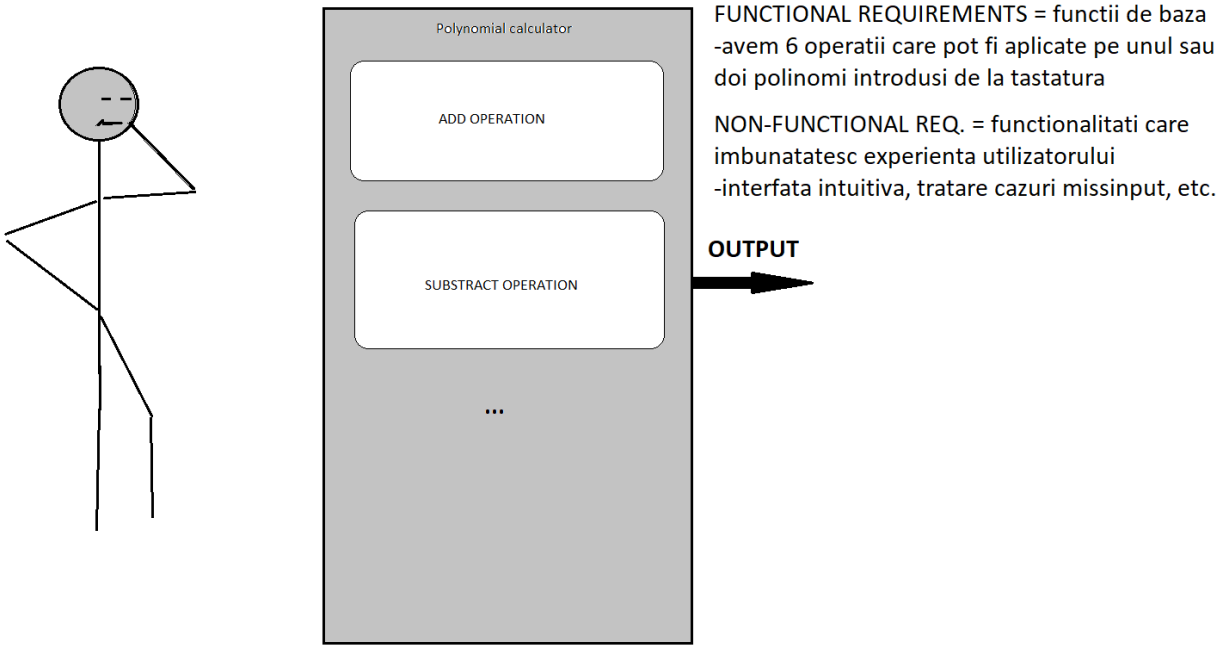


NUME STUDENT: IRIMEȘ DAVID
GRUPA: 30227

# 1. Obiectivul temei

• Obiectiv principal: proiectarea și implementarea unui calculator polinomial cu o interfață grafică dedicată prin care utilizatorul poate insera polinoame, poate selecta operația matematică care trebuie efectuată și poate vizualiza rezultatul.
• Subobiective
  - Analiza problemei și identificarea cerințelor ➔cap. 2
  - Proiectarea calculatorul polinom ➔ cap. 3
  - Implementarea efectivă calculatorul polinom ➔cap. 4
  - Faza de testare a calculatorului polinom ➔cap. 5

