

**Simulator de cozi**

**Documentatie**

**Facultatea: Automatica si Calculatoare**

**Roman Alexandru | Grupa 30228**

**CUPRINS**

1. **Obiectivul si descrierea proiectului**
2. **Analiza problemei, modelare, scenarii, cazuri de utilizare**
3. **Proiectare**
4. **4.Implementare**
5. **Rezultate**
6. **Concluzii**

**1**. **Obiectivul si descrierea proiectului**

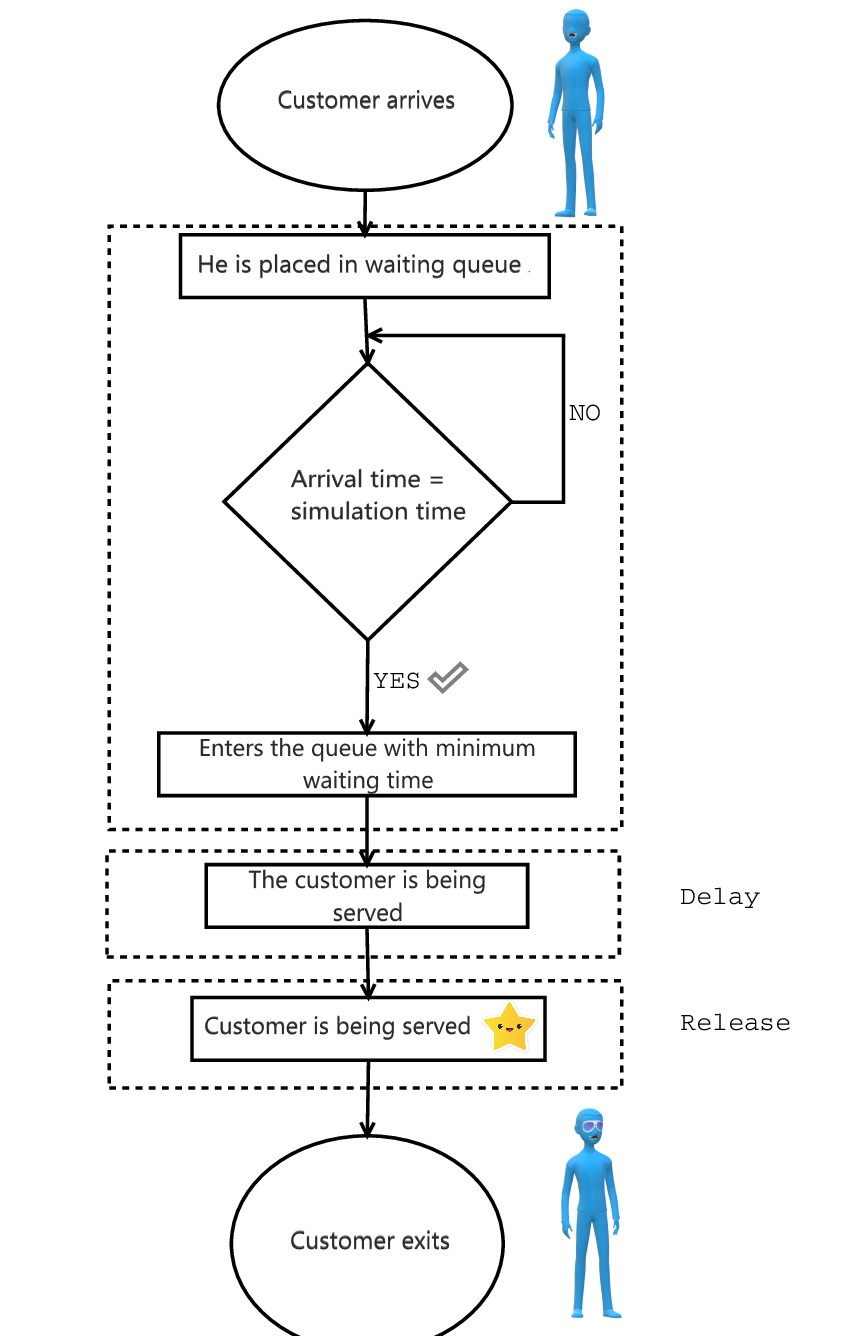
**Teoria statului la coadă** sau **teoria așteptării**  este studiul matematic al liniilor de așteptare, sau [cozi](https://en.wiktionary.org/wiki/queue). Modelul unei așteptări (unui stat la coadă) este construit astfel încât lungimea cozii și timpul de așteptare pot fi prezise. Teoria așteptării este, în general, considerată o ramură a operațiunilor de cercetare, deoarece rezultatele sunt adesea folosite în luarea deciziilor de business în funcție de resursele necesare.

