DOCUMENTAȚIE

TEMA 2

NUME STUDENT: Ghiorghe Sorana
GRUPA: 30225

Cuprins

1. Obiectivul temei

Obiectiv principal

Obiectivul principal al temei este proiectarea și implementarea unui simulator de cozi alaturi de o interfata grafica utilizand fire de executie astfel incat fiecare coada sa fie reprezentata de un fir. Cozile vor primi clienti care sosesc la un timp aleator, acestia fiind amplasati in functie de strategia selectata.

Obiectivele secundare

1.	Determinarea claselor necesare pentru implementarea proiectului
2.	Crearea unei interfețe grafice • Adaugarea unei sectiuni de introdus date • Adăugarea unui panou pentru vizualizare în timp real al cozilor.
3.	Gestiunea interacțiunii dintre interfață si activitatea threadurilor • Determinarea conținutului fiecărei clase pentru a minimiza nevoia de duplicare a metodelor și a campurilor. • Stabilirea nevoilor fiecarei clase pentru a-si efectua rolul principal și crearea de referinte către clasele de care are nevoie în desfășurare.
4.	Sincronizare și structuri de date relevante pentru thread safety • Abordare inițială de printare a conținutului propriu de catre fiecare coada • Abordare ulterioară de iterare a cozilor și afisarea clientului curent. • Adaugarea zonelor de cod synchronized pentru afisarea în ordine cronologică.
5.	Documentațe • Elaborarea unei documentații care să cuprindă detalii despre implementarea aplicației pentru o înțelegere corespunzătoare asupra simulatorului și modul în care funcționează.

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

Problema abordată constă în managementul ineficient al cozilor care poate conduce la un timp de așteptare ridicat pentru clienți și utilizarea necorespunzătoare a resurselor. În acest context, aplicația propune testarea a 2 strategii diferite pentru repartizarea clienților într-un mod cât mai eficient.testarea a 2 strategii diferite pentru repartizarea clientilor intr-un mod cat mai eficient.

Cerințe Funcționale

- Utilizatorul poate introduce informații legate de simulare de la tastatură, incluzând numărul de cozi, numărul de clienți, etc.
- Aplicația trebuie să pornească fire de execuție multiple în paralel cu firul principal care porneste simularea pentru o simulare corectă, care respectă datele de intrare.

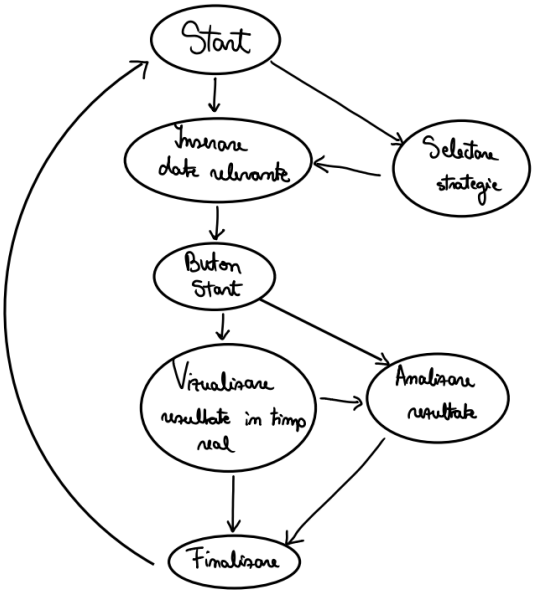
- Firele de execuție trebuie să fie implementate corespunzător și datele trebuie protejate pentru un acces thread-safe.
- Utilizatorul poate urmări progresul cozilor printr-un jurnal de evenimente relevant și concis, care respectă desfășurarea evenimentelor într-un mod cronologic.
- Fiecare coadă servește câte un singur client la orice moment dat, pentru o durată stabilită la generarea clientului.
- Afișarea statisticilor generate în urma rulării când timpul de execuție este îndeplinit.

