DOCUMENTATIE

**TEMA 2**

## NUME STUDENT: Szabo Szilard

## GRUPA: 30229

**CUPRINS**

1. [Obiectivul temei 3](#_TOC_250006)
2. [Analiza problemei, modelare, scenarii, cazuri de utilizare 3](#_TOC_250005)
3. Proiectare....................................................................................................................................5
4. Implementare...............................................................................................................................7
5. Rezultate.....................................................................................................................................10
6. Concluzii....................................................................................................................................10
7. Bibliografie................................................................................................................................11

# Obiectivul temei

*Obiectivul temei descrise este de a proiecta și implementa o aplicație de gestionare a cozilor care atribuie clienților cozi astfel încât timpul de așteptare să fie minimizat. Aplicația trebuie să simuleze sosirea unui număr de N clienți, care intră în Q cozi, așteaptă, sunt serviți și apoi părăsesc coada. Scopul principal este de a minimiza timpul total petrecut de fiecare client în cozi și de a calcula timpul mediu de așteptare. Aceasta implică luarea în considerare a momentului de sosire al fiecărui client, durata de servire și numărul de cozi disponibile. Aplicația trebuie să ofere, de asemenea, o interfață utilizator grafică pentru a permite utilizatorului să introducă datele de intrare necesare, cum ar fi numărul de clienți, numărul de cozi, intervalul de simulare și timpul minim și maxim de sosire și servire a clienților.*

# Analiza problemei, modelare, scenarii, cazuri de utilizare

*Analiza problemei:*

*Problema care trebuie rezolvată este legată de gestionarea cozilor într-un sistem. Scopul este de a minimiza timpul de așteptare al clienților în cozi și de a asigura că fiecare client este atribuit cozi într-un mod eficient și echitabil. Sistemul trebuie să fie capabil să manipuleze mai multe cozi, fiecare având propriul procesor, și să simuleze sosirea clienților și durata lor de servire.*

*Modelare:*

*Modelul problemei este reprezentat de un sistem de cozi, unde fiecare coadă este asociată cu un procesor. Scopul este de a minimiza timpul de așteptare al clienților în cozi și de a asigura că clienții sunt atribuiți cozi în mod eficient și echitabil.*

