# 

**DOCUMENTATIE TEMA 1**

**CALCULATOR DE POLINOAME**

**Varvara Raluca Ana-Maria**

**Grupa 30222**

**Profesor Laborator: Oana Andreea Marin**

Contents

[1 1](#_Toc66992892)

[1. Obiectivul temei 2](#_Toc66992893)

[2. Analiza 3](#_Toc66992894)

[3. Proiectarea sistemului 4](#_Toc66992895)

[3.1 Proiectarea generală a sistemului 4](#_Toc66992897)

[3.2 Decizii de proiectare 5](#_Toc66992898)

[3.3 Divizarea în subsisteme/pachete 5](#_Toc66992899)

[3.4 Împărțirea în clase 6](#_Toc66992900)

[3.5 Diagrama de clase 8](#_Toc66992901)

[3.6 Structuri de date 9](#_Toc66992902)

[3.7 Algoritmi 9](#_Toc66992903)

[3.8 Interfață 10](#_Toc66992904)

[4. Implementare 12](#_Toc66992905)

[5. Rezultate 18](#_Toc66992906)

[6. Concluzii 19](#_Toc66992907)

[7. Bibliografie 19](#_Toc66992912)

# **Obiectivul temei**

Obiectivul principal al acestei teme este proiectarea și implementarea unui calculator polinomial, un sistem de procesare al polinoamelor, acestea fiind de o singură variabilă și cu coeficienți întregi. Proiectul va prezenta o interfață grafică user-friendly prin care utilizatorul poate introduce două polinoame și prin care va putea selecta operația care se va efectua între ele.

Sub-obiectivele principale ale temei reprezintă pașii care au fost urmați pentru atingerea obiectivului final:

* analizarea problemei și identificarea cerințelor – se identifică cerințele funcționale și nonfuncționale și se prezintă use-case-urile (se găsește în capitolul 2);
* proiectarea calculatorului – se vor alege structurile de date pentru atingerea obiectivului final, alegerea unui pattern arhitectural( capitolul 3)
* implemenatrea calculatorului – se vor descrie clasele de care este nevoie pentru reprezentarea polinoamelor și cele care fac posibilă realizarea interfeței grafice (capitolul 4);
* testarea calculatorului, vor fi descrise câteva scenarii de testare a calculatorului folosindu-ne de Junit Test (capitolul 5).

# **Analiza**

**Cerințele functționale** vor include următoarele:

* introducerea de polinoame de către user și validarea corectitudinii acestor date ( de exemplu “x^2+3x+1” este o introducere corectă pe când “ x la a doua plus unu” nu este o introducere corectă);
* alegerea operației de către utilizator
* adunarea a două polinoame;
* scăderea a două polinoame;
* înmulțirea a două polinoame;
* împărțirea a două polinoame;
* derivarea primului polinom;
* integrarea primului polinom.

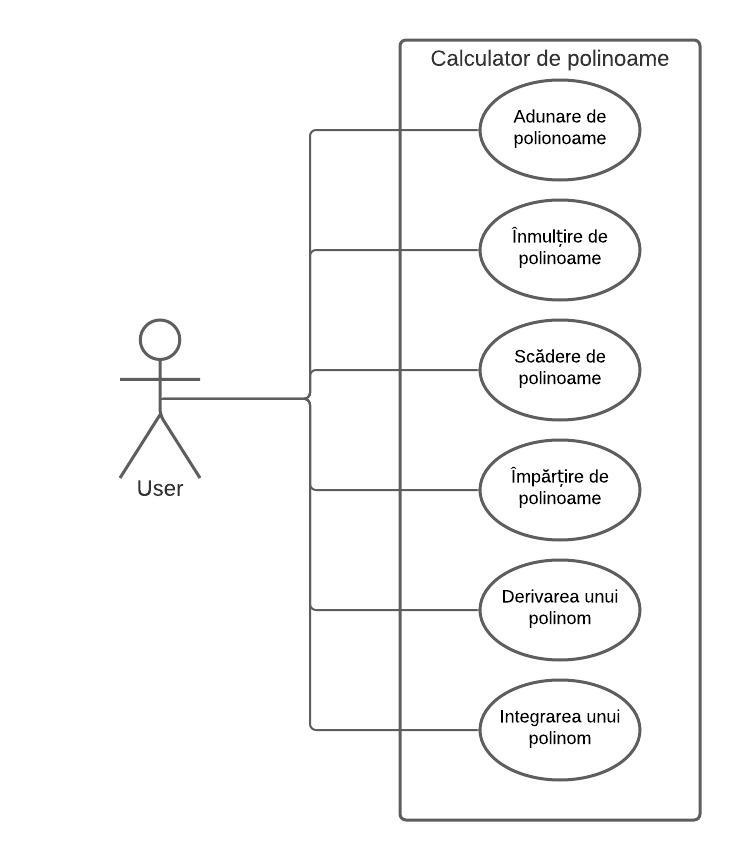


Fig. 1

**Cerintele non-funcționale** vor fi următoarele:

* Calculatorul polinomial ar trebui să fie intuitiv și ușor de folosit de către user;
* Incapsularea datelor;
* Păstrarea datelor pentru a se putea face mai multe operații pe aceleași date;
* Eficiența programului atât din punct de vedere al memoriei (nu am stocat în polinoame monoamele cu coeficient 0 și în clasele Operation nu am stocat date) cât și din punct de vedere al timpului de executie.

**Use case:** adunarea a două polinoame

**Actorul principal:** user-ul

**Scenariul principal de success:**

1. User-ul introduce 2 polinoame în interfața grafică, primul în campul 1, *Primul polinom*, iar al doilea în al doilea camp, *Al doilea polinom*, ambele polinoame având termeni de forma “**coeficientx^grad**”, unde coeficienții 1 pot fi ignorați, atât și puterile 1, respective 0, unde se ignoră și parte de “x^”;
2. User-ul selectează operația de adunare, apăsând pe butonul corespunzător acesteia. În cazul ecuațiilor matematice care se execută pe un singur polinom, primul va fi ales, iar operația se va face pe acesta.
3. Calculatorul efectuează operația de adunare și afișează rezultatul câmpul rezervat acestuia
4. Polinoamele rămân introduce în campurile text astfel se poate alege o altă operație care să se execute pe ele.

**Secvența alternativă:** polinoame incorecte

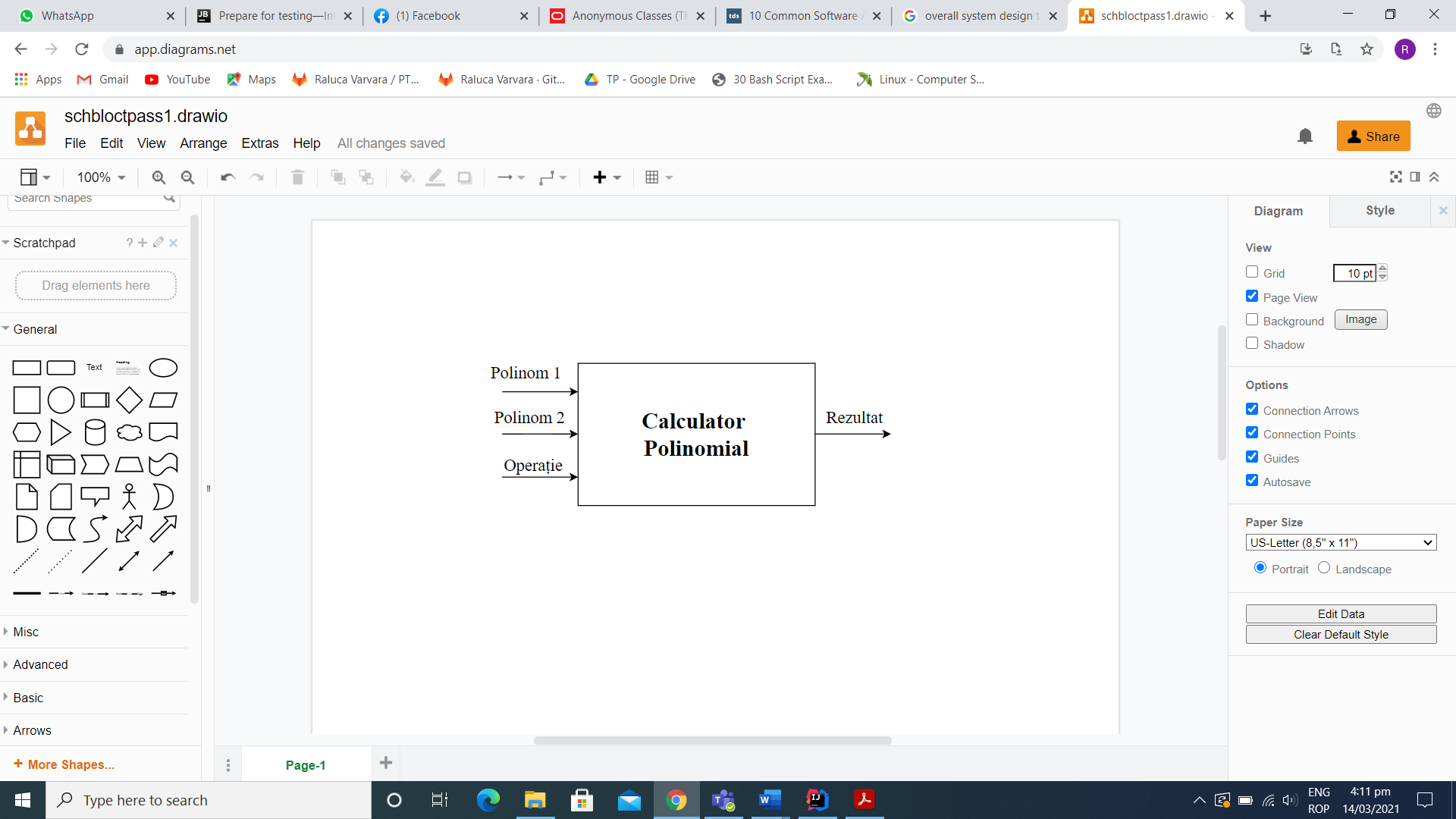
Userul introduce 2 polinoame scrise incorect de exemplu cu mai multe variabile sau cu coeficienți raționali neîntregi, sau introduce un text. Acesta va fi atenționat, scenariul întorcându-se la pasul 1.

