**Documentație**

la disciplina

**Tehnici de programare**

**Tema 2 – Managementul unor cozi cu**

**Ajutorul thread-urilor**

**Popa Cristian Mihai, grupa 30224**

**An academic: 2021-2022**

Cuprins

[1. Obiectivul temei: 3](#_Toc99664753)

[2. Analiza problemei: 3](#_Toc99664754)

[3. Proiectare: 4](#_Toc99664755)

[4. Implementare: 5](#_Toc99664756)

[5. Concluzie: 9](#_Toc99664757)

[6. Bibliografie: 10](#_Toc99664758)

# Obiectivul temei:

* **Obiectiv principal:**

Obiectivul acestei teme consta in creare unei aplicații care sa simuleze o situație reala des întâlnita in ziua de astăzi. Aplicația simulează o coada de așteptare, asemănătoare cu ce am găsi in anumite instituții publice sau când suntem in mediul online, iar un server trebuie sa proceseze eficient mai multe cereri.

Aplicația trebuie sa primească input de la utilizator, sub forma de un string. Aplicația prezinta in total 7 câmpuri unde utilizatorul va introduce informația necesare pentru simulare.

* **Obiectiv secundare**:
  + Revizualizarea unor paradigme de programare orientata pe obiect: Interfețe, Thread-uri si cum funcționează etc.;
  + Dezvoltarea unor clase de obiect adecvate: crearea unor clase potrivite pentru o persoana(in cazul nostru, client) si coada, astfel încât informația sa fie ușor de sintetizat;
  + Dezvoltarea unor algoritmi eficienți si ușor de înțeles pentru prelucrarea in paralel a mai multor cereri;
  + Implementarea operațiilor si structuri de date specifice pentru Thread-uri, astfel încât programul sa funcționeze conform cerinței;
  + Implementarea unei interfețe grafice ușor de utilizat si ușor de înțeles;
  + Implementarea soluției pentru a avea un calculator funcțional de polinoame

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

# Analiza problemei:

In viața de zi cu zi întâlnim adeseori situații in care trebuie sa așteptam. Ca suntem la o coada la o benzinărie sau magazin, sau suntem in mediul online si așteptam răspunsul la o cerere lansata de noi(ex: sa ni se afișeze email-urile dintr-o anumita data). Problema de acum ne cere sa cream un sistem de management al unor cozi de așteptare. Aplicația va distribui fiecare client la cate cozi avem astfel încât timpul de așteptare sa fie cat mai mic. Utilizatorul va trebui sa introducă in 5 câmpuri informațiile necesare despre simulare:

* Număr de clienți;
* Număr de cozi disponibile;
* Timpul total al simulării(in secunde);
* Timpul de sosire(va fi un interval de forma a-b);
* Timpul de servire necesar unei persoane(de ex. persoana care sta in fata) care va fi tot un interval de genul [a-b]

Pentru a funcționa corect, programul presupune introducerea datelor sub o anumita forma, ușor de înțeles. De exemplu, pentru ultimele doua câmpuri, utilizatorul va introduce pur si simplu cele 2 capete ale intervalului: „2-10” de exemplu. Pentru restul câmpurilor, utilizatorul introduce pur si simplu numerele dorite.

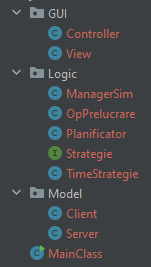
Programul verifica fiecare input al utilizatorului, deoarece pot apărea greșeli(de exemplu: utilizatorul introduce litere sau alte semne nepermise) . In cazul unei erori, utilizatorul va fi notificat, iar după poate sa introducă alt polinom, programul va rula in continuare. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

După introducerea datelor, simularea va porni iar utilizatorul va putea vizualiza in căsuța de text mare(Area Text Box) de la sfârșitul aplicației, cum evoluează cozile in timp real. După ce se termina simularea, utilizatorul va putea deschide un fișier .txt care se afla in fișierul sursa al proiectului pentru a vedea si timpul de așteptare si servire medie. De asemenea, utilizatorul are la dispoziție 3 butoane, care la apăsarea lor se vor completa automat câmpurile cu niște exemple, pentru a reduce timpul de vizualizare a funcționalității aplicației. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