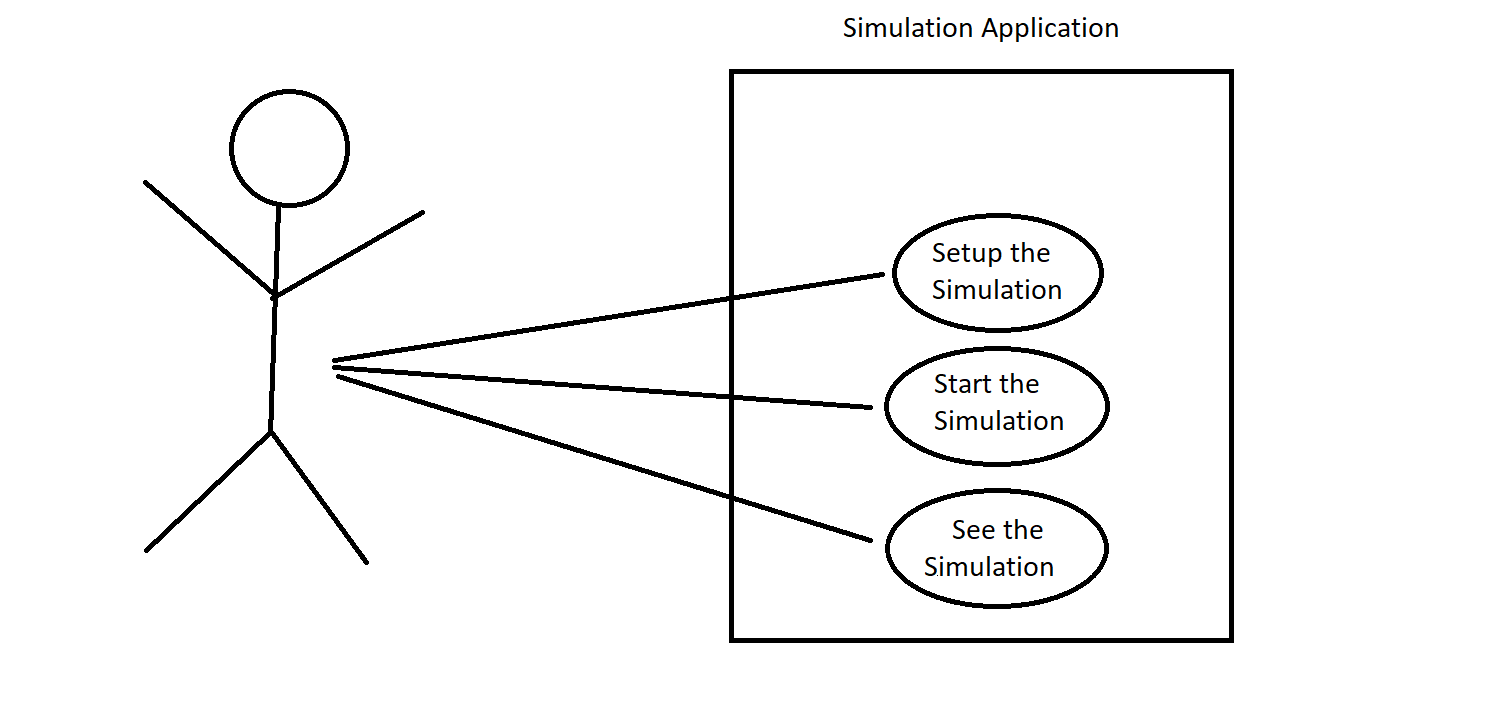
*Scenarii:*

* *Un client ajunge la sistem și nu există cozi disponibile. În acest caz, clientul trebuie să aștepte până când o coadă este disponibilă.*
* *Un client ajunge la sistem și există mai multe cozi disponibile. În acest caz, clientul este atribuit cozi cu cel mai scurt timp de așteptare.*
* *Un client este atribuit unei cozi și începe să aștepte. Dacă un alt client ajunge și este atribuit aceleiași cozi, trebuie să fie servit în ordinea în care a ajuns în coadă.*
* *Durata de servire a clienților poate varia și poate fi diferită de la un client la altul. Acest lucru trebuie luat în considerare la atribuirea clienților la cozi.*

*Cazuri de utilizare:*

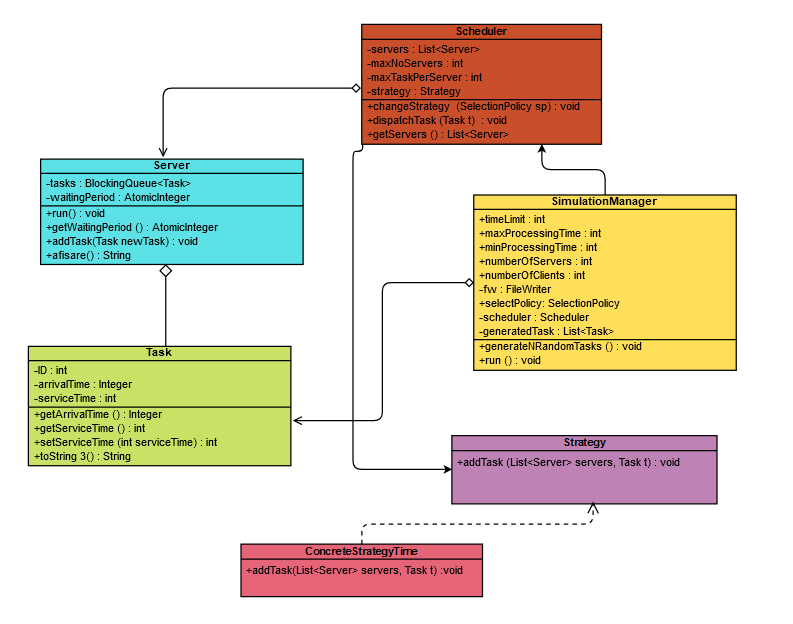
* *Un magazin care trebuie să gestioneze o coadă de clienți la casele de marcat.*
* *O companie care trebuie să gestioneze cozi de clienți care așteaptă să fie transferați la departamente diferite.*
* *Un restaurant care trebuie să gestioneze cozi de clienți care așteaptă să fie serviți.*

Diagrama use-case:



# Proiectare

*Diagrama UML de clase este:*

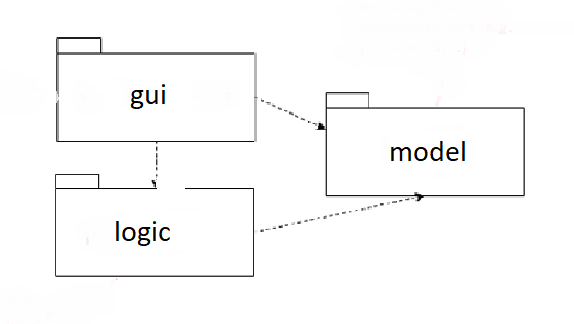


În cadrul acestui proiect, am implementat o simulare a unui sistem de cozi cu mai multe servere. Am creat mai multe clase care modelează diferite aspecte ale sistemului, precum: Server, Task, Scheduler și SimulationManager. De asemenea, am implementat o interfață numită Strategy, care definește o strategie de adăugare a unei sarcini în coadă.

* *Clasa Task - descrie o sarcină, cu proprietăți precum id-ul sarcinii, timpul de sosire și timpul de procesare.*
* *Clasa Server - reprezintă un server și are o coadă cu sarcini de procesat și un timp de așteptare total.*
* *Clasa SimulationManager - administrează simularea, cu proprietăți precum timpul maxim, timpul minim și numărul de servere și sarcini de simulat. Acesta generează și ordonează sarcinile, apoi le trimite la serverul potrivit, pe baza politicii de selecție specificată.*
* *Clasa Scheduler - este responsabilă pentru gestionarea serverelor și procesarea sarcinilor pe baza strategiei de selecție specificate.*
* *Interfața Strategy - definesc o interfață pentru strategiile de selecție a sarcinilor.*

*Pentru a ilustra integrarea acestor clase, ar putea fi util să faci un diagramă UML care prezintă relațiile dintre acestea. De exemplu, clasa SimulationManager utilizează o instanță a clasei Scheduler, care la rândul său utilizează o strategie de selecție implementată într-o clasă care implementează interfața Strategy. Clasa Scheduler utilizează, de asemenea, o listă de servere care sunt reprezentate de instanțe ale clasei Server. Aceste servere sunt create și inițializate în constructorul clasei Scheduler. La rândul lor, clasa Server utilizează o coadă de sarcini și un timp de așteptare total, care sunt gestionate în interiorul acesteia.*

Diagrama UML de pachete este:

**

Model contine clasele Task si Server.

Logic contine clasele: SimulationManager, ConcreteStrategyTime, Scheduler si interfata Strategy.

*Paradigma OOP (Orientată Obiect) este un stil de programare care se concentrează pe obiecte și interacțiunile dintre ele. Aceasta implică utilizarea unor concepte cum ar fi încapsularea, abstractizarea, moștenirea și polimorfismul pentru a structura și organiza codul în module reutilizabile și ușor de întreținut.*

*În paradigma OOP, obiectele sunt entități care conțin date și comportament. Acestea pot fi create ca instanțe ale unei clase, care este o definiție a obiectului și a proprietăților și metodelor sale. În OOP, codul este împărțit în clase, care sunt organizate într-o ierarhie de moștenire, în care clasele copil moștenesc proprietățile și metodele clasei părinte.*

*Încapsularea implică împachetarea datelor și metodelor relevante pentru un obiect și ascunderea acestora de utilizatorii exteriori. Abstractizarea implică definirea unei interfețe pentru un obiect sau o clasă, astfel încât să se poată utiliza acestea fără a fi necesar să cunoaștem toate detaliile de implementare. Moștenirea permite crearea de noi clase, bazate pe proprietățile și metodele unei clase părinte, iar polimorfismul permite utilizarea unor obiecte în mai multe moduri, în funcție de context.*

*OOP poate oferi numeroase beneficii, precum modularitate, flexibilitate și reutilizabilitate.*

# Implementare

*Clasa Task:*

* *Campuri:*
  + *id - id-ul task-ului, de tip int*
  + *arrivalTime - momentul de timp la care task-ul a ajuns la coadă, de tip int*
  + *serviceTime - timpul necesar pentru procesarea task-ului, de tip int*
* *Metode:*
  + *Task(int id, int arrivalTime, int serviceTime) - constructor care initializează toate câmpurile clasei*
  + *int getId() - metoda pentru obținerea id-ului task-ului*
  + *void setId(int id) - metoda pentru setarea id-ului task-ului*
  + *int getArrivalTime() - metoda pentru obținerea momentului de timp la care task-ul a ajuns la coadă*
  + *void setArrivalTime(int arrivalTime) - metoda pentru setarea momentului de timp la care task-ul a ajuns la coadă*
  + *int getServiceTime() - metoda pentru obținerea timpului necesar pentru procesarea task-ului*
  + *void setServiceTime(int serviceTime) - metoda pentru setarea timpului necesar pentru procesarea task-ului*
  + *String toString() - metoda pentru reprezentarea sub formă de string a unui obiect de tip Task*

