Tema 1
Calculator de polinoame

Disciplina: Tehnici de programare

Mariciuc Andrei-Alexandru

Cuprins

Obiectivul temei

Observația de la care începe proiectul este timpul crescut și dificultatea de a efectua operații cu polinoame folosind tehnicile clasice: pixul și foaia. Astfel, dorim să dezvoltăm un sistem software capabil de a realiza operații precum: adunarea, scăderea, înmulțirea, împărțirea, derivarea și integrarea polinoamelor într-un timp foarte scurt. De asemenea, ne dorim ca modul de interacțiune cu sistemul să fie unul cât mai intuitiv și firesc, de aceea, implementarea unei interfețe grafice, capabilă să preia informații despre polinoame ușor și să furnizeze răspunsuri rapid, se concretizează în obiectivul principal al acestei teme. Alte obiective secundare pot fi reprezentate de următoarele:

- Analiza atentă a nevoilor unui presupus utilizator și modelarea unei soluții care să rezolve problema data *[cap. Analiza Problemei.*];*
- Proiectarea unei ierarhii de clase eficientă, care să faciliteze întreținerea aplicației și care să faciliteze înțelegerea codului de alți programatori *[cap. Proiectare]*;
- Dezvoltarea unor algoritmi de adunare, scădere, înmulțire, împărțire, derivare și integrare pentru polinoame folosindu-ne de definițiile matematice *[cap. Proiectare.Algortimi]*;
- Design-ul unei interfețe cu utilizatorul folosind API-ul Swing *[cap. Implementare]*;
- Implementarea unui pattern ținând cont de regulile Regex, pentru a extrage șirul de monoame din care este alcătuit un polinom *[cap. Proiectare.Algortimi]*;
- Afișarea polinoamelor într-o formă cât mai apropiată cu cea obișnuită în realitate folosind LaTeX *[cap. Implementare]*;
- Dezvoltarea unui sistem rudimentar de undo, care să permită vizualizarea istoriei intrărilor din calculator *[cap. Implementare]*;

Analiza problemei

Putem începe analiza prin identificarea unor cerințe funcționale fundamentale (“trebuie”) urmate de cele opționale („ar fi drăguț”):

- Aplicația trebuie sa lase utilizatorul sa insereze polinoame în variabila *x;*
- Calculatorul trebuie să permită introducerea sub o formă intuitivă => avem nevoie de o tastatură;

- Calculatorul de polinoame trebuie se permită efectuarea operațiilor de adunare, scădere, înmulțire, împărțire, derivare și integrare;
- Calculatorul trebuie să afișeze polinoamele rezultate în urma efectuării operațiilor;
- Ar fi bine dacă, putem revenii printr-un sistem *undo* la datele introduse anterior;
- Ar fi drăguț, un mod de afișare a polinoamelor cât mai apropiat cu notațiile clasice din matematică;
- Ar fi tare dacă, putem introduce datele de intrare și prin intermediului unor fișiere;

Modelare

Analiza bazată pe cerințele funcționale de mai sus conduce spre o reprezentare grafică a interacțiunii unui presupus utilizator cu sistemul descrisă în *figura 1*.

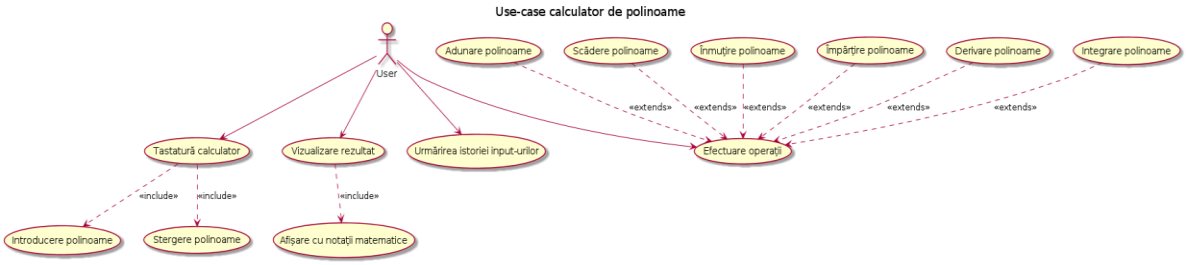


Figura 1 *Use-case calculator de polinoame*

Odată pornită aplicația, utilizatorul intră în contact direct cu toate funcționalitățile principale ale caculatorului de polinoame. El are posibilitatea de a introduce polinoamele, dar și modifica input-urile deja existente, prin intermediul unei tastaturi dedicate. De asemenea, poate vizualiza rezultatul operației alese (adunare, scădere, înmulțire, împărțire, derivare, integrare), fie în format clasic computațional (ex: 1+x^2+x^3+...), fie folosind notații matematice (ex: $1 + x^2 + x^3 + \cdots$). Aceste rezultate sunt furnizate la momente diferite în timp, dar se poate revenii la ele prin intermediul istoricului, care va furniza sub forma unei stive temporale, polinoamele predecesoare introduse de către utilizator

Scenarii

În continuare se vor prezenta două scenarii de utilizare, unul care va trata cazul fericit, iar altul, cazul mai puțin fericit de interacțiune a utilizatorului cu sistemul.

