

**FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE**

**~SPECIALIZAREA: CALCULATOARE SI TEHNOLOGIA INFORMATIEI~**

**Grupa 30226**

**An 2, Semestrul 2**

**Arseniuc Anamaria**

**Queues Simulator**

**-Documentatie-**

**Cuprins**

1. Obiectivul temei . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .pg 3
2. Analiza problemei, modelare, scenarii, cazuri de utilizare . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . pg3-4
3. Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . pg 4-6
4. Implementare . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . pg 7-9
5. Rezultate . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . pg 9
6. Concluzii . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . pg 10
7. Bibliografie . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .pg 10
   1. **Obiectivul temei**

“*Queues are commonly used to model real world domains. The main objective of a queue is to provide a place for a "client" to wait before receiving a "service". The management of queue based systems is interested in minimizing the time amount their "clients" are waiting in queues before they are served. One way to minimize the waiting time is to add more servers, i.e. more queues in the system (each queue is considered as having an associated processor) but this approach increases the costs of the service supplier. When a new server is added the waiting customers will be evenly distributed to all current available queues. The application should simulate a series of clients arriving for service, entering queues, waiting, being served and finally leaving the queue. It tracks the time the customers spend waiting in queues and outputs the average waiting time. To calculate waiting time we need to know the arrival time, finish time and service time. The arrival time and the service time depend on the individual clients – when they show up and how much service they need. The finish time depends on the number of queues, the number of clients in the queue and their service needs.”*

**Cerinta Proiectului**

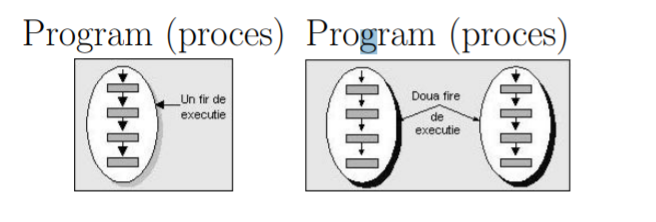
*“Proiectati si implementati o aplicatie de simulare care vizeaza analiza sistemelor bazate pe cozi de asteptare cu scopul de a determina si minimiza tipul de asteptare al clientilor.”*

**Obiectivul principal** al acestei temei este de a implementa o aplicatie care simuleaza un anumit numar dat de catre utilizator de cozi cu clienti, avand in vedere mai multe aspecte care ar putea sta la baza unei mai bune fluidizari ale acestora, astfel se simuleaza o serie de clienti, generati random la un anumit interval de timp setandu-li-se momentul de timp la care au ajuns la casa si timpul de service, urmand ca la plecarea de la casa de marcat sa i se seteze timpul. Se tine cont de plasarea acestora la niste case de marcat (cozi), optimizand alegerea acestora in functie de timpul de asteptare pe fiecare casa de marcat.

* 1. **Analiza problemei, modelare, scenarii, cazuri de utilizare**

1. **Analiza problemei:**

Un *fir de executie* (thread) este o succesiune secventiala de instructiuni care se executa ın cadrul unui proces. In Java putem crea threaduri fie extinzand clasa Thread, fie implementand interfata Runnable. Este important insa ca in clasa noastra thread sa existe o metoda *public void run()* care constituie corpul principal de instructiuni care se va executa in cadrul threadului.



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.

Am considerat ca pentru simularea sistemelor bazate pe cozi trebuie sa avem in considerare ca numarul cozilor este aleator, el este dat de catre un simulator, astfel algoritmul utilizat pentru acesta trebuie sa fie unul general si sa respecte simularea cozii ca si in viata reala. Pentru realizarea simularii cozilor am luat modele din viata reala: asteptarea la diferite cozi la un magazin, asteptarea la banca pentru plata facturilor sau pentru efecturarea diferitelor tranzactii.

Am rezolvat problema folosind sapte clase: clasele Task si Server (din pachetul TS) , clasa Scheduler (din pachetul cu acelasi nume) si clasa SimulationManager impreuna cu clasa principala, Main (din pachetul Simulation). Analizand asteptarea clientilor la coada trebuie sa luam in considerare *ca un client nu poate iesi mai repede din coada daca este altcineva inaintea sa.* El trebuie sa astepte sa plece clientul dinaintea sa si abia dupa plecarea acestuia sa aiba comanda procesata si sa plece din coada. Mai trebuie sa avem in vedere ca un client va merge tot timpul la casa la care asteapta mai putini oameni, deoarece timpul sau de asteptare va fi mai mic.

