Analiza unui sistem de gestiune al unor cozi

Gherghel Daniel-Andrei

Grupa 30224

Îndrumător de laborator: Claudia Pop

Cuprins

[1. Cerințe funcționale 3](#_Toc5120257)

[2. Obiective 3](#_Toc5120258)

[2.1. Obiectiv principal 3](#_Toc5120259)

[2.2. Obiective secundare 3](#_Toc5120260)

[3. Analiza problemei 3](#_Toc5120261)

[4. Proiectare 6](#_Toc5120262)

[4.1. Diagrama de clase UML 6](#_Toc5120263)

[4.2. Clase și algoritmi folosiți 6](#_Toc5120264)

[5. Concluzii și dezvoltări ulterioare 8](#_Toc5120265)

[6. Bibliografie 9](#_Toc5120266)

# Cerințe funcționale

Să se implementeze o aplicație care are scopul de a analiza sistemele bazate pe cozi, pentru a determina și minimize timpul de așteptare al clienților.

# Obiective

## Obiectiv principal

Obiectivul principal al proiectului este de a dezvolta o aplicație care simulează un sistem de cozi.

## Obiective secundare

* Dezvoltarea de use case și scenarii
* Proiectarea claselor
* Alegerea structurilor de date
* Dezvoltarea algoritmilor
* Implementarea soluției

# Analiza problemei

Un sistem de Queue Management (sistem de management a cozilor) este o soluție de gestionare a timpului petrecut de clienți la coada de așteptare și de calculare a numărului de persoane care circulă în companii, centre de servicii, bănci, instituții publice, spitale și clinici medicale.

Introducerea unui astfel de sistem duce interacțiunea clienților cu serviciile oferite la un nivel superior prin creșterea satisfacției acestora și a eficienței personalului.

Motive pentru a folosi un sistem de management a cozilor :

* Reducerea timpului de așteptare al clienților ;
* Timpul de așteptare este cunoscut pentru clienți ;
* Creșterea eficienței personalului ;
* Obținerea de statistici referitoare la perioadele de timp.

In executarea aceste lucrari am utilizat threadurile si tehnica numita Multithreading.

“Multithreading” înseamnă capacitatea unui program de a executa mai multe secvenţe de

cod în acelaşi timp. O astfel de secvenţă de cod se numeşte fir de execuţie sau thread. Limbajul

Java suportă multithreading prin clase disponibile în pachetul java.lang. În acest pachet există 2

clase Thread şi ThreadGroup, şi interfaţa Runnable. Clasa Thread şi interfaţa Runnable oferă

suport pentru lucrul cu thread-uri ca entităţi separate, iar clasa ThreadGroup pentru crearea unor

grupuri de thread-uri în vederea tratării acestora într-un mod unitar.

Există 2 metode pentru crearea unui fir de execuţie: se creează o clasă derivată din clasa

Thread, sau se creează o clasă care implementeză interfaţa Runnable.

Crearea unui fir de execuţie prin extinderea clasei Thread

Se urmează etapele:

- se creează o clasă derivată din clasa Thread

- se suprascrie metoda public void run() moştenită din clasa Thread

- se instanţiază un obiect thread folosind new

- se porneşte thread-ul instanţiat, prin apelul metodei start() moştenită din clasa Thread.

Apelul acestei metode face ca maşina virtuală Java să creeze contextul de program

necesar unui thread după care să apeleze metoda run().

Metoda main() are propriul fir de execuţie. Prin apelul start() se cere JVM crearea şi

pornirea unui nou fir de execuţie. Din funcţia start() se va ieşi imediat. Firul de execuţie

corespunzător metodei main() îşi va continua execuţia independent de noul fir de execuţie creat.

Implementările Java depind de platformă. Un program Java care foloseşte fire de execuţie

poate avea comportări diferite la execuţii diferite pentru aceleaşi date de intrare. Platformele

Windows folosesc cuante de timp (firele de execuţie sunt administrate într-o manieră RoundRobin).

