Tema 2

Gestionare cozi

Proiect realizat de

Bursuc Andrei

Seria A, Grupa 30224

Contents

[**1.** **Obiectivul temei** 3](#_Toc479331388)

[**2.** **Analiza problemei, modelare, scenarii, cazuri de utilizare** 3](#_Toc479331389)

[**3.** **Proiectare** 4](#_Toc479331390)

[I. Decizii de proiectare 4](#_Toc479331391)

[II. Diagrama UML 4](#_Toc479331392)

[III.Proiectarea claselor și relațiile dintre ele 5](#_Toc479331393)

[IV. Interfață utilizator 14](#_Toc479331394)

[**4.** **Concluzii si dezvoltari ulterioare** 15](#_Toc479331395)

[**5.** **Bibliografie** 16](#_Toc479331396)

# **Obiectivul temei**

Acest proiect urmareste realizarea unui program ce simuleaza 3 sau mai multe case sau ghisee, de tipul celor din centrele comerciale sau din institutiile publice. Programul trebuie sa fie capabil sa gestioneze in cadrul timpului dat de catre utilizator, in mod eficient, prin intermediul caselor, un flux continuu de clienti, cozile fiind monitorizare in mod direct prin intermediul unei interfete grafice, dar si a unui logger unde este afisat timpul la care clientul a intrat la casa si a iesit de la casa, dar si cat a asteptat la casa. Datele legate de simulare precum durata simularii, numarul de cozi, service time-ul unui client vor fi date prin intermediul interfetei.

# **Analiza problemei, modelare, scenarii, cazuri de utilizare**

Problema pe care o ridica aceasta aplicatie este data de modul in care modelam programul astfel incat fluxul de clienti care apare la case sa fie gestionat in mod eficient si aplicatia sa stie sa repartizeze fiecare client la casa cu cel mai mic timp de asteptare.De asemenea cozile trebuie monitorizare in timp real prin intermediul interfetei grafice, utili-zatorul fiind capabil sa vada exact cati clienti asteapta la o casa, in acelasi timp intr-o casuta text din interfata el va vedea momentele la care un client s-a asezat la coada la o casa, a iesit de la casa dar si timpul pe care un client l-a asteptat la coada. In plus mai stie timpul mediu de asteptare la casa, perioada in care la coada la casa nu a astepta nimeni,dar si momentul la care la coada au astepta cei mai multi clienti.

Aplicatia primeste ca date de intrare, oferite de utilizator prin intermediul interfetei: timpul minim si maxim dintre aparitia a doi clienti la coada, timpul minim si maxim de servire a unui client, durata simularii si numarul de cozi.

In cadrul programului fiecare coada va fi reprezentata de un thread ( thread-urile pentru cozi sunt definite prin intermediul unei clase unice ce implementeaza interfata Runnable), clientii vor fi generati prin intermediul unui thread special care va rula in timpul de simulare dat de utilizator, de asemenea la finalul executiei acestui thread vom opri si celelalte thread-uri din rulare. Practic acest thread va genera un client ce va fi preluat de o casa dupa care se va opri pentru un timp random (a carui valoare este intre timpul minim si maxim dintre aparitia a doi clienti) si isi va continua rularea. Pe langa aceste thread-uri exista in plus un thread care se ocupa de scrierea textului in logger-ul din interfata grafica, dar si un thread special care monitorizeaza fiecare casa si determina timpii medii de asteptare, ora de varf pentru fiecare casa si timpul in care casele au fost goale. Toate aceste thread-uri sunt gestionate de o clasa care preia datele de la utilizator si porneste thread-urile. Pe interfata este prezent un singur buton care porneste programul, 6 casute in care vor fi introduse datele, 5 progress bar-uri in care va fi afisata starea fiecarei case in timp real si un JTextArea care joaca loc de logger in care va fi afisat timpul la care un client se aseaza sau iese de la casa si la final datele extra despre care am vorbit mai sus.

# **Proiectare**

## I. Decizii de proiectare

Pentru realizarea aplicației am optat pentru utilizarea arhitecturii MVC (Model ). Ideea este separarea interfeței utilizator într-un Model, un View (vedere, vizualizare) (creează afişajul, interacționând cu Modelul după nevoi), şi un Controller (răspunde la cererile utilizatorului, interacționând atât cu Vizualizarea cât şi cu Controlorul după nevoi.