1. **Modelarea:**

Cozile sunt de obicei utilizate pentru a modela domenii din lumea reală. Obiectivul principal al unei cozi este pentru a oferi un loc pentru un „client“ să aștepte înainte de a primi un „serviciu“. Managementul coadă sisteme pe bază este interesat în minimizarea cantității timpului lor „clientii“ sunt în așteptare în cozi înainte ca acestea sa fie servite.

Orice coadă poate fi văzută ca o pereche casă de servire - client care așteaptă la coadă, această corespondență fiind modelată după următorul aspect : fiecare coadă are clienți care trebuie procesați. Dar clientul poate să aleagă ( în funcție de numărul de case de servire puse la dispoziție) cărei case de servire să fie asociat. Modul de asociere depinde de cum este văzută problema.

1. **Cazuri de utilizare:**

Simularea se poate folosi in viata reala pentru a determina anumite date , cum ar fi timpul mediu de asteptare al cozilor la un supermarket, la banca pentru plata facturilor sau pentru efecturarea diferitelor tranzactii, sau la spitale.

* 1. **Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages)**

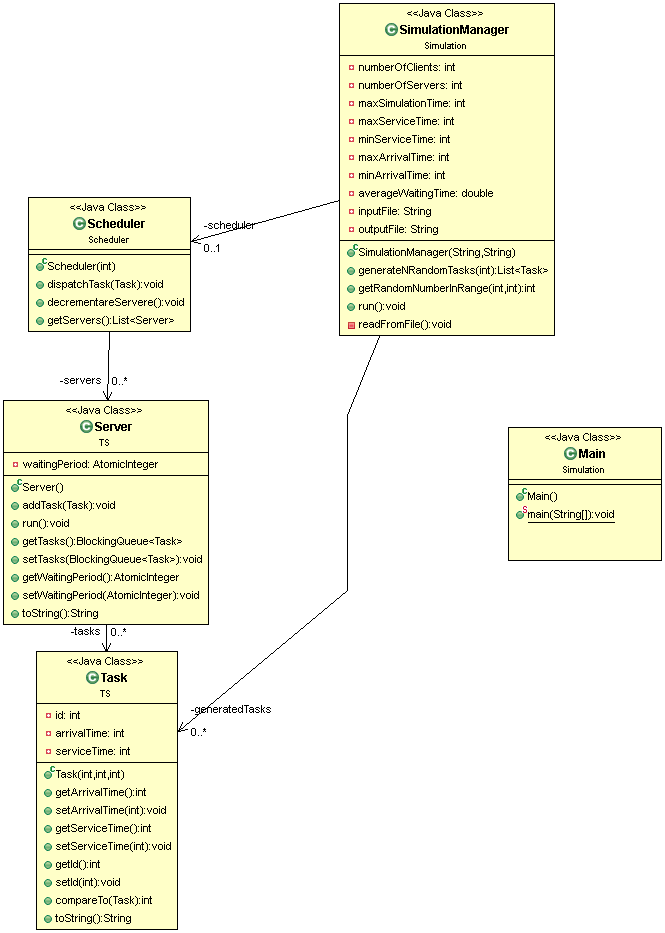
Proiectul a fost facut in Eclipse-Java, iar limbajul de programare folosit este Java.

Proiectarea acestei teme pe baza cerintelor are ca arhitectura urmatoarele: componentele aplicatiei, si modul de comportare in diverse situatii date de catre utilizator prin intermediul fisierelor text.

**Diagrama UML**

UML (Unified Modeling Language) este un limbaj standard pentru descrierea de modele si specificatii software. UML a fost la bază dezvoltat pentru reprezentarea complexității POO al căror fundament este structurarea programelor pe clase, și instanțele acestora (numite și obiecte ). Cu toate acestea,datorită eficienței și clarității în

reprezentarea unor elemente abstracte, UML este utilizat dincolo de domeniul IT.



