Simulator de cozi

**1.Obiectivul temei**

Obiectivul principal al temei este proiectarea si implementarea unei aplicatii pentru analizarea unui sistem de cozi bazata pe determinarea si minimizarea timpului de asteptare al clientului, iar mai apoi aceasta aplicatie va fi folosita de un simulator pentru a testa corectitudinea aplicatiei.

Sub-obiective:

* Analiza problemei si identificarea necesitatilor
* Proiectarea aplicatiei
* Implementarea aplicatiei
* Testarea aplicatiei

**2. Analiza problemei, modelare, scenariu, cazuri de utilizare**

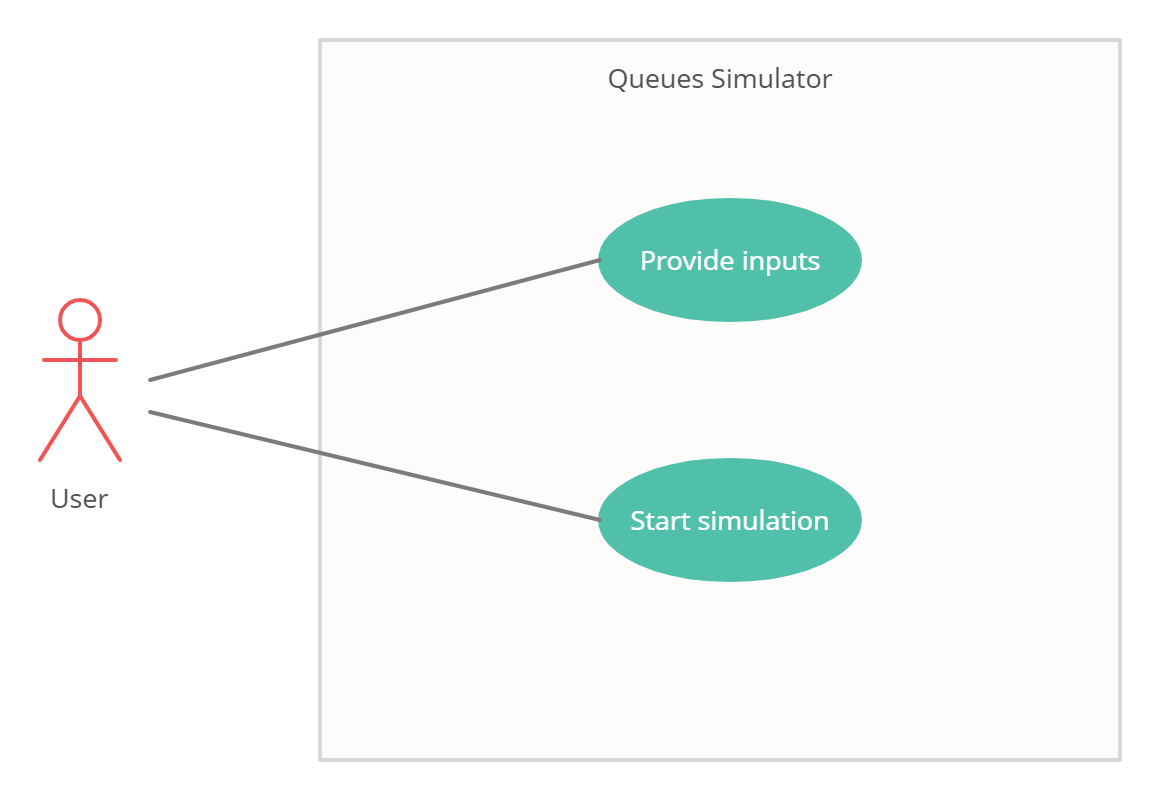
Aplicatia va trebui sa genereze o situatie in care mai multi clienti (care au timpi de sosire diferiti) sunt repartizati la cozi in functie de o anumita strategie, iar odata ajunsi primii in coada vor fi nevoiti sa astepte serviciul dorit (timpi de procesare). Astfel pentru generarea acestor clienti vom avea nevoie de urmatoarele input-uri: numarul de clienti (tasks), numarul de cozi (servers), timp de sosire minim, timp de sosire maxim, timp de procesare minim, timp de procesare maxim si limita de timp (durata simularii).

