DOCUMENTATIE

TEMA *2*

NUME STUDENT: RADU DARIUS - RAZVAN

GRUPA: 30225

# 

# CUPRINS

[1. Obiectivul temei 3](file:///D:\Faculta\AN2_SEM2\TP\Laborator\PT2021-2022_Documentation_Template.doc#_Toc95297885)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 3](file:///D:\Faculta\AN2_SEM2\TP\Laborator\PT2021-2022_Documentation_Template.doc#_Toc95297886)

[3. Proiectare 3](file:///D:\Faculta\AN2_SEM2\TP\Laborator\PT2021-2022_Documentation_Template.doc#_Toc95297887)

[4. Implementare 5](file:///D:\Faculta\AN2_SEM2\TP\Laborator\PT2021-2022_Documentation_Template.doc#_Toc95297888)

[5. Rezultate 5](file:///D:\Faculta\AN2_SEM2\TP\Laborator\PT2021-2022_Documentation_Template.doc#_Toc95297889)

[6. Concluzii 8](file:///D:\Faculta\AN2_SEM2\TP\Laborator\PT2021-2022_Documentation_Template.doc#_Toc95297890)

[7. Bibliografie 9](file:///D:\Faculta\AN2_SEM2\TP\Laborator\PT2021-2022_Documentation_Template.doc#_Toc95297891)

# Obiectivul temei

Obiectivul temei a fost sa proiectam un simulator de cozi. Primim ca date de intrare de la utilizator, numarul de clienti, numarul de cozi disponibile, timpul maxim de rulare a simularii, precum si minimul si maximul de procesare a clientilor.

Aplicatia genereaza aleator un set de clienti, respectandu-se datele de intrare, care urmeaza apoi a fi procesati in functie de 2 strategii (in functie de cel mai scurt timp de asteptare la coada sau in functie de cea mai scurta coada). Strategia este aleasa, de asemenea, de utilizator.

De asemenea, aplicatia trebuie sa calculeze timpul mediu de asteptare si timpul mediu de servire pentru clientii.

Datele vor fi citite si afisate pe o interfata grafica.

# Analiza problemei, modelare, scenarii, cazuri de utilizare

Modelarea problemei a fost facuta in mare parte dupa exemplul alaturat prezentarii temei, astfel incat cozile sunt modelate ca servere, care primesc task-uri (clienti) pe care trebuie sa le proceseze. Serverele sunt monitorizate si primesc task-uri de la un scheduler, care prin atributul strategy consulta timpul de asteptare la fiecare server in parte si ia o decizie cu privire la coada careia sa ii asigneze urmatorul client pentru a facilita eficienta.

Use case: simularea unor cozi de la magazin, la care asteapta clienti.

Actor principal: utilizator

Pasi:

* Introducem datele de simulare.
* Alegerea strategiei de simulare.
* Pornirea simularii, apasand butonul de Start Simulation.
* Se deschide fereastra pentru afisarea datelor.
* La fiecare moment de timp, se adauga sau scoate un client din coada. (lucru vizibil in tabel).
* Afisarea timpului mediu de asteptare.

# Proiectare

**Diagram, schematic

Description automatically generatedDiagrama de pachete** folosite in cadrul aplicatiei noastre poate fi obseravata in dreapta. Este o diagrama clasica pentru arhitectura MVC, ce prezinta dependentele pachetelor (view-ul este dependent de model, iar controller-ul este dependent atat de model, cat si de view).

Fiecare pachet contine clase ce au un anumit rol bine definit.

In pachetul „controller” exista clasa „SimulationManager”, ce are rolul de a coordona desfasurarea simularii (creierul simularii), clasa „Scheduler” ce contine linsta de cozi si se ocupa cu schimbul intre strategiile de simulare, interfata „Strategy” si clasele „TimeStrategy” si „ShortestQueueStrategy”, ce implementeza interfata „Strategy”, pentru a se ocupa de selectia cozii in care vom insera clientul.

In pachetul „view”, avem doua clase „DisplayData”, care e folosita pentru afisarea datelor pe interfata grafica si „HomePage”, pentru inserarea datelor si pornirea simularii.

In pachetul „model” avem clasele, „Task”, ce reprezinta modelarea efectiva a clientului, „Server”, care este coada si un enum „SelectionPolicy”, pentru schimbarea intre strategii.

