DOCUMENTATIE

TEMA *2 – QUEUE MANAGEMENT*

NUME STUDENT: PETRENCIUC AMELIA-ANDREEA

GRUPA: ………30225…………

# 

# CUPRINS

[1. Obiectivul temei 3](#_Toc95297885)

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

[3. Proiectare 5](#_Toc95297887)

[4. Implementare 8](#_Toc95297888)

[5. Rezultate 16](#_Toc95297889)

[6. Concluzii 17](#_Toc95297890)

[7. Bibliografie 18](#_Toc95297891)

# Obiectivul temei

Acest proiect presupune crearea unui program în Java pentru gestionarea cozilor. Programul primește următoarele date de intrare: numărul de cozi (Q), numărul de clienți (N) care urmează să fie serviți și durata maximă pentru care cozile sunt deschise (Tsimulation). De asemenea, se introduc intervalele de timp pentru când clienții sunt gata să intre în coadă (minim și maxim) și durata de servire a unui client (minim și maxim).

Scopul este de a genera aleator un număr de clienți, în funcție de datele de intrare, cu timpuri aleatorii, dar care să respecte intervalele introduse. Pentru eficiență, trebuie implementate metode astfel încât fiecare client care așteaptă în coadă să fie redirecționat către coada cu cel mai mic timp de așteptare. Programul trebuie să folosească fire de execuție (thread-uri): fiecare coadă are un fir de execuție asociat, astfel încât fiecare coadă să poată funcționa independent.

Este important ca fiecare coadă să fie gestionată eficient și să se asigure că clienții sunt serviți într-un mod just și rapid, respectând prioritățile și timpurile de așteptare minime.

Top of Form

Sub-obiectivele proiectului includ:

1.Proiectarea (Capitolul 3): Acest capitol va descrie organizarea proiectului, includând structura generală a programului, relațiile între diferitele componente și modul în care acestea interacționează.

2. Implementarea (Capitolul 4): Aici veți găsi informații detaliate despre clasele și modulele care alcătuiesc programul. Acest capitol va cuprinde descrieri ale claselor, metodelor și variabilelor, precum și modul în care acestea sunt implementate pentru a îndeplini cerințele proiectului.

3. Testarea proiectului (Capitolul 5): Acest capitol se va concentra pe procesul de testare a programului pentru a asigura că funcționează corect și în conformitate cu specificațiile. Va include planuri de testare, scenarii de testare și rezultatele testelor efectuate pentru a valida funcționalitatea și corectitudinea programului.

# 

# Analiza problemei, modelare, scenarii, cazuri de utilizare

*Cerințele funcționale ale proiectului includ:*

* *Configurarea simulării: Utilizatorul trebuie să poată configura parametrii simulării, cum ar fi numărul de cozi, numărul de clienți, timpul maxim de simulare etc.*
* *Pornirea simulării: Utilizatorul trebuie să poată iniția simularea după ce a configurat parametrii.*
* *Afișarea evoluției cozilor în timp real: Aplicația trebuie să ofere o interfață grafică sau textuală care să prezinte în timp real cum se schimbă starea coziilor în timpul simulării.*

*Cerințele non-funcționale ale proiectului includ:*

* *Ușurința în utilizare: Aplicația trebuie să fie intuitivă și ușor de folosit pentru utilizatori, astfel încât să poată configura și porni simularea fără dificultăți sau instrucțiuni complexe.*