Pe langa aceste input-uri, utilizatorul va putea selecta strategia dorita pentru repartizarea clientilor in cozi. In aceasta aplicatie se vor realiza doua strategii:

1. Timp de asteptare minim: clientul va fi repartizat la coada cu timplu de asteptare minim
2. Numar de client minim: clientul va fi repartizat la coada cu cei mai putini clienti

Principalele proprietati ale aplicatiei sunt:

* Singurul actor este utilizatorul
* Acesta beneficiaza de functionalitatile simulatorului de cozi prin interfata grafica dedicata
* Utilizatorul va putea introduce de la tastatura imputurile necesare pentru a incepe simularea, acestea fiind specificate in interfata grafica
* Utilizatorul va putea selecta strategia in functie de care clientii se vor aseza la o anumita coada(timp de asteptare minim, numar de clienti minim)
* Va exista un buton de start pentru a incepe simularea
* Simularea va incepe imediat dupa apasarea butonului de start



Caz de utilizare:

Actor principal: utilizatorul

Scenariu:

1. Utilizatorul introduce datele necesare pentru crearea simularii
2. Utilizatorul selecteaza strategia dorita (timp de asteptare minim sau numar de clienti minim)
3. Utilizatorul apasa butonul de start pentru inceperea simularii
4. Aplicatia va porni simularea iar rezultatele acesteia vor fi afisate in interfata grafica. Mai exact, la fiecare secunda se va afisa starea cozilor.

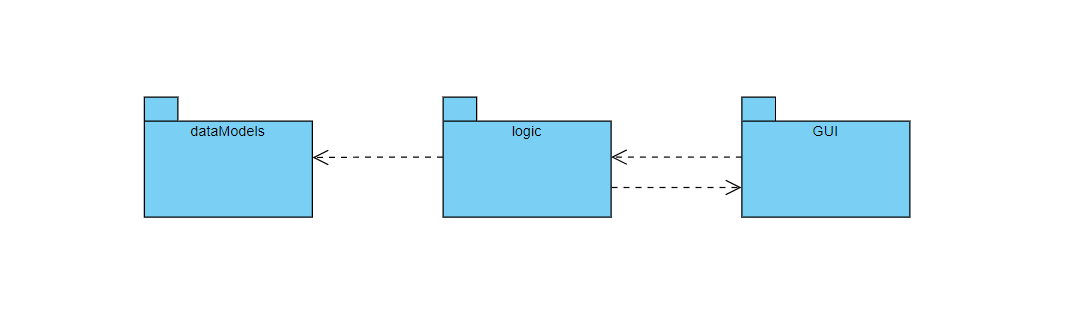
**3.Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, algoritmi, interfata utilizator)**

Se va imparti proiectul in trei pachete: dataModels, logic si GUI.

In pachetul dataModels se vor crea clasele necesare pentru reprezentarea clientilor (Task) si cozilor (Server).

In pachetul logic se va realiza logica simularii, clasa principala fiind SimulationManager.

In pachetul GUI se va realiza interfata grafica care va folosi functionalitatile din logic.



In primul rand pentru a putea identifica un client si a eficientiza modul de repartizare al acestuia vom avea nevoie sa asociem un identificator unic fiecarui client si sa tinem minte timpul de sosire si cel de procesare.

In al doilea rand, o coada va fi conceputa ca un sir de clienti pentru care se va retine timpul de asteptare in fiecare moment al simularii. Fiecare coada va fi un thread separate, in consecinta clasa care va reprezenta cozile va implementa interfata Runnable.

Astfel, pachetul dataModels va contine doua clase: Task (client), Server (coada).

