**Documentatie-Tema 2**

**Queuing based systems**

***Calacean Ionut Eugen***

***Facultatea de automatica si calculatoare***

***Grupa 30229***

***Indrumator laborator:Pop Cristina***

**Cuprins**

1. **Obiectivul temei.**
2. **Analiza problemei, modelare, scenarii, cazuri de utilizare.**
3. **Proiectare(decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator).**
4. **Implementare.**
5. **Rezultate.**
6. **Concluzii.**
7. **Bibliografie.**

**1.Obiectivul temei**

Cozile sunt comun utilizate pentru a modela domeniile lumii reale. Principalul obiectiv al unei cozi este de a asigura un loc pentru un “client” ca acesta sa astepte inainte de a primi un “serviciu”. Gestionarea sistemelor bazate pe cozi este focusata pe minimizarea timpului scurs in care “clientii” asteapta in cozi inainte sa fie serviti. O metoda de a asigura minimizarea timpului de asteptare este adaugare unor servere de procesare( fiecare coada se considera ca are atasata un sistem de procesare) dar aceasta abordare duce la cresterea costului de intretinere a procesului de servire(angajati mai multi, case mai multe). Cand un nou server de procesare este adaugat, clientii in asteptare vor fi distribuiti spre toate cozile valabile.

Aplicatia ar trebui sa simuleze o serie de clienti care sosesc pentru servire, intrari in cozi, asteptari, serviri si in final plecarea clientilor din coada. Masoara timpul pe care clientii il petrec asteptand in coada si afiseaza spre iesire media timpului de asteptare. Pentru a calcula timpul de asteptare avem nevoie de timpul de sosire, timpul de finalizare si timpul de servire. Timpul de sosire si timpul de servire depind de client – cand apar acestia si cat timp de servire au nevoie in functie de cate produse au. Timpul de finalizare depinde de numarul de cozi, de numarul de clienti in coada si de timpul fiecaruia de servire.

**Cerinta principala** a temei este: “Proiectati si implementati o aplicatie de simulare care sa anaizeze sistemele bazate pe cozi pentru a determina si minimiza timpul de asteptare al clientilor”, astfel obiectivul principal al temei este acela de a crea o interfata care sa primeasca niste setari ca input si sa ofere o simulare virtuala a cozilor la un magazin spre exemplu, sa calculeze timpul de asteptare mediu, ora de varf, timpul de servire mediu,si sa incerce minimizarea timpului in care fiecare client asteapta sa fie servit.

**Date de intrare:**

**-**Minimum si maximum intervalului timpului de sosire intre clienti;

**-**Minimum si maximul timpului de servire/client;

**-**Numarul de cozi;

**-**Intervalul de simulare;

**-**Alte informatii pe care le consideram necesare: am ales numarul de clienti;

**Date de iesire:**

**-** Media timpului de asteptare, de servire si timpul in care cozile sunt goale pentru intervalul de simulare;

**-**Zona in care se inregistreaza evenimentele si principalele date de sistem;

**-**Evolutia cozilor;

**-**Ora de varf pentru intervalul de simulare;

**Obiective secundare:**

* Tratarea cazurilor de utilizare, prin diagrame use-case, aici prezentandu-se scenarii de utilizare a viitoarei aplicatii(capitol 2);
* Implementarea unei interfete secundare pentru setarile aplicatiei(care preia inputul de la utilizator) si il transmite sistemului de gestionare(capitol 4);
* Principala interfata sa simuleze grafic aparitia clientilor, intrarea acestora in cozi si parasirea cozilor dupa timul de servire(capitol 4);
* Crearea cozilor propriu-zise formate din thread-uri, astfel aplicatia va rula in paralel, simuland cat mai fidel realitatea;
* Crearea clientilor si asezarea acestora in cozi sub forma de ArrayBlockingQueue pentru a profita de partea de sincronizare automata a proceselor creata in Java pentru BlockingQueue;
* Prezentare concluzii, posibilitati de dezvoltare ulterioara si cunostintele deprinse in urma proiectului(capitol 6).

