DOCUMENTATIE

TEMA 2

**QUEUES MANAGEMENT APPLICATION USING THREADS AND SYNCHRONIZATION MECHANISMS**

NUME STUDENT: MOLNÁR ZSANETT-INGRID

GRUPA: 302210

# CUPRINS

[1. Obiectivul temei 3](file:///E:\Facultate\Anul%202\SEM%202\TP\PT2021-2022_Documentation_Template.doc#_Toc95297885)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 3](file:///E:\Facultate\Anul%202\SEM%202\TP\PT2021-2022_Documentation_Template.doc#_Toc95297886)

[3. Proiectare 3](file:///E:\Facultate\Anul%202\SEM%202\TP\PT2021-2022_Documentation_Template.doc#_Toc95297887)

[4. Implementare 3](file:///E:\Facultate\Anul%202\SEM%202\TP\PT2021-2022_Documentation_Template.doc#_Toc95297888)

[5. Rezultate 3](file:///E:\Facultate\Anul%202\SEM%202\TP\PT2021-2022_Documentation_Template.doc#_Toc95297889)

[6. Concluzii 3](file:///E:\Facultate\Anul%202\SEM%202\TP\PT2021-2022_Documentation_Template.doc#_Toc95297890)

[7. Bibliografie 3](file:///E:\Facultate\Anul%202\SEM%202\TP\PT2021-2022_Documentation_Template.doc#_Toc95297891)

**1. Obiectivul temei**

Principalul obiectiv a temei este realizarea unei aplicații care are ca scop analiza sistemelor bazate pe cozi de așteptare și distribuirea unui set de clienți generat aleatoriu către aceste cozi astfel încât timpul de așteptare să fie minim.

Pentru a îndeplini obiectivul principal, trebuie urmați câțiva pași esențiali, și anume obiectivele secundare:

* **Analiza problemei și identificare cerințelor:** constă în identificarea principalelor obiecte și atributelor acestora (se va detialia în capitolul: *2*. *Analiza problemei, modelare, scenarii, cazuri de utilizare*)
* **Proiectare**: constă în proiectarea unei soluții care să se plieze pe toate cazurile pe care le introduce utilizatorul și se referă la structura, arhitectura proiectului (se va detalia în capitolul: *3. Proiectare)*
* **Implementare:** se referă la clasele și metodele utilizate pentru implementarea simulatorului de cozi și crearea unei interfațe user-friendly (se va detalia în capitolul: *4. Implementare)*
* **Testare:** constă în verificarea corectitudinii implementării realizate și observarea rezultatelor finale, concluziile (se va detalia în capitolul: *5. Concluzii)*

# 

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

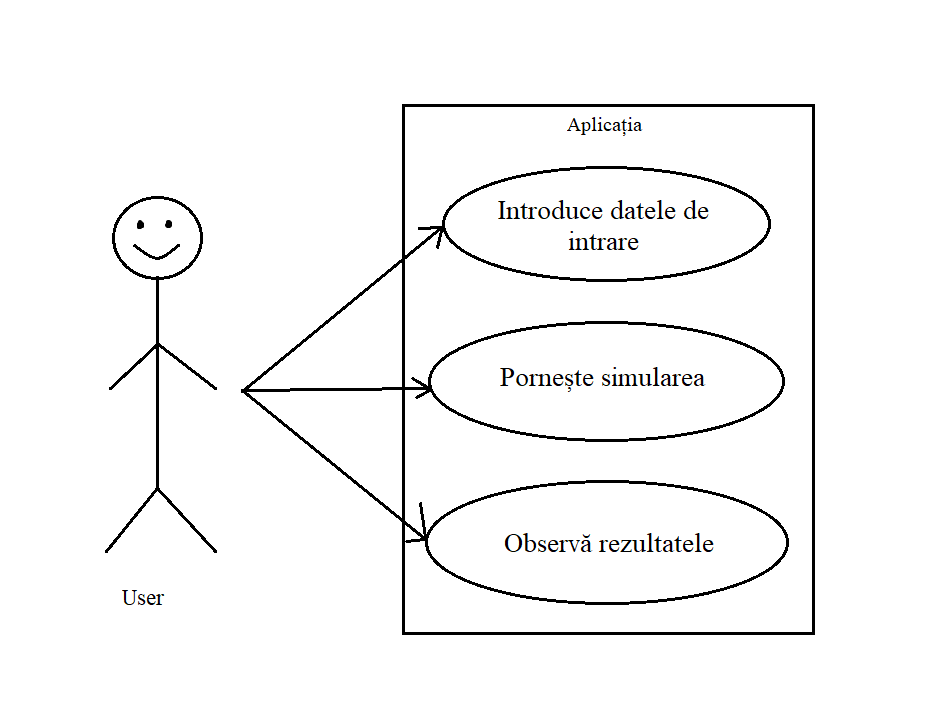
Cerințele funcționale ale proiectului sunt:

