SIMULARE COZI CU THREAD-URI

DOCUMENTAȚIE

Studentă: Aurică Alina

Materie: Tehnici de programare

Profesor îndrumător: Mitrea Dan

CUPRINS:

1. Obiectiv
2. Studiul problemei
3. Implementare

* Diagrame UML
* Clase și metode
* GUI

1. Testare
2. Concluzii
3. Bibliografie
4. OBIECTIV:

Obiectivul proiectului este de a implementa și proiecta o simulare a cozilor de la un supermarket, având o strategie bazată pe timpul de așteptare la coadă. Pe scurt, aplicația trebuie să introducă în cozi clienți în funcție de timpul de așteptare specific fiecărei cozi, trebuie să proceseze clientul și trebuie să-l elemine de la coadă, după ce comanda lui a fost realizată.

Vizualizarea acestor cozi se face printr-o interfață grafică, unde se observă dinamica fiecărei cozi în funcție de momentul de timp în care ne aflăm.

Structura internă a aplicației este susținută prin thread-uri. Un thread, în contextul programării în Java, este calea urmată la executarea unui program. Thread-urile sunt folosite pentru execuția în paralel a aceleiași bucăți de cod, dar având valori diferite ale variabilelor globale.

1. STUDIUL PROBLEMEI:

În mod general, noi avem clienți pe care dorim să-i așezăm la cozi. Cozilor le sunt asociate thread-uri pentru a putea funcționa simultan.

Cele 2 strategii care pot fi implementate pentru punerea în cozi sunt:

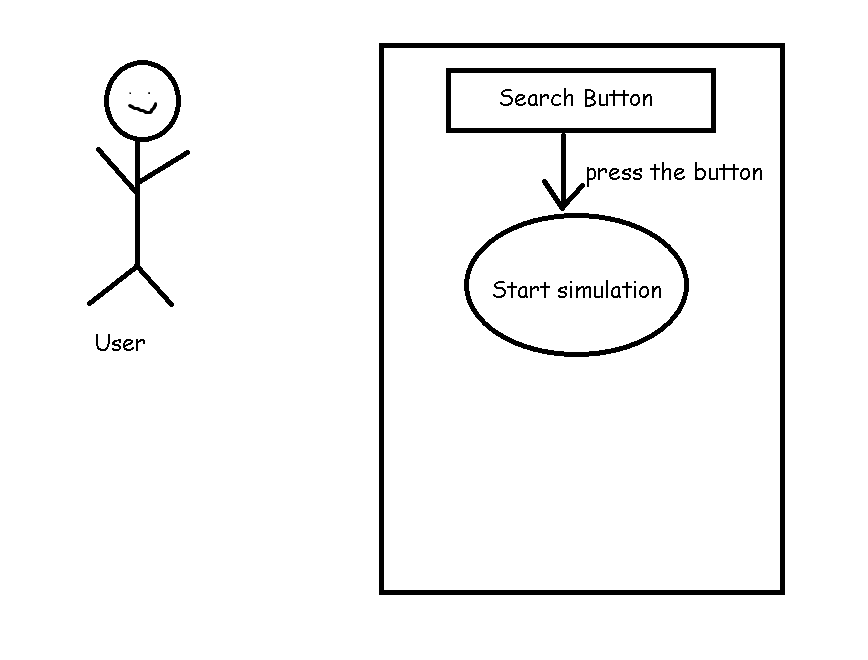
* În coada care are waitingTime-ul cel mai mic, se adaugă un nou client.
* În coada care are cei mai puțini clienți se adaugă un nou client.

Metoda pe care am ales să o implementez și care este cea mai eficientă din punct de vedere al timpului este bazata pe prima strategie, cea cu waitingTime-ul.

