



**UNIVERSITATEA TEHNICĂ**  
DIN CLUJ-NAPOCA

-Tehnici de programare-  
Tema 2: Simulator de cozi

Student: Iamnițchi Bogdan

Grupa: 30225

# **Cuprins:**

## 1. Cerințe funcționale

### 1.1. Enunț

### 1.2. Descriere

### 1.3. Date de intrare

### 1.4. Date de ieșire

## 2. Obiectivul temei

### 2.1. Obiectiv principal

### 2.2. Obiective secundare

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

## 4. Proiectare

### 4.1. Structuri de date

### 4.2. Diagrama de clase UML

## 5. Implementare

## 6. Concluzii

## 7. Bibliografie

# 1. Cerințe funcționale

## 1.1. Enunț:

Proiectați și implementați o aplicație de simulare care vizează analiza sistemelor bazate pe cozi pentru determinarea și minimizarea timpului de așteptare al clienților.

## 1.2. Descriere:

Cozile sunt utilizate ca model în lumea reală. Obiectivul principal al unei cozi este să pună elementul într-un loc cât mai favorabil. Ideea principală este de a minimiza timpul de așteptare creat de cozile respective. O cale de a minimiza timpul de așteptare este de a face mai multe cozi. În principal aplicația ar trebui să simuleze un exemplu uzual din viața reală precum ceea ce se întâmplă într-un supermarket. Când se deschide o nouă casă de marcat (se adaugă un nou server), clienții în așteptare vor fi distribuiți uniform la toate cozile disponibile curente, alegând coada care are timpul de așteptare minim.

## 1.3. Date de intrare:

- Numărul de clienti (N)
- Numărul de cozi (Q)
- Timpul maxim de simulare a aplicației
- Minimul și maximul timpului de sosire un client
- Minimul și maximul timpului de servire a unui client

## 1.4. Date de ieșire:

Ieșirea aplicației este dată de o interfata cu utilizatorul dar și un fișier text care conține un jurnal al execuției aplicației și timpul mediu de așteptare al clienților.

# 2. Obiectivul temei

## 2.1. Obiectivul principal:

Tema are ca obiectiv principal proiectarea și implementarea unei aplicații “simulator de cozi” pentru determinarea și minimizarea timpului de așteptare al clienților. Datele de intrare vor fi introduse de către utilizator într-un fișier, urmând să fie procesate și afișate într-un alt fișier text.

## 2.2. Obiective secundare:

- a. Utilizarea programării orientate pe obiect
- b. Alegerea structurilor de date (abstractizarea datelor)
- c. Împărțirea pe clase (modelarea problemei)
- d. Implementarea unui Random Client Generator
- e. Multithreading: un thread per coadă
- f. Structuri de date sincronizate adecvate pentru a asigura siguranța thread-ului
- g. Închiderea și deschiderea cozilor în mod dinamic
- h. Generarea unui fișier .jar
- i. Implementarea soluției (proiectarea ideii)
- j. Testare

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

Aplicația simulează (prin definirea unui timp de simulare *tsimulation*) o serie de N clienți care sosesc pentru un serviciu, intrarea în cele Q cozi, așteptarea, procesarea serviciului și, în final, părăsirea cozilor.

Toți clienții sunt generați la începutul simulării și sunt caracterizați de trei parametri: ID (un număr între 1 și N), *tarrival* (timpul de simulare când sunt gata să meargă la coadă) și *tservice* (intervalul de timp sau durata necesară pentru a fi servit clientul de către casier). Aplicația urmărește timpul total petrecut de fiecare client la coadă și calculează timpul mediu de așteptare. Fiecare client este adăugat la coadă cu un timp minim de așteptare când timpul său este egal cu timpul de simulare, urmând să fie procesat atunci când ajunge în capul cozii.

În modelarea problemei am luat în considerare diferite opțiuni având în vedere implementarea rezolvării, urmând exemplul din ppt-ul pus la dispoziție de către profesor. Această aplicație este modelată în așa fel încât să îndeplinească toate cerințele funcționale ale unui simulator de cozi. Ca mod de organizare a informației este folosită încapsularea, accentul fiind pus pe abstractizarea datelor care împreună cu proprietățile și caracteristicile lor sunt grupate în obiecte, reunite mai apoi în clase.

