MINISTERUL EDUCAȚIEI ȘI CERCETĂRII ȘTIINȚIFICE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
CATEDRA CALCULATOARE

Simulare Cozi

Documentație

Şeican Lucian Alexandru

Grupa 30221

Cuprins

1. Obiective

În cadrul acestei teme principala cerinţă este realizarea unui program care simulează funcţionarea unui număr de cozi şi distribuirea clienţilor între aceste cozi. În cadrul acestei teme este necesară utilizarea de threaduri şi a conceptului de multithreading.

Obiectivul acestei teme este proiectarea şi implementarea simulatorului de cozi conform cerinţei enunţate mai sus. Acesta va prezenta o interfaţă grafică utilizată pentru culegerea datelor şi pentru afişarea cozilor şi evoluţiei acestora în timp real. Acest program va trebui de asemenea să păstreze şi un log al evenimentelor din timpul rulării programului.

În cadrul implementarii acestui proiect se va ţine seama de utilizarea conceptelor programării orientate pe obiecte în vederea implementării proiectului cerut. Astfel se va urmări utilizarea de clase şi tehnici/operaţii specifice OOP , precum şi respectarea convenţiilor de nume Java. Toate acestea au ca scop utilizarea la nivel maxim a capabilităţilor programării orientate pe obiect în vederea aprofundării cunoştiinţelor legate de aceasta.

2. Analiza Problemei

Pentru realizarea simulării cozilor vom utiliza conceptele de threaduri si multithreading. Un thread sau fir de execuţie defineşte cea mai mică unitate de procesare ce poate fi programată spre execuţie de către sistemul de operare. Threadurile sunt folosite pentru a eficientiza execuţia programelor , executând porţiuni distincte de cod în paralel în interiorul aceluiaşi proces.

În cadrul ecestei teme threadurile vor fi folosite pentru simularea cozilor. Astfel, fiecare coadă va avea propriul thread care va realiza operaţiile specifice fiecărei cozi precum preluarea, procesarea şi ştergerea clienţilor. Pe lângă threadurile corespunzătoare cozilor va fi necesară utilizarea unui thread care va avea funcţionalitatea unui planificator de activiăţi. Acesta va distribui clienţii spre cozi conform unei strategii de distribuţie(la realizarea acestei teme am ales utilizarea strategiei de distribuţie bazată pe numărul de clienţi a fiecărei cozi, astfel fiecare client va fi trimis la coada cu cel mai mic număr de clienţi).

Fiecare client este caracterizat de un timp de sosire şi un timp de procesare. Timpul de sosire reprezintă intervalul de timp dintre începerea simulării şi momentul în care clientul va merge la coadă. Timpul de procesare reprezintă durata necesară pentru un client să fie procesat şi apoi să părăsească coada din momentul în care i-a venit rândul la o coadă.

Aceste threaduri destinate cozilor respectiv planificatorului de activităţi vor trebui să se sincronizeze şi să lucreze în paralel în mod corepunzător pentru realizarea operaţiilor cerute. Astfel, în momentul în care un thread al unei cozi va fi gol el va trebui sa astepte dupa threadul planificator sa îi transmită un client pentru a îşi relua funcţionarea.

Pe lângă simularea cozilor programul va ţine şi un log al evenimentelor. În această vedere am ales să utilizez un fişier text numit “log.txt” pentru păstrarea logului. Programul va prelua numărul clienţilor de la utilizator şi va realiza o simulare pe 3 cozi şi pe un interval de timp prestabilit..

Cazuri de utilizare:

- Nu se introduce numărul de clienţi şi se porneşte simularea: programul va arunca o excepţie şi va înceta execuţia
- Se introduce un număr de clienţi şi se porneşte simularea : programul va funcţiona în mod normal(chiar dacă numărul introdus este negativ se va comporta ca şi în cazul în care nu ar exista niciun client).

