**FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE**

**DEPARTAMENTUL CALCULATOARE**

**Tema 1**

la disciplina

**Tehnici de programare**

**Titlu**

***„Simulator de cozi”***

Nume si prenume : Trif Gheorghe Andrei

Grupa: 30225

An academic : 2018-2019

**1.Obiectivul temei**

Propuneti, proiectati si implementati un sistem de simulare a cozilor unui magazin care are ca scop reducerea timpului de asteptare a clientilor la coada.

|  |  |  |
| --- | --- | --- |
| Obiective secundare | Descriere | Capitolul |
| Dezvoltare de use case-urii si scenarii in legatura cu intelegera temei propuse. | Se va dezvolta un comportament pe care programul va incerca sa le urmeze. | 1 |
| Analiza problemei si modelarea acesteia | Se va prezenta proiectarea OOP a problemei ,diagrame UML ,pachete si structuri de date folosite. | 2 |
| Dezvoltarea algoritmilor | Se va perzenta operatile alese  si descrierea lor. | 3 |
| Impelmentarea solutiei | Se va prezenta codul sursa aferent proiectari. | 3 |
| Testare aplicatiei | Prezentarea ca de cazuri favorabile de rulare si modul de utilizare ,dar si anumite defecte. | 4 |
| Dezvoltare ulterioara | Noi ideii de a inbunatati proiectul existent. | 5 |

Scopul acestui proiect este de a implementa un simulator de gestiune al cozilor si de a descrie modul in care acestea pot fii create si manipulate cu ajutorul paradigmelor OOP.

Cozile sunt frecvent folosite pentru a modela obiecte din lumea reala care se bazeaza pe principiul fifo (first in first out) adica primul venit primul servit .Obiectivul principal al unei cozi in viata reala este de a oferii un loc pentru un "client" care asteapta anumite serviici si produse si care nu poate sari in fata altor clienti care au ajuns innaintea lui la coada.Acest tip de sisteme bazate pe cozi urmaresc sa eficientizeze timpul de asteptare astfel incat clienti sa primeasaca serviciile cat mai rapid.Astfel prima idee care ne sare in minte este adaugarea a cat mai multor cozi care sa micsoreze timpul de distribuire al serviciilor , acest lucru poate duce la cheltuieli suplimentare ale furnizorului si care poate fi putin inposibil in viata de zi cu zi.Astfel ideea pe care ne bazam nu este cea de alocare a cat mai multor cozi ci cea in care incercam sa alocam clienti in mod uniform cozilor disponiblie in acel moment de timp.Acest lucru va spori considerabil modul de livrare al serviciilor.

2.Analiza problemei si modelarea acestei

2.1.Analiza problemei

În programare o coada este o structura de date logice (implementare se face cu alte structuri de date) si omogene (toate elementele sunt ce acelasi tip).Aceasta prezinta doua operatii de baza: adaugarea unui lement si extragerea unui element . Singurul lucru care difera fata de alte structuri este principiul sau de feunctionare fifo.

Cozile ca structuri de date port fi implementate prin liste dar si vectori in functie de cerintele porblemei.

Cateva proprietati ale cozilor

1. Adaugarea unui element in coada se face tot timpul la final;
2. Stergera unui element se face tot timpul de la inceput;

Prin analiza problemei, ne referim la un prim set abstract de operații și proprietăți prin care încercăm să depistăm eventualele însușiri și comportamente ale proceselor necunoscute. Programarea orientată ne oferă aici un avantaj clar, tocmai fiindcă ea permite să taclăm problema de la un nivel superiror, fără a mai fi constrâși, într-o așa măsură, de caracteristicile tehnice.

Această strategie de conceptualizare, mai poartă numele și de bottom-up design. Este foarte avantajoasă din prisma găsirii componentelor constituente, deoarece pot fi găsite, relativ ușor, structuri cu o legătură directă în lumea reală( obiecte, acțiuni etc.). Din păcate această versatilitate vine cu prețul complexității, ea crescând pe măsură ce se avansează pe nivelele inferioare.Bottom -up poate fi foarte utila pentru crearea unor solutii rapide dar nu foarte complexe fiind utilizata cel mai adesea de un singur programator care lucreaza independent .