Exista multe cercetari si studii, multe lucrari scrise pe aceasta tema si ele urmaresc atat intelegerea procesului in sine prin impartirea lui in etape si in tipuri de cozi de asteptare, intelegerea perceptiei clientului si a proceselor psihologice implicate, cat si la gasirea celor mai eficiente metode de adresare a cozilor in scopul reducerii timpilor si a eficientizarii interactiunii angajat-client.

Acest proiect proiect, descris in limbajul java a fost creat pentru a minimiza timpul de asteptat la coada de catre clienti la un anumit numar de cozi. Obiectivul acestui sistem care monitorizeaza asezarea la cozi a clientilor este de a pozitiona clientii in fuctie de timpul de asteptare si timpul de servire astfel incat sa se proceseze un numat cat mai mare de clienti intr-un interval cat mai mic de timp si cozile sa functioneze la capacitate maxima.

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

Mai jos, putem vedea diagrama use case a sistemului care ne arata stagiile prin care trece fiecare client.



Proiectul are implementat in Clasa client ( despre care vom vorbi putin mai tarziu) un generator de clienti. Acesta genereaza o lista de clienti care va fi considerata un waiting queue. Astfel putem zice ca atunci cand ajung clientii sunt plasati intr-un waiting queue. Simularea acestor cozi dureaza un anumit timp si incepe de la 0. In momentul in care timpul simulariii ajunge la o valoare, de exemplu secunda 5, lista de clienti( waiting queue este verificata, iar clientii care au arrival time egal cu simulation time, adica care au ajuns in acel moment de timp, sunt prealuati si asezati la coada cu waiting time cel mai mic. Cozile (dupa cum voi explica mai tarziu cand vom vorbii despre implementarea si structura oop a aplicatiei) au un waiting time care semnifica timpul care ar trebuii sa astepte un client nou, pus la coada pentru a fii servit. Deci cand clientul este preluat este plasat optim astfel incat acesta sa astepte cat mai putin posibil. Acesta asteapta timpul necesar la coada iar apoi este servit pentru un anumit timp. Cand acesta este gata acesta iese din coada, urmatorul client vine sa fie servit si asa mai departe.

