**DOCUMENTATIE TEMA 2**

**QUEUE SIMULATOR**

**Nume prenume: Pașca Maria**

**Grupa 30229**

**Profesor Laborator Assist Antal Marcel**

Contents

[1. Cerinte Functionale 3](#_Toc476131445)

[2. Obiective 3](#_Toc476131446)

[2.1. Obiectiv Principal: 3](#_Toc476131447)

[2.2. Obective Secundare: 3](#_Toc476131448)

[3. Analiza Problemei 3](#_Toc476131449)

[4. Proiectare 3](#_Toc476131450)

[4.1. Structuri de date 3](#_Toc476131451)

[4.2. Diagrama de clase 3](#_Toc476131452)

[4.3. Algoritmi 3](#_Toc476131453)

[5. Implementare 4](#_Toc476131454)

[6. Testare 4](#_Toc476131455)

[7. Concluzii si Dezvoltari Ulterioare 4](#_Toc476131456)

[8. Bibliografie 4](#_Toc476131457)

# Cerinte Functionale

Cerința a fost dezvoltarea unui program care simulează evoluția unor cozi într-un magazin și returnarea rezultatului într-un fișier .txt și într-un frame în timp real. Utilizatorul are la dispoziție o interfață grafică prin care poate comunica cu programul.

Utilizatorul trebuie să introducă parametrii simulării în câmpurile precizate în interfața grafică, anume numărul de clienți, numărul de cozi, durata simulării (în secunde), intervalul în care clienții pot ajunge la coadă și intervalul pentru durata de servire a clienților. În cazul în care sunt lăsate câmpuri necompletate va apărea o eroare în consolă în care este specificat faptul că nu pot fi lăsate câmpuri goale.

# Obiective

## Obiectiv Principal:

Obiectivul principal este proiectarea și implementarea unui program care să simuleze evoluția unor cozi spre exemplu într-un magazin, astfel încât timpul de așteptare să fie cel mai mic. Programul include, de asemenea, o interfață grafică prin care utilizatorul poate introduce parametrii simulării, precum numărul de clienți, numărul de cozi, durata simulării, intervalul în care clienții pot ajunge la coadă și intervalul pentru durata timpului de servire a clienților. La finalul execuției programului, informațiile aferente simulării vor fi scrise într-un fișier de tip .txt, unde, după prezentarea statusului cozilor și clienților în fiecare secundă, este afișată o statistică cu timpul mediu de așteptare și timpul mediu de servire. Aceste informații sunt afișate și într-un frame care arată evoluția în timp real a simulării.

## Obective Secundare:

|  |  |  |
| --- | --- | --- |
| **Obiectiv Secundar** | **Descriere** | **Capitol** |
| Dezvoltarea de use case-uri si scenarii | Descrierea modului de funcționare a programului în diferite situații | 3 |
| Alegerea structurilor de date | Prezentarea pe scurt a structurilor de date folosite | 4 |
| Diagrama de clase | Diagrama UML a proiectului | 4 |
| Implementarea solutiei | Descrierea programului pe larg, descrierea claselor, metodelor și relației dintre clase | 5 |
| Testare |  | 6 |

# Analiza Problemei

În cadrul acestui program, există un singur use-case în care utilizatorul introduce numărul de cozi, numărul de clienți, timpul de simulare, timpul minim de sosire, timpul maxim de sosire, timpul minim de servire și timpul maxim de servire și urmărește rezultatul simulării pe ecran și îl primește și la final în documentul txt generat de program.

Use Case Name: Simularea cozilor

Actori:

* Utilizatorul
* View-ul
* Controller-ul
* Modelul

Trigger:

* Utilizatorul a apăsat butonul de start simulation

Precondition:

* Utilizatorul a introdus de la tastatură parametrii simulării

Normal Flow:

1. Utilizatorul introduce corect valorile în textFieldurile aferente.
2. Utilizatorul apasă butonul de Start Simulation.
3. Listenerul butonului de Start simulation din controller va apela metoda startSimulation din Controller.
4. Metoda startSimulation va crea o instanță de store, prin care se va crea threadul principal.
5. În store sunt generați clienții ipotetici și este creat scheduler-ul, prin care se creează threadurile pentru cozi, și va crea o instanță nouă de ViewLogs, frame în care vor veni afișate rezultatele simulării.
6. Pe parcursul simulării, este apelată metoda printDetails din clasa store pentru a da mai departe string-ul instanței de ViewLogs pentru afișarea pe ecran.
7. Utilizatorul va vedea rezultatul calculului pe ecran în timp real.
8. Utilizatorul va închide programul.

Alternate Flow:

1A. Utilizatorul lasă câmpuri goale.

1. Va apărea o eroare în consolă care informează utilizatorul că nu pot rămâne câmpuri goale.

1B. Utilizatorul introduce alte caractere în afară de cele numerice.

