**DOCUMENTAȚIE TEMA 1**

**CALCULATOR POLINOAME**

**Student: Raul - Alexandru Gorgan**

**Grupa: 30227**

**Profesor Laborator: Dan Mitrea**

Contents

[1. Cerinte Functionale 3](#_Toc476131445)

[2. Obiective 3](#_Toc476131446)

[2.1. Obiectiv Principal: 3](#_Toc476131447)

[2.2. Obective Secundare: 3](#_Toc476131448)

[3. Analiza Problemei 3](#_Toc476131449)

[4. Proiectare 3](#_Toc476131450)

[4.1. Structuri de date 3](#_Toc476131451)

[4.2. Diagrama de clase 3](#_Toc476131452)

[4.3. Algoritmi 3](#_Toc476131453)

[5. Implementare 4](#_Toc476131454)

[6. Testare 4](#_Toc476131455)

[7. Concluzii si Dezvoltari Ulterioare 4](#_Toc476131456)

[8. Bibliografie 4](#_Toc476131457)

# Cerințe Functionale

Se cere implementarea funcțiilor de bază pe polinoame, pentru a evidenția modul în care acestea pot fi create și manipulate prin intermediul paradigmei programării orientate pe obiecte. Printre operațiile de bază se numără cele de adunare, scădere, înmulțire și împărțire, dar și cele de derivare și integrare. Aplicația trebuie să poată interacționa cu utilizatorul într-un mod cât mai ușor pentru acesta, printr-o interfață grafică: utilizatorul scrie cele două polinoame, selectează operația dorită iar rezultatul îi este afișat pe ecran.

# Obiective

## Obiectiv Principal:

Obiectivul principal al acestui proiect este de propunere, proiectare și implementare al unui calculator de polinoame de o singură variabilă si cu coeficienți intregi. Acesta este o unitate de calcul care primește la intrare două polinoame, efectuează operații asupra acestora și afișează la ieșire polinomul rezultat în urma operațiilor. Aplicația trebuie integrată într-o interfață grafică cu utilizatorul, pentru a face mai ușoară și mai atractivă comunicarea.

## Obective Secundare:

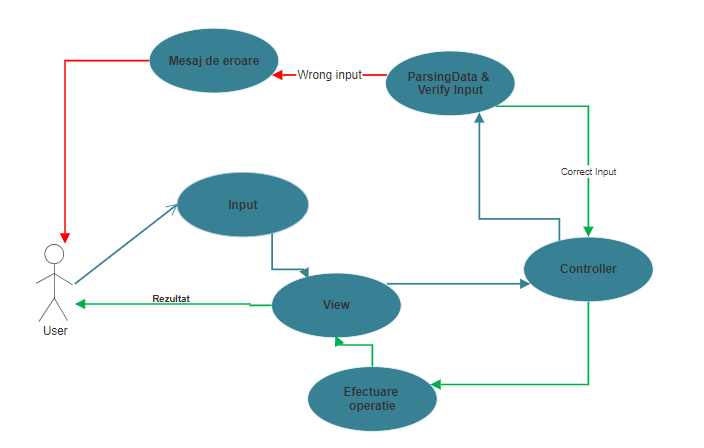
Pentru a putea îndeplinii obiectivul principal al acestui proiect, este nevoie de stabilirea unor obiective secundare care ne vor conduce la realizarea lui. Stabilirea acestora este importantă înainte de implementarea unei aplicații, deoarece o proiectare bună poate aduce multe beneficii, atât pentru proiectant cât și pentru utilizator. O bună alegere a soluției poate face ca aplicția să fie mai ușor de scris, întreținut, chiar și de promovat. Utilizatorul dorește să aibă o aplicație care efectuează calculele într-un timp minim, să fie user-friendly, să lucreze ușor și intuitiv. Aceste avantaje pot fi rezultatul unei analize în detaliu și a unei proiectări de calitate.

Un obiectiv secundar care intră în discuție la acest proiect este analiza problemei prin dezvoltarea de use case-uri și de scenarii, care vor fi prezentate în capitolul numărul 3. Acesta ne va ajuta la indeplinirea unui alt obiectiv secundar și anume alegerea structurilor de date folosite, alegerea fiind argumentată la capitolul 4. Tot la capitolul 4 vor fi dezvoltate și celelalte obiective secundare: împărțirea pe clase, respectând paradigmele programării orientate pe obiecte și dezvoltarea algoritmilor esențiali. După analiza în detaliu a problemei, la capitolul 5 se descrie pe larg implementarea soluției, aceasta fiind urmată de faza de testare, ultimul obiectiv secundar în vederea finalizării aplicației.