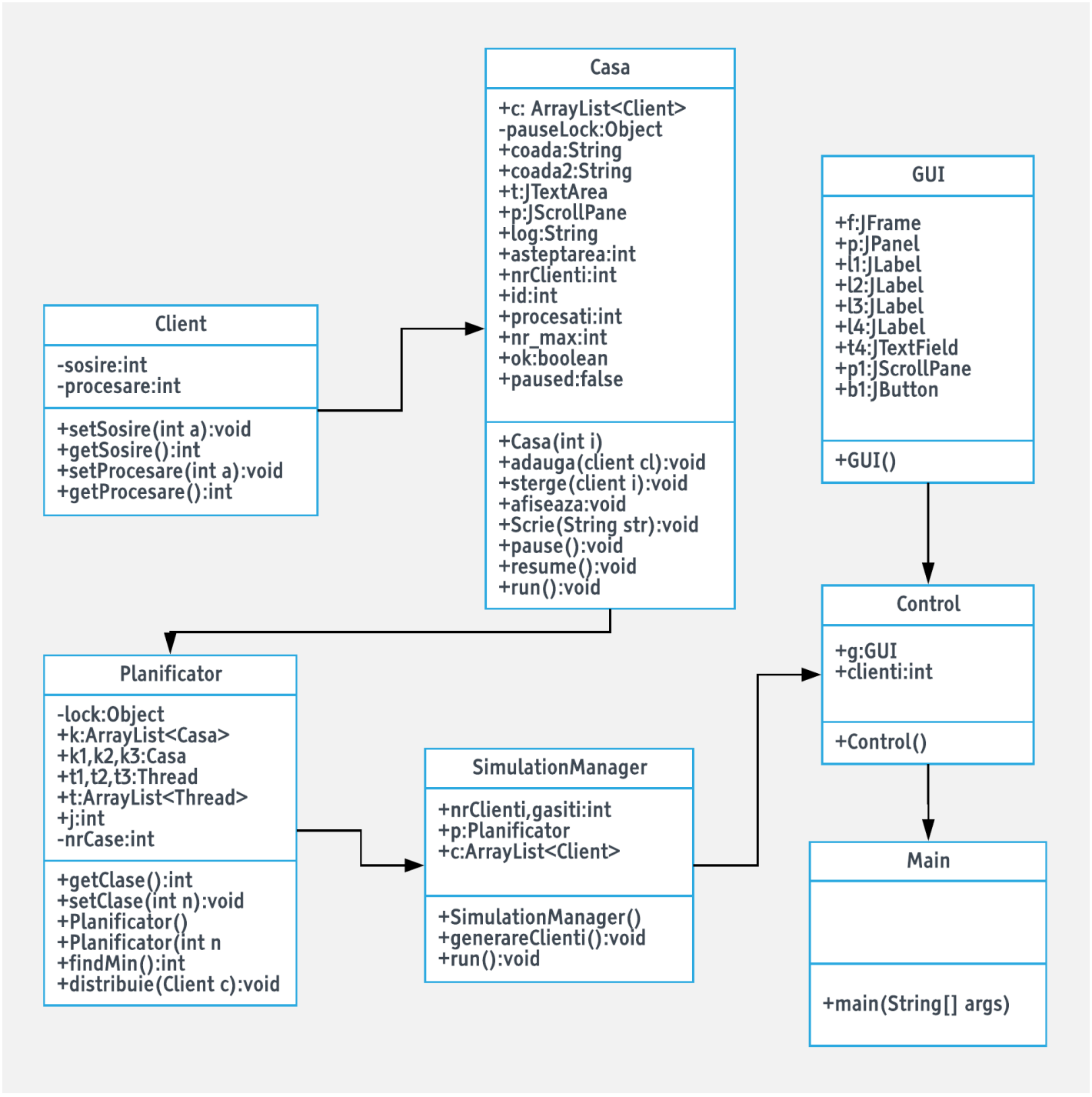
Asumpţii: vom presupune că simularea se realizează pe un număr prestabilit de cozi(în acest caz pe 3 cozi).

