

DOCUMENTATIE

Tema 2

NUME STUDENT: Rafa Ioana-Sorina
GRUPA: 30223

CUPRINS

1.	Obiectivul temei.....	Error! Bookmark not defined.
2.	Analiza problemei, modelare, scenarii, cazuri de utilizare	Error! Bookmark not defined.
3.	Proiectare	Error! Bookmark not defined.
4.	Implementare	Error! Bookmark not defined.
5.	Rezultate	Error! Bookmark not defined.
6.	Concluzii.....	Error! Bookmark not defined.
7.	Bibliografie	Error! Bookmark not defined.

1.Obiectivul temei

Proiectarea si implementarea aplicatiilor de management al cozilor Atribuiți clienții la cozi pentru a minimiza timpii de așteptare.

Obiective secundare:

1. Definiți clase cu date specifice pentru a stoca informații Despre clienți și cozile aferente
2. Generați aleatoriu N clienți și informațiile de identitate ale acestora Pentru că actrița la acea vreme
3. Crearea de cozi Q și metode de procesare aferente, folosind clase date definite anterior
4. Calculați timpul total de așteptare pentru fiecare client și determinați timpul mediu de așteptare
5. Creați o interfață de utilizator pentru introducerea datelor Intrare
6. Testați aplicația cu diferite seturi de date de intrare

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

Cerințele funcționale includ:

- Aplicația trebuie să faciliteze simularea unui număr de N clienți care sosesc, se înscriu în cozi, așteaptă, sunt serviți și pleacă.
- Fiecare client trebuie să fie definit de trei aspecte: identitatea (ID-ul), momentul sosirii și ritmul de servire.
- Aplicația trebuie să calculeze durata totală petrecută de fiecare client în cozi și să determine media timpului de așteptare.
- Fiecare client trebuie să fie adăugat în coadă în funcție de cel mai scurt timp de așteptare disponibil, atunci când momentul sosirii sale este mai mare sau egal cu momentul de simulare.

Cerințele non-funcționale includ:

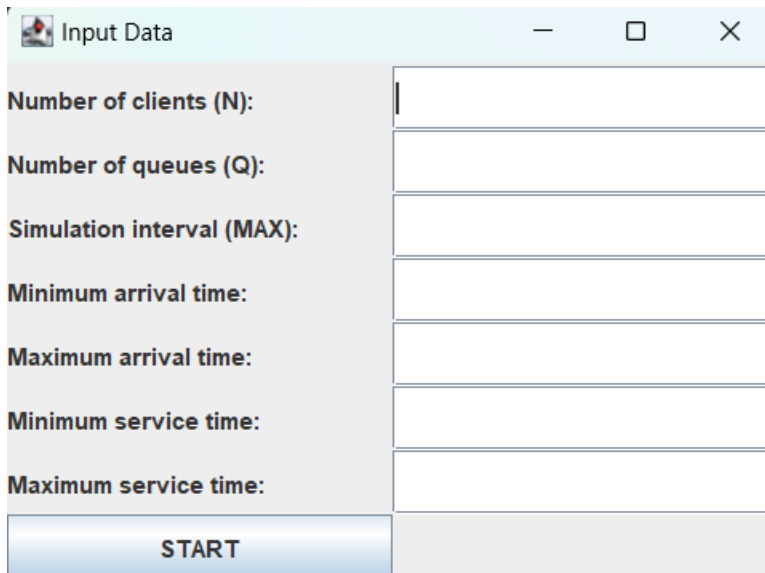
- Performanță: aplicația trebuie să poată realiza simularea într-un interval rezonabil de timp și să gestioneze un volum mare de clienți și cozi.
- Ușurință de utilizare: aplicația trebuie să ofere o interfață simplă și intuitivă pentru introducerea datelor de intrare și afișarea rezultatelor.
- Fiabilitate: aplicația trebuie să gestioneze erorile și să furnizeze o simulare precisă și consecventă.
- Scalabilitate: aplicația trebuie să fie capabilă să gestioneze variabilitatea în numărul de cozi și clienți, fără a compromite performanța sau fiabilitatea.

Cazurile de utilizare pentru aplicația de gestionare a cozilor includ:

- Introducerea parametrilor de intrare: Utilizatorul introduce numărul de clienți, numărul de cozi, intervalul de simulare și limitele pentru duratele minime și maxime de așteptare și servire a clienților.
- Utilizarea unei interfețe prietenoase: Aplicația oferă o interfață intuitivă și ușor de utilizat pentru introducerea parametrilor de intrare și vizualizarea informațiilor despre clienți și cozi.
- Generarea automată a clienților în funcție de datele de intrare: Odată ce utilizatorul a introdus parametrii, aplicația generează automat o serie de clienți cu caracteristici specifice (ID, momentul sosirii și durata servirii).
- Repartizarea unui client în funcție de cea mai scurtă coadă: Aplicația direcționează fiecare client către coada cu cea mai mică durată de așteptare.

