***TEMA 2***

***QUEUES MANAGEMENT APPLICATION USING***

***THREADS AND SYNCHRONIZATION MECHANISMS***

***Materie: Tehnici de Programare***

***Elev: Pavel Mădălina- Denisa***

***Grupa: 30223***

***Specializare: CTI RO, an 2***

*Cuprins*

[*Obiectivul temei* 2](#_Toc131095220)

[*Analiza problemei, modelare, scenarii, cazuri de utilizare* 3](#_Toc131095221)

[*Proiectare* 4](#_Toc131095222)

[*Implementare* 5](#_Toc131095223)

[*Rezultate* 6](#_Toc131095224)

[Concluzii 9](#_Toc131095225)

[Bibliografie 10](#_Toc131095226)

# *Obiectivul temei*

Principalul obiectiv al temei este proiectarea și implementarea unei aplicații de gestionare a cozilor care atribuie clienții astfel încât timpul de așteptare să fie minimizat.

Cozile sunt folosite frecvent pentru a modela domenii reale. Obiectivul principal al unei cozi este de a furniza un loc pentru un "client" pentru a astepta înainte de a primi un "serviciu". Managementul sistemelor bazate pe cozi are interesul de a minimiza timpul pe care "clienții" îl petrec așteptând la coada înainte de a fi serviți. Una dintre modalitățile de minimizare a timpului de așteptare este de a adăuga mai mulți servere, adică mai multe cozi în sistem (fiecare coadă este considerată a avea un procesor asociat), dar această abordare crește costurile furnizorului de servicii.

Aplicația de gestionare a cozilor ar trebui să simuleze (prin definirea unui timp de simulare 𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛) o serie de N clienți care sosesc pentru serviciu, intră în Q cozi, așteaptă, sunt serviți și părăsesc în cele din urmă cozile. Toți clienții sunt generați când simularea este pornită și sunt caracterizați de trei parametri: ID (un număr între 1 și N), 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 (timpul de simulare când sunt pregătiți să intre în coadă) și 𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 (intervalul de timp sau durata necesară pentru a servi clientul; adică timpul de așteptare când clientul se află în fața cozii). Aplicația urmărește timpul total petrecut de fiecare client în cozile și calculează timpul mediu de așteptare. Fiecare client este adăugat la coadă cu cel mai mic timp de așteptare când timpul său de sosire este mai mare sau egal cu timpul de simulare (𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 ≥ 𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛).

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

Conceptul de fir de execuție reprezintă cea mai mică unitate de procesare programabilă de către sistemul de operare și este utilizat pentru a eficientiza execuția programelor prin rularea paralelă a diferitelor secțiuni de cod în cadrul aceluiași proces. Implementarea aplicațiilor folosind firele de execuție necesită implementarea interfeței Runnable sau extinderea clasei Thread, iar diferența dintre cele două metode constă în faptul că Runnable este o interfață implementată de clasele care o folosesc, în timp ce Thread este o clasă părinte.

În procesul de analiză a problemei și identificare a claselor și funcționalităților acestora, programarea orientată pe obiect ne permite să implementăm o aplicație folosind informații de suprafață. Multithreading-ul este util pentru a rula simultan mai multe secțiuni de cod, astfel încât fiecare sarcină să fie executată independent, permițând obținerea rezultatelor într-un timp optim. În absența thread-urilor, programul nostru ar rula ca un singur fir de execuție, iar fiecare acțiune ar fi executată în mod secvențial, așteptând finalizarea acțiunilor precedente.

Exemplu de use-case:

1. Utilizatorul introduce numarul de clienti

2. Utilizatorul introduce numarul de cozi

3. Utilizatorul introduce timpul maxim al simularii

4. Utilizatorul introduce tMinArrival si TMaxArrival

5. Utilizatorul introduce tMinService si TMaxService

6. Utilizatorul apasa pe butonul “START SIMULATION”.

Dupa apasarea butonului in caseta de text din dreapta se va afisa continutul cozilor(inclusiv cea de asteptare) pentru fiecare moment de timp.