3. Proiectare

În faza de proiectare a acestei teme am ales să structurez programul în următoarele clase:

- Casa
- Client
- Control
- GUI
- Main
- Planificator
- SimulationManager

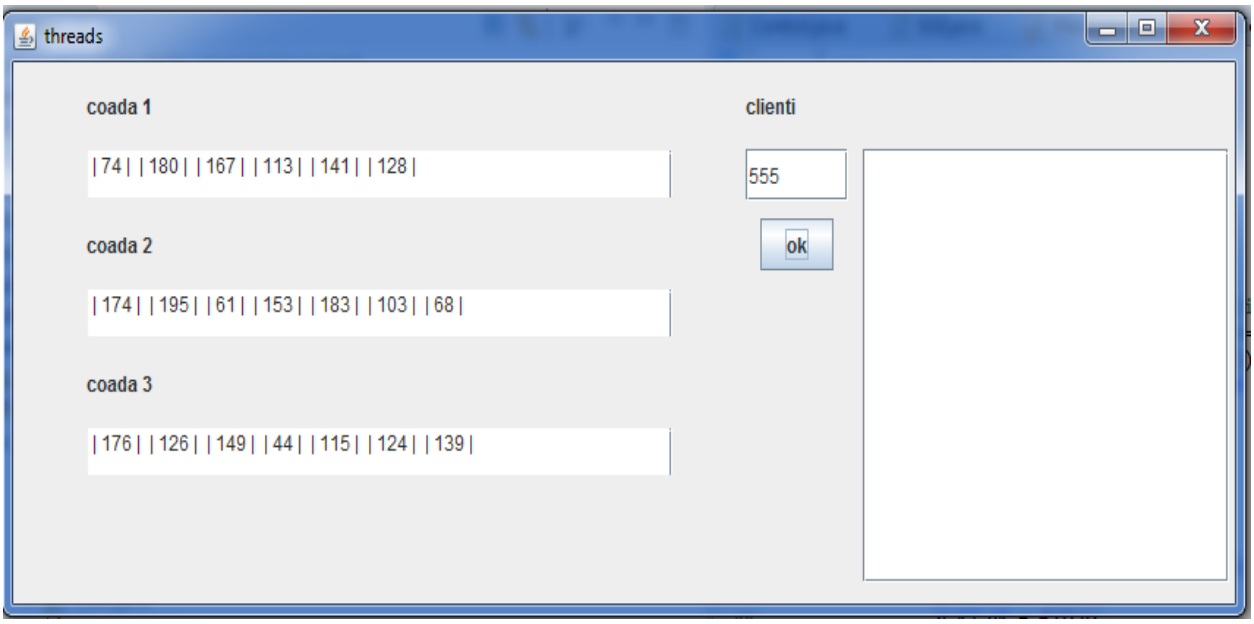
Aceste clase sunt structurate comform diagramei UML următoare:


Casa
+c: ArrayList<Client>
-pauseLock:Object
+coada:String
+coada2:String
+t:JTextArea
+p:JScrollPane
+log:String
+asteptarea:int
+nrClienti:int
+id:int
+procesati:int
+nr_max:int
+ok:boolean
+paused:false
+Casa(int i)
+adauga(client cl):void
+sterge(client i):void
+afiseaza:void
+Scrie(String str):void
+pause():void
+resume():void
+run():void
GUI
+f:JFrame
+p:JPanel
+l1:JLabel
+l2:JLabel
+l3:JLabel
+l4:JLabel
+t4:JTextField
+p1:JScrollPane
+b1:JButton
+GUI()
Client
-sosire:int
-procesare:int
+setSosire(int a):void
+getSosire():int
+setProcesare(int a):void
+getProcesare():int
Control
+g:GUI
+clienti:int
+Control()
Planificator
-lock:Object
+k:ArrayList<Casa>
+k1,k2,k3:Casa
+t1,t2,t3:Thread
+t:ArrayList<Thread>
+j:int
-nrCase:int
+getClase():int
+setClase(int n):void
+Planificator()
+Planificator(int n
+findMin():int
+distribuie(Client c):void
SimulationManager
+nrClienti,gasiti:int
+p:Planificator
+c:ArrayList<Client>
+SimulationManager()
+generareClienti():void
+run():void
Main
+main(String[] args)

Pentru proiectarea acestui program am ales să utilizez clasa Client care conţine timpul de sosire şi timpul de procesare ca şi informaţii caracteristice fiecărui client. Clasa casă conţine o listă de clienţi. Astfel, fiecare casă are un grup de clienţi pe care ii va procesa.

