**Documentație Tema 2**

**Simulator de cozi**

**Student: Tocilă Emilia**

**Grupa 302210**

**Profesor laborator : Antal Marcel**

**Cuprins**

1. Obiectivul temei...........................................................................3
2. Analiza problemei, modelare, scenarii, cazuri de utilizare..........3
3. Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfețe, relații, packages, algoritmi, interfața utilizator)........................................................................4
4. Implementare................................................................................5
5. Rezultate.......................................................................................8
6. Concluzii....................................................................................9
7. Bibliografie.................................................................................10

* **Obiectivul temei**
* **Obiectivul principal**

Propuneți, proiectați si implementați o aplicație care să analizeze sistemele bazate pe cozi determinând și minimizând timpul de așteptare al clienților la coadă.

* **Obiective secundare**
* Dezvoltarea de use case-uri si scenarii (cap. 2) – descrisă execuția programului
* Alegerea structurilor de date (cap. 3) – descrise structurile care pot implementa facil problema
* Împărțirea pe clase (cap. 3) – clasele ce vor dezvolta implementările
* Dezvoltarea algoritmilor (cap. 4) – implementarea algoritmilor & metodelor
* Implementarea soluției (cap. 4)
* Rezultate (cap. 5) – verificare
* **Analiza problemei, modelare, scenarii, cazuri de utilizare**

Principalul obiectiv al unei cozi este să ofere un spațiu pentru un client în care să aștepte înainte să fie servit. Aplicația simulează o serie de clienți care vin pentru un serviciu, intrând în cozi, așteaptă, sunt serviți și apoi părăsesc coada. Management-ul sistemelor bazate pe cozi este axat în jurul minimizării timpului pe care clienții îl petrec așteptând în coadă înainte să fie servit. Felul în care am gândit și implementat acest lucru a fost să am mai multe cozi prin thread-uri, iar atunci când un client nou vine, acesta va fi repartizat la coada cu timpul de așteptare cel mai mic. Ca intrări, vom avea timpul minim și maxim pentru momentele de venire și timpul serviciului, numărul de cozi, timpul de simulare și numărul de clienți. Ca ieșiri, vom afișa peak time, off-peak time, timpul mediu de serviciu și timpul mediu de așteptare.

**Diagrama use-case:**

Introducerea datelor de intrare

Afișarea în timp real a evoluției cozilor

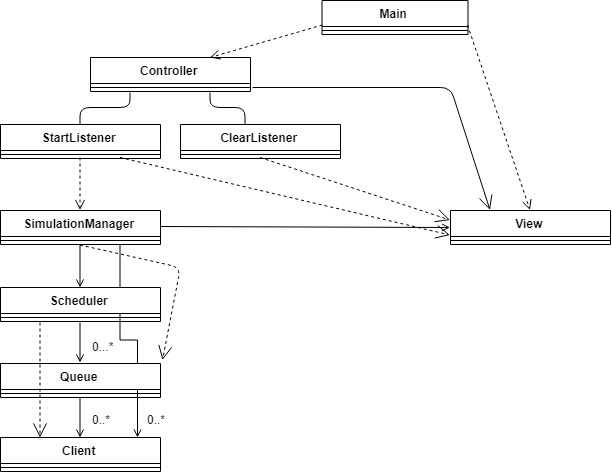
Utilizator

Afișarea rezultatelor despre simulare

**Scenariu:**

1. Utilizatorul introduce datele de intrare(timpul de simulare, timpul minim și maxim al momentului sosirii clienților, timpul minim și maxim al serviciului fiecărui client, numărul de cozi și numărul de clienți).
2. Utilizatorul apasă butonul start.
3. Pe interfață va apărea evoluția cozii și la finalul simulării, rezultatele cu privire la simulare.

* **Proiectare**
* **Diagrama UML**



* **Decizii de proiectare**

Am împărțit proiectul în clasele Client, Queue, Scheduler și SimulationManager pentru a facilita rezolvarea problemei. Pentru realizarea interfeței grafice conform implementării MVC, am creat clasa ” Controller” și ” View”.

