MINISTERUL EDUCAȚIEI NAȚIONALE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Facultatea de Automatică și Calculatoare
Secția Calculatoare și Tehnologia Informației

4/3/2021

TEMA 2
QUEUES SIMULATOR
DOCUMENTAȚIE

Sirghi Paula
GRUPA 30223

Contents

1. **Obiectivul temei** 2
2. **Analiza problemei, modelare, scenarii, cazuri de utilizare** 2
3. **Proiectare** 3
4. **Implementare** 4
   - Clasa Simulation 4
   - Clasa SimulationFrame 5
   - Clasa SimulationFrame2 6
   - Clasa Buton 6
   - Clasa Task 7
   - Clasa Server 7
   - Clasa Scheduler 7
   - Interfața Strategy 8
   - Clasa ConcreteStrategyTime 8
   - Enumerația SelectionPolicy 8
5. **Rezultate** 8
6. **Concluzii** 10
7. **Bibilografie** 10

1. Obiectivul temei

Obiectivul acestei teme este reprezentat de implementarea unei aplicații ce are ca scop proiectarea unui sistem bazat pe cozi penrtu a determina și minimiza timpul de așteptare al clienților. Conform cerințelor problemei, fiecare client nou venit va fi adăugat la coada cu cel mai scurt timp de așteptare la momentul de timp egal cu timpul de sosire al acestuia, urmând ca, după ce a stat în coadă pe prima poziție un timp egal cu timpul său de servire, să fie scos din aceasta, făcând loc următorilor clienți.

Modul de citire al datelor introduse se face simplu, urmând ca textul introdus să fie convertit conform unui șablon în concordanță cu cerințele temei, iar rezultatul va fi afișat atât într-un fișier, cât și într-o interfață grafică care se actualizează la fiecare secundă. Astfel, se va afișa timpul, cozile și clienții din fiecare coadă cu timpul de procesare actualizat (scade timpul de procesare a fiecărui client din cap de coadă la fiecare secună). În cazul introducerii unor date eronate, utilizatorul va fi atenționat cu un mesaj de eroare.

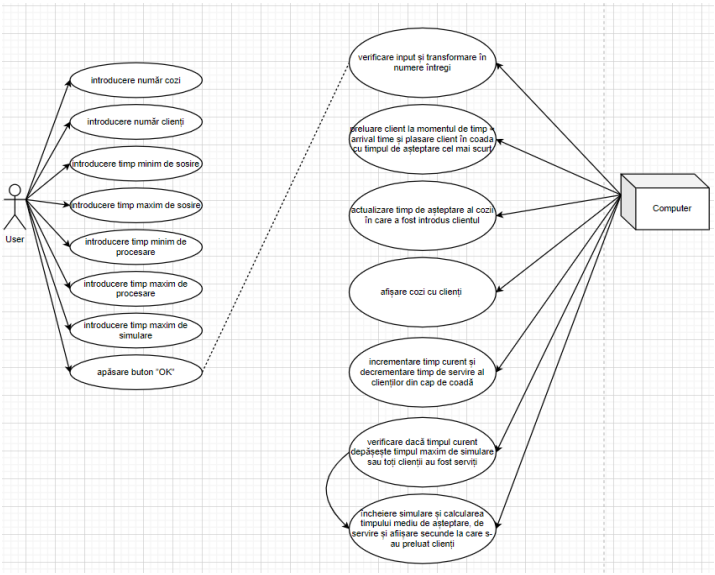
2. Analiza problemei, modelare, scenarii, cazuri de utilizare

Analiza problemei constă, inițial, în înțelegerea cerinței problemei și implicit implementarea unui plan inițial pentru rezolvarea sa care, după cum am menționat mai sus, necesită un algoritm de împărțire a clienților la cozi în funcție de timpul cel mai scurt și timpul de sosire al acestora. Ca prim pas, vom alege clasele necesare (substantive) și metodele reprezentative pentru fiecare (verbe) pentru a putea realiza o aplicație funcțională și ușor de înțeles pentru orice cunoscător de java.