Arhitectură:

• **Model** -Această parte a controlatorului manipulează operațiunile logice și de utilizare a informației (trimisă dinainte de către rangul său superior), în cazul acestei probleme modelul este reprezentat de: clasa SimulationManager care se ocupa de initializarea si pornirea datelor cat si colectarea datelor de utilizator si transmiterea lor mai departe spre threadul ce genereaza clienti; clasa Server ce defineste practic un obiect de tip coada; clasa EventListener ce implementeaza interfata IEL si ajuta la monitorizarea caselor, de asemenea ea instantiaza si un thread ce numara clientii de la fiecare casa o data la 100ms; clasa Logger ce ajuta la scrierea textului in casuta din interfata.

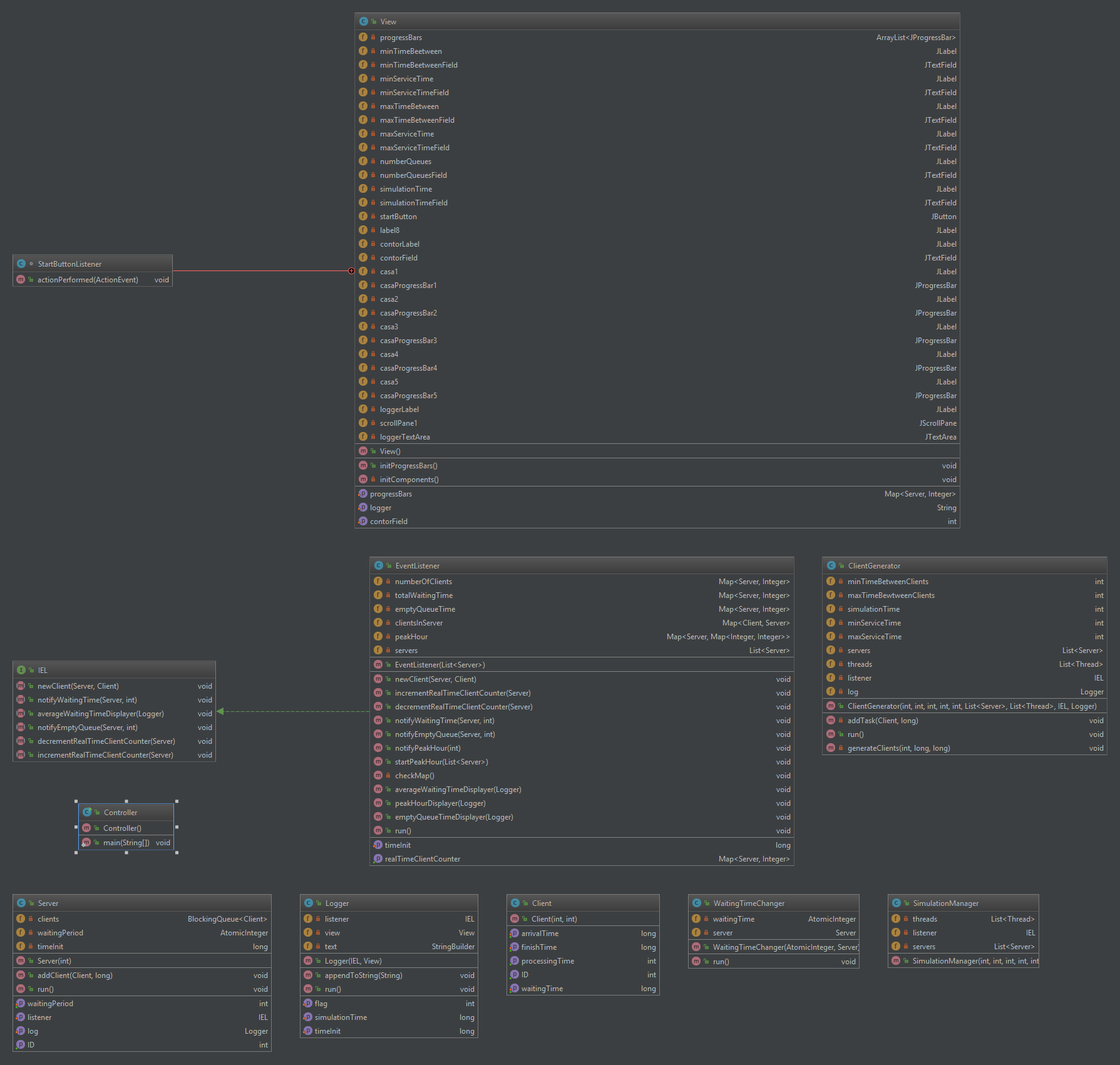
• **View**- Acestui membru al familiei îi corespunde reprezentarea grafică, sau mai bine zis, exprimarea ultimei forme a datelor: interfața grafică ce interacționează cu utilizatorul final. Rolul său este de a evidenția informația obținută până ce ea ajunge la controller. Interfața a fost realizată cu ajutorul bibliotecii Javax.Swing utilizând un plugin special pentru IntelIJ denumit JFormDesigner, ce permite realizarea mai ușoară a unei interfețe atractive pentru utilizator și ușor de folosit.

