DOCUMENTATIE

TEMA 1

FLORICA STEFANIA

GRUPA: 30228

# 

# CUPRINS

[1. Obiectivul temei 3](#_Toc95297885)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 3](#_Toc95297886)

[3. Proiectare 4](#_Toc95297887)

[4. Implementare 6](#_Toc95297888)

[5. Rezultate 10](#_Toc95297889)

[6. Concluzii 11](#_Toc95297890)

[7. Bibliografie 11](#_Toc95297891)

# Obiectivul temei

Obiectivul principal al temei este implementarea unui calculator care realizează diverse operații matematice pe structuri de tip polinom. Un polinom este o unealtă matematica cu numeroase aplicații in domeniul Computer Science sau Automatizări. Se dorește construirea unei aplicații cu o interfață grafica pentru utilizator cat mai simplista, astfel încât sa poată fi folosita cu ușurință de orice tip de utilizator, fie ca acesta are sau un o anume pregătire tehnica.

Printre principalele operații pe care ne dorim sa le efectuam asupra unor polinoame se numără adunarea, scăderea, înmulțirea, împărțirea, derivarea si integrarea. Primele patru dintre ele implica existenta a doi operanzi, in timp ce ultimele doua se efectuează pe cate un singur operand.

Obiectivele secundare constau in implementarea acestor operații, ținându-se cont de paradigmele programării orientate pe obiect, folosindu-se anumite structuri de date specifice si arhitectura aplicației de tip MVC. Aceste aspecte vor fi prezentate in detaliu in cadrul secțiunii 3, in care se descrie proiectarea aplicației.

# Analiza problemei, modelare, scenarii, cazuri de utilizare

* **2.1 Analiza si modelarea problemei**

Analiza problemei presupune extragerea din enunțul problemei a substantivelor (Polinom, Monom), care devin clasele-candidat, si a verbelor (aduna, scade, împarte etc.), care se vor concretiza in metodele implementate in clasele respective. Odată ce am parcurs aceasta etapa, putem considera ca am pus bazele rezolvării problemei, urmând sa ne axam pe descrierea funcțională a acesteia, pe comportamentul aplicației la introducerea anumitor date de carte utilizator.

Aplicația poate fi utilizata de un număr larg de categorii de persoane, de aceea modul in care va fi implementata interfața grafica reprezintă un important aspect, deoarece ea realizează comunicarea intre utilizator si funcționalitatea aplicației. GUI-ul va fi descris in detaliu in secțiunea 4, însă, pe scurt, acesta cuprinde doua componente de tip JTextField pentru introducerea polinoamelor sub forma de String-uri si un al treilea pentru afișarea rezultatului operației selectate.

Implementarea depinde de la programator la programator, dar orice soluție trebuie sa funcționeze conform standardelor.

* **2.2 Scenarii si cazuri de utilizare**

Cazurile de utilizare ale aplicației se împart in doua mari categorii: operații pe doua polinoame, respectiv pe un singur polinom.

Figura 1 reprezintă o diagrama Use-Case pentru aplicație, in care sunt reliefate principalele scenarii de utilizare a calculatorului de polinoame.

1. Utilizatorul introduce date de forma *„**anxn+an-1xn-1+…+a1x1+a0”* corespunzătoare operației pe care dorește sa o efectueze. Pentru primele 4 operații, utilizatorul trebuie sa completeze primele doua casete, in timp ce pentru derivare si integrare, se cere doar completarea primei casete, fapt precizat printr-un mesaj informativ in GUI.
2. Utilizatorul selectează operația dorita, apăsând butonul corespunzător.
3. Rezultatul este afișat in caseta rezervata acestei operațiuni.

In implementarea calculatorului, am luat in considerare si cazurile excepționale, in care utilizatorul introduce un sir de caractere care nu corespunde formei generale a unui polinom, fiind astfel avertizat printr-o apariția unei ferestre cu un mesaj corespunzător (in arhitectura interna a aplicației, acest caz se tratează prin aruncarea unei excepții pentru ca programul sa nu se oprească brusc.)

De asemenea, am avut in vedere si cazul in care utilizatorul introduce un polinom corect din punct de vedere structural, însă gradele monoamelor nu sunt in ordine descrescătoare, aceasta situație excepționala fiind tratata prin ordonarea monoamelor in cadrul obiectului stocat in memorie pentru a facilita operațiile ulterioare.

