DOCUMENTAȚIE

TEMA 2

NUME STUDENT: Pescaru Silviu-Mihaita

GRUPA: 30225

# CUPRINS

[1. Obiectivul temei 3](#_Toc95297885)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 3](#_Toc95297886)

[3. Proiectare 5](#_Toc95297887)

[4. Implementare 6](#_Toc95297888)

[5. Rezultate 10](#_Toc95297889)

[6. Concluzii 11](#_Toc95297890)

[7. Bibliografie 11](#_Toc95297891)

# Obiectivul temei

Obiectivul principal al temei este implementarea unei aplicații Java de management a unor cozi de clienți, astfel încât clienții să fie adăugați la coada cu timpul minim de așteptare.

Obiective secundare: analiza problemei și identificarea cerințelor, modelarea și implementarea unei aplicații de simulare, testarea aplicației de simulare.

Am avut de implementat următoarele:

Interfața grafică

Managerul simulării

Cozile

Clienții

Controller-ul

Alte clase și funcții ajutătoare

# Analiza problemei, modelare, scenarii, cazuri de utilizare

Analiza:

Programul adaugă în fiecare secundă de simulare clienții care ajung în coada cu timpul de așteptare cel mai scurt, urmând să îi scoată din coadă după ce au fost serviți.

Modelare:

Clasa Task reprezintă clienții, cu atributele specifice: ID-ul fiecăruia, timpul la care ajung în coadă și timpul de servire.

Clasa Server reprezintă cozile și conține coada în care sunt înșirați clienții, precum și timpul de așteptare.

Scenarii:

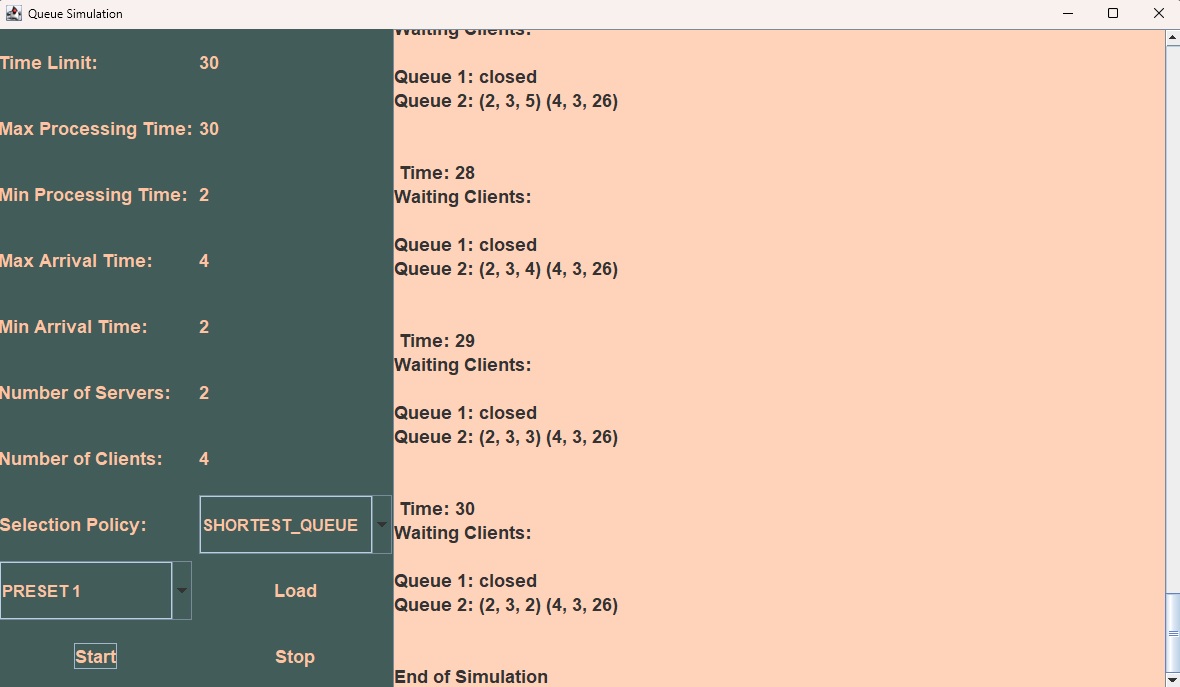
Programul incepe o data cu apasarea butonului Start, avand initial toate cozile goale. Dupa ce a trecut timpul minim de arrival clientii sunt pusi la coada. Programul de simulare a cozilor se va incheia atunci cand toti clientii au fost serviti sau cand timpul limita setat de user in interfata, se va atinge.

