DOCUMENTATIE

TEMA *2 QUEUES MANAGEMENT APPLICATION USING*

THREADS AND SYNCHRONIZATION MECHANISMS

NUME STUDENT: Ricu Alexandru Razvan
GRUPA: 30224

CUPRINS

1. Obiectivul temei

Scopul proiectului este dezvoltarea unei aplicații de gestionare a cozilor, care atribuie clienților cozi în așa fel încât timpul de așteptare să fie minimizat. Interfața grafică va fi intuitivă pentru utilizatori, facilitând accesul la funcționalități.

- *Procesul începe cu analiza nevoilor utilizatorilor și stabilirea cerințelor (se va detalia în capitolul: 2. Analiza problemei, modelare, scenarii, cazuri de utilizare).*
- *Urmează proiectarea detaliată a arhitecturii software și a interfeței. (se va detalia în capitolul: 3. Proiectare)*
- *Implementarea se concentrează pe scrierea codului și integrarea funcționalităților. (se va detalia în capitolul: 4. Implementare)*
- *Testarea este esențială pentru asigurarea corectitudinii și stabilității aplicației. (scenariile pentru testare vor fi prezentate în capitolul: 5. Concluzii)*

Scopul final este dezvoltarea unei aplicații robuste și ușor de utilizat pentru gestionarea eficientă a cozilor.

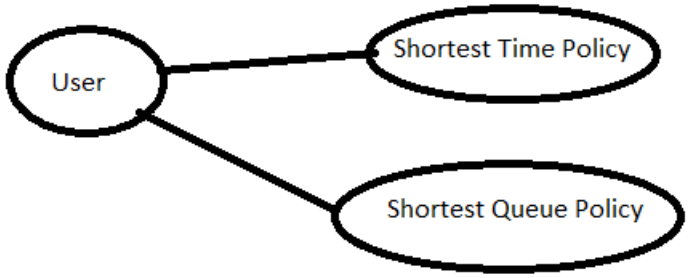
2. Analiza problemei, modelare, scenarii, cazuri de utilizare

Cerințe Funcționale:

- Shortest Time Policy: Utilizatorul poate analiza evolutia cozilor din punct de vedere al acestui policy.
- Shortest Queue Policy: Utilizatorul poate analiza evolutia cozilor din punct de vedere al acestui policy.
- Interfață Grafică: O interfață grafică intuitivă care permite utilizatorului să introducă infomratii despre simulare și să isi aleaga policy-ul dorit.
- Interfata Animata: O interfata grafica ce se modifica real-time si permite utilizatorului sa observe evolutia cozilor.

Cerințe Non-Funcționale:

- Performanță: Aplicația trebuie să fie rapidă și responsive la interacțiunile utilizatorului.
- Ușurință de Utilizare: Interfața grafică trebuie să fie prietenoasă și ușor de înțeles pentru utilizatori de toate nivelurile de experiență.
- Fiabilitate: Aplicația trebuie să fie robustă și să gestioneze corect diversele scenarii de utilizare.



Use-case-urile sunt prezentate mai jos, actorul principal al acestora fiind utilizatorul:

• Shortest Time Policy:

Scenariu de succes:

a) Utilizatorul introduce date legate de simulare în interfața grafică.

b) Utilizatorul selectează shortest time policy din drop-down menu si apasa butonul Start.

c) Se deschide un nou panel in care se poate observa evolutia real time a cozilor impreuna cu statistici cum ar fi Peak Hour, Avg Service Time, Avg Waiting Time.

• Shortest Length Policy:

Scenariu de succes:

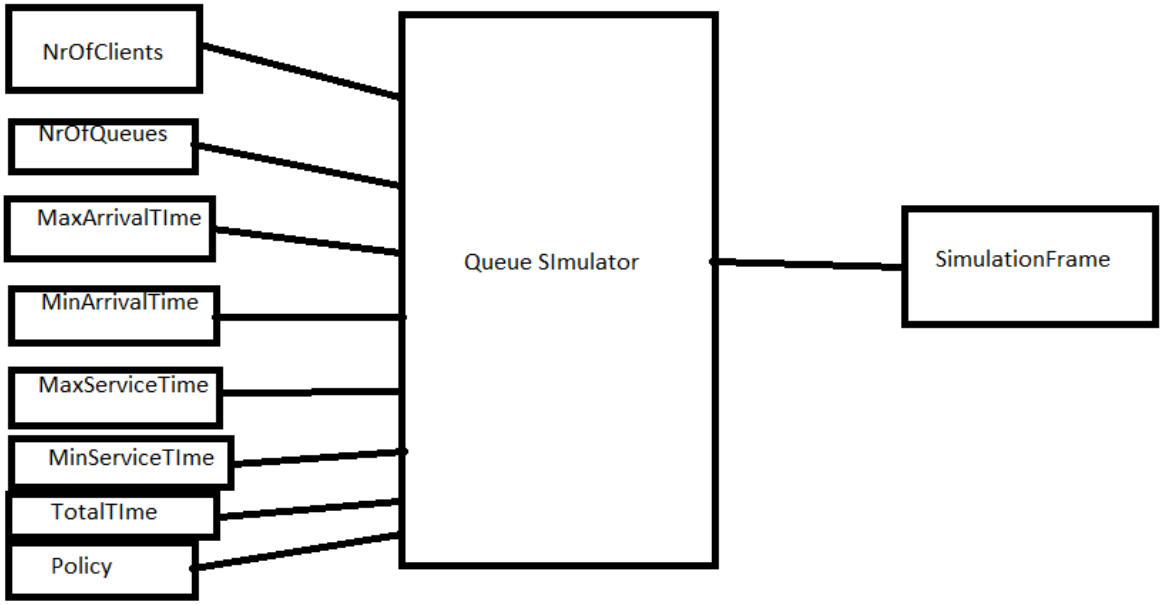
a) Utilizatorul introduce date legate de simulare în interfața grafică.

b) Utilizatorul selectează shortest length policy din drop-down menu si apasa butonul Start.

c) Se deschide un nou panel in care se poate observa evolutia real time a cozilor impreuna cu statistici cum ar fi Peak Hour, Avg Service Time, Avg Waiting Time.

Scenariul alternativ pentru toate use-case-urile: nu se introduc date valide simularii.

3. Proiectare



Pentru a implementa funcționalitățile calculatorului polinomial, avem nevoie de 8 intrări esențiale: parametrii simularii și tipul policy-ului (Shortest Queue sau Shortest Time).

Pentru a gestiona aceste aspecte, am ales să implementez un model arhitectural MVC hybrid. Acest model împarte aplicația în trei componente distincte:

- Model: Aici sunt încapsulate modelele datelor de bază, cum ar fi Task-ul si Server-ul. Clasa task reprezinta clientul din coada, iar clasa server reprezinta coada in sine si implementeaza interfata Runnable. Această structură permite gestionarea eficienta a clientilor si cozilor.

- GUI (Gui, Controller si SimulationFrame): Această componentă conține modulul care afișează informațiile utilizatorului și obține datele de la Controller pentru a le afișa, precum și un modul care, în funcție de input-ul primit din GUI, creeaza un nou panel unde se poate observa evolutia real time a cozilor in functie de policy-ul selectat(SimulationFrame).

- BusinessLogic (ConcreteStrategyQueue,ConcreteStrategyTime,Scheduler,SelectionPolicy,SimulationManager,Strategy): Clasa SimulationManager coordoneaza fiecare task in parte si il trimite unei cozi in functie de policy-ul selectat. Clasa SelectionPolicy este de tip enum pentru o mai buna vizibilitate a policy-ului selectat in cod. Clasa Scheduler porneste thread-urile pentru fiecare server in parte si in functie de policy-ul selectat, trimite task-ul catre un server anume. Pentru a ajunge la o abstractizare cât mai mare, ne folosim și de o interfață "Strategy" implementată de clasele ConcreteStrategyQueue,ConcreteStrategyTime pentru a implementa diferite modalitati de a adauga un task la serverul trimis ca parametru.