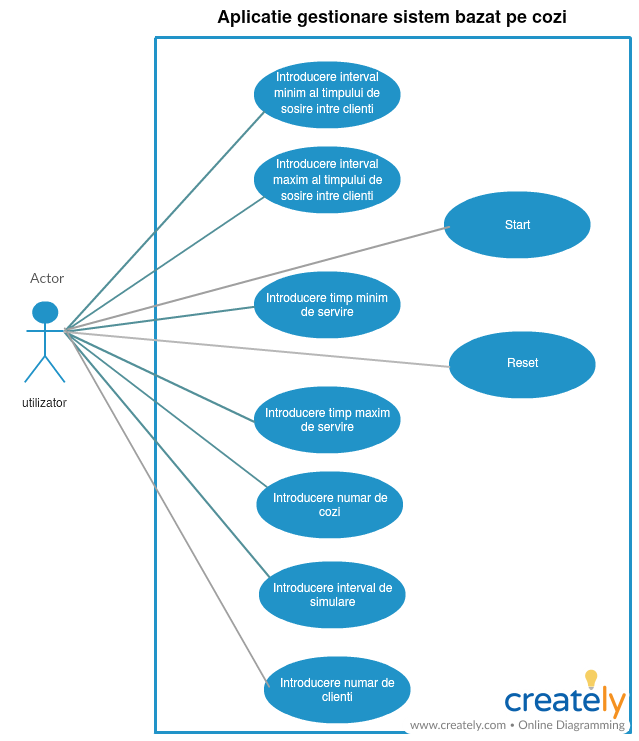
**2.Analiza problemei, modelare, scenarii, cazuri de utilizare**

Analiza preliminara a problemei ne imparte sarcina in crearea in primul rand a unei clase “Client” pentru manipularea acesteia in program, crearea unui sistem de gestionare capabil sa imparta clientii la cozi, crearea cozilor propriu zise( foarte important- sub forma de Thread-uri), si a unei interfete care sa simuleze aplicatia, pe langa o interfata de setare a acesteia.

Fiind o aplicatie care simuleaza o problema din lumea reala, avem nevoie de un timp al simularii, astfel, “timpul” sistemului de gestionare care imparte clientii la cozii si “timpul” asociat cozilor sunt aceleasi si le vom referi mai tarziu in program drept “currTime”.

Tot din procesul de modelare al problemei face parte si identificarea colectiilor de care avem nevoie: se poate observa ca in sistemul nostru de gestionare principal, avem nevoie de un ArrayList de clienti, pentru a simula oarecum interiorul magazinului(toti clientii care sunt inauntru in acel moment), avem nevoie si de un ArrayList de cozi, deoarece numarul de cozi ne este dat de catre utilizator si astfel acestea sunt mult mai maleabile. Iar in interiorul fiecarei cozi, avem nevoie de o colectie de clientii, dar de aceasta data vom folosi ArrayBlockingQueue deoarece aceasta structura consta intr-o coada care aditional suporta operatii care asteapta coada sa devina “non-empty” cand se extrage un element si asteapta spatiu sa devina “available” in coada cand se stocheaza un element, astfel se face oarecum partea de sincronizare in spate.

In continuare voi prezenta scenarii si cazuri de utilizare sub forma de diagrame use-case:



* **Use case: introducere cele 7 setari**
* **Primary actor:utilizatorul**
* **Main succes scenario:**

1. Utilizatorul porneste aplicatie
2. Utilizatorul introduce cele 7 cerinte ale ferestrei de Settings
3. Utilizatorul apasa butonul start
4. Aplicatia verifica daca inputul este unul corect
5. In cealalta fereastra deschisa se simuleaza corect plasarea clientilor la cozile cele mai putin costisitoare ca timp
6. Utilizatorul poate cere resetarea aplicatiei dupa incheierea simularii
7. Utilizatorul preia datele din “Log of events” si din rezultatele de simulare afisate in partea de sus a ferestrei

* **Use case: 1 din cele 6 operatii**
* **Primary actor:utilizatorul**
* **Alternative Sequences:**

