***DOCUMENTAȚIE***

TEMA 2 TEHNICI DE PROGRAMARE

***Aplicație pentru simularea și***

***gestiunea cozilor***

*Pogăcean Rahela- Alexandra,*

Calculatoare și Tehnologia Informației

Anul II, seria B, grupa 30226

**Cuprins**

1. **Obiective**
   1. **Obiectiv principal**
   2. **Obiective secundare**
2. **Analiza problemei**
   1. **Descriere**
   2. **Scenarii și cazuri de utilizare**
3. **Proiectare**
   1. **Decizii de proiectare**
   2. **Diagrame UML**
   3. **Structuri de date**
   4. **Proiectare clase și algoritmi**
4. **Implementare**
   1. **Metode**
   2. **Interfața utilizator**
5. **Rezultate și concluzii**
6. **Bibliografie**
7. **Obiective**
   1. **Obiectivul principal**

Obiectivul fundamental al acestei lucrări de laborator este proiectarea și implementarea unei aplicații destinate analizei sistemelor bazate pe cozi, pentru a determina și minimiza timpul de așteptare al clienților (cozile din viața reală, de exemplu coada de clienți dintr-un supermarket).

* 1. **Obiective secundare**

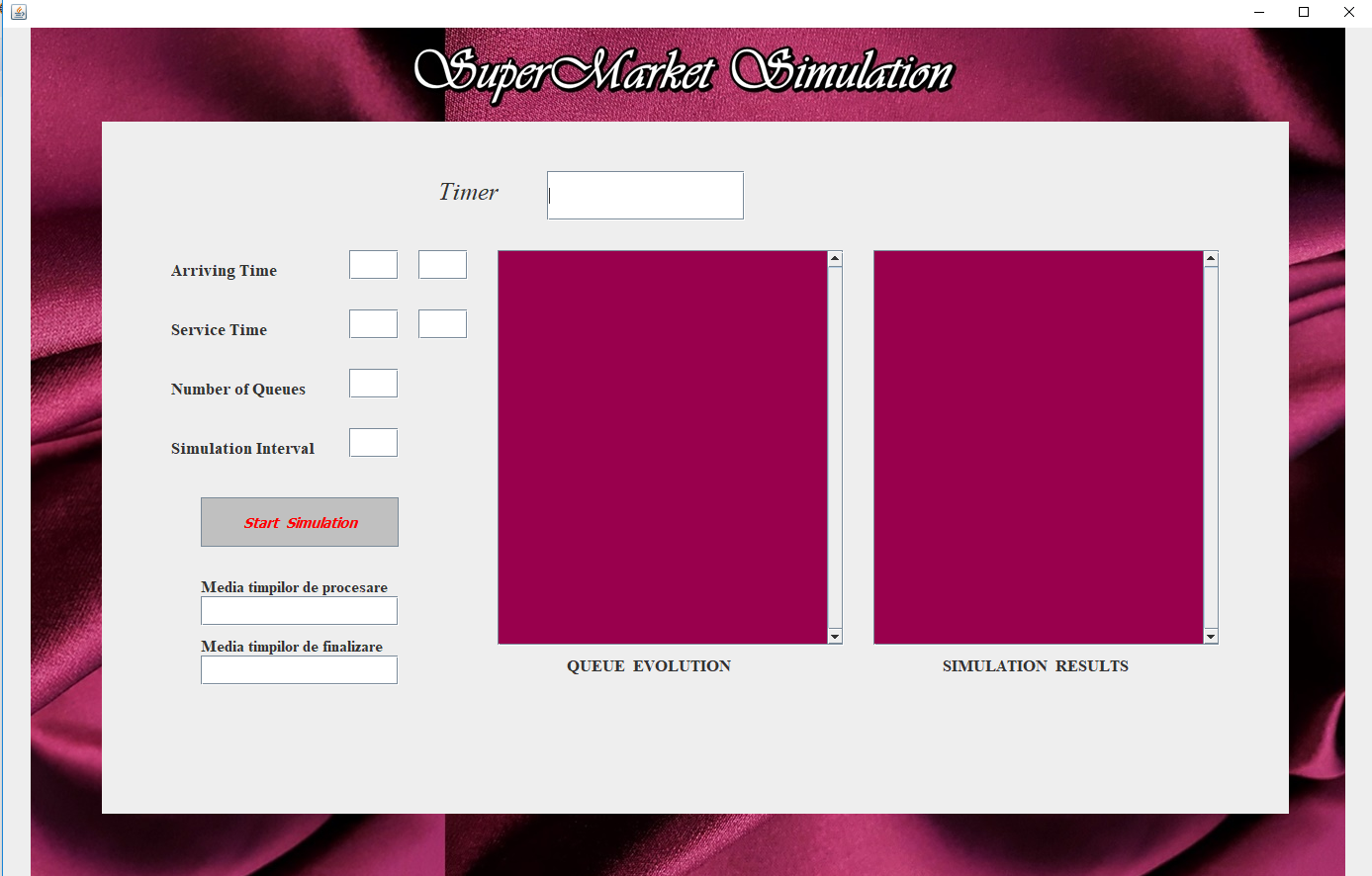
Un prim obiectiv secundar este constituit de respectarea paradigmelor programării orientate pe obiecte, structurarea codului in clase, ceea ce face ca acesta să fie mai lizibil și mai facil de urmărit. Realizarea diagramelor de clase și a use case-urilor reprezintă un alt obiectiv secundar al acestui proiect, precum și dezvoltarea ulterioară, propunându-se o serie de îmbunătățiri ale aplicației pentru o versiune mai complexă.

