**DOCUMENTAȚIE TEMA 2**

**SIMULATOR DE COZI**

**Ovreiu Auraș Dănuț**

**Grupa 30229**

**Profesor Laborator Assist Antal Marcel**

Contents

[1. Cerințe funcționale 3](#_Toc68768084)

[2. Obiective 3](#_Toc68768085)

[2.1. Obiectivul principal 3](#_Toc68768086)

[2.2. Obiective secundare 3](#_Toc68768087)

[3. Analiza problemei 3](#_Toc68768088)

[4. Proiectare 4](#_Toc68768089)

[4.1. Structuri de date 4](#_Toc68768090)

[4.2. Algoritmi 4](#_Toc68768091)

[4.3. Diagrame de clase 5](#_Toc68768092)

[5. Implementare 6](#_Toc68768093)

[5.1. Clase 6](#_Toc68768094)

[5.2. Metode 8](#_Toc68768095)

[6. Testare 10](#_Toc68768096)

[7. Concluzii și Dezvoltări Ulterioare 11](#_Toc68768097)

[8. Bibliografie 11](#_Toc68768098)

# 1. Cerințe funcționale

La această temă se cere implementarea unui simulator de cozi pentru a determina și a minimiza timpul de așteptare al clienților. Fiecare client sosește la un anumit moment de timp în care este adăugat la una dintre cozi astfel încât să poată fi servit cât mai repede posibil. După ce clientul este servit acesta este eliminat din coada în care se află, simularea terminându-se atunci când toți clienții au fost serviți sau timpul de simulare a expirat.

# 2. Obiective

## 2.1. Obiectivul principal

Obiectivul principal al acestei teme este proiectarea și implementarea unei aplicații care să simuleze o serie de cozi la care așteaptă un anumit număr de clienți, eficientizarea acesteia realizându-se pe baza unor criterii bine definite de la început.

## 2.2. Obiective secundare

|  |  |  |
| --- | --- | --- |
| **Obiectiv Secundar** | **Descriere** | **Capitol** |
| Descrierea cerinței | se prezintă sumar funcționalitatea proiectului | 3 |
| Alegerea structurilor de date | se aleg structuri de date care permit stocarea clienților și a cozilor, precum și gestionarea acestora | 4 |
| Împărțirea pe clase | proiectul se împarte pe clase grupate în pachete astfel încât să se poată afișa rezultatele obținute în simulare pe interfața grafică în timp real sau doar după terminarea simulării(atunci când se apasă pe buton) | 4 |
| Dezvoltarea algoritmilor | se vor implementa algoritmi pentru ca timpul mediu de așteptare al clienților să fie minim | 4 |
| Implementarea soluției | se va descrie fiecare clasă cu metode si câmpuri importante, precum și implementarea interfeței utilizator | 5 |
| Testare | se verifică funcționarea corectă a aplicației prin rularea mai multor teste | 6 |

# 3. Analiza problemei

În cadrul acestei teme sunt impuse o serie de specificații, după cum urmează:

* folosirea paradigmelor de programare orientată obiect(încapsulare, folosirea de clase aferente, denumirea claselor, metodelor, pachetelor, variabilelor în mod corect)
* utilizarea unui generator de numere random pentru setarea detaliilor despre clienți
* utilizarea a câte unui thread pentru fiecare coadă
* realizarea unei interfețe grafice dedicate în care să fie posibilă vizualizarea detaliilor din testul curent, precum și evoluția simulării în timp real
* afișarea rezultatelor simulării pe interfața grafică sau într-un fișier text

Cozile sunt de obicei utilizate pentru a modela domenii din lumea reală. Obiectivul principal al unei cozi este de a oferi un loc pentru un „client“ să aștepte înainte de a primi un „serviciu“.

Modelarea problemei a fost facută în mare parte după exemplul alăturat prezentării temei, astfel încât cozile sunt modelate ca servere, care primesc task-uri (clienți) pe care trebuie să le proceseze. Serverele sunt monitorizate și primesc task-uri de la un scheduler, care consultă timpul de așteptare la fiecare server în parte și ia o decizie cu privire la coada căreia să îi asigneze următorul client pentru a facilita eficiența. Pentru a determina cărei cozi să îi fie asignată clientul, există două opțiuni: SHORTEST\_QUEUE(se va alege coada cu cei mai puțini clienți) și SHORTEST\_TIME(se va alege coada cu cel mai mic timp de așteptare-suma timpilor de servire a clienților). De asemenea, scheduler-ul este responsabil și pentru închiderea și deschiderea serverelor atunci când este nevoie.