Scopul aplicației este de a minimiza timpul de așteptare al clienților prin optimizarea repartizării lor în cozi, evitând astfel creșterea costurilor de serviciu prin adăugarea de noi cozi. De asemenea, aplicația înregistrează durata totală petrecută de fiecare client în cozi și calculează media timpului de așteptare.

3. Proiectare

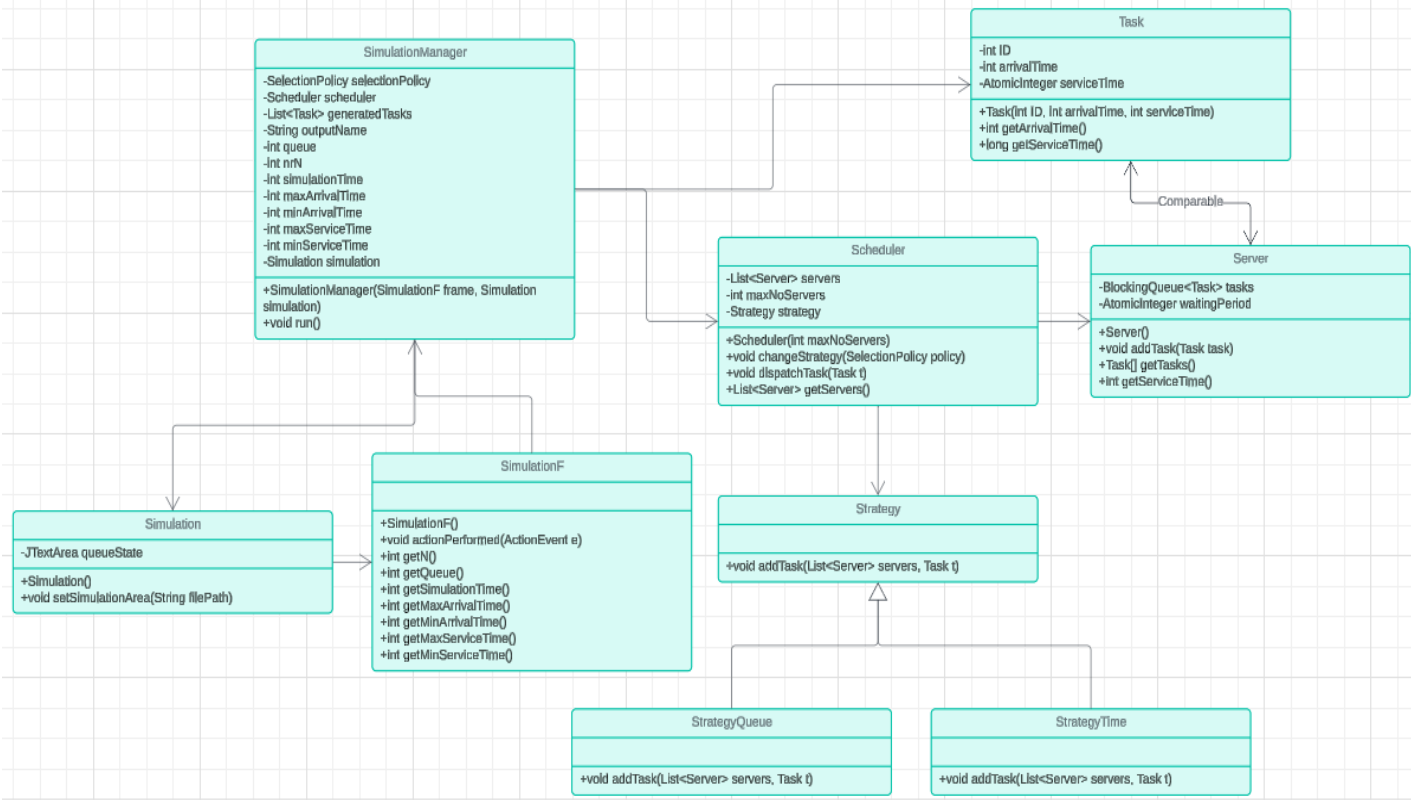


Am create clasele:

1. Scheduler
2. SelectionPolicy
3. SimulationManager
4. Strategy
5. StrategyQueue

6. StrategyTime
7. Simulation
8. SimulationF
9. Server
10. Task

Diagrama UML:



4. Implementare

Clasa Scheduler:

1. Attribute:

servers: O listă de obiecte Server, care reprezintă cozi pentru servirea clienților.

maxNoServers: Numărul maxim de cozi care pot fi create.

strategy: O strategie de selecție a cozii pentru a împărți sarcinile (clienții) între cozi.

