DOCUMENTATIE

TEMA 2

APLICATIE PENTRU GESTIONAREA COZILOR FOLOSIND THREAD-URI (FIRE DE LUCRU) SI MECANISME DE SINCRONIZARE

NUME STUDENT: Alexandru Andrei-Denis

GRUPA: 30221

# CUPRINS

[1. Obiectivul temei 3](#_Toc95297885)

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

[3. Proiectare 3](#_Toc95297887)

[4. Implementare 3](#_Toc95297888)

[5. Rezultate 3](#_Toc95297889)

[6. Concluzii si dezvoltari ulterioare 3](#_Toc95297890)

[7. Bibliografie 3](#_Toc95297891)

# Obiectivul temei

* 1. **Obiectivul principal**

Obiectivul principal al temei este reprezentat de crearea unei aplicatii pentru implementarea unui tip de simulator dedicat analizei unui sistem de gestionare a perechilor client-coada. Mai exact, vorbim despre un numar de N clienti ce necesita obtinerea/realizarea unui serviciu (referit de-a lungul implementarii cu termenul de Task), clienti ce se aseaza la Q cozi, asteapta pana cand le vine randul de a fi seriviti apoi parasesc coada. Pe langa dinamica task-urilor in cozi de-a lungul unui interval de timp, se vor studia si timpul mediu de asteptare in coada pentru a fi servit, durata medie de servire a clientului si „ora de varf” a simularii (momentul de timp in care in cozi se afla cel mai mare numar de clienti).

* 1. **Obiective secundare**
* **Dezvoltarea de scenarii si use-case-uri -** In cadrul computerelor, un scenariu reprezinta succesiunea de interactiuni „foreseeable” dintre roluri(cunoscuti in UML - Unified Modeling Language - si sub numele de „*actori*”) si sistemul reprezentat de obicei de catre computer. In cazul acestei teme, rolul este reprezentat de utilizatorul aplicatiei. Fiecare scenariu are un scop(goal). Asemanator, un use-case este o lista de actiuni sau evenimente care definesc un set de interactiuni dintre *actor* si sistem. Mai multe use-case-uri se pot grupa pentru a defini un scenariu. - Capitolul 2
* **Alegerea structurilor de date -** Gasirea unor structuri de date adecvate pentru a duce la capat obiectivul principal - Capitolul 3
* **Impartirea pe clase -** Folosirea unei arhitecturi MVC (Model-View-Controller) pentru a proiecta o interfata grafica (GUI - Graphic User Interface) - Capitolul 3
* **Dezvoltarea algoritmilor -** Capitolul 3
* **Implementarea solutiei -** Descrierea fiecarei clase cu campurile si metodele sale importante, precum si descrierea interfetei pentru utilizator - Capitolul 4
* **Testarea -** Descriera catorva scenarii de testare a comportamentului sistemului, pentru un diferite cantitati de clienti si cozi.

# Analiza problemei, modelare, scenarii, cazuri de utilizare

Utilizarea programului presupune introducerea de catre utilizator a datelor teoretice ale simularii, urmand a analiza in timp-real transformarile pe care cozile le sufera pana la finalizarea simularii, in functie de strategia aleasa. Particularitatile simularii pe care trebuie sa le seteze utilizatorul sunt: numarul de clienti sositi, numarul de cozi deschise, intervalul de timp maxim in care se petrece simularea, intervalul de timp corespunzator „sosirii” clientilor (arrival time interval) si intervalul limitator al duratei de procesare si realizare a serviciului solicitat de acestia (service time interval).

Preluarea si efectuarea acesor operatii se va efectua dupa apasarea butonului „start simulation”, strategia dupa care aplicatia se va ghida fiind cea de „shortest-time”, insemnand deci ca noii clienti vor fi asezati automat la coada cu cel mai mic timp de asteptare. Pentru functionarea corecta a aplicatiei, utilizatorul este responsabil cu introducerea unor date initiale corecte.

