**FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE**

**DEPARTAMENTUL CALCULATOARE**

**TEMA 2**

**Simulator de cozi**

**Documentație**

**Nume: Litu Saviana**

**Grupa: 30221**

**An academic 2020 – 2021**

**Cuprins**

[**Obiectivul temei** 3](#_Toc68976022)

[**Analiza problemei, modelare, scenarii, cazuri de utilizare** 4](#_Toc68976023)

[**Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfețe, relații, packages, algoritmi, interfața utilizator)** 5](#_Toc68976024)

[**Implementare** 9](#_Toc68976025)

[**Rezultate** 17](#_Toc68976026)

[**Concluzii** 20](#_Toc68976027)

[**Bibliografie** 20](#_Toc68976028)

# **Obiectivul temei**

Obiectivul acestei teme este de a propune și implementa o aplicație Java ce are scopul de a analiza sistemele bazate pe cozi de așteptare, în vederea determinării și minimizării timpului de așteptare al clienților.

Pentru a realiza acest obiectiv principal, definim următoarele obiective secundare:

|  |  |  |
| --- | --- | --- |
| **Obiectiv** | **Descriere** | **Capitol** |
| Analiza problemei | Se analizează enunțul problemei și se dezvoltă un design inițial | **2** |
| Alegerea structurilor de date potrivite | Se aleg structuri de date care să faciliteze sincronizarea thread-urilor | **3** |
| Implementarea | Se vor crea clase în concordanță cu design-ul dorit | **3** |
| Proiectarea algoritmilor | Se vor scrie algoritmi pentru a calcula ora de vârf, timpul mediu al unui serviciu, timpul mediu de așteptare | **3** |
| Realizarea interfeței grafice | Se va realiza interfața utilizator utilizând JavaFX | **4** |
| Testarea | Se vor efectua teste pentru diferite input-uri | **4** |

# **Analiza problemei, modelare, scenarii, cazuri de utilizare**

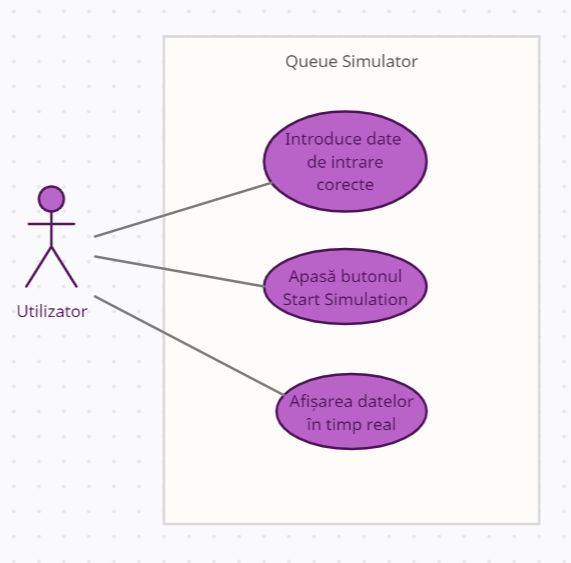
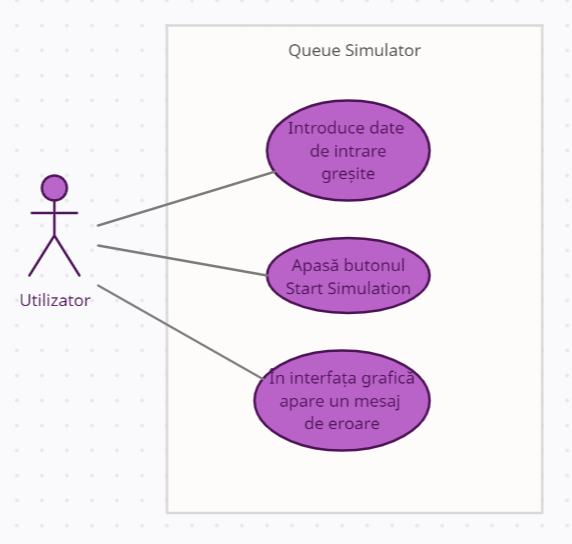
Cozile sunt des folosite în domenii din lumea reală. Principalul lor obiectiv este acela de a asigura un loc unde un ”client” poate aștepta să i se ofere un ”serviciu”. Management-ul sistemelor bazate pe cozi de așteptare vizează minimizarea timpului pe care un ”client” trebuie să îl aștepte până când este servit. O modalitate de a minimiza acest timp de așteptare este cea de a adauga mai multe servere, adică mai multe cozi în sistem, dar această abordare crește costurile furnizorului de servicii.

Analizând enunțul temei propuse, observăm că trebuie să generăm N clienți care așteaptă să fie serviți în Q cozi de așteptare, sunt serviți, apoi părăsesc cozile. Totul se întâmplă într-un interval de timp definit prin variabila tsimulation. Un client este caracterizat de parametrii ID, timpul la care sunt pregătiți să intre în coada de așteptare și intervalul de timp necesar servirii clientului.

Datele de intrare necesare sunt: numărul de clienți, numărul de cozi, intervalul de timp alocat simulării, intervalul în care se poate încadra timpul la care clientul ajunge la coadă și intervalul de timp în care clientul poate fi servit.

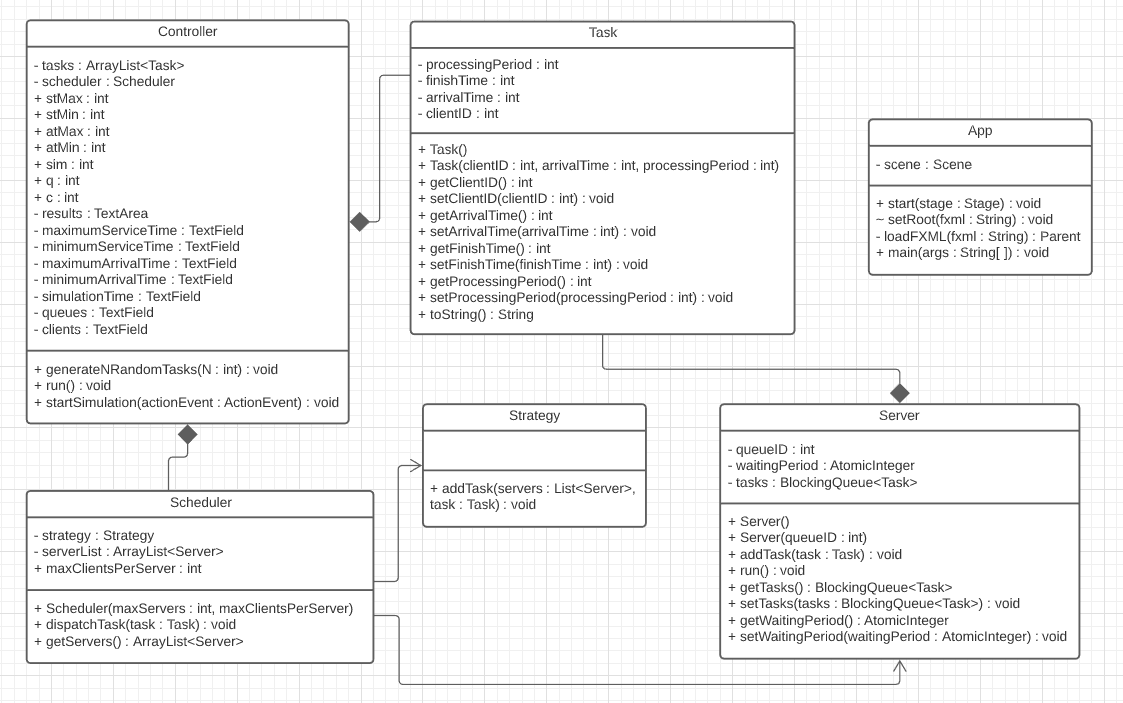
Așadar, trecem la partea de modelare. Voi folosi clasele: Task – corespunde clientului, Server – corespunde cozii, Strategy – clasă în care implementez adăugarea unui client în coadă pe baza timpului minim de așteptare, Scheduler – clasă în care pornesc câte un thread pentru fiecare coadă creată și Controller – clasă în care generez clienții, îi așez în cozi, calculez ora de vârf, timpul mediu al unui serviciu și timpul mediu de așteptare și fac conexiunea cu interfața grafică.

În continuare, voi prezenta cazul de utilizare ideal și un caz de utilizare alternativ, care apare în momentul în care utilizatorul introduce date greșite (de exemplu: introduce litere în loc de numere întregi în oricare dintre field-uri).

1. Caz ideal
2. Utilizatorul introduce de la tastatură datele de intrare corecte, adică numere întregi, cu minArrivalTime < maxArrivalTime, minServiceTime < maxServiceTime, maxArrivalTime < simulationTime, maxServiceTime < simulationtime.
3. Utilizatorul apasă butonul Start Simulation.
4. Rezultatele sunt afișate în timp real în interfața grafică și scrise într-un fișier text.
5. Caz alternativ (de eroare)
6. Utilizatorul introduce de la tastatură date care nu sunt numere întregi sau care nu respectă condițiile impuse anterior.
7. Utilizatorul apasă butonul Start Simulation.
8. În interfața grafică apare un mesaj de eroare.
9. Utilizatorul introduce noi date de intrare de la tastatură.