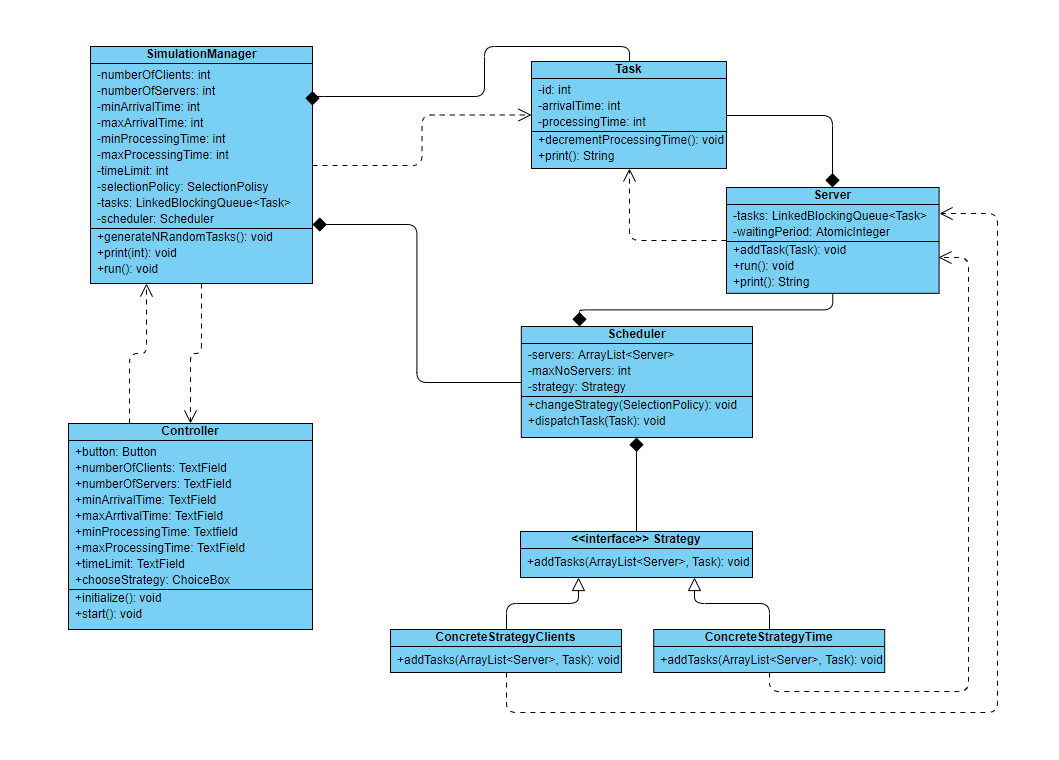
Cum am precizat si mai sus, clasa principala din pachetul logic va fi SimulationManager care va crea propriu-zis simularea. De aici reiese ca aceasta clasa va avea ca atribute input-urile pe care utilizatorul va fi nevoit sa le introduca: numberOfClients, numberOfServers, minArrivalTime, maxArrivalTime, minProcessingTime, maxProcessingTime si timeLimit.

Pe langa aceasta clasa care gestioneaza simularea vom avea nevoie de inca o clasa pentru gestionarea cozilor: Scheduler. Utilitatea acesteia va fi de a lansa cozile(thread-urile) si de a adauga clienti in acestea oricand SimulationManager-ul cere acest lucru.

In final, pachetul logic trebuie sa mai contina o interfata Strategy care va fi implementata in doua moduri: timp de asteptare minim (ConcreteStrategyTime) si numar de clienti minim (ConcreteStrategyClients).

Partea de interfata grafica am ales sa o realizez in JavaFX si cuprinde sapte campuri TextField in care utilizatorul va introduce datele de intrare (numberOfClients, numberOfServers, minArrivalTime, maxArrivalTime, minProcessingTime, maxProcessingTime si timeLimit), un ChoiceBox de unde se va selecta strategia dorita, un buton de Start si un TextArea pentru afisarea starii cozilor la fiecare moment de timp.

Diagrama UML pentru clase:



**4. Implementare**

***Clasa Task:***

Ca atribute, aceasta clasa contine id-ul clientului, timpul de sosire si timpul de procesare, toate trei private.

private int id;  
 private int arrivalTime;  
 private int processingTime;

Metoda print() va fi folosita pentru afisarea in timp real a starii unui client in timpul simularii.

public String print*() {* String s = new String*()*;  
 s = "(" + this.id + "," + this.arrivalTime + "," + this.processingTime + ")";  
  
 return s;  
 *}*

In rest, aceasta clasa mai contine cate functii de get si set care nu necesita explicate.