Timpul minim de servire trebuie să fie mai mic sau egal cu timpul maxim de servire.

Timpul minim de ajuns la coadă trebuie să fie mai mic sau egal cu timpul maxim de ajuns la coadă.

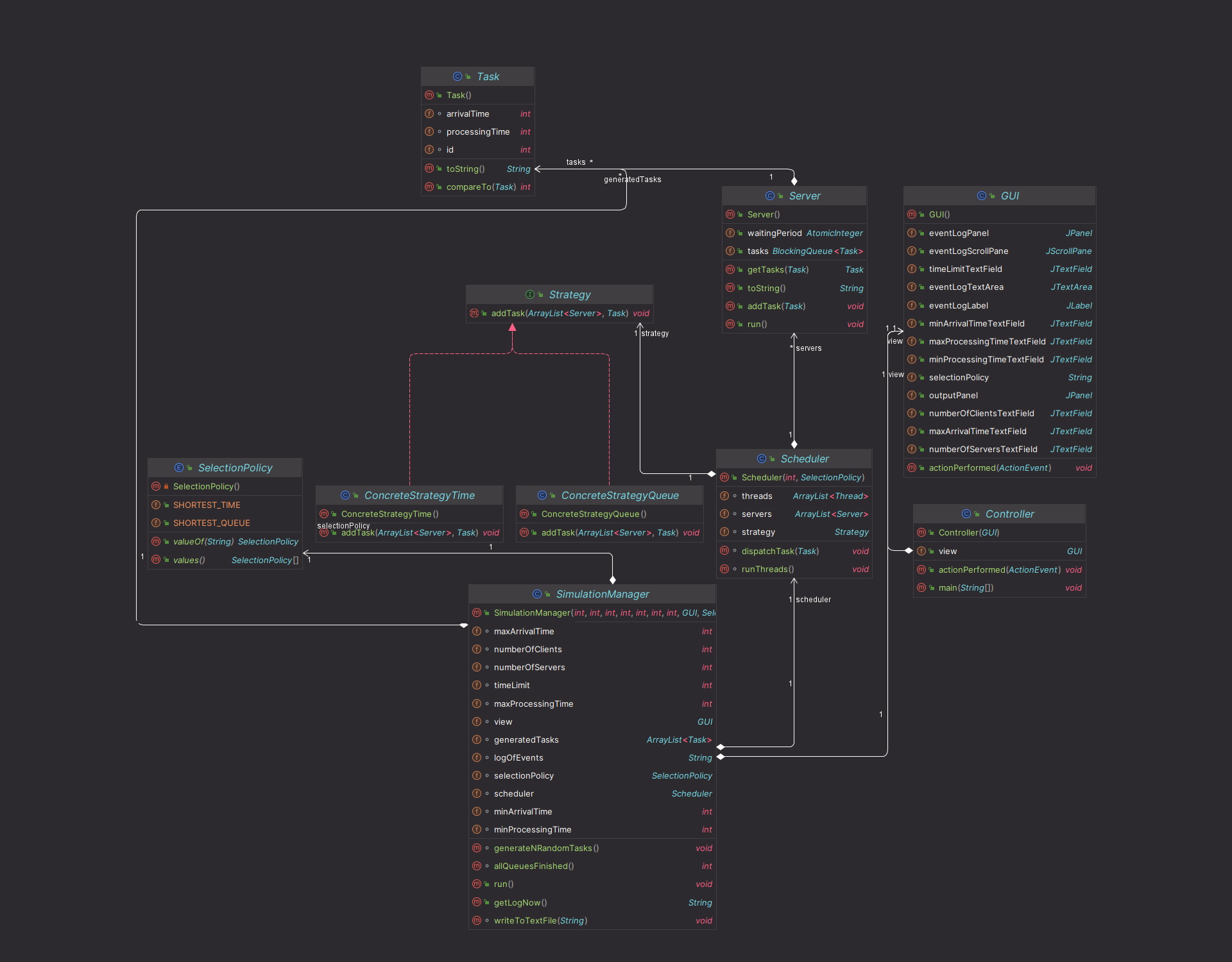
Use cases:

În interfața grafică pot fi inserate datele în câmpurile din partea stângă a ferestrei. Se mai poate selecta unul din cele 3 preseturi, dar și politica de alegere a clientilor pe baza celei mai scurte cozi sau cel mai scurt timp de așteptare. După inserare utilizatorul poate apăsa pe butonul “Load”, după care butonul “Start” pentu a porni simularea.

