Simularea cozilor

Universitatea Tehnică Cluj-Napoca

Facultatea de Calculatoare și Automatică

Departamenul de Calculatoare și Tehnologia informației

Nume: Muresan George

Grupa: 30226

Materie: Tehnici de programare1.Obiectivul temei

Cerinta proiectului este: „Propuneti si implementati o aplicatie de simulare pentru a analiza sistemele bazate pe cozi si pentru a reduce timpul de asteptare al clientilor de exemplu intr-un supermarket”.

Obiectivul temei este de a implementa o aplicatie cu interfata grafica pentru a fi folosita ca simulator de cozi. Acest simulator este implementat folosind limbajul de programare Java si conceptele de programare orientata obiect.

Coada este o structură de date abstractă, pentru care operatia de inserare a unui element se realizează la un capăt, în timp ce operatia de extragere a unui element se realizează la celălalt capăt.

Coada este un tablou de date cu proprietatea FIFO (First In First Out) avand ca operatii specifice adaugarea (se realizeaza intodeauna ca ultim element al cozii) si scoaterea din coada (intotdeauna va fi servit primul client din coada). In esenta, primul client care se aseaza la coada va fi primul client servit. De asemenea clientii se pot aseza doar la sfarsitul cozii.

Cozile sunt frecvent intalnite atat in lumea reala cat si in lumea informaticii. Obiectivul pricipal al unei cozi este de a oferi un loc pentru client in asteptarea unui anumit serviciu. Aplicatia urmareste minimizarea timpului de asteptare.

2.a) Analiza problemei

În partea de analiză a problemei trebuie depistate principalele clase, respectiv caracteristicile și funcționlitățile acestora și realizate legăturile dintre ele dar si implementarea opeartiilor. Programarea orientata pe obiecte oferă avantajul de a putea începe dezvoltarea unui proiect folosind doar informațiile de la suprafață, fără a fi nevoie de implementarea efectivă a funcționalităților. Această strategie este cunoscută sub numele “Top-Down”. Ea este foarte avantajoasă din punctul de vedere al găsirii componentelor constituente, deoarece pot fi găsite, relativ ușor, structuri cu o legătură directă în lumea reală( obiecte, acțiuni etc.). Din păcate această versatilitate vine cu prețul complexității, ea crescând spre măsură ce se avansează pe nivelele inferioare.In subiectul de fata trebuie sa analizam rezolvarea cozilor dintr-un magazin.

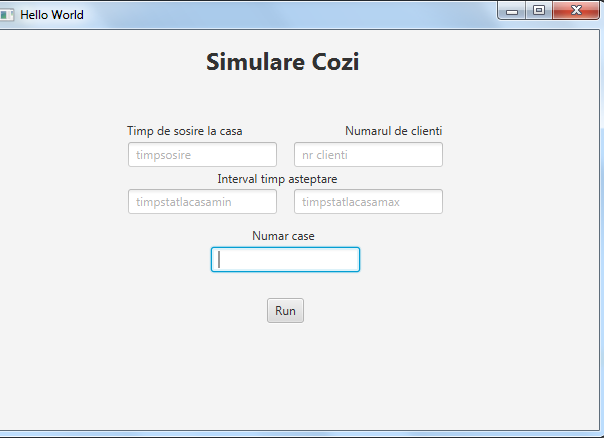
Cozile sunt tratate asemanator atat in lumea reala cat si in lumea virtuala. Scopul unei cozi este de a oferi o ordine de servire a clientilor in functie de timpul la care acestia s-au asezat la coada. Principalul obiectiv al cozilor este de a reduce timpul de asteptare al clientilor.

Fiecare client este identificat prin timpul de servire.

2.b) Modelarea

In partea de modelare interfața cu utilizatorul este stratul cel mai de deasupra al unei aplicații, de aceea realizarea ei a reprezentat punctul de pornire al acestui proiect. În primul rând, ea trebuie să fie cât mai intuitivă și permisivă în ceea ce privește modul de introducere a datelor.