1. **Analiza problemei**

**2.1 Descriere**

Cozile sunt, cel mai adesea, utilizate pentru a modela domeniile din lumea reală. Obiectivul fundamental al unei cozi este de a oferi și asigura un spațiu pentru client, un loc în care să aștepte înainte de a primi un serviciu. Managementul bazat pe sistemele de gestiune a cozilor este interesat să minimizeze timpul de așteptare al clienților, timp acordat înainte ca aceștia să fie serviți. O modalitate pentru a face acest lucru ar fi adăugarea mai multor cozi, case de marcat destinate servirii, însă aceasta metodă ar presupune o creștere semnificativă a cheltuielilor. În momentul adăugării unui nou server, clienții în așteptare vor fi distribuiți tuturor server-elor disponibile.

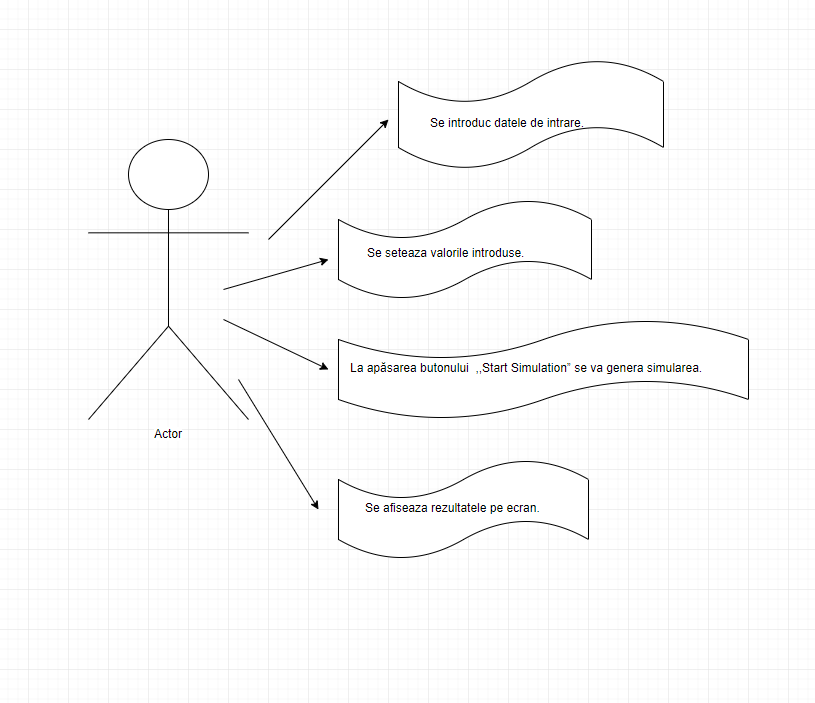
Aplicația ar trebui să simuleze o serie de clienți care sosesc pentru a fi serviți, intră în cozi, așteaptă, sunt serviți și în final, părăsesc coada. Se urmărește timpul pe care consumatorii îl petrec așteptând în cozi, precum și statisticile privind media timpilor de așteptare. Pentru a calcula timpul de așteptare, trebuie să cunoaștem timpul de sosire, timpul de finalizare și cel de servire (procesare). Timpul de sosire și cel de servire depind de fiecare client în parte (momentul de timp în care ei apar și serviciile de care au nevoie). Timpul de finalizare depinde de numărul de cozi, de numărul de clienți din coadă și de necesitățile lor în ceea ce privește servirea.



