Tehnici de Programare – Tema 2

*Simulator de cozi*

Popovici Ruxandra

*Grupa 30227*

1. Obiectivul Temei

Obiectivul principal al proiectului de fata a fost proiectarea si implementarea unui simulator de cozi dedicat analizei unui sistem de determinare si minimizare a timpului de asteptare a clientilor. Se foloseste o interfata grafica dedicata prin intermediul careia un utilizator va putea introduce de la tastatura numarul clientilor, numarul de cozi(servere) disponibile, timpul maxim de simulare, intervalul de timp in care pot ajunge clientii si intervaul de timp in care acestia pot fi serviti.

Obiectivele secundare ale proiectului:

* Analizarea obiectivelor proiectului si a problemelor legate de implementare prin modelarea de scenarii corespunzatoare cazurilor de utilizare, astfel incat sa identificam corect cerintele;
* Proiectarea simulatorului astfel incat toate necesitatile utilizatorului sa fie indeplinte;
* Implementarea solutiei optime descoperite prin scrierea de cod;
* Testarea simulatorului, observandu-i si analizandu-i concret comportamentul pentru mai multe cazuri, luand in considerare scenarii diferite de folosire si diferite date de intrare.

2. Analiza problemei (*modelare, scenarii,cazuri de utilizare)*

Prin analiza problemei intelegem examinarea cerintelor, potentialelor probleme si a diferitelor modalitati in care aplicatia noastra ar putea fi folosita, pusa la incercare: introducerea de date (timpii de simulare, numarul de clienti si cozi) care va duce la generarea unui raspuns corespunzator din partea simulatorului in functie de corectitudinea lor, in functie de intervalele de timp si incadrarea caracteristicilor fiecarui client in acestea.

Dorim ca utilizatorul sa fie atentionat cu privire la corectitudinea datelor, astfel incat va primi mesaj de eroare daca acestea nu respecta cerintele(sa fie numere intregi)

De asemenea, programul trebuie sa functioneze in mod adecvat si sa genereze raspunsul dorit, urmand principiul diagramei urmatoare:

Intervalul in care clientii pot ajunge

Numar de clienti

Numar de cozi

*Simulator de Cozi*

*~Schema generala a proiectului~*

Rezultat / Fisierul \*.txt

Timp de simulare

Intervalul in care clientii pot fi serviti

Diagrama Use Case demostreaza mai bine functionalitatea si ideea intregului proiect:

**User**

Dupa cum putem observa si mai sus, intregul sistem reprezinta aplicatia creata, in care User-ul reprezinta, de fapt, actorul / utilizatorul, care se asteapta sa primeasca un raspuns / rezultat dupa ce actioneaza asupra lui.

* Cerinte functionale ale simulatorului:
* Trebuie sa permita utilizatorului sa introduca datele necesare pornirii simularii;
* Trebuie sa permita utilizatorului sa porneasca simularea;
* Trebuie sa permita utilizatorului vizionarea rezultatului cerut – log of events, adica mersul simularii, pentru fiecare secunda trecuta si in functie de clientii generati.
* Altele:
* Simulatorul trebuie sa fie intuitiv si usor de folosit de catre utilizator.
* Sa genereze un rezultat adecvat in caz de success sau sa afiseze un mesaj de eroare in caz ca datele primite nu au respectat cerintele si formatul cautat.
* Scenariu de utilizare:

1. Utilizatorul introduce numarul de clienti de la tastatura: ex: 4;
2. Utilizatorul introduce numarul de cozi de la tastatura: ex. 2;
3. Utilizatorul introduce intervalul de simulare de la tastatura: ex: 60;
4. Utilizatorul introduce celelalte doua intervale pentru caracteristice clientilor generati: timpii de sosire si timpii de servire: amandoua fiind caracterizate de un minim si un maxim: ex. [2, 30] (fiecare client poate sa soseasca doar la un timp / o valoare din acest interval), [2, 4] (fiecare client poate sa fie servit doar intr-un timp / valoare din acest interval);
5. Utilizatorul apasa butonul de “Start” (“Adunare”), pornind aplicatia;
6. Aplicatia va genera 4 clienti la alegere, folosindu-se de datele introduse si va “porni” doua servere;
7. Timp de 60 de secunde, ori pana cand toti clientii sunt serviti, aplicatia va rula, va afisa in terminal in timp real modul in care se desfasoara simularea si va introduce aceleasi date intr-un fisier \*.txt de tip “Log Of Events”;
8. Utilizatorul iese din aplicatie.

Un scenariu alternativ de utilizare:

1. Utilizatorul introduce date gresite de la tastatura: ex. caractere in loc de numere intregi, sau spatii goale;
2. Aplicatia afiseaza un mesaj de eroare si sterge datele introduse;
3. Se revine la primii pasi din scenariul anterior.
4. Proiectare *(decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator)*

Proiectarea in limbaj OOP a presupus mai multi pasi, in functie de deciziile luate: am urmarit implementarea folosind modelul MCV (Model-View-Controller), fiind un model tip sistem interactiv si repartizarea lor in pachete, asa cum se poate observa si in diagrama de mai jos:

**data\_models**

**user\_interface**

**main**

**business\_logic**

QueueModel

Scheduler

Server

Task

Strategy

ConcreteStrategyQueue

ConcreteStrategyTime

SelectionPolicy

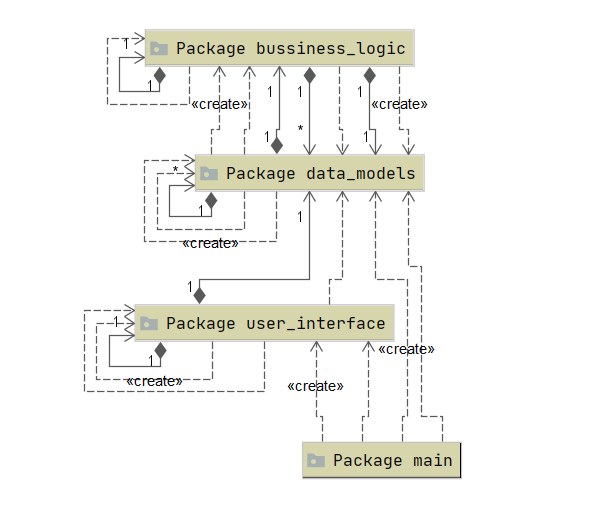
SimualtionManager

QueueController

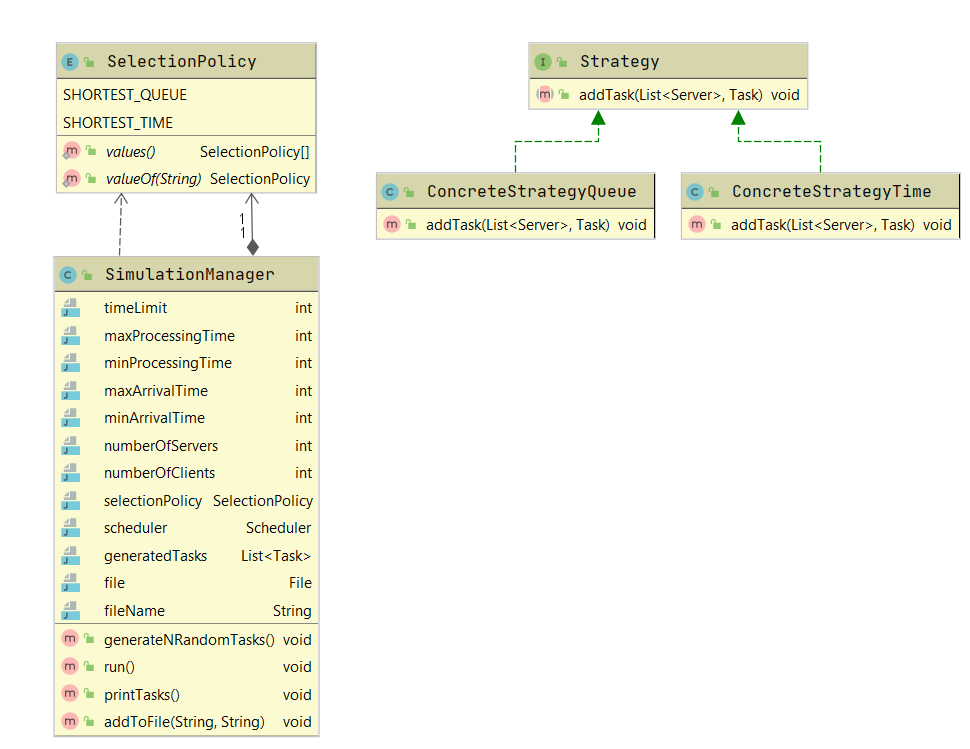
QueueView

QueueMCV

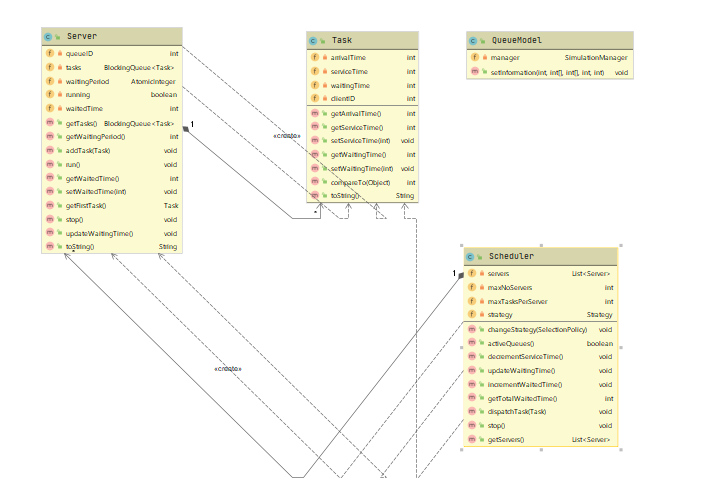
Diagrama UML a proiectului ajuta la o mai buna si amanuntita intelegere ai analiza a claselor, pachetelor, metodelor implementate in fiecare din acestea, cum se “leaga” si “comunica” intre ele:



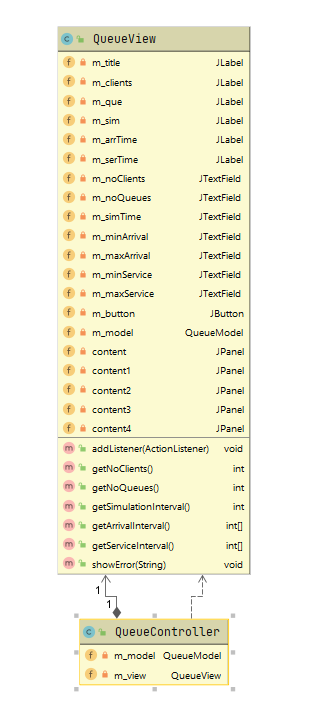
***Pachetul business\_logic:***



***Pachetul data\_models:***



***Pachetul user\_interface:***

******

Structura de tip MCV separa fiecare clasa in functie de functionalitatile metodelor din ele.

“data\_models” contine clase ce reprezinta “structurile” datelor la care se asteapta aplicatia: Task, reprezentand clientii, Server, reprezentand cozile la care acestia vor fi distribuiti, Scheduler pentru efectuarea de operatii pe servere;

“user\_interface” le contine pe cele care au de a face cu interactiunea dintre utilizator si calculatorul / aplicatia propriu-zisa: GUI, JFrame, butoanele cu ActionListenere-le respective, etc..

“business\_logic” are ca si clase chiar logica din spatele aplicatiei cese ocupa cu pornirea thread-urilor, cu alegerea strategiei, cu simularea propriu-zisa.

“main” reprezinta clasa care porneste aplicatia si care leaga toate celelalte pachete si clase impreuna.