O altă etapă reprezentativă constă în stabilirea intrărilor și ieșirilor aplicației noastre. În cazul de față avem șapte intrări reprezentative pentru numărul de cozi, numărul de clienți, timpul simulării, timpul minim și maxim de sosire și timpul minim și maxim de servire, toate acestea fiind necesare pentru realizarea temei. Datele de intrare sunt reprezentate de JTextField-uri indicare de JLabel-uri cu nume reprezentative. În schimb, pentru ieșire vom avea numărul de cozi, timpul curent și clienții din fiecare coadă în funcție de timpul curent.

Am luat în calcul eventualele erori de introducere a unor date care nu sunt în conformitate cu intrările dorite, iar ca urmare utilizatorul va primi un mesaj de eroare în acest caz.

Use case-ul de mai jos evidențiază pașii ce sunt parcurși atât de utilizator, cât si de computer pentru a se obține rezultatul așteptat în momentul execuției programului.



3. Proiectare

Pentru reprezentarea temei am ales un model MVC (Model View Controller) și am folosit 3 pachete: Model – conține algoritmul propriu-zis de împărțire a clienților la cozi în funcție de timpul de așteptare minim, View – conține interfața grafică a programului și Controler – conține simularea programului, inclusiv metoda main de unde se dă startul execuției acestuia.

Diagrama UML este reperezentată mai jos, iar în ea putem observa clasele proiectului cu legăturile specificate, dar șiu metodele și structurile de date folosite.