Datele de intrare pentru această aplicație sunt intervalul minim și intervalul maxim de sosire, timpul minim și timpul maxim de servire, numărul de cozi și intervalul de simulare. Datele de intrare trebuie să fie numere naturale, preferabil nenule. Scopul proiectului este de a crea și dezvolta un simulator care analizează un sistem bazat pe cozi pentru determinarea și reducerea timpului de așteptare al clienților cu ajutorul limbajului Java.

* 1. **Scenarii și cazuri de utilizare**

Cazurile de utilizare vor fi realizare folosind diagrame use case. O diagramă use case este o reprezentare grafică a interacțiunii dintre utilizator și aplicație. Se va prezenta un actor, precum și acțiunile pe care acesta are posibilitatea de a le realiza. Folosirea aplicației pentru gestiunea sistemului de cozi presupune parcurgerea succesivă a mai multor etape. În primul rând, se realizează introducerea datelor de intrare necesare de către utilizatorul final în câmpurile libere (casetele de JtextField). în acest sens, se setează timpul minim și timpul maxim de sosire, timpul minim și maxim de procesare sau servire, numărul de servere și intervalul de simulare. În cazul introducerii unor date valide, la apăsarea butonului ,,Start Simulation” se va genera simularea cozilor, afișându-se pe ecran rezultatele (timpul actual, evoluția cozii, acțiunile clienților).



1. **Proiectare**

**3.1 Decizii de proiectare**

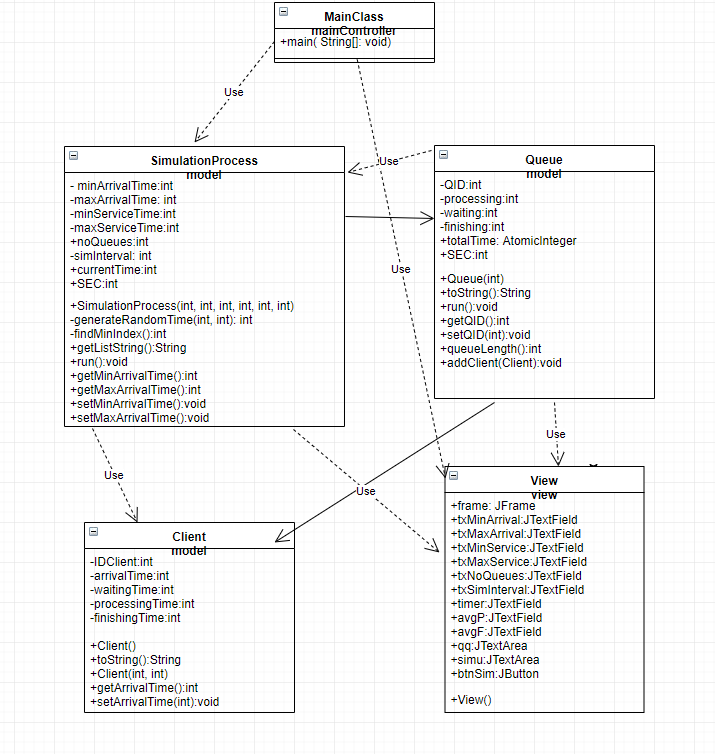
Am optat pentru divizarea acestei teme în mai multe task-uri. Pentru început, am decis să proiectez interfața grafică. Am testat funcționalitatea acesteia prin introducerea datelor de intrare (numerelor specificate) și apoi am afișat acele valori în spațiile destinate răspunsului. Proiectarea interfeței grafice, asigurându-mă de corectitudinea datelor introduse și de afișarea corectă a lor a fost urmată de implementarea sistemului de cozi cu ajutorul firelor de execuție (Threads). Pe acest concept s-a fundamentat întreaga temă, el constituind cea mai mică unitate de procesare ce poate fi programată spre execuție de către sistemul de operare. Utilizarea sa se datorează necesității de eficientizare a execuției programelor, executând porțiuni distincte de cod în paralel în interiorul aceluiași proces. Mai mult decât atât, am implementat conceptul de Multithreading, care se referă la un program ce execută simultan mai multe fire de execuție. În Java există două metode principale de creare a thread-urilor (fie prin extinderea clasei Thread- metodă pe care am implementat-o actualmente, fie prin implementarea interfeței Runnable). Se impune aici o condiție esențială- clasa Thread trebuie neapărat să conțină o metodă public void run() ce cuprinde blocul de instrucțiuni executate în cadrul firului de execuție. Utilizarea mai multor thread-uri necesită o sincronizare a acțiunilor acestora.

