

DOCUMENTAȚIE
TEHNICI DE PROGRAMARE
TEMA 2 – Aplicație de gestiune a cozilor

NUME STUDENT: ROTARIU LAURA-ALEXANDRA
GRUPA: 30224-2

CUPRINS

1.	Obiectivul temei	2
2.	Analiza problemei, modelare, scenarii, cazuri de utilizare.....	2
3.	Proiectare.....	4
4.	Implementare.....	6
5.	Rezultate.....	9
6.	Concluzii	10
7.	Bibliografie.....	11

1. Obiectivul temei

Obiectivul principal al temei este de a proiecta și implementa o aplicație de gestionare a cozilor care atribuie clienții la cozi astfel încât timpul de așteptare este minimizat, folosind mecanisme eficiente de alocare a cozilor, cu o interfață grafică dedicată în Java care să ofere utilizatorului o modalitate ușoară de a introduce datele necesare simulării: numărul de clienți, numărul de cozi, timpul de simulare, intervalul în care clienții ajung și intervalul de servire. Utilizatorul poate vedea după încheierea simulării cum au fost distribuiți clienții la cozi și rezultate precum timpul mediu de servire, timpul mediu de așteptare și ora de vârf.

Obiectivele secundare ale temei (pașii care trebuie urmați pentru îndeplinirea obiectivului principal):

- Analizarea problemei și identificarea funcționalităților de bază care vor fi oferite utilizatorilor prin intermediul interfeței grafice simple: introducerea datelor necesare simulării și vizualizarea rezultatelor și a alocării clienților la cozi. Este important în cadrul acestui pas să se stabilească modul simulării: distribuirea clienților la cozi după un anumit criteriu, existența mai multor cozi pentru a procesa clienții în paralel, existența unui fir de execuție separat care va fi responsabil cu păstrarea timpului de simulare și distribuirea clienților la coadă cu timp de așteptare minimizat. Acest pas va fi descris în secțiunea Analiza problemei, modelare, scenarii, cazuri de utilizare.

- Proiectarea aplicației pentru simulare: definirea claselor și a interfețelor necesare, implementarea structurilor de date care stochează clienții și cozile, definirea strategiei pentru a determina la ce coadă să fie pus fiecare client, implementarea logicii pentru așteptarea clienților să fie serviți, implementarea calculului parametrilor care ne interesează (timp mediu de așteptare, de servire, oră de vârf). Este importantă organizarea codului în clase și pachete în funcție de utilitatea acestuia. Acești pași vor fi descriși în secțiunile Proiectare și Implementare.

- Implementarea aplicației pentru simulare: scrierea codului pentru clasele și interfețele necesare proiectate la punctul anterior, implementarea strategiilor și a logicii pentru așteptarea clienților la coadă pentru a fi serviți, scrierea codului Java necesar pentru a introduce datele de către utilizator în interfața grafică. Este necesară utilizarea unor componente standard în interfață, precum panouri, butoane, casete de text, etichete pentru a permite utilizatorilor să introducă datele, să pornească simularea și să vadă rezultatele. Acest pas va fi descris în secțiunea Implementare.

- Testarea pentru a vedea dacă aplicația funcționează corect. Se va testa pe mai multe seturi de date și se vor urmări rezultatele pentru a identifica erori sau îmbunătățiri posibile. Acest pas va fi descris în secțiunea Rezultate.

