

Documentație simulare cozi

Fariseu Adriana-Teodora

Facultatea de Automatică și Calculatoare, Grupa 30225

Anul 2020-2021

# Obiectivul temei

Se cere realizarea unei aplicații care permite atât urmărirea în timp real a servirii unor clienți ce sunt sau așteaptă a fi distribuiți în mai multe cozi, dar și realizarea unor statistici în legătură cu rezultatele acestei simulări (timp mediu de așteptare, timp mediu de servire, ora de vârf). În același timp, se dorește ca această simulare să fie cât mai asemănătoare procesării paralele, impunându-se lucrul cu thread-uri

Pentru a atinge acest obiectiv principal, va trebui să se trateze următoarele obiective secundare:

|  |  |  |
| --- | --- | --- |
| Obiectiv | Descriere | Capitolul în care este tratat |
| Analiza problemei | Pentru siguranța că se vor satisface toate cerințele problemei, trebuie să se stabilească ce trebuie să se urmărească prin examinarea cerințelor funcționale, non-funcționale și a cazurilor de utilizare | [2](#_Analiza_problemei) |
| Proiectarea | Cunoscând cerințele problemei, se poate trece la maparea acesteia în paradigma de programare pusă la dispoziție | [3](#_Proiectare) |
| Implementarea | Se implementează efectiv rezultatele mapării realizate anterior, eventual descoperindu-se noi nevoi pe parcurs | [4](#_Implementare) |
| Testarea | Se testează aplicația pentru a ne asigura că funcționează cum trebuie și că rezolvă problema cerută | [5](#_Testare) |

# Analiza problemei

***Cerințe funcționale***

Pentru ca aplicația să rezolve problema cerută, următoarele obiective trebuie neapărat îndeplinite:

**Generarea unor seturi de numere aleatoare**. În timp ce anumite valori sunt date de utilizator, un anumit număr de perechi de numere trebuie generate aleator, respectând în același timp limitarea într-un interval dat

**Lucrul pe fișiere**. Se cere crearea și scrierea unui fișier .txt cu evoluția cozilor și a clienților de servit

**Organizarea pe fire de lucru**. Este menționat faptul că fiecare coadă trebuie să aibă corespondență într-un fir de lucru, fiind de fapt principalul lucru care se urmărește în realizarea aplicației. Despre această corespondență(începerea thread-ului, încheierea, întreruperi) se va vorbi în detaliu în capitolul de proiectare

***Cerințe non-funcționale***

După ce sunt implementate corect funcționalitățile de bază, următoarele funcționalități ar îmbunătăți aplicația:

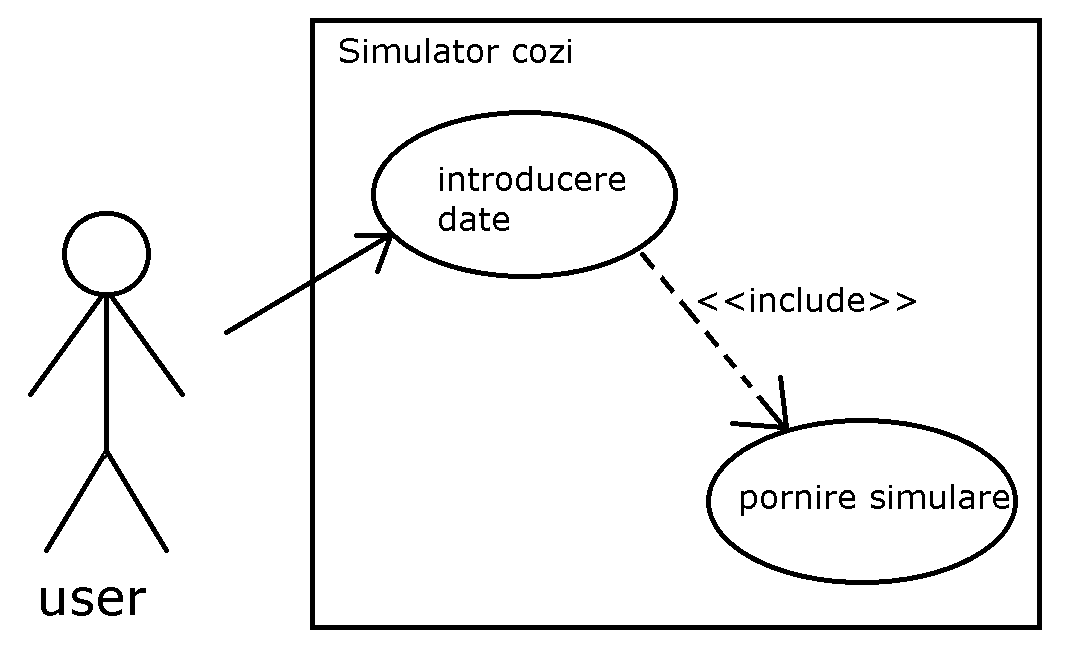
**Calculul statisticilor**. La terminarea simulării, ar fi util să se culeagă detalii cu privire la:

* Timpul mediu de așteptare
* Timpul mediu de servire
* ”Ora de vârf”( momentul la care cei mai mulți clienți se află în decurs de servire)

**Interfață grafică**. Pentru o utilizare și navigare mai accesibilă în aplicație, se poate implementa o interfață grafică ușor de utilizat. Aceasta trebuie să cuprindă o parte pentru introducerea datelor inițiale și afișarea cu animație a simulării

**Pauza și încheierea prematură a simulării**. Uneori ar fi de dorit să se pună pauză simulării în cazul în care vrem să observăm un pas mai în detaliu. De asemenea, în cazul în care nu mai sunt clienți de servit, nu ar mai avea rost ca aplicația să ruleze

***Scenarii de utilizare:***

******

În principiu, singurul scenariu de utilizare este simularea cozilor

Scenariu de utilizare: simularea evoluției servirii unor clienți la un număr de cozi

Actor principal: utilizator

Cazul de utilizare principal:

1. Se introduc datele de intrare:
   1. Nr clienți
   2. Nr cozi
   3. timp maxim simulare
   4. Interval timp sosire : min,max
   5. Interval timp servire: min,max
   6. Strategia de alegere a cozii (după timpul minim de așteptare sau după numărul de clienți)
2. Se apasă un buton de pornire simulare
3. Se generează intern valorile random pentru clienți și thread-urile necesare
4. Se pornește simularea pentru timpul maxim sau până când nu mai sunt clienți ce se procesează
5. Se încheie simularea și se finalizează fișierul cerut

Scenarii alternative:

* 1. Se introduc date incorecte (timp negativ,flotant, caractere care nu sunt cifre, limita maximă < limita minimă într-un interval etc.)
* Nu se începe simularea și se revine la pasul 1
  1. Utilizatorul pune pauză simulării
* Se oprește simularea până când se repornește dintr-un buton

**Situații de introducere eronată a datelor**. După cum s-a menționat, există situații în care simularea nu trebuie începută din cauza faptului că utilizatorul oferă date eronate din interfața grafică fie din neștiință, fie din pură inițiativă maniacală. De aceea, trebuie să ne asigurăm că datele introduse sunt corecte, stabilind în primul rând de ce tipuri avem nevoie. În acest caz, toate numerele ce trebuie date pentru simulare trebuie să fie numere naturale (întregi pozitive în unele cazuri nenule), deci avem următoarele cazuri de date eronate:

* Litere sau alte simboluri în input în afară de cifre. Deoarece nu s-ar putea parsa un întreg
* Un număr cu virgulă. Pentru că se dau cardinale de mulțimi (număr de clienți, număr de cozi) și pentru că eșantionarea timpului se realizează exact din secundă în secundă
* Un număr pozitiv: Același motiv ca la numerele cu virgulă
* Un câmp gol: se poate introduce o valoare default, dar nu ar avea sens în cazul tuturor câmpurilor goale
* Un număr egal cu 0: nu ar avea sens din punct de vedere logic să se înceapă simularea pe nimic
* Capătul inferior al intervalului mai mare decât cel superior (pentru câmpurile unde trebuie să se dea minimul și maximul pentru generare de numere)

# Proiectare

Clase

Principalele substantive ce pot fi deduse din enunț sunt: Client, Coadă, Simulare. Pe baza acestora deci, se va realiza organizarea modelului care rezolvă problema cerută. Alături de acestea, pentru interfața grafică, trebuie să se realizeze o fereastră pentru introducerea datelor simulării și una pentru urmărirea ei, având clase de controller corespunzătoare pentru interacțiunea cu utilizatorul

Timeline

Description automatically generated with low confidence

Clasa de statistici. Se va ocupa de reținerea evoluției statisticilor în fiecare secundă, având nevoie de operații atomice. Statisticile care trebuie calculate sunt:

Peak time-ul (pentru care trebuie să se rețină atât secunda, cât și numărul de clienți la momentul respectiv

Timpul mediu de așteptare (pentru care trebuie să se rețină numărul de clienți serviți și

Pachete

Pe lângă pachetele pentru Model-View-Controller, se mai poate deduce pachetul pentru strategiile de alegere a cozii unde se repartizează următorul client în așteptare.

Diagram

Description automatically generated

Design Patterns

***Modelul Model-View-Controller***

Este printre cele mai folosite pattern-uri arhitecturale, presupunând separarea claselor într-o parte de logică din spare pentru rezolvarea problemei în sine (Model), partea de interfață grafică (View) și o parte care face legătura dintre interfață și acțiunile utilizatorului (Controller)

***Strategy Pattern***

Este un pattern arhitectural care presupune alegerea dinamică unui algoritm la rularea programului, față de simpla scriere a unui singur algoritm pentru rezolvarea unei problemei.

În cazul acestui proiect, schimbarea strategiei s-a putut aplica la alegerea cozii în care să se distribuie următorul client din lista de așteptare, acest lucru putându-se realiza:

* Prin a ține cont de numărul de clienți din coadă
* În funcție de timpul de așteptare total din acea coadă (suma tutor timpilor de servire ai clienților din coadă).