Același scenariu se va repeta pentru fiecare dintre use-case-urile prezentate în Fig 1. cu excepția ca pentru operatiile de integrare și derivare va fi necesar numai primul polinom, în cazul în care în al doilea text field se va introduce o dată incorectă, aceasta va fi ignorată. De asemenea o secvență aleatoare este prezentă pentru cazul operațiilor de împarțire și module, dacă se încearcă să se împartă la zero, se va arunca o excepție și va apărea un pop-up care să atenționeze utilizatorul.

# **Proiectarea sistemului**



## Proiectarea generala a sistemului



Sistemul va avea 3 intrari si o iesire, 2 intrari care vor reprezenta datele, cele doua polinoame, si inca o intrare care va specifica ce functionalitate se aplica.

## Decizii de proiectare

Din punct de vedere al proiectarii am luat urmatoarele decizii

* Sa folosesc o arhitectura de tip layered (decizie explicate in capitolul 3.3)
* Pachetul gui din arhitectura precizata anterior sa il implementez in stil MVC(3.3)
* Sa definesc o structura/clasa ajutatoare pentru Polinom, astfel am impartit polinomul in Monoame, si functionalitatile pentru polinom vor fi ajutate de functionalitatile de la monoame
* Pentru ca sa fie in concordanta cu paradigma poo am impartit proiectul astfel incat fiecare clasa sa aiba cat mai putine functionalitati, sa aiba un singur scop
* Am implementat in clasa Monom interfata Comparable pentru a putea sorta un Polinom in functie de grad
* Folosirea un REGEX pentru a imparti si valida un string, daca se poate imparti in monoame si respective transforma in polinom ("[+-]?[0-9]\*(x(\\^[0-9]+)?)?")

## Divizarea in subsisteme/pachete

Am decis să împart problema astfel: un sistem care să rețină datele persistent, un sistem care să efectueze funcționalitătile, operațiile, și un sistem care să se ocupe de interfața grafică cu utilizatorul, să valideze datele și să le afișeze. Astfel am abordat o **arhitectura de tip layered** (fig. 2)**.** Cele 3 pachete pe care le-am implementat sunt **dataModel**, care determină structurile, acest pachet conținând clasele **Monom** și **Polinom**, pachetul **businessLogic** este cel care se ocupă de implementarea funcționalităților și are clasele **OperationMonom** și **Operation Polinom**, iar pachetul **gui** se va ocupa de interfața grafică cu utilizatorul, iar pentru acest pachet am ales să folosesc o altă tehnică pe care am folosit-o până acum pentru proiectele cu interfața grafică, un pattern architectural de tip **MVC** (Fig. 3): acesta având clasele **CalcPolModel, CalcPolView, și CalcPolController**.

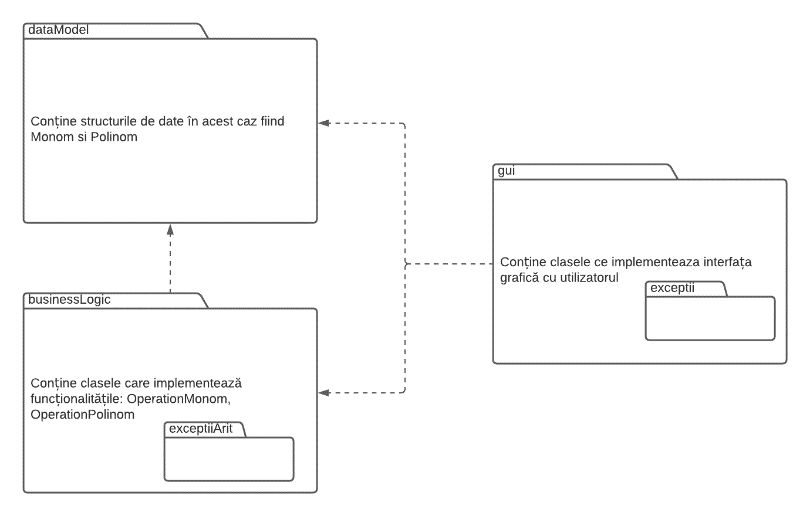
 

Fig. 2 Fig. 3

Am ales un pattern architectural de tip Layered deoarece împarte problema în subprobleme asemănătoare pentru fiecare pachet, după cum am descris mai sus:

* Subproblema de stocare
* Subproblema de funcționaliți
* Subproblema de interacționare cu utilizatorul.

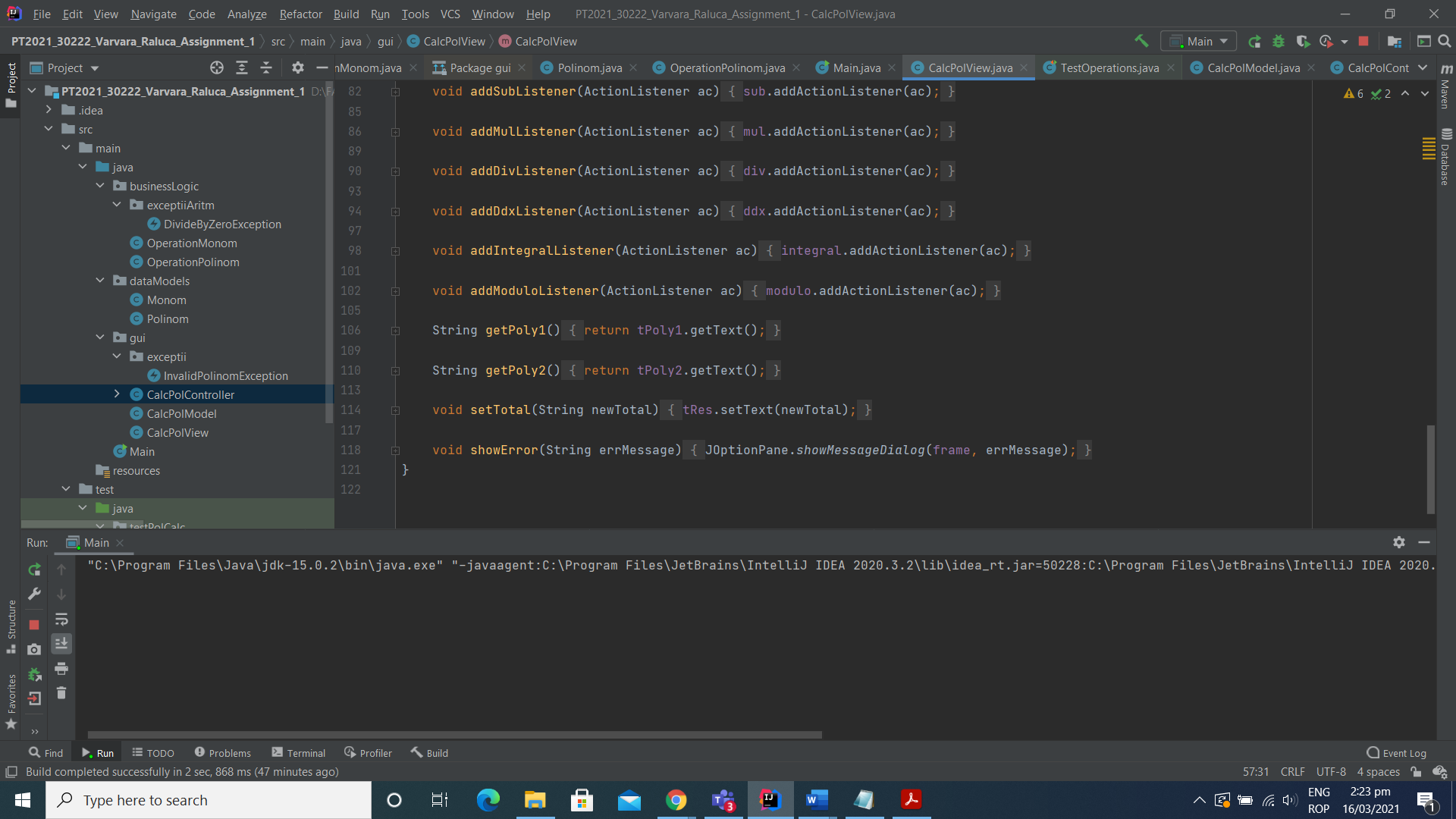
Este un mod ușor și intuitiv de implementare și împărtire în pachete, și nu există confuzii în legătură cu interacționarea dintre pachete.

