**FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE**

**DEPARTAMENTUL CALCULATOARE**



**Simulator de comenzi depozit. Documentație**

Grigorescu M. Alexandru

Departamentul de calculatoare, Universitatea Tehnică, 30223

Tehnici de programare

Prof. dr. ing. Tudor Cioara

16 Mai, 2024

# Cuprins

[Cuprins 2](#_Toc166769160)

[Obiectivul temei 3](#_Toc166769161)

[Determinarea unor cazuri de utilizare 3](#_Toc166769162)

[Proiectarea algoritmilor de prelucrare a polinoamelor 3](#_Toc166769163)

[Modelarea tipurilor de date utilizate 3](#_Toc166769164)

[Crearea unei arhitecturi flexibile 3](#_Toc166769165)

[Implementarea unei interfețe grafice prietenoase cu utilizatorul 3](#_Toc166769166)

[Analiza problemei, modelare, scenarii, cazuri de utilizare 4](#_Toc166769167)

[Cerințe Funcționale 4](#_Toc166769168)

[Cerințe non funcționale 4](#_Toc166769169)

[Cazuri de utilizare 4](#_Toc166769170)

[Proiectare 5](#_Toc166769171)

[Routing 5](#_Toc166769172)

[Evenimente 5](#_Toc166769173)

[Interfața grafică 5](#_Toc166769174)

[Integrarea interfeței grafice 6](#_Toc166769175)

[Structura pachetelor 7](#_Toc166769176)

[3 Layer Arhitecture 8](#_Toc166769177)

[Routing 8](#_Toc166769178)

[Interfața grafică 9](#_Toc166769179)

[Concluzii 9](#_Toc166769180)

[Bibliografie 11](#_Toc166769181)

# Obiectivul temei

Un sistem de management al comenzilor pentru un depozit este ca un dirijor pentru o orchestră complexă de operațiuni logistice. Similar cu sistemul de administrare a cozilor din magazine, acesta are ca scop principal organizarea și optimizarea fluxului de mărfuri în depozit pentru a minimiza timpii de așteptare și a maximiza eficiența operațională.

### 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 cozilor constituie prima etapă în dezvoltarea aplicației propuse.

### 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

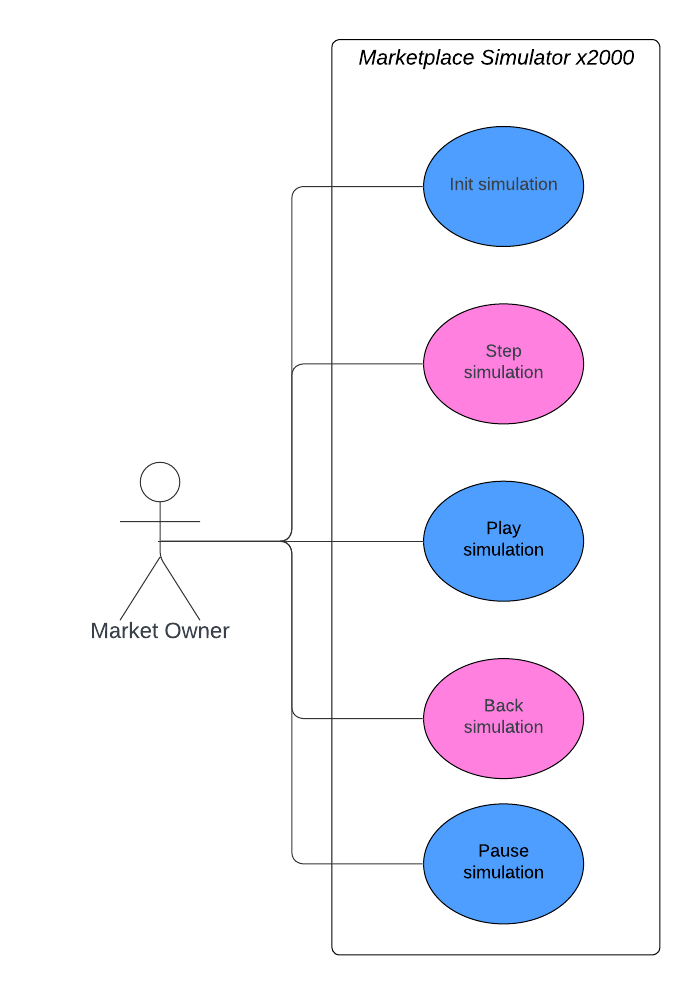
### Cerințe Funcționale

Aplicația *Marketplace Simulator x2000* trebuie să ofere următoarele servicii:

* CRUD pentru clienti
* CRUD pentru produse
* CRUD pentru comenzi
* Sistem de Log-uri

### Cerințe non funcționale

Aplicația *Emag* trebuie să aibă următorul comportament:

* Trebuie să informeze utilizatorul în cazul erorilor de sistem
* Trebuie să aibă o interfață prietenoasă cu utilizatorul
* Trebuie să funcționeze pe orice tip de calculator/laptop

### Cazuri de utilizare

Diagrama alăturată prezintă toate cazurile de utilizare tratate de *Emag.* În continuare se poate observa cum fiecare request al utilizatorului este procesat de sistem.

# A diagram of a computer system Description automatically generatedProiectare

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). Totodată datorită volumului de muncă mare asincron am ales să combin arhitectura MVC cu una bazată pe evenimente. În următoarele secțiuni voi detalia modul de funcționare al sistemului.

### Routing

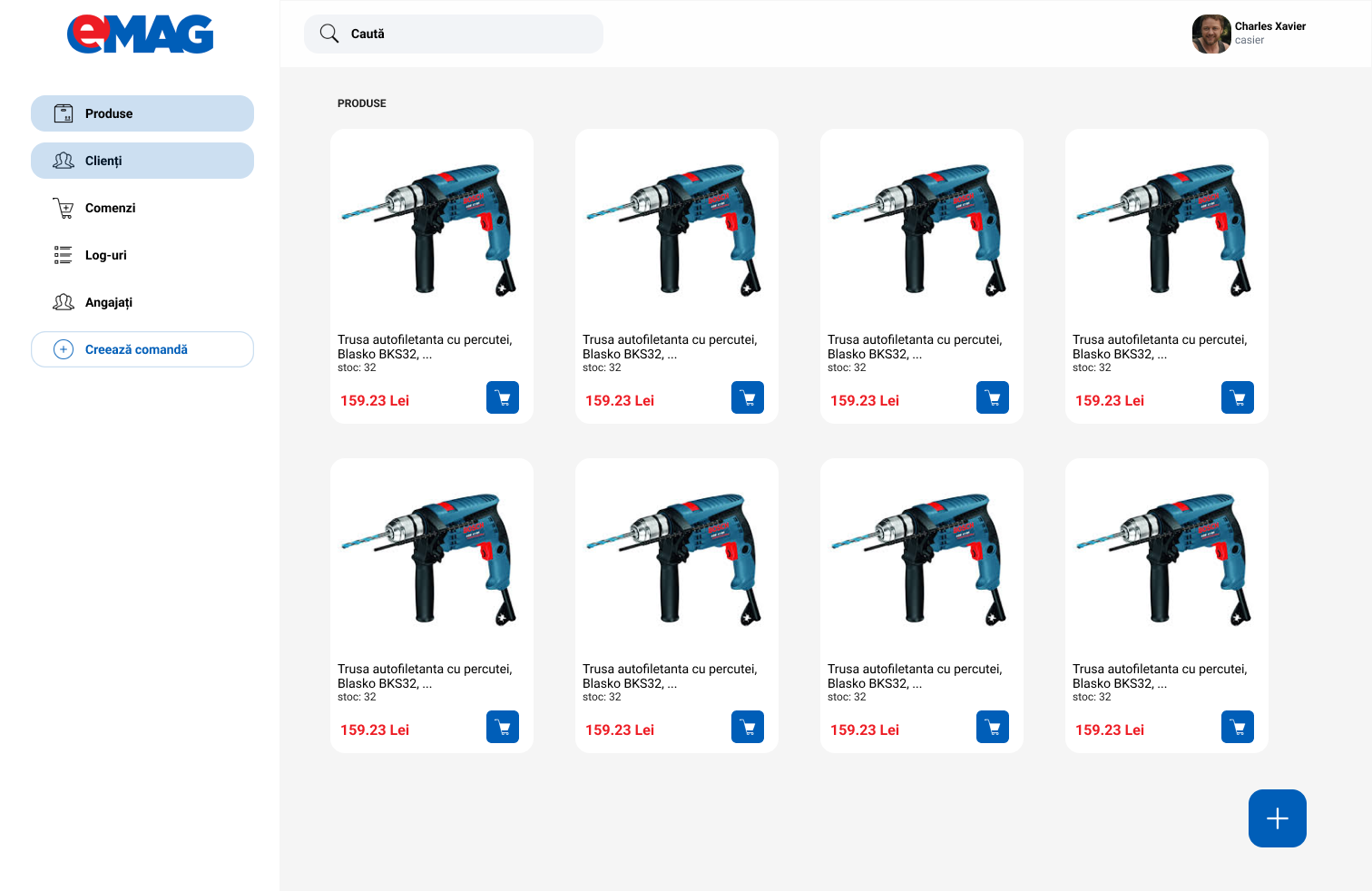
A diagram with text and a black arrow