# **Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfețe, relații, packages, algoritmi, interfața utilizator)**

* **Diagrama UML**



* **Structuri de date**

Structurile de date folosite sunt:

* Int
* Double
* ArrayList
* AtomicInteger
* BlockingQueue
* TextField și TextArea – specifice JavaFX, se referă la field-urile în care se introduc datele și în care se afișeaza output-ul

Tipurile de date AtomicInteger și BlockingQueue sunt folosite pentru a asigura sincronizarea thread-urilor. BlockingQueue este o clasă Java, care reprezintă o coadă folosită pentru ca thread-urile să efectueze operații de adăugare și ștergere de elemente asupra ei. AtomicInteger este o clasă care facilitează incrementarea unui număr în mod concurent, din mai multe thread-uri, într-un mod sigur, fără a sincroniza accesul la variabilă. O operație atomică este o operație care poate fi executată în parallel de mai multe thread-uri, fără a folosi cuvântul cheie ”synchronized” sau alte metode de sincronizare.

* **Proiectarea claselor**

*Clasa App*: clasă în care se află metoda main, deci din care se rulează aplicația.

*Clasa Task*: clasă de bază a acestui program, care se referă la client și are următoarele atribute: clientID, arrivalTime, finishTime și processingPeriod. Primul reprezintă id-ul atribuit fiecărui client generat, arrivalTime este timpul la care clientul este pregătit să intre în coadă, finishTime este timpul la care clientul părăsește coada, iar processingPeriod se referă la timpul alocat servirii clientului. Această clasă cuprinde doi constructori, gettere și settere pentru toate atributele și o metodă toString pentru afișarea clienților.

*Clasa Server*: clasă principală, care se referă la coadă și are următoarele atribute: queueID, waitingPeriod(AtomicInteger) și tasks(BlockingQueue<Task>). Primul este id-ul atribuit fiecărei cozi. WaitingPeriod și tasks sunt obiecte ale claselor AtomicInteger, respectiv BlockingQueue, folosite pentru că fiecărui server îi corespunde un thread, iar aceste două clase ajută la sincronizarea thread-urilor. Această clasă cuprinde doi constructori, gettere și settere pentru cele trei atribute și metodele addTask, run(deoarece Server implementează interfața Runnable) și toString pentru afișarea cozilor.

*Clasa Strategy*: clasă în care implementez doar metoda pentur adăugarea unui client într-o coadă în funcție de timpul minim de așteptare.

*Clasa Scheduler*: clasă în care creez și pornesc câte un thread pentru fiecare coadă creată și are atributele maxClientsPerServer, serverList și strategy. Primul va lua valoarea numărului de clienți introdus din interfața grafică și va reprezenta mărimea unui server. ServerList este lista de servere cu care lucrez, iar strategy este un obiect al clasei Strategy, folosit pentru a-i accesa metoda addTask. Această clasă conține un constructor unde inițializez datele, inclusiv thread-urile, o metoda dispatchTask, unde apelez metoda addTask din clasa Strategy și metoda getServers.

*Clasa Controller*: clasă de unde se face management-ul simulării, dar și legătura cu interfața grafică. Aici folosesc atribute de tipul TextField si TextArea pentru field-urile din interfață, dar și atribute de tip întreg care corespund acestor field-uri, pentru a stoca datele introduse. De asemenea, folosesc și un obiect din clasa Scheduler și o lista de task-uri. Metodele acestei clase sunt: generateNRandomTasks, run și startSimulation. Prima este metoda în care generez clienții aleator, cu arrivalTime și processingPeriod obținute folosind un obiect din clasa Random. Folosesc metoda run deoarece Controller implementează interfața Runnable, folosind thread-uri. În aceasta apelez metoda de generare a clienților, calculez peakHour, averageServiceTime și averageWaitingTime, adaug în coadă un clienții, afișez datele în interfața grafică în timp real, dar le scriu și în fișiere text, folosind obiecte din clasele FileWriter și BufferedWriter. Metoda startSimulation este apelată atunci când în interfață se apasă butonul Start Simulation. Aceasta extrage datele introduse și le salveaza în variabile, totodată verificând corectitudinea lor, inițializeaza lista de task-uri și scheduler-ul, creează un nou thread și un timer pentru oprirea thread-urilor și a programului, la 10 secunde după afișarea datelor de ieșire.

* **Algoritmi**

Algoritmii folosiți sunt cei pentru calcularea orei de vârf, a timpului mediu al unui serviciu și timpul mediu de așteptare. Aceștia au fost implementați în clasa Controller, în metoda run().

* **peakHour**

În metoda run folosesc o variabilă numită currentTime, care este incrementată la fiecare pas executat până când se ajunge la timpul maxim al simulării, furnizat în interfață. Atunci când afișez fiecare coada, rețin în variabila countForServers și mărimea fiecărei cozi. Momentul în care această variabilă are valoarea maximă, peakHour ia valoarea lui currentTime. La sfârșit afișez această valoare în interfața grafică, dar și în fișierul text, alături de numărul de task-uri care se făceau la acel moment.

* **averageWaitingTime**

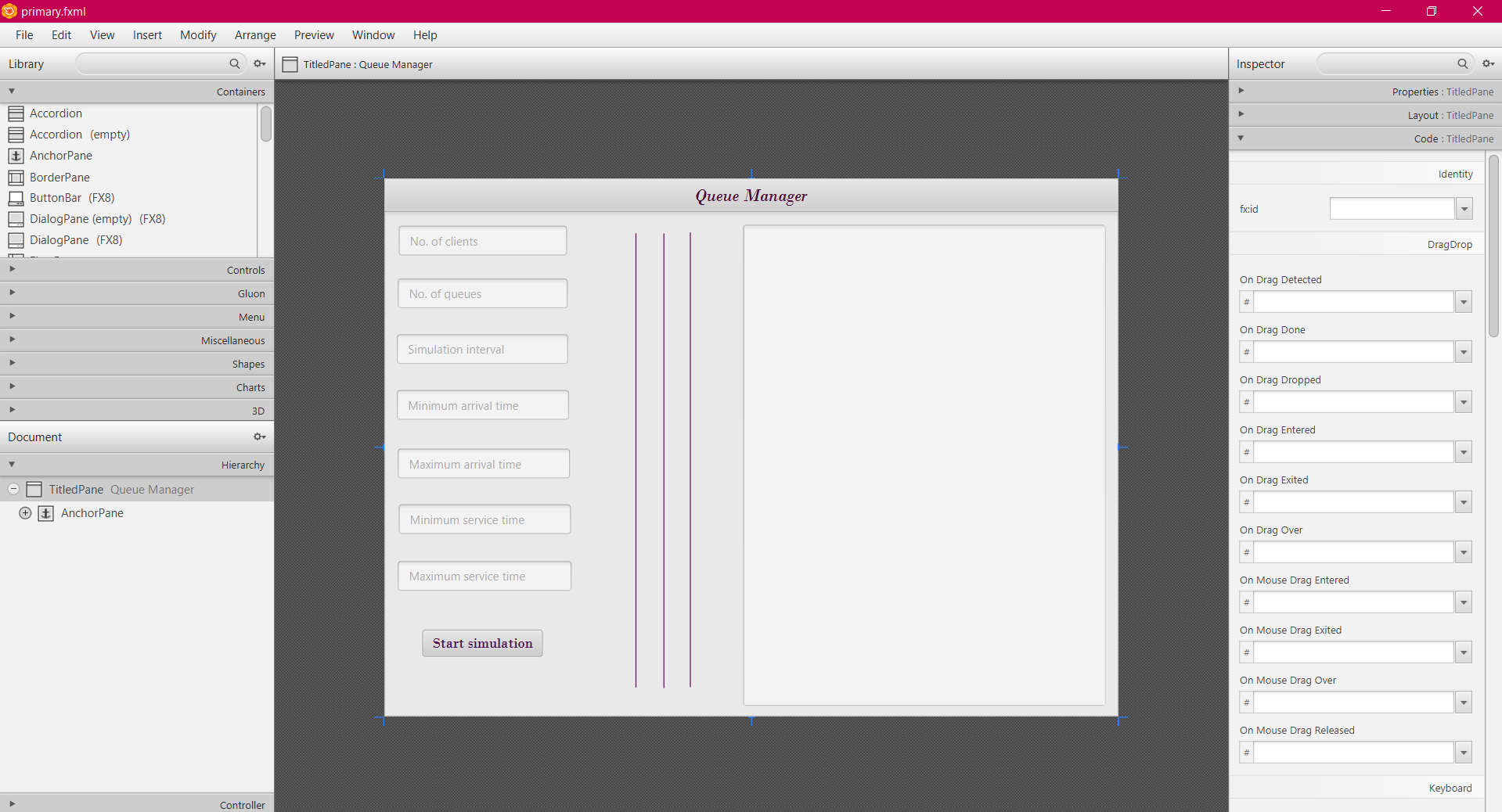
În metoda run, cât timp currentTime este mai mic decât timpul alocat simulării, atunci când adaug un client într-o coadă, adaug la variabila averageWaitingTime diferența dintre finishTime-ul și arrivalTime-ul task-ului respectiv. După ce se ajunge la timpul simulării, împart averageWaitingTime la numărul de clienți, pentru a obține valoarea căutată și o afișez atât în interfață, cât și în fișierul text.

