

MANAGER DE COZI

Buciuman Mihai Catalin
Grupa 30227

Cuprins

1. Obiectivul temei
2. Analiza problemei
3. Proiectare
4. Implementare
5. Rezultate
6. Concluzii si Dezvoltari Ulterioare
7. Bibliografie

1. Obiectivul temei

Obiectivul principal al acestei teme este de a proiecta și implementarea un sistem de manageriere a unor cozi și simularea pe o perioadă determinată a clienților la cozi.

Obiectivele secundare ale acestei teme sunt reprezentate de realizarea unei interfete grafice pentru adăugarea parametrilor simulării. Parametri care sunt ceruți de interfață reprezintă timpul minim și maxim în care se generează următorul client la coadă, timpul minim și timpul maxim pe care îl poate petrece un client la o coadă, timpul simulării și numărul total de cozi. Interfața grafică cerută trebuie să prezinte o animație prin care să se observe cum se desfășoară simularea și un loc în care să se afișeze detaliile despre evenimintele din coadă respective statistica simulării.

2. Analiza problemei

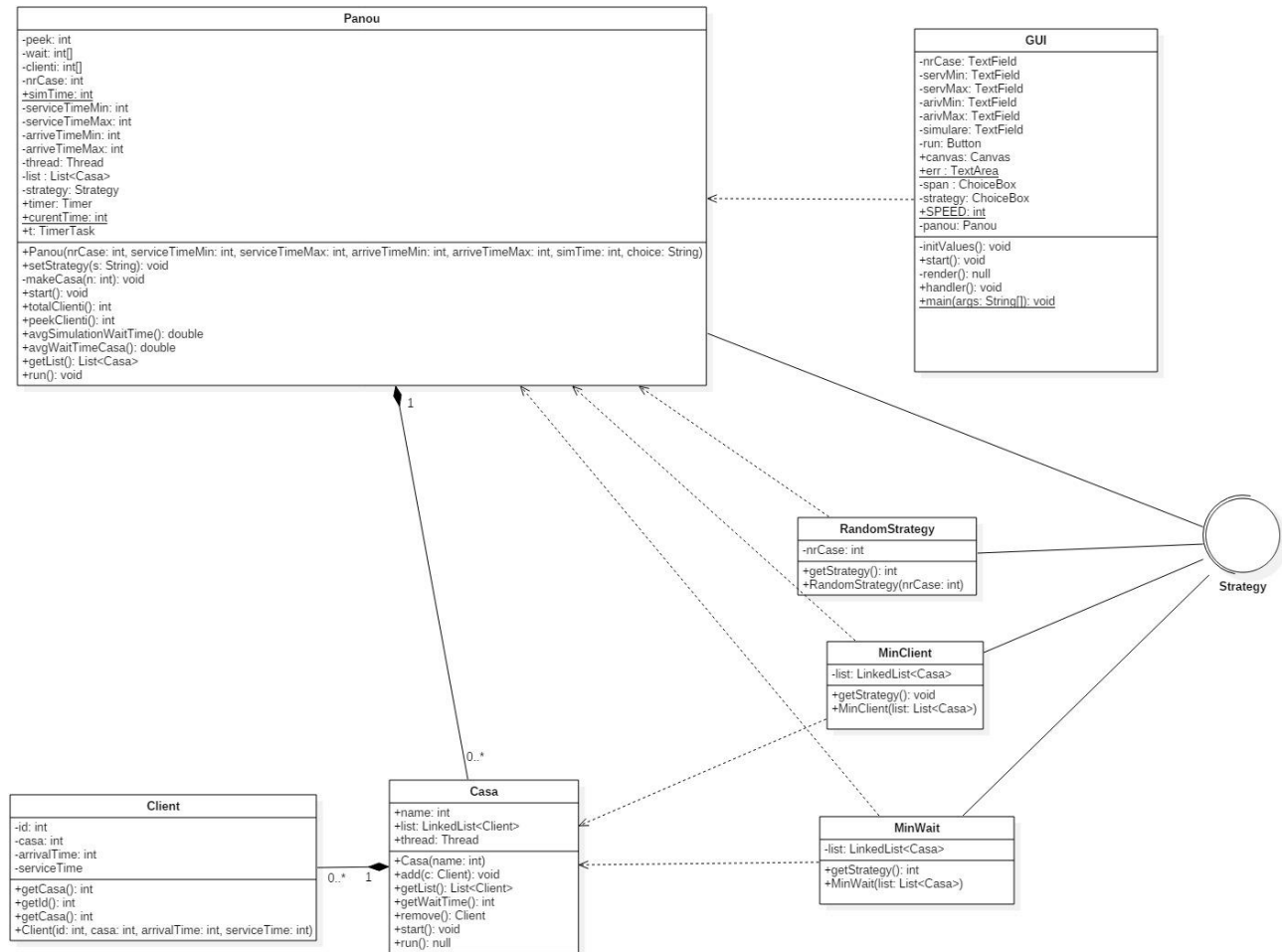
Programul funcționează pe baza unor cozi pe care le putem denumi “Case” și în care fiecare client își va aștepta rândul până când acesta va fi servit. Fiecare coadă funcționează pe baza principiului “First In First Out” sau mai bine spus “Primul venit Primul servit”.

