# DOCUMENTATIE TEMA 4: SISTEM DE MANAGEMENT

# AL UNUI RESTAURANT

# AGHENITEI BIANCA

CALCULATOARE ROMANA GRUPA 30225



# Obiectivul temei

**Proiectarea și implementarea unei aplicații de simulare care vizează analiza sistemelor bazate pe cozi pentru determinarea și minimizarea timpului de așteptare al clienților**

Aplicația ar trebui să simuleze (definind un timp de simulare 𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛) o serie de N clienți care sosesc pentru serviciu, care intră în cozi Q, așteaptă, sunt deserviți și, în final, părăsesc cozile. Toți clienții sunt generați la începerea 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, adică ora la care clientul a terminat cumpărăturile) și 𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 (intervalul de timp sau durata necesară pentru a-l servi pe client de către casier, adică timpul de așteptare când clientul este în fața cozii). Aplicația urmărește timpul total petrecut de fiecare client la cozi și calculează timpul mediu de așteptare. Fiecare client este adăugat la coadă cu un timp minim de așteptare când timpul său 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 este mai mare

decât sau egal cu timpul de simulare (𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 ≥ 𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛).

Următoarele date ar trebui considerate date de intrare citite dintr-un fișier text pentru aplicație:

- Numărul de clienți (N);

- Numărul de cozi (Q);

- Interval de simulare (𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛

𝑀𝐴𝑋);

- Ora minimă și maximă de sosire (𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙

𝑀𝐼𝑁 ≤ 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 ≤ 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙

𝑀𝐴𝑋);

- Timp de serviciu minim și maxim (𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒

𝑀𝐼𝑁 ≤ 𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 ≤ 𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒

𝑀𝐴𝑋);

Ieșirea aplicației este un fișier text care conține un jurnal al execuției aplicației și timpul mediu de așteptare al clienților

**Obiective secundare:**

* 1. Analzia problemei si dezvoltarea de scenarii de utilizare
  2. Alegerea structurilor de date
  3. Impartirea pe pachete si clase, alegerea strcuturii proiectului
  4. Generarea unui fisier .jar
  5. Generarea fisierelor de test si verificarea corectitudinii lor

# Analiza problemei, modelare, scenarii, cazuri de utilizare

Cozile sunt utilizate în mod obișnuit pentru modelarea domeniilor din lumea reală. Obiectivul principal al unei cozi este de a oferi un loc unde un „client” să aștepte înainte de a primi un „serviciu”. Gestionarea sistemului bazat pe cozi este interesată să reducă la minimum perioada în care „clienții” lor așteaptă la cozi înainte de a fi deserviți. O modalitate de a minimiza timpul de așteptare este adăugarea mai multor servere, adică mai multe cozi în sistem (fiecare coadă este considerată ca având un procesor asociat), dar această abordare crește costurile furnizorului de servicii.

Acest proiect va modela comportamentul unui numar constant de cozi care va fi citit din fisierul de intrare. Utilizatorul va rula in linia de comanda fiserul .jar , trimitand ca argumente calea fisierului de intrare, respectiv a celui de iesire. Programul va procesa datele de intrare din fisier si va afisa in fisierul de iesire starea cozilor la momentele discrete cand acestea de pot schimba, dupa urmatorul model:

*Time 0*

*Waiting clients: [(4 6 27), (3 9 27), (2 17 5), (1 25 2)]*

*1:closed cu waiting time:0*

*2:closed cu waiting time:0*

*Time 1*

*Waiting clients: [(4 6 27), (3 9 27), (2 17 5), (1 25 2)]*

*1:closed cu waiting time:0*

*2:closed cu waiting time:0*

*Time 2*

*Waiting clients: [(4 6 27), (3 9 27), (2 17 5), (1 25 2)]*

*1:closed cu waiting time:0*

*2:closed cu waiting time:0*

*Time 3*

*Waiting clients: [(4 6 27), (3 9 27), (2 17 5), (1 25 2)]*

*1:closed cu waiting time:0*

*2:closed cu waiting time:0*

La final se va afisa in fisier timpul mediu petrecut de clienti in coada dupa modelul:

*Averange time: 21.5*

**Cazuri de utilizare:**

Daca datele de intrare sunt corecte se va genera un fisier log cu numele precizat in apel care contine starea fiecarei cozi la momente discrete.

In cazul in care calea fisierului de intrare este invalida, programul va arunca o exceptie care va fi semnalata in terminal. De asemenea, el nu va rula daca argumentele nu sunt trimise corect. Formatul apelului corect este:

*java -jar PT2020\_Group\_FirstName\_LastName\_Assignment\_2.jar in.txt out.txt*

# 3.Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator)

Pentru modelarea comportamentului concurent al mai multor cozi, am optat pentru folosirea Threadurilor Java (fire de executie).

