**DOCUMENTAȚIE TEMA 2**

**QUEUES SIMULATOR**

**Student: Raul - Alexandru Gorgan**

**Grupa: 30227**

**Profesor Laborator: Dan Mitrea**

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)

# Cerințe Funcționale

Se cere implementarea unei aplicații care simulează funcționarea cozilor cu scopul de a evidenția modul de creare și manipulare a firelor de lucru prin intermediul paradigmei programării orientate pe obiecte. Aplicația trebuie să poată interacționa cu utilizatorul într-un mod cât mai ușor pentru acesta, printr-o interfață grafică: utilizatorul oferă numărul de cozi, numărul de clienți, timpul de simulare, intervalul de timp în care să ajungă clienții în coadă și intervalul de timp în care un client stă în fața cozii. Rezultatul simulării se va afișa real-time în interfață dar se va și menține într-un fișier text, pentru a putea fi analizat ulterior.

# Obiective

## Obiectiv Principal:

Obiectivul principal al acestui proiect este de propunere, proiectare și implementare al unui simulator pentru cozi folosind threaduri și tipuri și structuri de date safe pentru acestea. Simulatorul e o aplicație care primește la intrare numărul de cozi, numărul de clienți, timpul de simulare, intervalul de timp în care să ajungă clienții în coadă și intervalul de timp în care un client stă în fața cozii. La ieșire, aplicația va afișa parcursul simulării și rezultatele calculate în timpul ei (ex: timpul mediu de așteptare, timpul mediu de procesare, momentul de timp cel mai aglomerat etc.). Aplicația trebuie integrată într-o interfață grafică pentru a face mai ușoară și mai atractivă comunicarea, utilizatorul fiind cel care își alege propriul set de date de intrare și poate alege momentul când să înceapă simularea.

## Obiective Secundare:

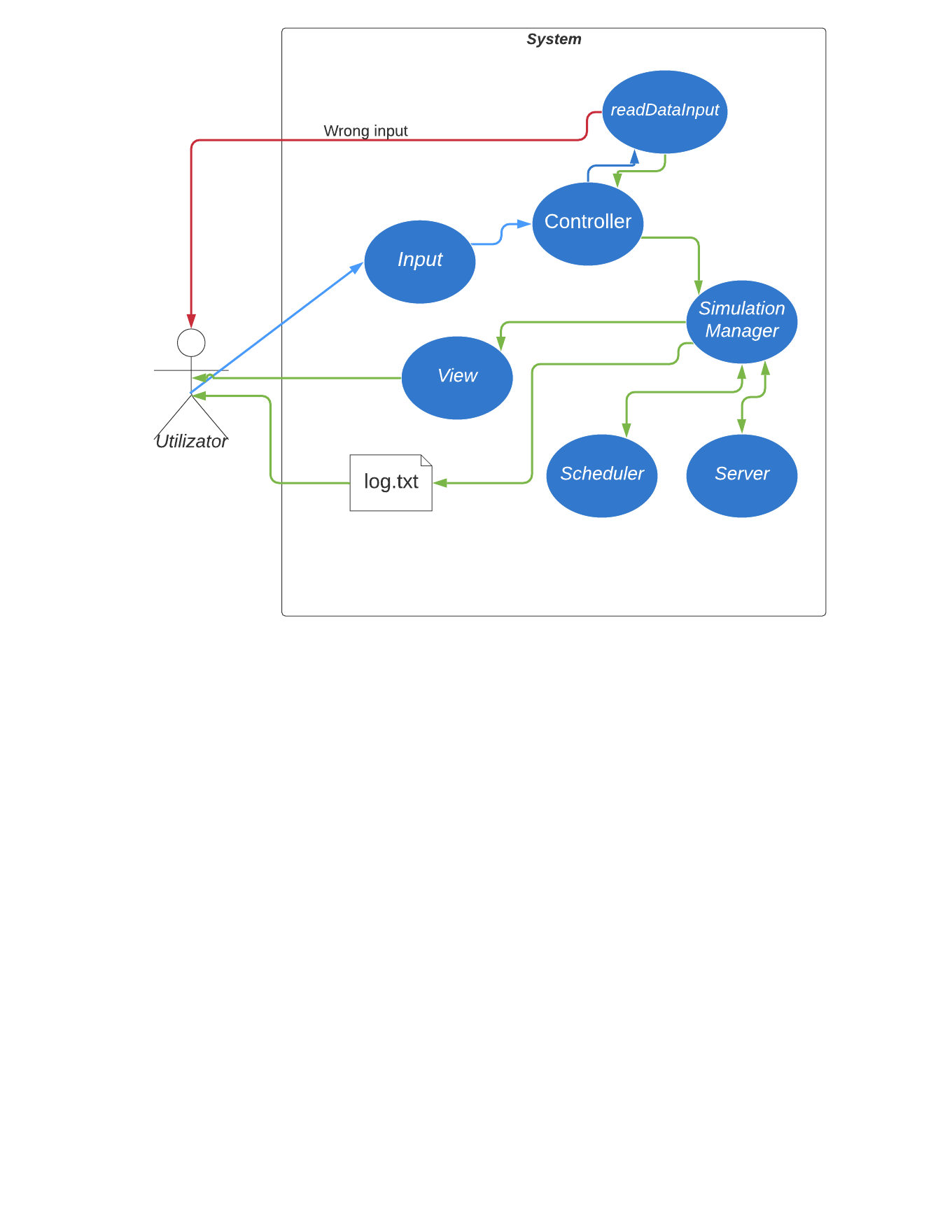
Pentru a putea îndeplinii obiectivul principal al acestui proiect, este nevoie de stabilirea unor obiective secundare care ne vor conduce la realizarea lui. Stabilirea acestora este importantă înainte de implementarea unei aplicații, deoarece o proiectare bună poate aduce multe beneficii, atât pentru proiectant cât și pentru utilizator. O bună alegere a soluției poate face ca aplicația să fie mai ușor de scris, întreținut, chiar și de promovat. Utilizatorul dorește să aibă o aplicație care efectuează calculele într-un timp minim, să fie user-friendly, să lucreze ușor și intuitiv. Aceste avantaje pot fi rezultatul unei analize în detaliu și a unei proiectări de calitate.