• **Controller** - Cu acest element putem controla accesul la aplicația noastră. Pot fi fișiere, scripts sau programe, in general orice tip de informație permisă de interfață. În acest fel putem diversifica conținutul nostru de o formă dinamică și statică, în același timp. În cazul aplicației de față în cadrul controller-ului instantiem un obiect de tip View in care este definit listener-ul pentru butonul din aplicatie, in listenerul din buton este instantiat un obiect de tip SimulationManager care practic porneste simularea cozilor.

Codul a fost scris si compilat utilizând IDE-ul IntelIJ IDEA, mediu de dezvoltare software dezvoltat de catre compania JetBrains ( cunoscută anterior sub numele de IntelIJ ). El vine in doua variante ”Community Edition” și ”Ultimate Edition”, aceasta din urmă fiind disponibilă gratis studenților. Prima versiune a IntelIJ a fost lansată in Ianuarie 2001 și a fost unul dintre primele IDE-uri pentru java care dispunea de funcții avansate de ” Code Navigation” și ”Code Refactoring”. Ultima versiune 12.1 dispune de suport pentru Java 8, posibilitatea de dezvoltare de interfață grafică pentru Android, Play 2.0 și Scala.

## II. Diagrama UML

În continuare este prezentată diagrama UML a aplicației generata cu ajutorul IDE-ului IntelIJ IDEA:



## 

## III.Proiectarea claselor și relațiile dintre ele

**Clasa Simulation Manager**

Aceasta clasa este definita de urmatoarele variabile:

private List<Thread> **threads**;  
private IEL **listener**;  
private List<Server> **servers**;

Lista servers contine obiectele de tip server ce vor fi folosite pentru a instantia threadurile. Lista Threads contine thread-urile propriu-zise iar listener este folosit pentru a instantia obiectul ce va monitoriza casele de a lungul simuarii.

Constructorul clasei este:

public **SimulationManager**(int minTimeBetweenClients, int maxTimeBetweenClients, int simultationTime, int numberQueues, int minServiceTime, int maxServiceTime,View view)  
{  
  
  
 **threads**=new ArrayList<Thread>();  
 **servers**=new ArrayList<Server>();  
  
 for(int **i**=0;**i**<numberQueues;**i**++)  
 {  
  
 **servers**.add(new Server(**i**));  
 }  
  
 **listener**=new EventListener(**servers**,view);  
 Logger **log**=new Logger(**listener**,view);  
  
 for(int **i**=0;**i**<numberQueues;**i**++)  
 {  
  
 **servers**.get(**i**).setListener(**listener**);  
 **servers**.get(**i**).setLog(**log**);  
 }  
  
 for(Server **server**:**servers**)  
 {  
 Thread **thread**=new Thread(**server**);  
 **threads**.add(**thread**);  
 **thread**.start();  
 }  
  
 ClientGenerator **gen**=new ClientGenerator(minTimeBetweenClients,maxTimeBetweenClients,simultationTime,minServiceTime,maxServiceTime,this.**servers**,this.**threads**,**listener**,**log**);  
 Thread **t**=new Thread(**gen**);  
 **t**.start();  
  
  
}

Constructorul primeste ca parametrii principalele date oferite de utilizator prin intermediul interfetei: timpul minim si maxim dintre aparitia a doi clienti la coada, timpul minim si maxim de servire a unui client, durata simularii si numarul de cozi, dar si un obiect de tip View instantiat in Controller.

In constructor instantiem obiectele de tip Server (atatea cat numaru de servere dat de utilizator prin interfata) si le adaugam in lista Servers. Mai departe instantem obiecte de tip Listener si Logger si le setam pe toate obiectele din lista Servers, instantiem Thread-urile le adaugam in lista threads si le pornim. In final instantiem generatoru de clienti si pornim threadul specifc.

**Clasa Client**

Aceasta clasa defineste un client, ce este identificat prin variabilele:

private int **ID**;  
private long **arrivalTime**;  
private long **finishTime**;  
private int **processingTime**;  
private long **waitingTime**;

Variabilele sunt folosite la identificare clientului (ID) , dar si salvarea datelor legate de momentul in care a ajuns la casa, a parasit casa, timpul in care acesta poate fi procesat si timpul pe care l-a asteptat la coada.