# Analiza Problemei

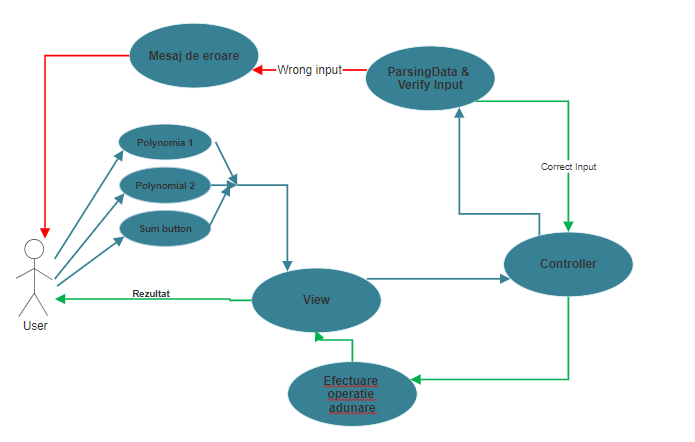
Analiza problemei începe cu identificarea use case-urilor și a scenariilor. Niciodata nu garantăm că utilizatorul o să introducă datele exact în formatul specificat de noi sau o să urmeze pașii în ordine, așa că este nevoie de analiza tuturor cazurilor de utilizare și a cazurilor limită. Aplicația trebuie să acopere toate aceste cazuri și să afișeze mesaje sugestive utilizatorului.

Diagramele de use case sunt o variantă vizuală a pașilor care se parcurg la efectuarea unei operații în aplicație.



Aceasta este diagrama generală de use case. Parcursul albastru e comun celor două scenarii (scenariul bun și scenariul alternativ). Dacă inputul e corect, parcursul va urma parcursul verde, dacă nu, va urma parcursul roșu.

Se pot define use case-uri pentru fiecare operație. Pentru operația de adunare, avem următorul use case:



Use case: adunarea polinoamelor

Actorul principal: utilizatorul

Scenariul de success:

1. Utilizatorul introduce cele două polinoame (în formatul cerut) în text-field-urile corespunzătoare.
2. Utilizatorul alege operația de adunare, prin click pe butonul „Sum” din interfața grafică.
3. Aplicația extrage polinomii, calculează rezultatul și îl afișează în interfața grafică, în tex-field-ul corespunzător rezultatului.

Scenarii alternative:

* Utilizatorul introduce greșit polinoamele, de exemplu: introduce două sau mai multe variabile, caractere greșite și care fac imposibilă împărțirea stringului în monoame.
* Utilizatorul scrie doar un polinom, lasă un polinom necompletat și alege o operație la care este nevoie de două polinoame.
* Utilizatorul lasă ambele polinoame necompletate și alege operația, caz în care, orice operație ar selecta, aceasta nu se va putea efectua
* În toate aceste cazuri, va fi aruncată o excepție definită de programator, iar execuția aplicației va continua întorcându-se la pasul 1.

Pentru operațiile de scădere, înmulțire, derivare (un singur polinom) și integrare (un singur polinom), scenariile vor fi asemănătoare.

La operația de împărțire, se adaugă un scenariu alternativ:

* Utilizatorul alege funcția de împărțire a polinoamelor dar polinomul al doilea este „0” sau are gradul mai mare decât al primului. (va fi aruncată o exceptie definită de programator).

# Proiectare

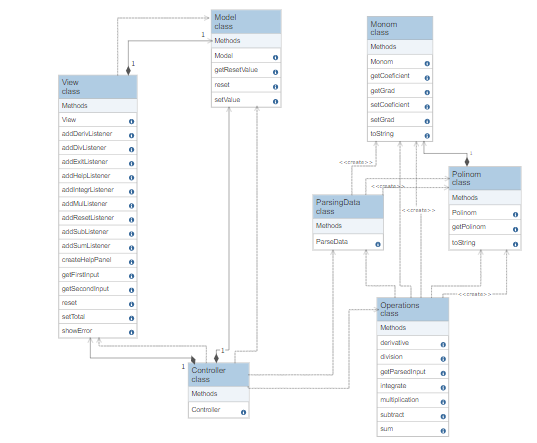
## Structuri de date

Pornind de la cerința și de la analiza problemei, sistemul are ca intrări două polinoame și o operație, iar ca ieșire un polinom rezultat, fiind nevoie în implementare de două obiecte: un obiect monom, care are câmpurile coeficient și grad și un obiect polinom, care are o listă de monoame. Astfel, instanțele de clase Monom și Polinom reprezintă obiectele cu care se va lucra. În clasa monom este supraîncărcată metoda toString deoarece i-am adaugat un parametru boolean ca să specific dacă vreau afișarea coeficientului cu zecimale sau fără. Astfel, varianta String a polinomului constă în concatenarea versiunii String a fiecarui monom din listă. La implementarea listelor de monoame din cadrul unui polinom se foloseste ArrayList. Acest mod general de a privi lucrurile reprezintă primul nivel în design-ul aplicației.

