MINISTERUL EDUCAȚIEI ȘI CERCETĂRII ȘTIINȚIFICE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE

Simulare cozi

Documentație

Ciubotaru Andrei-Mihai

Grupa: 30227

Cuprins

1.Obiectivul temei

Obiectivul acestei teme este sa simulam si sa analizam un sistem de de procesare al cozilor. Un exemplu din lumea reala il poate reprezenta cozile de clienti de la un supermarket.

Aplicatia va dispunde de o interfata, astfel utilizatorul va alege datele de intrare, iar apoi va urmari simularea, interfata se va actualiza in timp real astfel vom putea vedea la ce moment si la ce coada a intrat un client si cand a iesit.

2.Analiza problemei

Pentru inceput trebuie sa stim ce e aceea o coada. Coada se bazeaza pe principiul FIFO(first in first out) astfel primul client care se va aseza la coada va fi primul procesat. Pentru a putea implementa mai multe cozi care sa isi desfasoara activitatea in acelasi timp vom folosi thread-urile.

Utilizatorul introduce de la tastatura date pentru a putea rula aplicatia. Aceste date sun reperezentate de numarul de cozi, am ales sa ma rezum la maxim 7 cozi, timpul minim si maxim de sosire dintre clienti, timpul minim si maxim de servire, strategia dupa care se vor aseza clientii la coada si intervalul de simulare.

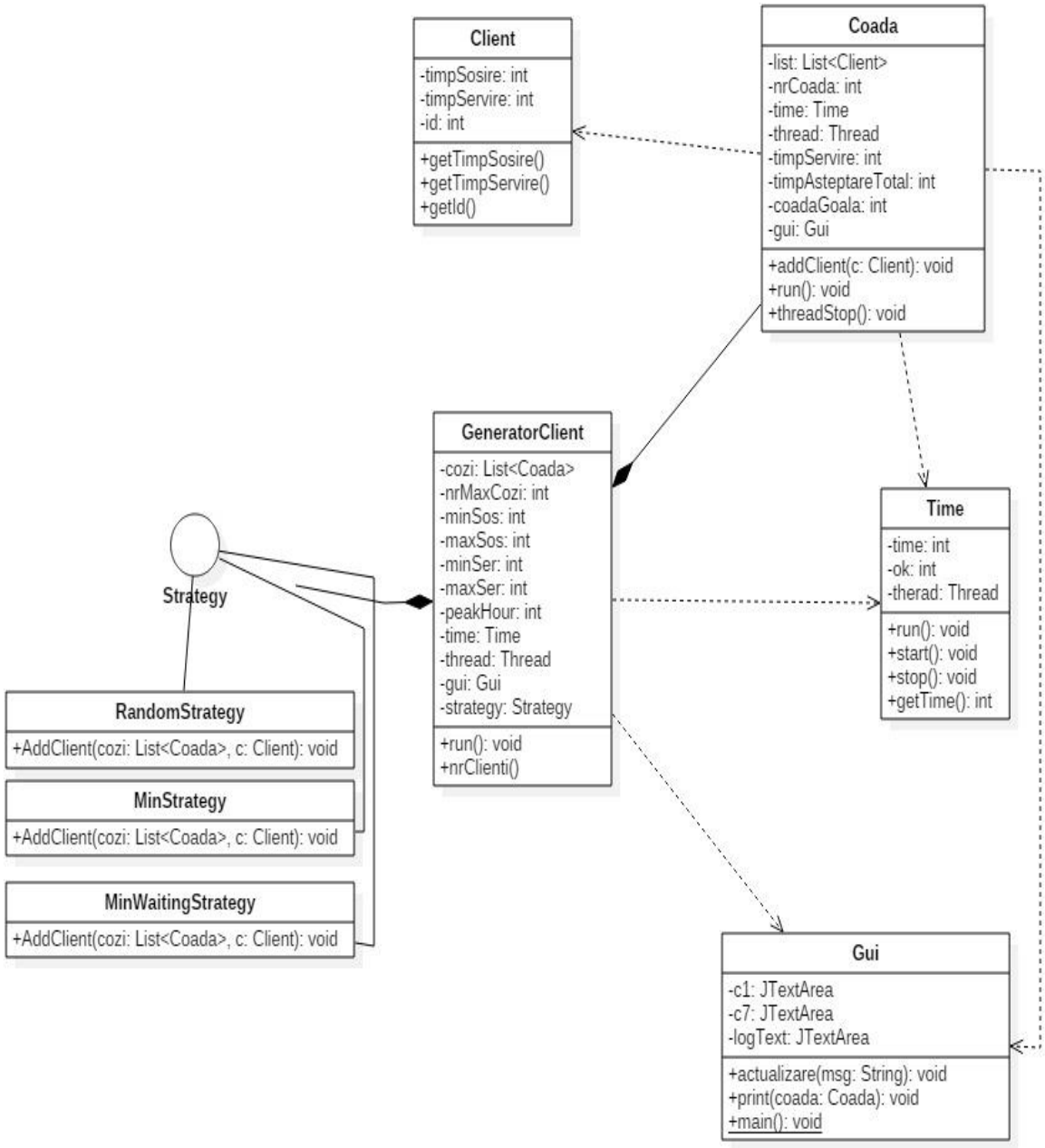
La sfaristul simularii vor fi afisate pentru fiecare coada timpul mediu de asteptare si de servire, timpul cat coada a fost goala si momentul in care la cozi au fost cei mai multi clienti.

