DOCUMENTAȚIE

Sistem de procesare a clienților într-un magazin

Chișluca Miruna

Grupa 30222

Contents

[1. Obiective 3](#_Toc6531299)

[2. .Analiza problemei 4](#_Toc6531300)

[3. Proiectare 5](#_Toc6531301)

[4. Implementare 6](#_Toc6531302)

[5. Rezultate 9](#_Toc6531303)

[6. Concluzii 9](#_Toc6531304)

[7. Bibliografie 9](#_Toc6531305)

# *Obiective*

Obiectivul principal al temei reprezinta proiectarea si implementarea unui sistem care genereaza simuleaza un caz din realitate in care oamenii se aseaza la niste cozi si asteapta sa le fie procesate cerintele.

Pentru a indeplini acest obiectiv, se are in vedere atingerea unor alte obiective secundare precum proiectarea structurata a aplicatiei pentru eficienta operatiilor necesare indeplinirii cerintei, crearea unei clase care sa reprezinte o coada de asteptare din realitate, folosirea thread-urilor pentru a simula corect, in timp real aceasta procesare a clientilor si implementarea unei interfete care sa simuleze corect acest lucru.

Thread-urile sunt folosite in această aplicatie atat pentru a genera un anumit numar de clienti, cat si pentru procesarea constanta la fiecare coada existenta deschisa in aplicatie.

# *.Analiza problemei*

Scopul aceste aplicatii este de a simula cat mai aproape de realitate o coada la un magazin la care asteapta un individ pentru a fi servit. Aceste cozi functioneaza pe principiul primul venit, primul servit, ceea ce se implementeaza cu o coada de tipul FIFO.

Aceste cozi vor reprezenta casele de marcat si vor contine clienti care asteapta sa fie procesati. Operatiile unei cozi de tip FIFO sunt enqueue, adica adaugarea la sfarsit, care reprezinta in aceasta aplicatie sosirea unui nou client la casa de marcat si dequeue, care va reprezenta incheierea cu succes a procesarii clientului aflat la inceputul cozii.

Un thread este un fir de lucru care ruleaza in interiorul programului. Un program poate contine mai multe thread-uri, care functioneaza in paralel si impart resursele alocate procesului.

In realitate, la un magazin, clientii vin la diferite momente de timp si se aseaza la o anumita coada, in cazul in care sunt disponibile mai multe, in functie de anumite criterii. Pentru a simula acest lucru, vom folosi un thread care, pe tot parcursul simularii, va genera clienti aleatori.

Procesarea unui client la casa de marcat se realizeaza din nou cu ajutorul unui thread, care pe tot parcursul simularii verifica daca clientul a asteptat suficient la casa de marcat pentru a putea fi procesat si eliminat din coada.

Aplicatia functioneaza in urmatorul mod:

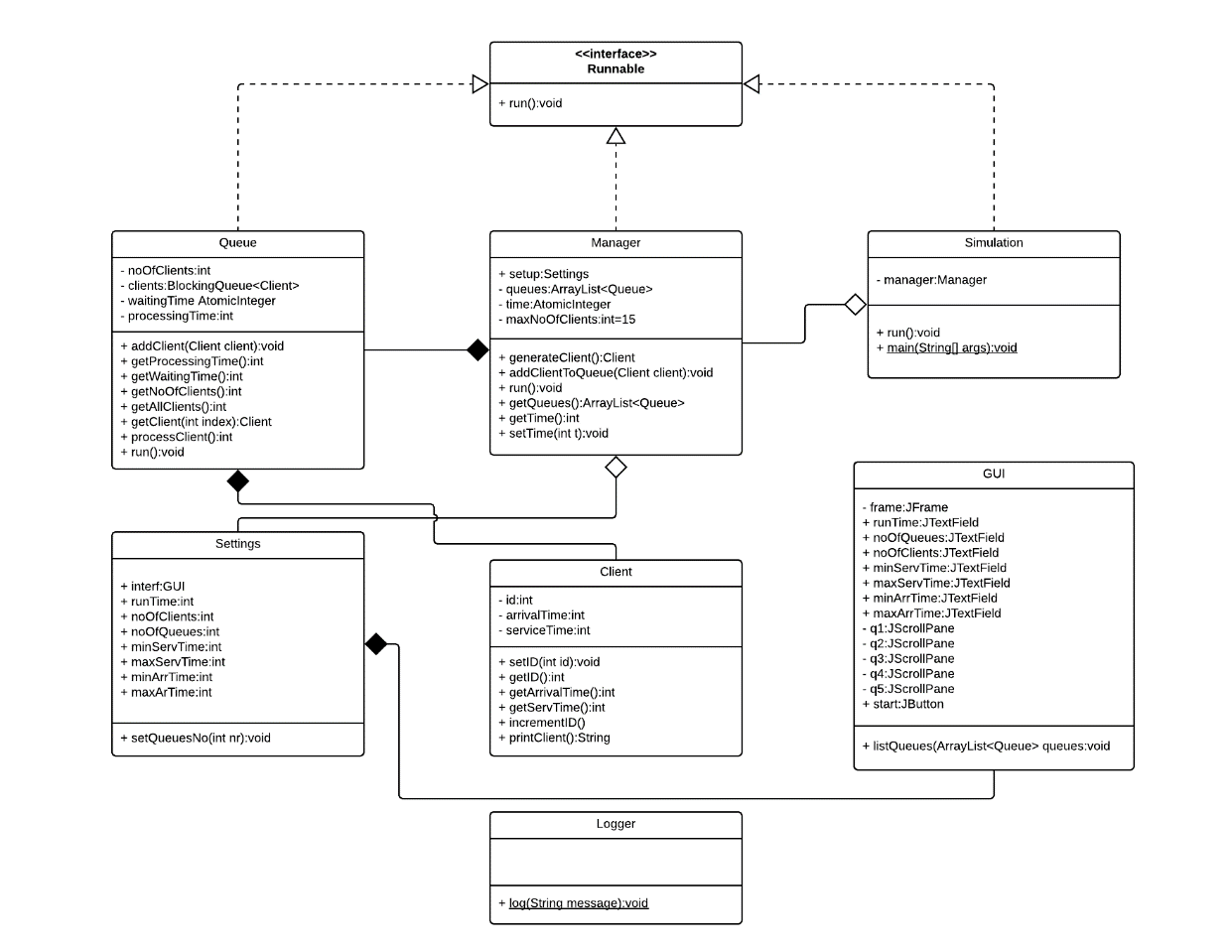
1. Se seteaza anumiti timpi necesari pentru realizarea simularii, anume: timpul de rulare al simularii, numarul de cozi deschise, numarul de clienti care vor urma sa fie generati, intervalul de timp in care poate ajunge urmatorul client, intervalul de timp necesar pentru procesarea unui client.
2. Dupa efectuarea setarii, se apasa butonul de *Start* care va porni simularea. Interfata afiseaza cozile si clientii generati, dar si modul in care acestia se aseaza la o coada si procesarea lor.
3. La finalul simularii, dupa ce a trecut timpul setat in prima parte, intr-un fisier se gasesc anumite informatii referitoare la executia simularii

# *Proiectare*

Proiectul respecta structura MVC, modelul fiind reprezentat de clasele care reprezinta clientul si coada, view-ul este reprezentat de interfata grafica, iar cotrolul este realizat de un manager care genereaza clienti si ii trimite la coada la care urmeaza sa astepte sa fie procesati.

Actiunile care se realizeaza in aceasta simulare trebuie sa se execute simultan, astfel ca am folosit thread-uri. Pentru functionarea corecta, pe langa modificatorul *synchronized* care permite ca metoda sa fie accesata de catre un singur thread la un anumit moment, am folosit BlockingQueue si AtomicInteger care sunt structuri ce realizeaza sinconizarea in mod automat.

BlockingQueue este o interfata care implementeaza o coada care asteapta ca aceasta sa nu fie goala, pentru a face dequeue, si pentru enqueue, asteapta sa se elibereze din spatiu in cazul in care este plina.