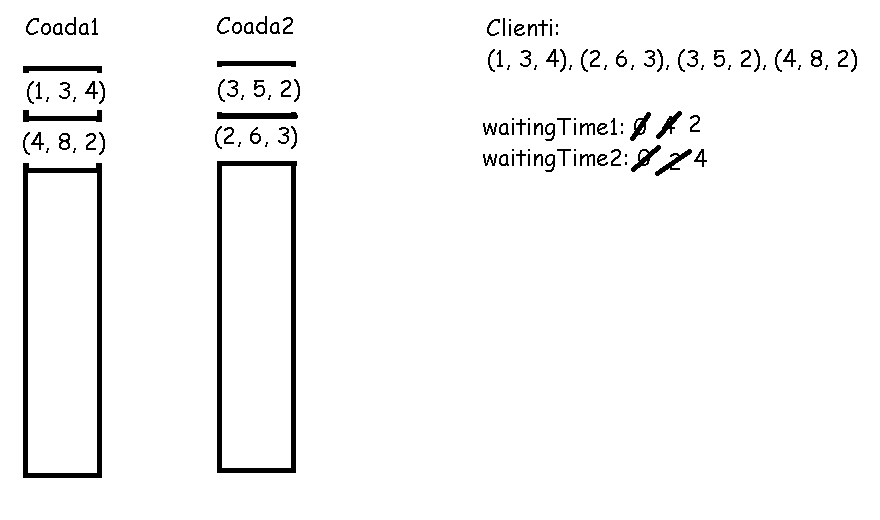
Clienții sunt generați random, în cadrul unei metode, pentru a valida funcționalitatea proiectului pentru orice caz. Metoda clasei Random, rand.nextInt() este folosită pentru timpul de venire și pentru timpul de procesare. Se folosește acestă metodă deoarece ne dorim să avem timpi numere întregi, fiind mult prea complicat să lucrăm într-o unitate de măsură a timpului mai mică decât secunda.

private void generateNRandomTasks(){  
 //generateClient = new ArrayList<Client>();  
 Random rand = new Random();  
 for(int i = 1; i <= nrClienti; i++)  
 {  
 //System.out.println(nrClienti);  
 int arrivalT = rand.nextInt(maxArrivalTime - minArrivalTime) + minArrivalTime;  
 int processingT = rand.nextInt(maxProcessingTime - minProcessingTime) + minProcessingTime;  
  
 Client c = new Client(i, arrivalT, processingT);  
 //System.out.println(c.toString());  
 generateClient.add(c);  
 }  
  
 Collections.*sort*(generateClient);  
}

Un mod ușor de a vizualiza ceea ce e capabil să facă aplicația, la care se adaugă și o imagine per ansamblu a funționalității este diagrama Use Case:



De altfel, în imaginea de mai jos voi ilustra funcționalitatea strategiei alese.



După cum putem observa, la timpul t = 3, a fost introdusă în coada1, clientul 1. I s-a mărit waitingTime-ul cu tService. La momentul t = 5, s-a introdus în a doua coadă clientul 3, deoarece prima coadă mai are o secundă din timpul de procesare al primului client. WaitingTime-ul cozii 2 crește cu tService. Pe măsură ce secundele trec, din timpul de servire pentru un client se scade câte o secundă, până când tService devine egal cu 0 și este eliminat din coadă. Simultan se scade și din waitingTime-ul cozii. Acest lucru se repetă în mod sincron, pentru fiecare listă în paralel, până când ultima persoană este procesată.

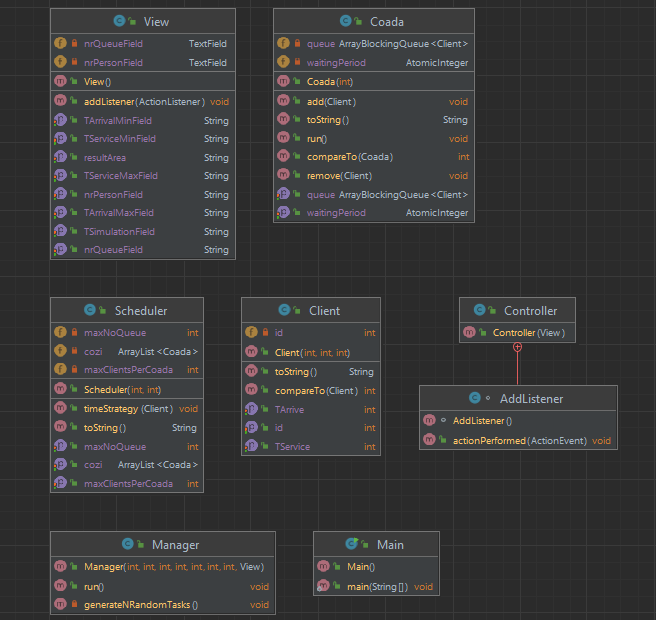
1. IMPLEMENTARE:

* DIAGRAMA UML:

Unified Modeling Language (UML) este un limbaj standard pentru descrierea de modele si specificații pentru software. UML a fost dezvoltat la început pentru reprezentarea complexității programelor care erau construite utilizând programarea orientată pe obiect, al căror fundament este structurarea programelor pe clase și instanțele acestora (obiecte).

Am utilizat UML-ul pentru a vedea mult mai clar împărțirea pe clase a codului, precum și variabilele instanță și metodele pe care le-am utilizat în dezvoltarea strategiei cozilor cu thread-uri.

În această situație, un real ajutor îl constituie și împărțirea claselor în pachete, fiind astfel mult mai ușor de urmărit codul și de legat componentele.

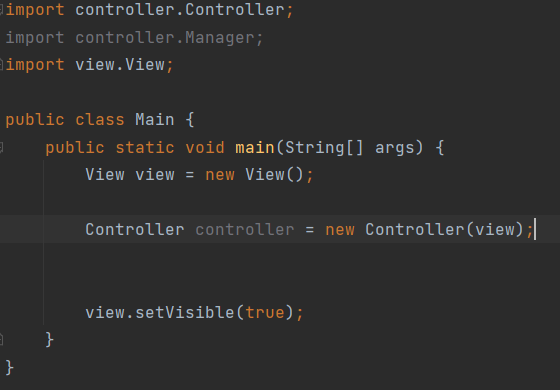


* CLASE ȘI METODE:

Pentru structurarea codului am utilizat modelul MVC (Model, View, Controller), adică am împărțit codul în 3 pachete, la care am adăugat clasa Main.

Clasa Main controlează tot MVC-ul, având două instanțe; una a clasei View, pe care o setăm vizibilă, pentru a putea vedea interfața grafică (GUI) și cealaltă este o instanță a clasei Controller, care leagă Model-ul de View, fiind principalul punct de comandă al celor două.

Fără această clasă, care are rol unificator, utilizatorul nu s-ar putea bucura nici de interfața grafică cu utilizatorul (GUI), și nici de funcționalitățile simulării unei cozi. Noi vedem în timp real introducerea, procesarea și excluderea clienților din cozi și vedem acest lucru pe fiecare coadă în parte.



Pachetul model conține 3 clase: Client, Coada și Scheduler.

Clasa Client:

Clasa client conține 3 variabile instanță declarate private, pentru care se realizează metode de get și set. Cele 3 variabile instanță sunt: id, tArrival și tService și sunt de tip întreg. Ele sunt introduse ca parametri ai constructorului clasei specificate anterior.

public class Client implements Comparable<Client>{  
 private int id;  
 private int tArrive;  
 private int tService;  
  
 public Client(int id, int tArrive, int tService){  
 this.id = id;  
 this.tArrive = tArrive;  
 this.tService = tService;  
 }

Clasa implementează interfața Comparable cu parametrul Client și definește metoda compareTo, deoarece în codul programului în clasa Manager, dorim să sortam crescător în funcție de tArrive o listă de clienți.

public int compareTo(Client c){  
 return this.tArrive - c.tArrive;  
}

Pe lângă acestă metodă, în clasă suprascriem și metoda toString, care ne ajută să afișăm într-un anumit format clienții.