Din moment ce aceste criterii se bazează foarte mult pe comparare, totul s-a putut reduce la folosirea de clase Comparator cu metode de compare() suprascrise

Organizarea Thread-urilor

***Sincronizare și concurența***

În programarea concurentă, modificarea simultană a unor resurse folosite de mai multe thread-uri poate duce la inconsistența datelor. O soluție la această problemă este fie folosirea blocurilor de synchronized, fie folosirea unor structuri de date specifice și sigure pentru multithreading. Având în vedere aceste lucruri, am folosit BlockingQueue și PriorityBlockingQueue pentru cozile serverelor și clienții în așteptare și timpuri de date atomice pentru valorile ce se modifică (timpul de servire) și synchronizedLists pentru lista de cozi

***Mecanismul wait-notify***

Prezentarea suport sugerează punerea firelor de coadă pe sleep pe un timp egal cu procesarea task-ului pentru a simula servirea clienților. Însă dacă doar se pune thread-ul pe sleep fără a ține cont de evoluția simulării, se poate risca apariția unor anomalii (thread-urile de coadă continuă să ruleze în spate atunci când thread-ul de simulare este oprit din butonul interfeței grafice). Din acest motiv, ar trebui să se implementeze un mecanism de notificare condiționată, oferit de metodele wait() și notifyAll() din clasa obiect. Astfel, în fiecare clasă de Server s-a păstrat referința Simulation Manager-ului din care face parte, apelându-se wait() pe acesta cu scopul de a face server-ul să aștepte până ce Manager-ul apelează notifyAll(), trezind simultan toate serverele la fiecare secundă pentru a se reactualiza în timp real

De asemenea, trebuie să avem grijă ca următorul client să nu se distribuie într-o coadă înainte ca thread-urile să înceapă să decrementeze timpul primului client (lucru care strică rezultatul atunci când se adaugă un client la o coadă goală) Din moment ce sunt mai multe fire de lucru care trebuie să comande o singură execuție, un mecanism cu wait-notify ar fi greu de implementat. Din fericire, pentru astfel de situații există clasa specială CyclicBarrier, similară CountDownLatch-ului, doar că reutilizabilă după inițializare. Această clasă permite așteptarea mai multor thread-uri pe o ”barieră” până când un anumit număr de fire de lucru în așteptare pe aceasta este atins. În momentul când un număr suficient de thread-uri așteaptă pe bariera ciclică, aceasta ”se va sparge”, executând o acțiune stabilită de o clasă Runnable

Structuri de date folosite

Toate structurile de date folosite au fost utilizate în varianta lor thread safe pentru evitarea folosirii blocurilor synchronized pentru acestea

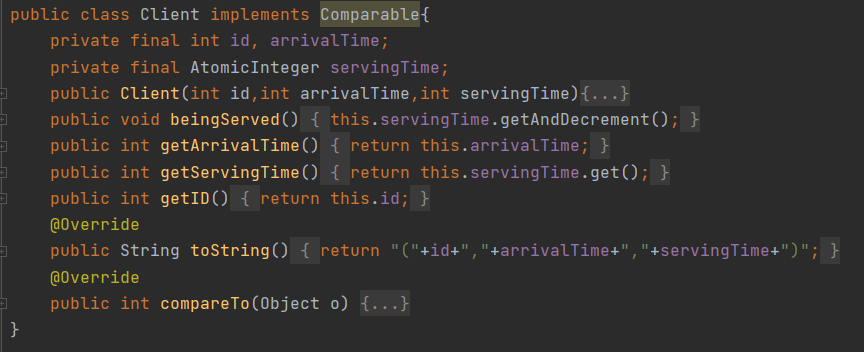
După cum le spune și numele, pentru implementarea cozilor de clienți am folosit o coadă de tip FIFO, având nevoie de acces doar la primul element din listă

Pentru a avea o evidență mai ușoară a clienților ce trebuie distribuiți într-o coadă la un anumit moment, am folosit o coadă de prioritate datorită operațiilor mai eficiente per total ( găsire minim O(1) și extragere minim O(log n), față de un ArrayList sau un LinkedList, unde astfel de operații ar dura O(n) și O(n) sau O(1) respectiv), în care clienții sunt sortați după timpul de sosire

Cozile corespondente firelor de lucru au fost păstrate într-o listă pentru a le păstra ordinea după index și pentru că alegerea minimului se realizează de fiecare dată în mod diferit

# Implementare

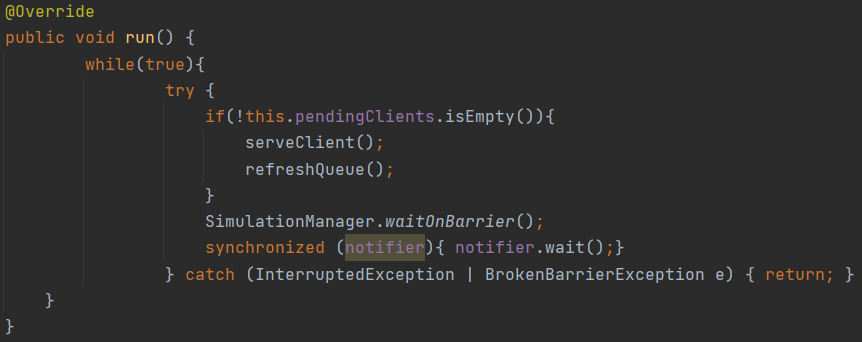
***Pachetul model***

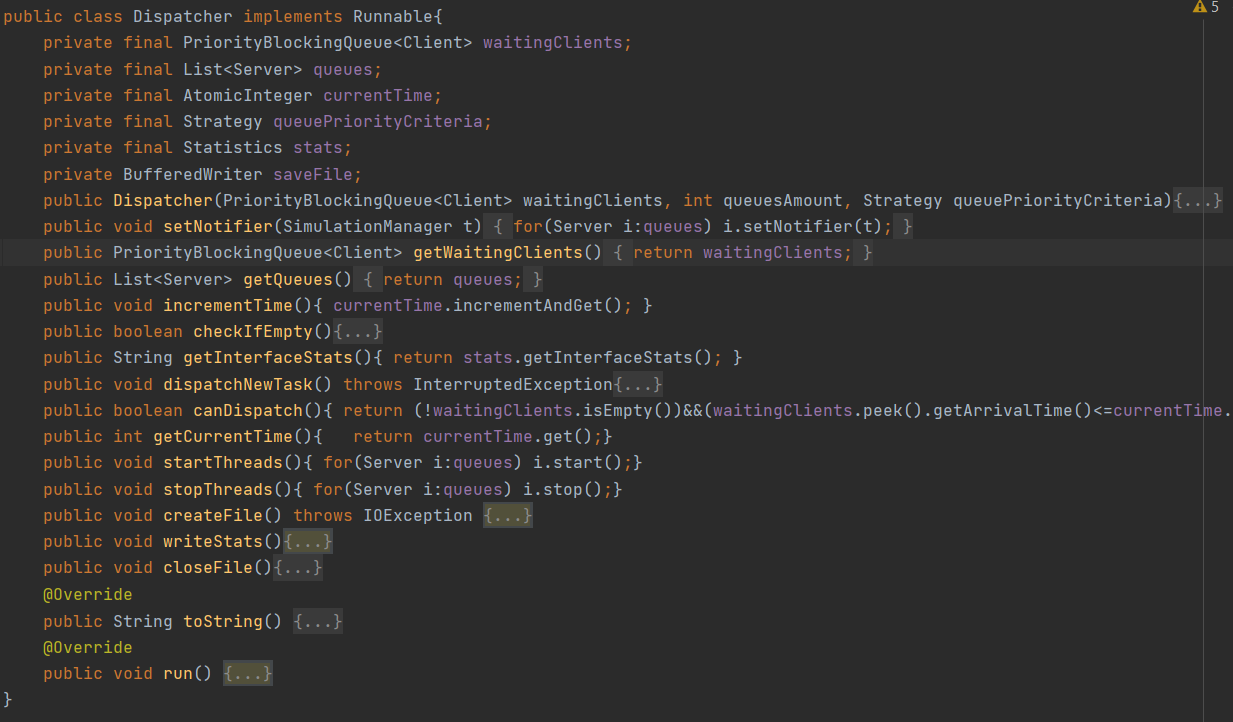


Clasa corespondentă clienților cu câmpurile cerute. De asemenea, conține metoda care simbolizează servirea lui în coadă.

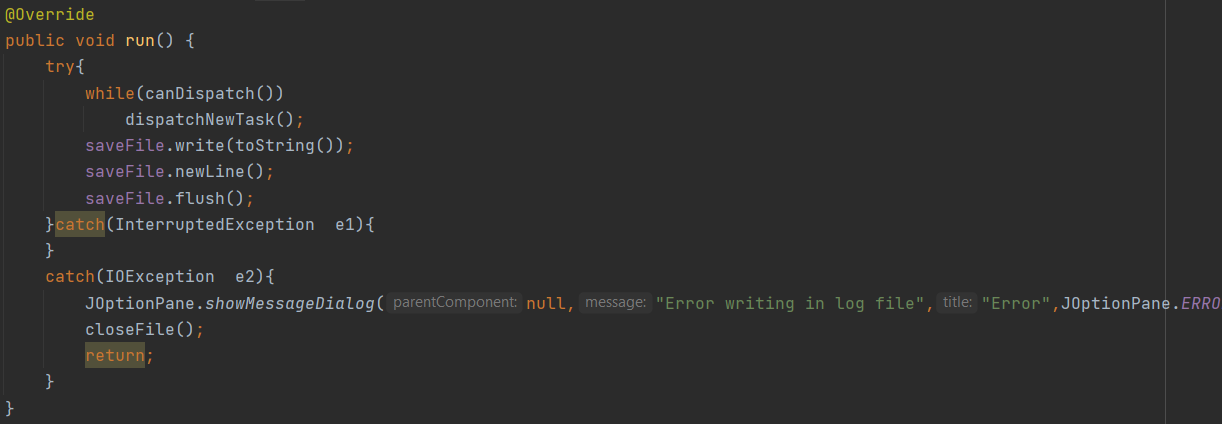


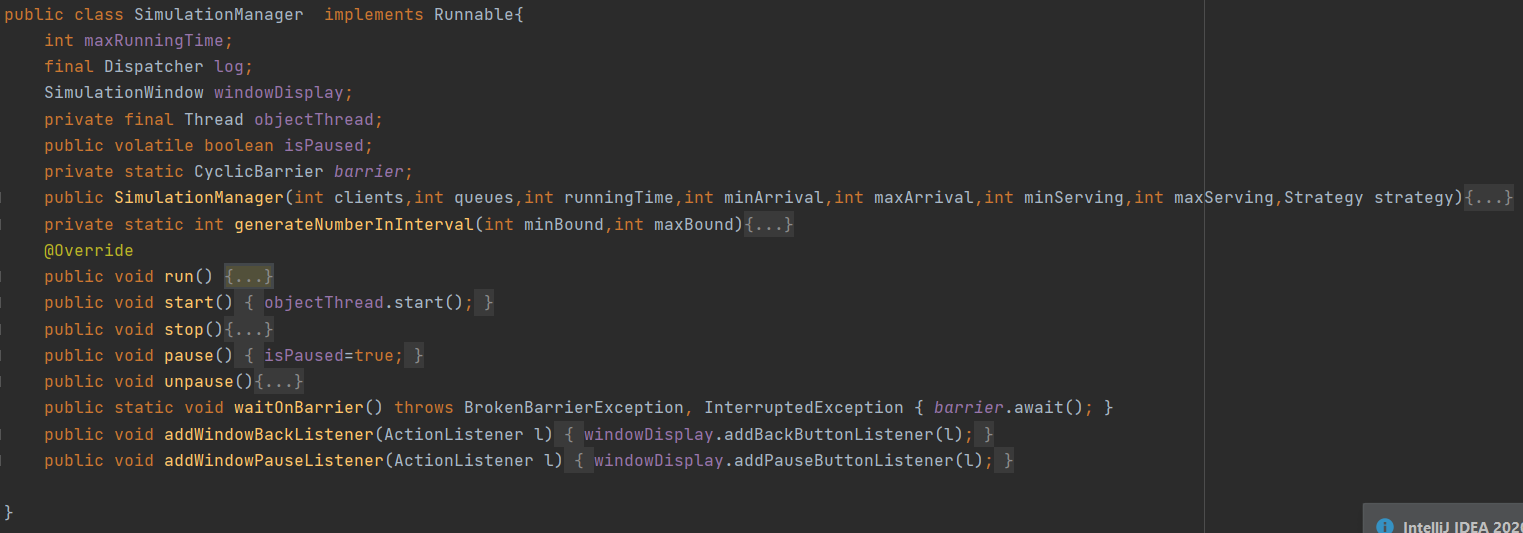
Cozile care servesc clienții și care vor fi corespondente câte unui fir de lucru (implementând deci Runnable). Aici se găsesc metode de scoatere din coadă când timpul ajunge la 0 și servire a clientului din față (decrementarea timpului de servire)



