**FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE**

**CATEDRA CALCULATOARE**

Simulare cozi

Documentație

Ilies Oana-Elena

Grupa 30222 | An 2 semestrul 2

Cuprins

1. Obiectiv

2. Studiul probelemei

3.Implementare

3.1 Diagrame UML

3.2 Clase

3.3 GUI

3.4 Metode

4.Concluzii

5.Dezvoltari ulterioare

6.Bibliografie

1.Obiectiv

Obiectivul temei de laborator a fost de a proiecta și implementa o aplicație care să simuleze cozi de clienți, luând în considerare mai multe aspecte care pot duce la o fluidizare mai bună a acestora. Această aplicație simulează sosirea unei serii de clienți la un magazin, ținând cont de plasarea acestora în cozi și monitorizând timpul pe care aceștia îl petrec la coadă și timpul pe care îl petrec fiecare în parte fiind servit. Pentru a calcula timpul de așteptare, trebuie să se cunoască ora de sosire a clientului și timpul de servire, care sunt generați aleatoriu pentru fiecare client în parte, astfel încât fiecare client să fie diferit.

2.Studiul problemei

Cozile reprezintă un concept fundamental în domeniul modelării sistemelor din lumea reală, fiind utilizate într-o gamă largă de domenii, precum transportul public, servicii medicale sau comerțul cu amănuntul. Scopul principal al unei cozi este de a oferi un spațiu în care clienții pot aștepta înainte de a fi serviți. Sistemele de management al cozilor se concentrează în principal pe minimizarea timpului pe care clienții îl petrec în așteptarea în cozi înainte de a fi serviți.

Orice coadă poate fi văzută ca o pereche între casa de servire și clientul care așteaptă la coadă. Această corespondență poate fi modelată în diferite moduri, în funcție de specificul problemei. De exemplu, fiecare coadă poate avea un set de clienți care trebuie procesați, iar clientul poate alege care dintre casele de servire disponibile îi convine cel mai bine, în funcție de numărul de clienți existenți în coadă și de timpul de așteptare estimat.