**Diagrama UML**

**Pachete si Clase**

Pentru o mai buna organizare a acestui proiect, l-am implementat in 3 pachete, si anume:

1. **Pachetul TS :** cuprinde clasele de pornire, si anume *Task* in care tinem id-ul, timpul de sosire si timpul de procesare si clasa *Server* care e construita dintr-o coada (BlockingQueue) de clienti (task-uri)
2. **Pachetul Scheduler :** care cuprinde clasa cu acelasi nume, fiind responsabila cu trimitirea clientilor la cozile cele mai avantajoase pentru ei, deoarece se doreste ca clientul sa se puna la coada cu cel mai mic timp de asteptare si sa fie procesat cat mai repede
3. **Pachetul Simulation :** are clasa *SimulationManager* (genereaza random “n” client, cu un timp de sosire si de procesare cuprinse intre limitele citite din fisier, face simularea si afiseaza datele intr-un alt fisier) si clasa principal, *Main*, care creeaza un obiect din clasa anterioara si imi porneste thread-ul pentru acesta.

**Structuri**

**BlockingQueue**

**Pentru construirea unei cozi de Task-uri am folosit interfata J**ava BlockingQueue, pentru a putea adauga si a sterge concurrent clienti. Termenul de blocare a cozii vine de la faptul că Java BlockingQueueeste capabil să blocheze firele care încearcă să insereze sau să ia elemente din coadă. De exemplu, dacă un thread încearcă să ia un element și nu mai rămâne niciunul în coadă, thread-ul poate fi blocat până când există un element de luat. Dacă blocajul de apelare este sau nu depinde de ce metode apelați BlockingQueue.

Un BlockingQueue este de obicei utilizat pentru ca un fir de executie (Thread) să producă obiecte, pe care un alt fir îl consumă. Iată o diagramă care ilustrează acest principiu:

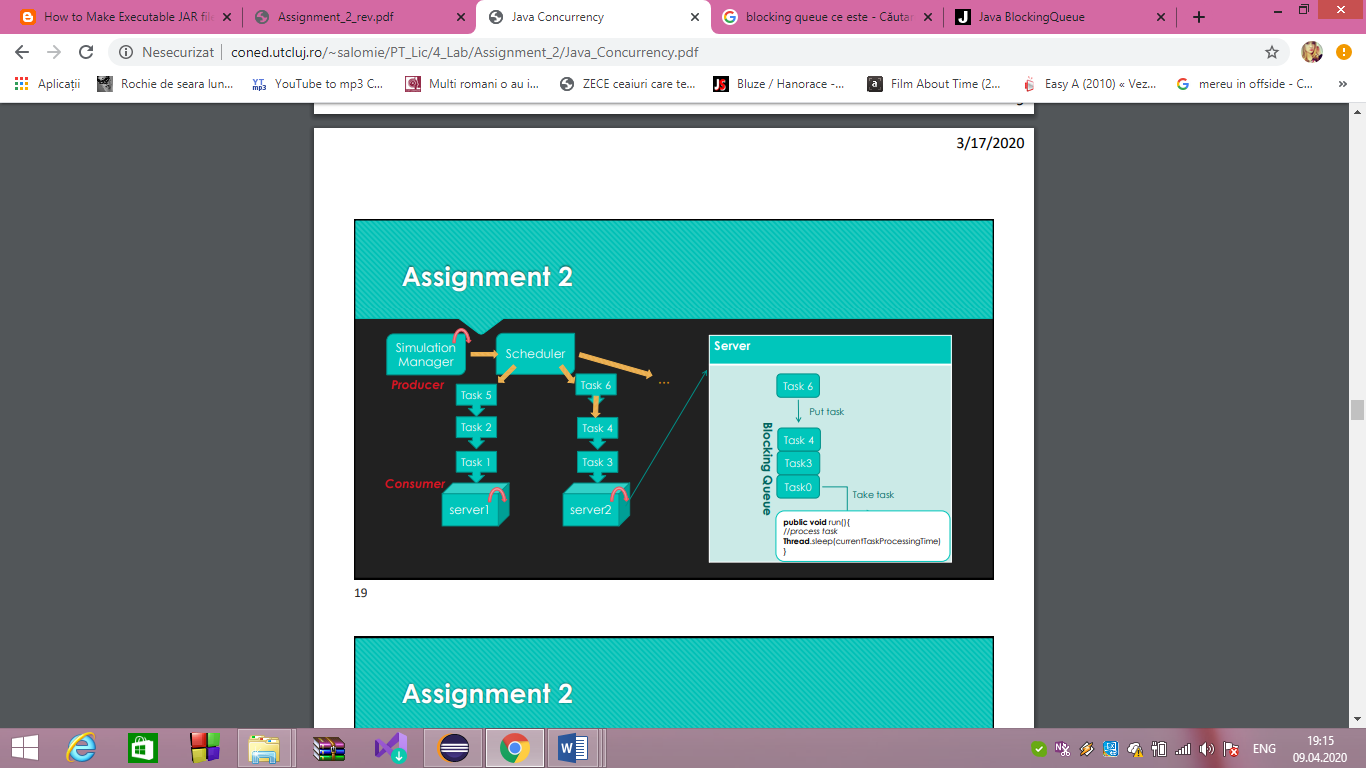
|  |
| --- |
| Un BlockingQueue cu un thread pus în el, și un alt thread preluând din el. |
| ***Un BlockingQueue cu un thread pus în el, și un alt thread preluând din el.*** |