***Clasa Server:***

Ca atribute, aceasta clasa contine o lista de task-uri (clienti) si un AtomicInteger folosit pentru contorizarea timpului de asteptare. Pentru implementarea listei am folosit LinkedBlockingQueue<> pentru a atomiza operatiile pe aceasta lista, AtomicInteger fiind folosit din acelasi motiv.

private LinkedBlockingQueue*<*Task*>* tasks;  
 private AtomicInteger waitingPeriod;

Aceasta clasa implementeaza interfata Runnable si in consecinta metoda sa principala va fi metoda run(). Odata cu lansarea thread-ului (cozii) in executie se va crea un while() infinit care va astepta clientii iar mai apoi ii va procesa. In timpul procesarii thread-ul va fi adormit pentru o secunda cee ace reprezinta trecerea unei unitati de timp dupa care timpul de procesare al clientului respectiv se va decrementa.

public void run*() {* Task t = null;  
 while*(*true*) {* if *(*t == null*) {* t = tasks.peek*()*;  
 *}* else *{* try *{* Thread.*sleep(*1000*)*;  
 *}* catch *(*InterruptedException e*) {* e.printStackTrace*()*;  
 *}* this.waitingPeriod.decrementAndGet*()*;  
 t.decrementProcessingTime*()*;  
 if *(*t.getProcessingTime*()* == 0*) {* this.tasks.poll*()*;  
 t = null;  
 *}  
 }  
 }  
 }  
  
}*

Metoda print() va fi folosita pentru afisarea in timp real a starii unui client in timpul simularii.

public String print*() {* String s = new String*()*;  
 Object*[]* t = tasks.toArray*()*;  
  
 for*(*int i=0; i<t.length; i++*)* s = s + *((*Task*)*t*[*i*])*.print*()* + " ";  
  
 return s;  
 *}*

Metoda addTask() este folosita pentru a adauga un task nou in lista si totodata pentru a incrementa timpul de asteptare cu timpul de procesare al task-ului.

public void addTask*(*Task newTask*) {* this.waitingPeriod.addAndGet*(*newTask.getProcessingTime*())*;  
 this.tasks.add*(*newTask*)*;  
 *}*

***Clasa SimulationManager:***

Aceasta clasa are ca atribute input-urile pe care utilizatorul va trebui sa le introduca in interfata grafica. Acestea sunt necesare pentru generarea datelor necesare simularii (numarul de clienti (tasks), numarul de cozi (servers), timp de sosire minim, timp de sosire maxim, timp de procesare minim, timp de procesare maxim si limita de timp (durata simularii)). Pe langa aceste input-uri clasa mai are ca atribute o lista de task-uri in care se vor adauga clientii generate, un scheduler pentru a lansa cozile la care vor fi repartizati clientii si un TextArea folosit pentru afisarea in interfata grafica.

private int numberOfClients;  
 private int numberOfServers;  
 private int minArrivalTime;  
 private int maxArrivalTime;  
 private int minProcessingTime;  
 private int maxProcessingTime;  
 private int timeLimit;  
 private SelectionPolicy selectionPolicy;  
  
 private LinkedBlockingQueue*<*Task*>* tasks;  
 private Scheduler scheduler;  
  
 TextArea queues;

Aceasta clasa implementeaza interfata Runnable si in consecinta cea mai importanta metoda este run(). Aceasta metoda este cea care creeaza toate conditile necesare aplicatiei folosind restul claselor. In ea se genereaza clientii, se lanseaza cozile (prin scheduler) si se simuleaza trecerea timpului intr-un while(). Cand se depaseste limita de timp simularea se incheie (toate thread-urile).