Description automatically generated with medium confidencePentru 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.

### Evenimente

das

### 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.

### 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).

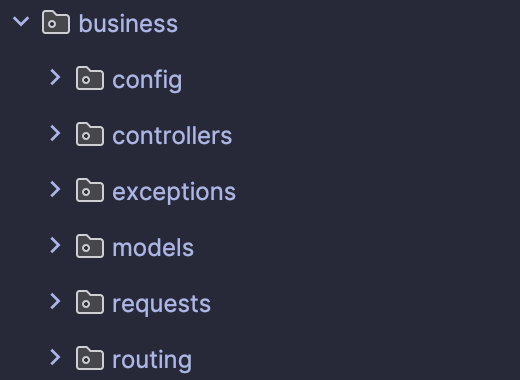
Conform metodologiei JavaFX, view-ul este construit din două părți: fișierul .fxml și o clasă Controller care se ocupă de interacțiunea cu aplicația. În acest controller, sunt definite variabile pentru câmpurile de text pentru P și Q (polinoame), mesajul de eroare, ComboBox și input-ul utilizatorului. 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).

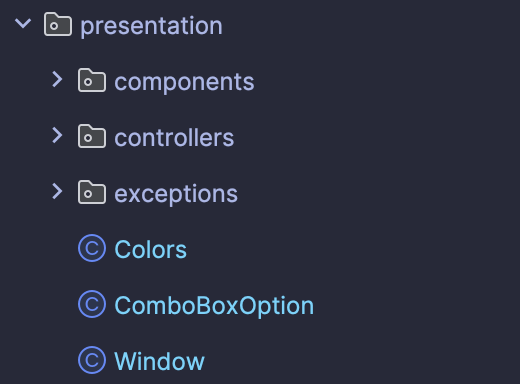
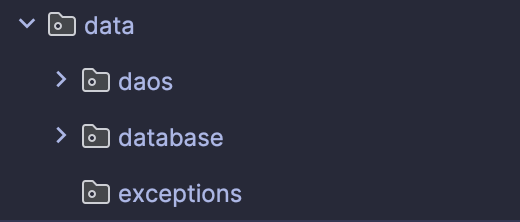
A screenshot of a computer

Description automatically generatedA white text box with black text

Description automatically generatedAceastă 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.