### SCENARIU DE UTILIZARE:


ACTOR: Un utilizator care dorește să observe modul de lucru a cozilor într-un supermarket (sau orice alt magazin) și să vadă care este timpul mediu de așteptare a clientilor la cozi.

PRE-CONDITIONS: Datele de intrare au fost introduse corect în fisierul text din care se vor citi datele problemei având în vedere rularea programului.

POST-CONDITIONS: Se va genera un fișier text care va conține pașii simulării începând cu timpul 0 pana la *tsimulation*. La fiecare pas se va afișa lista de clienți în așteptare și fiecare coadă cu clienții din acea coadă.

NORMAL FLOW: Utilizatorul introduce corect datele de intrare în fisier, salvează modificările, rulează programul, deschide fisierul de output pentru a observa cum funcționează aplicația.

### Exemplu fisier de ieșire

 out-test-1 - Notepad

Fișier Editare Format Vizualizare Ajutor

---

Time 0

Waiting clients: (4,3,3); (3,11,3); (1,17,3); (2,25,3);

Queue 1:

Queue 2:

Time 1

Waiting clients: (4,3,3); (3,11,3); (1,17,3); (2,25,3);

Queue 1:

Queue 2:

...

Time 27

Waiting clients:

Queue 1: (2,25,2);

Queue 2:

Time 28

Waiting clients:

Queue 1: (2,25,1);

Queue 2:

Average waiting time: 3.0

---

## 4. Proiectare

În ceea ce privește proiectarea aplicației am optat pentru folosirea a mai multe clase, bine-delimitate, astfel încât să fie mult mai clară și mult mai simplă ideea de implementare.

Prima clasă cu care am început a fost clasa **Client**, gândindu-mă că această aplicație are la bază lucrul cu clienți. Fiecare client are ID, `t_arrival`, `t_service` și un `t_wait` (timpul de așteptare din momentul în care s-a pus la coadă până când acesta este procesat și părăsește coada). Astfel, am definit variabilele instanței ale clasei, urmând să fac un constructor pentru acestea, gettere și settere, și o metodă `toString()` pentru a putea furniza informațiile despre un client sub forma de String, restul urmând a fi completat pe parcursul dezvoltării aplicației.

După clasa Client am construit **Server** care este corespondentul unei cozi. Un obiect de tip Server are o coadă care conține mai mulți clienți, are un *waitingTime* specific cozii și un flag care ne va ajuta să deschidem și să închidem coada în mod dinamic. Fiecărui obiect de tipul Server îi este asociat un thread care se va ocupa de executia operațiilor specifice unei cozi. Această clasă are un constructor pentru initializarea variabilelor instanței, urmând să aibă metode de adăugare și scoatere a unui client din coadă, cât și o metodă `run` specifică thread-ului. Alte completări se vor face în timpul implementării soluției.

Următoarea clasă este clasa **Scheduler** care se ocupă de managementul cozilor. Această clasă conține mai multe cozi, numărul cozilor fiind dat în constructorul specific obiectului de tipul Scheduler, urmând să fie create un număr de *Q* cozi. Spunând că această clasă se ocupă cu managementul cozilor, putem trage concluzia că aceasta se ocupă cu punerea clienților în cozi, după un anumit criteriu (în cazul nostru, cel mai mic timp de așteptare).

Ultima clasă este **SimulationManager** care are cel mai important rol în functionarea aplicației. Ea se va ocupa cu citirea și scrierea datelor în fișier, cu generarea clienților cu timpul `t_arrival` și `t_service` random. Acestea nu sunt singurele funcționalități ale clasei, aceasta clasă coordonează toată activitatea folosindu-se de un thread care are metoda `run`. Pe lângă cele menționate, clasa SimulationManager conține metoda `main`, unde are loc crearea obiectului de tip SimulationManager și asocierea lui cu un thread.

Am implementat și o interfață **Strategy** pe care o implementează alte două clase **ShorestTimeStrategy** și **ShortestQueueStrategy** care se ocupă cu repartizarea clienților după un anumit criteriu selectat de utilizator.

