

Documentație

Aplicatie de simulare a cozilor

Student: Costea Ovidiu-Bogdan

Grupa:30229

Contents

1.Obiectivul temei:.....	3
2.Analiza problemei ,modelare,cazuri de utilizare	3
2.1Analiza problemei	3
2.2. Modelarea.....	3
2.3.Cazuri de utilizare.....	3
3.Proiectare.....	4
3.1. Structuri de date.....	4
3.2. Clase	5
3.3Algoritmi folosiți	5
3.4. Pachete	5
3.5. Interfețe	6
3.6.Interfața cu utilizatorul	6
4. Implementare	7
5.Rezultat	13
6.Concluzie	13
7.Bibliografie.....	14

1. Obiectivul temei:

Scopul principal al acestei teme a fost dezvoltarea unei aplicații, în limbajul de programare Java, capabile să simuleze cozile unui magazin. Prin simpla analiză a cozilor va trebui să determinăm coada cu timpul cel mai mic de așteptare, pentru a minimiza astfel timpul de așteptare al clienților.

Proiectul realizat de mine simulează cozile unui magazin în timp real, are o interfață prietenoasă implementată cu ajutorul pachetelor Swing și AWT și oferă posibilitatea setării din interfața grafică a numărului de clienți, cozi și intervalele de servire, generare de clienți.

2. Analiza problemei, modelare, cazuri de utilizare

2.1 Analiza problemei

La o primă citire a cerinței ne-am putea gândi la supermarketurile la care mergem în fiecare zi (ex: Lidl). Acolo există un număr de cozi (threaduri) și un alt thread reprezintă intrarea în magazin unde fiecare client își alege coada la care vrea să meargă. Toate acestea ar trebui realizate într-un interval de simulare introdus de către utilizator. La final trebuie să afișăm timpul mediu de așteptare pentru fiecare coadă în parte, dar și timpul în care au fost cei mai mulți clienți la coadă.

2.2. Modelarea

Pentru implementarea acestui sistem de simulare am avut nevoie de mai multe clase. Dintre care una pentru clienți care reține informații legate de timp și un ID unic al fiecărui client în parte pentru a fi vizibilă corectitudinea funcționării sistemului.

Pe lângă această clasă am mai avut nevoie de o clasă care va reprezenta cozile din magazin și o altă care va ilustra "intrarea în supermarket", utilizată doar pentru dirijarea clienților spre casa care oferă un timp minim de așteptare. Alte clase au mai fost utilizate pentru implementarea interfeței grafice și realizarea designului MVC.

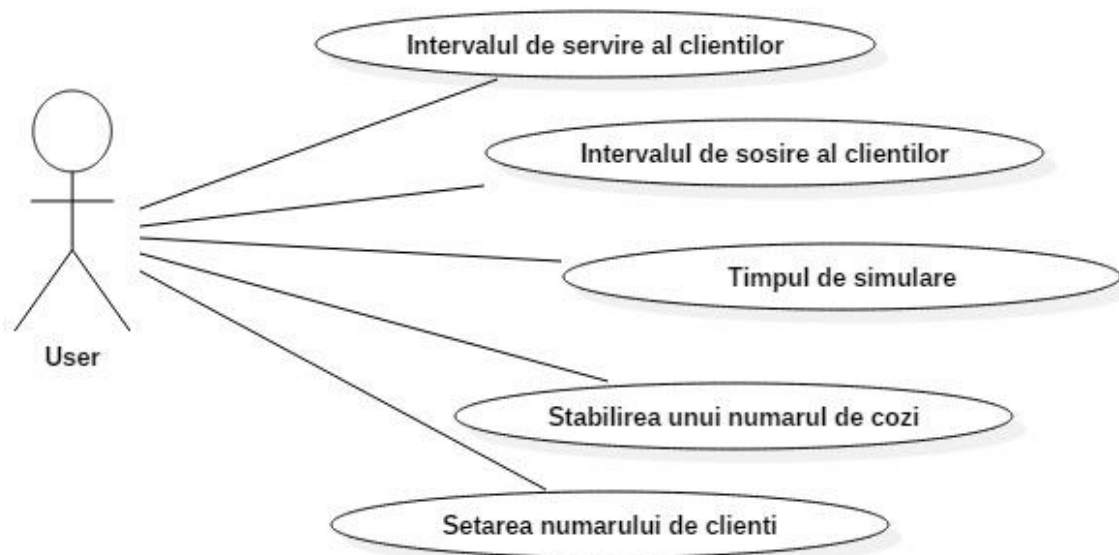
2.3. Cazuri de utilizare

Utilizarea acestei aplicații urmărește respectarea unor reguli înainte începerii unei noi simulări. Astfel toate câmpurile trebuie completate, dar unele necesită o mai mare grijă înainte executiei.

