SIMULATOR DE COZI

DOCUMENTATIE

Munteanu Cezar-Lucian

Grupa 30226

Profesor Coordonator : Dorin Vasile Moldovan

Contents

1. Cerinte Functionale …………………………………………………… 3
2. Obiective ………………………………………………………………... 3

2.1 Obiectiv principal …………………………………………………. 3

2.2 Obiective secundare …………………………………………….. 3

3. Analiza Problemei ……………………………………………………… 4

4. Proiectare ………………………………………………………………... 5

4.1 Structuri de date …………………………………………………… 5

4.2 Diagrama de clase ………………………………………………... 5

5. Implementare …………………………………………………………… 6

6. GUI ……………………………………………………………………….. 12

7. Concluzii si posibilitati de dezvoltare ……………………………… 13

8. Bibliografie ……………………………………………………………… 14

1. Cerinte Functionale

Se cere dezvoltarea si implementarea unei aplicatii care doreste sa analizeze sistemele niste sisteme bazate la randul lor pe cozi pentru a optimiza si minimiza timpul de asteptare al clientilor .

1. Obiective
   1. Obiectiv principal :

Trebuie dezvoltata o aplicatie care sa simuleze intr-un timp anume dat, de secunde , repartizarea unor client cu timp de sosire si servisarea aleatory ( intr-un interval stabilit inainte ) la un numar de Q cozi de asteptare . Acest sistem are ca scop optimizarea timpului de asteptare de catre client la cozi pana cand ajung sa fie serviti . Aceasta aplicatie trebuie sa obtina statistici privind numarul maxim de client care au stat la o coada , ora de varf , timpul mediu de asteptare si timpul mediu de servisare .

* 1. Obiective secundare :

Pentru a putea ajunge la obiectivul final trebuie ca problema mare sa fie impartita in probleme mai mici, respective obiective mai mici . Asadar obiectivele secundare sunt :

* **Dezvoltarea de use cas-uri si scenarii :** reprezinta scenariile in care programul merge , respective cazurile in care merge si modul in care trebuie utilizat programul pentru ca simularea sa fie cat mai reala si fara erori produse din cauza thread-urilor.
* **Alegerea structurilor de date :** Pentru a putea lucra cu thread-uri avem nevoie sa alegem niste structuri de date care sa fie thread-safe , cu alte cuvinte , niste structure de date care pot sa fie accesate doar de un singur thread la un moment dat . Ca de exemplu : synchronizedList.
* **Dezvoltarea algoritmilor :** pentru a putea implementa simularea de thread-uri trebuie sa ne gandim la diferiti algoritmi pentru a putea simula comportamentul concurrent al mai multor cozi.
* **Verificarea rezultatului :** Pentru a vedea cum s-a comportat simularea, trebuie sa verificam pe tot parcursul ei , fiecare secunda rezultatul generat .