**Diagrama UML:**

**Diagram, schematic

Description automatically generated**

Clasa „Task” => reprezinta modelarea clientului nostru, ce va fi generat si repartizat intr-o anumita coada, in functie de strategia aleasa. Clasa, pe langa constructor si metodele de toString, equals si Setters and getters, contine metoda de „decremenetServiceTime”, ce este folosita pentru a decremeneta serviceTime-ul fiecarui client la fiecare moment de timp, pentru a se observa la afisare cat timp mai are de stat in coada.

Clasa „Service” => reprezinta coada de asteptare pentru clienti. Clasa implementeaza interfata Runnable (pentru a fi trimisa dupa catre un thred). Pentru atribute (lista de task-uri si numarul de asteptare) folosim tipuri ce se autosincronizeaza (BlockingQueue si AtomicInteger), acestea vor rezolva problema concurentei dintre Thred-uri. De asemenea avem metoda addTask, ce adauga un task in coada si incrementeaza waitingTime-ul acestei cozi. Evident, fiind un Runnable, avem functia de Run ce implementeaza functionalitatea cozii (servirea clientiilor pe rand).

Clasa „HomePage” => este o interfata grafica, in care vom introduce datele de simulare si de unde vom porni simularea.

Clasa „DisplayData” => este o interfata grafica, in care vom afisa coziile, si in timp real fiecare client care intra/iese din cozi, precum si numarul clientilor ramasi, timpul mediu de servire si de asteptare. Afisarile se fac cu metodele printData().

Clasa „SimulationManager” => este creierul mare ce coordoneaza toata functionalitatea aplicatiei. Clasa implementeaza si ea interfata runnable.

Avem metodele: controllerHomePage(), ce se ocupa de Listener-ul de pe butonul de start al simularii. Aceasta metoda se ocupa de legatura dintre partea de view si partea de control a aplicatiei.

Metoda readData(), citeste datele introduse de utilizator si suprascrie parametrii de simulare.

Metoda generateTasks(), genereaza n clienti random.

Metoda removeTasks(), scoate clientii din coada cand se termina servirea lor

Metoda decrementQueue(), se ocupa de decrementarea serviceTime-urilor clientiilor din fiecare coada.

Metodele printQueue() si printQueueF() care se ocupa de printarea clinetiilor din cozi atat pe ecran cat si in fisierele de log.

Clasa „Scheduler” => contine lista de servere si se ocupa de crearea strategiei alese de noi.

Clasa „ShortestQueueStrategy” => implementeaza interfata „Strategy”. Implementeaza functia „addTask”, care se ocupa de alegerea cozii celei mai scurte, pentru a adauga task-ul.

Clasa „ShortestQueueStrategy” => implementeaza interfata „Strategy”. Implementeaza functia „addTask”, care se ocupa de alegerea cozii cu cel mai mic timp de asteptare, pentru a adauga task-ul.

De asemenea avem interfata „Strategy”, ce contine metoda addTask, si enum-ul SelectionPolicy, ce se ocupa de separarea celor 2 strategii.

**Structuri de date folosite**:

Pentru rezolvarea concurentei dintre Thred-uri, am folosit LinkedBlockingQueue.

Pentru stocarea serverelor am folosit LinkedList.

Pentru stocarea tuturor task-urilor generate am folosit ArrayList.

# Implementare

Metode din pachetul ModelPackage:

1. Clasa TASK:

Metoda decrementServiceTime(): aceasta metoda, dupa cum am spus si la punctul anterior, este folosita pentru decrementarea ServiceTime-ului, pentru ca in momentul trecerii unui moment de timp, sa fie destul de clar fiecare client cat mai are de stat in coada. Functionaliatea este simpla, folosim operatorul de decrementare (--) pe atributul „serviceTime”, al task-ului referit.

Metoda equals(Object obj): am inclus-o pentru ca am crezut ca voi avea nevoie de o oarecare comparare intre task-uri, dar acest lucru nu a fost necesar, deci functia nu este folosita. Poate fi stearsa.

Metoda toString(): este folosita pentru a usura printarea fiecarui task. Functia este cea clasica, generata de mediul de lucru.