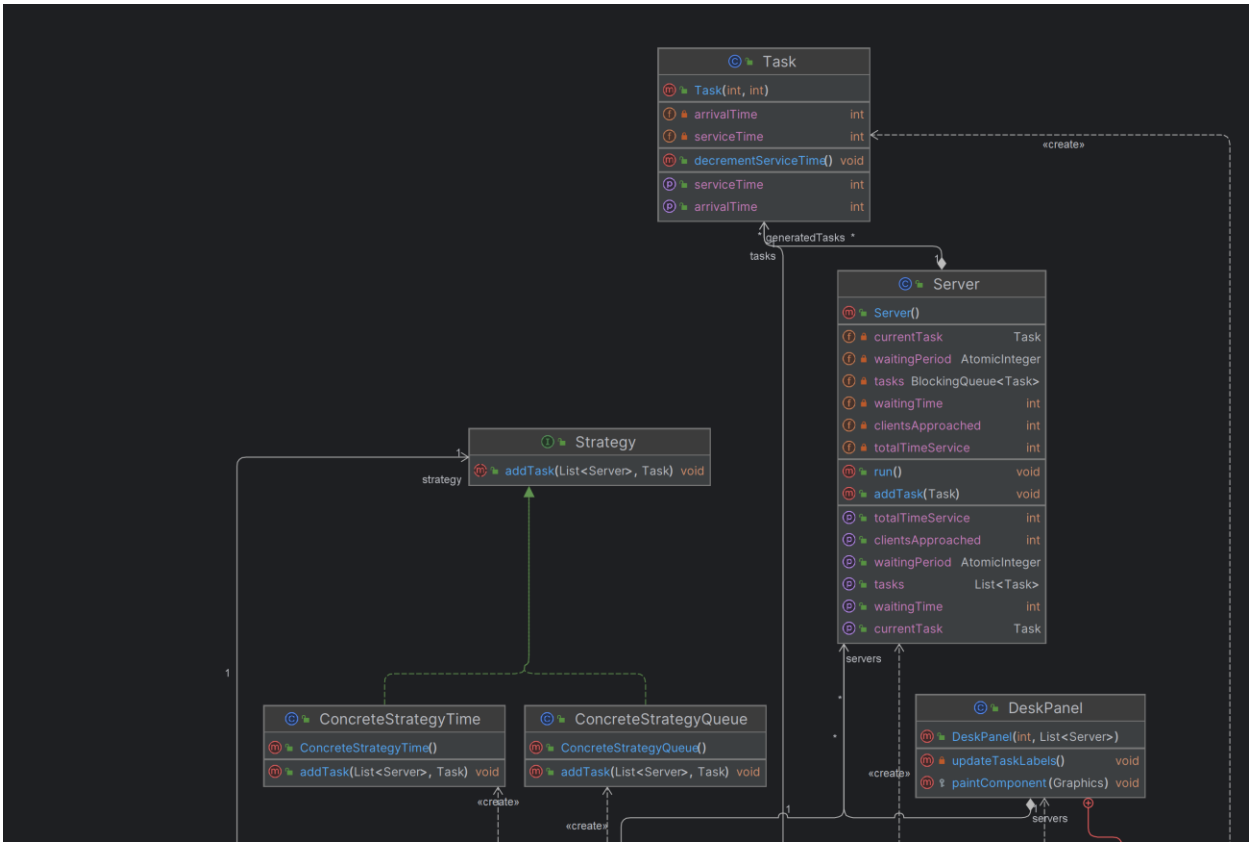
Prin implementarea acestui model arhitectural, putem gestiona eficient interacțiunea utilizatorului cu aplicația, separând clar business logic-ul (BusinessLogic) de interfața utilizatorului (Gui) și gestionând intrările utilizatorului prin intermediul Controlerului.