# 4. Proiectare

## 4.1. Structuri de date

Principalele structuri de date folosite sunt lista(List<Client> client=new ArrayList<>(), this.threads=new ArrayList<>(maxNoServers)) și blockinq queue(this.clienti=new ArrayBlockingQueue<>(nrMax)). Acestea permit memorarea mai ușoară a datelor(în cazul de față a clienților, threadurilor, cozilor etc) și mai eficientă, operațiile pe aceste structuri de date fiind mai la îndemână pentru simulatorul de cozi. Alte structuri folosite pentru a reține mai ușor clienții și cozile au următoarea formă:

public Client(int ID, int arrivalTime, int serviceTime){  
 this.ID=ID;  
 this.arrivalTime=arrivalTime;  
 this.serviceTime=serviceTime;  
}

public Coada(int nrMax){  
 this.clienti=new ArrayBlockingQueue<>(nrMax);  
 this.waitingPeriod=new AtomicInteger();  
}

## 4.2. Algoritmi

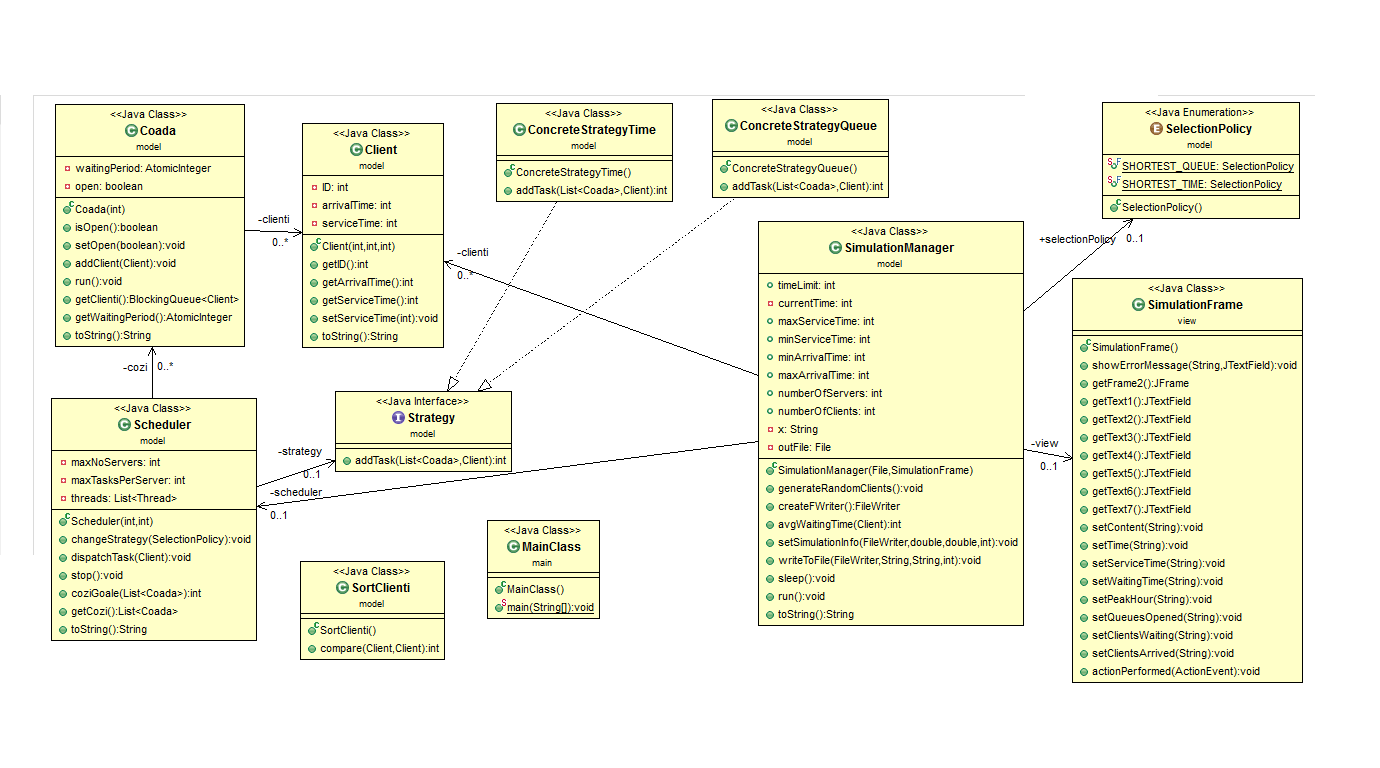
Algoritmii utilizați la această temă sunt cei de determinare a cozii cu cel mai mic timp de așteptare, respectiv cu cel mai mic număr de clienți.