* Aplicația de simulare trebuie să permită utilizatorului să configureze simularea
* Aplicația de simulare trebuie să permită utilizatorului să pornească simularea
* Aplicația de simulare trebuie să afișeze evoluția cozilor în timp real

Cerințele non-funcționale ale proiectului sunt:

* Aplicația de simulare trebuie să fie intuitivă și ușoară de folosit de către utilizator

Use-caseurile sunt niște acțiuni sau evenimente care definesc interacțiunile dintre un rol și un sistem pentru a îndeplini o cerință. Aceasta diagramă use-case reprezintă use-caseurile și actorii proiectului (în cazul nostru un singur actor există, și anume userul aplicației).

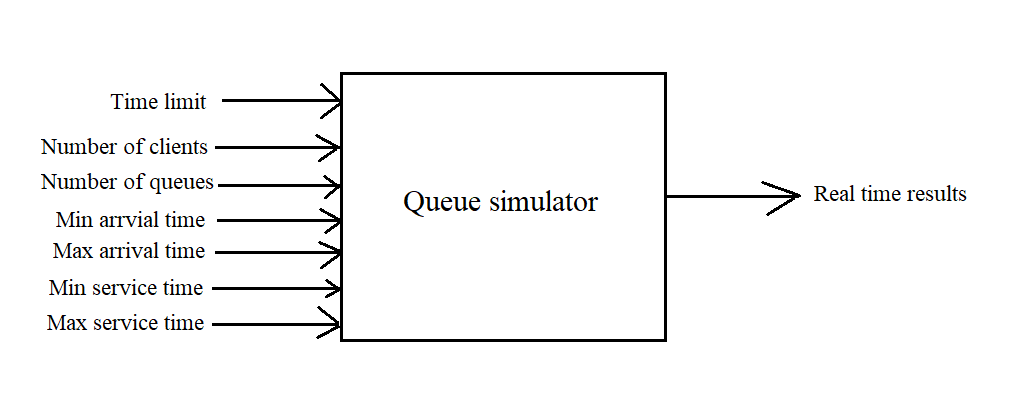


Descrierea use-caseurilor:

1. Utilizatorul introduce toate datele necesare pentru simulare în câmpurile: time limit, number of clients, number of queues, min și max arrival time, min și max service time.
2. Utilizatorul apasă butonul de start
3. Aplicația afișează evoluția cozilor în timp real

Odată ce s-a terminat pasul 3, utilizatorul are două posibilități: ori apasă butonul de reset, astfel revenind la pasul 1, ori apasă butonul start din nou, pornind o nouă simulare cu aceleași date de intrare. Deși datele de intrare sunt aceleași, rezultatul probabil va fi diferit, deoarece clienții se generază aleator.

**3. Proiectare**



Pentru a realiza funcționalitățile simulatorului de cozi, avem nevoie de 7 intrări: **Time Limit**, care e durata simulării, **Number of Clients** (numărul de clienți), **Number of Queues** (numărul cozilor), **Min arrival time** (timpul minim de sosire a clienților), **Max arrival time** (timpul maxim de sosire a clienților), **Min service time** (timpul minim de servire a unui client), **Max service time** (timpul maxim de servire a unui client).