1. Analiza problemei

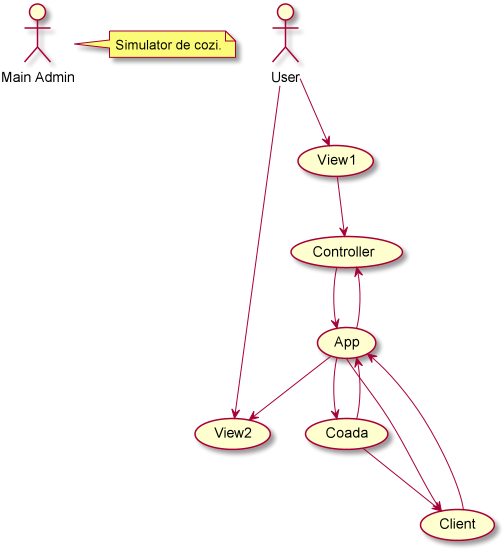


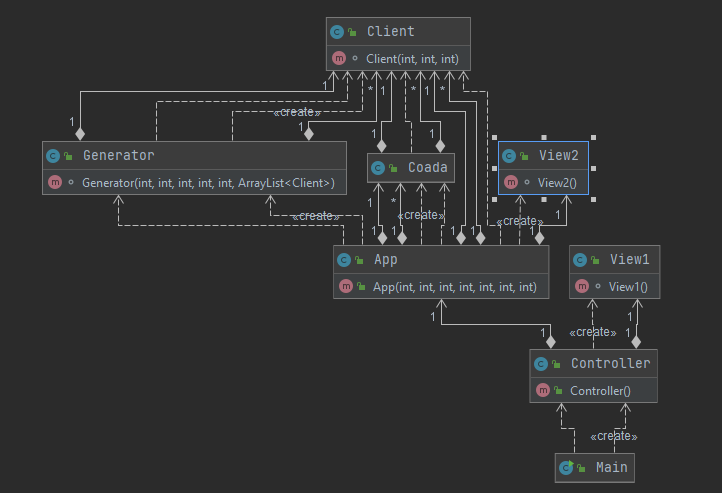
Diagrama USE-CASE reprezinta cea mai simla reprezentare a interactiunii unui utilizator cu sistemul/programul , astfel intelegand procesul de interactiune mult mai usor . In cazul acestei probleme USE-CASE-ul este unul relative simpu , pentru ca avem doar un utilizator care poate interactiona doar cu interfata grafica . Prin urmare , un utilizator va folosi interfata , va introduce datele necesare pentru desfasurarea simularii ; timpul de simulare ; numarul de client , cozi , si timpii maximi si minimi de servisare si de sosire in zonele de text respective . La apasarea butonului Adauga, o noua fereastra se va afisa unde va avea loc simularea respectiva .

1. Proiectare
   1. Structuri de date

Pentru a retine clientii care trebuie generati la inceputul simularii am ales sa folosesc o structura de tip ArrayList pentru a sti mereu clientul care trebuie sa fie procesat de catre thread-uri in urma sortarii dupa timpul de sosire . Pentru a putea simula comportamentul unei cozi utilizand thread-uri , trebuie sa fim foarte atenti la ce structure de date folosiim pentru ca thread-urile impart aceeasi zona din memorie din program . Eu am ales sa construiesc pentru fiecare coada , o lista specifica de client , care ii corespunde fiecarui thread / coada in parte , pentru a nu aparea situatia cand doua cozi proceseaza acelasi client . Clientul va fi stability de catre o unitate de tip “manager” si il va atribui thread-ului corespunzator . Pentru a tine mai multe cozi, am ales sa folosesc un arrayList pentru ca am nevoie de index-ul cozii pentru a putea determina unde va ajunge clientul . Am ales aceste structuri pentru ca reprezinta un mod efficient de a face aceste lucruri pentru ca nu se pierde timp la selectarea unui thread si nici nu facem risipa de memorie , alocam cat este nevoie .

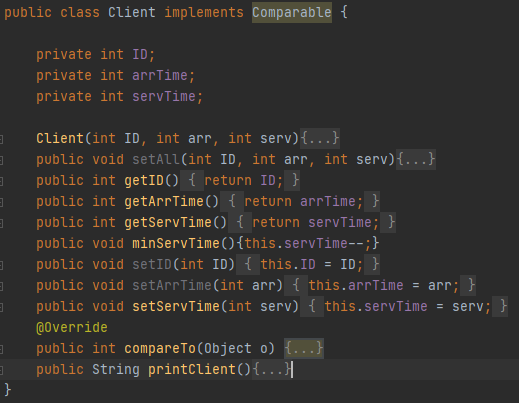
* 1. Diagrama de clase

Diagrama de clase (UML) este o structura de diagrama statica care descrie structura unui sistem afisand clasele , atributele si metodele implementate si relatia dintre obiectele proiectului . Mai jos avem reprezentata o diagrama UML simplificata pentru a putea urmarii mai usor relatiile dintre clase . Dupa cum putem observa , intre clasele noastre apar relatii de agregare , compoziie si asiciere .



