MINISTERUL EDUCAȚIEI ȘI CERCETĂRII ȘTIINȚIFICE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
CATEDRA CALCULATOARE

Queue Simulator

Documentație

Dimitriu David

Grupa 30228 | An II semestrul 2

Cuprins

1. Obiectivul temei:

Obiectivul temei a fost sa proiectam in Java un simulator de cozi. Acest simulator primeste ca date de intrare numarul de cozi disponibile, numarul de clienti care vor urma sa fie serviti, si timpul maxim pentru care era deschis “magazinul”. In plus, pentru clienti se specifica timpul maxim si minim la care acestia isi termina cumparaturile, si intervalul de timp (prin minim si maxim) pe care trebuie sa il astepte la casa pentru a le fi scanate produsele. Proiectul trebuie sa genereze pornind de la aceste date de intrare o colectie de clienti cu date aleatorii care sa respecte intervalele de timp. Cerinta presupune de asemenea si inchiderea sau deschiderea automata a cozilor, astfel incat, la inceput toate cozile sa fie inchise, urmand sa fie deschise cand primul client este gata sa fie procesat de acea coada, iar apoi inchise iar cand nu mai sunt clienti care asteapta. Trebuie prevazut un “organizator de cozi” care se eficientizeze sistemul, astfel incat fiecare client sa fie directionat spre coada cu cel mai mic timp de asteptare. Simularea va fi afisata pe al doilea view intr-un text field, unde se vor afisa pentru fiecare perioada de timp (de cate o secunda) clientii care fac cumparaturile si clientii care si-au terminat cumparaturile, distribuiti la cozile aferente. Tot in al doilea view am afisat timpul mediu de asteptare. De asemenea, datele de intrare se citesc dintr-un fisier de intrare. Trebuie ca aplicatia sa fie implementata pe thread-uri (fire de executie), asfel incat fiecare coada are un thread asociat si functioneaza independent de celelalte.

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

Modelarea problemei se face în mare parte conform exemplului atașat prezentării temei, astfel încât cozile sunt modelate ca servere care primesc sarcinile pe care trebuie să le proceseze (clienții). Serverele sunt monitorizate și primesc sarcini de la scheduler, care consultă fiecare server cu privire la timpul de așteptare și decide ce coadă să aloce următorului client pentru eficiență.
Interactiunea user-ului cu aplicatia se va face prin intermediul celor doua View-uri care reprezinta partea de front a aplicatiei.In primul view user-ul va introduce datele de simulare,va configura simularea cozilor.Un exemplu de imput ar fi : 2, 30 pentru ArrivalTime(min,max), 6 pentru numarul de client, 2 pentru numarul de cozi, 50 de secunde intervalul de simulare si 2, 7 ServiceTime(min,max).Trebuie luat in considerare ca intervalele trebuie date ca despartite de virgula si spatiu.Folosind regex am preluat

din text field intervalul,atribuind minimului,respectiv maximului valorile dorite.User-ul poate verifica datele introduce apasand pe butonul Validate.Apoi va apasa pe butonul de start si va incepe simularea.Va aparea al doilea view,in care vor fi afisate in timp real cozile si clientii cu arrival time si service time-ul fiecaruia.

3. Proiectare (decizii de proiectare, diagrame UML, structure de dare, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator).

Pentru proiectarea m-am ghidat dupa structura data in prezentare.Am proiectat programul in 3 pachete,view,model si bussinesLogic. Am folosit de asemenea structura mvc. Pachetul view include clasele View si outPutView care sunt responsabile cu interactiunea cu user-ul programului. In pachetul models avem clasele Task si Server reprezentand clientii si cozile.Iar in pachetul bussinesLogic avem clasele : Controller care uneste view-urile cu logica programului,Scheduler, clasa enum SelectionPolicy, SimulationManager care se ocupa de logica simularii si SortTask care are rolul de a sorta task-urile dupa arrivalTime.

