MINISTERUL EDUCAȚIEI ȘI CERCETĂRII ȘTIINȚIFICE
UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Calculator de polinoame. Documentație

Grigorescu M. Alexandru
Departamentul de calculatoare, Universitatea Tehnică, 30223
Tehnici de programare
Prof. dr. ing. Tudor Cioara
21 Martie, 2024

Cuprins

Obiectivul temei

Este dorită elaborarea unei aplicații dedicate manipulării polinoamelor, o componentă esențială în studiul matematicii și în numeroase domenii tehnice și științifice. Pentru realizarea proiectului trebuie respectate următoarele etape esențiale pentru a asigura dezvoltarea eficientă și eficace a aplicației.

Determinarea unor cazuri de utilizare

Cazurile de utilizare (sau usecase-urile) sunt scenarii specifice în care utilizatorii vor interacționa cu sistemul nostru și vor utiliza funcționalitățile oferite de acesta. Prin definirea și înțelegerea acestor usecase-uri, putem ghida procesul de dezvoltare și ne asigurăm că aplicația noastră va satisface nevoile utilizatorilor într-un mod complet și eficient. Informații suplimentare și detaliate sunt furnizate în cadrul Capitolului 2, unde subiectul este explorat în profunzime.

Proiectarea algoritmilor de prelucrare a polinoamelor

Procesul de proiectare a algoritmilor pentru manipularea polinoamelor constituie prima etapă în dezvoltarea aplicației propuse. Această etapă implică identificarea și definirea operațiilor de bază, adunarea, scăderea, înmulțirea, împărțirea, derivarea și integrarea polinoamelor. În plus, este necesară analiza eficienței și optimizarea algoritmilor pentru a asigura o performanță optimă a aplicației în condiții diverse de utilizare. Informații suplimentare și detaliate sunt furnizate în cadrul Capitolului 2, unde subiectul este explorat în profunzime.

Modelarea tipurilor de date utilizate

Alegerea unor structuri de date adecvate este crucială pentru eficiența algoritmilor implementați. De asemenea, este important să se țină cont de aspectele legate de gestionarea memoriei și de eficiența în timpul de execuție. Informații suplimentare și detaliate sunt furnizate în cadrul Capitolului 2, unde subiectul este explorat în profunzime.

Crearea unei arhitecturi flexibile

O arhitectură bine definită și flexibilă reprezintă o condiție prealabilă pentru o dezvoltare rapidă și eficientă a aplicației. Prin adoptarea unor principii de proiectare software, cum ar fi modularitatea, coeziunea și cuplajul redus, se asigură o structură a codului care poate fi ușor extinsă și modificată în viitor, în funcție de cerințele și feedback-ul utilizatorilor. Informații suplimentare și detaliate sunt furnizate în cadrul Capitolului 3, unde subiectul este explorat în profunzime.

Implementarea unei interfețe grafice prietenoase cu utilizatorul

Interfața grafică reprezintă componenta prin care utilizatorii interacționează cu aplicația și, în consecință, joacă un rol crucial în experiența lor de utilizare. Crearea unei interfețe grafice intuitive și prietenoase, care să faciliteze utilizatorilor accesul și utilizarea funcționalităților oferite de aplicație, necesită o atenție deosebită acordată designului ergonomic, navigației intuitive și feedback-ului vizual. Informații suplimentare și detaliate sunt furnizate în cadrul Capitolului 3, unde subiectul este explorat în profunzime.

Analiza problemei, modelare, scenarii, cazuri de utilizare

Cerințe Funcționale

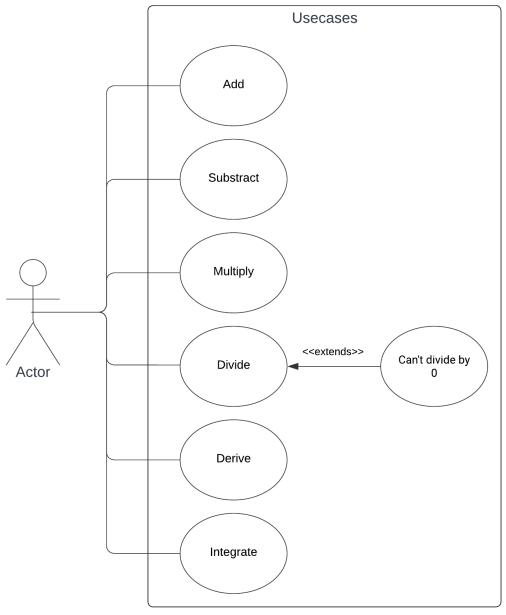
Aplicația *calculatordepolinoame.ro* trebuie să ofere următoarele servicii:

- Trebuie să permită utilizatorului să introducă unul sau două polinoame
- Trebuie să permită utilizatorului să aleagă tipul de operație matematică
- Trebuie să adauge două polinoame
- Trebuie să scadă două polinoame
- Trebuie să înmulțească două polinoame
- Trebuie să împartă două polinoame
- Trebuie să deriveze un polinom
- Trebuie să integreze un polinom
- Trebuie să trateze cazurile de excepții

Cerințe non funcționale

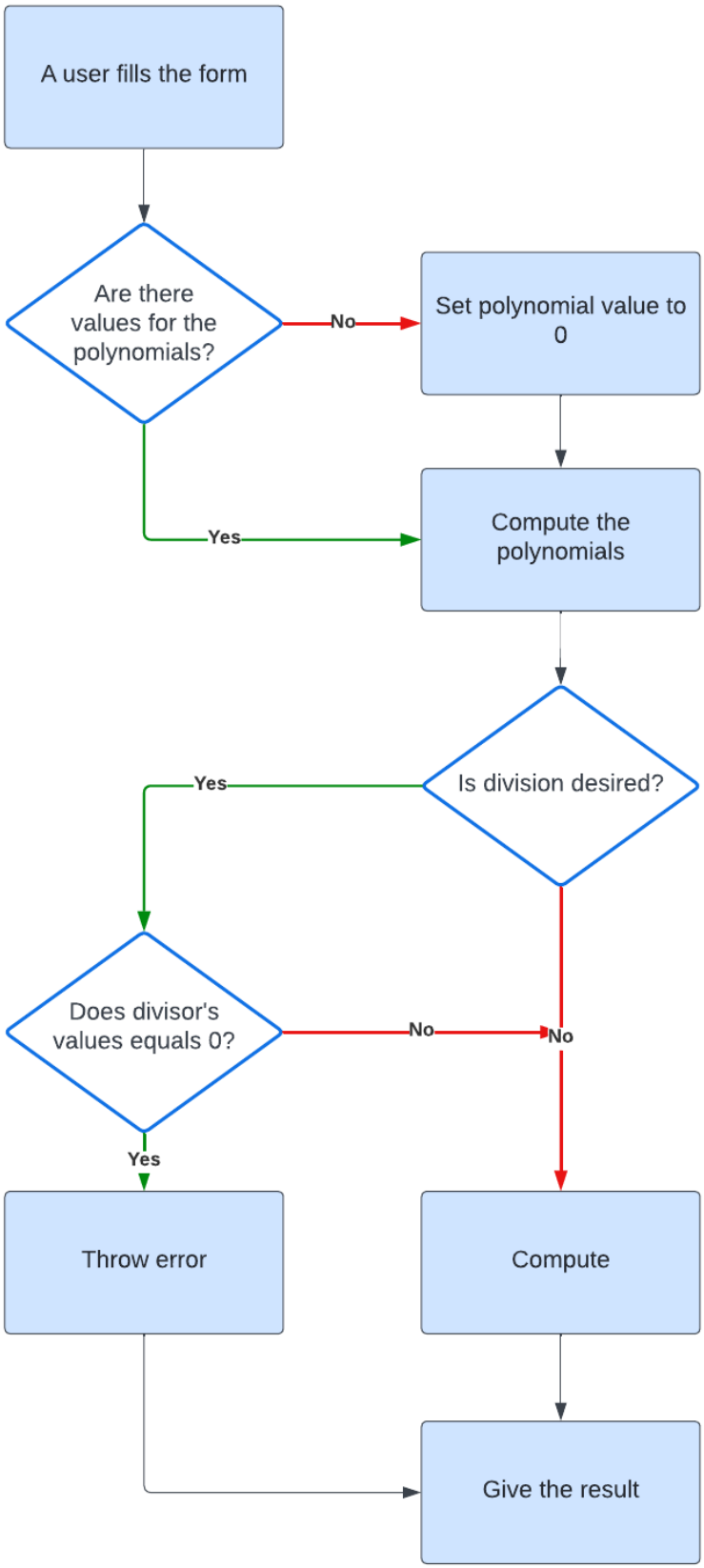
Aplicația *calculatordepolinoame.ro* trebuie să aibă următorul comportament:

- Trebuie să informeze utilizatorul în cazul erorilor de sistem
- Trebuie să aibă o interfață prietenoasă cu utilizatorul
- Trebuie să poată returnă rezultatul oricărui tip de ecuație în sub o secundă
- Trebuie să funcționeze pe orice tip de calculator/laptop
- Trebuie să fie construit modular, astfel încât adăugarea noilor funcționalități să fie rapidă



Cazuri de utilizare

Diagrama alăturată prezintă toate cazurile de utilizare tratate de *calculatoruldepolinoame.ro.* În continuare se poate observa cum fiecare request al utilizatorului este procesat de sistemul nostru. În dorința de îmbunătății cât mai mult experiența utilizatorului, sistemul nostru tratează orice input invalid drept polinomul “0”.


A user fills the form
Are there values for the polynomials?
No
Set polynomial value to 0
Yes
Compute the polynomials
Is division desired?
Yes
Does divisor's values equals 0?
No
No
Yes
Throw error
Compute
Give the result

Proiectarea algoritmilor de prelucrare a polinoamelor

Algoritmii de prelucrare a polinoamelor sunt realizați după cum urmează. Pentru simplitate în discurs ne vom referi la primul polinom drept P și la al doilea drept Q. Monomul este cea mai mică parte a unui polinom, el este compus din exponent și coeficient.

Adunarea, scădere

Pentru a aduna două polinoame am însumat coeficienții monoamelor de același grad. Analog am scăzut din coeficientul monomului lui P coeficientul monomului lui Q.

Înmulțirea

Pentru realizarea acestei operații trebuie să înmulțim pe rând fiecare element din P cu fiecare element din Q.

Împărțirea

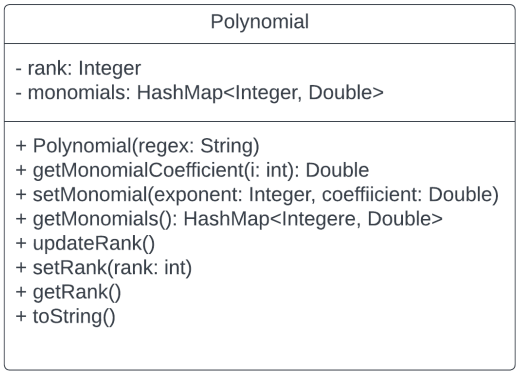
Operația de împărțire poate să returneze atât câtul împărțirii, cât și restul acesteia. Arhitectura sistemul (vezi cap. 3) filtrează cererile de calcul pentru împărțiri astfel încât sunt evitate cazurile în care împărțitorul este 0 sau câtul urmează sa fie 0 (rangul lui P este mai mic decât cel al lui Q). Pentru a calcula restul împărțim pe primul monom al lui P cu primul monom al lui Q. Acest rezultat (R) este înmulțit cu Q și scăzut din P. P ia valoarea ultimei operații, iar procesul se repetă până când rangul lui P este mai mic decât al lui Q. În acest moment, valoarea restului este salvată în P. Câtul urmează aceeași regulă, însă de data asta valorile lui R sunt adăugate într-un nou polinom, polinomul cât.

Derivarea, integrarea

Pentru efectuarea acestei operațiuni, este parcurs fiecare monom în parte și sunt aplicate regulile de derivare, respectiv integrare.

Modelarea structurilor de date

Odată ce am definit toate operațiile în secțiunea anterioară putem înțelege cum ar trebui să se comporte un polinom. Știm că este format dintr-o listă de monoame și are un rang. Totodată din usecase-uri putem trage concluzia că pentru a crea un polinom avem nevoie de un șir de caractere care îl definește matematic. Cât despre comportament știm următoarele: trebuie să își poate seta și obține monoamele și rangul.



Din diagrama UML se observă în plus două metode care în esență fac același lucru, însă au implementări diferite: *updateRank*(calculează singur rangul polinomului O(numărul de monoame)) și *setRank*(setează rangul la o valoarea specificată O(1)).