Pentru determinarea cozii cu cel mai mic timp de așteptare, parcurgem lista de cozi cu două bucle for-each. În prima buclă determinăm coada cu timpul minim de așteptare, iar în cea de-a doua adăugăm clientul sosit la coada

respectivă. În mod asemănător, pentru aflarea și adăugarea clientului la coada cu cel mai mic număr de clienți aflăm mai întâi care este coada cu cea mai mică dimensiune, adică cea la care se află cel mai mic număr de clienți la momentul actual și adăugăm ulterior clientul la acea coadă.

## 4.3. Diagrame de clase

Unified Modeling Language sau UML pe scurt este un limbaj standard pentru descrierea de modele si specificații pentru software. UML a fost la bază dezvoltat pentru reprezentarea complexității programelor orientate pe obiect, al căror fundament este structurarea programelor pe clase, și instanțele acestora ( numite și obiecte ). Cu toate acestea, datorită eficienței și clarității în reprezentarea unor elemente abstracte, UML este utilizat dincolo de domeniul IT.



# 5. Implementare

## 5.1. Clase

***Clasa MainClass:***

Este clasa în care spunem programului ce să execute. În această clasă se realizează deschiderea interfeței grafice și este pornită simularea de către threadul principal.

***Clasa Client:***

Este clasa în care ne creăm clienții reprezentați de un ID și caracterizați de timpul de sosire și cel de procesare.

***Clasa Coadă:***

Este clasa în care ne creăm cozile caracterizate prin timpul de așteptare și o structură de tip coadă în care sunt stocați clienții aflați la coadă la momentul curent.

***Clasa ConcreteStrategyQueue:***

Este clasa în care se determină coada cu cel mai mic număr de clienți la momentul actual.

***Clasa ConcreteStrategyTime:***

Este clasa în care se determină coada cu cel mai mic timp de așteptare.

***Clasa Scheduler:***

Această clasă este responsabilă de management-ul serverelor(cozilor) și este reprezentară de o listă de cozi, un număr maxim de servere(numărul de cozi) și de task-uri într-un server(numărul de clienți), o strategie aplicată pentru a eficientiza procesul și de o listă de threaduri, câte unul pentru fiecare coadă, care răspund de gestionarea cozilor la fiecare secundă din simulare.

***Clasa SelectionPolicy:***

Este o clasă conținând doar un enum pentru selecția strategiei ce trebuie aplicată la adăugarea clienților la coadă. Strategiile respective sunt SHORTEST\_QUEUE și SHORTEST\_TIME.

***Clasa SimulationManger:***

Este clasa responsabilă de simularea aplicației. Aceasta conține o serie de variabile ce o caracterizează în mod direct. În primul rând se remarcă variabilele care se dau ca input pentru aplicație(timeLimit, maxServiceTime, minServiceTime, minArrivalTime, maxArrivalTime, numberOfServers, numberOfClients) și sunt și afișate pe interfața grafică. În continuare mai avem un String “x” instanțiat ca fiind String-ul gol în care se vor reține informațiile detaliate despre evoluția simulării, “selectionPolicy” pentru a selecta strategia dorită, “scheduler” pentru a instanția management-ul cozilor, “outFile” pentru afișarea în fișier a informațiilor deduse din simulare, “view” pentru a instanția interfața responsabilă de afișarea informațiilor pe ecran și “clienti” pentru a instanția lista de clienți care urmează să vină la cozi.