* **averageServiceTime**

În metoda run, cât timp currentTime este mai mic decât timpul alocat simulării, atunci când adaug un client într-o coadă, adaug la variabila averageServiceTime valoarea de processingPeriod a clientului respectiv. După ce se ajunge la timpul simulării, împart averageServiceTime la numărul de clienți, pentru a obține valoarea căutată și o afișez atât în interfață, cât și în fișierul text.

* **Interfața utilizator**

Pentru a realiza interfața grafică folosind JavaFX am creat un proiect Java de tip Maven și am dezvoltat interfața incluzând dependințe specifice, apoi modelând aspectul în SceneBuilder. Am ales cadranul de bază, de tipul AnchorPane, în care să se afle toate TextField-urile (pentru datele de intrare), TextArea (pentru afișarea rezultatelor în timp real) și butonul pentru start.



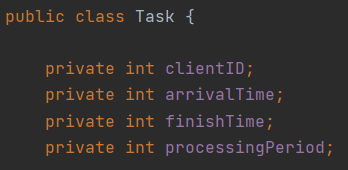
În acest screenshot se poate vedea cum arata mediul în care mi-am realizat interfața utilizator.

Fiecare TextField, dar și TextArea, au în secțiunea Code un fx:id, căruia îi dau un nume, apoi îl declar în clasa Controller, cu o adnotație @FXML.

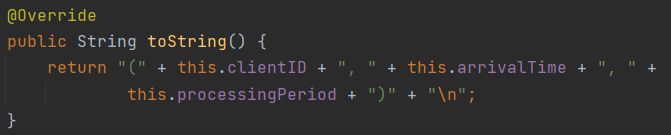
Butoanului Start simulation îi corespunde metoda startSimulation din clasa Controller, fiind precedată de adnotația @FXML. Atunci când butonul este apăsat, datele de intrare sunt stocate în variabile, pentru a putea fi folosite în restul programului, se verifică dacă acestea sunt numere întregi și verifică condițiile impuse, apoi se creează un nou thread și un timer, care să oprească programul la 10 secunde după afișarea rezultatelor pe ecran.

# **Implementare**

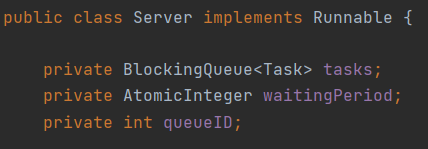
Încep prin a prezenta clasa *Task*. Acestea sunt atributele, care, conform cerinței, ajută la identificarea fiecărui client în parte:



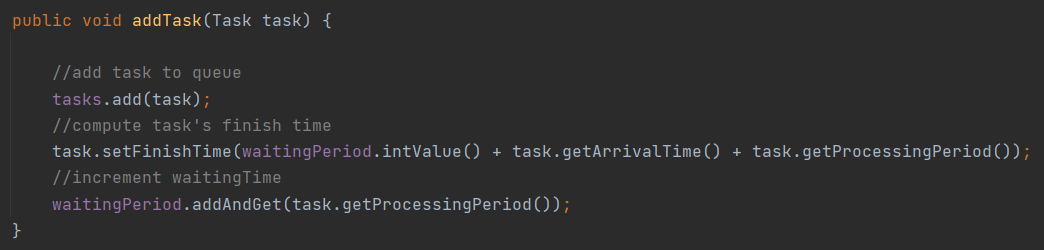
Fiecare atribut are getter și setter. Pe lângă acestea, am mai folosit și o metodă suprascrisă toString, pentru afișarea clienților:



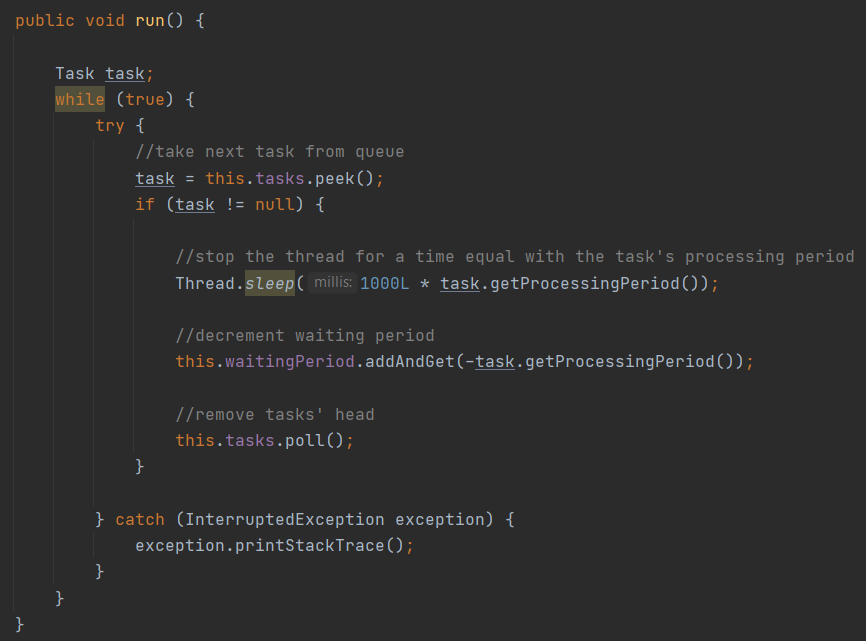
Clasa *Server* dispune de următoarele atribute, al căror rol este de a stoca id-ul fiecărui server (queueID), de a reține lista de clienți a fiecărei cozi (tasks) și de a stoca timpul de așteptare în coadă pentru fiecare client (waitingPeriod):



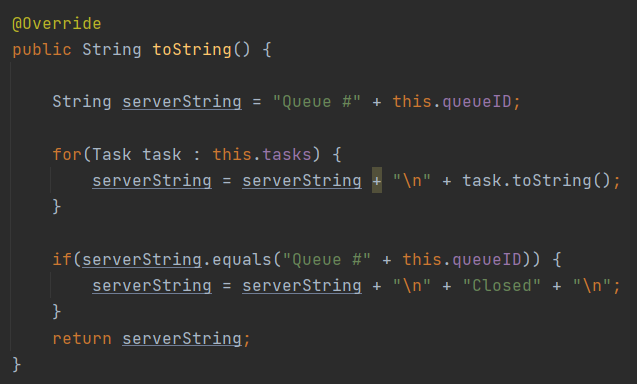
Metoda addTask, după cum spune și numele, adaugă un task în lista de task-uri, dar în același timp incrementează waitingPeriod-ul și calculează finishTime-ul pentru task-ul respectiv:



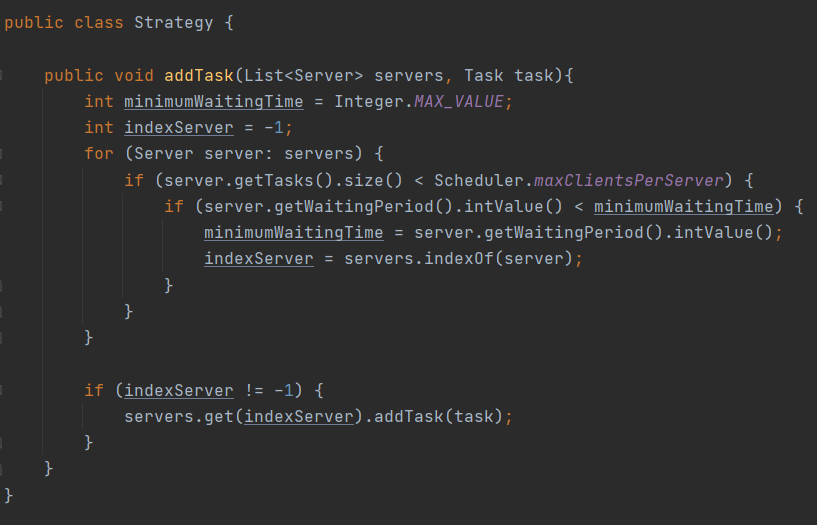
Metoda run trebuie să fie implementată deoarece clasa Server implementează interfața Runnable, pentru a utiliza thread-uri:



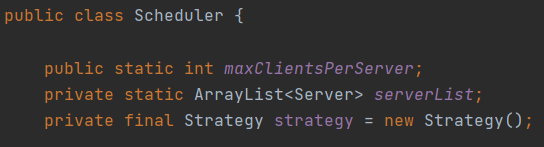
La fel ca în cazul clasei Task, am suprascris metoda toString, pentru a afișa fiecare coadă în parte, cu statusul CLOSED sau cu lista de clienți care se află la coadă, după caz:



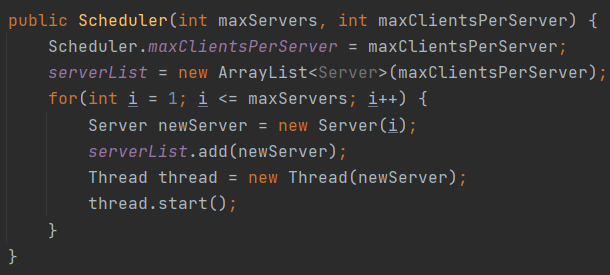
Clasa *Strategy* nu are atribute, ci doar metoda addTask, cu ajutorul căreia clienții intră în cozi după timpul minim de așteptare:



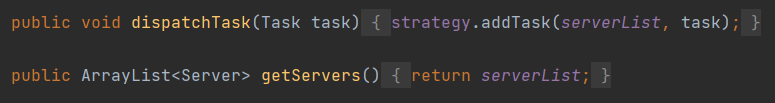
Clasa *Scheduler* are atributele maxClientsPerServer, serverList și strategy, folosite pentru a stoca numărul maxim de clienți care pot exista la o coadă (adică numărul de clienți introdus de utilizator), pentru a ține evidența unei liste de cozi, respectiv pentru a realiza adăugarea clienților în coadă conform strategiei implementate:



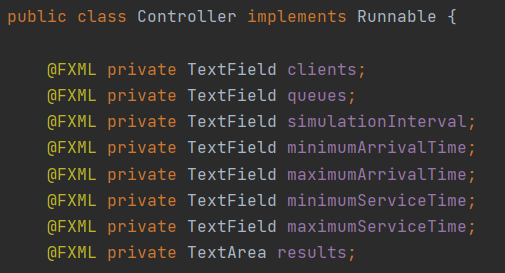
Constructorul acestei clase are rolul de a inițializa atributele, dar și de a crea și porni câte un thread pentru fiecare server:



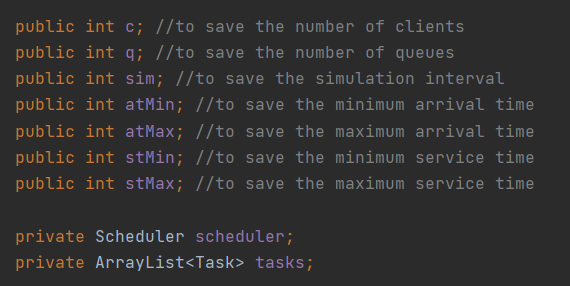
Ultimele două metode sunt folosite pentru a apela metoda addTask din clasa Scheduler, respectiv pentru a accesa lista de servere:



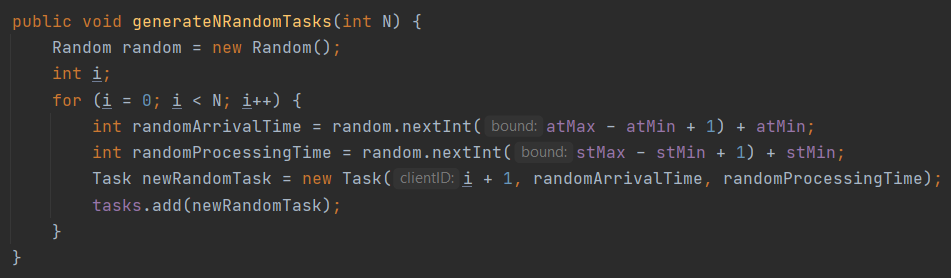
Clasa *Controller* are rolul de a face legătura între program și interfața grafică. Așadar, conține atribute specifice JavaFX, care corespund field-urilor în care se introduc datele, zonei în care se afișează rezultatele:

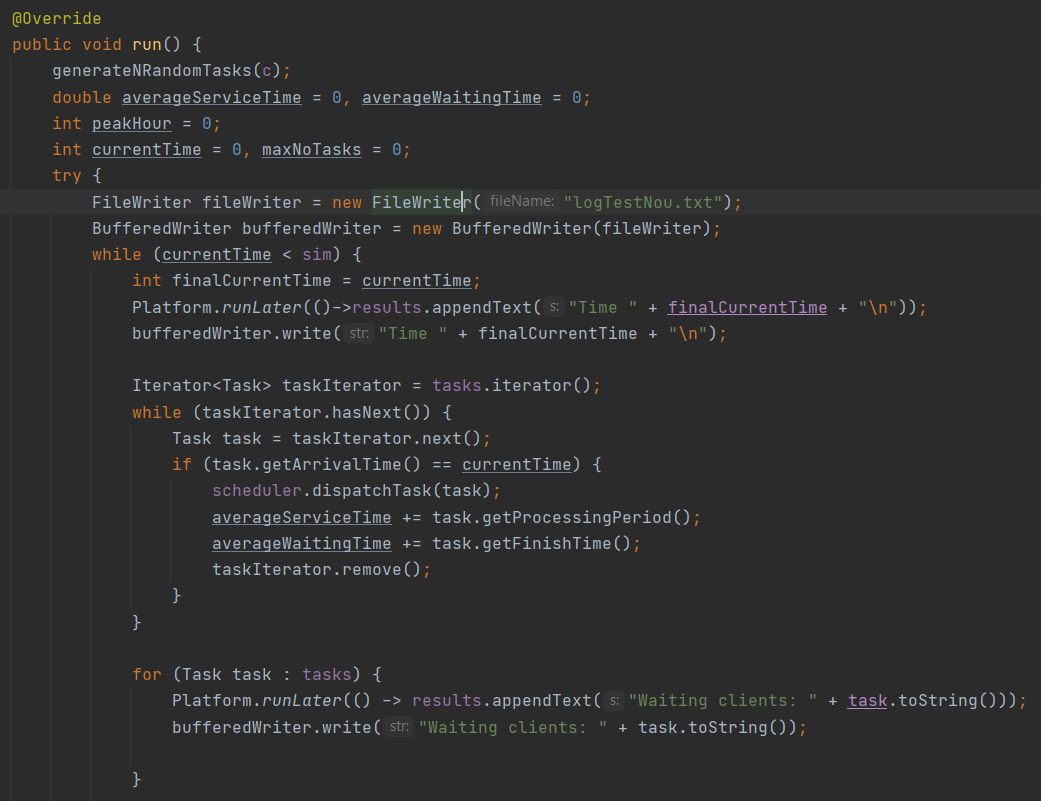


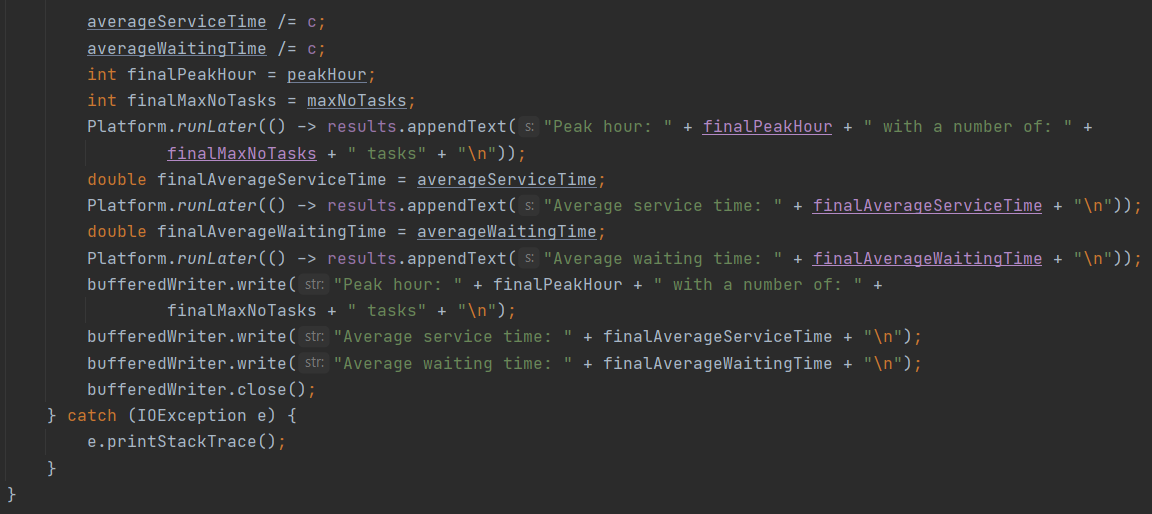
Alte atribute folosite sunt următoarele:



Metoda prin care se creează clienți este generateNRandomClients, care primește un număr întreg N, care va avea valoarea numărului de clienți introdus de utilizator. În cadrul acesteia, folosesc un obiect al clasei Random pentru a genera numere întregi random pentru arrivalTime și processingTime fiecărui client. Apoi, după generarea acestor numere, se creează un nou client care este caracterizat de ele și este adăugat în lista de clienți definită anterior (tasks):

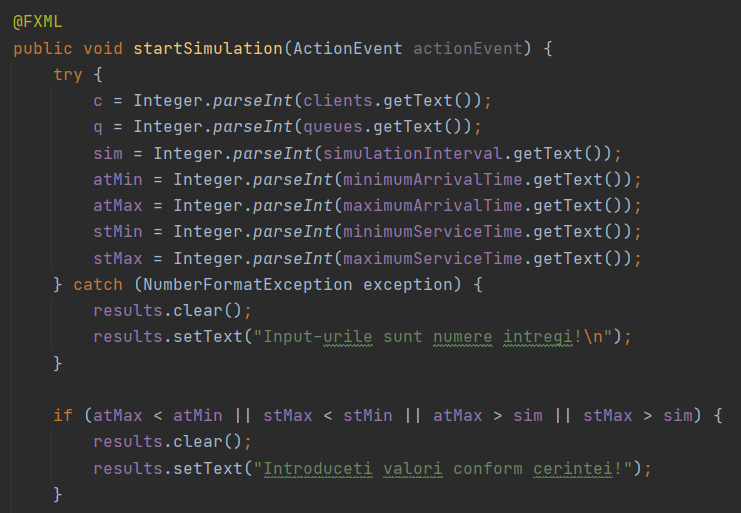


Metoda run este implementată deoarece clasa Controller implementează interfața Runnable, pentru a crea thread-uri. În cadrul ei se apelează metoda de generare a clienților, se adaugă clienții în cozi, se calculeaza peakHour, averageServiceTime și averageWaitingTime și se scriu rezultatele în interfața grafică și în fișierul text.