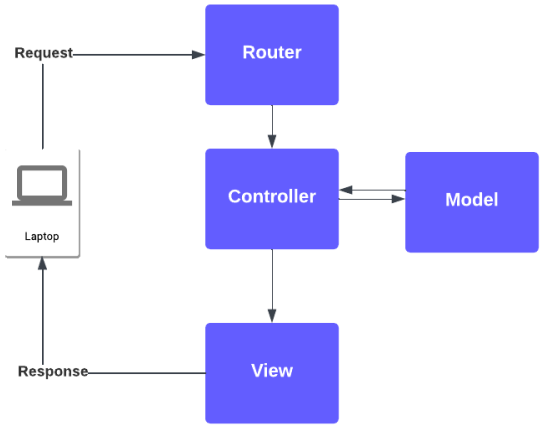
Argumentul împotriva instituirii unei clase specializate *Monomial* în contextul matematicii se întemeiază pe stabilitatea și durabilitatea definițiilor matematice. În matematică, definițiile sunt adesea considerate entități stabilite și puțin susceptibile la schimbări. Prin urmare, este puțin probabil ca o clasă dedicată *Monomial* să fie modificată în timp.

În plus, alegerea de a nu implementa o clasă specializată pentru *Monomial* este justificată și de simplitatea soluției alternative propuse. Această alternativă constă în utilizarea unui simplu *HashMap*, care oferă o implementare eficientă și ușor de înțeles pentru gestionarea monoamelor.

Prin urmare, prin evitarea unei clase specializate în favoarea unei soluții mai generice și simplificate, se obține o structură mai ușor de întreținut și de înțeles în contextul matematic.

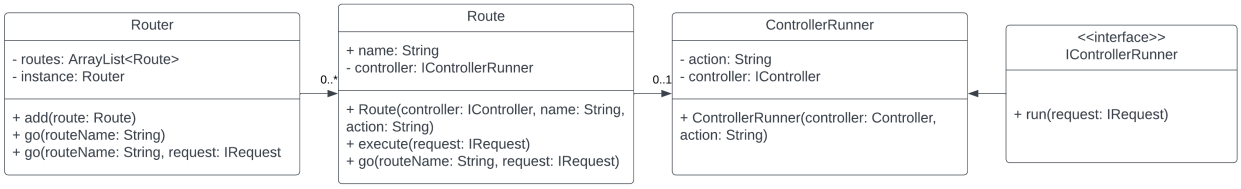
Proiectare

Întrucât aplicația dispune de o interfață grafică am ales o arhitectură de tipul MVC. Modul în care datele sunt transmise de la utilizator la aplicație și înapoi la utilizator este descris în diagramă de mai jos (figura 1). În următoarele secțiuni voi detalia modul de funcționare al sistemului



Routing

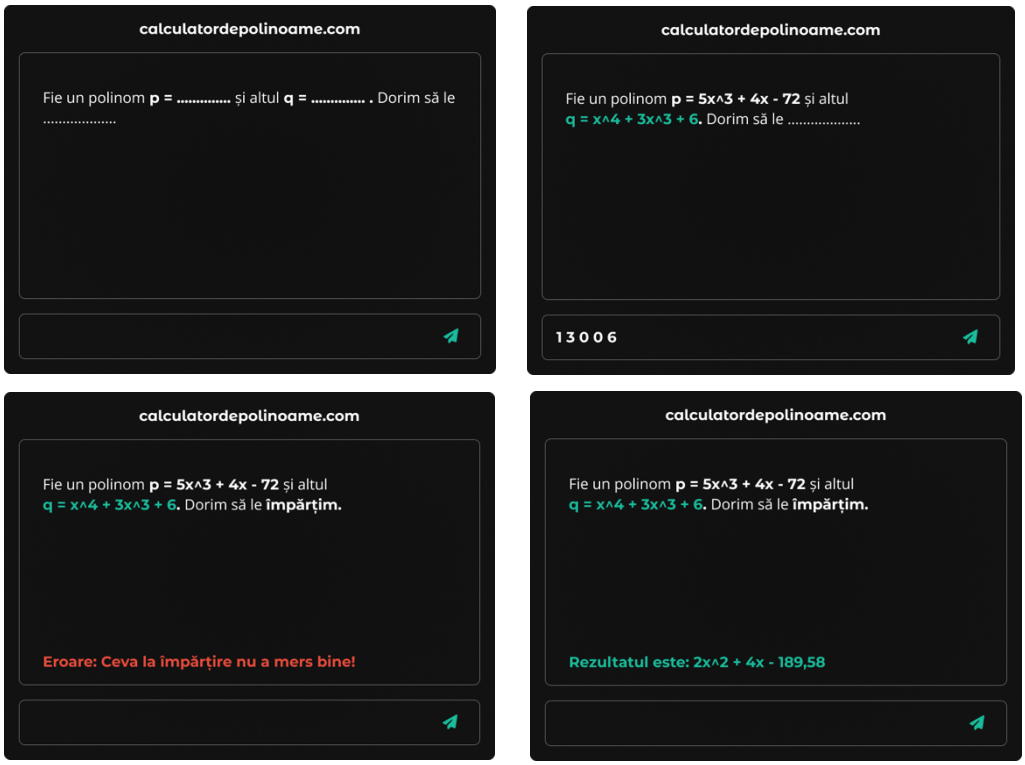
Pentru a asigura o dezvoltare rapidă pe viitor, am creat un sistem de rutare care permite adăugarea rapidă și ușoară a rutelor noi. În cadrul aplicației, există o singură instanță a Router-ului care poate fi accesată din orice loc. Procesul începe odată ce cererea este trimisă către Router, care o direcționează către ruta corespunzătoare. Apoi, cererea este trecută către o clasă numită ControllerRunner, care, la rândul său, apelează metoda action din controler, având cererea ca parametru. Detaliile acestei implementări sunt prezentate în figura 3. Prin acest sistem, se îmbunătățește flexibilitatea și eficiența în gestionarea rutelor și a controlerelor, permițând o dezvoltare agilă și o adaptare rapidă la cerințele viitoare.