@Override  
public String toString(){  
   
 return "( " + this.id + ", " + this.tArrive + ", " + this.tService + " )";  
}

Clasa Coadă:

Clasa Coadă funcționează ca și un thread, deoarece implementează interfața Runnable. Această interfață vine o metoda run() ce trebuie implementată. Metoda run a acestei clase trebuie să proceseze câte un element din coadă. Mai intâi îi scade timpul de servire Clientului, iar când acesta ajunge să fie egal cu 0, elimină clientul din coadă. În același timp, decrementează cu 1 și waitingPeriod-ul cozii respective.

Trebuie menționat ca thread-ul este pus pe așteptare în momentul în care lucrăm cu acesta. Pentru a atinge această funcționalitate, avem nevoie de un apel al metodei statice sleep(), prin intermediul clasei Thread.

@Override

public void run() {  
  
 while (true) {  
 if(!queue.isEmpty()) {  
 Client c = queue.element();  
 c.setTService(c.getTService()-1);  
  
 if(c.getTService() == 0) {  
 queue.remove(c);  
 }  
 }  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 //ceva  
 }  
 waitingPeriod.decrementAndGet();  
 //this.waitingPeriod.addAndGet(-1);  
 }  
}

Această coadă conține o metodă add() care introduce clienți în coadă și adaugă la timpul de așteptare al cozii, timpul de servire al clientului. Metoda are și parametrul synchronized între cuvântul care specifiă vizibilitatea și cuvântul care specific tipul de dată returnat, deoarece pot fi adăugați clienți simultan în 2 cozi diferite.

public synchronized void add(Client c){  
 this.queue.add(c);  
 this.waitingPeriod.addAndGet(c.getTService()); //cu ce ar trebui incrementat waitingPeriod????  
 System.*out*.println("wPeriod: " + this.waitingPeriod);  
}

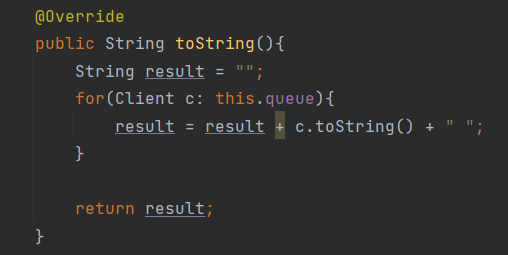
Pe lângă interfața Runnable, Coada implementează și interfața Comparable, pentru a sorta un ArrayList de cozi in funcție de waitingPeriod:

@Override  
public int compareTo(Coada c) {  
 return this.waitingPeriod.get() - c.waitingPeriod.get();  
}

Este important de menționat că waitingPeriod este un AtomicInteger, adică este o variabilă sincromizată, ce are capacitatea de a fi citită doar după ce este modificată. De altfel, pentru prima dată am implementat un ArrayBlockingQueue<Client> care reprezintă una din formele de implementare a colecției Queue. Scopul ei este de a putea folosi metodele de add() și remove() specifice acestei colecții. De asemenea, este o reprezentare în abstract a noțiunii de zi-cu-zi (neabstractă) numită coadă. Variabila prin care îi setăm lungimaea este reprezentată de numărul maxim de clienți pe care îi avem în total. Această decizie este luată pentru că, de la bun început, conștientizăm că într-o coadă, la un anumit timp, nu pot fi adăugați mai mult de numărul maxim de clienți pe care îi avem la început.

public class Coada implements Runnable, Comparable<Coada>{  
 private ArrayBlockingQueue<Client> queue;  
 private AtomicInteger waitingPeriod;  
  
 public Coada(int q){  
 queue = new ArrayBlockingQueue<Client>(q);  
 waitingPeriod = new AtomicInteger(0);  
 }

Pe lângă toate acestea, avem pentru fiecare variabilă instanță getter și setter, plus o suprascriere a metodei toString pentru a afișa o coadă și elementele din ea.