Dupa cum putem observa, datele de intrare si de iesire reies foarte bine din imaginea de mai jos .Pentru a putea duce la bun sfarsit aceasta problema: a timpului de asteptare a clientilor intr-un magazin trebuie sa abordam functionarea cozilor astfel ca, functionarea unei cozi se bazeaza pe principiul FIFO (First In First Out), care permite introducerea unui element exclusiv la finalul cozi, in timp ce deservirea si scoaterea unui element din coada se realizeaza pe primul element al cozii.



Pentru efectuarea propriu-zisa a cozilor avem nevoie sa cunoastem ce inseamna conceptul de thread. Threadul (firul de execuție) definește cea mai mică unitate de procesare ce poate fi programată spre execuție de către [sistemul de operare](https://ro.wikipedia.org/wiki/Sistem_de_operare). Este folosit în programare pentru a eficientiza execuția programelor, executând porțiuni distincte de cod [în paralel](https://ro.wikipedia.org/wiki/Calcul_paralel) în interiorul aceluiași [proces](https://ro.wikipedia.org/w/index.php?title=Proces_(sisteme_de_operare)&action=edit&redlink=1). Câteodata însă, aceste portiuni de cod care constituie corpul threadurilor, nu sunt complet independente și în anumite momente ale execuției, se poate întampla ca un thread să trebuiască să aștepte execuția unor instructiuni din alt thread, pentru a putea continua execuția propriilor instrucțiuni. Această tehnică prin care un thread asteaptă execuția altor threaduri înainte de a continua propria execuție, se numește sincronizarea threadurilor.

Un fir de execuţie se poate afla în Java în mai multe stări, în funcţie de ce se întâmplă cu el la un moment dat. Atunci când este creat, dar înainte de apelul metodei start, singurele metode care se pot apela pentru firul de execuţie sunt metodele start şi stop. Metoda start lansează firul în execuţie prin apelul metodei run . Metoda stop omoară firul de execuţie încă înainte de a fi lansat. Dacă este apelata metoda start pentru firul de execuţie va trece în starea running . În această stare, instrucţiunile din corpul metodei run se execută una după alta.

Execuţia poate fi oprită temporar prin apelul metodei sleep care primeşte ca argument un număr de milisecunde care reprezintă intervalul de timp în care firul trebuie să fie oprit. După trecerea intervalului, firul de execuţie va porni din nou.

Dacă dorim oprirea firului de execuţie pe timp nedefinit, putem apela metoda suspend. Aceeaşi stare este folosită şi pentru oprirea temporară cu sleep. În cazul apelului suspend însă, execuţia nu va putea fi reluată decât printr-un apel al metodei resume. După acest apel, firul va intra din nou în starea running .

O altă cale de a ajunge în starea not running este aceea de a apela o metodă sau o secvenţă de instrucţiuni sincronizată după un obiect. În acest caz, dacă obiectul este deja blocat, firul de execuţie va fi oprit până în clipa în care obiectul cu pricina apelează metoda notify sau notifyAll.

Deoarece tema proiectului este aceea de “simulare a cozilor”, urmatoarele clase care se contureaza sunt :



2.c) Scenarii

Utilizarea normala a aplicației presupune capabilitatea user-ului de a citi .

Scenariile care pot sa apară și sa ducă la oprirea din funcționare a aplicatiei sau eventual la probelme de implementare pot fii:

* Introducerea de text, fără semnificație numerică, de exemplu :”dahsidgsai”
* De asemenea, este interzisa introducerea unui interval ce are timpul de servire 0, deoarece se considera ca fiecare client are nevoie de un timp pentru a fi servit.
* La introducerea datelor sunt analizate toate cazurile care ar putea genera exceptii si sunt excluse acele cazuri in care intervalele specificate nu sunt valide.