## Diagrama de clase

Organizarea claselor în pachete îl reprezintă nivelul al doilea de design. Deoarece aplicația este proiectată după modelul arhitectural Model-View-Controller, vor fi trei pachete care se ocupă de buna funcționare: pachetul pentru modele, pentru logica aplicației și pentru interfața grafică cu utilizatorul. Acest tip de structură oferă multe avantaje de proiectare, fiind mult mai ușor de intreținut.

Nivelul al treilea în design-ul unei aplicații îl reprezintă împărțirea pachetelor în clase. Acest lucru trebuie făcut după o analiză a necesităților și a resurselor. Prin urmare, pachetul Model conține clasele Monom și Polinom, fiecare într-un subpachet, și clasele Model, Operations și ParsingData. Pachetul UI contine partea care se ocupă de GUI și partea care controlează întreg sistemul. Prin urmare, clasele rezultate pot fi vizualizate într-o diagramă de clase:



## Algoritmi

După ce s-a realizat împărțirea în clase, design-ul problemei continua cu analiza metodelor. Trebuie proiectați algoritmii folosiți la calcule și anume, algoritmii de adunare, scădere, înmulțire, împărțire a două polinoame și operațiile de derivare și integrare a unui polinom.

**Algoritmul de adunare** a două polinoame constă în identificarea monoamelor de grad egal și adunarea coeficienților acestora, iar monoamene care nu au corespondent, se trec direct în rezultat. Am optat pentru un algoritm care se bazează pe if-uri, tot procesul petrecându-se într-o buclă while. Parcurg listele celor două polinoame cu iteratori. Iterația la elementul următor se face doar pe baza unor condiții.

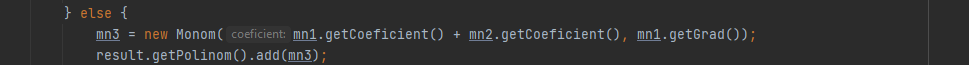
Dacă gradul primului monom e mai mare decât gradul celui de-al doilea, atunci în rezultat se adaugă primul și acesta se și iterează.



Altfel, dacă gradul celui de-al doilea e mai mare, atunci al doilea monom se adaugă la rezultat și acesta se iterează.

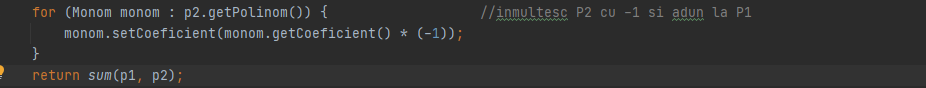


Altfel, înseamnă că cele două monoame au grad egal. Se creează un nou monom cu gradul acela și coeficientul fiind suma celor două. Noul monom se adaugă la rezultat.

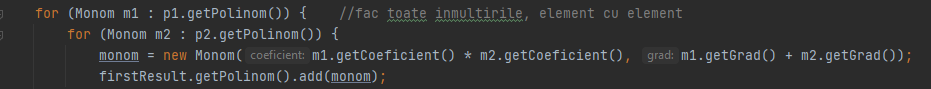


Logica principală se află în aceste linii de cod, restul fiind doar verificări că mai există elemente în liste.

**Algoritmul de scădere** nu este altceva decât o adunare cu polinomul al doilea înmulțit cu -1.



**Algoritmul de inmulțire** face înmulțirea fiecărui monom din primul polinom cu fiecare monom din al doilea.



Apoi face sume succesive între elementele polinomului rezultat, pentru ca la final să fie un polinom valid (toate elentele sunt cu grad diferit).

**Algoritmul de împărțire** constă în câțiva pași succesivi:

* se împarte primul monom al pui p (primul polinom) la primul monom al lui q (al doilea polinom) iar rezultatul se adaugă la cât.



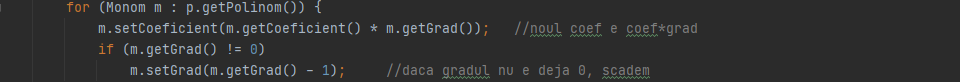
* acum se obține restul: se înmulțește câtul cu q iar rezultatul se scade din p.



* Se repetă acești pași, noul p fiind restul. Se continuă până când gradul restului e mai mic decât gradul lui q.

**Algoritmul de derivare** constă în parcurgerea listei de monoame, iar fiecărui monom:

* Noul grad este actualul grad - 1 (doar dacă e mai mare decât 0).
* Coeficientul devine actualul coeficient înmulțit cu gradul.



