**- TEMA II -**

**SIMULAREA SI ANALIZAREA UNUI SISTEM BAZAT PE COZI**

Marc Andrei – Aurel

Grupa 30224

Laborator – Tehnici de Programare

Cuprins

1. Obiectivul temei
2. Analiza problemei
3. Proiectare
4. Implementare
5. Rezultate
6. Concluzii
7. Bibliografie

Obiectivul temei

Conceptul de thread (fir de execuție) definește cea mai mică unitate de procesare ce poate fi programată spre execuție de către [sistemul de operare](https://ro.wikipedia.org/wiki/Sistem_de_operare). Este folosit în programare pentru a eficientiza execuția programelor, executând porțiuni distincte de cod [în paralel](https://ro.wikipedia.org/wiki/Calcul_paralel) în interiorul aceluiași [proces](https://ro.wikipedia.org/w/index.php?title=Proces_(sisteme_de_operare)&action=edit&redlink=1). Câteodata însă, aceste portiuni de cod care constituie corpul thread-urilor, 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 thread-uri înainte de a continua propria execuție, se numește sincronizarea thread-urilor.

Coada este o structura de date logica (implementarea este facuta utilizand alte structuri de date) si omogena (toate elementele din care este compusa sunt de acelasi tip). Structura are doua operatii de baza, si anume crearea cozii, listarea sau parcurgerea cozii, prelucrarea, adaugarea si extragerea unui element din coada. In afara acestor operatii fundamentale, pot fi implementate si alte operatii utile, precum testul de structura vida, obtinerea primului element. Disciplina de acces pe care se bazeaza coada este FIFO ( First In, First Out ). Coada poate fi implementata in mai multe moduri, cea mai utilizata implementare fiind cea utilizand liste sau vectori.

**Obiectivul temei** este dezvoltarea unei aplicatii ce simuleaza si analizeaza sistemele bazate pe cozi pentru determinarea si minimizarea timpului de asteptare al clientilor. Evolutia cozilor poate fi vizualizata prin intermediul interfetei.

Cozile sunt utilizate, in general, pentru modelarea domeniilor din viata reala. Principalul obiectiv al unei cozi este de a furniza un loc de asteptare pentru un client, in vederea primirii unui serviciu. Management-ul sistemului bazat pe cozi consta in minimizarea timpului de asteptare a clientilor in cozile de care apartin, inainte sa fie serviti. In timp ce clientii apar, se va cauta coada cu timpul de asteptare minim. Aplicatia simuleaza aparitia unor clienti random, intrand in cozi, asteptand, primind servicii pentru un anumit timp.

Datele de input sunt: intervalul minim si maxim in care clientii pot sa ajunga, intervalul minim si maxim al timpului de servire, numarul de cozi, intervalul de simulare.

Datele de output sunt: timpul mediu de asteptare, timpul mediu de servire, timpul in care cozile au fost goale, un event log pentru a primi informatii despre clienti (timpul la care au ajuns, timpul de servire), evolutia cozii, reprezentata grafic si ora de varf a intervalului de simulare.

Analiza problemei

Tema consta in implementarea unui sistem ce simuleaza un set de cozi, ce sunt umplute cu clienti cu diferite timpuri de aparitie, asteptare si servire. Tinta principala este de a minimiza timpul de asteptare a tuturor clientilor. Pentru a putea realiza acest lucru, avem nevoie de detaliile clientilor, prezentate mai sus, si anume timpul de sosire, timpul de asteptare si timpul de servire. Toti acesti parametri sunt dati random, de clasa **Simulator**, in metoda **generateRandomClients()**, unde se genereaza un numar dat de clienti care la randul sau este random. Acesti clienti generati vor fi introdusi la momentul sosirii lor, in coada cea mai optima, gasita cu ajutorul metodei **getTheMinimalQueue()**, care ia la rand fiecare coada si compara, doi cate doi, timpii de asteptare. Coada cu timpul minim de asteptare va fi returnata iar clientul va fi introdus in capatul cozii.