Toate câmpurile trebuie completate cu numere strict mai mari decât 0, în cazul în care unul dintre câmpuri conține valori negative sau utilizatorul va primi o atenționare. Intervalele de servire/sosire a clienților trebuie urmărite tiparul timpului minim strict mai mic decât timpul maxim, altfel se va primi o atenționare.

În cazul în care toate cerințele necesare sunt completate simularea va începe.

Diagrama USE-CASE:



3. Proiectare

Aplicatia a fost dezvoltată printr-o abordare top-down, sistemul fiind proiectat ca un tot, apoi acesta a fost divizat în subsisteme. Clasele au fost realizate într-un mod abstract, acestea așteptându-se la un rezultat specific fără a furniza instrucțiuni de returnare a acelui rezultat. După ce un model acceptabil a fost atins metodele au putut fi definite.

Motivul utilizării acestei abordări a fost faptul că oferă un foarte bun nivel de abstractizare, acest lucru îmbunătățind viteza de dezvoltare a aplicației, un cod mai ușor de întreținut și mai puțin predispus la erori.

3.1. Structuri de date

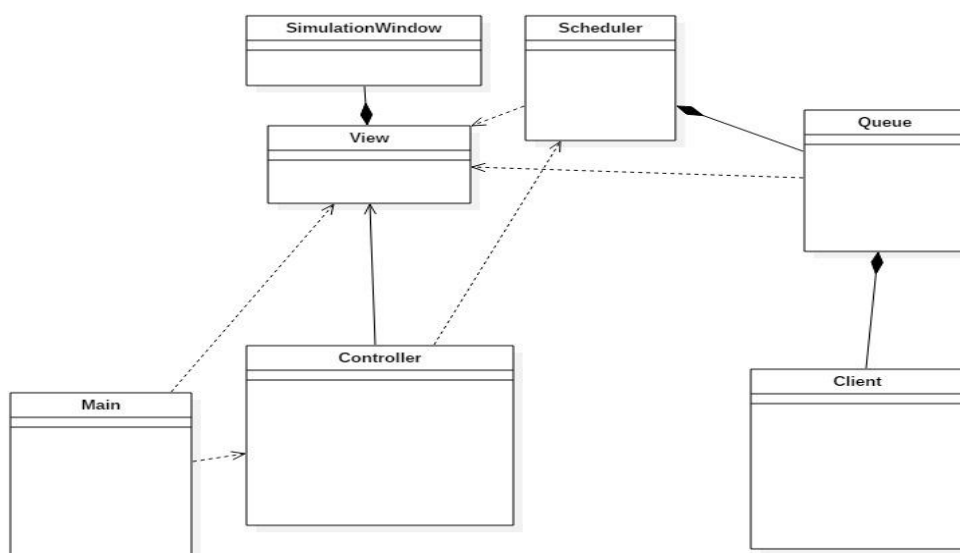
Din cauza faptului că numărul de clienți conținuți de fiecare coadă este variabil am optat pentru utilizarea unei structuri de date de tip ArrayList, care se va comporta exact ca o coadă (FIFO - First In First Out). O utilizare a unui simplu array de clienți ar putea fi o greșeală destul de mare mai ales în cazul depășirii limitei impuse acestui array cauzând astfel erori destul de grave. Acest tip de structură s-a utilizat și în cazul cozilor care supra cum bine am stabilit pot fi setate execuției simulării.

3.2. Clase

Funcționalitatea programului conține următoarele șapte clase. Clasa "main" reprezintă clasa care execută programul.