* **Structuri de date**

Pentru a reprezenta cozile de clienți, am folosit structura ArrayBlockingQueue<Client>, lucrând foarte bine cu thread-uri. În mod evident, am avut nevoie de o coadă, deoarece merge pe principiul FIFO (first in, first out); primul client din coadă va fi cel care este servit, iar ultimul este cel care a intrat ultimul. În clasa Scheduler avem un ArrayList<Queue>, adică o listă cu toate cozile, pentru a porni thread-urile acestora în constructor și pentru a parcurge cozile cu scopul de a decide care este cea mai bună coada la care următorul client va fi adăugat pentru a minimiza timpul de așteptare. În clasa SimulationManager avem ArrayList<Client> pentru a păstra toți clienții generați random și pentru a-i introduce în cozi.

* **Implementare**

**Clasa Client:**

Această clasă se ocupă cu crearea și gestionarea clienților din cozi. Avem variabilele instanță ID, arrivalTime, serviceTime, waitingTime și doIncrementWaitingTime. Aceasta din urmă va fi true atât timp cât clientul așteaptă sa fie servit, și false atunci când acesta este servit. Clasa conține getter-e și setter-e, precum și două metode care incrementează sau decrementează anumite câmpuri (decrementServiceTime: cu fiecare moment de timp în care clientul este servit, valoarea câmpului serviceTime scade, până ajunge la 0 (clientul poate părăsi coada); incrementWaitingTime: cu fiecare moment de timp în care clientul așteaptă să fie servit, valoarea câmpului waitingTime crește în cazul în care variabila booleană doIncrementWaitingTime este true). Clasa conține si metoda suprascrisă compareTo, motiv pentru care clasa Client implementează Comparable<Client>. Motivul pentru care folosim această metodă este pentru a sorta clienții generați random în SimulationManager după timpul sosirii (astfel încat primul client din acea listă va fi următorul care va intra într-una din cozi – sunt ordonați ”cronologic”). În final, avem metoda toString care returnează un String cu ID-ul clientului și timpul de procesare (cât timp de serviciu i-a rămas).

**Clasa Queue:**

Clasa Queue impementează interfața Runnable, pentru a executa o porțiune de cod in paralel în interiorul aceluiași proces (programare concurentă). Vom avea mai multe cozi care vor avea evoluții diferite, însă toate cozile vor urmări aceiași pași (preluarea unui client, servirea clientului și plecarea clientului). Pentru aceasta, păstrăm clienții în structura de date ArrayBlockingQueue<Client> (queue).

Constructorul inițializează coada.

Pentru a mări timpul de așteptare tutuor clienților din coadă care nu sunt serviți, avem ArrayList<Client> (allClients), unde vom păstra toți clienții care au fost vreodată în coada respectivă. Trebuie să-i păstrăm pe toți, nu doar pe cei care au rămas, deoarece la final trebuie să facem media timpului de așteptare al tuturor în momentul finalizării simulării. De aceea avem o variabilă instanță waitingTime, unde vom aduna toți acești timpi ai clienților. Metoda incrementWaitingTimeForAllClients face întocmai ce spune numele. Metoda addClient adaugă în coadă un client nou, dar și in ArrayList-ul nostru ce conține toți clienții. De asemenea, tot în interiorul acestei metode, adăugam la variabila instanță serviceTime timpul serviciului clientului adăugat. Această variabilă ne va folosi pentru a calcula timpul mediu al serviciului fiecărui client din simulare. Variabila instanță done (boolean) joacă rolul de a opri thread-urile (este un flag). Prin metoda setDone, care va fi apelată din SimulationManager, se va seta done pe true, adică timpul de simulare s-a terminat, astfel, thread-urile trebuie oprite. Dacă am face while(true), thread-urile vor merge la infinit, iar procesorul va merge foarte greu. Avem getter-e și setter-e (getServiceTime, getWaitingTime, getNrOfClientsInQueue, setDone).