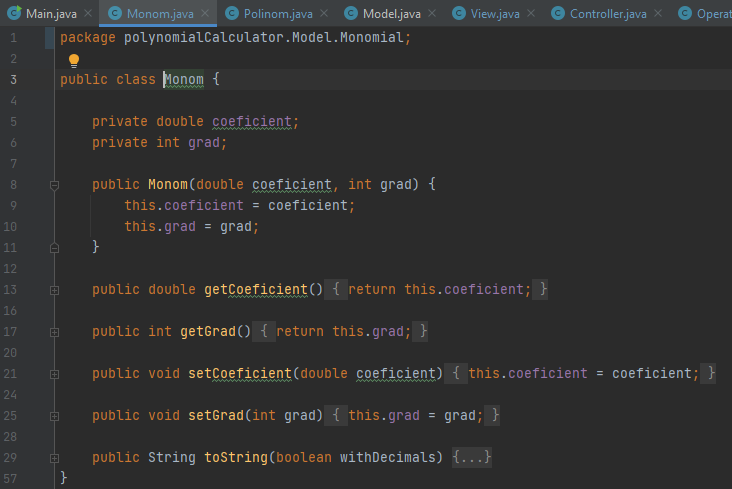
**Algoritmul de integrare** constă și el în parcurgerea listei:

* La grad se aduna 1.
* Coeficientul se împarte cu grad + 1.



# Implementare

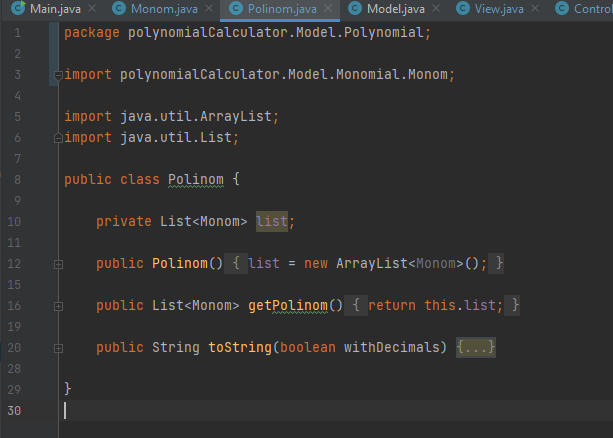
În urma analizei problemei și a implementării design-ului, au reieșit următoarele clase: Monom și Polinom, care reprezintă structurile de date ale aplicației, Model, View, Controller, care reprezintă pattern-ul arhitectural al aplicației, ParsingData, o clasă care conține o metodă de parsare a inputului în monoame, Operations, în care sunt implementate operațiile pe polinoame și clasele IllegalDivisionException, NoInputException și WrongInputException, care sunt exceptii definite de programator.

**Clasa Monom**

Această clasa are ca variabile instanță coeficientul si gradul, coeficientul fiind double, dar afișez convertit la int la operațiile care nu dau rezultat cu virgula și cu două zecimale la operațiile division și integrate. Grad este de tip int și datorită regex, doar pozitiv. Metodele clasei, pe lângă constructor, sunt gettere si settere și toString. Această metodă toString este supraîncărcată deoarece primește un parametru boolean cu care aleg daca să afișez cu zecimale sau nu coeficientul.

**Clasa Polinom**

Această clasă modelează un polinom, care are ca variabilă instanță o listă de monoame, implementată cu ArrayList. Aceasta e o metoda mai ușoară de a manipula o colecție de date decât vector. Se pot parcurge listele cu foreach și se poate adăuga și șterge cu un efort computațional mai mic.



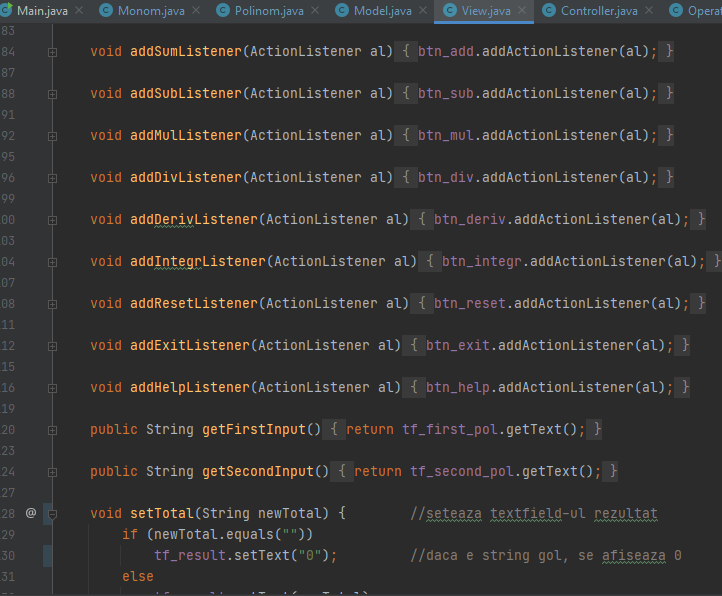
Această clasă conține și ea medota toString cu paramentru, pe langă construnctor și getter pentru listă. Metoda toString concatenează stringurile fiecărui monom, rezultând polinomul.