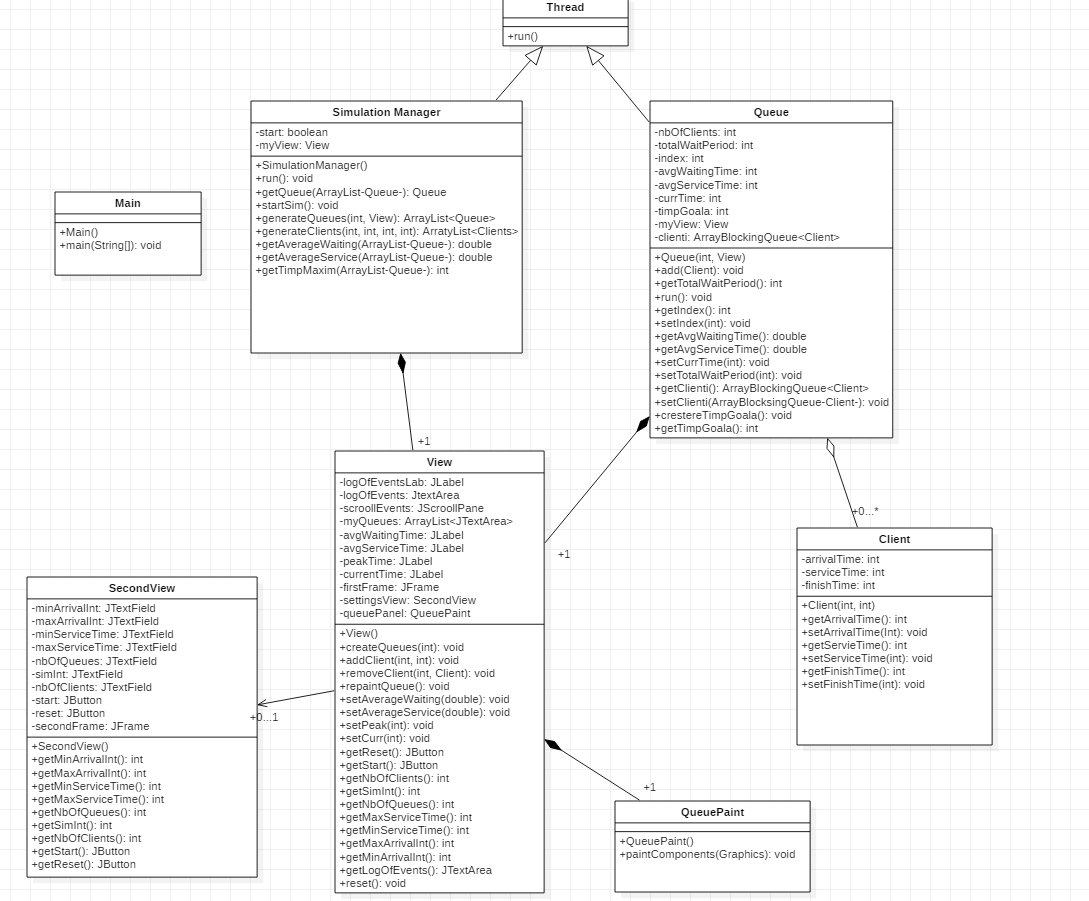
1. Datele introduse nu respecta o logica minima

* Intervalul minim al timpului de sosire dintre clienti este mai mare decat intervalul maxim
* Timpul minim de servire este mai mare decat timpul maxim

1. Datele introduse sunt invalide

* Utilizatorul ofera ca input altceva decat date numerice

**3.Proiectare**



Clase folosite:

1. Clasa **Main**: - principala clasa a programului, cu ajutorul ei se instantiaza un obiect de tipul Simulation manager si se porneste cu metoda start() , pentru a trece Thread-ul in modul Running;
2. Clasa **SimulationManager:-** aceasta clasa am implementat-o ca avand un dublu rol, atat acela de sistem de gestionare propriu zis cat si acela de “Scheduler” creand atat cozile cat si clientii, si apicand strategia pentru minimizarea timpului de asteptare. Aceasta strategie este in functie de timpul minim de asteptare. Pe langa acest rol, are functia de a porni Thread-urile paralele, adica cozile de procesare;
3. Clasa **Queue:-** clasa cu rol de coada(server de procesare), aceasta contine o colectie de clienti pe care ii proceseaza in ordinea sosirii pe principiul “primul venit primul servit”, cand clientul a fost servit acesta se sterge din coada respectiva, facand loc altui client. Tot la nivel de clasa se mentine si o variabila in care se updateaza mereu timpul total de asteptare pentru acea coada. Pe acest timp se bazeaza impartirea clientilor la cozi din Scheduler. O alta variabila este folosita la contorizarea timpului in care coada respectiva a fost goala de-a lungul simularii noastre;
4. Clasa **Client:-** in aceasta clasa se mentin ca variabile instanta timpul de sosire, timpul de servire si timpul de finalizare pentru fiecare client, toate acestea fiind utile in calcularea timpilor medii si perioadei de varf;
5. Clasa **View:-**in cadrul acestei clase se construieste interfata grafica a aplicatiei, se ofera acces la butoane, un log al evenimentelor si o zona dinamica in care se construiesc cozile in functie de inputul primit de la utilizator;
6. Clasa **SecondView:-** este o clasa auxiliara pentru clasa View, pentru a nu supraincarca interfata grafica principala, se construieste aici o fereastra separata pentru a introduce setarile aplicatiei(timpii minimi si maximi, numarul de cozi, numarul de clienti, timpul de simulare) , astfel aceasta clasa se afla in asociere directa cu clasa View;
7. Clasa **QueuePaint:-**aceasta clasa extinde un JPanel si este folosita pentru a superscrie metoda paintComponents(), lucru util cand se foloseste metoda repaint() pentru a construi dinamic lista de cozi in interfata grafical
8. Clasa **Thread:-** este extinsa de catre 2 clase pentru a putea folosi o procesare in paralel a clientilor, si pentru a putea simula un timp real prin folosirea metodei sleep().