# 2. Analiza problemei, modelare, scenarii, cazuri de utilizare




NUME STUDENT: IRIMEȘ DAVID
GRUPA: 30227

# 3. Proiectare

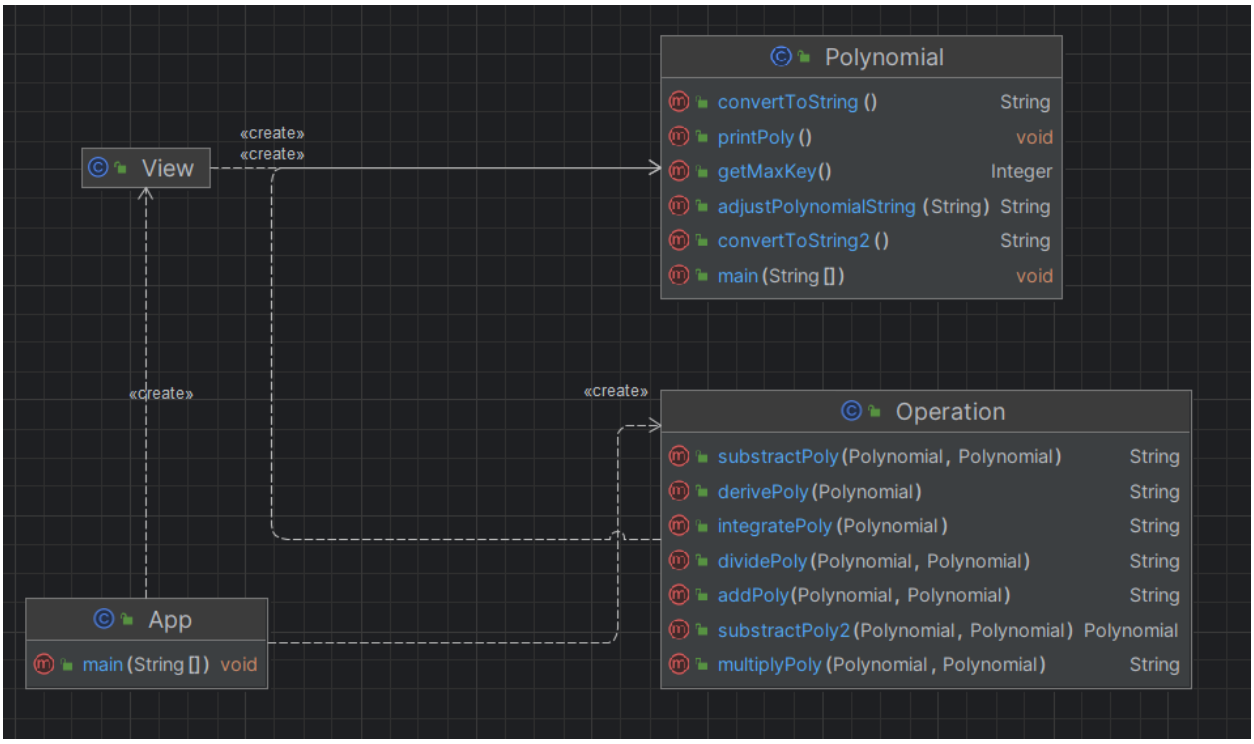
Tema se imparte in doua mari fronturi initial si implicit, doua pachete:
-partea de interfata grafica (GUI), unde utilizatorul va avea contact cu aplicatia ➔ aceasta se va realiza in pachetul View, si mai exact in clasa cu acelasi nume, care extinde clasa predefinita Jframe (se va lucra cu Java Swing)
-partea de logica/implementare de operatii, unde se adauga de fapt functionalitatile de baza ale aplicatiei ➔ aceasta se va regasi in pachetul Model

Singura clasa din pachetul pentru interfata va fi View, care va fi implementata si cu un form, pentru usurinta dispunerii elementelor in fereastra de afisaj.
Pentru pachetul Model avem clasele:

- Polynomial: care va retine un polinom folosind un Map, de ce? Pentru ca astfel accesul la memorie este rapid si polinoamele pe care le dam ca si input trebuie sa fie in forma finala, adica nu putem avea doua monoame cu acelasi grad, fapt ceea ce face ca Map sa fie cea mai buna structura pentru memorare. Ca principiu, constructorul va asigura traducerea unui string in polinom.
- Operation: care va contine toate metodele specifice operatiilor care trebuie implementate.

OBS: o sa avem o clasa App de unde porneste de fapt executia

Diagrama UML de clase:



Pe langa aceste pachete mai avem si un pachet special pentru testare unitara cu JUnit, unde se va verifica functionalitatea fiecarei operatii.