Metoda run este executată din momentul în care a fost creat thread-ul. Cât timp variabila done este false, adică cât timp încă continuă simularea, dacă coada nu este goala, este păstrat în variabila locala c de tipul Client clientul din capul cozii (adică este preluat spre procesare). Se apelează metoda c.setFalseDoIncrementWaitingTime (nu se va mai mări timpul de așteptare, pentru că este servit clientul). Cât timp clientul este servit (while(c.getServiceTime() > 0), thread-ul este suspendat o secundă cu sleep(1000) , este decrementat timpul serviciului clientului. După ce clientul a fost servit, acesta este eliminat din coadă (queue.take()).

Metoda toString returnează un String cu informațiile despre toți clienții din coadă.

**Clasa Scheduler:**

Clasa Scheduler are variabilele instanță un ArrayList de cozi și o primitivă nrOfQueues. În constructorul acestei clase, se crează nrOfQueues cozi și se pornesc thread-urile acestora.

Metoda processClient are o variabilă instanță bestQueue de tipul Queue unde va fi repartizat clientul către cea mai bună coadă (cea cu timpul de așteptare cel mai mic) și o variabilă instanță minWaitingTime inițializată cu Integer.MAX\_VALUE unde va fi timpul minim de așteptare dintre toate cozile. Se parcurg cozile și se determină coada ideală. După ce a fost stabilită coada ideală, clientului trimis ca parametru îi este setată variabila do pe true (adică se poate începe incrementarea timpului de așteptare până în moemntul în care este servit) și este adăugat în această coadă. Metoda getQueues returnează ArrayList-ul ce conține toate cozile.

**Clasa SimulationManager:**

Această clasă este cea mai complexă, modelând simularea. Ca variabile instanță, aceasta conține un obiect scheduler, datele de intrare introduse de utilizator, un ArrayList cu clienții generați random, variabila done care va fi transmisă cozilor, un obiect view pentru comunica cu acesta și a-i transmite String-urile pentru a le afișa, variabila currentTime și de asemenea variabile necesare pentru a calcula rezultatele și concluziile ce reies din simulare.

Constructorul primește de la controller datele de intrare, inițializează ArrayList-ul de clienți cu numărul clienților, apelează constructorul Scheduler și apelează metoda generateClients.

Metoda generateClients primește ca parametru numărul de clienți și le atribuie valori random între limitele primite de la utilizator pentru arrivalTime și serviceTime. După ce au fost generate aceste valori, este creat un client și adăugat în ArrayList. La final, după adăugarea tuturor clienților, apelăm metoda Collections.sort(clients) pentru a-i sorta în funcție de timpul de sosire.

Metoda processQueues va fi apelată in metoda run. În variabila locală j va fi păstrat numărul cozii, iar în qEvol, Stringul cu informații despre cozi. Dacă mai există clienți în ArrayList, cât timp primul client din listă are timpul de sosire egal cu timpul current, clientul este procesat (scheduler.processClient(clients.get(0)), variabilei instanță i se adaugă timpul de servire al clientului introdus, variabila instanță nrOfClientsThisDay este incrementată, se scoate din ArrayList clientul, iar apoi se mai verifică o dată dacă mai există clienți in ArrayList. Apoi se parcurg cozile cu un for după ce au fost introduși clienții cu timpul de sosire corespunzător, se incrementează timpul de așteptare al tuturor clienților unde este necesar (q.incrementWaitingTimeForAllClients()); dacă moemntul de timp este ultimul din simulare, se va adăuga variabilei instanță timpul de așteptare al tuturor clienților care au făcut parte din simulare (if(currentTime == simulationTime - 1) averageWaitingTime += q.getWaitingTime();); se adaugă String-ului qEvol informațiile despre cozi (#nrOfQueue: info - qEvol += j + ": " + q.toString() + "\n"; j++;). La finalul metodei se incrementează variabila instanță currentTime.