1. Va apărea o eroare în consolă care informează utilizatorul că nu pot fi introduse alte caractere în afară de cele numerice.

1C. Utilizatorul introduce o valoare mai mare în câmpul pentru limita minimă a unui interval decât valoarea aferentă limitei maxime.

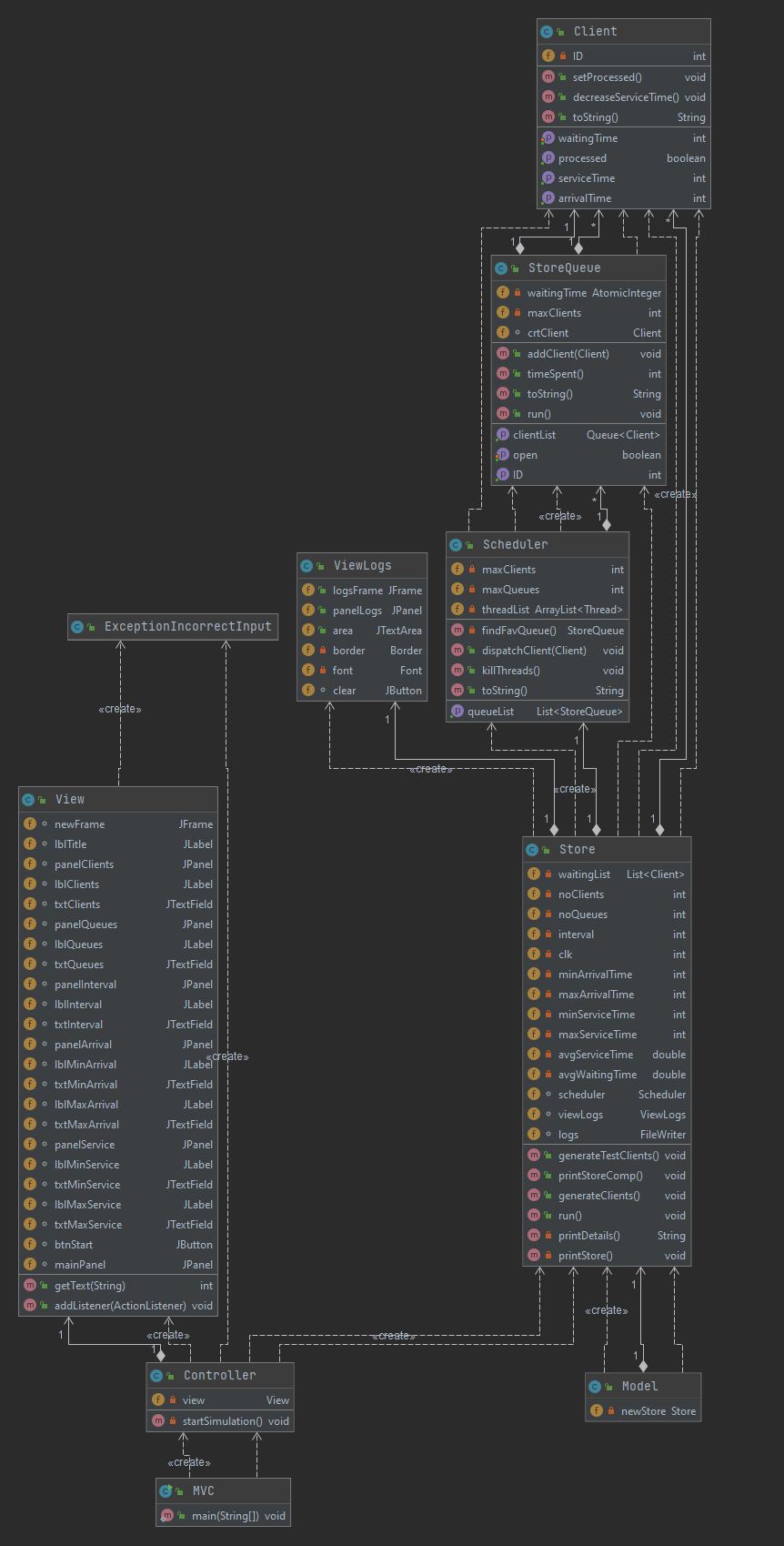
1. Va apărea o eroare în consolă care informează utilizatorul că limita minimă a intervalului e mai mare decât limita maximă a aceluiași interval.

# Proiectare

## Structuri de date

Structura de date cea mai răspândită în proiect este Singly-LinkedList-ul, care sunt folosite pentru a ține listele de clienți, de threaduri și de cozi în aproape toate clasele din proiect. De asemenea este folosit și AtomicInteger-ul, în cadrul StoreQueue, pentru a avea waitingTime-ul thread safe.