**Clasa Model**

Clasa Model reprezintă modelul aplicației, care comunică cu view. Este una dintre cele 3 componente ale pattern-ului arhitectural folosit (Model-View-Controller). Câmpurile clasei sunt o constantă RESET\_VALUE, care se pune în view la polinomul rezultat atunci când se face reset și inițializare și un BigDecimal displayed\_value de care mă folosesc la inițializarea valorii din model, prin functia setValue și la resetare. Folosesc gettere și settere pentru a putea utiliza valorile în view.

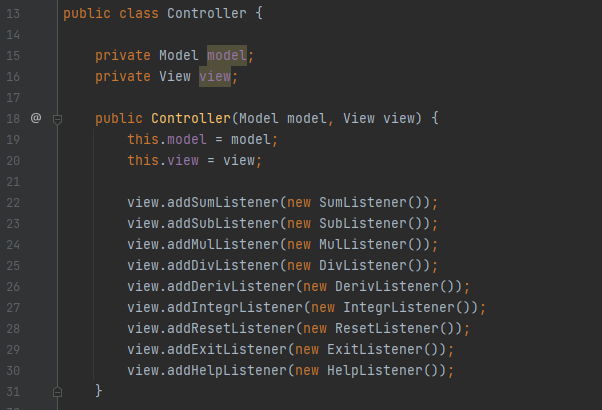
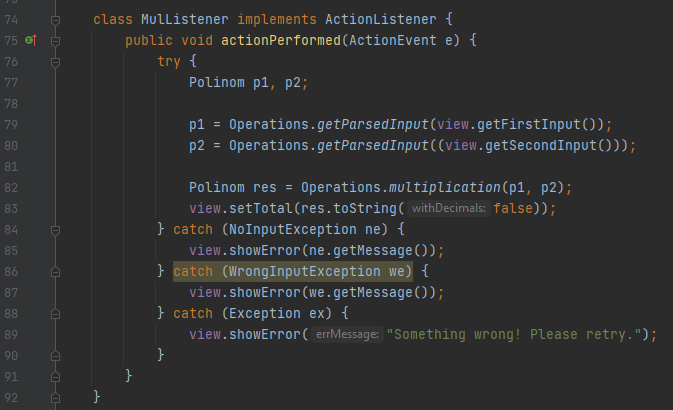
**Clasa View**

View este a doua componentă din pattern-ul architectural, care comunică cu modelul și realizează interfața grafică cu utilizatorul. Câmpurile acestei clase sunt reprezentate de o instantă de model și toate componentele din interfață: butoane, labels, text-fields etc.

Cea mai importantă metoda este constructorul, în care practic se construiește interfața, adăugându-se componentele. Apoi metodele prin care se adaugă un actionListener pe butoane sunt importante deoarece fără ele, butoanele nu ar face nimic. Acestea fac legătura intre Controller-ul aplicației, în care sunt implementate aceste acțiuni pe butoane și apăsarea acestuia in view. Exemple de metode în care se adaugă actionListener: addSumListener, addDerivListenr etc. Se observă și metodele prin care se iau polinoamele date de utiizator în view și metoda prin care se scrie rezultatul în urma operației dorite.

Această clasă va fi prezentată în detaliu la sfârșitul acestui capitol.

**Clasa Controller**

Clasa Controller este a treia componentă a pattern-ului arhitectural. Instanța clasei Controller comunică atât cu modelul cât și cu view. La nivel de câmpuri, clasa Controller are un model și un view, pe care le ia ca parametri în constructor. Această clasă conține câte o clasă internă pentru fiecare buton din view. Se face acest lucru ca să se poată adăuga actionListener pe buton și să se efectueze orice operație se dorește. În constructor se apelează, pentru ficare buton, metoda prin care se adauga actionListener, trimitând ca parametru o instanță a clasei interne care implementează metoda actionPerformed.

În această clasă se folosește toată logica aplicației, care se găsește în clasele Operations și ParsingData. Fiecare buton are atașată o acțiune (o operație din clasa Operations).