Fiecare client dispune de un timp de servire și de un timp de sosire. Acestea ne arată evoluția cozi în timp. Timpul de sosire este necesar deoarece ne arată în cât timp ajunge clientul la o anumită casă și timpul de servire ne arată cât anume stă la casă.

În concluzie avem nevoie pentru această problemă de Clienți de Case și de un Panou de comandă prin care dirijăm evoluția cozilor.

Perioada de simulare ne arată cât va rula programul nostru.

3.Proiectare



Pentru proiectarea acestei probleme am ales sa folosesc o clasa pentru Client , o clasa pentru Casa si o clasa foarte importanta, Panou. Clasa Panou se refera la panoul de comanda al acestui program prin care dirijam evenimentele simulari. In clasa de panou vom implementa interfata Runnable pentru a putea lucra cu Thread-uri . Aceasta clasa va genera noi client pe baza input-ului dat de utilizator prin interfata grafica.

Clasa de Clienti retine informati despre pozitia si despre timpul alocat fiecarui client. Fiecare client este introdus la o casa dupa generare si din acest lucru este foarte important sa ii retinem in niste variabile numarul clientului sau id-ul acestuia si casa de la care face parte.

Clasa Casa implementeaza si ea la randul ei interfata Runnable pentru a putea lucra cu Thread-uri acest lucru este essential deoarece se va lucra cu mai multe case in programul principal si fiecare casa va merge concurrent.

Clasa de GUI a fost facuta in JavaFX pentru a avea o interfata grafica cat mai placuta si pentru a fi facut programul cat mai “User Friendly” astfel incat utilizatorul sa poata utiliza programul cu placere

Interfata Strategy reprezinta un Strategy pattern prin care putem selecta acele strategii dorim sa avem atunci cand generam noi clienti. Strategia este folosita pentru a genera un anumit indice la o anumita casa astfel in cat pe acea pozitie sa se introduca urmatorul nostru client in coada. RandomStrategy, MinClient si MinWait sunt strategiile pe care le vom folosi pentru generare.

RandomStrategy este o clasa care implementeaza metoda de generare din interfata Strategie si ma ajuta sa pot crea un nou Client. Metoda implementata va alege aleatoriu o pozitie.

MinClient este o clasa care implementeaza metoda de generare din interfata Strategie si ma ajuta sa pot crea un nou Client. Metoda gaseste coada cu cel mai mic numar de clienti si returneaza indicele casei.

MinWait este o clasa care implementeaza metoda de generare din interfata Strategie si ma ajuta sa pot crea un nou Client. Metoda gaseste coada cu cel mai mic numar de asteptare la coada si returneaza indicele casei.

4. Implementare

Proiectul a fost organizat in trei pachete principale : GUI , Simulare si Strategy

In pachetul de Simulare se afla clasele care se ocupa de partea principala a programului adica functionarea pe Thread-uri , generarea si managerierea cozilor.

Panou

Clasa care este defapt panoul de comanda are ca variabile de clasa nrCase care se refera la numarul total de case (sau cozi) pe care programul nostru le prezinta , simTime care este o variabila statica reprezentand timpul simularii introdus tot de utilizator in interfata , serviceTimeMin si serviceTimeMax care sunt

variabilele declarate de utilizator pentru timpul de servire dat de utilizator, arriveTimeMin si arriveTimeMax care reprezinta timpul de asteptare, strategy care reprezinta strategia dorita pentru generarea clientilor in cozi (data de utilizator).

Cum am specificat si mai sus , clasa foloseste Thread pentru rularea programului dar si un Timer care imi masoara timpul de rulare al programului specificat de simTime.

Cel mai important element din aceasta clasa (atribut de clasa) este lista de case care este sincronizata pentru lucrul cu mai multe Thread-uri .

```
private List<Casa> list = Collections.synchronizedList(new  
LinkedList<Casa>());
```

Timer-ul implementat ca sa poata functiona are nevoie de un TimerTask cu care sa poata numara unitatile de timp . TimerTask-ul implementeaza o clasa anonima de tipul TimerTask.

```
TimerTask t=new TimerTask() {  
    @Override  
    public void run() {  
        curentTime++;  
        if(curentTime>simTime){  
            timer.cancel();  
        }  
    }  
};
```

Deoarece in constructor este specificat doar un String cu numele strategiei dorite , am facut o metoda prin care voi putea seta Strategia in functie de valoarea String-ului. Metoda este setStrategy(String s)

```
public void setStrategy(String s) {  
    if(s.equals("Min Clienti"))strategy=new MinClient(list);  
    else if(s.equals("Min Wait"))strategy=new MinWait(list);  
    else strategy=new RandomStrategy(nrCase);  
}
```