- Scenariul fericit - polinoame corect introduse:
    1. Utilizatorul introduce prin intermediul interfeței grafice, două polinoame corecte respectând formatul următor: $\sum_{i=0}^{n} a_n \cdot x^n \geq 0$;
    2. Utilizatorul selectează una din operațiile existente menționate mai sus (add, sub, mul, div, deriv, integ);
    3. Calculatorul efectuează calculul și afișează în interfața grafică rezultatul cerut de utilizator sub cele două formele discutate mai sus (notații matematice și notații digitale);
    4. Un caz special este cel în care utilizatorul vrea sa revină la unul din input-urile anterioare, el trebuie să dea click pe *textfield-ul*
- Scenariul mai puțin fericit - input greșit/efectuarea unor operații imposibile:
    1. Utilizatorul introduce un polinom cu format greșit (de exemplu: $\sum_{i=0}^{n} a_n \cdot xxx..x^n$ sau $\sum_{i=0}^{n} x^n a_n$ sau sum(a_n*x^^…^n)). Pentru a restrânge domeniul problemei, fără a afecta vreo funționalitate și în continuare aproximarea soluției probleme să rezolve problema propusă, utilizatorul nu va avea voie sa introducă alte caractere decât cele din mulțimea următoare {x, [0-9], [+-*/]}. Alt caz special fericit, este atunci când se încearcă apăsarea butonului de delete, când nu mai sunt caractere de șters sau împărțirea a două polinoame identice nule.
    2. Pentru cazurile descrise mai sus, interfața grafică va arunca o fereastră pop-up care va descrie problema întâmpinată (“polinom introdus greșit”, ”împarțire la zero”, ”tasta indisponibilă”), cu posibilitatea de a revenii în aplicație din starea în care a fost lasată printr-o singură apăsare de buton: ”OK!”.

Cazuri de utilizare

- Calculatorul poate fi folosit cu ușurință pentru aritmetica polinoamelor de grad superior în ipostaze educaționale și chiar industriale;
- Poate fii folosit de atât de studenți cât și de elevi pentru efectuarea temelor și proiectelor într-un timp cât mai rapid, sau pentru pregătirea pentru examene și teste;
- Poate fii folosit de ingineri/studenți pentru calculul unor integrale de tip $\frac{P(x)}{Q(x)}$;