## Diagrama de clase



# Implementare

Pentru acest proiect am ales să merg pe patternul arhitectural MVC (Model, View, Controller). Astfel, clasele pot fi împărțite în 3 categorii, cele care țin de View, anume View și ViewLogs, cele care țin de Controller, cu clasa eponimă și Model, anume Client, Store, StoreQueue, și Scheduler. De asemenea, mai este și clasa ExceptionIncorrectInput, care extinde Exception.

În clasa View este contruită interfața grafică în care utilizatorul are posibilitatea de a introduce în câmpurile aferente, parametrii simulării. Aceasta este formată din 6 JPaneluri, panelClients, în care sunt nested labelul și textFieldul pentru numărul de clienți, panelQueues, în care sunt ținute labelul și textFieldul pentru numărul de cozi, panelInterval, care conține labelul și textFieldul pentru durata simulării, panelArrival, care ține labelurile și textFieldurile aferente intervalului în care pot ajunge clienții la coadă, panelService, în care sunt ținute labelurile și textFieldurile pentru intervalul duratei procesării comenzii unui client, și mainPanel, în care sunt ținute toate aceste paneluri, pe lângă labelul în care este ținut titlul programului și butonul care activează simularea.

Clasa View mai conține metoda getText, care ia ca parametru un string a cărui valoare desemnează numele unui câmp. Acesta conține un string Text, care ia valoarea din textFieldul precizat de stringul introdus ca parametru. Pe urmă, în metodă se mai verifică dacă valoarea acelui câmp este validă sau nu, adică să nu fie câmp gol și să nu conțină alte caractere în afară de cele numerice.

Clasa Controller face legătura între interfața grafică și logica programului astfel: Listener-ul butonului de Start Simulation este prezent în această clasă și tot în această clasă îi sunt trimise obiectului model parametrii simulării citiți din textBoxurile prezente în interfață și îi este zis ce să facă cu ele. Aceasta conține și metoda startSimulation, în cadrul căreia e apelată metoda getText din View pentru a popula câmpurile de valori pentru parametrii simulării care urmează să fie trimiși constructorului Store-ului. Dacă parametrii sunt corecți, se va începe simularea.

În clasa Store se regăsesc parametrii simulării, anume numărul de clienți, numărul de cozi, durata simulării, timpul minim de sosire, timpul maxim de sosire, durata minimă de servire și durata maximă de servire, aici generându-se aleatoriu și clienții prin metoda generateClients. Aceasta se folosește de valorile pentru timpul minim de sosire și timpul maxim de sosire pentru generarea aleatorie a timpului de sosire a clientului curent și de valorile pentru durata minimă de servire și durata maximă de servire pentru generarea aleatorie a duratei de servire a clientului curent, care, mai apoi, este adăugat în waiting list-ul store-ului, care desemnează clienții care încă nu au ajuns la nicio coadă. De asemenea, aici se află și threadul principal, din care vor fi distribuiți clienții celorlalte cozi, aspect care se va face pe seama contorului de timp clk, care se incrementează până ajunge la valoarea duratei simulării. Astfel, la fiecare semnal de tact se va verifică dacă sunt clienți care urmează să ajungă la coadă, în cazul favorabil trimițând comanda scheduler-ului să îl atribuie cozii celei mai favorabile. Tot aici se face și scrierea în fișierul txt și în interfața grafică, prin metodele printStore, care scrie în fișier, și printDetails, care construiește string-ul care pe urmă este trimis ca parametru metodei append specifice TextPane-ului din ViewLogs. Clasa View este folosită pentru realizarea interfaței GUI cu Java Swing. În realizarea ei s-au folosit un frame, 5 paneluri, 6 butoane, 2 text field-uri pentru a scrie polinoamele care urmează să fie folosite în operații și 4 label-uri, dintre care unul este pentru a afișa rezultatul operației. Metodele prezente în această clasă sunt getInput1 și getInput2, care returnează conținutul text field-urilor, adică polinoamele introduse de utilizator, setTxtOutput, prin care labelul pentru output primește rezultatul operației, și metodele prin care butoanelor le sunt atribuite listener-urile.

Clasa Client conține câmpurile ID, waiting time, arrival time, service time și booleanul isProcessed, necesare identificării și rulării simulării de cozi. Metodele specifice acestei clase sunt getterele și setterele și, de asemenea, suprascrierea metodei toString pentru a returna un string de forma (id, arrival time, service time). Booleanul isProcessed este folosit pentru a verifica dacă clientul curent a fost procesat sau nu, adică dacă a fost adăugat unei cozi sau încă este în așteptare.