Metoda run funcționează astfel: Cât timp momentul curent este mai mic decat timpul de simulare, este apelată metoda processQueues. Apoi, în blocul try, se vor calcula rezultatele simulării. În primul rând, se calculează numărul de clienți din fiecare moment de timp parcurgând cozile (nrOfClientsThisHour += q.getNrOfClientsInQueue();); se suspendă thread-ul o secundă (pentru a afișa toate informațiile), se determină peakHour (numărul maxim de clienți într-un moment de timp din tot timpul simulării) și offPeakHour, în mod similar; se apelează view.setQueues(qEvol) pentru a afișa la fiecare moment informații despre cozi, iar qEvol este resetat. După blocul catch, sunt variabila done este setată (adică thread-urile cozilor se opresc – timpul curent a ajuns la timpul de simulare) și se calculează averageServiceTime și averageWaitingTime. Apoi este apelată metoda view.setInfo(info) pentru a afișa aceste rezultate (peakHour, offPeakHour, averageServiceTime, averageWaitingTime).

**Clasa View:**

Se ocupă cu aspectul interfeței cu utilizatorul. Această componentă conține 7 zone de text unde utilizatorul poate scrie datele de intrare, două zone de text ce nu pot fi modificate de utilizator unde vor apărea rezultatele (evolution of queues - JScrollPane și information), 2 butoane; unul pentru a porni simularea și celălalt pentru a reseta câmpurile unde apar rezultatele.

Metoda reset setează textul din cele 2 câmpuri destinate rezultatelor la ””.

Metodele getSimTime, getMinSTime, getMinSTime, getMaxSTime, getMinATime, getMaxATime, getNrQ și getNrC returnează string-urile corespunzătoare datelor introduse de utilizator.

Metoda setQueues adaugă la textul din Evolution of queues prin append informații despre cozi la fiecare moment de timp.

Metoda setInfo setează în zona de text ”Information” datele care au rezultat din simulare.

Metodele add---Listener (--- : Start, Clear) leagă butoanele de funcționalitatea lor, butoanele sunt ”ascultate”, ”urmărite”.