Clasa Scheduler:

Această clasă conține un ArrayList de cozi (ArrayList<Coada> cozi), un număr maxim de cozi (maxNoQueue) și un numar maxim de clienți pentru fiecare coadă (maxClientsPerCoada). Ultimele două variabile instanță (atributele clasei) sunt oferite ca parametru constructorului.

În interiorul constructorului se generează cozile, se adaugă în lista de cozi și se pornește câte un thread din fiecare coadă.

public Scheduler(int maxNoQueue, int maxClientsPerCoada)  
{  
 this.cozi = new ArrayList<Coada>();  
 for( int i = 0; i < maxNoQueue; i++) {  
 Coada queue = new Coada(maxClientsPerCoada);  
 this.cozi.add(queue);  
 Thread t = new Thread (queue);  
 t.start();  
 }  
}

În acestă clasă se implementează și o metodă care se bazează pe strategia de timp menționată în capitolul anterior:

public void timeStrategy(Client client)  
{  
   
 Collections.*sort*(this.cozi);  
 System.*out*.println(this.cozi.get(0).getWaitingPeriod());  
 this.cozi.get(0).add(client);

}

Înainte să adaugăm într-o coadă, sortam lista de cozi prin apelul metodei sort, cu ajutorul clasei Collections. Același mecanism se regăsește și în clasa Manager, pentru sortarea listei de clienți.

Restul codului din acestă clasă este reprezentat de o metodă toString() și de gettere și settere.

Pachetul Controller conține, la rândul său, 2 clase: Controller și Manager.

Clasa Controller:

Această clasă leagă view-ul de clasa Manager și se ocupă cu logica programului. Ea conține în constructor o instanțiere a clasei View și un apel al metodei addListener prin intermediul view-ului. Ca parametru al metodei, se instanțiază clasa AddListener().

În interiorul clasei, avem o clasă internă, AddListener() care implementează ActionListener și suprascrie metoda actionPerformed(). În acestă metodă se preia din interfață valori precum numărul de Clienți, numărul de Cozi, timeLimit (durata de funcționare a unei cozi) și invervalele de timp pentru sosirea clienților și pentru servirea lor.

int nrClienti = Integer.*parseInt*(view.getNrPersonField());

Valorile sunt de timp întreg, iar ceea ce se preia din UI, din textField sunt valori de tip String. Astfel, devine necesară o conversie de tip și din acest motiv apelăm metoda parseInt().

Tot în această clasă internă este instanțiat Manager-ul și i se dă start thread-ului principal, care va porni cozile.

class AddListener implements ActionListener {  
  
 public void actionPerformed(ActionEvent e) {  
  
 int nrClienti = Integer.*parseInt*(view.getNrPersonField());  
 System.*out*.println(nrClienti);  
 int nrCozi = Integer.*parseInt*(view.getNrQueueField());  
 System.*out*.println(nrCozi);  
  
 int timeLimit = Integer.*parseInt*(view.getTSimulationField());  
 System.*out*.println(timeLimit);  
  
 int maxProcessingTime = Integer.*parseInt*(view.getTServiceMaxField());  
 System.*out*.println(maxProcessingTime);  
 int minProcessingTime = Integer.*parseInt*(view.getTServiceMinField());  
 System.*out*.println(minProcessingTime);  
  
 int maxArrivalTime = Integer.*parseInt*(view.getTArrivalMaxField());  
 System.*out*.println(maxArrivalTime);  
 int minArrivalTime = Integer.*parseInt*(view.getTArrivalMinField());  
 System.*out*.println(minArrivalTime);  
  
 gen = new Manager(nrClienti, nrCozi, timeLimit, maxProcessingTime, minProcessingTime, maxArrivalTime, minArrivalTime, view);  
 Thread t = new Thread(gen);  
 t.start();  
  
 //view.setResultField(gen.result);  
 }  
}

Clasa Manager:

Constructorul acestei clase primește ca parametri toate valorile extrase din interfața grafică, alături de un obiect al clasei View. În interiorul ei sunt inițializate un scheduler, un view și se apelează metoda generateNRandomTasks(), care generează random clienții. Aceștia sunt adăugați într-o listă cu vizibilitatea public. La finalul metodei antemenționate, lista este sortată după tArrival al fiecărui client.

public Manager(int nrClienti, int nrCozi, int timeLimit, int maxProcessingTime, int minProcessingTime, int maxArrivalTime, int minArrivalTime, View view){  
 this.nrClienti = nrClienti;  
 System.*out*.println(nrClienti);  
 this.nrCozi = nrCozi;  
 System.*out*.println(nrCozi);  
  
 System.*out*.println("Inainte de scheduler");  
  
 this.scheduler = new Scheduler(nrCozi, nrClienti);  
 this.view = view;  
  
 this.timeLimit = timeLimit; //constructor  
 System.*out*.println(timeLimit);  
 this.maxProcessingTime = maxProcessingTime;  
 System.*out*.println(maxProcessingTime);  
 this.minProcessingTime = minProcessingTime;  
 System.*out*.println(minProcessingTime);  
 this.maxArrivalTime = maxArrivalTime;  
 System.*out*.println(maxArrivalTime);  
 this.minArrivalTime = minArrivalTime;  
 System.*out*.println(minArrivalTime);  
  
  
 generateNRandomTasks();  
 for(Client c: generateClient)  
 {  
 System.*out*.println(c.toString());  
 }  
  
 System.*out*.println("ies din constructor");  
}

Clasa implementează și interfața Runnable, ceea ce înseamnă că suprascrie metoda run(). Această metodă se ocupă cu logica de la fiecare secundă a timpului curent. Atât timp cât nu am depășit limita de timp, se extrage din lista de clienți un client și se adaugă într-o coadă pe baza strategiei de timp. Se actualizează interfața grafică și fișierul în care se scrie. Crește timpul curent și se oprește simularea dacă atât cozile, cât și lista de clientți sunt goale.

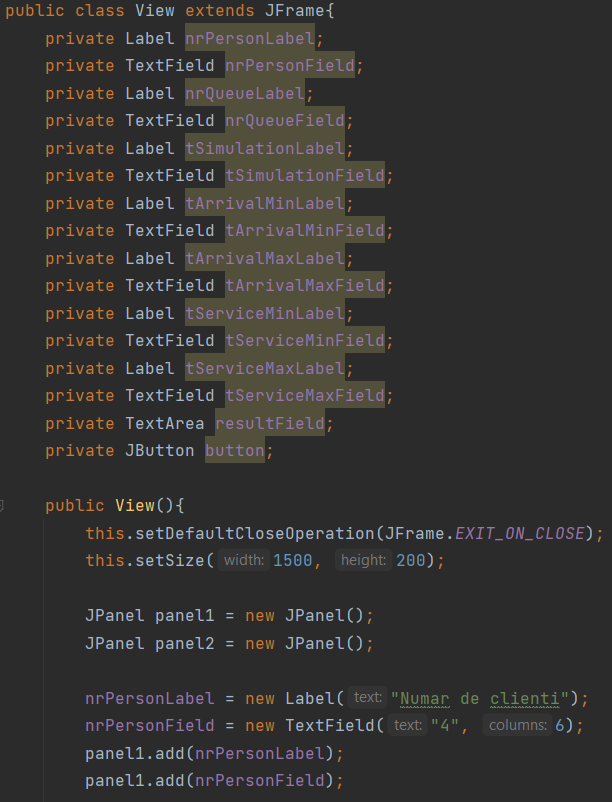
@Override  
public void run() {  
 int currentTime = 0;  
 int maxClienti = 0; double sumWaitingTime = 0.0;  
   