2. Constructor:

Scheduler(int maxNoServers): Inițializează un obiect Scheduler cu un număr maxim de cozi specificat. Creează, de asemenea, obiecte Server corespunzătoare numărului maxim de cozi și pornește un fir de execuție (thread) pentru fiecare coadă.

3. Metode:

- changeStrategy(SelectionPolicy policy): Schimbă strategia de selecție a cozii în funcție de politica specificată (SHORTEST_QUEUE sau SHORTEST_TIME).
- dispatchTask(Task t): Distribuie o sarcină (un client) către o coadă folosind strategia specificată.
- getServers(): Returnează lista de obiecte Server asociate Scheduler-ului.
-

Această clasă utilizează și enumerația SelectionPolicy pentru a specifica politica de selecție a coziilor (cel mai scurt timp sau cea mai scurtă coadă). De asemenea, folosește interfețele StrategyQueue și StrategyTime pentru a implementa strategiile specifice de selecție a coziilor.

În esență, Scheduler are rolul de a gestiona și coordona coziile și interacțiunea cu acestea în funcție de politica de selecție specificată.

Clasa SimulationManager:

1. Atribute:

- selectionPolicy: Politica de selecție a coziilor.
- scheduler: Obiectul Scheduler responsabil pentru gestionarea coziilor și distribuirea sarcinilor (clienților) către ele.
- generatedTasks: O listă de sarcini (clienți) generate aleatoriu pentru simulare.
- outputName: Numele fișierului de ieșire pentru înregistrarea rezultatelor simulării.
- Alte atribute pentru parametrii simulării, cum ar fi numărul de cozi, numărul de clienți, timpul de simulare, timpul de sosire și durata de servire maximă și minimă.

2. Constructor:

- SimulationManager(SimulationF frame, Simulation simulation): Inițializează un obiect SimulationManager cu parametri specificați de către interfața grafică. Generează, de asemenea, sarcini aleatorii și pornește un fir de execuție pentru fiecare coadă.

3. Metode:

- `generateNRandomTasks()`: Generează o listă de sarcini aleatorii în funcție de parametrii specificați.
- `run()`: Implementează logica principală a simulării. Iterează prin fiecare moment de timp și distribuie sarcinile în funcție de timpul de sosire. Actualizează și înregistrează starea cozilor și a clienților în fișierul de ieșire. Actualizează și afișează zona de simulare în interfața grafică.
- Alte metode auxiliare pentru manipularea și gestionarea simulării.

Această clasă joacă un rol important în coordonarea simulării și în interacțiunea cu interfața grafică, asigurând că simularea se desfășoară corect conform parametrilor specificați și că rezultatele sunt prezentate corespunzător utilizatorului.

Clasa StrategyQueue:

Această clasă implementează o strategie care selectează coada cu cea mai scurtă lungime și adaugă sarcina în acea coadă.

1. Metoda `addTask`:
 - Iterează prin lista de servere și găsește serverul cu cea mai mică lungime a cozii.
 - Adaugă sarcina în coada găsită.

Clasa StrategyTime:

Această clasă implementează o strategie care selectează coada cu cel mai scurt timp de servire și adaugă sarcina în acea coadă.

1. Metoda `addTask`:
 - Iterează prin lista de servere și găsește serverul cu cel mai mic timp de servire.
 - Adaugă sarcina în coada găsită.

Ambele strategii sunt utile în diferite contexte și pot fi selectate în funcție de nevoile specifice ale aplicației sau ale mediului de simulare. Utilizarea acestor strategii în cadrul clasei `Scheduler` permite flexibilitate în gestionarea coziilor și optimizarea timpului de așteptare al clienților.

Clasa Simulation:

Această clasă reprezintă fereastra principală a aplicației și afișează starea cozilor într-o zonă de text.

1. Atribute:

- queueState: Componentă JTextArea pentru afișarea stării cozilor.

2. Constructor:

- Simulation(): Inițializează fereastra cu titlul "Queue Simulation" și o zonă de text pentru afișarea stării cozilor.

3. Metode:

- setSimulationArea(String filePath): Actualizează zona de simulare cu conținutul din fișierul specificat.

Clasa SimulationF:

Această clasă reprezintă fereastra pentru introducerea datelor de intrare pentru simulare.

1. Atribute:

- Componente JLabel și JTextField pentru introducerea datelor de intrare, cum ar fi numărul de clienți, numărul de cozi, intervalul de simulare și alte parametri relevanți.
- Butonul "START" pentru a iniția simularea.

2. Constructor:

- SimulationF(): Inițializează fereastra și componentele pentru introducerea datelor de intrare.