*Clasa Server:*

* *Campuri:*
  + *tasks - coada de task-uri a serverului, de tip List<Task>*
  + *maxTasks - numărul maxim de task-uri ce pot fi procesate de server în același timp, de tip int*
  + *waitingPeriod - timpul total de așteptare al tuturor task-urilor din coada, de tip AtomicInteger*
* *Metode:*
  + *Server() - constructor care setează numărul maxim de task-uri ce pot fi procesate de server în același timp cu valoarea implicită 1*
  + *Server(int maxTasks) - constructor care setează numărul maxim de task-uri ce pot fi procesate de server în același timp cu o valoare specificată*
  + *void addTask(Task t) - adaugă un task la coada serverului*
  + *void removeTask(Task t) - elimină un task din coada serverului*
  + *void run() - metoda pentru procesarea task-urilor aflate în coadă*
  + *int getMaxTasks() - metoda pentru obținerea numărului maxim de task-uri ce pot fi procesate de server în același timp*
  + *void setMaxTasks(int maxTasks) - metoda pentru setarea numărului maxim de task-uri ce pot fi procesate de server în același timp*
  + *AtomicInteger getWaitingPeriod() - metoda pentru obținerea timpului total de așteptare al tuturor task-urilor din coada*
  + *void setWaitingPeriod(AtomicInteger waitingPeriod) - metoda pentru setarea timpului total de așteptare al tuturor task-urilor din coada*
  + *String afisare() - metoda pentru reprezentarea sub formă de string a tuturor task-urilor din coadă*

*Clasa SimulationManager:*

* *Campuri:*
  + *int simulationInterval: reprezintă intervalul de simulare al sistemului în milisecunde.*
  + *Scheduler scheduler: instanță a clasei Scheduler utilizată pentru gestionarea serverelor și a sarcinilor.*
  + *InputReader reader: instanță a clasei InputReader utilizată pentru citirea datelor de intrare.*
  + *OutputWriter writer: instanță a clasei OutputWriter utilizată pentru scrierea datelor de ieșire.*
* *Metode:*
  + *public void start(): metoda care pornește simularea sistemului.*
  + *private void sleep(int interval): metoda utilitară pentru suspendarea execuției simulării cu un interval dat în milisecunde.*
  + *private void displayStatus(): metoda utilitară pentru afișarea stării serverelor în timpul simulării.*

*Clasa Scheduler:*

* *Campuri:*
  + *List<Server> servers: lista de instanțe ale clasei Server utilizate pentru simularea serverelor.*
  + *int maxNoServers: numărul maxim de servere acceptate în sistem.*
  + *int maxTasksPerServer: numărul maxim de sarcini pe care un server le poate gestiona în același timp.*
  + *Strategy strategy: instanță a unei clase care implementează interfața Strategy utilizată pentru distribuirea sarcinilor.*
* *Metode:*
  + *public Scheduler(int maxNoServers, int maxTasksPerServer): constructor care primește ca parametri numărul maxim de servere și numărul maxim de sarcini pe care un server le poate gestiona în același timp. Instanțiază serverele și pornește thread-urile corespunzătoare pentru fiecare server.*
  + *public void changeStrategy(SelectionPolicy sp): metoda care primește o politică de selecție și schimbă strategia utilizată pentru distribuirea sarcinilor în funcție de aceasta.*
  + *public void dispatchTask(Task t): metoda care primește o sarcină și o distribuie către un server utilizând strategia curentă.*
  + *public List<Server> getServers(): metoda care returnează lista de servere din sistem.*

*Clasa ConcreteStrategyTime:*

* *Metode:*
  + *public void addTask(List<Server> servers, Task t): metoda care primește o listă de servere și o sarcină și distribuie sarcina către serverul care are cel mai scurt timp de așteptare.*

*Interfața Strategy:*

* *Metoda:*
  + *void addTask(List<Server> servers, Task t) - metoda care adaugă un task în coada*

# Rezultate

# 

*Pentru a testa buna functionalitate a aplicatiei noastre, am realizat trei teste, pentru situatii diferite, cu scopul de a verifica daca aceasta poate procesa corect sarcinile primite si poate livra rezultatele corecte.*