Interfața grafică

Interfața a fost proiectată folosind Figma, o unealtă ce permite crearea rapidă și ușoară a design-urilor. Aplicația poate fi în una dintre cele trei stări: inițială, în curs de completare, sau în stare de succes sau eroare. Abordarea în crearea design-ului a fost minimalistă, evitând supraîncărcarea utilizatorului cu toate opțiunile disponibile încă de la început. În schimb, sunt prezentate doar opțiunile relevante pentru stadiul curent al aplicației. Pentru a îmbunătăți experiența utilizatorului, am inclus și o schemă de culori pentru text, care reprezintă polinomul în curs de scriere. La finalizarea procesului, utilizatorul își selectează operația dorită prin intermediul unui meniu drop-

down. Orice rezultate sau erori generate sunt afișate la finalul secțiunii de text. Aceste elemente contribuie la o experiență mai plăcută și mai intuitivă pentru utilizator.

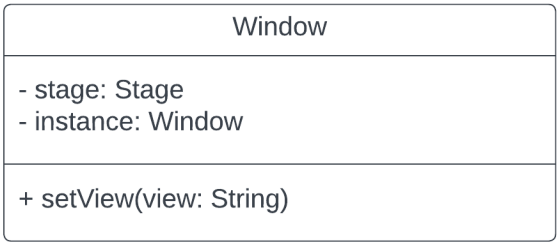


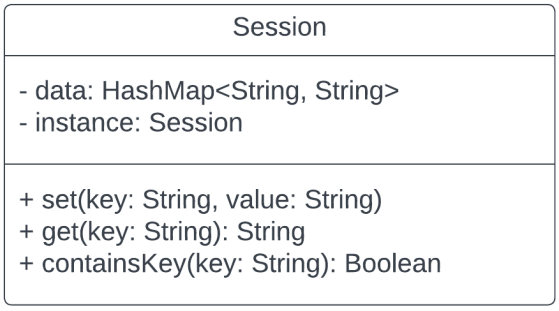
Integrarea interfeței grafice

Pentru crearea interfeței grafice, s-a utilizat JavaFX, menținând consistența în culori și formatare cu ajutorul unor clase ajutătoare: *Colors* (care salvează culorile personalizate folosite în proiect, cum ar fi roșu și verde) și *ComboBoxOptions* (deoarece un *ComboBox* nativ nu poate avea valori separate pentru textul afișat pe ecran și valoarea folosită în cerere, această clasă permite acest lucru).

Conform metodologiei JavaFX, view-ul este construit din două părți: fișierul .fxml și o clasă Controller care se ocupă de interacțiunea cu aplicația. În acest controller, sunt definite variabile pentru câmpurile de text pentru P și Q (polinoame), mesajul de eroare, ComboBox și input-ul utilizatorului. În ComboBox sunt enumerate operațiile posibile pe care le poate realiza utilizatorul. Atunci când se apasă pe câmpul unui polinom, acesta devine selectat, inputul primește valoarea acestuia și culoarea se schimbă în verde. Atunci când butonul de trimitere este apăsat, este creată o cerere și este trimisă către router (vezi secțiunea de Routing).