In clasa pe langa constructor avem o serie de getere si setere cum ar fi:

public void setWaitingTime(long waitingTime) {  
 this.**waitingTime** = waitingTime;  
}  
  
  
public long getFinishTime() {  
 return **finishTime**;  
}

**Clasa Logger**

Clasa Logger implementeaza interfata Runnable si este folosita pentru a scrie in logger toate datele colectate de la obiectele de tip server, ea insasi definind un thread care verifica tot timpul daca au fost colectate informatii noi si le afiseaza in text-box-ul din interfata. Principalele variabile definite in clasa sunt:

private IEL **listener**;  
private View **view**;  
private StringBuilder **text**;  
private int **flag**;  
private long **timeInit**;  
private long **simulationTime**;

Principalele metode sunt:

public synchronized void appendToString(String s)  
{  
 **text**.append(s);  
 **text**.append("\n");  
}

Aceasta metoda este folosita in thread-uri pe post de afisor ea colectand tot timpul text pe care mai departe il ataseaza StringBuilder-ului.

public void run() {  
 long **time**=0;  
 EventListener **listen**=(EventListener) **listener**;  
 int **size**=0;  
 while(true)  
 {  
 Thread.sleep(1);

if((**time**-**timeInit**)<=**simulationTime**)  
 **view**.setContorField((int)(**time**-**timeInit**));  
 if(**text**.toString().length()>**size**)  
 **view**.setLogger(**text**.toString());  
 **size**=**text**.toString().length();  
 **view**.setProgressBars(**listen**.getRealTimeClientCounter());  
 if(**flag**==1)  
 {**listen**.averageWaitingTimeDisplayer(this);  
 **listen**.emptyQueueTimeDisplayer(this);  
 **listen**.peakHourDisplayer(this);  
 **flag**=2;  
 }  
 else if(**flag**==2)  
 break;  
   
 }  
}

In metoda run definim mai intai un obicet de tip EventListener facand cast la obiectul listener primit ca parametru, apoi intr-un while masuram tot timpul valoarea timpului si folosim valoara ca sa setam contorul din interfata ce ne arata cat timp mai are de rulat programul. Mai departe verificam daca marimea StringBuiler-ului s-a modificat si este nevoie sa scriem in casuta text din interfata, dupa in view setam progress bar-urile preluand datele culese in timp real de listener. In final atunci cand metoda run din generatorul de clienti se inchide activeaza un flag ce va notifica metoda run de aici sa afiseze datele legate de timpul mediu de asteptare, ora de varf, timpul in care casele au fost goale ( culele de listener) si sa modifice din nou flag-ul cu 2 pentru ca la urmatoarea iteratie sa iesim din while si sa incheiem rularea thread-ului. De mentionat este ca la fiecare iteratie din while punem threadul in sleep o miniescunda pentru ca programul sa nu foloseasca prea multe resurse.

**Clasa ClientGenerator**

Variabilele clasei sunt:

private int **minTimeBetweenClients**;  
private int **maxTimeBewtweenClients**;  
private int **simulationTime**;  
private int **minServiceTime**;  
private int **maxServiceTime**;  
private List<Server> **servers**;  
private List<Thread> **threads**;  
private IEL **listener**;  
private Logger **log**;

Acestea stocheaza practic valorile trimise ca parametru in constructorul clasei,o parte din valori ( cele de tip int ) au fost preluate mai intai de SimulationManager din View si transmise mai departe constructorului unui obiect de tipul acestei clase.

Aceasta clasa implementand interfata Runnable are ca principala metoda Run:

public void run() {  
  
 int **i**=0;  
 long **timeInit**=System.currentTimeMillis();  
 long **time**=**timeInit**;  
 **log**.setTimeInit(**timeInit**);  
 **log**.setSimulationTime(**simulationTime**);  
 EventListener **listen**=(EventListener)**listener**;  
 **listen**.setTimeInit(**time**);  
 Thread **t**=new Thread(**listen**);  
 **t**.start();  
 Thread **logThread**=new Thread(**log**);  
 **logThread**.start();  
  
 generateClients(**i**, **timeInit**, **time**);  
 **t**.interrupt();  
 for(Thread **thread**:this.**threads**)  
 **thread**.interrupt();  
 **log**.setFlag(1);  
  
}

Aici, in obiectul log ( ce se ocupa de afisare in TextBox) setam timpul initial de la care a pornit simularea si durata simularii (prin intermediul unor metode din log), setam timpul initial din obiectul listen si intantiem un Thread cu el pe care ulterior il pornim. Mai departe apelam metoda generateClients. In finalul metodei oprim din rulare thread-urile specifice cozilor, threadul listener-ului si notificam thread-ul obiectului log ca trebuie sa afiseze ultimul set de date si sa se opreasca ulterior.