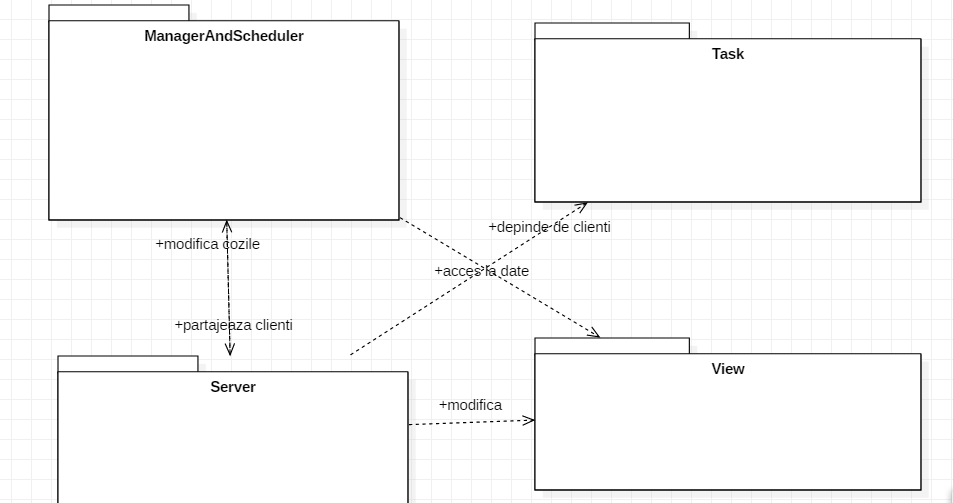
Ca alte tipuri de date folosite, se folosesc colectiile in SimulationManager pentru a genera colectia noastra de cozi si colectia de clienti existenti, tot din cadrul colectiillor, pentru diferitele cozi se foloseste ArrayBlockingQueue care este o coada dar cu unele metode sincronizate, dat fiind faptul ca folosim Thread-uri pentru procesare paralela, acest lucru este util.

Variabilele in clasele care extind Thread sunt de tipul **“volatile”** lucru care marcheaza faptul ca sunt stocate in **memoria principala.**  Cu alte cuvinte, orice citire a unei variabile volatile va fi efectuata din memoria principala a computerului si nu din cache-ul CPU, iar fiecare scriere a unei variabile volatile se va face in memoria principala, nu in CPU cache.

Acest **“volatile”** garanteaza vizibilitatea schimbarilor variabilelor de-a lungul mai multor Threaduri. Intr-o aplicatie multithreading unde Thread-urile opereaza pe variabile non-volatile, fiecare Thread poate copia variabilele din memoria principala in CPU cache cand le foloseste, pentru motive de performanta. Asta inseamna ca, fiecare Thread poate copia variabilele in CPU cache al unui procesor diferit. Cu variabile non-volatile, nu este garantat cand JVM-ul citeste date din memoria principala in CPU caches, sau cand scrie date din CPU caches in memoria principala. Aceasta poate cauza unele probleme cum ar fi desincronizarea operatiilor pe o variabila, fapt care duce la erori majore.

Folosind variabile **“volatile”** operatiile se fac in memoria principala, astfel fiecare Thread vede valoarea curenta a variabilelor, fara a exista dubii sau probleme.