1. Implementare

Clasa Client:



In aceasta clasa realizam modelul unui client, care din cerinta ni se zice ca va avea un id , un timp de sosire si un timp pe care il pentrece cand este servit . Cu ajutorul acestei clase vom putea genera ulterior un sir de client, intr-un mod aleator , si cu ajutorul interfetei Comparable , reuseste sa sorteze crescator dupa timpul de sosire . Avem metoda printClient() pentru a putea afisa un client sub forma de String (id, arrTime, servTime) .

Aceasta clasa mai contine si metode de get si set pentru fiecare particularitate in parte dintre cele ale clientului dat . . .

Clasa Coada :



Aceasta clasa sta la baza simulraii si tot odata aceasta clasa estea aceea ce va implementa un thread . Am ales sa mostenesc clasa Thread pentru ca nu avem nevoie de mosteniri de alte clase . Acest thread va simula o coada . Pentru a evita problemele sincronizarii cu alte thread-uri, clasa Coada are o lista de client care semnifica clientii care asteapta sa fie procesati . Pe langa aceste arratList avem si dimensiunea cat si poz la care se afla un client in aceasta date ca date.

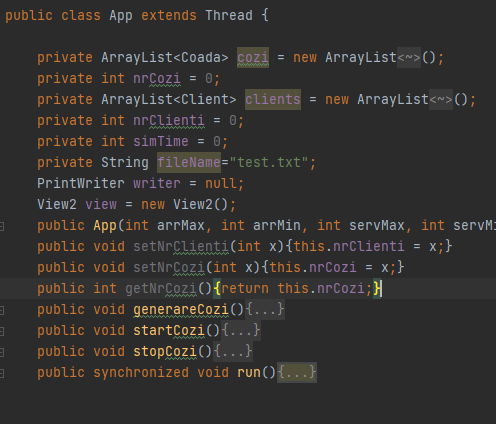
Pentru a putea minimiza timpul de asteptare pe care il au clientii , clasa Coada continua cu un waitingTime . Asadar , clientul urmator care va fi adaugat la coada va creste acest waitingTime . Mai mult acest waitingTime ne ajuta sa selectam care client trebuie sa fie adaugat in coada respective bazandu-ne pe care coada are waitingTime-ul mai mic.

public synchronized void run()

Aceasta functie run, suprascrisa , modeleaza comportamentul thread-ului dup ace va fi apelata metoda .start() . Aici vrem sa simulam comportamentul unei cozi individuale , asadar in aceasta functie verificam daca coada curenta poceseaza un client , daca da , atunci dup ace asteapta o secunda , ii va scadea o unitate din timpul de servisare , daca ajunge la 0 atunci se va oprii prin intermediul variabilei Boolean tureValue , semnaland ca in momentul current niciun element nu este procesat . In acest caz , cauta in lista specifica urmatorului client .

Totodata, aceasta clasa mai contine metode si de set si get care ajuta la obtinerea si setarea datelor particularizate despre fiecare coada in parte cat si metode de adaugare sau stergere din coada si inca cateva care se leaga de waitingTime .

Clasa App:



Despre aceasta clasa putem sa spunem ca este thread-ul de control . Aceste thread se opreste cand timpul current este mai mare decat timpul de simulare dat . In acest timp cautam in lista de client persoanele care trebuie sa ajunga la timpul current . Daca timpul de sosire respective nu este egal cu cel al simularii , atunci pe baza rationamentului descries mai sus , luam clientii si ii bagam in cozile respective . Stergem din lista de client pe care i-am procesat . Dupa aceea thread-ul o sa atepte o secunda , atunci crestem timpul current al simularii . Acest procedeu se reia .

List<Client> lista = Collections.*synchronizedList*(this.clients);

De mentionat ca aici am folosit Collections.synchronizedList care ne ajuta la sincronizarea thread-urilor.