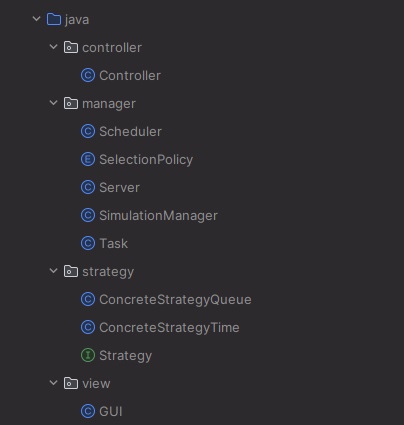


# Proiectare

Diagrama UML:



Ierarhia de fișiere a proiectului:



# Implementare

Clasa `**ConcreteStrategyQueue**` este o implementare a interfeței `**Strategy**`.

Metoda "addTask" este suprascrisă din interfața "Strategy" și are rolul de a adăuga un client în coada serverului cu cea mai mică lungime. Pentru a face acest lucru, metoda caută serverul cu cel mai mic număr de clienți în coadă și adaugă clientul la acesta.

Clasa `**ConcreteStrategyTime**` este o implementare a interfeței `**Strategy**`.

Metoda "addTask" este suprascrisă din interfața "Strategy" și are rolul de a adăuga un client în coada serverului cu cel mai mic timp de așteptare. Pentru a face acest lucru, metoda caută serverul cu cel mai scurt timp de așteptare și adaugă clientul la acesta.

Clasa `**Scheduler**` conține trei atribute: `servers`, `threads` și `strategy`.

Atributul `servers` este o listă de obiecte de tip `Server`, reprezentând fiecare server din sistem. Atributul `threads` este o listă de fire de execuție (`Thread`) corespunzătoare fiecărui server, folosite pentru a procesa sarcinile primite.

Atributul `strategy` este un obiect de tipul unei clase care implementează un anumit algoritm de planificare.

Constructorul clasei `Scheduler` primește ca parametru numărul de servere și creează câte un obiect de tip `Server` pentru fiecare server, stocându-le în lista `servers`. De asemenea, constructorul creează și câte un fir de execuție corespunzător fiecărui server, stocându-le în lista `threads`.

Metoda `changeStrategy` este utilizată pentru a schimba algoritmul de planificare al sistemului. Aceasta primește ca parametru o politică de selecție a serverului și setează algoritmul de planificare corespunzător acelei politici, creând un nou obiect de tipul unei clase care implementează acel algoritm de planificare.

Metoda `dispatchTask` primește ca parametru un client (task) și îl planifică pe unul dintre servere, folosind algoritmul de planificare curent.

Metoda `runThreads` este utilizată pentru a porni toate firele de execuție corespunzătoare serverelor, astfel încât acestea să poată procesa sarcinile primite.

Clasa **Server** este o clasă ce implementează un server, care primește sarcini de la clienți și le procesează. Aceasta implementează interfața Runnable, deci poate fi executată ca un fir de execuție separat.

Atributul tasks este o coadă blocantă (BlockingQueue) de sarcini (Task) pe care serverul le primește. Atributul waitingPeriod este un AtomicInteger ce reprezintă timpul total de așteptare al tuturor sarcinilor din coadă.

Metoda addTask adaugă o nouă sarcină în coada de sarcini a serverului și actualizează timpul total de așteptare.

Metoda getTasks primește o sarcină ca parametru și o returnează.

Metoda run este metoda apelată atunci când firul de execuție al serverului este pornit. Aceasta execută un buclă infinită care așteaptă timp de 1 secundă la fiecare iterație. În fiecare iterație se scade timpul de procesare a primei sarcini din coadă și se actualizează timpul total de așteptare. Dacă timpul de procesare al primei sarcini este zero, aceasta este eliminată din coadă.

Metoda toString returnează o reprezentare sub forma de șir de caractere a tuturor sarcinilor din coadă.

Scopul acestei clase, numită SimulationManager, este de a administra simularea procesării sarcinilor de către mai mulți servere. Aceasta implementează interfața Runnable, astfel încât să poată fi executată într-un fir de execuție separat.

Constructorul acestei clase primește mai mulți parametri, printre care numărul de servere, numărul de clienți, limitele de timp și timpul de procesare, GUI-ul, politica de selecție a serverului și multe altele. De asemenea, se generează un număr specific de sarcini aleatorii.

