DOCUMENTATIE

TEMA 2

NUME STUDENT: FANGLI DARIA

GRUPA: 30225

# CUPRINS

[1. Obiectivul temei 3](#_Toc95297885)

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

[3. Proiectare 6](#_Toc95297887)

[4. Implementare 7](#_Toc95297888)

[5. Rezultate 11](#_Toc95297889)

[6. Concluzii 12](#_Toc95297890)

[7. Bibliografie 12](#_Toc95297891)

# Obiectivul temei

Proiectul presupune dezvoltarea unei aplicații Java pentru administrarea cozilor, având ca bază datele de intrare precum numărul de cozi, numărul total de clienți care trebuie deserviți, și durata maximă a simulării. De asemenea, sunt furnizate și intervalele de timp în care clienții sunt pregătiți să intre în coadă și durata de servire a unui client.

Scopul proiectului este să genereze aleator clienți, având în vedere intervalele specificate, și să îi direcționeze către coada cu cel mai mic timp de așteptare. Pentru a asigura eficiența, fiecare coadă are alocat un fir de execuție propriu, permițându-le să funcționeze independent.

O atenție deosebită este acordată gestionării eficiente a cozilor, astfel încât clienții să fie serviți în mod echitabil și rapid, respectând prioritățile și minimizând timpii de așteptare.

În cadrul proiectului, sunt stabilite următoarele sub-obiective:

Proiectarea sistemului: Acest capitol va descrie structura generală a proiectului, relațiile dintre componentele sale și modul în care acestea interacționează între ele.

Implementarea funcționalităților: Aici se va detalia modul în care sunt implementate clasele și modulele proiectului. Vor fi prezentate informații despre clase, metode și variabile, explicându-se modul în care acestea îndeplinesc cerințele proiectului.

Testarea și validarea: Acest capitol se va concentra pe procesul de testare a aplicației, cu scopul de a verifica funcționalitatea corectă și conformitatea cu specificațiile. Vor fi elaborate planuri de testare și vor fi prezentate rezultatele testelor efectuate pentru validarea funcționalității și corectitudinii aplicației.

# Analiza problemei, modelare, scenarii, cazuri de utilizare

Cerințele funcționale ale proiectului includ:

Configurarea simulării: Utilizatorul trebuie să poată ajusta parametrii simulării, cum ar fi numărul de cozi, numărul de clienți, și durata maximă a simulării, înainte de a porni procesul de simulare.

Inițierea simulării: Utilizatorul trebuie să poată începe simularea după ce a configurat parametrii, activând astfel procesul de gestionare a cozilor.

Monitorizarea în timp real: Aplicația trebuie să ofere o interfață grafică sau textuală care să prezinte în timp real evoluția stării cozilor pe parcursul simulării, astfel încât utilizatorul să poată urmări cum se schimbă dinamica și volumul cozilor.

Cerințele non-funcționale ale proiectului includ:

Ușurința în utilizare: Aplicația trebuie să ofere o experiență intuitivă și simplă pentru utilizatori, permițându-le să configureze și să inițieze simularea fără a întâmpina dificultăți sau a avea nevoie de instrucțiuni complexe. Interfața utilizator trebuie să fie accesibilă și prietenoasă, facilitând interacțiunea și navigarea.