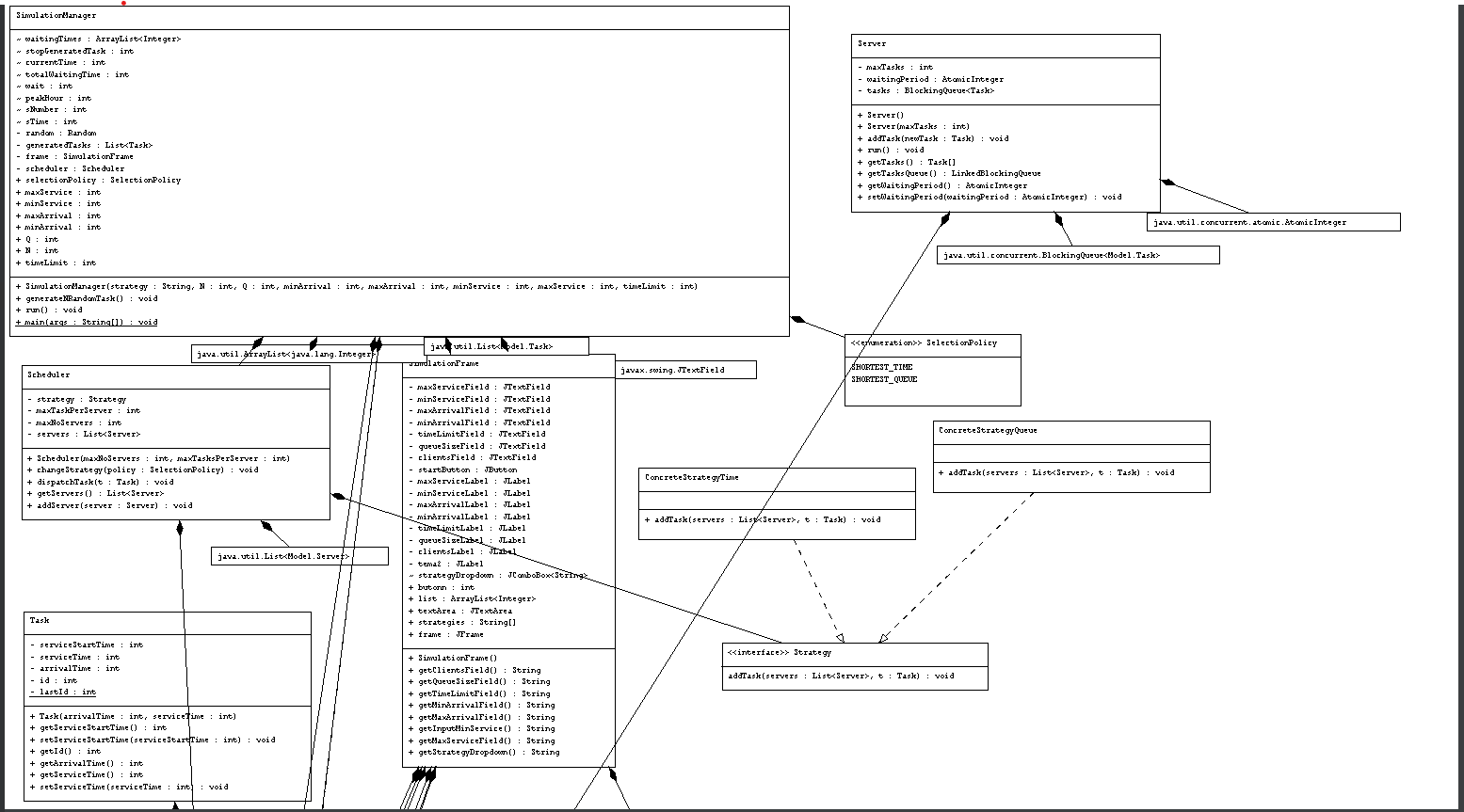
Proiectul curent are ca obiectiv simularea unei serii de clienți care sosesc la cozi într-un interval de timp dat. După sosirea lor, clienții se vor așeza la coada cu cel mai mic număr de clienți și vor aștepta să le vină rândul pentru a fi serviți. Pentru a realiza acest lucru, trebuie să cunoaștem momentul sosirii fiecărui client la casă și timpul de servire de care are nevoie. Mai mult decât atât, trebuie să estimăm timpul pe care fiecare client îl va petrece în coadă până când va ajunge la vârful acesteia, deoarece fiecare client dorește să aștepte cât mai puțin posibil. Acest timp de așteptare depinde de numărul de clienți existenți în coadă și de timpul de servire necesar pentru fiecare client în momentul în care ajunge în fața cozii. Interfața grafică a aplicației permite utilizatorului să introducă date de intrare, cum ar fi intervalul de sosire și timpul de servire minim și maxim al clienților. Numărul de case disponibile este presetat la trei și nu poate fi ales de utilizator. Detaliile privind procesul de alegere a coziilor și de servire a clienților pot fi vizualizate în partea de jos a interfeței grafice a aplicației.

# 3.Implementare:

3.1 Diagrame UML:

Unified Modeling Language (UML) este un limbaj standardizat pentru descrierea modelelor și specificațiilor de software, utilizat în principal pentru reprezentarea programelor orientate pe obiect. Acest limbaj oferă o modalitate de a modela și descrie structurile, relațiile și comportamentul sistemelor software, utilizând o serie de diagrame și elemente grafice.

Pe lângă utilizarea sa în programarea orientată pe obiect, UML este utilizat și în alte domenii, cum ar fi managementul de proiecte și designul proceselor de afaceri, datorită capacității sale de a reprezenta informații abstracte într-un mod clar și eficient. UML poate fi utilizat pentru a modela diverse aspecte ale unui sistem, inclusiv structura, comportamentul și interacțiunile dintre componente. În plus, UML poate fi folosit ca un limbaj comun pentru a facilita comunicarea și colaborarea între membrii echipei de dezvoltare software și cu clienții sau utilizatorii sistemului.