# 4. Implementare

VIEW:
Aici definim campurile pentru citire si afisare si butoanele care permit aplicarea de operatii. Action Listeners o sa se regaseasca aici, fiind putin cod, iar eventualele exceptii se trateaza aici, avand, dupa caz, mesaje specifice si in consola si in interfata.

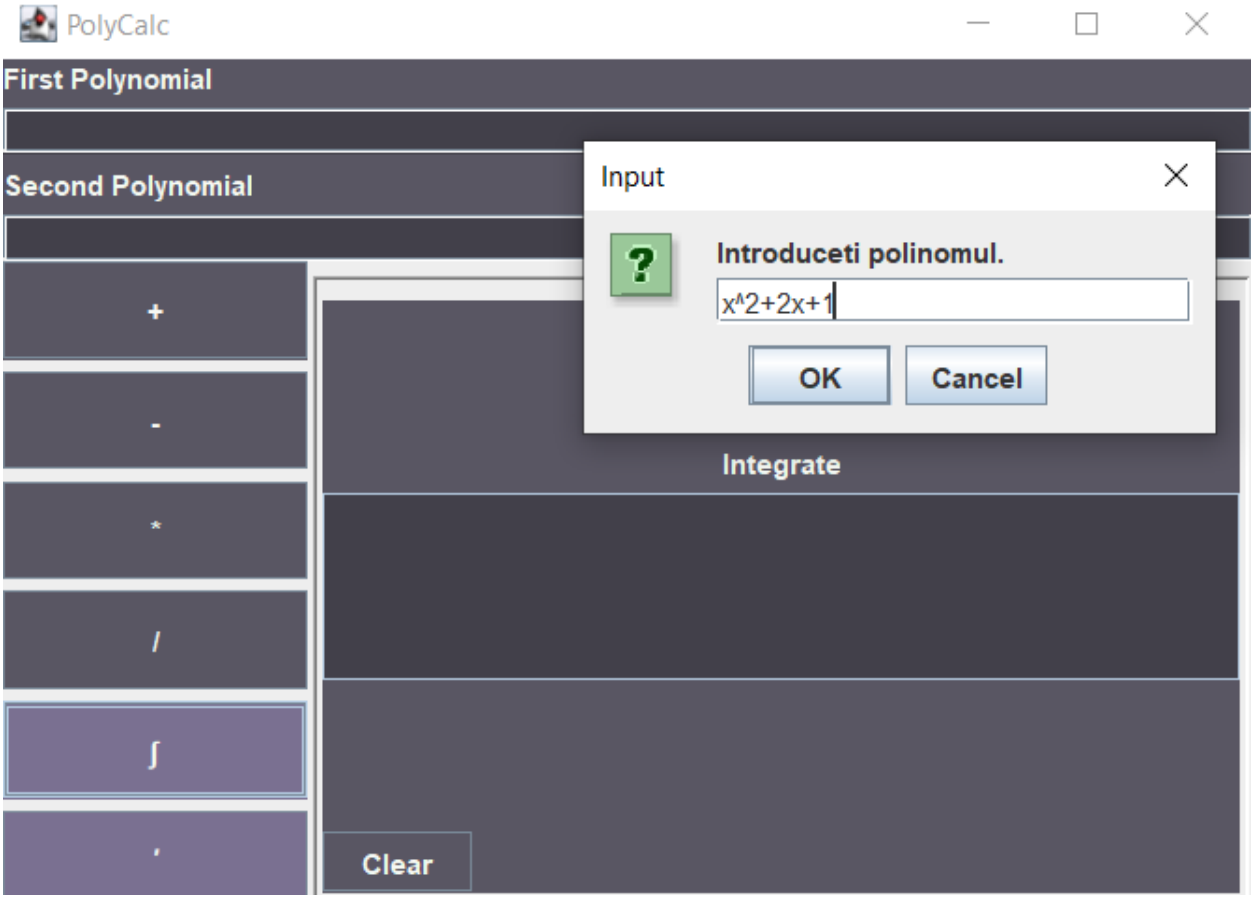
Ex:

```
substractButton.addActionListener((e) -> {
        outputLabel.setText("Substract");
        String s1 = textField1.getText();
        String s2 = textField2.getText();
        s1 = Polynomial.adjustPolynomialString(s1);
        s2 = Polynomial.adjustPolynomialString(s2);
        if(s1 == "" || s2 == "") {
            System.out.println("EXCEPTION: <NoInput>");
            outputText.setText("");
        }
        else{
            Polynomial op1 = new Polynomial(s1);
            Polynomial op2 = new Polynomial(s2);
            outputText.setText(Operation.substractPoly(op1,op2));
        }
});
```