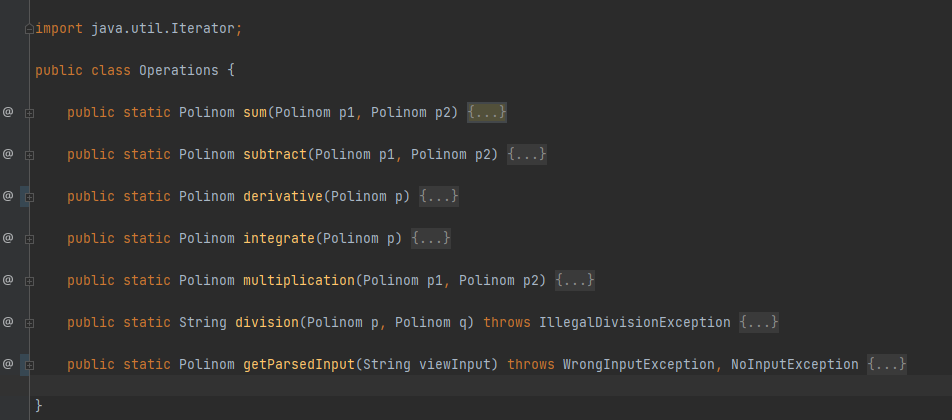
Clasele interne din Controller sunt implementate în stilul pozei numărul doi de mai sus. Aceasta este clasa care descrie evenimentul și ce se va întâmpla la apăsarea butonului. Totul se pune într-un bloc try-catch pentru a intercepta orice exceptie care apare și pentru a evita prabușirea aplicației. Mai întâi, se formează polinoamele din cele 2 intrări cu funcția getParsedInput. Această funcție apelează funcția parseData, prin care se formează monoamele și se aruncă eventualele exceptii cu privire la introducerea greșită a polinoamelor sau alte exceptii. Apoi se face operația dorită, în cazul de față, înmlțirea. Se scrie rezultatul în view. Acesta este scenariul bun, dar dacă apar unele excepții, acestea vor fi interceptate de blocurile catch.

Clasa Controller mai are pe lângă clasele interne pentru operații și clase interne care implementează acțiunea butoanelor de reset pentru aplicație, de închidere a aplicației și de afișare pop-up a unui textArea cu informații despre modul de folosire a aplicației.

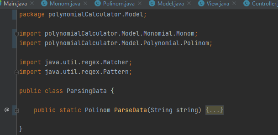
**Clasa Operations**

Aceasta e o clasă care conține implementarea tuturor operațiilor pe polinoame: adunare, scădere, înmulțire, împărțire, derivare și integrare. Metodele sunt statice ca să fie chemate pe clasă, să nu se instanțieze obiect de clasa Operations. Algoritmii pe care se bazează aceste metode sunt prezentați în capitolul 4.3.

Pe lângă operațiile de calcul cu polinoame, clasa mai are o metodă, numită getParsedInput. Această metodă primește la intrare conținutul unui textfield din view și returnează polinomul aferent. În această metodă se apelează metoda statică din clasa ParsingData, care practic face parsarea.

****

**Clasa ParsingData**

**** Această clasă conține metoda statică de parsare a intrării. Se ia polinomul tastat de utilizator și se trimite ca parametru acesti funcții. Dacă totul decurge în scenariul bun, intrarea este împărțită în monoame, iar funcția returnează polinomul rezultat.

În această metodă, împart stringul de intrare în polinoame cu ajutorul unui regex, un pattern pentru input, care împarte în trei grupuri plus grupul 0 (intregul monom). Grupul 1 conține semnul coeficientului (+, - sau null dacă este primul termen din monom și nu i s-a dat explicit semn). Grupul 2 conține coeficientul și caracterul x la urmă. Pentru a lua coeficientul, iau tot grupul în afară de ultimul caracter (x). Grupul 3 conține „^grad”, iar pentru a lua gradul iau stringul incepând de la al doilea caracter. Aici sunt 2 cazuri alternative când e null, dacă gradul e 0 sau 1. Verific acest lucru uitându-ma iar la grupul 2 și verificând dacă conține x la final, dacă da, gradul e 1 altfel e 0.

Verificarea că s-a introdul un string greșit nu e în această metodă, dar e într-o metodă din clasa Operations (getParsedInput) care se bazează pe aceasta.



**Clasele de excepții**

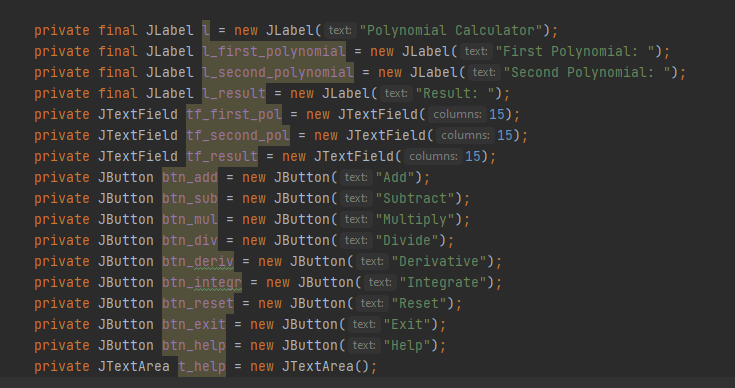
Sunt trei clase de excepții: IllegalDivisionException care este aruncată când polinomul al doilea de la împărțire e 0 sau are grad mai mare decât primul; NoInputException care e aruncată când se dă string gol la intrare; WrongInputException, aruncată când se dă un input greșit.