Tot aici ne folosim de librariile:

import java.io.FileNotFoundException;  
import java.io.PrintWriter;

Cu ajutorul acestora noi vom putea genera fisierul in care sa scriem rezultatele cerute in urma simularii cozilor, mai exact cele trei fisiere de test cerute in cerinta cat si timpii medii si peak hour-ul.

Metoda

public synchronized void run()

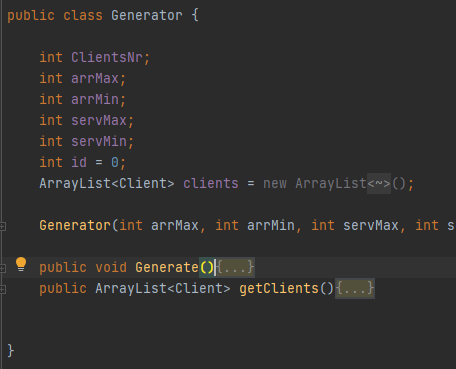
Aceasta este functia de control a thread-ului care va controla la randul ei celelalte thread-uri formate din cozi si va afisa pe interfata cat si in fisierul generat.

Tot aici mai avem inca doua metode importante ce constau in pornirea si oprirea thread-urilor de tip coada mai exact startCozi() si stopCozi() . Acestea doua iau fiecare coada in parte si le aplica metoda start() respective stop().

Totodata , clasa App mai contine si niste metode primitive cum ar fi setNrClienti, setNrCozi, respective getNrCozi , care ajuta la modificarea mai usoara a elementelor clasei.

In aceasta clasa array-ul de client va fi generat cu ajutorul clasei Generator care va genera random clientii.

Clasa Generator:

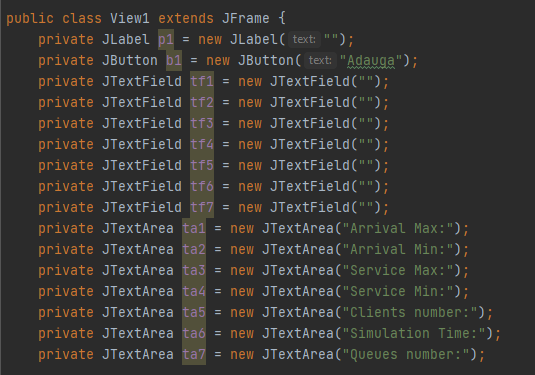


Aceasta clasa va genera clientii random si ii va salva intr-un Array de tip client pe care il va returna la final.

Pentru a genera random datele vom folosi libraria java.util.Random si vom apela functia random.nextInt().

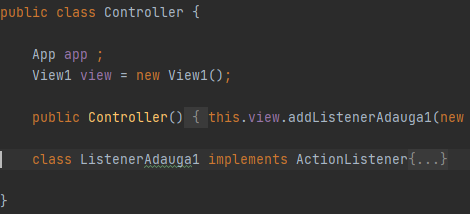
Totodata, in aceasta clasa o sa sortam clientii genrati random , cu ajutorul Collections.sort(); care ne sorteaza clientii dupa cum am suprascris in clasa Cliet dupa arrTime.

Clasa View1



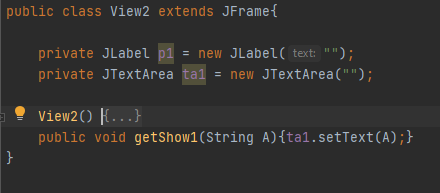
Aceasta clasa creaza view-ul initial pe care user-ul il are. Contine mai multe campuri de text , in care user-ul va trebui sa introduca datele cerute pentru a efectua simularea cat si un buton care v-a efectua actiunea.

Clasa Controller



Aceasta clasa preia inputul din interfata initiala si il va trimite mai departe la mode . Aici se implementeaza actiunea de definitie a butonului Adauga , care va deschide o noua fereastra.