Cerințe Non-funcționale

- Interfața prin care utilizatorul interacționează cu aplicația trebuie să fie intuitivă și ușor de utilizat.
- Afișarea în timp real a progresului realizat de fiecare coadă și alte informații relevante.
- Utilizatorul trebuie să poată folosi aplicația fără erori sau buguri neașteptate.
- Sistemul de gestionare al thread-urilor trebuie să amplaseze clienții la cozi și în cazurile în care două cozi ar produce un efect identic.
- Programul trebuie să fie pregătit pentru o utilizare necorespunzătoare și să atenționeze în cazul unui input greșit.

Cazuri de utilizare

Aplicația va fi utilizată pentru a simula cozi cu ajutorul thread-urilor, în funcție de criterii specificate de utilizator în timp ce acesta analizează deciziile făcute de simulare. Aplicația poate fi utilizată pentru a observa îmbunătățirile pe care strategiile le aduc din punctul de vedere al timpului de așteptare.



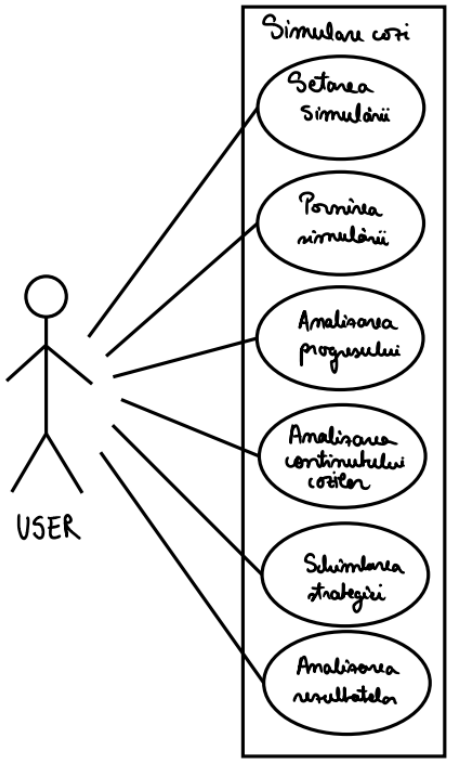
Comportamentul așteptat al aplicației

Scenariu principal de succes:

- Utilizare corectă a simulatorului
  - Actor primar: utilizatorul
  - Utilizatorul introduce date de simulare valide și nenule, care respectă regulile impuse (minim < maxim).
  - Utilizatorul selectează una dintre cele două strategii disponibile: ShortestQueue și ShortestTime.
  - Utilizatorul începe simularea prin apăsarea butonului de start.
  - Aplicația generează clienții și cozile, afișând la interval de o secundă progresul realizat de simulare: conținutul cozilor și clientul care este servit în secunda curentă.
  - După timpul de executie, aplicația afișează informații legate de statistică, precum timpul mediu de așteptare, secunda la care au existat cei mai mulți clienți, etc.

Scenarii alternative:

- Introducerea datelor invalide
  - Utilizatorul introduce date de simulare, dar acestea nu sunt valide sau sunt nule. Programul detectează acest lucru și afișează un mesaj de eroare pe ecran, solicitând reintroducerea datelor și atenționând zona care nu corespunde.
  - Se revine la pasul inițial, utilizatorul fiind încurajat să reintroducă datele.
- Neselectarea unei strategii
  - Utilizatorul nu selectează o strategie, iar aplicația afișează un mesaj de eroare pe ecran, cerând utilizatorului să selecteze o strategie validă.e revine la pasul inițial



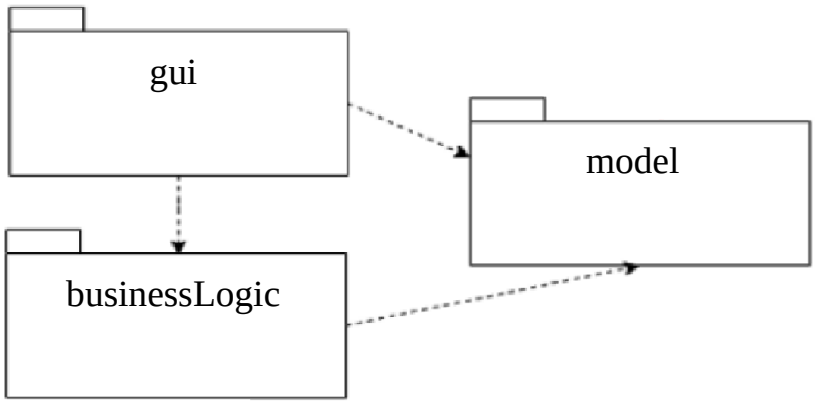
3. Proiectare

a. Proiectarea OOP

Aplicația este structurată în 3 pachete principale, cel de businessLogic fiind cel mai dezvoltat din punct de vedere al numărului de clase. Clasele au fost definite în așa fel încât să se respecte principiul OOP de abstractizare și encapsulare a datelor.