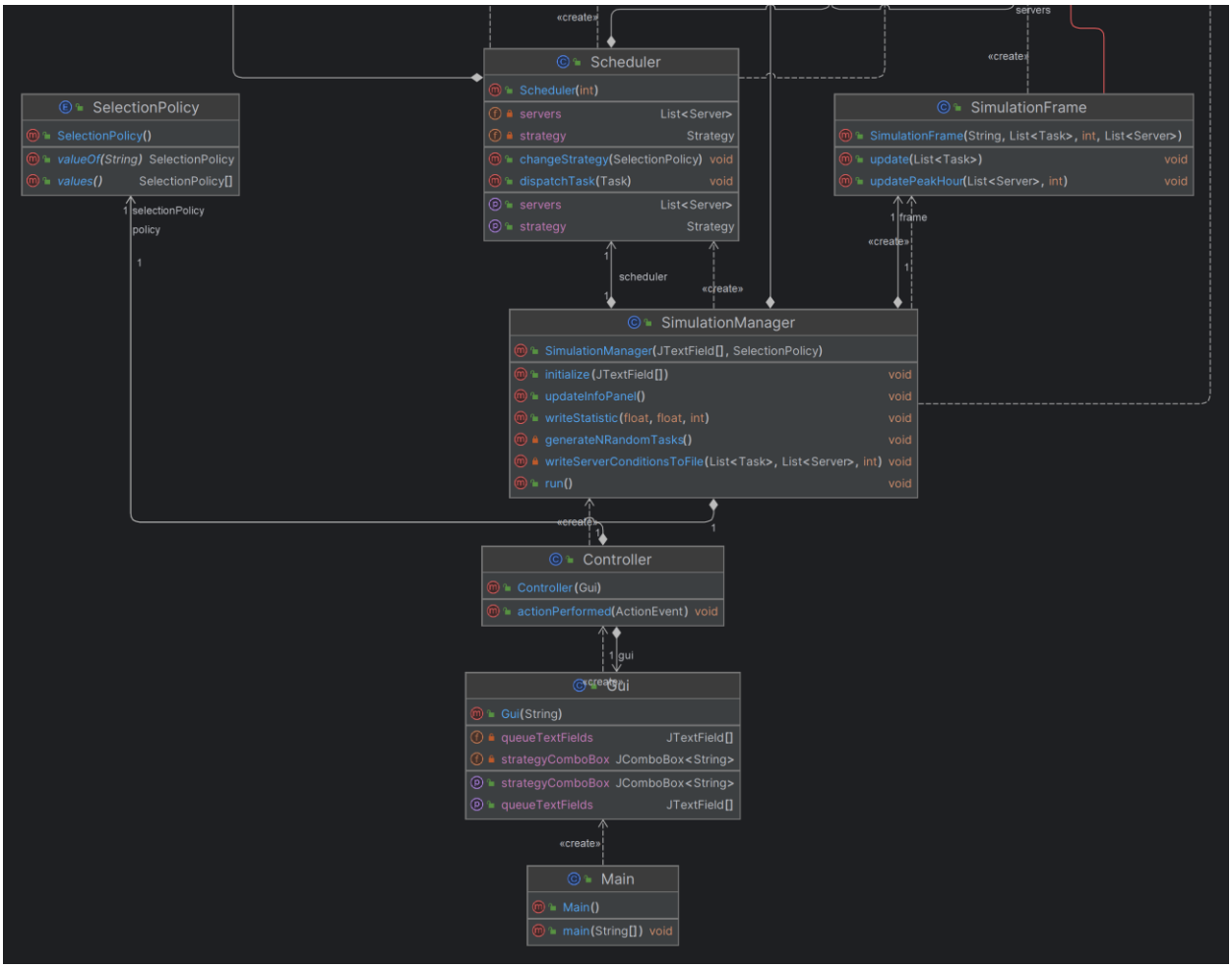
Pe lângă cele trei pachete menționate, am introdus și următoarea clasa:

• Clasa Main: În această clasă se găsește punctul de intrare în aplicație, de unde este deschis queue simulator-ul. Clasa Main este responsabilă pentru inițializarea și lansarea aplicației.

Structura de date pentru reprezentarea serverului este un blockingQueue (de tipul Task), fiecare Task având un arrivalTime si un serviceTime (de tip întreg), iar pentru reprezentarea cozilor din Scheduler este un List (de tipul Server).

Diagrama UML a proiectului este urmatoarea:





4. Implementare

Pentru a asigura o înțelegere completă a funcționalității Queue Simulator with Threads, voi descrie detaliat clasele menționate în secțiunile anterioare:

- Clasa Task ;

  Este responsabilă pentru reprezentarea unei sarcini în cadrul unui sistem de gestionare a cozi. Ea definește o serie de caracteristici și comportamente care descriu și controlează modul în care aceste sarcini sunt gestionate. Iată o prezentare a clasei Task:

  - Câmpuri:
    - arrivalTime: Reprezintă timpul la care sarcina a fost adăugată în sistem sau a sosit în coadă.
    - serviceTime: Indică timpul necesar pentru procesarea completă a sarcinii.