Modelele arhitecturale definesc structuri pentru sisteme software în termeni de subsisteme predefinite și responsabilităților acestora.

Pentru proiectarea simulatorului de cozi am ales modelul arhitectural de implementare **MVC (Model View Controller)**, care împarte aplicația noastră în trei părți:

* **Componentele Model:** încapsulează datele de bază și funcționalitățile
* **Componentele View:** afișează informații utilizatorului, obține datele pe care le afișează de la Model
* **Controller:** Fiecărui View îi este asociată o componentă Controller. Controllerele primesc intrarea de obicei sub forma unor evenimente care denotă mișcarea mouse-ului, activarea butoanelor mouse-ului sau intrarea de la tastatură. Evenimentele sunt traduse în cereri de service, care sunt trimise fie la Model sau la View

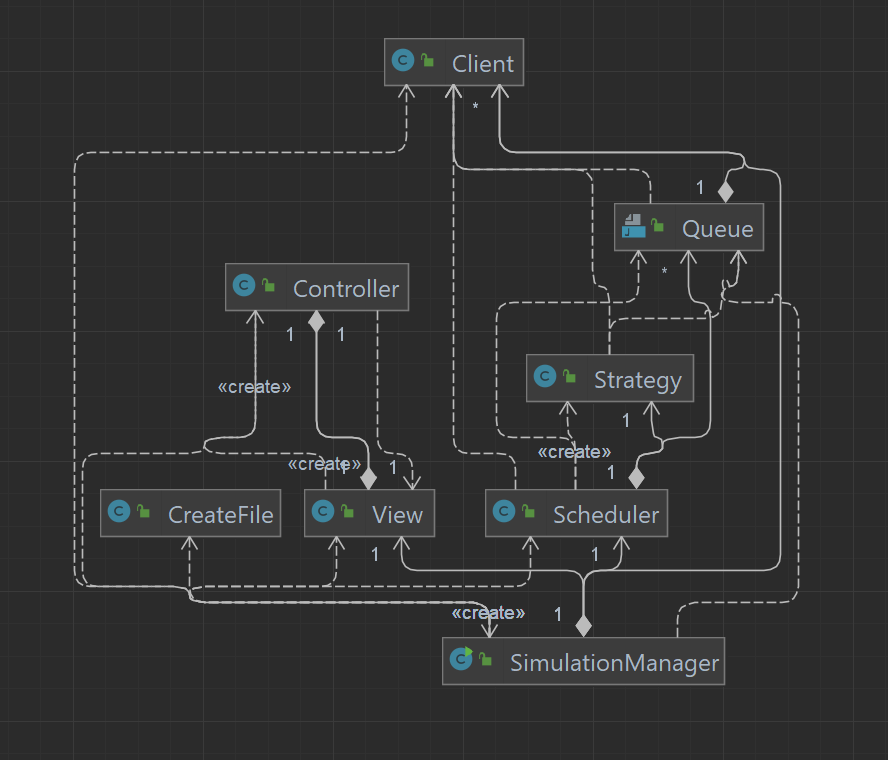
Clasele pe care le-am utilizat pentru realizarea acestei structuri sunt următoarele:

* **Clasa Client:** Clientul reprezintă obiectul principal care participă la simulare. Pe ei se vor efectua acțiuni: se vor adăuga și se vor șterge din cozi. Clienții aplicației se generează aleator. Această clasă conține atributele și metodele caracteristice unui client.
* **Clasa Queue:**  Clasa conține atributele și metodele caracteristice unei cozi. O codă poate conține mai mulți clienți.
* **Clasa Scheduler:** În această clasă se vor crea atâtea cozi cât se precizează de către utilizator. Tot aici, pentru fiecare coadă, se pornește câte un thread, iar când se termină simularea, se vor opri aceste threaduri.
* **Clasa Strategy:** Această clasă va adăuga clienții în coada potrivită, adică în coada unde timpul de așteptare este minim în momentul respectiv (adică unde trebuie să aștepte cel mai puțin).
* **Clasa SimulationManager:** Aceasta este clasa principală care realizează funcționalitatea aplicației de simulare. Aici se va crea threadul principal, se vor genera clienți aleatorii, respectiv e clasa Main a proiectului.