# Proiectare

* 1. **Alegerea structurilor de date ­**

Structurile de date principale folosite in dezvoltarea aplicatiei au fost listele, si datorita caracterului concurent al mecanismului de procesare, BlockingQueue-urile. Un BlockingQueue este o structura de date de tip coada, care restrictioneaza si blocheaza extragerea si introducerea elementelor din ea, in cazul in care este goala sau, respectiv s-a atins limita maxima de elemente. Un fir de lucru ce incearca sa scoata un element dintr-o coada goala va fi blocat pana cand un alt fir de lucru va introduce in coada respectiva un element. Analog, acelasi lucru se va intampla cu introducerea unui element peste limita maxima de elemente posibile.

Sa presupunem ca ne aflam intr-o institutie publica, cum ar fi spre exemplu o gara. Utilizarea BlockingQueue-urilor are loc in intermediul clasei Server. Un obiect de tipul Server este reprezentarea virtuala a unui ghiseu din realitate, in fata caruia se creeaza o coada cu clienti ce doresc a fi serviti. Clientii sunt reprezentati de obiectele de tip Task, carora le este asociat un timp de „venire in institutie” si un timpul necesar satisfacerii cererii (ex: durata de timp de la plasarea comenzii pana la tiparirea biletului de tren). Aceste doua perioade se vor genera la intamplare pentru fiecare Client, pentru a translata pe cat de realist se poate dinamica cererilor dintr-o institutie reala.

* 1. **Impartirea pe clase ­­-** Diagrama UML

O imagine care conține text, electronice, negru

Descriere generată automat

Pachetele in care au fost imparite clasele prezentate in diagrama UML de mai sus sunt

* Pachetul GUI (Graphical User Interface) - Clasa View si clasa Controller, responsabile pentru aspectul si functionarea interfetei grafice
* Pachetul Business Logic - Clasele SimulationManager, Scheduler, Selection Policy, ShortestQStrategy, TimeStrategy si interfata Strategy, responsabile pentru descrierea corecta si realista a mecanismului sistemului de procesare si administrare a cozilor si clientilor
* Pachetul Model - Clasele Server si Task. Clasa Task contine detaliile necesare pentru descrierea corecta a clientilor, in timp ce clasa Server este o clasa de tip „runnable” ce contine un BlockingQueue de Taskuri si metoda run, responsabila cu simularea dinamicii cozii.
* Pachetul Main - Clasa Main care face legatura dintre cele 3 piese de baza ale arhitecturii Model View Controller.
  1. **Algorimti demni de mentionat**

Pentru a descrie un sistem cat se poate de asemanator cu o situatie reala, aplicatia este nevoita sa functioneze pe principiul de relationare intre Producator si Consumator. Concret, a fost nevoie de o unitate care sa se ocupe de generarea aleatoare a Clientilor si de una responsabila cu distribuirea acestora in cozi, dar si cu „servirea” (eliminarea) acestora odata ce timpul simularii a ajuns in punctul favorabil.

# Implementare

* 1. **Pachetul Model**

Clasa **Task**

Dupa cum am mentionat anterior, un obiect apartinator clasei Task reprezinta Clientul din situatia reala. Acesta este caracterizat printr-un ID unic(in cazul de fata, un numar intreg, autoincrementarea avand loc intr-o clasa ulterioara), momentul de timp in care Clientul ajunge in interiorul simularii (Arrival Time, *arrTime)* - similar cu intrarea in gara, in exemplul dat anterior - cat si timpul necesar satisfacerii cererii acestuia (Execution time, *exTime*), ambele fiind de asemenea reprezentate de numere intregi pentru a simplifica simularea, chiar daca in cazul real duratele sunt variate si se pot exprima mai usor in minute si secunde. Ultima variabila instanta a clasei este tot de tip intreg, responsabila de contorizarea timpului petrecut de client in cadrul cozii pana cand ii vine randul (*waitingTime)*.

Restul clasei Task este populata cu metode de tip setter si getter, necesare pentru a obtine informatii imporante despre Client(Task) datorita ascunderii acestora prin intermediul incapsularii (ex: *private* int ID). Pe langa acestea, clasa contine si o metoda de comparare intre Task-uri, comparatia fiind facuta intre timpul de sosire al fiecarui dintre cele doua task-uri, returnand diferenta.

Clasa **Server**

Clasa Server contine detaliile principale legate de o coada, cat si metodele minime necesare gestionarii clientilor din ea.