De asemenea, in clasa task mai avem getters pentru fiecare atribut. Folosindu-se incapsularea, a fost nevoie de implementarea acestor gettere. Setter-ele nu au fost necesare, din pricina faptului ca nu suprascriem niciun atribut al task-ului, in exteriorul clasei de Task.

1. Clasa SERVER:

Metoda addTask(Task newTask): are rolul de a introduce un task in interiorul listei de task-uri din interiorul server-ului, adica de a aseza un client la coada. De asemenea mai incrementeaza timpul de asteptare al cozii, cu timpul de servisare al clientului nou adaugat. Clasa server fiind un runnable, va trebui sa avem grija la problema concurentei, deci implementarea listei de task-uri, o vom face folosinf un LinkedBlockingQueue, iar pentru pastrarea timpului de asteptare vom folosi un AtomicInteger. Aceste structuri se ocupa automat de problema concurentei. Introducerea intr-un BlockingQueue se face cu metoda „put”, dar trebuie sa inconjuram apelul metodei cu o structura try-catch, pentru a testa exceptia: InterruptedException.

De asemenea, am pus synchronized acestei metode, pentru a fi complet sigur ca problema concurentei este rezolvata pe deplin.

Metoda toString(), ce nu este folosita in mod special, am folosit-o pentru testare, in timpul procesului de implementare.

Getters: pentru ca folosim incapsularea este necesara folosirea getter-elor pentru accesarea campurilor private.

Pentru ca clasa server este o clasa runnable, avem metoda de run, specifica oricarei clase ce implementeaza interfata runnable. Metoda run, se ocupa cu scoaterea din coada a clientului client si procesarea acestuia. Am implementat astfel: scoatem clientul din coada, cu metoda „take”, aceasta metoda se ocupa sa nu scoata null in caz ca coada este nula, apoi punem thred-ul la somn atat timp, cat clientul se proceseaza (am avut grija sa inmultesc serviceTime-ul clientului cu 1000L, pentru ca argumentul metodei sleep, este de ordinul milisecundelor).

Metode din pachetul ControllerPackage:

1. Clasa SCHEDULER

Metoda changeStrategy(SelectionPolicy policy): aceasta metoda se ocupa de crearea strategiei in functie de care vom alege coada in care adaugam task-ul in coada. SelectionPolicy, este un enum (prezent in pachetul ModelPackage), ce contine tipurile SHORTEST\_QUEUE si SHORTEST\_TIME. In functie de acestea se va face selectia si crearea clasei de strategie corespunzatoare pentru strategia selectata de utilizator. Selectia se face simplu cu un if. Daca policy este SHORTEST\_QUEUE, atributul strategy din scheduler va fi instantiat cu constructorul clasei ShortestQueueStrategy, iar daca este SHORTEST\_TIME, va fi instantiat cu constructorul clasei TimeStrategy.

Metoda dispatchTask(Task task): aceasta metoda se ocupa cu apelarea functiei ce face selectia efectiva a cozii in care trebuie sa adaugam clientul. Aceasta functie are 2 implementari. Implementarea care va fi folosita, va fi cea din clasa cu care a fost instantiat atributul strategy, in metoda changeStrategy.

De asemenea avem un getter, pentru returnarea listei de servere. Il folosim pentru parcurgerea tuturor serverelor in afara clasei de schaduler (la afisari, decrementari sau scoateri din cozi).

1. Clasa TimeStrategy. Aceasta clasa implementeaza interfata Strategy, ce contine definitia metodei addTask

Metoda addTask (LinkedList<Server> servers, Task task): se ocupa cu selectarea cozii in care vom insera task-ul primit ca parametru. Am implementat astfel: parcurgem lista de servere (cozi) si calculam minimul dintre waitingTime-urile fiecarei cozi, precum si stocam serverul care are waitingTime-ul minim intr-o variabila s. La final, daca variabila este diferita de null vom apela functia de addTask din interiorul clasei server, pentru a adauga task-ul in coada efectiva.

1. Clasa ShortestQueueStrategy. Aceasta clasa implementeaza interfata Strategy, ce contine definitia metodei addTask

Metoda addTask (LinkedList<Server> servers, Task task): se ocupa cu selectarea cozii in care vom insera task-ul primit ca parametru. Am implementat astfel: parcurgem lista de servere (cozi) si calculam lungimea minima dintre lungimile cozilor, precum si stocam serverul care are lungimea minima intr-o variabila s. La final, daca variabila este diferita de null vom apela functia de addTask din interiorul clasei server, pentru a adauga task-ul in coada efectiva.