Clasele prezentate mai sus realizează funcționalitățile aplicației, astfel se încadrează la componenta de **Model** a structurii MVC. Pe lângă clasele Model, avem următoarele clase:

* **Clasa View:** în această clasă este realizată interfața utilizatorului (GUI). În modelul MVC, aceasta reprezintă componenta **View**.
* **Clasa Controller:** această clasă “controlează” acțiunile aplicației, adică în cazul apăsării unui buton se ocupă de evenimentele, acțiunile care trebuie să urmeze pentru ca aplicația noastră să îndeplinească cerințele și să funcționeze corect. În arhitectura MVC, această clasă reprezintă **Controller**-ul.
* **Clasa CreateFile:** această clasă nu are un rol esențial în funcționalitatea proiectului, are doar rolul de a scrie în fișiere rezultatele simulărilor.

Diagrama UML a proiectului:



**4. Implementare**

Pentru înțelegerea corectă și bună a funcționalității Aplicației de simulare a cozilor, în acestă secțiune se vor descrie detaliat clasele menționate în secțiunea anterioară.

* **Clasa Client:**

Această clasă implementează obiectul Client, care reprezintă un element dintr-o coadă. Clientul reprezintă obiectul principal care participă la simulare. Pe ei se vor efectua acțiuni: se vor adăuga și se vor șterge din cozi. Clasa are următoarele câmpuri:

- **int ID:** un număr unic caracteristic fiecărui client.

- **int arrivalTime:** timpul sosirii clientului.

- **int serviceTime:** timpul necesar servirii clientului, timpul cât stă la capul cozii.

Clasa are următoarele metode:

* **public Client(int ID, int arrivalTime, int serviceTime):** constructor
* **gettere, settere**
* **public int compareTo(Client o):** compară arrivalTime-ul a doi clienți și returnează diferența acestora
* **public String toString()**

Clasa Client implementează interfața Comparable. Acest lucru ne va fi util atunci când lista de clienți va trebui sortată crescător după timpul de sosire. Este obligatorie să apară metodele interfeței și în clasa care o implementează. În cazul nostru, metoda **compareTo** e această metodă.

* **Clasa Queue:**

Aceasta este clasa destinată cozilor, elemente esențiale proiectului nostru. O coadă poate conține mai mulți clienți. Clasa implementează interfața Runnable, deoarece pentru fiecare coadă va exista un thread separat. Fiecare coadă va procesa în paralel clienții.

Clasa are următoarele câmpuri:

- **int ID:** un număr unic, caracteristic fiecărei cozi.

**- BlockingQueue<Client> clients:** coada de clienți a cozii.

**-** **AtomicInteger actualWaitingTime:** timpul de așteptare actual a cozii. Este suma timpurilor de procesare a clienților care se află în coadă în momentul respectiv.

**-** **AtomicInteger overallWaitingTime:** suma timpurilor de procesare a tuturor clienților care au fost cândva în coadă.

**-** **boolean oprire:** o valoare booleană, care are rolul de controla oprirea threadurilor.

Clasa are următoarele metode:

- **public Queue(int ID):** constructor

- **public void addClient(Client c):** metodă care adaugă un client în coadă. Odată adăugat în coadă, adaugă și timpul acestuia la actualWaitingTime și overallWaitingTime.