***Clasa SortClienti:***

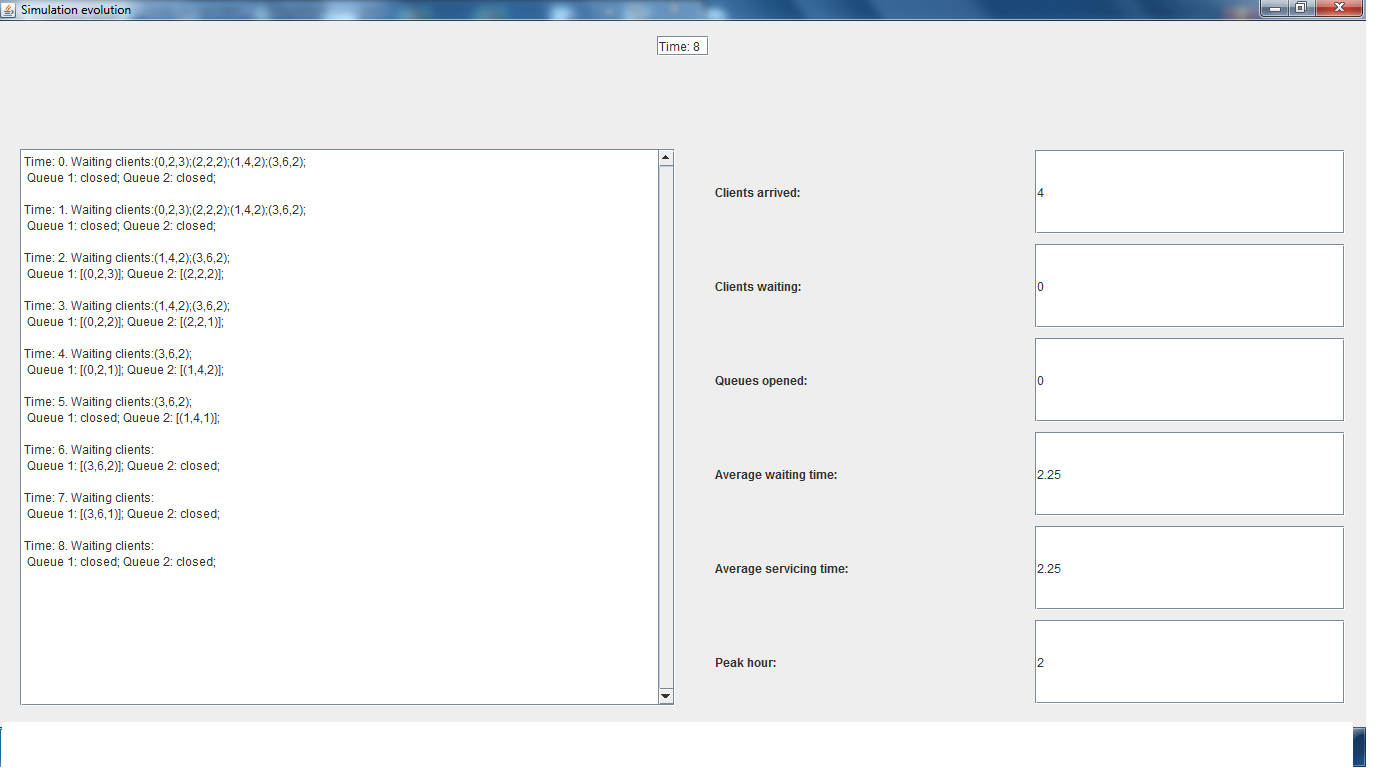
Este clasa în care se realizează sortarea clienților după timpul de sosire.