# *Proiectare*

Pentru proiectarea acestei aplicații am folosit design-pattern-ul Model-View-Controller, structurându-mi astfel proiectul în 3 pachete:

* *Model:* acest pachet conține clasele **Client** și **Server**, care constituie practic baza proiectului, unde sunt implementate metodele care tin de fiecare coada in parte si respectiv de clienti;
* *View*: în acest pachet apar doar elementele necasare construirii interfeței grafice, crearea frame-ului principal și poziționarea fiecarui element în interfața utilizatorului;
* *Controller*(aici: org.example): clasele din acest pachet conectează interfața grafică(GUI-ul) de operatiile pe cozi, dar si coada de asteptare de cozile in care vor fi introdusi clientii.

Diagram

Description automatically generated

# *Implementare*

***Clasa Client***

In aceasta clasa am salvat practiv informatiile despre un client: id-ul, tArrival si tService. Aceasta implementeaza interfata Comparable, care ne ajuta sa sortam clientii in coada de asteptare dupa momentul in care au ajuns.

***Clasa Server***

In aceasta clasa practic sunt implementate toate operatiile care tin de o coada. Aceasta contine atributele: queueClients(lista de clienti corespunzatoare fiecarei cozi), waitingPeriod(perioada de asteptare la o coada), timeW(variabila folosita pentru average waiting time) si timeS(variabila folosita pentru averageServiceTime).

Este de mentionat faptul ca waitingPeriod este o variabila de tip AtomicInteger. Este importantă în cadrul programării concurente, în special în situațiile în care mai multe fire de execuție (threads) accesează și modifică aceeași variabilă integer. În astfel de cazuri, utilizarea AtomicIntger asigură faptul că modificările sunt efectuate într-un mod consistent, evitând astfel erorile de sincronizare și concurență.

* ***public void run()***

In metoda run() asociata practic Threadului corespunzator unei anumite cozi am luat fiecare client din aceasta coada si i-am decrementat tService-ul(asta inseamna ca va avea de asteptat dupa fiecare moment de timp cu 1 secunda mai putin la coada). Daca tService ajunge la 1, clientul va fi scos din coada(adica va fi servit). De asemenea, pentru fiecare moment de timp, waitingPeriod -ul cozii va fi decrementat.

***Clasa ConcreteStrategyTime***

Aceasta clasa are o singura metoda:

* ***public void addClient(List<Server> queues, Client c)***

Aceasta metoda adauga un client C la coada care are timpul de asteptare minim.

***Interfata Strategy***

Contine declaratia metodei addClient de mai sus.

***Clasa Controller***

Aceasta clasa contine un actionListener, care leaga interfata de codul propriu-zis. La crearea unui obiect de tipul SimulationManager, se va porni si Threadul asociat.

***Clasa Scheduler***

Aceasta clasa conecteaza clasa Server de clasa SimulationManager. De asemenea, in constructorul acesteia am creat cate un Thread pentru fiecare coada salvata in campul listofQueues, pe care l-am pornit. Am implementat si metoda isEmpty(), care verifica daca mai avem sau nu elemente intr-o coada. Aceasta ne ajuta sa oprim simularea cand nu mai exista client de servit.

***Clasa SimulationManager***

În cadrul acestei clase se petrec majoritatea evenimentelor. Această clasă este cea mai importantă clasă a aplicației, întrucât aceasta implementează majoritatea metodelor. Atributele acestei clase sunt: nbQueues, nbClients, simInterval, minArrival, maxArrival, minService, maxSerivce, listOfClients, listOfQueues, un obiect al clasei Scheduler. Aceasta clasa contine metodele:

* ***public void generateRandomClients()***

Aceasta metoda genereaza nbClients clienti, adica pentru fiecare client un id, un timp de sosire si un timp de servire. Pentru asta am folosit functia Random.

* ***public String printInt(int time)***

Aceasta metoda este folosita pentru afisarea rezultatelor in interfata.

* ***public void Print1(int currentTime)***

Aceasta metoda este folosita pentru afisarea rezultatelor in consola.

* ***public void writeToFile(String fileName, String content)***

Aceasta metoda este folosita pentru afisarea rezultatelor in fisiere text.

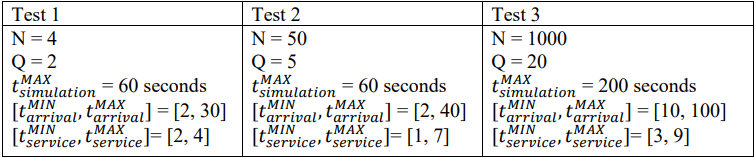
* ***public void run()***

Cea mai importanta metoda din aceasta clasa. Este rulata in momentul in care activam Threadul corespunzator obiectului de tip SimulationManager din clasa Controller.