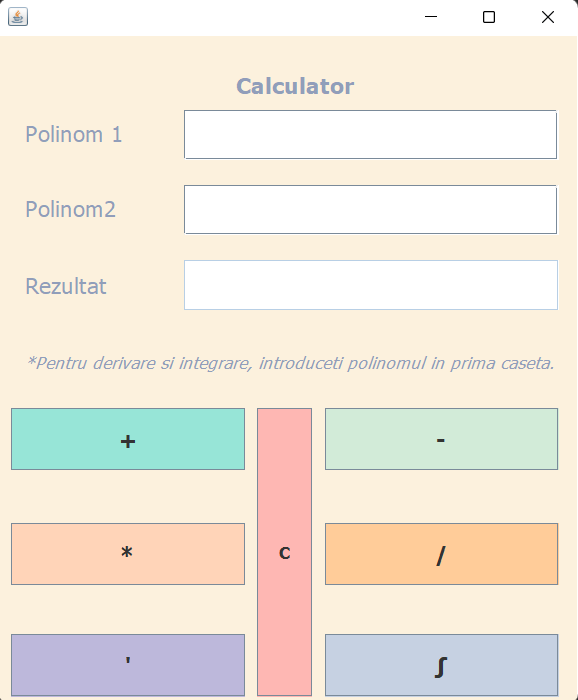
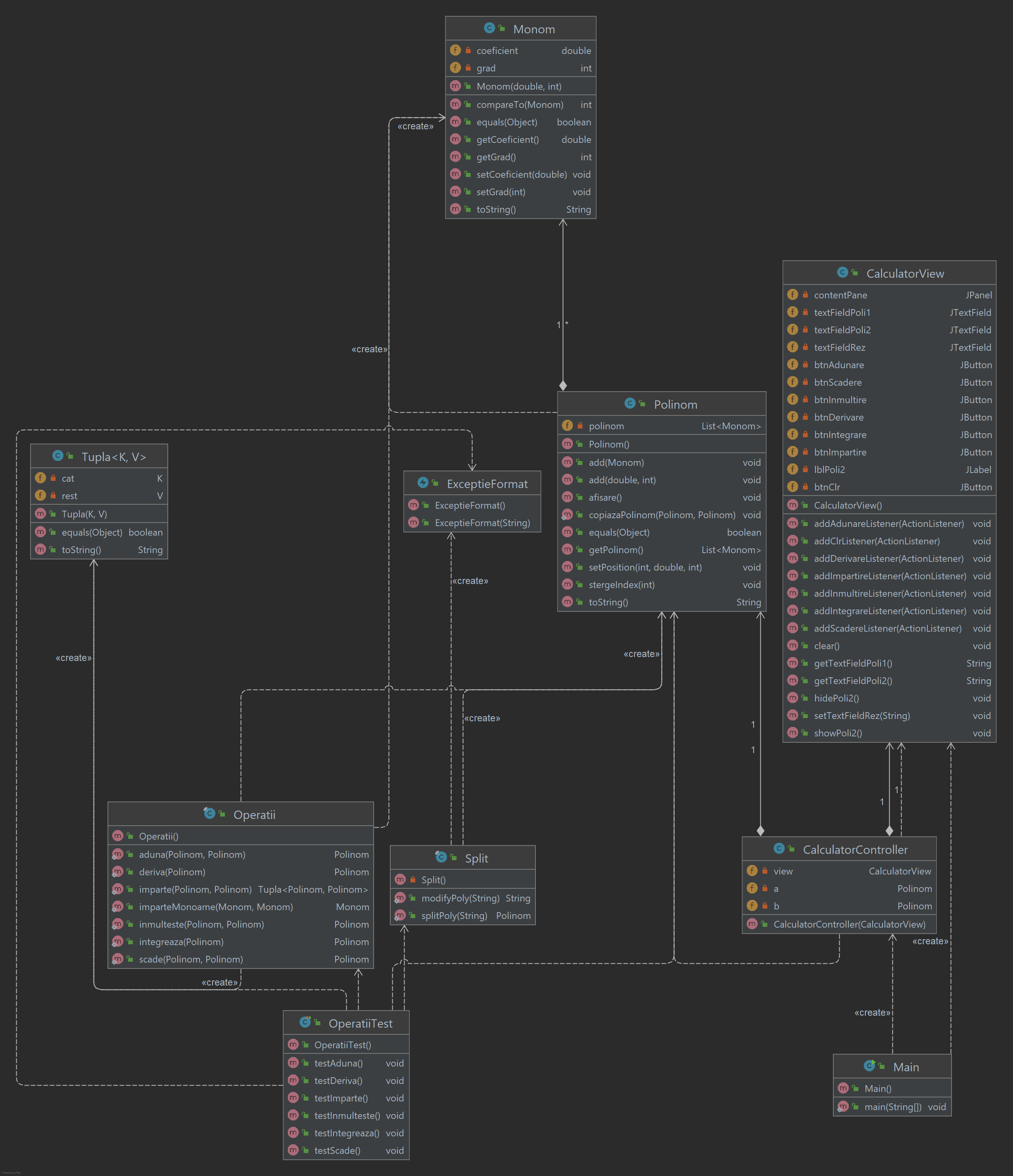