Clasa *Task* reprezinta un client, ce are ca atribute: “id” (unic, incepand de la 0 la numarul de clienti), “arrivingTime” care reprezinta timpul la care clientul isi termina de facut cumparaturile si ajunge la casa, fiind pregatit sa fie procesat, “serviceTime” care reprezinta perioada de timp necesara pentru procesarea sau scanarea produselor. Ca metode se gasesc: get si set pentru fiecare argument, constructorul clasei si metoda toString care are ca rol printarea unui task impreuna cu atributele lui.

Clasa *Server*, care reprezinta o coada, locul in care sunt prelucrati clientii (Task-urile) si are ca argumente: “tasks”, care este de timp BlockingQueue<Task> si care reprezinta coada cu clientii aflati la casa, care asteapta sa fie procesati, “waitingPeriod”, care repprezinta timpul de asteptare pentru ultimul client aflat la coada pana cand acesta este terminat de procesat, “open” care este de tip Boolean si reprezinta starea serverului, adica daca e deschis sau inchis, “id” care reprezinta numarul serverului si e unic. De asemenea, si aici se gaseste si “totalWaitingTime” de tip AtomicInteger. Ca metode se gasesc: get, set (pentru toate atributele), constructor cu parametri, addTask, toString si run (deoarece aceasta clasa trebuie sa implementeze Runnable).

Metoda run din clasa Server :

```
public void run(){
        while (this.open) { //cat timp serverul e deschis,coada
```

```
                while (tasks.peek() != null) {      //cat timp are clienti de procesat
                    try {
                        assert this.tasks.peek() != null;    task.peek()
                        Thread.sleep(1000L*this.tasks.peek().getServiceTime());
                        assert tasks.peek() != null;
                        waitingPeriod.addAndGet(-tasks.peek().getServiceTime());
                        assert tasks.peek() != null;
                        this.tasks.peek().setServiceTime(0);

                    }catch (Exception ignored){}
                }
            }
        }
```

Clasa *Scheduler*, care este responsabila pentru management-ul serverelor, are ca atribute: “servers”, care e o colectie de obiecte,lista, de tip “Server”, (ArrayList<Servers>) , “maxTasksPerServer”, care reprezinta numarul maxim de clienti la o singura coada, “maxNoServers”, care reprezinta numarul maxim de servere, “threads”, care este o colectie de obiecte de tip “Thread” (ArrayList<Thread>), Ca metode se gasest: get si set pentru fiecare argument, constructor cu parametri, , minTimeQueue, dispatchTask, killThreads,.

Clasa *SimulationManager*, implementeaza si metoda main, este simulatorul aplicatiei si are ca argumente in mare datele de pe view,minArrivalTime, maxArrivalTime,nOfClients,nOfQueues,timeLimit,minServiceTime, maxServiceTime,outView reprezentand al doilea view,scheduler de tip Scheduler,o lista de task-uri (List<Task> taskList si variabile de clasa : averageServiceTime,peakH care reprezinta “ora cea mai aglomerata” , maxQueue,averageWaitingTime,initialNoClients, txtForFile care reprezinta o copie la a string-ului cu care setam text field-ul din view-ul destinat afisarii .Avem constructorul clasei in care initializam toate atributele,alegand si policy-ul , adica abordarea problemei, am instantiai scheduler iar mai apoi am apelat metoda de sortarea pe lista de task-uri. Aici avem metoda generateNRandomTasks() care are ca rol de a genera nOfClienti(nr de client dat ca input) cu un arrivalTime si serviceTime random. In aceasta metoda apelam si metodele randomizedServiceTime() si randomizedArrivalTime() care ajuta la generare. Mai continue metoda run() in care

setam thread-urile si practice logica din spatele simularii,generam string-ul care urmeaza sa fie afisat pe view si calculam peakH, averageWaitingTime si averageServiceTime.