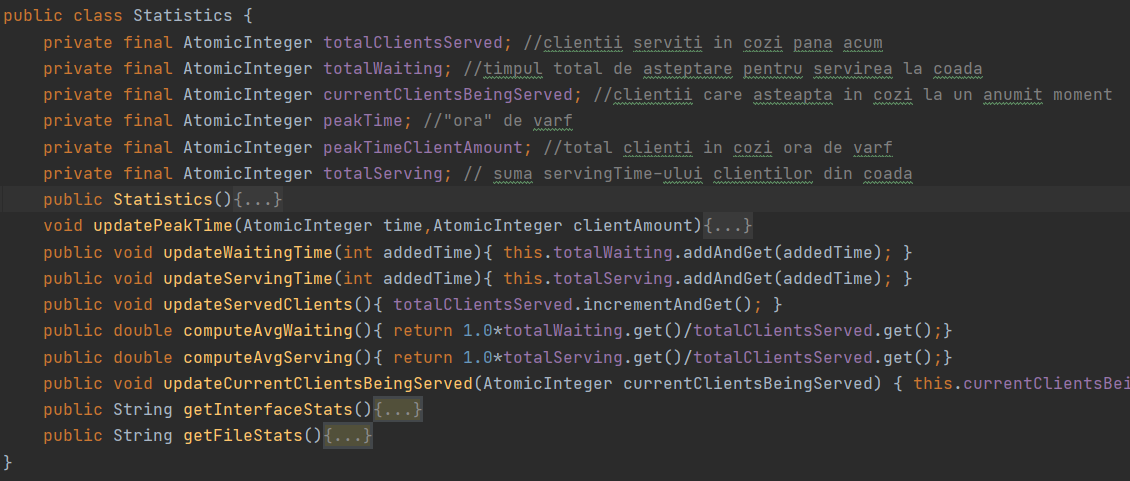
Metoda run din această clasă este scrisă aproape pseudocodic pentru a acorda mai multă atenție fluxului firului de lucru. În cazul în care există client de servit, el va fi procesat,după se va verifica dacă timpul de procesare a ajuns la 0 pentur a fi scos. După aceste lucruri, se notifică bariera din Manager care va aștepta ca toate serverele să proceseze clienții curenți înainte să adauge alții noi. După aceea, thread-ul se pune pe wait pe monitorul manager-ului pentru a putea face simultan toate thread-urile să decrementeze la următoarea iterație

Este o clasă responsabilă pentru distribuirea următorului client din lista de așteptare într-o coadă conform unei anumite strategii. La rândul ei este Runnable pentru a executa acțiunea care are loc la spargerea barierei din Manager



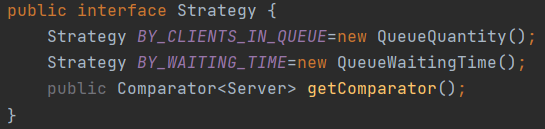


Este clasa care se ocupă de simularea principală, doar notificând thread-urile de cozi și apelând metoda pentru distribuirea unui client în coadă. Ea coordonează thread-urile de Servere, dar și așteaptă ca acestea să termine execuția de iterație înainte să se treacă

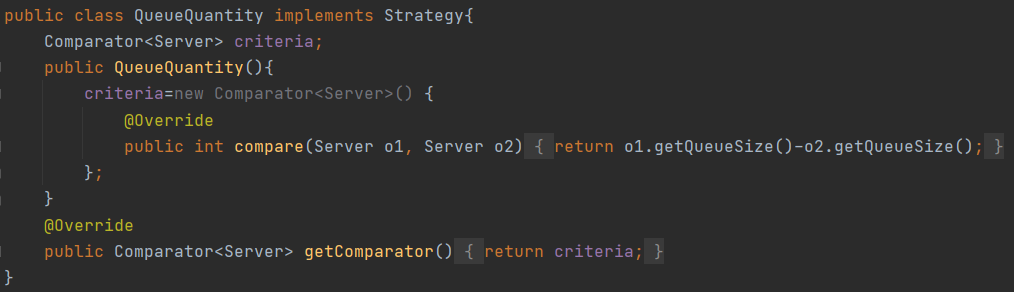


Clasă suport cu operații atomice care se ocupă de menținerea statisticilor de la fiecare pas al simulării

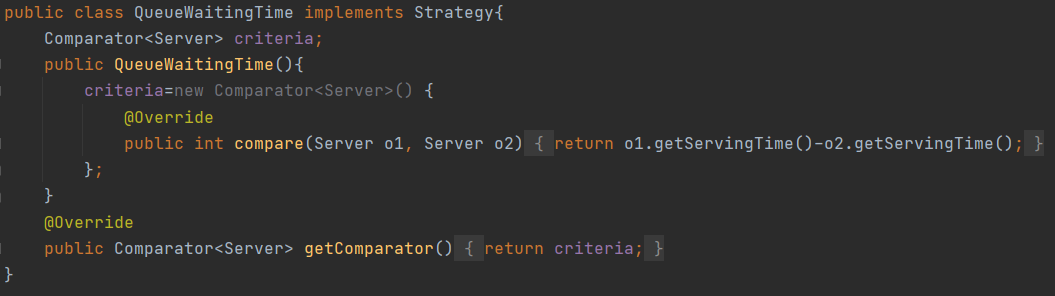
***Pachetul model.strategy***

******

Este interfața pentru strategiile de selecție a cozii de distribuție pentru un nou client

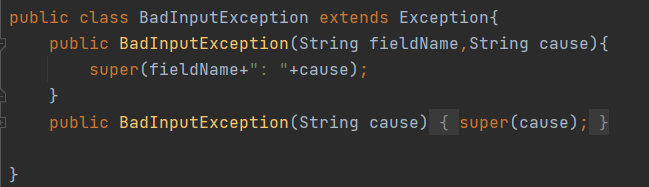


Clasa pentru strategia de selecție prin lungimea cozii (câți clienți sunt într-o coadă)



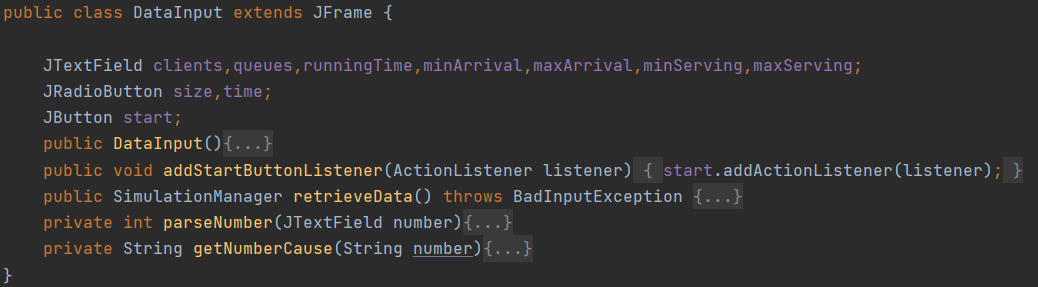
Clasa pentru strategia de selecție prin timpul total de așteptare (suma timpilor de servire a cliențior din coadă)

***Pachetul view***

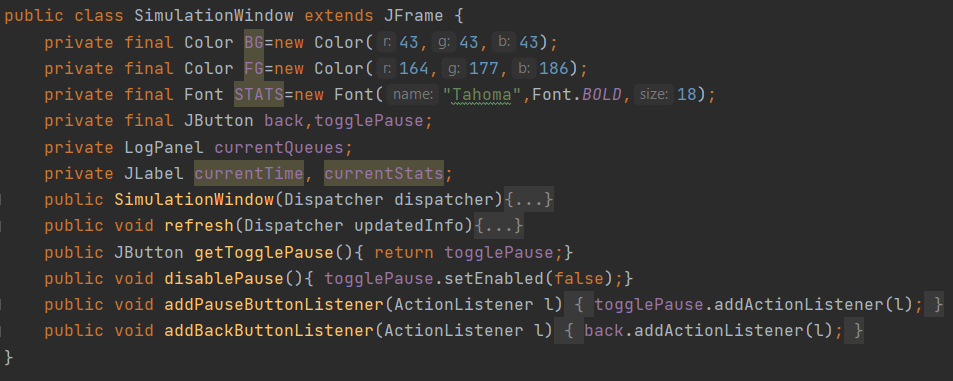
******

Clasa pentru excepțiile de introducere a datelor

***view.windows***

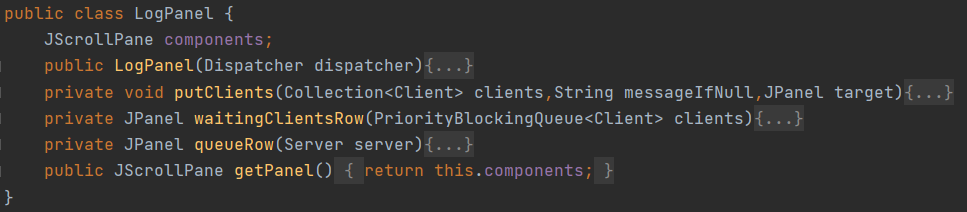
******

Fereastra pentru introducerea datelor cu câmpurile pentru informațiile de introdus și butonul de pornire. De asemenea, conține metode de verificare a inputului și identificare a motivelor de eroare