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

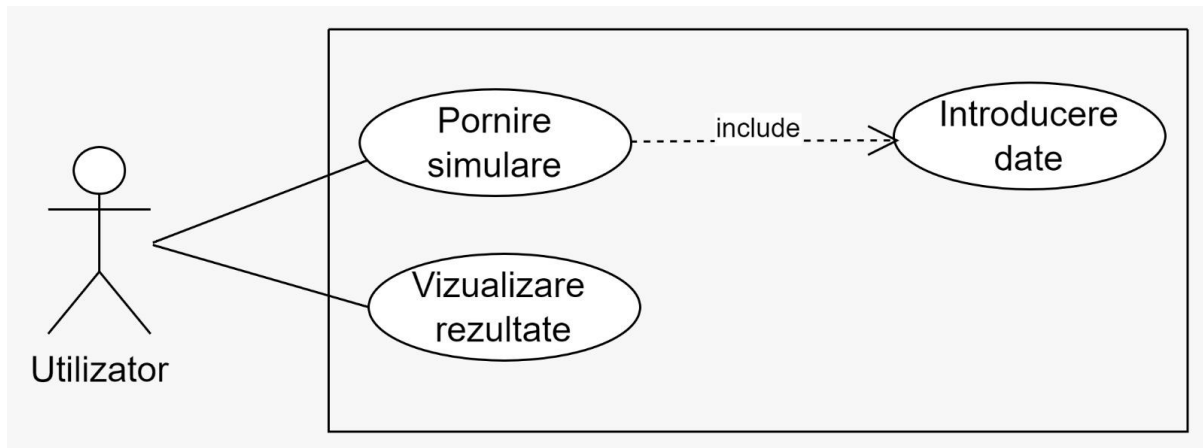
• Cerințe funcționale:

- Aplicația de simulare trebuie să permită utilizatorilor să introducă datele necesare simulării.
- Aplicația de simulare trebuie să permită utilizatorilor să pornească simularea.
- Aplicația de simulare trebuie să calculeze rezultatele de interes: timpul mediu de așteptare, timpul mediu de servire și ora de vârf.
- Aplicația de simulare trebuie să permită utilizatorilor să vadă rezultatele obținute în urma simulării și modul în care au fost alocați clienții la cozi.

• Cerințe non-funcționale:

- Aplicația de simulare trebuie să fie intuitivă și ușor de folosit de către utilizator.
- Aplicația de simulare trebuie să realizeze simularea și să afișeze rezultatele într-un timp scurt.
- Aplicația de simulare trebuie să funcționeze indiferent de sistemul de operare folosit.

- **Cazuri de utilizare:**



- **Descrieri use-case-uri:**

Use case: Pornire simulare(după introducerea datelor necesare pentru simulare)

Actor principal: utilizator

Scenariul principal cu succes:

1. Utilizatorul introduce valorile pentru numărul de clienți, numărul de cozi, intervalul de simulare, timpul minim și maxim de sosire și timpul minim și maxim de servire pentru clienți în interfața grafică.
2. Utilizatorul apasă butonul care pornește simularea din interfața grafică.
3. Aplicația citește datele introduse și începe simularea, iar când este gata, afișează un mesaj pe ecran pentru ca utilizatorul să poată vedea rezultatele obținute și dispunerea clienților la cozi.

Scenariul alternativ: Datele necesare nu sunt introduse sau sunt introduse incorect

- Utilizatorul nu introduce datele necesare pentru a porni simularea sau introduce date care nu sunt numere și astfel este imposibilă simularea.
- Se afișează un mesaj și se cere utilizatorului să introducă datele necesare simulării și scenariul se va întoarce la primul pas.

Use case: Vizualizare rezultate

Actor principal: utilizator

Scenariul principal cu succes:

1. Utilizatorul așteaptă afișarea unui mesaj în interfața grafică care validează terminarea simulării și apasă pe butonul “See log of events” și vede cum au fost dispuși clienții la cozi și pe butonul “See results” și vede datele de interes (timpul mediu de așteptare, timpul mediu de servire și ora de vârf). La apăsarea acestor butoane se deschide câte un fișier cu datele respective.