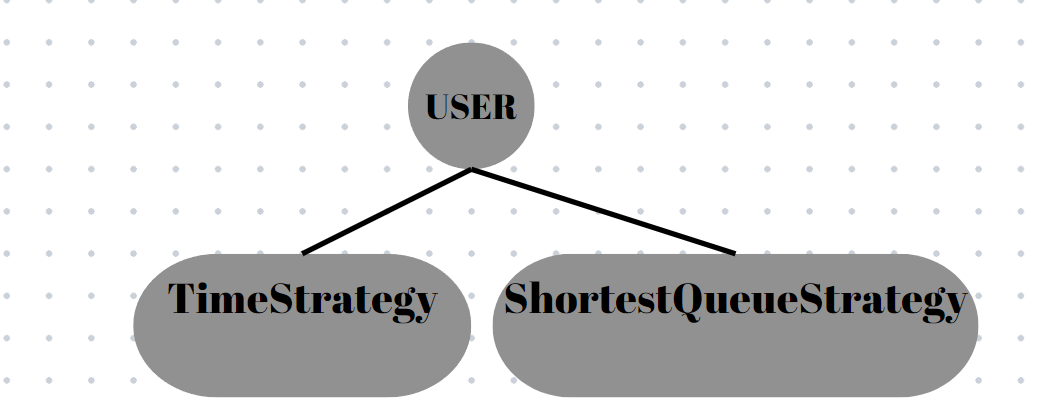
**Caz de utilizare:** Adăugare date

**Actor principal**: Utilizatorul

**Scenariu principal de succes**:

1. Utilizatorul introduce în interfața grafică: numar de client, numar de cozi, interval de simulare, timp minim de servire, timp maximi de servire, timp minim de asteptare, timp maxim de asteptare.
2. Utilizatorul selectează optiune dorita (shortest queue sau in functie de timp).
3. Se afișează rezultatele pe ecran, apar clientii pe ecran si sunt simulate cozile de asteptare.

Această diagramă Use Case reprezintă funcționalitățile sistemului și actorul implicat (în acest caz, singurul actor este utilizatorul), precum și relațiile dintre acestea.

**

# Proiectare

Schema bloc a aplicatiei este urmatoarea:



*Codul este structurat in trei pachete: Model, Simulation, Logic.*

Clasele pe care le-am utilizat pentru realizarea acestei structuri sunt următoarele:

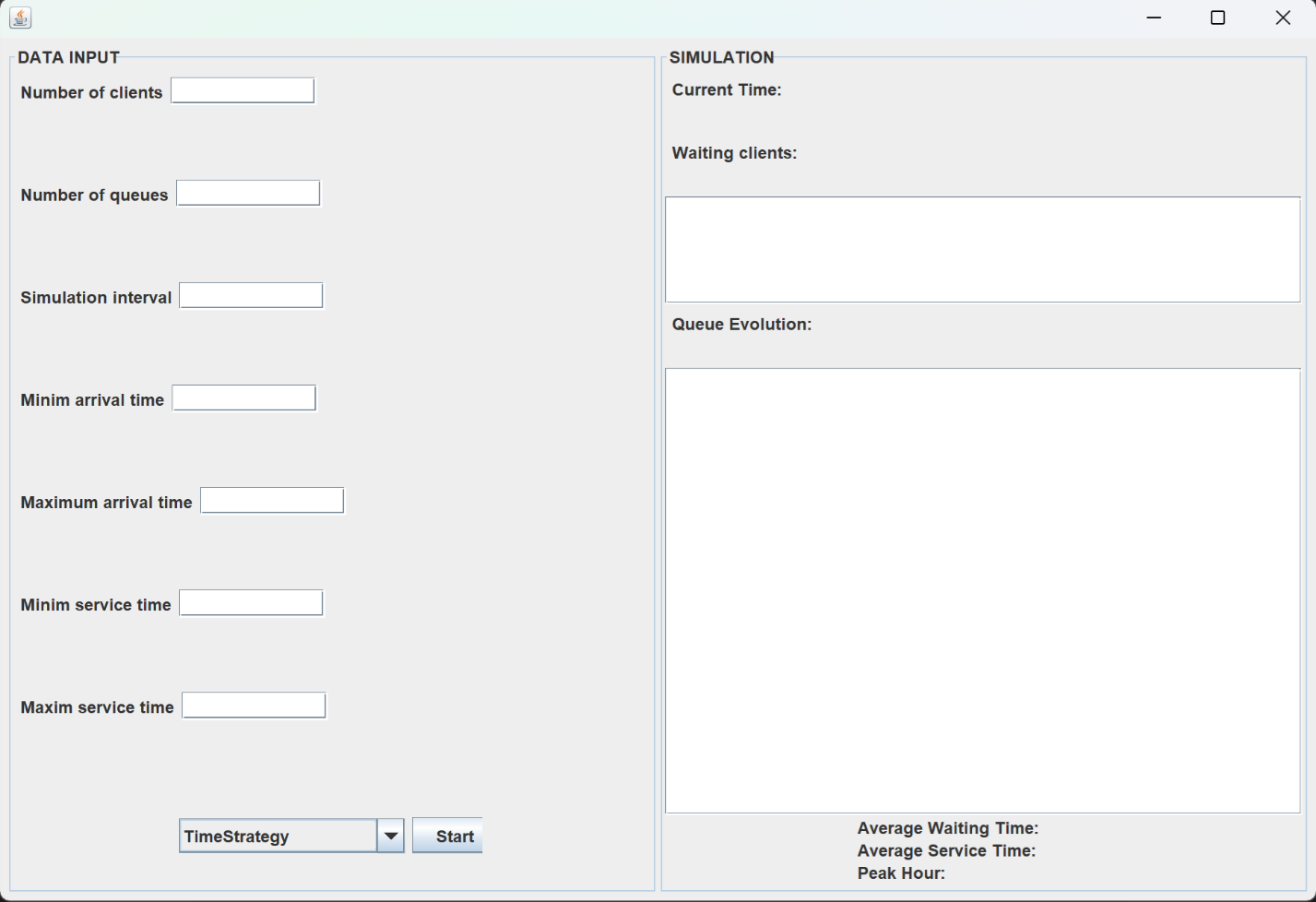
Pachetul **Model** contine:

* Clasa **Task** reprezintă entitatea centrală în simulare, fiind subiectul acțiunilor care se desfășoară: adăugarea și eliminarea din cozi. Clienții sunt generați aleatoriu în cadrul aplicației. Această clasă encapsulează caracteristicile și comportamentul specific al unui client.Clasa
* Clasa **Server** descrie o coadă și include atât atributele cât și metodele specifice acesteia. O coadă poate găzdui mai mulți clienți simultan.

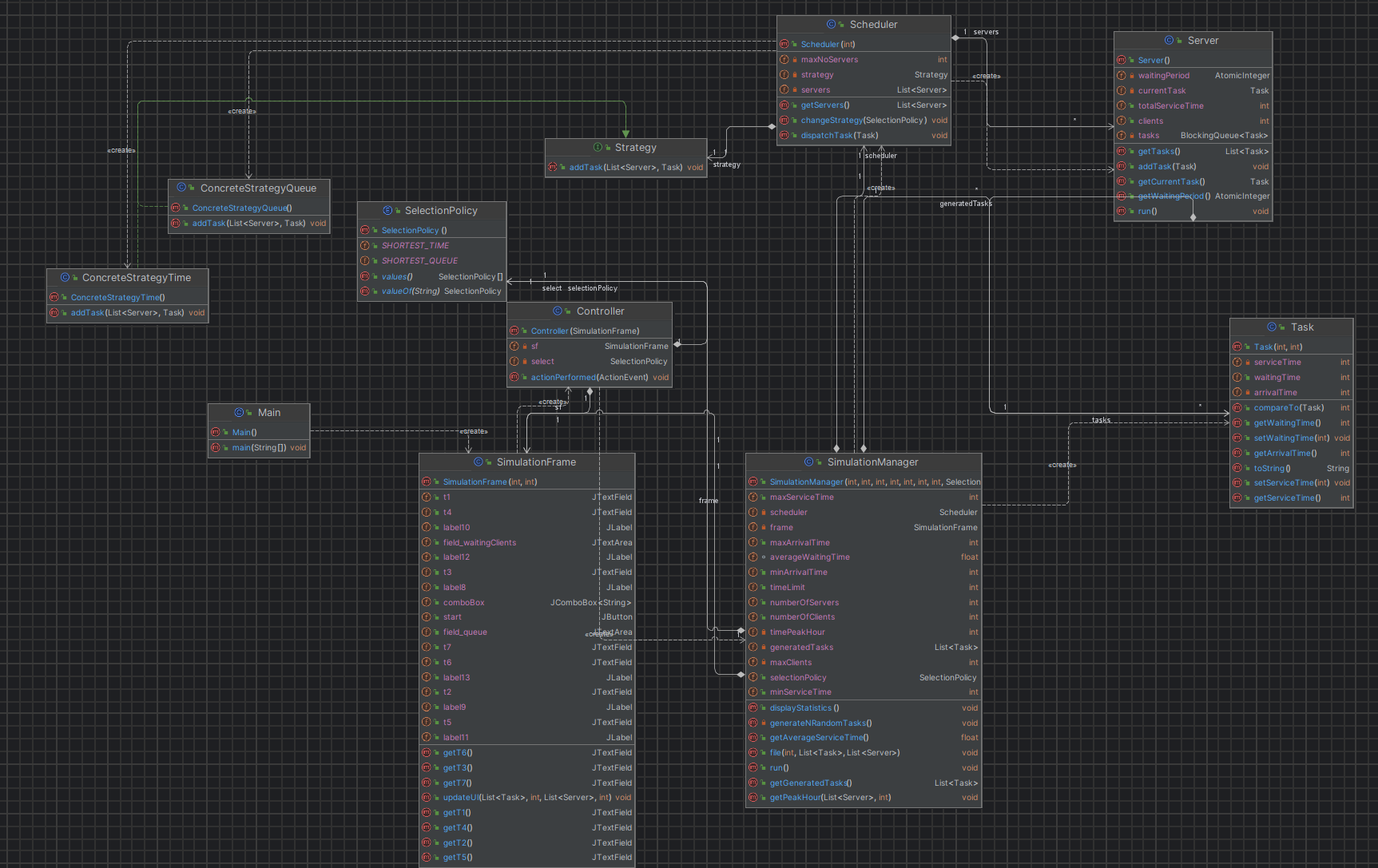
Pachetul **Logic** contine:

* În clasa **Scheduler** se vor inițializa și gestiona cozi în conformitate cu specificațiile utilizatorului. Pentru fiecare coadă creată, un fir de execuție (thread) va fi pornit. La încheierea simulării, aceste fire de execuție vor fi oprite corespunzător.
* Interfața **Strategy** definește comportamentul pentru adăugarea clienților în cozi, fie în funcție de numărul de clienți deja prezenți în coadă, fie în funcție de timpul estimat de finalizare pentru toți ceilalți clienți deja în așteptare.
* Clasele **ConcreteStrategyTime** și **ConcreteStrategyQueue** sunt clase care respectiv implementează interfața Strategy. Prima clasă adaugă clienții în funcție de timpul estimat, iar a doua în funcție de dimensiunea cozii.
* Enumeratia **SelectionPolicy** reprezinta cele 2 strategii.
* Pachetul **Simulation** contine:
* Clasa **SimulationManager:** responsabilă de gestionarea simulării și a interacțiunii dintre diferitele componente ale acesteia. Ea se ocupă de inițializarea simulării cu parametrii specificați de utilizator, generarea aleatoare a clienților și monitorizarea progresului simulării în timp real. Clasa coordonează și actualizează interfața grafică, gestionează despășirea sarcinilor la cozi și înregistrează datele relevante într-un fișier text. De asemenea, calculează și afișează statisticile finale ale simulării, cum ar fi timpul mediu de servire al unui client.