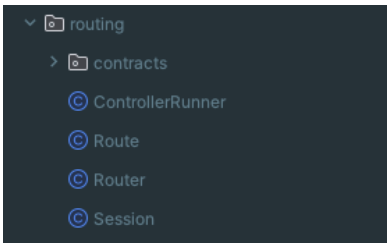
Această interfață este integrată în restul aplicației prin intermediul unui sistem de view-uri. Acest sistem are la bază o clasă *Window* de tip singleton care oferă un API pentru controllere, prin intermediul căruia pot fi setate view-urile după numele acestora. Pentru a putea transmite date de la controller la view, a fost creată o clasă Session care se comportă similar cu cea implementată de browser.

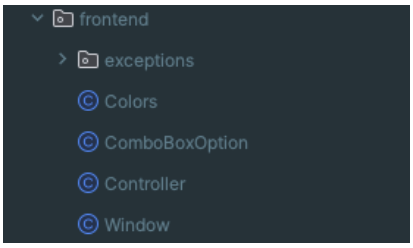


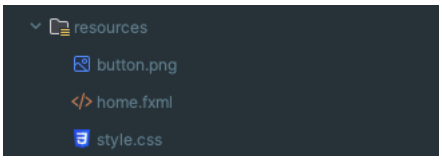


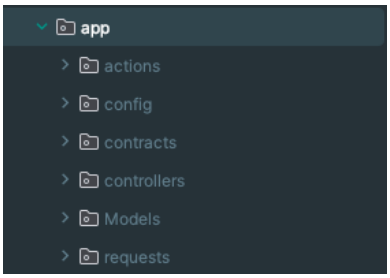
Structura pachetelor

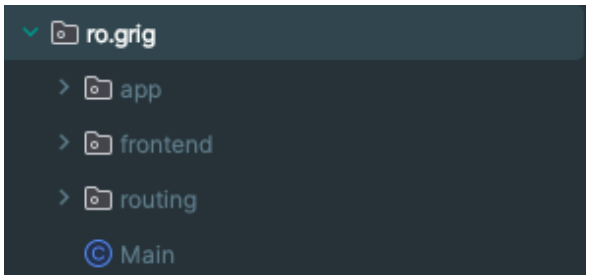
Pachetele pentru această aplicație au fost structurate într-o manieră care permite modularizarea aplicației, făcând ușor convertirea modului de lucru într-un framework. Pachetele care se ocupă de gestionarea flow-ului de date sunt separate de pachetul în care se poate găsește logica afacerii și implementarea acesteia (*app*). În *resources* se pot găsi fișiere folosite de JavaFX views, CSS și imagini.











Implementare

Pentru implementare am ales un sistem bazat pe acțiuni. Pentru fiecare funcție pe care trebuie să o îndeplinească aplicația este creat un contract (o interfață) care descrie modul de funcționare. Pentru fiecare contract se pot găsi una sau mai multe implementări (în funcție de cerințele afacerii). Implementarea algoritmilor pentru operațiile matematice au fost discutați în *Capitolul 2: Proiectarea algoritmilor de prelucrare a polinoamelor*

Containere de injecție de dependințe

Asocierea unui contract cu o implementare se realizează prin intermediul unui Container de injecție de dependințe (DI Container, găsit în aplicație sub numele de *AppProvider*). El ne permite să ne referim în cod direct la contract, fără să ne intereseze implementarea. Asocierea între contract și implementare se realizează în *Main*, la începutul aplicației. Codul următor demonstrează crearea legăturii:

```
AppProvider.set(AddsPolynomials.class, new AddPolynomials());
AppProvider.set(SubtractsPolynomials.class, new SubtractPolynomials());
AppProvider.set(MultipliesPolynomials.class, new MultiplyPolynomials());
AppProvider.set(DividesPolynomials.class, new DividePolynomials());
AppProvider.set(DerivesPolynomials.class, new DerivePolynomials());
AppProvider.set(IntegratesPolynomials.class, new IntegratePolynomials());
```

Totodată pentru a putea face tranziția de la operația primită prin request la contractul dorit am creat un *CalculatorProvider* care se ocupă de parsarea operației la contract. Alegerea acestei abordări s-a datorat preferinței pentru flexibilitatea și ușurința de modificare a soluției, în detrimentul unei funcții cu mulți if-uri sau unui switch lung. Mai jos găsiți codul de inițializare al CalculatorProvider-ului:

```
CalculatorProvider.set("add", AddsPolynomials.class);
CalculatorProvider.set("subtract", SubtractsPolynomials.class);
CalculatorProvider.set("multiply", MultipliesPolynomials.class);
CalculatorProvider.set("divide", DividesPolynomials.class);
CalculatorProvider.set("derive", DerivesPolynomials.class);
CalculatorProvider.set("integrate", IntegratesPolynomials.class);
```