 while(currentTime < timeLimit) {  
 int sumClienti = 0; double maxWaitingTime = 0.0;  
 if(generateClient.size() != 0) {  
 Client c = generateClient.get(0);  
 while (currentTime == c.getTArrive() && generateClient.size() != 0) {  
 scheduler.timeStrategy(c); generateClient.remove(0);  
 if(generateClient.size() != 0)  
 c = generateClient.get(0);}}  
 for(int i = 0; i < nrCozi; i++) {  
 sumClienti += scheduler.getCozi().get(i).getSize();  
 for(Client c: scheduler.getCozi().get(i).getClientList())  
 maxWaitingTime += c.getTService();  
 if(scheduler.getCozi().get(i).getSize() != 0) {  
 maxWaitingTime = maxWaitingTime / scheduler.getCozi().get(i).getSize();  
 sumWaitingTime += maxWaitingTime;}}  
 if(sumClienti > maxClienti) { maxClienti = sumClienti; pickHour = currentTime; }  
  
 result = result + currentTime + "\n" + generateClient + "\n" + scheduler.toString() + "\n";  
 view.setResultArea(result); scriereFisier(result);  
 currentTime++;  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();}  
 if(coziP() == 0 && generateClient.size() == 0)  
 break;  
 }  
 sumWaitingTime = sumWaitingTime / (currentTime - 1);  
 result += "pickHour: " + pickHour + "\n" + "average service time: " + medServiceTime + "\n" + "average waiting time: " + sumWaitingTime;  
 view.setResultArea(result); scriereFisier(result);  
}

Pentru a actualiza fișierul în care se scrie, am creat o metodă scriereFisier(), cu parametru un element de tip String, unde are loc crearea și scrierea într-un fișier.

* GUI

Interfața grafică cu utilizatorul este creată prin clasa View și legată de restul codului prin Controller. Pentru a crea o astfel de interfață avem nevoie de librării speciale, precum: javax.swing și java.awt. Acestea sunt librării create special pentru GUI.

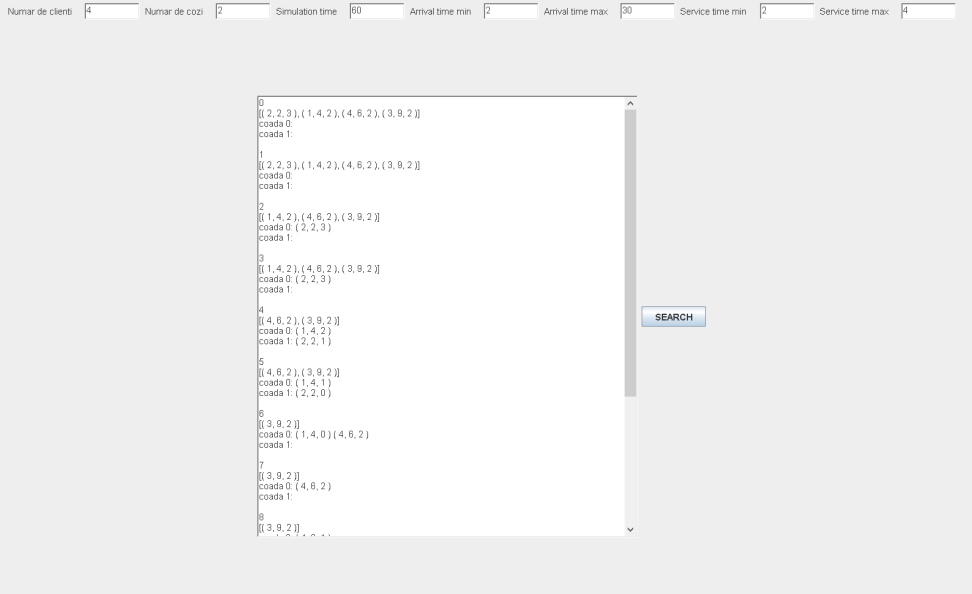
Aceasta este o porțiune din clasa View, care conține getterere și setterere pentru fiecare JtextField și addListener() pentru fiecare buton:



Observăm cum toate atributele clasei sunt declarate private și inițializate în constructor. Apoi sunt adăugate în JPanel.