Deși pentru a putea face posibilă interacțiunea grafică cu utilizatorul am mai creat o clasă de **Controller** și una de **GUI**. Prin intermediu acestora extrag informațiile introduse de utilizator și fac posibilă crearea unei simulări.

Clasele SimulationManager și Server implementează interfața Runnable, suprascriind metoda `run()`, iar clasa Client implementează interfața Comparable.

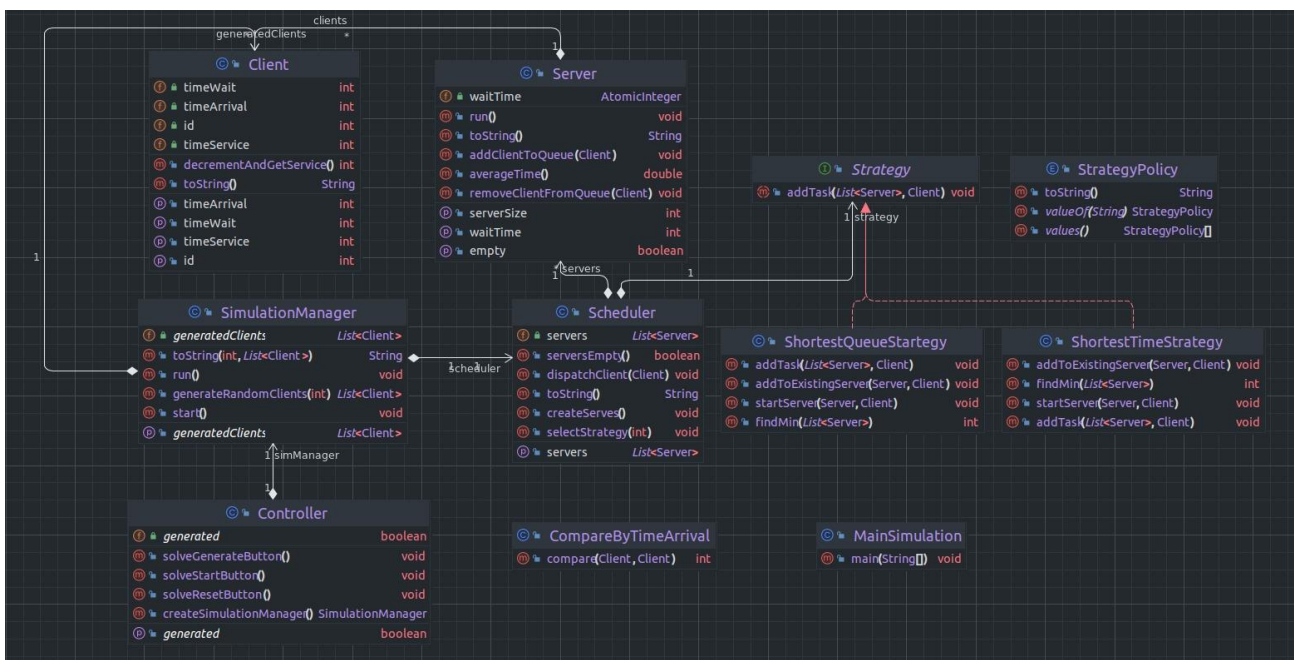
## 4.1. Structuri de date

Pentru a facilita modul de lucru și a obține o eficiență cât mai bună am folosit ArrayList, o structură de date care implementează List. Am ales această structură de date deoarece în comparație cu un vector, ArrayList are deja implementate metodele de add(), remove(), getSize(), get(), (de care mă pot folosi atunci când doresc să construiesc un polinom, care este de fapt o listă de monoame), iar pentru un vector aceste metode ar trebui definite, implementate și mai apoi folosite. Am folosit o structură ArrayList pentru a ține clienții generați random (acei waiting clients) și o altă structură ArrayList pentru a ține cozile.