b. Diagrama UML a claselor

c. Diagrama UML a pachetelor



d. Structuri de date

Structura principală utilizată pentru clienți din thread-uri este un BlockingQueue, care asigură accesarea corespunzătoare a datelor stocate. Un alt tip de date folositor în lucrul cu thread-uri este Atomic Integer, definit pentru WaitingTime în clasa Server.

BlockingQueue este o structură de date optimizată pentru lucrul cu threaduri, fiind o coada care așteaptă automat să respecte starea operației activate (așteaptă existența unui obiect pentru recuperarea lui, așteaptă disponibilitatea spațiului pentru stocarea lui). Aceasta este utlizată pentru gestiunea tuturor clienților.

e. Algoritmi folositi

Algoritmii utilizați sunt în principal legați de iterarea clienților într-un mod sigur. Aceștia sunt prezenți în principal în clasa Scheduler. De asemenea, sunt prezenți doi algoritmi de strategie: ShortestQueue și ShortestTime. Aceștia reprezintă de asemenea o iterare asupra datelor și utilizarea informației stocate în clasa Statistic și Scheduler pentru optimizarea distribuirii clienților.

4.Implementare

În dezvoltarea acestui proiect, am utilizat 3 module principale: businessLogic, care realizează pornirea și setarea simulării, model pentru proiectarea clienților și a thread-urilor, și interfața grafică pentru urmărirea activității acestor două module.

4.1 Clasa Client

Aceasta reprezintă clasa care determină una dintre cele doua structuri principale pe care se defineste proiectul.

Câmpuri:

`private final int id;`
`private final int arrivalTime;`
`private int serviceTime;`
	Datele necesare pentru a descrie un client.

Constructor:

`public Client(int id, int arrivalTime, int serviceTime)`	Crearea obiectului client cu datele relevante pentru identificarea acestuia.

Metode:

`public void decreaseServiceTime()`	Realizează updatarea fiecărui client în timp ce acesta este servit.
`@Override` `public String toString()`	Afișare relevantă pentru fiecare client.

Gettere:

`public int getServiceTime()`
`public int getArrivalTime()`
`public int getId()`
	Pentru accesarea datelor reprezentative clientilor.

4.2 Clasa Server

Această clasă implementează interfața Runnable și reprezintă un thread pentru fiecare coadă. Activitatea acesteia este de a servi clientul care se află poziționat într-un BlockingQueue și a decrementa ServiceTime-ul clientului pentru a reprezenta servirea în timp real. Această clasă este gestionată și creată de Scheduler.

Câmpuri:

`private final BlockingQueue<Client> clients;`	BlockingQueue este utlizat pentru gestiunea tuturor clientilor intr-un mod sigur.
`private static AtomicInteger waitingPeriod;`	Structura de date thread-safe, utilizata pentru generarea statisticii de la final, prin incrementarea cand se serveste un client si respectiv decrementarea cand se indeparteaza un client.
`private Client currentClient;`	Clientul servit la momentul actual. Definit astfel pentru a putea fi accesat prin gettere de alte clase pentru afisare.

Constructor:

`public Server()`	Crearea si initializarea celor doua structuri de date.

Metode:

`public void run()`	Metoda functionala care determina activitatea threadului. Cat timp programul este activ, procesul este activ. Cand exista un nou client amplasat in varful cozii, acesta este servit cu metoda `serveClient(currentClient)` detaliata mai jos.
`private void serveClient(Client client) throws InterruptedException`	Cât timp condiția `client.getServiceTime() > 0` este îndeplinită, threadul așteaptă o secundă și apoi decrementează timpul de servire al clientului. Când condiția nu mai este îndeplinită, clientul este eliminat din blockingQueue și din variabila de currentClient. De asemenea, se modifică waitingPeriod relativ cu clientul actual.