In continuare se va prezenta diagrama de pachete:



Pachetul ManagerAndScheduler depinde de view in sensul ca de aici isi construieste toate partile componente, in functie de inputul de la utilizator primit in View. De acest View depinde si Pachetul Server, deoarece acesta face modificari in View pe parcursul procesarii clientilor din coada. De Pachetul Task, care contine de fapt clientii, depinde pachetul Server, acesta fiind bazat pe o colectie de clienti. Intre Pachetele Server si ManagerAndScheduler exista o relatie de interdependenta prin faptul ca fac schimb de informatii, Schedulerul distribuie clientii la cozile create de el insusi iar Serverul modifica Lista proprie de clienti, fapt care duce la modificare cozii si la schimbarea campurilor prin pachetul care raspunde de crearea, popularea cozilor si crearea si distribuirea eficienta din punct de vedere temporar a clientilor.

**4.Implementare**

**Clasa Client**

Aceasta clasa contine 3 variabile principale de tip integer, cu modificator de acces private si tip volatile, variabile care stabilesc timpul de sosire la coada al clientului, timpul de servire necesar acestuia si timpul de finalizare. Toate aceste 3 informatii vor fi unterior folositoare pentru calcularea timpul de asteptare mediu, timpului de servire mediu si perioadei de varf. Din punct de vedere al metodelor folosite, se folosesc doar gettere si settere.

**Clasa Queue**

Clasa care extinde clasa Thread si suprascrie metoda run() pentru a putea simula procesarea paralela a clientilor pe mai multe cozi.

In aceasta clasa se retin mai multe informatii sub forma de intregi. Contine astfel numarul de clienti, timpul total de asteptare pentru coada curenta, un index- folositor in view pentru construirea indexata a cozilor, un timp mediu de asteptare si un timp mediu de servire pentru coada curenta. In plus mai contine un timp in care coada curenta este goala si, cel mai important, o colectie de clienti sub forma de ArrayBlockingQueue<Client>.

Ca metode folosite, amintesc metoda de adaugare a unui client in lista de clienti, moment in care se creste si timpul total de asteptare/coada cu timpul de servire al clientului respectiv. O alta metoda este cea de incrementare a timpului in care coada este goala, fapt verificat din alta clasa, 2 metode pentru calcularea timpului mediu de asteptare si de servire pentru coada curenta, in care se imparte timpul total de asteptare si de servire calculat, la numarul de clienti din coada.

Cea mai importanta metoda este metoda run(), aceasta metoda lucreaza mereu cu clientul din varful cozii noastre(cel care este primul la rand). Intr-o bucla infinita se verifica daca timpul de servire al clientului curent a ajuns la 0, in caz afirmativ se elibereaza locul clientului atat in coada propriu zisa cat si in coada din interfata. Se trece la urmatorul client, daca acesta exista, aici se creste timpul total de asteptare/ coada cu timpul de asteptare total pentru fiecare client in parte, si se creste la fel si timpul de servire total/coada. In finalul metodei se scade timpul de servire al clientului curent cu o secunda(pentru ca pe atata timp e gandit sa ruleze Threadul, si se scade si perioada totala de asteptare in momentul prezent a cozii. Se pune Thread-ul pe sleep pentru o secunda si se revine in while.

Metoda run este prezentata in continuare:

public void run() {

// **TODO** Auto-generated method stub

Client currClient=clienti.peek();

int distributed=1;

while(true)

{

currClient=clienti.peek();

if(currClient != null)

{

if(currClient.getServiceTime() == 0)

{

clienti.poll();

myView.removeClient(this.index,currClient);

currClient=clienti.peek();

distributed=1;

}

if(currClient != null)

{

if(distributed == 1)

{

avgWaitingTime+= currClient.getFinishTime()-currClient.getArrivalTime() ;

avgServiceTime+= currClient.getServiceTime();

nbOfClients++; //clienti serviti prin acea coada

distributed=0;

}

currClient.setServiceTime(currClient.getServiceTime() -1);

totalWaitPeriod--;

}

}

try {

currTime++;

*sleep*(1000);

}

catch(InterruptedException e)

{

e.printStackTrace();