Pentru ultimele doua operatii, ne trebuie doar un input, iar in cazul apasarii butonului, am optat pentru aparitia unei ferestre de dialog de unde se poate face citirea unicului operand:

```
integrateButton.addActionListener((e) -> {
        outputLabel.setText("Integrate");
        String input = JOptionPane.showInputDialog( parentComponent: null, message: "Introduceti polinomul.");
        if(input != null && !input.isEmpty()) {
            System.out.println("<Execute operation>");
            input = Polynomial.adjustPolynomialString(input);
            Polynomial poly = new Polynomial(input);
            outputText.setText(Operation.integratePoly(poly));
        }else{
            System.out.println("<Canceled>");
        }
});
```

Interfata in sine:



POLYNOMIAL:

```
Map <Integer, Double> monomials = new HashMap<>();

21 usages  new *
public static String adjustPolynomialString(String poly) {...}
10 usages  new *
public Polynomial(){}
24 usages  new *
public Polynomial(String poly) {...}
no usages  new *
public void printPoly() {...}
8 usages  new *
public String convertToString(){...}
1 usage  new *
public String convertToString2(){...}
10 usages  new *
public Integer getMaxKey(){...}
```

Avem un camp si o multime de metode care, pe rand:

- Permit memorearea unui polinom→gradul=Integer=cheia; coeficientul=Double=valoarea
- adjustPolynomialString()→Adauga “^0”,”^1”,elimina spatiile,etc →folosire regex: "(?=[-+])")
- Constructori→Def. polinom cu un String ca input unde folosim regex:([+]?\d*\.?\d*)?x\^?(\d+)?
- printPoly() → pentru testare in early stages (afiseaza monoamele)
- convertToString() 1 sau 2 →convertirea la string mai ‘fancy’(prima, scapa de caractere inutile la afisare, “^0”… afiseaza in ordine descrescatoare a gradelor), sau mai de folos (a doua - folosita pentru operatia de impartire)
- getMaxKey() pentru a gasi monomul cu cel mai mare grad → tot pentru impartire (obs: se putea folosi de la inceput TreeMap)

OPERATION:

- are doar metode statice, asociate cu fiecare operatie.
- printre ele, impartirea se dovedeste a fi cea mai grea, urmand un algoritm exact:
  1. se verifica daca gradul deimpartitului este mai mare decat cel al impartitorului