1. Implementare

* *Pachetul “data\_models”*

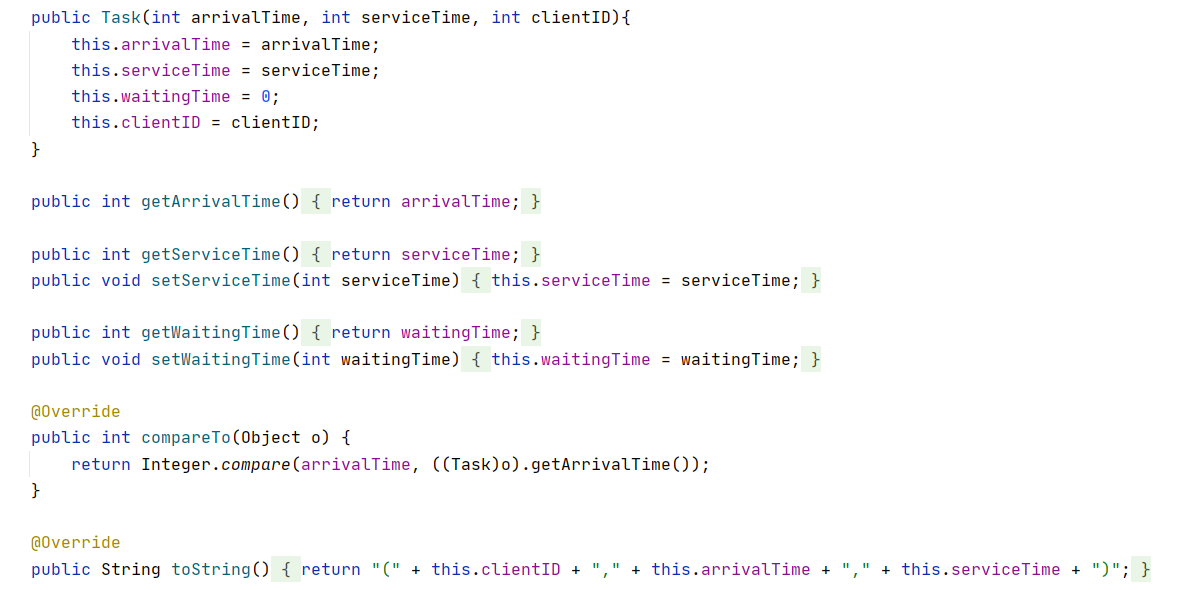
***Clasa “Task”***

Reprezinta clientul propriu zis, care va fi distribuit, in functie de anumite criterii la coada cea mai portivita.

Acesta este caracterizat prin urmatoarele atribute: *arrivalTime* (“timp de sosire”), *serviceTime* (“timp de servire”) si *clientID* (“ID-ul clientului”).

ArrivalTime este caracteristica care ne spune la ce moment al simularii poate fi atribuit un client unei cozi (cand ajunge acesta la coada), iar serviceTime reprezinta cat ii va lua unei cozi sa-l proceseze (timpul atribuit de catre o coada pentru fiecare client in parte). Pe langa acestea, mai avem si un waitingTime care reprezinta timpul petrecut de fiecare client in coada, folosindu-ne la a calcula, la final, timpul total de asteptare.

Inafara de constructor, getter si setter, am implementat, suprascriind metoda “toString”, o functie de afisare a clientului ce va ajuta la “estetica”: afisam, pe rand, ID-ul clientului, dupa care timpii caracteristici, si inca o functie compareTo, care va ajuta, dupa ce generam clientii random, la a-i sorta in functie de arrivalTime.



***Clasa “Server”***

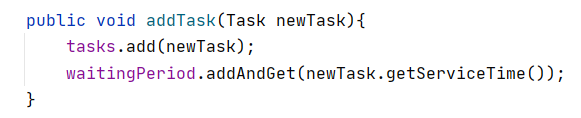
Aceasta implementeaza structura propriu-zisa a cozii, implementand interfata ***Runnable*** => fiecare coada va fi, de fapt un thread care va merge in paralel cu toate celelalte pornite odata cu ea.

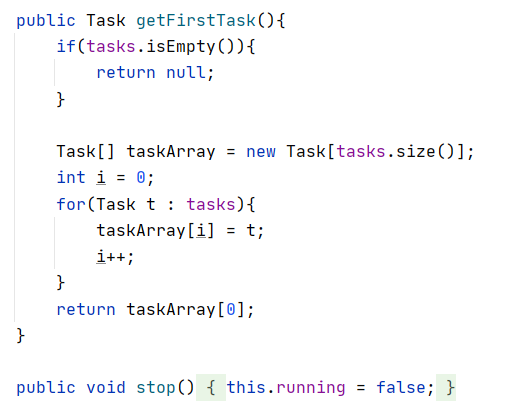
Implementam, deci, in aceasta clasa, metoda “run”, care va face ca fiecare server sa “doarma” atunci cand nu exista clienti in el, iar in caz ca exista, va “astepta” pe clietul respectiv cat ar dura sa fie servit, dupa care il va elimina din coada si va trece la urmatorul: procesul se va repeta, deci, pe fiecare client, cat timp coada va fi activa.

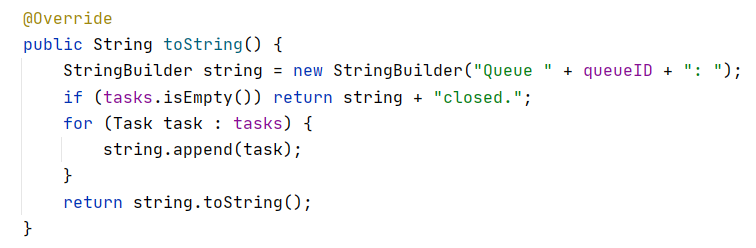


Metoda “addTask” va adauga clienti la coada si va adauga la variabila waitingPeriod timpul de servire al acestuia, in timp ce “getFirstTask” va returna “capul” cozii, primul element al acesteia, iar “stop” va seta variabila “running” a cozii la *false* (conditie care va determina cat timp va sta activ coada / serverul, adica thread-ul).

Metoda “toString” din aceeasi clasa ajuta la afisarea fiecarei cozi in parte: in caz ca aceasta va mai avea task-uri / clienti activi, ii va afisa pe acestia, iar in caz ca va fi goala, va fi urmata de cuvantul “closed!”.







***Clasa “Scheduler”***

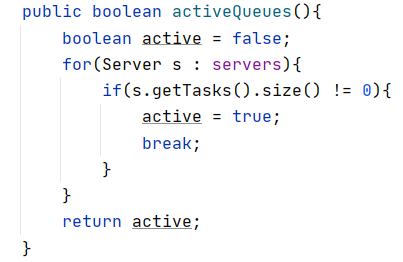
Contine o lista de servere / cozi si serveste la adaugarea de clienti in functie de cerinte, la coada cea mai portivita.

Aici vom face inserarea, ajutandu-ne de un obiect de tip “Strategy” prin metoda “dispatchTask”:

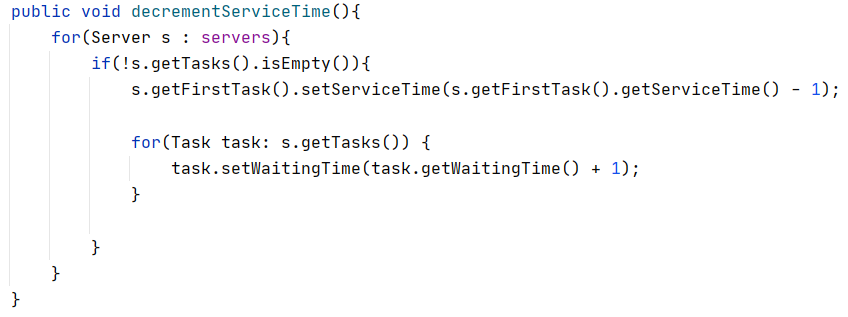


Aici sunt implementate si metodele care vor fi apelate la fiecare incrementare a timpului de simulare a aplicatiei:

“activeQueues” returneaza *true* daca cel putin o coada este activa (are clienti ce mai trebuie procesati);