4.3 Clasa SimulationManager

După numele clasei, acesta este creierul principal al programului, legând toate clasele definite între ele. De asemenea, implementează interfața Runnable și definește threadul principal al simulației, descris mai jos.

Câmpuri:

`private final Display display;` `private Scheduler scheduler;`	Legături cu celalalte clase, cu Display pentru afișarea progresului și cu Scheduler pentru impartirea clientilor în functie de strategie.
`private final ArrayList<Client> clients = new ArrayList<>();`	Structura de date în care sunt stocați clienții înainte de a fi repartizați unui thread / unei cozi.

`private int nbOfClients;`
`private int nbOfServers;`
`private int simulationInterval;`
`private int minArrivalTime;`
`private int maxArrivalTime;`
`private int minServiceTime;`
`private int maxServiceTime;`
`private SelectionPolicy s;`
	Datele simulării care sunt preluate din interfață.

Constructor:

`public SimulationManager(Display display)`	Crearea unei legaturi intre manager si interfata grafica pentru ca datele modificate in intermediul simularii sa fie vizualizate in timp real in intermediul interfetei grafice.
`public void simMan()`	O extensie a constructorului principal, care definește date necesare pentru pornirea simulării dupa ce au fost primite în interfața grafică. Această secțiune este separată de constructor deoarece metoda `generateRandomClients()` are nevoie de informații preluate din interfața grafica iar simulation manager este initializat înainte pentru a nu duplica variabilele.

Metode:

`public void generateRandomClients()`	Generează clienți în mod aleator pe baza datelor citite din interfață.
`@Override public void run()`	Metoda funcțională cea mai amplă a proiectului, definind întregul proces de simulare, incluzând afișarea rezultatelor în interfața grafică și în fișier, rezolvând clienții a caror `client.getArrivalTime()` este egal cu timpul curent din simulare. De asemenea, așteaptă o secundă înaintea următoarei iterații pentru a oferi timp suficient threadurilor să execute `Thread.sleep(1000)`. Dupa încheierea intervalului de simulare, sunt afișate și computate statisticile relevante în urma datelor produse în simulare.
`public String remainingClients()`	Afișarea clienților generați a căror `arrivalTime` nu corespunde cu timpul curent.
`public void appendToFile(String content)`	Realizează scrierea în fișier a informațiilor produse de simulare.

Settere:

`public void setStrategy(SelectionPolicy s)`
`public void setNbOfClients(int nbOfClients)`
`public void setNbOfServers(int nbOfServers)`
`public void setSimulationInterval(int s)`
`public void setMinArrivalTime(int minArrivalTime)`
`public void setMaxArrivalTime(int maxArrivalTime)`
`public void setMinServiceTime(int minServiceTime)`
`public void setMaxServiceTime(int maxServiceTime)`
	Settere accesate în clasa Display pentru a oferi managerului datele introduse de utilizator necesare simularii.

4.4 Clasa Scheduler

Această clasă determină interacțiunea cu thread-urile programului și amplasarea clienților în funcție de strategia selectată. Clasa este responsabilă și de afișarea conținutului fiecărui thread, fiind singura clasă care are acces la ele în acest mod.

Campuri:

`private final ArrayList<Server> servers = new ArrayList<>();`	Salvarea cozilor intr-un array list pentru iterarea cu usurință.
`private Strategy strategy;`	Strategia adoptata pentru impartirea clienților.

Constructor:

`public Scheduler(int numberOfServers,SelectionPolicy sel)`	Asignarea strategiei selectate si crearea unui numar determinat de threaduri, care isi incep executia aici. `t.start()`

Metode:

`public void dispatchClient(Client client)`	In functie de strategie, se adauga clientul cu `strategy.addTask(servers,client);` Add task este o metoda din interfata Strategy care are doua implementari descrise in sectiunea respectiva.
`public void changeStrategy(SelectionPolicy s)`	Se atribuie o strategie de selectie a clientilor pentru a asigura distribuirea lor corespunzatoare.
Urmatoarele metode sunt prezente pentru informatii suplimentare referitoare la cozi, pentru a fi utilizate in implementarea statisticilor, afisajelor si decizilor luate de strategii.	
`public int amountOfPeopleWaiting()`	Numarul total de persoane care se afla in toate cozile la un moment dat. Utilizat pentru a descoperi peak time.
`public static int amountOfPeopleWaitingInS(Servers)`	Numarul de persoane care se afla intr-o singura coada specificata. Utilizat pentru strategia shortestQueue.
`public String contentInQueues()`	Metoda pentru afisarea intregului continut al cozilor in textArea desemnat acestei reprezentari.
`public String topOfQueues()`	Afiseaza clientul care este servit in momentul in care este apelata metoda, sau „closed” in cazul in care nu exista un client. Utilizata pentru afisarea progresului in timp real in textArea desemnat acestei reprezentari.
`public void averageTime(int currentTime)`	Indeplineste incapsularea threadurilor cand este nevoie de acestea pentru a afla average waiting time.

4.5 Interfata Strategy

Aceasta interfata determina doua strategii diferite, implementate prin aceasi semnatura. Metoda definita in interfata este implementata in clasa care corespunde fiecarei strategii pentru a atinge abstractizarea claselor care implementeaza interfata.

`void addTask(List<Server> servers, Client client);`	Definirea semnăturii metodei care va fi construită în clasele menționate mai jos.

4.5.1 Clasa ShortestQueueStrategy

`void addTask(List<Server> servers, Client client)`	Initializarea serverului cautat si a celui mai scurt timp de asteptare cu primul server din lista si apoi iterarea tuturor serverelor, verificand conditia `if (server.getWaitingPeriod() <= shortestTime)` pentru fiecare server in parte.

4.5.2 Clasa ShortestTimeStrategy

`void addTask(List<Server> servers, Client client)`	Initializarea serverului cautat si a celui mai scurt timp de asteptare cu primul server din lista si apoi iterarea tuturor serverelor, verificand conditia `if (serv < shortestServ)` pentru fiecare server in parte, unde shortestServ este serverul cautat si serv reprezinta un server din iteratie. Acestea sunt definite cu ajutorul metodei `amountOfPeopleWaitingInS (server)` din clasa Scheduler.

4.5.3 Enumeratia SelectionPolicy

`public enum SelectionPolicy { SHORTEST_TIME, SHORTEST_QUEUE }`	Referirea la cele doua strategii intr-un mod intuitiv.

4.6 Clasa Statistic

Clasa statistic este definita pentru a separa datele rezultate în urma simulării de alte clase care au un alt scop principal.

Campuri:

`private static double averageST;`
`private static double averageWT;`
`private static int peakHour;`
`private static int peakPeople = 0;`
	Campuri cu rolul de a stoca rezultatele obtinute din simulare. Acestea sunt utilizate pentru salvarea informatiei relevante si afisarea cu usurinta la final.

`private static final ArrayList<Integer> allWT = new ArrayList<>();`	Salvarea timpului de asteptare total peste toate threadurile la fiecare timp. Indicele reprezinta secunda iar valoarea stocata la acel indice este timpul de asteptare inregistrat la momentul respectiv.

Metode:

`public static void peakHourDecider(int people, int currentTime)`	Metoda este apelată la fiecare incrementare a timpului cu ajutorul metodei `scheduler.amountOfPeopleWaiting()` pentru campul people, pentru a determina numărul maxim de persoane care a existat concomitent amplasat la cozi.
`public static void addWaitingTime(int currentTime, ArrayList<Server> servers) s)`	Formarea structurii `ArrayList<Integer> allWT` la fiecare incrementare a timpului, pentru a putea fi iterat la finalizarea simularii.
`public static void averageWaitingTime(int totalTime)`	Aflarea rezultatului *averageWT* Prin parcurgerea ArrayListului mentionat anterior și crearea unei medii a acestor termeni.
`public static void setAverageST(double averageST)`	Determinarea unei medii a timpului de servire intre toti clienții generați.
`public static String showStats()`	Metoda care se ocupă cu afișarea rezultatului la finalul fișierului / finalul simulării.