* Clasa **Controller** este responsabilă de gestionarea acțiunilor utilizatorului în cadrul interfeței grafice. Ea implementează interfața ActionListener pentru a recepta și procesa evenimentele de acțiune generate de utilizator. Atunci când utilizatorul apasă butonul de pornire a simulării, clasa Controller extrage parametrii specificați din interfață și inițializează o nouă instanță a clasei SimulationManager pentru a gestiona simularea. De asemenea, clasa monitorizează schimbările în strategia de selecție a coziilor și actualizează această selecție în funcție de opțiunile selectate de utilizator.
* Clasa **SimulationFrame** este o fereastră grafică care permite utilizatorului să configureze și să monitorizeze simularea. Ea conține componente pentru introducerea parametrilor simulării, cum ar fi numărul de clienți și de cozi, intervalul de simulare și timpii de sosire și servire. De asemenea, afișează în timp real evoluția cozilor și a clienților în așteptare într-un mod interactiv. Această clasă gestionează, de asemenea, interacțiunea cu clasa Controller pentru a recepta acțiunile utilizatorului, cum ar fi pornirea simulării și schimbarea strategiei de selecție a coziilor.
* Clasa **MainClass** declanseaza intrarea în aplicație, de unde se inițializează și se deschide simularea cozilor. Această clasă este responsabilă pentru gestionarea fluxului principal al aplicației și pornirea interfeței grafice.