Celelalte clase "View", "SimulationWindow", "Client", "Queue", "Scheduler", "Controller" contribuie la crearea designului MVC. Interfața grafică este implementată de clasele "View" și "SimulationWindow" care moștenesc clasa JFrame. În clasa "Controller" vom face legăturile dintre pechetele "view" și "model" pentru a sigura un răspuns vizibil utilizatorului. Clasele "Scheduler", "Queue", "Client" se ocupa stric de simularea cozilor unui magazin și reținerea datelor despre fiecare client, primele doua clase extind clasa Thread sau implementeaza interfata Runnable, lucru care ajuta executia simultana a unui cod in mai multe procese.

Diagrama UML de clase :



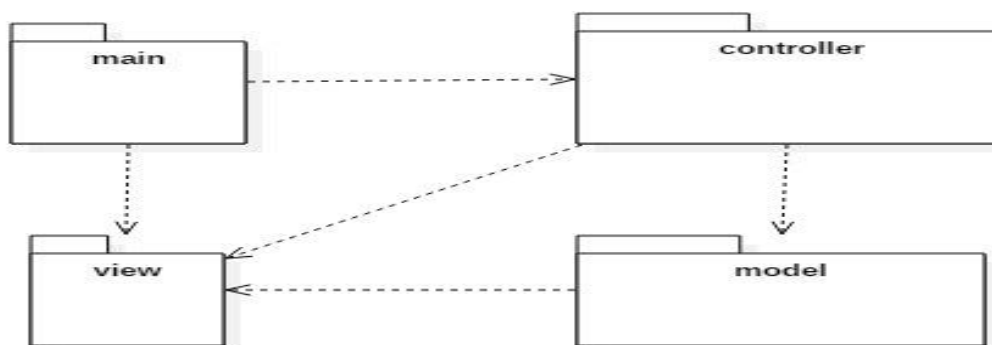
3.3 Algoritmi folosiți

Algoritmi folosiți în dezvoltarea acestei probleme sunt destul de simpli. Pentru fiecare moment al simulării în care avem unul sau mai mulți clienți noi căutăm coada care are timpul de așteptare minim. După așezarea clientului/ clienților, fiecare coadă trebuie să proceseze (aștepte) un timp egal cu timpul de procesare al primului client. Acest algoritm se va repeta atât timp cât încă timpul actual este mai mic sau egal cu timpul de simulare ales de utilizator.

3.4. Pachete

Cât posibil am încercat încadrarea fiecărei clase în câte un pachet, astfel am ajuns să folosim pachetele: model, view, controller, main. Pachetul model conține clasele Client, Queue și

Scheduler(folosita pentru alegerea unei cozi pentru utilizator nou venit). Pachetul view contine clasa View in care a fost implementata interfata grafica pentru setarea campurilor necesari simulari si clasa SimulationWindow , care deschide o noua fereastra in care va fi posibila urmarirea in timp real a evolutiei cozilor. Celelalte pachete contin cate o clasa denumite la fel ca numele pachetului.