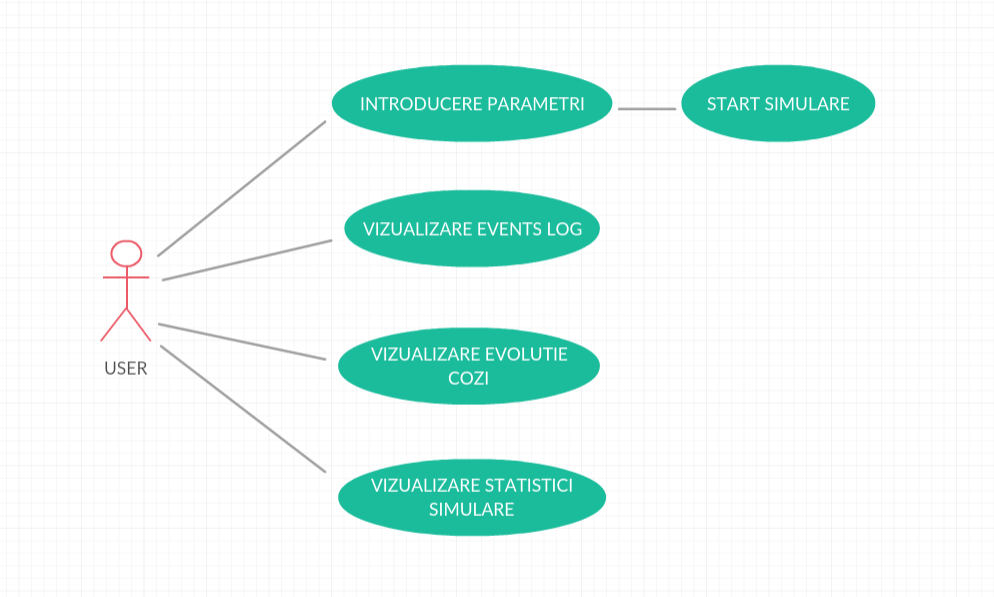
Fiecare coada trebuie sa „stea pe loc” un interval de timp egal cu timpul de servire al primului client din coada. Dupa ce se scurge acest timp, clientul este scos din coada si se ia urmatorul, repetandu-se aceeasi operatie. In momentul in care coada este goala, se numara secundele, rezultand **emptyTime**-ul cozii. Vom folosi, pentru fiecare coada, un thread separat, conform principiului multithreading.

Pentru a afisa Events Log-ul, vom folosi un JTextArea, in care vom scrie din **Simulator**, clientii la timpul corespunzator sosirii lor, precum si timpul la care acestia vor pleca, acesta fiind suma dintre momentul de timp la care respectivul ajunge si timpul de servire. Acest text este afisat din clasa **Queue**, pe masura ce un client este sters din coada.

Pentru a afisa evolutia cozilor, vom folosi tot un JTextArea, in care se vor suprascrie de fiecare data continutul cozilor, astfel putandu-se observa clientii care ajung si pleaca la diferite momente de timp.

Ca date de intrare, vom avea neaparat nevoie de intervalul de sosire al clientilor, intervalul de servire, date prin limita inferioara si superioara a acestora, de timpul de simulare si de numarul de cozi deschise, la care clientii sunt pusi in asteptare.

Diagrama Use Case, care prezinta actiunile ce pot fi efectuate de catre un user prin intermediul interfetei.



Un user obisnuit va putea, conform diagramei, sa introduca parametrii necesari pentru a porni simularea (momentul minim si maxim de sosire, intervalul de timp de servire, intervalul de simulare si numarul de cozi), dupa care, in urma pornirii, poate vizualiza evenimentele in timp real, prin intermediul Events Log-ului, precum si evolutia cozilor si plasarea clientilor in cozi. La sfarsit, ii vor fi disponibile statisticile (timpul mediu de asteptare, timpul mediu de servire, ora de varf, timpul in care cozile au stat goale).