Interfata grafica este urmatoarea:



Proiectul are urmatoarea diagrama UML:



# Implementare

Proiectul Gestionare Cozilor contine urmatoarele clase:

* Clasa **Task**:

Clasa **Task** din pachetul **model** reprezinta o sarcină(clientul) care trebuie efectuată în cadrul unei simulări. Descrierea clasei **Task**:

**Câmpuri:**

* arrivalTime: Reprezintă timpul de sosire al clientul.
* serviceTime: Reprezintă timpul necesar pentru a finaliza sarcina.

**Constructor:** public Task(int arrivalTime, int serviceTime): Constructorul clasei Task care primește două argumente, arrivalTime și serviceTime, și inițializează câmpurile corespunzătoare.

Clasa are următoarele metode:

* getArrivalTime(): Returnează timpul de sosire al clientului.
* getServiceTime(): Returnează timpul necesar pentru a finaliza sarcina.
* setServiceTime(int serviceTime): Setează timpul necesar pentru a finaliza sarcina la o valoare specificată.
* toString(): Returnează o reprezentare sub formă de șir a obiectului Task, incluzând timpul de sosire și timpul necesar pentru finalizare.
* compareTo(Task o): Implementarea metodei compareTo din interfața Comparable. Compară timpul de sosire al unui client cu timpul de sosire al unei alt client și returnează diferența dintre ele.
* Clasa **Server**:

Clasa **Server** din pachetul **model** reprezintă un server care e folosit pentru a gestiona cozile de client în cadrul unei simulări. Iată o descriere a clasei **Server**:

**Câmpuri:**

* + tasks: O coadă blocantă (BlockingQueue) de obiecte Task care reprezintă sarcinile sau activitățile ce trebuie gestionate de către server.
  + waitingPeriod: Un atom AtomicInteger care reprezintă perioada totală de așteptare a tuturor sarcinilor în coadă.
  + currentTask: Un obiect Task care reprezintă sarcina actuală pe care serverul o gestionează.

Clasa are următorii constructori:

**Constructor:**

* + public Server(): Inițializează câmpurile serverului. Coada de clienti este inițializată cu o coadă blocantă (LinkedBlockingQueue), iar perioada de așteptare este inițializată.

Clasa are următoarele metode: getteri si setteri.