private void generateClients(int i, long timeInit, long time) {  
 while(time<timeInit+this.**simulationTime**)  
 {  
 try {  
 Random **rand**=new Random();  
 int **numarProduse**=**rand**.nextInt((this.**maxServiceTime**-this.**minServiceTime**+1))+this.**minServiceTime**;  
 Client **client**=new Client(i,**numarProduse**);  
 i++;  
 this.addTask(**client**,timeInit);  
 int **interval** = **rand**.nextInt((this.**maxTimeBewtweenClients** - this.**minTimeBetweenClients** + 1)) + this.**minTimeBetweenClients**;  
 Thread.sleep(**interval**);  
 time = System.currentTimeMillis();  
 }  
 catch(InterruptedException e)  
 {  
 JOptionPane.showMessageDialog(null,e.getMessage());  
 }  
 }  
}

In aceasta metoda care ruleaza un timp egal cu simulationTime-ul dat de utilizator, instantiem clienti cu un ID si un processing time (numarProduse) generat random si apelam metoda addTask ce va fi explicata mai jos. In final generam un interval random dintre clienti si oprim threadul curent un timp egal cu acel interval.

private synchronized void addTask(Client client,long timeInit)  
{  
  
 int **min** = Integer.***MAX\_VALUE***;  
 for (Server **server** : **servers**) {  
 if (**server**.getWaitingPeriod() < **min** ) **min** = **server**.getWaitingPeriod();  
 }  
  
 for (Server **server** : **servers**) {  
 if (**server**.getWaitingPeriod() == **min** ) {  
 {  
 **server**.addClient(client,timeInit);  
 break;  
 }  
  
 }  
 }  
}

Metoda addTask calculeaza initial timpul minim de asteptare din cozi si mai apoi parcurge inca o data cozile si apeleaza metoda addClient pentru a adauga clientul in coada respectiva.

**Clasa Server**

Aceasta clasa este folosita pentru a defini casele ce gestioneaza clientii (cozile), un obiect de acest tip fiind definit de variabilele:

private AtomicInteger **waitingPeriod**;  
private int **ID**;  
private long **timeInit**;  
private Logger **log**;  
private IEL **listener**;

Demn de mentionat este waitingPeriod care practic indica timpul de asteptare la casa timp necesar atunci cand vrem sa aflam la care coada clientul are mai putin de asteptat.

Implementand interfata Runnable principala metoda din clasa este:

public void run() {  
  
 while(true)  
 {  
 long **timeInitEmpty**=0;  
 long **timeFinalEmpty**=0;  
 try {  
 **timeInitEmpty** = System.currentTimeMillis();  
 Client **client** = **clients**.take();  
 **timeFinalEmpty** = System.currentTimeMillis();  
 if (**timeFinalEmpty** - **timeInitEmpty** > 0)  
 **listener**.notifyEmptyQueue(this, (int) (**timeFinalEmpty** - **timeInitEmpty**));  
 this.**listener**.newClient(this, **client**);  
 WaitingTimeChanger **tm**=new WaitingTimeChanger(**waitingPeriod**,this);  
 Thread **ts**=new Thread(**tm**);  
 **ts**.start();  
 Thread.sleep(**client**.getProcessingTime());  
 **ts**.interrupt();  
 **client**.setFinishTime(System.currentTimeMillis()-**timeInit**);  
 **client**.setWaitingTime(**client**.getFinishTime() - **client**.getArrivalTime());  
 **log**.appendToString("Clientul " + **client**.getID() + " a iesit de la casa " + this.**ID** + " la " + **client**.getFinishTime()+" ms!" );  
 **log**.appendToString("Clientul "+**client**.getID()+ " a asteptat " + **client**.getWaitingTime()+" ms la casa "+this.**ID**+"!");  
 **listener**.notifyWaitingTime(this, (int) **client**.getWaitingTime());  
 **listener**.decrementRealTimeClientCounter(this);  
 } catch (InterruptedException e) {  
 **log**.appendToString("Casa "+this.**ID**+" s-a inchis!");  
 if(**timeInitEmpty**>0 && **timeFinalEmpty**==0)  
 {  
 **timeFinalEmpty**=System.currentTimeMillis();  
 **listener**.notifyEmptyQueue(this,(int)(**timeFinalEmpty**-**timeInitEmpty**));  
 }  
 break;  
 } }  
}

In cadrul metodei preluam fiecare client din BlockingQueue-ul clients (in acelasi timp preluam timpul de inainte de take si de dupa take si daca acesta este mai mare ca 0 inseamna ca la coada nu s-a aflat nimeni si metoda take a asteptat pana cand a putut prelua un client, aceasta valoare o trimitem mai departe prin metoda notifyEmptyQueue din listener).