**AtomicInteger**

Operatiile atomice sunt necesare intr-o aplicatie multithreading pentru a evita inconsistenta datelor.

1. **Implementare**

**Metode si Clase**



1. ***CLASA TASK:*** contine urmatoarele attribute de clasa : id-(id-ul clientului), arrivalTime a (timpul de sosire a clientului) si serviceTime (timpul de servire a clientului).

Metode: contine *gettere, settere* pentru atributele clasei,

*toString()* - o metoda utilizata pentru afisarea datelor clientului.

1. ***CLASA SERVER***: e construita dintr-o coada de Task-uri, si o varibila atomica WaitingPeriod care se decrementeaza de curentul thread odata ce un Task este complet si se incrementeaza de clasa Scheduler atunci cand se adauga noi Task-uri in cozi.

Metode: *addTask()*- adauga un nou task intr-o coada si incrementeaza variabila atomica WaitingPeriod

*Run()* – punem pe sleep thread-ul cozii respective cu un timp egal cu timpul de procesare al task-ului si decrementam variabila WaitingPeriod

*Gettere si settere* pentru atributele clasei

*toString()* pentru afisarea cozii de Task-uri in fisier.

1. ***CLASA SCHEDULER :*** are un singur atribut (o lista de Servers / cozi), Constructorul clasei creeaza “noOfServers” Thread-uri si le si porneste totodata.

Metode*: dispatchTask(Task t)-* metoda primeste ca si parametru un task si are rolul de a-I gasi coada/serverul cel mai potrivit (cu timpul de asteptare cel mai mic), si dup ace il gaseste, il adauga in coada.

*decrementareServere(*) – decremneteaza timpii de procesare a clientilor care sunt primii in coada, iar daca mai are de stat doar un timp in coada, la urmatoarea iteratie este scos din coada

*getServers()* – returneaza lista de Servers

1. ***CLASA SIMULATIONMANAGER***- cea mai complexa, cea in care se face citirea si afisarea din/in fisier, cat si prelucrarea datelor. Ca si attribute ar fi numberOfClients, numberOfServers, maxSimulationTime, maxServiceTime, minServiceTime, maxArrivalTime, minArrivalTime – date scrise de utilizator in fisierul de intrare, averageWaitingTime- timpul mediu de asteptare al clientilor, o lista de task-uri -> generatedTasks, acei clienti care sunt generati random in functie de limitele impuse de utilizator si care asteapta a fi prelucrati.

In constructorul clasei se extrage path-ul fisierelor de intrare si iesire date ca si argumente in linia de comanda, apelam metoda de citire a datelor din fisierul de intrare si pastrarea acestor in atributele clasei pentru a fi utilizate mai tarziu, creem 1 obiect instanta a clasei Scheduler si pastram in variabila generatedTaks o lista de task-uri rezultata in urma apelarii metodei generateNRandomTasks(numberOfClients).