Scenariul alternativ:

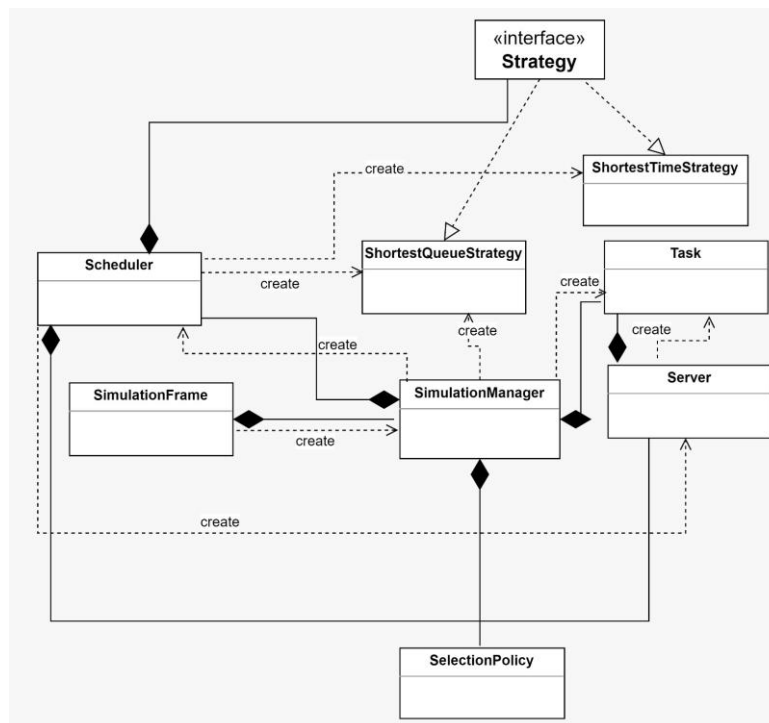
- Utilizatorul apasă prea repede și simularea nu este finalizată și trebuie să mai aștepte pentru a vedea rezultatele.

3. Proiectare

- **Proiectarea OOP a aplicației:**

Pentru a rezolva specificațiile problemei, am ales să implementez clasele: Server, Task, Scheduler, SimulationManager, SimulationFrame, interfața Strategy implementată de clasele ShortestQueueStrategy și ShortestTimeStrategy și enumerația SelectionPolicy, incluse în diferite pachete în funcție de utilitatea lor. Clasa Server este inclusă în pachetul org.example.Model și implementează interfața Runnable, putând fi executată cu un fir de execuție separat. Clasa Task este inclusă tot în pachetul org.example.Model și aici am implementat un model de sarcină care trebuie procesată de un sever. Clasele Scheduler, SimulationManager, ShortestTimeStrategy, ShortestQueueStrategy, interfața Strategy și enumerația SelectionPolicy sunt incluse în pachetul org.example.BusinessLogic. Clasa Scheduler are o listă de servere și o strategie pentru a aloca sarcinile acelor servere și pornește câte un fir de execuție nou pentru fiecare server, permițându-le să fie executate în paralel. Interfața Strategy este implementată de clasele ShortestQueueStrategy și ShortestTimeStrategy care definesc modalitatea în care vor fi dispuși clienții la coadă. Clasa SimulationManager implementează logica de simulare a unui sistem de cozi cu mai multe servere care deservește o serie de sarcini generate aleatoriu și utilizează clasa Scheduler pentru aceasta și enumerația SelectionPolicy. Clasa SimulationFrame este inclusă în pachetul org.example.GUI și aici am implementat interfața grafică în care utilizatorul va introduce datele necesare simulării și va putea cere deschiderea fișierelor cu rezultatele simulării.

- **Diagrama UML de clase:**



- **Structurile de date folosite:**

Structurile de date pe care le-am utilizat în implementarea aplicației sunt:

- *ArrayList<T>*: pentru a stoca lista de servere și lista de sarcini generate;
- *BlockingQueue*: coadă blocantă care permite mai multor thread-uri să acceseze aceeași coadă în mod concurent și care blochează accesul la ea atunci când e plină sau goală; am utilizat-o pentru a stoca obiecte de tipul clasei Task care sunt adăugate de mai multe thread-uri;
- *enum*: pentru a selecta politica de alegere a cozii la care se va pune clientul;
- *Task*: clasa creată pentru a stoca clienții;
- *Server*: clasa creată pentru a gestiona clienții de la fiecare coadă;
- *ShortestQueueStrategy* și *ShortestTimeStrategy*: clase create pentru a defini strategia;
- *FileWriter*: pentru a scrie rezultatele simulării în fișiere;
- *Scheduler*, *SimulationManager*: clase create pentru a planifica și simula cozile și distribuirea la cozi a clienților.