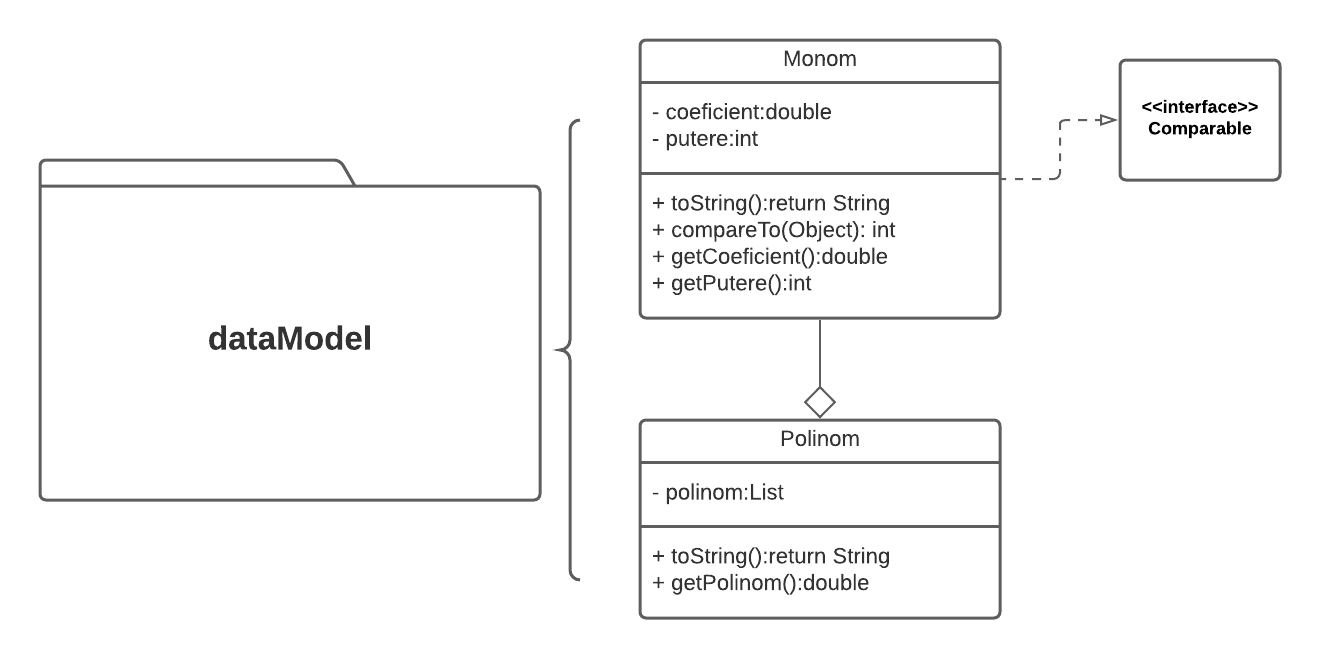
Un dezavantaj este că pachetele și clasele din pachete nu comunică între ele dinamic, cum ar fi într-un model MVC.

Pentru pachetui **gui** am ales un pattern arhitecturat mvc deoarece parte de interfața este mult mai dinamică, și de cele mai multe ori partea de interfață se implementeaza astfel. Clasele vor avea următoarele funcționalități:

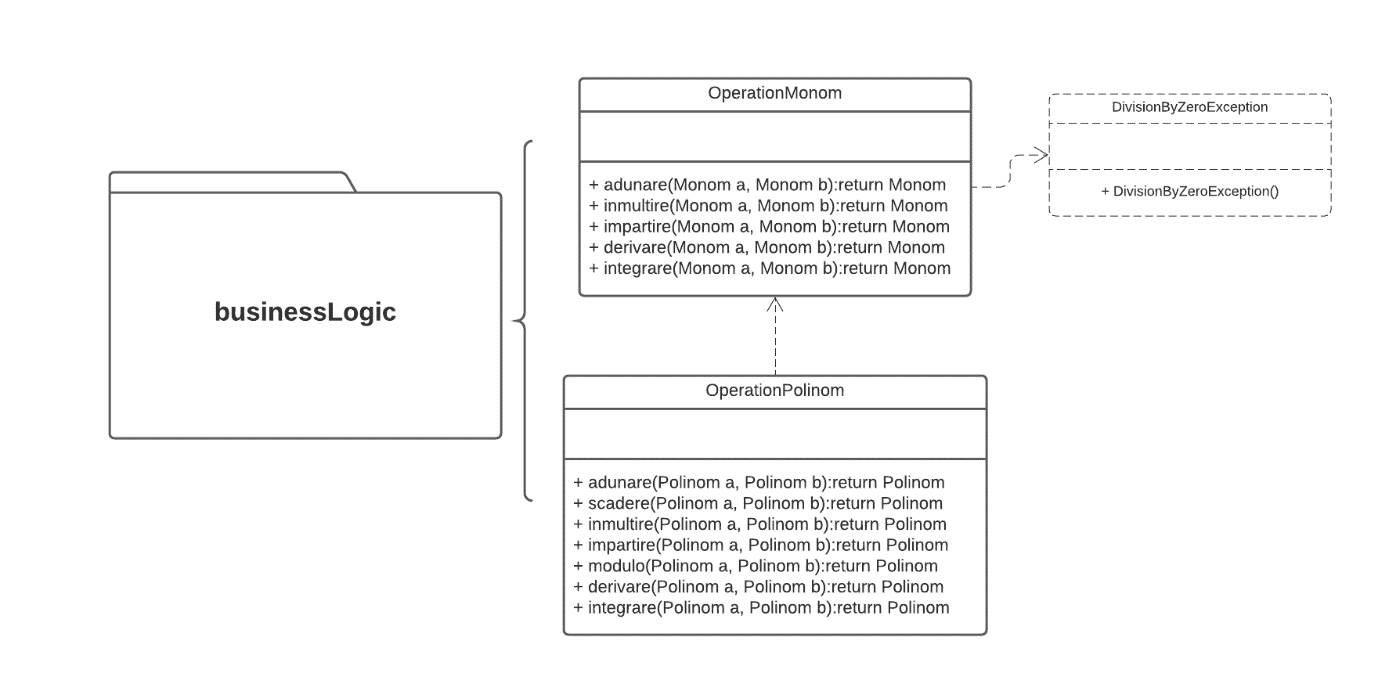
* **CalcPolModel:** este modelul din arhitectura și se va ocupa cu reținerea datelor cu care se va lucra și manipularea datelor, cum utilizatorul poate să introducă numai un String și acesta trebuie transformat în Polinom. Cum nu putem fi siguri de corectitudinea introducerii datelor de utilizator, modelul se va ocupa de validarea și transformarea lor din șir de caractere în polinom, și de setarea și returnarea datelor stocate.
* **CalcPolView:** este view-ul din arhitectura și definește frame-ul, containerele și componentele, va exista frameul principal, 3 text field-uri pentru cei doi operanzi și rezultat și câte un buton pentru fiecare operație.
* **CalcPolController:** este controlerul arhitecturii și va defini câte un ActionListener pentru fiecare element interactiv din interfață (butoane, text field), și ce se va întâmpla când se va interacționa cu fiecare din aceste elemente.