**Interfața Grafică cu Utilizatorul**

Pentru realizarea interfeței am folosit Java Swing. Panoul principal este de tip BoxLayout, aranjate pe axa y deoarece este compus din mai multe elemente, în vederea unei aranjări optime și atractive. Panoul principal (numit content) este format practic doar dintr-un singur panou, iar celelalte elemente nu sunt decât porțiuni goale (rigid areas) pentru a da un aspect mai bun aplicației. Panoul principal conține numele aplicației, centrat și mărit. Apoi, urmează o panoul cu aranjare GridLayout de 10 pe 2, în care am integrat toate elementele care folosesc la aplicație.

Am adăugat labels, precum: „First polynomial”, „Second Polynomial”, „Result”. În dreptul fiecărui label am pus câte un textfield, unde utilizatorul introduce cele 2 polinoame și vede rezultatul. Apoi am adăugat pe rând fiecare buton: Add, Subtract, Multiply, Divide, Derivative, Integrate, Reset, Exit, iar după introducerea unui spațiu rigid. Am adăugat și butonul Help.

Acestea sunt elementele care compun interfața:



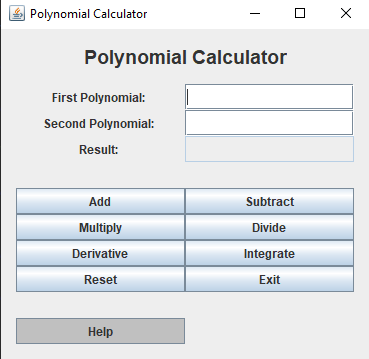
Panoul care conține elementele e definit astfel:



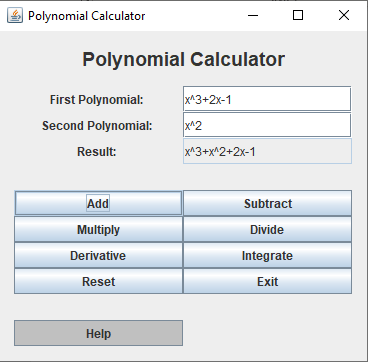
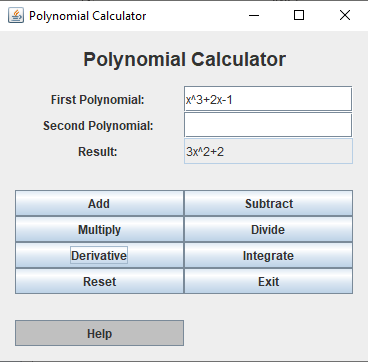
Panoul principal, care conține și spațiile goale, pentru aspect este definit astfel:



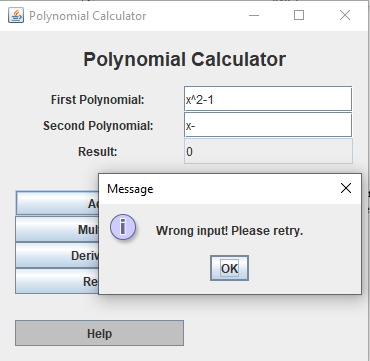
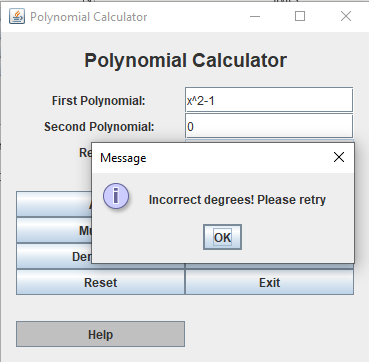
Aplicația arată în felul următor:

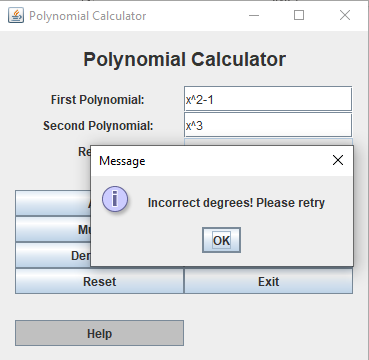
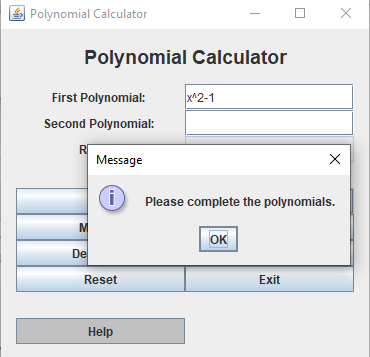