3.2 Clase

Pentru implementare am folosit aceste clase:

In pachetul **Model** am implementat:

A.Clasa Server:

->este folosită pentru a modela un server care poate procesa sarcini primite într-o coadă. Clasa are trei câmpuri private:

"tasks": o coadă blocantă de sarcini care așteaptă să fie procesate de server;

"waitingPeriod": un AtomicInteger care reprezintă perioada de așteptare (în milisecunde) a unei sarcini înainte de a fi procesată de server;

"maxTasks": un int care reprezintă numărul maxim de sarcini pe care serverul le poate procesa simultan

B.Clasa Task:

->are patru câmpuri private și un constructor public. Câmpurile clasei includ:

"lastId": un câmp static de tip int folosit pentru a genera un identificator unic pentru fiecare obiect de tip Task creat;

"id": un câmp de tip int care reprezintă identificatorul unic al obiectului Task;

"arrivalTime": un câmp de tip int care reprezintă momentul de timp la care obiectul Task a ajuns;

"serviceTime": un câmp de tip int care reprezintă timpul necesar pentru a procesa (a servi) obiectul Task.

Clasa Task conține metode de acces și de modificare pentru fiecare dintre aceste câmpuri, precum și o metodă publică "setServiceStartTime" care setează timpul de început al procesării Task-ului. Această clasă este folosită pentru a modela un Task sau o sarcină care trebuie procesată într-un sistem.

In pachetul **GUI** am implementat interfata

A.SimulationFrame

In pachetul BusinessLogic am implementat

A.ConcreteStrategyQueue

Această clasă reprezintă o strategie specifică pentru adăugarea de sarcini în cozi de servere într-un sistem de simulare a serverelor. Clasa implementează interfața Strategy și are o metodă addTask care primește o listă de obiecte Server și o sarcină Task pe care trebuie să o adauge în coada serverului cu cel mai mic număr de sarcini în așteptare.

B.ConcreteStrategyTime

Aceasta clasă reprezintă o strategie de alegere a serverului în cadrul unui sistem de cozi, în care se dorește minimizarea timpului de așteptare al sarcinilor.

C.Scheduler

Aceasta clasa este numita Scheduler si implementeaza planificarea pentru servere. Ea contine o lista de servere, un numar maxim de servere si un numar maxim de sarcini per server. Clasa utilizeaza si o strategie de selectie pentru a alege serverul disponibil pentru a primi sarcina.

D.SelectionPolicy(enum)

Este un enum ce contine**: SHORTEST\_QUEUE, SHORTEST\_TIME;**

E.SimulationManager

Aceasta clasa este responsabilă pentru gestionarea simularii unui sistem de cozi și pentru rularea acestei simulări. Acesta primește datele necesare (cum ar fi timpul limită, numărul de cozi și limitele de timp de sosire și de serviciu) și generează un număr specificat de sarcini aleatorii cu intervale de sosire și de serviciu aleatorii.

În timpul rulării, clasa gestionează evenimentele care apar, cum ar fi sosirea sarcinilor, eliberarea cozi, actualizarea timpului de așteptare și înregistrarea evenimentelor într-un fișier de jurnal.

De asemenea, clasa calculează și afișează rezultatele finale ale simulării, cum ar fi ora de vârf, timpul mediu de așteptare și timpul mediu de serviciu.

F.Strategy(interface)

Prezinta interfata pentru metodele addTask din clasele corespunzatoare strategiilor