Fig. 1 (Diagrama Use-Case) Fig.2 (Interfata grafica pentru utilizator)

# Proiectare

Etapa de proiectare presupune stabilirea structurilor de date utilizate pentru stocarea si interpretarea datelor primite de la utilizator, precum si crearea de clase si pachete conform principiilor Programării Orientate pe Obiect.

Am ales polinomul sa fie o obiect format din mai multe obiecte de tip Monom, care, la rândul lui conține coeficient si grad. Pentru o mai buna înțelegere a întregii arhitecturi interne a aplicației, am apelat la schițarea unei diagrame de clasa UML (*Unified Modeling Language*), folosită pentru reprezentarea vizuală a claselor și a interdependențelor. UML este notația internațională standard pentru analiza şi proiectarea orientată pe obiecte.



De asemenea, am folosit o arhitectura de tip MVC de împărțire a claselor in pachete corespunzătoare rolului fiecăreia dintre acestea. In proiect se poate observa prezenta a trei pachete, models, view si controller. Pachetul „models” conține clasele Polinom, Monom si clasele utilizare Split si Operatii, unde am implementat metode statice pe care le-am folosit pentru parsarea String-ului introdus de utilizator, respectiv pentru efectuarea operatiilor pe structurile de tip polinom.

# Implementare

In cadrul acestei secțiuni a documentației, voi descrie in detaliu implementarea fiecarei clase din proiect. Toate clasele respecta principiile POO, in special incapsularea (variabilele-instanta sunt private), se regasesc elemente de polimorfism (suprascrierea metodelor).

**Clasa CalculatorView**

Contine implementarea efectiva a interfetei grafice, folosind elemente de tip Java Swing (JPanel, JTextField, JButton, JLabel). Clasa are ca variabile-instanta mai multe instante ale claselor mentionate anterior din pachetul Java Swing. Clasa CalculatorView mosteneste clasa JFrame, deci la crearea unui obiect de tipul acestei clase, se va deschide o fereastra pe care va se vor integra toate funcționalitățile aplicatiei. In constructor se instantiaza obiecte pentru fiecare variabila instanta sau se instantiaza si obiecte locale, pe care nu va fi nevoie sa le modificam in nici un scenariu de utilizare (i.e.: obiectele permanente de tip JLabel). Tot aici am setat si dimensiunile panelului pe care am adaugat restul componentelor, apeland de fiecare data metoda panel.add(<<*component>>*).

public class CalculatorView extends JFrame *{* private JPanel contentPane;  
 private JTextField textFieldPoli1;  
 private JTextField textFieldPoli2;  
 private JTextField textFieldRez;  
 private JButton btnAdunare;  
 private JButton btnScadere;  
 private JButton btnInmultire;  
 private JButton btnDerivare;  
 private JButton btnIntegrare;  
 private JButton btnImpartire;  
 private JLabel lblPoli2;  
 private JButton btnClr;  
  
 public CalculatorView*() {* setDefaultCloseOperation*(*JFrame.*EXIT\_ON\_CLOSE)*;  
 setBounds*(*100, 100, 480, 570*)*;  
 setLocationRelativeTo*(*null*)*;  
 contentPane = new JPanel*()*;  
 contentPane.setBackground*(*new Color*(*252, 241, 221*))*;  
 contentPane.setBorder*(*new EmptyBorder*(*5, 5, 5, 5*))*;  
 setContentPane*(*contentPane*)*;  
 contentPane.setLayout*(*null*)*;  
  