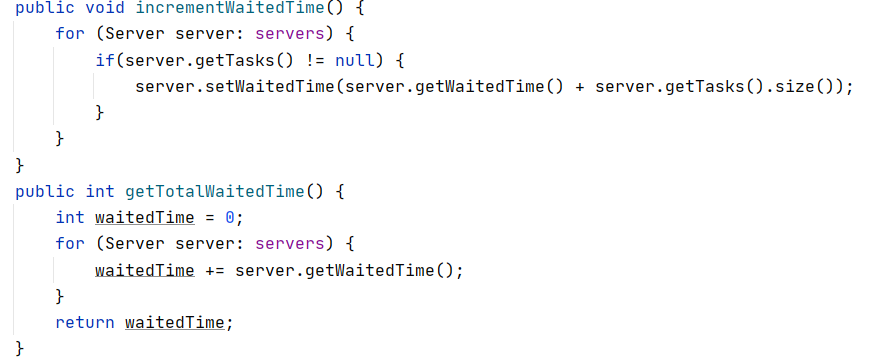


“decrementServiceTime” va decrementa timpul de servire a fiecarui prim client din toate serverele sale si va incrementa, in acelasi timp, timpul de asteptare pentru restul clientilor din acea coada;

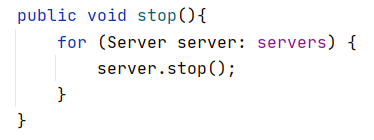


“incrementWaitedTime” va seta adauga la timpul de asteptare a fiecarui server numarul de clienti care se afla la coada in momentul respectiv;

**“**getTotalWaitedTime**”** va parcurge serverele si va returna suma tuturor timpilor de asteptare ai acestora.

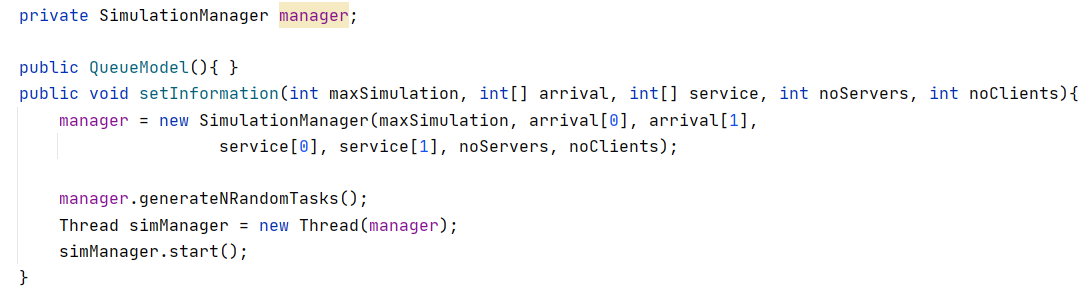


“stop” va parcurge toate serverele si le va opri threadurile.



***Clasa “QueueModel”***

Aceasta va contine un obiect de tip SimulationManager si va fi cea care, apelata, va porni intreaga simulare.

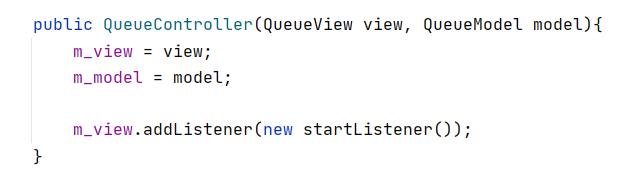


* *Pachetul “user\_interface”*

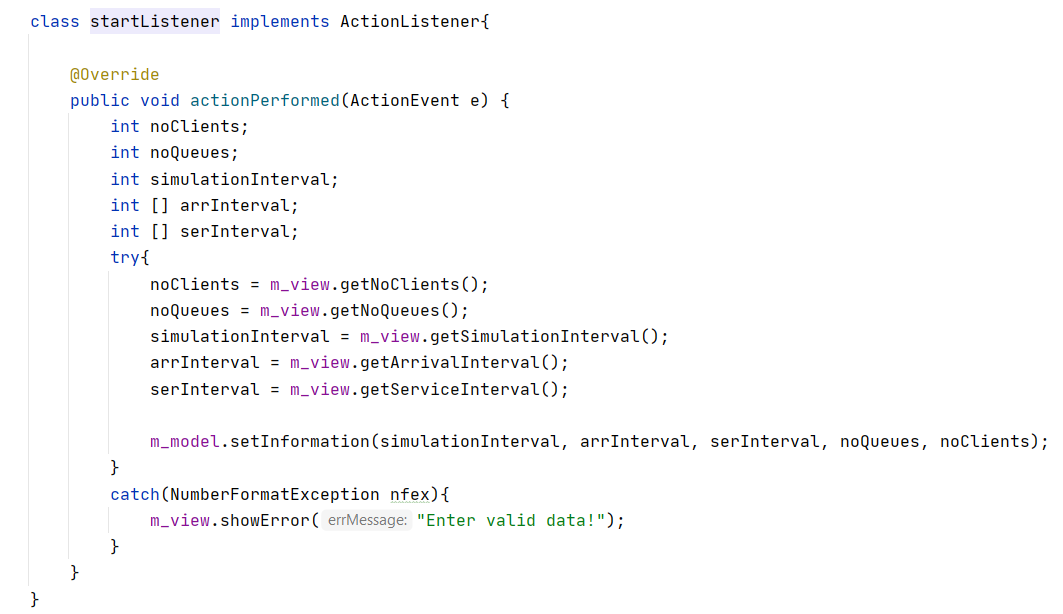
***Clasa “QueueController”***

Contine un obiect de tip “QueueModel” si unul de tip “QueueView”.

Aici se face, de fapt, legatura dintre view(interfata grafica) si model(simularea propriu-zisa).



Constructorul acestei clase va adauga si un Listener la actionarea caruia se va porni simularea, prin culegerea de informatii(setarea datelor de la tastatura) si transmiterea acestora spre a fi procesate:



***Clasa “QueueView”***

Aici se va crea interfata grafica cu care va interactiona utilizatorul.

Prin intermediul ei, utilizatorul nu trebuie sa se gandeasca cum se va ajunge / cum trebuie sa se ajunga la un rezultat, el doar va introduce date (prin TextField-uri), va trimite comenzi (prin butoane) si va primi rezultate, adica un raspuns de la aplicatie.