* addTask(Task newTask): Adaugă o nou client în coada de sarcini a serverului. De asemenea, actualizează perioada totală de așteptare a clientilor.
* run(): Implementează metoda run() a interfeței Runnable, care reprezintă comportamentul cozii atunci când este executat. Coada continuă să preia clienti din coada sa și să le proceseze în buclă infinită. Procesarea unei client implică scăderea timpului de servire al acestuia în timp ce se simulează procesarea, actualizarea perioadei de așteptare și actualizarea statisticii legate de numărul total de clienți și timpul total de servire.
* getCurrentTask(): Returnează clientul curent pe care coada o gestionează.
* getWaitingPeriod(): Returnează perioada totală de așteptare a clientilor în coada.
* getTasks(): Returnează o listă imutabilă a clientilor din coada.
* Clasa **Scheduler**:

Clasa **Scheduler** din pachetul **logic** reprezintă un planificator care gestionează distribuirea clientilor către cozile de asteptare și schimbarea strategiei de selecție a cozilor. Descriere a clasei **Scheduler**:

* **Câmpuri**:
  + **servers**: O listă de obiecte **Server** care reprezintă cozile disponibile pentru procesare.
  + **maxNoServers**: Numărul maxim de cozi pe care le poate gestiona planificatorul.
  + **strategy**: O interfață **Strategy** care reprezintă strategia de selecție a cozilor. Aceasta poate fi una din două strategii concrete: **ConcreteStrategyQueue** pentru selectarea cozii cu cei mai putini clienti sau **ConcreteStrategyTime** pentru selectarea cozii cu cel mai mic timp de procesare.
  + **Constructor:** public Scheduler(int maxNoServers): Inițializează planificatorul cu numărul maxim de cozi. Creează și pornește fire de execuție pentru fiecare coada.
  + **Metode**:
  + changeStrategy(SelectionPolicy policy): Schimbă strategia pe baza politicii specificate (SHORTEST\_QUEUE sau SHORTEST\_TIME). Inițializează o nouă instanță a strategiei corespunzătoare și o asociază cu câmpul strategy.
  + dispatchTask(Task t): Distribuie o sarcină către unul dintre cozile disponibile utilizând strategia curentă.
  + getServers(): Returnează lista de cozi gestionate de planificator.

În esență, clasa **Scheduler** controlează distribuirea sarcinilor către servere și permite schimbarea strategiei de selecție a serverelor în funcție de politica specificată. Aceasta reprezintă un nivel de abstractizare între sarcinile primite și serverele disponibile, gestionând fluxul de sarcini și aplicând logica necesară pentru selectarea serverului potrivit pentru fiecare sarcină.

* **Interfata Strategy, Enum Selection Policy**:

Enumerația `SelectionPolicy` definește două politici posibile pentru selectarea serverului în cadrul planificatorului:

1. `SHORTEST\_QUEUE`: Această politică selectează scoada care are cea mai scurtă coadă cu clienti în momentul distribuirii unui nou client. Aceasta este o abordare eficientă pentru a minimiza timpul total de așteptare al clientilor.

2. `SHORTEST\_TIME`: Această politică selectează coada care are cel mai mic timp de procesare rămas pentru clienti. Cu alte cuvinte, se selectează cloada care se așteaptă să termine cu clientul în cel mai scurt timp posibil.

Interfața `Strategy` definește o metodă pentru adăugarea unei client la lista de clienti a cozilor, utilizând o anumită strategie. Aceasta permite implementarea diferitelor strategii de distribuire a clientilor către cozi, fără a fi nevoie să modificăm codul sursă al clasei `Scheduler`.

Metoda `addTask(List<Server> servers, Task t)` este implementată în clasele care implementează această interfață și definește logica specifică pentru distribuirea clientilor către cozi. Această metodă primește lista de cozi disponibile și clientul care trebuie distribuit și decide cum să distribuie clientul pe baza logicii strategiei respective. Implementările concrete ale acestei interfețe vor fi `ConcreteStrategyQueue` și `ConcreteStrategyTime`, fiecare implementând logica specifică a selecției cozii în conformitate cu politica corespunzătoare.