4.3 Clasa Display

Această clasă realizează crearea interfeței grafice, având de asemenea clase lambda pentru implementarea ActionListenerului. Aceasta determină preluarea datelor tastate de utilizator și afișarea în timp real a progresului realizat de interfață. Implementarea interfeței grafice a fost realizată utilizând elemente Swing UI, amplasate pe un panou fără manager de aspect pentru a oferi libertate în poziționarea componentelor.

Câmpuri:

`private SimulationManager s;`	Legatura interfeței cu BusinessLogic.

`private final JPanel mainPanel;`
`private final JScrollPane progressScrollPane;`
`private final JScrollPane queueContentScrollPane;`
	Paneluri pentru afișarea conținutului pe interfața grafică. Al doilea și al treilea sunt legate fiecare de propriul JTextArea.

`private final JTextArea progressTextArea;` `private final JTextArea queueContentTextArea;`	Cele două JTextArea menționate anterior.

`private JTextField nrClientsTextField;`
`private JTextField nrQueuesTextField;`
`. . .`
`private JTextField serMaxTextField;`
	JTextFielduri diferite pentru preluarea inputului de la utilizator.

`private final JLabel timerLabel = new JLabel();`	Singurul label care nu este definit local în metoda de creare a conținutului interfeței grafice, deoarece este modificat cu timpul curent la fiecare iterație a programului.

`private JButton startButton;`
`private JRadioButton queueStRadioButton;`
`private JRadioButton timeStRadioButton;`
	JButton simplu pentru a incepe simularea. Sunt utilizate și două JRadioButtons pentru selectarea strategiei. Grupul de butoane este derminat local înainte ca butoanele să fie adăugate panelului principal.

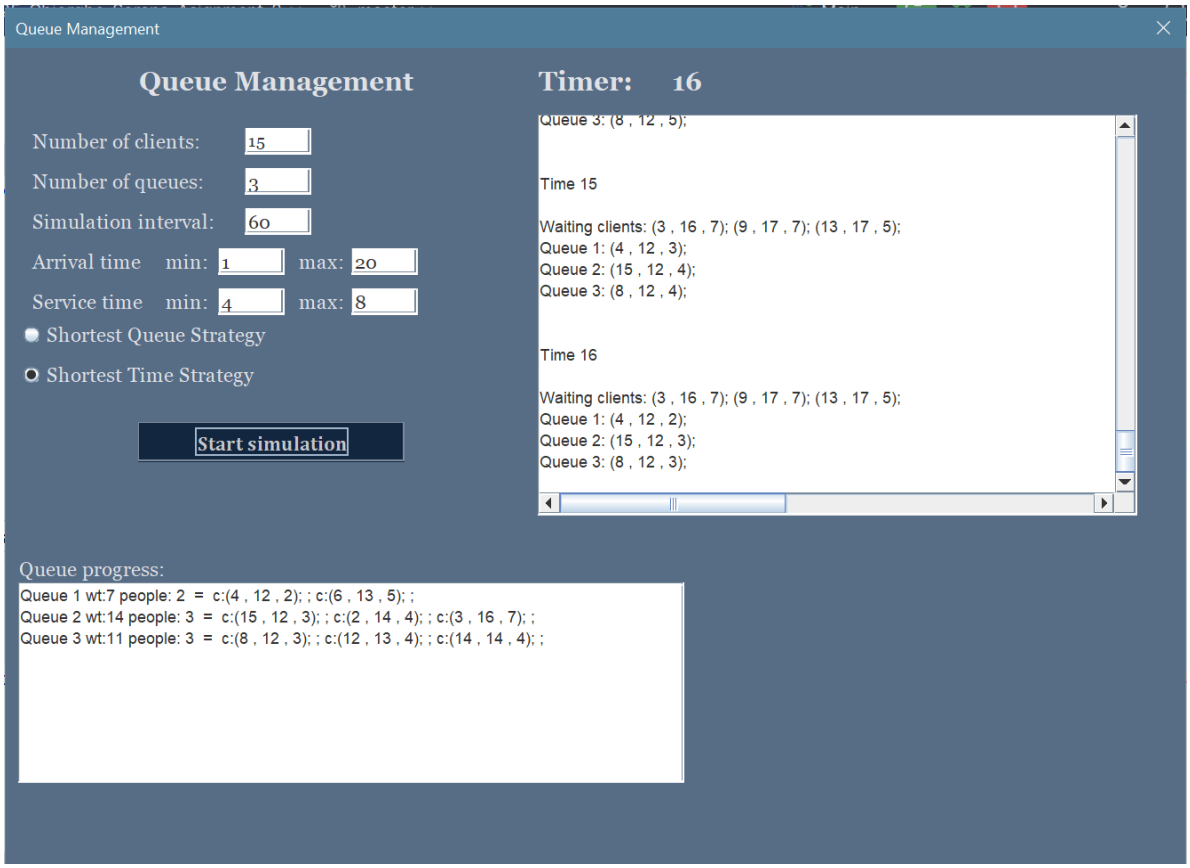
Constructor:

`public Display(Frame parent)`	Creează interfața grafică, apelând diverse metode pentru amplasarea componentelor. Aici există și un ActionListener al butonului start, care începe threadul cu runnable din SimulationManager dupa ce au fost introduse datele.

Metode:

`private void getData()`	Asignarea datelor din SimulationModel valorile tastate și selectate în interfață.
`private void createContent()`	Adăugarea componentelor în interfața grafică.
`public void addToMainTextArea(String s)`	Adăugarea informației generate la fiecare secundă este adaugată la textArea pentru progres.
`public void setContentTextArea(String s)`	Modificarea afișorului cu dimensiunea fiecărei cozi și alte informații relevante.

Aspectul interfeței poate fi observat în imaginea de mai jos. Colțul din stânga sus reprezintă meniul de selecție, urmat de butonul principal. Panoul din dreapta conține un jurnal al activității curente a cozii și a clienților care încă nu au ajuns. Panoul din stânga jos determină conținutul cozilor în întregime și informații legate de numărul de persoane amplasate în coadă și timpul de așteptare. Aceste date sunt utilizate în funcție de strategia selectată și pot fi utilizate drept o confirmare că simularea alege coada optimă conform strategiei în amplasarea clienților noi.



4. Rezultate

Simularea a fost pornită pentru 3 seturi diferite de date pentru ambele strategii cu scopul de a observa diferența rezultatelor generate de acestea.

Strategy	Test 1	Test 2	Test 3
	$N = 4$ $Q = 2$ $t_{simulation}^{MAX} = 60$ seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [2, 30]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [2, 4]$	$N = 50$ $Q = 5$ $t_{simulation}^{MAX} = 60$ seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [2, 40]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [1, 7]$	$N = 1000$ $Q = 20$ $t_{simulation}^{MAX} = 200$ seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [10, 100]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [3, 9]$
Shortest time	Average waiting time: 0.12 Average service time: 2.5 Peak hour: at second 15 with 1 people	Average waiting time: 1.52 Average service time: 3.08 Peak hour: at second 30 with 11 people	Average waiting time: 70.83 Average service time: 4.49 Peak hour: at second 100 with 599 people
Shortest queue	Average waiting time: 0.12 Average service time: 2.75 Peak hour: at second 5 with 1 people	Average waiting time: 2.5 Average service time: 3.62 Peak hour: at second 40 with 14 people	Average waiting time: 108.38 Average service time: 5.5 Peak hour: at second 100 with 663 people

Primul test realizat pe un număr mic de clienți nu a generat un caz pentru care două persoane ajung la același timp / o persoană ajunge când o coadă este ocupată cu o persoană deja, așadar nu se poate observa o diferență majoră între cele două strategii. Cozile sunt goale într-o proporție foarte mare pe parcursul simulării, determinând astfel un waitingTime aproape inexistent. Raportul între clienți și numărul de cozi este o coadă la 2 clienți.

Al doilea test este realizat pe un număr mai mare de clienți cu un timp de servire mai variat, așadar există cazuri în care două persoane ajung la același timp / o persoană ajunge când o coadă este ocupată cu un client deja. Existând astfel de cazuri, se poate observa diferența dintre cele două strategii: strategia ShortestTime are un timp mediu de așteptare cu o secundă mai mic decât strategia ShortestQueue. Timpul de așteptare este relativ mic în ambele strategii. Raportul între clienți și numărul de cozi este o coadă la 10 clienți.