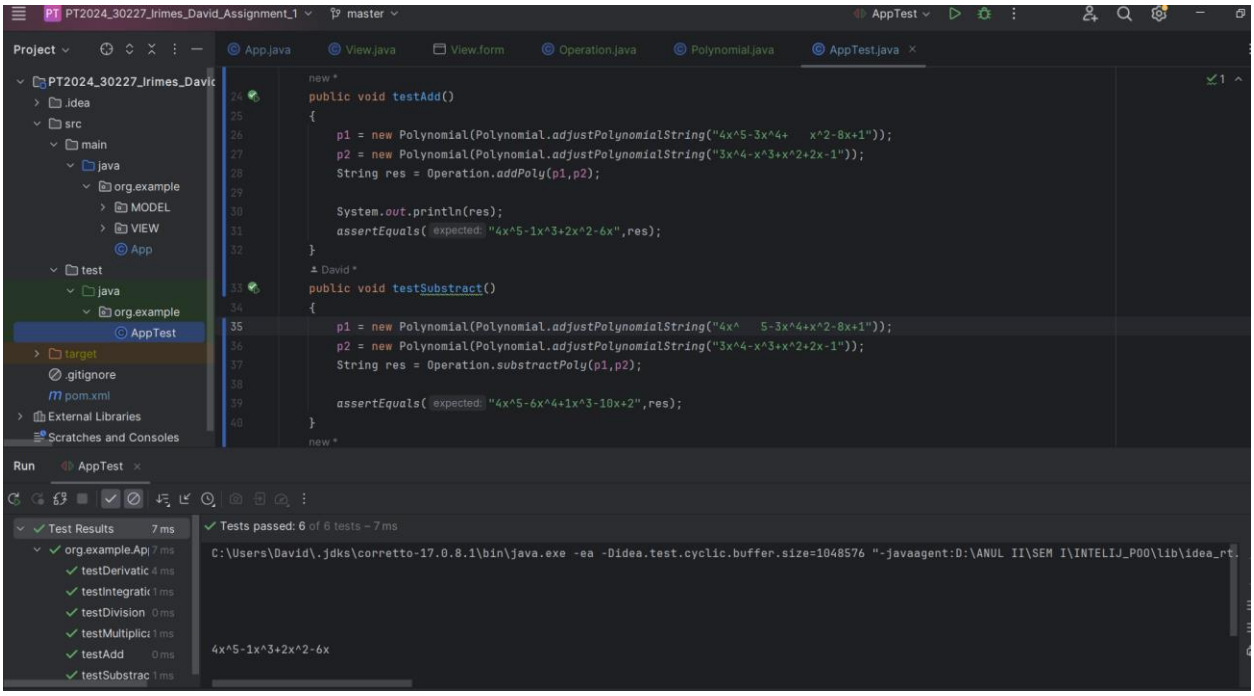
2. Alegerea primului termen: Începe cu primul termen din polinomul dividend și împarte-l la primul termen al polinomului divizor.
3. Înmulțirea: Înmulțește polinomul divizor cu rezultatul obținut la pasul anterior și scrie rezultatul sub polinomul dividend.
4. Scădere: Scade polinomul obținut prin înmulțire din polinomul dividend.
5. Alegerea următorului termen: Alegeți următorul termen din polinomul dividend și adăugați-l la rezultatul obținut la pasul anterior pentru a obține un nou dividend.
6. Repetiție: Repetă pașii 2-5 până când termenii din dividend sunt de grad mai mic decât cei din divizor.\

# 5. Rezultate

Rezultatele au fost obtinute prin testarea unitara cu JUnit. Am folosit stringuri care sa foloseasca toate procesele de pre-editare/ pregatire a inputului, cum ar fi spatii fara sens, lipsa scrierii explicite  a gradului “^0” sau “^1”, lipsa semnului primului monom, etc.

Orice coeficient care nu este numar intreg sau suficient de aproape de un numar intreg (marja de eroare <0.1) va fi scris cu doua zecimale dupa virgula.

Testarea unitara:



# 6. Concluzii

1. Am invatat de existenta regex si utilitatea sa.
2. Am invatat si am implementat pentru prima data testarea unitara pe JUnit
3. Am invatat cum sa lucrez cu GitLab

Posibile dezvoltari:

- Daca coeficientul este 1 sau -1, se va afisa in string-ul rezultat chiar daca e redundant (prea multe if-uri)
- Interfata lasa de dorit si s-ar putea dezvolta astfel incat in fereastra de dialog de la operatia de integrare si derivare sa se poata alege optiunea de a selecta un input scris deja in campurile de mai sus, in cazul in care utilizatorul foloseste aplicatia prima data si nu stie ca nu trebuie sa foloseasca acele campuri pentru operatiile de integrare si derivare.

# 7. Bibliografie


- Algoritmul matematic de impartire a doua polinoame: https://youtu.be/_FSXJmESFmQ?si=8_txmjbZzqkW_ocK
- Cum se foloseste regex: https://coderpad.io/blog/development/the-complete-guide-to-regular-expressions-regex/