

**FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE**

**DEPARTAMENTUL CALCULATOARE**

***Tehnici de programare fundamentale***

***Tema 2 – Simulator de cozi***

Matioc Bogdan Darius, 30224

Profesor îndrumător: David Sera

Cuprins

[1. Obiectivul temei 3](#_Toc68433518)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 3](#_Toc68433519)

[2.1 Analiza problemei 3](#_Toc68433520)

[2.2 Modelare 3](#_Toc68433521)

[2.3 Scenarii și cazuri de utilizare 3](#_Toc68433522)

[3. Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator) 4](#_Toc68433523)

[4. Implementare 4](#_Toc68433524)

[5. Rezultate 6](#_Toc68433525)

[6. Concluzii 10](#_Toc68433526)

# Obiectivul temei

Obiectivul acestei teme este ade a realiza un simulator de cozi. Mai mult decât atât aplicația trebuie să arate în timp real modul în care evolueaza aceasta simulare, mai precis trebuie să poată să fie observate toate modificările aduse cozilor în fiecare secundă.

Obiectivele secundare care fac posibilă realizarea unei astfel de aplicații sunt:

* Definirea claselor corespunzătoare
* Implementarea unei strategii de a poziționa task-urile în coadă
* Realizarea interfeței grafice
* Preluarea datelor necesare din interfața grafică
* Testarea și verificarea corectitudinii implementării realizate

# Analiza problemei, modelare, scenarii, cazuri de utilizare

## Analiza problemei

În primul rând, o să încep prin a prezenta mai precis modul în care funcționează și ce presupune mai exact acest simulator de cozi. Pentru a face o analogie cu lumea reală, viața de zi cu zi, putem să asociem aceste cozi cu cozile de la magazine. Asemenea acelor cozi, în cazul aplicației noastre casa de marcat o să se regăsească sub numele de *server,* iar clienții o să se regăsească sub numele de *clients.* Clientii o să aibă un timp la care aceștia o să fie asignați unei cozi (arrival time) și un alt timp de procesare (processing time). Ca o asociere cu viața reală, arrival time poate să reprezinte timpul în care un client își realizează cumpărăturile, iar apoi se așază la o coadă. În ceea ce privește processing time, acest timp poate să fie asociat cu durata de scanare a cumpărăturilor odată ce clientul a ajuns la casă. Acești timpi menționați anterior, arrival time și processing time, o să fie generați în mod aleatoriu pentru fiecare client, prin specificarea unui minim și a unui maxim pentru fiecare dintre cei doi timpi.

## Modelare

Din punct de vedere al modelării, se pot distinge două părți principale. Prima parte este reprezentată de interfața grafică. Utilizatorul interacționează în mod direct cu aceasta. Interfața este ușor de folosit și sugestivă, din punct de vedere al funcționalității pe care o are. Așadar, utilizatorul are posibilitatea de a introduce orice date dorește privind timpul simulării, numărul de servere, numărul de taskuri și intervalele pe baza cărora se generează timpii aferenți fiecărui task în parte. Mai mult decât atât, utilizatorul poate urmări în timp real evoluția cozilor și a taskurilor care vor fi afișate în fiecare secundă într-o casetă text. Cea de-a doua parte este reprezentată de partea de cod aflată în spatele interfeței, care face posibilă întreaga funcționalitate a programului și înglobează algoritmii necesari pentru toate operațiile.

## Scenarii și cazuri de utilizare

Pornește simularea

Introduce date de intrare

Sterge textul din caseta rezultat

Utilizator

Descriere caz de utilizare:

1. Utilizatorul introduce toate datele necesare pentru a completa câmpurile de: minim și maxim arrival time, minim și maxim processing time, timpul de simulare, numărul de cozi si clienti.
2. Utilizatorul apasă butonul de start
3. Simulatorul afișează evoluția cozilor

Odată ajunși la pasul 4 , utilizatorul are în continuare două posibilități, odată ce afișarea rezultatului de la pasul 3 s-a încheiat, prima dintre ele presupune a apăsa butonul de resetare, astfel revenind la pasul 1 prezentat anterior, a doua posibilitate este aceea de a apăsa din nou butonul start, păstrând datele introduse anterior, astfel revenind la pasul 2 prezentat mai sus. În cazul alegerii celei de-a doua opțiuni rezultatul afișat o să fie diferit față de rezultatul afișat la prima rulare, deoarece taskurile sunt generate in mod aleatoriu.

