

Dictionar

O aplicație care permite căutarea în ierarhii de clase serializate JSON



May 25, 2016

Ștefănescu marian, seria A, grupa 30223

U.T.C. Calculatoare & Tehnologia Informatiei

Cuprins

[1. Introducere 2](#_Toc451955371)

[2. Analiza problemei și modelarea acesteia 3](#_Toc451955372)

[2.1. Cazuri de utilizare 4](#_Toc451955373)

[2.2. Scenarii 4](#_Toc451955374)

[3. Proiectare 5](#_Toc451955375)

[3.1. Clasa Panel 6](#_Toc451955376)

[3.2. Clasa Cuvant 7](#_Toc451955377)

[3.3. Clasa Clasa 8](#_Toc451955378)

[3.3.1. Crearea claselor de echivalență 8](#_Toc451955379)

[1.3.2. Găsirea unui cuvânt în structura de clase 9](#_Toc451955380)

[1.3.3. Adăugarea consistentă a cuvintelor la dicționar 9](#_Toc451955381)

[1.3.4. Implementare compozită a dicționarului 9](#_Toc451955382)

[1.4. Clasa JSON 10](#_Toc451955383)

[1.5. Clasa Factory 11](#_Toc451955384)

[1.6. Rezultate 12](#_Toc451955385)

[2. Concluzii 12](#_Toc451955386)

[3. Bibliografie 13](#_Toc451955387)

# Introducere

Dicționarele sunt un exemplu clasic de structuri de date masive care implică primele dificultăți atât în definirea lor corectă, cât și în implementarea generală. Un dicționar, la modul general, este o colecție de date grupate după un număr de proprietăți asemănătoare, proprietatea putând fi orice element din colecție pe care îl definim ca reprezentat. În cazul dicționarelor clasice, acest reprezentant este chiar prima literă a unui cuvânt, iar mai apoi, în variante mai extinse întregi structuri arborescente numite Trie unde un cuvânt se definește ca totalitatea muchiilor de la rădăcină pănă la o anumită frunză.

Aplicația mea constă în simularea unui dicționar simplist folosind funcțiile Lambda, definite în Java 8 care permit manipularea foarte ușoară a colecțiilor folosind Stream-uri, filtre și Predicate. De asemenea, aplicația oferă capabilități de adăugare a cuvintelor și vizualizare a claselor de echivalentă.

# Analiza problemei și modelarea acesteia

Prin analiza problemei, ne referim la un prim set abstract de operații și proprietăți prin care încercăm să depistăm eventualele însușiri și comportamente ale proceselor necunoscute. Programarea orientată ne oferă aici un avantaj clar, tocmai fiindcă ea permite să taclăm problema de la un nivel superiror, fără a mai fi constrâși, într-o așa măsură, de caracteristicile tehnice.

Această strategie de conceptualizare, mai poartă numele și de bottom-up design. Este foarte avantajoasă din prisma găsirii componentelor constituente, deoarece pot fi găsite, relativ ușor, structuri cu o legătură directă în lumea reală( obiecte, acțiuni etc.). Din păcate această versatilitate vine cu prețul complexității, ea crescând spre măsură ce se avansează pe nivelele inferioare.

De cele mai multe ori se pornește de la specificația proiectului, căutându-se:

* Substantive, care devin eventuale clase candidat
* Verbe ce ar putea juca rolul metodelor din clasă.

Odată realizat pasul de mai sus, ar trebui să avem o idee foarte generală asupra problemei. Pasul următor constă în descrierea funcțională a acesteia. Ce trebuie să facă aplicația la intrările X?

Programul va putea fi accesat de un număr ridicat de persoane, de aceea interfața cu utilizatorul devine punctul de pornire al proiectului. Ea trebuie să permită, într-o manieră cât mai convenabilă, comunicarea utilizatorului cu aplicația.