        - waitingTime: Stochează timpul total pe care sarcina îl petrece în așteptare înainte de a fi procesată.
    - Constructor:
        - public Task(int arrivalTime, int serviceTime): Inițializează un obiect Task cu timpul de sosire și timpul de serviciu specificate.
    - Metode:
        - public void decrementServiceTime(): Scade timpul de serviciu al sarcinii. Aceasta simulează progresul în procesarea sarcinii.
        - public void setWaitingTime(int waitingTime): Setează timpul de așteptare al sarcinii la o valoare specificată.
        - public int getArrivalTime(): Returnează timpul de sosire al sarcinii.
        - public int getServiceTime(): Returnează timpul de serviciu al sarcinii.
        - public int getWaitingTime(): Returnează timpul de așteptare al sarcinii.
- Clasa Server:

    Reprezintă un server sau un punct de serviciu în cadrul sistemului de cozi. Aceasta gestionează sarcinile primite și le procesează în ordinea sosirii lor. Iată o prezentare detaliată a clasei Server:
    - Câmpuri:
        - Tasks: Reprezintă o coadă blocantă de sarcini (Task) care urmează să fie procesate de către server.
        - waitingPeriod: Un contor atomic care monitorizează perioada totală de așteptare a sarcinilor în coadă.
        - clientsApproached: Numărul total de clienți care au fost abordați de către server.
        - clientsServed: Numărul total de clienți care au fost serviți de către server.
        - totalTimeService: Timpul total de serviciu oferit de către server.
        - finishedTasks: O listă care conține sarcinile care au fost finalizate de către server.
    - Constructor:
        - public Server(): Inițializează un obiect Server cu o coadă vidă de sarcini, un contor de perioadă de așteptare și alte câmpuri asociate.

- Metode:
  - public void addTask(Task newTask): Adaugă o sarcină nouă în coada de sarcini a serverului, actualizând în mod corespunzător contorul de perioadă de așteptare.
  - public void run(): Implementează logica principală a serverului. În bucla infinită, serverul preia sarcinile din coadă și le procesează până când programul este oprit.
  - public List<Task> getTasks(): Returnează o copie a listei de sarcini a serverului.
  - public AtomicInteger getWaitingPeriod(): Returnează contorul de perioadă de așteptare a serverului.
  - public Task getCurrentTask(): Returnează sarcina curentă care este procesată de către server.
  - public int getTotalTimeService(): Returnează timpul total de serviciu oferit de către server.
  - public int getClientsApproached(): Returnează numărul total de clienți care au fost abordați de către server.
  - public int getClientsServed(): Returnează numărul total de clienți care au fost serviți de către server.
  - public ArrayList<Task> getFinishedTasks(): Returnează lista de sarcini finalizate de către server.

- Clasa ConcreteStrategyQueue:

  Reprezintă o implementare specifică a unei strategii pentru adăugarea sarcinilor la servere în cadrul sistemului de cozi. Această clasă implementează interfața Strategy, care definește un set comun de metode pentru strategiile de gestionare a sarcinilor. Iată o prezentare a clasei ConcreteStrategyQueue:
  - Metode:
    - public void addTask(List<Server> servers, Task t): Această metodă implementează logica pentru adăugarea unei noi sarcini la unul dintre serverele disponibile. Ea primește o listă de servere (servers) și o sarcină (t) și determină serverul optim pentru a procesa sarcina.
    - Pentru fiecare server din lista dată, se verifică dacă serverul nu are deja o sarcină în curs de procesare (getCurrentTask() == null) și dacă coada de sarcini a serverului este goală

(getTasks().isEmpty()). În acest caz, sarcina este atribuită acestui server și metoda se încheie.
        - În caz contrar, se determină serverul cu cea mai mică dimensiune a cozii de sarcini. Sarcina este adăugată apoi la coada acestui server.

- Clasa ConcreteStrategyTime:

  Reprezintă o implementare specifică a unei strategii pentru adăugarea sarcinilor la servere în cadrul sistemului de cozi. Similar cu clasa ConcreteStrategyQueue, această clasă implementează interfața Strategy, care definește un set comun de metode pentru strategiile de gestionare a sarcinilor. Iată o prezentare a clasei ConcreteStrategyTime:
  - Metode:
    - public void addTask(List<Server> servers, Task t): Această metodă implementează logica pentru adăugarea unei noi sarcini la unul dintre serverele disponibile, bazată pe timpul minim de așteptare. Ea primește o listă de servere (servers) și o sarcină (t) și determină serverul cu cel mai mic timp de așteptare pentru a procesa sarcina.
    - Pentru fiecare server din lista dată, se determină timpul minim de așteptare (min) din toate serverele.
    - Apoi, se caută serverul care are timpul de așteptare egal cu min și se adaugă sarcina la coada acestui server.
- Clasa Scheduler:

  Reprezintă un program de planificare care gestionează serverele și sarcinile în cadrul sistemului de cozi. Aceasta coordonează procesul de asignare a sarcinilor la servere utilizând diverse strategii de planificare. Iată o prezentare a clasei Scheduler:
  - Câmpuri:
    - servers: O listă de servere disponibile în sistem.
    - maxNoServers: Numărul maxim de servere pe care le poate gestiona scheduler-ul.
    - strategy: Strategia de planificare utilizată pentru a alege serverul și a distribui sarcinile.
  - Constructor:
    - public Scheduler(int maxNoServers): Inițializează scheduler-ul cu numărul maxim de servere specificat. Creează serverele corespunzătoare și le pornește în fire de execuție.