Inainte ca toate sa se poata genera un client , lista noastra de case trebuie sa aiba toate casele specificate de utilizator . Pentru asta am implementat o metoda private cu care voi initializa casele . Metoda mai declara si variabilele de clasa wait si clienti cu care pot sa generez la finalul simularii statistica acesteia. Metoda implementata se numeste makeCasa si are ca parametru numarul de case.

```
private void makeCasa(int n){
    wait = new int[nrCase];
    clienti = new int[nrCase];
    for(int i=0;i<n;i++)list.add(new Casa(i));
}
```

Ca Thread-urile implementate in clasa Casa cat si Timer-ul si Thread-ul acestei clase sa functioneze trebuie sa apela pe fiecare metoda de start. Am implementat o metoda cu care acest lucru este posibil in clasa Panou.

```
public synchronized void start(){//pornim thread-urile si timerele
    thread.start();
    timer.scheduleAtFixedRate(t,1000/ GUI.SPEED,1000/ GUI.SPEED);
    for(int i=0;i<list.size();i++){
        list.get(i).start();
    }
}
```

Pentru procesarea Statistici de la finalul unei simulari s-au implementat urmatoarele metode

```
public int totalClienti(){//numarul total de clienti din simulare
    int n=0;
    for(int i=0;i<nrCase;i++){
        n+=clienti[i];
    }
    return n;
}
public int peekClienti(){//functie care imi numara totalu de clienti la
un anumit moment
    int n=0;
    for(int i=0;i<list.size();i++){
        n+=list.get(i).getList().size();
    }
    return n;
}
private double avgSimulationWaitTime(){//timpu mediu de asteptare pentru
simulare
    double k=0;
    for(int i=0;i<nrCase;i++){
        k+=wait[i];
    }
    return (k/totalClienti());
}
private String avgWaitTimeCasa(){//timpu mediu de asteptare pentru casa
String[] rez=new String[nrCase];
```

```
String r="\n";
for(int i=0;i<nrCase;i++){
    rez[i]="Casa "+(i+1)+" average time:
"+String.valueOf((double)wait[i]/clienti[i])+"\n";
}
for(int i=0;i<nrCase;i++){
    r+=rez[i];
}
return r;
}
```

Metoda de totalCienti Numara totalul de clienti care sau generat pe tot parcusul simulari .

Metoda de peekCienti este folosita pentru a vedea cati clinti sunt generati in ora de varf a simulari

Metoda de avgSimulationWaitTime calculeaza timpul total de asteptare de la case , de pe tot parcursul simulari . “WaitTime” sau timpul de masurare se refera la suma tuturor timpurilor de servire de la o coada , adica timpul pe care il asteapta fiecare client la casa (nu la coada).

Metoda de avgWaitTimeCasa reprezinta timpul total de asteptare la fiecare casa . Aceasta metoda returneaza un String pe care noi il vom afisa in TextArea-ul dedicate pentru lista de evenimente

Cea mai importanta metoda din aceasta clasa este metoda implementata de Runnable si cu care vom putea genera clienti pentru simulare.

Metoda are in componenta un while care functioneaza doar daca timpul simulari (simTime) este mai mare ca si timpul current (currentTime) .

Indiferent de ce strategie vom alege timpul de asteptare si timpul de servire se vor genera aleator.

Dupa generarea timpului de asteptare se face un sleep pe Thread presupunand ca acest timp este timpul care trebuie ca un client sa ajunga la coada .

Dupa ce strategia a fost facut se adauga in coada urmatorul element respectiv client.

Pentru a lista evenimentele se cheama TextArea-ul care este static in clasa GUI si se pune urmatorul eveniment , adaugarea in lista. Metoda prin care se face acest lucru are nevoie sa blocheze o anumita pozitie din TextArea pentru a nu se face race. Pentru a indeplini acest lucru am chemat o metoda specifica thread-ului pe care functioneaza interfata facuta in javaFX aceasta blocand thread-ul. Metoda respectiva implementeaza metode din Runnable dar am ales sa folosesc un lamda function.

```

public void run() { //aici porneste thread-ul si generatorul
    int n,serv,id=0,max=0,aux;
    Random r = new Random();
    while (simTime>curentTime) {
        n = arriveTimeMin + r.nextInt(arriveTimeMax-arriveTimeMin+1); //genereaza random
arrive time
        try {
            Thread.sleep(n * 1000/ GUI.SPEED +1); //timpu de asteptare in functie de Speed
selectat de utilizator
        } catch (InterruptedException e) {}
        serv = serviceTimeMin + r.nextInt(serviceTimeMax-serviceTimeMin+1); //genereaza
random service time
        int indiceCase=strategy.getStrategy(); id++; //se selecteaza strategia data de
utilizator
        int s=id,n1=n,serv1=serv;
        if(curentTime>simTime) break;
        list.get(indiceCase).add(new Client(id,indiceCase+1,n,serv)); //adaugare in lista
        wait[indiceCase]+=serv; //pentru calculul timpului mediu se aduna toate
serviceTime-urile generate
        clienti[indiceCase]++; //pentru calculul timpului mediu se aduna toti clienti
        if(max<(aux=peekClienti())) { //la fiecare unitate de timp se face update la
peekTime
            peek=curentTime;
            max=aux;
        }
        if(simTime>=curentTime) javafx.application.Platform.runLater( () ->
GUI.err.appendText(curentTime+": Client "
            +s+" intra la casa "+(indiceCase+1)+" SERVICE TIME: "+ serv1+" ARRIVE
TIME: "+n1+
            " WAIT TIME: "+list.get(indiceCase).getWaitTime()+"\n")); //se scrie in
TextArea adaugarea
    }
    //se afiseaza Statistica
    javafx.application.Platform.runLater( () -> GUI.err.appendText("STOP
SIMULARE\n\nStatistica\nPeek Time: " +peek));
    javafx.application.Platform.runLater( () -> GUI.err.appendText("\nAverage Simulation
Waiting Time: "
        +avgSimulationWaitTime()));
    javafx.application.Platform.runLater( () -> GUI.err.appendText("\nAverage Waiting
Time: " +avgWaitTimeCasa()));
    simTime=0; //pentru oprirea sigura a thread-urilor
    }
}