# Proiectare:

* **Alegerea unor clase de obiecte de baza adecvate:**
  + O coada, descrisa de clasa Server(deoarece am presupus ca vom avea o coada de așteptare in online) va fi formata din unul sau mai mulți clienți(cereri pt. server).
  + Clientul este descris de clasa Client, ce poseda următoarele atribute: „idClient” ce este unic pentru fiecare utilizator creat, „arrivalTime” ce reprezintă timpul cand va ajunge clientul iar ultimul parametru este dat de „serviceTime” ce reprezintă timpul necesar de servire al clientului. Acești doi parametrii din urma vor fi generați aleator din intervalele date la tastatura de utilizator.
  + Client poseda câteva metode, getter si setter, un constructor si alte metode de afișare si o metoda de comparație.
  + Clasa Server poseda doua atribute speciale private. Conține o lista speciala(făcută pentru operații cu Thread-uri) ce va tine clienții si un tip special de int, ce reprezintă perioada de așteptare. De asemenea avem si 2 constructori, unul default si unul care ne seteaza timpul de asteptare.
  + Clasa Server este o clasa ce conține mai multe metode decât Client. Practic, ea reprezintă coada pe care dorim sa o simulam. Clasa Server implementează interfața „Runnable” ce ne obliga sa suprascriem metoda „run()”, ceea ce ne ajuta sa lucram cu mai multe fire de execuție/thread-uri. Clasa run practic, la execuție, in timp ce creste timpul simulării, ea scade timpul de servire al clientului aflat in fruntea cozii, iar când acel timp este 0, prima persoana din coada va fi data afara. De asemenea se poate adaugă alte persoane la coada, astfel mărind-se timpul de așteptare la coada respectiva. In rest mai avem câteva getter si setter si o metoda de afișare a clienților ce se afla in coada.
  + Aceste doua clase stau la baza programului, iar ceea ce conține fiecare este foarte intuitiv si ușor de înțeles.
* **Împărțirea programului pe clase:**

Programul este împărțit in 3 pachete mari după modelul Model-View-Controller (MVC). Cele 3 pachete sunt:

* Pachetul GUI (Grafical User Interface) ce cuprinde 2 clase, Controller si View ce stau la baza funcționarii aplicației. Clasa View este responsabila pentru crearea interfeței grafice, iar clasa Controller ne ajuta sa interacționăm cu Text-filed-urile si butoanele de pe Form;
* Pachetul Logic conține toate cele 4 clase si o interfața necesare pentru simulările pe care dorim sa le generam
* Pachetul Model conține clasele descrise mai sus: Client si Server, cele ce stau la baza unei simulari;
* In plus, mai avem o clasa numita „MainClass” ce nu aparține niciunei clase(deși s-ar putea încadra in clasa Model) care deschide interfața si inițializează controller-ul.
* Astfel, clasele sunt organizate in 3 categorii respectând principiul MVC.
* Împărțirea lor pe clase:
* 

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

* **Algoritmi folosiți:**

Cum prelucrează programul datele din câmpurile date ca si input? Nu este foarte mare filozofie, întrucât primele câmpuri se încearcă convertirea datelor introduse, iar pentru ultimele doua câmpuri unde avem introduse niște intervale, programul împarte acel string in doua numere in funcție de semnul specific „-”. Programul poate identifica si trata si cazurile speciale: când utilizatorul introduce litere sau anumite caractere speciale, sau lasă anumite câmpuri goale. In acest caz, utilizatorul va fi notificat de aplicație, repetând introduce datele din nou. De asemenea, utilizatorul poate schimba ce date dorește din cele 3 exemple date.

# Implementare:

In cele ce urmează, se vor prezenta modul in care au fost implementate clasele de obiecte, din cele 3 pachete. Nu vor fi incluse setter si getter, deoarece se subînțelege ca au fost folosite.