Proiectare

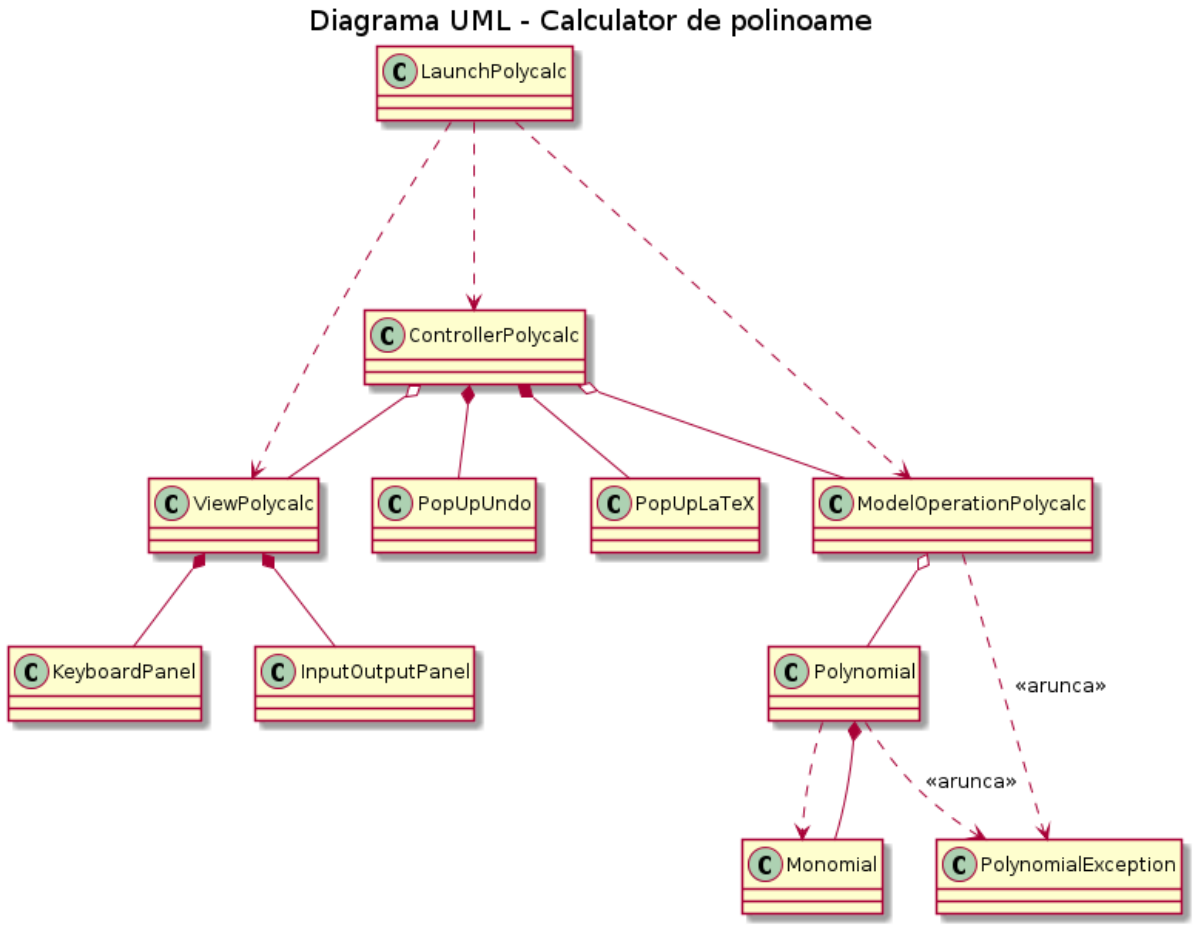
Pentru proiectarea unei soluții aleg să folosesc modelul MVC (Model View Controller). Mai întâi trebuie să proiectăm *modelul,* așadar, luând în considerare fundamentul teoretic matematic al unui polinom și având în vedere baza de reprezentare (B = $\{1, x, x^2, x^3, ..., x^n\}$) în spațiul vectorial al polinoamelor, pe care o folosim pentru această problemă, putem considera un polinom ca fiind o listă de monoame, unde un monom este de forma: $a_n * x^n$. Astfel gândind, este clară definirea unor clase pentru pentru aceste două entități, hai să le denumim *Polynomial* și *Monomial.* Considerând fundamentele teoretice din matematică, se distinge o relație de compoziție și de dependență între cele două clase: "*Polynomial este compus din Monomial și Polynomial nu poate exista fără Monomial*". Tot aici, trebuie să avem o entitate care să fie capabilă să opereze cu polinoamele pe care tocmai l-am descris. Hai să numim această clasă *ModelOperationPolycalc,* ea va fi responsabilă de toate operațiile pe polinoame pe care le poate efectua calculatorul nostru. Este adevărat că această clasă operează pe polinoame, însă între *ModelOperationPolycalc* și *Polinomial* este doar o relație de agregare, întrucât existența unui polinom nu este condiționată de existența celui care efectuează operații pe el.

Pentru *view*, avem nevoie de o tastatură pe care aleg s-o reprezint prin mai multe butoane, de două câmpuri de text - pentru introducerea operanzilor de tip polinom (text field) și de o etichetă (label), pentru afișarea rezultatului. Mai multe detalii se vor prezenta în *Proiectare.Interfață utilizator.*

Pentru tratarea excepțiilor definite în scenariul de utilizare mai puțin fericit, se va crea o clasa numită *PolynomailException*, aceasta va fi responsabilă pentru intercepția tuturor excepțiilor legate de de polinoame. În caz de nevoie, ea va fi aruncată atât de clasa *Polynomial* cât și de clasa *ModelOperationPolycalc.*

Diagrama UML

În cele ce urmează este prezentată diagrama UML, care pornește de la ideile din prima secțiune a acestui capitol. Detalierea metodelor și atributelor se va face în *Proiectare.Proiectare clase.*



Structuri de date

O structură de date foarte importantă este lista înlănțuită, care este folosită foarte frecvent din cauza bazei alese pentru reprezentarea polinoamelor. Aceasta a fost utilizata prin intermediul clasei *List<Type>* și printr-o extensie a sa, numită *ArrayList<Type>*.

O altă structură de date care a fost folosită, este reprezentată de arborii binari. Prin intermediul clasei *TreeMap<Key Type, Value Type>*, am putut considera *Key* ca fiind gradul polinomului, iar *Value* ca fiind coeficientul său. Astfel, munca depusă pentru adunare, scăderea și simplificarea polinoamelor a fost mult redusă și îmbunătățită din punct de vedere al performanței.