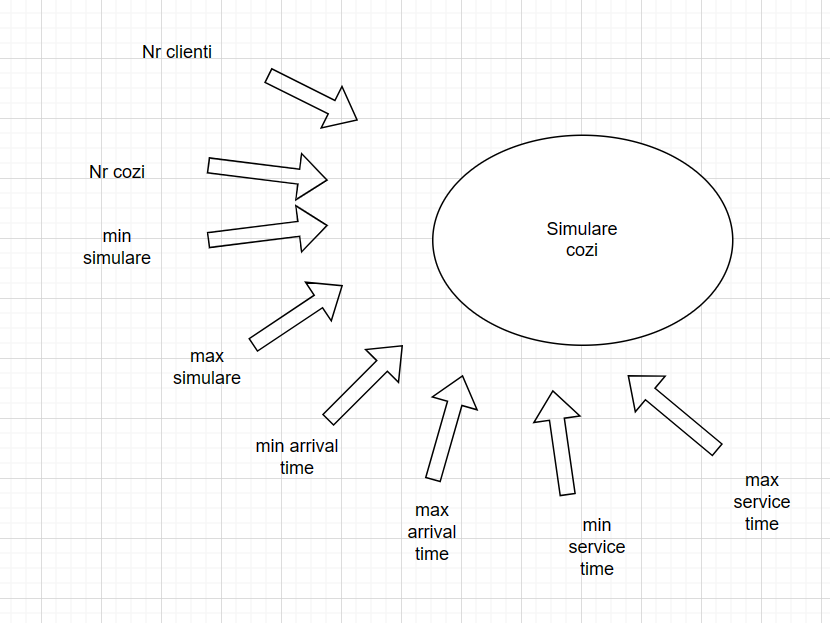
**Scenariu de utilizare**: Configurare simulare

Actor principal: Utilizatorul

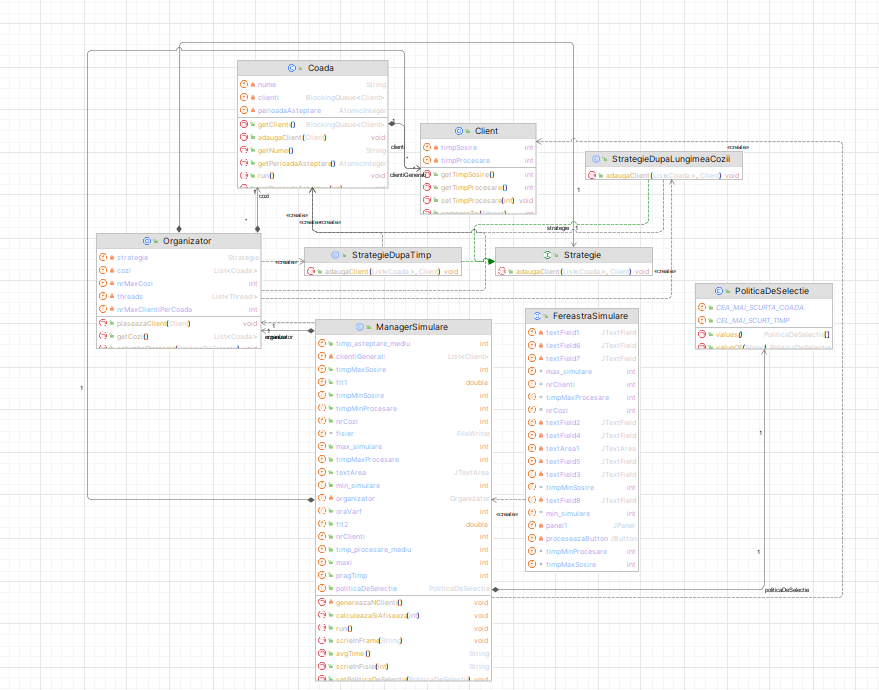
Scenariu principal de succes:

1. Utilizatorul introduce în interfața grafică detalii precum numărul de clienți, numărul de cozi, intervalul de simulare, durata minimă și maximă de servire, și intervalul minim și maxim de așteptare.
2. Sistemul afișează rezultatele pe ecran, prezentând clienții și simulând cozile de așteptare conform configurației introduse.

Această descriere prezintă modul în care utilizatorul interacționează cu sistemul pentru a configura o simulare, precum și rezultatul acestei acțiuni, inclusiv afișarea rezultatelor pe ecran și simularea efectivă a cozilor de așteptare.