3.Proiectare

Fiecare coada va avea propriul thread. Am ales sa mai folosesc un thread pentru a genera clientii si inca unul pentru a putea stii la ce moment al simularii ma aflu. Fiecare client are un timp de sosire si de asteptare generat random intere ce le doua limite introduse de utilizator. Clientul va fi asezat la coada in functie de startegia aleasa. Am ales sa implementez 3 strategii: strategia „Random”, clientul este plasat aleatoriu la o casa, strategia „Min_Clienti”, urmatorul clientul este plasat la coada la care se afla cei mai putini clienti si strategia „Min_Servire”, urmatorul client asezandu-se la coada la care timpul de servire e cel mai mic(practic se va aseza la coada la care sunt cele mai putine produse). Clientii vor fi generati atat timp cat nu este depasita durata de simulare. De exemplu daca ne aflam la momentul 96 si umeaza sa intre un client la momentu 102, iar durata de simulare este 100, acesta nu va mi fi adugat si simularea se va termina.

Diagrama UML presupune modelarea unui sistem prin lucrurile care sunt importante pentru acesta. Aceste lucruri sunt modelate folosind clase.

3.1.Diagrama de clase



4.Implementare

4.1.Clase si pachete

Am organizat proiectul in trei pachete: Coada, Manager si Gui.

In pachetul Coada se afla clasele Client si Coada. Clasa client are ca si atribute timpul de sosire , timpul de servire si id. Clasa Coada implementeaza interfata Runnable deoarece fiecare coada va rula pe un thread separat. Aceste clase reperzinta baza acestei aplicatii.

O clasa foarte importanta este clasa GeneratorClient care are rolul de a genera clienti si de ai trimite la o anumita coada. Aceasta clasa se afla in pachetul Manager alaturi de clasa Time, interfata Strategy, si de clasele care implementeaza aceasta interfata RandomStrategy, MinStrategy si MinWaitingStrategy. Am folosit interfata Strategy pentru a putea avea mai multe startegii de plasare a unui client la o coada.