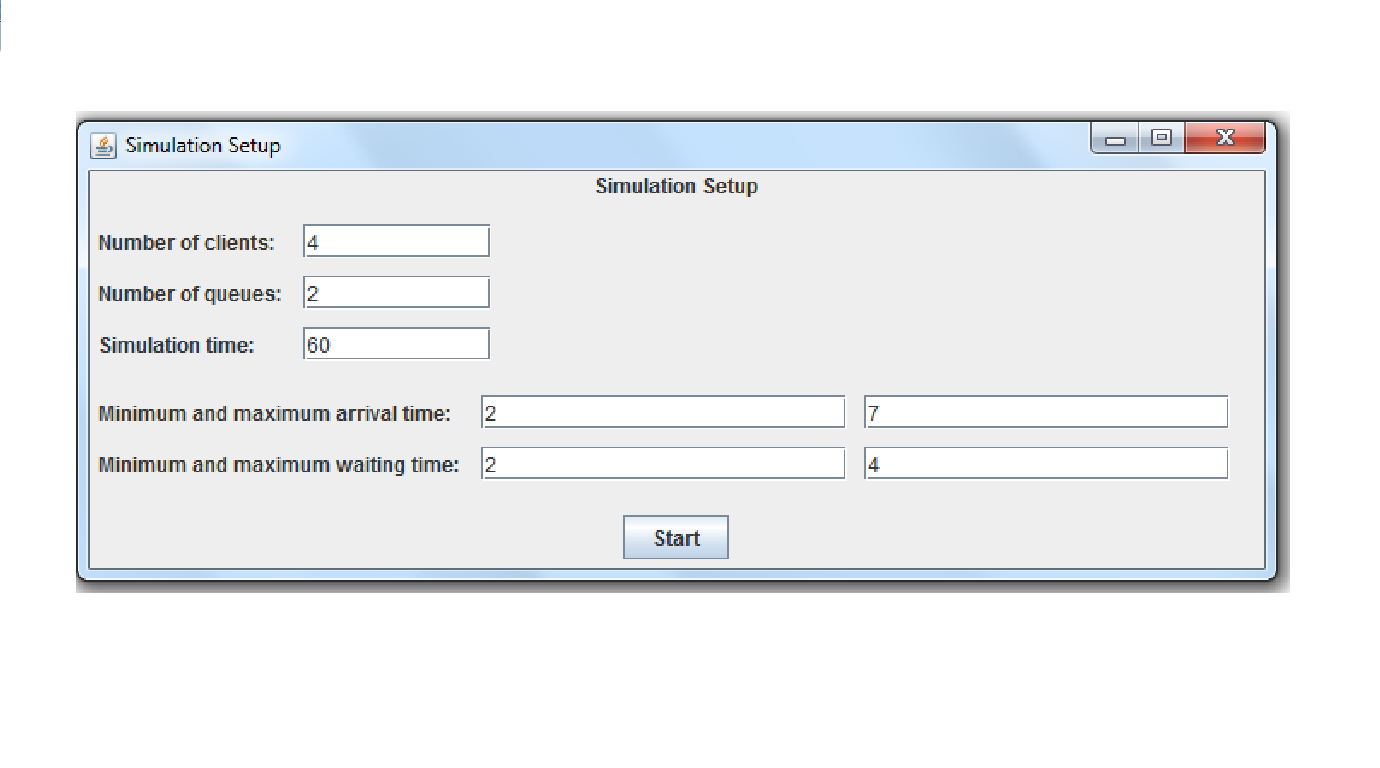
***Clasa Strategy:***

Reprezintă o interfață responsabilă de adăugarea clienților în cozi dupa criteriile menționate mai sus.

***Clasa SimulationFrame:***

Este clasa responsabilă de afișarea interfeței grafice. Aceasta este constituită din două frame-uri, unul pentru afișarea parametrilor simulării și celălalt pentru afișarea în timp real a evoluției simulării. Din această clasă poate fi pornită simularea prin intermediul unui buton de start, însă care poate doar să afișeze informațiile din simulare doar după ce aceasta din urmă s-a încheiat.





## 5.2. Metode

***Metode utilizate în clasa Client***

În această clasă se regăsesc metode pentru reținerea și modificarea caracteristicilor clientului(getteri și setteri). Tot aici întâlnim și metoda toStrting pentru afișarea cât mai corectă a informațiilor despre client(ID-ul său, timpul de sosire și cel de servire).

***Metode utilizate în clasa Coada***

În această clasă utilizăm getteri și setteri pentru o variabilă “open” care ne indică dacă o coadă este închisă(nu mai are clienți) sau deschisă, dar și getteri pentru informații specifice cozii, cum ar fi coada în care sunt reținuți clienții și perioada de așteptare a cozii, care reprezintă de fapt suma timpilor de așteptare a clienților săi.

Tot aici avem o metodă denumită addClient, care așa cum îi zice și numele este folosită pentru a adăuga un nou client în structura de tip coadă, mărind totodată și timpul de așteptare al cozii cu timpul de servire al clientului adăugat.

Mai departe este suprascrisă metoda run(), clasa implementând interfața Runnable. Această metodă descrie evoluția cozii pe parcursul simulării. Cât timp coada este deschisă și mai are încă clienți thread-ul asociat este adormit cu o anumită perioadă de timp(în cazul de față o secundă), timp după care perioada de așteptare a cozii, dar și timpul de servire al clientului care se află în fața cozii se decrementează. Atunci când timpul de servire al clientului devine 0 înseamnă că acesta a fost servit și va fi eliminat din coadă, urmând să fie procesat următorul client care stă la rând. Întregul proces este descris printr-o buclă while, acesta terminându-se când ultimul client din coadă a fost servit, moment în care variabila booleană “open” va fi setată la false, semnalând închiderea cozii.

Această clasă mai conține și metoda toString() care are rolul de a afișa clienții care stau la rând. Dacă coada este goală sau timpul său de așteptare ori timpul de servire al clientului din fața cozii este zero se va afișa mesajul “closed”. Altfel se vor afișa informații despre clienții din coadă(ID, timp sosire, timp servire).

***Metode utilizate în clasa ConcreteStrategyQueue:***

Această clasă conține doar metoda addTask() suprascrisă prin interfața Strategy ce adaugă un client în coada cu cel mai mic număr de clienți.

***Metode utilizate în clasa ConcreteStrategyTime:***

Această clasă conține doar metoda addTask() suprascrisă prin interfața Strategy ce adaugă un client în coada cu cel mai mic timp de așteptare.

***Metode utilizate în clasa Scheduler:***

O primă metodă din această clasă este changeStrategy(), care nu face decât să selecteze strategia dorită.