* 1. **Diagrame UML**

Unified Modeling Language(UML) este un limbaj standard pentru descrierea de modele si specificatii software. UML oferă o largă gamă de diagrame pentru modelarea diferitelor situații în cadrul unui proiect de dezvoltare software.

Aceste diagrame sunt de mai multe tipuri: de activitate, de componente, de clasă, package, de secvență, use-case și deployment.

În continuare, va fi prezentată diagrama de clase și de pachete pentru sistemul de gestiune a cozilor implementat.



Principalele relații ce se stabilesc între clase sunt următoarele: MainClass folosește SimulationProcess și View, SimulationProcess folosește Client și View și depinde de Queue, Queue folosește SimulationProcess și View și depinde de Client.

* 1. **Structuri de date**

Ca structuri de date mai inedite, din categoria colecțiilor concurente am ales să folosesc BlockingQueue pentru lista de clienți. Implementarea acestei interfețe este thread-safe, asigură sincronizarea, necesită importarea pachetului java.util.concurrent package. De asemenea, am optat pentru utilizarea tipului AtomicInteger pentru timpul total al fiecărei case de marcat și are rolul de contor atomic ce poate fi folosit în mod concurent de mai multe thread-uri.

* 1. **Proiectare clase**

1. **Implementare**
   1. **Metode**
   2. **Interfața utilizator**

În cele ce urmează, voi prezenta proiectarea claselor, algoritmii folosiți, implementarea metodelor și a interfeței utilizator.

Întregul proiect este structurat similar modelului MVC (Model- View- Controller), care este un model arhitectural utilizat în ingineria software. Succesul modelului se datorează izolării logicii față de interfața cu utilizatorul, rezultând o aplicație modularizată, ușor de înțeles și eficientă. Aplicația este structurată în mai multe pachete, câte unul pentru fiecare funcționalitate (model, view, mainController). Vederea și controlorul principal conțin câte o clasă cu aceeași denumire.

Modelul manipulează operațiunile logice și de utilizare de informație, constând în instanțierea a două obiecte de tipul polinom pe care se vor aplica operațiile menționate anterior. Lăsând la o parte abordarea de ansamblu, superficială și încercând pătrunderea în adevărata logică a programului, prin descoperirea gândirii din spatele acestei aplicații, gândire ce a guvernat, de fapt, întreg mecanismul, se observă structurarea pe nivele. Astfel, pentru a crea funcționalități complexe, s-a pornit de la lucruri și concepte simple, de bază, care au fost ulterior dezvoltate, interconectate și interclasate cu scopul de a obține un tot unitar. Astfel, clasa care a constituit punctul de plecare este Client, având ca atribute propriul identificator (IDClient), timpul de sosire, de așteptare, de procesare și de finalizare, caracteristici individuale, specifice fiecărui client în particular. Toate aceste variabile sunt de tipul int, având modificatorul de acces private, motiv pentru care am generat getters și setters. Constructorul are drept parametri timpul de sosire și cel de procesare, întrucât aceste date vor fi generate inițial în procesul de simulare. Metoda toString() permite o afișare corespunzătoare necesității.