Metoda generateNRandomTasks() generează sarcinile aleatorii și le sortează în funcție de momentul de sosire. Metoda allQueuesFinished() verifică dacă toate cozile serverelor au fost finalizate.

Metoda run() reprezintă nucleul acestei clase și este cea care conduce simularea. În interiorul acesteia, se simulează sosirea și procesarea sarcinilor, iar acestea sunt atribuite serverelor în conformitate cu politica de selecție specificată. În timpul simulării, evenimentele sunt înregistrate într-un jurnal, care este afișat și în interfața GUI. La finalul simulării, jurnalul este scris într-un fișier text.

Metoda getLogNow() returnează jurnalul evenimentelor care au loc în prezent. Metoda writeToTextFile() scrie jurnalul evenimentelor într-un fișier text, cu un nume specific.

Clasa **Task** este utilizată pentru a stoca informații despre un client în cadrul unui sistem de simulare. Această clasă are trei atribute: ID-ul clientului, timpul de sosire și timpul de servire.

Pentru a permite sortarea clientilor după timpul lor de sosire, clasa implementează interfața Comparable și supraîncarcă metoda compareTo. Acest lucru permite compararea a doi clienți pe baza timpului lor de sosire.

În plus, clasa Task supraîncarcă și metoda toString, care returnează un șir de caractere cu informații despre client în formatul "(ID, timp de sosire, timp de servire)". Această metodă este utilă pentru a afișa informații despre client în interfața grafică a utilizatorului sau în alte mesaje.

Clasa "**Controller**" este o clasă Java care implementează interfața "ActionListener" și conține metoda "actionPerformed", care definește acțiunile care trebuie luate atunci când utilizatorul interacționează cu elementele GUI ale aplicației. Această clasă primește o referință la obiectul GUI ca parametru de intrare și este responsabilă pentru preluarea datelor introduse de utilizator și inițierea procesului de simulare a sistemului de coadă.

Metoda "actionPerformed" este declanșată atunci când utilizatorul apasă unul dintre butoanele GUI ("Start" sau "Stop"). Dacă butonul apăsat este "Start", clasa preia datele introduse de utilizator prin intermediul elementelor GUI (numărul de servere, numărul de clienți, timpul de sosire minim și maxim al clienților, timpul de servire minim și maxim al clienților, limita de timp a simulării și politica de selecție a serverelor) și inițiază obiectul "SimulationManager" pentru a începe simularea sistemului de coadă. În caz contrar, dacă butonul apăsat este "Stop", aplicația se va opri.

Tot în aceasta clasa este și main-ul care creeaza un obiect de tip GUI ce initializeaza interfata grafica și implicit executia programului.

