DOCUMENTATIE

TEMA *2*

NUME STUDENT: Anghel Mihai Radu

GRUPA: 30225

# Obiectivul temei

*Obiectivul principal al acestei teme este de a proiecta și implementa o aplicație care își propune să analizeze sistemele bazate pe cozi prin simularea unei serii de N clienți care sosesc pentru servicii, intră în Q cozi, așteaptă, sunt serviți și, în cele din urmă, părăsesc cozile, și calcularea timpului mediu de așteptare, timpului mediu de servire și a orei de vârf.*

*Obiectivele secundare pentru implementarea cozii sunt:*

*• Analiza problemei și identificarea cerințelor*

*• Proiectarea aplicației de simulare*

*• Implementarea aplicației de simulare*

*• Testarea aplicației de simulare*

# Analiza problemei, modelare, scenarii, cazuri de utilizare

*Cozile sunt folosite frecvent pentru a modela domenii din lumea reală. Obiectivul principal al unei cozi este de a oferi un loc unde un "client" poate aștepta înainte de a primi un "serviciu".*

*Managementul sistemelor bazate pe cozi își propune să minimizeze timpul în care "clienții" lor așteaptă în cozi înainte de a fi serviți. Un mod de a minimiza timpul de așteptare este de a adăuga mai multe servere, adică mai multe cozi în sistem (fiecare coadă este considerată a avea un procesor asociat), dar această abordare crește costurile furnizorului de servicii.*

*Aplicația de management al cozilor simulează o serie de N clienți care sosesc pentru servicii, intră în Q cozi, așteaptă, sunt serviți și, în cele din urmă, părăsesc cozile. Toți clienții sunt generați când simularea începe și sunt caracterizați prin trei parametri: ID (un număr între 1 și N), 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 (timpul de simulare când sunt pregătiți să intre în coadă) și 𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 (intervalul de timp sau durata necesară pentru a servi clientul; adică timpul de așteptare atunci când clientul se află în fața cozii). Aplicația urmărește timpul total petrecut de fiecare client în cozi și calculează timpul mediu de așteptare. Fiecare client este adăugat la coadă cu timpul minim de așteptare atunci când timpul său de sosire este mai mare sau egal cu timpul de simulare.*

*Pentru a modela cozile, am utilizat clasa Server care reprezintă coada. Fiecare server are o coadă (implementată prin intermediul BlockingQueue) care conține sarcinile care urmează să fie preluate și servite. Serverul este reprezentat prin câteva variabile de instanță, cum ar fi id-ul serverului, timpul de așteptare și o variabilă booleană care indică dacă serverul rulează sau nu.*

*Clasa Task reprezintă o sarcină în sistemul bazat pe cozi și are trei variabile de instanță: id-ul sarcinii, timpul de sosire și timpul de servire. Aceasta este utilizată pentru a crea sarcini și a le adăuga la coadă prin intermediul metodei addTask din clasa Server.*

*În concluzie, clasele Server și Task sunt utilizate pentru a modela sistemul bazat pe cozi. Serverul reprezintă un procesor care preia sarcinile din coadă și le servește, iar clasa Task reprezintă sarcinile care așteaptă să fie preluate de server. Aceste clase sunt utilizate în implementarea aplicației de simulare a sistemelor bazate pe cozi.*

*Scenariile și cazurile de utilizare:*

*Un client nou sosește la coadă. Caz de utilizare: când un client nou sosește, acesta trebuie adăugat la coada cu cel mai mic timp de așteptare disponibil.*

*Un client este servit și părăsește coada. Caz de utilizare: când un client este servit, acesta trebuie eliminat din coadă și timpul total petrecut de acesta în coadă trebuie înregistrat.*

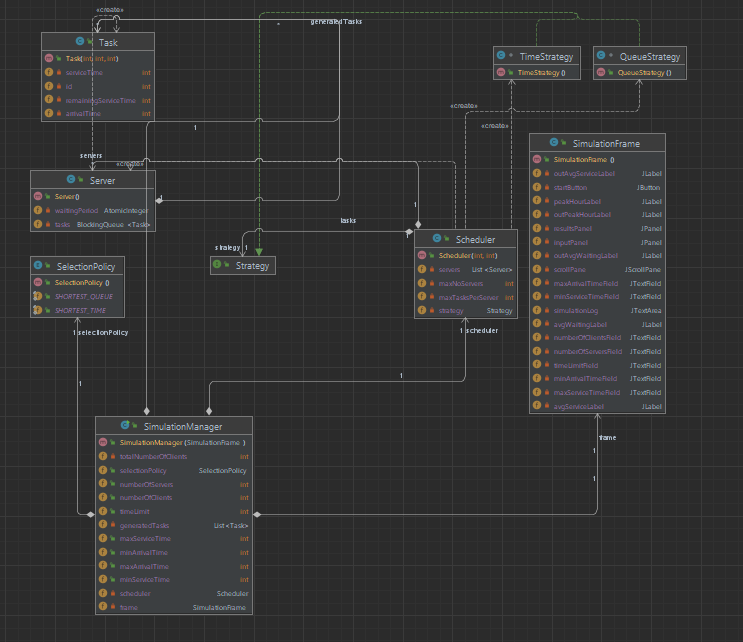
*Se calculează timpul mediu de așteptare. Caz de utilizare: la finalul simulării, aplicația trebuie să calculeze timpul mediu de așteptare pentru toți clienții.*