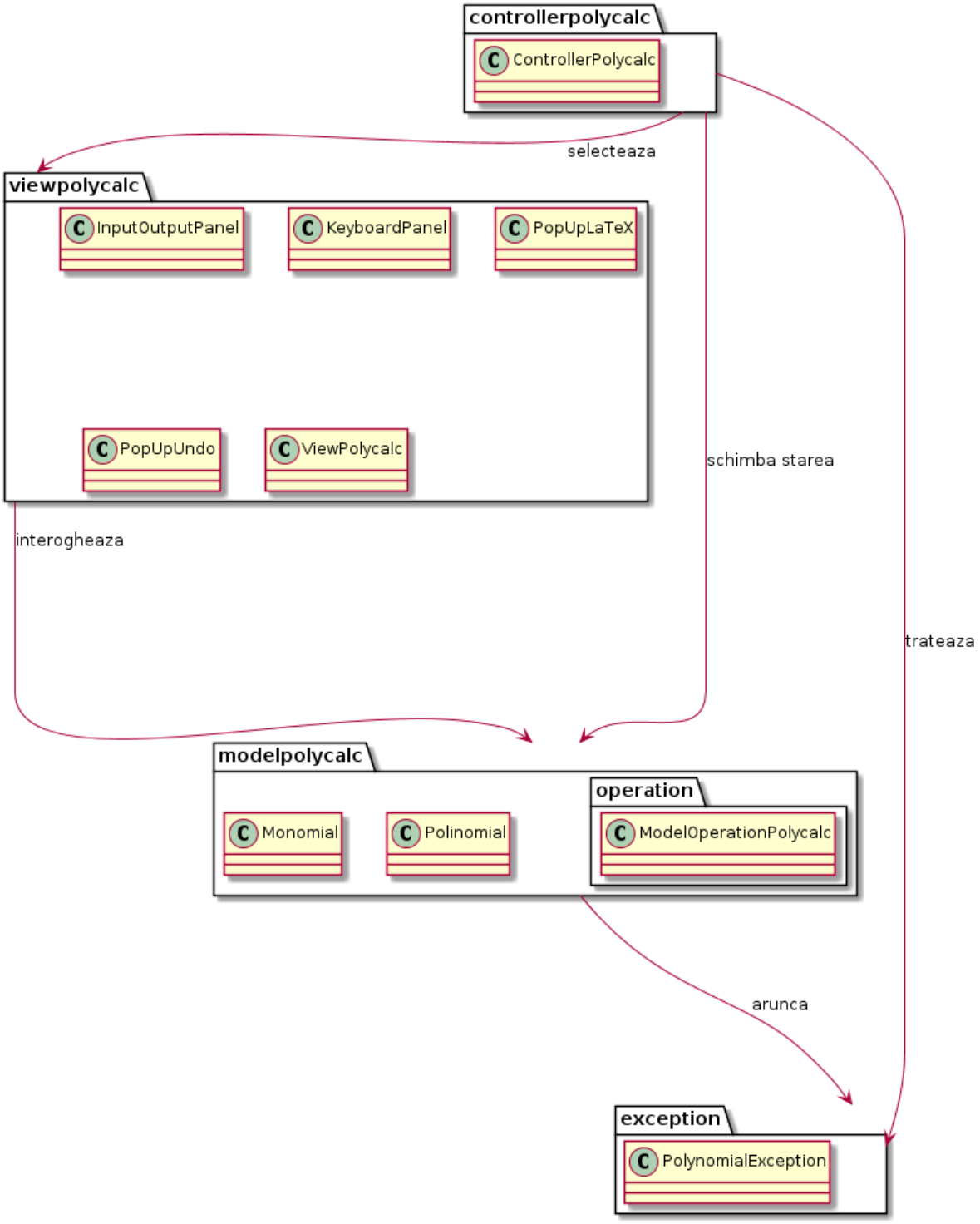
Relații

Relațiile din partea dreaptă a diagramei UML a claselor, au fost prezentate chiar prima secțiune a acestui capitol. În continuare vom discuta partea de View și Controller. Clasa *LaunchPolycalc* este responsabilă de lansarea aplicației, ea este dependentă de clasele *ViewPolycalc, ModelOperationPolycalc* și *ControllerPolycalc.* Clasa *ControllerPolycalc are ViewPolycalc* și *ModelOperationPolycalc,* adică se realizează astfel o relație de agregare între view și controller, respectiv model. Mai mult, în **compoziția** clasei *ControllerPolycalc* intră clasele responsabile de cerințele funcționale

" nice to have", și anume *PopUpUndo, PopUpLaTeX*. Pentru o mai bună structurare, clasa ViewPolycalc va fi **compusă** din două clase *KeyboardPanel (tastatura)* și *InpuOutputPanel (I/O polinoame)*.