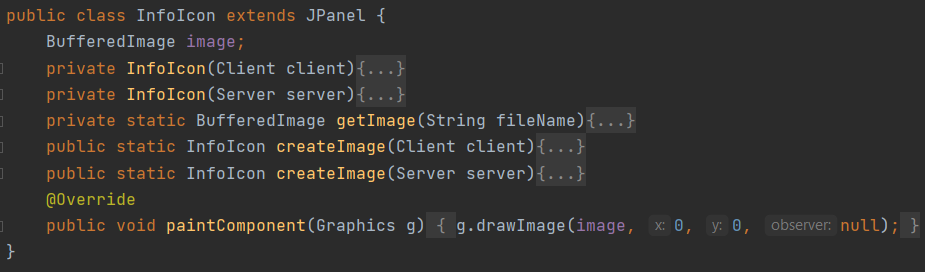


Fereastra pentru simularea în sine

***view.utility***

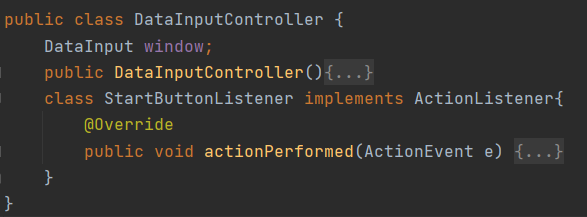
******

Zona în care sunt afișate cozile și clienții în așteptare

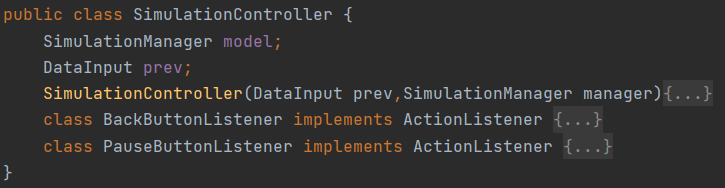


Imaginile din aplicație

***Pachetul controller***

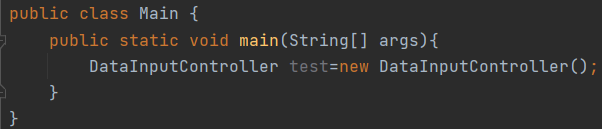
******

Clasa pentru evenimentele din fereastra de introducere a datelor(pornire simulare)



Clasa pentru evenimentele din fereastra de simulare (pauză și încheiere prematură)

***Pachetul default***

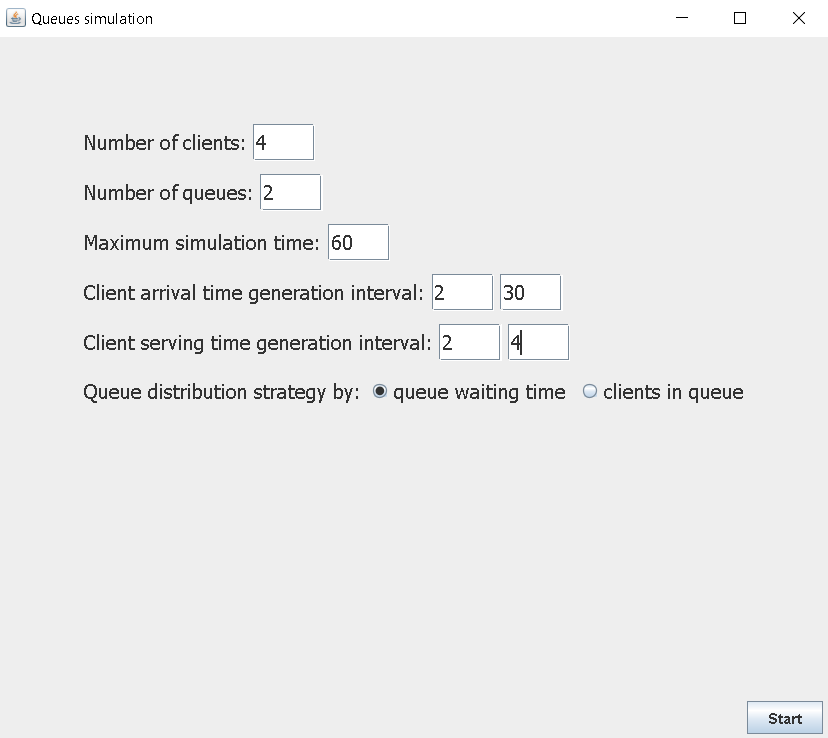


Clasa pentru inițializarea ferestrei cu metodă de main

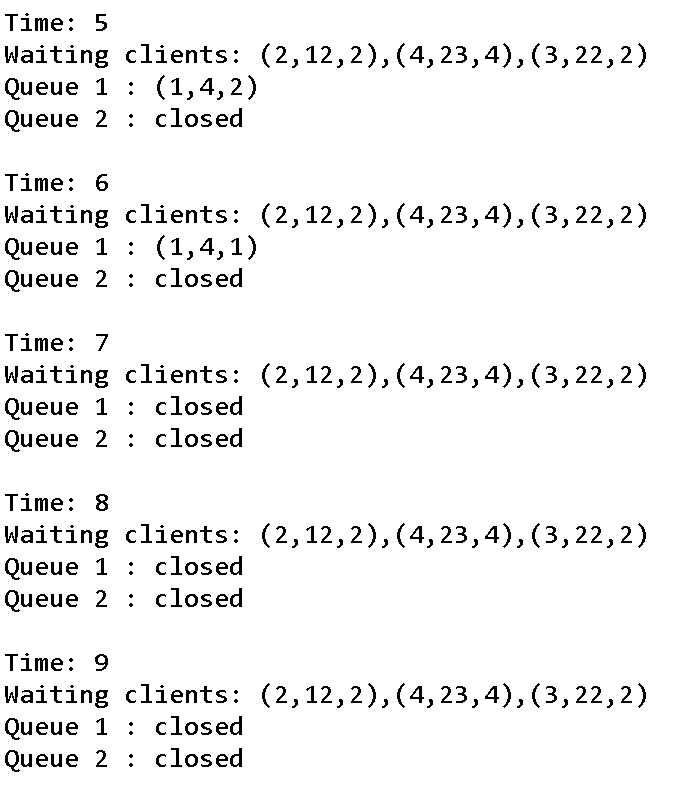
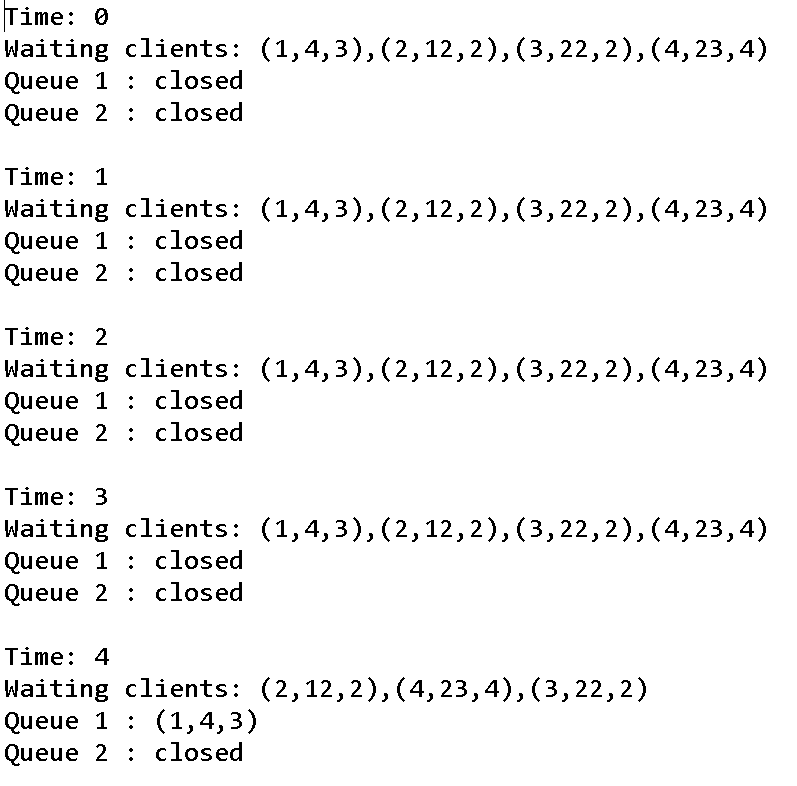
# Testare

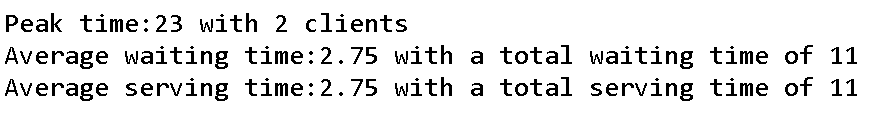
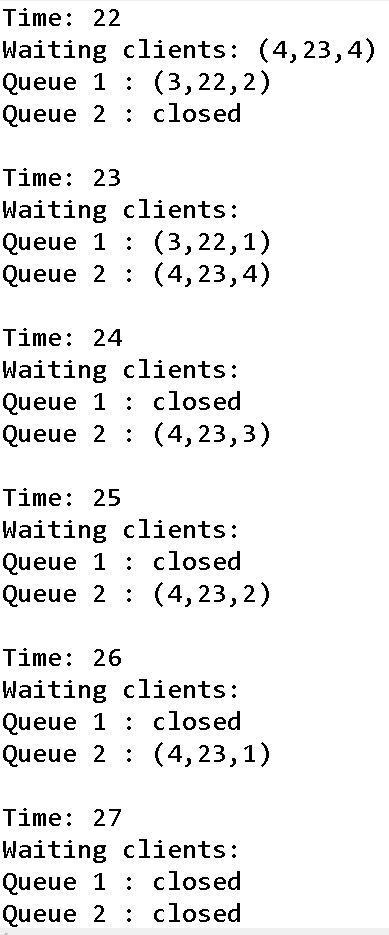
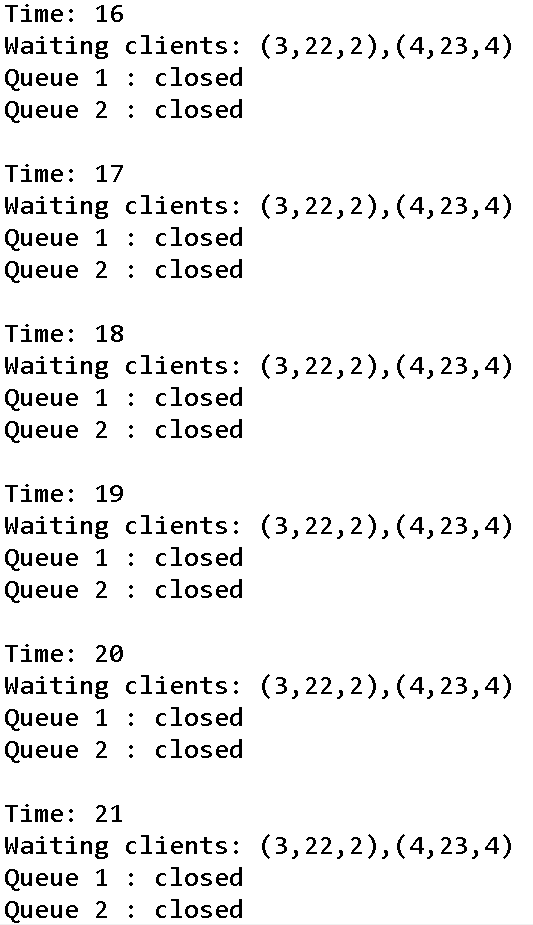
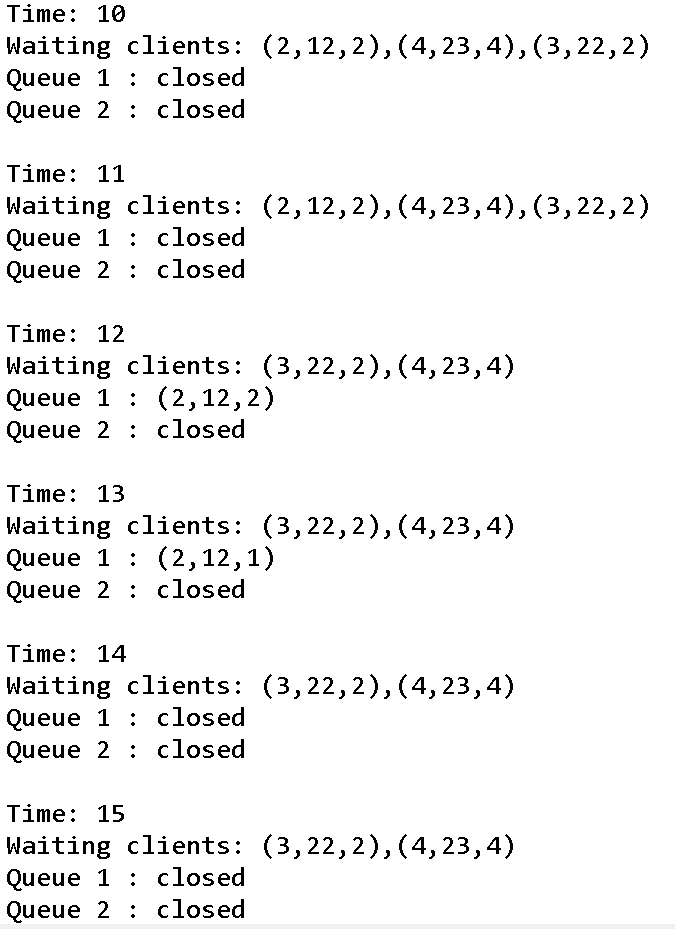
Din cauza naturii inițializării simulării (generarea randomizată a clienților), nu se pot efectua teste printr-un framework de testare, bazându-ne pe observarea practică a evoluției programului. Pentru demonstrarea funcționalității, se rulează următoarele teste:

Testul 1

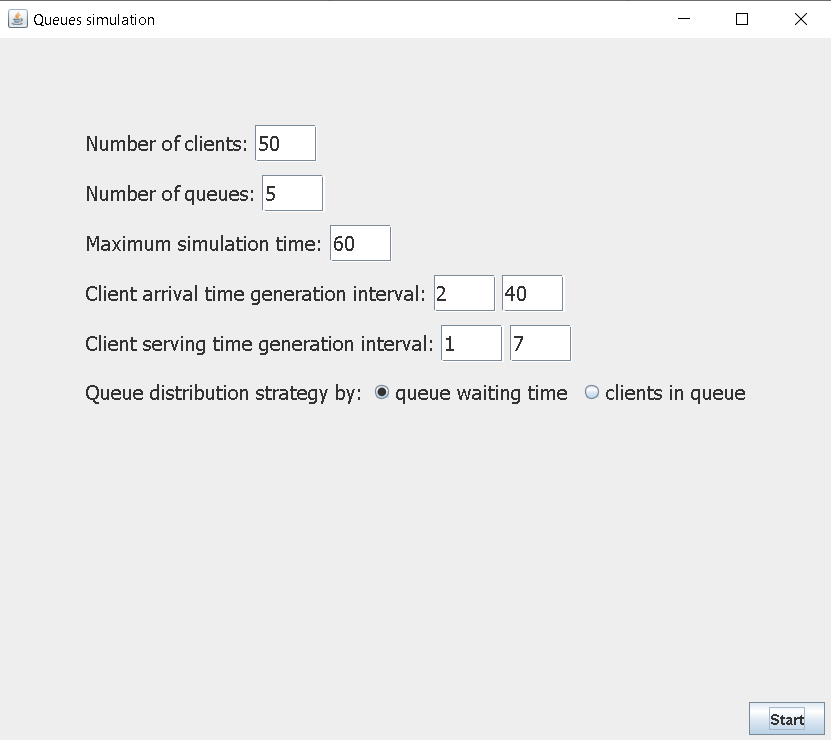


Fișier log:





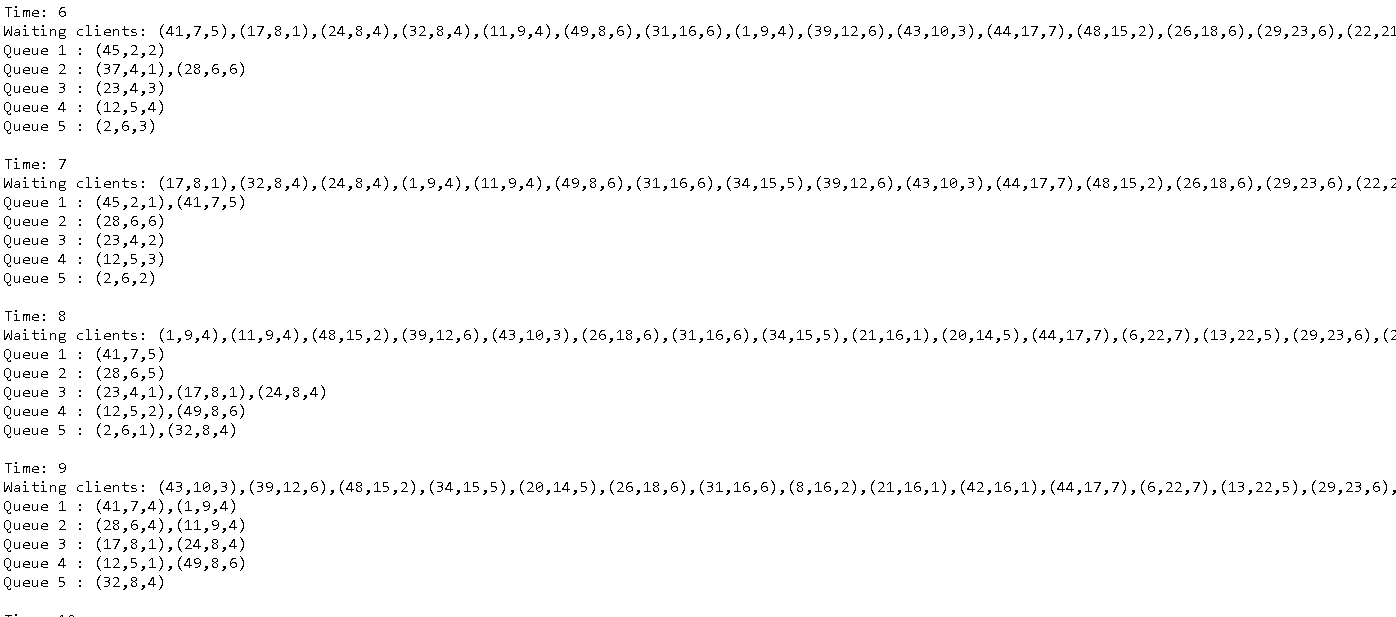
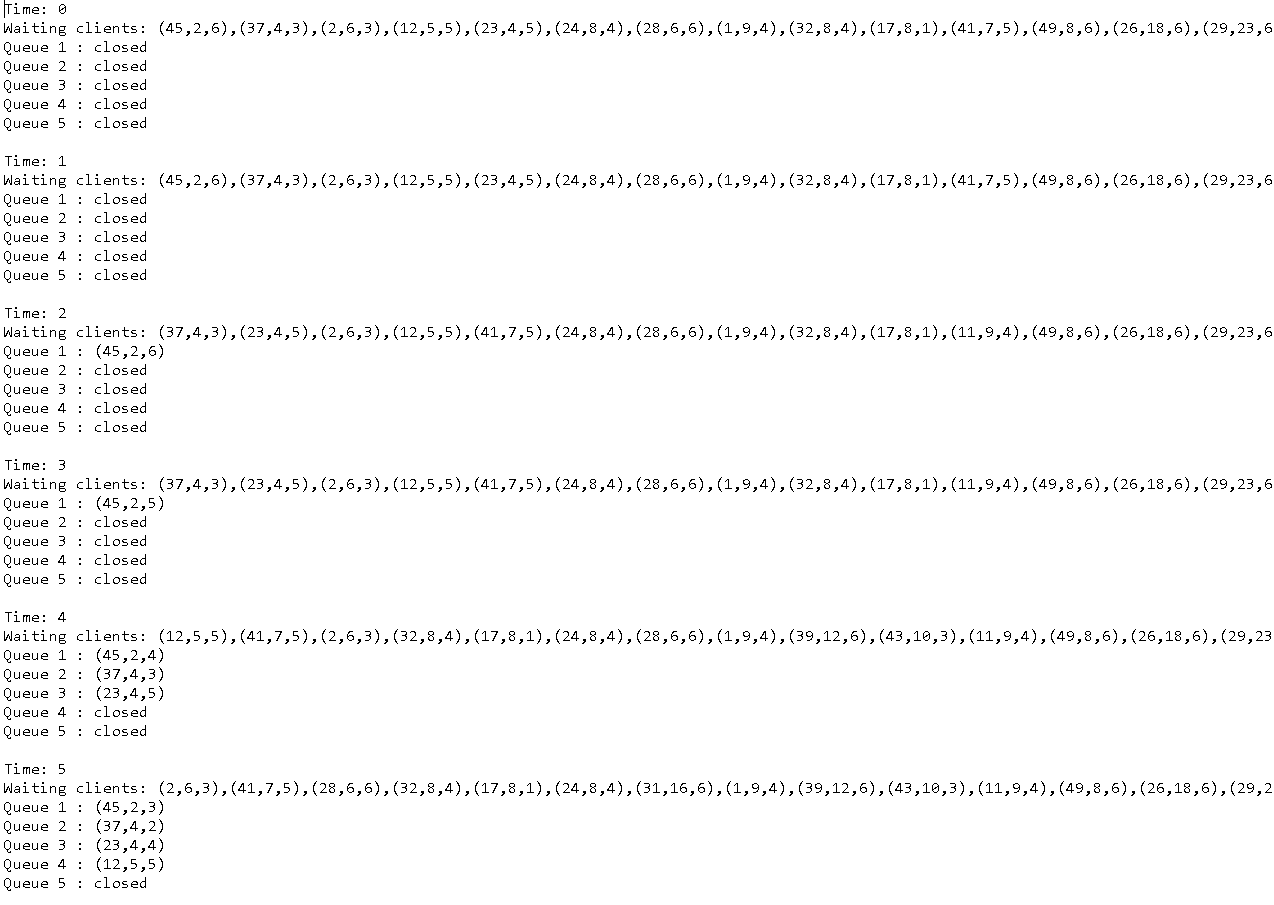
Test 2