Dupa punem in sleep thread-ul cu un timp egal cu cu processingTime-ul clientului dar inainte instantiem un thread special care intra in sleep cate o milisecunda si decrementeaza waitingTime-ul de la casa pentru a optimiza repartizarea clientilor la casa.

In final trimitem la log prin metoda appendToString mesajele pe care utilizatorul le va vedea legate de momentul in care clientul a parasit coada si cat a asteptat la coada. Deoarece thread-ul va fi inchis folosindu-se metoda interrupt si astfel va fi aruncata o exceptie care va fi prinsa de catch unde vom trimite la log un mesaj ca s-a inchis casa si transmitem la listener timpul in care casa a fost goala chiar inainte de inchidere.

public synchronized void addClient(Client client,long timeInit)  
{  
 try {  
 this.**timeInit** = timeInit;  
 if (client.getArrivalTime() == 0)  
 {  
 **clients**.put(client);  
 **listener**.incrementRealTimeClientCounter(this);  
 client.setArrivalTime(System.currentTimeMillis() - timeInit);  
 **log**.appendToString("Clientul " + client.getID() + " s-a asezat la coada la casa " + this.**ID** + " la " + client.getArrivalTime() + "!");  
 **log**.appendToString("Clientul "+client.getID()+" are timp de procesare "+client.getProcessingTime()+"!");  
 **waitingPeriod**.addAndGet(client.getProcessingTime());  
 }  
 }  
 catch(InterruptedException e)  
 {}  
}

Metoda addClient apelata din metoda addTask din ClientGenenerator pune efectiv clientul in lista de clienti a obiectului cu metoda put, seteaza timpul in care clientul s-a asezat la casa si trimite la log mesaje legate de timpul la care s-a asezat la casa si id-ul casei, dat si timpul lui de procesare.

**Clasa EventListener**

Aceasta clasa e definita de variabilele:

private Map<Server,Integer> **numberOfClients**;  
private Map<Server,Integer> **totalWaitingTime**;  
private Map<Server,Integer> **emptyQueueTime**;  
private Map<Client,Server> **clientsInServer**;  
private Map<Server,Integer> **realTimeClientCounter**;  
private Map<Server,Map<Integer,Integer>> **peakHour**;  
private List<Server> **servers**;  
private long **timeInit**;

Fiecare monitorizand numarul total de clienti care au fost la casa, timpul total pe care l-au astepta toti clientii la casa, timpul in care casa a fost goala si orele de varf.

Aceasta clasa contine mai multe metode fiecare monitorizand in timp real situatia de la case. De exemplu:

public void notifyWaitingTime(Server server,int waitingTime)  
{  
 int **time**;  
 if (**totalWaitingTime**.containsKey(server)) {  
 **time**=**totalWaitingTime**.get(server);  
 }  
 else  
 {  
 **totalWaitingTime**.put(server,0);  
 **time**=0;  
 }  
 **totalWaitingTime**.put(server,**time**+waitingTime);  
  
}

Aceasta metoda creste pentru serverul dat timpul total de asteptare, sau daca serverul nu era in lista de servere il pune in Map-ul totalWaitingTime cu valoarea 0 pentru timp. Pe acelasi principiu functioneaza si metoda notifyEmptyQueueTime care contabilizeaza insa valorile din Map-ul emptyQueueTime.

Mai avem metodele de tipul:

public void peakHourDisplayer(Logger log)  
{  
 if(!**peakHour**.isEmpty()) {  
 log.appendToString("");  
 for (Map.Entry<Server, Map<Integer, Integer>> **entry** : **peakHour**.entrySet()) {  
  
 Map<Integer, Integer> **values** = **entry**.getValue();  
 int **max** = 0, **max2** = 0;  
 if (!**values**.isEmpty()) {  
 for (Map.Entry<Integer, Integer> **entry2** : **values**.entrySet()) {  
 **max** = **entry2**.getValue();  
 **max2** = **entry2**.getKey();  
  
 }  
 }  
 **max** = **max** - **max** % 100;  
 log.appendToString(" La ora de varf " + **max** + " ms pentru casa " + **entry**.getKey().getID() + " au fost " + **max2** + " clienti!");  
 }  
 }  
}

Care sunt folosite pentru a afisa datele culese in timpul rularii caselor pentru fiecare casa in parte. Alte metode de acest gen sunt averageWaitingTimeDisplayer si emptyQueueTimeDisplayer. Toate aceste metode primesc ca parametru logger-ul la care transmit datele ce vor fi afisate mai departe pentru client. In aceasta metoda practic preluam toate datele de la case preluate din 100 in 100 de ms si calculam maximul pentru a afla si afisa ora de varf de la fiecare casa.