```

Casa

Clasa aceasta detine ca attribute de clasa variabila name care reprezinta numele casei (sau numarul ex: Casa 1) , lista de Clienti si Thread-ul thread.

Metodele pentru procesarea listelor ce case sunt add care adauga un client nou la finalul casei , remove care functioneaza ca un deque pentru o coada , acesta returneaza elementul de pe pozitia 0 si dupa il sterge.

La fel ca si in clasa Panou si aici avem o metoda de start pentru pornirea Thread-urilor .

Cand am descris clasa Panou am pomenit de “wait time” . In aceasta clasa avem o metoda cu care putem sa scoatem acest timp de asteptare pe fiecare casa .

```
public int getWaitTime(){//wait time pentru fiecare casa
    int rez=0;
    for(Client t:list)rez+=t.getServiceTime();
    return rez;
}
```

Metoda care imi implemteaza functionarea thread-ului este run . Aici , la fel ca si in clasa de Panou se proceseaza elementele listei , in cazul aceste clase , lista de clienti.

Metoda are un while cu care validam functionarea thread-ului in functie de simTime care este o variabila statica din Panou. Fiecare element din casa va fi sters dupa ce va fi servit , de acea lasam un sleep pe timpul de servire.

Ca sa nu se faca race intre thread-uri pe lista se foloseste un bloc de sincronizare in care se fac actiunile pe lista .

```

public void run() {
    while (Panou.simTime>0) {
        if (list.size() > 0) {
            try {
                Thread.sleep(list.get(0).getServiceTime() * 1000/
GUI.SPEED +1); //timpu de servire in
                // functie de Speed selectat de utilizator
            } catch (InterruptedException e) {
            }
            if(Panou.curentTime>Panou.simTime)break;
            synchronized (list){
                Client c=this.remove(); //se sterge din casa primul client
servit
                javafx.application.Platform.runLater( () ->
GUI.err.appendText(Panou.curentTime + ": Client " + c.getId() +
                    " iese de la casa " +c.getCasa()+ "\n")); // se
adauga in TextArea stergerea din casa
            }
        }
    }
}

```

Client

Este clasa care retine despre fiecare client . Are ca variabile de clasa un id care este necesar pentru recunoasterea clientului care iese si intra la o anumita casa , casa la care a fost repartizat acest client in urma generarii , timpul de servire si timpul de asteptare.

Clasa Client contine doar gettere.

Strategie

Am ales ca clasele pentru strategie sa fie in propriul lor pachet pentru ca asa se pot observa mult mai usor strategiile implementate si modificarile care se pot aduce unei strategii.

Strategiile implementate sunt : RandomStrategy care imi returneaza aleator un indice pentru o casa in care va fi generat un nou client, MinClient care merge pe lista si imi ia pozitia casei cu cel mai mic numar de clienti si MinWait care merge pe lista si imi ia pozitia casei cu cel mai mic waiting time .

În acest pachet se mai pot aduce îmbunătățiri și se pot implementa strategii noi pentru plasarea cât mai eficientă a clienților la coadă.