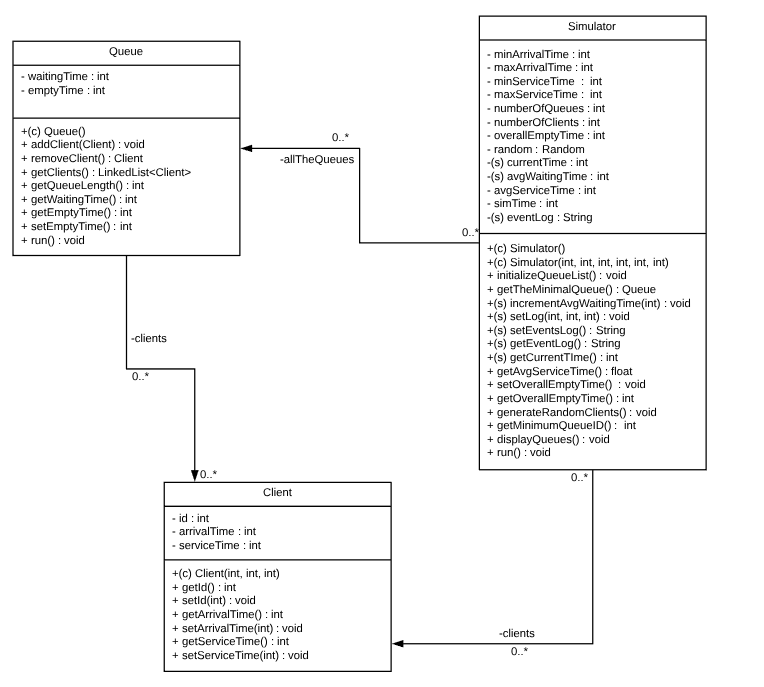
Proiectare

Diagrama **UML** a fost realizată cu ajutorul site-ului [www.lucidchart.com](http://www.lucidchart.com).

Clasa **Client** este clasa ce retine informatii despre un client: id-ul sau, timpul de sosire si timpul de servire. Clasa are ca metode doar getter-e si setter-e, alte metode nefiind necesare.

Clasa **Queue** este clasa care stocheaza un anumit numar de clienti, folosint un LinkedList <Client> . De asemenea, are ca atribute si timpul de asteptare la coada pe care o reprezinta, si timpul in care aceasta ramane goala. In aceasta clasa sunt scosi din coada clientii care isi termina actiunile.

Clasa **Simulator** este clasa care realizeaza simularea propriu – zisa : genereaza clienti random si ii pune in coada potrivita, si face bucla ce ruleaza in intervalul timpului de simulare dat.

Clasa **GUI** este folosita pentru implementarea interfetei cu utilizatorul. Aceasta contine zonele de text unde sunt dati parametrii pentru simulare, un buton pentru a incepe simularea, zone de text pentru afisarea event log – ului si a evolutiei cozii, precum si label-uri pentru afisarea statisticilor din urma simularii.

Structura proiectului dezvoltat este Maven, un sistem de build și administrare a proiectelor, scris în Java. Proiectul este descris într-un fișier XML, denumit POM (Project Object Model) , care conține informații despre module, precum și despre dependențele proiectului. Acesta îi definește proiectului o denumire unică, formată din group ID și artifact ID.

Implementare

Am ales, pentru stocarea clientilor in coada, sa folosesc un LinkedList <Client> , deoarece este mai usor de folosit pentru situatia data, motivul fiind faptul ca se pot realiza operatiile de enqueue si dequeue mai usor, deoarece sunt deja implementate.

Clasa **Client** – folosita pentru stocarea informatiilor despre un client, si anume ID-ul sau, timpul sau de sosire si timpul de servire. Clasa are ca metode doar getter-e si setter-e, nefiind nevoie de altele.

* **Client(id, arrivalTime, serviceTime)** este constructorul clasei
* **getId()** si **setId()** sunt getter-ul, respectiv setter-ul ID-ului clientului
* **getArrivalTime()** si **setArrivalTime()** sunt getter-ul, respectiv setter-ul pentru timpul de sosire al clientului
* **getServiceTime()** si **setServiceTime()** sunt getter-ul, respectiv setter-ul pentru timpul de servire al clientului