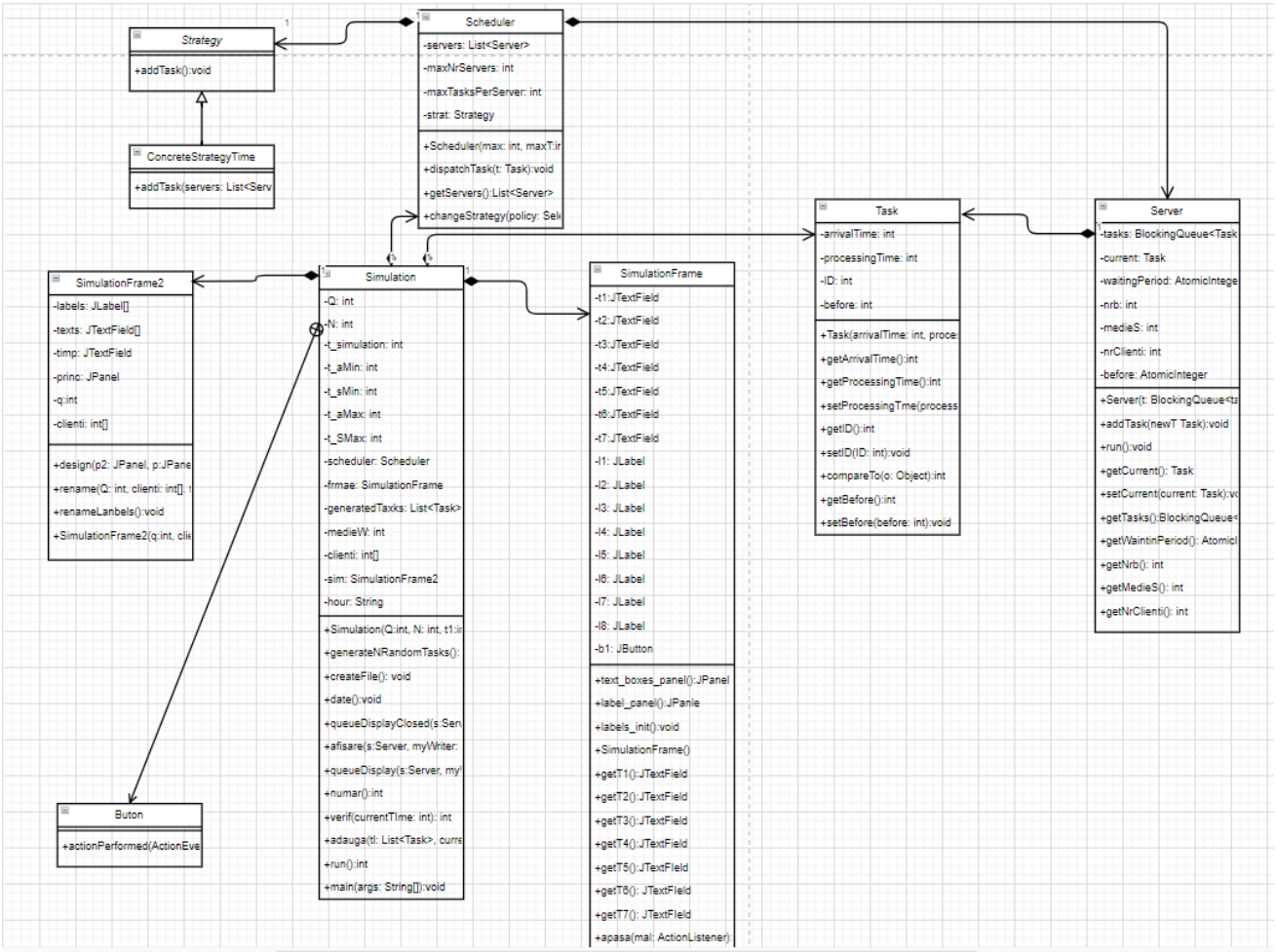
4. Implementare

Descrierea claselor se va face pe baza diagramei UML de clase reprezentată mai sus.

- Clasa Simulation

Această clasă reprezintă clasa de bază a proiectului, clasa de unde se începe simularea, fiind clasa ce conține, pe lângă alteșe și metoda statica main.

În metoda main instanțiem un obiect de tip Simulation cu datele: Q=2, N=4, t_simulation=20, arrival_time=(2,10), processing_time=(2,10) și realizăm crearea fișierului în



care vor fi scrise datele de ieșire, în cazul în care acesta nu există. Tot în această metodă se crează și se dă startul thread-ului menționat mai jos.

Totodată, clasa Simulation implementează Runnable și deci conține suprascrierea metodei run pentru a se putea forma thread-ul inițial al aplicației, thread din care se vor porni celelalte threaduri, cele reprezentative pentru fiecare coadă. În metoda run se realizează scrierea clienților care așteaptă să fie puși la coadă, apelul metodei adaugă, al metodei queuesDisplay și al metodei verif. În run, ținem evidența timpului curent care se va incrementa la fiecare secundă, fapt posibul datorită apelului sleep(1000). Tot aici se reactualizează datele din interfața grafică reprezentată de clasa SimulationFrame2 care reprezintă output-ul aplicașiei, clienții fiind reprezentați prin ”*”.

Metoda adaugă, după cum îi spune și numele, preia clienții cu timpul de sosire egal cu timpul curent și îi adaugă în coada cu timpul de așteptare cel mai scurt, folosindu-se de metoda add a clasei Task, metodă descrisă mai jos, la clasa Task. Tot în această metodă, ținem cont de timpul pe care îl așteaptă fiecare client până să fie servit, adăugându-l la medieW și de secunda la care a fost adăugat în coadă. Adăugarea clienților în cozi este precedată de ștergerea lor din generatedTasks pentru a nu mai fi luați în condiserare în viitor.

Metoda queuesDisplay afișează în fișier conținutul fiecărei cozi și timpul curent, actualizând totodată numărul de clienți ce vor fi afișați în interfața grafică. Dacă o coadă e goală se va afișa mesajul ”closed” lângă aceasta. Tot aici găsim și apelul metodei afișare care afișează în fișier clienții din cozi și queuesDisplayClosed care afișează mesajul closed lângă cozile închise.