2.d) Cazuri de utilizare

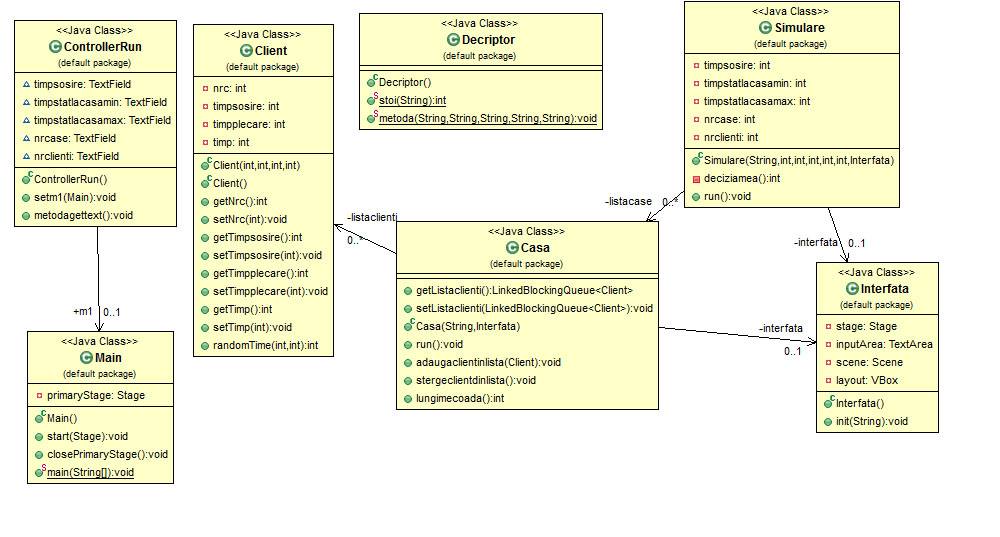
Aplicația se poate folosi pentru:

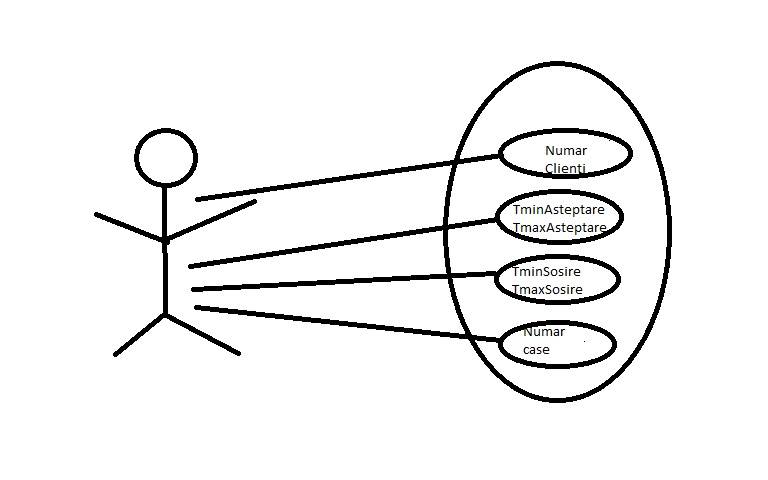
* Simularea cozilor dintr-un magazin
* Simularea cozilor pentru cumpararea unor bilete de tren.

3.Proiectare

**Interfața grafică** este realizată cu JavaFX Scene Builder gasit in IntelliJ(una dintre ele), iar cealalta este realizata manual. Acesta generează un fișier .fxml care este introdus în proiect. ActionEvent-urile ale fiecărui buton sunt setate în JavaFX Scene Builder prin introducerea în câmpul “On Action” numele metodei care trebuie apelata și id-ul Text field-urilor, în câmpul “id”.

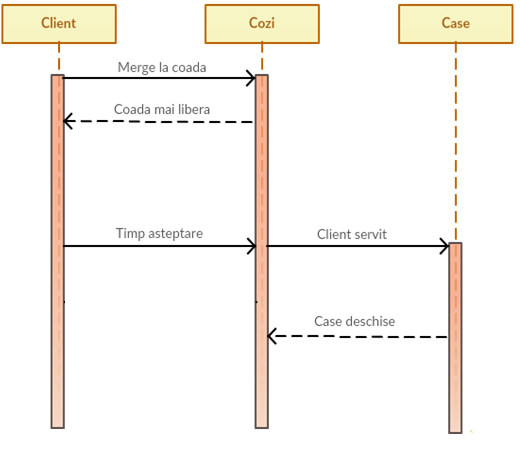
Dupa cum puteti observa, mai jos este reprezentata **diagrama UML** a claselor .