De cele maimulte ori se porneste de la specificarea proiectului cautandu-se verbe sau substantive relevante pentru crearea de clase si pentru crearea de metode.Astfel odata ce am realizat pasul de mai sus vom avea o idee foarte clara despre comportamentul aplicatiei pe care urmeaza sa o creeam si care sunt funtiile sale de baza.

O astfel de aplicatie a fii folosita defoarte multe persoane sau poate chiar de o companie astfel un prim pas in creearea sa e sa determinam intrarile pe care un anumit utilizator ar dori sa le poata accesa.Acest lucru se face cel mai probabil prin cadrul interfetei dintre program si utilizator.Aceasta interfata trebuie sa fie foarte usor de folosit dar si foarte precisa astfel incat sa nu induca in eroare utilizatorii.Deoarece in zilele noastre mai exista persoane care nu sunt invatate cu astfel de lucruri si care ,la prima vedere le poate crea o stare de confuzie.

In cadrul Simulatorului de cozi se specifica foarte clar ca principala sa caracteristica o sa fie eficientizarea timpului de servire pe care un client il are in momentul in care acesta va ajunge intr-o coada.Astfel eu in proiectare am ales sa folosesc mai multe zone de tip text in care un utilizator poate introduce anumite valori necesare unei simulari corecte , cum ar fii : un timp minim si maxim de sosire dar si un timp min si maxim de servire , precum si perioada de simulare.

|  |
| --- |
| Planificare |
| i : int  client : Client  e : ScheduledExecutorService  Coada1 : Simulare  Coada2 : Simulare  Coada3 : Simulare  generator : Generator |
| C:  Planificare();  M:  Min(int,int,int) : int |

2.2.Proiectarea claselor

|  |
| --- |
| Client |
| timpSosire : int  timpServire : int  id : int |
| C:  Client(int ,int,int);  M:  getServireTime() : int  setServireTime (int) : void  setSosireTime (int) : void  getSosreTime () : int  setID(int) : void  getID() : int; |

|  |
| --- |
| window |
| minatime:TextField  maxatime:TextField  minstime:TextField  maxstime:TextField  timptf:TextField  text:TextArea  start:JButton  clear:JButton  simulator :Runnable  sim:ScheduledExecutorService |
| C:  window()  M:  Show(String):void  getMinServire(): int  getMaxServire(): int  getMinSosire(): int  getMaxSosire(): int  getTimp() : int |

|  |
| --- |
| Generator |
| timpSosire : int  timpServire : int  r : Random  nrClientiGenerati : int  i : int  generator : Runnable  make : ScheduledExecutorService |
| C:  Generator()  M:  getNrClienti() : int  getClienti() : Client |

|  |
| --- |
| window |
| Coadaclient:BlockigQueue<Client>  client :Client  T :Thread  Id:int  finTime :long  currTime :long  initTime :long  nrclienti :int  avAsteptare: long  avServire: long  block:Runnable  e: : ScheduledExecutorService |
| C:  Simulare(int)  M:  run : void  addClient(Client):void  getAvgAsteptare (): long  getAvgServire (): long  getCapacity() : int  isReady(): int |

Simulatorul de cozi este alcatuit din 5 clase: Client , Generator , Simulare ,Planificare si window care este clasa interfata.

In clasa window este creata interfata dintre utilizator si programul care face simularea.Aceasta contine butoanele precum si campurile de input unde vor fi introduse datele despre simulare.In aceasta clasa se afla si butonul de start care are rol de pornire a simulari dar sibutonul de curatare care dupa o simulare are rolul de a reseta campul unde sunt afisate rezultatele.