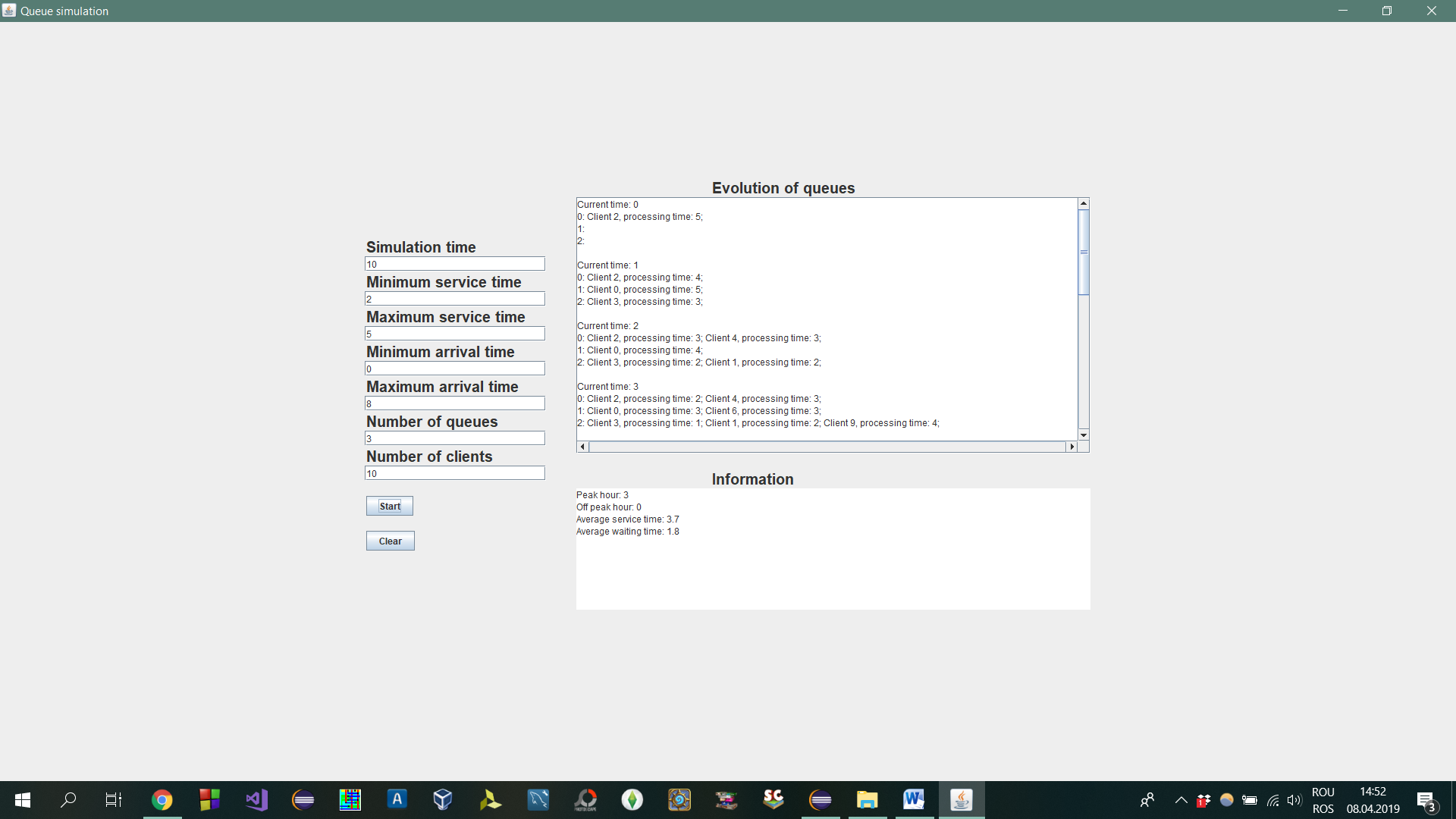
**Clasa Controller:**

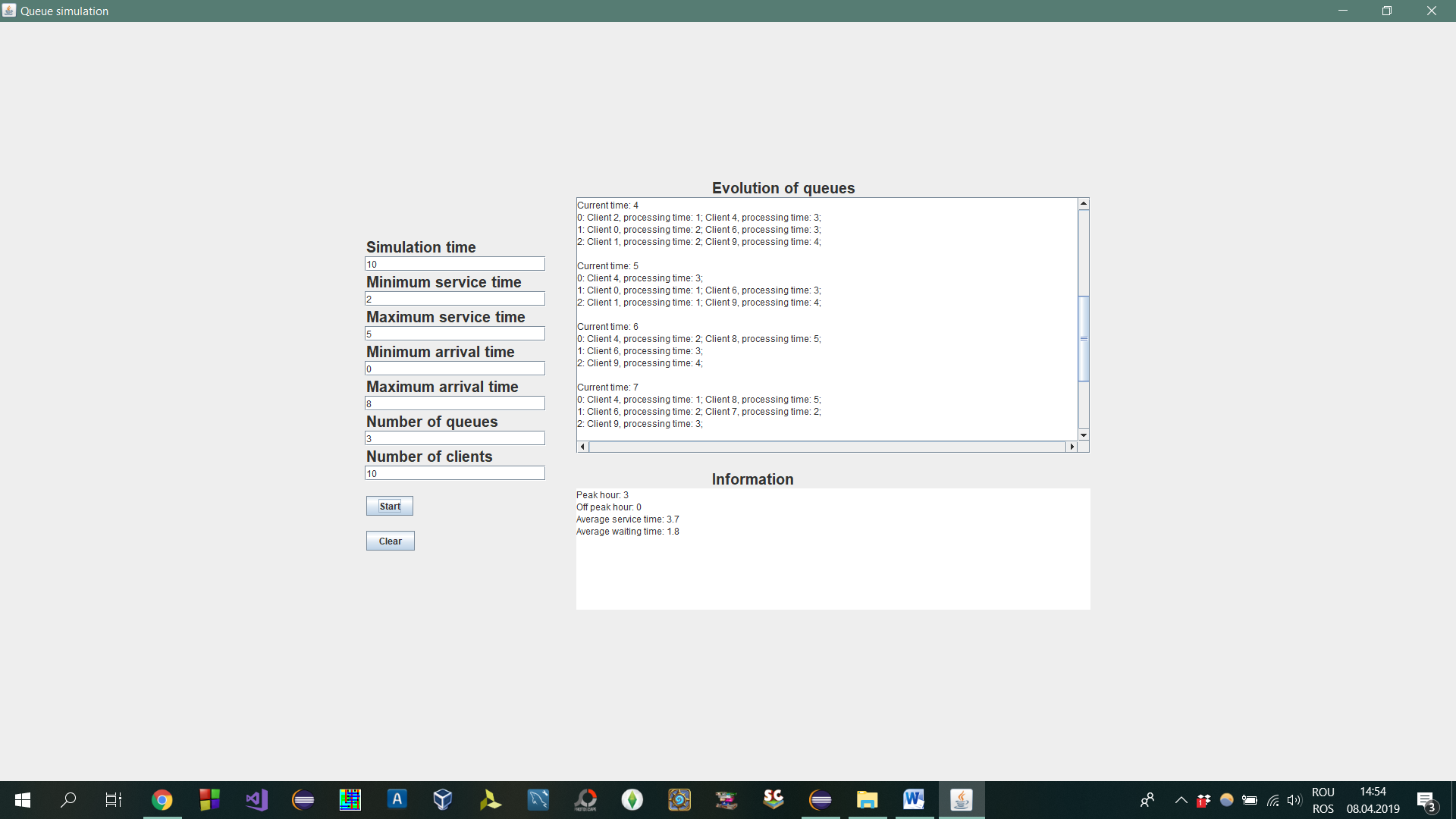
Se ocupă cu corelarea logicii din spatele programului cu partea cu care utilizatorul va interacționa direct. Clasele ---Listener (--- : Start, Clear) care implementează ActionListener au metoda actionPerformed, cu parametrul ActionEvent e, fac întocmai acest lucru. Variabilele locale din metoda actionPerformed din clasa StartListener reprezintă primitivele de tip int care vor fi preluate de la interfață și transmise către SimulationManager. Pentru a face asta, trebuie efectuată o parsare de la String la Integer (Integer.parseInt(view.get---()))). După acest lucru, vom apela constructorul clasei SimulationManager, dând ca parametri aceste variabile și pornim thread-ul (Threat t = new Thread(sim); t.start();).