Diagrama **use case:**



**actor**

**Diagrama de secventa:**



Aplicatia mea foloseste 7 clase, fiecare fiindu-mi necesara pentru indeplinirea temei si mai exact “simularea cozilor “.

**Clasa Client** este una dintre cele mai esentiale clase.Fiecare client are un id, timpul de sosire la casa,timpul de plecare de la casa si timpul care este nevoie sa astepte pana este servit, toate acestea fiind declarate ca si int-uri.Tot aici avem constructorul, get-erele si set-erele adecvate.

**Clasa Casa** este urmatoare clasa pe care am implementat-o si ea contine doar o lista de clienti si o interfata creata manual pentru implementarea si afisarea EventLog-ului.Aceasta clasa extinde clasa Thread.Lista de clienti este de tipul LinkedBlockingQueue.In aceasta clasa am ales sa implementez mai multe metode pe care am sa vi le prezint mai tarziu .

**Clasa Simulare** este clasa unde imi pornesc atat casele cat si clienti.Toate atributele pe care le am aici sunt prezentate si in interfata grafica, mai exact private List<Casa> listacase=new ArrayList<Casa>(100); private int timpsosire;private int timpstatlacasamin;private int timpstatlacasamax;private int nrcase;private int nrclienti; private Interfata interfata., Am aici o lista de case, interfata grafica create manual dar si toate datele de intrare.

**Clasa Decriptor** este o clasa cu cateva metode implementate.Una dintre ele schimba din string in int si cealalta este metoda prin care iau datele de intrare din interfata si pornesc simularea.

**Clasa Interfata** este o clasa simpla prin care imi generez o interfata pentru a putea afisa EventLog-ul, in care implementez o metoda pe care o voi folosi si in celelalte clase (Casa si Simulare) care pur si simplu imi ia un text si mi-l pune intr-un textArea.

**Clasa ControllerRun** este clasa specifica primei interfete in care introduce doar datele de intrare si pe care o voi inchide atunci cand voi apela o metoda care se activeaza printr-un buton.

**Clasa Main** este clasa in care imi pornesc prima interafata grafica si in care am o metoda care mi-o si inchide.

4. Implementare si testare

Bun.Ca prima metoda implementata am ales metoda **randomTime** care imi genereaza un numar random intre 2 valori pe care le primeste ca si parametru.Putem observa aceasta metoda in Clasa Client prezentata mai jos :

**public class** Client {  
 **private int nrc**;  
 **private int timpsosire**;  
 **private int timpplecare**;  
 **private int timp**;  
  
 **public** Client(**int** nrc, **int** timpsosire, **int** timpplecare, **int** timp) {  
 **this**.**nrc** = nrc;  
 **this**.**timpsosire** = timpsosire;  
 **this**.**timpplecare** = timpplecare;  
 **this**.**timp** = timp;  
 }  
 **public** Client()  
 {}  
  
 **public int** getNrc() {  
 **return nrc**;  
 }  
  
 **public void** setNrc(**int** nrc) {  
 **this**.**nrc** = nrc;  
 }  
  
 **public int** getTimpsosire() {  
 **return timpsosire**;  
 }  
  
 **public void** setTimpsosire(**int** timpsosire) {  
 **this**.**timpsosire** = timpsosire;  
 }  
  
 **public int** getTimpplecare() {  
 **return timpplecare**;  
 }  
  
 **public void** setTimpplecare(**int** timpplecare) {  
 **this**.**timpplecare** = timpplecare;  
 }  
  
 **public int** getTimp() {  
 **return timp**;  
 }  
  
 **public void** setTimp(**int** timp) {  
 **this**.**timp** = timp;  
 }  
  
 **public int randomTime(int minservtime,int maxservtime )  
 {  
 int randtimp=minservtime + (int)(Math.*random*() \* ((maxservtime - minservtime) + 1));  
 return randtimp;  
 }**  
}

Urmatoare metoda implementata este adaugare de clienti in lista, aceasta metoda se numeste **“adaugaclientiinlista”** si primeste ca parametri, un client, adaugarea fiind facuta cu metoda predefinita a listelor .add, din cate putem observa mai jos avem si notifyAll care anunta thread-urile sa “se trezeasca”.