Metoda verif apelează metoda numar care contorizează numărul de cozi goale. Dacă numărul de cozi goale este egal cu numărul total de cozi și nu mai avem niciun element în generatedtasks sau timpul curent a ajuns mai mare decât timpul maxim de simulare, vom opri simularea, vom închide frame-ul cu simularea și vom scrie în fișier average waiting time, average serving time și hour, conform cerințelor aplicației.

Constructorul pentru această clasă actualizează datele problemei în funcție de parametrii specificați și creează o nouă interfață grafică pentru afișarea rezultatelor problemei. Tot aici este apelata metoda date.

Metoda date apelează metoda generateNRandomTasks și afișează task-urile nou create.

Metoda generateNRandomTasks generează N clienți aleatori în conformitate cu datele problemei și îi ordonează în funcție de timpul lor de sosire, reactualizându-le ID-ul.

- ## Clasa SimulationFrame

Reprezentativă pentru GUI-ul inițial în care utilizatorul introduce date pentru o noua simulare a programului.

Aceasta conține 7 JtextField-uri, 8 Jlabel-uri și un Jbutton, fapt care poate fi observat în poza de mai jos a interfației grafice în care utilizatorul introduce noi date pentru o nouă execuție a programului.

În constructorul clasei se realizează apelul metodelor text_boxes_panel, labels_init și label_panel. Tot aici se formează panourile interfeței grafice, se setează dimensiunea sa și se afișează.

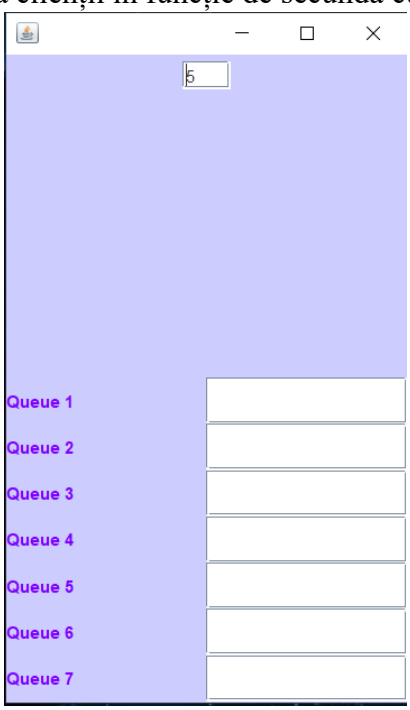
Metoda text_boxes_panel adaugă în panoul principal JtextField-urile menționate mai sus și butonul, setând totodată culorile lor.

Metoda labels_init setează culorile labeluri-lor și a butonului, acestea urmând să fie adăugate în panoul principal prin intermediul metodei label_panel.

Alte metode ale acestei clase sunt getterele getT1, getT2, getT3, getT4, getT5, getT6 și getT7 și metoda aasă care îi adaugă butonului un ActionListener care va fi detaliat în descrierea clasei Buton.

- Clasa SimulationFrame2

Această clasă conține un tablou de Jlabel-uri și unul de JtextField-uri, un JtextField, un Jpanel, un întreg q și un array de clienți tot de tipul int. Aspectul acestei interfețe în care se vor afișa clienții în funcție de secunda curentă, poate fi observat în imaginea de mai jos.



Constructorul acestei clase actulaizează datele de intrare cu parametrii acestuia, apelează metodele renameLabels și design și setează dimensiunea Jframe-ului, setând totodată vizibilitatea sa.

Metoda renameLabels actualizează conținutul label-urilor și al text-field-urilor în funcție de datele de intrare ale aplicației.

Metoda rename este apelată din clasa Simulation, aceasta reactualizând conținutul text-field-urilor si a label-urilor și apelând metoda design.

Metoda design setează culorile Frame-ului, layout-ul panourilor folosite și adaugă label-urile și text-field-urile dorite în interfața grafică.

- Clasa Buton

Această clasă reprezintă o clasă internă pentru clasa Simulation și rescrie metoda actionPerformed. Aici se reactualizează datele de intrare ale aplicației în funcție de datele introduse de utilizator în GUI, se apelează metoda date, createFile, constructorul clasei

Scheduler, menționate mai sus, se reactualizează conținutul array-ului de clienti la 0 și se pornește un nou thread al aplicației, apelându-se, totodatp și metoda rename.