Metoda run() din clasa SimulationManager :

```
@Override
public void run() {
	int currentTime = 0;

	while (currentTime < timeLimit) {
		StringBuilder txt = new StringBuilder();
		int ctTime = currentTime;

		for (int i = 0; i < nOfClients; i++) {
			if(taskList.get(i).getArrivalTime() == ctTime) {

				try {
					scheduler.dispatchTask(taskList.get(i));
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
				taskList.remove(taskList.get(i));
				nOfClients--;
				i--;
			}
		}

		txt.append("Time:").append(currentTime).append("\n").append("Waiting clients: ").append(taskList).append(", ");

		for(Server server : scheduler.getServers()){
			if(!server.getTasks().isEmpty()) {
				assert server.getTasks().peek() != null;
				if (server.getTasks().peek().getServiceTime() == 0)
					server.getTasks().remove();
			}
		}

		nrOfClients = 0;
		for (Server server : scheduler.getServers()) {
```

```
            txt.append("\nQueue ").append(server.getId()).append(": ");
            nrOfClients += server.getTasks().size();
            if (server.getTasks().isEmpty()) {
                txt.append("closed\n");
            } else{
                // BlockingQueue<Task> tasks = server.getTasks();
                for (Task task : server.getTasks()) {
                    txt.append(task.toString()).append(" ");
                }
                txt.append("\n");

            }
        }