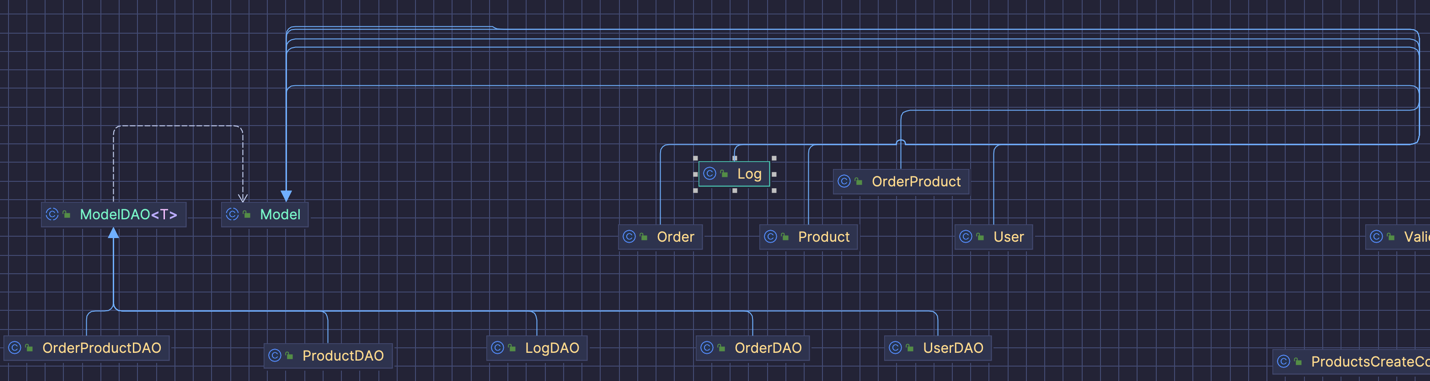
****

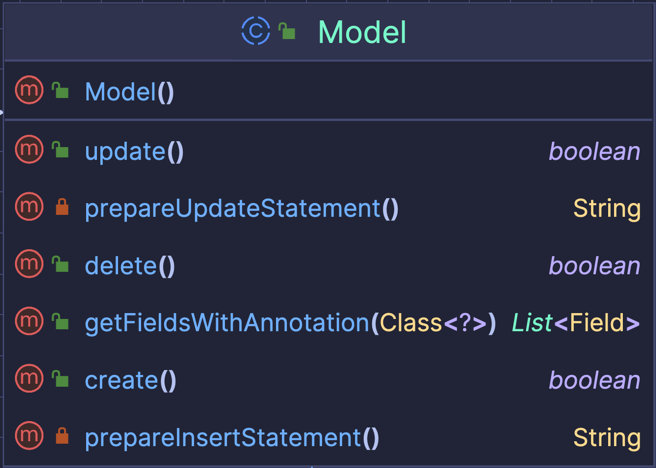
**Implementare**

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:*

### 3 Layer Arhitecture

#### DAO, Model





### 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("users.index", UserController.*class*, "index"));  
Router.*add*(*new* Route("users.create", UserController.*class*, "create"));  
Router.*add*(*new* Route("users.store", UserController.*class*, "store"));  
Router.*add*(*new* Route("users.edit", UserController.*class*, "edit"));  
Router.*add*(*new* Route("users.update", UserController.*class*, "update"));  
Router.*add*(*new* Route("users.delete", UserController.*class*, "delete"));  
  
Router.*add*(*new* Route("products.index", ProductController.*class*, "index"));  
Router.*add*(*new* Route("products.create", ProductController.*class*, "create"));  
Router.*add*(*new* Route("products.store", ProductController.*class*, "store"));  
Router.*add*(*new* Route("products.edit", ProductController.*class*, "edit"));  
Router.*add*(*new* Route("products.update", ProductController.*class*, "update"));  
Router.*add*(*new* Route("products.delete", ProductController.*class*, "delete"));  
  
Router.*add*(*new* Route("orders.index", OrderController.*class*, "index"));  
Router.*add*(*new* Route("orders.create", OrderController.*class*, "create"));  
Router.*add*(*new* Route("orders.store", OrderController.*class*, "store"));  
  
Router.*add*(*new* Route("logs.index", LogController.*class*, "index"));

### 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 doarclasa 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

# Concluzii

În urma dezvoltării proiectului emag.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