Un obiectiv secundar care intră în discuție la acest proiect este analiza problemei prin dezvoltarea de use case-uri și de scenarii, care vor fi prezentate în capitolul numărul 3. Acesta ne va ajuta la îndeplinirea unui alt obiectiv secundar și anume alegerea structurilor de date folosite, alegerea fiind argumentată la capitolul 4. Tot la capitolul 4 vor fi dezvoltate și celelalte obiective secundare: împărțirea pe clase, respectând paradigmele programării orientate pe obiecte și dezvoltarea algoritmilor esențiali. După analiza în detaliu a problemei, la capitolul 5 se descrie pe larg implementarea soluției, aceasta fiind urmată de faza de testare, ultimul obiectiv secundar în vederea finalizării aplicației.

# Analiza Problemei

Analiza problemei începe cu identificarea use case-urilor și a scenariilor. Niciodată nu este garantat că utilizatorul o să introducă datele exact în formatul specificat sau o să urmeze pașii în ordinea corectă, așa că este nevoie de analiza tuturor cazurilor de utilizare și a cazurilor limită. Aplicația trebuie să acopere toate aceste cazuri și să afișeze mesaje sugestive utilizatorului.

Diagramele de use case sunt o variantă vizuală a pașilor care se parcurg la o simulare:



Aceasta este diagrama generală de use case. Parcursul albastru e comun celor două scenarii (scenariul încheiat cu succes și scenariul alternativ / încheiat cu afișarea unui mesaj de eroare). Dacă inputul e corect, parcursul va urma linia verde, dacă nu, va urma linia roșie.

Use case general:

Actorul principal: utilizatorul

Scenariul de success:

1. Utilizatorul introduce cele numărul de clienți, numărul de cozi, timpul simulării, intervalele de timp pentru arrivalTime și processingTime.
2. Utilizatorul pornește simularea prin click pe butonul „Start” din interfața grafică. Butonul „Start” devine roșu.
3. Aplicația extrage datele de la utilizator, creează toate entitățile necesare iar simularea începe.
4. Este afișată real-time în interfață evoluția cozilor. La sfârșitul simulării, butonul „Start” redevine verde.

Scenarii alternative:

* Utilizatorul introduce greșit datele, oferă caractere care nu se pot converti la tipul întreg.
* Utilizatorul scrie greșit intervalele de timp, nefiind posibilă începerea simulării.
* În toate aceste cazuri, va fi aruncată o excepție definită de programator, iar execuția aplicației va continua întorcându-se la pasul 1.

# Proiectare

## Structuri de date

Pornind de la cerință și de la analiza problemei, sistemul are ca intrări numărul de task-uri, numărul de cozi, timpul de simulare și limitele inferioare și superioare ale timpului în care task-ul e pregătit să stea în coadă și ale timpului de procesare al task-urilor (timpul în care acesta stă în vârful cozii). Ca răspuns, sistemul afișează evoluția cozilor în funcție de timp (în secunde). Astfel, principalele instanțe de obiect cu care se lucrează sunt: Task (clienții) și Server (adică coada). Aceste obiecte sunt procesate și manipulate cu ajutorul altor clase (ex: Scheduler sau SimulationManager).