public void run*() {* int currentTime = 0;  
 generateNRandomTasks*()*;  
 scheduler = new Scheduler*(*this.numberOfServers, selectionPolicy*)*;  
 while *(*currentTime <= timeLimit*) {* int i = 0;  
 Task t = tasks.peek*()*;  
 while *(*t != null && t.getArrivalTime*()* <= currentTime*) {* scheduler.dispatchTask*(*t*)*;  
 tasks.poll*()*;  
 t = tasks.peek*()*;  
 *}* this.print*(*currentTime*)*;  
  
 try *{* Thread.*sleep(*1000*)*;  
 *}* catch *(*InterruptedException e*) {* e.printStackTrace*()*;  
 *}* currentTime++;  
 *}  
 }  
  
}*

Metoda generateNRandomTasks() este folosita pentru generarea clientilor ale caror timpi de sosire si timpi de procesare trebuie sa respecte anumite conditii (input-urile utilizatorului). La finalul metodei va fi generata o lista de clienti sortata in ordinea timpilor de sosire.

private void generateNRandomTasks*() {* tasks = new LinkedBlockingQueue*<*Task*>()*;  
 Random rand = new Random*()*;  
  
 ArrayList*<*Task*>* createTasks = new ArrayList*<*Task*>()*;  
  
 for*(*int i=0; i<numberOfClients; i++*) {* int processingTime = minProcessingTime + rand.nextInt*(*maxProcessingTime - minProcessingTime + 1*)*;  
 int arrivalTime = minArrivalTime + rand.nextInt*(*maxArrivalTime - minArrivalTime + 1*)*;  
 Task t = new Task*(*i+1, arrivalTime, processingTime*)*;  
  
 createTasks.add*(*t*)*;  
 *}* Collections.*sort(*createTasks, new SortByArrivalTime*())*;  
 for *(*Task t : createTasks*) {* tasks.add*(*t*)*;  
 *}  
 }*

Metoda print() va afisa starea cozilor in interfata grafica in timp real si totodata va salva datele intr-un fisier.

public void print*(*int currentTime*) {* String out = new String*()*;  
  
 String s = new String*()*;  
 Object*[]* t = tasks.toArray*()*;  
 for*(*int i=0; i<t.length; i++*)* s = s + *((*Task*)*t*[*i*])*.print*()* + " ";  
  
 out = "Time " + currentTime + "\n";  
 out = out + "Waiting clients: ";  
 out = out + s + "\n";  
  
 for*(*int i=0; i<this.scheduler.getServers*()*.size*()*; i++*) {* int index = i+1;  
 out = out + "Queue " + index + ": " + this.scheduler.getServers*()*.get*(*i*)*.print*()* + "\n";  
 *}* out = out + "\n";  
  
 queues.appendText*(*out*)*;  
 *}*

***Clasele ConcreteStrategyClient si ConcreteStrategyTime***

Aceste clase implementeaza interfata Strategy si reprezinta implementarile individuale ale celor doua strategii de repartizare a clientilor in cozi: dupa timpul minim de asteptare sau dupa numarul de clienti.

Ambele clase implementeaza doar metoda addTask(). Pentru primul caz se va parcurge lista de cozi si se va adauga clientul in cea cu timpul de asteptare minim.

public void addTask*(*ArrayList*<*Server*>* servers, Task t*) {* int min = servers.get*(*0*)*.getWaitingPeriod*()*.get*()*;  
 int iMin = 0;  
  
 for *(*int i=1; i<servers.size*()*; i++*) {* if*(*min > servers.get*(*i*)*.getWaitingPeriod*()*.get*()) {* min = servers.get*(*i*)*.getWaitingPeriod*()*.get*()*;  
 iMin = i;  
 *}  
 }* servers.get*(*iMin*)*.addTask*(*t*)*;  
 *}*

Iar pentru al doile caz se va parcurge lista de cozi si se va adauga clientul in cea la care asteapta cei mai putini clienti.

public void addTask*(*ArrayList*<*Server*>* servers, Task t*) {* int min = servers.get*(*0*)*.getTasks*()*.size*()*;  
 int iMin = 0;  
  
 for *(*int i=1; i<servers.size*()*; i++*) {* if*(*min > servers.get*(*i*)*.getTasks*()*.size*()) {* min = servers.get*(*i*)*.getTasks*()*.size*()*;  
 iMin = i;  
 *}  
 }* servers.get*(*iMin*)*.addTask*(*t*)*;  
 *}*