*“*[*Limbajul de programare Java*](https://ro.wikipedia.org/wiki/Java_(programming_language))*și*[*Java virtual machine*](https://ro.wikipedia.org/w/index.php?title=Java_virtual_machine&action=edit&redlink=1)*(JVM) au fost proiectate pentru a suporta*[*programarea concurentă*](https://ro.wikipedia.org/w/index.php?title=Programare_concurent%C4%83&action=edit&redlink=1)*și toate execuțiile au loc în contextul*[*firelor de execuție*](https://ro.wikipedia.org/wiki/Fir_de_execu%C8%9Bie)*. Obiectele și resursele pot fi accesate de multiple fire de execuție separate.*

[*Concurența*](https://ro.wikipedia.org/wiki/Concuren%C8%9B%C4%83_(informatic%C4%83))*este abilitatea de a rula mai multe părți ale unui program sau mai multe programe în paralel. Dacă sarcini consumatoare de timp pot fi realizate asincron sau în paralel, acest îmbunătăți debitul și interactivitatea programului.”*

Asadar, fiecare coada va fi reprezentata de un obiect „Server” care implementeaza interfata runnable, deci ii putem asocia threaduri.

*„java.lang.Runnable is an interface that is to be implemented by a class whose instances are intended to be executed by a thread. There are two ways to start a new Thread – Subclass Thread and implement Runnable. There is no need of subclassing Thread when a task can be done by overriding only run() method of Runnable.*

Pe langa threadurile pe care le asociem cozilor, va exista si un thread principal care retine timpul curent al simularii si are scopul de a distribui fiecare client la coada potrivita cand t arrival al sau este egal cu timpul simularii.

Threadul unei cozi va fi pornit atunci cand se adauga un element in coada. Din secunda in secunda dorim sa actualizam starea cozilor, sa decrementam timpii de procesare pentru elementele din coada si eventual sa le stergem din coada pe cele ale caror timp a expirat. Pentru aceasta, threadul va fi oprit (sleep) din metoda run() timp de o secunda si se va trezi doar sa faca actualizarile necesare.

De asemenea, faptul ca listele sunt alocate dinamic presupune ca threadurile sa nu ramama in viata cat timp aceastea sunt goale deoarece ar consuma resurse ale procesorului. Pentru aceasta, la fiecare moment cand threadul va fi treaz, el va verifica daca lista sa este goala si daca da, atunci el va muri. De fiecare daca cand adaugam un task la coada verific daca coada este goala si daca da, atunci trebuie sa creez un thread nou cu metoda start().

Pentru a realiza acestea am optat pentru urmatoarele clase:

* 1. **Clasa Task**

-retine ID-ul, timpul de procesare si timpul la care poate intra in coada si timpul petrecut in coada al fiecarul task;

-are getteri si setteri pentru arrivalTime, fiindca acesta se schimba la fiecare secunda petrecuta in coada;

-suprascrie clasa toString pentru afisarea dorita in fisier;

* 1. **Clasa Server**

-reprezinta o coada si dupa cum am explicat,implementeaza interfata Runnable. Asadar, metoda run() trebuie sa actualizeze starea cozii din secunda in secunda si sa o inchida daca aceasta este goala.

-retine o coada de Taskuri de tipul BlockingQueue (o coada thread-safe);

*„BlockingQueue implementations are thread-safe. All queuing methods achieve their effects atomically using internal locks or other forms of concurrency control. However, the bulk Collection operations addAll, containsAll, retainAll and removeAll are not necessarily performed atomically unless specified otherwise in an implementation.”*

-retine o variabila waitingPeriod, care reprezinta durata pentru care coada va ramane activa daca nu se mai adauga elemente intre timp. Cu alte cuvinte, ea semnalizeaza cate secunde ar trebui sa astepte un task nou pana sa fie procesat daca este adaugat in coada. Pe baza acestui criteriu vom stabili in ce coada este cel mai potrivit sa adaugam un task.

**4.Interfata Strategy**

-clasele care implementeaza interfata stretegy se ocupa de distribuirea efectiva a clientilor in cozi in functie de politica stabilita. Astfel, **clasa ConcreteStrategyTime** va pune clientul al caruia este randul in coada care are cel mai mic timp de asteptare sau **clasa ConcreteStrategyQueue** poate pune acelasi client in coada in care are cel mai mic numar de clienti. Se pot defini o multutudine de alte strategii dupa care pot opera cozile. Ele trebuie doar sa implementeze interfata Strategy si sa contina metoda addTask.

**3.Clasa Scheduler**

**-**toate operatiile la coada trec prin aceasta clasa organizatoare. Ea retine o lista cu toate cozile, numarul lor si dimensiunea lor maxima, cat si politica dupa care vom insera un task nou in cozi.