De asemenea, clasa View extinde JFrame, clasă care se ocupă cu crearea propriu-zisă a ferestrei pe care noi o numim convențional Interfață Grafică cu Utilizatorul.

Aceasta este interfața grafică pe care o vizualizează utilizatorul, după ce apasă pe butonul search:

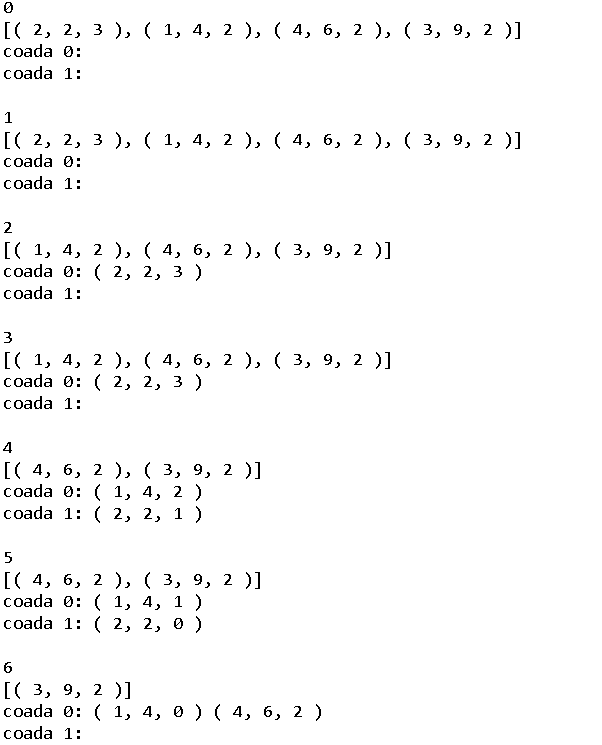


Interfața conține un singur buton, care pornește thread-urile și afișarea în interiorul ferestrei. De asemenea, conține 7 textField-uri și un textArea în care afișăm rezultatul. Cele 7 textField-uri sunt create pentru a introduce numărul de clienți, numărul de cozi, timpul de simulare și intervalele de timp pentru venirea clienților, dar și pentru servirea acestora. Interfața este structurată în 2 JPanel-uri, pentru a obține acest aspect al interfeței. Toate aceste panel-uri au fost adăugate într-un al 3-lea JPanel, care este inclus la rândul său într-un JFrame.

1. TESTARE:

Testarea codului se realizează prin afișare atât în GUI, cât și într-un fișier test, unde putem urmări punerea și scoaterea din cozi, precum și scăderea timpului de servire al fiecărui client.

Mai jos avem o afișare în fișierul text, în timp ce mai sus se regăsește o afișare în interfața grafică cu utilizatorul. Pentru afișarea în fișier a trebuit să tratăm codul printr-o excepție de tip try and catch.



1. CONCLUZII:

Concluzia acestui proiect pe care tocmai l-am realizat este că un astfel de dispozitiv ușurează foarte mult viața oamenilor care lucrează în domenii precum comerț, domeniul bancar, mai precis în fiecare domeniu în care se așteaptă la coadă.

În ceea ce mă privește, proiectul mi-a oferit oportunitatea de a-mi dezvolta lucrul cu limbajul Java, mai precis cu paradigma programării orientate pe obiect. Mi-a oferit oportunitatea de a crea un proiect complex, ce utilizează numeroase librării, precum: JUnit, javax.swing și java.awt. De asemenea, m-a ajutat să-mi reamintesc tehnicile de programare învățate semestrul trecut și să-mi îmbunătățesc lucrul cu thread-uri. Pe lângă toate acestea, mi-am readus aminte modul prin care pot utiliza interfața Comparable pentru a realiza sortări în funcție de criterii prestabilite.

1. BIBLIOGRAFIE:

<https://dsrl.eu/courses/pt/materials/A2_Support_Presentation.pdf>

<https://stackoverflow.com/>

<https://www.geeksforgeeks.org/>

<https://www.javatpoint.com/how-to-generate-random-number-in-java>