Clasa Scheduler conține două liste, anume queueList și threadList, care țin cozile generate și threadurile aferente cozilor, pe lângă câmpurile care țin numărul maxim de clienți și numărul maxim de cozi. În cadrul constructorului, sunt create cozile și threadurile pentru fiecare coadă în parte. Metodele aferente clasei Scheduler sunt findFavQueue, prin care se caută coada cea mai favorabilă pentru adăugarea unui nou client. Pentru ca o coadă să fie favorabilă, timpul petrecut în așteptarea la acea coadă trebuie să fie minim, iar, în caz că timpul petrecut în așteptare este minim pentru mai multe cozi, întâietate are cea cu mai puțini clienți în așteptare. O altă metodă specifică acestei clase este dispatchClient, în cadrul căreia este apelată metoda findFavQueue, clientul fiind atribuit acelei cozi favorabile, totodată fiindu-i atribuit și waiting time-ul și valoarea true booleanului de isProcessed. O altă metodă, de această dată pentru lucrul cu threaduri, este metoda killThreads, care închide fiecare thread deschis pentru cozi. În plus, are și suprascrierea metodei toString, care creează stringuri pentru fiecare queue după formatul ”Queue id: queue list”.

Clasa StoreQueue desemnează coada magazinului. Aceasta are ca și câmpuri o listă pentru clienți – clientList, unui AtomicInteger – waitingTime, folosit pentru a ține timpul de așteptare pentru threadul în cauză, maxClients, pentru numărul maxim de clienți, crtClient, clientul curent care se află în fața cozii și ID, care ține id-ul cozii curente. Aceasta conține metoda addClient, prin care un client este adăugat cozii și este incrementat waiting time-ul cu valoarea duratei de servire a clientului în cauză. De asemenea, conține și suprascrierea metodei toString, prin care se construiește string-ul specific acestei cozi, care poate fi de forma ”closed”, dacă coada este goală, sau ”client1; client2; …” dacă coada conține clienți. Metoda principală din această clasă este suprascrierea metodei run(), prin care se procesează, cât timp este deschisă, coada. Astfel, la fiecare secundă se procesează conținutul cozii, adică clientul curent, cel care se afla primul la coadă este scos din listă și atribuit câmpului crtClient, prin decrementarea duratei de servire. În momentul în care durata ajunge la 0, crtClient este suprascris cu următorul client din listă. De asemenea, această metodă va scrie în consolă pentru coada curentă, care este clientul procesat la fiecare pas.

Clasa ExceptionIncorrectInput extinde Exception, astfel că în cadrul constructorului, care primește ca parametrii wrongInput, câmpul din interfața grafică la care a existat problema, why, motivul pentru care a apărut eroarea, și frame, frame-ul în care va veni afișat mesajul, pe lângă mesajul din consolă, și se apelează super() cu mesajul care trebuie afișat. Această clasă se folosește în cadrul claselor controller și view, dacă valorile din text field-uri nu sunt conforme standardelor, adică conțin și alte caractere pe lângă cele de la 0 la 9, limita minimă a unui interval este mai mare decât limita maximă sau chiar sunt câmpuri goale rămase după ce s-a apăsat butonul de start Simulation. Tot aici se apelează și metoda showMessageDialog a JOptionPane, pentru afișarea mesajului de eroare și pe ecran pentru utilizator.

Clasa ViewLogs este folosită pentru a afișa în timp real stadiul simulării cozilor. Aceasta nu este instanțiată din Controller, deoarece scrierea în textArea se face în cadrul metodei suprascrise run() din clasa Store.

Clasa MVC este cea din care programul este rulat.

Toate clasele, inclusiv cele ce țin de UI, au până în 300 de linii, iar metodele până în 30 de linii, mai puțin metoda run() din Store, care conține 46 de linii, însă necesare pentru buna funcționare a simulării.

# Testare

Testarea s-a făcut pe cele 3 teste de la finalul pdf-ului, rezultatele putând fi vizualitate în fișierele text din repository ”logs\_1.txt”, ”logs\_2.txt” și ”logs\_3.txt”.

# Concluzii si Dezvoltari Ulterioare

În principiu, proiectul reușește să realizeze toate funcționalitățile dorite, anume de simulare a cozilor în funcție de niște parametrii trimiși de utilizator, de scriere în consolă a rezultatului și de generare a unui fișier .txt care conține aceste informații a simulării și de furnizare a statisticilor pentru timpul mediu de așteptare și timpul mediu de servire. Dezvoltările ulterioare pe care le văd să fie făcute sunt, în primul rând la interfața grafică, în special pentru viewLogs, și furnizarea informațiilor despre ora de vârf, în care cozile sunt cele mai aglomerate.

Prin acest proiect am reușit să am o înțelegere mai bună asupra threadurilor și utilizării lor, totodată îmbogățindu-mi cunoștințele care țin de realizarea unei interfețe grafice și de folosire a Java Swing.

# Bibliografie

<https://www.geeksforgeeks.org/multithreading-in-java/>

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html>

<https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html>