- Metode:
  - public void changeStrategy(SelectionPolicy policy): Schimbă strategia de planificare în funcție de politica de selecție dată (SHORTEST_QUEUE sau SHORTEST_TIME). Creează o nouă instanță a strategiei corespunzătoare și o asociază scheduler-ului.
  - public void dispatchTask(Task t): Desemnează o sarcină pentru a fi gestionată de către strategia curentă a scheduler-ului. Strategia este responsabilă pentru alegerea serverului adecvat și adăugarea sarcinii în coada acestuia.
  - public List<Server> getServers(): Returnează lista de servere gestionate de către scheduler.
  - public Strategy getStrategy(): Returnează strategia de planificare curentă a scheduler-ului.

- Enumerarea SelectionPolicy definește politica de selecție disponibilă pentru scheduler. Aceasta permite specificarea modului în care sunt selectate serverele pentru gestionarea sarcinilor în cadrul sistemului de cozi. Iată o prezentare succintă a enumerației SelectionPolicy:
  - Valori:
    - SHORTEST_QUEUE: Această valoare indică faptul că serverul cu cea mai scurtă coadă de sarcini ar trebui să fie selectat pentru procesarea unei noi sarcini.
    - SHORTEST_TIME: Această valoare indică faptul că serverul cu cel mai mic timp de așteptare ar trebui să fie selectat pentru procesarea unei noi sarcini.
- Clasa SimulationManager

  Reprezintă un manager de simulare care coordonează și controlează procesul de simulare a unui sistem de cozi. Aceasta gestionează crearea sarcinilor, planificarea serverelor, actualizarea interfeței grafice și scrierea datelor de simulare într-un fișier. Iată o prezentare detaliată a clasei SimulationManager:
  - Câmpuri:
    - TimeLimit: Limita de timp pentru simulare.
    - minArrivalTime, maxArrivalTime: Timpul minim și maxim de sosire al sarcinilor.
    - minProcessingTime, maxProcessingTime: Timpul minim și maxim de procesare al sarcinilor.

        - numberOfServers: Numărul de servere din sistem.
        - numberOfClients: Numărul total de clienți (sarcini) care vor fi generați pentru simulare.
        - selectionPolicy: Politica de selecție a serverului pentru gestionarea sarcinilor.
        - scheduler: Un obiect de tip Scheduler care gestionează serverele și strategia de planificare.
        - frame: O instanță a interfeței grafice de simulare.
        - generatedTasks: O listă de sarcini generate pentru simulare.
    - Constructor:
        - Public SimulationManager(JTextField[] textFields,SelectionPolicy policy): Inițializează managerul de simulare cu valorile specificate în câmpurile din câmpurile de text și politica de selecție dată. Inițializează scheduler-ul și interfața grafică de simulare.
    - Metode:
        - private void generateNRandomTasks(): Generează un număr specificat de sarcini aleatorii cu intervale de sosire și timp de procesare specificate. Sarcinile sunt sortate în funcție de timpul de sosire.
        - public void run(): Implementează logica principală a simulării. Procesează sarcinile în funcție de timpul de sosire și actualizează starea interfeței grafice în timp real.
        - public void initialize(JTextField[] input): Inițializează câmpurile managerului de simulare cu valorile specificate în câmpurile de text.
        - private void writeServerConditionsToFile(List<Task> generatedTasks, List<Server> servers, int time): Scrie informații despre starea serverelor și sarcinilor într-un fișier text.
        - public void updateInfoPanel(): Actualizează panoul de informații cu statisticile finale ale simulării.
        - public void writeStatistic(float avgServ,float avgWait,int peakHr): Scrie statisticile finale ale simulării într-un fișier text.
- Interfața Strategy

    Definește un set comun de metode pentru diferite strategii de gestionare a sarcinilor în cadrul unui sistem de cozi. Această interfață este

implementată de către clasele care definesc strategii specifice pentru distribuirea sarcinilor către servere. Iată o prezentare a interfeței Strategy:

- Metode:
  - void addTask(List<Server> servers, Task t): Această metodă este responsabilă pentru adăugarea unei noi sarcini (Task) la unul sau mai multe servere (Server) din sistemul de cozi. Implementarea acestei metode poate varia în funcție de strategia specifică utilizată (de exemplu, adăugarea sarcinii la serverul cu cea mai scurtă coadă sau la serverul cu cel mai mic timp de așteptare).