Clasa Time genereaza timpul simularii si datorita ei putem afla la ce moment al simularii ne aflam. Aceata are ca variabile instanta o variabila time care va reperezenta timpul si un thread. Acest thread va porni la inceputul executiei aplicatiei si va rula pana la sfarsit. Cu ajutorul metodei getTime() vom stii la ce moment al simularii ne aflam.

```
public class Time implements Runnable{
    private int time;
    private Thread thread;
    private int ok;
    public Time() {
        time=0;
        thread= new Thread(this);
    }
    public void setTime(int time) {
        this.time = time;
    }
    public int getTime() {
        return time;
    }
    public void start() {
        ok=1;
        thread.start();
    }
    public void stop(){
        ok=0;
    }
    @Override
    public void run() {
        while(ok==1) {
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            time++;
        }
```

Clasa GeneratorClient are rolul de a genera clienti random. Acesta clasa are ca variabile instanta minSos, maxSos, minSer, maxSer reperzentand limitele pentru generarea timpului de sosire si de servire. nrMaxCozi reprezinta numarul de cozi. Lista de cozi este reprezentata de List<Coada> cozi. Am ales sa implementez aceata lista ca ArrayList cu o exceptie, acesta lista trebuie sa fie *synchronizedList* deoarece avem mai multe thred-uri si doua thread-uri pot accesa lista in acelasi timp, spre exemplu este generat un lient care intra la coada la timpul 21 dar la acelasi timp un alt client iese.

(cozi= Collections.*synchronizedList*(new ArrayList<Coada>();)

Durata reprezinta intervalul de simulare, iar peakHour timpul la care la cozi se afla cei mai multi clienti.

```
this.nrMaxCozi=nrMaxCozi;
this.minSos=minSos;
this.maxSos=maxSos;
this.minSer=minSer;
this.maxSer=maxSer;
this.durata=durata;
peakHour=0;
if(strategie.equals("Random"))
    this.strategy=new RandomStrategy();
else if(strategie.equals("Min_Clienti"))
    this.strategy=new MinStrategy();
else if(strategie.equals("Min_Servire"))
    this.strategy=new MinWaitingStrategy();
cozi= Collections.synchronizedList(new ArrayList<Coada>());
time=new Time();
int i;
for(i=1;i<=nrMaxCozi;i++) {
    Coada coada=new Coada(i,time,gui);
    cozi.add(coada);
}
this.gui=gui;
thread = new Thread(this);
thread.start();
```

In constructor sunt initalizate toate variabilele instanta si in functie de nrMaxCozi este creata lista de cozi. Cozile vor fi numerotate de la 1 la nrMaxCozi. In functie de strategia introdusa de utilizator atributul strategy va executa o anumita metoda.

```
public class RandomStrategy implements Strategy {

    @Override
    public void addClient(List<Coada> cozi, Client c) {
        int nrCozi = cozi.size();
        Random rand = new Random();
        int nr = rand.nextInt(nrCozi);
        cozi.get(nr).addClient(c);
    }
}
```

Aceasta clasa implemeteaza interfata Strategy si suprascrie metoda addClient. Astfel se genereaza un nr random intre 0 si nrCozi-1 ce reprezinta nr cozii, iar clientul c va fi adugat la coada respectiva.

```
public class MinStrategy implements Strategy {

    @Override
    public void addClient(List<Coada> cozi, Client c) {
        int min=cozi.get(0).list.size();
        int nr=0;
        int j=0;
        for (Coada i : cozi) {
            if(i.list.size()<min) {
                min=i.list.size();
                nr=j;
            }
            j++;
        }
        cozi.get(nr).addClient(c);
    }

}
```

Pentru a adauga clientul la coada cu cei mai putini clienti am salvat numarul de clienti de la prima coada intr-o variabila numita min, iar apoi am comparat cu numarul clientilor de la celelalte cozi si am salvat nr cozii. La sfarsit adaug clientul la coada cu indexul nr.

```
public class MinWaitingStrategy implements Strategy {
    @Override
    public void addClient(List<Coada> cozi, Client c) {
        float min=cozi.get(0).getTimpServire();
        int nr=0;
        int j=0;
        for (Coada i : cozi) {
            if(i.getTimpServire()<min) {
                min=i.getTimpServire();
                nr=j;
            }
            j++;
        }
        cozi.get(nr).addClient(c);
    }
}
```

Pentru a adaugarea la coada in functie de timpul de servire al celorlalti clienti algoritmul este asemanator cu cel de la adugare in functie de nr clientilor doar ca de acesta data comparam timpul de servire de la fiecare coada.