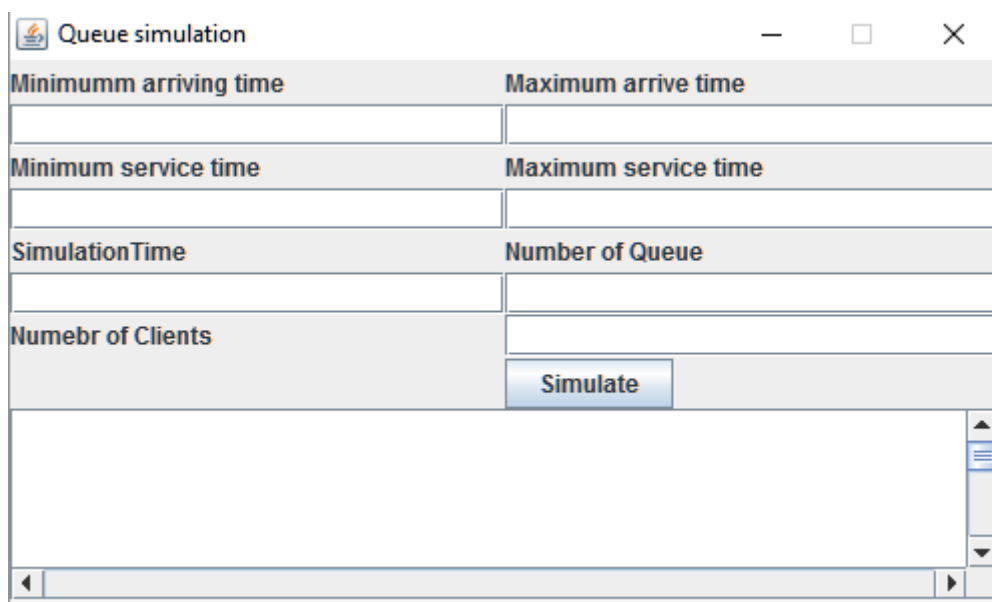


3.5. Interfețe

Clasa Scheduler implementeaza interfața Runnable pentru a face posibila rulara in acelasi timp cu executia proceselor din clasa Queue care extinde clasa Thread. Aceasta interfața necesita implementarea metodei run().

3.6. Interfața cu utilizatorul

Este folosita o interfata grafica foarte simpla și destul de intuitivă care conține sapte TextFielduri, sapte Labeluri, un TextArea si un Buton.



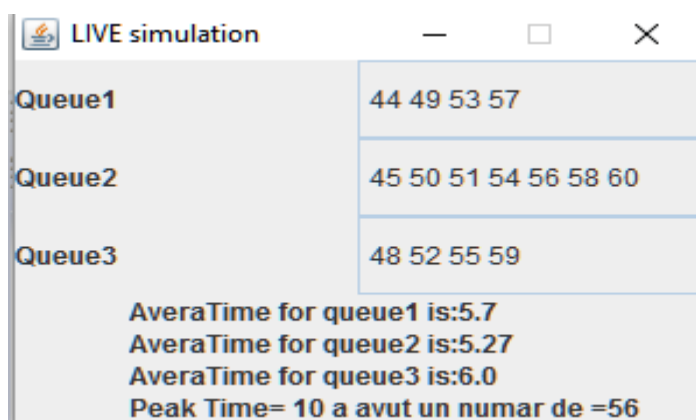
The screenshot shows a window titled "Queue simulation" with standard Windows window controls (minimize, maximize, close). The window contains a form with the following fields and controls:

- Minimum arriving time** and **Maximum arrive time** (Note: typo in image)
- Minimum service time** and **Maximum service time**
- SimulationTime** and **Number of Queue**
- Numebr of Clients** (Note: typo in image)
- A **Simulate** button
- A large text area at the bottom for output or logs.

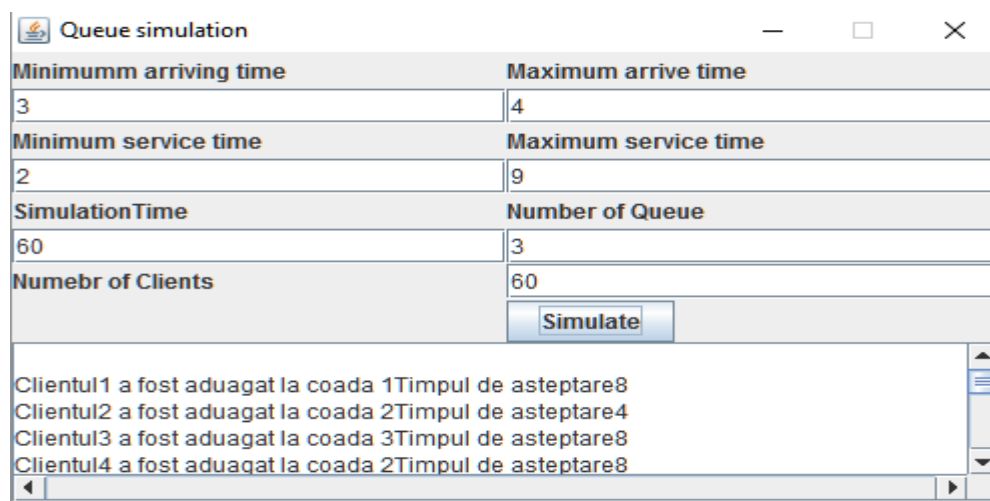
În acord cu regulile de completare a campurilor mentioante si mai sus , utilizatorul trebuie să introducă cate un numar pentru fiecare textfield in parte. Odată ce datele au fost introduse mai

ramâne decât să apăsăm pe butonul Simulate. O dată ce este detectat un eveniment al butonului programul își începe simularea. Pe ecran va apărea o nouă fereastră creată de clasa SimulationWindow în care se va putea observa în timp real clienții care se află în coadă la momentul actual.