O clasă importantă este Queue, care extinde clasa Thread. Aceasta se caracterizează prin următoarele atribute (id-ul propriu, timpul de procesare, așteptare, finalizare, timpul total al casei, lista de clienți, precum și o constantă SEC necesară atunci când apelăm sleep. Metoda public void run() presupune eliberarea clienților din coadă în urma procesării lor. Astfel, atât timp cât există clienți în coadă (consumers.size()>0), ne poziționăm pe primul client și folosim timpul acestuia de procesare, înmulțit cu constanta SEC(2000) pentru a apela metoda sleep a thread-ului, ce are drept scop suspendarea execuției firului pentru perioada specificată. Este un mod eficient de face timpul procesorului să fie disponibil și altor thread-uri care rulează pe sistemul computer-ului. Timpul de așteptare al fiecărui client se determină prin diferența dintre timpul curent al simulării (SimulationProcess.currentTime) și timpul de finalizare al clientului (topClient.getFinishingTime()). Ca rezultate vizibile, se vor afișa pe ecran timpul curent al procesului de simulare, plecarea clientului din coadă, specificându-se ID-ul consumatorului respectiv, ID-ul cozii în care era plasat, dar și timpul așteptat. Plecarea acestuia determină ștergerea lui din lista de consumatori, dar și actualizarea timpului total al casei, care va scădea cu valoarea timpului de procesare al clientului.

**public** **void** run() {

**while**(**true**) {

**if**(consumers.size()>0) {

Client topClient=consumers.peek();

processing=topClient.getProcessingTime();

**try** {

*sleep*(processing\****SEC***);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**int** wait=SimulationProcess.*currentTime*-topClient.getFinishingTime();

View.*simu*.append("\n CURRENT= "+ SimulationProcess.*currentTime*);

View.*simu*.append("\nClientul "+topClient.getIDClient() +

" a PLECAT din coada "+ QID+", a ASTEPTAT "+wait +" secunde");

consumers.remove();

totalTime.set(totalTime.get()-processing);//modific timpul casei

}

}

}

Următoarea clasă din pachetul model este SimulationProcess, care extinde clasa Thread și a cărei funcționalitate se bazează pe generarea de clienți. Atributele prezente sunt timpul minim și maxim de sosire, timpul minim și maxim de procesare (citite de către utilizator în interfața grafică, necesare pentru generarea timpilor de sosire și procesare ai fiecărui client), numărul de cozi (citit de la tastatură), intervalul de simulare (citit), precum și un ArrayList de cozi. În constructor, se parcurg toate cozile și se dă start la fiecare thread coadă. Metoda

GenerateRandomTime(int minTime, int maxTime) are scopul de a genera random timpul fiecărui client, cu ajutorul limitei maxime și minime introduse de către utilizator. Metoda findMinIndex caută indexul minim, adică în ce coadă clientul poate fi introdus (care casă este mai disponibilă, în funcție de numărul de clienți pe care îl conține în momentul respectiv). Astfel, se compară succesiv dimensiunile cozilor și se alege cea cu număr minim de consumatori. Metoda getListString() returnează un string, permițând afișarea clienților din fiecare coadă, dar și timpul cozii pe parcurs.

Metoda public void run() funcționează în felul următor: atât timp cât timpul actual se încadrează în intervalul de simulare ales, se generează random clienți (timp de sosire și de procesare), marcând acest fapt prin afișarea acțiunilor în interfață. Până la sosirea clientului, se simulează trecerea timpului, secundă după secundă, dând sleep la thread 1 SEC(2000) . Timpul de așteptare al clientului ia, inițial, valoarea timpului de procesare. Timpul de finalizare se calculează însumând timpul curent cu cel de procesare. Apoi, se caută indexul minim și se plasează consumatorul într-o anumită coadă, se actualizează timpul total al casei prin adunarea vechii valori cu timpul de procesare al clientului, se incrementează contorul de clienți și se afișează adăugarea în listă, menționându-se și timpul de procesare, dar și cel la care ar trebui să se termine servirea.