Tot in costructor pornim thread-ul.

```
public void run() {
    Random rand = new Random();
    int max=0;
    int id=0;
    time.start();
    while(time.getTime()<durata) {
        int sos=minSos + rand.nextInt(maxSos-minSos+1);
        int ser=minSer + rand.nextInt(maxSer-minSer+1);
        if(time.getTime()+sos>durata)
            break;
        id++;
        try {
            Thread.sleep(sos*500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        Client c=new Client(sos,ser,id);
        strategy.addClient(this.cozi,c);
```

```
            if(max<nrClienti()){
                max=nrClienti();
                peakHour=time.getTime();
            }
        }
        gui.actualizare("STOP");
        for (Coada i: cozi)
            i.threadStop();
        time.stop();
        gui.actualizare("Peak Hour: " + peakHour);
    }
```

In metoda run este prezentat modul in cere sunt generati clientii si mometul in care sunt introdusi in coada in functie de timpul de sosire. La inceput este pronit thread-ul care generaza timpul simulrii. In while verificam daca timpul la care ne alfam plus timpul de sosrie al urmatorului client nu depaseste durata simularii, daca aceasta conditie este indeplinita thread-ul va fi pus sa atstepte un timp egal cu timpul de sosire apoi clientul va fi introdus in coada coerspunzatoare. Am ales ca timpul de simulare sa fie de doua ori mai rapid decat cel de introdus de utilizator, astfel simularea se va termina mai repede, de aceea thread-ul ateapta sos*500 milisecunde.

Pentru a afla timpul la care la cozi se afla cei mai multi clienti am folosit o functie care imi calculeaza numarul total de clienti.

```
public int nrClienti() {
    int nr_total=0;
    for(Coada i: cozi) {
        nr_total+=i.list.size();
    }
    return nr_total;
}
```

Daca functia returneaza un numar mai mare decat max, peakHour ia valoarea timpului de la acel moment, iar max se actualizeaza.

La sfarsitul simularii opresc thread-urile de la fiecare coada, astfel clientii nu vor mai fi procesati. De asemena actualizez interfata.

Clasa Coada are ca atribute o lista de clienti, numarul cozii, timpul de servire, timpul total de asteptare , timpul in care coada este goala, nr de clienti serviti si instante ale claselor Time, Thread si Gui.

```
public void addClient(Client c) {
    list.add(c);
    nrClienti++;
    gui.actualizare("->Client id " + c.getId() + " sosit la coada " + nrCoada + "
la timpul " + time.getTime() +
                    "          timp sosire: " + c.getTimpSosire() +  " timp servire: "
+ c.getTimpServire());
    gui.print(this);
}
```

Metoda addClient adauga un client in lista de clienti si actualizaeaza interfata. In momentul in care intra un client nou la coada, in log se va afia „-> Client id X sosit la casa X la timpul X”. Timpul este preluat din clasa Time. Pentru

a putea urmari mai usor simularea si a vedea daca este ok am ales sa afisez si timpul de sosire si de servire. Metoda gui.print() are rolul de a afisa evolutia cozii.

```
public void run() {
    int lastSevTime=0;
    int id=0;
    int timpAsteptare=0;
    while(ok || list.size()>0) {
        if(list.size()!=0) {
            timp+=list.get(0).getTimpServire();
            if(lastSevTime-list.get(list.size()-1).getTimpSosire()>0) {
                timpAsteptare += lastSevTime - list.get(list.size() -
1).getTimpSosire();
                timpAsteptareTotal += timpAsteptare;
            }
            else timpAsteptare=0;
            lastSevTime=list.get(list.size()-1).getTimpServire();
            try {
                Thread.sleep(list.get(0).getTimpServire() * 500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            timpServire+=list.get(0).getTimpServire();
            nrClientiSer++;
            id=list.get(0).getId();
            list.remove(0);
            gui.actualizare("<-Client id " + id + " iesit de la coada " + nrCoada +
" la timpul " + timp);
            gui.print(this);
        }
        else {
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            timp=time.getTime();
            coadaGoala++;
        }
    }
}
```

In aceasta metoda sunt procesati clientii unei cozi.