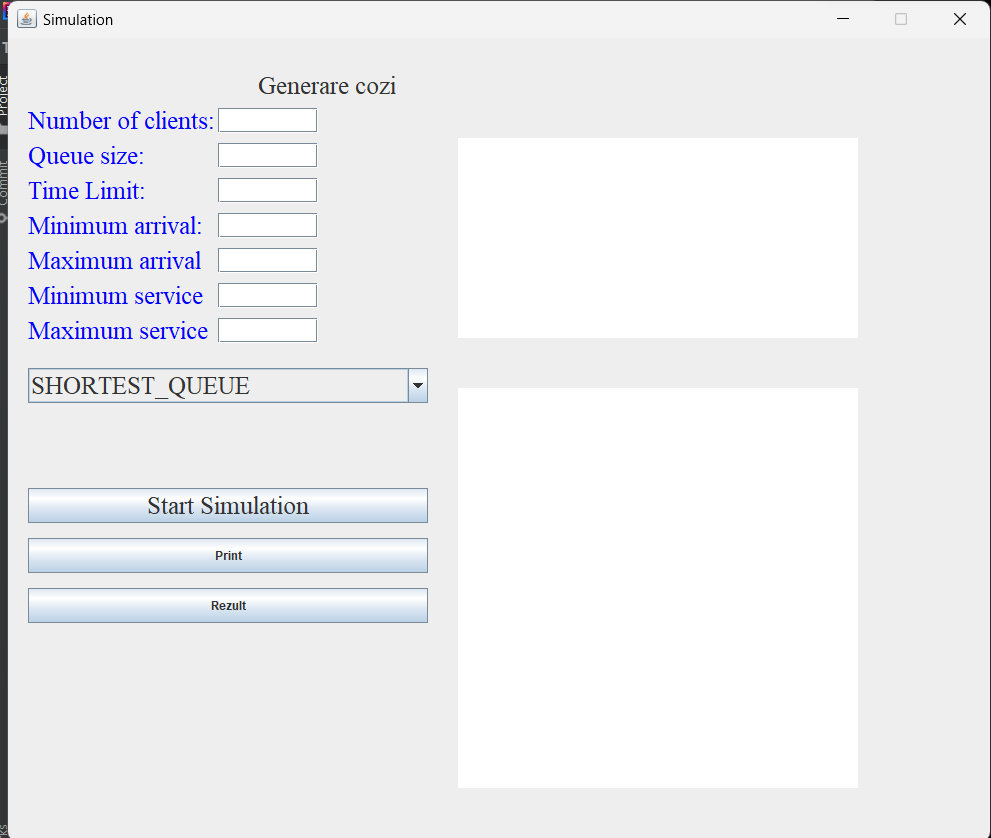
3.3 GUI

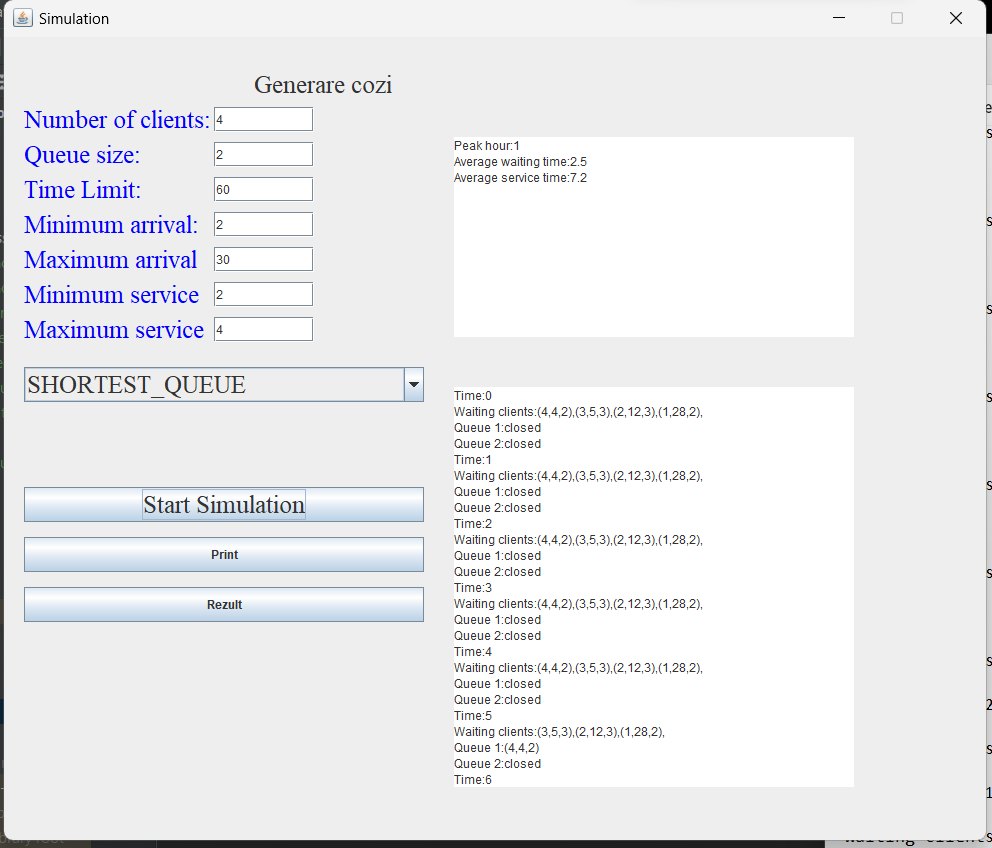
Această clasă, SimulationFrame, reprezintă interfața grafică a utilizatorului pentru simularea unui sistem de cozi. Ea conține elemente grafice, precum etichete, câmpuri de introducere a datelor, dropdown-uri și butoane, care permit utilizatorului să introducă parametrii pentru simulare și să inițieze simularea.