## Împărțirea în clase

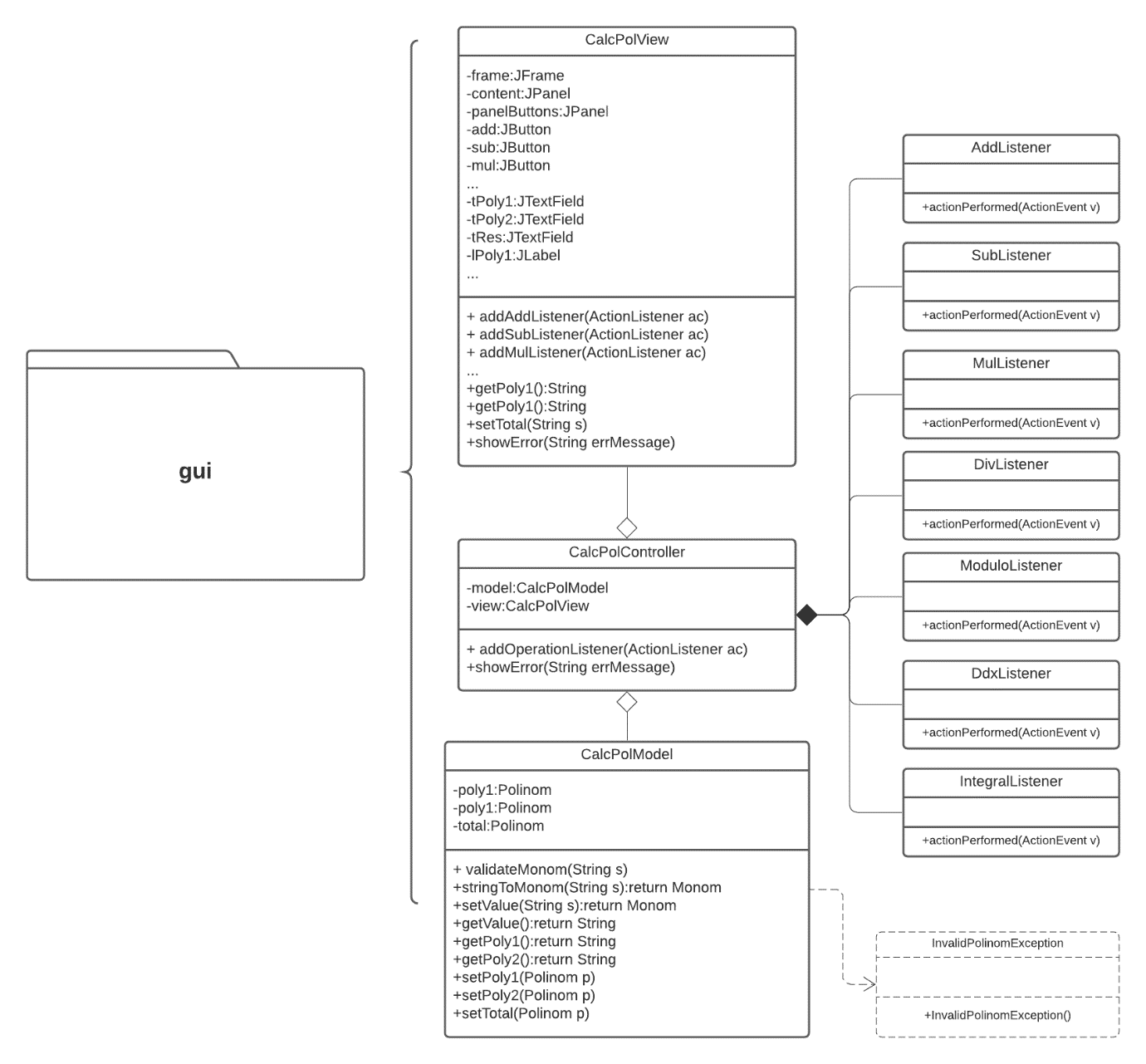




Pachetul dataModel va reține clasele care definesc modelele de date și care le vor putea stoca persistent astfel am împărțire în subprobleme și am implementat o clasă Monom și altă clasă Polinom care va avea o listă de moname. **Relația** dintre aceste două clase este de agregare, un Polinom depinde de Monom, are mai multe Monoame, dar este numai o relație de agregare, nu de compoziție deoarece Monom își păstreaza durata de viață și după distrugerea Polinomului.



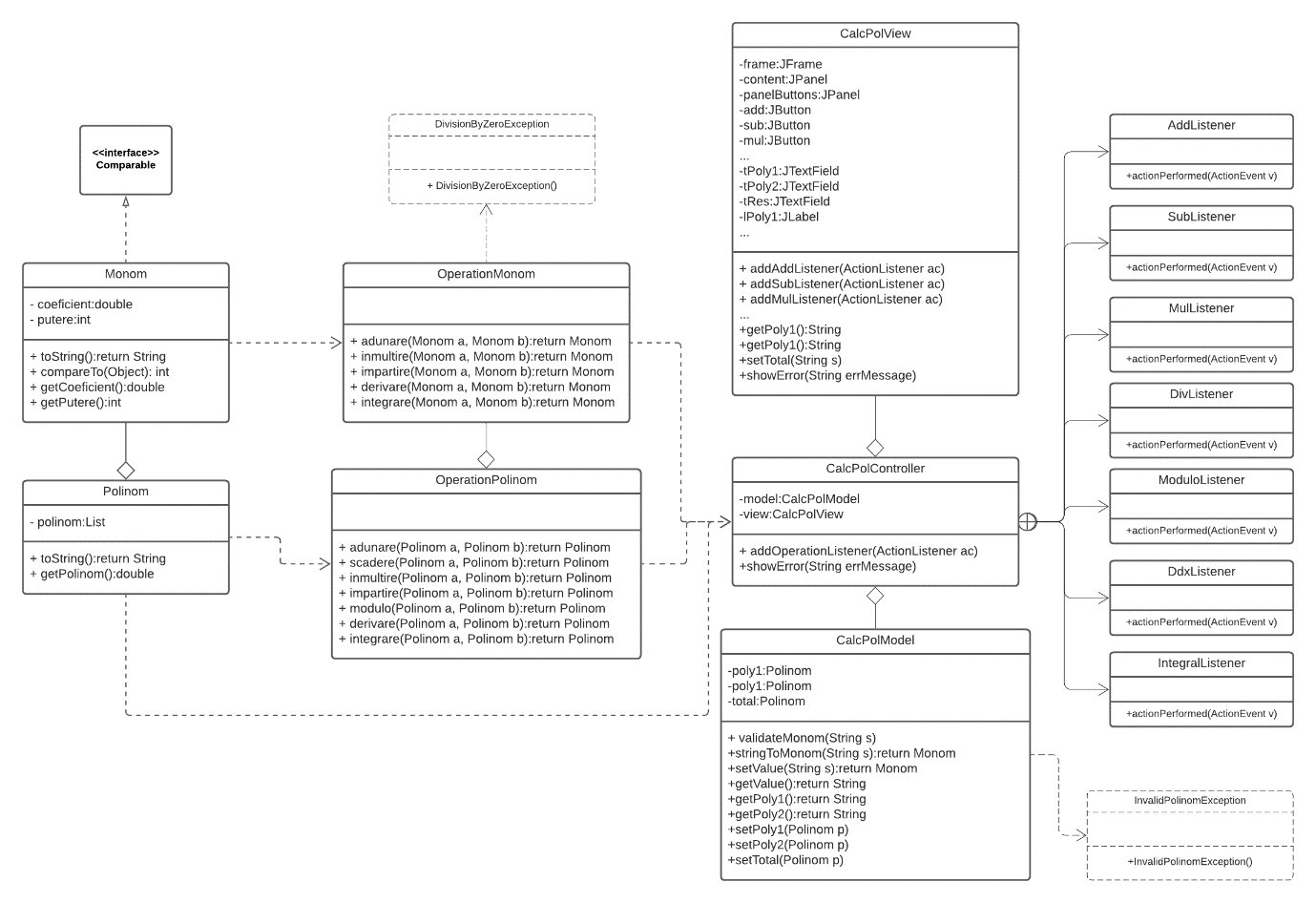
Pachetul businessLogic va defini clasele care se vor ocupa cu funcționalitățile, adică operațiile. Pentru a sparge problemele în probleme mai mici am definit o clasă OperationMonom care se ocupă cu operații pe Monoame și clasa OperationPolinom care se ocupă cu operațiile pe Polinoame. Între aceste 2 clase este o relație de dependență, OperationPolinom depinde de OperationMonom, deoarece prima o folosește pe a doua și nu ar putea funcționa corect fără aceasta, fiecare metodă ar fi eronată. De asemenea ambele clase pot arunca excepția definite DivisionByZeroException.



Pachetul gui am decis sa il implementez cu ajutorul unei arhitecturi MVC, tipica interfetelor cu utilizatorul, avand cate o clasa pentru Model, View si Controller, controllerul avand cate o clasa interna separata pentru fiecare operatie in parte. Controllerul are un View si un Model de care se foloseste in fiecare metoda actionPerformed din clasele interne asa ca este o **relatie de agregare.** Intre controller si clasele interne este o **relatie de compozitie**, intrucat acestea nu exista in afara controlerului.