**3. Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator)**

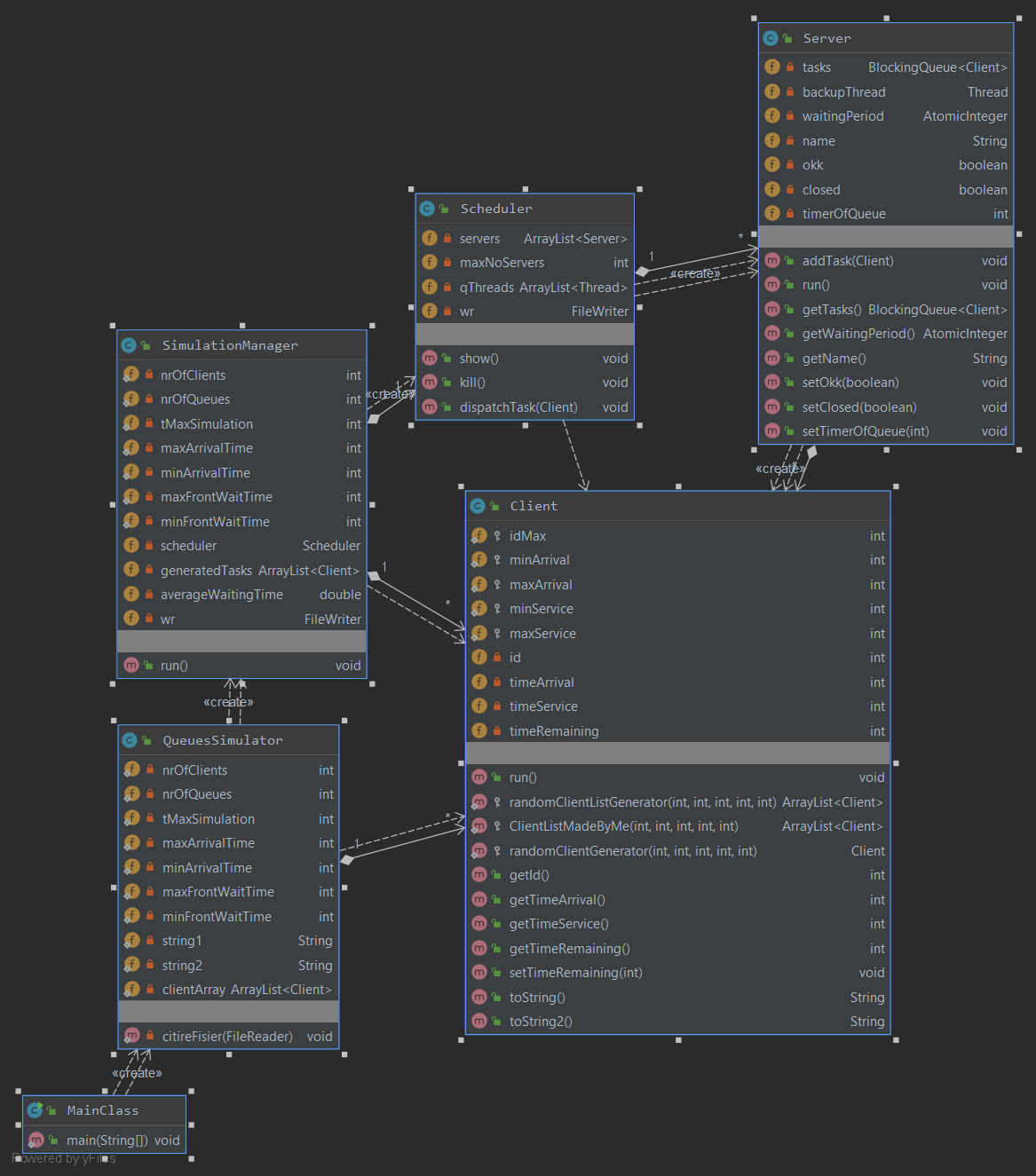
In proiectarea acestui simulator de cozi am decis sa il structurez in 6 clase: MainClass in care avem metoda main(), QueuesSimulator, SimulationManager, Scheduler, Client, Server. Toate aceste vor fii explicate in detaliu mai jos cand vom vorbii de implementare.

Ca si structuri de date am folosit ArrayList uri pentru a stoca atat clientii, cat si un ArrayList in care vor fii stocate serverele . Serverele sunt instantele runnable din clasa Server si se ocupa de procesarea clientilor. Fiecare server este controlat de cate un thread si are un BlockingQueue propriu pentru clienti.

Toate clasele au cel putin o relatie de dependenta cu o alta clasa: Intre MainClass si QueuesSimulator de exemplu care face legatura intre simulare si fisierele in care scriem / citim imformatii. QueuesSimulator la randul ei instantiaza clasa SimulationManager. Toate clasele au un rol important in proiect avend dependinde una fata de cealalta. Relatii de mostenire nu sunt intre cele 6 clase. Clasele SimulationManager si Server implementeaza clasa Runnable deoarece sunt rulate de catre threaduri si au nevoie de metoda run(), metoda implementata din clasa Runnable. De asemenea putem observa 5 relatii de compozitie in diagrama UML. De exemplu clasa Client este parte a claselor QueuesSimulator, SimulationManager si de asemenea a clasei Server. Acestea nu pot exista fara clasa client, asa cum, (alt exemplu) clasa Scheduler nu poate functiona fara clasa Server.

Proiectul nu are o interfata utilizator. Acesta afiseaza atat in consola cat si in fisiere de output derularea simularii si arata serverele impreuna cu cozile de clienti si starea lor, in fiecare secunda a simularii.

De asemenea, am creat un executable jar file care poate fii rulat din linia de comanda ( Command Prompt ) impreuna cu argumentele sale, care sunt 2 fisiere .txt si anume fisierul de intrare si cel de iesire. El a fost creat conform cerintei, sa ruleze cu aceste 2 argumente desii am vrut initial sa pun un al treilea argument care ar fii fost timpul real in ms care sa fii reprezentat cat dureaza o secunda a simularii. Aceasta valoare am lasat-o pe 1000 deci o secunda in timp real va avea 1000 ms adica tot o secunda in timp simulare pentru a avea o simulare cat mai aproape de realitate.