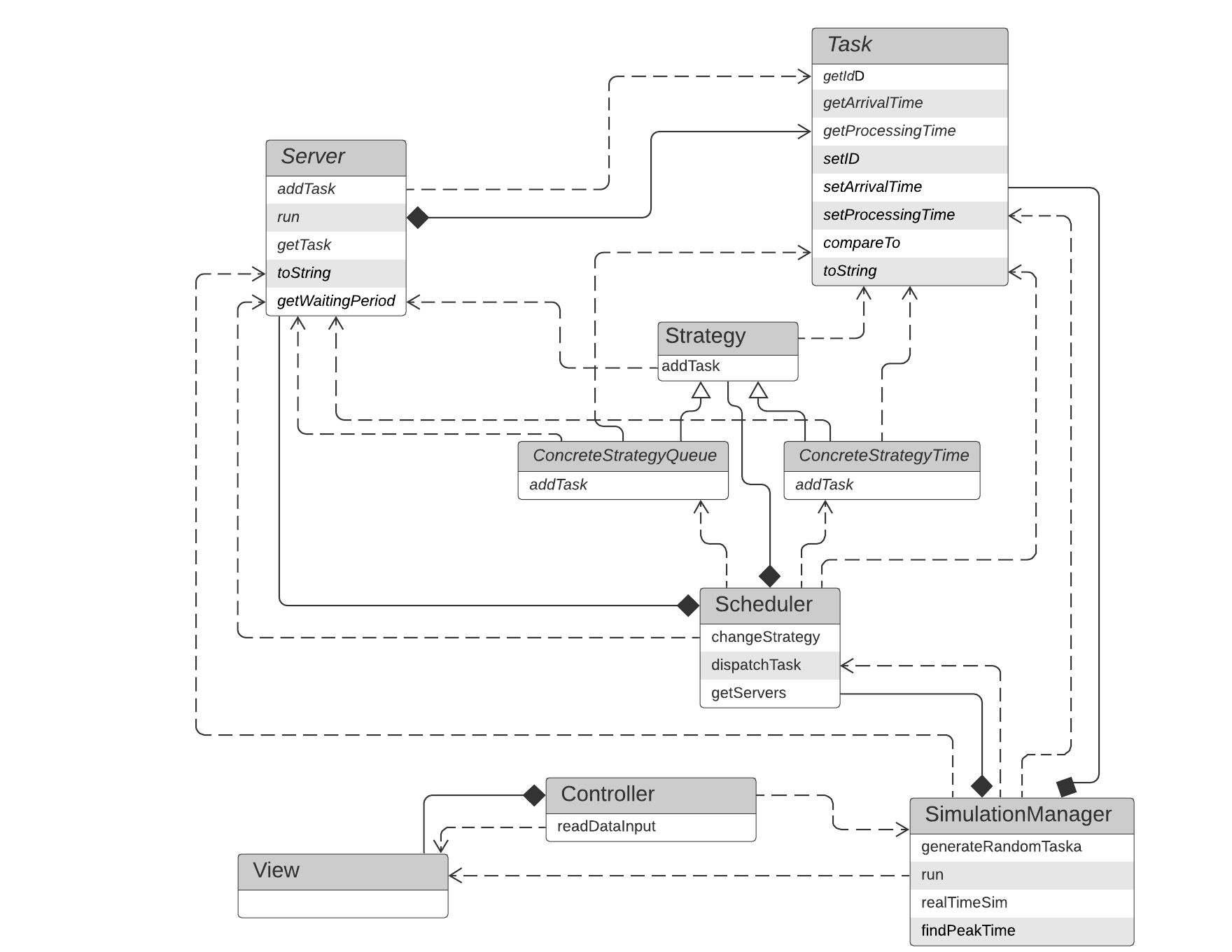
Deoarece se dorește ca implementarea să fie Thread safety, nu se va folosi tipul int pentru menținerea timpului de așteptare într-o coadă, ci se va folosi tipul AtomicInteger. Acest lucru rezolvă problema acțiunilor compuse, când, în cazul threadurilor, acestea pot accesa simultan variabila, lucru care poate duce la inconsistențe. Tipul AtomicInteger face operația de extragere și procesare într-un singur pas. Pentru a menține thread safety, se folosesc colecții de date sigure (colecții concurente) precum BlockingQueue pentru implementarea listei de cozi (Servere).

## Diagrama de clase

Organizarea claselor în pachete îl reprezintă nivelul al doilea de design. Deoarece aplicația este proiectată după modelul arhitectural Model-View-Controller, vor fi trei pachete care se ocupă de buna funcționare: pachetul pentru modele, pentru logica aplicației și pentru interfața grafică cu utilizatorul. Acest tip de structură oferă multe avantaje de proiectare, fiind mult mai ușor de întreținut.

Nivelul al treilea în design-ul unei aplicații îl reprezintă împărțirea pachetelor în clase. Acest lucru trebuie făcut după o analiză a necesităților și a resurselor. Prin urmare, pachetul Model conține clasele: Task (clientul), Server (coada), Scheduler (planificatorul simulatorului) și subpachetul Strategy. Acesta lui conține clasele care se ocupă de strategia folosită: interfața Strategy, care conține metoda neimplementată de adăugare a unui task într-o coadă, cele două clase care o implementează, ConcreteStrategyQueue (adăugare în funcție de numărul de clienți din coadă) și ConcreteStrategyTime (adăugare în funcție de timpul de așteptare) și o clasă enum care va ține cele două tipuri de politici de selecție. Pachetul View conține clasa cu același nume care se ocupă de implementarea interfeței grafice. Pachetul Controller conține clasa SimulationManager care se ocupă de managementul simulării și clasa Controller care pune actionListener pe butoane și trimite intrările către simulator. Un alt pachet, Exception conține excepțiile create de programator pentru a controla orice situație care ar duce la distrugerea integrității aplicației sau a rezultatelor afișate.

Prin urmare, clasele rezultate în urma analizei pot fi vizualizate într-o diagramă de clase:



## Algoritmi

După ce s-a realizat împărțirea în clase, design-ul problemei continuă cu analiza metodelor. Trebuie proiectați algoritmii folosiți în cadrul simulării și anume: algoritmii metodelor run() pe care le folosesc threadurile, algoritmul de selecție al tipului de politică aleasă, algoritmul de generare random al task-urilor.

**Algoritmul de generare random a task-urilor** constă în crearea unui număr de n task-uri cu datele oferite de utilizator în interfață, n fiind numărul task-urilor ce trebuie generate. Algoritmul a fost proiectat în felul următor: se declară o variabilă de tipul Task; repetând de n ori într-un for, se iau un timp de procesare și un timp de ajungere în coadă cu funcția Math.random() apoi se creează un obiect Task cu aceste date, ID-ul fiind contorul la care se adaugă 1; obiectul creat se adaugă în lista de Task-uri; lista va fi sortată după timpul de ajungere (arrivalTime) cu ajutorul metodei compareTo din interfața Comparable. Lista cu Task-uri va fi folosită de SimulationManager pentru a împărții task-urile în cozi.

**Algoritmul de schimbare a politicii de selecție** constă în verificarea parametrului dat și anume: dacă se dă politica SHORTEST\_QUEUE, obiectul de tip strategy se va instanția ca fiind ConcreteStrategyQueue, iar selecția cozii se va face în funcție de numărul de clienți din coadă; dacă se dă politica SHORTEST\_TIME, obiectul de tip strategy se va instanția ca fiind ConcreteStrategyTime, iar selecția cozii se va face în funcție de timpul de așteptare de la acea coadă.