# Proiectare



În proiect există 3 pachete:

* Pachetul org.example unde este implementat codul pentru gestionarea cozilor
* Pachetul GUI unde este implementat codul pentru interfața grafică
* Pachetul model unde sunt instantiate obiectele de lucru, clientul si coada.

Am folosit LinkedBlockingQueue, un tip de lista (coada) care este thread-safe.

Am folosit AtomicInteger, un tip de data care este, de asemenea, thread safe

# Implementare*.*

**Clasa Client** din pachetul org.example:

1. **Atribute:**
   * **timpSosire**: Reprezintă momentul de timp la care clientul sosește în sistem.
   * **timpProcesare**: Indică durata de timp necesară pentru a servi acest client în cadrul sistemului.
2. **Constructor:**
   * Clasa are un constructor care primește ca parametri timpul de sosire și timpul de procesare al clientului și îi atribuie valorile corespunzătoare atributelor.
3. **Metode:**
   * **getTimpSosire()**: Returnează timpul de sosire al clientului.
   * **getTimpProcesare()**: Returnează timpul de procesare al clientului.
   * **setTimpProcesare(int timpProcesare)**: Permite actualizarea timpului de procesare al clientului.
   * **compareTo(Object o)**: Implementează interfața **Comparable** pentru a permite compararea între obiecte de tip **Client** pe baza timpului de sosire. Metoda **compareTo** compară două obiecte **Client** și returnează 0 dacă timpul de sosire al obiectului curent este mai mic sau egal cu timpul de sosire al obiectului dat ca parametru, altfel returnează -1.

**Clasa Coada** din pachetul org.example

1. **Atribute:**
   * **clienti**: Este o coadă blocantă (**BlockingQueue**) care conține obiecte de tip **Client**, reprezentând clienții așteptând să fie serviți în coadă.
   * **perioadaAsteptare**: Este un obiect de tip **AtomicInteger** care reprezintă perioada totală de așteptare a clienților în coadă.
   * **nume**: Numele cozi, pentru identificare.
2. **Constructor:**
   * Clasa are un constructor care primește ca parametru numele cozii și inițializează coada de clienți și perioada de așteptare.
3. **Metode:**
   * **adaugaClient(Client clientNou)**: Adaugă un client nou în coadă.
   * **run()**: Implementează interfața **Runnable** pentru a permite executarea cozi ca un fir de execuție. În cadrul acestei metode, coada este verificată în mod continuu pentru clienți, iar aceștia sunt serviți conform timpului de procesare. Pe măsură ce clienții sunt serviți, timpul de procesare este decrementat, iar clienții sunt eliminați din coadă atunci când sunt serviți complet.

**Clasa StrategieDupaTimp** din pachetul org.example

1. **Metoda adaugaClient(List<Coada> cozi, Client c):**
   * Această metodă este suprascrisă din interfața **Strategie** și implementează logica de adăugare a unui client (**Client c**) într-una din cozile disponibile (**List<Coada> cozi**), utilizând o strategie bazată pe timpul de așteptare estimat în fiecare coadă.
   * Variabila **time\_max** este inițializată cu o valoare mare pentru a reprezenta timpul maxim de așteptare posibil.
   * Variabila **coada\_min** este inițializată cu o coadă goală, pentru a reține coada cu timpul minim de așteptare.
   * Se parcurg toate cozile din lista **cozi**, iar pentru fiecare coadă, se verifică dacă perioada de așteptare estimată (**coada.getPerioadaAsteptare().intValue()**) este mai mică decât **time\_max**. Dacă este, coada minimă (**coada\_min**) este actualizată cu coada curentă, iar **time\_max** este actualizat cu perioada de așteptare a cozi minime.
   * După parcurgerea tuturor cozilor, clientul este adăugat în coada minimă găsită (**coada\_min**).
   * Apoi, perioada de așteptare estimată a cozi minime este actualizată prin adăugarea timpului de procesare al clientului adăugat.