- **Interfețele definite**

- Am implementat interfața Strategy care definește o metodă addTasks care primește două argumente (o listă de obiecte Server și un obiect Task). Scopul metodei este de a adăuga sarcina specificată în lista de servere date, utilizând o anumită strategie. Datele specifice strategiei nu sunt specificate în interfață, urmând a fi implementate în cadrul claselor care o implementează. Interfața este utilă deoarece se pot defini mai multe strategii pentru dispunerea clienților la coadă. Clasele ShortestQueueStrategy și ShortestTimeStrategy implementează interfața definită și metoda addTasks în funcție de necesitățile strategiei.

4. Implementare

- *Clasa Task* - definește un client și implementează interfața Comparable pentru a sorta clienții. Atributele clasei sunt private și sunt ID-ul de tip întreg care reprezintă un identificator unic pentru fiecare instanță de tipul Task, arrivalTime și serviceTime de tip întreg care sunt momentul de sosire, respectiv timpul de serviciu pentru fiecare obiect de acest tip. Clasa are un constructor cu trei parametri care inițializează atributele precizate anterior.

Metodele clasei Task sunt:

- void setServiceTime(int serviceTime): setează valoarea atributului serviceTime de tip privat;
- int getID(): accesează atributul ID de tip privat;
- int getServiceTime(): accesează atributul serviceTime de tip privat;
- int getArrivalTime(): accesează atributul arrivalTime de tip privat;
- String toString(): suprascrie metoda toString din clasa Object astfel încât să se afișeze sarcina(clientul) sub formă de șir de caractere;
- int compareTo: metodă ce trebuie implementată deoarece clasa implementează interfața Comparable<Task> și care compară două instanțe de tipul clasei Task pe baza momentului de sosire și dacă acesta este egal, pe baza timpului de servire.

- *Clasa Server* – implementează interfața Runnable pentru a fi executată într-un fir de execuție separat. Atributele clasei sunt private: o coadă blocantă tasks care conține obiecte de tipul Task(sarcinile), un AtomicInteger waitingPeriod ce reprezintă perioada totală de așteptare a sarcinilor din coadă și maxTasks de tip întreg ce reprezintă numărul de sarcini ce pot fi adăugate în coadă. Clasa are un constructor care primește numărul maxim de sarcini și inițializează coada și AtomicInteger-ul waitingPeriod.

Metodele sunt:

- void addTask(Task task): care este o metodă sincronizată ce adaugă o sarcină în coadă și actualizează waitingPeriod cu durata serviciului acelei sarcini;

- void run(): metoda din interfața Runnable care rulează într-un fir de execuție separat și preia sarcinile din coadă, executându-le pe rând;
- void decrementWaitingPeriod(): metodă ce actualizează variabila waitingPeriod prin decrementare cu 1;
- void removeTask(int i): metodă sincronizată ce elimină o sarcină din coadă prin parcurgerea sa cu un iterator și eliminarea sarcinii de la indexul specificat ca parametru;
- Task [] getTasks(): metodă ce returnează un array cu sarcinile din coadă;
- AtomicInteger getWaitingPeriod() metodă ce returnează valoarea atributului waitingPeriod.