O alta metoda implementata este “**stergeclientdinlista”** care nu are niciun parametru, este de tipul synchronized ca si **“adaugaclientiinlista”,** putem observa si aici ca atunci cand lista este goala thrreadul sa astepte 0 secunde, iar daca nu sa-mi stearga clientul din capatul cozii, listaclienti.peek() ne duce exact la ultimul client de la coada, tot aici afisam pe ecran dar si in interfata si bininteles anutam threadurile cu notifyAll;

O alta metoda implementta este **“lungimecoada**”, care imi returneaza cati clienti am la o coada, putem observa ca este tot de tipul synchronized.

Dupa cum bine stim threadurile trebuie pornite prin metoda **“run”** care aici va incerca sa stearga clienti din coada si va astepta atata timp cat un client este nevoit sa stea pan ace este servit.

Toate aceste metode le putem observa mai jos, in clasa Casa:

**public class** Casa **extends** Thread {  
 **private** LinkedBlockingQueue<Client> **listaclienti**= **new** LinkedBlockingQueue<>(100);  
 **private** Interfata **interfata**;  
 **public** LinkedBlockingQueue<Client> getListaclienti() {  
 **return listaclienti**;  
 }  
  
 **public void** setListaclienti(LinkedBlockingQueue<Client> listaclienti) {  
 **this**.**listaclienti** = listaclienti;  
 }  
  
 **public** Casa(String name,Interfata interfata) {  
 **super**(name);  
 **this**.**interfata** = interfata;  
 *//this.listaclienti = listaclienti;* }  
 **public void run()  
 {  
 try{  
 while(true)  
 {  
 Client c1=listaclienti.peek();  
 if(c1!=null)  
 {  
 int timpdeasteptare = c1.getTimp();  
  
 *sleep*(timpdeasteptare \* 1000);  
  
 stergeclientdinlista();  
 }  
 }  
  
 }  
 catch(InterruptedException e)  
 {  
 System.*out*.println("Intrerupere casa ");  
 }  
 }**  
 **public synchronized void adaugaclientinlista(Client c )  
 {  
 listaclienti.add(c);  
 notifyAll();  
 }  
 public synchronized void stergeclientdinlista() throws InterruptedException  
 {  
 while(listaclienti.size()==0)  
 wait();  
 Client cl=listaclienti.peek();  
 listaclienti.remove(cl);  
 System.*out*.println("Clientul "+cl.getNrc()+" a parasit "+getName());  
 interfata.init("Clientul "+cl.getNrc()+" a parasit "+getName());  
 notifyAll();  
  
 }  
 public synchronized int lungimecoada(){  
 notifyAll();  
 int sizecoada=listaclienti.size();  
 return sizecoada;  
 }  
}**

Urmatoare clasa pe care am sa v-o prezint este clasa Simulare, in care putem observa metodele **“deciziamea”** care returneaza pozitia unde am cea mai scurta coada(a cata casa).Cum am implementat-o ? Simplu, am presupus ca prima casa are cei mai putini clienti, dupa care am verificat fiecare casa, iar daca am gasit o casa cu o coada mai mica am retinut acea casa .

O alta metoda implementata aici este metoda **“run”** unde ii ordonam thread-ului ce sa incerce sa faca, mai exact atata timp cat mai sunt clienti el sa creeze clienti care vor primi timpii necesari si parametrii necesari, dupa care sa-I adauge in lista de case care are fiecare o lista de clienti si il punem sa “astepte” timpul pe care il primesc clienti cat timp de sosire la casa .Tot aici afisam pe ecran dar si in interfata creata manual prin metoda **“init”** pe care am sa v-o prezint urmatoarea.