În cazul Dicționarului, se cunoaște că aplicația trebuie să implementeze operațiile de căutare pe colecții elementare – căutare, inserare, editare și ștergere. Modul de codificare al informației, dar și formatul acesteia ocupă un rol foarte important.

În cadrul Dicționarului m-am decis ca intrările aplicației să codifice string-uri.

## Cazuri de utilizare

Din momentul stabilirii metodei de intrare accentul se va pune, paradoxal, mai mult pe cazurile limită, care vor fi tratate mai special. Schematic, în CP, vorbim de următoarele:

* User-ul introduce un polinom “dezordonat”, iar rezultatul este un polinom minificat și ordonat
* Suma/Diferența/Produsul și Împărțirea între primul polinom și al doilea
* Calcularea rezultatelor numerice pentru primul polinom și al doilea
* Cuvinte speciale utilizate în string-urile ce conțin polinoamele
* Trasarea graficului pentru primul polinom

## Scenarii

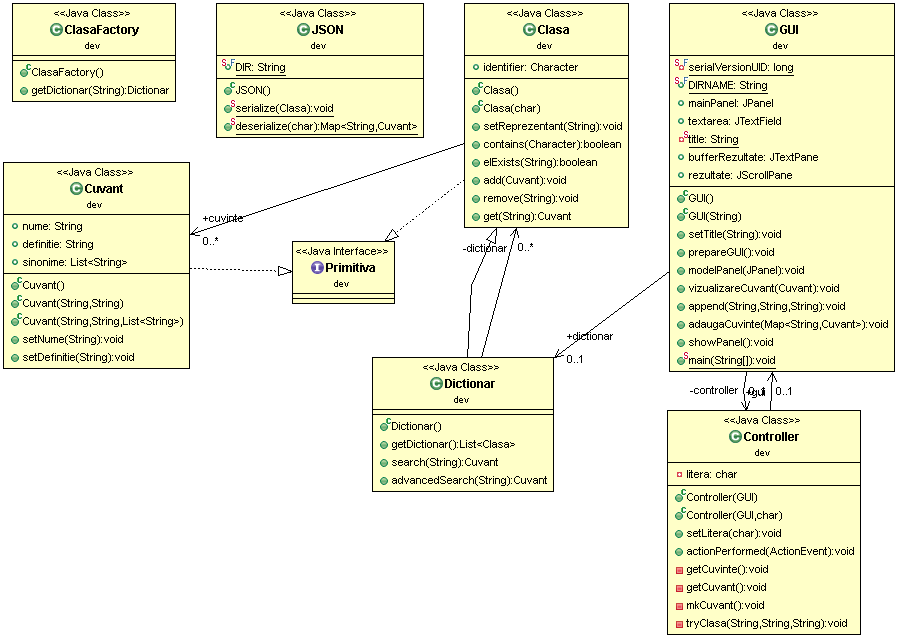
Utilizarea normală a programului presupune introducerea în caseta de căutare a unui cuvânt așteptat și verificarea acestuia în Dicționar. El este de fapt o macrostructură, formată din colecții de clase .

Totuși, cazul de mai sus este puțin probabil și adesea lucruri neașteptate pot să apară. Dintre acestea se remarcă :

* Introducerea de text, fără semnificație numerică, în câmpurile test
* Introducerea de necunoscute suplimentare(pe langă x)
* Folosirea altor cuvinte cheie decât cele specificate
* Scriere corectă a polinoamelor, dar moduri sintactice care diferă puțin de la user la user

# Proiectare

Clasa **GNU**– cu ajutorul ei se creează interfața grafică folosită în tot proiectul. Tot aici se instanțiază și asculatătorii. Ei au rolul de a surprinde eventualele acțiuni ale utilizatorilor prin intermediul evenimentelor. Tocmai de aceea, la nivelul câmpurilor de text și a butoanelor există câte un ascultător separat.