***Clasa Scheduler***

Aceasta clasa are ca scopuri lansarea propriu-zisa a cozilor (thread-urilor) in executie, gestionarea strategiei folosite si adaugarea de clienti in cozi in timpul simularii. Astfel atributele sale sunt:

private ArrayList*<*Server*>* servers;  
 private int maxNoServers;  
 private Strategy strategy;

Constructorul clasei realizeaza lansarea cozilor. Astfel in el se vor crea un numar de thread-uri egal cu numarul de cozi. Aceste thread-uri sunt setate ca fiid demoni, pentru ca la finalul simularii sa se opreasca.

public Scheduler*(*int maxNoServers, SelectionPolicy selectionPolicy*) {* this.servers = new ArrayList*<*Server*>()*;  
 this.maxNoServers = maxNoServers;  
 changeStrategy*(*selectionPolicy*)*;  
  
 for *(*int i=0; i<this.maxNoServers; i++*) {* Server s = new Server*()*;  
 servers.add*(*s*)*;  
 Thread server = new Thread*(*s*)*;  
 server.setDaemon*(*true*)*;  
 server.start*()*;  
 *}  
 }*

Metoda chageStrategy() selecteaza strategia de repartizare a clientilor in cozi in functie de alegerea utilizatorului din interfata grafica.

public void changeStrategy*(*SelectionPolicy policy*) {* if*(*policy == SelectionPolicy.*SHORTEST\_TIME)* this.strategy = new ConcreteStrategyTime*()*;  
 if*(*policy == SelectionPolicy.*SHORTEST\_QUEUE)* this.strategy = new ConcreteStrategyClients*()*;  
 *}*

***Clasa Controller***

Aceasta clasa are ca atribute mai multe TextField-uri in care utilizatorul va introduce datele care ii vor fi solicitate de interfata grafica, un TextArea in care se va afisa starea simularii in timp real, un Button care reprezinta butonul de start, si un choiceBox din care se va selecta strategia dorita.

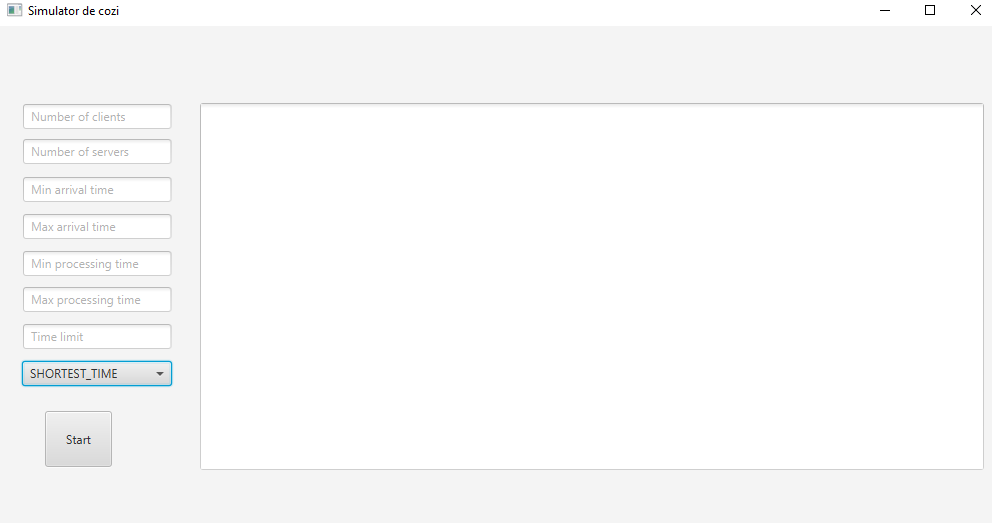
public TextArea queues;  
 public Button button3;  
 public TextField numberOfServers;  
 public TextField numberOfClients;  
 public TextField minArrivalTime;  
 public TextField maxArrivalTime;  
 public TextField minProcessingTime;  
 public TextField maxProcessingTime;  
 public TextField timeLimit;  
 public ChoiceBox chooseStrategy;