- ## Clasa Task

Aceasta este o clasă reprezentativă pentru clienți și implementează Comparable, ca urmare rescriind metoda compareTo pentru a putea fi folosită la sortarea clienților în ordine crescătoare în funcție de arrivalTime-ul lor.

Constructorul clasei actualizează arrivalTime-ul, processingTime-ul, ID-ul și before-ul clientului.

Alte metode ale aceste clase sunt getterele și setterele pentru că am folosit încapsularea datelor pentru processingTime, ID și before.

- ## Clasa Server

Această clasă este reprezentativă pentru cozi și implemetează Runnable, suprascriind metoda run pentru a fi posibilă crearea unui thread pentru fiecare coadă a aplicației. Metoda run scoate, pe rând capul cozii și îi decrementează processing time-ul până ajunge la 0, progresiv, pe secunde. În cazul în care coada este goală thread-ul este pus pe pauză pentru un interval de o secundă, fapt posibil prin apelul metodei Thread.sleep(1000).

Constructorul clasei inițializează coada de task-uri, numărul ei, media, numarul de clienți și before în funcți de datele introduse de utilizator sau la 0 pentru before, medie și număr clienți.

Metoda addTask actualizează elementul curent al cozii în cazul în care aceasta este goală la task-ul specificat ca parametru. Totodată, se adaugă acest task la coada de task-uri, se adaugă perioada de procesare a acestuia la perioada de procesare a cozii, se incrementează numărul de clienți și se actualizează before-ul task-ului cu valoarea before, urmând sa se adauge la acest before timpul de servire al clientului nou adăugat în coadă. Pentru verificarea corectitudinii, am ales să afișez după fiecare adăugare în coadă și task-ul adăugat.

Pe lângă metodele menționate mai sus am mai adăugat și gettere și settere petru current = elementul curent al cozii, tasks= coada de clienți, waitingPeriod= perioada de servire a cozii, nbr= numărul cozii, medieS= suma tuturor timpilor de procesare dintr-o coadă care va fi folosită la calculul timpului mediu de servire și nrClienti= numărul de clienți adăugați, per total, în coadă.

- ## Clasa Scheduler

Această clasă se ocupă cu distribuția clienților în cozi în funcție de strategia aleasă, în cazul de față fiind reprezentată de adăugarea clienților la coada cu timpul de așteptare cel mai scurt.

Constructorul clasei inișializează maxNrServers, maxTasksPerServer, servers, strategia și formează maxNrServers cozi și un thread pentru fiecare, urmând ca acesta să fie pornit prin apelul metodei start(). MaxNrServers reprezintă numărul maxim de cozi, maxTasksPerServer reprezintă numărul maxim de clienți dintr-o codă, iar servers reprezină cozile.

Metoda dispatchtask apelează metoda addTask a strategiei folosite. Startegia poate fi schimbată prin ntermediul metodei changeStrategy în cazul unei dezvoltări ulterioare a programului.

În plus, avem și getter-ul getServers pentru a obține cozile, metodă necesară datorită folosirii încapsulării datelor.

- Interfața Strategy

Această interfață prezintă metoda addTask care va fi suprascrisa de clasa ConcreteStrategyTime după cum este specificat mai jos.

- Clasa ConcreteStrategyTime

Această clasă implementează interfața Strategy și conține algoritmul de repartizare a clienților la coada cu timpul de procesare cel mai scurt. Acest algoritm este descris in metoda addTask. Am luat un număr întreg minim care va fi setat la numărul primei cozi cu waitingPeriod-ul cel mai mic. Apoi, am adăugat în coada cu numărul mini task-ul nou ce este transmis ca parametru alături de List-ul de cozi.