**Algoritmul metodei run() din clasa Server** va fi folosită de thread-urile create în clasa Scheduler (câte un thread pentru fiecare server. Algoritmul constă în următorii pași: se extrage (fără a se șterge) task-ul din vârful cozii; dacă acesta există, i se decrementează ProcessingTime-ul și se decrementează și waitingTime-ul cozii; dacă acum task-ul are processingTime egal cu 0 (adică a fost complet procesat) se scoate din coadă; apoi se pune thread-ul pe pauză pentru o secundă cu metoda sleep().

**Algoritmul metodei run() din clasa SimulationManager** are rolul de a trimite fiecare task la coada potrivită în momentul în care arrivalTime-ul lui e mai mare sau egal cu timpul curent de simulare. Se scriu permanent rezultatele simulării în fișierul deschis apoi se incrementează timpul de simulare și se pune thread-ul pe pauză timp de 1 secundă. Se scriu în fișier și alte date de interes (peak time, timpul mediu de așteptare, timpul mediu de servire) iar apoi se închide fișierul.

# Implementare

În urma analizei problemei și a implementării design-ului, au reieșit următoarele clase: Task și Server, care reprezintă structurile de date ale aplicației, Scheduler care e planificatorul simulării și clasa care creează threadurile, clasele care țin de implementarea strategiei: interfața Strategy, enum-ul SelectionPolicy și clasele ConcreteStrategyQueue și ConcreteStrategyTime, clasa View care conține implementarea interfeței, clasa SimulationManager care controlează întreaga simulare și clasa pentru excepții WrongInputException.

**Clasa Task**

Această clasă modelează obiectul principal de lucru în această aplicație. Obiectul de tip Task poate fi asemănat din lumea reală cu procesele care urmează să fie executate de procesor, clienții unui magazin care așteaptă la coadă etc. Variabilele instanță ale clasei sunt trei variabile de tip int: un ID, arrivalTime, timpul la care clientul e pregătit să se pună la coadă și processingTime, timpul petrecut de client in fața cozii pentru a fi servit. Metodele importante din această clasă sunt: metoda compareTo care e folosită în SimulationManager în funcția de creare random a task-urilor, metoda toString care returnează o formă vizuală user-friendly a unui obiect de tip Task.

**Clasa Server**

Clasa Server este reprezentarea unei cozi, structuri ce vor fi folosite pentru a face managementul obiectelor de tip Task. Variabilele instanță ale acestei clase sunt: un BlockingQueue de obiecte de tip Task și un AtomicInteger waitingPeriod care stochează timpul total de așteptare în cadrul cozii. Aceste două structuri au fost alese în detrimentul tipului ArrayList și int pentru că oferă conceptul de Thread safety. O metodă importantă din cadrul clasei este metoda addTask prin care se adaugă un task în listă și se crește timpul de așteptare al cozii cu timpul de procesare al task-ului. O altă metodă importantă este toString() care oferă o variantă afișabila a cozii și a listei de task-uri. Cea mai importantă metodă din cadrul clasei este metoda run() care este folosită de thread-ul care conține server-ul actual. Această metodă ia elementul din vârful cozii (fără să îl șteargă), se pune pauză execuției thread-ului pentru o secundă, iar dacă task-ul există, i se decrementează timpul de procesare și se decrementează și timpul de așteptare în coadă. Dacă task-ul a fost procesat complet, este extras din coadă.

**Clasa Scheduler**

Clasa Scheduler este planificatorul simulării și clasa care creează thread-urile pentru fiecare Server (coadă). Variabilele instanță sunt: o listă de Servere (inițializată în constructor), un obiect de tipul Strategy și numarul de servere. În cadrul constructorului, se inițializează lista de servere, se construiesc serverele și li se creează câte un thread. Metoda changeStrategy setează strategia de selecție a cozilor de către task-uri, în funcție de parametrul primit (de tipul SelectionPolicy). Metoda dispachTask alege coada potrivită pentru fiecare task în funcție de strategia aleasă.

**Interfața Strategy**

Această interfață se folosește pentru a facilita procesul de alegere a politicii de selecție a cozii. Conține o metoda addTask neimplementată, care urmează a fi implementă de clasele concrete ConcreteStrategyQueue și ConcreteStrategyTime. Enum-ul SelectionPolicy conține constante care reprezintă politicile de selecție. Aceste sunt folosite în clasa Scheduler.

**Clasa ConcreteStrategyQueue**

Această clasă implementează interfața Strategy și anume implementează metoda addTask. Această metodă caută coada cu numărul cel mai mic de clienți și i se adauga task-ul.

**Clasa ConcreteStrategyTime**

Și această clasă implementează interfața Strategy și anume implementează metoda addTask. Această metodă caută coada cu cel mai mic timp de așteptare și i se adauga task-ul. Aceasta va fi politica de selecție a cozii folosită în aplicație.

**Clasa Controller**

Clasa Controller este cea care implementează actionListener pentru butoanele aplicației și citește datele furnizate de utilizator prin intermediul interfeței. Acest lucru se face prin funcția readDataInput care, la citire, verifică dacă s-au introdus greșit, lipsesc datele, sau orice fel de excepție și le aruncă. Acestea vor fi prinse în metoda actionListener. În această metodă, după citirea și parsarea datelor, se creează un obiect de tipul SimulationManager și se pornește un thread pe acest obiect (aici începe simularea).

**Clasa SimulationManager**

Această clasă este cea mai importantă pentru aplicație deoarece realizează întreg managementul simulării. Aici este inițializat Scheduler-ul, este aleasă politica de selecție a cozii de către task-uri, se generează task-urile aleator. Clasa are ca variabile instanță un obiect de tip Scheduler, unul de tip SelectionPolicy, un obiect de tip View (interfața), o lista de task-uri și timpul de simulare.