## Diagrama de clase

Unified Modeling Language (prescurtat UML) este un limbaj standard pentru descrierea de modele și specificații software. Diagrama de clase UML este folosită pentru reprezentarea vizuală a claselor și a interdependențelor, taxonomiei și a relațiilor de multiplicitate dintre ele. Diagramele de clasă sunt folosite și pentru reprezentarea concretă a unor instanțe de clasă, așadar obiecte și a legăturilor concrete dintre acestea.



## Structuri de date

Singura structura de date predefinita pe care am folosit-o este ArrayList, lista de monoame din Polinom pe care am declarat-o ca List si am instantiat-o cu ArrayList. Clasa ArrayList este o matrice redimensionabilă, care poate fi găsită în pachetul java.util.

De asemenea structura de date se poate considera si Monom, clasa declarata de mine, deoarece este folosita pentru a organiza datele si a le prelucra mai eficient.

## Algoritmi

Algoritmii sunt folositi sunt cei pentru implementarea functionalitatilor.

**Algoritmul de adunare**

Se ia un polinom auxiliar la care adaugam toate elementele ambelor polinoame, pe care la sortam cu ajutorul metodei **Collection.sort(),** care este o sortare stabila, deci termenii cu acelasi grad din polinomul a, primul polinom, vor ramane inaintea termenilor din polinomul b, acest feature ne va ajuta pentru metoda de scadere. Apoi parcurgem polinomul auxiliar, monoamele din lista, cu o bucla for each si retinem un monom anterior celui curent, daca au acelasi grad acestea se vor aduna.

**Algoritmul de scadere**

Foloseste algoritmul de adunare, si aduna primul polinom, cu al doilea inmultit cu 0.

**Algoritmul de inmultire**

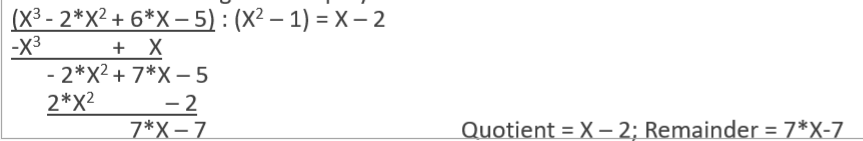
Am considerat 2 polinoame, rezultatul si unul auxiliar. Vom inmulti monoamele din primul polinom pe rand cu al doilea polinom in polinomul auxiliar, si dup ace s-a terminat inmultirea cu un monom, auxiliarul se va aduna la rezultat si va fi reinitializat.

**Algoritmul de impartire**

**Pas 1** – impartim primul monom din deimpartit la primul monom din impartitor si obtinem primul termen din cat

**Pas 2** **-** inmultimul impartitorul cu termenul rezultat din pasul anterior, si scadem din deimpartit obtinand restul

**Pas 3 –** repetam pasii anterior considerand deimpartit ca fiind restul obtinut la pasul anterior pana cand gradul restului este mai mic decat gradul impartitorului.



**Algoritmul de derivare**

Parcurgem polinomul si derivam fiecare termen adaugandu-l la rezultat.

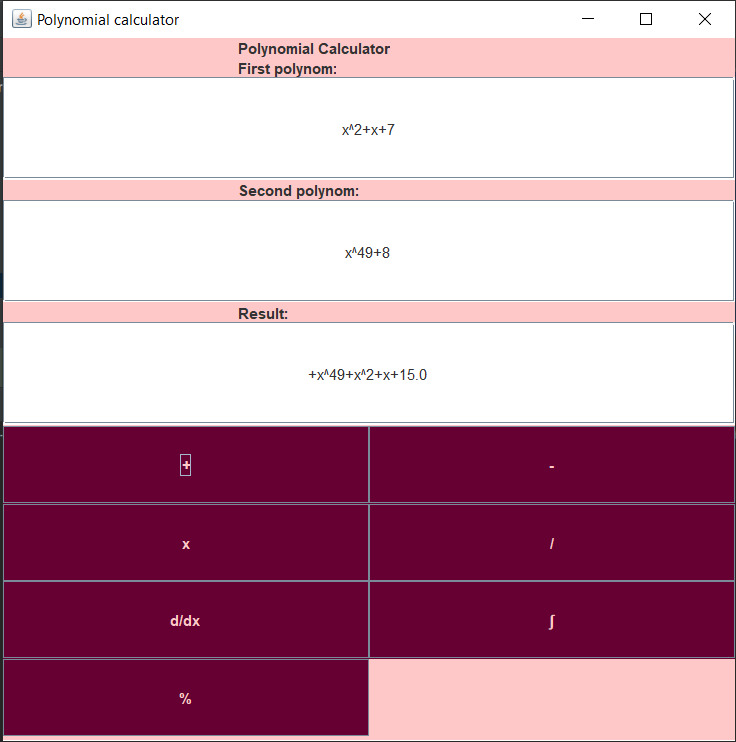
**Algoritmul de integrare**

Parcurgem polinomul si integram fiecare termen adaugandu-l la rezultat.

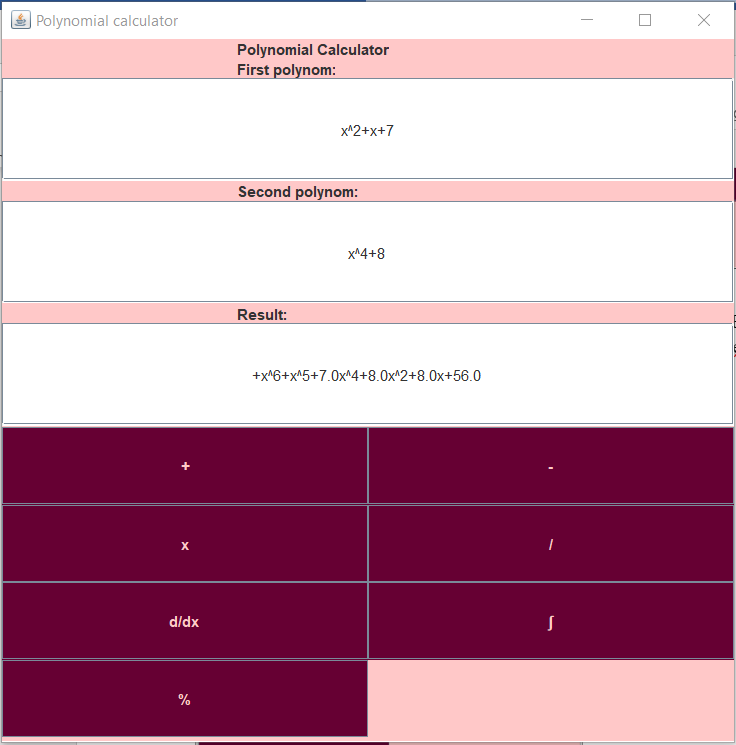
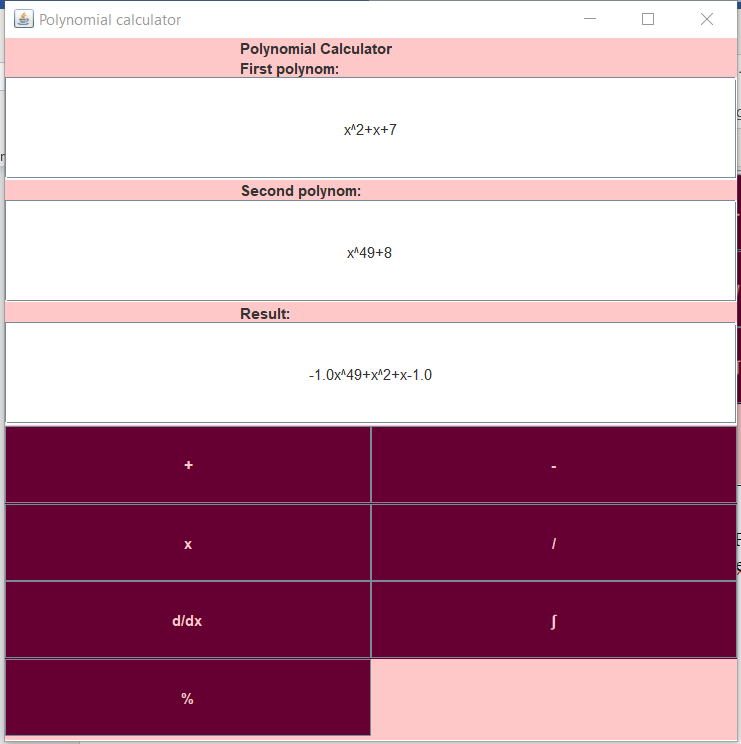
## Interfata

Am ales sa implementez o interfata simpla si intuitiva, usor de folosit de catre user. Are 2 campuri in care se pot introduce polinoame, si inca un camp in care se va afisa rezultatul in urma operatiei. Desupra fiecarui camp este un label care il descrie. Exista butoane pentru fiecare operatie si nu este necesar sa se introduca de mai multe ori datele daca se vrea sa se execute mai multe operatii pe aceleasi date.