Clasa Client este clasa de baza a aplicatiei ea defineste obiectul real care este supus diferitelor operatii care au loc in interiorul aplicatiei.Dupa o ampla sinteza am stabilit ca pentru un ca un client sa fie definit acesta trebuie sa aiba urmatoarele atribute :id pentru a putea stii ce client sa create si care dinte clienti suporta actiunea , un alt atribut este timpul de servire care ne spune cat timp ii ia unui cient din momentul cand ajunge primul la coada pana cand este servit si timpul de sosire este un atribut necesar deoarece astfel putem calcula media de asteptare a unui client in coada respectiva.Ca si metode aceasta clasa are doar metodele simple de set si get.

Clasa Generator este clasa care genereaza un nou client cu un timp de servire sau mai mare sau egal ca minimul introdus de la interfata.Un client este creat cu ajutorul unui executor de servicii care ruleza intr-un thread generatorul cale implementeaza interfata Runnable;

Clasa Planificare in aceasta clasa este implementata ideea de eficientizare a timpului de asteptare intr-o coda astfel in aceasta clasa adaugam clienti in coada in functie de coada care are cel mai mic nr de clienti;

Clasa Simulare aici are loc simularea propriu zisa a operatiilor de adaugare si de stergere a unui client cu ajutorul unui thread putem simula timpul de servire al cilientului prin adormirea threadului respectiv si prin oprirea sa odata ce operatia a fost terminata.Tot aici sunt calculate cele 2 medii de timp care sunt afisate utilizatorului.

3.Impementarea Claselor

3.1.Clasa Client

Aceasta clasa nu este o clasa complexa in interior este definirea simplificata a unui cilient care ajunge la coada la un anumit timp si asteapta o anumita perioada sa fie servit.

**public** Client(**int** x, **int** aTime, **int** sTime) {

**this**.id = x;

**this**.timpServire = sTime;

**this**.timpSosire = aTime;

}

3.2.Clasa Generator

Aceasta clasa reprezinta generatorul efectiv de clienti care genereaza un client cu un anumit timp de sosire si un anume timp de servire.Astfel aici cu ajutorul executorului de servicii rulam inrt-un thread un task de tip Runnable.

ScheduledExecutorService make = Executors.*newScheduledThreadPool*(1);

**private** Runnable generator=**new** Runnable() {

@Override

**public** **void** run() {

timpSosire =window.*getMinSosire*() + r.nextInt(window.*getMaxSosire*() - window.*getMinSosire*());

timpServire = window.*getMinServire*() + r.nextInt(window.*getMaxServire*() - window.*getMinServire*());

i++;

Client client=**new** Client(i,timpSosire,timpServire);

clienti.add(client);

nrClientiGener++;

}

Ca sa declar lista clienti am folosit libraria java.util.ArrayList care contine diverse metode pentru lucrul cu liste.

Ca metode am mai declarat opetoda care returneaza clientul cu un anumit index din lista si o metoda care returneaza un numar de clienti.

3.3.Clasa Simulare

In aceasta clasa este implementata simulare unei cozi in care daca coada respectiva nu este goala ii stergem capul si asteptam un anumit timp "timp de servire" dupa care reluam procesul pana cand timpul curent este egal cu timpul de simulare.Tot aici calculam si media timpului se asteptare si servire ca fiind suma tuturor timpilor de asteptare sau servire supra nr de clienti;

**while** (currTime <= finTime) {

**if** (coadaclienti.isEmpty() == **false**) {

**try** {

client = coadaclienti.take();

T= **new** Thread(block)

initTime = initTime + client.getSosireTime();

nrclienti++;

T.start();

T.join();

avServire =avServire + client.getServireTime();

avAsteptare = avAsteptare + (System.*currentTimeMillis*() - initTime);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

Tot in aceasta clasa sunt adaugate date despre fiecare coada intr-un fisier care are rol de logger.

3.4.Clasa Planificare

In aceasta clasa am creat un sexecutor cu 3 thread-uri care executa fiecare coada si face operatia de simulare prezentata in clasa anterioara.Tot in aceasta clasa adaugam clienti generati de generator in coada.Un client se va adauga in totdeauna in coada cu cei mai putini clienti deoarece mergem pe principiul ca in acea coada va fii servit mai repede .Astfel distribuim fiecare client uniform pe toate cele 3 cozidisponibile.