Packages

Organizarea pachetelor a fost insipirată din modelul MVC, așa cum se poate observa în diagrama de mai jos:



Algoritmi

În continuare descriu principalii algoritmi folosiți în proiect.

1. **Algoritm pentru parsarea unui polinom în listă de monoame**. Acesta primește ca input un string și trebuie să genereze ca output o listă de monoame ordonată dupa grad, sau să arunce o excepție în cazul în care input-ul nu este un polinom definit corect.
    a. Despărțim problema în doi pași. Despărțirea în monoame și apoi simplificarea si sortarea dupa grad. Înainte de asta, ne vom asigura ca toate polinoamele introduse au coeficienti cu semn, așa încât testăm dacă primul element are sau nu semn, iar dacă nu are, îl adăugăm noi.

```
if (input.charAt(0) != '+' && input.charAt(0) != '-')
    input = "+" + input;
computeMonomial(input);
simplifiesArranges();
```

    b. Metoda *computeMonomial(<String>)* se folosește de expresiile de tip regex, astfel pentru început se împarte stringul de input considerând cazurile: unul din semnele +- urmat de kx^n sau x^n sau kx sau x sau k, unde x - necunoscuta, n - gradul monomului și k - coeficient. După ce facem match la acest pattern, încercăm să adăugăm monoamele găsite și înacelași timp să reconstituim input-ul. La urma, dacă input-ul nu este egal cu reconstituirea atunci se golește lista de monoame și se aruncă o excepție.

```
private void computeMonomial(String input) throws PolynomialException {
    String regexPattern = "[+-](\\d+x\\^\\d+|x\\^\\d+|\\d*x|\\d+)";
    Pattern pattern = Pattern.compile(regexPattern);
    Matcher matcher = pattern.matcher(input);

    String result = "";

    while (matcher.find()) {
        result += matcher.group();
        poly.add(new Monomial(matcher.group()));
    }

    if (!result.equals(input)) {
        poly = new ArrayList<>();
        throw new PolynomialException("Nu ai introdus un polinom corect!");
    }
}
```

c. Metoda *simplifiesArranges()*, se folosește de clasa *TreeMap<K, V>*, pentru a simplifica și ordona descrescător, în funcție de grad, lista de monoame concepută la pasul anterior. Funcția *merge*, va adauga dacă la pur și simplu un coeficient în drepul gradului, dacă nu mai exista vreun grad existent în tree, iar daca există se va folosi de funcția *sum* pentru a face merge intre coeficienți, adică, va efectua adunare pentru coeficienți corespondenți aceluiași grad. **Complexitate: O(nlogn)**

```
public void simplifiesArranges() {
    TreeMap<Integer, Double> buffer = new TreeMap<>();

    for (Monomial monom : poly)
        buffer.merge(monom.getDegree(), monom.getCoef(), Double::sum);

    poly = new ArrayList<>();
    for (var it : buffer.entrySet())
        if(it.getValue() != 0)
            poly.add(new Monomial(it.getValue(), it.getKey()));
}
```

2. **Algoritm pentru adunarea și scăderea a două polinoame.** Chiar dacă sunt operații diferite, am reușit totuși să le efectuez pe ambele în același timp. Pentru a calcula ușor coeficienții polinomului rezultant m-am folosit de aceași colectie *TreeMap*. Parcurgând fiecare lista de monome corespunzătoare polinoamelor de intrare, aceastea sunt introduse în tree folosind funcția *merge* ca mai sus. La urma parcurgem mulțimea de <grad, coeficient> rezultată și se creiază polinomul rezultat în urma operației. Operația este selectată de parametrul funcției *op*, daca *op = 1 => resutlPoly = input1 + input2, atlfel dacă op = -1 => resultPoly = input1 - input2.* **Complexitate: O(nlogn), n - max(gradul lui input1, input2)**

```
private Polynomial operationAddSub(Double op) {
    Polynomial resultPoly = new Polynomial();
    TreeMap<Integer, Double> buffer = new TreeMap<>();

    for (Monomial monom : input1.getPoly())
        buffer.put(monom.getDegree(), monom.getCoef()); // input1 si input2
sunt deja procesate!!

    for (Monomial monom : input2.getPoly())
        buffer.merge(monom.getDegree(), op * monom.getCoef(), Double::sum);

    for (var it : buffer.entrySet())
        if (it.getValue() != 0)
            resultPoly.addMonomial(new Monomial(it.getValue(), it.getKey()));
    return resultPoly;
}
```