1. Clasa **Cuvânt –** clasă de bază ce permite **memorarea informațiilor despre cuvântul individual, adică definiția sa și lista de sinonime care îl însoștesc.**
2. Clasa **Clasa**– Clasa composite ce agregă cuvintele în primele structuri mai mari prin care se grupează în funcție de litera cu care încep.
3. Clasa **Monom** – clasa fundamentală a proiectului ce definește structura de monom, entitate ce conține doar un coeficient ( real sau întreg), o putere și, opțional, un String pentru a printa rapid monomul. Tot aici, se află și operațiile fundamentale: +, –, \*, / .
4. Clasa **Cartezian** – nu este neapărat necesară, între ea și celelalte clase existând doar o dependență foarte slabă. Această clasă permite trasarea, relativ dinamic, a polinoamelor, ținându-se cont de valorile de intrare.

Pe lângă tipurile de date primitive existente în clasele specificate mai sus, se folosesc cu preponderență și colecții. Una din cele mai importante colecții se află în clasa Polinom, ce conține o listă de monoame. Desigur, există o de compoziție între un polinom și elementele sale fundamentale, monoamele – în absența lor, o instanță Polinom nu există.

În ansamblu, există o similitudine între modul de proiectare a claselor și MVC pattern. Panel joacă rolul unui View, PolyInterpreter este Controller-ul și în final Polinom, Monom și Cartezian reprezintă clase model.

## Clasa Panel

Conține un Jframe pe care se construiește întreaga interfață.

mainFrame.setSize(900,530);

mainFrame.setLayout(**new** GridLayout(1,2));

S-a adoptat un layout de tip grid, în special deoarece chiar de la început se pot remarca două zone distincte ale interfeței cu utilizatorul:

1. Partea de control, reprezentată de câmpurile și butoanele prin care utilizatorul introduce datele și poate să vadă primele rezultate concrete asupra polinoamelor;
2. Partea de afișare grafică, ce folosește clasa Cartezian într-un mod aproape separat de fluxul normal al programului, afișând automat polinomul 1.

În partea de control, există câteva segmente de cod de aceeași natură, care se repetă cu mici variații. De exemplu, cele sașe zone în care se afișează informațiile despre polinoame și rezultatul operațiilor elementare sunt structural identice, atâta doar că își procură informația din alte locuri.

controlLabel = **new** JLabel("", JLabel.***LEFT***);

controlLabel.setPreferredSize(**new** Dimension(350,20));

statusLabel = **new** JLabel("",JLabel.***LEFT***);

statusLabel.setPreferredSize(**new** Dimension(350,20));

JPanel child\_mess = **new** JPanel();

child\_mess.setLayout(**new** GridLayout(2,1));

child\_mess.add(controlLabel);

child\_mess.add(statusLabel);

În snippet-ul de mai sus se poate remarca structura de bază a unei zone de afișare a datelor despre polinoame. Cele două label-uri se află într-un panel cu layout-ul tot de tip grid, deoarece la redimensionarea ecranului, lățimea se modifică și vrem o oarecare repoziționare a câmpurilor.

Metoda showEventDemo() conține cu precădere ascultătorii, care sunt utilizați mai departe în PolyInterpreter.

<component>.setActionCommand("example");

<component>.addActionListener(**new** PolyInterpreter(**this**));

## Clasa Cuvant

Este clasa care gestionează toate acțiunile venite din partea utilizatorilor și le corelează cu funcționalitățile preexistente. Tocmai de aceea, această clasă implementează interfața **ActionListener** și implicit metoda **actionPerformed()**. Există astfel o singură metodă care gestionează toate evenimentele venite din partea de view și asta fiindcă nu avem nevoie de un paralelism la nivelul tratării acțiunilor – în sensul că utilizatorul nu va executa două sau mai multe acțiuni în mod simutan.

Una din cele mai importante trăsături a clasei este că ea conține tot timpul două String-uri. Utilizatorul va introduce tot timpul informația sub formă de text în cele 2 JTextField-uri. Informația este neschimbată atâta timp cât utilizatorul nu actualizează datele din text field-uri. Rămâne doar să “selectăm” între diferitele acțiuni folosindu-ne de ActionCommand setate în clasa Panel.