Crearea unui fir de execuţie folosind interfaţa Runnable

Este o modalitate extrem de utilă atunci când clasa de tip Thread care se doreşte a fi

implementată moşteneşte o altă clasă (Java nu permite moştenirea multiplă). Interfaţa Runnable

descrie o singură metodă run().

Se urmează etapele:

- se creează o clasă care implementează interfaţa Runnable

- se implemetează metoda run() din interfaţă

- se instanţiază un obiect al clasei folosind new

- se creează un obiect din clasa Thread folosind un constructor care are ca parametru un

obiect de tip Runnable (un obiect al clasei ce implementează interfaţa)

- se porneşte thread-ul creat la pasul anterior prin apelul metodei start().

Un fir de execuţie se poate afla la un moment dat în una din următoarele stări: running

(rulează), waiting (adormire, blocare, suspendare), ready (gata de execuţie, prezent în coada de

aşteptare), dead (terminat). Fiecare fir de execuţie are o prioritate de execuţie. În general threadul cu prioritatea cea mai mare este cel care va accesa primul resursele sistem.

O problemă de excludere mutuală este exemplul producător-consumator. Producătorul

furnizează date pe care consumatorul le utilizează mai departe. Ce se întâmplă dacă vitezele de

producere şi consum diferă? În acest scop se utilizează metodele wait(), notify() şi notifyAll() ale

clasei Object. Apelul lui wait() va trece obiectul apelat în starea Blocked. El rămâne blocat până

la apelul unei metode notify() sau notifyAll() pentu acelaşi obiect. Condiţia necesară pentru a se

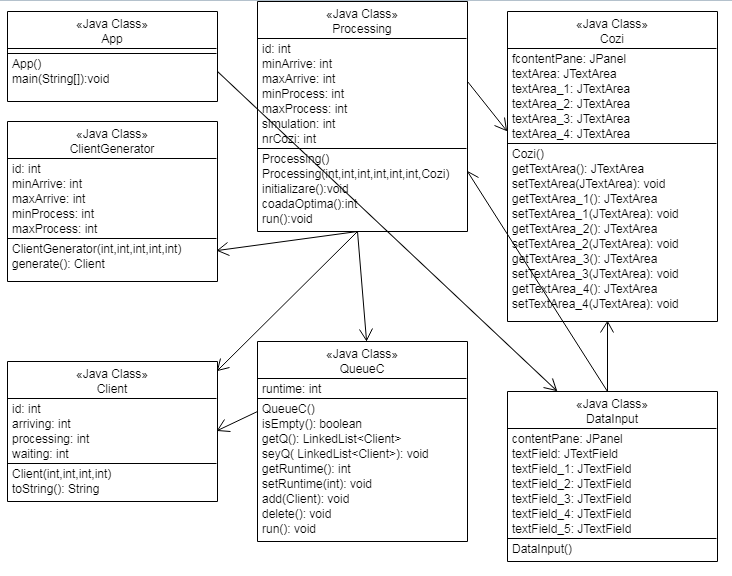
apela una dintre aceste metode este ca apelul lor să se facă în interiorul metodelor synchronized.

Pentru metoda wait() se poate specifica o durată maximă de aşteptare. În acest caz, firul de

execuţie râmâne blocat până când timpul expiră sau un alt fir de execuţie apelează notify().

# Proiectare

## Diagrama de clase UML



## Clase și algoritmi folosiți

Aplicatia contine un pachet, PT2019.demo.Assignment2 , care este alcatuit din 7 clase.

Clasa **App** contine functia main de unde se porneste practice apicatia, la rularea acesteia de creeaza o fereastra unde pot fi introduse informatiile pe care se doreste sa se faca simularea.

Clasa **Client** contine informatiile cu privire la o persoana care doreste sa intre intr-o coada si trebuie sa fie procesate. Informatiile unui client sunt urmatoarele: id, arriving time, processing time si waiting time. Waiting time reprezinta timpul total pe care clentul treebuie sa astepre pana cand va fi procesat. Aceasta durata de asteptare include si timpul propriu de procesare al fiecarui client.