Cat timp nu este depasita durata simularii si coada nu e vida (daca ok este 0 inseamna ca threadul a fost oprit) thread-ul este pus sa astepte un timp egal cu timpul de servire al clietului din varful cozii, iar apoi este scos din coada respectiv din lista de clienti a acelei cozi. Altfel daca lungimea listei este 0 la coada nu sunt clienti, thread-ul este pus sa astepte cate o secunda, iar variabila coadaGoala, care reprezinta cat timp coada a fost goala, creste.

Atunci cand un client paraseste coada in log se afiseaza „<-Client id X iesit de la coada X la timpul X”. Timpul este calculat cu ajutorul variabilei timp care reprezinta timpul la care a venit primul client la coada, iar timpul la care iese este dat de timpul de intrare plus timpul de servire ala fiecarui client care iese. De asemenea se actualizeaza evolutia cozii.

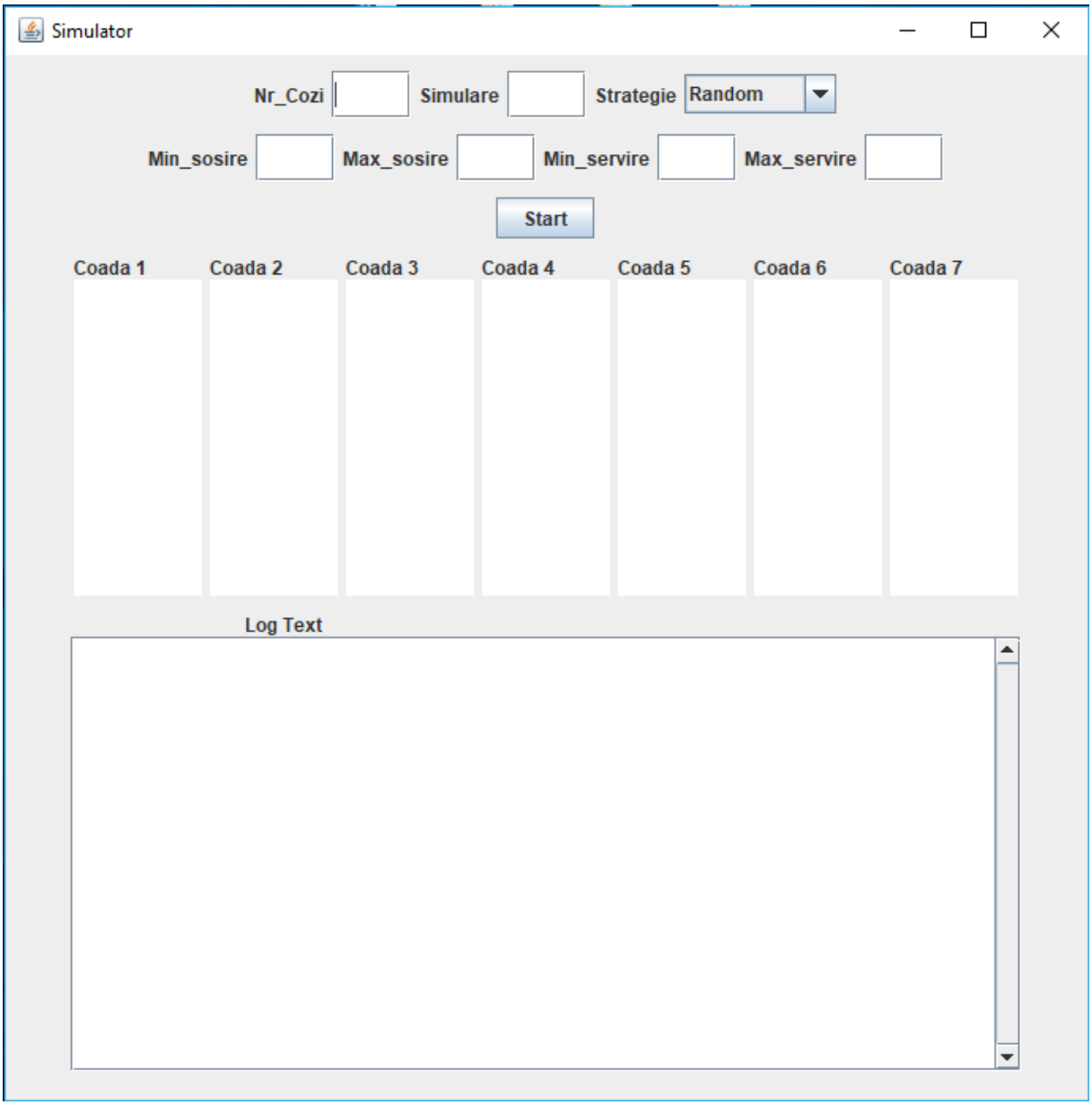
Pentru a calcula timpul mediu de servire am adunat la variabila timpServire timpul de servire al fiecarui calient care a iesit de la coada si am impartit la nummarul de clienti serviti.