-este specifica unei simulari;

-initial creeaza toate serverele si stabileste o strategie, care poate fi schimbata ulterior cu metoda changeStrategy()

-cand este randul sa se insereze un Task nou in cozi, clasa principala de simulare va apela metoda dispatchTask() din aceasta clasa, care va apela metoda de adaugare efectiva specifica, addTask, in functie de strategia aleasa.

**4.Clasa SimulationManager**

**-**pregateste simularea, citind si retinand datele din fisierul de intrare

-genereaza clienti random conform indicatiilor citite

-culege rezultatele si le scrie in fisierul de iesire

-are specific Threadul principal al programului, deci implementeaza interfata Runnable. Metoda run() este apelata din main si va rula pana la finalul programului. Metoda run() se ocupa de retinerea timpului curent al simularii si de delegarea taskurilor al aror arrival time este egal cu timpul curent catre Scheduler, care va avea grija de ele.

-threadul se termina singur cand timpul de simulare atinge timpul maxim de simulare citit din fisier si implicit termina executia programului.

# Rezultate

Rezultatele sunt cele asteptate, atat pentru date de intrare mici, cat si pentru unele mai mari, cum ar fi un numar mai mare de cozi, clienti sau un timp de simulare mai lung. Pentru un timp de simulare mai lung am modificat timpul relativ al simularii de la o secunda la 100 de milisecunde pentru a putea observa rapid rezultatele in fisierul lor.

Asadar, clientii sunt introdusi in cozi cand timpul lor de arrival este egal cu timpul curent, iar la o unitate de timp este decrementat timpul de procesare. Cand timpul de procesare ajunge la 0, acestia sunt scosi imediat din coada, astfel ca taskurile cu timpul 0 nici nu apar prezente in cozi.

Am testat politica „Shortest\_time”, confrom careia clientii sunt adaugati mereu in coada cu timpul de asteptare minim si functioneaza.

De asemenea am analizat rezultatele pentru timpul de asteptare mediu si ele sunt cele asteptate: daca clientii sunt alesi astfel incat toti sa aibe timp sa fie procesati si cozile sunt destul de incapatoare si nu foarte multe, este normal ca timpul mediu de asteptare sa fie mai mare decat timpul mediu de procesare al taskurilor, deoarece unele dintre ele vor trebui sa astepte si in cozi care nu sunt goale.

# Concluzii

In urma finalizarii acestui proiect consider ca am fixat unele dintre notiunile programarii orientate pe obiect, m-am familiarizat cu lucrul cu obiecte si interfete si am fost introdusa unor notiuni cu totul noi pentru mine, precum lucrul cu threaduri, generarea unui fisier .jar si rularea lui.

Desi la crearea claselor si a metodelor nu am intalnit probleme, lucrul cu threaduri Java mi-a dat de furca. Chiar daca notiunile sunt foarte asemenatoare, implementarea threadurilor s-a dovedit a fi oarecum diferita fata de ce stiam de dinainte de la sisteme de operare. Din fericire am avut prezentarea powerpoint de pe site-ul domnului profesor pentru informatii necesare si indicii referitoare la crearea claselor, astefel ca am reusit sa rezolv treptat toate problemele si nelamuririle pe care le aveam initial.

Obiectul sisteme de operare s-a dovedit totusi de folos spre final, cand am generat fiiserul .jar si am modificat programul sa citeasca argumentele la apelul din linia de comanda. Nu am avut probleme cu aceasta parte, fiindca a fost foarte asemenator cu ce eram invatata sa fac in Linux.

Din pacate timpul nu mi-a permis sa adaug mai mult proiectului, insa m-am gandit la mai multe posibilitati de imbunatatire ulterioara a lui:

* folosirea unui regex pentru a citi datele de intrare;
* realizarea unei interfete grafice care simuleaza activitatea cozilor;
* calculul mai eficient al timpului mediu de asteptare in coada;
* posibilitatea alegerii unei noi politici de adaugare in coada;

- tratarea explicita a cazurilor in care datele de intrare sunt incorecte si notificarea utilizatorului; -tratarea mai multor exceptii, atat la calcul, cat si la introducerea datelor;

-permiterea introducerii datelor cu coeficienti reali;

-imbunatatiri la interfata grafica: imagini, culori;

-adaugarea unei melodii pe fundal;

In concluzie, pot spune ca m-am distrat realizand acest proiect si sunt multumita cu rezultatul final. Sunt fericita ca am fixat niste notiuni importante de OOP, care ma vor ajuta la dezvoltarea in viitor a unor proiecte mai complexe. . Lucrul cu threaduri mi se pare foarte facil fiindca te poti folosi la maxim de resursele calculatorului si poti obtine timpi de executie mult mai mici daca stii sa imparti eficient taskurile.