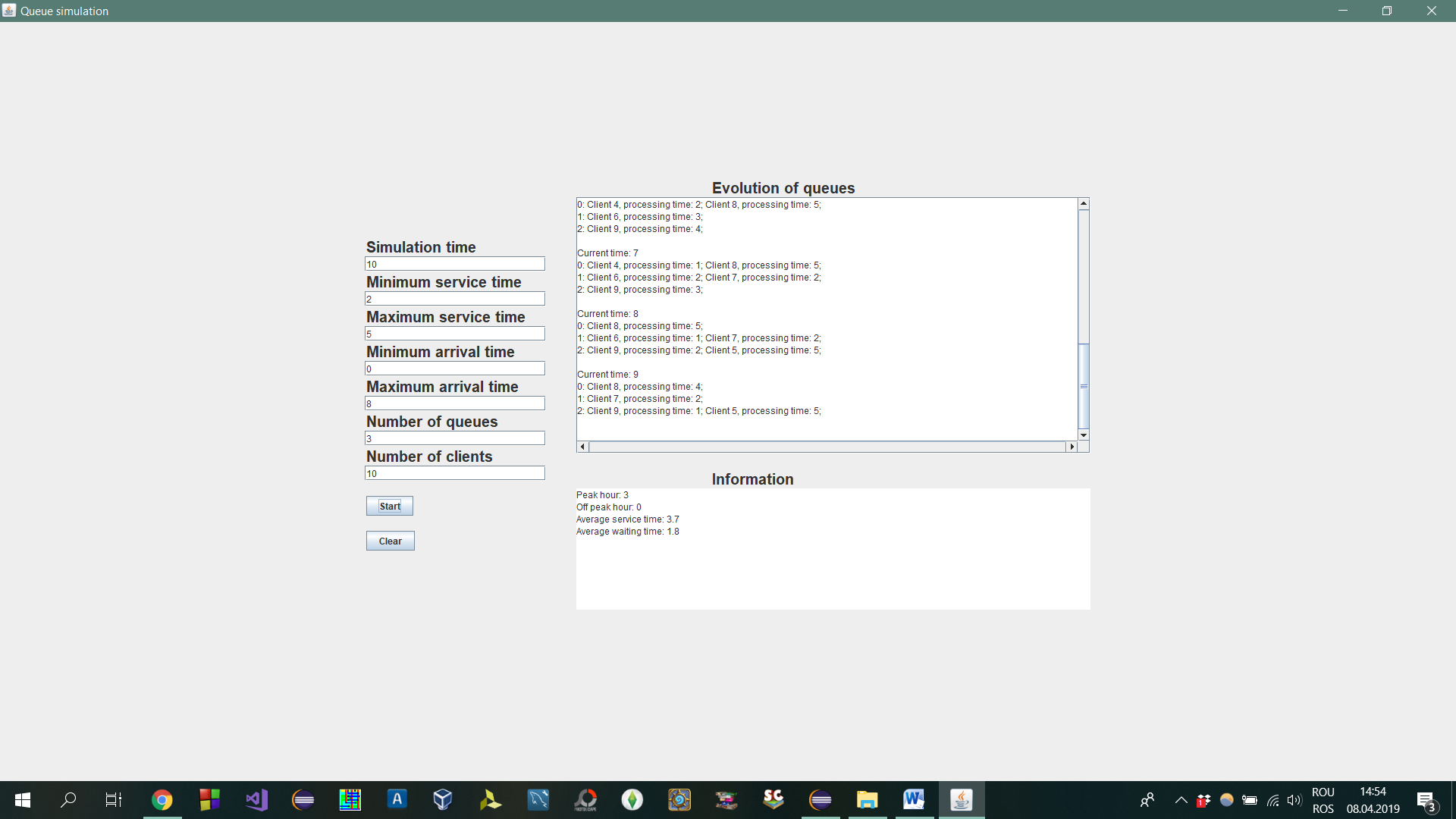
**Clasa Main:**

Clasa Main leagă View-ul de Controller și de model.

* **Rezultate**







* **Concluzii**

În urma acestui proiect, am învățat despre programarea concurentă, beneficiile pe care le aduce: mai multe fire de execuție într-un process care merg pe aceleași principii, fiind executate simultan, împărțind resursele (memorie, fișiere). Creând un thread pentru fiecare ”job”, se îmbunătățește performanța acestora, fiind executate în paralel. Având dată specificația problemei care trebuie rezolvată, decizia care trebuie luată este cum trebuie ea împărțită în procese și câte anume sunt necesare și mai ales cum trebuie acestea să interacționeze.

Aplicația creată pentru această temă mi-a adus noi cunoștințe pe care le-am folosit și au îmbunătățit în mod considerabil implementarea programului folosind această tehnică de programare.

* **Bibliografie**

<http://www.stackoverflow.com>

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.html>

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

<http://coned.utcluj.ro/~marcel99/PT2019/Tema%202/>