Metoda generateRandomTask are rolul de a genera task-uri în funcție de valorile date de utilizator. Cea mai importantă metoda este metoda run(). În această metodă se iau task-urile care au arrivalTime egal cu timpul curent de simulare și apelează funcția dispachTask pentru a pune task-ul în coada potrivită. Conținutul actual este scris în fișierul jurnal, se incrementează timpul curent iar thread-ul este pus pe pauză timp de o secundă. Acest lucru se repetă atât timp cât timpul actual este încă mai mic decât timpul de simulare ales.

**Clasele pentru exceptie**

Clasele WrongInputException și NoIntervalException sunt folosite la extragerea argumentelor din interfața grafică. Dacă se dau argumente incorecte, care nu se pot converti la tipul întreg, este aruncată o excepție de tipul WrongInputException. Dacă la timpii de sorire în coadă și timpii de procesare se furnizează date care nu reprezintă un interval valid, se va arunca o excepție de tipul NoIntervalException.

**Clasa View**

View este a doua componentă din pattern-ul architectural MVC, care comunică cu modelele aplicației și cu controller-ul și realizează interfața grafică cu utilizatorul. Câmpurile acestei clase sunt reprezentate de toate componentele din interfață: butoane, labels, text-fields și un text-area.

Cea mai importantă metoda este constructorul, în care practic se construiește interfața, adăugându-se componentele. Apoi metodele prin care se adaugă un actionListener pe butoane sunt importante deoarece fără ele, butoanele nu ar face nimic. Acestea fac legătura între SimulationManager, care e controller-ul aplicației, în care sunt implementate acțiunile realizate la apăsarea butoanelor din interfață. Metodele prin care se adaugă actionListener: addStartListener, addExitListenr. Se observă și metodele prin care se extrag datele introduse de utilizator în cadrul interfeței.

**Interfața Grafică cu Utilizatorul**

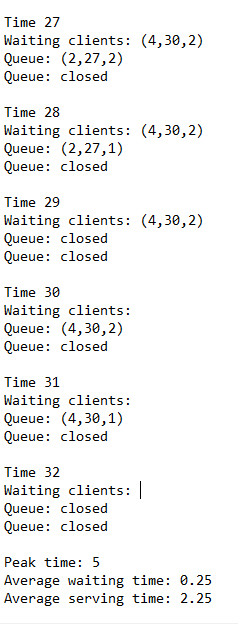
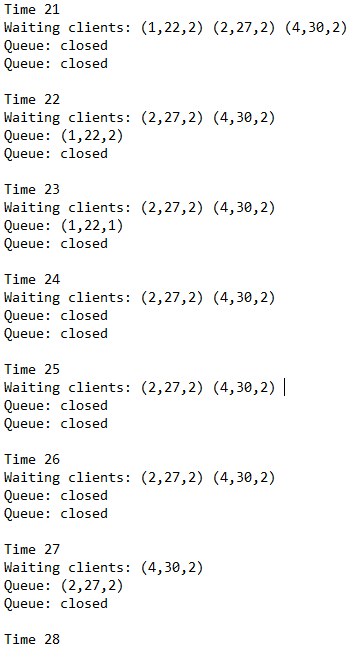
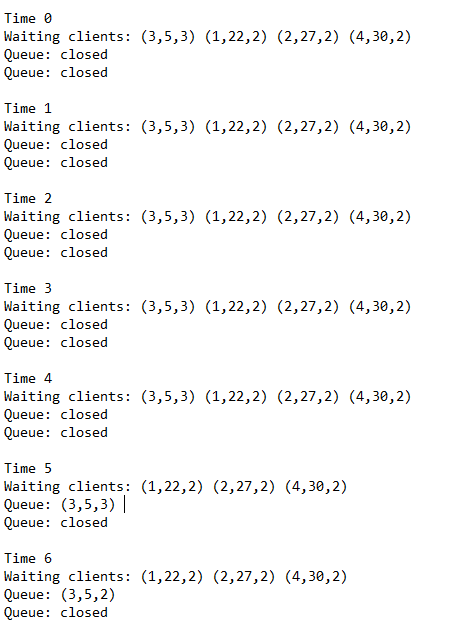
Pentru realizarea interfeței am folosit Java Swing. Clasa care o implementează este clasa View care extinde JFrame. Panoul principal este de tip BoxLayout, aranjate pe axa y deoarece este compus din mai multe elemente, în vederea unei aranjări optime și atractive. Panoul principal (numit content) este format practic din mai multe panouri și porțiuni goale (rigid areas) pentru a da un aspect mai bun aplicației. Panoul principal conține numele aplicației, centrat și mărit, toate panourile care conțin label-urile și text-field-urile, panoul care conține butoanele, iar apoi text-area care conține cozile în timp real.

Am adăugat labels, precum: „Introduce the number of queues”, „Simulation interval”, „Arrival time bounds”. În dreptul fiecărui label am pus câte un textfield, unde utilizatorul introduce numărul de cozi, numărul de clienți, timpul de simulare și intervalele pentru arrivalTime și processingTime. Apoi am adăugat pe rând butoanele „Start” care la apăsare pornește simularea și butonul „Exit” prin care se închide aplicația. Am adăugat și un text area în care se afișează în timp real conținutul cozilor.