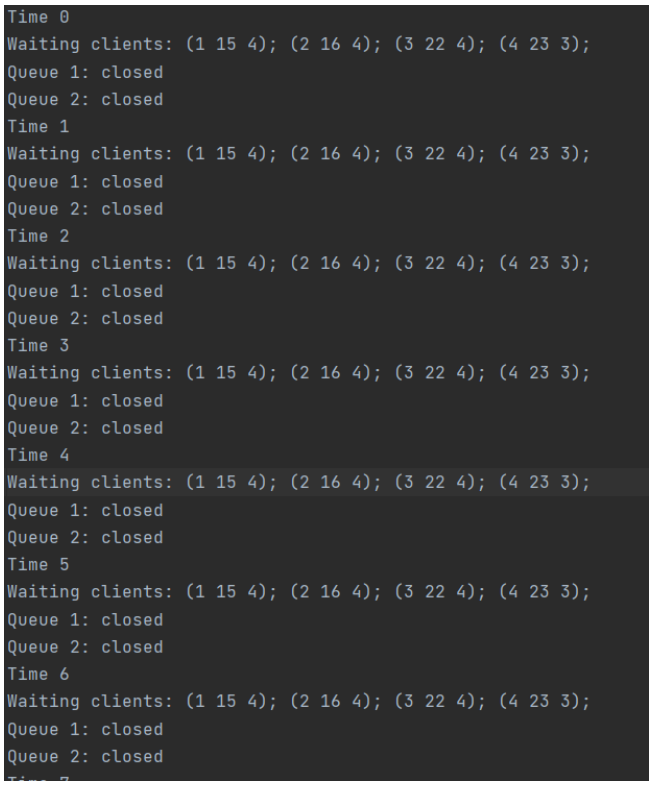
- Enumerația SelectionPolicy

Această enumerație conține posibilele strategii ale plasării clienților în cozi, dar eu am implementat doar strategia SHORTEST_TIME. Astfel, se pot adăuga alte strategii programului în cazul unor dezvoltări ulterioare ale aplicației.