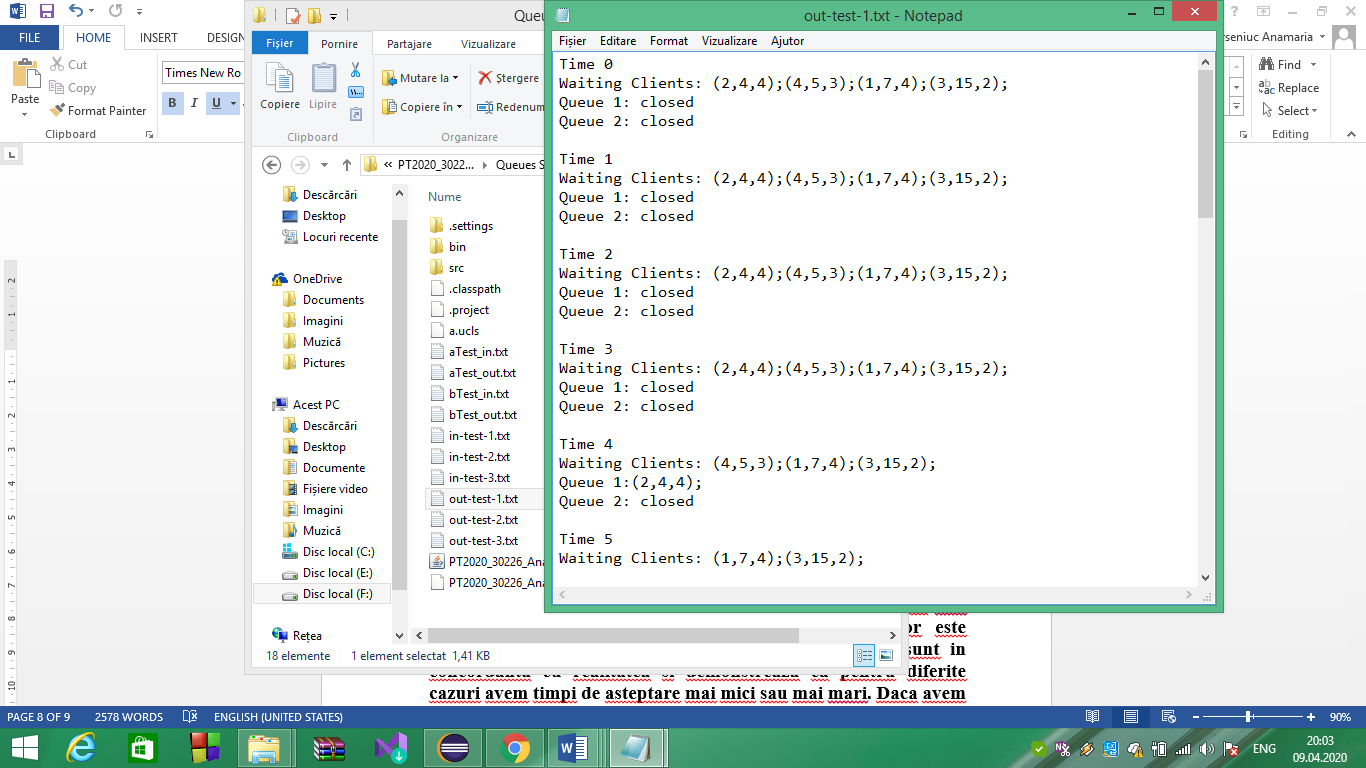
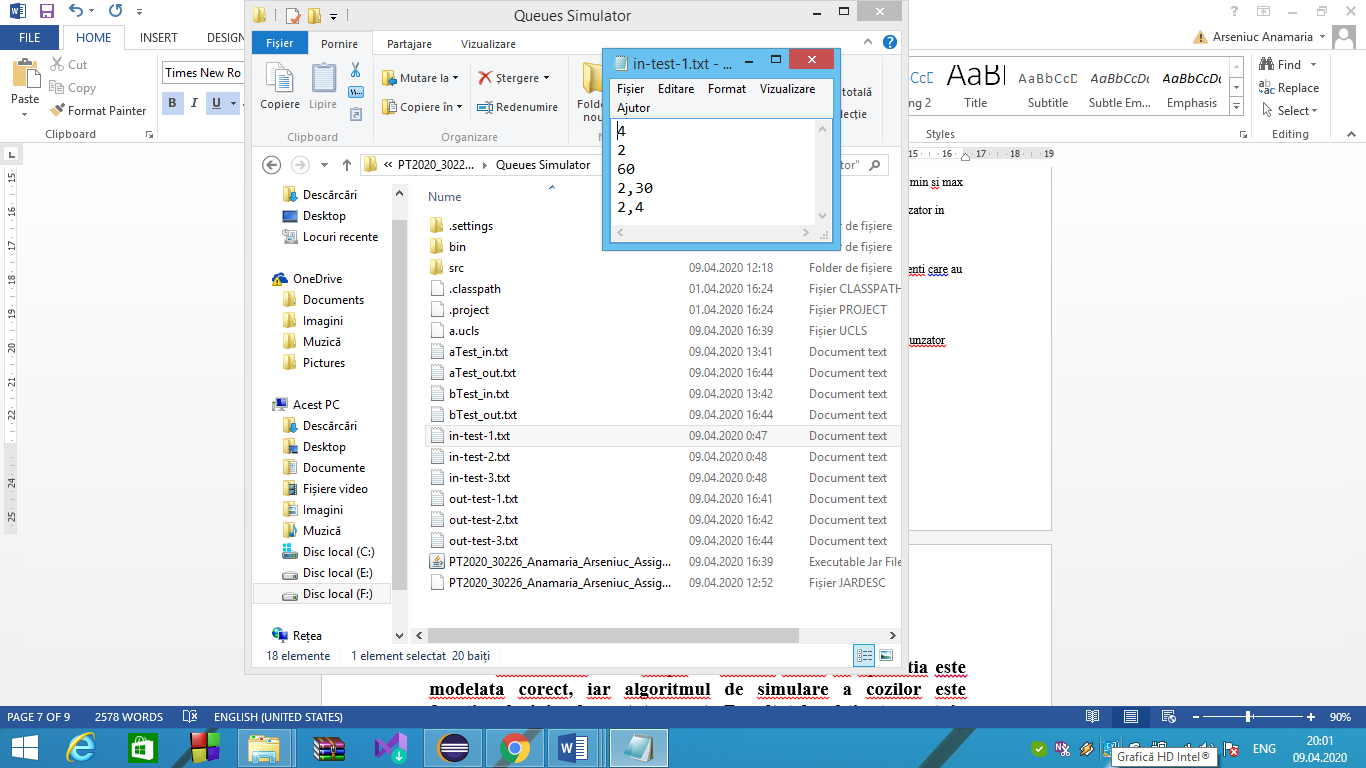
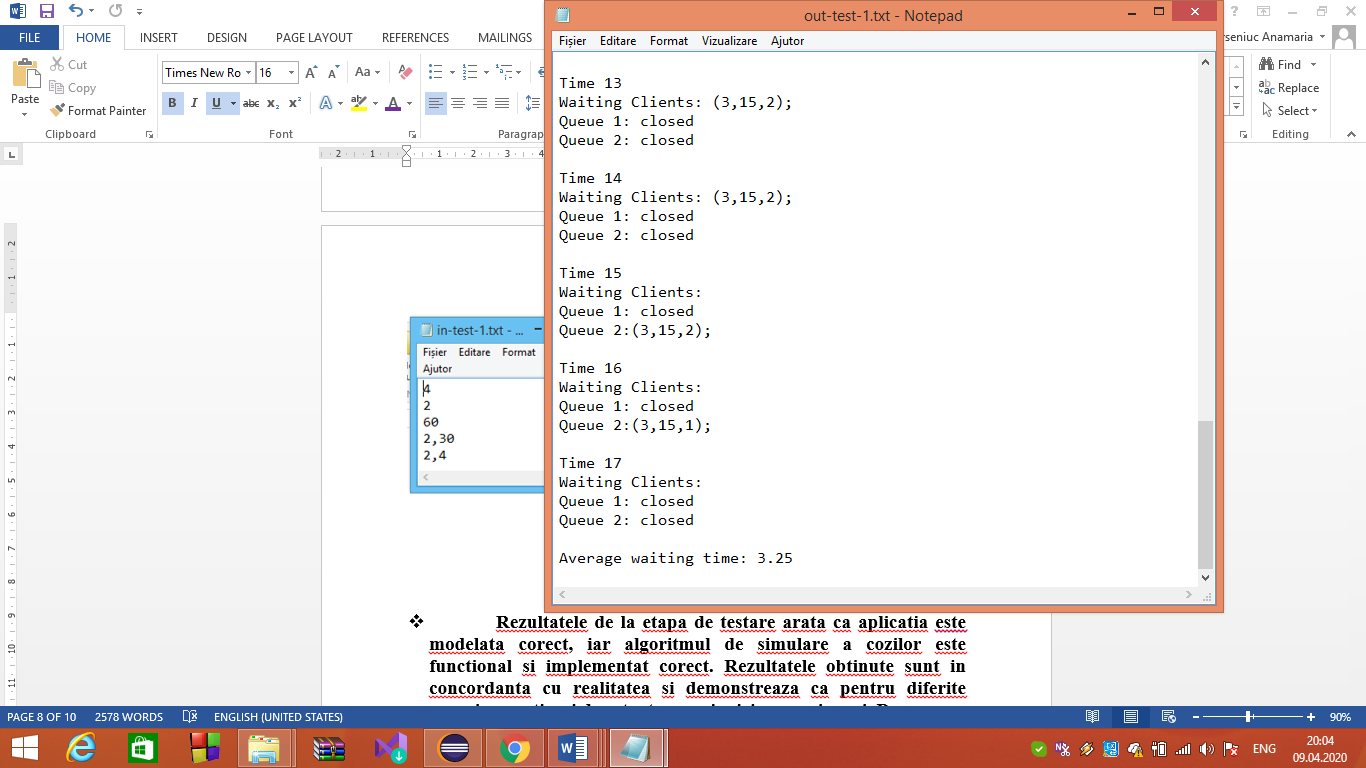
Metode*: generateNRandomTasks(numberOfClients)-* returneaza o lista de N client generati random