Pentru a calcula timpul mediu de asteptare, in variabila timpAsteptare am calculat cat timp asteapta un client fata de clientul din fata lui(=timpul de servire al primului client – timpul de sosire al urmatorului client). Daca la coada nu se afla niciun client atunci timpAsteptare este 0, altfel se aduna cu timpul de astepate al clientului curent. Timpul total de asteptare este suma timpilor de astepare al fiecarui client. Iar media este timpul total impartit la nr de clienti serviti.

```
        public void threadStop() {
    ok=false;
    thread.stop();
    gui.actualizare("Coada " + nrCoada + ":\n" + "Average service time: " +
timpServire/nrClientiSer +
            " Average waiting time: " + timpAsteptareTotal/nrClientiSer + " Empty
queue: " + (coadaGoala-1));
}
```

Metoda threadStop() opreste thread-ul curent si se actualizeaza interfata cu informatiile specifice acelei cozi.

4.2.GUI



Interfata este alcatuita din campurile destinate introducerii datelor de simulare, un buton, o sectiune in care este prezentata evolutia cozilor si un log text.

Pentru log am folosit o variablia de tipul JtextArea.

```
JPanel panel3 = new JPanel();
JLabel l5 = new JLabel("Log Text");
logText = new JTextArea(17,53);
logText.setLineWrap(true);
logText.setWrapStyleWord(true);
logText.setEditable(false);
JScrollPane scroll = new JScrollPane(logText);
scroll.setVerticalScrollBarPolicy (ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
panel3.add(l5);
panel3.add(scroll);
panel3.setLayout(new BoxLayout(panel3,BoxLayout.Y_AXIS));
```

Log-ul este actualizat cu ajutorul metodei append(). Astfel la continutul log-ului se adauga mesajul transmis ca parametru.

```
public void actualizare(String msg) {
    logText.append(msg+"\n");
}
```

Pentru reprezentarea cozilor am folosit tot variabile de tipul JtextArea.