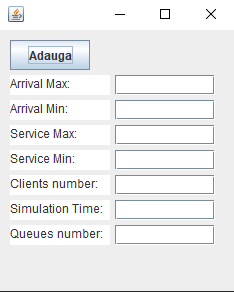
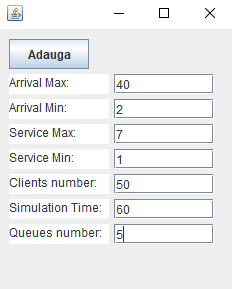
Clasa View2



Intrucat la inceputul interfetei grafice preluam datele , numarul de cozi si numarul de client . Nu putem sa stim dinainte cate cozi vom avea sau informatiile despre client , asadar , dupa ce aflam aceste lucruri construim un nou Frame unde va avea loc simularea . Se va afisa Q JPane-luri care vor contine numele fiecarei cozi , iar apoi se vor crea text field-uri care vor afisa pentru fiecare coada individual , clientul current si coada de asteptare specifica cat si datele despre client si la final mediile cerute cat si peak hour .

1. GUI ( Graphic User Interface )

Pentru realizarea interfetei grafice am folosit pachetul Java.Swing . Am folosit un frame in care am introdus mai multe panel-uri , urmand apoi sa folosesc functiile din librarie pentru a le aseza dupa bunul plac .

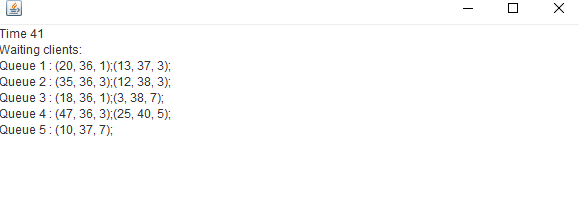
 

Aceasta este prima fereastra pe care utilizatorul o vede . Aici trebuie sa introduca in text field-ruile corespunzatoare , niste date valide pentru a se realiza correct simularea , iar apoi trebuie sa apese pe butonul Adauga .

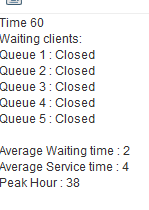
La apasarea acestui buton , o noua fereastra se va deschide , si aici se va afisa simularea cozilor cu toate datele precizate mai sus la clase . Avem in partea de sus , lista de client generate , iar in dreptul fiecarei cozi avem clientul current care este procesat .

Langa el avem lista de asteptare . Putem observa cum un client din lista initiala intra in coada si este procesat .

Totodata putem observa si timpul la care se afla simularea cat si numarul pentru fiecare coada in parte .



Dupa terminarea simularii , la final se vor afisa si mediile cerute cat si peak hour , dupa cum este precizat in cerinta .



1. Concluzii si posibilitati de dezvoltare

Acest proiect m-a ajutat foarte mult in intelegerea thread-urilor , sincronizarea lor si modul in care trebuie sa lucreze . Nu am mai lucrat cu fire de lucru pana acum , si mi-a facut placer sa invat lucruri noi , cum ar fi si synchronizedList() .

Ca posibilitati de dezvoltare s-ar putea incerca imbunatatirea semnalului de ceas , pentru a inlatura orice decalaj intre thread-ul din main si celelalte thread-uri . Practic atucni cand thread-ul main sta o secunda , atunci sa se anunte celelalte thread-uri ca a trecut o secunda , astfel nu am mai rebui sa fie anumite decalaje la afisare . O imbunatatire considerabila se poate face asupra interfetei grafice, s-ar putea adauga culori si animatii pentru fiecrare client care intra in coada si totodata iese .

1. Bibliografie

* <https://www.baeldung.com/java-synchronized-collections> ;
* <https://www.tutorialspoint.com/java/java_multithreading.htm> ;
* <https://www.geeksforgeeks.org/synchronized-in-java/> ;