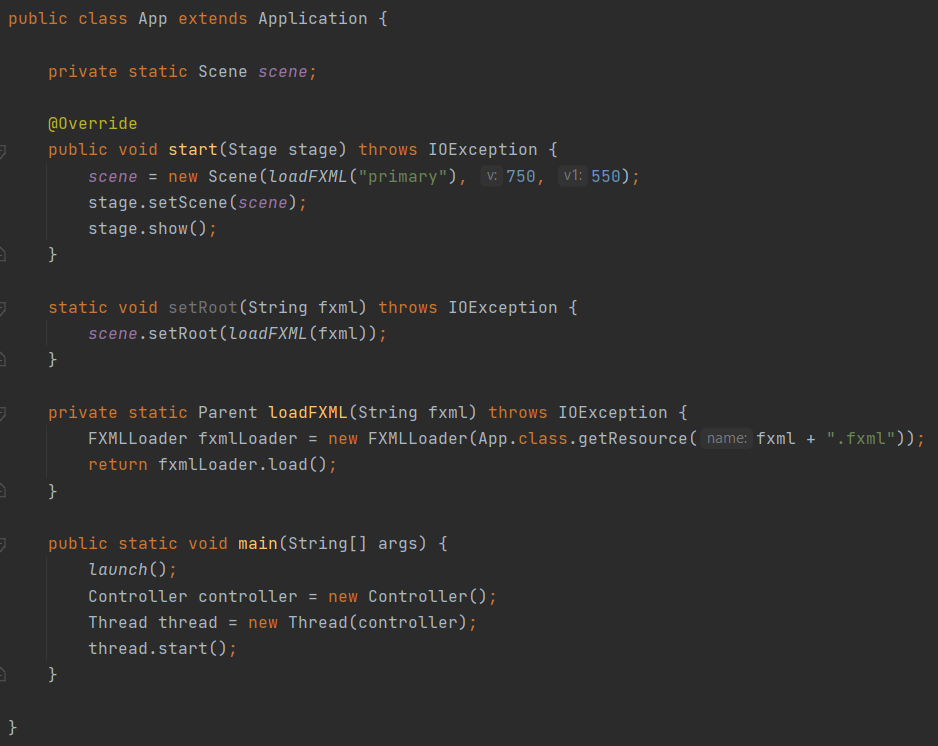
Ultima metodă este startSimulation, precedată de o adnotație @FXML, deoarece ea reprezintă ”răspunsul” la apăsarea butonului Start simulation din interfața grafică. În cadrul ei se salvează datele introduse de utilizator în variabile întregi și se verifică corectitudinea lor.



Apoi se inițializează lista de task-uri și scheduler-ul pe care le-am declarat la începutul clasei, se creează un nou thread și un timer pentru a opri execuția programului, la 10 secunde după terminarea timpului alocat simulării:



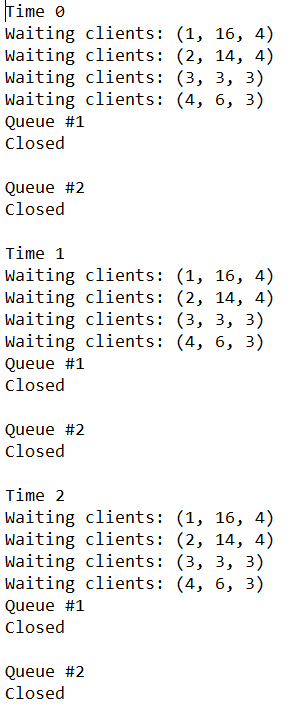
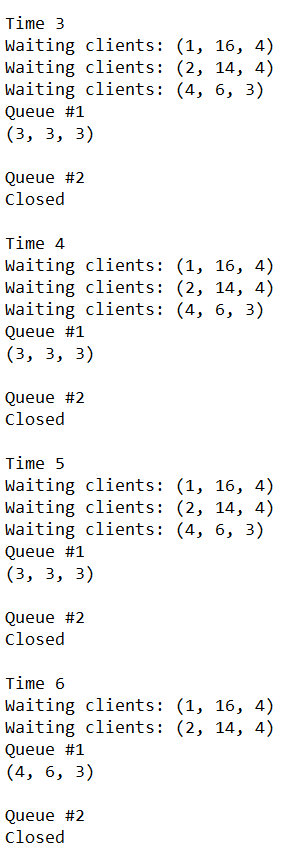
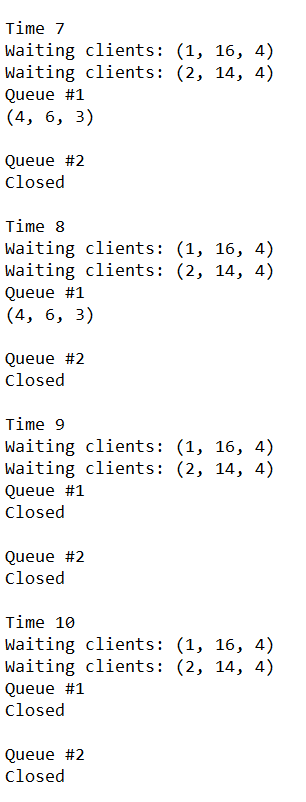
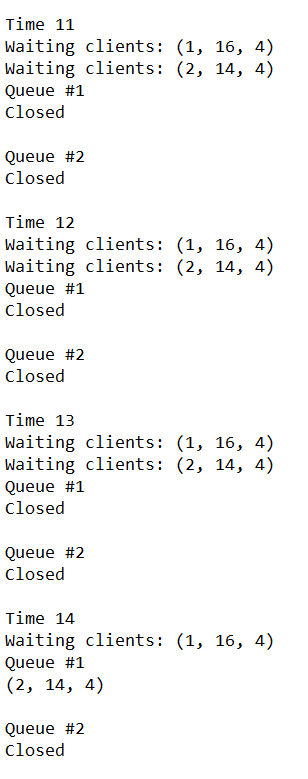
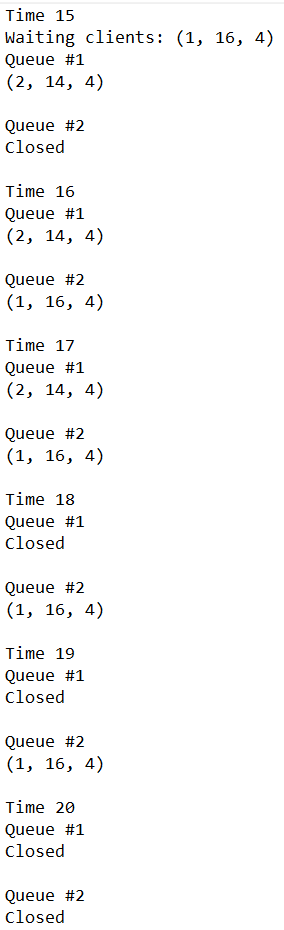
Ultima clasă este clasa App, care este Main-ul acestui program. Aceasta a fost creată automat atunci când mi-am setat dependințele pentru proiect, fiind de tip Maven, iar de aici rulez aplicația:

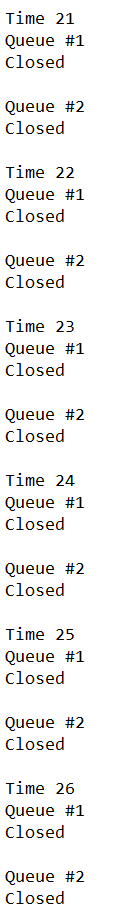
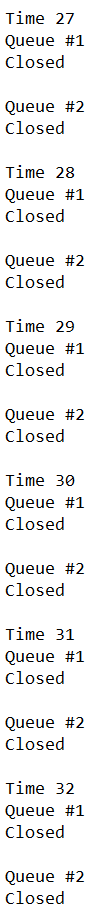
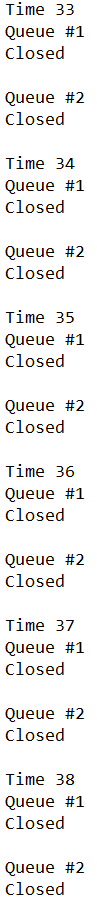
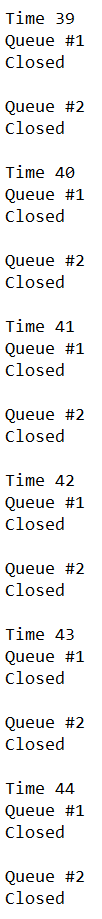
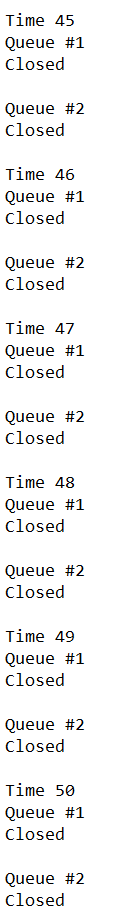
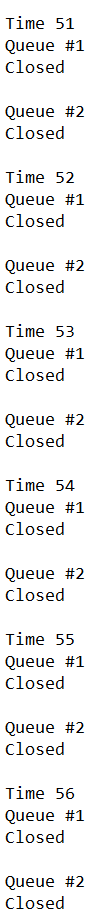
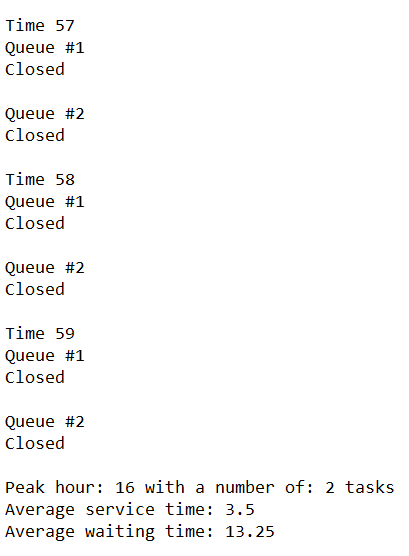