Indiferent de tipul acțiunii, la fiecare nou apel trebuie să creăm noi polinoame (deoarece informațiile se schimbă), dar asta nu înseamnă că modul de creare al acestora diferă. Metoda crearePolinom() este prima metodă care face cea dintâi parsare a string-urilor.

1. Se verifică dacă există cuvintele cheie **integ** sau **deriv** în fata polinomului. Daca da, atunci se va crea un polinom integrat, respectiv derivat și se vor face operațiile asupra acestor noi polinoame.
2. Dacă nu s-au găsit cuvinte cheie se încearcă “spargerea” string-ului. Pentru a nu fi nevoiți să folosim metode puțin diferite în funcție de context, am ales o abordare mai puțin elegantă, dar eficientă prin care putem trata polinoame cu coeficienți pozitivi sau negativi.

String actualBuffer =(buffer.contains("-")) ? buffer.replace("-"

,"+-") : buffer;

finalBuffer = (actualBuffer.charAt(0) == '+') ?

actualBuffer.substring(1) : actualBuffer;

monoame = finalBuffer.split("\\+");

Polinom poli = **new** Polinom(monoame, panel.x);

La final, această metodă trimite la constructorul clasei Polinom un vector de String-uri, care vor și acolo parsate suplimentar la nivelul fiecărui monom.

# Clasa Clasa

Clasa ce permite crearea efectivă a polinoamelor, ca și colecții de monoame. Ea are un singur constructor public, cel care necesită un vector de String-uri și doi constructori privați ce sunt folosiți strict în structura internă a clasei Polinom, la crearea de polinoame intermediare necesare pentru diferite operații.

La nivelul constructorului public, partea de parsare laborioasă se află în clasa Monom, rezultatul fiind că, în lipsa unei excepții, apelul acestui constructor va crea cu siguranță un obiect de tipul Polinom, ce va conține un ArrayList<Monom> de monoame.

Pe lângă toate acestea, constructorul conține și o metodă de sortare și minificare (*sort\_and\_minify()*). În lipsa acestei metode, ne-am putea întâlni cu situații în care am avea, în același polinom, monoame de acelasi grad. Sortarea este mai mult o operație auxiliară, dar benefică, eficientizând alte metode.

## Crearea claselor de echivalență

În cazul favorabil, adunarea a două polinoame se rezumă la a suma coeficienții monoamelor, rând pe rând cât timp gradul lor este egal. Totuși, sumarea trebuie să fie valabilă pe caz general.

Să luăm cele două polinoame de mai sus. Un algoritm mai general de însumare se reduce la :

În felul acesta tratăm adunarea pentru cele trei cazuri:

**public** Polinom add(Polinom poli2) {

// Definim polinomul rezultant in care vom memora suma

Polinom poli3 = **new** Polinom(x);

**for**(**int** i=0;i<**this**.monom.size();i++) {

//In acest moment polinoamele sunt minime si sortate

**int** p1 = **this**.monom.get(i).getPutere();

**int** c1 = **this**.monom.get(i).getCoeff();

**int** i\_pow = poli2.getIndexPutere(p1);

**if**(i\_pow == -1) {

// Nu am gasit deci adaugam efectiv un nou monom

poli3.monom.add(**new** Monom(c1, p1));

} **else** {

// Am gasit ceva in polinomul 2 deci sumam

**int** p3 = poli2.monom.get(i\_pow).getPutere();

**int** c3 = poli2.monom.get(i\_pow).getCoeff();

poli3.monom.add(**new** Monom(c1+c3, p3));

// Eliminam ce avem deja

poli2.monom.remove(i\_pow);

}

}

// Pasul final mai e sa adaugam ce a ramas in poli2

**for**(**int** j=0;j<poli2.monom.size();j++) {

**int** p2 = poli2.monom.get(j).getPutere();

**int** c2 = poli2.monom.get(j).getCoeff();

poli3.monom.add(**new** Monom(c2, p2));

}

**return** poli3;

}

## Găsirea unui cuvânt în structura de clase

Nu este decât un caz particular de adunare, unde coeficienții polinomului 2 sunt cu semnul inversat.