- Clasa Controller

Acționează ca un controler în cadrul aplicației GUI și gestionează evenimentele utilizatorului, precum acțiunile de clic pe butoane sau selecția unui element dintr-un meniu derulant. Iată o prezentare a clasei Controller:

- Câmpuri:
  - gui: O referință către obiectul Gui asociat, care reprezintă interfața grafică a aplicației.
- Constructor:
  - public Controller(Gui gui): Inițializează câmpul gui cu referința la obiectul Gui asociat.
- Metode:
  - public void actionPerformed(ActionEvent e): Implementează logica pentru gestionarea evenimentelor de acțiune. În funcție de comanda acțiunii, această metodă va efectua diferite acțiuni. Dacă comanda acțiunii este "Start", se inițializează și se pornește o nouă simulare utilizând SimulationManager, pe baza politicii de selecție specificată și a parametrilor introdusi de utilizator. Dacă comanda acțiunii este "Switch strategies", se obține strategia selectată din meniul derulant al GUI-ului și se actualizează politica de selecție în conformitate cu strategia selectată.

- Clasa Gui

Reprezintă interfața grafică a aplicației și este responsabilă pentru crearea și gestionarea componentelor GUI, precum câmpuri de text, butoane și meniuri derulante. Iată o prezentare a clasei Gui:

- Câmpuri:

        - queueTextFields: Un vector de câmpuri de text utilizat pentru introducerea parametrilor de simulare.
        - strategyComboBox: Un meniu derulant pentru selectarea strategiei de simulare.
    - Constructor:
        - public Gui(String title): Inițializează interfața grafică cu titlul specificat. Creează și plasează componentele GUI pe fereastra principală.
    - Metode:
        - public JTextField[] getQueueTextFields(): Returnează vectorul de câmpuri de text pentru parametrii de simulare.
        - public JComboBox<String> getStrategyComboBox(): Returnează meniul derulant pentru selecția strategiei de simulare.

- Clasa SimulationFrame

  Este responsabilă pentru afișarea și actualizarea interfeței grafice a simulării. Aceasta include informații despre starea curentă a simulării, precum și vizualizarea cozilor de așteptare și a activității serverelor. Iată o prezentare a clasei:
    - Câmpuri:
        - contentPane: Panoul principal al ferestrei, care conține toate celelalte componente.
        - information: Panoul care afișează informații despre simulare, cum ar fi timpul trecut, ora de vârf, timpul mediu de servire și timpul mediu de așteptare.
        - deskPanel: Panoul care afișează vizualizarea cozilor de așteptare și a activității serverelor.
        - LeftPanel: Panoul din stânga care afișează lista de sarcini generate.
        - timePassedField, peakHourField, avgServiceField, avgWaitingField: Câmpuri de text pentru afișarea informațiilor despre simulare.
    - Constructor:
        - public SimulationFrame(String title, List<Task> generatedTasks, int numberOfServers,List<Server> servers): Inițializează fereastra de simulare cu titlul specificat și parametrii de intrare. Afișează lista de sarcini generate și panoul

de informații, precum și vizualizarea cozilor de așteptare și a activității serverelor.

- Metode:
  - public void update(List<Task> generatedTasks): Actualizează afișarea listei de sarcini generate și panoul de informații.
  - public void updatePeakHour(List<Server>servers, int time): Actualizează ora de vârf în funcție de numărul maxim de clienți observat până în acel moment.
  - public class DeskPanel extends JPanel: Clasa internă care reprezintă panoul pentru afișarea cozilor de așteptare și a activității serverelor. Aceasta se ocupă de desenarea grafică a cozilor și a sarcinilor procesate de fiecare server.