**public** **void** run() {

*currentTime* = 0;//ora curenta

**int** arrive, process;

**int** sumFinish=0;//suma timpi finalizare

**int** sumProc=0;//suma timpi procesare/servire

**int** ctClients=0;//nr clienti

View.*qq*.setText(getListString());

**while** (*currentTime* < *simInterval*) {

arrive=*generateRandomTime*(minArrivalTime, maxArrivalTime);

process=*generateRandomTime*(minServiceTime, maxServiceTime);

Client c = **new** Client(arrive, process);// constructor cu sosire si procesare

c.setIDClient(ctClients);

View.*simu*.append("\n CURRENT= "+ *currentTime*);

View.*simu*.append("\nClientul "+ c.getIDClient()+" va SOSI in "+ arrive +" secunde\n");

**try** {

**for**(**int** i=0; i<c.getArrivalTime();i++) {

*currentTime*++;//simulez trecerea timpului, trecerea fiecarei secunde

View.*timer*.setText(""+ *currentTime*);

*sleep*(***SEC***);

}

} **catch** (InterruptedException e) {

e.printStackTrace();

}

c.setWaitingTime(process);//timpul de asteptare al clientului, initializat cu timpProcesare

c.setFinishingTime(*currentTime*+c.getProcessingTime());//finalizare=timpul curent+procesare

**int** index = findMinIndex();

*q*.get(index).addClient(c);//adaug clientul in coada pe pozitia libera

*q*.get(index).getTotalTime().set(*q*.get(index).getTotalTime().get() + process);//actualizez timpul casei

ctClients++;//actualizez nr clienti

sumProc+=process;

sumFinish+=c.getFinishingTime();

View.*qq*.setText("");

View.*qq*.setText(getListString());

View.*simu*.append("\n CURRENT= "+ *currentTime*);

View.*simu*.append(" \nClientul "+c.getIDClient() +

" a fost ADAUGAT in coada "+ Integer.*toString*(index)+"\n" +"timp procesare client= "+ process+"=>iesire:"+ c.getFinishingTime()+"\n");

}

**int** avgFinish=sumFinish/ctClients;//timpul mediu de finalizare

View.*avgF*.setText(avgFinish+ "");

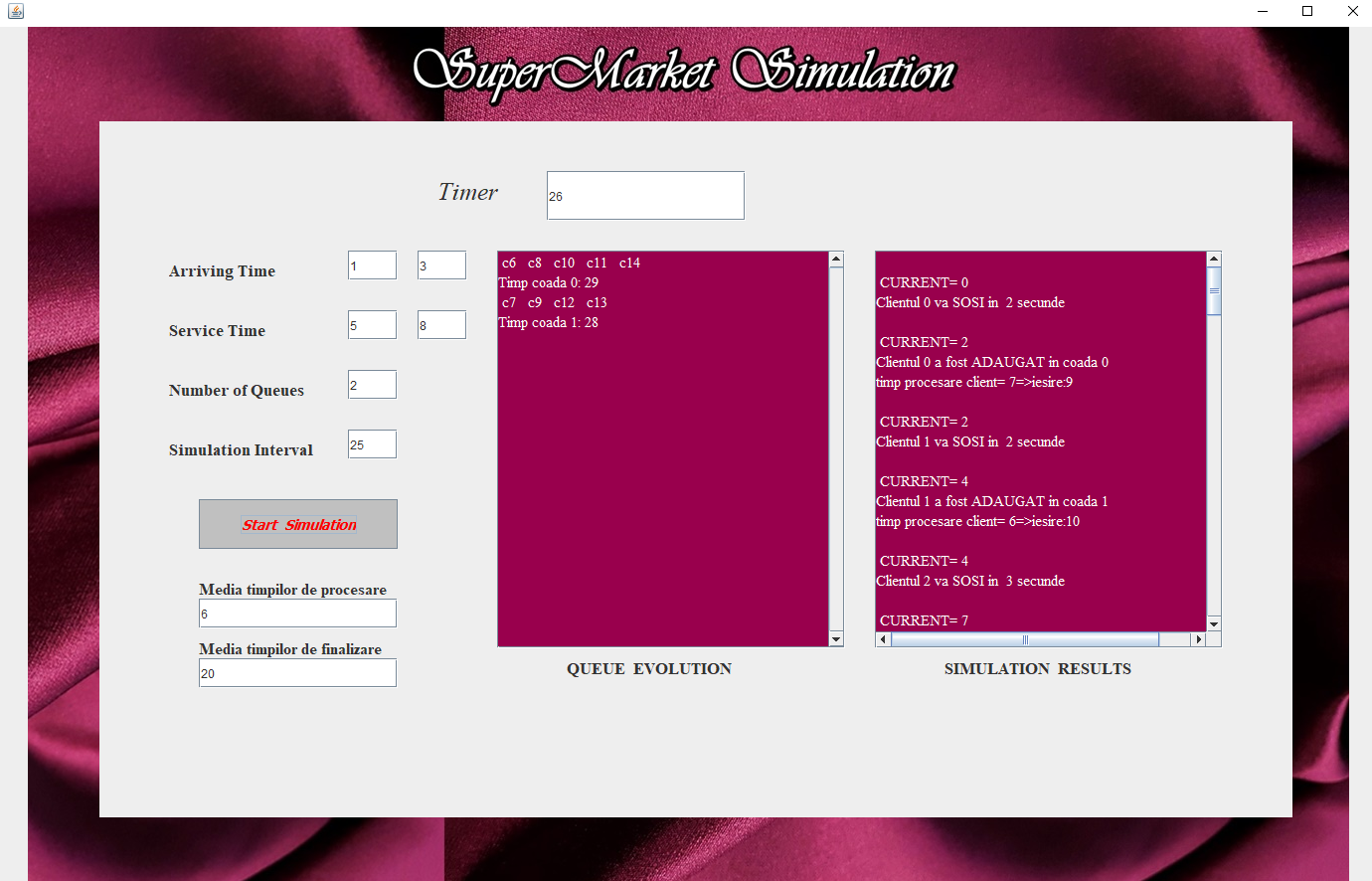
**int** avgProc=sumProc/ctClients;//timpul mediu de servire