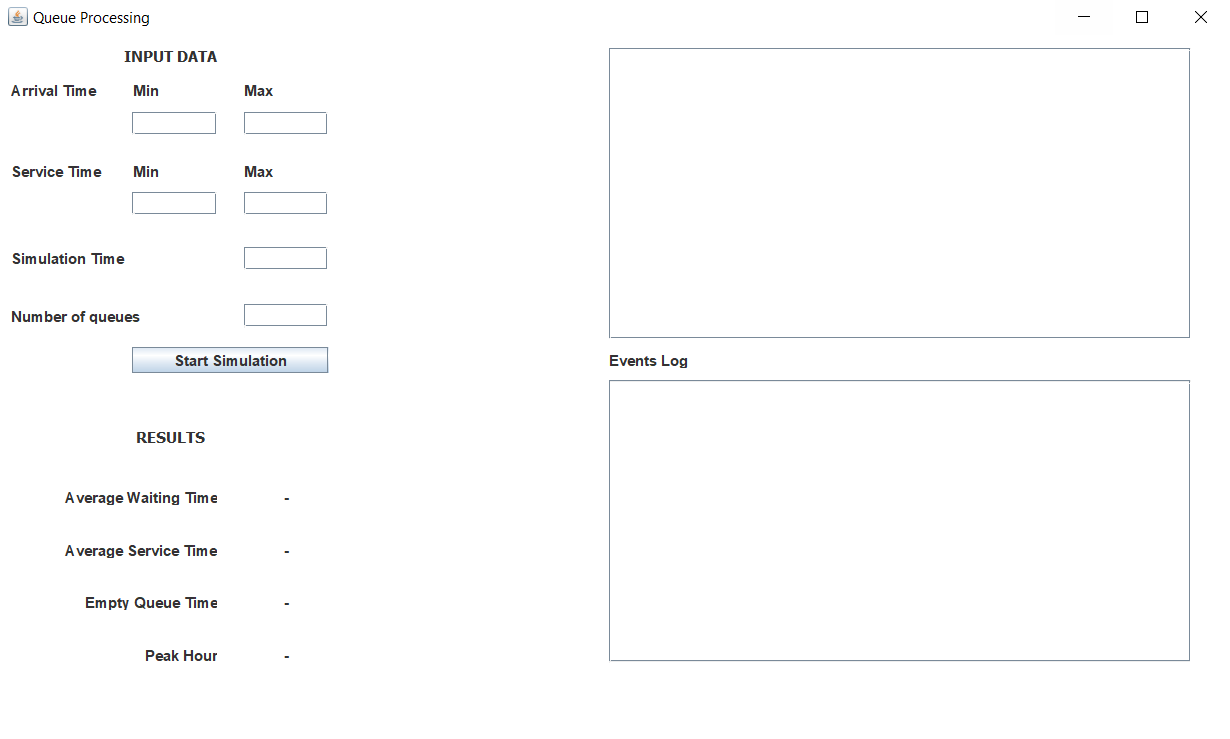
Clasa **Queue** – folosita pentru stocarea unui anumit numar de clienti, prin atributul sau **clients**, un LinkedList <Client>. De asemenea, are ca atribute si timpul de asteptare la coada pe care o reprezinta, si timpul in care aceasta ramane goala.

* **Queue()** este constructorul clasei
* **addClient(Client)** returneaza void, iar functia ei este de a adauga un client in coada
* **removeClient()** returneaza un Client, iar functia ei este de a scoate un client din coada
* **getClients()** returneaza un LinkedList <Client> , iar functia ei este de a returna coada propriu – zisa, retinuta in cadrul clasei
* **getQueueLength()** returneaza int, iar functia ei este de a returna marimea cozii (numarul de clienti din coada)
* **getWaitingTime()** returneaza int, iar functia ei este de a calcula timpul de asteptare al cozii respective
* **getEmptyTime()** si **setEmptyTime()** sunt getter – ul, respectiv setter – ul pentru timpul in care coada este goala.
* **run()** este metoda caracteristica Thread-urilor. Aceasta are functia de a lua prima persoana din coada, de a se pune in asteptare un interval de timp egal cu timpul de servire al clientului din varful cozii, acesta fiind scos din coada ulterior. Daca coada este goala, se incrementeaza atributul de emptyTime si se pune in asteptare un interval de timp care corespunde unei secunde in simulare. In momentul plecarii unui client, se scrie in **events log** detaliile clientului (id-ul si timpul la care acesta a plecat). Se transmite si timpul mediu de asteptare al cozii pentru adunarea sa la timpul mediu de asteptare al tuturor cozilor, din Simulator.