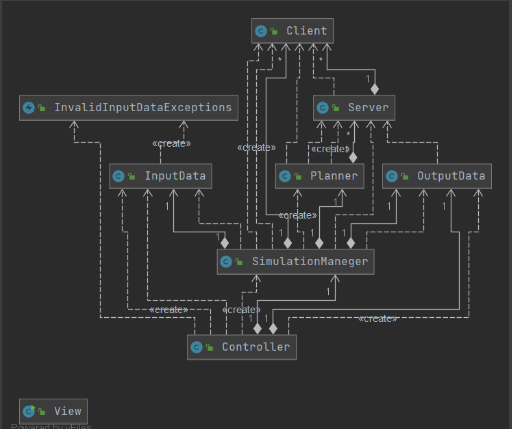
# Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator)

Una din deciziile de proiectare pe care le-am luat vizavi de acest calculator de polinoame contă în structurarea proiectului așa încât să respecte convenția Model-View-Controller. Prin această structurare proiectul este mai lizibil și mai coerent din punct de vedere logic. Așadar, prin clasele care înglobează această aplicație se poate distinge cu ușurință cărei componente, dintre cele trei menționate mai sus, aparțin.

Algoritmii care stau la baza acestui simulator de cozi sunt constituiți de cuprinsul celor două metode de run prin intermediul cărora este realizată dinamicitatea acestei aplicații. De asemenea, un alt algoritm important este acela care realizează sortarea. Alte metode ajutătoare integrate în acest proiect sunt metodele *toString* care au fost adaptate așa încât afișarea rezultatelor în urma operațiilor efectuate să fie cât mai lizibilă, ușor de urmărit, așa încât verificarea rezultatului să se facă fără probleme.

Interfața utilizator este ușor de folosit și intuitivă din punct de vedere al funcționalității. Aceasta permite personalizarea datelor de intrare și urmărirea rezultatului fără probleme.

**Diagrama UML**



# Implementare

Aplicația realizată respectă structura MVC (Model-View-Controller), așadar o să descriu pe rând clasele cuprinse în aceste trei componente:

Partea de modelcuprinde clasele care realizează, care implementează algoritmii cuprinși în proiect. Această parte manipulează operațiunile logice și de utilizare de informație pentru a rezulta o formă ușor de înțeles. Așadar, această parte înglobează următoarele clase:

* **Client (-id: int, -arrivalTime : int, -processingTime : int, -** **timeInQueue : int)**

Această clasă este utilizată pentru a stoca clienti care urmează apoi să fie introdusi în cozi. Variabilele instanță ale acestei clase reprezintă, în această ordine, numărul de identificare al clientului, timpul la care o sa fie asignat unei cozi, timpul de procesare până ce este scos din coadă odată ce a ajuns capul cozii și timpul pe care în așteaptă până ce ajunge să fie capul cozii. Această clasă implementează metode de ***set*** și *get***.** De asemenea, clasa implementează și o metodă *toString*, utila in momentul printarii. Pe de altă parte, această clasă implementează *Comparable* așa încât clientii o să fie sortati crescător în funcție de *arrivalTime.*

* **Server (-id : int, -tasks : BlockingQueue<Task>,-waitingPeriod : AtomicInteger)**

Aceasta este clasa prin intermediul căreia se creează serverele la care o să fie asignate taskurile. Variabilele instanță ale acestei clase reprezintă, în această ordine, coada cu taskurile aferente serverului în cauză, timpul de așteptare la serverul respectiv, numărul de identificare al serverului. Această clasa implementează *Runnable,* iar metoda run este cea mai importantă metodă cuprinsă, deoarece prin prisma ei se realizează dinamicitatea aplicației, această metodă fiind aceea care scoate rând pe rând taskurile afară din servere, în funcție de timpul de procesare curent al taskului. **.** De asemenea, clasa implementează și o metodă *toString*, care a fost modificată, așa încât afișarea taskurilor și a timpilor asociați lor să fie cât mai ușor de urmărit. Clasa conține și o metodă de adăugare a unui task nou.

* **ConcreteStrategyTime**

Această clasă implementează o metodă importantă prin intermediul căreia se stabilește strategia de adăugare a unui nou task unui server. Mai precis, metoda menționată are rolul de a adăuga întotdeauna taskul curent serverului la care timpul de așteptare este cel mai mic. Prin această strategie aplicația este mai eficientă, micșorând timpul de așteptare al taskurilor cât de mult posibil.

* **Planner (-servers : List<Server>, -strategy : Strategy)**

Cu ajutorul acestei clase, mai precis, prin intermediul constructorului clasei sunt create serverele și apoi este pornit câte un thread pentru fiecare dintre aceste servere. Totodată, această clasă are o metodă *get* prin care se returnează lista de servere. Am ales ca aceasta clasa sa detina un field de tipul clasei Strategy chiar daca am implementat o singura strategie. Am facut aceasta alegere deoarece am dorit ca aplicatie sa permita update fara a fi necesare modificari mari.