Clasa planificator conţine casele care urmează a fi utilizate pentru simulare pe care le iniţializează şi crează thread-uri pentru fiecare. De asemenea se ocupă cu distribuţia clienţilor la case.Clasa SimulationManager se ocupă de generarea clienţilor şi de rularea simulării propriu-zise.

Clasa GUI conţine casetele text,etichetele,frame-urile, panourile si celelalte elemente necesare realizării interfeţei grafice:



4. Implementare

În cadrul programului dezvoltat am utilizat următoarele clase cu metodele şi variabilele aferente:

- **Clasa Client**: conţine variabilele caracteristice unui client: timpul de sosire şi timpul de procesare.

```
private int sosire;
private int procesare;
```

De asemenea clasa Client mai conţine şi metode getteri şi setteri pentru timpul de sosire şi timpul de procesare şi un constructor care dă valori random timpilor de sosire şi de procesare:

```
public void setProcesare(int a);
public int getProcesare();
public void setSosire(int a);
public int getSosire();
```

- **Clasa Casa**: conţine metodele:

Adauga: metodă care adaugă un nou client la casa , apoi afişează toţi clienţtii de la casa respectivă şi scrie în log:

```
public void adauga(Client cl) {
	if(this.nrClienti<=this.nr_max) {
	c.add(cl);
	asteptare = asteptare + cl.getProcesare();
	this.nrClienti++;
	afiseaza();
	log = "";
	log = log + "clientul cu timpul de procesare " + cl.getProcesare() + "
	s-a asezat la coada " + this.id + "\n";
	try {
		Scrie(log);
	} catch (IOException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	}
	}
	else { System.out.println("coada este plina");}
		}
```

Sterge: metodă care şterge un client din lista de clienţi ai casei şi afişează toţi clienţii rămaşi:

```
public void sterge(Client i) {
	this.c.remove(i);
	asteptare = asteptare - i.getProcesare();
	this.nrClienti--;
	procesati++;
	afiseaza();
	log = "";
	log = log + "clientul cu timpul de procesare " + i.getProcesare() + " a
	iesit din coada " + this.id + "\n";
	try {
		Scrie(log);
	} catch (IOException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	}

		}
```

Metoda afiseaza care afişează toţi clienţii prezenţi la casă la un moment dat, metoda Scrie utilizată pentru a scrie logul intr-un fişier text şi metodele pause şi resume folosite pentru pauzarea şi reluarea rulării threadului casei.

Suprascrierea metodei run care realizează procesarea clienţilor şi stoparea respetiv reluarea activităţii casei.

```
public void run() {
		while(ok){
			try {
				if(this.nrClienti>0) {
				Thread.sleep(c.get(0).getProcesare());

				this.sterge(c.get(0));}
				else {
					synchronized(pauseLock) {
						if (paused) {
					try {
					    pauseLock.wait();
					} catch (InterruptedException ex) {
					    break;
					}
				}
			}
			}
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			if(this.nrClienti == 0) {
			System.out.println("-----"+ this.id+ "-----");
```

```
                paused = true;}
                }
```

- **Clasa Planificator** conţine metode constructor care crează un anumit număr de case si thread-uri corespunzătoare acestora
  Metodele importante ale acestei clase sunte metodele care se ocupă cu distribuţia clienţilor la case astfel:

- Metoda findMin găseşte casa cu numărul minim de clienţi

```
public int findMin() {
            int min = this.k.get(0).nrClienti;
            for(int i = 1;i<this.getCase();i++) {
                  if(this.k.get(i).nrClienti<min) {
                        min = this.k.get(i).nrClienti;
                  }
            }
            return min;
}
```

- Metoda distribuie care trimite clienţii la casa determinată de către metoda findMin

```
public void distribuie(Client c) {
            for(int i = 0;i<this.getCase();i++) {
                  int nr = this.findMin();
                  //System.out.println(nr);
                  if(this.k.get(i).nrClienti == nr) {
                        this.k.get(i).adauga(c);
                        if(this.k.get(i).paused == true) {
                        this.k.get(i).resume();}
                        return;
                  }
            }
      }
```