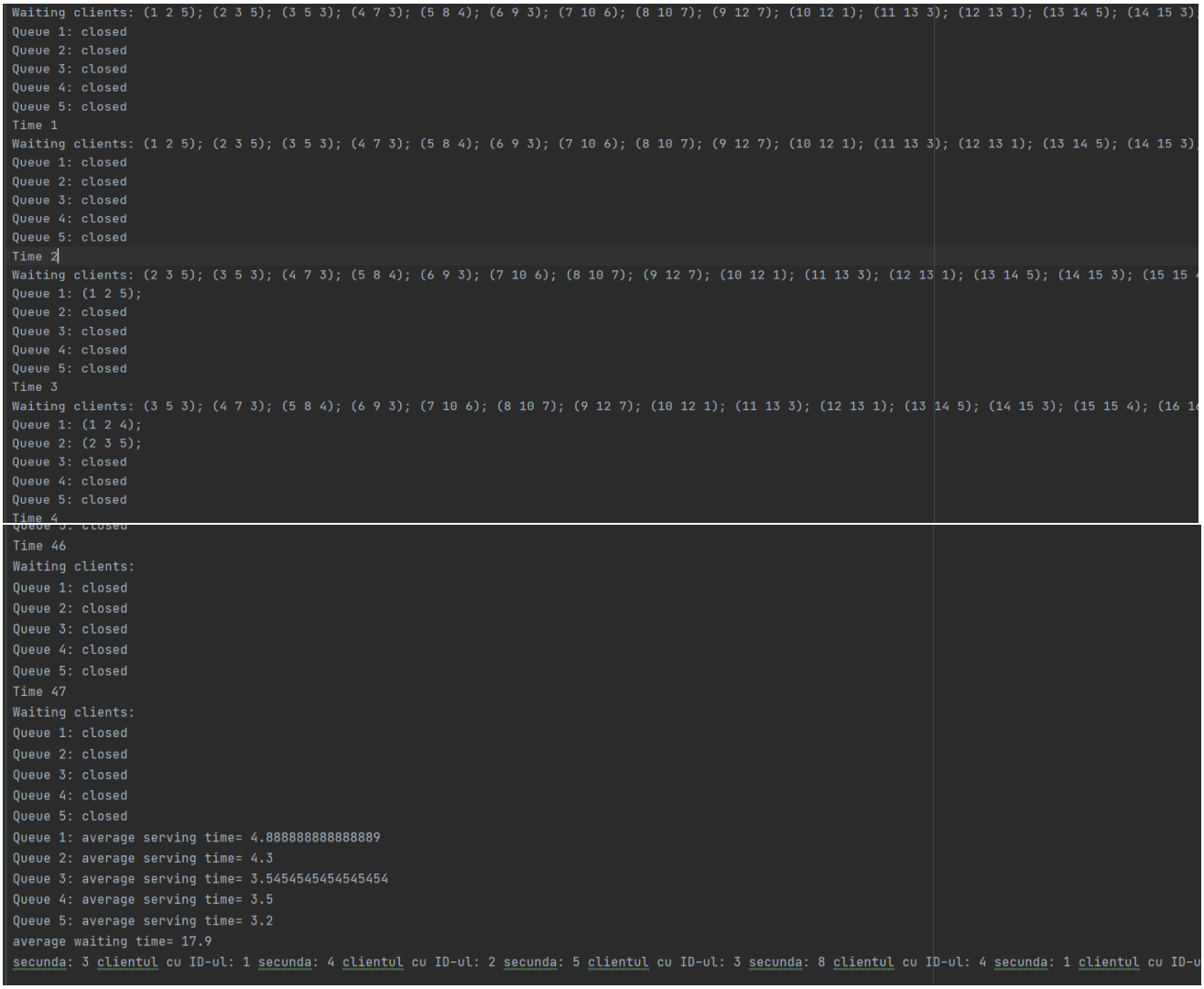
5. Rezultate

Testele le-am realizat în Intellij în funcție de cele trei exemple de simulare date în lucrarea de laborator și am observat corectitudinea rezulatelor. Simulările au pornit în momentul apăsării butonului ”OK” din interfața grafică a programului.

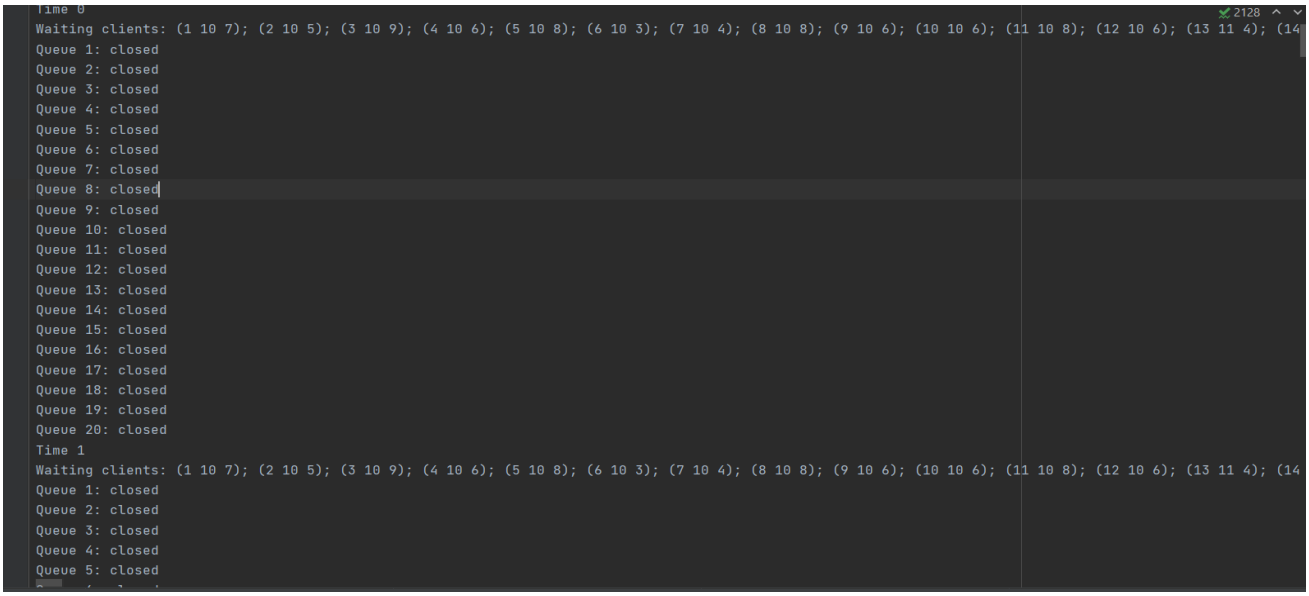
Pentru exemplul 1 se poate observa rezultatul în imaginea de mai jos și în fișierul trimis o dată cu tema.