3. **Algoritm pentru înmulțirea a două polinoame.** Se folosește aceeași idee ca la adunare și scădere, doar că aici se consideră toate combinațiile de monoame. **Complexitate: O((n*m)log(n+m)), n - gradul lui inpu1, m - gradul lui input2**

```
public Polynomial mul() {
    TreeMap<Integer, Double> buffer = new TreeMap();

    for (var monom1 : input1.getPoly())
        for (var monom2 : input2.getPoly())
            buffer.merge(monom1.getDegree() + monom2.getDegree(),
monom1.getCoef() * monom2.getCoef(), Double::sum);

    Polynomial resultPoly = new Polynomial();

    for (var it : buffer.entrySet())
        resultPoly.addMonomial(new Monomial(it.getValue(), it.getKey()));

    return resultPoly;
}
```

4. **Algoritm pentru derivarea unui polinom.** Se parcuge lista de monoame și se construiește un polinom de ieșire ținând cont de formula: $\frac{d(a_n * x^n)}{dx} = a_n(n-1)x^{n-1}$. **Complexitate: O(n), n - gradul lui input1**

```
public Polynomial deriv() {
    Polynomial resultPoly = new Polynomial();

    for (Monomial monom : input1.getPoly())
        resultPoly.addMonomial(new Monomial(monom.getCoef() *
monom.getDegree(), monom.getDegree() - 1));
    return resultPoly;
}
```

5. **Algoritm pentru integrarea unui polinom.** Se parcuge lista de monoame și se construiește un polinom de ieșire ținând cont de formula: $\int a_n x\text{^}n = a_n \frac{x^{n+1}}{n+1}$. **Complexitate: O(n), n - gradul lui input1**

```
public Polynomial integ() {
    Polynomial resultPoly = new Polynomial();

    for (var it : input1.getPoly())
        resultPoly.addMonomial(new Monomial(it.getCoef() / (it.getDegree() +
1), it.getDegree() + 1));
    return resultPoly;
}
```

6. **Algoritm pentru împărțirea a două polinoame.** Vrem să împărțim P(x) la Q(x), pentru asta trebuie să ne asigurăm că gradul(P) >= gradul(Q) și că Q nu este polinomul identic nul (dacă este, atunci vom arunca o excepție și un mesaj corespunzător). Cât timp P(x) nu este polinomul identic nul și gradul(P) >= gradul(Q), atunci luăm monomul cu gradul cel mai mare din P și monomul cu gradul cel mai mare din Q, împărțim aceste două monoame și rezultatul îl adăugăm în cât. Ne rămâne să înmulțim monomul obținut cu Q și rezultatul obținut să efectuăm o scădere P = P(x) - M(x) * Q(x), unde M(x) - monomul obținut la primul pas. Totul se repetă pentru noul polinom P(x) până ce condițiile nu mai sunt satisfăcute. La urmă în varibaila quo avem câtul iar în p avem restul. Se returnează o lista în care primul parametru este câtul și al doilea resutul.

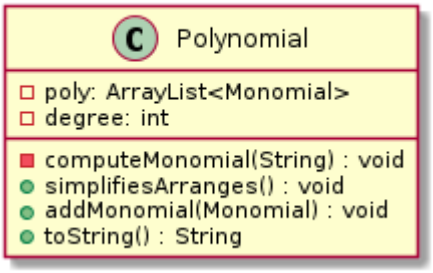
```
public List<Polynomial> div() throws PolynomialException {
    Polynomial p = input1;
    Polynomial q = input2;
    Polynomial quo = new Polynomial();

    if (q.getDegree() > p.getDegree()) {
        Polynomial aux;
        aux = p;
        p = q;
        q = aux;
    }

    if (q.toString().equals("0"))
        throw new PolynomialException("Impartire la zero a polinoamelor!");

    while (!p.toString().equals("0") && p.getDegree() >= q.getDegree()) {
        Monomial pMon = p.getPoly().get(p.getPoly().size() - 1);//monomul
mare lui p
        Monomial qMon = q.getPoly().get(q.getPoly().size() - 1);//monomul
mare lui q
        Monomial quoPart = new Monomial(pMon.getCoef() / qMon.getCoef(),
                pMon.getDegree() - qMon.getDegree()); // calculam catul prin
impartire!
        quo.addMonomial(quoPart);//adaugare la cat!
        Polynomial polQuoPart = new Polynomial();
        polQuoPart.addMonomial(quoPart);//cu asta o sa inmultim!
        Polynomial intermediarMul = new ModelOperationPolycalc(q,
polQuoPart).mul();//inmultirea intermediara
        p = new ModelOperationPolycalc(p, intermediarMul).sub(); //scaderea!
    }

    List<Polynomial> result = new ArrayList<>();
    result.add(quo);
    result.add(p);
    return result;
}
```

Implementare

Se prezintă în continuare toate clasele cu o scurtă descriere.