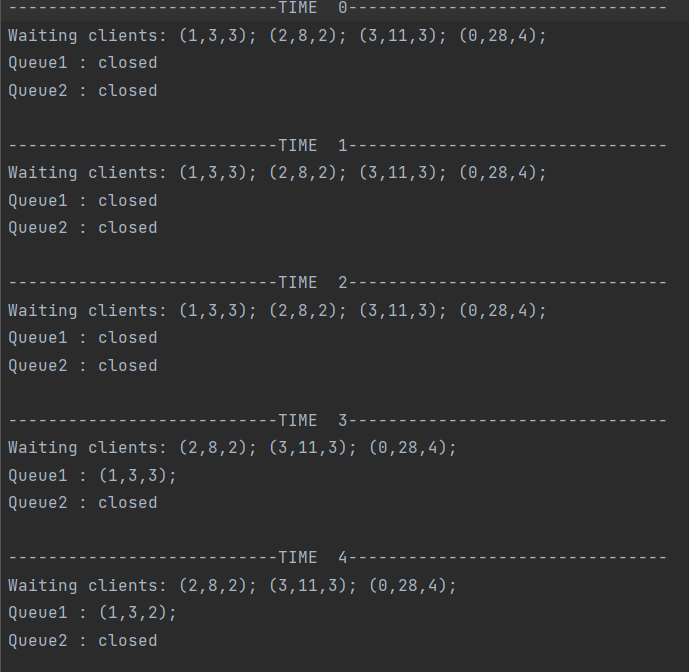
In aceasta metoda preluam fiecare client din lista de asteptare, il trimitem spre o coada-cea cu waitingTime-ul minin(folosind metoda dispatchClients ) si apoi stergem clientul din lista de asteptare.

# *Rezultate*

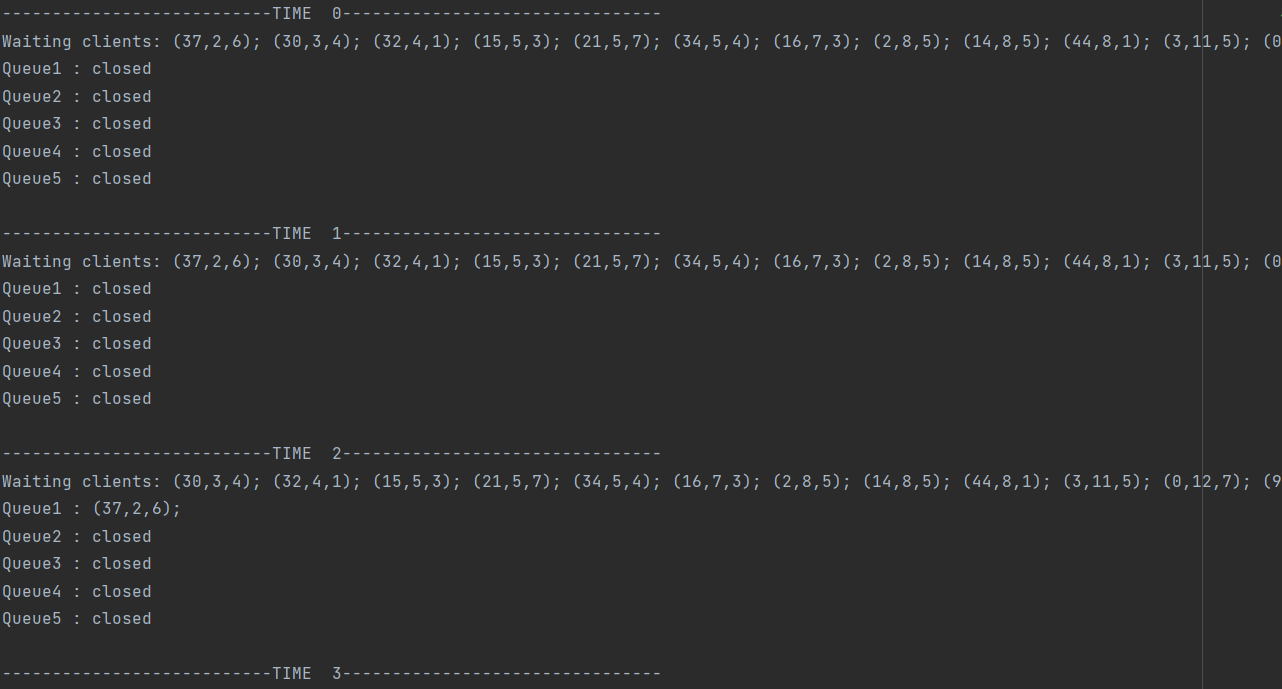
Pentru testarea programului am folosit pe langa testele personale, cele 3 teste date. S-a obtinut o simulare destul de realista a procesului de intrare si iesire din cozi, in functie de timpul minim de asteptare corespunzator fiecarei dintre aceste cozi. Rezultatele pentru acestea au fost salvate in fisiere txt, atasate documentatiei.