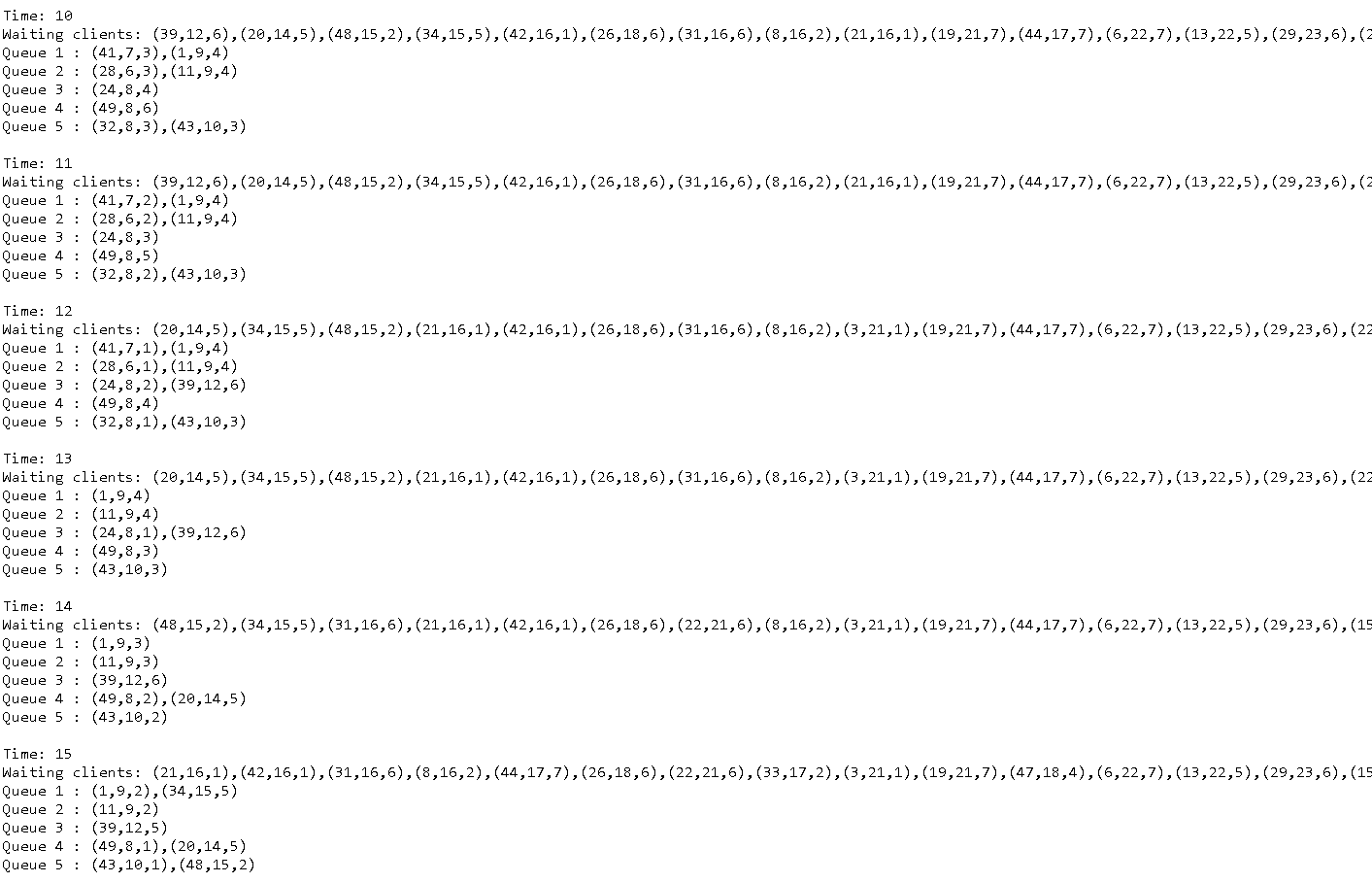


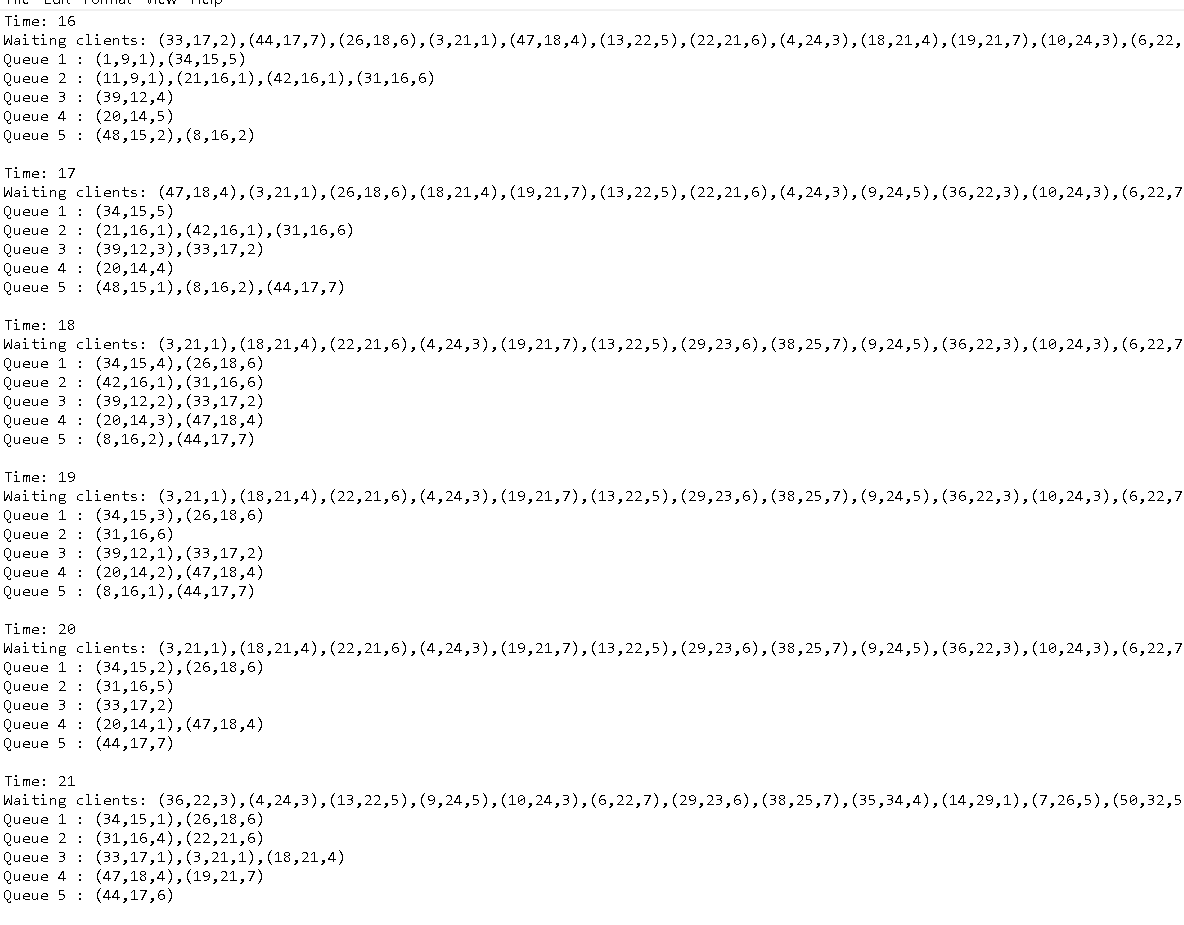
Un test cu un număr de clienți bun de arătat în interfața grafică