Acestea le putem observa mai jos :

**public class** Simulare **extends** Thread {  
 **private** List<Casa> **listacase**=**new** ArrayList<Casa>(100);  
 **private int timpsosire**;  
 **private int timpstatlacasamin**;  
 **private int timpstatlacasamax**;  
 **private int nrcase**;  
 **private int nrclienti**;  
 **private** Interfata **interfata**;  
  
 **public** Simulare(String name, **int** timpsosire, **int** timpstatlacasamin,**int** timpstatlacasamax, **int** nrcase, **int** nrclienti,Interfata interfata) {  
 **super**(name);  
 *//this.listacase = listacase;* **this**.**timpsosire** = timpsosire;  
 **this**.**timpstatlacasamin** = timpstatlacasamin;  
 **this**.**timpstatlacasamax**=timpstatlacasamax;  
 **this**.**nrcase** = nrcase;  
 **this**.**nrclienti** = nrclienti;  
 **this**.**interfata** = interfata;  
 **for**(**int** i=0;i<nrcase;i++)  
 {  
 Casa c1=**new** Casa(**"casa"**,interfata);  
 **listacase**.add(c1);  
 c1.start();  
 }  
 }  
  
  
 **private int deciziamea()  
 {  
 int min;  
 min=listacase.get(0).lungimecoada();  
 int poz=0;  
 for(int i=1;i<nrcase;i++)  
 {  
 int lungime=listacase.get(i).lungimecoada();  
 if(lungime<min) {  
 min = lungime;  
 poz=i;  
 }  
 }  
 return poz;*//retin pozitia unde am cea mai scurta coada* }**  
  
 **public void run()  
 {  
 try  
 {  
 int i=0;  
 while(i<nrclienti)  
 {  
 int cpoz=deciziamea();  
 i++;  
 Client cl=new Client();  
 cl.setNrc(i);  
 cl.setTimp(cl.randomTime(timpstatlacasamin,timpstatlacasamax));  
 cl.setTimpsosire(cl.randomTime(timpsosire,timpsosire));  
 cl.setTimpplecare(-1);  
  
 System.*out*.println("Clientul "+cl.getNrc()+" a fost adaugat la casa "+Integer.*toString*(cpoz));  
 interfata.init("Clientul "+cl.getNrc()+" a fost adaugat la casa "+Integer.*toString*(cpoz));  
  
  
 System.*out*.println("Clientul "+cl.getNrc()+" sta la casa "+cl.getTimp()+" secunde");  
 interfata.init("Clientul "+cl.getNrc()+" sta la casa "+cl.getTimp()+" secunde");  
  
  
 System.*out*.println();  
  
 listacase.get(cpoz).adaugaclientinlista(cl);  
 *sleep*(cl.getTimpsosire()\*1000);  
  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}**

Dupa cum v-am promis urmatoare metoda pe care v-o arat este **“init”,** prezenta in clasaInterfata atasata mai jos.Din cate puteti observa metoda init are ca parametru un string pe care il primeste si il afiseaza intr-un textArea din interfata.

**import** javafx.geometry.Insets;  
**import** javafx.scene.Scene;  
**import** javafx.scene.control.TextArea;  
**import** javafx.scene.layout.VBox;  
**import** javafx.stage.Stage;  
  
  
**public class** Interfata {  
 **private** Stage **stage**;  
 **private** TextArea **inputArea**;  
 **private** Scene **scene**;  
 **private** VBox **layout**;  
 **public** Interfata(){  
 **stage** = **new** Stage();  
 **stage**.setTitle(**"Simulare"**);  
 **inputArea** = **new** TextArea();  
 **inputArea**.setText(**"-------------------Chestii de afisat---------------"**);  
 **layout** = **new** VBox(10);  
 **layout**.setPadding(**new** Insets(20, 20, 20, 20));  
 **layout**.getChildren().addAll(**inputArea**);  
 **scene** = **new** Scene(**layout**,500,500);  
 **stage**.setScene(**scene**);  
 **stage**.show();  
 }  
  
