***Aplicație de gestionare a cozilor***

***Minimizarea timpului de așteptare la coade***

-Assigment 2-

Student : Brehuescu Andrei

Grupa: 30224

Faculatatea CTI RO,an II

Contents

[1.Obiectivul temei 3](#_Toc36557168)

[1.1 Obiectivul principal 3](#_Toc36557169)

[1.2 Obiective secundare 3](#_Toc36557170)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 4](#_Toc36557171)

[2.1 Cerințe funcționale 4](#_Toc36557172)

[2.2 Use-case 4](#_Toc36557173)

[3. Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfețe, relații, packages, algoritmi, interfața utilizator) 6](#_Toc36557174)

[3.1 Decizii de proiectare 6](#_Toc36557175)

[3.2 Diagrama UML 6](#_Toc36557176)

[3.3 Proiectarea claselor, pachete, interfețe 7](#_Toc36557177)

[4. Implementare 7](#_Toc36557178)

[5. Rezultate 9](#_Toc36557179)

[6. Concluzii 10](#_Toc36557180)

[7. Bibliografie 10](#_Toc36557181)

# 1.Obiectivul temei

## Obiectivul principal

Principalul scop al acestei teme este proiectarea și implementarea unui sistem de gestionarea al clienților, astfel încât să se minimizeze timpul de așteptare la coadă pentru fiecare client. În funcție de cozile deschise când un client intră în coadă se calculează timpul total de așteptare pentru coadă și i se atribuie o coadă cu un timp minim.

Gestionarea clienților se face folosind o strategie de alegerea cozii cu timpul minimal de procesare a tuturor clienților aflați deja în coadă, iar clientul este repartizat la acea coadă. Cozile au un comportament de FIFO ( first in first out ).

## Obiective secundare

Pentru proiectare , problema este împărțită în mai multe obiective secundare, urmând ca acestea sa fie implementate separate, acestea fiind:

1. Stabilirea atributelor claselor și comportamentul obiectelor pentru a semăna cât mai mult cu o transpunere din lumea reală ( variabile, metode, interfețe, strategii de selecție al cozilor);
2. Transpunerea clienților și a cozilor din lumea reală sub forma unor clase ( descrise în capitolele de Proiectare și Implementare);
3. Crearea de obiecte ce transpun clasele și clienții;
4. Implementarea metodelor necesare;
5. Citirea datelor dintr-un fișier de intrare ( număr de client, timp minim respectiv maxim, timp limită de simulare, simularea se poate opri mai devereme daca nu mai sunt clienți de procesat sau la atingerea timpului limită, etc) , cât timp simularea este rulată într-un timp mai mic de timpul limită, clienții pot ajunge la coadă pentru a putea fi procesați mai departe ,datorită acestei citiri din fișier nu este necesară o interfața grafică;
6. Crearea și scrierea unui fișier de ieșire care va conține rezultatele împărțite pe timpi de așteptare cu detalii la fiecare timp despre client și statusul lor ( poziția în coadă, timpul de procesare rămas, clienții în așteptare);
7. Utilizarea de interfețe Comparable ( pentru ordonarea clienților ) și Runnable pentru utilizarea mai multor fișiere de execuție;
8. Definirea interfețelor pentru diferite strategii de alegere a clienților ( În funcție de timpul minim de procesare, în funcție de lungimea minimă a cozii) , pentru această temă a fost implementată doar strategia de selecție a cozii cu timpul minim de procesare pentru distribuirea clienților.

# 2. Analiza problemei, modelare, scenarii, cazuri de utilizare

## 2.1 Cerințe funcționale

Cerințe:

* Distribuirea corectă a clienților la coadă în funcție de timpul minim total de procesare al clienților deja aflați la coadă;
* Simularea în timp real a evoluției cozii și generearea rezultatelor într-un fișier de ieșire;
* Utilizare Multithreading ;
* Dinamica cozilor ( rularea threadurilor doar când acestea au client de procesat, în caz contrar acestea trebuie oprite/ închise);
* Respectarea FIFO;
* Scăderea timpului de procesare a clientului la începutul cozii în cadrul fiecarei procesări;
* Când timpul de procesare ajunge la 0, clientul este considerat procesat și eliminat din coadă;
* Calcularea timpului mediu petrecut de client la coadă;

Toate aceste cerințe trebuie implementate pentru o aplicație cu o funcționalitate minimală, putând fi ulterior dezvolatată și îmbunătățită adăugând funcționalități noi. Prin funcționarea corectă a aplicației se înțelege că în procesarea clienților și distribuția acestora se va ține cont de câteva aspecte:

* Distribuirea se va facea la coada cu ordinul cel mai mic dar care are și cel mai mic timp de procesare;
* Pentru fiecare client trebuie să se aleagă coada cu cel mai mic timp de procesare astfel încat acesta va fi procesată în cel mai scurt timp;
* Thread-urile trebuie sincrozate;
* Interpretarea corectă a datelor de intrare și generarea de date corecte la ieșire;
* Interpretarea argumentelor date din linia de comandă la rularea fișierului .jar generat;

## 2.2 Use-case

Date de intrare:

Ca date de intrare se vor furniza într-un fișier de intrare:

* Număr de client
* Număr de cozi
* Timpul maxim de simulare
* Timpul minim de sosire , Timpul maxim de sosire
* Timpul minim de așteptare , Timpul maxim de așteptare

După furnizarea acestor date de intrare , aplicația va furniza un fișier de ieșire în care este prezentată situația cozilor, respectiv a clienților ce așteaptă sau sunt deja la coadă și timpul actual în simulare. Fiecare bloc simulat va avea un format de forma:

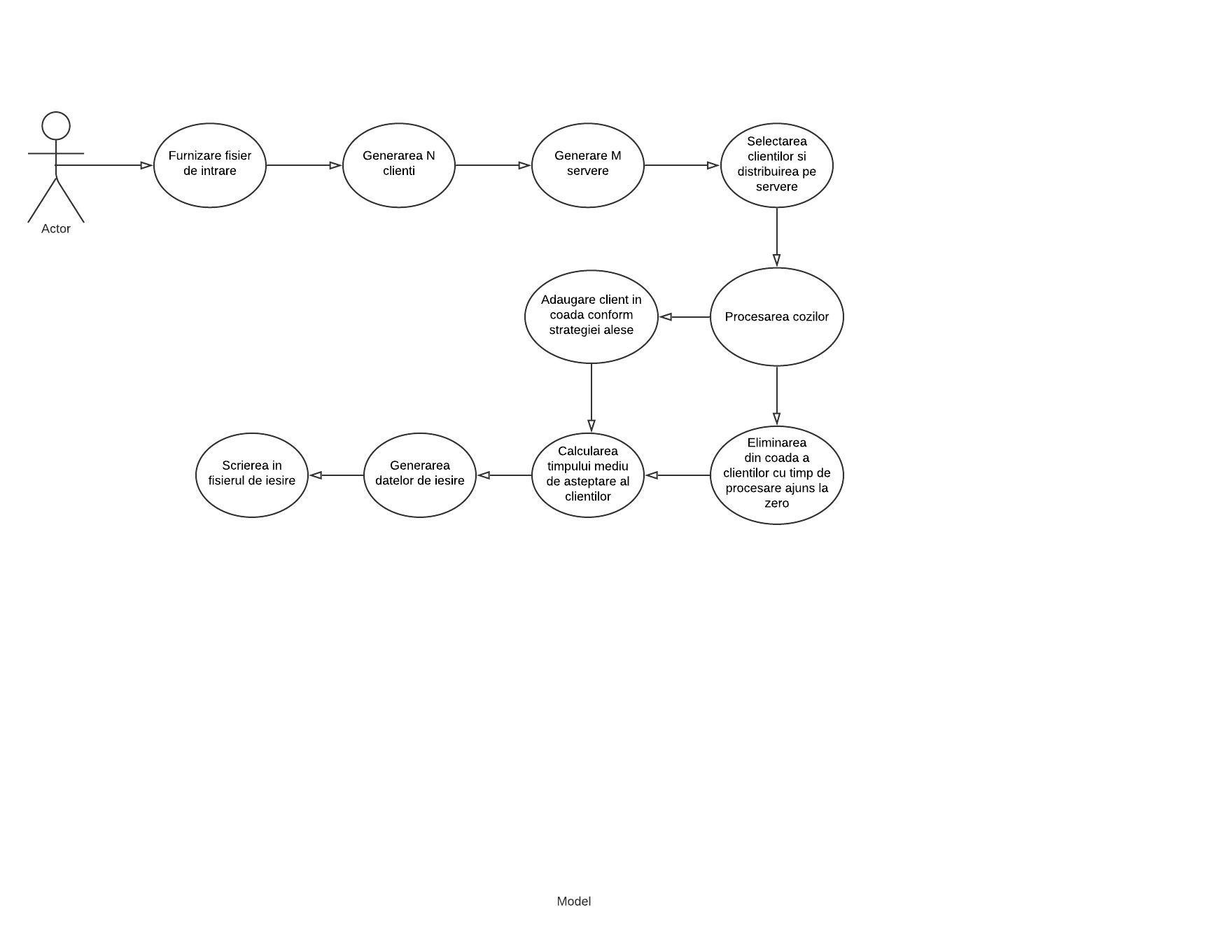
Date de ieșire:

* Timp <timp\_actual>
* Waiting clients: <Lista clienților care nu sunt încă la o coadă>
* Queue 1: <Lista clienților din coadă>
* …..
* Queue n: <Lista clienților din coadă>

După generarea tuturor blocurilor ( atingerea timpului limită de simulare sau procesarea tuturor clienților înainte de timpul limită este generată o linie pentru timpul mediu de așteptare)

* Average waiting time: <timp mediu de așteptare la coadă>

#### 2.3 Use-case Diagram



# 3. Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfețe, relații, packages, algoritmi, interfață utilizator)

## 3.1 Decizii de proiectare

Proiectarea orientată pe obiecte reprezintă o paradigmă de transpunere a obiectelor din lumea reală a obiectelor în clase și pachete. Tema nu specifică existența unei interfețe grafice de interacțiune cu utilizator de aceea aplicația nu respectă pachetele Model-View-Controller și din specificațiile temei nu va dispune nici de teste unitare. Aplicația este structurată în 3 pachete (Controller , Model, Main )

## 3.2 Diagrama UML

## 3.3 Proiectarea claselor, pachete, interfețe

Pachetele aplicației sunt Controller, Main, Model. Pachetul Main conține doar metoda main() și este punctul “de pornire” al aplicației. Pachetul controller conține clasa SimulationManager și este utilizată pentru monitorizarea proceselor din aplicație și furnizarea datelor de intrare/ ieșire, precum și generarea clienților și stabilirea execuției. Clasa Model conține datele transpuse în diferite obiecte: Task ( transpunerea clienților sub forma de task ce trebuie procesată), Strategy ( interfața utilizată pentru selectarea clienților pentru coadă în funcție de diferite strategii, timp minim total), Server ( reprezintă coada propiu zisă care rulează pe thread-ul ei , separate de celelalte cozi), Scheduler (se ocupa de generarea cozilor și comutarea între diferite strategii), ConcreteStrategyTime și ConcreteStrategyQueue ( implementări ale interfeței strategy , acestea vor fi folosite la adăugarea clienților în coadă în funcție de strategia alesă în Scheduler).

# 4. Implementare

**Clasa MainClass**

Clasa MainClass conține metoda main() în care este definită o instanță a clasei SimulationManager și lansată spre execuție într-un thread.

**Clasa SimulationManager**

Clasa SimulationManager este Controllerul aplicației , această clasă se ocupa de citirea și scrierea datelor cât și de generarea datelor urmând apoi a fi procesate. Atributele clasei sunt timeLimit ( timpul limită al simularii în caz că nu au fost procesați toți clienții până la atingerea timpului limită, programul se oprește), maxProcessingTime și minProcessingTime ( reprezintă limita timpului minim, respectiv maxim pentru generarea unui număr random de timp pentru procesarea clienților ) , numberOfServers ( numărul de cozi ), numberOfCliets ( numărul de client , sau task –uri ce trebuie generate), minArrivalTime si maxArrivalTime( limitele între care trebuie generat un număr aleator pentru timp de sosire al unui client la coadă) și outputFile, un atribut privat folosit la scrierea în fișier.

Clasa mai conține 4 atribute legate de execuția programului și anume: threadRunning ( care oprește toate threadurile în caz că simularea a ajuns la timpul limită dar cozile încă au client de procesat), selectionPolicy care reprezintă strategia de selecție a clienților când sunt repartizați la cozi (în acest caz SHORTEST\_TIME), scheduler ( o instanță către clasa ce se ocupă de programarea clienților și distribuirea lor în cozi) și o listă cu task-urile generate (clienții).

Constructorul clasei primește lista de argumente pentru a deschide fișierele de intrare/ ieșire și începe excuția programului (apelează metodele de citire, generare date, procesare și mai apoi scriere).

Metoda generateNRandomTasks() generează un număr de numberOfClients client cu un ID, un număr aleator pentru timpul de ajungere la coada( cuprins între minArrivalTime și maxArrivalTime) și un număr aleator pentru timpul de procesare.

Metoda readFromFile() primește ca argument un string ce reprezintă calea absolută sau relativă către fișierul de citire și începe stocarea într-un Array a valorilor citite, urmând a fi transmise atributelor clasei. Metoda citește câte o linie din fișier și folosind metoda split(regex) , unde regex =”,” separă linia în funcție de caracterul ‘,’ . Această separare a fost necesară pentru citirea limitelor superioare și inferioare a timpului de procesare și de ajungere la coadă.

Metoda writeToFile() primește ca argumente fișierul, lista de servere și un operand utilizat pentru scrierea anumitor date. În timpul rulării metoda este apelată pentru scrierea la finalul fișierului.