Clasa are un constructor care primeste toate aceste informatii pentru a crea un obiect de tipul Client. Clasa mai contine si metodele de get si set pentru toate atributele sale.

Clasa **ClientGenerator** contine urmatoarele campuri: id, minArrive, maxArrive, minProcess si maxProcess. Constructorul primeste toate aceste informatii, cu care creeaza un obiect de tipul ClientGenerator. Clasa implementeaza metoda generate() care creeaza si returneaza un client cu un id dat, cu un waiting time initial 0, cu un timp de sosire generat random intre minArrive si maxArrive si un timp de procesare de asemenea generat random intre minProcess si maxProcess. Generarea random se face cu ajutorul unui obiect de tipul Random care apeleaza metoda .nextInt().

**int** arrive = rand.nextInt( ( maxArrive – minArrive ) + 1) + minArrive;

**int** process = rand.nextInt( ( maxProcess – minProcess ) + 1) + minProcess;

**return** **new** Client(**this**.id,arrive,process,0);

Clasa **QueueC** contine un runtime o coada sub forma unui linkedList si metodele care impleenteaza operatiile de: adaugare si stergere din coada.

Clasa contine urmatoarele metode: getQ() care returneaza coada propriu zisa, isEmpty() care verifica daca coada este vida, caz in care returneaza true, iar in caz contrar false, getRuntime() si setRuntime(), runtime-ul reprezentand intervalul de timp in functie de care se decide cat sa “doarma” threadul. Metoda add() primeste ca parametru un Client,aici se seteaza timpul de asteptare, waiting time, astfel, daca coada este goala atunci timpul de asteptare este timpul propriu de procesare, iar daca in coada exista déjà alti clenti atunci timpul sau de asteptare se calculeaza ca suma dintre timpul de asteptare al ultimului client plus timpul propriu de procesare.

Metoda delete() executa stergerea primului client din coada. Iar in final, ca ultima metoda se foloseste suprascrierea metodei run() care este executata de threadul care primeste coada respective, in prima faza se verifica daca coada este vida, in caz contrar se face un sleep pe thread de timpul de procesare inmultit cu runtime-ul cozii, dupa care se apeleaza metoda de stergere.

Clasa **Cozi** reprezinta o intervata care contine 5 textarea-uri, unul pentru fiecare coada, spatii unde sunt afisate in tim real continutul fiecarei cozi, reactualizarea lor efectuandu-se din secunde in secunda. In clasa DateInput se creeaza o noua instant a aceste clase, mai precis pe actiunea pusa pe butonul de start, de unde devine vizibila si se trensmite mai departe prin constructorul Processing.

Clasa **DataInput** reprezinta o interfata care este pornita din clasa App, unde este creata o instant a acestei clase si setata ca vizibila. Odata ce este pornita se pot introduce datele pe care se doreste sa se efectueze simularea. Prin apasarea butonului Start se executa urmatoarele comenzi: se extrag de pe textField-uri informatiile si sunt puse in variabile, se creeaza o instanta a clase Cozi si cu ajutorul acestor informatii se creeaza un obiect de tipul Processing. Pentru o rulare corecta a aplicatiei se creeaza un Thread pe noul obiect de tipul Processing si se porneste cu apelarea metodei start() deoarece am ales metoda de implementare a interfetei Runable.

Si in final clasa **Processing** care reprzinta motorul aplicatiei implementeaza urmatoarele metode si declara atributele necesare pentru implementarea metodelor.

In prima faza se declara atributele necesare pentru a putea fi preluate datele din interfata DataInput: id, minArrive, maxArrive, minProcess, maxProcess, simulation care reprezinta durata de simulare, practice intervalul de timp in care se genereaza client, numarul de cozi, dar l-am considerat constant si o declarare de obiect de tipul Cozi. De asemenea tot aici am declarat o lista de cozi, unde vor fi create si puse toate cozile din system.