3. Metode:

- getN(), getQueue(), getSimulationTime(), etc.: Metode pentru a obține valorile introduse de utilizator pentru diferiții parametri ai simulării.
- main(String[] args): Punctul de intrare în aplicație, care creează o instanță a clasei SimulationF.

Aceste două clase lucrează împreună pentru a permite utilizatorului să introducă datele de intrare pentru simulare și pentru a afișa starea cozilor în timpul simulării. Apoi, clasa SimulationManager este utilizată pentru a iniția și a coordona simularea pe baza datelor introduse.

Clasa Server:

Această clasă reprezintă un server care poate servi sarcini (clienți) și gestionează coada de sarcini.

1. Atribute:

- `tasks`: O coadă blocantă (`BlockingQueue`) pentru a stoca sarcinile care așteaptă să fie servite.
- `waitingPeriod`: Un contor atomic pentru a urmări perioada totală de așteptare a sarcinilor în coadă.

2. Constructor:

- `Server()`: Inițializează un nou server cu o coadă goală și o perioadă de așteptare inițială de 0.

3. Metode:

- `addTask(Task task)`: Adaugă o sarcină în coadă și actualizează perioada totală de așteptare.
- `run()`: Implementează logica de servire a sarcinilor. Serverul verifică periodic dacă există sarcini în coadă și le servește în ordinea sosirii lor.
- `getTasks()`: Returnează un array cu toate sarcinile din coadă.
- `getServiceTime()`: Calculează perioada totală de servire rămasă pentru toate sarcinile din coadă.

Clasa Task:

Această clasă reprezintă o sarcină individuală (client) care trebuie să fie servită de un server.

1. Atribute:

- `ID`: Identificatorul unic al sarcinii.
- `arrivalTime`: Momentul de timp la care sarcina ajunge în sistem.
- `serviceTime`: Durata necesară pentru servire, reprezentată ca un contor atomic.

2. Constructor:

- `Task(int ID, int arrivalTime, int serviceTime)`: Inițializează o nouă sarcină cu atributele specificate.

3. Metode:

- `getArrivalTime()`: Returnează momentul de timp al sosirii sarcinii în sistem.
- `getServiceTime()`: Returnează durata rămasă de servire a sarcinii.
- `setServiceTime(int serviceTime)`: Actualizează durata de servire a sarcinii.
- `compareTo(Task other)`: Implementează metoda `compareTo` pentru a permite sortarea sarcinilor după momentul de sosit.

Aceste două clase formează baza funcționalității de simulare a coziilor și sarcinilor în cadrul aplicației tale. Clasa Server se ocupă de gestionarea coziilor și a procesului de servire a sarcinilor, în timp ce clasa Task reprezintă sarcinile individuale care trebuie să fie procesate de servere.

5.Rezultate

Am introdus următoarele date de intrare:

Test 1	Test 2	Test 3
$N = 4$ $Q = 2$ $t_{simulation}^{MAX} = 60 \text{ seconds}$ $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [2, 30]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [2, 4]$	$N = 50$ $Q = 5$ $t_{simulation}^{MAX} = 60 \text{ seconds}$ $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [2, 40]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [1, 7]$	$N = 1000$ $Q = 20$ $t_{simulation}^{MAX} = 200 \text{ seconds}$ $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [10, 100]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [3, 9]$

Rezultatele după rulearea programului au fost scrise pentru fiecare simulare în parte într-un fișier text cu nume specific fiecărui test: “Test1”, “Test2” și “Test3”.

6. Concluzii

Am dezvoltat o aplicație eficientă de gestionare a coziilor, care a reușit să minimizeze timpul de așteptare al clienților. Prin utilizarea unei simulări a timpului de execuție și a unui set de date de intrare specificat de utilizator, am reușit să distribuim clienții în cozi și să alocăm resursele într-un mod optim. Strategia noastră a fost să adăugăm clienții în coada cu cel mai mic timp de așteptare atunci când timpul lor de sosire este mai mare sau egal cu timpul de simulare curent.

Din această temă, am învățat să lucrăm cu thread-uri pentru a gestiona simultan clienții și cozile, să implementăm o afișare live pentru a monitoriza desfășurarea simulării și să folosim scrierea în fișiere în Java pentru a salva rezultatele obținute.

Există câteva îmbunătățiri pe care le putem aduce:

- Adăugarea unui sistem de prioritizare pentru clienții importanți.
- Integrarea unei opțiuni de plată online pentru a îmbunătăți experiența utilizatorilor.

7.Bibliografie

https://dsrl.eu/courses/pt/materials/PT_2024_A2_S1.pdf

https://dsrl.eu/courses/pt/materials/PT2024_A3_S2.pdf

https://www.tutorialspoint.com/java/util/timer_schedule_period.htm