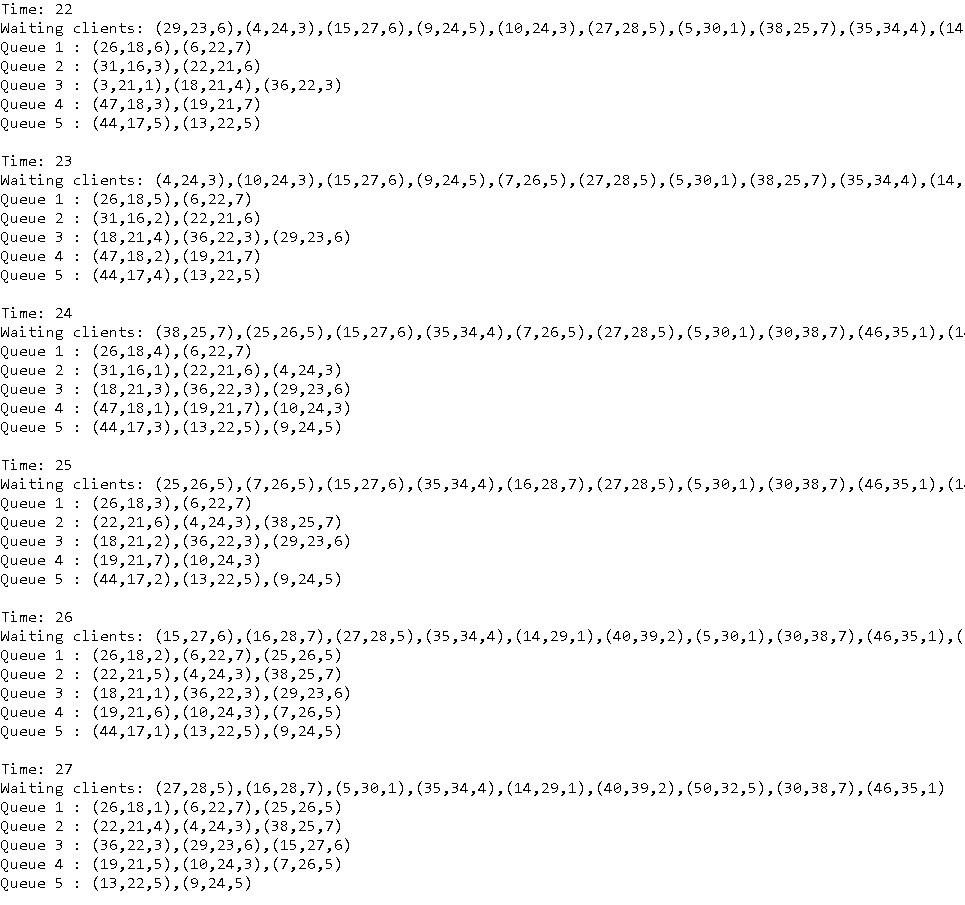


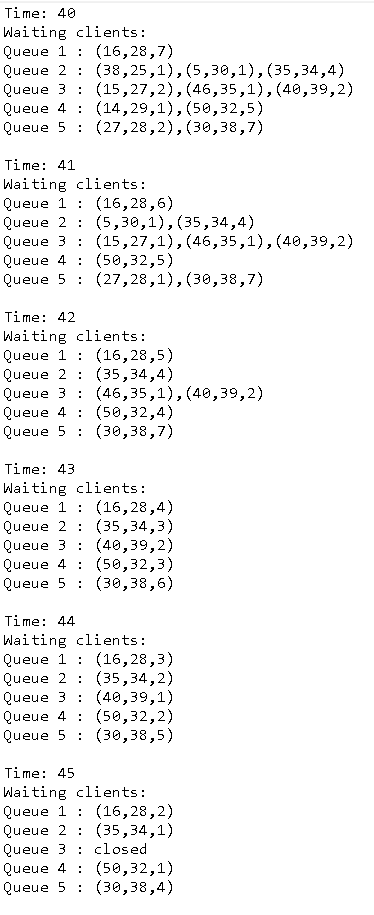
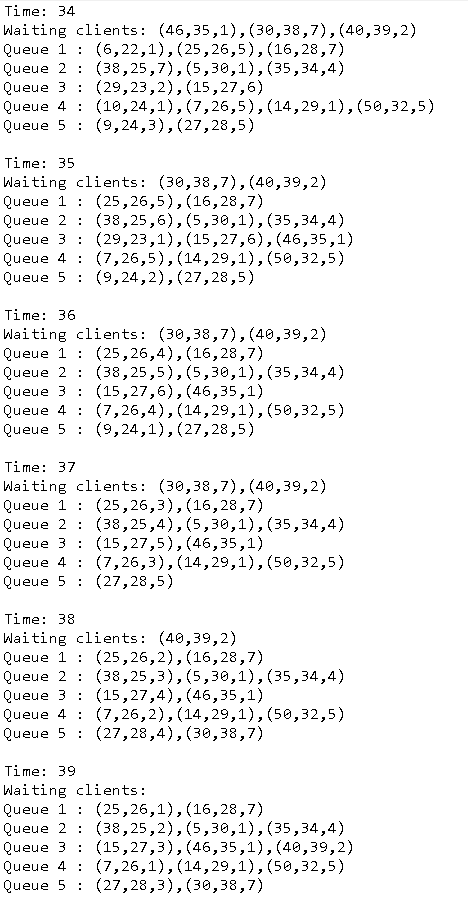
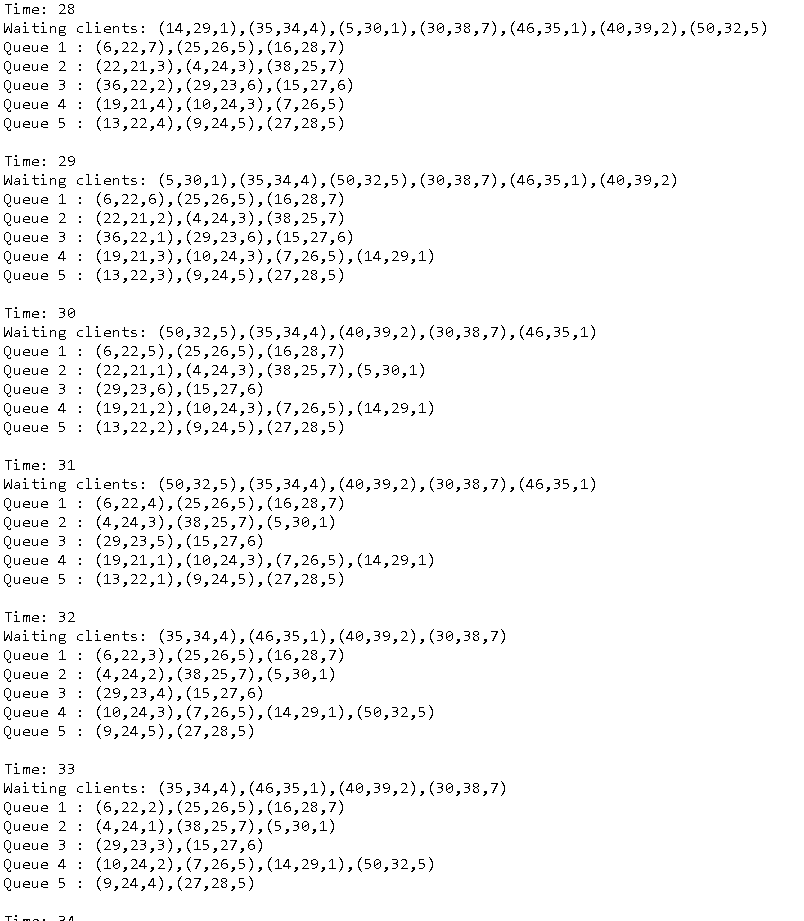
Fișier log:

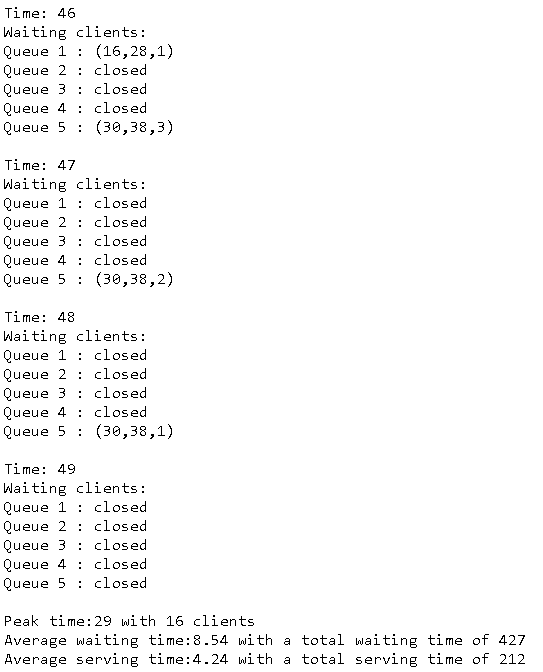




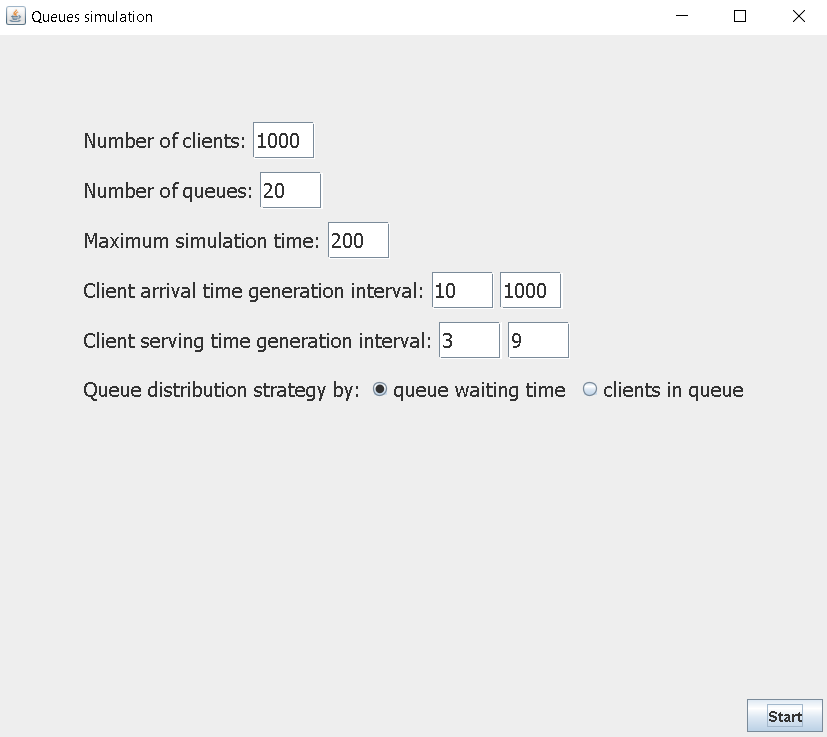




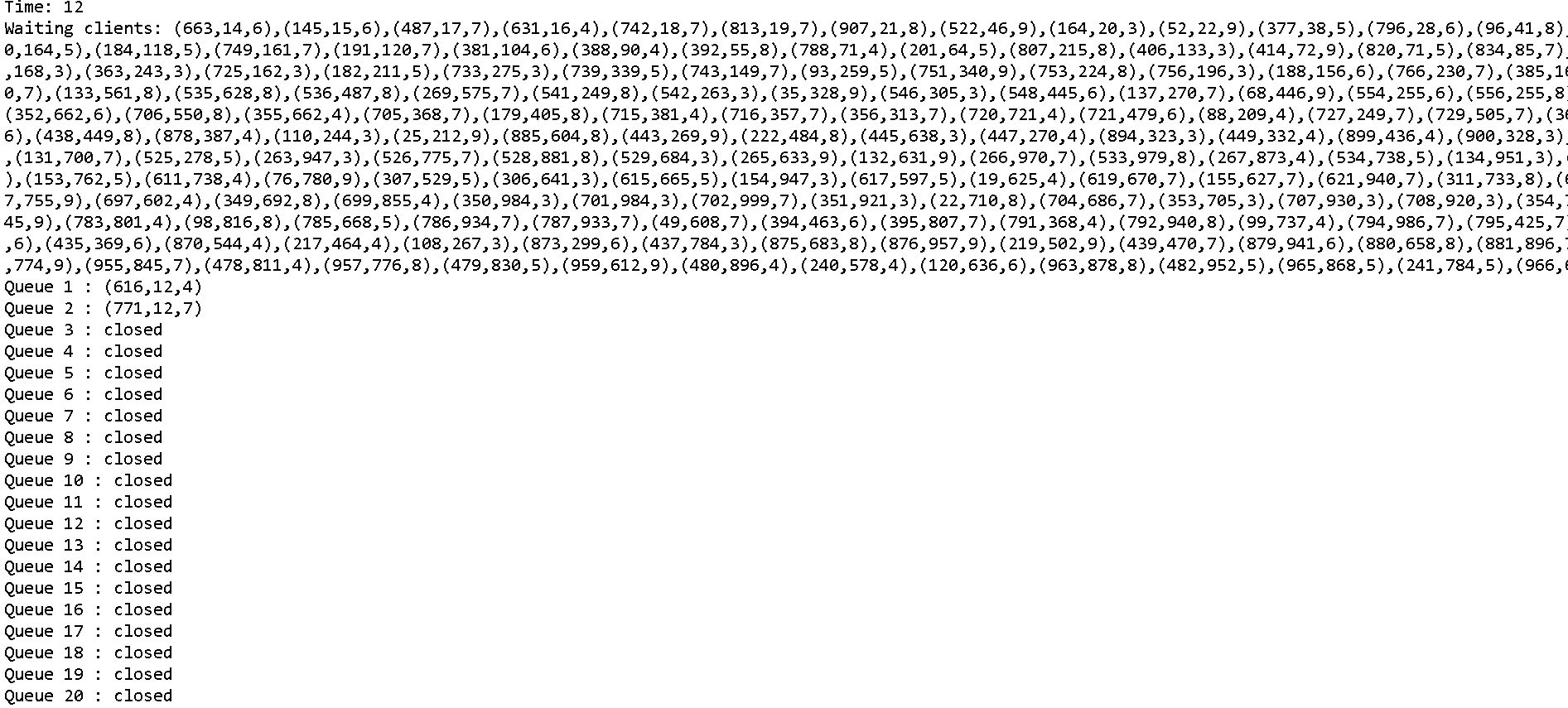




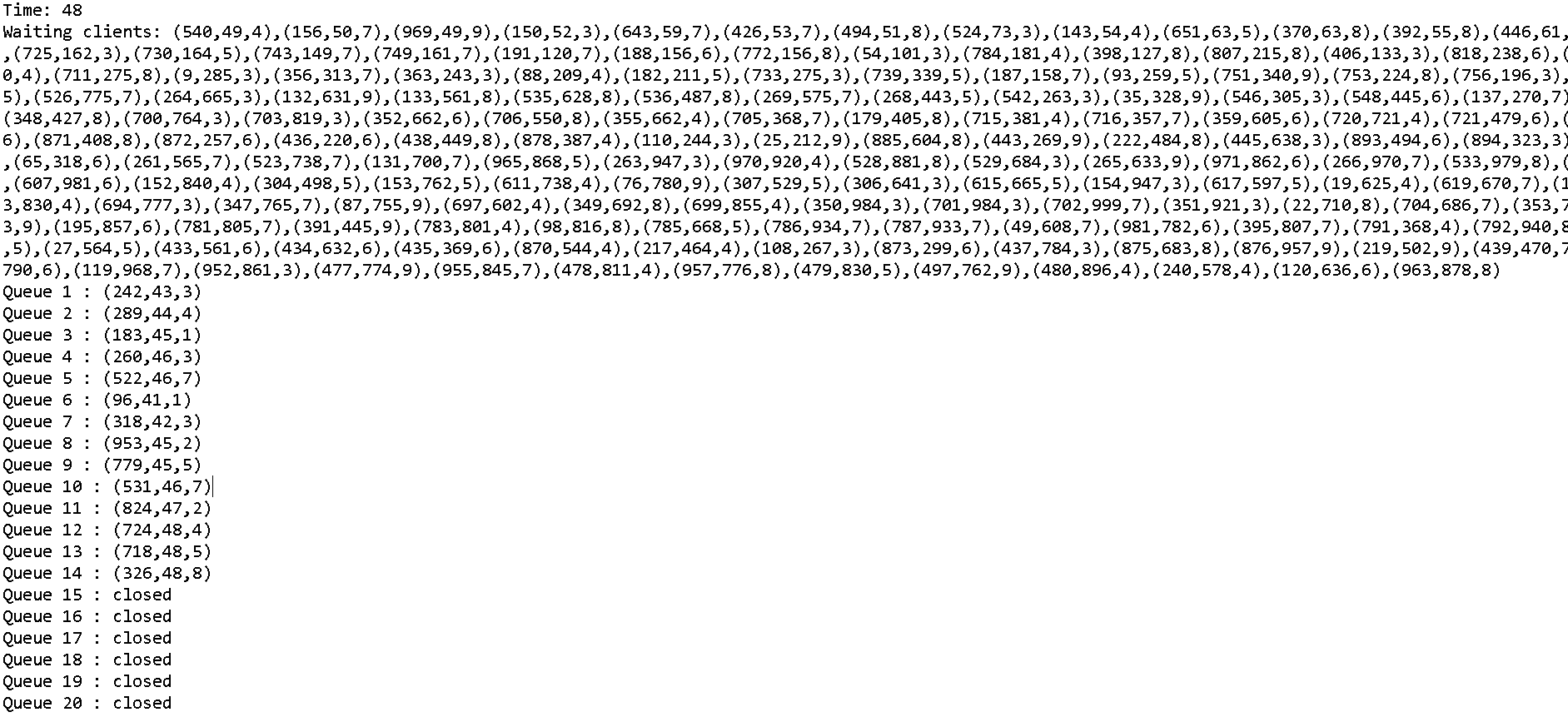
Test 3. Un test mai amplu. Nu s-au putut servi toti clienții înaintea terminării



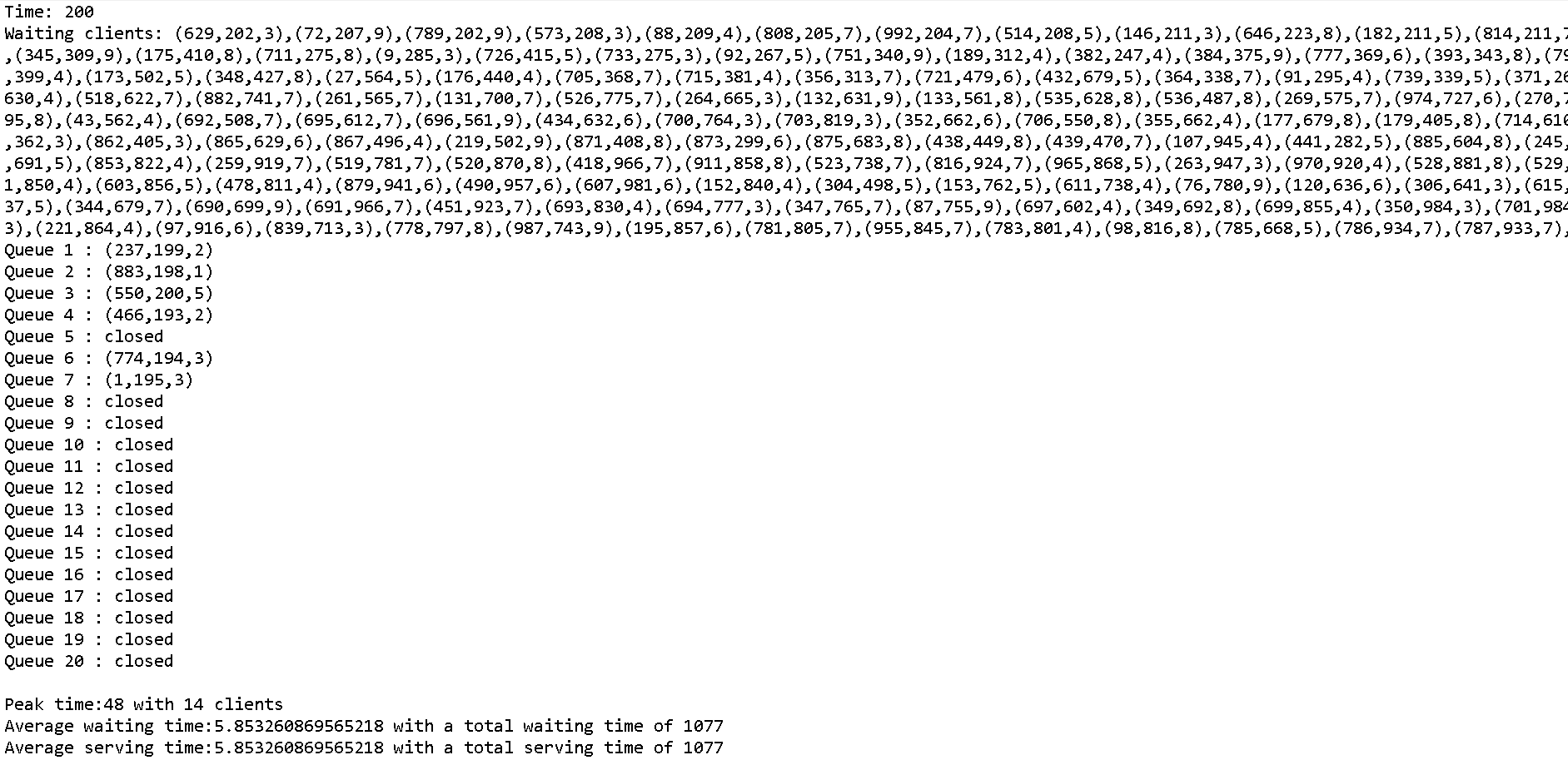
Câteva highlight-uri (vă rog frumos nu mă puneți să fac screenshot la tot fișierul, ca o să plâng)



Momentul intrării primilor clienți



Peak time



Ultimul moment al simulării + statisticile

# Concluzii

Acest proiect a fost un ajutor foarte mare în înțelegerea mecanismului de fire de lucru la un nivel mai practic. Am avut ocazia să înțeleg mai bine conceptele de sincronizare, ciclu de viață al unui thread, lacăte, monitoare și multe altele. Firele de lucru aduc o optimizare foarte mare în realizarea unor proiecte complexe, unde mai multe lucruri trebuie să ruleze simultan, deci înțelegerea acestora este un avantaj enorm. Chiar dacă nu sunt expertă în ele, acest proiect mi-a adus o fundație mai solidă de cunoștințe

Sugestia de Strategy Pattern a fost de asemenea o ocazie de a afla ceva nou și util în metode de programare și de a experimenta cu alegerea dinamică a algoritmilor pentru realizarea anumitor task-uri

Singurul meu regret este că mi-a luat atât de mult să mă prind de funcționarea firelor de lucru și de ce mecanisme de sincronizare pot utiliza, încât nu am apucat să învăț și să experimentez cu JavaFX cum îmi planificasem inițial. Un alt regret este că am stat până la 5 dimineața ca să pot deschide o amătâtă de imagine in IntelliJ pentru că nu știu cum îmi lua calea relativă și IntelliJ nu îmi dădea un URL pentru imagine, ci doar suferință

# Dezvoltări ulterioare

**Realizarea unui grafic cu numărul de clienți din cozi pe secundă**. Ar fi un lucru foarte interesant de realizat și chiar aș fi vrut să am o astfel de funcționalitate dacă aș mai fi avut timp pentru implementarea ei

**Îmbunătățirea interfeței grafice**. În mod normal aș fi trecut prin mai mulți pași în proiectarea și realizarea unei interfețe grafice, însă am vrut să mă axez pe înțelegerea unui lucru noi (lucrul cu thread-uri), față de ceva pe care m-am axat considerabil la proiecte anterioare

**Mai mult control asupra timpului execuției**.. S-ar putea reține ultimele 5-8 stări ale simulării pentru un buton de rewind sau s-ar putea alege derularea mai rapidă/lentă a simulării (prin a avea o variabilă care să fie transmisă ca parametru pentru sleep)

**Pornirea simulării de la timpul de sosire al primului client din lista de așteptare**. Mai mult pentru cazurile în care minimum arrival time este mare

# Bibliografie și resurse

<https://www.journaldev.com/1037/java-thread-wait-notify-and-notifyall-example>

<https://www.baeldung.com/java-cyclicbarrier-countdownlatch>

<https://www.baeldung.com/java-cyclic-barrier>

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

<https://howtodoinjava.com/java/multi-threading/multithreading-difference-between-lock-and-monitor/>