 JPanel panel = new JPanel*()*;  
 panel.setBackground*(*new Color*(*252, 241, 221*))*;  
 panel.setBounds*(*0, 10, 476, 550*)*;  
 contentPane.add*(*panel*)*;  
 panel.setLayout*(*null*)*;  
  
 textFieldPoli1 = new JTextField*()*;  
 textFieldPoli1.setBackground*(*new Color*(*255, 255, 255*))*;  
 textFieldPoli1.setFont*(*new Font*(*"Tahoma", Font.*PLAIN*, 14*))*;  
 textFieldPoli1.setBounds*(*148, 50, 300, 40*)*;  
 panel.add*(*textFieldPoli1*)*;  
 textFieldPoli1.setColumns*(*10*)*;  
  
 textFieldPoli2 = new JTextField*()*;  
 textFieldPoli2.setBackground*(*new Color*(*255, 255, 255*))*;  
 textFieldPoli2.setFont*(*new Font*(*"Tahoma", Font.*PLAIN*, 14*))*;  
 textFieldPoli2.setColumns*(*10*)*;  
 textFieldPoli2.setBounds*(*148, 110, 300, 40*)*;  
 panel.add*(*textFieldPoli2*)*;

De asemenea, am inplementat si cateva metode de tip setter si getter pentru a prelua String-urile introduse de utilizator in text field-uri, respectiv de a seta textul pentru caseta corespunzatoare afisarii rezultatului operatiei selectate de utilizator.

Ultimul tip de metode implementate in cadrul acestei clase sunt cele de tip add<<component>>Listener, care primesc ca parametru un obiect instanta a clasei ActionListener, asociind astfel fiecarui buton din cadrul interfetei grafice un „ascultator” care urmareste daca butonul este apasat de catre utilizator. Aceste metode vor fi apelate in constructorul clasei CalculatorController, unde se va implementa logica din spatele apasarii fiecarui buton.



**Clasa Monom**

public class Monom implements Comparable*<*Monom*>{* private double coeficient;  
 private int grad;

//…

Contine doua variabile-instanta care respecta principiul incapsularii (private, nu pot fi accesate direct din afara clasei, doar prin setteri si getteri): coeficient, de tip double, si grad, de tip int, modeland astfel forma unui monom din matematica (*coeficient*\*x^*grad*).

Tot aici am suprascris si metoda toString() pentru putea afisa un obiect de tip Monom sub o forma convenabila, tinand cont de cazurile particulare, cand grad=0 sau grad=1.

@Override  
public String toString*() {* String coef="";  
 int k=0;  
 if*(*coeficient==0*)* return "";  
 if*(*coeficient==-1 && grad!=0*)* return "-x"+*((*grad==1*)*?"":*(*"^"+grad*))*;  
 if*(*coeficient==1 && grad!=0*)* return "x"+*((*grad==1*)*?"":*(*"^"+grad*))*;  
 else coef=String.*valueOf(*coeficient*)*. replaceFirst("\\.0\*$|(\\.\\d\*?)0+$", "$1");

return *(*grad==0*)*?coef:*(*coef+*(*grad>0?*(((*coef.compareTo*(*""*)*==0*)*?"":"\*"*)*+*((*grad==1*)*?"x":"x^"+grad*))*:""*))*;  
*}*

De asemenea, am implementat o functionalitate pentru a renunta la „trailing zeros” in cazul coeficientilor intregi, care sunt stocati in variabile de tip double.

Pentru ca rezultatul final afisat pe interfata sa fie cat mai ordonat, clasa implementeaza interfata Comparable<Monom> si implicit, trebuie implementata si metoda compareTo(), care va veni in ajutorul metodei Collections.sort(); aceasta va fi apelata pentru a ordona monoamele din cadrul unui polinom.