*Se modifică intervalul de timp necesar pentru a servi un client. Caz de utilizare: dacă managementul sistemului dorește să optimizeze timpul necesar pentru a servi clienții, poate modifica intervalul de timp necesar pentru a servi un client, ceea ce va influența timpul de așteptare pentru clienți.*

*În viața reală, această aplicație este folosită ca un simulator de clienți pentru una sau mai multe cozi. De exemplu,* *sunt mai multe case de marcat la un magazin, iar oamenii se așează la coadă, având timpul de sosire generat aleator din inputul dat de către utilizator. Programul generează numărul de clineți pe care îl introducem din input, random. Aceștia, o dată ajunși la coadă, sunt serviți într-un interval de timp random ales între minimul si maximul adăugat de la utilizator. O dată serviți, clienții vor părăsi coada, urmând ca următorul client să fie servit, până la finalizarea procesului.*

# Proiectare

*Aplicația creată de mine este o aplicație care generează n clienți la q cozi. Această aplicație are o interfață grafică care permite utilizatorilor să introducă numărul de clineți dorit, numărul de cozi, timpul maxim de simulare, timpul minim de sosire și de servire și timpul maxim de servire și de sosire.*

**

# Implementare

***Pachetul Model*** *conține clasele Task, Server și ServerListener.*

***Clasa Task*** *reprezintă o sarcină individuală, adică un client cu un ID, un timp de sosire și un timp de servire.*

* *id: ID-ul sarcinii.*
* *arrivalTime: Timpul când sarcina sosește.*
* *serviceTime: Timpul necesar pentru a finaliza sarcina.*
* *RemainingServiceTime: Timpul ramas pana la finalizarea comenzii*

*Clasa conține și metode getter și setter pentru fiecare atribut.*

***Interfața ServerListener*** *conține o singură metodă, serverUpdated(Server server), care este apelată atunci când starea serverului este actualizată.*

***Clasa Server*** *reprezintă un server care poate procesa sarcini. În cazul meu, clasa server definește o coadă. Clasa implementează interfața Runnable, astfel încât poate fi rulată într-un fir de execuție separat.*

* *Constructorul: inițializează cozile de așteptare și perioada de așteptare, setează un ID unic pentru server și îl incrementează pentru a fi utilizat de serverul următor și setează starea de rulare a serverului.*
* *addTask(Task newTask): adaugă o nouă sarcină în coada serverului și actualizează perioada de așteptare în conformitate cu durata de servire a sarcinii adăugate.*
* *stop(): oprește rularea serverului prin actualizarea stării de rulare.*
* *getTasks(): returnează o listă cu sarcinile din coada serverului.*
* *getWaitingPeriod(): returnează perioada totală de așteptare a tuturor sarcinilor din coada serverului.*
* *removeCurentTask(Task task): elimină o sarcină din coada serverului și actualizează perioada de așteptare. Dacă sarcina nu poate fi eliminată din coadă, metoda afișează un mesaj de eroare. Dacă există un obiect ServerListener setat, metoda apelează metoda serverUpdated() pentru a notifica ascultătorii despre modificările stării serverului.*

***Interfața Strategy*** *definește o metodă addTask(). Această interfață este utilizată pentru a defini strategiile de alegere a serverelor în funcție de criteriile specificate. Metoda addTask() primește o listă de servere și o sarcină și returnează serverul specificat conform strategiei specifice.*

***TimeStrategy*** *implementează interfața Strategy, care are o singură metodă addTask(). Această clasă definește o strategie de alegere a serverului cel mai puțin ocupat pentru a procesa o sarcină nouă.*

* *addTask(List<Server> servers, Task t): această metodă primește o listă de servere și o sarcină și returnează serverul cu cea mai mică perioadă de așteptare. Metoda utilizează stream-uri și metoda min() pentru a găsi serverul cu cea mai mică perioadă de așteptare și Comparator.comparing() pentru a obține perioada de așteptare a fiecărui server. Dacă nu există servere disponibile, metoda aruncă o excepție de stare ilegală. Această metodă este utilizată pentru a adăuga sarcini noi în coada serverului cel mai puțin ocupat, utilizând strategia specificată de clasa ConcreteStrategyTime.*