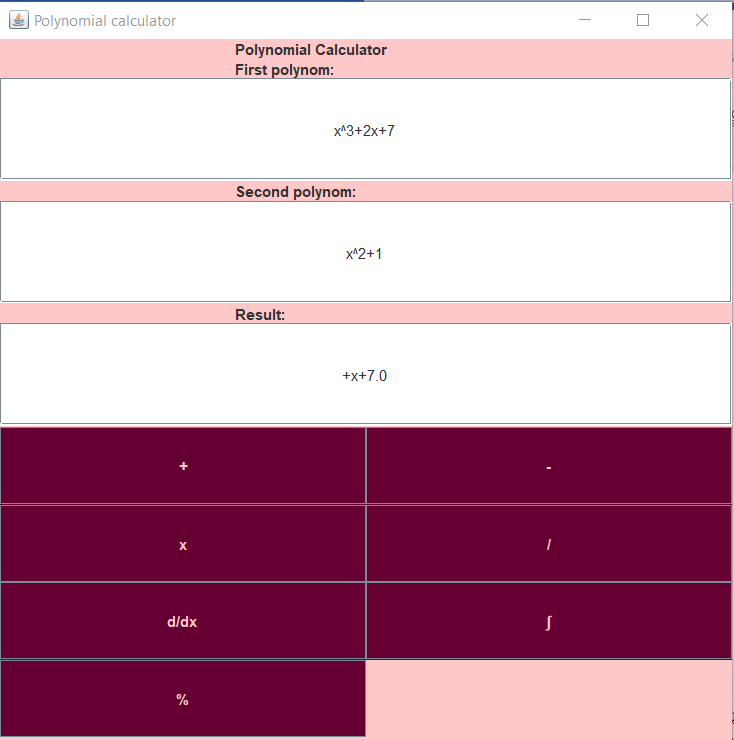
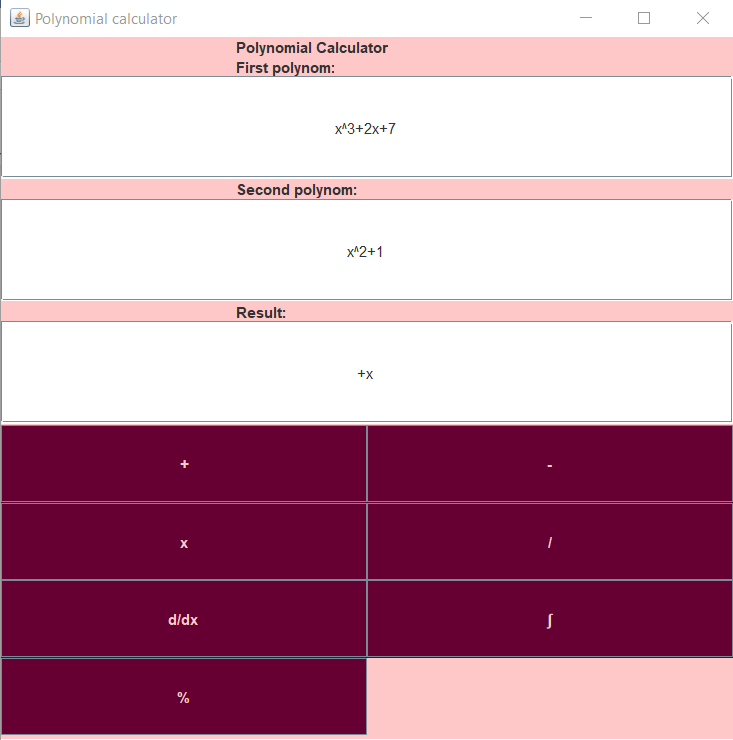
Fiecare buton are un semn cu operatia pe care o excuta (+, -, x, /, %, d/dx, )



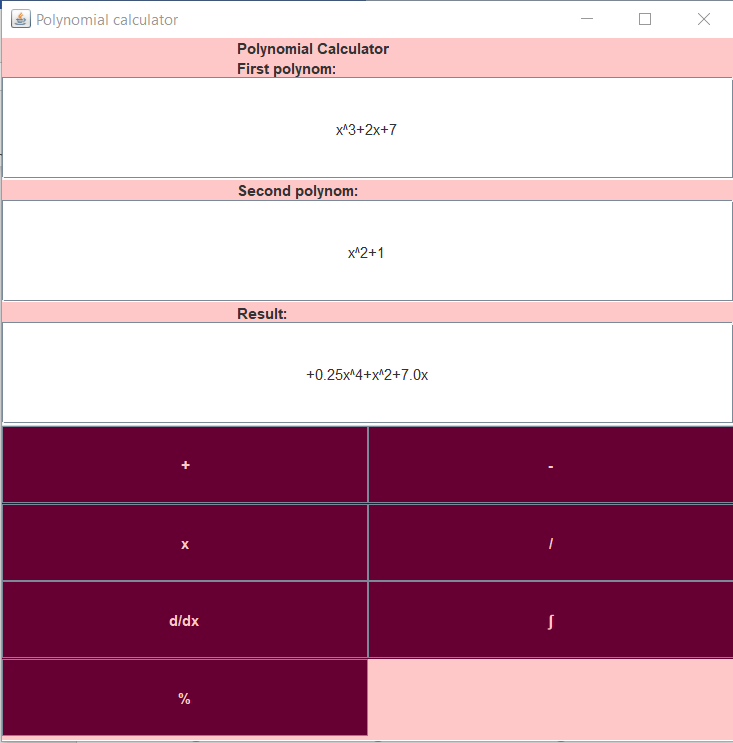
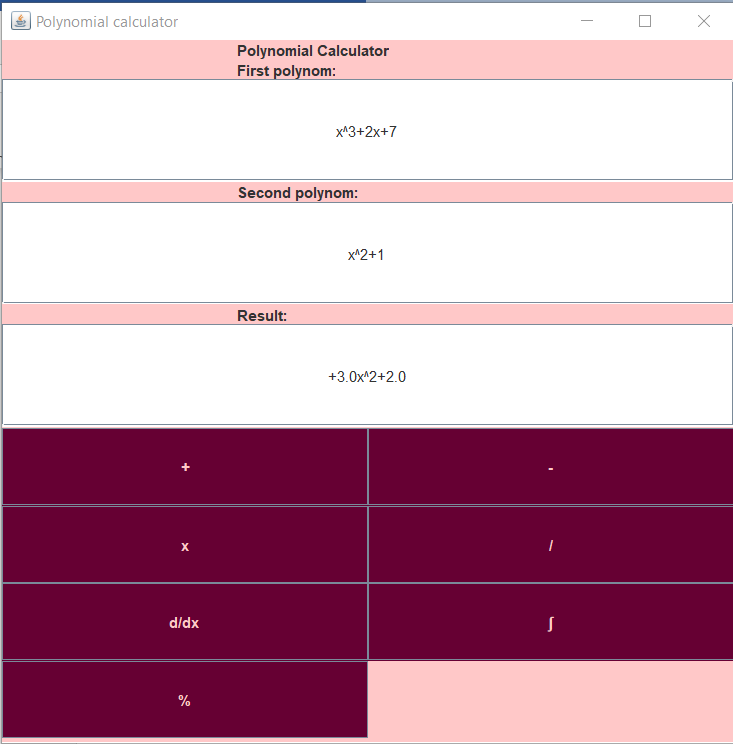
Am ales sa implementez interfata cu Java Swing. Swing este un set de instrumente widget GUI pentru Java. Face parte din Oracle Java Foundation Classes (JFC) - un API pentru furnizarea unei interfețe grafice de utilizator (GUI) pentru programele Java.



Scadere Inmultire



Impartire Modulo



Derivare Integrare

# Implementare

**Pachetul dataStructures**

**Clasa Monom**

Clasa monom este folosită pentru a descrie structura de date pentru monom. Monomul este elemuntal de bază din care se formează un polinom, acesta are mai multe monoame, iar pentru o divizare eficientă în subprobleme, am decis să implementez clasa aceasta.

**Atribute:**

* **double coeficient**: care este de tipul double, deoarece double poate reprezenta și întregi, polinoamele introduse având toate coeficienți întregi, iar tipul double ne trebuie pentru integrare, unde coeficienții de cele mai multe ori rezultă în a fi raționali, neîntregi;
* **int putere**: este de tipul int deoarece lucrăm numai cu puteri naturale >=0.

Ambele atribute sunt **private**, folosind conceptul specific progrămarii orientate pe obiect de incapsulare a datelor.