Datorită asemănării semnificative între implementările celor două clase, este ușor de înțeles modul lor de funcționare observând doar una dintre ele:

```
public class AppProvider {
    private final Map<Class<?>, Object> bindings = new HashMap<>();
    private static final AppProvider instance = new AppProvider();

    public static <T> void set(Class<T> clazz, T target) {
        instance.bindings.put(clazz, target);
    }

    public static <T> T get(Class<T> clazz) {
        return clazz.cast(instance.bindings.get(clazz));
    }
}
```

Routing

După cum a fost discutat în secțiunea anterioară, obiectivul principal al arhitecturii este să creeze un mediu ușor de manipulat și extins. Deși a fost prezentat fluxul router-ului anterior, în această secțiune se dorește o expunere mai detaliată a modului în care rutele sunt integrate în aplicație. Codul de mai jos oferă o ilustrare comprehensivă a procesului de definire a rutelor. Fiecare rută este compusă dintr-un nume unic, utilizat pentru navigare, precum și un controller asociat și o acțiune specifică, localizată în interiorul controller-ului respectiv. Această abordare nu numai că simplifică gestionarea rutelor în aplicație, dar și permite o mai mare flexibilitate în adaptarea și extinderea funcționalității acesteia în viitor.

```
Router.add(new Route("home", PolynomialController.class, "index"));
Router.add(new Route("calculate", PolynomialController.class, "calculate"));
```

Procesarea operațiilor

Modul în care operațiile sunt procesate este simplu, permițând adăugarea și ștergerea operațiilor cu ușurință. Tot ce trebuie făcut este să se creeze un contract, să se implementeze acesta, să se asocieze între ele cu ajutorul clasei *AppProvider* și să se asocieze unui cuvânt cu contractul în *CalculatorProvider*. Algoritmul care parsează request-ul și execută operația dorită se poate găsi în controller. Aici, funcția compute execută operația după ce controller-ul a validat datele de intrare. Datorită modului în care a fost implementat constructorul pentru clasa *Polynomial*, orice input invalid va fi transformat în zero. Astfel, nu mai este necesară validarea polinomului introdus de utilizator.

```
public static void calculate(PolynomialCalculateRequest request) {
    if (!CalculatorProvider.getAvailableOperations().contains(request.operation)) {
        Session.set("error", "Operația selectată nu există");
        Router.go("home");
        return;
    }

    Polynomial P = new Polynomial(request.P);
    Polynomial Q = new Polynomial(request.Q);

    if (Q.getRank() == 0 && Objects.equals(request.operation, "divide")) {
        Session.set("error", "Nu poți împărții la zero");
        Router.go("home");
        return;
    }

    Session.set("result", compute(request.operation, P, Q));
    Router.go("home");
}

private static String compute(String operation, Polynomial P, Polynomial Q) {
    if (operation.equals("divide")) {
        DividesPolynomials divider = AppProvider.get(DividesPolynomials.class);
        return "Câtul este: " + divider.handle(P, Q).toString() +
                " și restul este: " + divider.getRemainder(P, Q).toString();
    }

    return AppProvider.get(CalculatorProvider.get(operation)).handle(P, Q).toString();
}
```

Interfața grafică

În procesul de afișare al unui view pot apărea trei tipuri de erori potențiale. Pentru fiecare dintre aceste situații, am stabilit un set de coduri de eroare, menite să faciliteze procesul de depanare al programului. Conform convenției stabilite, codul de eroare este alcătuit din codul procesului care a generat eroarea, codul acțiunii care a cauzat-o și pasul la care a intervenit aceasta. Deoarece doar clasa Window are capacitatea de a genera astfel de erori și este definită ca primul element, aceasta va fi identificată cu codul 1. Mai jos găsiți lista completă a erorilor definite:

- Fereastra nu a fost creată: cod de stare 110
- View-ul nu a fost creat: cod de stare 120
- Imposibilitate de citire din fișier: cod de stare 121

Testare

În cadrul testării, a fost adoptată o abordare bazată pe comportament și dezvoltare ghidată de comportament (BDD), folosind Cucumber. Această metodă permite redactarea de teste ușor de înțeles pentru toate părțile implicate în proiect, deoarece acestea sunt exprimate într-un limbaj natural, apropiat de limba engleză, în loc de cod complex.