GUI

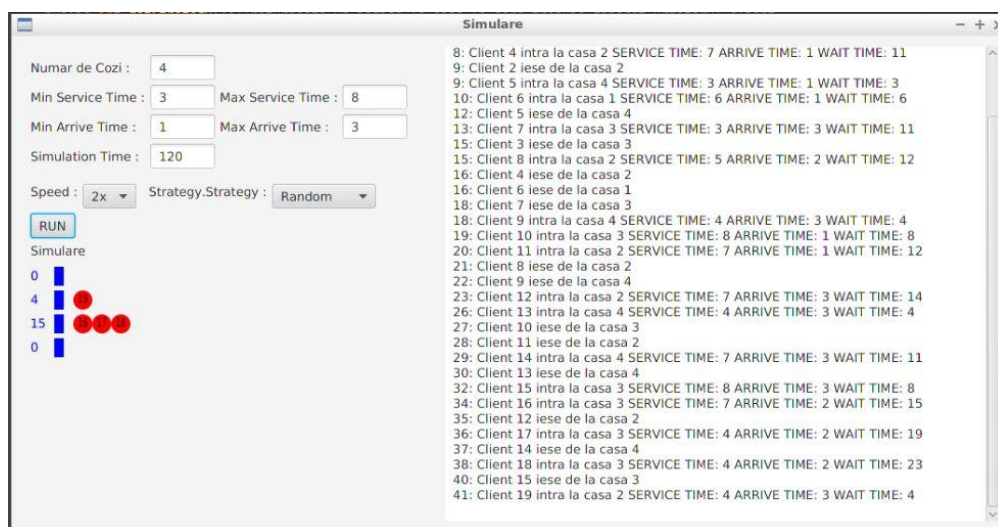
Pachetul de GUI conține clasa GUI care este cea care mă ajută să implementez interfața grafică a acestui proiect.

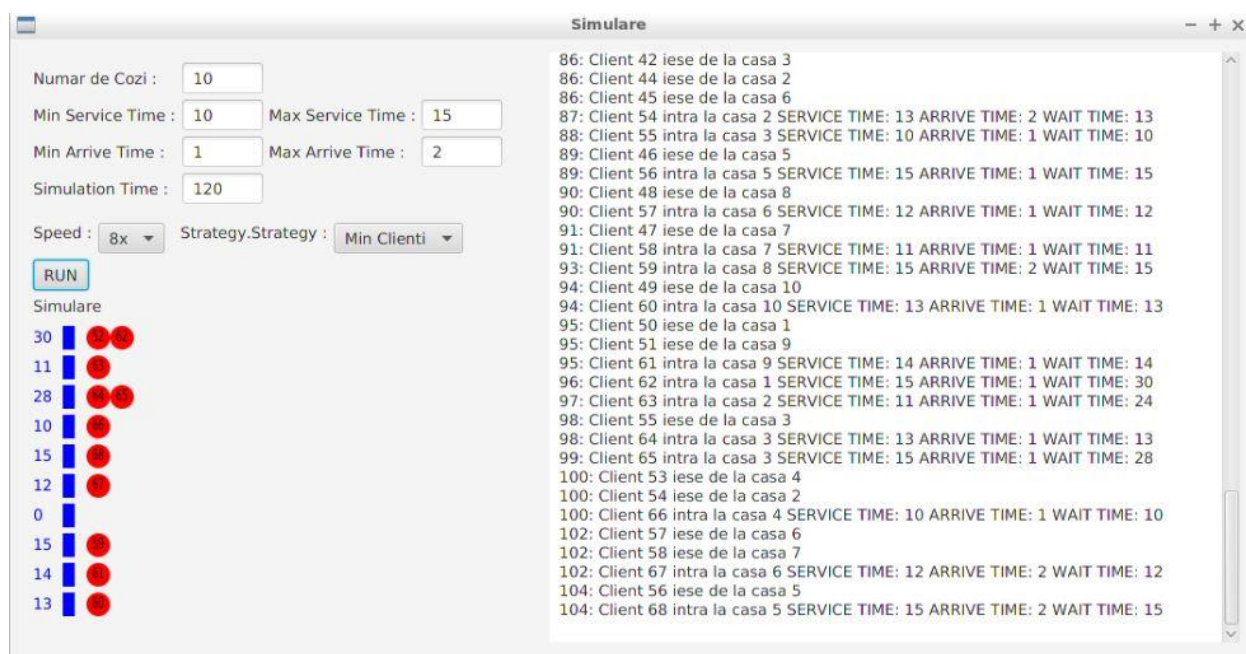
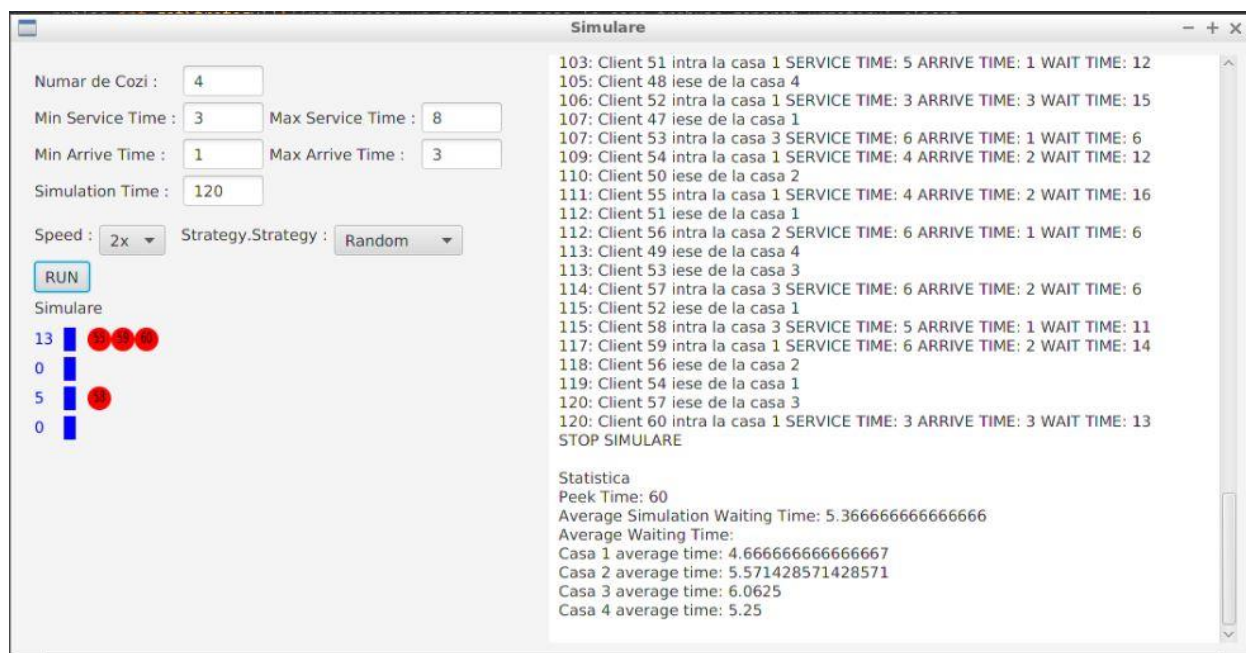
Pentru fiecare element care trebuie specificat în simulare s-a adăugat un `TextField` sau un `ChoiceBox`. Pentru funcționarea corectă a simulatorului este obligatorie scrierea în fiecare `TextField` a valorii dorite pentru simulare. Există pe numărul cozilor o limită, nu este permis mai mult de 10 case într-o simulare. Orice nerespectare a acestei convenții va semnaliza un mesaj de eroare și simularea nu va avea loc.