- **gettere, settere**

- **public void setOprire():** setează atributul de oprire true, care era false inițial

- **public String returnClientList()**: returnează un String cu lista clienților aflați în coadă.(ex. Client1(1,2,5), Client3(3,6,2))

- **public int returnSize():** returnează numărul de clienți aflați în coadă

- **public void run():** metoda interfeței Runnable. În această metodă se realizează procesarea clienților în cozi. Din coada respectivă, se ia clientul de la capul acesteia (primul), apoi se oprește threadul pentru atâta timp, cât durează servirea clientului. După ce s-a servit clientul și a trecut timpul lui de procesare, din timpul de așteptare a cozii se va scade timpul lui de procesare (deoarece el nu va mai face parte din coadă), și se șterge din coada respectivă.

* **Clasa Scheduler:**

Această clasă are rolul de a crea atâtea threaduri câte cozi sunt, de a le porni și de a le opri atunci când aceștia nu mai conțin clienți. Pe lângă acest lucru, el are și rolul de a distribui clienții în cozile potrivite, adică în cozile unde timpul de așteptare e minim.

Clasa are următoarele câmpuri:

- **List<Queue> queues:** lista de cozi a aplicației.

- **List<Thread> threads:** o listă de threaduri. Fiecare coadă are threadul lui.

- **int maxNoQueues:** numărul maxim de cozi. Acest număr este dat de utilizator în interfață.

- **int maxClientsPerQueue:** numărul maxim de clienți care se pot afla într-o coadă. Acest număr este egal de fapt cu numărul clienților dat de către utilizator, deoarece mărimea unei cozi atunci ar fi cea mai mare când toți clienții sunt acolo.

- **Strategy strategy:** clasa scheduler va distribui clienții în coada potrivită pe baza unei strategii, adică în cazul nostru pe baza timpului minim de așteptare. Prin acest field se face legătura între cele două clase.

Clasa are următoarele metode:

- **public Scheduler(int maxNoQueues, int maxClientsPerQueue):** constructor, care, pe lângă inițializarea câmpurilor, creează atâtea threaduri câte cozi sunt și le și pornește:

*threads.add(new Thread(queues.get(i)));*

*threads.get(i).start();*

**- gettere, settere**

**- public void dispatchClient(Client c):** trimite clientul primit ca și argument la coada unde timpul de așteptare e minim, apelând metoda *addClient* din clasa Strategy.

**-** **public void opresteCozi():** prin această metodă reușește clasa Scheduler să oprească toate threadurile cozilor. Parcurge cu un foreach toate cozile și pentru fiecare apelează metoda *setOprire* a clasei Queue.

**-** **public int returnMaxSize():** această metodă returnează dimensiunea celei mai lungi cozi dintre toate. Acesta ne va fi de folos în afișarea peak hour-ului (adică timpul la care erau cele mai multe persoane într-o coadă)

**-** **public float calculateAverageWaitingTime() :** acesta calculează timpul mediu de așteptare, adică adună overallWaitingTime-ul tuturor cozilor și împarte suma la numărul cozilor.

* **Clasa Strategy:**

Această clasă realizează strategia de distribuire a clienților, și anume strategia timpului minim de așteptare. Clasa nu conține câmpuri, ci conține doar o singură metodă:

- **public void addClient(List<Queue> queues, Client c):** această metodă parcurge lista de cozi a aplicației, căutând coada care are timpul minim de așteptare. Inițiază o variabilă minWait și o variabilă k cu timpul de așteptare, respectiv cu indexul primei cozi. Apoi, parcurgând restul cozilor cu un foreach, compară timpul acestora cu minWait. Când s-a găsit coada care are cel mai mic timp de așteptare, se salvează indexul acesteia în k, apoi clientul c primit ca și argument se inserează în coada cu indexul k.

* **Clasa SimulationManager:**

Aceasta este clasa principală care realizează funcționalitatea aplicației de simulare. Aici se va crea threadul principal, se vor genera clienți aleatorii, respectiv e clasa Main a proiectului.

Clasa are următoarele câmpuri:

- **int timeLimit:** timpul cât durează simularea, introdus de către utilizator.