În timpul simulării zona componentei TextArea se va reactualiza automat la momente actuale cu informații legate de clienți. Mesajele se afișează doar atunci când un client se așează la coadă și iese de la coadă. În momentul când un nou cumpărător este asociat unei cozi pe lângă numele său și numărul randului la care se așează am optat să afișez și timpul de servire al acestuia.



În imaginea de mai sus putem observa fereastra care va conține evoluția cozilor precum și rezultatele în urma simulării (timpul mediu de așteptare și momentul în care au fost cei mai mulți oameni).



Putem observa mesajele care apar în acest TextArea chiar în timpul simulării.

4. Implementare și Testare

Programul a fost scris prin implementarea claselor deja menționate. În fiecare clasă au fost implementate metode folosind algoritmi deja descriși mai sus, acestea au fost

declarate public ,iar campurile private fiind accesibile doar cu ajutorul getterelor si setterelor pentru a asigura permisiunile la aceste date.

Clasa Client:

```
package model;

public class Client {
    //Declararea campurilor
    private int id,arrivalTime,processTime;
    //Constructori pentru Client
    public Client()
    {
        this.id=0;
        this.arrivalTime=0;
        this.processTime=0;
    }
    public Client(int id,int arrival,int processTime)
    {
        this.id=id;
        this.arrivalTime=arrival;
        this.processTime=processTime;
    }
    //Gettere si settere pentru client
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public int getArrivalTime() {
        return arrivalTime;
    }
    public void setArrivalTime(int arrivalTime) {
        this.arrivalTime = arrivalTime;
    }
}
```

Prima data a fost specificat pachetul din care face parte clasa, ,câmpurile și constructori. Așa cum se poate vedea clasa Client contine informatii legate de id unic de identificare al fiecarui comparator, timpul cand a fost asezat la o coada si timpul de servire al acestuia. Campurile clasei care apar si in imagine au fost declarate sugetiv pentru o mai buna intelegere a codului.

Metodele care ne ofera posibilitatea să setăm sau să primim valoarea câmpurilor clasei "Client" ,apar si in imagine declarate in functie de modul lor de functionare si denumite cu get si respective set.

Clasa Queue:

Clasa Queue face parte din pachetul "model" impereună cu clasele Client și Scheduler, acesta pe langa campurile care retin timp de asteptare, numarul de client,

numarul casei si multe alte campuri contine o lista de client care reprezinta coada propriu-zisa la momentul simularii. De precizat este faptul ca aceasta clasa extinde clasa Thread, cu ajutorul careia putem crea mai multe procese ce ruleaza in acelasi timp.

Pe langă aceste gettere si settere mai avem și metodele care ne furnizeaza informatii legate de client si de bilantul dupa simulare.

```
// Functie care trimite o lista de clienti
public synchronized String convertClientsListToString()
{
    String listClients="";
    for(Client client:clients)
    {
        System.out.print(client.getId()+ " ");
        listClients=listClients+" "+client.getId();
    }
    System.out.println();
    return listClients;
}

//Returneaza numarul de clienti din lista
public synchronized int clientsNumber()
{
    return this.clients.size();
}

//Returneaza AverageTime pentru fiecare coada
public synchronized String showAverageTime(int i)
{
    double partialResult=(double)this.totalWaitingTime/this.numberOfClients;
    Double result=new Double(Math.round(partialResult*100)/100.0);
    return result.toString();
}
```

Toate metodele au fost sincronizate nu produce erori firelor de lucru nestiind ce valori au variabilele folosite de fiecare instanta.