**Clasa Polinom**

Are ca variabila instanta o lista cu obiecte de tip Monom, pentru a transpune imaginea matematica a unui polinom conform principiilor POO.

public class Polinom *{* private List*<*Monom*>* polinom;  
  
 public Polinom*()  
 {* polinom= new ArrayList*<*Monom*>()*;  
 *}* public void add*(* double coeficient,int grad*)  
 {* polinom.add*(*new Monom*(*coeficient,grad*))*;  
 *}* public void add*(* Monom monom*)  
 {* polinom.add*(*monom*)*;  
 *}*

Printre restul metodelor pe care le-am implementat in cadrul acestei clase se numara si doua metode *add()*, supraincarcare: prima primeste ca parametrii un coeficientul si gradul unui monom, prin intermediul carora creeaza un obiect de tip Monom ce va si adaugat in lista de monoame, iar cea de-a doua primeste un Monom existent deja in memorie, pe care doar il adauga in lista.

Am suprascris si metoda toString() pentru a putea afisa un polinom conform formei sale reale, astfel putand fi interpretat cu usurinta de utilizator. Desigur, am luat in considerare cazurile speciale, pentru ca forma finala a String-ului returnat sa fie una cat mai placuta vizual. De exemplu, in cazul coeficientilor negaviti se va afisa doar semnul `-`, nu `+-`, restul cazurilor particulare fiind deja tratate in detaliu pentru fiecare Monom in parte, simplifind astfel implementarea acestei metode.

**Clasa Split**

Este o clasa utilitate, final, din care nu se pot instantia obiecte, fara variabile-instanta, ce contine doar doua metode statice ce vin in ajutorul parsarii String-urilor date de utilizator si transformarea si stocarea lor in memorie sub forma de obiecte de tip Polinom.

Prima metoda, *modifyPoly()*, primeste ca parametru un String (introdus de utilizator) pe care il modifica astfel incat sa corespunda cu formula generala a polinomului (*anxn+ +an-1xn-1+…+a1x1+a0x0*). Se parcurge String ul caracter cu caracter si se formeaza un alt String auxiliar, in care se adauga treptat caracterele din cel initial, alaturi de anumite modificari pentru a simplifica operatie de separare in monoame. De exemplu, in cazul in polinomului „-x+1”, se va adauga -1 in loc de -, iar la `x` se va concatena „^1”. Similar vom proceda si pentru monomul 1 cu grad 0, adaugand „x^0”. In urma acestor modificari, String-ul va arata astfel: „-1x^1+1x^0”.

public static String modifyPoly*(*String poly*)* throws ExceptieFormat *{* if*(*poly.compareTo*(*""*)*==0*)* throw new ExceptieFormat*(*"Introduceti polinoame!"*)*;  
 char*[]* polyChar = poly.toCharArray*()*;  
 String aux="";

…

…

…for *(*int i = 0; i < polyChar.length - 1; i++*) {* if *(*i == 0 && polyChar*[*i*]* == 'x'*)* aux += '1';  
 if *(*polyChar*[*i*]* != '\*' && polyChar*[*i*]*!=' '*)* aux += polyChar*[*i*]*;  
 if *(*polyChar*[*i + 1*]* == 'x' && *(*polyChar*[*i*]* == '+' || polyChar*[*i*]* == '-'*))* aux += '1';  
 //if(i>0 && Character.isDigit(polyChar[i]) && (polyChar[i+1]=='+' || polyChar[i+1]=='-') && !aux.contains("^")) aux+="x^0";  
 if *(*polyChar*[*i*]* == 'x' && polyChar*[*i + 1*]* != '^'*)* aux += "^1";  
 *}*aux+=polyChar*[*polyChar.length-1*]*;  
*}*if*(*aux.charAt*(*aux.length*()*-1*)*=='x'*)* aux+="^1";  
if*(*Character.*isDigit(*aux.charAt*(*aux.length*()*-1*))* && aux.charAt*(*aux.length*()*-2*)*!='^'*)* aux+="x^0";  
System.*out*.println*(*"Dupa modificare:\n"+aux*)*;  
return aux;

Cea de-a doua metoda, *splitPoly(),* efectueza, dupa indica si numele, separarea polinomului in monoane. Initial, se vor inlocui toate caracterele `^` cu stringul gol, „” (practic se sterg), dupa care se va apela metoda *split()* pe String-ul returnat de metoda *modifyPoly()*. Metoda split() se apeleaza expresia regulata „*((?=*\\+*)*|*(?=*\\-*)*|x*)*”, ce evidentiaza faptul ca separarea se face fie prin `+`, fie prin `-`, fie prin `x`, rezultatul fiind un array de String-uri pe care l-am stocat intr-o lista cu obiecte de tip String. Aceasta va avea forma: coefn, gradn, coefn-1, gradn-1,…, coef0, grad0. Se parcurge apoi lista si se creeaza monoane din perechile de numere de pe pozitiile i, i+1 din lista de String-uri (se incearca transformarea lor in valori de tip double, respectiv int). In cazul in care parsarea nu este posibila, inseamna ca utilizatorul nu a introdus date corespunzatoare, iar metoda arunca o noua exceptie, pe care o vom intercepta in momentul in care vom apela metoda in clasa CalculatorController.