Aplicatia respecta paradigmele OOP; clasele existente in proiect au variabile private, putand fi modificate si accesate prin getters si setters.

# *Implementare*

Aplicatia contine 7 clase: Client, Queue, Manager, Settings, GUI, Simulation si Logger.

CLIENT

Clasa Client modeleaza un client. Acesta are 3 campuri: ID, arrivalTime care reprezinta la ce secunda din simulare ajunge clientul la una din cozi si serviceTime care reprezinta numarul de secunde in care este procesat clientul, adica cate secunde va astepta inainte sa plece cand ajunge la inceputul cozii.

Timpii arrivalTime si serviceTime se calculeaza aleator, in functie de setarile facute la inceputul simularii.

Metodele clasei sunt setters si getters pentru a seta Id-ul clientului si pentru a obtine timpii amintiti mai sus.

QUEUE

Clasa Queue reprezinta coada la care clientul se aseaza si isi asteapta randul pentru a fi procesat. Campurile acestei clase sunt: noOfClients care reprezinta cati clienti au stat la coada respectiva, un BlockingQueue<Clients> care reprezinta coda de clienti, un AtomicInteger care reprezinta timpul de asteptare la coada in fiecare secunda a simularii, si processingTime care reprezinta totalul timpilor de service ai clientilor care au stat la coada respectiva.

O metoda importanta este metoda addClient, care adauga un client in lista de clienti, modifica timpul de asteptare al cozii si numarul de clienti al cozii.

O alta metoda importanta este metoda processClient care returneaza primul client.

Clasa Queue implementeaza interfata Runnable, iar metoda run() apeleaza metoda de procesare a unui client, pune thread-ul in starea de sleep pentru un timp egal cu impul de procesare al clientului pe care il proceseaza, dupa care il elimina din lista.

MANAGER

Aceasta clasa se ocupa de generarea clientilor si asignarea lor in cozi. Campurile ei sunt: un ArrayList<Queues> care reprezinta cozile existente, o variabila setup de tipul Settings care contine valorile datelor introduse in GUI, un AtomicInteger time, care reprezinta timpul simularii si un vector de thread-uri care vor fi cozile.

Constructorul acestei clase creeaza thread-urile pentru cozi si le porneste. Pentru a nu complica programul, am decis sa creez 5 cozi de la inceput si in functie de cate cozi alege utilizatorul in partea de setare, atatea vor fi active.

In aceasta clasa exista o metoda generateClient care creeaza un nou client, cu ajutorul parametrilor introdusi in partea de setare. Metoda addClientToQueue adauga clientii in cozi. In lumea reala, un om alege coda la care sa astepte in functie de cati oameni mai sunt inaintea lui sau in functie de cate produse au acestia. Pentru a face acest lucru posibil in simulare, in aceasta metoda se determina care dintre cozile deschise are cei mai putini clienti si care are cel mai mic waitingTime. Dupa ce se determina acest lucru, clientul este adaugat ori la coada cea mai scurta, ori la coada cu cel mai mic timp de asteptare.

La fel ca si clasa Queue, Manager este de fapt un thread, deci implementeaza interfata Runnable.

In metoda run() a aceste clase se intampla urmatoarele lucruri: se creeaza un nou AtomicInteger t, care va reprezenta fiecare secunda a simularii. Se genereaza un client nou. Campul arrivalTime al acestui client va reprezenta la ce secunda a simularii acesta ar trebui sa ajunga la coada. Acest arrivalTime al clientului generat va fi salvat in variabila time. Dupa generarea clientului se verifica daca timpul curent al simularii, salvat in t, este egal cu time, timpul la care se aseaza cleintul la coada. Daca acesti timpi sunt egali, se adauga clientul la una din cozi, cu ajutorul metodei addClientToQueue. In fiecare secunda a simularii, thread-ul se pune in starea sleep pentru o secunda. Daca clientul curent a fost adaugat la una din cozi, se genereaza un nou client si repeta aceeasi pasi.