Metoda run() se ocupă de execuția thread-ului în care se află instanță clasei din metoda main, aceasta selecteaza pe rând clienții și în funcție de timpii de sosire și de procesare sunt atribuite serverelor corespunzatoare. Metoda rulează cât timp mai sunt clienți de procesat sau până la atingerea timpului limită.

**Clasa ConcreteStrategyQueue**

Aceasta clasă poate oferi o altă strategie de selecție a clienților pentru repartizarea în cozi, cum ar fi alegerea cozii de lungime minima. Tema a cerut implementarea unei strategii orientate pe timpul minim, de aceea aceasta clasă nu oferă implementarea metodei addTask(). Această clasă precum și altele de același timp pot fi dezvoltări ulterioare.

**Clasa ConcreteStrategyTime**

Clasa oferă selecția cozilor în funcție de timpul minim de procesare , aceasta este și strategia cerută în cadrul temei , de aceea este implementată față de ConcreteStrategyQueue.

**Clasa Scheduler**

Conține ca atribute : lista serverelor( cozile existente) , numărul maxim de cozi ( maxNoServer ), tipul strategiei ( strategy) și un câmp static folosit la calcularea timpului mediu de așteptare ( totalTime ).

Constructorul clasei primește numărul maxim de servere și generează serverele respective și un thread pentru fiecare după care este lansat în execuție. Thread-ul se va suspenda pentru ca inițial cozile sunt goale, când va fi introdus un client în server va fi creat un thread nou iar daca serverul ramane din nou fără client threadul va muri și altul va fi creat dacă este necesar.

Metoda changeStrategy() poate fi utilizată în dezvoltări ulterioare pentru alegerea unei strategii diferite de a selecta coada la care este atribuit clientul când acestă ajunge la timpul corespunzator.

**Interfața Strategy**

Oferă un model comportamental pentru strategiile ce ar trebui implementate. Acestă clasă conține doar antetul metodei addTask().

**Clasa Server**

Clasa conține ca atribut : un id ( id-ul cozii), o listă de task-uri pentru coadă (lista clienților aflați la coadă) și un AtomicInteger ,waitingPeriod, în care este calculat la fiecare procesare , timpul total de așteptare al acelei cozi. Constructorul clasei primește un id,și inițializeaza atributele. Metoda addTast() este responsabilă de suspendarea ( uciderea) threadului și crearea unui thread nou doar când există clienți la coadă, aceștia fiind adăugati imediat după deschiderea threadului.

Clasa implementează intefața Runnable și oferă implementare pentru metoda run(), în această metodă se sincronizează toate threadurile prin sleep(1000) (o secundă timp real) și la fiecare secundă se decrementează timpul de procesare al clientului din capul cozii. Metoda rulează cât timp în lista de task-uri există cel puțin 1, altfel threadul curent este lăsat să moară.

**Clasa Task**

Clasa conține ca atribute: id (id –ul clientului), arrivalTime (timpul în care clientul ajunge la coadă), finishTime (timpul de terminare al procesarii clientului) și processingTime (timpul de procesare curent).

Constructorul clasei este apelat de SimulationManager la generarea Clienților ( considerate task-uri în acestă aplicație) , inițializând atributele clasei. FinishTime –ul va fi calculat în momentul ieșirii din coadă.

Clasa conține gettere și settere și implementează interfața Comparable pentru o sortare ascendentă în funcție de timpul de ajungere la coadă. Sortarea este realizată dupa generearea tuturor task-urilor .

# 5. Rezultate

Specificațiile temei nu solicită existența unor clase de test în JUnit. Un exemplu de intrare/ ieșire este :

Date de intrare:

4

2

60

2,30

2,4

Rezultate generate

Time:0

Waiting clients: (4,2,3);(2,10,2);(1,13,2);(3,17,4);

Queue 1: closed

Queue 2: closed

Time:1

Waiting clients: (4,2,3);(2,10,2);(1,13,2);(3,17,4);

Queue 1: closed

Queue 2: closed

Time:2

Waiting clients: (2,10,2);(1,13,2);(3,17,4);

Queue 1:(4,2,3);

Queue 2: closed

………………………

Time:20

Waiting clients:

Queue 1:(3,17,1);

Queue 2: closed

Time:21

Waiting clients:

Queue 1: closed

Queue 2: closed

Average waiting time: 2.75

# 6. Concluzii

Din aceasta temă am învățat să lucrez cu mai multe fire de execuție și definirea unui tip de date “enum” pentru stocarea diferitelor constante. Aplicația actuală poate fi îmbunătățită prin adaugarea mai multor strategii de selecție precum și prin crearea unei intefețe grafice .

# 7. Bibliografie

1. <http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_2/Java_Concurrency.pdf>
2. <https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html>
3. <https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html>