- **Clasa GUI:** este clasa care implementează interfaţa vizuală a programului şi conţine iniţializarea tuturor elementelor grafice necesare pentru realizarea interfeţei precum şi un constructor care setează dimensiunile şi locaţiile acestor elemente şi le face vizibile utilizatorului.
- **Clasa Simulation Manager:** se ocupă de generarea clienţilor prin intermediul metodei generareClienţi şi de realizarea simulării propriu-zise:

```
public void generareClienti() {
            for(int i = 0;i<nrClienti;i++) {
                  Client ct = new Client();
                  this.c.add(ct);
            }
      }
```

În vederea realizării simulării de cozi se suprascrie metoda run în interiorul căreia se pornesc thread-urile caselor, se simulează trecerea timpului prin intermediul variabilei time şi a metodei sleep() care se apelează la fiecare iteraţie a acestei variabile. De asemenea se verifică şi dacă toţi clienţii existenţi au fost procesaţi, caz în care se opreşte execuţia programului.

```
public void run() {
	int time = 0;
	this.generareClienti();
	p.t1.start();
	p.t2.start();
	p.t3.start();

	while(time <= 300) {
		for(int i = 0;i<nrClienti;i++) {
			if(this.c.get(i).getSosire() == time) {
				p.distribuie(this.c.get(i));
				System.out.println("client nou");

				gasiti++;
			}
		}
	if(p.k1.procesati + p.k2.procesati + p.k3.procesati == nrClienti) {
			p.k1.ok = false;
			p.k2.ok = false;
			p.k3.ok = false;
		}
		time++;
		try {
			Thread.sleep(1000);
		} catch (InterruptedException e) {

			e.printStackTrace();
		}
	  }
	}

	}
```

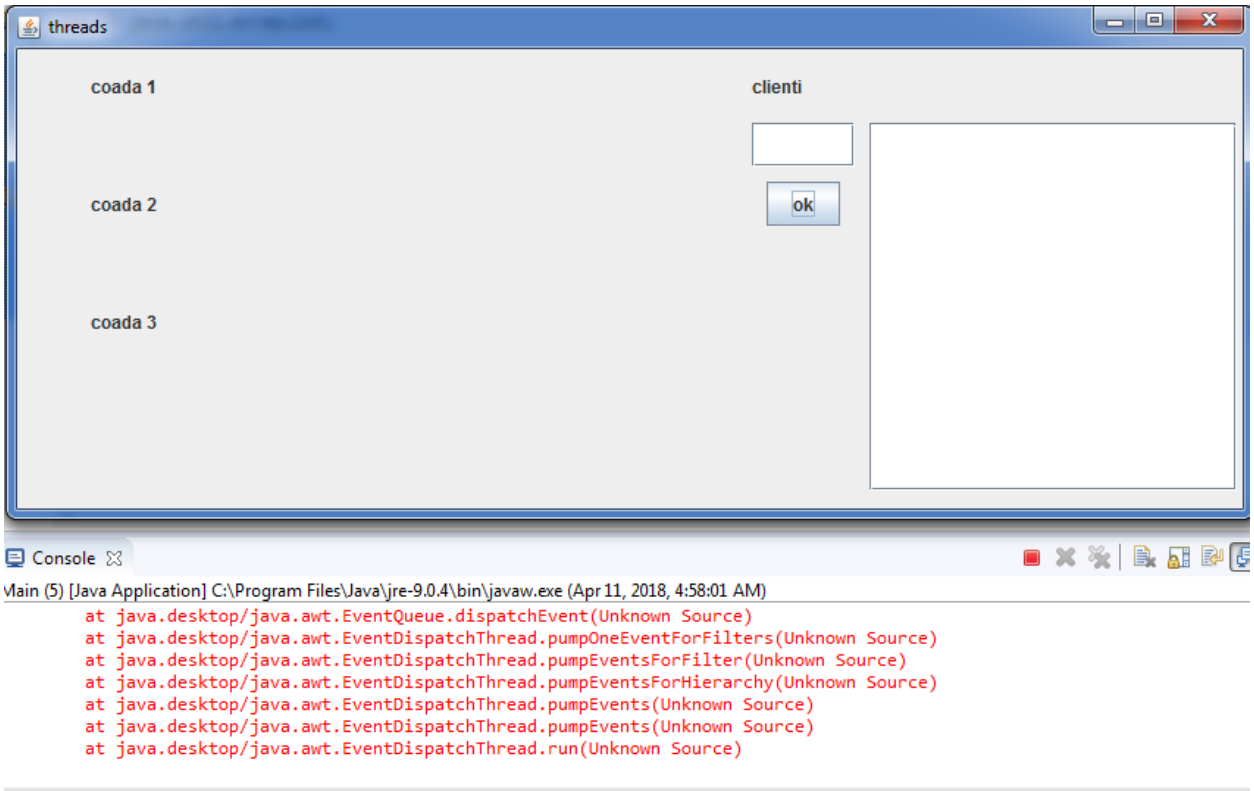
- **Clasa planificator**: conţine un obiect de tip GUI şi se ocupă cu preluarea datelor de la utilizator prin intermediul interfeţei grafice, iniţializarea unui obiect din clasa SimulationManager şi a unui thread dedicat acestuia.