**- int minProcessingTime:** timpul minim de procesare a clienților, introdus de către utilizator

**- int maxProcessingTime:** timpul maxim de procesare a clienților, introdus de către utilizator

**- int minArrivalTime:** timpul minim de sosire a unui client, introdus de către utilizator

**- int maxArrivalTime:** timpul maxim de sosire a unui client, introdus de către utilizator

**- int numberOfQueues:** numărul cozilor, introdus de către utilizator

**- int numberOfClients:** numărul clienților, introdus de către utilizator

**- Scheduler scheduler:** simularea are un scheduler, care se ocupă de crearea, pornirea, oprirea threadurilor și de distribuirea clienților

**- List<Client> generatedClients:** lista de clienți generați aleator

**-** **View v:** interfața aplicației

Clasa are următoarele metode:

* **public SimulationManager(int timeLimit, int minProcessingTime, int maxProcessingTime, int numberOfQueues, int numberOfClients, int minArrivalTime, int maxArrivalTime, View v):** constructor, care pe lângă inițializarea fieldurilor are rolul de a genera un numberOfClients de clienți apelând metoda generateRandomClients().
* **gettere, settere**
* **public void generateRandomClients():** generează un numberOfClients de clienți, îi inserează în lista de clienți, apoi sortează lista crescător după timpul de sosire.
* **public float averageServiceTime():** calculează timpul mediu de servire, care e egal cu suma timpurilor de servire / nr clienți.
* **public float averageWaitingTime():** calculează timpul mediu de așteptare, care e egal cu suma timpurilor de procesare / nr cozi.
* **public void run() :** metoda interfeței Runnable, care constituie threadul principal, a simulării. Această metodă contorizează timpul simulării, incrementându-l în fiecare secundă în while-ul principal. În acest while, atâta timp, cât timpul curent de simulare e mai mic decât timpul limită de simulare, se efectuează următoarele acțiuni:pentru fiecare coadă (atâta timp cât lista noastră de clienți nu e goală) dacă timpul curent de simulare coincide cumva cu timpul de sosire a primului client din lista de clienți generatedClients, atunci acesta va fi trimis la o coadă apelând scheduler.dispatchClient(generatedClients.get(0));, iar apoi șters din lista de așteptare.Tot în această metodă, se afișează starea simulării în casuța de rezultat de pe interfață și în fișier, împreună cu peak hour, average waiting time și average service time. După ce s-a golit lista de clienți în așteptare, toate threadurile se opresc.
* **Clasa View:**

În această clasă este implementată interfața utilizatorului. Clasa extinde JFrame. Are următoarele câmpuri:

- **JFrame frame:** fereastra aplicației

Căsuțe text pentru introducerea datelor:

**- JTextField nrClients**

**- JTextField nrQueues**

**- JTextField minArrival**

**- JTextField maxArrival**

**- JTextField minServiceTime**

**- JTextField maxServiceTime**

**- JTextField timeLimit**

**- JtextArea result:** aici se afișează evolutia cozilor în timp real

Mesaje afișate pe GUI:

**- JLabel msg1**

**- JLabel msg2**

**- JLabel msg3**

**- JLabel msg4**

**- JLabel msg5**

**- JLabel msg6**

**- JLabel msg7**

Butoane:

**- JButton start:** pornește simularea aplicației

**- JButton exit:** închide fereastra aplicației

**- JButton reset:** șterge conținutul căsuțelor text

- **Controller controller:** un obiect de tip controller, care face legătura între apăsarea butoanelor din View și acțiunile care trebuie să urmeze (implementate în Controller).