- *Clasele ShortestQueueStrategy și ShortestTimeStrategy* – implementează interfața Strategy care definește comportamentul pentru adăugarea sarcinilor la o listă de servere. Metoda addTasks primește o listă de servere și o sarcină și implementează strategia pentru adăugarea sarcinii la unul dintre serverele din listă. Clasa ShortestQueueStrategy definește comportamentul pentru adăugarea sarcinilor la serverul cu cea mai scurtă coadă. Parcurge lista de servere pentru a găsi serverul cu cea mai scurtă coadă și adaugă sarcina la acel server. Clasa definește comportamentul pentru adăugarea sarcinilor la serverul cu cel mai scurt timp de așteptare. Parcurge lista de servere pentru a găsi serverul cu cel mai scurt timp de așteptare și adaugă sarcina la acel server. Clasele nu au atribute, doar metoda addTasks(List <Server> servers, Task task).

- *Clasa Scheduler* – această clasă definește planificatorul care se ocupă cu distribuirea sarcinilor către servere, folosind o strategie specifică, determinată de obiectul de tip SelectionPolicy atribuit la metoda changeStrategy. Clasa are ca unul dintre atribute de tip privat o listă sincronizată de servere, care sunt inițializate în constructorul clasei Scheduler, fiind create maxNoServers servere care pot avea maxTasksPerServer sarcini, număr specificat de parametrul din constructor. Fiecare server este asociat cu un Thread care rulează metoda run() din clasa Server. De asemenea, clasa are ca atribut un obiect de tipul clasei Strategy care este inițializat în funcție de valoarea parametrul SelectionPolicy din metoda changeStrategy(SelectionPolicy).

Clasa are metodele:

- void changeStrategy(SelectionPolicy policy): pentru a inițializa strategia;
- void dispatchTask(Task task): care adaugă un nou obiect Task în sistemul de cozi utilizând strategia determinată prin obiectul de tip Strategy asociat;
- List<Server> getServers(): returnează lista de servere din sistem.

- *Clasa SimulationManager* – implementează logica de simulare pentru un sistem de cozi. Aceasta utilizează o instanță a clasei Scheduler pentru a distribui sarcinile în mai multe cozi în funcție de politica de selecție specificată. Clasa mai are ca și atribute o listă de sarcini generate, două fișiere pentru a scrie rezultatele și dispunerea clienților la cozi, limita de timp pentru simulare, un JTextField pentru a afișa în interfața grafică când se termină simularea și mai multe variabile de tip întreg și double pentru a calcula timpii de interes (timpul mediu de așteptare, timpul mediu de servire și ora de vârf). Constructorul clasei primește numărul de servere și numărul de clienți, intervalul de timp pentru sosirea clienților și intervalul pentru servirea clienților și inițializează o listă de sarcini, strategia și generează o listă de sarcini aleatorii pentru a fi procesate. Clasa are metodele:

- void generateNRandomTasks(int numberOfClients, int minArrivalTime, int maxArrivalTime, int minServiceTime, int maxServiceTime): care generează un număr de sarcini aleatorii, folosind parametri specificați și stochează aceste sarcini în lista numită generatedTasks; tot aici se calculează și timpul mediu de servire;
- void processTasks(): care verifică fiecare server dacă are task-uri în coadă sau este liber, iar dacă este liber, scrie în fișier că serverul este închis, altfel decrementează timpul de servire a primului task din coadă și timpul de așteptare pentru coada respectivă și se scrie în fișierul fileLogs dispunerea clienților la cozi. Când un task își termină timpul de servire, este eliminat din coada serverului respectiv. Aici se calculează și timpul mediu de așteptare și ora de vârf;
- void writeResultsFile(): metodă în interiorul căreia se scriu în fișierul fileResults rezultatele simulării (timpii de interes);
- void writeLogEvents(): metodă în interiorul căreia se scriu în fișierul fileLogs timpul curent și clienții care încă nu au dispuși la cozi;
- void run(): metodă care implementează logica de bază a simulării, cum ar fi pornirea simulării, întreruperea acesteia. Simularea are loc pentru un timp introdus de utilizator în interfața grafică (timeLimit). Se parcurge lista de clienți generați și sunt dispuși la coadă clienții care au timpul de servire egal cu timpul

curent al simulării, se asignează clientul unei cozi, după care se elimină din listă. Simularea se încheie când se termină timpul de simulare sau când nu mai sunt clienți nerepartizați și toate cozile sunt închise;

- void startSimulation(): inițializează două fișiere, log_events.txt și results.txt pentru a scrie evenimentele din simulare și rezultatele finale. Această metodă trebuie apelată la începerea simulării;
- void setTimeLimit(int timeLimit): pentru a seta timpul de simulare a aplicației.