**Clasa Operatii**

Este o clasa utilitate de tip final in care am implementat metodele pentru operațiile intre polinoame. Toate metodele sunt statice.

Metoda *aduna()* primește ca parametrii doua polinoame si returnează un obiect de același tip, rezultat al adunării celor doi parametrii. Operația a fost implementata astfel: am creat doi iteratori pentru fiecare dintre cele doua liste din structura celor doua polinoame trimise ca parametrii. Am utilizat algoritmul de interclasare a doua liste pentru implementarea operației de adunare. Astfel, cat timp ambii iteratori nu au parcurs in întregime listele, se compara un monom din primul polinom cu unul din al doilea, adaugandu-se la rezultat cel cu gradul cel mai mare dintre ele si se trece mai departe in lista respectiva sau, in cazul in care gradele sunt egale, se adaugă un monom cu coeficientul egal cu suma coeficienților celor doua monoame menționate anterior si se trece mai departe in ambele liste. După ce unul dintre iteratori ajunge la finalul listei pe care o parcurge, se iese din while si restul monoamelor neparcurse ramase in cealaltă lista se adaugă in lista din polinomul-rezultat.

Metoda *scade()* funcționează pe același principiu ca adunarea, singura diferența fiind ca atunci când se întâlnesc doua monoame cu același grad, monomul rezultat din scăderea lor are coeficientul egal cu diferența coeficienților monoamelor. La final, după ieșirea din bucla while, daca au mai rămas monoame in lista celui de-al doilea polinom, acestea se vor adăuga in rezultat cu semn schimbat.

Metoda *inmulteste()* parcurge cele doua liste de monoame si se adauga la rezultat un monom cu coeficientul egal cu produsul coeficientilor monoamelor curente si cu gradul egal cu suma gradelor celor doua monoame. In urma acestui proces, lista-rezultat poate contine mai multe monoame cu acelasi grad, se trebuie comasate intr-unul singur. Astfel, pentru fiecare monom din rezultat se verifica daca in dreapta lui exista alte monoame cu acelasi grad, si daca da, se aduna coeficientul celui gasit la cel curent si se elimina.

Metoda *imparte()* retuneaza rezultatul impartirii a celor doua polinoame trimise ca parametrii ca instanta a clasei Tupla<Polinom, Polinom>, cu doua campuri, cat si rest. Algoritmul este cel clasic, de impartire a doua polinoame „pe hartie”. Cat timp restul are gradul mai mare sau egal decat impartitorul, se imparte primul monom din rest cu primul monom din impartitor, dar rezultatul se adauga la lista de monoane ,,cat” din „Tupla”. Se creeaza apoi un polinom auxiliar in care inseram doar monomul rezultat in urma impartirii si apelam metoda *inmulteste(Polinom impartitor, Polinom auxiliar),* dupa care se apeleaza scade() cu parametrii rest si polinomul returnat de metoda *inmulteste()*.

Metoda *deriva()* primește ca parametru un polinom, a carui lista de monoame o parcurge si adauga la rezultat un monom cu coeficientul egal cu produsul dintre grad si coeficient si cu gradul mai mic cu o unitate.

Metoda *integreaza()* este similara cu metoda *deriva()*, se face o parcurgere a listei de monoame si se adauga la rezultat un monom cu coeficientul egal cu raportul dintre coeficientul curent si gradul curent, si gradul mai mare cu o unitate.

**Clasa CalculatorController**

Face legatura intre interfata grafica (view) si forma datelor stocate in memorie (models). Clasa are ca variabila-instanta un obiect de tipul CalculatorView, pentru care se defineste un anumit comportament la interactionarea utilizatorului cu GUI. Am ales sa definesc actiuni pentru fiecare buton din view in controller, prin apelarea metodelor *add<<component>>Listener()*, folosind clase anonime care implementeaza interfata ActionListener si implicit, si metoda *action performed().* Am procedat astfel pentru fiecare buton din interfata.