Clasa **Organizator** din pachetul org.example

1. **Constructorul Organizator(int nrMaxCozi, int nrMaxClientiPerCoada):**
   * Constructorul primește numărul maxim de cozi (**nrMaxCozi**) și numărul maxim de clienți per coadă (**nrMaxClientiPerCoada**).
   * Inițializează o strategie implicită, în acest caz **StrategieDupaTimp**.
   * Creează lista de cozi (**cozi**) și pentru fiecare coadă, creează un obiect de tip **Coada**, îi atribuie un nume și îl pornește într-un fir de execuție separat folosind clasa **Thread**.
2. **Metoda schimbaStrategia(PoliticaDeSelectie politica):**
   * Această metodă permite schimbarea strategiei folosite pentru plasarea clienților în cozi, în funcție de o anumită politică (**PoliticaDeSelectie**).
   * Dacă se specifică politica **CEL\_MAI\_SCURT\_TIMP**, se setează strategia la **StrategieDupaTimp**, iar dacă se specifică **CEA\_MAI\_SCURTA\_COADA**, se setează strategia la **StrategieDupaLungimeaCozii**.
3. **Metoda plaseazaClient(Client c):**
   * Această metodă plasează un client (**Client c**) în una dintre cozi, utilizând strategia definită.
   * Dacă strategia este **StrategieDupaTimp**, clientul este plasat în coada cu cel mai mic timp estimat de așteptare.
   * Dacă strategia este **StrategieDupaLungimeaCozii**, clientul este plasat în coada care are cea mai mică lungime.
4. **Metoda getCozi():**
   * Această metodă returnează lista de cozi gestionate de organizator.

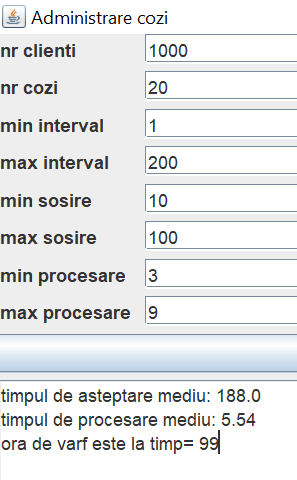
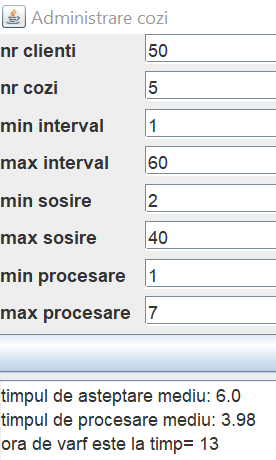
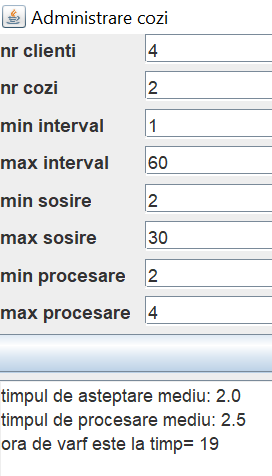
Clasa **FereastraSimulare**

1. **Constructorul FereastraSimulare():**
   * Inițializează interacțiunea cu componentele interfeței grafice, cum ar fi câmpurile de introducere a datelor și butonul de procesare.
   * Atașează ascultători la câmpurile de introducere pentru a captura valorile introduse de utilizator.
2. **Metoda main(String[] args):**
   * Lansează aplicația, creând și afișând fereastra principală.
3. **Acțiunea butonului proceseazaButton (metoda actionPerformed(ActionEvent e)):**
   * Captură valorile introduse de utilizator pentru numărul de clienți, numărul de cozi și alte parametri relevanți pentru simulare.
   * Creează un obiect de tip **ManagerSimulare** cu parametrii introdusi și obiectul **textArea1** pentru afișarea rezultatelor.
   * Pornește un fir de execuție separat pentru simularea gestionată de **ManagerSimulare**.