* **Clasa SimulationManager**:

Clasa **SimulationManager** din pachetul **simulation** este responsabilă pentru gestionarea și coordonarea simulării sistemului. Iată o descriere detaliată a clasei:

**Câmpuri:**

* timeLimit: Durata totală a simulării, specificată în unități de timp.
* maxArrivalTime, minArrivalTime: Timpul maxim și minim de sosire a clientilor.
* maxServiceTime, minServiceTime: Timpul maxim și minim necesar pentru a finaliza sarcina clientului.
* numberOfServers: Numărul total de cozi disponibile în simulare.
* numberOfClients: Numărul total de clienți (sarcini) care trebuie să fie procesați în cadrul simulării.
* selectionPolicy: Politica de selecție a serverului utilizată în cadrul simulării.
* scheduler: Un obiect Scheduler care gestionează distribuirea clientilor către cozi.
* frame: Un obiect SimulationFrame pentru actualizarea interfeței utilizatorului.
* generatedTasks: O listă de client generate aleatoriu pentru a fi procesate în cadrul simulării.
* totalWaitingTime: Timpul total de așteptare al clientilor în cadrul cozii.
* timePeakHour: Momentul (timpul) în care a avut loc vârful de activitate (numărul maxim de clienți) în cadrul simulării.
* maxClients: Numărul maxim de clienți observat în timpul simulării.

**Constructor:**

* public SimulationManager(...): Inițializează simularea cu parametrii specificați și creează un obiect Scheduler. Generează sarcinile pentru simulare și le sortează după timpul de sosire.

**Metode:**

* run(): Implementează metoda run() a interfeței Runnable. Coordonarea simulării: în fiecare unitate de timp, verifică sosirea noilor clienti, distribuie clientii către cozi, actualizează interfața utilizatorului și scrie datele de simulare într-un fișier text.
* file(...): Scrie datele de simulare într-un fișier text, inclusiv sarcinile aflate în așteptare și starea fiecărui server.
* displayStatistics(): Afișează statisticile finale ale simulării, inclusiv timpul mediu de așteptare, timpul mediu de servire și vârful de activitate.
* getAverageServiceTime(): Calculează timpul mediu de servire al clientilor în cadrul simulării.
* getAverageWaitingTime(): Calculează timpul mediu de așteptare al clientilor în cadrul simulării.
* getPeakHour(...): Identifică momentul (timpul) în care a avut loc vârful de activitate (numărul maxim de clienți) în cadrul simulării.
* **Clasa SimulationFrame**:

Clasa SimulationFrame este o fereastră grafică pentru interfața utilizatorului în cadrul simulării

**Câmpuri:**

* t1 până la t7: câmpuri de text pentru introducerea datelor de intrare ale simulării.
* label11, label12, label13, label10, label9, label8: etichete pentru afișarea informațiilor despre simulare.
* field\_waitingClients, field\_queue: zone de text pentru afișarea clienților care așteaptă și a stării cozilor.
* start: butonul de start pentru începerea simulării.
* comboBox: caseta de selecție pentru alegerea strategiei de simulare.
* generatedTasks: lista de sarcini generate pentru simulare.
* currentTime: timpul curent al simulării.
* scheduler: obiectul Scheduler folosit pentru gestionarea simulării.

**Constructor:**

* Inițializează interfața grafică, setând dimensiunile ferestrei și creând elementele UI necesare pentru introducerea datelor și afișarea informațiilor despre simulare.

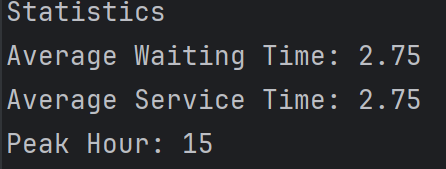
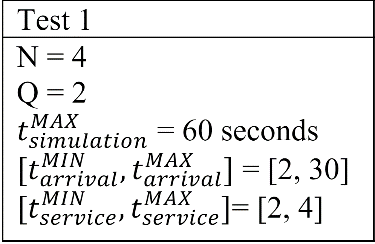
**Metode:**