De exemplu, in cazul apasarii butonului `+`, se procedeaza astfel: se preia textul introdus de utilizator in cele doua textField-uri destinate acestei operatiuni si este trimis ca parametru metodei polySplit() din clasa utilitara Split, rezultatele salvandu-se in obiecte de tip Polinom. Rezultatul este apoi afisat in textField-ul pentru rezultat. Toate aceste operatiuni sunt integrare intr-un bloc try-catch care va intercepta eventualele exceptii in cazul introducerii de String-uri necorespunzatoare si va avertiza utilizatorul prin aparitia unei ferestre de tipul JOptionPane cu un mesaj adecvat, acesta putand ulterior sa reintroduca date corecte. Restul butoanelor functioneaza la fel, iar butonul „Clear” are rolul de a reseta interfata in orice moment.



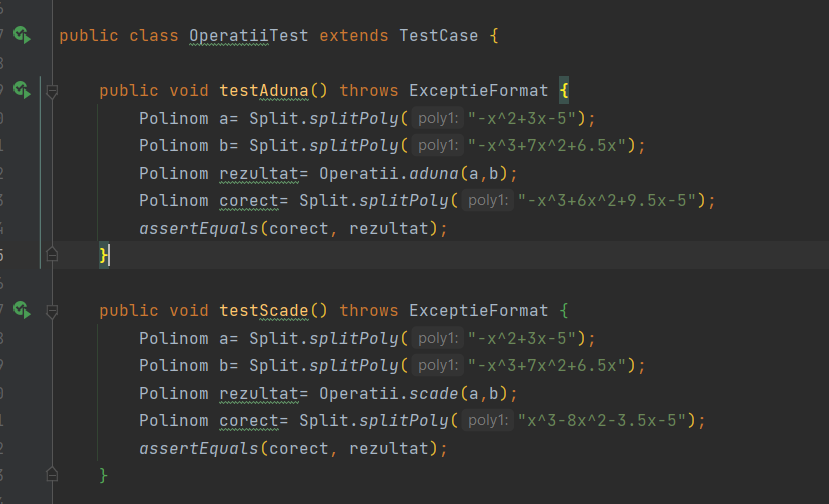
**Clasa Main**

In aceasta clasa am instantiat un obiect din clasa CalculatorView pentru a deschide fereastra ce contine interfata grafica. Acestui obiect i-am asociat un controller, instantiind si un obiect instanta a clasei CalculatorController prin trimiterea la constructor ca parametru obiectul de tip CalculatorView.

# Rezultate

Pentru testarea aplicatiei, am folosit framework-ul de testare regresiva JUnit, folosind asertiuni- metode de bază din clasa Assert utilizate pentru testare. Ordinea parametrilor dintr-o metodă de tip assert este, în general, parametru așteptat, parametru actual. De exemplu, pentru testarea metodei *aduna*, am luat doua obiecte de tip Polinom, pe care le-am parsat folosind metoda splitPoly() din clasa utilitara Split. Am comparat apoi rezultatul polinomul returnat de metoda *aduna* cu rezultatul așteptat, pe care l-am reținut intr-un alt obiect instanță a clasei Polinom.

Testele au trecut cu succes, programul având un oarecare grad de flexibilitate la nivelului modului de procesare a polinoamelor și parsare a expresiilor expectionale, dupa cum am precizat in sectiunea anterioara.



# Concluzii

Consider ca acest proiect m-a ajutat la aprofundarea cunoștințelor legate limbajul de programare Java, de paradigmele POO, dar si in implementarea cat mai buna a unei interfețe grafice pentru utilizator. Evident, programul poate fi îmbunătățit pe viitor pentru a avea acuratețe mai mare, GUI-ul sa fie mai „prietenos”. O alta imbunatatire ar putea fi extinderea posibilitatilor de parsare a String-urilor date de utilizator pentru ca aplicatia sa poata percepe si alte forme de polinoame, cum ar fi cele in care exista termeni de acelasi grad in cadrul unui polinom, fiind necesar sa ii comasam intr-unul singur.

# Bibliografie

materiale de la disciplina *Programare Orientata pe Obiecte*(semestrul 1)

https://regex101.com/

https://www.w3schools.com/java