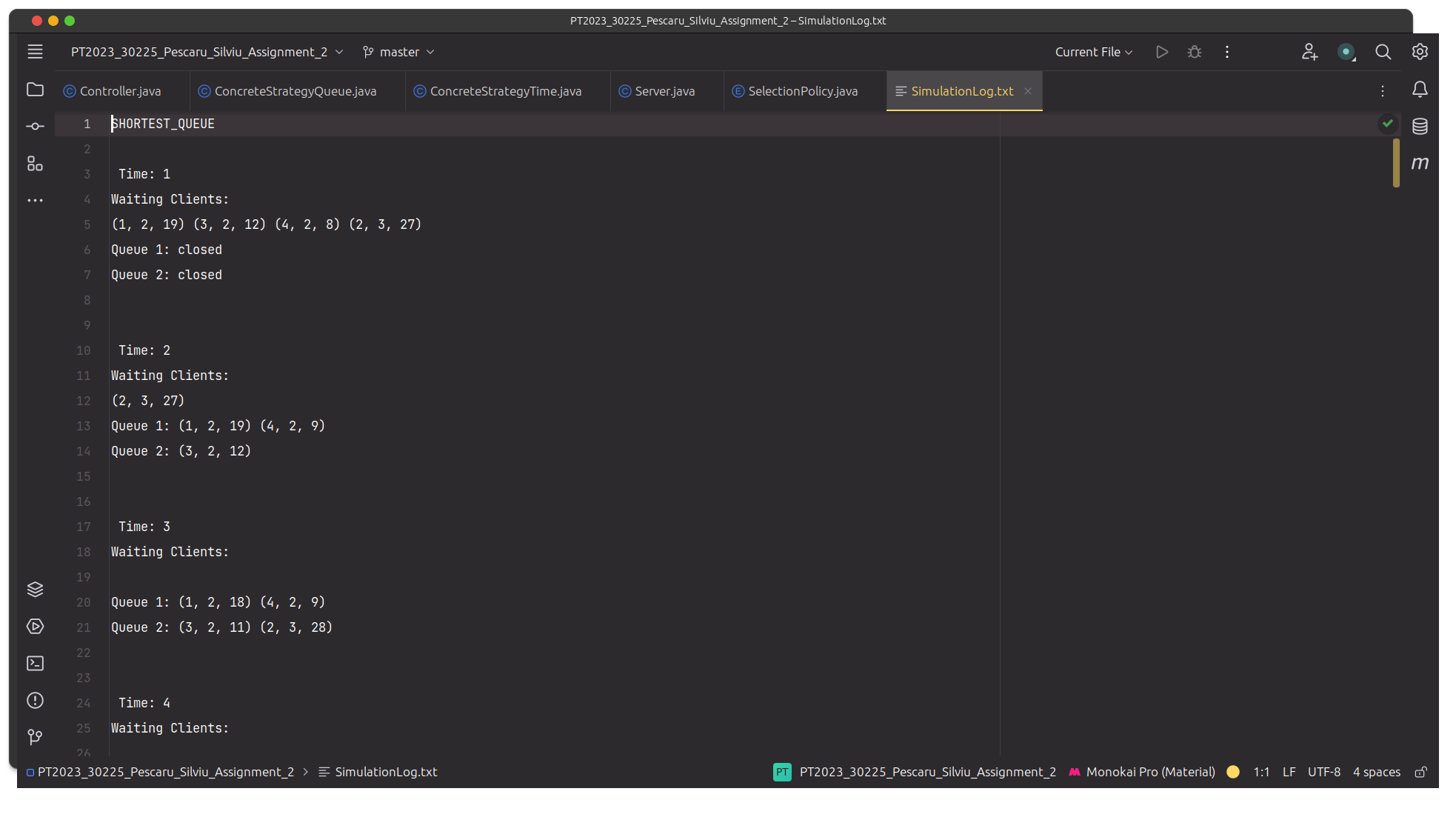
Clasa **GUI** este o componenta vizuala a unei aplicatii de simulare a unei cozi. Aceasta clasa extinde clasa JFrame si implementeaza interfata ActionListener pentru a putea reactiona la evenimente. Clasa contine un numar de campuri JTextField pentru introducerea de date de intrare ale utilizatorului si o serie de panouri pentru afisarea de date de iesire.

Constructorul clasei este utilizat pentru a initializa si construi fereastra GUI. Acesta creaza si adauga la fereastra un panou de intrare, un panou de iesire si un panou de jurnalizare a evenimentelor. Panoul de intrare contine etichete si campuri de text pentru introducerea datelor de intrare ale utilizatorului. Panoul de iesire este utilizat pentru a afisa rezultatele simulatiei si jurnalul evenimentelor. Panoul de jurnalizare a evenimentelor este utilizat pentru a afisa mesajele de jurnalizare generate de simulare.

Clasa are, de asemenea, o serie de metode si variabile pentru a gestiona interactiunea cu utilizatorul si pentru a comunica cu clasa Controller pentru a iniția simularea. Variabila selectionPolicy este folosita pentru a stoca politica de selectare a cozi de catre clienti. Clasa implementeaza metoda actionPerformed pentru a putea reactiona la evenimente, cum ar fi apasarea butonului "Start Simulation".

# Rezultate

Rezultatele testărilor pot fi găsite în fișierele text “SimulationLog.txt”, sau în aplicație, prin introducerea datelor în interfața grafică și rularea simulării.



# Concluzii

Cu ocazia acestei teme am realizat cum pot implementa un sistem de management al cozilor ce poate fi implementat la diverse restaurante, institutii (ex. platirea impozitelor), etc.

# Bibliografie

[https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html](https://www.geeksforgeeks.org/introduction-to-java-swing/)

https://dsrl.eu/courses/pt/materials/PT2023\_A2\_S1.pdf

https://www.w3schools.com/java/java\_enums.asp

https://www.baeldung.com/java-string-to-enum

https://examples.javacodegeeks.com/java-development/desktop-java/swing/jtextfield/create-jtextfield-with-border/

https://examples.javacodegeeks.com/java-development/desktop-java/swing/java-swing-boxlayout-example/

https://examples.javacodegeeks.com/java-swing-layouts-example/

https://stackoverflow.com/questions/15507639/how-do-i-center-a-jtextfield

https://chat.openai.com