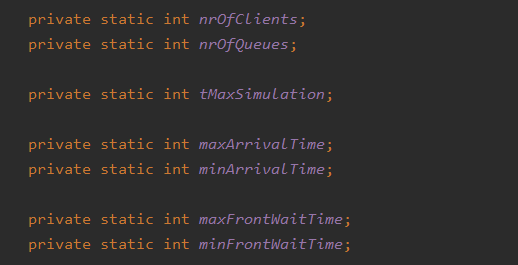


**4. Implementare**

In aceasta sectiune vom vorbii despre cum am implementat clasele simulatorului de cozi precum si campurile si functiile acestora.

Clasa **MainClass:** in aceasta clasa este instantiata clasa QueuesSimulator care se ocupa cu managementul fisierelor si unde este instantiata clasa SimulationManager. Aceasta clasa, din punctul meu de vedere, este singura clasa care poate lipsii, clasa QueuesSimulator putand fii instantiata si intr-un main din propria clasa dar am preferat sa creez o clasa MainClass doar pentru metoda main(). Acest lucru poate fii util deoarece proiectul poate fii dezvoltat in ceva mai complex.

Clasa **QueuesSimulator:** este clasa care se ocupa de interactiunea simularii care se intampla in clasa SimulationManager cu fisierele de intrare si iesire. Aceasta clasa primeste prin constructor fisierele si le deschide. Ea citeste din fisier, caracter cu caracter, cu ajutorul metodei citireFisier() care primeste un fileReader si este apelata din constructor. Metoda modifica variabilele globale citindule valorie din fisier.



In aceasta clasa se instantiaza si clasa SimulationManager.

Clasa **SimulationManager:** Aceasta este clasa care controleaza simularea. Ea implementeaza clasa Runnable iar in clasa QueuesSimulator este instantiata, instanta fiind atrubuita unui thread. Ea are aceleasi variabile globale citite din fisier primite din QueuesSimulator plus o instanta a clasei Scheduler (despre care vom vorbii mai jos), un ArrayList care contine clientii generati, unui FileWriter pentru a scrie in fisier si o variabile averageWaitingTime care face media aritmetica intre timpurile de procesare ale clientilor.

In constructor se foloseste metoda randomClientLiistGenerator() care furnizeaza lista cu clienti si ii ordonam dupa arrival time. Este instantiat Scheduler cu numerul de cozi si executa medota run(). Aici avem un while care se incrementeaza pana la tMaxSimulation iar o iteratie dureaza o secunda. Lista de clienti este parcursa iar cand simulation time este egal cu arrival time al clientului, acesta este trimis catre coada cu cel mai mic timp de astepare cu ajutorul metodei dispatchTask din clasa Scheduler(). Intre timp, threadul simularii doarme o secunda cat timp clientii care sunt in prezent la cozi sunt procesati pana secunda viitoare cand va verifica iar lista de clienti si ii va distribuii urmatoarei liste cu timp minim(intre timp unele cozi se vor goli daca timpul clirntilor procesati s-a scurs si daca se gasesc cozi goale timpul de asteptare va fii 0 iar clientul urmator va fii distrubuit coziii goale).

La sfarsitul simularii metoda Kill() a clasei Scheduler va fii apelata iar threadurile care se ocupa de cozi se vor oprii, iar simularea va lua sfarsit, apoi inchinzandu-se fileWriter ul folosit pentru afisarea decurgerii simularii in fsier. De asemenea se afiseaza average waiting time ul. Daca acesta a fost afisat pe ecran in timpul simularii insemna ca simularea s-a oprit deoarece este in afara while-ului.

Clasa **Client:** aceasta clasa contine caracteristicile unui client : id, timeArrival, timeService si timeRemaining care este egal cu timeService dar scade dupa ce clientul intra in coada iar cand acesta ajunge la 0 clientul iese din aceasta (a fost procesat).Pe urma avem metoda randomClientLiistGenerator() care genereaza un ArrayList cu clienti. Acestia au valori random generati (cu ajutorul clasei Ranom) in conformitate cu parametrii cititi din fisier. Exista de asemenea metoda clientListMadeByMe in care iti poti introduce proprii clienti in caz ca vrei niste velori precise ale clientilor generati. Ea returneaza de asemenea un ArrayList de clienti dar nu este folosita in imlpementare deoarece pe noi ne intereseaza diversitatea clientilor.