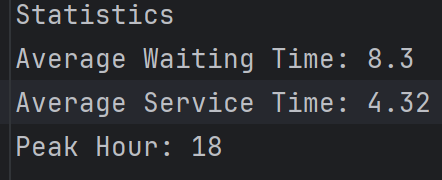
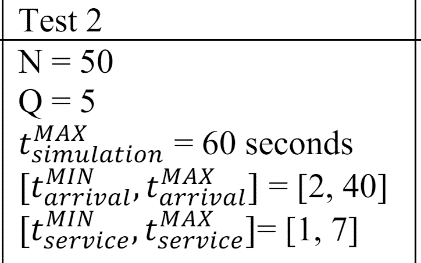
* updateUI(...): actualizează interfața utilizatorului cu informațiile actuale despre simulare, inclusiv timpul curent, clienții care așteaptă și starea cozilor.
* getT1() până la getT7(): metode pentru a obține referințele către câmpurile de text pentru a accesa valorile introduse de utilizator.
* setLabel10(...), setLabel9(...), setLabel8(...): metode pentru a seta etichetele din cadrul interfeței.
* setField\_waitingClients(...), setField\_queue(...): metode pentru a seta zonele de text din cadrul interfeței
* **Clasa Controller**:

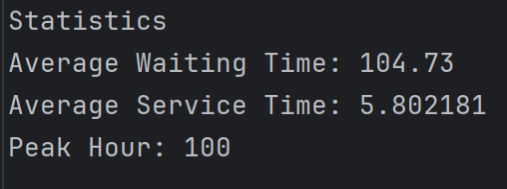
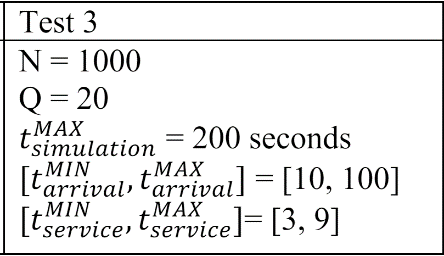
Clasa **Controller** servește ca un controler pentru interacțiunea utilizatorului cu interfața grafică a simulării:

* **Câmpuri**:
  + **sf**: Referință către obiectul **SimulationFrame**, adică fereastra grafică a simulării.
  + **select**: Politica de selecție a serverului selectată de utilizator.
* **Constructor**:
  + Inițializează controlerul cu un obiect **SimulationFrame** primit ca parametru.
* **Metode**:
  + **actionPerformed(ActionEvent e)**: Implementează metoda **actionPerformed** a interfeței **ActionListener**, care gestionează evenimentele de acțiune generate în interfața utilizatorului.
  + Dacă sursa acțiunii este butonul de start (**sf.start**), se extrag valorile introduse de utilizator din câmpurile de text și se creează un nou obiect **SimulationManager** cu aceste valori. Apoi, un fir de execuție separat este creat pentru a rula simularea.Dacă acțiunea este de a schimba selecția în combobox (**sf.comboBox**), se actualizează politica de selecție a serverului (**select**) pe baza selecției utilizatorului

# Rezultate







# Concluzii

*În încheierea acestui proiect, pot spune că inițial am întâmpinat unele dificultăți, dar cred că am reușit să rezolv problemele destul de bine. Am învățat cum să lucrez cu thread-urile în Java și cum să le gestionez corect, iar în plus, am descoperit lucruri noi despre Java Swing .*

*În ceea ce privește actualizările viitoare ale aplicației, mi-ar plăcea să îmbunătățesc interfata. În plus, intenționez să simplific și să curăț codul și să restructurez interfața utilizatorului pentru a o face mai receptivă și mai ușor de utilizat.*

Bibliografie

1. <https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>
2. <https://www.tutorialspoint.com/java/util/timer_schedule_period.htm>
3. <http://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-andthreadpoolexecutor.html>
4. [*https://dsrl.eu/courses/pt/*](https://dsrl.eu/courses/pt/)