**Metode:**

* **public double getCoeficient() ­–**  getter pentru coeficient;
* **public int getPutere()** – getter pentru putere;
* **@Override public String toString() –** suprascrierea metodei toString pentru o afișare mai ușoară și mai eficientă a întregului Polinom, tehnica specifică poo de împărțire a problemei în subprobleme;
* **@Override public int compareTo(Object o) ­–** este metoda definită țn interfața Comparable implementată de această clasă, metoda care ne va ajuta pentru operațiile pe polinoame. Compararea se face după putere, adică după grad, pentru a menține în ordine monoamele în polinom.

După cum se observă nu există nici un setter, deoarece nu vrem să modificăm datele în acest program, vrem să putem refolosi pentru mai multe operații dacă este nevoie.

**Clasa Polinom**

Este clasa care definește structura polinomului întreg, care este o listă ordonată de monoame (elementul de baza al polinomului). Între clasele Polinom și Monom este o relație de compoziție, un Polinom are unul sau mai multe Monoame.

**Atribute:**

* **private List<Monom> polinom = new ArrayList<Monom>()** un polinom este definit ca o listă de monoame. Este un atribut privat pentru a respecta paradigma poo. Este definit cu interfața List și instanțiat cu clasa ArrayList, care implementează aceasta interfață.

**Metode:**

* **public List<Monom> getPolinom()**  - este necesar pentru a accesa variabila instanta privată;
* **@Override public String toString()** – este suprascrierea metodei toString din clasa mama Object, și se parcurge cu o buclă foreach fiecare monom din lista de polinoame, și în funcție de semnul coeficientului se pune plus sau minus înaintea termenului.

**Pachetul businessLogic**

**Clasa OpertionMonom**

Această clasă are responsabilitatea de a defini funcționalitățile pentru clasa Monom. Am decis să creez această clasă pentru a nu oferi prea multe funcționalități clasei Monom

**Atribute:**

* clasa **NU are atribute,** deoarece este o clasă care se ocupă exclusiv de funcționalități.

**Metode:**

Metodele de la clasa monom sunt considerate mai mult ca niște metode ajutătoare pentru operațiile care se vor executa pe polinoame, astfel lucrăm modular, și se vor putea testa erorile mai usor. Am folosit o tehnică de implementare pentru clasele de Operatii bottom up, detectând problemele mici și implementându-le, implementând soluțiile pentru problemele mai mari mai apoi.

* **public Monom adunare(Monom a, Monom b)** este strict o metodă ajutătoare considerându-se că dacă se adună două monoame cu grade diferite va rezulta un polinom, astfel această metodă adună numai 2 monoame cu aceeași putere. **Metoda returnează un nou monom care reprezintă suma parametrilor.** Dacă se încearcă să se adune 2 monoame de grade diferite, rezultatul acestei adunări va fi unul eronat, dar este o metodă care se poate apela numai în cadrul clasei OperationPolinom, și am avut grijă să nu se apeleze greșit, am urmărit mai multe exemple și cu debuggerul pentru fiecare operație care folosește această metodă.
* **public Monom înmulțire(Monom a, Monom b)** poate înmulți orice 2 monoame deoarece rezultatul o să fie tot un monom, orice 2 monoame ar fi introduse ar produce rezultatul corect. Aceasta este, la fel, o metodă ajutătoare pentru metodele din clasa de operații pentru polinoame. Se vor înmulți coeficienții și se vor aduna puterile. **Metoda returnează un nou monom care reprezintă înmulțirea parametrilor.**
* **public Monom împărțire(Monom a, Monom b) throws DivideByZeroException** este similară cu cea de înmulțire, se împart coeficienții și se vor scădea puterile. Dar dacă se vrea să se împartă cu un monom care are coeficientul 0 se va arunca o excepție definite de mine, și se va alarma utilizatorul.
* **public Monom integrare(Monom a)** returnează monomul integrat, se folosește formula simpla de integrare a unui monom . Cum nu se poate împărți la 0 niciodată deoarece puterea minimă e 0 => n+1 = 0+1 = 1, nu se aruncă nici o excepție.
* **public Monom derivare(Monom a)** returnează monomul derivate și se folosește formula simplă de derivare a monomului .

**Clasa OperationPolinom**

Este o clasă specială pentru operațiile pe polinoame, nu are nici o variabilă instantă ci numai polinoame. Funcționalitatea acestei clase este de a efectua operațiile pe polinoame și a returna rezultatul pentru fiecare dintre aceste operații.

**Atribute:**  la fel ca la clasa anterioară, cum aceasta se ocupa tot numai cu funcționalități, operații, nu are nevoie de variabile instanta.

**Metode:**

* **public Polinom adunare(Polinom a, Polinom b):** se ia un polinom auxiliar la care adăugăm toate elementele ambelor polinoame, pe care le sortăm cu ajutorul metodei **Collection.sort(),** care este o sortare stabilă, deci termenii cu același grad din polinomul a, primul polinom, vor rămâne înaintea termenilor din polinomul b, acest feature ne va ajuta pentru metoda de scădere. Apoi parcurgem polinomul auxiliar, monoamele din lista, cu o buclă for each și reținem un monom anterior celui curent, dacă au același grad acestea se vor aduna. Dacă coeficientul rezultatului este 0 îl ignorăm și nu îl mai adaugăm la rezultat.
* **public Polinom scădere(Polinom a, Polinom b):** pentru operația de scădere m-am bazat pe formula  **a – b = a + (-b),** astfel am înmultit cu monomul -1 (coeficient = -1, putere = 0), fiecare termen al polinomului b pentru a-i schimba semnul, iar apoi am apelat metoda de adunare (this.adunare(…)).
* **public Polinom înmulțire(Polinom a, Polinom b):**  am considerat 2 polinoame, rezultatul și unul auxiliar. Vom înmulți monoamele din primul polinom pe rând cu al doilea polinom în polinomul auxiliar, și după ce s-a terminat înmulțirea cu un monom, auxiliarul se va aduna la rezultat și va fi reinițializat.
* **public Polinom împărțire(Polinom a, Polinom b) throws DivideByZeroException:** se folosește algoritmul de împărțire descris în capitolul 3. Proiectare, la subcapitolul de algoritmi, și se rețin 2 polinoame auxiliare, câtul și restul și se returnează câtul. Se poate arunca Exceptia DivideByZero din metoda de împărțire din clasa OperationMonom.
* **public Polinom modulo(Polinom a, Polinom b) throws DivideByZeroException:** se apelează metoda de împărțire și se returnează diferența dintre deîmpărțit și înmulțirea dintre cât și împărțitor pe baza formulei : **Deîmpărțit = Împărțitor\*Cât+Rest.**
* **public Polinom derivare(Polinom a):**  se iterează peste lista de monoame a polinomului a și se crează un rezultat din fiecare monom derivat.
* **public Polinom integrare(Polinom a):** se iterează peste lista de monoame a polinomului a și se crează un rezultat din fiecare monom integrat.

**Clasa DivideByZeroException**

Este o clasa care **extinde clasa Exception** și are numai un constructor care apelează constructorul super cu mesajul potrivit, atenționează dacă se împarte cu 0.

**Pachetul gui**

A fost implementat cu o arhitectura MVC.

**Clasa CalcPolModel**

Este clasa care se ocupa cu prelucrarea datelor si stocarea lor din partea de interfata, tipi cunei clase model dintr-o arhitectura MVC. Retine datele cu care va lucra calculatorul si prelucreaza sirurile de caractere introduce de utilizator astfel inca sa rezulte polinoame, si daca sunt date introduce gresit va arunca o exceptie.

**Atribute:**

* private Polinom poly1 – primul operand
* private Polinom poly2 – al doilea operand
* private Polinom total – rezultatul operatiei intre operanzii de mai sus.