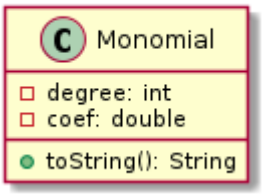
1. **class Polynomial** - se ocupă cu procesarea și stocarea unor polinoame primite sub formă de șiruri de caractere. Polinoamele sunt strocate sub formă de listă de monoame. Metoda *toString()* are grijă să afișeze frumos lista de polinoame.

Polynomail Class



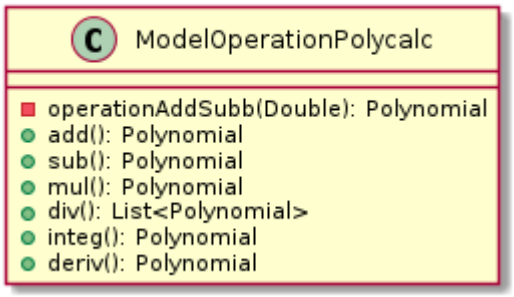
2. **class Monomial** - procesează un monom primit ca string și-l desparte în coeficient și grad ca variabile separate.

Monomial Class



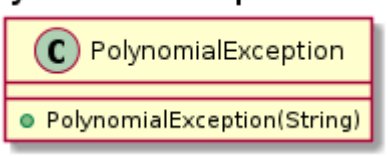
3. **class ModelOperationPolycalc** - metodele acestei clase au fost prezentate în secțiunea dedicată algoritmilor. Pe scurt, această clasa efectuează operațiile cerute de cerință asupra polinoamelor.

ModelOperationPolycalc Class

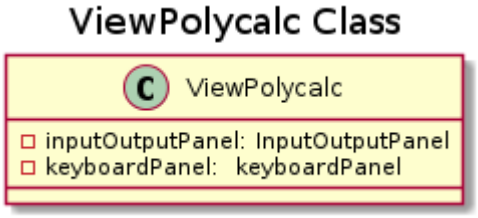


4. **class PolynomialException** - această clasă este cea care se va ocupa de aruncarea excepțiilor care pot intervenii în procesarea de polinoame și în efectuarea operațiilor pe polinoame.

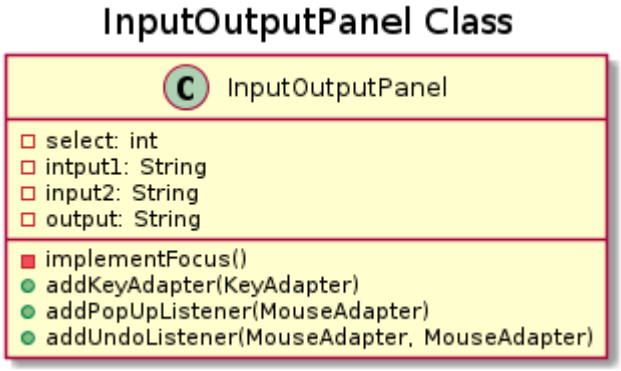
PolynomialException Class



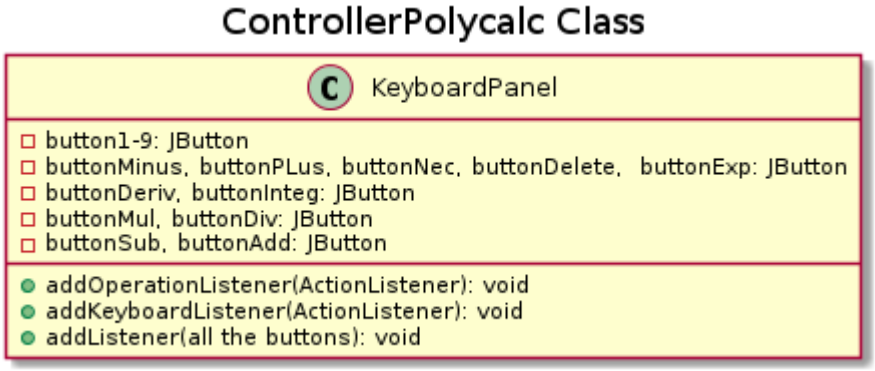
5. **class ViewPolycalc** - pentru a gestiona mai ușor interfața cu utilizatorul, am împărțit view-ul în două: tastatură și I/O. Clasa actuală conține două obiecte care vor fi reunite și afișate ca întreg.