Deoarece utilizatorul alege cati clienti vor fi generati pe parcursul simularii, in cazul in care timpul simularii nu este destul de lung iar intervalele setate pentru timpul de asteptare al unui client si timpul de sosire, este posibil sa nu se genereze toti clientii, pentru ca se ajunge mai repede la sfarsitul simularii.

De asemenea, daca timpul de asteptare este foarte mare, iar simularea este prea scurta, in cazul in care mai sunt clienti la sfarsitul timpului de simulare in coada, acestia nu vor mai fi procesati.

Toate aceste detalii legate de cum se termina simularea sunt vazute in loggerul generat.

In plus, am decis sa fac o limita de clienti pentru cozi. Am ales sa nu las mai mult de 15 clienti la o coada, astfel, in cazul in care o coada este plina, adica are 15 clienti, se va deschide o noua coada automat. Acest lucru se face tot in metoda run(), unde se verifica de fiecare data numarul de clienti din cozi, cu fiecare secunda care trece din simulare.

SETTINGS

Aceasta clasa contine actionListeners pentru campurile din interfata grafica. Constructorul acestei clase ia datele introduce de utilizator si le salveaza in niste variabile folosite ulterior in clasa Manager, pentru a genera clientii cu intervalele stabilite si pentru a stii cat timp trebuie simularea sa ruleze. Pentru a fi mai usor de folosit si a nu scrie foarte multi getters am decis sa las variabilele acestei clase publice.

GUI

Clasa GUI reprezinta interfata aplicatiei. Pentru a reprezenta cozile am folosit 5 JScrollPane in care afisez un JList care contine clientii din fiecare coada.

Aceasta clasa are o singura metoda, si anume listQueues, care creeazi 5 JList-uri care contin clientii. Aceasta metoda este apelata de catre thread-ul Manager ca sa afiseze in fiecare secunda a simularii continutul cozilor.

SIMULATION

Aceasta clasa este folosita doar pentru a crea thread-ul pentru Manager si a-l porni.

LOGGER

Clasa Logger este folosita pentru a crea fisierul in care vor fi scrise datele despre evolutia simularii. Ea contine o singura metoda statica care realizeaza scrierea unei linii noi in fisier.

# *Rezultate*

Am rulat simularea pentru urmatoarele date de intrare:

Timp de rulare: 50

Nr. cozi: 2

Nr. clienti: 22

Timp sosire min:1 max:3

Timp de asteptare min:3 max:6

Rezultatele acestei simulari sunt:

Starting simulation...

t=2 Client 1 arrived at queue 1 and will have to wait 3s.

t=4 Client 2 arrived at queue 2 and will have to wait 3s.

t=5 Client 3 arrived at queue 1 and will have to wait 4s.

t=6 Client 4 arrived at queue 1 and will have to wait 5s.

t=7 Client 5 arrived at queue 2 and will have to wait 4s.

t=9 Client 6 arrived at queue 2 and will have to wait 5s.

t=11 Client 7 arrived at queue 1 and will have to wait 5s.

t=13 Client 8 arrived at queue 2 and will have to wait 4s.

t=14 Client 9 arrived at queue 1 and will have to wait 4s.

t=15 Client 10 arrived at queue 2 and will have to wait 4s.

t=16 Client 11 arrived at queue 1 and will have to wait 4s.

t=18 Client 12 arrived at queue 2 and will have to wait 4s.

t=20 Client 13 arrived at queue 1 and will have to wait 5s.

t=22 Client 14 arrived at queue 2 and will have to wait 5s.

t=23 Client 15 arrived at queue 1 and will have to wait 4s.

t=25 Client 16 arrived at queue 2 and will have to wait 4s.

t=26 Client 17 arrived at queue 1 and will have to wait 4s.

t=28 Client 18 arrived at queue 2 and will have to wait 4s.

t=29 Client 19 arrived at queue 1 and will have to wait 3s.

t=30 Client 20 arrived at queue 2 and will have to wait 5s.

t=31 Client 21 arrived at queue 1 and will have to wait 3s.

t=33 Client 22 arrived at queue 2 and will have to wait 3s.

... Simulation ended

Simulation ran for 50 seconds.