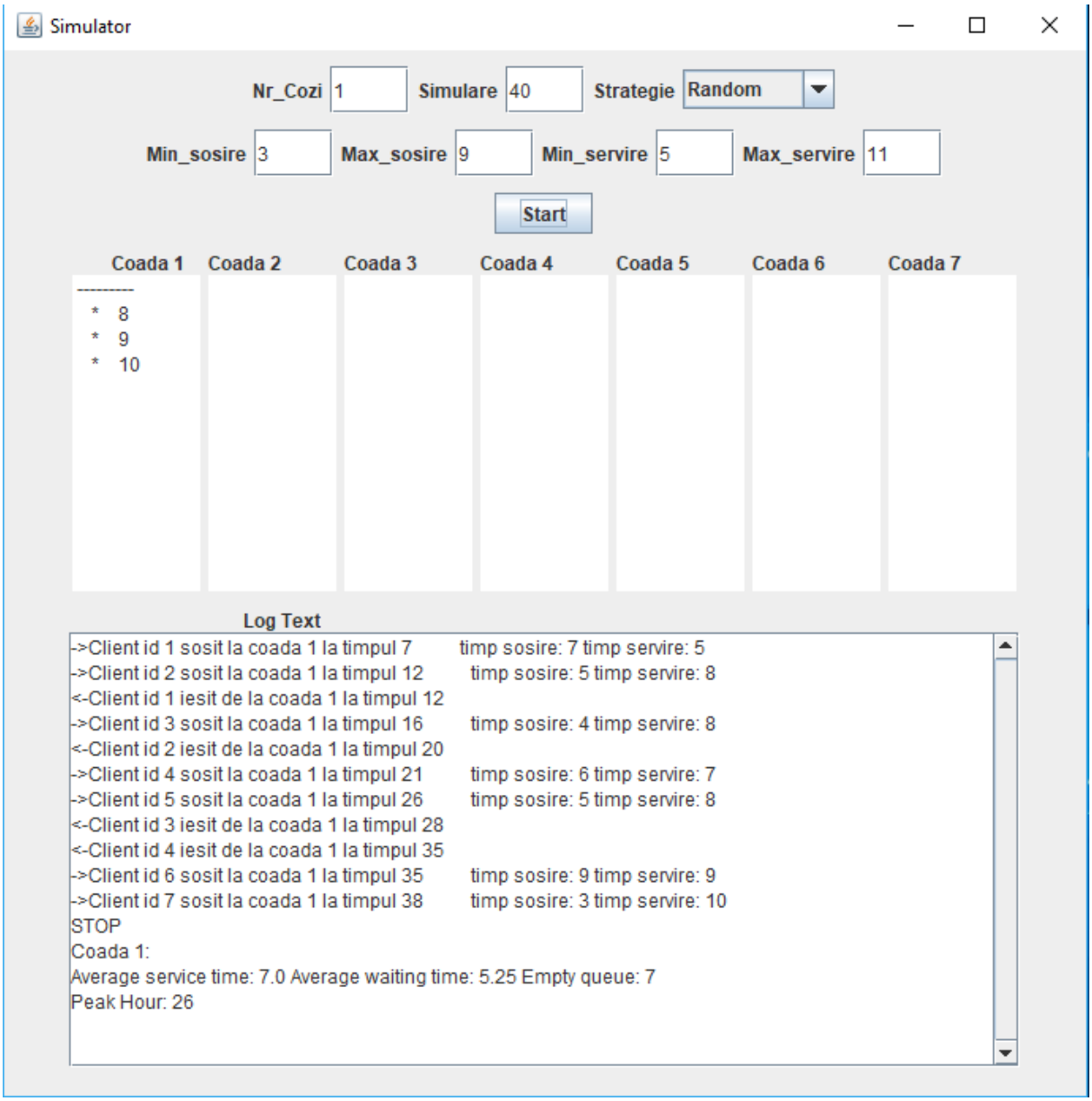
```
public void print(Coada coada) {
    if(coada.getNrCoada()==1) {
        c1.setText("");
        c1.append(" ---------\n");
        for (int i = 0; i < coada.list.size(); i++) {
            c1.append("    *    " + coada.list.get(i).getTimpServire() + "\n");
        }
    }
    else if(coada.getNrCoada()==2) {
        c2.setText("");
        c2.append(" ---------\n");
        for (int i = 0; i < coada.list.size(); i++) {
            c2.append("    *    " + coada.list.get(i).getTimpServire() + "\n");
        }
    }
```

Metoda print() primeste lisata de cozi si in functie de nr cozilor actualizeaza text field-ul corespunzator. Casa este reprezentata de „---------” ,iar clientii de * plus timpul de servire.

5.Testare

Pentru a testa aplicatia, utilizatorul introduce datele si apasa butonul „Start” si va vedea evoluita cozilor si informatiile furnizate de log.

6.Rezultate



Se poate vedea evolutia cozii si informatiile furnizate de log.

7.Concluzii

Aplicatia poate fi dezvoltata adaugand mai multe statistici.

Am invatat sa folosesc multi-threading, si sa fac o interfata care se actualizeaza in timp real.

8.Bibliografie

www.stackoverflow.com