View.*avgP*.setText(avgProc+ "");

}

Interfața grafică, implementată în clasa View (pachetul view) este compusă dintr-un frame (o fereastră), un panel format din mai multe JLabels, mai multe JtextFields, destinate introducerii de către user a datelor revendicate (timpul minim și maxim de sosire, timpul minim și maxim de servire, numărul de cozi, precum și intervalul (perioada) de simulare. Mai există o altă casetă de JtextField în care se poate observa timpul actual (trecerea timpului, secundă după secundă) și două zone de JtextArea, una pentru evoluția cozii (arată clienții din fiecare coadă, dar și timpul total pe coadă), iar cealaltă pentru rezultatele simulării (sosirea și plecarea consumatorilor, cu precizarea timpilor specifici). Aceste rezultate se generează în urma apăsării butonului Start Simulation și, datorită multitudinii lor, a fost necesară folosirea câte unui JScrollPane pentru fiecare JTextArea.

Interfața ActionListener a butonului este implementată în clasa MainClass din pachetul mainController, clasă care face conexiunea dintre funcționalitatea programului și interfața acestuia. Astfel, în metoda main se instanțiază o nouă vedere, se implementează ActionListener prin metoda actionPerformed, în cadrul căreia se citesc de la tastatură datele necesare, se instanțiază un nou SimulationProcess, primind ca parametri informațiile citite și se pornește thread-ul. În acest fel, prin apăsarea butonului se va porni execuția programului.



1. **Rezultate și concluzii**

În urma acestei teme mi-am dezvoltat abilitățile în ceea ce privește proiectarea unei interfețe grafice, am reușit să înțeleg mai bine noțiunea de thread, concept pe care l-am parcurs până acum doar teoretic, neavând ocazia de a realiza o aplicație care necesită punerea lui în practică.

1. **Bibliografie**

<https://ro.wikipedia.org/wiki/Fir_de_execu%C8%9Bie>

<https://www.geeksforgeeks.org/blockingqueue-interface-in-java/>

<https://www.geeksforgeeks.org/stack-peek-method-in-java/>

<http://users.utcluj.ro/~igiosan/Resources/POO/Curs/POO11.pdf>