}

}

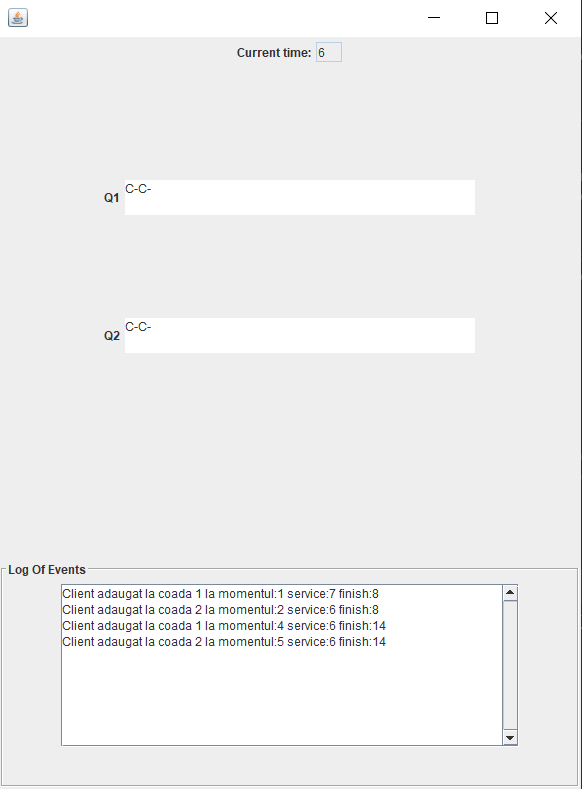
}

D

**Clasa View**

Aceasta clasa construieste practic interfata grafica in care utilizatorul poate vedea o simulare in timi real a evolutiei cozilor si o evolutie gradata a timpilor medii si a perioadei de varf in functie de numarul de clienti prezenti, cat si un spatiu unde se salveaza parcursul evenimentelor de la cozi, si timpul in care cozile au ramas goale pe parcursul simularii.

Din punct de vedere al componentelor folosite aici se identifica in mod special un JTextArea pentru casuta de evenimente, acestuia ii este asociat un JScrollPane deoarece in cele mai multe cazuri informatia depaseste spatiul pus la dispozitie. Pe laga acestea mai avem un panou special de tip QueuePaint(care este o alta clasa care extinde de fapt un JPanel)- aici se vor construi in mod dinamic cozile noastre. Alte componente sunt, bineinteles, JLabel-urile pentru etichetarea informatiei prezente in output. Ca urmare a aranjarii acestora prin folosirea majoritara a unor Layout-uri de tip BoxLayout am obtinut interfata grafica din figura urmatoare:



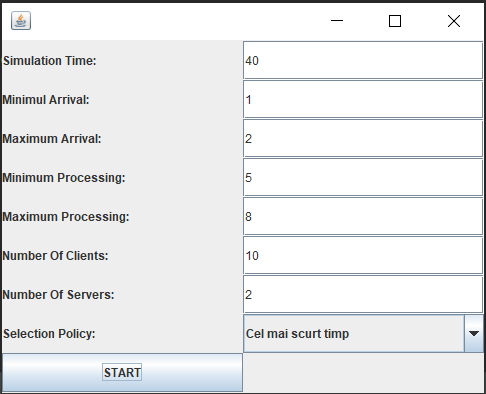
Ca metode importante din aceasta clasa, avem metodele de adaugare si de stergere a unui client care practic lucreaza cu sirul de caractere afisat in dreptul fiecarei cozi cu mentiunea ca se afiseaza doar ultima cifra a numarului (C12- devine C2) dar acest fapt este vizibil in simulare datorita log-ului de evenimente.

O metoda importanta este cea care creaza cozile in mod dinamic in functie de numarul de cozi dat de utilizator. Aceasta metoda practic initializeaza de fiecaer data un JLabel, un JTextArea si un JPanel pe care le adauga treptat la un alt Panel de tipul Clasei QueuePaint cu ajutorul careia se apeleaza apoi repaint() pentru a afisa cozile.

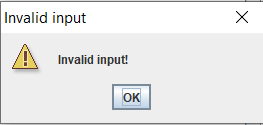
**Clasa SecondView**

Aceasta clasa este oarecum auxiliara la clasa anterioara, care are de altfel si o variabila instanta de timpul clasei curente. Aici sunt introduse urmatoarele: minimul intervalului timpului de sosire dintre clienti si maximul, minimul timpului de servire/client si maximul, numarul de cozi, numarul de clienti si intervalul de simulare.Tot aici utilizatorul poate alege intre a apasa butonul start sau reset, pentru a reseta aplicatia.

Interfata pentru setarile aplicatiei arata in felul urmator:



Tot in cadrul acestei interfete, prin metode de tip gettere se preiau toti intregii din inputul utilizatorului, si se verifica daca inputul este unul valid. In cazul in care nu este valid( nu este compus doar din caractere numerice), ci contine si alt tip de caractere, metodele vor verifica fiecare input in parte si vor afisa in acest caz un mesaj de tip WARNING\_MESSAGE care arata in felul urmator:



**Clasa SimulationManager**

Cea mai importanta clasa din acest pachet contine ca variabile instanta o valoare booleana si o instanta a clasei View si extinde clasa predefinita in Java-Thread pentru a putea simula un timp real, folosind metoda sleep().