1. p2.monom.coeficient \* (-1)
2. add(p1, p2)

## Adăugarea consistentă a cuvintelor la dicționar

**public** Polinom times(Polinom poli2) {

Polinom poli3 = **new** Polinom(x);

**for**(**int** i=0;i<**this**.monom.size();i++) {

Monom m1 = **this**.monom.get(i);

**for**(**int** j=0;j<poli2.monom.size();j++) {

Monom m2 = poli2.monom.get(j);

Monom m3 = m1.times(m2);

// Pentru simplificarea procesului, doar adaugam monoamele

poli3.monom.add(m3);

}

}

poli3.sort\_minify();

**return** poli3;

}

Ne folosim de funcția de sortare care la final va ordona și minifica polinomul rezultat.

## Implementare compozită a dicționarului

**public** ArrayList<Polinom> divide(Polinom impartitor) {

ArrayList<Polinom> polis = **new** ArrayList<Polinom>();

Polinom cat = **new** Polinom(x);

Polinom rest = **new** Polinom(x);

rest = **this**;

**if**(rest.getGrad() < impartitor.getGrad()) {

cat.monom.add(**new** Monom(0,0));

} **else** {

**int** i=3;

**while**(rest.getGrad() >= impartitor.getGrad() && i>=0) {

Monom dm = cautaMaxim(rest);

Monom im = cautaMaxim(impartitor);

Monom m = dm.divide(im);

Polinom pm = **new** Polinom(m, x);

cat.monom.add(m);

rest = rest.sub(impartitor.times(pm));

i--;

}

}

**if**(rest.monom.isEmpty()) rest.monom.add(**new** Monom(0,0));

polis.add(cat);

polis.add(rest);

**return** polis;

}

Se poate remarca că în algoritmii prezentați mai sus se fac anumite operații asupra monoamelor. Aceste metode sunt descrise în clasa Monom.

# Clasa JSON

Clasa care conține operațiile elementare de adunare, scădere, înmulțire și împărțire. Aceste operații nu sunt complexe în vreo manieră și nu necesită explicații suplimentare.

La nivelul acestei clase, se află metoda validare(). Ea nu este decât o implementare a unui arbore decizional pentru a trata diferitele cazuri ce apar la introducerea datelor.

**private** **boolean** validare(String monom) {

// Verificam daca exista caractere ciudate - cu exceptia lui ^

**if**(!monom.matches("^[a-zA-Z0-9\\^\\\*\\- ]\*")) **return** **false**;

Pattern polyFormat = Pattern.*compile*("\\^");

Matcher m = polyFormat.matcher(monom);

String s = **new** String();

**while**(m.find()) {

s = m.group();

}

**if**(s.isEmpty()) { // nu contine ^ =>

buffer = monom.split("[a-zA-Z]");

**if**(buffer.length == 0) {

coeficient = 1;

putere = 1;

} **else** {

coeficient = (!buffer[0].isEmpty()) ? Integer.*parseInt*(buffer[0]) : 1;

putere = (buffer[0] == monom) ? 0 : 1;

}

} **else** { // contine ^ =>

buffer = monom.split("\\^");

**try** {

String nrStr = **new** String();

**for**(**int** i = 0; i < buffer[0].length(); i++){

**char** c = buffer[0].charAt(i);

**if**(c==45) nrStr += c;

**if**(c > 47 && c < 58)nrStr += c;

}

coeficient = (nrStr.isEmpty()) ? 1 : Integer.*parseInt*(nrStr);

putere = Integer.*parseInt*(buffer[1]);

} **catch**(NumberFormatException e) { //

System.***out***.println("Formatul nu este valid");

}

}

**return** **true**;

}

Schematizat, clasa de mai sus face următoarele verificări asupra string-ului formatat, înainte de a crea monoamele specifice:

1. Verifică dacă string-ul conține caractere speciale care nu au ce căuta în descrierea unui polinom(simboluri de tipul **!**, **?**, **;** etc.)
2. Se caută apoi după **^** și se tratează separat situațiile: când există și când nu.
3. Căutăm apoi coeficienții din fața lui x, și tratăm cazurile când nu există nimic în fața acestuia, sau când nu există x.