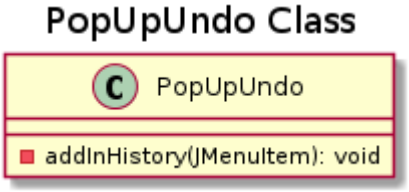
6. **class InputOutputPanel** - asigură view-ul întrărilor și ieșirilor calculatorului de polinoame. Are metode de adăugare de listeneri, care vor fi apelate de către controller, pentru toate componentele care se pot vizualiza în figura de mai jos.



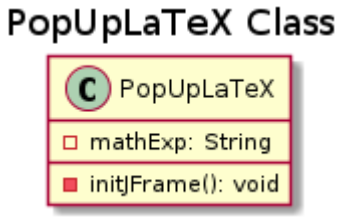
7. **class KeyboardPanel** - asigură view-ul tastaturii și are metode pentru adăugarea de listeneri, care de asemenea vor fi apelate de controller.



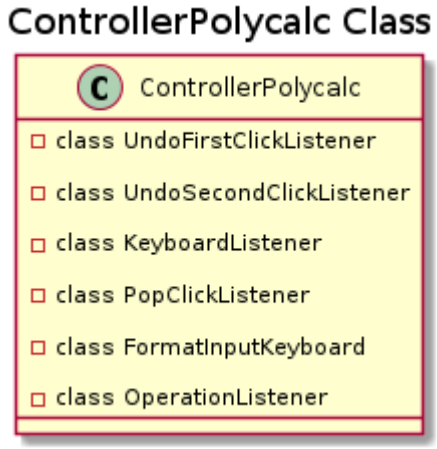
8. **class PopUpUndo** - cu ajutorul acestei clase se afișează istoricul celor două JTextField-uri din clasa InputOutputPanel.



9. **class PopUpLaTeX** - această clasă se folosește de libraria *jlatexmath* pentru a genera ecuații matematice pentru input-uri care respectă sintaxa LaTeX. Dacă utilizatorul dă click pe JLabel-ul de output, va avea posibilitatea de a genera o fereastră separată cu polinomul afișat conform cu notațiile din matematică.

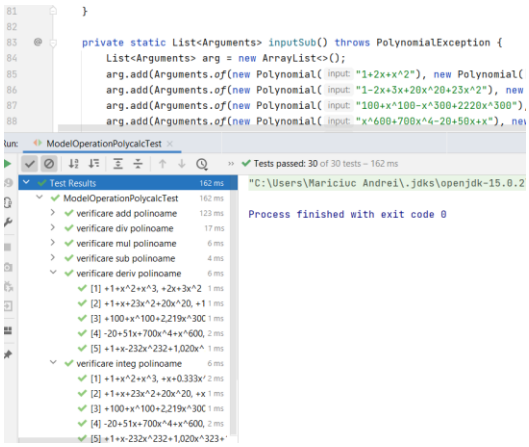


10. **class ControllerPolycalc** - tot controlul aplicației este concentrat în această clasă. Toți listenerii sunt definite ca inner classes în controller și trimise la view prin constructorul clasei.



Rezultate

Am testat folosind JUnit5 toate metodele din model. Testele au fost generate folosind matlab, pentru a mă asigura că nu greșesc la calcule manual. Am testat atât operațiile cât și transformarea unui polinom din string în lista de monoame. Majoritatea testelor au fost parametrizate și am considerat câte 5 cazuri pentru fiecare metoda.



Concluzii

Voi prezenta câteva din lucrurile pe care le-am câștigat prin intermediul temei:

- să lucrez cu pattern-uri regex;
- am aprofundat modelul MVC;
- să lucrez cu PlantUML editor, întrucât toate diagramele din documentație au fost procesate prin intermediul unui limbaj, prezent în anexele atașate cu documentația;
- am aprofundat colecțiile precum *TreeMap<>*;
- mi-am reamintit cum se divid două polinoame;
- am învățat tehnici de a scrie cod cât mai condensat;
- am aprofundat metodele de testing cu JUnit5;
- m-am inițiat librăriei jlatexmath;

Dezvoltări ulterioare:

- Posibilitatea încărcării din fișiere text și salvarea output-ului în fișiere text. Asta ar putea da posibilitatea de a procesa polinoame mult mai mari;
- Implementarea unor funcții noi precum determinarea rădăcinilor;
- Afișarea graficului polinomului într-un sistem de coordonate cartezian;
- Procesarea polinoamelor de mai multe variabile;
- Descompunerea în produs;

Biliografie

1. https://docs.oracle.com/javase/7/docs/api/java/text/NumberFormat.html
2. https://plantuml-editor.kkeisuke.com/
3. https://regexone.com/
4. https://regexr.com/
5. https://jar-download.com/?search_box=jlatexmath
6. https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html