Urmează metoda dispatchTask(), în care se adaugă clientul la coadă conform strategiei alese. Atunci când coada este închisă aceasta se va deschide și se va porni un nou thread în locul ei.

Metoda stop() setează toate cozile ca fiind închise pentru a semnala terminarea procesului de simulare, iar metoda coziGoale() verifică dacă toate cozile sunt goale.

Și în această clasă regăsim metoda toString() în care se afișează numărul cozii și informațiile despre clienții din cozi preluate din metoda toString() a clasei Coada.

***Metode utilizate în clasa SimulationManager:***

În această clasă se regăsesc metode importante care contribuie la realizarea simulării aplicației.

O primă metodă este generateRandomClients() în care sunt generați clienți în mod aleator(timpii de sosire și servire se pot afla într-un interval definit la începutul simulării) și apoi sortați după timpul de sosire.

Metoda răspunzătoare de procesarea clienților este run(), în care informațiile obținute sunt salvate într-un fișier text. Tot procesul are loc într-o buclă while care se termină când a expirat timpul de simulare. Atunci când timpul de simulare devine egal cu cel de sosire al primului client acesta este eliminat din listă și adăugat în coadă conform criteriilor menționate mai sus. Tot aici se vor afla și niște detalii adiționale, cum ar fi timpul mediu de așteptare(cât a stat fiecare client în medie la coadă), timpul mediu de servire(media timpilor de servire al clienților) și ora de vârf(momentul din simulare când au existat cei mai mulți clienți la cozi), informații care vor fi afișate atât pe interfața grafică cât și în fișier.

Afișarea informațiilor se bazează și în această clasă pe metoda toString() care înglobează toate informațiile acumulate din simulare.

***Metode utilizate în clasa SimulationFrame:***

Această clasă răspunde de afișarea interfeței grafice, având o serie de elemente caracteristice acesteia.

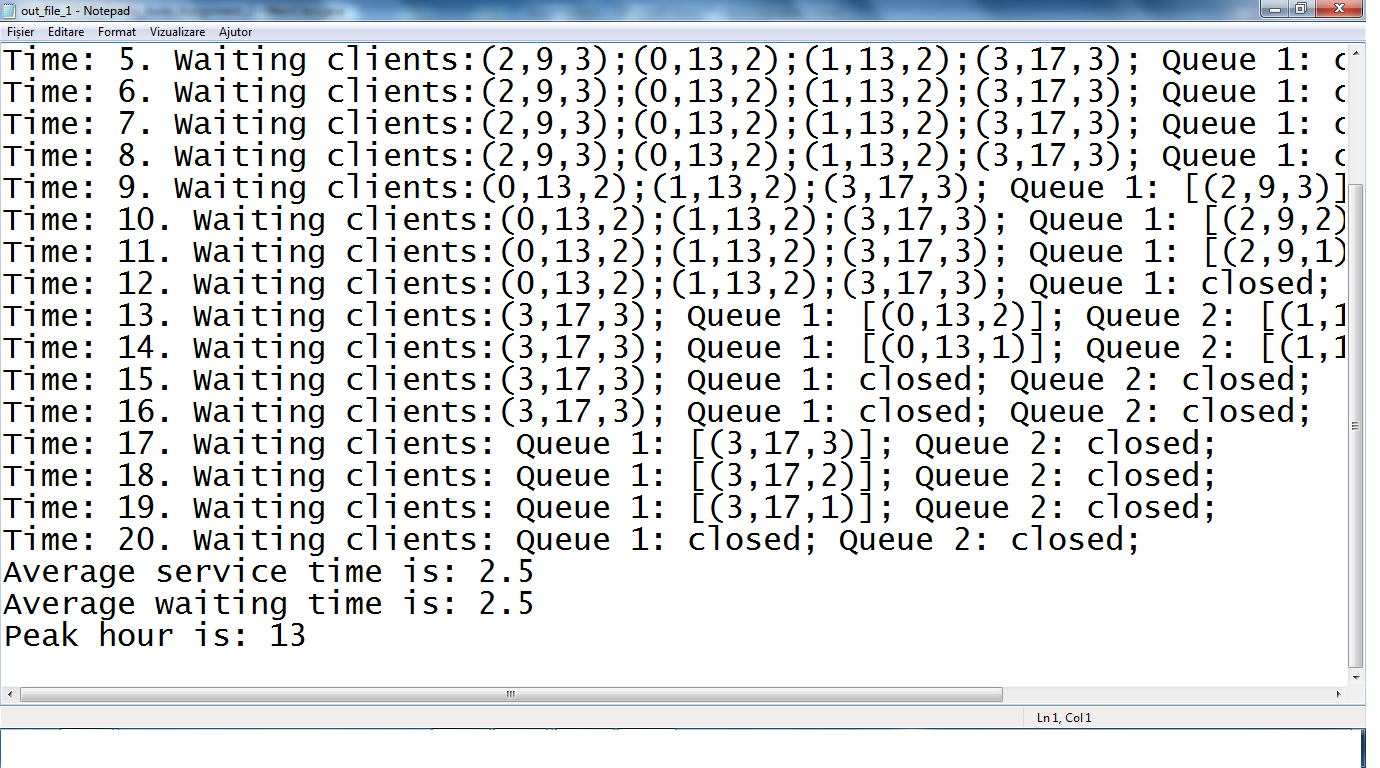
Principalele componente ce pot fi văzute sunt text field-uri, label-uri, dar și un buton de start care pornește simularea și un scroll pane în care se pot afișa informații din simulare, oricât de lungi ar fi.