Clasa Main: instanţiază un obiect din clasa Control:

```
public static void main(String[] args) throws InterruptedException {

Control c = new Control();

	}

}
```

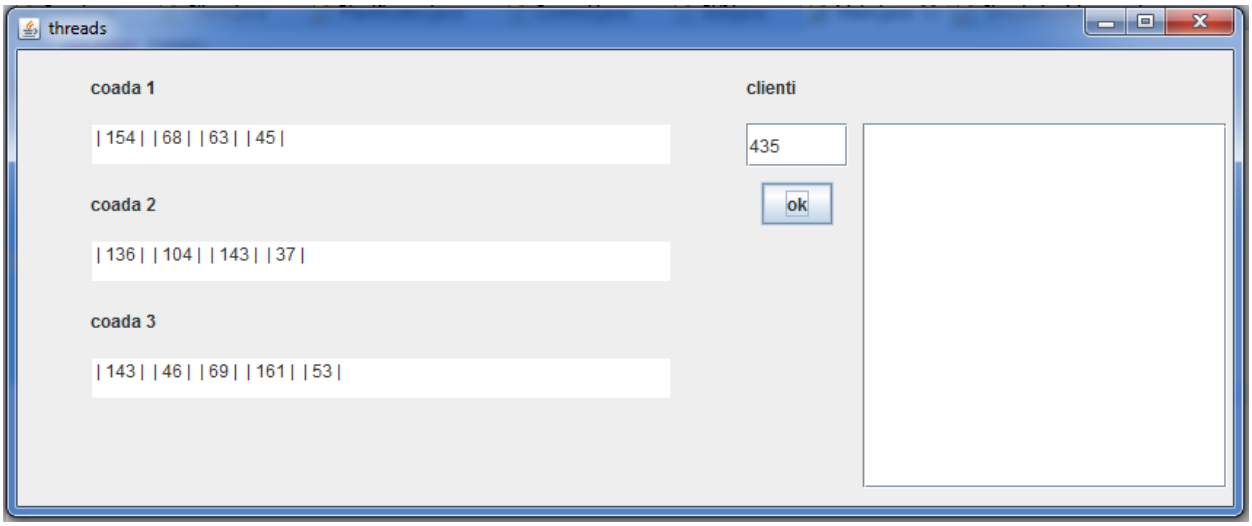
5. Testare şi rezultate

În continuare vom testa toate scenariile de utilizare descrise în capitolul 1:

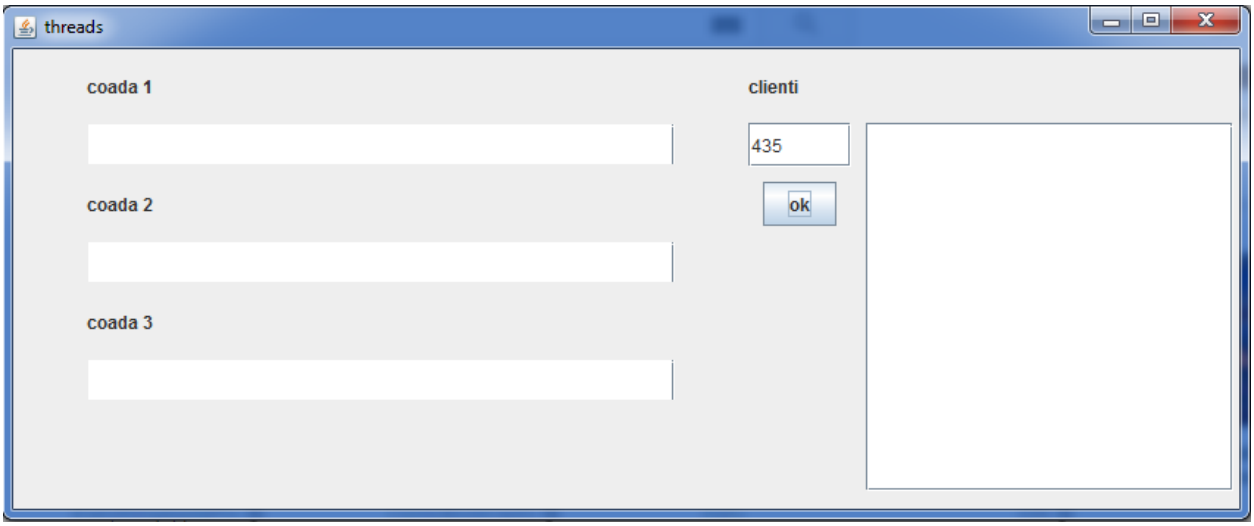


După cum se poate observa, dacă se încearcă rularea simulării fără introducerea numărului de clienţi programul va arunca o excepţie.

În cazul în care se va introduce un număr de clienţi programul va funcţiona în condiţii normale, distribuind clienţii la coada cu cel mai mic număr de clienţi atunci când timpul de sosire al fiecăruia coincide cu timpul curent al simulării.



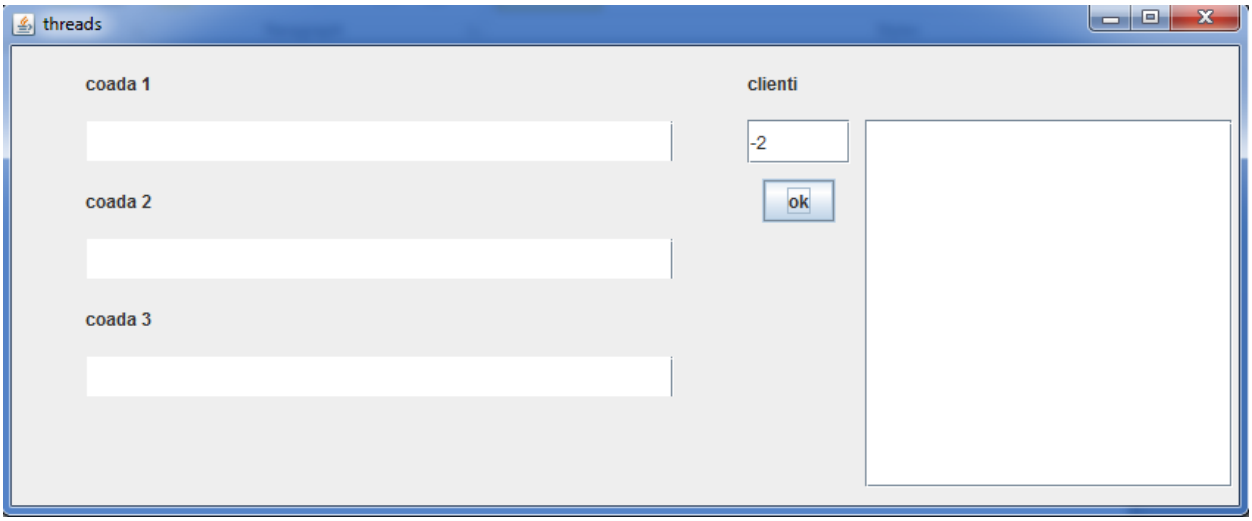
După finalizarea execuţiei programului putem observa faptul ca cele 3 cozi vor fi goale