Ca metode regasite aici, amintesc de metodele pentru calcularea timpui mediu de asteptare si de servire pe fiecare coada, astfel se imparte media timpilor pe cozi la numarul de cozi: asfel calculam timpul mediu de asteptare la nivel de cozi, care datorita minimizarii create de noi este aproximativ egal cu timpul mediu de asteptare pentru clienti.

Tot in cadrul acestei clase se afla si 2 metode de generare clienti, respectiv cozi, care tin cont de inputul de la utilizator. Voi prezenta cele 2 metode in sectiunea care urmeaza:

public static ArrayList<Queue> generateQueues(int nbOfQueues,View myView)

{

ArrayList<Queue> queues=new ArrayList<Queue>(nbOfQueues);

Queue q;

for(int i=0;i<nbOfQueues;i++)

{

q=new Queue(i,myView);

queues.add(q);

queues.get(i).start();

}

return queues;

}

public ArrayList<Client> generateClients(int nbOfClients,int minArrivalInt, int maxArrivalInt, int minServiceTime,int maxServiceTime)

{

ArrayList<Client> clients=new ArrayList<Client>(nbOfClients);

Random rand=new Random();

int serviceT,arrivalT;

int prev\_arrivalT=0;

int currClients=0;

while(currClients < nbOfClients)

{

serviceT=rand.nextInt(maxServiceTime-minServiceTime+1)+minServiceTime;

arrivalT=rand.nextInt(maxArrivalInt-minArrivalInt+1)+minArrivalInt+prev\_arrivalT;

prev\_arrivalT=arrivalT;

clients.add(new Client(arrivalT,serviceT));

currClients++;

}

return clients;

}

Se observa ca in metoda de generare a cozilor, acestea le leaga si de interfata, si se pornesc toate serverele de procesare(cozile), acestea extinzand clasa Thread, pentru a lucra in paralel.

Cealalta metoda, de generare a clientilor, se foloseste de Clasa Random predefinita in Java pentru a genera niste clienti aleatori, folosindu-se de inputul utilizatorului si de setarile acestuia pentru client.

Metoda urmatoare, are rolul de Scheduler, aceasta stabilind care este coada cu timpul minim de asteptare unde se va pune urmatorul client:

public static Queue getQueue(ArrayList<Queue> queues)

{

Queue queue=null;

int minWaitingTime=Integer.***MAX\_VALUE***;

for(Queue q: queues)

{

if(minWaitingTime> q.getTotalWaitPeriod())

{

minWaitingTime=q.getTotalWaitPeriod();

queue=q;

}

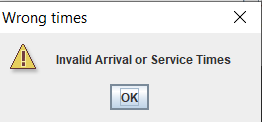
}

return queue;//coada cu cel mai mic total wait period

}

Se observa faptul ca dupa o iteratie prin cozi, se returneaza coada cu timpul minim de asteptare.

Tot in cadrul acestei clase, se verifica si inputul utilizatorului din punct de vedere logic: daca acesta ofera un timp mimim mai mare decat cel maxim sau invers, aplicatia va detecta acest lucru si va oferi un mesaj de tip WARNING\_MESSAGE de tipul:



Voi prezenta in continuare din metoda startSim() care preia inputul si porneste efectiv simularea, doar bucla care simuleaza exact lucrul unor cozi de procesare clienti:

while(currTime < simInt)

{

myView.setCurr(currTime+1);

currentWaitingT=*getTimpMaxim*(queues);

if(timpMax < currentWaitingT)

{

timpMax=currentWaitingT;

peakTime=currTime;

myView.setPeak(peakTime);

}

for(Client c: random)

{

if(c.getArrivalTime()==currTime)

{

myQ=*getQueue*(queues);

c.setFinishTime(c.getArrivalTime()+c.getServiceTime()+myQ.getTotalWaitPeriod());

myQ.add(c);

myView.addClient(myQ.getIndex(),clientCurent);

myView.getLogOfEvents().append("Momentul:"+currTime+"\n"+"Clientul "+ clientCurent++ + " adaugat la coada "+(myQ.getIndex()+1)+ " Service time: "+c.getServiceTime()+"arr:"+c.getArrivalTime()+"\n");