Clasa **Simulator** este clasa care realizeaza simularea propriu – zisa : genereaza clienti random si ii pune in coada potrivita. Are ca atribute timpul minim si maxim de sosire, timpul minim si maxim de servire, numarul de cozi, numarul de clienti, timpul total in care cozile au fost goale, timpul la care s-a ajuns cu simularea, timpul mediu de asteptare si de servire, timpul de simulare, clientii generati random, lista de cozi si event log-ul.

* **Simulator()** si **Simulator(minArrivalTime, maxArrivalTime, minServiceTime, maxServiceTime, numberOfQueues, simTime)** sunt constructorii clasei
* **initializeQueueList()** initializeaza HashMap-ul de cozi cu un numar dat de instante Queue
* **getTheMinimalQueue()** returneaza un Queue si are functia de a gasi coada optima – coada cu cel mai mic timp de asteptare
* **incrementAvgWaitingTime()** are functia de a calcula timpul mediu de asteptare
* **setLog()** creeaza un String din id-ul, timpul de sosire si servire al clientului pentru a-l afisa pe event log
* **setEventsLog()** concateneaza event log-ul cu un String dat ca parametru metodei
* **getEventLog()** este getter-ul pentru event log
* **getCurrentTime()** este getter-ul pentru a obtine timpul curent al simularii
* **getAvgServiceTime()** este getter-ul pentru timpul mediu de servire
* **setOverallEmptyTime()** face suma tuturor timpilor in care cozile sunt goale, iar **getOverallEmptyTime()** este getter-ul pentru timpul in care cozile au fost goale
* **generateRandomClients()** genereaza un numar dat de clienti random, care sunt pusi in structura de data clients. Pe parcurs se calculeaza si timpul mediu de servire
* **displayQueues()** afiseaza evolutia cozilor la fiecare secunda a simularii
* **run()** este metoda caracteristica Thread-urilor. In aceasta metoda se face simularea propriu – zisa. Se initializeaza lista de cozi, se genereaza clientii si se pornesc thread-urile pentru fiecare coada. Apoi, intr-o bucla pana la timpul de simulare, se ia fiecare client, iar daca acesta are timpul de sosire egal cu timpul curent, se adauga in coada optima si se scriu in event log detaliile despre client. Se afiseaza evolutia cozilor, iar apoi este pusa in asteptare un timp asociat unei secunde.

Interfata pentru utilizator a fost creata in clasa **GUI**.



Interfata este una usor de folosit. Utilizatorul trebuie sa introduca parametrii necesari simularii. Dupa ce utilizatorul introduce valori valide, la apasarea butonului „Start Simulation” va porni procesul de simulare. Pe parcursul simularii, clientii care apar vor putea fi vazuti in events log, precum si in TextArea-ul din dreapta sus, unde se vede evolutia cozilor. Statisticile pot fi vizualizate in stanga jos, in label-urile marcate cu „ - ”.

Rezultate

Nu putem testa exact rezultatele folosind JUnit, insa am testat fiecare pas in implementare pentru a asigura faptul ca implementarea este una corecta. De asemenea, intr-o alta clasa, cu o metoda main, se poate testa functionalitatea proiectului astfel:

**public** **static** **void** main(String[] args) {

Simulator simulator = **new** Simulator(1, 10, 1, 6, 5, 25);