Metodele principale sunt getterii și setterii pentru componentele celor două frame-uri.

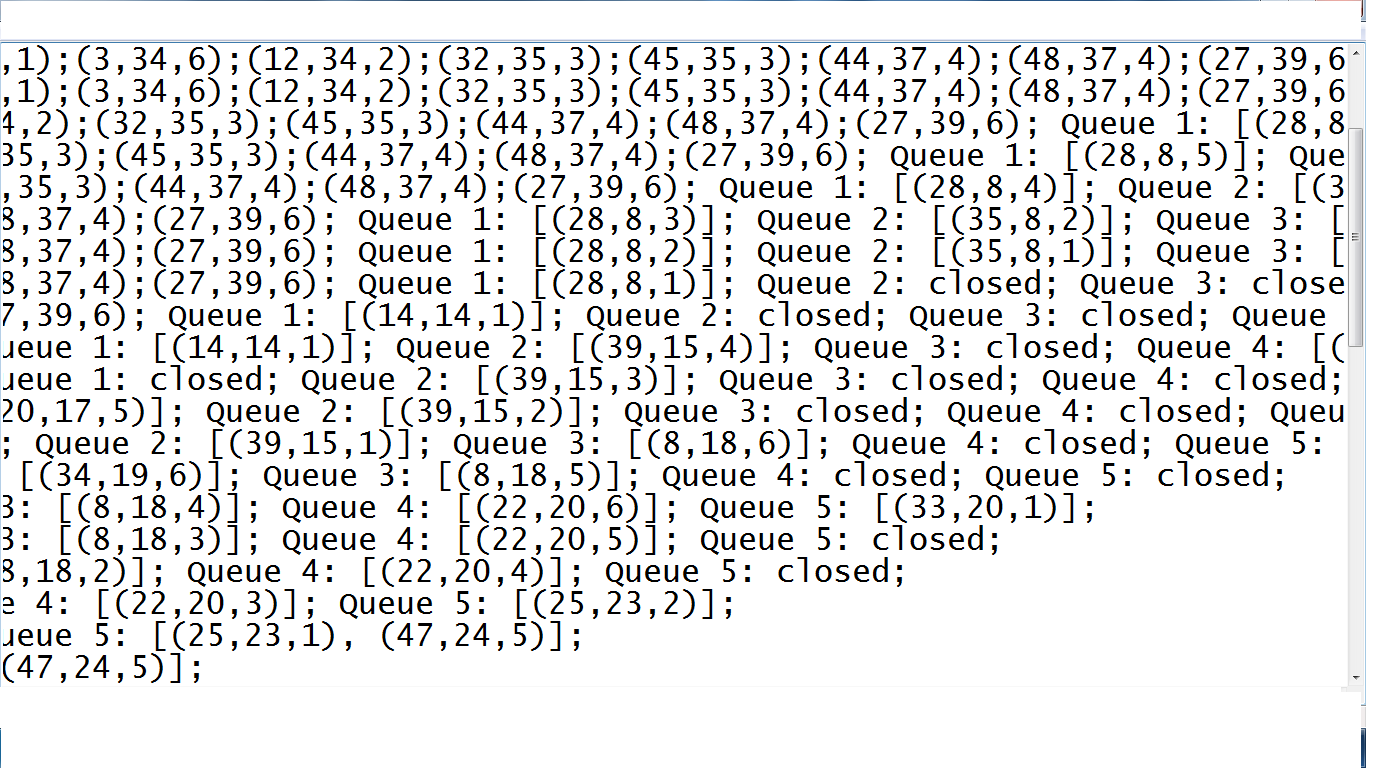
Metoda actionPerformed() este apelată la apăsarea butonului de start, iar dacă informațiile introduse în primul frame sunt valide(numere întregi pozitive) se va deschide un nou frame în care se vor afișa rezultatele simulării atât pe interfața grafică, cât și în fișier. Din această metodă se pornește și un thread care să dea drumul la simulare.