*Primul test a fost realizat cu urmatoarele date: numarul de servere = 2, numarul de clienti = 4, timpul maxim de procesare = 4, timpul minim de procesare = 2, timpul maxim de sosire = 30, timpul minim de sosire = 2 . Acest test a fost rulat cu strategia "Shortest Time" cu o limita de timp egala cu 60 de secunde. Rezultatele obtinute sunt cele asteptate si se regasesc in fisierul "test1.txt".*

*Al doilea test a fost realizat cu urmatoarele date: numarul de servere = 5, numarul de clienti = 50, timpul maxim de procesare = 7, timpul minim de procesare = 1, timpul maxim de sosire = 40, timpul minim de sosire = 2 . Acest test a fost de asemenea rulat cu strategia "Shortest Time" cu o limita de timp egala cu 60 de secunde. Rezultatele obtinute sunt cele asteptate si se regasesc in fisierul "test2.txt".*

*Ultimul test a fost rulat cu datele de intrare: numarul de servere = 20, numarul de clienti = 1000, timpul maxim de procesare = 9, timpul minim de procesare = 3, timpul maxim de sosire = 100, timpul minim de sosire = 10. Rezultatele obtinute au fost similare cu cele asteptate si se gasesc in fisierul "test3.txt".*

*Pe baza testelor efectuate putem concluziona ca aplicatia noastra functioneaza in conformitate cu specificatiile initiale si poate procesa cu succes sarcinile primite, indiferent de strategia folosita.*

# Concluzii

Din acest proiect, am invatat cum sa proiectezi si sa implementez o aplicatie care utilizeaza thread-uri si sincronizare pentru a gestiona un numar mare de obiecte de acelasi tip (in cazul nostru, servere si sarcini). De asemenea, am invatat cum sa utilizez pattern-uri de proiectare, precum Strategy si SimulationManager, pentru a face aplicatia mai modulara si mai usor de extins.

Dezvoltari ulterioare:

1. *Adăugarea de noi strategii de programare: În prezent, proiectul include doar o strategie de programare bazată pe cel mai scurt timp de așteptare. Puteți extinde proiectul prin adăugarea de noi strategii de programare, cum ar fi strategia bazată pe prioritate sau strategia bazată pe complexitatea sarcinilor.*
2. *Implementarea unui sistem de monitorizare a performanței: Pentru a evalua eficiența sistemului de server, puteți implementa un sistem de monitorizare a performanței care să colecteze date despre timpul de așteptare, timpul de procesare și timpul total de rulare a sarcinilor.*
3. *Adăugarea de funcționalități de configurare: Puteți extinde proiectul prin adăugarea de funcționalități de configurare, cum ar fi setarea numărului maxim de servere, numărul maxim de sarcini pe server și durata maximă a timpului de așteptare.*
4. *Implementarea unui sistem de raportare: Pentru a genera rapoarte detaliate despre performanța sistemului, puteți implementa un sistem de raportare care să ofere date despre numărul total de sarcini procesate, numărul total de servere disponibile și timpul mediu de așteptare.*
5. *Integrarea cu un sistem de stocare: Pentru a păstra un istoric al sarcinilor procesate, puteți integra proiectul cu un sistem de stocare pentru a stoca datele despre sarcini și performanța sistemului.*

# Bibliografie

Referintele pe care le-am consultat pe parcursul implementarii temei sunt:

1. [*https://dsrl.eu/courses/pt/*](https://dsrl.eu/courses/pt/)
2. [*https://www.geeksforgeeks.org/java-program-to-write-into-a-file/*](https://www.geeksforgeeks.org/java-program-to-write-into-a-file/)
3. [*https://online.visual-paradigm.com/app/diagrams/#diagram:proj=0&type=ClassDiagram&gallery=/repository/f31a486f-8471-48db-8dbe-35c94ad9812c.xml&name=UML%20Class%20Diagram%20Example%3A%20Phone%20Book*](https://online.visual-paradigm.com/app/diagrams/#diagram:proj=0&type=ClassDiagram&gallery=/repository/f31a486f-8471-48db-8dbe-35c94ad9812c.xml&name=UML%20Class%20Diagram%20Example%3A%20Phone%20Book)
4. [*https://www.geeksforgeeks.org/java-threads/*](https://www.geeksforgeeks.org/java-threads/)