Diferite metode, mai ales cele de la sfarsit, ajuta programatorul in a face legatura cu celelalte clase si componente ale MCV-ului (transmiterea datelor, adica aparametrilor corespunzatori simularii - ex. “*getNoClients*”, “*getArrivalInterval*”, adaugarea de actionListener-e corespunzatoare – *ex. “AddListener”*-, afisarea de erori in functie de raspunsul primit de la Controller – *ex. showError* -, etc.):



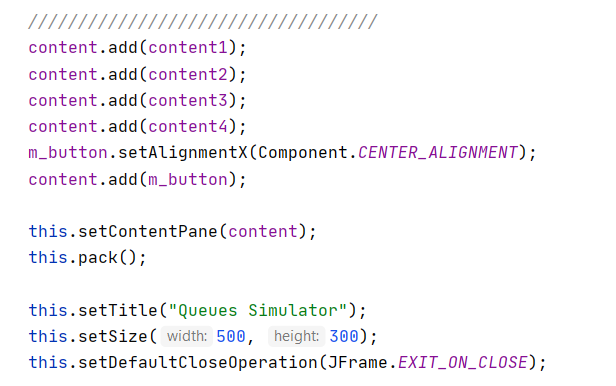
Toate acestea sunt posibile datorita faptului ca “QueueView” mosteneste clasa JFrame:



astfel incat constructorul clasei noastre creeaza insasi fereastra cu care va interactiona utilizatorul (interfata). Vom folosi tot felul de obiecte din pachetele “swing” si “awt” de tip JLabel, JTextField, JButton, si JPanel de care un user se va folosi pentru transmiterea de date si informatii, iar un programator pentru prelucrarea si modelarea lor.



Am impartit Frame-ul in mai multe panouri (panel-uri) pentru o mai buna organizare a componentelor din GUI, folosind metode de tip setLayout, add, pack, setAlignment.



* *Pachetul “business\_logic”*

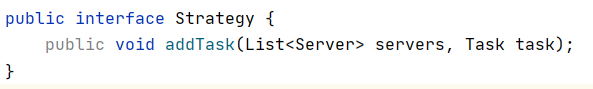
***Enumul “SelectionPolicy”***

Va ajuta in alegerea unei politici de selectare portivite pentru simulare:

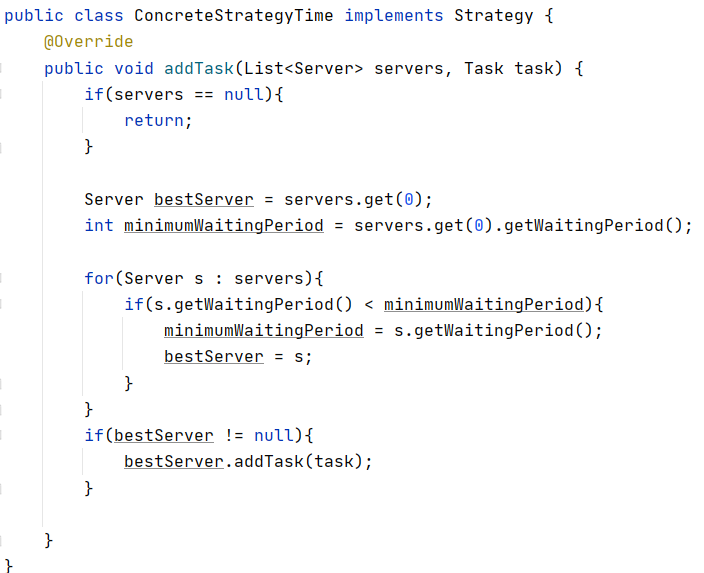
Avem optiunea de a alege intre *shortest\_time (*cel mai scurt timp*),* sau *shortest\_queue (*cea mai scurta coada*),* atunci cand facem adaugarea de clienti in serverele unei simulari.

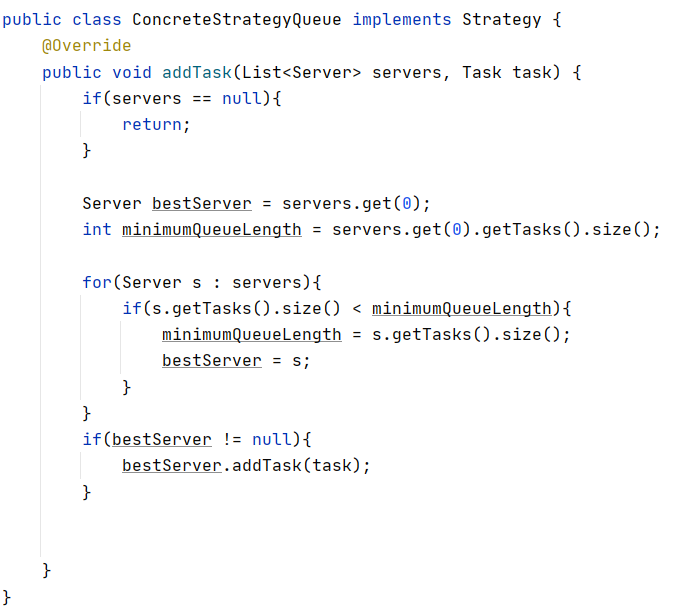
***Interfata “Strategy” si cele doua clase care o implementeaza***

Pentru a avea o singura functie de addTask, alegem sa folosim o interfata cu doua clase:



In clasa ***ConcreteStrategyTime*** se va cauta coada cu cel mai mic timp de asteptare si va adauga task-ul, clientul la aceasta, in timp ce in ***ConcreteStrategyQueue*** se va folosi ca si coada optima cea in care se afla cei mai putini clienti la momentul respectiv.

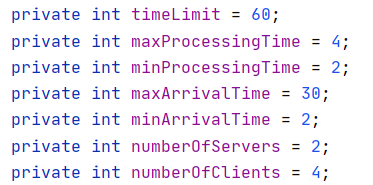




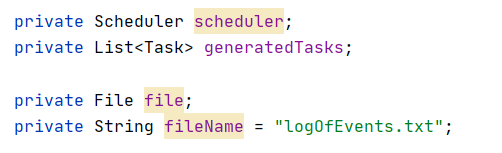
***Clasa “SimulationManager”***

Este clasa cu care vom putea porni efectiv toata simularea, threadurile si care va controla buna desfasurarea a acesteia:

Aici vom gasi parametrii / datele pe care utilizatorul le ofera aplicatiei:



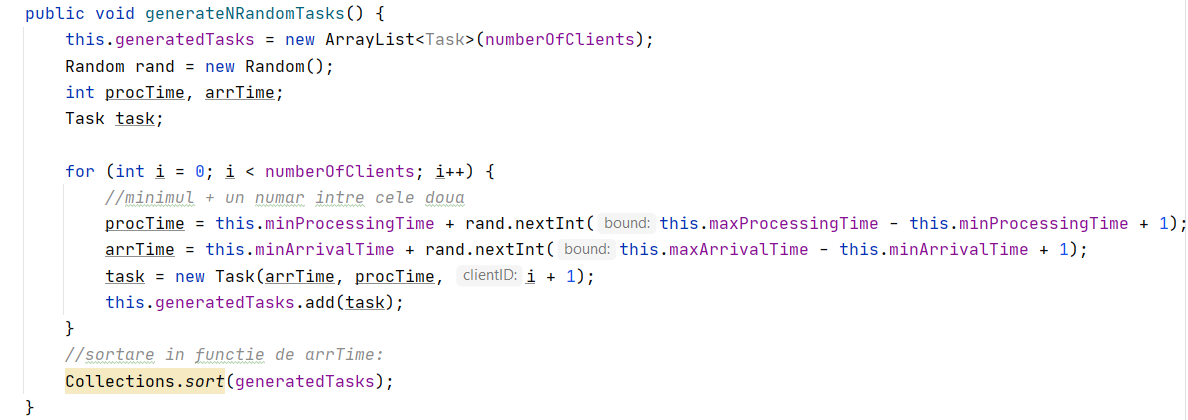
Tot aici vom avea un obiect de tip Scheduler in care se va afla lista noastra de cozi, o lista de clienti ce vor fi generati in mod aleator prin functia “generateNRandomTasks” si care vor fi dispusi serverelor / threadurilor pornite, cat si un numele unui fisier in care sa tinem evidenta evenimentelor ce au loc de la o secunda la alta a simularii:



Metodele clasei:

* *generateNRandomTasks*:

Se va folosi de datele introduse de utilizator pentru a genera N clienti random cu proprietatea ca: parametrii de tip “timp” vor fi alesi in intervalele de timp mentionate, iar ID-ul lor sa fie unic (este incrementat cu ajutorul “i”-ului din “for”). Acestia vor fi si sortati in functie de arrTime, astfel incat simularea sa ruleze cu succes.



* *run*

Folosindu-se de un while ca sa verifice ca timpul pe care il petrece pentru a indeplini sarcinile nu trece peste cel dat de catre utilizator ca timp limita, va asigna scheduler-ului cate un client daca timpul lui de sosire se potriveste cu cel curent al simularii. Daca task-ul a fost adaugat, se va scoate din lista si se va face verificarea cu urmatorul.

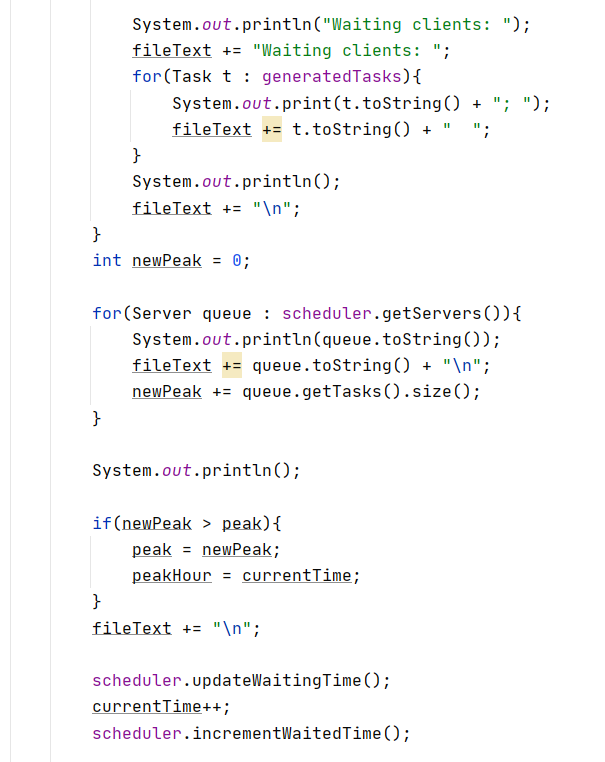
Daca nu mai sunt clienti potriviti cerintei, se trece mai departe, se afiseaza cozile, se verifica numarul de clienti din cozi (pentru a tine evidenta orei de varf), se incrementeaza timpul curent al simularii si se actualizeaza informatiile cu privire la timpii de servire a clientilor din cozi si la timpii de asteptare ale acestora.

Aplicatia nu se opreste doar atunci cand ajungem la timpul limita de simulare, astfel incat nu vrem sa o tinem pornita degeaba, deci se va face la sfarsit o alta verificare prin care ne asiguram atat ca nu mai sunt clienti in lista ce asteapta sa fie asignati unui server, cat si ca nu mai sunt cozi active.

Cand se iese din while, apelam metoda “stop” din “scheduler”, oprind, astfel, toate threadurile / cozile din a mai rula si insemnand timpul mediu de asteptare, ora de varf si faptul ca simularea a ajuns la sfarsit.

* *addToFile*

Va initializa un obiect de tip FileWriter cu care vom nota in fisierul text deschis datele adunate in metoda “run”.

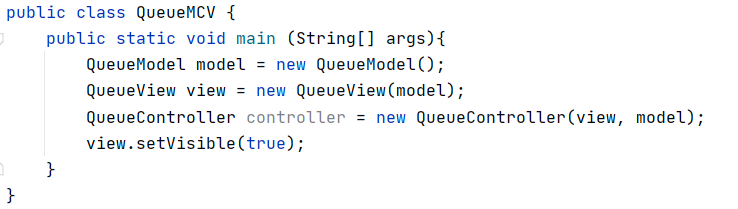
.



* *Pachetul “main”*

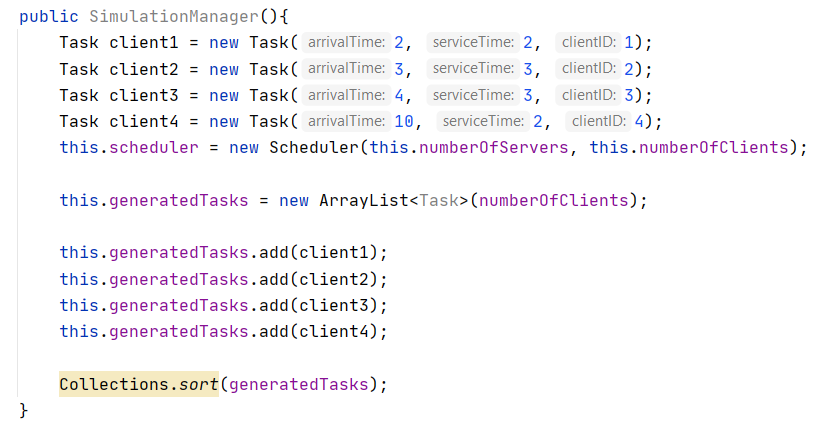
***Clasa “QueueMCV”***

Aceasta clasa porneste aplicatia si permite lucrul cu GUI, facand legatura intre toate celelalte clase si stabilind relatiile care vor duce la buna functionare a interfetei grafice si a programului din spatele acesteia.