 **public void init(String s){  
 this.inputArea.appendText("\n" + s);  
 }**}

Clasa Decriptor are doar 2 metode pe care le puteti observa mai jos, metoda **“stoi”** primeste ca parametru un string si returneaza in integer, am facut-o cu try catch pentru cazul in care utilizatorul introduce date incorecte.Cealalta metoda se numeste chiar **“metoda”** din lipsa de inspiratie, aceasta primeste mai multe stringuri ca parametrii pe care cu ajutorul metodei **“stoi”** ii converteste in int, iar tot aici imi creeez un obiect de tipul Simulare care primeste noile date convertite si tot aici pornesc si simularea .Putem observa ca ambele metode sunt statice pentru a putea fi apelate si in alte clase cum ar fii ultima clasa ControllerRun.

**public class** Decriptor {  
  
 **public static int stoi(String s1)  
 {  
 try  
 {  
 int rez=Integer.*parseInt*(s1);  
  
 return rez;  
  
 }  
  
 catch( NumberFormatException e)  
 {  
 System.*out*.println("aloooooooooooooo..... date incorecte");  
 System.*exit*(1);  
  
 }  
 return -1;  
 }**  
 **public static void metoda(String timpsosiree,String timpstatlacasamaxx,String timpstatlacasaminn,String nrcasee,String nrclientii ) throws IOException {  
 int timpsosire=*stoi*(timpsosiree);  
  
 int timpstatlacasamax=*stoi*(timpstatlacasamaxx);  
 int timpstatlacasamin=*stoi*(timpstatlacasaminn);  
  
 int nrcase=*stoi*(nrcasee);  
 int nrclienti=*stoi*(nrclientii);  
  
 Interfata interfata = new Interfata();  
  
 Simulare s1=new Simulare("SIMULATOR",timpsosire,timpstatlacasamin,timpstatlacasamax,nrcase,nrclienti,interfata);  
 s1.start();  
  
 }**}

Clasa ControllerRun este detinatoare peste prima interfata grafica cea din care sunt luate datele de intrare.Putem observa ca aici ne instantiem un obiect de tipul main adica prima noastra interfata cea cu care operam, pe care o vom inchide in metoda **“metodagettext”** unde apelam Decriptor.metoda de stringurile primite din fisierul FXML .

**public class** ControllerRun {  
 @FXML  
 TextField **timpsosire**;  
 @FXML  
 TextField **timpstatlacasamin**;  
 @FXML  
 TextField **timpstatlacasamax**;  
 @FXML  
 TextField **nrcase**;  
 @FXML  
 TextField **nrclienti**;  
  
 **public** Main **m1**;  
 **public void** setm1(Main m1){  
 **this**.**m1** = m1;  
 }  
  
 **public void metodagettext() throws IOException {  
 String s1=timpsosire.getText();  
 String s3=timpstatlacasamax.getText();  
 String s32=timpstatlacasamin.getText();  
 String s2322=nrcase.getText();  
 String s22322=nrclienti.getText();  
 Decriptor.*metoda*(s1,s3,s32,s2322,s22322);  
 m1.closePrimaryStage();  
  
 }**}

5.Concluzii

Ca o concluzie, acest proiect m-a ajutat sa imi consolidez cunostintele de programare orientata pe obiect dobandite in primul semestru si sa imi organizez munca in baza paradigmelor POO de asemenea acest proiect m-a ajutat sa inteleg mult mai bine conceptual de thread pe care il voi folosi si in viitor.

In cadrul dezvoltarilor ulterioare se pot aminti urmatoarele: imbunatatirea interfetei grafice, inchiderea si deschiderea caselor in functie de numarul de clienti, crearea unui algoritm mai eficient pentru distribuirea clientilor la case, folosirea timpului real de sosire a clientilor la case, genererea individuala a caselor si a clientilor etc .

.