***QueueStrategy*** *implementează interfața Strategy și definește o strategie de alegere a serverului cu cel mai puține sarcini în coadă pentru a procesa o sarcină nouă.*

* *addTask(List<Server> servers, Task t): această metodă primește o listă de servere și o sarcină și returnează serverul cu cel mai mic număr de sarcini în coadă. Metoda utilizează stream-uri și metoda min() pentru a găsi serverul cu cea mai mică dimensiune de coadă și Comparator.comparing() pentru a obține dimensiunea cozi fiecărui server. Dacă nu există servere disponibile, metoda aruncă o excepție IllegalMonitorStateException. Această metodă este utilizată pentru a adăuga sarcini noi în coada serverului cu cel mai mic număr de sarcini, utilizând strategia specificată de clasa ConcreteStrategyQueue.*

***Clasa Scheduler*** *reprezintă planificatorul sistemului de cozi și gestionează procesarea sarcinilor în funcție de politica de selecție specificată prin intermediul obiectului de strategie.*

* *Atributul servers reprezintă o listă de obiecte Server, care sunt serverele disponibile în sistem și care procesează sarcinile în paralel.*
* *ExecutorService este un executor de fir de execuție și este utilizat pentru a executa obiectele Server într-un fir de execuție separat.*
* *Metoda constructorului initializează lista servers și creează obiecte Server pentru fiecare server disponibil, apoi le adaugă la listă și le execută prin intermediul executorService.*
* *Metoda changeStrategy schimbă politica de selecție a sistemului de cozi, bazat pe valoarea specificată de SelectionPolicy. Acest lucru se realizează prin instantierea unei noi strategii ConcreteStrategyQueue sau ConcreteStrategyTime, în funcție de politica specificată.*
* *Metoda dispatchTask adaugă o sarcină la coada de așteptare a unui server, în funcție de politica de selecție specificată. Aceasta apelează metoda addTask a serverului selectat.*
* *Metoda queueStatus returnează o reprezentare sub formă de șir de caractere a stării curente a cozilor pentru fiecare server.*
* *Metoda shutdown oprește executorService.*

***Clasa SimulationFrame*** *implementeaza un simulator de cozi, care simuleaza fluxul de clienti care sosesc si sunt serviti de un numar finit de cozi (servere). Utilizatorul poate configura numarul de clienti, numarul de cozi, timpul de simulare, precum si intervalele de timp pentru sosirea si servirea clientilor. Aplicatia utilizeaza o strategie de selectie pentru a decide care coada va prelua urmatorul client care ajunge. Utilizatorul poate alege intre doua strategii de selectie - "cea mai scurta coada" si "cel mai scurt timp".*

*Interfata grafica a aplicatiei este construita cu ajutorul bibliotecii Swing. La pornirea aplicatiei, utilizatorul trebuie sa completeze informatiile necesare pentru simulare intr-un formular de configurare. Dupa ce utilizatorul a completat toate campurile necesare, acesta poate porni simularea apasand butonul "START".*

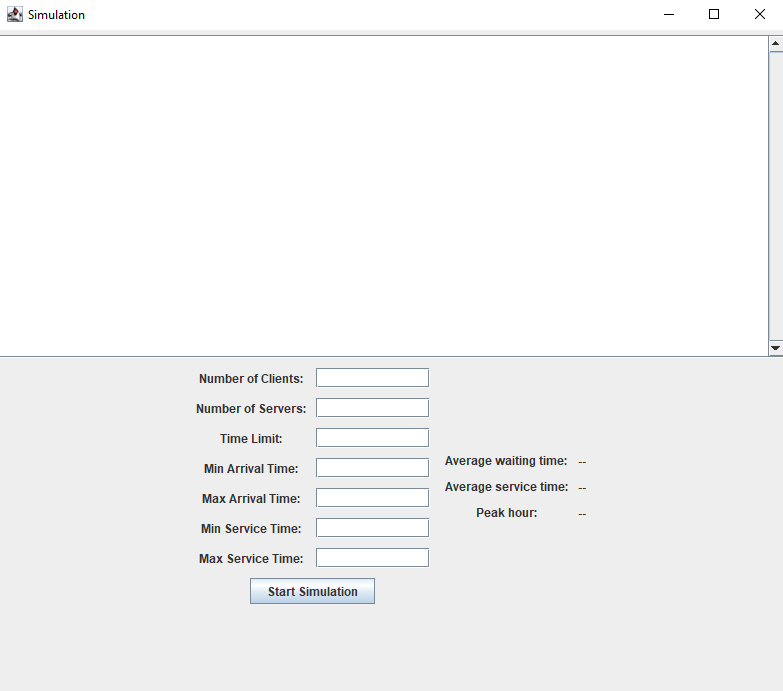
*Odata ce simularea este pornita, se deschide o fereastra cu interfata grafica a serverelor. Fiecare server este reprezentat printr-un panou care afiseaza numele cozii si clienții care asteapta sa fie serviti. In timpul simularii, utilizatorul poate urmari evolutia cozilor si poate observa cum clientii sunt serviti de servere.*