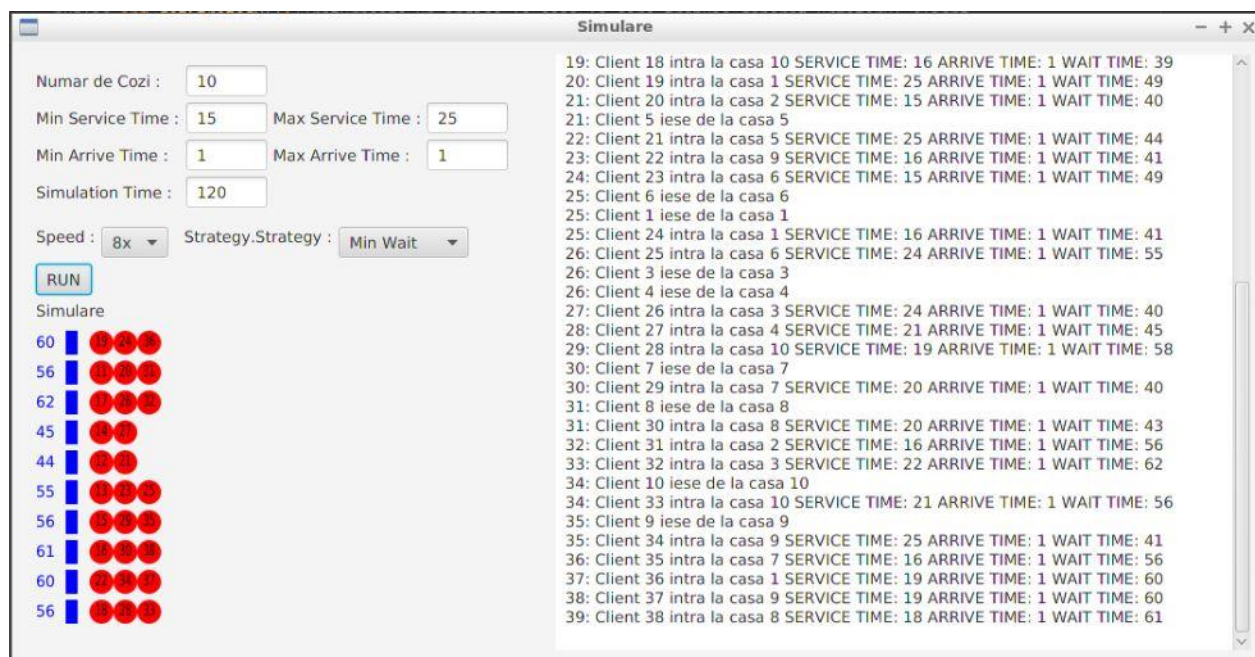
În clasa GUI au fost necesare implementarea a 4 metode :

- start , metoda care se ocupă de stage ,scene și de nodurile interfeței
- intValues ,metoda care inițializează fiecare componentă de clasă
- render , metoda prin care se poate vizualiza simularea acestor cozi de clienți
- handler , metoda care face legătura între interfața și utilizator . De aici se acționează butonul interfeței.