In interfata vor fi introduse datele care ulterior vor fi afisare in 2 fisiere, unul in care se va afisa log of events si unul in care se va afisa: peak hour, average waiting time si average service time.(Rezultatul aflat in fisiere se gaseste in folderul proiectului)

Cuprinde:

1. JFrame frame
2. String[] strategies
3. JTextArea textArea
4. ArrayList<Integer> list
5. int butonn
6. JComboBox<String> strategyDropdown
7. JLabel tema2
8. JLabel clientsLabel
9. JLabel queueSizeLabel
10. JLabel timeLimitLabel
11. JLabel minArrivalLabel
12. JLabel maxArrivalLabel
13. JLabel minServiceLabel
14. JLabel maxServiceLabel
15. JButton startButton
16. JButton prinFromFile
17. JButton rezul
18. JTextField clientsField
19. JTextField queueSizeField
20. JTextField timeLimitField
21. JTextField minArrivalField
22. JTextField maxArrivalField
23. JTextField minServiceField
24. JTextField maxServiceField
25. JTextArea textArea1
26. JTextArea textArea2





In momentul in care este apasat Start Simulation incepe simularea si datele sunt afisare in fisier, butonul print afiseaza datele ce tin de log of events, iar butonul rezult afiseaza rezultatul mediilor si al peak hour.

3.4 Metode

Clasa Server conține și câteva metode publice:

"addTask": această metodă primește o sarcină nouă de tip Task și o adaugă în coada de sarcini. De asemenea, crește valoarea "waitingPeriod" cu 1 și trezește toate thread-urile care așteaptă să ia sarcini din coadă;

"run": această metodă este folosită pentru a procesa sarcinile din coadă. Metoda procesează sarcinile în ordinea în care au fost adăugate în coadă și elimină doar cele care au fost finalizate cu succes (cu durata de procesare 0). Dacă o sarcină nu a fost finalizată cu succes, se așteaptă o perioadă de timp egală cu durata de procesare a sarcinii înainte de a fi procesată din nou;

ConcreteStrategyQueue: „addTask” parcurge lista de servere și calculează dimensiunea cozii de sarcini pentru fiecare server în parte. După ce găsește serverul cu cea mai mică coadă, metoda adaugă sarcina în coada acestui server utilizând metoda addTask din clasa Server. Dacă toate serverele din lista dată au cozi pline, metoda aruncă o excepție de tip IllegalStateException cu mesajul "Server plin".

@Override  
public void addTask(List<Server> servers, Task t) {  
 Server selectedServer = null;  
 int shortestQueueSize = Integer.*MAX\_VALUE*;//cea mai mica coada  
  
 for (Server s : servers) {  
 int queueSize = s.getTasks().length;  
 if (queueSize < shortestQueueSize) {  
 shortestQueueSize = queueSize;  
 selectedServer = s;  
 }  
 }  
  
 if (selectedServer == null) {  
 throw new IllegalStateException("Server plin");  
 }  
  
 selectedServer.addTask(t);  
}

ConcreteStrategyTime: "addTask" care primește o listă de obiecte de tipul "Server" și o sarcină de tipul "Task". Metoda parcurge toate serverele din listă, calculează timpul de așteptare pentru fiecare server și selectează serverul cu cel mai mic timp de așteptare. Dacă nu există server disponibil, se aruncă o excepție "IllegalStateException" cu mesajul "Server plin". După selectarea serverului potrivit, metoda adaugă sarcina în coada serverului.