- *Clasa SimulationFrame*- implementează interfața grafică pentru o aplicație de simulare a cozilor. Această clasă are ca și câmpuri un obiect de tipul SimulationManager pentru a apela metoda din această clasă la apăsarea unui buton adăugat pentru acest scop, o fereastră JFrame și 8 casete text JTextField. Clasa are un constructor unde sunt setate elementele interfeței grafice (panou, etichete, casete, text, butoane, separatori) și sunt adăugați ascultători butoanelor pentru a fi efectuate operații la apăsarea lor. Când se apasă pe butonul de începere a simulării se citesc datele introduse de utilizator și se inițializează un obiect de tipul clasei SimulationManager și se pornește un fir de execuție și se apelează metoda startSimulation() din acea clasă. La apăsarea butoanelor pentru vizualizarea rezultatelor simulării se deschide fișierul text corespunzător. Clasa are:

- metoda statică main care creează un obiect de tipul SimulationFrame și setează fereastra JFrame ca fiind vizibilă.

Pornirea aplicației are loc în această clasă.

Implementarea interfeței grafice:

Interfața grafică a aplicației de simulare este simplă și ușor de utilizat. Fereastra are ca titlu Queue Managemen. Există șapte casete text pentru a introduce datele, interval de sosire al clienților, intervalul de servire al clienților, timpul maxim de simulare, numărul de clienți și numărul de cozi și o casetă text pentru afișarea unui mesaj când se termină simularea. Sunt 3 butoane: butonul Start Simulation pentru a porni aplicația și a începe simularea și butoanele See log of events și See results, la apăsarea cărora se va deschide fișierul corespunzător pentru a vedea dispunerea clienților la coadă, respectiv timpii de interes. Am folosit etichete pentru a preciza utilitatea fiecărei casete text și am denumit butoanele specific și am folosit separatori pentru delimitare.

Utilizatorul trebuie să introducă datele necesare simulării în casetele text corespunzătoare și să apese butonul pentru începerea simulării, să aștepte afișarea unui mesaj conform căruia poate verifica fișierele și să apese pe butoanele ce deschid fișierele. După apăsarea butonului See results, caseta text cu mesajul se va curăța. Dacă utilizatorul nu introduce datele sau introduce altceva în afară de numere, se va afișa un mesaj cu JOptionPane și utilizatorul va putea introduce datele.

Queue management

Min arrival time: Max arrival time:

Min service time: Max service time:

Max simulation time:

Number of queues:

Number of clients:

Wait for the simulation to stop until a message is displayed below!


Start simulation See log of events See results

5. Rezultate

Am testat aplicația pe aceste seturi de date de intrare din tabelul de mai jos și am analizat fișierele text generate pentru a vedea dacă repartizarea a fost efectuată în mod corect. Am inclus fișierele generate cu evenimentele și rezultatele în repository-ul meu. De asemenea, am testat aplicația și pe alte exemple. În fișierul results.txt putem vedea timpii de interes și în fișierul log_events.txt putem urmări punerea și scoaterea din cozi, precum și decrementarea timpului de servire pentru fiecare client pus la coadă.

Test 1	Test 2	Test 3
$N = 4$ $Q = 2$ $t_{simulation}^{MAX} = 60 \text{ seconds}$ $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [2, 30]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [2, 4]$	$N = 50$ $Q = 5$ $t_{simulation}^{MAX} = 60 \text{ seconds}$ $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [2, 40]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [1, 7]$	$N = 1000$ $Q = 20$ $t_{simulation}^{MAX} = 200 \text{ seconds}$ $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [10, 100]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [3, 9]$

Un exemplu de utilizare și o parte din fișierul log_events.txt:

 Queue management

Min arrival time:

Max arrival time:

Min service time:

Max service time:

Max simulation time:

Number of queues:

Number of clients:

Wait for the simulation to stop until a message is displayed below!