Clasa **Server:**  este clasa care descrie comportamentul unuei cozi. Ea implementeaza clasa Runnable. Primeste prin constructor un nume care nu este afisat pe ecran si un flag de tip boolean care atunci cand este setat pe false, atunci coada este inchisa (threadul este inchis sau in starea de wait() ). In metoda run(), implementata din Runnable severifica coada cu metoda peek() din clasa BlockingQueue. Aceasta metoda se uita In coada si returneaza primul client din coada fara a-l scoate. Daca metoda peek() returneaza null atunci coada este goala si serverul intra in starea de sleep pana ce va primi notify din metoda dispatchTask() atunci cand va intra un nou client in coada. Cand primeste notify se “trezeste” si verifica din nou coada si intra in modul sleep o secunda (clientul este procesat o secunda). Dupa o secunda porneste si scade waiting time-ul cozii cu o secunda, precum si remaining time-ul clientului. Daca acesta este 0 clientul este eliminat din coada(a fost procesat) si preluat urmatorul.

Clasa **Scheduler:** Aceasta clasa este responsabila pentru management-ul cozilor. Cand se apeleaza constructorul ei din SimulationManager aceasta creaza nrOfQueues cozi si tot atatea threaduri iar apoi le porneste. Ea are 2 metoda cruciale penntru functionarea simulatorului de cozi precum si una de afisare a cozilor. Una dintre metoda este una despre care am mai mentionat si anume DispatchTask() care distribuie clientii cozilor cu timp de asteptare cel mai mic. Aceasta apeleaza si functia notify pentru a notifica threadurile ca a venit un client nou in coada in caz ca acestea sun in modul wait(). O alta metoda este metoda kill() care notifica threadurile in modul wait() si apoi seteaza flagul de rulare pe false astfel acestea oprindu-se.

Metoda show() a clasei este folosite pentru afisarea atat in consola cat si in fisier a starii cozilor.

**5. Rezultate**

Simulatorul de cozi implementat de mine se foloseste de 2 fisiere si anume fisierul pentru a citii parametrii respectiv pentru a afisa rezultatele simularii. Datorita jar-ului executabil, prograul se poate desemenea rula din linia de comanda.

Trei fisiere au fost rulate de acest program si anume: in-test-1.txt, in-test-2.txt, in-test-3.txt si se pot vedea solutiile iesite in fisierele: out-test-1.txt, out-test-2.txt, out-test-3.txt . De asemenea mai sunt inca 2 doar ca rulate din linia de comanda si anume: in1.txt si in2.txt respectiv out1.txt, out2.txt.

Instructiunea pentru a rula in linia de comanda :

Java -jar PT2020\_30228\_Roman\_Alexandru\_Assignment\_2.jar “fisier de intrare .txt” “ fisier de iesire .txt”

**6. Concluzii si posibilitati de dezvoltare ulterioara**

Ca o concluzie pentru aceasta documentatie a proiectului ar fii faptul ca in lumea actuala, cozile sunt folosite intr-o multitudine de domenii iar un simulator de cozi poate avea posibilitati infinite de dezvoltare ulterioara. Pe cat trece timpul si industria it evolueaza si se perfectioneaza, si celelalte industrii evolueaza odata cu ea, iar tot mai multe devin automatizate. Programul de simulare a cozilor poate fii dezvoltat intr-o sumedenie de lucruri precum un program care distribuie clientiii in liniile telefonice ale unei companii. O companie in care lucreaza n oameni care raspund la apeluri pentru a oferii servicii clientilor de exemplu. Cand clientii suna sunt selectati de catre un algoritm si pusi la coada la care asteapta cei mai putini clienti. Sau de exemplu iintr-un joc cu mai multe servere care este foarte jucat. Un algoritm asemenator poate redirectiona playerii catre serverele mai libere, eceptia fiind ca jucatorii(task-urile) vor iesii singure din servere iar serverele vor avea un numar de n playeri pe care ii poate sustine.

Din aceasta tema am folosit cum sa operez folosing threaduri deoarece a fost prima data cand am lucrat cu ele. Am invatat cum functioneaza si mi-a oferit o idee despre lucrurile ce pot fii dezvoltate folosind multithreading. De asemenea mi-am aprofundat si cunostintele in java in general si in programarea orientat pe obiecte.