Pentru a usura accesul la lista cu clienti asezati in coada, BlockingQueue-ul (despre care am vorbit anterior - vezi cap. 3.1.) cu Task-uri este declarat public, fiind accesibil din toate clasele aplicatiei, incalcand astfel principiul de incapsulare (acest lucru va putea fi rectificat in dezvoltarile ulterioare ale aplicatiei).

Pentru a rezolva problemele ce apar odata cu lucrul in mod concurent si de sincronizare, una din solutiile ce sunt recomandate este folosirea de principiul de Atomicitate. O actiune atomica este de fapt o actiune care se realizeaza „cu totul deodata” si nu poate fi oprita in timpul realizarii, evitand astfel interactiunile mai multor fire de lucru asupra unei zone de memorie. De aceea, variabila instanta waitingPeriod este de tip AtomicInteger. Pe langa waitingPeriod, despre un „ghiseu” de tip Server se mai pot cunoaste detalii cum ar fi: numarul de clienti serviti in total, timpul total petrecut pentru a servi toti acesti clienti, dar si timpul total petrecut de clienti in coada. Toate aceste detalii sunt reprezentate in variabile instanta de tip intreg.

Metode importante:

Metoda addTask se ocupa, sugestiv, de adaugarea in coada de clienti a unui client nou, dar si de actualiarea timpului total de asteptare, adaugand timpul necesar realizarii solicitarii la timpul total curent.

Metoda run este o metoda caracteristica interfetei Runnable, implementata si de catre thread-uri. Metoda se ocupa de simularea trecerii timpului in intermediul Serverului si de eliminarea clientilor al caror serviciu a fost realizat la momentul de timp curent.

* 1. **Pachetul BusinessLogic**

Interfata **Strategy** si Clasele **TimeStrategy** si **ShortestQStrategy**

Aplicatia este creata in ideea de a putea gestiona cozile in diferite moduri, bazate pe diferite seturi de reguli de comparatie, numite in cazul de fata Strategii.

Strategy este o interfata ce contine doar metoda de addTask, responsabila cu adaugarea in interiorul unui Server al unui Task. Interfata este implementata de ambele Clase mentionate in antet.

Clasa ShortestQStrategy nu contine variabile instanta, in schimb aceasta suprascrie metoda mentionata anterior. Ideea de baza este cautarea in lista de Cozi (Servers), coada cu lungimea cea mai scurta, la care sa se aseze Clientul.

Similar, Clasa TimeStrategy se ocupa de acelasi lucru, diferenta fiind gasita in modul de cautare a Serverului oportun, in cazul acesta cautandu-se Serverul cu perioada de asteptare cea mai mica (waitingPeriod, discutata anterior).

Enumeratia **SelectionPolicy**

Rolul acestei enumeratii este de a gestiona mai usor schimbul de Strategii pe care se bazeaza functionalitatea aplicatiei.

Clasa **Scheduler**

Un obiect de tip Scheduler este echivalentul „insititutiei” despre care s-a vorbit anterior. Acesta contine o lista de Cozi (Servere), o strategie, un spatiu limitat in care se pot aseza ghiseele (in realitate nu putem avea un numar infinit, asa ca numarul de servere este limitat de variabila instanta maxNoServers, iar numarul de clienti intr-o coada este limitat de variabila maxTasksPerService). In timp ce locatia fizica a serverelor e reprezentata de lista de Servere, executia acestora este reprezentata de lista de Fire de lucru, fiecare fir de lucru fiind creat pe baza unui server, in constructor.

Metode importante:

Metoda changeStrategy este responsabila cu schimbarea strategiei de lucru a aplicatiei, in functie de un parametru de tipul enumeratiei prezentata anterior.

Metoda getCrtClients se ocupa de obtinerea numarului total de clienti aflati la cozi la momentul curent de timp, in timp ce areAllQueuesEmpty verifica daca toate cozile sunt goale, fiind o metoda pentru incheierea simularii inainte de a se ajunge la timpul maxim dat de utilizator.

Metodele getAvgTime si getSchedulerTotalClientsServed au fost create pentru usurarea obtine datelor necesare crearii unei statistici finale, in care sa fie prezentate detalii precum timpul mediu petrecut la coada sau lungimea de timp medie a serviciilor.