Al treilea test oferă o statistică interesantă asupra simulării pe un număr foarte mare de clienți care trebuie să fie amplasați eficient la un număr limitat de cozi. Raportul între clienți și numărul de cozi este o coadă la 50 de clienți.

În ambele cazuri de simulare, cozile au rămas pline până la încheierea timpului maxim de simulare, însă cu un număr de clienți și timp de așteptare diferit. Observatii asupra cozilor la secunda 200:

	Strategia Shortest Queue	Strategia Shortest Time
Numarul clientilor	Rămân 14 – 17 clienți la fiecare coadă. 301 clienți rămași în total	Rămân 7 – 9 clienți la fiecare coadă. 151 clienți rămași în total
Timp de asteptare	Între 70 și 90 în funcție de coadă.	Între 28 și 34 în funcție de coadă.

După cum se poate observa, pentru strategia shortestQueue, timpul maxim al simulării ar fi trebuit extins cu aproximativ 1/3 din timpul dat pentru a putea încheia procesarea clienților în întregime, pe când acest raport pentru strategia shortestTime este de aproximativ 1/5. Numărul de clienți care nu sunt serviți cu strategia shortestQueue este dublu față de strategia shortestTime.

Așadar, concluzia este evidentă: strategia shortestTime eficientizează amplasarea clienților la coadă în funcție de timpul de care au nevoie pentru a fi serviți iar această eficiență se poate observa clar în rezultatele produse de simulare.

5. Concluzii

Acest proiect a reprezentat o oportunitate relevantă în dezvoltarea propriilor cunoștințe și abilități și în aprofundarea acestora. Am dobândit abilitatea de a lucra cu threaduri și implementarea acestora într-un mod sigur, folosind zone sincronizate unde era nevoie. Am învățat concepte legate de threaduri, cum ar fi lucrul cu structuri de date optimizate pentru threaduri (blockingQueue, AtomicInteger), utilizarea lock-urilor (ReentrantLock și CountdownLock) într-o fază inițială a proiectului, care ulterior m-a ajutat să formeze soluții mai optime de accesare a datelor.

Mi-am reîmprospătat cunoștințele legate de interfețe și clase statice, scrierea în fișier și diferite elemente de interfață grafică pentru a oferi progresul în timp real în zonele de text desemnate. Proiectul mi-a oferit oportunitatea de a aprofunda și repeta diferite elemente și paradigme ale limbajului de programare orientat pe obiect.

Aplicația determină o simulare cu o concluzie clară, implementând cele două strategii cu succes. În ceea ce privește viitoarele dezvoltări ale aplicației, se pot adăuga mai multe strategii pentru a observa efectul acestora în comparație cu celelalte două implementate. De asemenea, se poate complica modelul de simulare pentru a adăuga diferite abilități speciale fiecărei cozi, precum pauze prestabilite, eficiența sporită în cazul clienților cu timp mare de servire, cozi rapide care nu pot primi clienți cu timp de servire mai mare decât un prag prestabilit.

6. Bibliografie

1. *Git Lab - https://www.jetbrains.com/help/idea/gitlab.html*
2. *Java Threads - https://www.w3schools.com/java/java_threads.asp*
3. *Interfaces - https://www.geeksforgeeks.org/interfaces-in-java/*
4. *AtomicIniger Class - https://www.geeksforgeeks.org/interfaces-in-java/*
5. *Blocking Queue - https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html*
6. *Scrolling text area - https://tips4java.wordpress.com/2008/10/22/text-area-scrolling/*
7. *Automatic scrolling - https://stackoverflow.com/questions/2483572/making-a-jscrollpane-automatically-scroll-all-the-way-down*
8. *Java Create and Write to Files - https://www.w3schools.com/java/java_files_create.asp*
9. *Concurrency in Java - https://medium.com/why-david-y/concurrency-in-java-part-2-locks-locks-and-some-deadlocks-76a4c7af3be7*
10. *JRadioButton and Button Groups - https://www.geeksforgeeks.org/jradiobutton-java-swing/*
11. *Appending instead of overwritting - https://www.geeksforgeeks.org/jradiobutton-java-swing/*
12. *Resurse Laborator - https://dsrl.eu/courses/pt/materials/PT_2024_A2_S1.pdf*