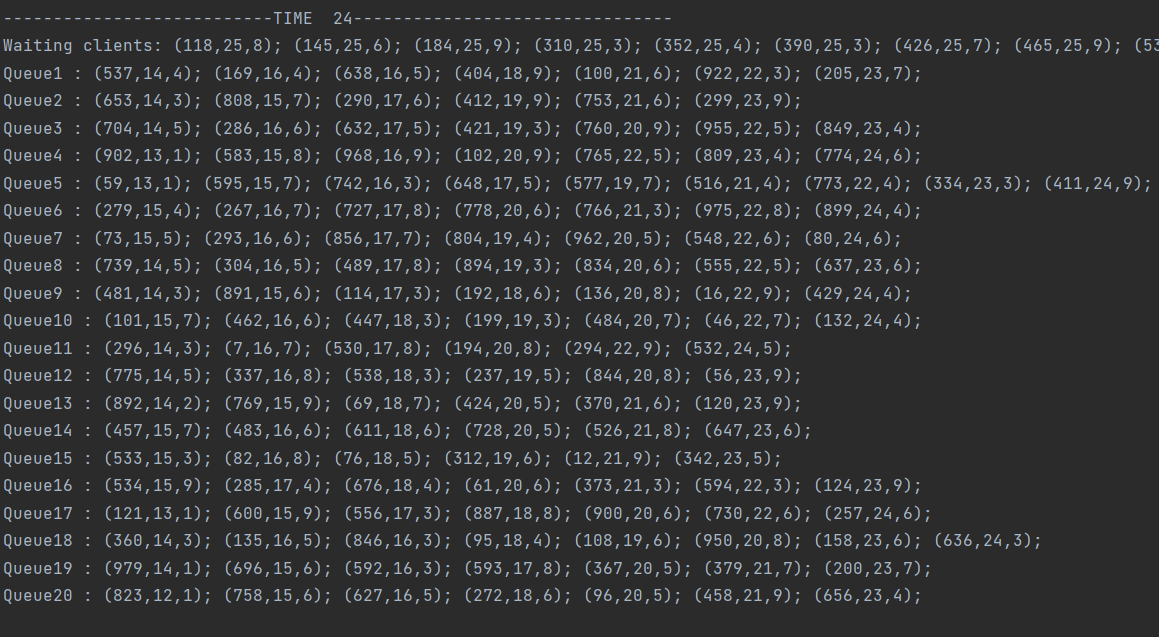
* testul 1



* testul 2



* testul 3



# Concluzii

În concluzie, implementarea acestei simulari în Java mi-a reamintit multe concepte poate uitate de la OOP, dar am utilizat si unele noi, cum ar fi utilizarea Threadurilor, cu care nu am mai lucrat la un astfel de nivel pan acum. De asemenea, am folosit elemente de sincronizare, tot un concept nou pentru mine.

Un astfel de proiect poate fi extins cu funcționalități suplimentare, cum ar fi stergerea rezultatelor anterioare din interfata prin adaugarea unui buton de refresh sau alte elemente ce tin de designul acesteia.(ex. Adaugarea unor simboluri care sa reprezinte oamenii care asteapta la coada).

# Bibliografie

<https://dsrl.eu/courses/pt/>

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

<https://www.geeksforgeeks.org/multithreading-in-java/>

<https://www.javatpoint.com/java-jscrollpane>

<https://stackabuse.com/bytes/fix-index-0-out-of-bounds-for-length-0-in-java/>