**while** (ok== 0) {

client = generator.getClient(i);

**int** min = Min(Coada1.getClient(), Coada2.getClient(), Coada3.getClient());

**if** (Coada1.getClient() == min) {

**if** (Coada1.getCapacity() != 0) {

Coada1.addClient(client);

i++;

}

} **else** {

**if** (Coada2.getClient() == min) {

**if** (Coada2.getCapacity() != 0) {

Coada2.addClient(client);

i++;

}

} **else** {

**if** (Coada3.getClient() == min) {

**if** (Coada3.getCapacity() != 0) {

Coada3.addClient(client);

i++;

}

}

}

}

3.4.Clasa Window

In aceasta clasa este creata interfata grafica a aplicatiei prin apelare interfetei JFrame in care adaugam butoanele si casetele text.Sunt existente cateva metode care au rolul de a lua textul din JTextFild si de al transmite in program.Tot aici sunt instantiati si ascultatori butoanelor de start si clear prin folosirea librariei ActionEvent

Tot cu ajutorul unui Executor de servitii pornim simularea propriu zisa

ScheduledExecutorService sim = Executors.*newScheduledThreadPool*(1);

**private** Runnable simulator= **new** Runnable(){

**public** **void** run(){

**new** Planificare();}

};

start.addActionListener(**new** ActionListener() {

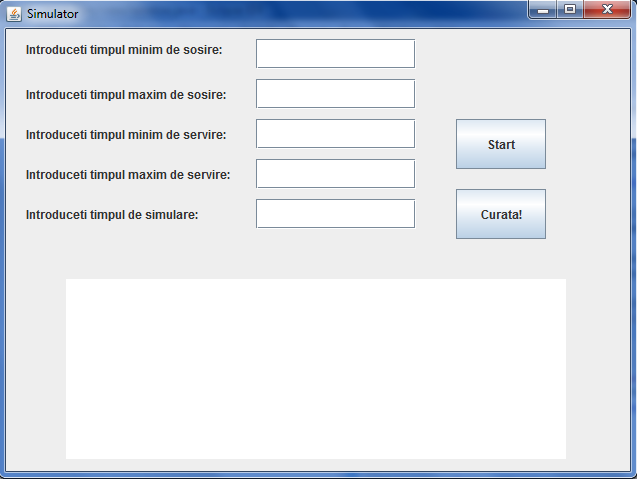
**public** **void** actionPerformed(ActionEvent e) {

sim.execute(simulator);

}

});

4.Testarea aplicatiei



Testarea acestei aplicatii este foarte simpla si usor de facut.Dupa ce sau completat toate campurile text se apasa pe start si se asteapta aparitia rezultatului simulari dupa ce rezultatele au aparut pentru o noua simulare se apasa pe butonul curata care va curata zona de output se reintroduc datele noii simulari si se apasa pe start din nou.

5.Dezvoltarea ulterioara

Din aceasta tema mi-am dat seama de numaratele moduri de a lucra cu readuri in java.Dar si cat de complexa poate fii o problema care la inceput pare usor de implementat.O prima dezvoltare ar fii extinderea domeniului de masurare si adaugarea de noi campuri cu noi caracteristici si nevoi ale unui client.

O alta ulterioara dezvoltare a acestei aplicatii de simulare a putea fii crearea unei magazin mai complex si cu comenzi si diferite tipuri de produse dar si clienti cu diferite pretentii si asteptari care ar putea renunta la asteptare in momentul in timpul de asteptare ar fii foarte mare.

Bibliografie

[1]. httpshttps://www.mkyong.com/java/java-scheduledexecutorservice-examples/

[2].https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledExecutorService.html

[3]. https://www.mkyong.com/tutorials/junit-tutorials

[4]. https://www.baeldung.com/java-blocking-queue

[5]. https://www.tutorialspoint.com/java/java\_multithreading.htm