Lucrând cu thread-uri trebuie să asigurăm siguranța fiecărui thread. Acest lucru se poate face folosind structuri de date sincronizate. Structurile de date sincronizate alese de mine sunt:

- ArrayBlockingQueue care implementează interfața BlockingQueue (această structură de date am folosit-o pentru a ține clienții într-o coadă)
- AtomicInteger - folosită pentru a ține timpul de așteptare într-o coadă.

## 4.2. Diagrama de clase UML



## 5. Implementare

Prima clasa, numita **Client**, implementeaza interfata Comparable si are 4 variabile instantate: *id*, *t\_arrive*, *t\_service* si *t\_wait*; un constructor cu parametrii de tip int pentru initializarea variabilelor instantate; 3 gettere ( pentru *t\_arrive*, *t\_service* si *t\_wait*); 3 settere ( pentru *t\_arrive*, *t\_service* si *t\_wait*); o metoda toString() care converteste obiectul de tip Client in obiect de tip String si o metoda compareTo(), care compara doi clienti dupa de timpul de sosire, iar în funcție de rezultatul comparării returnează -1,0,1.

Cea de-a doua clasa, **Server**, implementeaza interfata Runnable si are: o variabila *clients* de tip BlockingQueue <Client> care reprezinta o coada de clienti; o alta variabila *waitingTime* de tip AtomicInteger care reprezinta timpul de asteptare al cozii (suma timpilor de servire a clientilor din coada respectiva; o variabila volatila de tip boolean *flag* care are rolul de porni sau opri executia threadului corespunzator cozii; doua variabile statice *nrOfClients* - numarul de clienti care au fost procesati complet si *totalTime* – suma timpilor de asteptare a fiecarui client care a fost procesat complet. Mai departe, aceasta clasa are un constructor fara parametrii, unde initializeaza *clients* ca fiind de tipul ArrayBlockingQueue, *waitingTime* si *flag* ul il seteaza pe false si un getter pentru *waitingTime*.

Clasa Server contine mai multe metode:

- *addClient()* - in aceasta metoda se adauga un client in coada, se seteaza flagul pe true ( pentru ca thread ul sa-si inceapa executia) si se actualizeaza variabila *waitingTime* adunand *t\_service* a clientului care a fost adaugat in coada
- *removeClient()* – in metoda aceasta se scoate un client din coada, numarul de clienti procesati complet *nrOfClients* creste, se actualizeaza *totalTime* adunand *t\_wait* a clientului scos din coada, iar daca coada este goala flag ul este setat pe false
- *empty()* – aceasta metoda returneaza true daca coada este goala, iar in caz contrar returneaza false
- *toString()* - care converteste obiectul de tip Server in obiect de tip String
- *averageTime()* – metoda care returneaza timpul mediu de asteptare a clientilor in cozi, fiind calculat ca *totalTime* impartit la *nrOfClients*
- *run()* – aceasta metoda are o mare functionalitate, fiind o metoda impusa de catre interfata Runnable, necesara pentru a putea rula thread urile secundare specifice fiecărei cozi. Orice thread are o metoda run() care contine un while si se executa cat timp conditia din while e adevarata (in cazul nostru while( flag == true)). Pentru inchiderea si deschiderea cozilor in mod dinamic se foloseste un flag care devine true atunci cand un client intra in coada si devine false in momentul in care coada este goala, in acest fel oprindu-se executia thread-ului. In aceasta metoda are loc procesarea clientului aflat in capul cozii, procesarea facandu-se prin scaderea timpului de servire *t\_service* si intreruperea thread ului pentru o secunda ( Thread.sleep(1000) ). Mai apoi se



verifica daca timpul de servire a clientului este egal cu 0 atunci se apeleaza metoda `removeClient` pentru ca acesta sa fie scos din coada.