        if (nrOfClients > maxQueue) {
            maxQueue = nrOfClients;
            peakH = currentTime;
        }

        this.outView.setDisplayTextArea(txt);
        txtForFile = String.valueOf(txt);
        currentTime++;



        try {
            Thread.sleep(1000L);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    for(Server server : scheduler.getServers()){
        averageWaitingTime += server.getTotalWaitingTime();
    }

    scheduler.killTreads();

    outView.setAvgServiceTTxt(averageServiceTime);
    outView.setPeakHTxt(peakH);
    outView.setAvgWaitingTTxt((double) averageWaitingTime / initialNoClients);
```

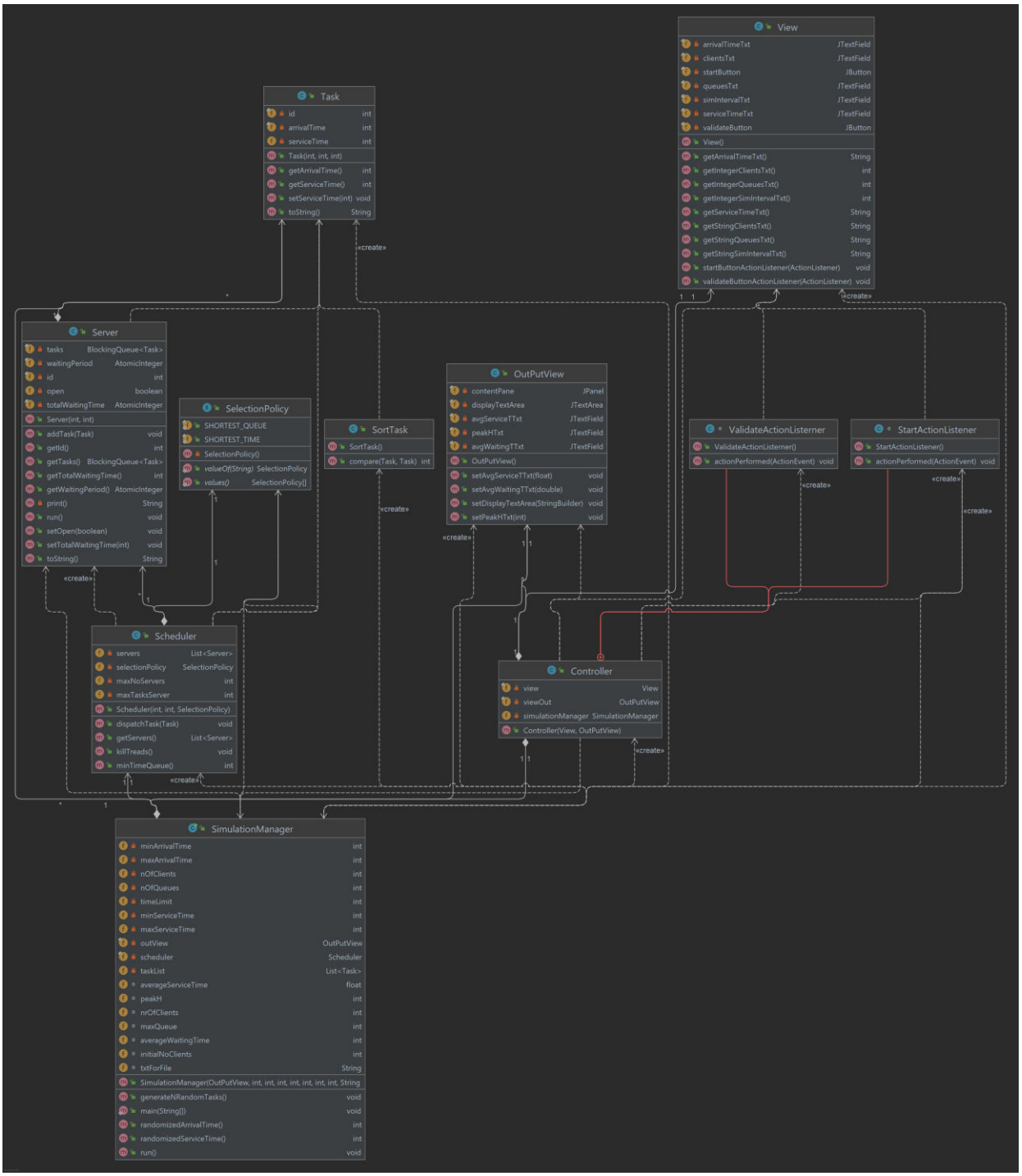
```
        // https://www.w3schools.com/java/java_files_create.asp de ajutor
        File myFile = new File("output3.txt");
        try {
            if(myFile.createNewFile()){
                //System.out.println("File created : " + myFile.getName() + "in " +
myFile.getAbsolutePath());
                FileWriter myWriter = new FileWriter("output.txt");
                myWriter.write(String.valueOf(txtForFile));
                myWriter.close();
                //System.out.println("Successfully wrote in the file");
            }else{
                //System.out.println("File already exists.");
            }
        } catch (IOException e) {
            System.out.println("Error!");
            e.printStackTrace();
        }

}
```