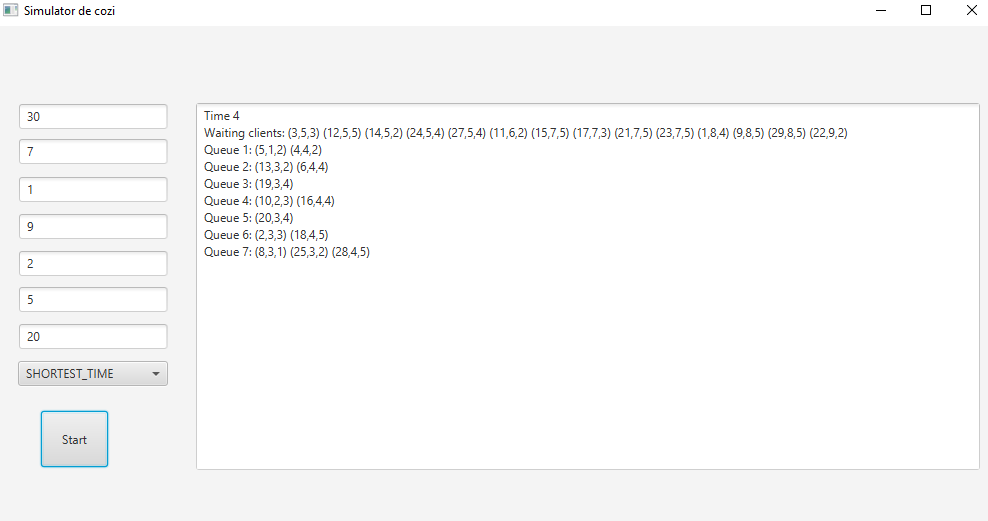
Metoda start() este cea care se va executa la apasare butonului Start din interfata grafica si in ea se trimite in executie un Thread cu implementarea din SimulationManager, deci cu simularea, in functie de strategia selectata de utilizator in choiseBox.

public void start*() {* if*(*chooseStrategy.getSelectionModel*()*.getSelectedItem*()* == choiseList.get*(*0*)) {* SelectionPolicy selectionPolicy = SelectionPolicy.*SHORTEST\_TIME*;  
 SimulationManager s = new SimulationManager*(*Integer.*parseInt(*numberOfClients.getText*())*, Integer.*parseInt(*numberOfServers.getText*())*, Integer.*parseInt(*minArrivalTime.getText*())*, Integer.*parseInt(*maxArrivalTime.getText*())*, Integer.*parseInt(*minProcessingTime.getText*())*, Integer.*parseInt(*maxProcessingTime.getText*())*, Integer.*parseInt(*timeLimit.getText*())*, selectionPolicy, queues*)*;  
 new Thread*(*s*)*.start*()*;  
 *}* else *{* SelectionPolicy selectionPolicy = SelectionPolicy.*SHORTEST\_QUEUE*;  
 SimulationManager s = new SimulationManager*(*Integer.*parseInt(*numberOfClients.getText*())*, Integer.*parseInt(*numberOfServers.getText*())*, Integer.*parseInt(*minArrivalTime.getText*())*, Integer.*parseInt(*maxArrivalTime.getText*())*, Integer.*parseInt(*minProcessingTime.getText*())*, Integer.*parseInt(*maxProcessingTime.getText*())*, Integer.*parseInt(*timeLimit.getText*())*, selectionPolicy, queues*)*;  
 new Thread*(*s*)*.start*()*;  
 *}  
 }*

**5. Rezultate**

Aplicatia a fost finalizata cu succes. Pentru testare am rulat de mai multe ori simularea si am interpretat starile afisate. Interfata grafica face ca aplicatia sa fie user-friendly. In imaginile de mai jos se poate observa functionalitate interfetei grafice.





**6. Concluzie**

In timpul implementarii acestui proiect a trebuit sa caut informatii legate de thread-uri si de utilizarea lor. Proiectul mai poate fi imbunatatit printr-o interfata grafica care sa faca simularea mai usor de urmarit si totodata mai captivanta.

**7. Bibliografie**

* <https://www.w3schools.com/java/java_threads.asp>
* <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingQueue.html>