* **Pachetul Gui:**
  + Clasa View:

package GUI;

public class View extends JFrame{....}

Clasa View conține si inițializează toate componentele vizuale menționate mai sus: 4 butoane, 5 TextField-uri, 5 laber-uri si un area text. Este sunt plasate un panouri invizibile care la rândul lor aranjate intr-un mod specific in fereastra principala(putem considera ca sunt toate afisate sub forma de o coloana). Câteva inițializări si metode din clasa View:

private JTextField text5 = new JTextField(10);  
  
private JTextArea area1 = new JTextArea("Aici va avea loc simuarea",10,30);  
private JScrollPane scroll = new JScrollPane(area1);

public JTextField getServiceTime()  
{  
 return text5;  
}  
  
public void clearAreaText()  
{  
 area1.setText("");  
}  
  
public void setAreaText(String s){  
 area1.setText(s);  
}

* + Clasa Controller:

public class Controller implements ActionListener{  
 private View view;  
  
case "Start":  
 view.clearAreaText();  
 if(ok) {  
 //List<Client> clienti = operatii.GenerateNClients(n, arrvTimeMin, arrvTimeMax, servTimeMin, servTimeMax);  
 ManagerSim sim = new ManagerSim(view, simTime, q, n, arrvTimeMin, arrvTimeMax, servTimeMin, servTimeMax);  
  
 Thread t = new Thread(sim);  
 t.start();  
 }  
 break;  
case "EX1":  
 view.getNCLients().setText("4");  
 view.getQThreads().setText("2");  
 view.getSimTime().setText("60");  
 view.getArrivalTime().setText("2-30");  
 view.getServiceTime().setText("2-4");  
 view.clearAreaText();  
 break;

....

}

Poate fi considerat creierul aplicației, întrucât comunica intre interfața vizuala si logica din spatele programului, ajuta la rezolvarea operației selectate si la livrarea rezultatului. Câteva metode.

* **Pachetul Logic:**
  + Clasa ManagerSim:

In aceasta clasa se poate vizualiza foarte bine care sunt pașii simulării. Aceasta clasa manageriala si administrează simularea. Practic, aici se generează cei N clienți aleator si sunt distribuiți la cele Q cozi(servere). Tot aici este format stringul ce se va afișa, pentru ca utilizatorul sa vadă simularea in timp real. Asa se genereaza n Clienti.

public void GenerateNClients(){  
 int stime = 0;  
 int wtime = 0;  
 for(int i=0; i<nrOfClients; i++){//?  
 Random r = new Random();  
 int arrivalTime = r.nextInt(ArrivalTimeMax - ArrivalTimeMin) + ArrivalTimeMin;  
 int serviceTime = r.nextInt(ServiceTimeMax - ServiceTimeMin) + ServiceTimeMin;  
 Client c = new Client(i, arrivalTime, serviceTime);  
 stime += serviceTime;  
 wtime += serviceTime;  
 clientList.add(c);  
 }  
 avgWaitTime = wtime/(nrOfClients\*1.0);  
 avgServTime = stime/(nrOfServers\*1.0);  
 clientList.sort(Client::compareTo);  
}

Aceasta metoda ne arata statusul cozii mari precum si a celor mici:

public String getStatus(){ //pt a vizualiza starea cozilor, precum si cei care sunt inca in coada principala  
 String str = new String();  
 str = str + "Clienti in coada:\n";  
 for(Client t : clientList)  
 str = str + " " + t.toString();  
 str = str + "\n";  
 int i=1;  
 for(Server s : planific.getServere()) {  
 if(s.toString().isEmpty())  
 str = str + "Coada " + i + ": goala!\n";  
 else  
 str = str + "Coada " + i + ": " + s.toString() + "\n";  
 ++i;  
 }  
 str = str + "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n";  
 return str;  
}

* + Clasa Planificator:

După cum sugerează numele, aceasta coada ne planifica tot ceea ce e legat de cozi. In primul rând, ne creează Q servere si Q thread-uri si ne pornește si acele thread-uri create anterior la începutul simulării. De asemenea, distribuie clientii prin metoda „dispatchClient”.

* + Interfata Strategie:

Descrie metoda de adăugare client la o coada, „addClient”, ce primește mai multe cozi si un singur client. Este ceva generic, deoarece adevărata metoda ce distribuie clienții in funcție de strategia „cel mai mic timp de servire la o coada” se afla in clasa următoare.