@Override  
 public void addTask(List<Server> servers, Task t) {  
 Server selectedServer = null;  
 int shortestWaitingPeriod = Integer.*MAX\_VALUE*;//cel mai scurt timp de asteptare  
  
  
 for (Server s : servers) {  
 int waitingPeriod = s.getWaitingPeriod().get();  
 if (waitingPeriod < shortestWaitingPeriod) {  
 shortestWaitingPeriod = waitingPeriod;  
 selectedServer = s;  
 }  
 }  
  
 if (selectedServer == null) {  
 throw new IllegalStateException("Server plin");  
 }  
  
 selectedServer.addTask(t);}}

Scheduler:

Metoda constructorului creaza o lista de servere si le adauga in aceasta lista. Fiecare server are o dimensiune maxima de sarcini pe care le poate executa, care este specificata prin parametrul maxTasksPerServer. In plus, pentru fiecare server, se creeaza un fir de executie, reprezentat de obiectul Server, si se lanseaza.

Metoda „changeStrategy” primeste ca parametru o politica de selectie, reprezentata prin enumeratia SelectionPolicy si schimba strategia folosita de Scheduler.

Metoda „dispatchTask” primeste o sarcina si o trimite la unul dintre serverele disponibile in functie de strategia aleasa.

Metoda ”getServers” returneaza lista de servere. In plus, clasa contine si o metoda addServer, care adauga un server nou in lista de servere.

SimulationManager:

Metoda "generateNRandomTask()" generează un număr de sarcini aleatorii în funcție de numărul N dat de utilizator și a intervalului de sosire și de serviciu specificat. Această metodă creează un obiect "Task" pentru fiecare sarcină generată, apoi le sortează în funcție de momentul sosirii.

public void generateNRandomTask() {  
 for (int i = 0; i < N; i++) {  
 int arrTime = random.nextInt(maxArrival - minArrival + 1) + minArrival;  
 int servTime = random.nextInt(maxService - minService + 1) + minService;  
 System.*out*.println(arrTime + " " + servTime);  
 generatedTasks.add(new Task(arrTime, servTime));  
 }  
 generatedTasks.sort(Comparator.*comparing*(Task::getArrivalTime));  
}

Metoda "run()" reprezintă punctul de intrare în simulare și este rulată ca un fir de execuție separat. Această metodă execută bucla principală a simulării și implementează logica procesării sarcinilor. În timpul simulării, metoda se asigură că sarcinile sunt distribuite serverelor în sistemul de cozi, își menține starea și calculează statistici precum numărul de clienți în coadă la un moment dat, vârful orarului, timpul mediu de așteptare și timpul mediu de servire. În plus, această metodă se ocupă de înregistrarea evenimentelor care au loc în timpul simulării într-un fișier text.