Ultima cea mai importanta metoda este metoda run aceasta clasa implementand interfata Runnable:

public void run() {  
  
 long **time**=0;  
 this.startPeakHour(**servers**);  
 while(true)  
 {  
 try{  
  
 **time**=System.currentTimeMillis();  
 if((**time**-**timeInit**)%100==0)  
 this.notifyPeakHour((int)(**time**-**timeInit**));  
 Thread.sleep(1);  
 }  
 catch (InterruptedException e)  
 {  
 break;  
 }  
 }  
}

Aceasta metoda apeleaza din 100 in 100 de ms metoda notifyPeakHour care memoreaza situatia de la casa in acel moment. Thread-ul intra in sleep cate o milisecunda la fiecare iteratie din while pentru ca programul sa foloseasca mai putine resurse.

**Clasa View**

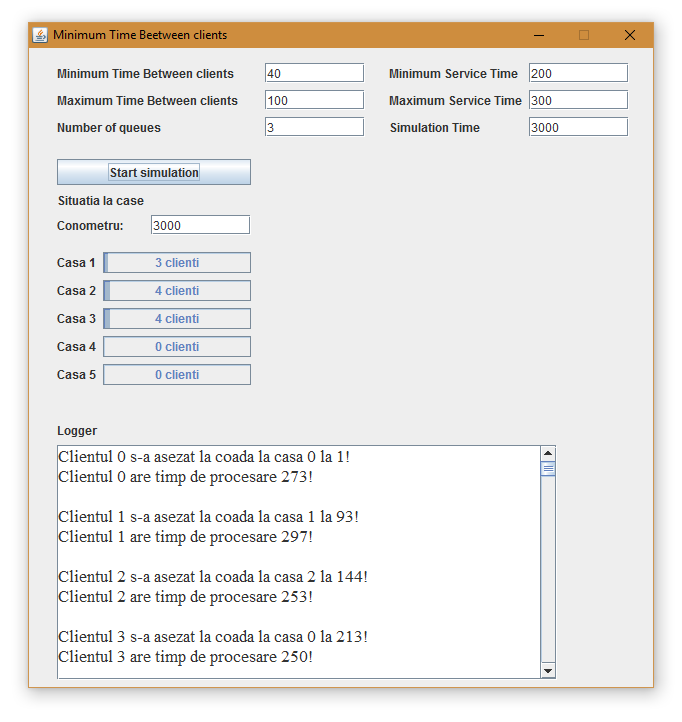
Interfata a fost realizata folosind pluginul JFormDesigner deci mare parte din cod a fost generat de plugin, dar trebuie mentionat listener-ul pentru butonul de start din interfata.

class StartButtonListener implements ActionListener  
{  
 public void actionPerformed(ActionEvent e) {  
 try {  
  
 int **minTimeBetweenClients**=new Integer(**minTimeBeetweenField**.getText());  
 int **maxTimeBetweenClients**=new Integer(**maxTimeBetweenField**.getText());  
 int **minServiceTime**=new Integer(**minServiceTimeField**.getText());  
 int **maxServiceTime**=new Integer(**maxServiceTimeField**.getText());  
 int **numberQueues**=new Integer(**numberQueuesField**.getText());  
 if(**numberQueues**<3 || **numberQueues**>5) throw new NumberFormatException("Numarul de cozi trebuie sa fie intre 3 si 5!");  
 int **simulationTime**=new Integer(**simulationTimeField**.getText());  
 if(**maxTimeBetweenClients**<**minTimeBetweenClients**)  
 throw new NumberFormatException("Timpul maxim dintre clienti este mai mic decat cel minim!");  
 if(**maxServiceTime**<**minServiceTime**)  
 throw new NumberFormatException("Timpul maxim de servire a unui client este mai mic decat timpul minim!");  
 if(**simulationTime**<=**maxServiceTime** || **simulationTime**<=**maxTimeBetweenClients**)  
 throw new NumberFormatException("Timpul de simulare dat este prea mic");  
 View.this.initProgressBars();  
 //IEL listener=new EventListener();  
 SimulationManager **manager**=new SimulationManager(**minTimeBetweenClients**,**maxTimeBetweenClients**,**simulationTime**,**numberQueues**,**minServiceTime**,**maxServiceTime**,View.this);  
  