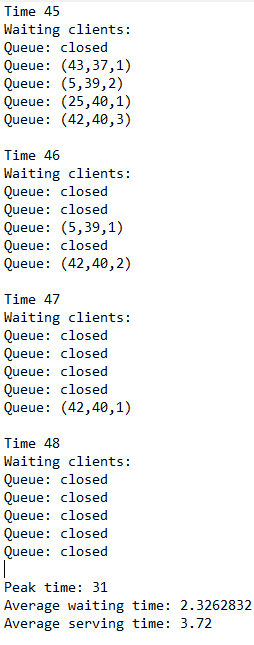
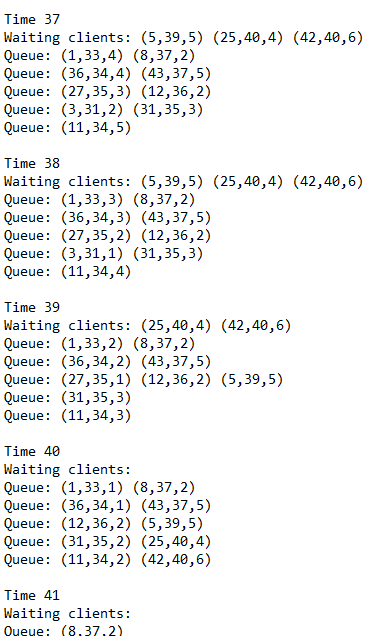
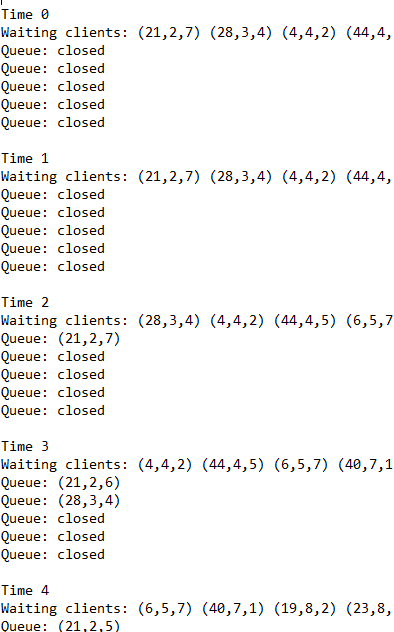
La introducerea greșită a datelor pot apărea mai multe mesaje de informare. Dacă se introduc caractere care nu se pot parsa la tipul întreg, se va afișa un mesaj de eroare de tipul WrongInputException. Dacă se introduce un interval care nu e valid, se va afișa un mesaj corespunzător.

# Testare

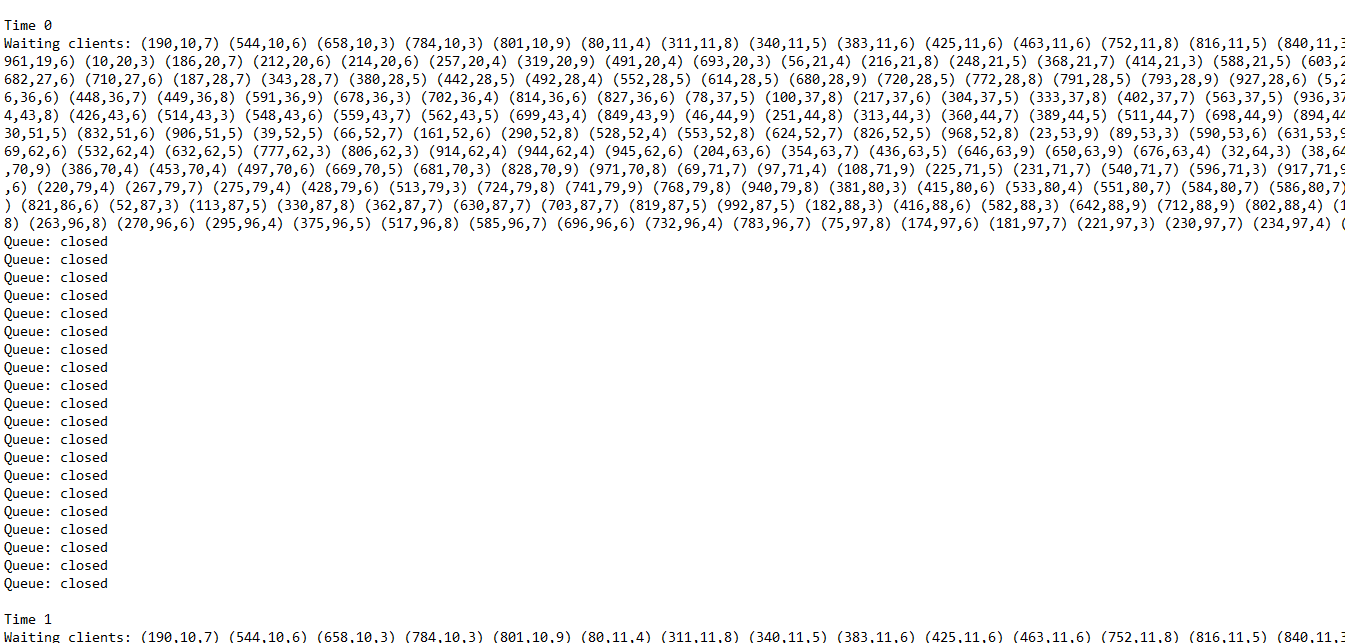
Pentru primul set de teste, rezultatele sunt următoarele:

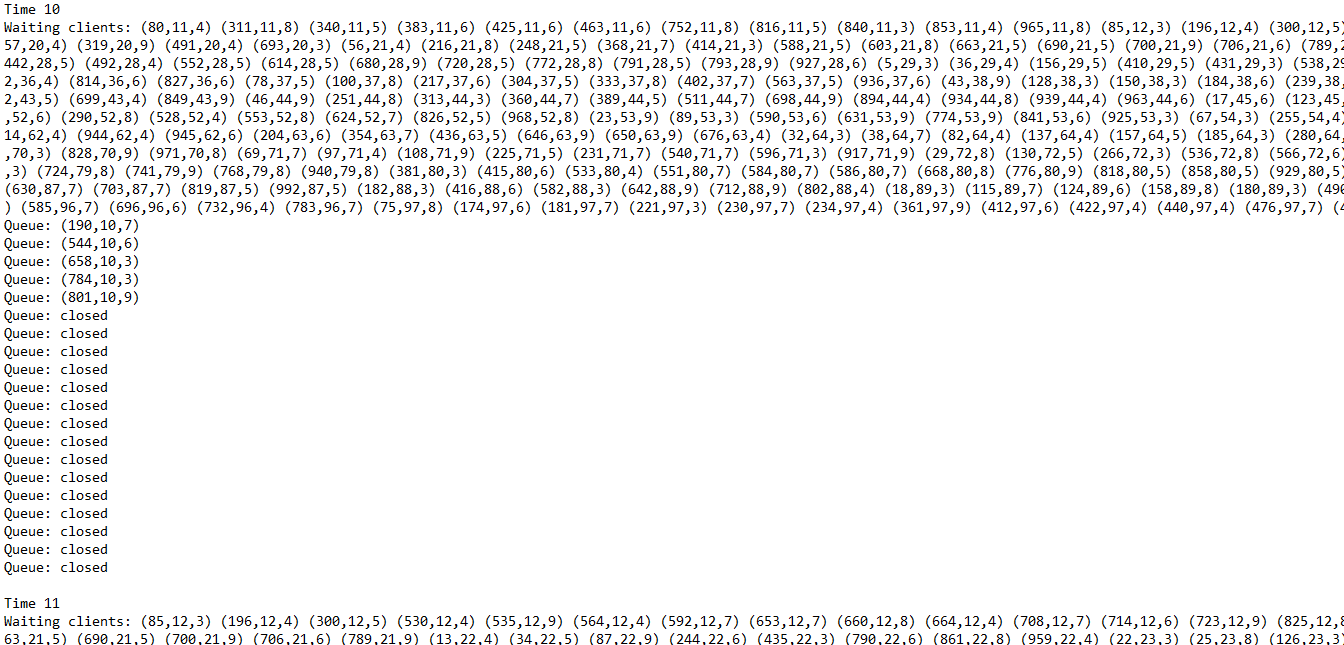


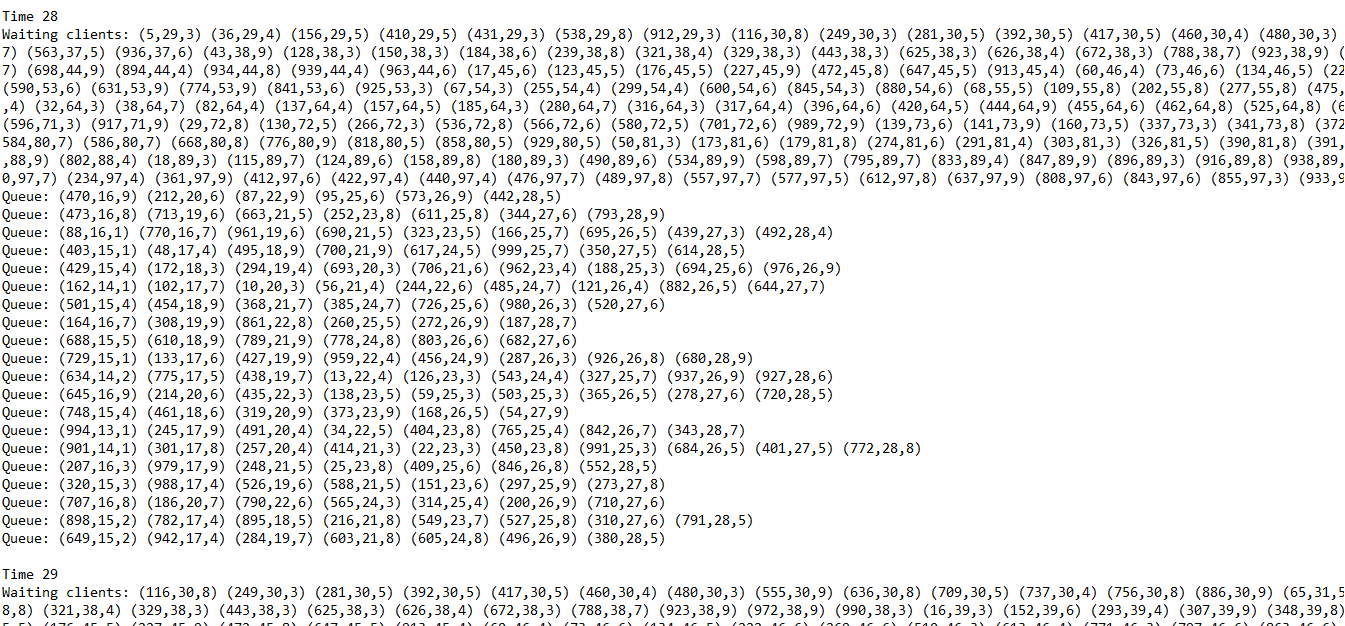
Pentru al doilea set de teste, rezultatele sunt:

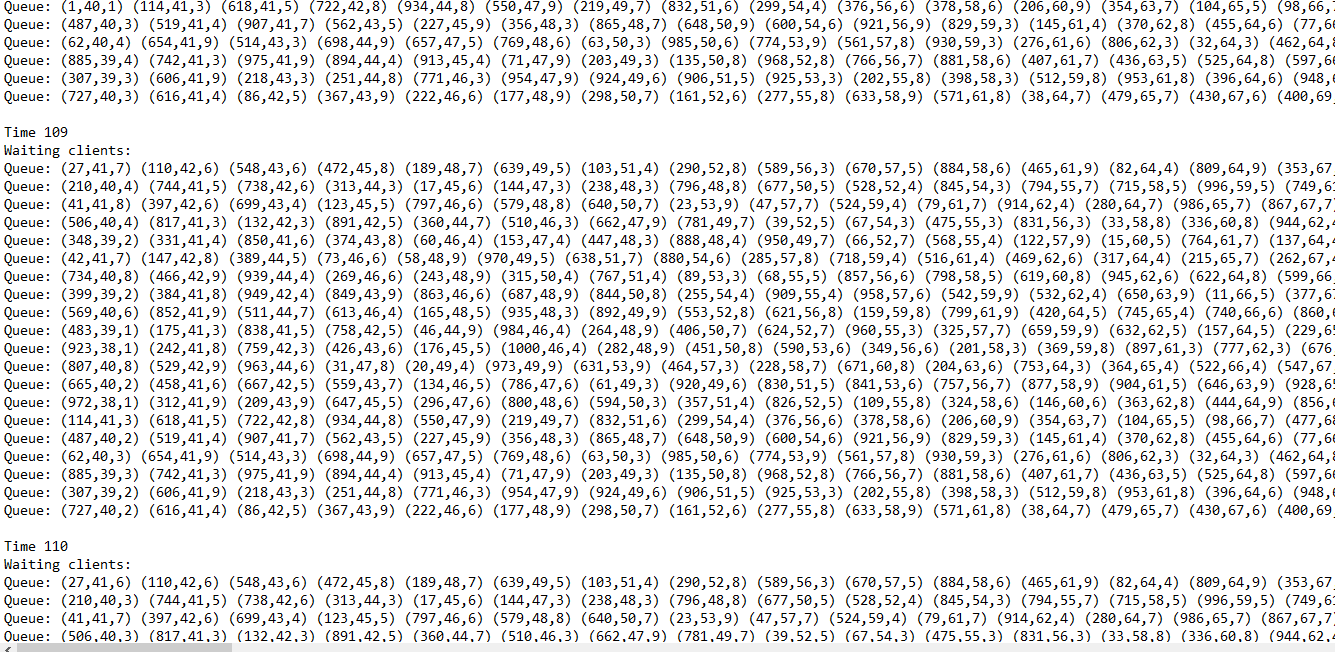


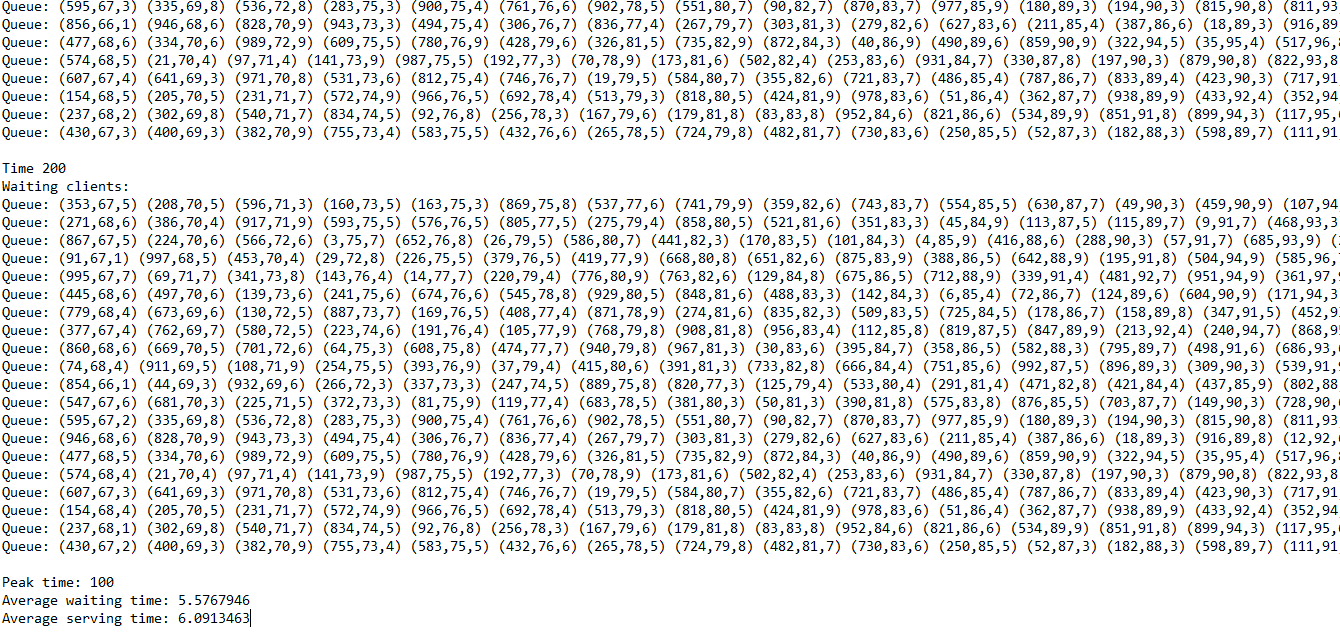
Pentru al treilea set de teste:











# Concluzii si Dezvoltări Ulterioare

În concluzie, înainte de implementarea propriu zisă a unei aplicații, e nevoie de o analiză în detaliu și de luarea unor decizii cu privire la structură și la modelul ales, astfel încât să fie cel mai potrivit cerințelor. Acest lucru conferă o implementare mai ușoară, mai logică, mai organizată și cu mai puțin efort. Poate fi un avantaj și pentru client/utilizator dacă aplicația e mai intuitivă, organizată și atractivă.

Ca dezvoltare ulterioară, aplicația poate să îi ceară utilizatorului (prin intermediul interfeței) politica de selecție a cozii de către clienți, adică se poate selecta dacă politica să fie în funcție de timpul de așteptare sau în funcție de numărul de clienți din coadă.

# Bibliografie

1. <http://tynerblain.com/blog/2007/04/09/sample-use-case-example/>
2. https://lucid.app/
3. ASSIGNMENT\_2\_SUPPORT\_PRESENTATION.pdf