Clasa **ManagerSimulare**

1. **Atribuții și inițializare:**
   * Variabilele de instanță sunt inițializate, inclusiv lista de clienți, politica de selecție și obiectul **Organizator** responsabil cu gestionarea cozilor.
2. **Constructor:**
   * Primește parametrii necesari pentru configurarea simulării și inițializează obiectul.
   * Generează clienții cu timpuri de sosire și procesare aleatorii în limitele specificate și calculează media timpului de procesare.
3. **Metoda run():**
   * Realizează simularea prin iterarea peste intervalul de timp specificat.
   * La fiecare iterație, plasează clienții în cozi în funcție de timpul de sosire și politica de selecție.
   * Calculează și afișează statistici relevante, cum ar fi timpul mediu de așteptare și de procesare.
4. **Metoda scrieInFisier(int timp):**
   * Construiește un șir de caractere pentru a fi scris în fișierul de ieșire, includând informații despre clienți și starea cozilor la un anumit moment de timp.
5. **Metoda avgTime():**
   * Returnează un șir de caractere care conține informații despre timpul mediu de așteptare, timpul mediu de procesare și ora de vârf în cadrul simulării.

# Rezultate



# Concluzii

Prin acest proiect, am reușit să construiesc o aplicație Java pentru gestionarea cozilor și simularea proceselor de servire a clienților. O concluzie care reflectă ce am învățat și ce mai pot explora este urmatoarea:

1. **Modelarea și implementarea entităților:** Am învățat să definesc și să implementez clase pentru clienți și cozi, inclusiv metode pentru manipularea acestora.
2. **Utilizarea firelor de execuție (thread-urilor):** Am folosit fire de execuție pentru a gestiona simultan mai multe cozi, permițându-le să funcționeze independent.
3. **Strategii de gestionare a cozi:** Am implementat strategii diferite pentru a distribui clienții în cozi, cum ar fi după timpul minim de așteptare sau lungimea minimă a cozii.
4. **Interacțiunea cu interfața grafică:** Am creat o interfață grafică simplă pentru a permite utilizatorului să configureze și să inițieze simularea.

Ce as mai putea imbunatati:

1. **Optimizarea performanței:** Pot să explorez modalități de îmbunătățire a performanței și eficienței aplicației, cum ar fi optimizarea algoritmilor și gestionarea resurselor.
2. **Testarea și depanarea:** Pot să învăț mai multe despre testarea unitară și de integrare pentru a asigura funcționarea corectă a aplicației și pentru a depista și rezolva erorile.
3. **Extinderea funcționalității:** Pot să adaug caracteristici suplimentare, cum ar fi monitorizarea în timp real a stării cozilor, gestionarea priorităților între clienți sau generarea de rapoarte detaliate despre performanță.

În ansamblu, acest proiect m-a familiarizat cu concepte și tehnologii esențiale în dezvoltarea aplicațiilor Java, dar există întotdeauna oportunitate pentru a continua să explorez și să îmi îmbunătățesc cunoștințele și abilitățile în programare.

# Bibliografie

1. [Java Threads - GeeksforGeeks](https://www.geeksforgeeks.org/java-threads/) – informatii despre thread-uri
2. [BlockingQueue Interface in Java - GeeksforGeeks](https://www.geeksforgeeks.org/blockingqueue-interface-in-java/)<https://www.geeksforgeeks.org/java-util-hashmap-in-java-with-examples/> - informatii despre blockingqueues
3. [Atomic Variables in Java with Examples - GeeksforGeeks](https://www.geeksforgeeks.org/atomic-variables-in-java-with-examples/)- informatii despre variabilele atomice
4. FUNDAMENTAL PROGRAMMING TECHNIQUES-https://dsrl.eu/courses/pt/materials/PT\_2024\_A2\_S1.pdf