# Clasa Factory

Este clasa cu care se desenează automat polinomul 1. Această clasă extinde JPanel și suprascrie metoda void paintComponent(Graphics g). Funcționalitatea principală se află în corpul a două metode, cea menționată și cea care calculează valoarea funcției date ca parametru.

Primul pas este să cream axele de coordonate, iar mai apoi să ne desenăm valori într-o manieră dinamică. Pentru aceasta este necesar să folosim un raport de scalare, pentru a ști fată de ce valori să ne raportăm. Ideea este simplă:

1. În loc să lăsăm dimensiunile fixe, lucrăm cu un raport
2. Apoi doar desenăm în locații k \* raport pentru a obține acea scalare dorită

**double** min = (-*maxValue*), max = *maxValue*, ratio = size/(max\*2), fx;

**for**(;min<=max;min+=0.0025){

//O rotunjire bruta - interesant oricum

min=Math.*round*(min\*1000.0)/1000.0;

fx=Math.*round*((**this**.f(min))\*1000.0)/1000.0;

g.drawLine((**int**)(size/2+(ratio\*min)), (**int**)(size/2-(ratio\*fx)),

(**int**)(size/2+(ratio\*min)), (**int**)(size/2-(ratio\*fx)));

}

Cheia unei afișări corespunzătoare este găsirea unui raport la care să raportăm tot timpul valorile de axa Ox cu cele de pe Oy. În esență ne trebuie un factor de scalare care să depindă de lățimea pe care putem desena în raport cu intervalul Ox pe care lucrăm. De aceea, cel mai important lucru este să definim un raport:

# Rezultate

Cu excepția unor mici cazuri când programul este forțat la în zona de input-uri, programul nu va rula corespunzător deoarece nu există parte de parsare pentru eventualele string-uri aleatoare.

În rest, atât rezultatele, cât și valorile se află în grila celor așteptate, programul având un oarecare grad de flexibilitate la nivelului modului de procesare a polinoamelor și parsare a expresiilor speciale.

# **Concluz**ii

Din această primă temă, mi-am dat seama, din păcate la final de importanța sepărării, la nivel conceptual, a parsării string-urilor fată de interpretarea lor de către clasele Polinom și Monom. Această greșeală am facut-o în principal, fiindcă nu am reușit să estimez complexitatea reală a problemei. O clasă separată, Parsare, de exemplu, ar fi fost o alternativă mult mai viabilă fiindcă în acest fel nu ar fi fost nevoie să existe vreo legătură între procesul de determinare al monoamelor din string-uri și crearea efectivă a polinomului.

Ca și dezvoltări ulteriore, ideea de a converti dintr-o bază polinomială într-o alta pare foarte atractivă, necesitând algoritmi pentru manipularea matricilor de transformare a bazelor. Desigur, o bază generală ar pune mari probleme de implementare, dar și din punct de vedere matematic, așa că implementarea unor baze standard(Laplace, Newton, Legendre) pare o alternativă fezabilă.

Ce am învățat este rolul pe care separarea obiectivelor îl are chiar la începutul problemei. Nu o dată, a trebuit să elimin bucăți semnificative de cod fiindcă încercam să compensez lipsa de rigurozitate din faza de planificare cu clauze suplimentare.

# Bibliografie

1. <http://stackoverflow.com>
2. <https://docs.oracle.com>
3. <http://forum.codecall.net/>