**Metode:**

* **public void validateMonom(String s) throws InvalidPolinomException** – este metoda care decide daca un string poate fi monom sau nu. Am create un regex care sa defineasca paternal unui monom "[+-]?[0-9]\*(x(\\^[0-9]+)?)?", acesta insemnand ca este de forma coeficient x ^ putere, unde coeficient si putere sunt int-uri si se poate avea semn in fata dar nu e necesar pentru numerele positive, iar coeficientul si puterea 1 pot lipsi, iar daca puterea este 0, partea de x^ va fi si ea ignorata. Cu ajutorul claselor Pattern si Mattcehr, daca stringul nu este de aceasta forma se va arunca o exceptie
* **public Monom stringToMonom(String s)** – aceasta este o metoda care transforma un string intr-o structura de tip Monom. Se parcurge sirul de caractere si prima oara se verifica daca este un semn la inceput, in acest caz se trece la urmatorul caracter si se formeaza un numar pana cand se da de litera x. Daca nu exista nici un numar se considera ca coeficientul este 1 iar apoi daca primul caracter era minus se schimba semnul coeficientului. Apoi se ia sirul de la final spre inceput pana se da de caracterul ^ sau se ajunge la indicele din parcurgerea anterioara si se determina numarul de la putere. Daca nu exista, si exista litera x in sir se considera ca putere era 1, daca nu exista nici caracterul ^ nici x puterea se considera 0.
* **public Polinom setValue(String s) throws InvalidPolinomException** – este metoda care se foloseste pentru a seta valorile variabilelor instanta. Se imparte sirul de caractere in subsiruri pe rand pana se ajunge la primul semn, daca este plus il va ignora pentru urmatorul subsir daca este minus nu. Acest subsir este validat daca se potriveste cu regexul pentru monom, daca este valabil, in caz contrar se rearunca exceptia metodei de validare monom. Daca este okay, acesta este transformat in monom si adugat la polinom. Acest pas se repeat pana cand se ajunge la finalul sirului. Subsirul se determina cu 2 indici, unul de inceput altul de final.
* **public String getValue() –** aceasta metoda returneaza valoarea totalului, rezultatului. Se trateaza exceptia ArrayIndexOutOfBounds care se poate arunca de metoda to string cand rezultatul da zero la o operatie, deoarece in fiecare operatie ignor monoamele rezultante cu coeficientul 0 iar daca rezultatul e 0 va fi polinomul null.
* **public Polinom getPoly1() –** urmatoarele metode sunt gettere si settere pentru operanzi
* **public Polinom getPoly2()**
* **public void setPoly1(Polinom poly1)**
* **public void setPoly2(Polinom poly2)**

**Clasa CalcPolView**

Aceasta clasa este view-ul MVC-ului, defineste strict cum va arata interfata si are metode de adaugare de Listeneri pentru fiecare componenta interactive.

**Atribute:**

* private final JFrame frame = new JFrame("Polynomial calculator");
* private final JPanel content = new JPanel();
* private final JPanel panelButtons = new JPanel();
* private final JLabel title = new JLabel("Polynomial Calculator" , SwingConstants.CENTER);
* private final JLabel lPoly1 = new JLabel("First polynom:");
* private final JLabel lPoly2 = new JLabel("Second polynom:");
* private final JLabel lRes = new JLabel("Result:");
* private final JTextField tPoly1 = new JTextField();
* private final JTextField tPoly2 = new JTextField();
* private final JTextField tRes = new JTextField();
* private final JButton add = new JButton("+");
* private final JButton sub = new JButton("-");
* private final JButton mul = new JButton("x");
* private final JButton div = new JButton("/");
* private final JButton ddx = new JButton("d/dx");
* private final JButton integral = new JButton("∫");
* private final JButton modulo = new JButton("%");

**Metode:**

* void addAddListener(ActionListener ac) – la urmatoarele metode se adauga numai ActionListener pentru fiecare buton
* void addSubListener(ActionListener ac)
* void addMulListener(ActionListener ac)
* void addDivListener(ActionListener ac)
* void addDdxListener(ActionListener ac)
* void addIntegralListener(ActionListener ac)
* void addModuloListener(ActionListener ac)
* String getPoly1() – acesta metoda si urmatoarea returneaza stringul introdus de utilizator in capurile pentru operanzi
* String getPoly2()
* void setTotal(String newTotal) – aceasta metoda seteaza text fieldul de la rezultat
* void showError(String errMessage) – acesta metoda afiseaza un pop up in cazul in care este o eroare la introducerea datelor sau la impartirea cu 0.

**Clasa CalcPolControler**

Este clasa care se ocupa cu interactiunea dintre model si view si implementeaza functionalitatile pachetului de gui, adauga fiecare ascultator pt fiecare buton.

**Atribute:**

* private CalcPolModel model;
* private CalcPolView view;

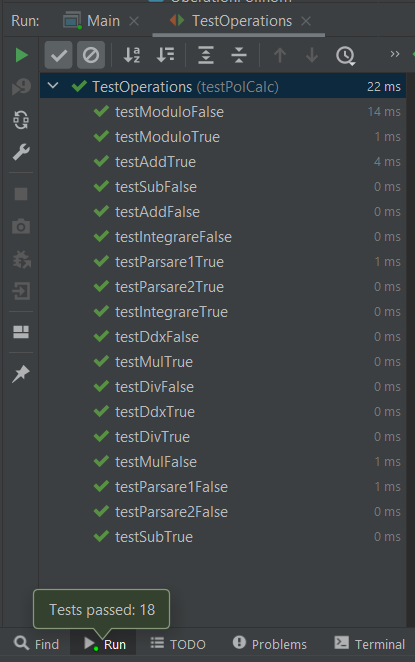
**Constructor: -** el va initializa modelul si view ul si va apela fiecare metoda de adaugare de ascultator definite pentru view cu clasele interioare corespunzatoare fiecarei operatii.

**Clase interne:**

Fiecare clasa interna definește metoda public void actionPerformed(ActionEvent e) :se încearcă să se seteze valorile operzanzilor din model cu valorile stringurilor din cele 2 text fielduri corespunzătoare. Daca se arunca o excepție se tratează și în clauza catch se vă apela metoda din view de a afișă o eroare cu un mesaj corespunzător.

* **class AddListener implements ActionListener** – după pașii descriși mai sus, în caz de success al introducerii datelor se apelează metoda de adunare, se setează variabila instanță din model de rezultat și apoi se vă afișă și în câmpul de rezultat din view.
* **class SubListener implements ActionListener** – după pașii descriși mai sus, în caz de success al introducerii datelor se apelează metoda de scădere, se setează variabila instanță din model de rezultat și apoi se vă afișă și în câmpul de rezultat din view.
* **class MulListener implements ActionListener** – după pașii descriși mai sus, în caz de success al introducerii datelor se apelează metoda de înmulțire, se setează variabila instanță din model de rezultat și apoi se vă afișă și în câmpul de rezultat din view.
* **class DivListener implements** **ActionListener** – la fel că mai sus dar se mai poate arunca și excepția împărțirii cu 0 și atunci se mai adaugă o clauza catch care să trateze, șiș a atenționeze utilizatorul de eroare
* **class ModuloListener implements ActionListener –** la fel că la împărțire
* **class DdxListener implements ActionListener** – la fel că la primele metode de adunare, scădere, dar se initializeaza numai primul operand și se apelează metoda de derivare
* **class IntegralListener implements ActionListener** – la fel că la derivare dar se apelează metoda de integrare

# Rezultate

Am făcut câte un test negativ și câte un test pozitiv pentru fiecare operație în parte și câte 2 teste pozitive și 2 negative pentru parsare, toate testele la rulare fiind corecte.

Am implementat și o metoda precedată de adnotația @Before pentru a se execută înainte de fiecare test, pentru a mă asigura că nu se vor schimba parametrii.

Scenariile pentru fiecare operație sunt ori cazul în care rezultatul este bun ori cazul în care rezultatul nu este corect. La parsare sunt mai multe cazuri, cazurile negative fiind următoarele: stringul nu este valid, nu poate fi transformat în polinom metoda testParsare2False prezentând acest caz, 0 este înlocuit de O, iar celalt caz negative fiind faptul că stringul este valid dar transformarea lui în polinom nu este corectă. Cazurile pozitive sunt când stringul este atât valid cât este și transformat corect în polinom.

# Concluzii

În concluzie realizarea acestui proiect a fost foarte utilă , amintind de o mare parte din principalele concepte ale porgramării orientate pe obiecte învățate semestrul trecut.

Am învățat să lucrez cu regex-uri, și un nou tip de arhitectura, arhitectura layered. M-am obișnuit mai bine cu alte paradigme ale programării orientate pe obiect cum ar fi atribuirea unei clase o singura responsabilitate și împărțirea pe pachete. Am învățat mai bine să fac diagrame UML și Ușe case-uri.

O dezvoltare ulterioara ar fi atenționarea mult mai clară a utilizatorului în cazul în care datele sunt introduse greșit, adică programul să îi spună utilizatorului exact care dintre polinoame a fost introdus incorect și să vină cu o sugestie de corectare. Și de asemenea să se permită sim ai multe formate de a introduce polinoamele, de exemplu spațierea să fie ignorata.

O alta dezvoltare ulterioară ar fi afișarea mai frumoasă a rezultatelor, să se afișeze numerele în format double dacă și numai dacă coeficienții sunt double, după integrare de exemplu, afișarea constantei la integrare și altele.

# Bibliografie

* <https://lucid.app/documents>
* <https://regexr.com/>
* <https://www.tomdalling.com/blog/software-design/model-view-controller-explained/>
* <https://www.informit.com/articles/article.aspx?p=336264&seqNum=3>
* ASSIGNMENT\_1\_SUPPORT\_PRESENTATION