1. Rezultate

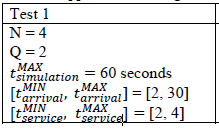
Am testat aplicatia folosindu-ma mai intai de un constructor al SimulationManager prin care am creat cei 4 clienti din exemplul dat in materialul de la laborator:



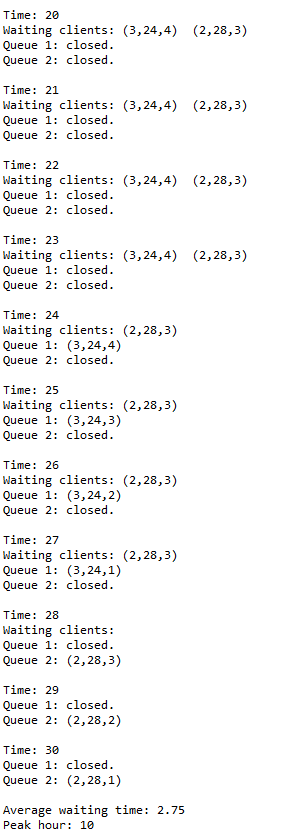
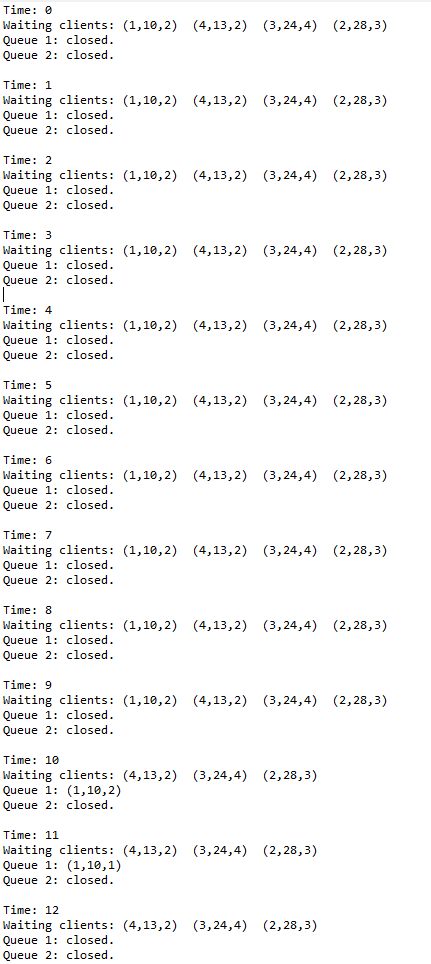
Avand rezultatele similare, am efectuat cele 3 teste propuse tot in materialul respectiv si am generat cele 3 fisiere \*.txt:

Exemplul 1:

***Input***



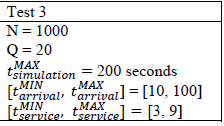
***Rezultat***

******

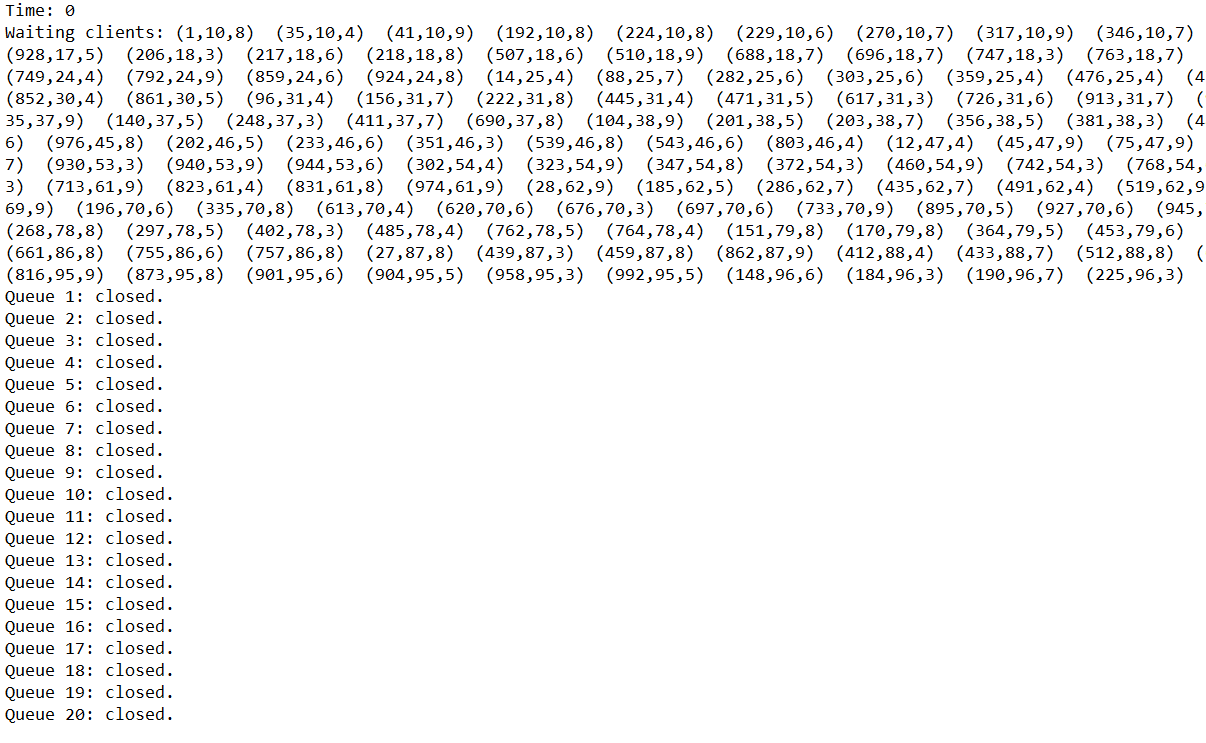
1. Concluzii

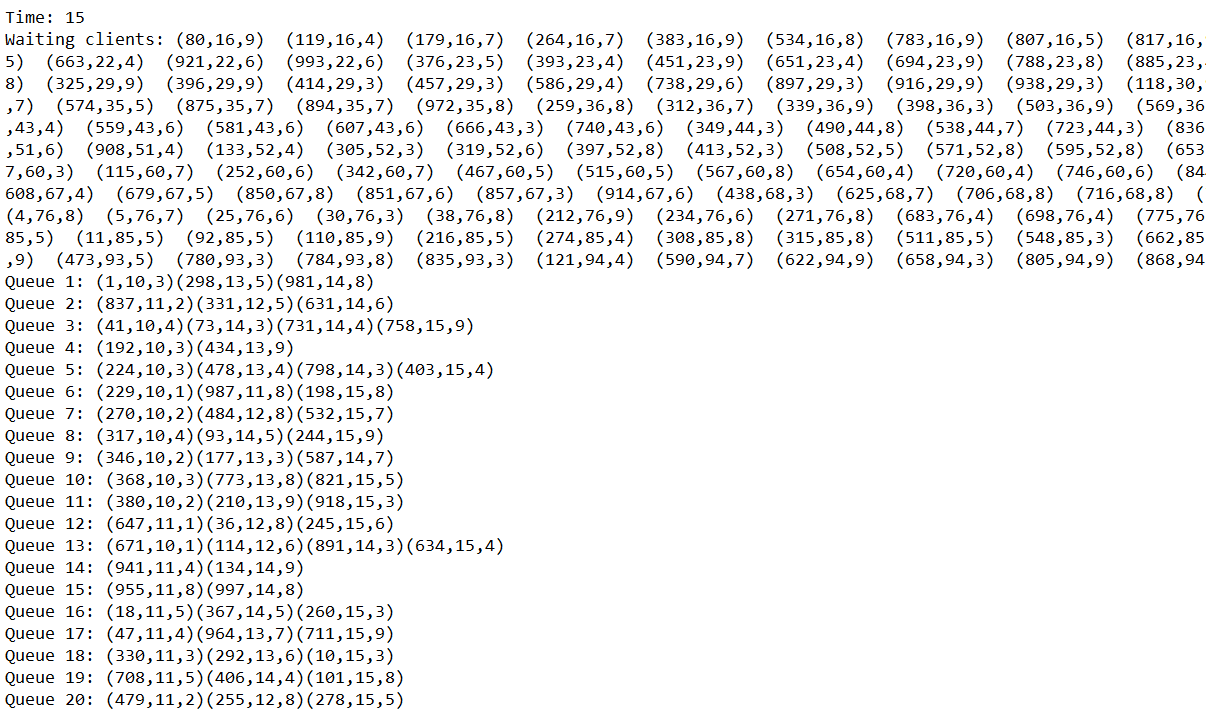
Exemplul 3:

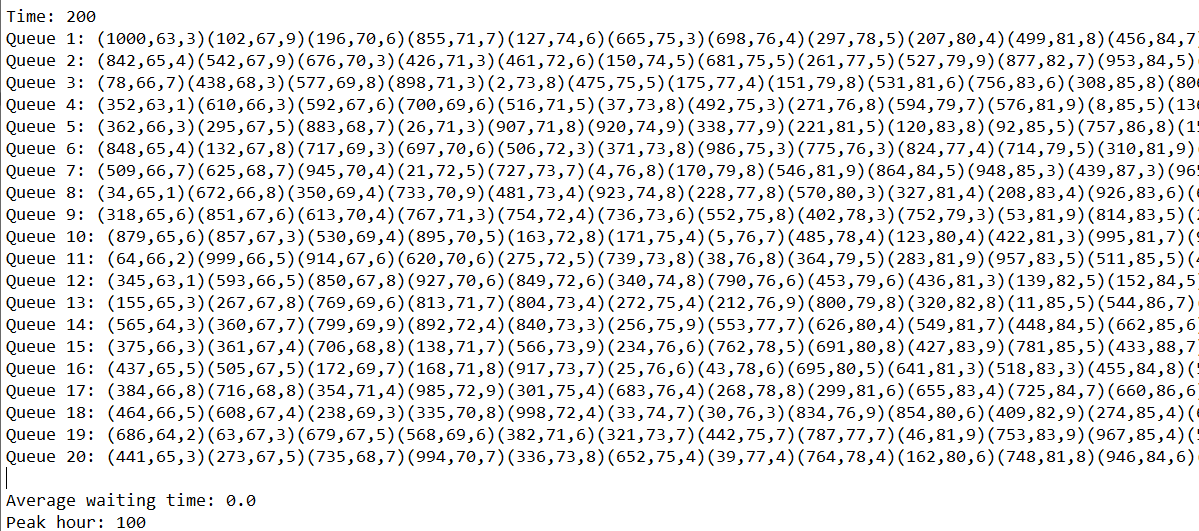
***Input***

******

***Rezultat***

******

******

******

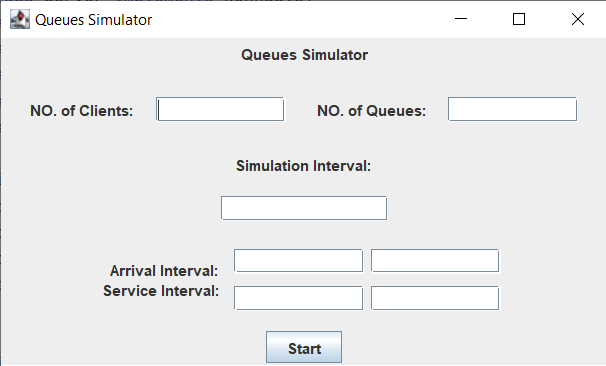
1. Concluzii

Prin implementarea acestui proiect s-a urmarit atat lucrul cu cozile, thread-urile, cat si intelegerea management-ului unui sistem bazat pe acestea, folosind multithreading-ul astfel incat toate serverele sa lucreze in paralel, sa fie sincronizate si sa fie sigure, organizarea codului pentru o mai buna si usoara intelegere.

Noutati pentru mine, fata de proiectul trecut, a fost lucrul cu thread-uri, ideea de BlockingQueue si AtomicInteger, si lucrul cu fisiere.

As fi putut adauga proiectului niste metode, ori chiar una, prin care sa fi facut afisarea in timp real a evenimentelor in interfata grafica, cat si alte metode de verificare a acorectitudinii datelor: ex. pentru intervale sa verific timpMin <= timpMax, etc..

Varinata finala a proiectului (GUI-ul):



1. Bibliografie

[*https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html*](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html)

[*https://howtodoinjava.com/java/multi-threading/atomicinteger-example/*](https://howtodoinjava.com/java/multi-threading/atomicinteger-example/)

[*https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html*](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html)

[*https://www.w3schools.com/java/java\_threads.asp*](https://www.w3schools.com/java/java_threads.asp)

[*https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html*](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html)

[*https://www.w3schools.com/java/java\_files\_create.asp*](https://www.w3schools.com/java/java_files_create.asp)

[*https://www.youtube.com/watch?v=eQk5AWcTS8w*](https://www.youtube.com/watch?v=eQk5AWcTS8w)

[*https://www.youtube.com/watch?v=a\_LBuCx1KTE*](https://www.youtube.com/watch?v=a_LBuCx1KTE)

*FUNDAMENTAL PROGRAMMING TECHNIQUES*

*ASSIGNMENT 2 – SUPPORT PRESENTATION -Teams-*