O alta clasa este ***Scheduler***. Ea se ocupa de managementul cozilor, creand cozi, pornind thread-uri specifice cozii si distribuind clientii in cozi. Aceasta clasa are o lista de cozi, iar prin intermediul constructorului cu parametru de tip `int` reprezentand numarul de cozi, ea adauga un numar `Q` de cozi in lista specifica. Clasa contine o metoda `dispatch()` care este responsabila cu adaugarea clientilor in cozi dupa un anumit criteriu ( criteriul folosit in acest program este timpul de asteptare minim ) si pornirea thread-ului cand e cazul. Pentru a adauga un client in coada corespunzatoare se testeaza urmatoarele cazuri:

- daca macar una din cozi este goala, clientul este adaugat in prima coada goala
- daca cozile nu sunt goale, se calculeaza care dintre cozi are timpul mai mic de asteptare, urman ca clientul sa fie adaugat in coada care are timpul de asteptare minim
- daca cozile au timpul de asteptare egal, clientul va fi adaugat in prima coada

Clasa mai contine metoda `emptyQueues()` care returneaza `true` daca toate cozile sunt goale, respectiv `false` daca mai exista clienti la cozi si metoda `toString()` - care converteste obiectul de tip `Scheduler` in obiect de tip `String`.

Ultima clasa implementata este clasa `SimulationManager()` care implementeaza interfata `Runnable` si are rolul de a coordona celelalte clase si thread uri. Clasa are un constructor cu 2 parametrii de tipul `String`, reprezentand numele fisierelor de intrarea si iesire, iar in cadrul acestuia se initializeaza numele fisierelor, se apeleaza metoda `readFile()` si metoda `generateRandomClients()` . Aceasta clasa contine mai multe metode, dintre care:

- `readFile()` – in aceasta metoda are loc citirea datelor din fisier si initializarea variabilelor cu datele corespunzatoare
- `generateRandomClients()` - in aceasta metoda are loc crearea a `N` clienti cu id din intervalul `[1, N]` si cu `t_arrival` si `t_service` generate random in intervalele specificate. Dupa crearea si adaugarea celor `N` clienti, se apeleaza sortarea acestora in functie de `t_arrival`.
- `toString()` - care creeaza un `String` care va fi mai apoi afisat in fisier sau consola
- `run()` – este o metoda impusa de catre interfata `Runnable`, necesara pentru a putea rula thread ul principal al aplicatiei. Thread-ul se executa cat timp timpul de simulare curent este mai mic decat timpul maxim de simulare si se opreste atunci cand timpul curent de simulare este egal cu timpul maxim de simulare sau daca cozile sunt goale si nu mai sunt clienti in asteptare. In acesta metoda se ia primul element din lista de clienti generati random (daca lista nu e goala) si se apeleaza `dispatch` pentru a pune clientul in coada, fiind apoi scos din lista

de clienti generati, aceste doua instructiuni fiind executate cat timp lista nu e goala si timpul de sosire este egal cu timpul de simulare. Astfel se vor lua in acelasi interval de timp toti clientii care au timpul de sosire este egal cu timpul de simulare, mai apoi avand loc scrierea in fisierul de output si intreruperea thread-ului pentru o secunda.

- *main()*- in aceasta metoda se creeaza un obiect de tipul *SimulationManager* a carui constructor primeste ca parametrii argumentele (numele fisierelor) date in linia de comanda la rularea fisierului jar din consola, urmand ca sa construiasca un thread asociat acelui obiect si sa-l porneasca.

## 6. Concluzii

Aceasta tema m-a ajutat sa-mi dezvolt abilitatile de lucru in ceea ce priveste programarea orientate pe obiecte, lucrul cu thread-uri (fiind primul meu proiect care a implicat utilizarea acestora) si nu numai.

Am invatat sa lucrez cu structuri de date sincronizate, pentru a oferi siguranta thread-urilor si pentru a le sincroniza. Totodata, am exersat modul de lucru cu fisiere text si mi-am dat seama mai bine ce inseamna procesul de abstractizare a datelor.

Aceasta tema mi s-a parut una destul de practica, simuland aspecte din viata reala.

## 7. Bibliografie

- <https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>
- [http://coned.utcluj.ro/~salomie/PT\\_Lic/4\\_Lab/Assignment\\_2/Java\\_Concurrency.pdf](http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_2/Java_Concurrency.pdf)
- <https://howtodoinjava.com/java/multi-threading/>
- <https://www.geeksforgeeks.org/arrayblockingqueue-class-in-java/>
- <https://stackoverflow.com/questions/1082580/how-to-build-jars-from-intellij-properly>