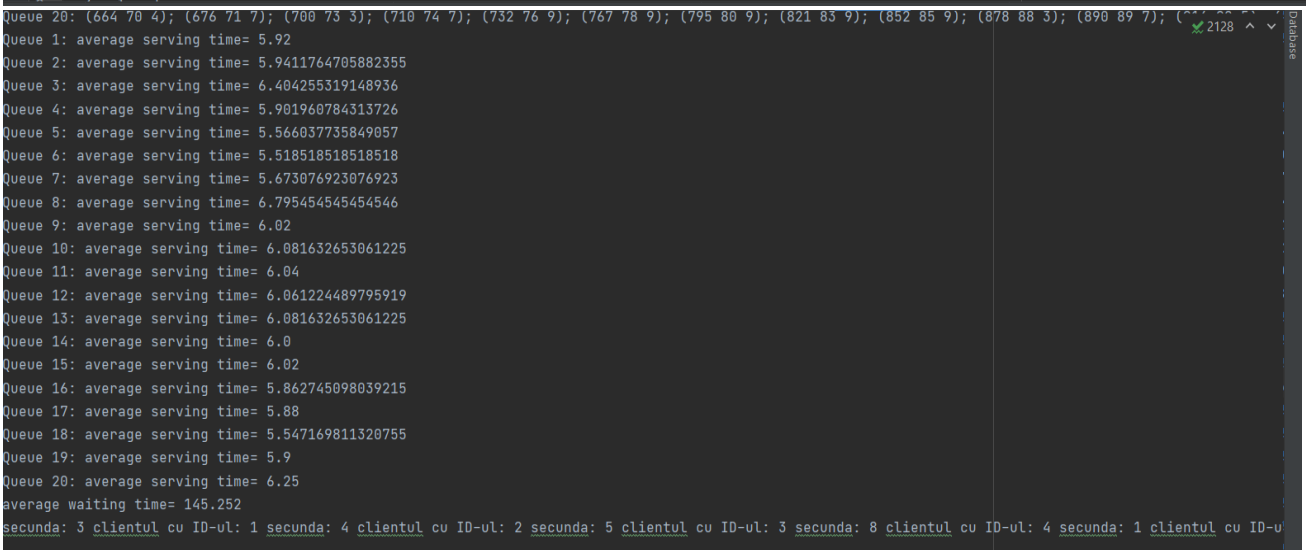


Pentru exemplu 2 se poate observa rezultatul în imaginea de mai jos și în fișierul trimis o dată cu tema.



Pentru exemplul 3 se poate observa rezultatul în imaginea de mai jos și în fișierul trimis o dată cu tema.





6. Concluzii

Din punctul meu de vedere, această tema a reprezentat o adevărată provocare pentru mine, dar am reușit să îi fac față cu brio. Un punct forte al acesteia este reprezentat de faptul că m-a ajutat să înțeleg thread-urile și m-a determinat să recapitulez anumite noțiuni învățate semestrul trecut. Conștientizarea importanței thread-urilor vine ca o urmare a realizării temei cu numărul doi.

7. Bibilografie

Resursele bibliografice sunt reprezentate de suportul de laborator și de suportul de curs de semestrul trecut de la materia Programare Orientată pe Obiect.