*getRandomNumberInRange(int min, int max)-* returneaza un numar random cuprins intre min si max

*readFromFile()-* citeste datele (cu ajutoru Scanner-ului) din fisierul de intrare dat de utilizator in linia de comanda si le pastreaza in atributele clasei

*run()* – imi deschide fisierul de iesire (folosind BufferedWriter) si se prelucreaza datele corespunzatoare cerintei (se pun in cozi clientii, se calculeaza timpul mediu de asteptare, se sterg acei clienti care au timpul de procesare < 1 )

1. ***CLASA MAIN*** – creeaza un obiect instanta al clasei SimulationManager si porneste un THREAD corespunzator acestuia.



**.**

**.**

**FISIERE DE INTRARE**

**SI IESIRE**

**numberOfClients**

**numberOfServers**

**maxSimulationTime**

**minArrivalTime,maxArrivalTime**

**minServiceTime,maxServiceTime**

**.**

**.**

1. **Rezultate**

Rezultatele de la etapa de testare arata ca aplicatia este modelata corect, iar algoritmul de simulare a cozilor este functional si implementat corect. Rezultatele obtinute sunt in concordanta cu realitatea si demonstreaza ca pentru diferite cazuri avem timpi de asteptare mai mici sau mai mari. Daca avem timpi mai mici de aparitie a clientilor si timpi mai mari de procesare a comenzilor, atunci, pentru a micsora timpul de asteptare a clientilor, este indicat sa avem mai multe cozi deschise. Programul permite, asadar, utilizatorului sa-si faca o idee despre cum ar trebui organizate cozile, pentru a reduce timpul de asteptare, sau daca trebuie sa astepti la coada sa stii sa te duci coada cu timpul de asteptare cel mai mic.

**6.Concluzii**

O concluzie a acestei teme ar fi chiar daca avem mai multe cozi, sau timp de procesare mic pentru a avea un timp de asteptare cat mai mic, in viata reala se poate ca un client sa se puna la coada, sa uite sa ia ceva, sa plece la coada si dupa sa revina si astfel se poate mari timpul de asteptare. Un alt fapt care ar putea sa creasca timpul de asteptare este atunci cand de exemplu la magazin sau la banca se greseste comanda, clientul este taxat mai mult, iar casa trebuie sa remedieze greseala, astfel timpul de asteptare creste datorita unei erori umane. Chiar daca simularea este corecta, ea nu poate prevedea toate situatiile neasteptate din viata de zi cu zi.

In aceasta tema am invatat cum sa lucrez cu thread-urile pentru a putea executa in acelasi timp mai multe metode necesare pentru a putea simula sistemul bazat pe cozi .

Ca si **dezvoltari ulterioare** ar fi urmatoarele:

* realizarea unei interfete grafice pentru a se introduce si a schimba datele de intrare mai usor de catre utilizator
* incercarea de implementare a situatiilor neprevazute, cum ar fi marirea timpulului de procesare al unei comenzi la un moment aleatoriu;

1. **Bibliografie**

<http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html#pageToc>

<https://stackoverflow.com/>

<http://inf.ucv.ro/documents/tudori/laborator8_53.pdf>

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html>

<https://www.w3schools.com/java/java_files_create.asp>

<https://www.baeldung.com/java-write-to-file>