Textfield-ul de la rezultat e setat a nu fi editabil. Dacă scriem 2 polinomi în cele 2 textfield-uri, în al treilea vom avea rezultatul. Dacă facem operația de adunare respectiv derivare:

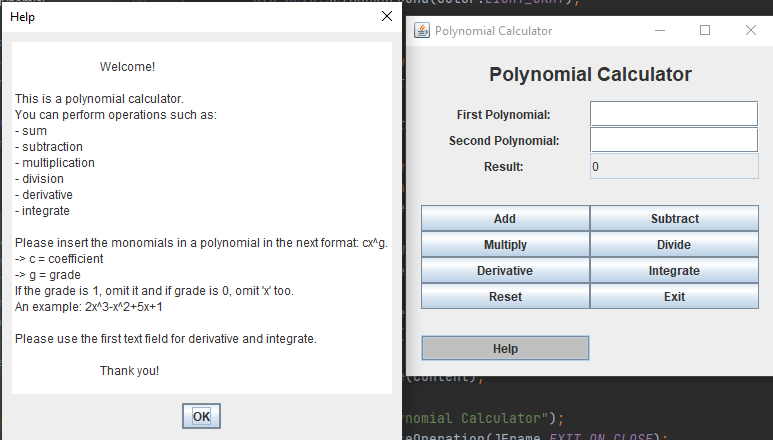
 

La introducerea greșită a polinoamelor pot apărea mai multe mesaje de informare. Dacă se introduce un polinom greșit avem situația din prima poza, dacă se încearcă împărțirea cu 0, situația din a doua poză, dacă se incearcă impărțirea cu grad mai mare la al doilea polinom, poza 3, iar dacă nu se scrie polinomul, poza 4 (rezultatul 0 e dat la apăsarea butonului Reset):

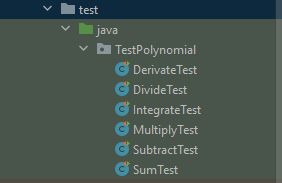
 

La apăsarea butonului Reset, se șterge conținutul textfield-urilor celor 2 polinoame și se pune 0 pe rezultat (constanta din clasa Model). La apăsarea butonului Exit, se închide aplicația. La apăsarea butonului Help, se deschide o fereastră pop-up care conține informații desprea aplicație și modul de folosire:

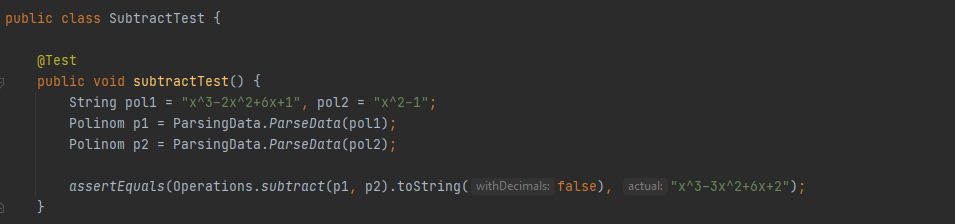


# Testare

Cu JUnit s-a testat fiecare operație. S-a făcut câte o clasă de test pentru fiecare operație, iar cu assertEquals am verificat rezultatul dat de operație cu cel calculat pe foaie.



Aici e un exemplu, clasa de test pentru operația de scădere.



# Concluzii si Dezvoltari Ulterioare

În concluzie, înainte de implementarea propriu zisă a unei aplicații, e nevoie de o analiză în detaliu și de luarea unor decizii cu privire la structură și la modelul ales, astfel încât să fie cel mai potrivit cerințelor. Acest lucru conferă o implementare mai ușoară, mai logică, mai organizată și cu mai puțin efort. Poate fi un avantaj și pentru client/utilizator dacă aplicația e mai intuitivă, organizată și atractivă.

Ca dezvoltare ulterioară, funcția de parsare s-ar putea modifica astfel încât să accepte gradele polinoamelor aleator și să facă operațiile corecte pe ele. O altă dezvoltare ar fi extinderea pe numere reale sau grade negative.

# Bibliografie

1. <http://tynerblain.com/blog/2007/04/09/sample-use-case-example/>
2. <https://regex101.com/>
3. https://cloud.smartdraw.com/
4. ASSIGNMENT\_1\_SUPPORT\_PRESENTATION.pdf