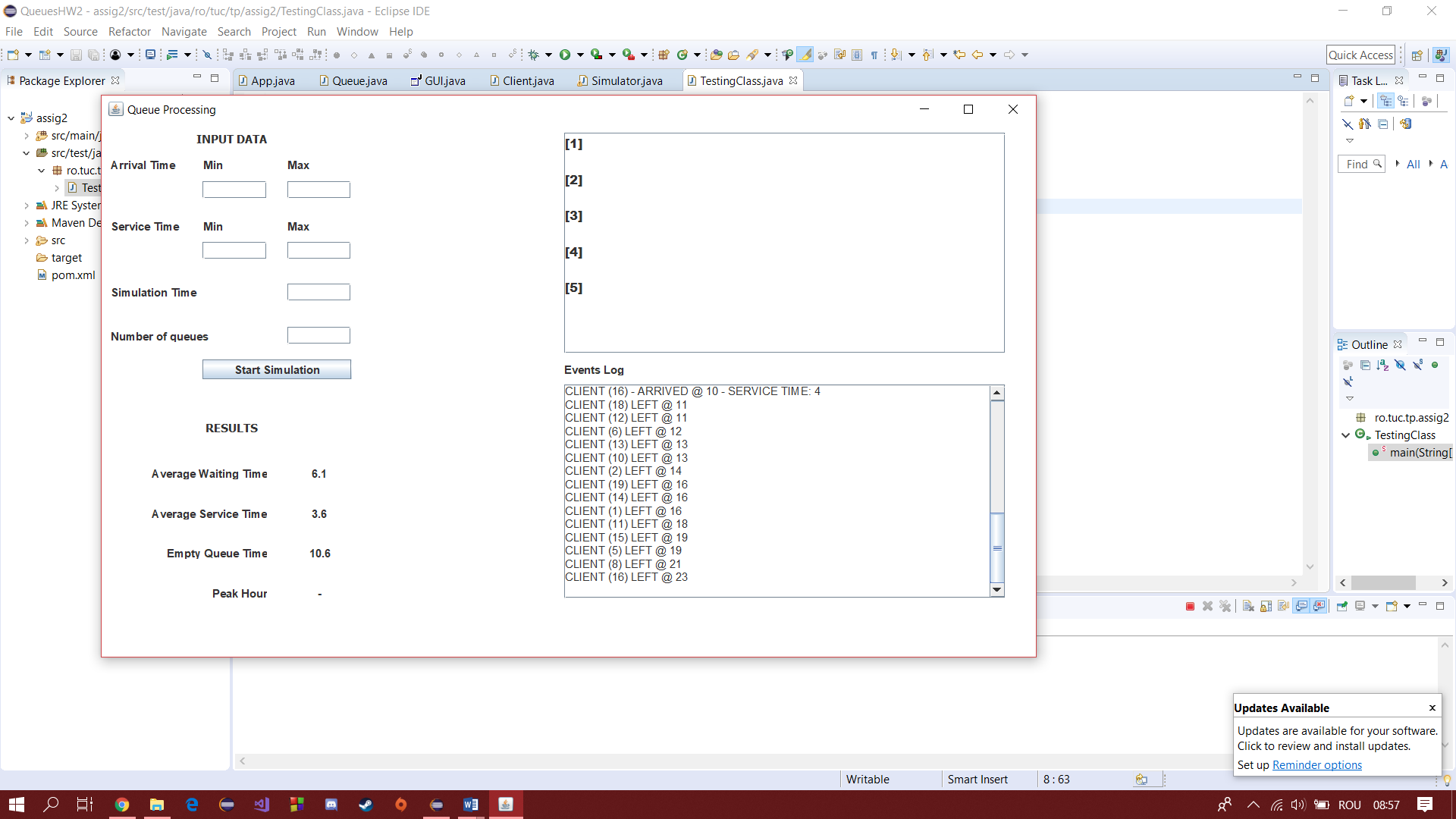
GUI gui = **new** GUI();

gui.setVisible(**true**);

simulator.start();

}

Se creeaza un simulator cu timpul de sosire al clientilor intre 1 si 10, cu timpul de servire intre 1 si 6, cu 6 cozi si cu timpul de simulare de 25 de secunde. Se instantiaza un obiect de tip GUI si se face vizibil, dupa care se porneste thread-ul simulatorului.



Alaturi sunt rezultatele primite in urma celei mai recente rulari.

Concluzii

In aplicatie s-a realizat implementarea unui sistem bazat pe cozi, un sistem care ar putea fi foarte inovativ in viata de zi cu zi, deoarece reduce semnificativ timpul de asteptare la diferitele cozi ce necesita un timp de asteptare crescut.

Bibliografie

<https://www.lucidchart.com/>

<https://stackoverflow.com/>

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