Clasa *SortTask* foloseste doar la sortarea clientilor in functie de arrivalTime.

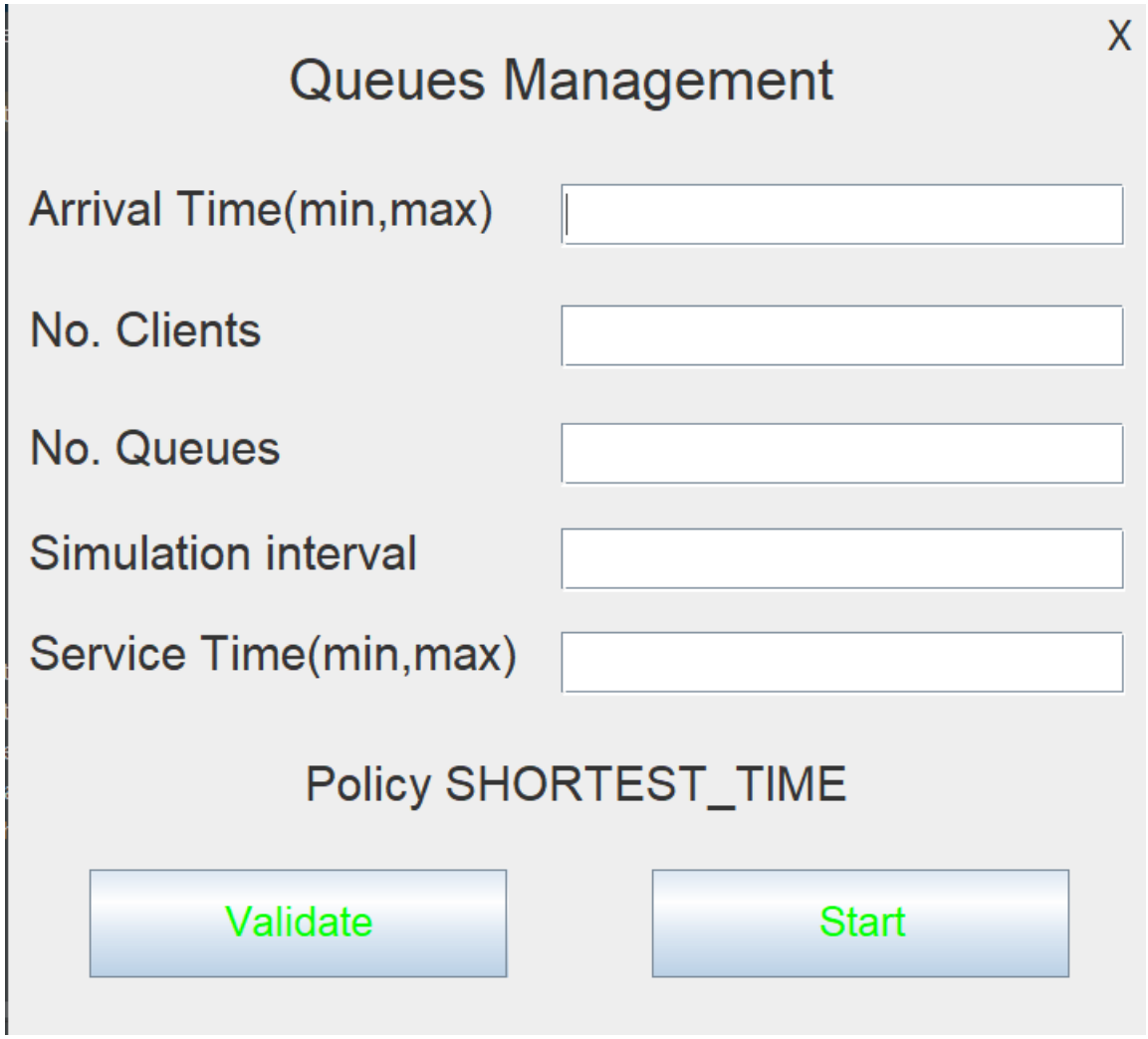
Clasa *Controller* are ca rol unirea front-uli aplicatiei cu back-ul ei,avand atributele view, viewOut si simulationManager.In constructor setam view,urile si cele 2 butoane de pe primul view,Validate si Start prin metoda validateButtonActionListener respectiv startButtonActionListener. Tot in aceasta clasa avem 2 inner clase in care este logica din spatele butoanelor.

Diagrama UML:

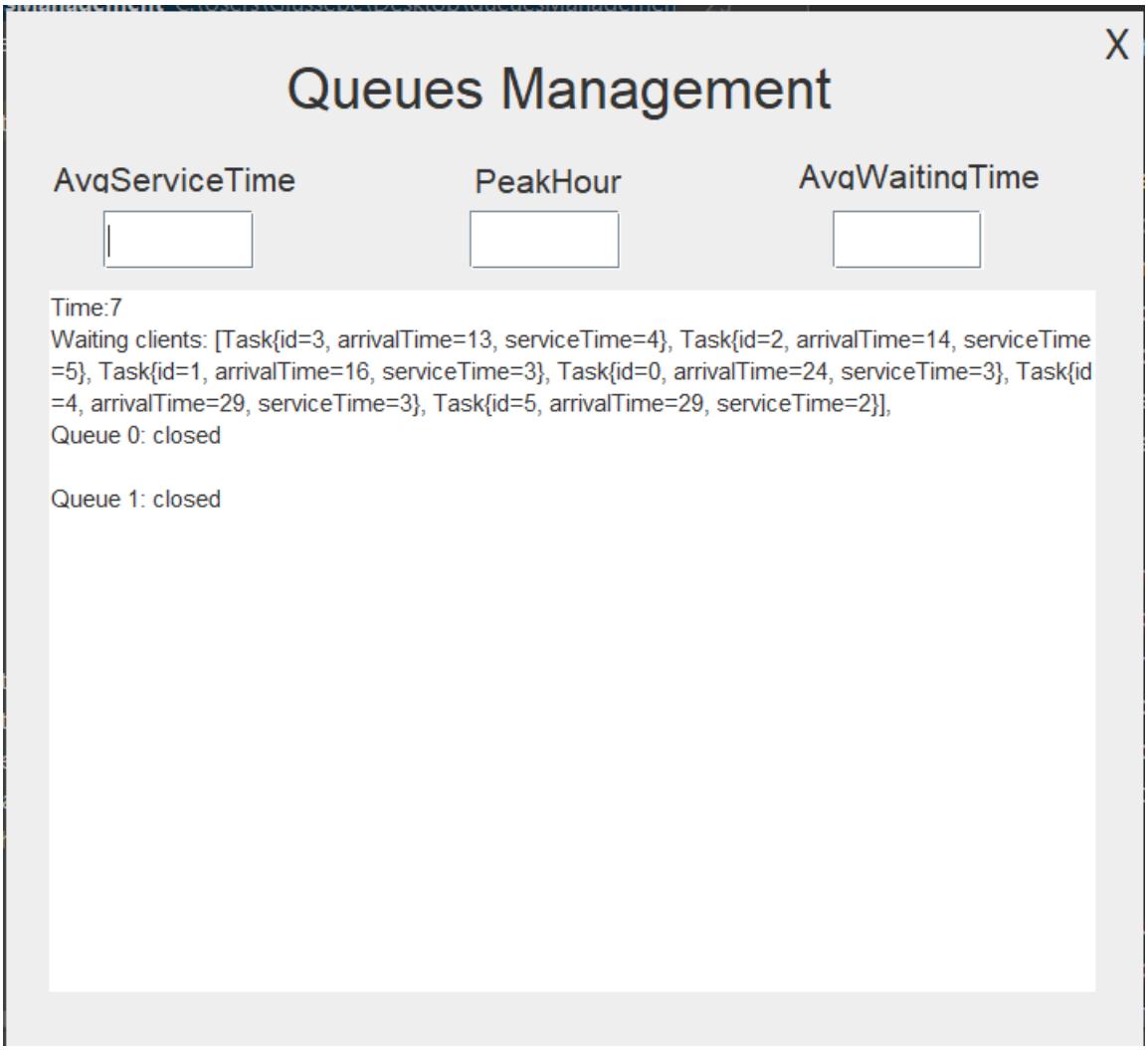


Interfata grafica:

Interfata pentru a seta datele de simulare :



Interfata pentru a afisa simularea :



Algoritmi folositi:

Partea de algorimtica din acest proiect consta in parcurgerea si procesarea Task-urilor pana cand acestea nu mai sunt. Pentru asta, au fost folosite bucle de tip while cu multe conditii pentru a asigura functionalitatea dorita.

4. Implementare:

Metode:

1. Clasa Task:

- toString() pentru a usura afisarea proprietatilor task-urilor.

2. Clasa Server:

- addTask(Task) in care adauga un nou Task la Server, marind timpul de asteptare al cozii.
- toString() ajuta doar la afisarea Task-urilor dintr-un server.
- run() descrie activitatea unui server cand este pornit: cat timp este deschis si mai are clienti de procesat, adoarme thread-ul asociat cu el pentru o secunda, scade timpul total de procesare si timpul de procesare a clientului care este procesat, iar apoi verifica daca a terminat cu el. Daca da, atunci il scoate din lista de asteptare, altfel ciclul se repeta. Dupa ce nu mai are clienti de procesat, serverul isi seteaza starea ca fiind inchis.