Pe lângă realizarea simulării şi afişarea cozilor în timp real programul a salvat şi un log al evenimentelor în fişierul “log.txt”

```
clientul cu timpul de procesare 144 s-a asezat la coada 1
clientul cu timpul de procesare 133 s-a asezat la coada 2
clientul cu timpul de procesare 148 s-a asezat la coada 3
clientul cu timpul de procesare 101 s-a asezat la coada 1
clientul cu timpul de procesare 61 s-a asezat la coada 2
clientul cu timpul de procesare 123 s-a asezat la coada 3
clientul cu timpul de procesare 132 s-a asezat la coada 1
clientul cu timpul de procesare 156 s-a asezat la coada 2
clientul cu timpul de procesare 118 s-a asezat la coada 3
clientul cu timpul de procesare 175 s-a asezat la coada 1
clientul cu timpul de procesare 185 s-a asezat la coada 2
clientul cu timpul de procesare 74 s-a asezat la coada 3
clientul cu timpul de procesare 189 s-a asezat la coada 1
clientul cu timpul de procesare 184 s-a asezat la coada 2
clientul cu timpul de procesare 99 s-a asezat la coada 3
clientul cu timpul de procesare 155 s-a asezat la coada 1
clientul cu timpul de procesare 144 a iesit din coada 1
clientul cu timpul de procesare 90 s-a asezat la coada 1
clientul cu timpul de procesare 41 s-a asezat la coada 2
clientul cu timpul de procesare 101 a iesit din coada 1
clientul cu timpul de procesare 133 a iesit din coada 2
clientul cu timpul de procesare 148 a iesit din coada 3
clientul cu timpul de procesare 61 a iesit din coada 2
clientul cu timpul de procesare 132 a iesit din coada 1
clientul cu timpul de procesare 123 a iesit din coada 3
clientul cu timpul de procesare 156 a iesit din coada 2
clientul cu timpul de procesare 118 a iesit din coada 3
clientul cu timpul de procesare 175 a iesit din coada 1
clientul cu timpul de procesare 74 a iesit din coada 3
clientul cu timpul de procesare 185 a iesit din coada 2
clientul cu timpul de procesare 99 a iesit din coada 3
clientul cu timpul de procesare 189 a iesit din coada 1
clientul cu timpul de procesare 184 a iesit din coada 2
clientul cu timpul de procesare 41 a iesit din coada 2
clientul cu timpul de procesare 155 a iesit din coada 1
clientul cu timpul de procesare 90 a iesit din coada 1
```

Un alt posibil caz de utilizare este cazul în care numărul de clienţi introdus este negativ, caz în care programul va funcţiona ca şi când numărul de clienţi introdus ar fi fost 0:



6.Concluzii

Programul pe care l-am avut de realizat în cadrul acestei teme a fost unul de o complexitate relativ ridicată.Cu toate acestea principalul subiect abordat în cadrul acestei teme şi anume thread-urile este unul extrem de important având o mulţime de aplicaţii practice. Programul pe care l-am dezvoltat în cadrul acestei teme poate fi îmbunătăţit în diverse moduri precum afişarea logului evenimentelor în interfaţa grafică pentru a putea fi accesat mai uşor de utilizator sau colectarea mai multor informaţii în cadrul acestui log . O altă modalitate prin care acest program ar putea fi îmbunătăţit l-ar reprezenta permiterea lucrului cu un număr variabil de case.

Toate acestea fiind spuse, pot spune ca această temă m-a ajutat să învăţ un număr mare de noi concepte legate de thread-uri şi multithreading si mi-a oferit experienţă în lucrul cu thread-urile, ceea ce îmi va fi foarte util în viitor .

7.Bibliografie

- www.stackoverflow.com
- www.youtube.com
- www.wikipedia.org
- www.baeldung.com
- Cursuri TP : I.Salomie, C.Pop

Tool utilizat pentru crearea diagramei UML: www.lucidchart.com