# **Rezultate**

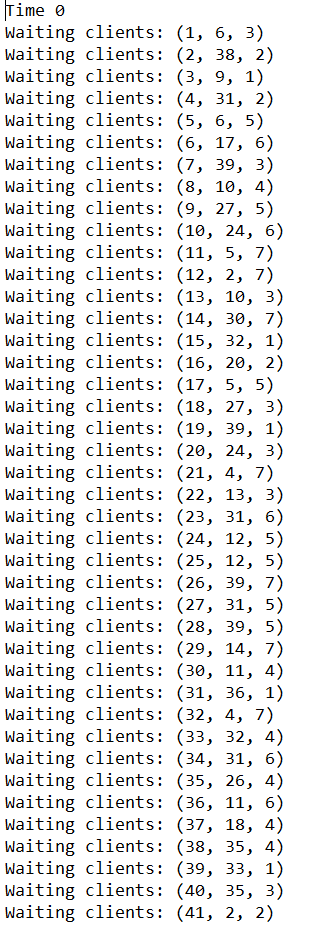
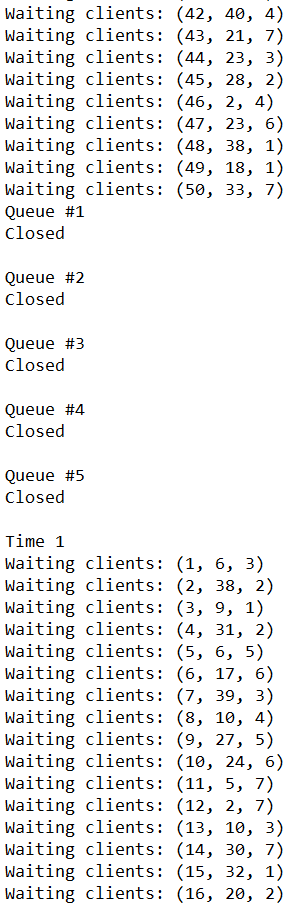
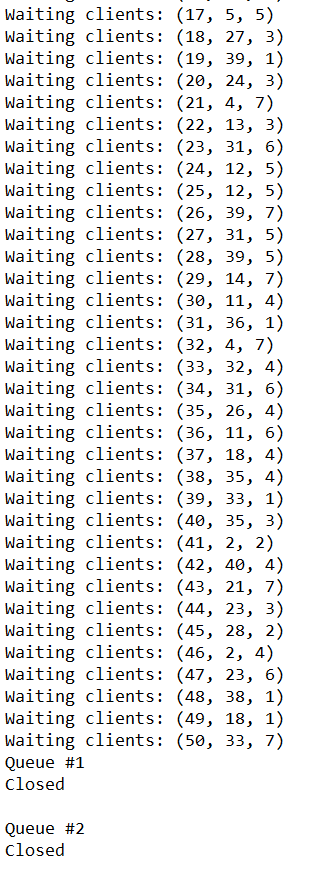
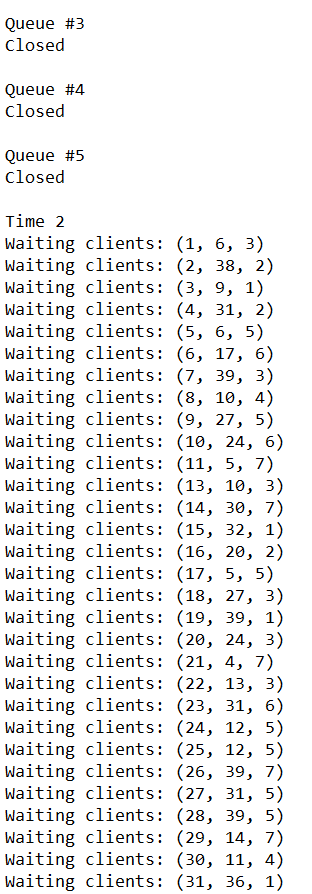
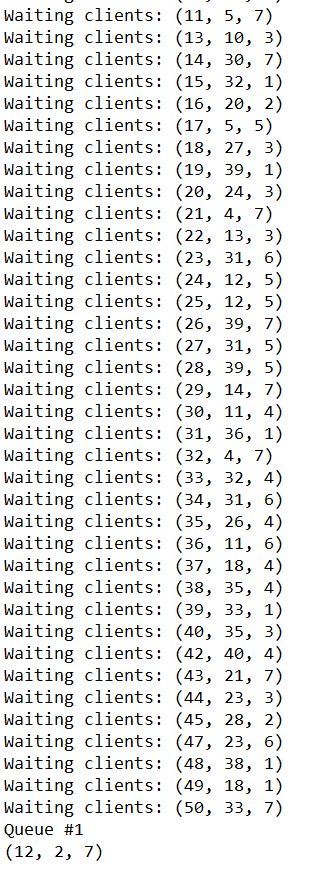
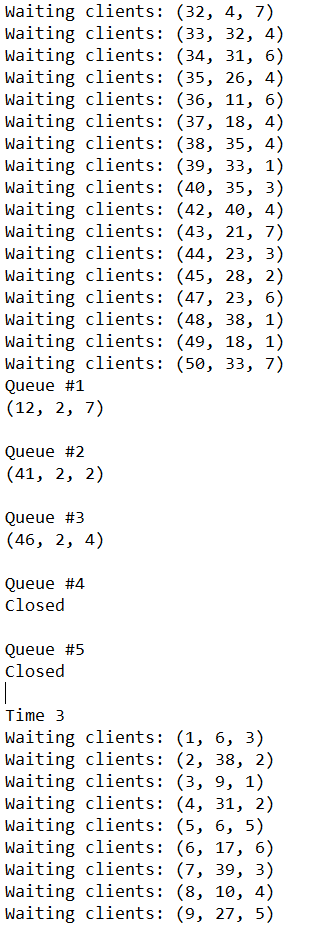
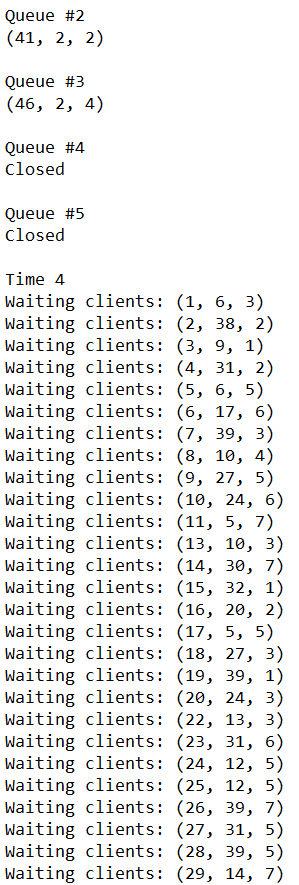
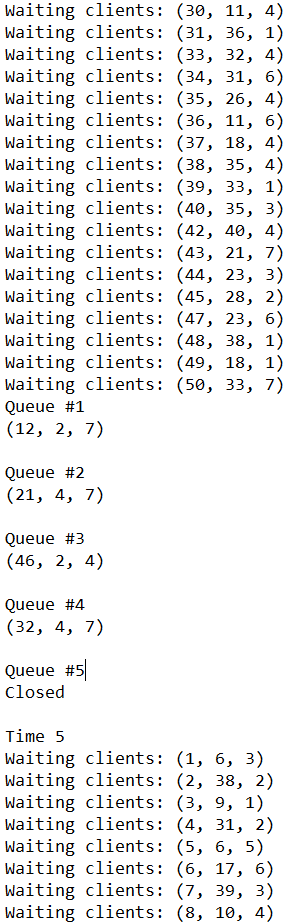
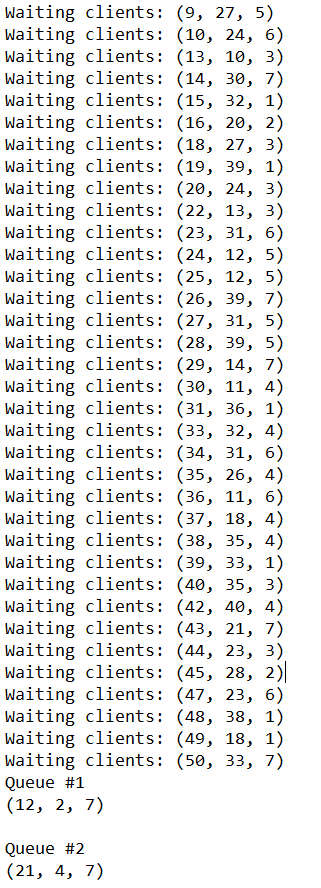
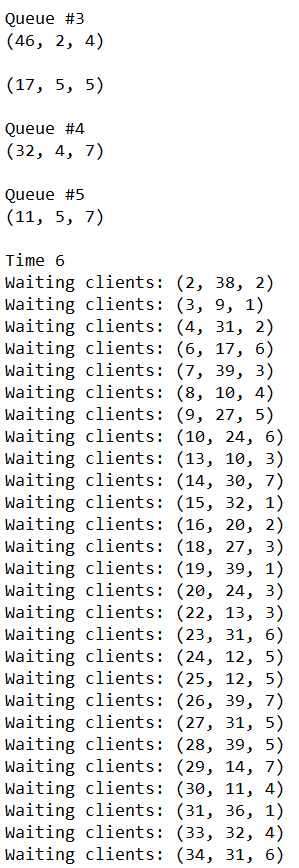
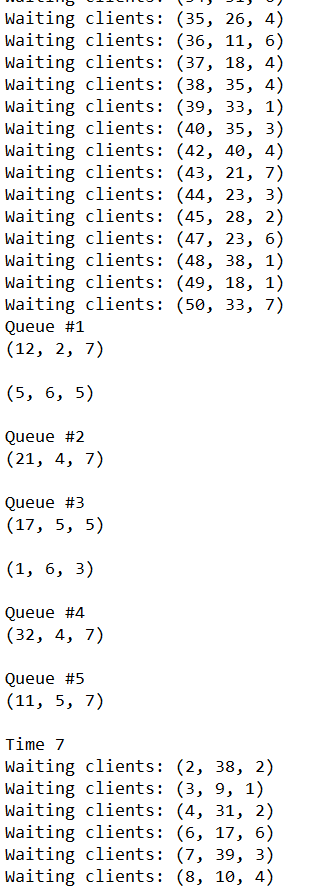
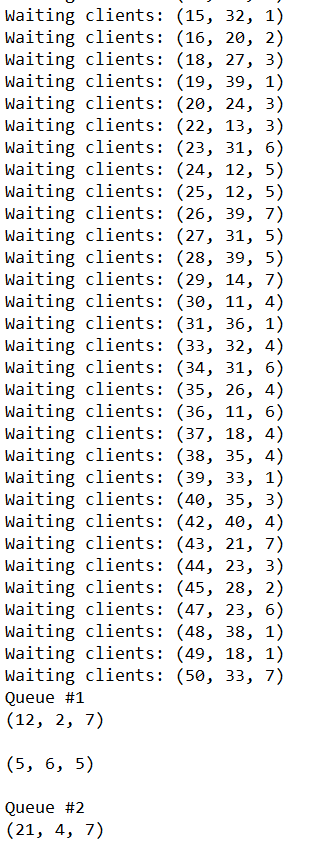
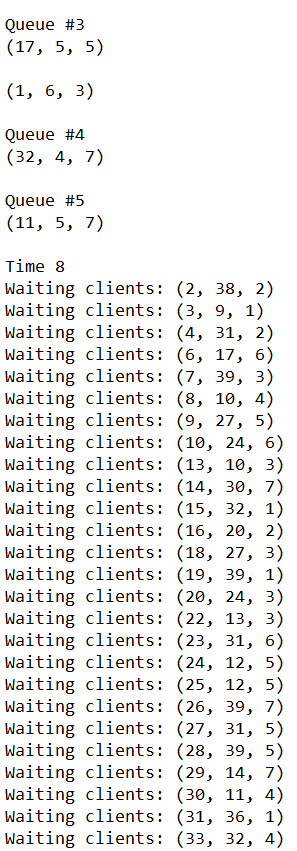
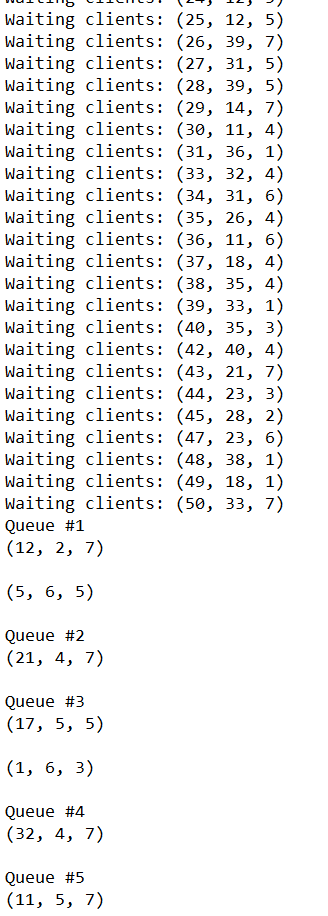
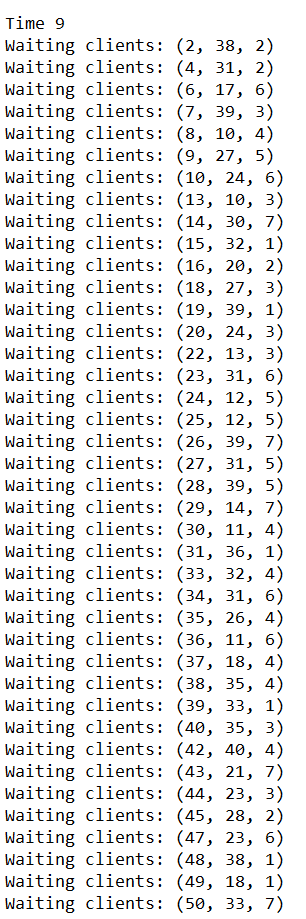
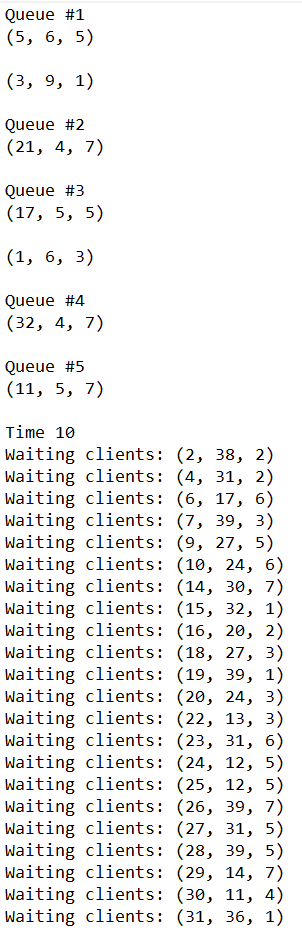
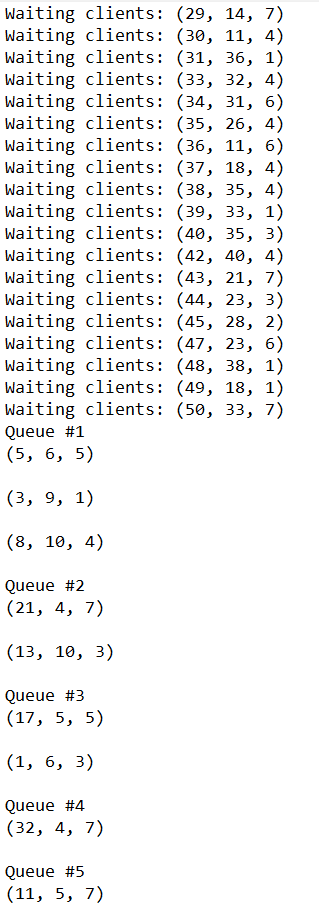
Am testat programul pentru input-urile din cerință și am afișat rezultatele în 3 fișiere text, pe care le voi încărca alături de proiect pe GitLab. Acum voi atașa câteva screenshot-uri cu rezultatele:

1. Test 1 (toate cele 60 de secunde)

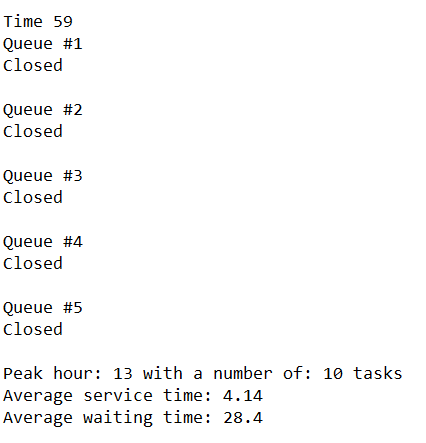
    

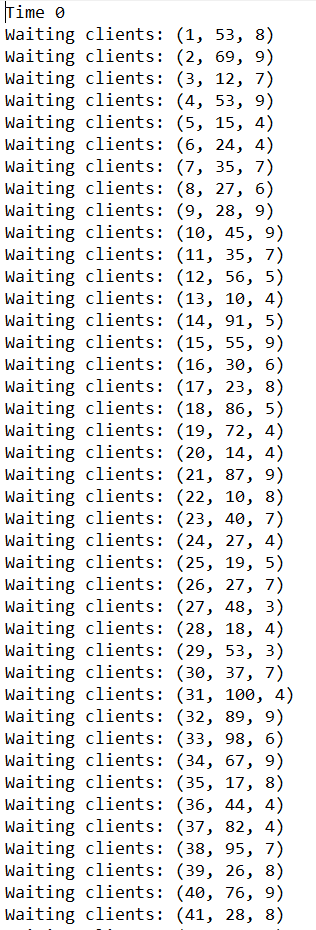
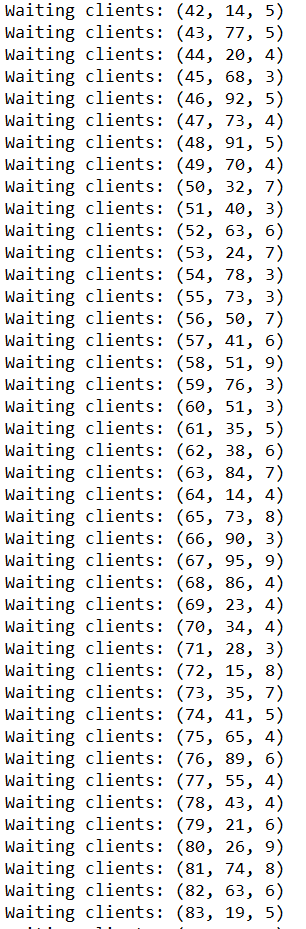
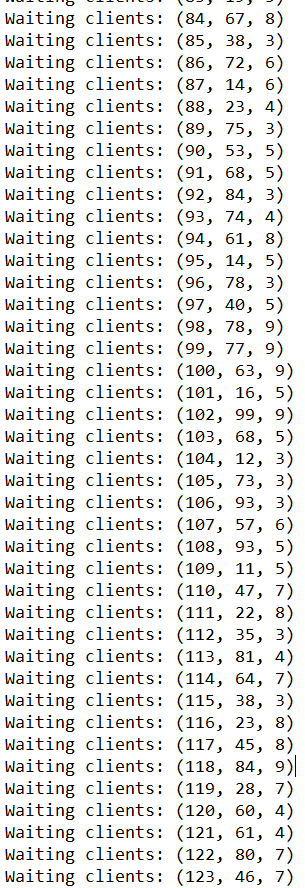
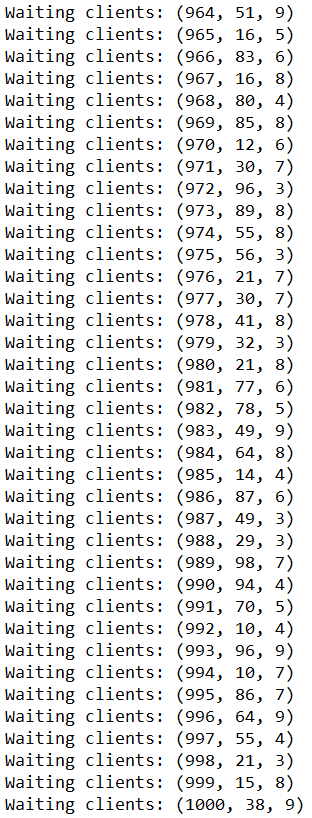
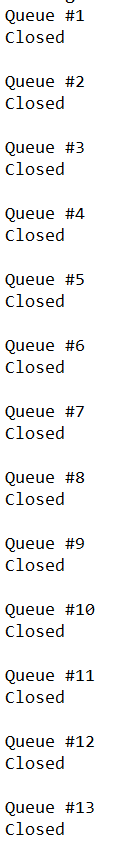
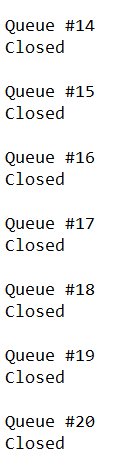
1. Test 2

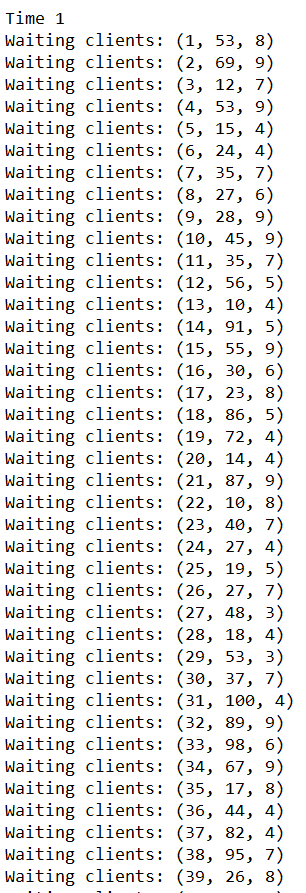
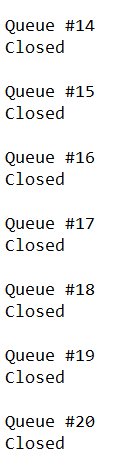
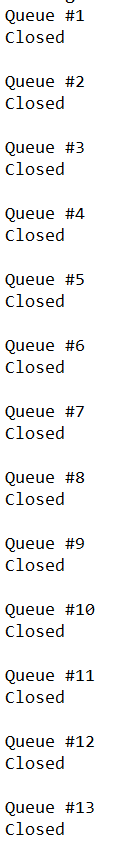
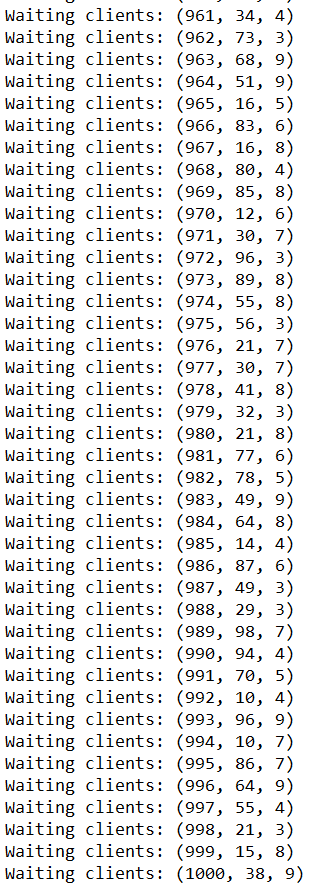
               etc.

Rezultate:

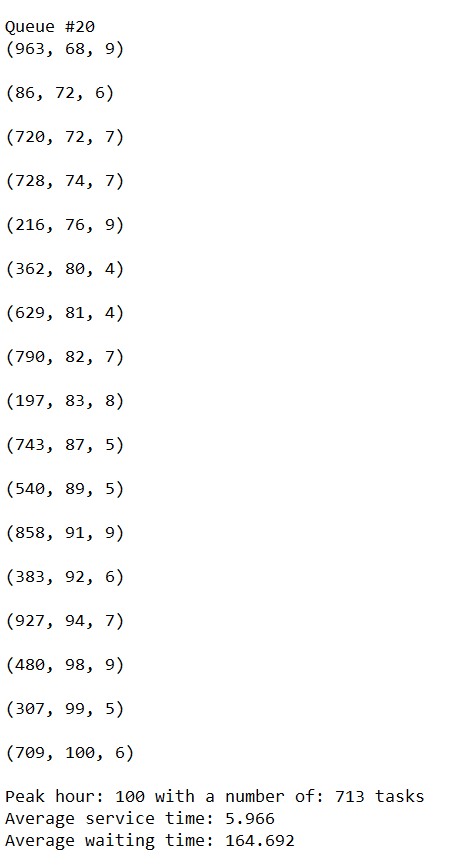


1. Test 3

  etc.   

etc. etc.

Rezultate:



# **Concluzii**

În concluzie, această temă m-a ajutat să mă familiarizez cu thread-urile și lucrul cu acestea, să descopăr clase noi, precum cele atomice și BlockingQueue, pe care le-am folosit la sincronizarea thread-urilor și să învăț să lucrez cu ele și noi moduri de a-mi structura proiectul. Cel mai mult mi-a plăcut să lucrez la interfața grafică, pe care am realizat-o folosind dependițe JavaFX în proiectul meu de tip Maven.

Ca dezvoltare ulterioară, mă gândesc să îmbunătățesc interfața grafică, să fie mai colorată și sa aibă reprezentări grafice pentru cozi și clienți, în loc de numere.

# **Bibliografie**

<https://docs.oracle.com/javase/7/docs/api/java/io/BufferedWriter.html>

<https://stackoverflow.com/questions/13784333/platform-runlater-and-task-in-javafx>

<https://docs.oracle.com/javase/8/javafx/api/javafx/application/Platform.html>

<https://docs.oracle.com/javase/7/docs/api/java/util/Timer.html>

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

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html>

<https://www.baeldung.com/java-strategy-pattern>