* **SimulationManager (-inputData : Inputdata, -outputData : OutputData, -planner : Planner, -isRun : boolean)**

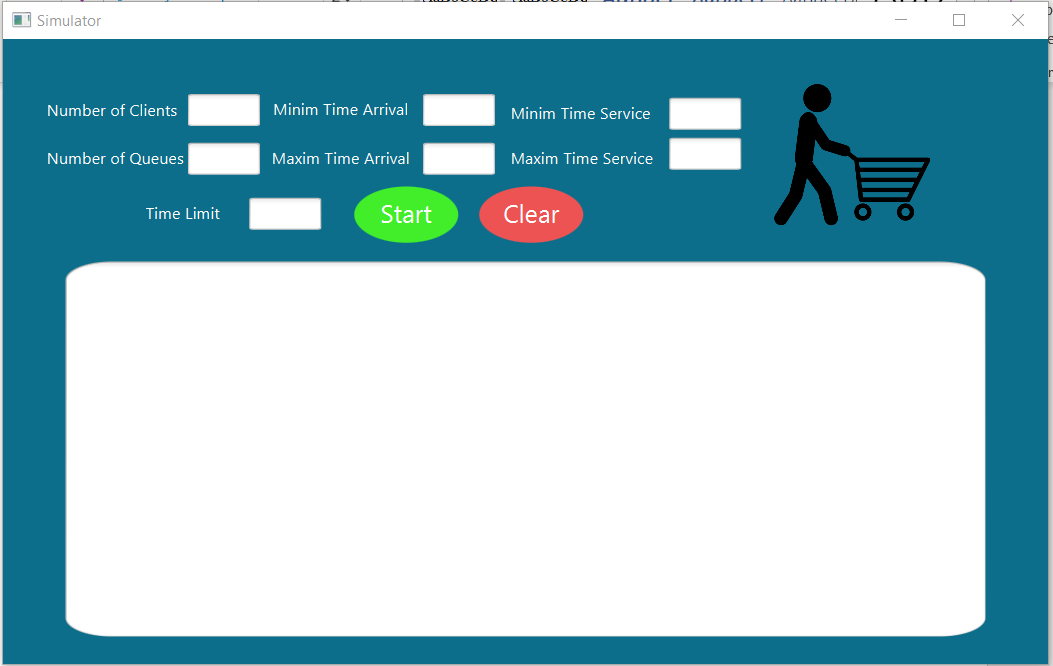
Această clasă reprezintă clasa principală care pune în mișcare lucrurile și aduce laolaltă majoritatea funcționalităților celorlalte clase. Variabilele instanță ale acestei clase cuprinde variabile care o să aibă valorile pe ca utilizatorul decide să le introducă în interfața grafică grupate intr-o singura clasa numita OutputData. Una dintre metodele acestei clase este *generateNRandomClients,* metodă cu ajutorul căreia se generează în mod aleatoriu n clients, care apoi sunt adăugate într-o listă. După ce toate taskurile se află în listă, ele o să fie sortate crescător în funcție de *arrivalTime*. Aceeași metodă calculează și *averageProcessingTime*. Dat fiind faptul că această clasă implementează *Runnable,* metoda *run* joacă un rol important în cadrul aplicației, deoarece prin intermediul ei se leagă funcționalitățile claselor prezentate anterior. În această metodă sunt apelate funcțiile care realizează adăugarea taskurilor unor servere. Tot aici, se calculează și *peakHour*, care reprezintă secunda în care au existat cele mai multe servere în cozi, și *averageWaitingTime*. De asemenea, se construiește la fiecare pas în variabila instanță *outputData* evoluția cozilor, care ulterior este afișată în interfața grafică.

* **InputData (-timeLimit : int, -maxTimeService : int, -minTimeservice : int, -maxTimeArrival : int, -minTimeArrival :int, -numberOfServers : int, -numberOfClients : int)**

Această clasă grupeaza toate datele pe care utilizatorul le va introduce;

* **OutputData (-report : StringBuilder)**

Această clasă contine datele de iesire actualizate constant (in fiecare secunda)



Pachetul Viewcuprinde codul care realizează interfața grafică. Prin intermediul clasei View, al fisierului View.fxml, din acest pachet i-a naștere partea vizuală a proiectului, aceea care interacționează cu utilizatorul.