În continuare voi spune cateva cuvinte despre metoda .Deoarece clasa "Queue" extinde clasa Thread este nevoie implementarea acestei metode pentru a face posibil lucrul cu mai multe fire de lucru. In aceasta am apelat metoda de extragere si aflarea informatiilor despre primul client id actstuaia ,dar si numarul de secunde in care coada trebuie sa-l proceseze. Pratic trebuie doar sa adormim threadul un numar de secunde egal

cu timpul de procesare.

```
// Metoda run specifica interfetei Runnable
public void run() {
    try {
        while(true)
        {
            //Luam informatiile legate de primul client din lista de clienti
            this.takeInformation();
            sleep(1000 * numberOfSecond);

            //Afisam un mesaj in LogEvent cum ca primul client a fost procesat
            this.view.addNewMessage("Clientul " + clientId + "a fost servit la casa " + this.id);
            this.waitingTime=this.waitingTime-this.numberOfSecond;

            //Stergem elementul si reactualizam simulator de cozi
            stergeClient();
            view.refreshSimulation(this.id, convertClientsListToString());
        }
    } catch (InterruptedException e) {
        System.out.println("Intrerupere");
        System.out.println(e.toString());
    }
}
```

Clasa Scheduler:

Clasa cea mai importanta, care rezolva problema asocierii unui client la o coda. Deoarece implementeaza interfata Runnable si acesta trebuie sa implementeze metoda run(). Unica diferenta intre implementarea interfetei Runnable sau extinderea clasei Thread este intatierea unui nou thread care dorim sa-l pornim.

Modul de functionare al acestei clase este urmatorul. Metoda run verifica daca nu cumva timpul de simulare s-a ,in cazul timoul actual este mai mic vom apela functia random care va genera numarul de client care vor fi generate in acest moment. Asupra problemei generari clientilor se ocupa functia addClient(). Generarea timpului de servire si al clientul vor avea loc in aceasta metoda la fel si asignarea la coada cu timpul de asteptare minim.

```

public void addClient(int clientID,int timeCounter)
{
    Random rand=new Random();
    //Generam timpul de servire
    int newClientServiceTime=rand.nextInt(this.maximumService-this.minimumService+1)+this.minimumService;
    Client client=new Client(clientID,timeCounter,newClientServiceTime);

    //Adaugam clientul creat in coada cea mai scurta
    int indexMinimumQueue=bestQueue();

    try
    {
        queues[indexMinimumQueue].adauga(client);
    }
    catch (InterruptedException e)
    {
        System.out.println(e.toString());
    }

    //Aflam un numarul de clienti de la momentul actual
    int newClientsNumber=this.totalNumberOfClients();
    if(newClientsNumber>this.maxNumberOfClients)
    {
        this.maxNumberOfClients=newClientsNumber;
        this.peakTime=timeCounter;
    }
}

```

Dupa cum se poate observa aceasta metoda creeaza un nou client si-l asigneaza cozi cu timpul cel mai scurt , numarul cozi va fi returnat de metoda bestQueue.

```

int bestQueue()
{
    int i=0;

    int min=queues[0].getWaitingTime();
    System.out.println("Minimul"+min+"Locat la pozitia"+i);
    //Parcurgem toate cozile pentru a determina coada cu waiting time-ul cel mai mic si vom returnat indecele ei
    for(int j=1;j<numberQueue;j++)
    {int min1=queues[j].getWaitingTime();
    if(min1<min&&min!=min1)
    {
        System.out.println("Minimul"+min1+"Indexul "+j);
        min=min1;
        i=j;
    }
    }
    return i;
}

```

Aceasta cauta printre elementele vectorului queues coada care are timpul de asteptare cel mai mic . Daca cumva se gaseste o coada care are timpul mai mic decat cel precedent se retine valoarea ,dar si pozitia acestei cozi. In final metoda va returna indexul cozi.

De afisarea rezultatelor in urma simulri se va ocupa showResult care va trimite clasei SimulationWindow un mesaj cu rezultatele cozi dar si indexul pentru a asocia raspunsul. Se va transmite media timpului de astptare al fiecărei cozi, dar si timpul in care au fost cei mai multi clienti si numarul acestora.

```
//Afisam rezultatele obtinute
public void showResults()
{
    //Afisam in SimulationWinfow
    //Afisam peak time
    view.resultShow(this.numberQueue,"Peak Time= "+this.peakTime+" a avut un numar de =" +this.maxNumberOfClients);

    for(int i=0;i<this.numberQueue;i++)
        view.resultShow(i,"AveraTime for queue"+(i+1)+" is:"+queues[i].showAverageTime(i));
}
```

Clasa View:

Cea mai importantă parte din view este constructorul care conține informații despre crearea interfeței grafice. Unele metode apelate în interiorul acestui constructor sunt doar câteva metode auxiliare create în interiorul acestei clase cu scopul de a crește înțelegerea codului.

```
package view;
import java.awt.*;

public class View extends JFrame{
    //Declararea componentelor vizuale
    private JTextField minArrive=new JTextField();
    private JTextField maxArrive=new JTextField();
    private JTextField minService=new JTextField();
    private JTextField maxService=new JTextField();
    private JTextField simulationTime=new JTextField();
    private JTextField numberQueue=new JTextField();
    private JTextField numberClients=new JTextField();
    private JTextArea logEvent=new JTextArea();
    private JScrollPane jsp;
    private JLabel ma=new JLabel("Minimumm arriving time");
    private JLabel mA=new JLabel("Maximum arrive time");
    private JLabel ms=new JLabel("Minimum service time");
    private JLabel mS=new JLabel("Maximum service time");
    private JLabel sm=new JLabel("SimulationTime");
    private JLabel nq=new JLabel("Number of Queue");
    private JLabel nc=new JLabel("Numebr of Clients");
    private JButton simulate=new JButton("Simulate");

    //Declararea panourilor
    private JPanel panel1=new JPanel();
    private JPanel panel2=new JPanel();

    //Declararea ferestrei care simuleaza in direct cozile
    private SimulationWindow simulationQueue;
```

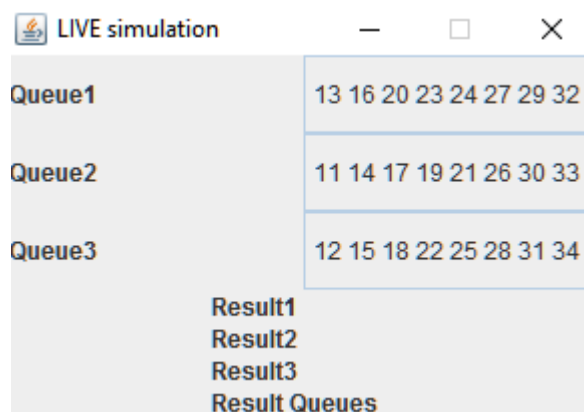
Ascultătorii pentru buton este :

```
//Ascultatori
public void addListenersSimulation(ActionListener a)
{
    simulate.addActionListener(a);
}
```

O implementare mai interesantă se poate găsi în clasa `SimulationWindow` care în funcție de numărul de cozi introdus de utilizator înaintea simulării va deschide o fereastră cu un număr egal de textfielduri și labeluri. Aici va fi posibilă urmărirea în timp real a procesării cozilor. De asemenea, după trecerea timpului de simulare, va fi vizualizarea rezultatelor care sunt determinate în urma simulării. Mărimea acelor ferestre este variabilă în funcție de numărul de cozi.

5. Rezultat

Rezultatul este vizibil în urma declanșării ascultătorului asociat butonului `simulate`, moment în care va apărea fereastra din imaginea de mai jos. Aceste rezultate depind doar de datele de intrare. Labelurile care conțin cuvântul "Result" se vor completa doar după terminarea simulării.



Pentru datele de intrare introduce în câmpuri vom obține client generat random la momente de timp alese de asemenea aleatoriu.

6. Concluzie

Chiar dacă această aplicație este o simulare a cozilor unui magazin pe viitor am putea încerca realizarea unei simulări care să determine coada cu timpul minim prin implementarea unor algoritmi mult mai eficienți care țin cont de mult mai mulți factori. S-ar mai putea implementa un graphic care să ilustreze timpul în care au fost cei mai mulți clienți și de asemenea introducerea unor animații care să reflecte mult mai bine clienți așezați la cozi.

7. Bibliografie

- <http://users.utcluj.ro/~igiosan/>
- http://inf.ucv.ro/documents/tudori/laborator8_53.pdf
- <https://stackoverflow.com/questions/40828704/how-to-compare-with-thread-state>
- <https://stackoverflow.com/questions/2560784/how-to-center-elements-in-the-boxlayout-using-center-of-the-element>
- <https://stackoverflow.com/questions/5908798/how-to-get-the-state-of-a-thread>
- <https://www.youtube.com/watch?v=PARFguh3ck8>