# 6. Testare

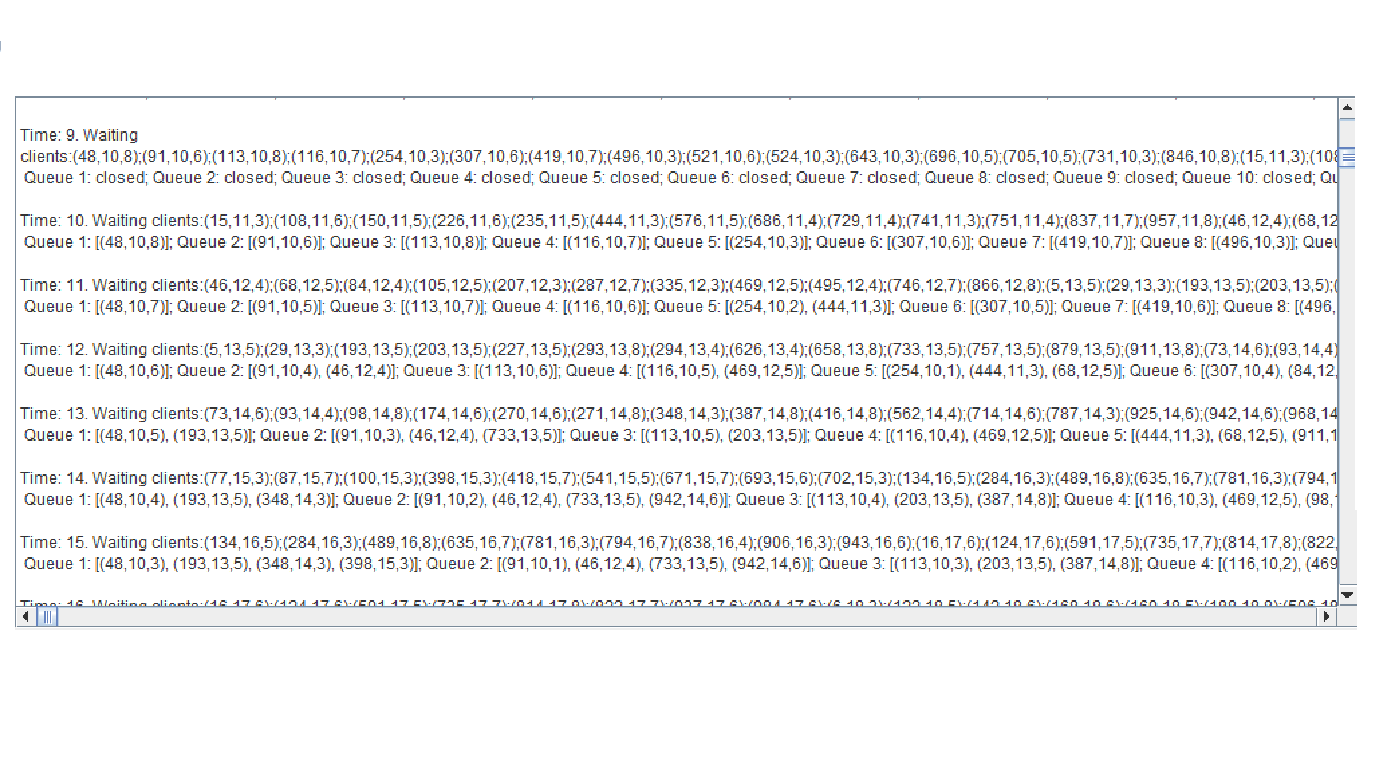
Pentru verificarea corectitudinii funcționării aplicației am rulat simularea pe baza informațiilor din cele 3 fișiere text ilustrate în cerința temei, rezultatele fiind la toate trei cele dorite.



Test 1:



Test 2:

****

Test 3:

# 7. Concluzii și Dezvoltări Ulterioare

În concluzie, sunt de părere că acest proiect m-a ajutat să îmi aprofundez cunoștințele legate de thread-uri, cu care nu am mai lucrat niciodata până acum, știind doar niște noțiuni de la alte materii.

Ca și dezvoltări ulterioare, s-ar putea implementa acest algoritm în viața reală, pentru gestionarea cozilor la supermarket, aeroport sau alte locuri aglomerate.

# 8. Bibliografie

1. Pdf-ul cu Java\_Concurrency

2. Wikipedia

3. Youtube