}

}

currTime++;

avgWaitingTime=*getAverageWaiting*(queues);

avgServiceTime=*getAverageService*(queues);

myView.setAverageService(avgServiceTime);

myView.setAverageWaiting(avgWaitingTime);

for(int i=0;i<nbOfQueues;i++)

{

if(queues.get(i).getClienti().isEmpty())

queues.get(i).crestereTimpGoala();

}

try {

*sleep*(1000);

}

catch(InterruptedException e)

{

e.printStackTrace();

}

}

for(int i=0;i<nbOfQueues;i++)

myView.getLogOfEvents().append("Coada "+(i+1)+" a stat goala pentru un timp="+queues.get(i).getTimpGoala()+"\n");

Se observa faptul ca in View tot timpul se updateaza timpul curent al simularii, apoi se itereaza prin timpii maximi de asteptare pentru fiecare coada si pe baza acestora , cand este timpul cel mai mare se updateaza si perioada de varf din magazin, adica momentul cand se astepta cel mai mult de catre un client pentru a fi procesat.

Urmeaza iteratia prin clienti, astfel daca avem un client care are timpul de sosire egal cu cel al simularii curente, il plasam in coada cu timpul de asteptare minim. Se updateaza si timpii medii dupa care se asteapta o secunda pentru a trece la noua iteratie prin bucla. La final, se seteaza timpii in care cozile au fost goale, fara ca serverele(cozile) sa proceseze ceva.

**5.Rezultate**

Pe baza simularii efectuate, la finalul acesteia, in partea de sus a interfetei grafice se pot obtine informatii cu privire la perioada de varf din simulare, timpul mediu de asteptare la fiecare coada de catre clienti, timpul mediu de servire la fiecare coada, iar in logul evenimentelor se pot obtine informatii cu privire la circultia clientilor prin cozi(cand a sosit un client, ce timp de servire are, la ce coada a fost pus, cand a fost terminat de servit un client, la ce moment a parasit coada).

In finalul simularii se pot obtine informatii cu privire la cat timp au stat goale cozile, fara a avea clienti de procesat. Pentru un manager, asta ar ajuta mult la gestionarea angajatilor in functie de orele de varf( mai multi angajati la ore de varf, mai putini angajati cand cozile stau goale impreuna un timp suficient de mare pentru a inchide o casa fara a afecta timpul de asteptare al clientilor).

Simularea efectuata ofera ca rezultat si o imitare vizuala a clientilor care vin si se plaseaza la cozi, fiind procesati intr-un timp anume si apoi parasind coada.

**6.Concluzii**

Ca o concluzie la acest proiect, se observa importanta incontestabila a gestionarii cat mai eficiente a resurselor puse la dispozitie(umane si materiale). O astfel de aplicatie pusa in practica intr-un supermarket ar ajuta mult la mimizarea timpului de asteptare la cozi in primul rand, fiind astfel utila pentru clienti, castigand timp valoros, pentru ca le-ar oferi informatii cu privire la coada unde trebuie sa se plaseze. O alta utilitate s-ar gasi in gestionarea angajatilor si in urmarirea activitatii acestora de catre manager, acesta ar stii cat timp sta o coada goala, cand este cel mai aglomerat, si astfel ar putea lua decizii in functie de acest lucru(deschiderea, inchiderea unei case, angajari sau eliberari de personal,etc).

Cel mai important aspect care m-a ajutat la dezvoltarea cunostintelor de programare este lucrul in paralel cu Threaduri, faptul ca aplicatia este altfel mult mai rapida, si fiecare serever lucreaza de unul singur, nedepinzand direct de celalat, cum era cazul daca aplicatia mea era non-multithreading.

Din punct de vedere al dezvoltarii ulterioare, s-ar putea implementa o metoda prin care simulatorul sa stie singur cand este nevoie sa mai deschida o casa sau sa inchida o casa(un server de procesare pentru clienti) in functie de timpul de asteptare al clientilor existenti. Tot pe aceasta tema a dezvoltarii, s-ar putea o statistica mult mai laborioasa, cum ar fi timpul petrecut in magazin sau gestionarea cantitatii de produse(timpului de servire) in functie de perioada din an respectiva- pentru a sti cand e nevoie de o suplimentare sau o scadere a stocurilor.

**7.Bibliografie**

[1] <https://www.tutorialspoint.com/java/java_multithreading.htm>

[2] <http://users.utcluj.ro/~crisb_pop/pt2018.html>

[3] <https://www.geeksforgeeks.org/multithreading-in-java/>

[4] <http://tutorials.jenkov.com/java-concurrency/volatile.html>