* + Clasa TimeStrategie:

Care implementează interfața de mai sus, descrie metoda „addClient”. Se va adauga in coada cu timpul cel mai mic de așteptare următorul client.

public void addClient(ArrayList<Server> servers, Client client) {  
 int min = servers.get(0).getPerioadaAsteptare();  
 Server aux = servers.get(0);  
 for(Server s : servers){ //cautam coada cu perioada cea mai mica de asteptare  
 if(min>s.getPerioadaAsteptare()){  
 aux = s;  
 min = s.getPerioadaAsteptare();  
 }  
 }  
 aux.addClient(client);  
}

* **Pachetul Model:**
  + Clasa Client: una din clasele se stau la baza aplicatiei

public class Client implements Comparable<Client>{  
 private int idCLient;  
 private int arrivalTime; //simulation time when they are ready to enter the queue  
 private int serviceTime; //time interval or duration needed to serve the client, when the client is in front of the queue  
  
 public Client(int id, int arrvTime, int servTime)  
 {  
 this.idCLient = id;  
 this.arrivalTime = arrvTime;  
 this.serviceTime = servTime;  
 }

. . . }

Dintre metodele importante, regăsim:

@Override  
public String toString(){  
 return "["+idCLient+","+arrivalTime+","+serviceTime+"]";  
}  
  
@Override  
public int compareTo(Client o) {  
 return this.arrivalTime - o.arrivalTime;  
}

Ne ajuta pentru sortarea automată a cozii principale in funcție de timpul de sosire.

* + Clasa Server:

public class Server implements Runnable{ //server = coada  
 private BlockingDeque<Client> clientList =new LinkedBlockingDeque<Client>();  
 private AtomicInteger perioadaAsteptare;

. . .}

Aceasta clasa procedează doua atribute, ce este un BlockingDeque de Clienti si un AtomicIntger ce reprezintă timpul de așteptare la coada. In aceasta clasa regăsim tot felul de metode folositoare pentru simulare. De exemplu, metoda necesara pentru prelucrarea thread-urilor, run():

while(true)  
{  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {e.printStackTrace();}  
 if(!clientList.isEmpty()){ //verificam dacam mai avem clienti in coada  
 clientList.element().setServiceTime(clientList.element().getServiceTime()-1);

public void mulScalar(int x)  
{  
 for(Monom m: polinom)  
 {  
 m.setNr(m.getNr() \* x);  
 }  
}  
  
public void mulMonom(Monom m)  
{  
 for(Monom myMonom: polinom)  
 {  
 myMonom.setNr(myMonom.getNr()\*m.getNr());  
 myMonom.setExp(myMonom.getExp()+m.getExp());  
 }  
}

...}

In rest mai avem o metoda de adăugat client si restul metodelor sunt pentru afișare si obținere a anumitor informații.

# Concluzie:

Aplicația dezvoltata respecta paradigmele programării orientate pe obiect si este un bun reminder al cunoștințelor învățate sem. trecut., in special „thread-urile”. Sistemul creat gestionează corespunzător o simulare a unei cozi de așteptare si oferă utilizatorului informații in timp real, iar la sfârșitul aplicație utilizatorul poate revedea informațiile din simulare in aplicație sau intr-un fișier txt. De asemenea metodele si structurile de date au fost special folosite pentru a opera cu thread-uri.

Deși este simpla aplicația si își deservește scopul, totuși este loc de îmbunătățiri si upgrade-uri. Aplicația poate avea pe viitor o alta strategie de a distribui clienții la cozile create. De asemenea se mai pot implementa niște câmpuri adiționale, pentru a se putea vedea informații in plus pentru a avea o simulare mai detaliata.

# Bibliografie:

<https://docs.oracle.com/en/>

<https://www.baeldung.com/java-write-to-file>

<https://www.w3schools.com/java/java_files_create.asp>

https://www.w3schools.com/java/java\_threads.asp

<https://www.jetbrains.com/help/idea/working-with-code-documentation.html>

http://tutorials.jenkov.com/java-concurrency/creating-and-starting-threads.html