public void run() {try (BufferedWriter logOfEvents = new BufferedWriter(new FileWriter("filename.txt"));  
 BufferedWriter finalRezult = new BufferedWriter(new FileWriter("rezultat.txt"))) {  
 while (currentTime < timeLimit && currentTime != stopGeneratedTask) { logOfEvents.write("Time:" + currentTime + "\n");  
 if (generatedTasks.size() == 1) {Task task = generatedTasks.get(0); int arrivalTime = task.getArrivalTime(); int serviceTime = task.getServiceTime();  
 stopGeneratedTask = arrivalTime + serviceTime + 2;}  
 logOfEvents.write("Waiting clients:");  
 for (int i = 0; i < generatedTasks.size(); i++) {Task t = generatedTasks.get(i);  
 logOfEvents.write("(" + t.getId() + "," + t.getArrivalTime() + "," + t.getServiceTime() + ")" + ",");} logOfEvents.write("\n");  
 for (int i = 0, count = 0; i < scheduler.getServers().size(); i++) { int aux\_i=i+1; logOfEvents.write("Queue " + aux\_i + ":");  
 Task[] tasks = scheduler.getServers().get(i).getTasks();  
 if (scheduler.getServers().get(i).getTasksQueue().size() == 0) {logOfEvents.write("closed ");} count = 0;  
 for (int j = 0; j < tasks.length; j++) {  
 if (tasks[j] != null) { count++;  
 logOfEvents.write("(" + tasks[j].getId() + "," + tasks[j].getArrivalTime() + "," + tasks[j].getServiceTime() + ") ");}}  
 if (peakHour <= count - 1) { peakHour = count;}  
 if (!scheduler.getServers().get(i).getTasksQueue().isEmpty()) {  
 Queue<Task> tasksQueue = scheduler.getServers().get(i).getTasksQueue();  
 if (!tasksQueue.isEmpty()) { Task t = tasksQueue.peek();  
 if (t.getServiceTime() == 1) { tasksQueue.remove(); t.setServiceStartTime(currentTime);  
 for (int j = 0; j < tasks.length; j++) {  
 if (tasks[j] != null) { count++;try { logOfEvents.write("(" + tasks[j].getId() + "," + tasks[j].getArrivalTime() + "," + tasks[j].getServiceTime() + ") ");  
 } catch (IOException exception) { throw new RuntimeException(exception);}  
 totalWaitingTime += (currentTime - tasks[j].getArrivalTime());}}  
 sTime+= totalWaitingTime; sNumber++;} else { t.setServiceTime(t.getServiceTime() - 1);}} }logOfEvents.write("\n"); }  
 for (int i = 0; i < generatedTasks.size(); i++) {Task g = generatedTasks.get(i);  
 if (g.getArrivalTime() == currentTime) {  
 scheduler.dispatchTask(g);generatedTasks.remove(i);i--;}}currentTime++;  
 synchronized (this) {try {wait(1000);} catch (InterruptedException e) {throw new RuntimeException(e);}}}  
 finalRezult.write("Peak hour:"+peakHour+"\n"+"Average waiting time:" + (double) totalWaitingTime / sNumber + "\n" +"Average service time:"+ (double)sTime/sNumber+"\n");  
 } catch (IOException e) {throw new RuntimeException(e);}}  
/\*

4.Concluzii

În concluzie, coziile reprezintă un concept fundamental în modelarea sistemelor din lumea reală și au o gamă largă de aplicații practice, de la transportul public la comerțul cu amănuntul și servicii medicale. Sistemele de management al cozilor se concentrează pe minimizarea timpului pe care clienții îl petrec în așteptare și încearcă să optimizeze procesul de servire a clienților.

Proiectul curent se concentrează pe simularea unei serii de clienți care sosesc la cozi într-un interval de timp dat și sunt serviți în funcție de numărul de clienți existenți în coadă și timpul de servire necesar. Utilizatorul poate introduce date de intrare, iar interfața grafică a aplicației permite vizualizarea procesului de alegere a coziilor și de servire a clienților.

În general, înțelegerea și gestionarea eficientă a coziilor sunt importante în multe aspecte ale vieții noastre de zi cu zi și pot contribui la îmbunătățirea experienței clienților și la creșterea eficienței sistemelor.

Si ca un plus in aceasta implementare am descoperit o noua tehnica de programare, cea a folosiri thread urilor si de asemena folosirea unei noi componente din interfata GUI „textArea” si „JScrollPane”.

5.Dezvlotari ulterioare

In acest punct tot ce as vrea adaug ar fi o cerinta deja specificata in proiect si anume afisarea in interfata live.

6.Bibliografie

1.Gross, D., & Harris, C. M. (2011). Fundamentals of queueing theory (4th ed.). John Wiley & Sons.

2.Kleinrock, L. (1975). Queueing Systems, Volume 1: Theory (1st ed.). John Wiley & Sons.

3. <https://docs.oracle.com/javase/8/docs/api/index.html>

4. <https://docs.oracle.com/javase/tutorial/>

5. <https://stackoverflow.com/questions/tagged/java>

6. <https://www.javacodegeeks.com/>