Clasa are următoarele metode:

- **public View() :** constructor, aici sunt implementate elementele de design ale calculatorului. Un astfel de element de design ar fi setarea mărimii ferestrei calculatorului ( frame.setSize(1000, 900); ). Panelul principal al ferestrei este adăugat din 7 panele (panel0, panel1, panel2, panel3, panel4, panel5, panel6). Fiecare panel are propriile sale componente

- **gettere:** returnează valorile introduse în căsuțele text

- **public void setResult(String s):** afișează în căsuța rezultatului stringul primit ca argument

- **public void clear():** șterge conținutul căsuțelor text

* **Clasa Controller:**

În această clasă sunt implementate acțiunile care trebuie să urmeze după apăsarea unui buton. Ea implementează interfața ActionListener. Are următoarele câmpuri:

* **View v**

Clasa are următoarele metode:

- **public Controller(View v) :** constructor

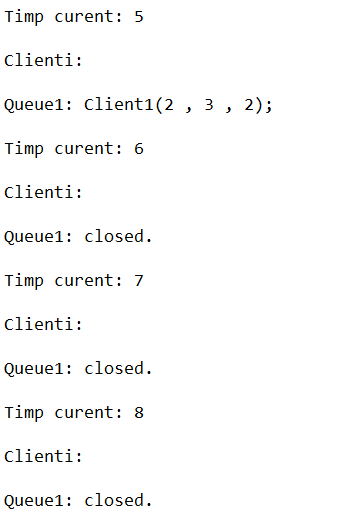
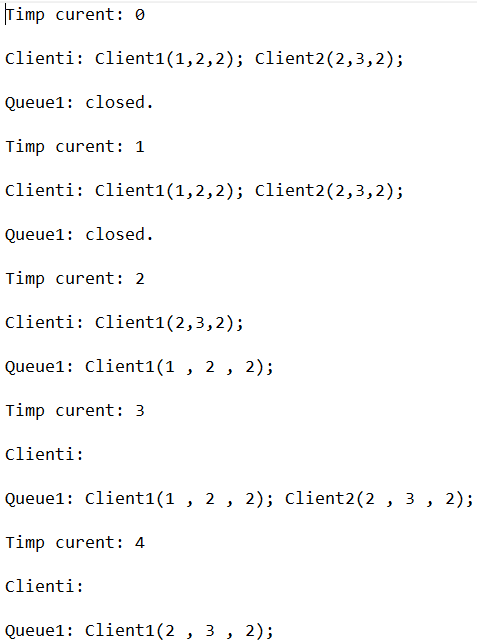
- **public void actionPerformed(ActionEvent e):** metodă specifică interfeței ActionListener, ce trebuie implementată în mod obligatoriu. În această metodă se verifică cu câte un if, care buton s-a apăsat, și în funcție de butonul apăsat, se execută acțiunile necesare. Dacă s-a apăsat butonul start, se pornește simularea. Dacă s-a apăsat butonul exit, se închide fereastra aplicației. Dacă s-a apăsat butonul reset, se șterge conținutul căsuțelor text.

* **Clasa CreateFile:** această clasă nu are un rol esențial în funcționalitatea proiectului, are doar rolul de a scrie în fișiere rezultatele simulărilor.

Clasa are o singură metodă, **public void writeInFile(String msg),** care scrie într-un fișier text stringul msg primit ca și argument.

**5. Rezultate**

Rezultatele simulării pot fi observate în fișierele text, generate de clasa CreateFile. Un exemplu de rezultat a simulării este:



**6. Concluzii**

Ca orice proiect sau aplicație, și acesta a avut dificultățile lui, din care se pot trage concluzii și idei, care ne pot fi utile în implementarea altor proiecte în viitor. Un lucru important ar fi că, pentru realizarea unei obiective, cerințe, trebuie mai întâi să găsim soluții sub-obiectivelor. Un obiectiv principal e alcătuit din mai multe sub-obiective. O altă concluzie ar fi că rezolvarea unei probleme impune programatorului să găsească structurile de date cele mai potrivite pentru caz. În cazul nostru, folosirea BlockingQueue-ului a fost mult mai optim decât folosirea unui vector.

**7. Bibliografie**

<https://www.vogella.com/tutorials/JUnit/article.html>

<https://docs.oracle.com/javase/tutorial/uiswing/>

<https://dsrl.eu/courses/pt/materials/A2_Support_Presentation.pdf>