- Clasa DeskPanel

Este o clasă internă a clasei SimulationFrame și reprezintă panoul responsabil pentru afișarea vizuală a cozilor de așteptare și a activității serverelor în cadrul simulării. Iată mai multe detalii despre această clasă:

- Câmpuri:
  - taskLabels: O listă de etichete pentru afișarea informațiilor despre sarcinile procesate de fiecare server.
  - servers: O listă de servere pentru care se afișează activitatea.
- Constructor:
  - public DeskPanel(int numberOfServers, List<Server> servers): Inițializează panoul cu numărul specificat de servere și lista de servere pentru care se afișează activitatea.
- Metode:
  - public void updateTaskLabels(): Actualizează lista de etichete pentru a reflecta ultima stare a sarcinilor procesate de fiecare server.
  - @Override protected void paintComponent(Graphics g): Suprascrie metoda paintComponent pentru a desena grafic cozile de așteptare și activitatea serverelor pe panou. Această metodă se ocupă de desenarea dreptunghiurilor colorate care reprezintă sarcinile și cozile de așteptare pentru fiecare server. De asemenea, afișează informațiile despre fiecare sarcină utilizând etichetele definite în câmpul taskLabels.

5. Rezultate

Testarea funcționalităților proiectului a fost efectuată folosind 3 seturi de date predefintie pentru simulare, fiecare rezultat fiind scris in fisier, la finalul fiecarei simulari. Cele trei cazuri de testare vor fi prezentate mai jos:

1) N = 4
Q = 2
tsimulationMAX = 60 seconds
[tarrivalMIN ,tarrivalMAX] = [2, 30]
[tserviceMIN ,tserviceMAX]= [2, 4]

```
Peak hour: 5
Average Service Time: 3.0
Average Waiting Time: 0.75
```

2) *N = 50*
Q = 5
tsimulationMAX = 60 seconds
[tarrivalMIN ,tarrivalMAX] = [2, 40]
[tserviceMIN ,tserviceMAX]= [1, 7]

```
Peak hour: 39
Average Service Time: 4.14
Average Waiting Time: 2.12
```

3)) N = 1000
Q = 20
tsimulationMAX = 200 seconds
[tarrivalMIN ,tarrivalMAX] = [10, 100]
[tserviceMIN ,tserviceMAX]= [3, 9]

```
Peak hour: 100
Average Service Time: 5.9492755
Average Waiting Time: 60.51831
```

6. Concluzii

Experiența acumulată în timpul implementării acestui proiect a adus la lumină câteva concluzii și idei relevante, care pot fi de folos în viitoarele proiecte:

- *Structurarea obiectivelor în sub-obiective a fost esențială pentru progresul eficient și gestionarea adecvată a proiectului. Împărțirea obiectivelor mari în etape mai mici și mai ușor de gestionat ne-a permis să ne concentrăm pe realizarea pașilor necesari pentru atingerea obiectivului final.*

- *Alegerea structurilor de date potrivite a avut un impact semnificativ asupra performanței și eficienței aplicației. Utilizarea adecvată a structurilor de date, cum ar fi BlockingQueue pentru gestionarea cozilor, a dus la o gestionare mai eficientă a sarcinilor și a redus timpul de așteptare al clienților.*

- *Abstractizarea codului prin intermediul interfețelor și claselor abstracte a permis o dezvoltare mai flexibilă și extensibilă a aplicației. Definirea unor interfețe clare pentru operațiile de bază și utilizarea claselor abstracte pentru a implementa comportamente comune au facilitat integrarea și extinderea funcționalităților fără a afecta structura existentă a codului.*

- *Lizibilitatea codului a fost un aspect crucial în dezvoltarea și întreținerea aplicației. Menținerea unui cod curat și lizibil prin limitarea dimensiunii metodelor și claselor a fost esențială pentru înțelegerea și gestionarea eficientă a codului pe parcursul proiectului.*

- *Gestionarea erorilor și feedback-ul utilizatorului au fost aspecte importante care ar putea fi îmbunătățite în dezvoltarea ulterioară a aplicației. Implementarea unui sistem robust de gestionare a erorilor și furnizarea unui feedback clar și util utilizatorului vor contribui la îmbunătățirea experienței generale a utilizatorului și la creșterea fiabilității aplicației.*

7. Bibliografie

1. https://dsrl.eu/courses/pt/
2. https://docs.oracle.com/javase%2F7%2Fdocs%2Fapi%2F%2F/java/util/concurrent/BlockingQueue.html