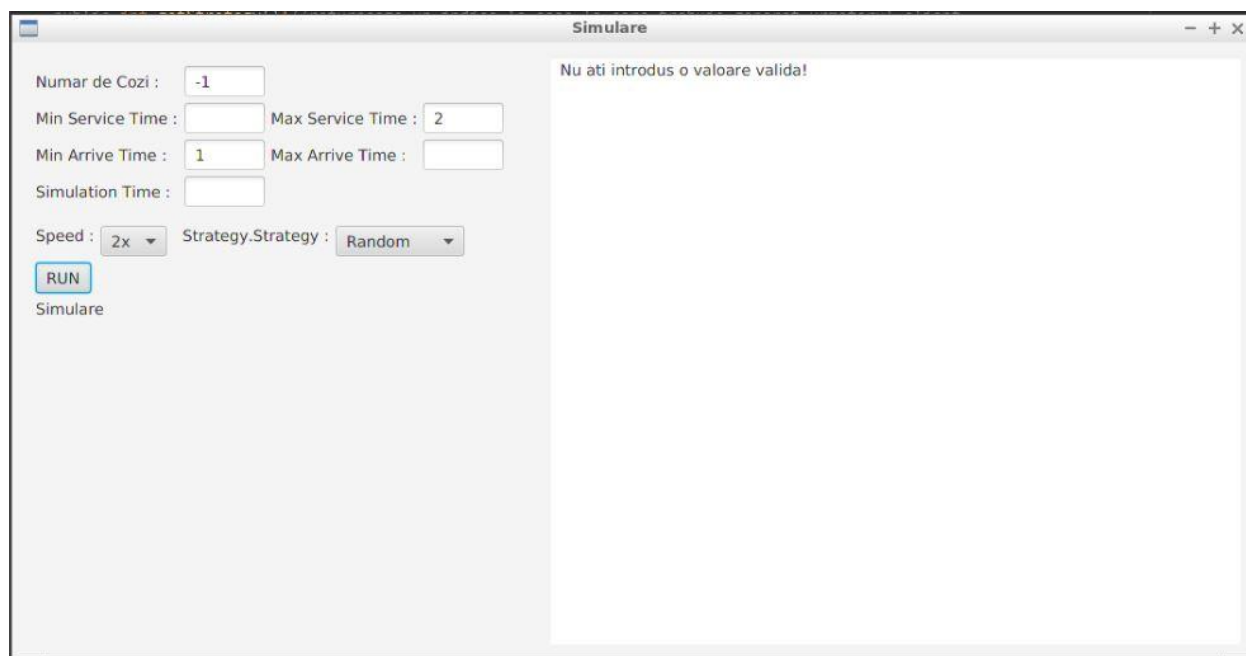
5. Rezultate

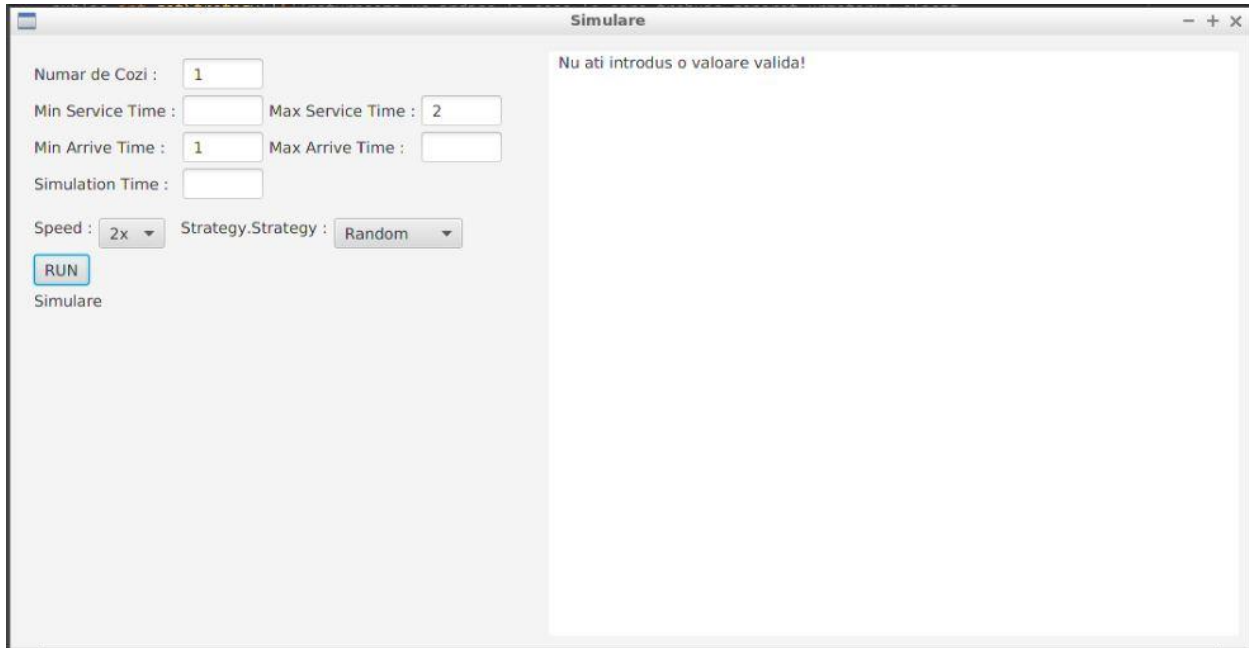




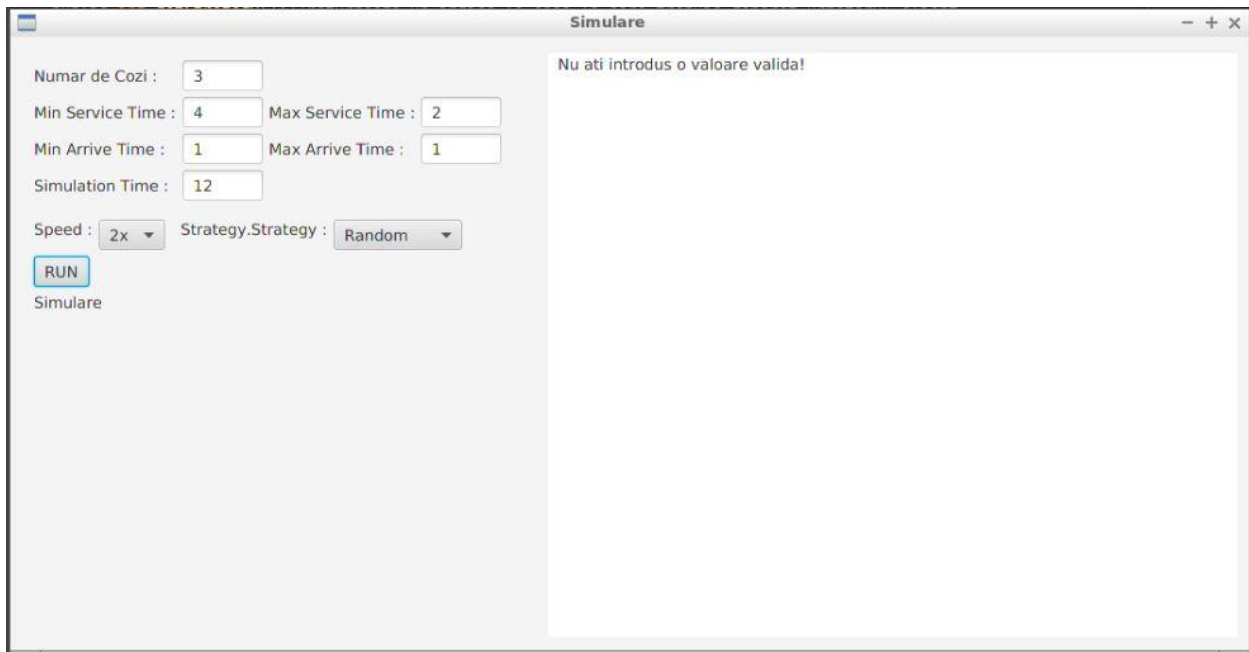


Mesaje De EROARE





The screenshot shows the 'Simulare' application window. On the left, there are input fields for simulation parameters: 'Numar de Cozi' (1), 'Min Service Time' (empty), 'Max Service Time' (2), 'Min Arrive Time' (1), 'Max Arrive Time' (empty), and 'Simulation Time' (empty). Below these are dropdown menus for 'Speed' (2x) and 'Strategy.Strategy' (Random), followed by a blue 'RUN' button and the label 'Simulare'. On the right, a large text area displays the message 'Nu ati introdus o valoare valida!'.



The screenshot shows the 'Simulare' application window with updated settings. The input fields now contain: 'Numar de Cozi' (3), 'Min Service Time' (4), 'Max Service Time' (2), 'Min Arrive Time' (1), 'Max Arrive Time' (1), and 'Simulation Time' (12). The 'Speed' dropdown is still 2x and 'Strategy.Strategy' is still Random. The 'RUN' button and 'Simulare' label are present. The right text area still displays 'Nu ati introdus o valoare valida!'.

6. Concluzii si Dezvoltari Ulterioare

Aplicatia este utila pentru corporati care au probleme cu aglomerati la ghisee sau firmelor care intampina probleme cu clienti nemultumiti datorita timpului de asteptare.

Fiind un simulator se poate observa cum o diferita strategie de asezare a clientilor la cozi poate eficientiza timpul de asteptare si de a crea mult mai multe benefici clientilor.

Cel mai important lucru de vazut la aceasta aplicatie pot fi posibilitatile de dezvoltare ulterioara cum ar fi implementarea mai multor strategii, optimizarea codului pentru a putea genera mai multe cozi de clienti.

Aceasta aplicatie ma ajutat sa aprofundez cunostintele mele de multi-threading si sa invat despre thread-uri in java .

7. Bibliografie

<https://www.journaldev.com/2377/java-lock-example-reentrantlock>
<http://www.oracle.com/technetwork/java/javafx/overview/index.html>