*Simularea ruleaza in background si este actualizata la fiecare secunda. Aceasta este realizata cu ajutorul unui obiect Timer din biblioteca Swing, care declanseaza un eveniment la fiecare secunda si face actualizari la interfata grafica.*

***Clasa SimulationManager*** *este clasa principală care gestionează simularea procesului de așteptare la cozi. Această clasă se ocupă de crearea și gestionarea obiectelor de tip Server, Scheduler, SetupFrame și SimulationFrame.*

* *Constructorul clasei SimulationManager primește un obiect de tip SetupFrame, unde se specifică parametrii simulării (numărul de clienți, numărul de cozi, timpul maxim de simulare, etc.) și apoi initializează Scheduler-ul cu numărul de cozi și de clienți specificate. De asemenea, se setează politica de selectare a coziilor (de exemplu, în cazul nostru, politica de selectare este cea a coziilor cu timp de așteptare minim). Apoi se creează o instanță de SimulationFrame, care este fereastra grafică care va afișa starea curentă a cozilor.*
* *Metoda updateFrame() se ocupă de actualizarea frame-ului de simulare. Această metodă este apelată de fiecare dată când se schimbă starea cozilor sau a clienților.*
* *Metoda stopSimulation() oprește simularea, adică oprește toate serverele.*
* *Metoda logToFile() este responsabilă de scrierea evenimentelor din cadrul simulării într-un fișier log.*
* *Metoda generateNRandomTasks() generează o listă de n obiecte Task, cu timpuri de sosire și de servire generate aleatoriu.*
* *Metoda run() este cea care gestionează simularea în sine. În cadrul acestei metode, se verifică la fiecare secundă dacă a sosit un client nou și se adaugă într-o coadă. Apoi se actualizează frame-ul de simulare cu noile informații. În final, se calculează și se afișează timpul mediu de așteptare.*
* *Metoda main() este metoda principală a aplicației și este responsabilă pentru crearea unei instanțe de SetupFrame și pentru pornirea simulării. Când utilizatorul apasă butonul "START", se creează o instanță de SimulationManager și se pornește un nou thread pentru simulare. Apoi se ascunde fereastra de setări a simulării.*

# Rezultate



# 

# Concluzii

*În concluzie, prin acest proiect am avut oportunitatea să înveț și să aplic cunoștințele legate de gestionarea și simularea cozilor cu clienți. Am învățat cum să lucrez cu firul de execuție, cum să folosesc interfețele grafice legate de threads-uri și cum să generez evenimente și să actualizez starea aplicației.*

*De asemenea, am fost expusă la mai multe concepte de programare, cum ar fi algoritmi de sortare, strategii de selecție și cum să lucrez cu fișiere.*

*Este importat să mențin aceste abilități pe care le-am dobândit și să evoluez. Mă pot ajuta în viitor să construiesc aplicații mai complexe și să dezvolt proiecte mai mari.*

*Această aplicație poate fi dezvoltată, uterior pentru diverse proiecte de simulare. De exemplu, se poate trece direct la o simulare de producție, pe baza acestei aplicații, sau gestionarea unui site pentru clienți.*

# Bibliografie

1. *Java Thread Pool Example using Executors* [*https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html*](https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html)
2. *java.util.Timer.schedule() Method* [*https://www.tutorialspoint.com/java/util/timer\_schedule\_period.htm*](https://www.tutorialspoint.com/java/util/timer_schedule_period.htm)
3. *The Java Tutorials* [*https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html*](https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html)
4. *Class FileWriter* [*https://docs.oracle.com/javase/7/docs/api/java/io/FileWriter.html*](https://docs.oracle.com/javase/7/docs/api/java/io/FileWriter.html)
5. *Java Threads* [*https://www.w3schools.com/java/java\_threads.asp*](https://www.w3schools.com/java/java_threads.asp)
6. *Java Threads* [*https://www.geeksforgeeks.org/java-threads/*](https://www.geeksforgeeks.org/java-threads/)
7. *Class AtomicInteger* [*https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html*](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html)
8. *Java Concurrency - AtomicInteger Class* [*https://www.tutorialspoint.com/java\_concurrency/concurrency\_atomic\_integer.htm*](https://www.tutorialspoint.com/java_concurrency/concurrency_atomic_integer.htm)