 } catch (NumberFormatException e2) {  
 JOptionPane.showMessageDialog(null,e2.getMessage());  
 }  
 }  
}

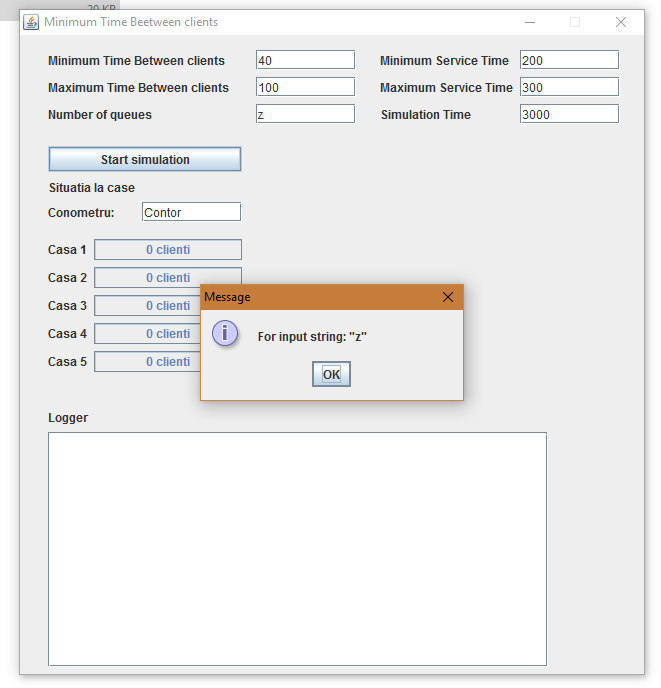
Aici preluam datele de la interfata, verificam daca ele sunt corecte si in caz ca nu sunt aruncam erori cu mesaje specifice pentru a instiinta utilizatorul ce date a introdus gresit. Dupa initializam progressBar-urile si folosim datele culese din interfata pentru a instantia un obiect de tipul SimulationManager ce va pune de altfel tot programul in miscare.

## IV. Interfață utilizator

Interfata este alcatuita din 6 textField-uri in care vom introduce principalele date de simulare: Minimum Time Between clients, Maximu Time Between clients, Minimum Service Time, Maximum Service Time, Number of queues si Simulation Time. De asemenea in interfata mai apare butonul de pornire dar si 5 progressbar-uri fiecare cu numele clasei corespunzatoare in stanga. In partea de jos avem un JTextArea unde o sa se afiseze informatii despre simulare.



In cazul in care un set de date a fost introdus gresit va fi afisat un mesaj de eroare de tipul:



Dupa cum se poate observa in prima poza cozile sunt monitorizate in timp real prin intermediul progressBar-urilor.

# **Concluzii si dezvoltari ulterioare**

In optinia mea aceasta aplicatie poate fi utilizata pentru a simula o situatie reala in care verificam cum se descurca un numar fix de case intr-un timp dat la un aflux de clienti care apar la anumite intervale aleatoare.

Dezvoltarea acestei aplicatii m-a ajutat sa inteleg mai bine un concept relativ nou pentru mine cel al thread-urilor care ruleaza concurential. De-a lungul dezvoltarii am invatat cum sa instantiez un thread, cum pot sa pornesc, sa opresc temporar un thread si cum sa ii intrerup rularea la nevoie. De asemenea am invatat ca tipurile de date care sunt accesate de mai multe thread-uri trebuie sa fie sincronizate, si faptul ca nu putem sti in ce ordine vor rula mai multe threaduri, aceasta ordine fiind aleasa de JVM. In final am inteles ca prin folosirea thread-urilor putem face un program sa fie mai rapid si mai atractiv pentru client.

Posibile dezvoltari ulterioare ar putea constitui in realizarea unei interfete grafice mai sugestive care sa foloseasca animatii pentru a reprezenta clientii care intra sau ies de la casa. De asemenea se poate oferi posibilitatea pentru client de a calcula anumite date pentru un interval specific de timp din simulare. In final se poate eficientiza si mai mult modul in care alegem coada in care asezam clientii pentru a obtine timpi medii de asteptare si mai mici.

# **Bibliografie**

1. <http://docs.oracle.com/javase/>
2. <http://stackoverflow.com/>
3. Kathy Sierra, Bert Bates. SCJP Sun Java Certified Programmer for Java 6. McGraw Hill, USA 2008
4. http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
5. http://www.tutorialspoint.com/java/util/timer\_schedule\_period.htm