In constructor se creeaza numarul dat de cozi, se seteaza runtime-ul pe o secunda si se adauga fiecare coada creeata in lista de cozi.

Prima metoda implementata este initializare(), care are rolul de a parcurge lista de cozi, de a prelua fiecare coada pe care se va initializa si porni cate un Thread.

Clasa contine de asemenea metodele de set si get pentru fiecare atribut al sau.

Urmatoarea metoda implementata, coadaOptima(), are rolul de a returna coada cea mai favorabila pe care trebuie sa o aleaga clientul. Astfel am considerat ca eficienta sa fie de ambele parti, si de cea a clientului si de cea a magazinului, in fiecare coada sa poata fi introdusi numai 3 clienti, in primul rand pentru o mai buna urmarire a functionalitatii, apoi se urmareste umplerea primei cozi, iar dupa umplerea acesteia sa se dechida o noua coada, dar deoarece generarea clientilor lucreaza in parlel cu procesarea lor, in momentul in care se proceseaza un client urmatorul client sa fie introdus in coada 1, astfel in lipsa clientilor sa poata fi usor de inchis o coada care nu mai este necesara. Astfel prin acesta metoda de alegere a numarului cozii, magazinul trebue sa tina deschis numarul minim necesar de cozi, iar pentru ca intr-o coada sunt doar 3 clienti, timpul de asteptare este relative mic.

Ultima metoda implementata reprezinta o suprascriere a metodei run(), unde are loc generarea si procesarea clientilor. In aceaste metoda am implementat doua while-uri, primul are ca numar de pasi intervalul de simulare, in interiorul acestuia apelez metoda initializare() pentru pornirea threadurilor, apoi generez un client nou pe baza unui index care creste la fiecare iteratie, aleg o coada optima, dupa care adaug clientul in coada aleasa. Dupa adaugarea clientului se face un sleep pe thread de o secunda , iar apoi se face o reactualizare a wating time-urilor cu ajutorul unui while care parcurge lista de cozi si pe urme fiecare coada, element cu element si in final se introduc in textArea continutul cozilor.

**if**(!cozi.get(0).getQ().isEmpty())

coziInterface.getTextArea().setText(cozi.get(0).getQ().toString());

Dupa terminarea primului while s-au generat un numar de clienti care nu s-au putut procesa in acelasi timp, asadar este nevoie de un al doilea while care sa se execute pana in momentul in care lista de cozi este vida. In interiorul acesteia am reimplementat actualizarea (popularea) textarea-urilor.

# Concluzii și dezvoltări ulterioare

Aplicația dezvoltată este ușor de folosit și simulează un sistem de management al cozilor, sistem întâlnit aproape la orice pas. Utilizatorul trebuie doar să introducă valori valide în câmpurile destinate și să apese pe butonul START.

Odata cu apasarea butonului start se incepe simularea care implica : generarea unui nou client la interval de o secunda, iar odata cu generarea primului client are loc pornirea threadurilor pe cozi, astfel fiecare coada nevida proceseaza indpendenta un client si in acelasi timp cu activitatea celorlate cozi. Activitatea de procesare a cozilor se realizeaza in paralel cu generarea clientilor. La terminarea intervalului de procesare se opreste procesul de generare al clientilor, urmand doar procesarea clientilor déjà generati.

O dezvoltare posibilă a aplicației ar putea fi creșterea numărului de clienți din fiecare coadă (în acest moment pot fi maxim 3 clienți la o coadă, deci în total maxim 15 de clienți). O astfel de aplicatie este foarte usor de utilizat si de implementat in orice magazin care lucreaza cu case de marcat sau orice institutie care se bazeaza pe asteptarea la cozi.

# Bibliografie

* <https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html>
* <https://stackoverflow.com/questions/5887709/getting-random-numbers-in-java>
* <https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>
* <https://www.geeksforgeeks.org/synchronized-in-java/>
* <http://inf.ucv.ro/documents/tudori/laborator8_53.pdf>
* <https://www.draw.io/>