3. Clasa Scheduler:

- minTimeQueue() caluleaza si cel mai mic timp de asteptare prin parcurgerea listei de servere.
- dispatchTask(Task) foloseste metoda minTimeQueue pentru a asigna task-ul serverului cu cea mai scurta coada, punand in coada serverului taskul.
- killThreads() parcurge lista de servere si seteaza starea acestora pe fals (adica incis) apeland pentru fiecare metoda setOpen() din server

4. Clasa SimulationManager:

- randomizedArrivingTime() returneaza un numar aleator intre minArrivalTime si maxArrivalTime
- randomizedProcessingTime() returneaza un numar aleator intre minProcessingTime si maxProcessingTime
- generateNRandomTasks() genereaza numberOfClients task-uri aleatorii folosind metodele randomizedProcessingTime si randomizedArrivingTime, si le adauga in lista generatedTasks, apoi le sorteaza dupa arrivalTime
- run() pentru fiecare task din cele generate aleator, apeleaza metoda dispatchTask si o asigneaza unui server, dupa care o scoate din lista de taskuri. Pentru fiecare unitate de timp scrie in fisier clientii care inca nu si-au terminat cumparaturile si clientii care asteapta la fiecare coada, scazand timpul de procesare a primului client din fiecare coada cu 1. Dupa ce termina cu toti clientii apeleaza killThreads si seteaza starea tuturor serverelor ca fiind inchise si scrie in fisier timpul mediu de asteptare. Tot aici am creat un string de tipul StringBuilder in care am tot concatenate rezultatele si am si creat sun fisier si am scris in el rezultatele.

- main() am create un obiect de tipul View si outPutView si un Controller caruia I le-am pasat.

5. Clasa Controller

- Class ValidateActionListener in care este logica din spatele butonului de validate.Am folosit regex si un format de interval facut “[0-9]+(, [0-9]+)+” cu care am verificat campurile introduce de user in view,cele ca interval. Pentru cele de tip number am folosit regex-ul “[0-9]”.Am verificat fiecare camp si in cazul in care nu era introdus correct am afisat un pop-up in care notificam user-ul cum sa introduca input-ul.

  -Class StartActionListener in care am facut logica pentru butonul de START.Aici ne-am create un obiect de tipul SimulationManager caruia i-am pasat datele de pe view si am creat un thread si l-am pornit.

6. Clasa Sortare

- compare(Task, Task) returneaza diferenta dintre arrivalTime pentru cele 2 task-uri

5. Rezultate.

Pentru testarea proiectului am facut mai multe teste, cu datele specificate in cerinta. Pentru primele 2 fisiere, rezultatele au fost corecte, aplicatia a functionat corect si a reusit sa termine cu succes toate task-urile generate aleator.

Am intampinat multe erori de-a lungul realizarii acestui proiect.Am intampinat si comportament nedorit, cum ar fi: unul dintre servere mergea bine iar clelelalte nu, sau se blocau toate serverele cand mai aveau doar un client si nu mai procesau acel client. De asemenea, am intampinat si problema rularii la infinit, programul ajungand intr-o stare in care nu se mai oprea si nu mai crestea timpul curent deloc, ramanand setata la 0. Cu toate acestea, intr-un final am reusit sa obtin rezultatele dorite, mai putin pentru timpul mediu de asteptare, care ramanea 0 tot timpul:

6. Concluzii:

Acest proiect a necesitat o documentare foarte detaliată și îndelungată, deoarece conceptul de threading era complet nou pentru mine și a trebuit să caut surse externe de informații despre acest concept. Până la urmă, am reușit să înțeleg ideea într-o oarecare măsură și am reușit să o implementez în proiect. Odată cu dezvoltarea ulterioară, algoritmul poate fi implementat în viața reală pentru gestionarea cozilor din supermarketuri, aeroporturi sau alte locuri aglomerate, prin niște porți automate, redirecționând clienții către coada cu cel

mai scurt timp de așteptare. S-ar putea implementa chiar si o metoda de calcul a timpului de procesare pentru fiecare client prin image recognition pe culoarele de dinainte de casele de marcat.

7. Bibliografie:

- https://dsrl.eu/courses/pt/materials/PT2021-2022_Assignment_2.pdf
- https://dsrl.eu/courses/pt/materials/A2_Support_Presentation.pdf
- https://www.w3schools.com/java/java_files_create.asp //pentru fisiere
- https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
- https://www.tutorialspoint.com/java/util/timer_schedule_period.htm

-https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html // thread-uri