Prin BDD, au fost definite scenarii de testare care descriu comportamentul dorit al aplicației, în loc să ne concentrăm pe implementarea internă a funcționalității. (vezi *Capitolul 2: Cazuri de utilizare*).

Cucumber permite redactarea acestor scenarii de testare sub forma unei "documentații vii" (living documentation), ușor de citit și de înțeles pentru toți membrii echipei. Aceste teste pot fi rulate automat și servesc ca un instrument puternic pentru a asigura că aplicația funcționează conform specificațiilor și că orice modificare nu afectează comportamentul existent.

În plus, prin utilizarea BDD și Cucumber, au fost izolate clar funcționalitățile și au fost identificate rapid eventualele probleme sau discrepanțe între cerințele clientului și implementarea efectivă a acestora în aplicație. Această abordare a contribuit la dezvoltarea și livrarea produselor de calitate, satisfăcând nevoile și așteptările utilizatorilor.

Scenarii

Pentru fiecare operație matematică, am definit scenarii de testare care descriu comportamentul dorit al funcției respective. Mai jos găsiți operaitons.feature

```
Feature: Polynomial Mathematical Operations

 Scenario: Adding two polynomials
   Given I have polynomial P: "3x^2 + 2x + 1"
   And I have polynomial Q: "4x^3 - x^2"
   When I add polynomials P and Q
   Then the result is: "1.0 + 2.0x + 2.0x^2 + 4.0x^3"

 Scenario: Subtracting two polynomials
   Given I have polynomial P: "3x^2 + 2x + 1"
   And I have polynomial Q: "4x^3 - x^2"
   When I subtract polynomials P and Q
   Then the result is: "1.0 + 2.0x + 4.0x^2 - 4.0x^3"

 Scenario: Multiplying two polynomials
   Given I have polynomial P: "3x^2 + 2x + 1"
   And I have polynomial Q: "4x^3 - x^2"
   When I multiply polynomials P and Q
```

```
    Then the result is: " - x^2 + 2.0x^3 + 5.0x^4 + 12.0x^5"

  Scenario: Dividing two polynomials
    Given I have polynomial P: "x^3 - 2x^2 + 6x - 5"
    And I have polynomial Q: "x^2 - 1"
    When I divide polynomials P and Q
    Then the result is: "Câtul este:  - 2.0 + x și restul este:  - 7.0 + 7.0x"

  Scenario: Derivative of a polynomial
    Given I have polynomial P: "3x^2 + 2x + 1"
    When I take the derivative of polynomial P
    Then the result is: "2.0 + 6.0x"

  Scenario: Integration of a polynomial
    Given I have polynomial P: "3x^2 + 2x + 1"
    When I integrate polynomial P
    Then the result is: "x + x^2 + x^3"
```

Concluzii

În urma dezvoltării proiectului calculatordepolinoame.ro, am acumulat cunoștințe substanțiale și am identificat perspective promițătoare pentru evoluția ulterioară a aplicației.

În ceea ce privește noțiunile dobândite, am ajuns la o înțelegere profundă a mecanismelor Dependency Injection (DI) și a rolului fundamental pe care DI Containerul îl joacă în gestionarea dependențelor în cadrul unei aplicații. De asemenea, am dezvoltat o perspectivă mai nuanțată asupra framework-urilor de backend, cu accent pe Laravel, și am înțeles modul în care acestea facilitează dezvoltarea aplicațiilor web complexe și scalabile.

Privind către viitor, am identificat mai multe direcții de interes pentru dezvoltarea ulterioară a aplicației. Printre acestea se numără extinderea capacității de gestionare și raportare a erorilor, optimizarea performanței prin implementarea unui sistem eficient de caching pentru operațiile frecvent utilizate, dezvoltarea unei funcționalități de istoric pentru monitorizarea și analiza operațiilor anterioare și extinderea funcționalității polinoamelor pentru a permite utilizatorilor să opereze cu funcții în calitate de coeficienți și exponente pentru monoame.

Bibliografie

1. *Operațiile matematice:* https://dsrl.eu/courses/pt/materials/PT_2024_A1_S1.pdf
2. *Setare și rulare Cucumber:* https://cucumber.io/docs/guides/10-minute-tutorial/?lang=java
3. *DI Container:* https://www.youtube.com/watch?v=y7EbrV4ChJs&ab_channel=LaraconEU
4. *Acțiuni:* https://www.youtube.com/watch?v=0Rq-yHAwYjQ&ab_channel=LaraconOnline