Fisierul .fxml ccuprinde elementele care se regăsesc în interfața grafică cum ar fi: butoane, casete text, etichete. Clasa View extinde interfata Application, implemantand metoda start, metoda absolut necesara pentru pornirea aplicatie. Infruusetarea interfetei a fost arealizata cu ajutorul unei bucati de cod in limbajul CSS. Aceasta bucata de cod se regaseste in fisierul style.css in directorul resurse.

După cum se poate observa interfața are, în primul rând, un titlu sugestiv care anunță întru-totul așteptările pe care utilizatorul trebuie să le aibă vizavi de această aplicație. Urmează apoi o serie de casete text, însoțite de niște etichete sugestive, care ajută utilizatorul în introducerea datelor dorite. După ce toate câmpurile au fost completate corespunzător, utilizatorul are posibilitatea de a apăsa pe butonul start. Apăsând acest buton în caseta text aflată în partea de jos a ferestrei o să apară în timp real rezultatul simulării. Odată ce simularea a luat sfârșit utilizatorul poate să apese butonul de clear pentru a goli toate câmpurile scrise.

Partea de *controller* cuprinde clasele prin intermediul cărora se realizează de fapt legătura dintre partea de model și de vizualizare. Prin intermediul acestor clase putem controla accesul la aplicația noastră.

Clasa **Controller** implementează metode *get* cu ajutorul cărora datele introduse de utilizator sunt preluate și convertite corespunzător cu tipul datelor utilizate în program.

# Rezultate

Pentru a verifica funcționalitatea aplicației realizate, am testat și am urmărit corectitudinea rezultatului raportat la exemplele primite în specificația proiectului. Rezultatele obținute în urma testării sunt:

* Pentru N = 4 Q = 2 𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛 𝑀𝐴𝑋 = 60 seconds [𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 𝑀𝐼𝑁 , 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 𝑀𝐴𝑋 ] = [2, 30] [𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 𝑀𝐼𝑁 , 𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 𝑀𝐴𝑋 ] = [2, 4] : <https://github.com/MatiocBogdanDarius/RezulateTesteTema2TP/blob/main/rezultat1.txt>
* Pentru N = 50 Q = 5 𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛 𝑀𝐴𝑋 = 60 seconds [𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 𝑀𝐼𝑁 , 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 𝑀𝐴𝑋 ] = [2, 40] [𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 𝑀𝐼𝑁 , 𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 𝑀𝐴𝑋 ] = [1, 7] : <https://github.com/MatiocBogdanDarius/RezulateTesteTema2TP/blob/main/rezultat2.txt>
* Pentru N = 1000 Q = 20 𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛 𝑀𝐴𝑋 = 200 seconds [𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 𝑀𝐼𝑁 , 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 𝑀𝐴𝑋 ] = [10, 100] [𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 𝑀𝐼𝑁 , 𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 𝑀𝐴𝑋 ] = [3, 9] : <https://github.com/MatiocBogdanDarius/RezulateTesteTema2TP/blob/main/rezultat3.txt>

Observatie: Deoarece rezultatele sunt fisiere de dimensiuni considerabile, am ales sa creez un repozitoriu GitHub public, sa incar fisierele acolo iar aici sa introduc doar link-uri spre aceste fisiere.

# Concluzii

Prin prisma acestei teme mi-am consolidat cunoștințele de programare orientată pe obiect. De asemenea, am reușit să înțeleg mai bine modul în care trebuie să îmi structurez codul și logica pe care trebuie să o aplic pentru a respecta formatul Model-View-Controller. Totodată, am învățat lucruri noi despre threaduri și modul în care acestea funcționează. Tot prin intermediul acestui proiect am reusit sa imi aprofundezi cunostintele despre Javafx (in acest proiect am creat prima mea animatie cu ajutorul Javafx) si am folosit pentru prima data limbajul CSS pentru a adauga un stil pentru o aplicatie desktop.

Din punct de vedere al dezvoltării ulterioare a acestui simulator de cozi, aplicația ar putea fi îmbunătățită și prin adăugarea unor noi strategii de plasare a unui task într-o coadă, cum ar fi adăugarea unui task la coada cea mai scurtă.

# Bibliografie

1. FUNDAMENTAL PROGRAMMING TECHNIQUES-ASSIGNMENT 2 – SUPPORT PRESENTATION
2. <http://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-andthreadpoolexecutor.html>
3. <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html>
4. <https://www.youtube.com/watch?v=XwZ7Lgy8qBo&t=1s&ab_channel=FacemSoft>
5. https://www.youtube.com/watch?v=bccrpscSLjs&ab\_channel=FacemSoft