Generated clients: 22/22

Average waiting time per client: 4.0454545

Peak hour was at second 31 with a total of 8 clients.

Queue 1 served 11 clients with a total of 44s as waiting time and was empty for 9s.

Queue 2 served 11 clients with a total of 45s as waiting time and was empty for 7s.

Queue 3 served 0 clients with a total of 0s as waiting time and was empty for 0s.

Queue 4 served 0 clients with a total of 0s as waiting time and was empty for 0s.

Queue 5 served 0 clients with a total of 0s as waiting time and was empty for 0s.

Pentru urmatoarele setari, timpul de rulare este prea scurt si procesarea clientilor nu ajunge la sfarsit.

Timp de rulare: 40

Nr. cozi: 2

Nr. clienti: 25

Timp sosire min:1 max:3

Timp de asteptare min:5 max:10

Rezultatul acestei simulari este urmatorul:

Starting simulation...

t=1 Client 1 arrived at queue 1 and will have to wait 8s.

t=3 Client 2 arrived at queue 2 and will have to wait 5s.

t=5 Client 3 arrived at queue 2 and will have to wait 8s.

t=6 Client 4 arrived at queue 1 and will have to wait 8s.

t=7 Client 5 arrived at queue 2 and will have to wait 8s.

t=9 Client 6 arrived at queue 1 and will have to wait 5s.

t=11 Client 7 arrived at queue 1 and will have to wait 9s.

t=12 Client 8 arrived at queue 2 and will have to wait 9s.

t=13 Client 9 arrived at queue 1 and will have to wait 7s.

t=14 Client 10 arrived at queue 2 and will have to wait 6s.

t=16 Client 11 arrived at queue 2 and will have to wait 7s.

t=18 Client 12 arrived at queue 1 and will have to wait 9s.

t=20 Client 13 arrived at queue 1 and will have to wait 6s.

t=21 Client 14 arrived at queue 2 and will have to wait 7s.

t=22 Client 15 arrived at queue 1 and will have to wait 7s.

t=24 Client 16 arrived at queue 2 and will have to wait 8s.

t=26 Client 17 arrived at queue 1 and will have to wait 7s.

t=28 Client 18 arrived at queue 2 and will have to wait 5s.

t=30 Client 19 arrived at queue 1 and will have to wait 8s.

t=31 Client 20 arrived at queue 1 and will have to wait 6s.

t=33 Client 21 arrived at queue 2 and will have to wait 7s.

t=34 Client 22 arrived at queue 2 and will have to wait 5s.

t=36 Client 23 arrived at queue 2 and will have to wait 5s.

t=38 Client 24 arrived at queue 1 and will have to wait 7s.

... Simulation ended

Simulation ran for 40 seconds.

Generated clients: 25/25

Average waiting time per client: 6.68

Peak hour was at second 35 with a total of 14 clients.

Queue 1 served 12 clients with a total of 87s as waiting time and was empty for 1s.

Queue 2 didn't finish processing the clients. 7 client(s) remaining.

Queue 2 served 12 clients with a total of 80s as waiting time and was empty for 3s.

Queue 3 didn't finish processing the clients. 7 client(s) remaining.

Queue 3 served 0 clients with a total of 0s as waiting time and was empty for 0s.

Queue 4 served 0 clients with a total of 0s as waiting time and was empty for 0s.

Queue 5 served 0 clients with a total of 0s as waiting time and was empty for 0s.

# *Concluzii*

Pentru aceasta tema am invatat cum se folosesc thread-urile si am descoperit cum se pot folosi anumite structuri de date pentru realizarea sincronizarii in mod transparent.

O viitoare dezvoltare a aplicatiei o reprezinta adaugarea unei logici care sa inchida o coada daca aceasta este goala sau inchiderea unei cozi daca numarul total de clienti este mic in comparatie cu numarul de cozi deschise. In acest caz, ar trebui implementat un algoritm care sa nu mai permita adaugarea unor noi clienti la o coada care s-a decis ca se va inchide dupa ce termina de procesat clientii existenti.

# *Bibliografie*

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html>

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html>

<https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html>