Check the log of events and the results!

Start simulation

See log of events

See results

```

log_events - Notepad
File Edit Format View Help
Time 0
Waiting clients:
(4,7,4) (2,13,2) (3,16,2) (1,24,3)
Queue 1: closed
Queue 2: closed
Time 1
Waiting clients:
(4,7,4) (2,13,2) (3,16,2) (1,24,3)
Queue 1: closed
Queue 2: closed
Time 2
Waiting clients:
(4,7,4) (2,13,2) (3,16,2) (1,24,3)
Queue 1: closed
Queue 2: closed
Time 3
Waiting clients:
(4,7,4) (2,13,2) (3,16,2) (1,24,3)
Queue 1: closed
Queue 2: closed
Time 4
Waiting clients:
(4,7,4) (2,13,2) (3,16,2) (1,24,3)
Queue 1: closed
Queue 2: closed
Time 5
Waiting clients:
(4,7,4) (2,13,2) (3,16,2) (1,24,3)
Queue 1: closed
Queue 2: closed
Time 6
Waiting clients:
(4,7,4) (2,13,2) (3,16,2) (1,24,3)
Queue 1: closed
Queue 2: closed
Time 7
Waiting clients:
(2,13,2) (3,16,2) (1,24,3)
Queue 1: [(4,7,4)]
Queue 2: closed
Time 8
Waiting clients:
(2,13,2) (3,16,2) (1,24,3)
Queue 1: [(4,7,3)]
Queue 2: closed
Time 9
Waiting clients:
(2,13,2) (3,16,2) (1,24,3)
Queue 1: [(4,7,2)]
Queue 2: closed
Time 10
Waiting clients:
(2,13,2) (3,16,2) (1,24,3)
Queue 1: [(4,7,1)]
Queue 2: closed
Time 11
Waiting clients:
(2,13,2) (3,16,2) (1,24,3)
Queue 1: closed
Queue 2: closed

```

6. Concluzii

În concluzie, proiectarea și implementarea unei aplicații menite să simuleze o serie de clienți care ajung pentru a fi serviți, dispunerea lor la cozi, așteptarea, servirea și plecarea lor din cozi este destul de dificilă atât din punct de vedere al implementării, cât și al interfeței grafice.

Această temă a ajutat la recapitularea conceptelor legate de Programarea orientată pe obiect, la acumularea unor noi cunoștințe, întrucât am învățat din această temă cum se folosește structura de date `BlockingQueue` și tipul de date `AtomicInteger` și la familiarizarea lucrului cu thread-uri în Java. De asemenea, am învățat că este foarte important să fie urmați pașii următori pentru a ajunge la finalizarea sarcinii: analizarea problemei și identificarea funcționalităților aplicației de simulare, proiectarea aplicației de simulare, apoi implementarea acesteia și testarea pentru a vedea dacă funcționează fără erori.

Posibile dezvoltări ulterioare ar fi afișarea în timp real a evoluției cozilor și a dispunerii clienților la coadă.

7. Bibliografie

1. *Fundamental Programming Techniques – Suport Presentations and Lectures* - <https://dsrl.eu/courses/pt/>
2. *BlockingQueue Interface in Java* - <https://www.geeksforgeeks.org/blockingqueue-interface-in-java/>
3. *AtomicInteger Class* - https://www.tutorialspoint.com/java_concurrency/concurrency_atomic_integer.htm
4. *Java Threads* - <https://www.geeksforgeeks.org/java-threads/>
5. *Synchronization in Java* - <https://www.mygreatlearning.com/blog/synchronization-in-java/>
6. <https://stackoverflow.com/>