Clasa **SimulationManager**

Clasa SimulationManager este cea mai mare si bogata dintre clasele aplicatiei, ce se ocupa in principiu de „trecerea timpului” si de obtinerea detaliilor legate de elementele descrise pana acum.

Variabilele instanta ale clasei sunt datele introduse de catre utilizator prin intermediul interfetei grafice, toate de tip intreg (ex: numar de servere, numar de clienti, interval de simulare, interval de sosire, interval de lungime temporala a serviciului cerut).

Pe langa acestea, se mai gasesc lista de Clienti sub forma de ArrayList, Scheduler-ul descris de clasa de mai sus, si elementele de GUI (un view si un controller).

Metode importante:

Metoda generateRandomTask, sugestiv, genereaza un numar de N clienti cu diferiti timpi de sosire si executie, pe baza datelor introduse de user, asezandu-le in lista de Clienti, sortati dupa timpul de sosire. Metoda este apelata prin intermediul constructorului.

Metodele getTotalSerTime si getQSerTimeRemaining sunt metode ce parcurg listele de taskuri, respectiv de servere, pentru a obtine date relevante pentru generarea statisticilor de la final.

De-a lungul simularii, la fiecare moment de timp trecut, se afiseaza in timp real dinamica listei de clienti si a cozilor prin intermediul metodei getQueuesInfo. Aceste date sunt plasate atat intr-o zona de text in GUI, cat si scrise in fisierul logs.txt cu ajutorul functiilor createFile si appendFile.

Metoda run asigura functionarea continua a simularii pana cand momentul maxim de timp este atins sau pana cand nu mai exista niciun client neservit in cadrul simularii.

* 1. **Pachetul GUI**

Clasa **View**

In clasa View sunt instantiate cele 7 TextField-uri pentru datele de intrare, precum si butonul pentru inceperea simularii, si o zona de text navigabila.

Metodele din clasa View constau in preluarea datelor de intrare sub forma de String din TextField-uri si parsarea acestora catre tipul intreg.

Clasa **Controller**

Clasa este responsabila cu crearea unei legaturi intre Model si View fara ca acestea sa „stie” una de alta. In interiorul acesteia exista o subclasa pentru definirea Listenerelui pentru buton din interfata grafica.

* 1. **Clasa Main**

public class Main {  
 public static void main(String[] args) {  
  
 View view = new View();  
 Controller controller = new Controller(view);  
 view.setVisible(true);

}

# 

# Rezultate

O imagine care conține masă

Descriere generată automat

O imagine care conține masă

Descriere generată automat O imagine care conține masă

Descriere generată automat

Average Waiting time: 1.0

Average Service time: 3.25

Peak hour: 4

# Concluzii si dezvoltari ulterioare

In urma analizarii codului si testarii aplicatiei, personal concluzionez ca aceasta este una total functionala atata timp cat datele de intrare respecta o configuratie asemanatoare cu realitatea.

Ce am invatat din aceasta tema? Am reusit sa imi fac o idee despre folosirea firelor de lucru si despre sincronizare.

In cazul dezvoltarii ulterioare, aplicatia ar putea benificia de o interfata grafica mai bogata din punct de vedere al aspectului. Din punct de vedere logic, ar trebui implementat si un sistem de verificare a datelor introduse de utilizator si afisarea unor avertismente in cazul introducerii eronate.

O alta dezvoltare ar putea fi gasirea unui strategii mai eficiente, concentrata asupra maximizarii productivitatii, cum ar fi implementarea unui sistem de programari.

# Bibliografie

<https://softwareengineering.stackexchange.com/questions/347498/difference-between-scenario-and-use-case>

<https://dsrl.eu/courses/pt/materials/A2_Support_Presentation.pdf>

<https://www.javaprogramto.com/2020/04/java-collection-sort-custom-sorting.html>

<https://stackoverflow.com/questions/4818699/practical-uses-for-atomicinteger>

<http://tutorials.jenkov.com/java-concurrency/blocking-queues.html>

<https://howtodoinjava.com/java/multi-threading/atomicinteger-example/>