1. Clasa SimulationManager

Aceasta clasa este controller-ul mare, ce controleaza atat interfata grafica, cat si fiecare proces din interiorul simularii.

Metoda controllerHomePage(HomePage homePage): aceasta metoda este apelata din interiorul constructorului SimulationManager-ului si are rolul de a controla evenimentele din frame-ul initial (de introducere de date pentru simulare). In loc sa mai fac o clasa pentru controller-ul frame-ului, am ales sa fac o simpla clasa, deoarece avem doar un Listener pe un singur buton. Acest listener l-am implementat cu o functie lambda.

La apasarea butonului de pe frame, va trebui sa se citeasca datele introduse si setate atributele SimulationManagerului, pentru a avea datele cu care sa lucreze. Acest lucru este facut prin apelarea metodei readData, pe care o vom explica. Apoi vom seta strategia selectata in scheduler. Generam cele n task-uri random, prin apelarea metodei generateTasks(), de asemenea va fi discutata in randurile ce urmeaza.

In final dupa ce am apasat butonul, vrem ca pagina de introdus date sa se inchida si sa se deschida pagina in care vizualizam cum se desfasoara simularea. (dam dispose la homePage si creem un displayData object). Inainte sa inceapa simularea propriuzisa trebuie sa creem coziile, cu un for de la 0 la numarul de servere, creem un server, ce-l adaugam in lista de servere, apoi pornim un thred ce are ca parametru server-ul nostru. La final pornim simularea, prin pornirea unui thred ce are ca parametru fix obiectul la care ne referim „this”, deoarece noi ne aflam in clasa SimulationManager (obiect creat in main).

Metoda readData(): seteaza fiecare parametru de simulare, cu varianta convertita in integer din textField-urile din homePage. Nimic foarte complicat, doar niste atribuiri si apel de metoda parseInt().

Metoda generateTasks(): intr-un for de la 0 la numarul de clienti, vom genera 2 numere aleatoare (avem grija sa fie pozitive, folosind Math.abs()), unul pentru arrival time si unul pentru service time. Pentru ca valoarea sa fie in intervalul specificat de noi vom face astfel: pentru a fi mai mica decat limita superioara, vom face „% limita superioara”, iar pentru a fi mai mare decat limita inferioara, in cazul in care este mai mic decat limita inferioara, vom mai adauga la numarul generat limita inferioara, pentru a translata numarul in interval.

De asemenea, in metoda asta, calculam timpul global de asteptare pentru a putea opri mai repede simularea, in cazul in care cozile se folesc mai repede. (adunam service time-urile). Calculam si media timpurilor de servire a clientilor, facand media aritmetica dintre service time-urile clientilor.

La final de tot, sortam lista de clienti in functie de arrival time, folosind o lambda function.

Metoda run(): se ocupa de coordonarea efectiva a simularii. Am implementat-o astfel: cat timp suntem inca in timpul permis, si nu s-au terminat clientii de pe hol, vom cauta acei clienti ce au arrivalTime-ul egal cu timpul curent. In cazul in care se gasesc astfel de clienti, scheduler-ul ii va trimite pentru a fi adaugati in coada, iar apoi ii scoatem din lista de clienti generati, pentru a putea afisa si clientii ce asteapta pe hol. Evdent decrementam de fiecare data cand gasim unul, variabila remainingClients, care numara cati clienti mai avem pe hol.

Dupa ce am trimis task-urile ce au arrivalTime-ul egal cu timpul curent, vom printa in fisierele de log, cat si in consola (pentru a vizualiza mai usor ce se intampla in simulare in cazul in care nu vrem sa ne uitam pe interfata grafica, pentru testare), apeland functiile printQueuesF, respectiv printQueues. Afisam si pe interfata grafica, apeland metoda printData din clasa DisplayData. Apoi incrementam timpul curent, puntem thred-ul la somn pentru o secunda, iar dupa ce reporneste, inseamna ca a trecut o unitate de timp, deci trebuie sa decrementam serviceTime-ul fiecarui client pentru a vizualiza mai usor in fisierul de log cat timp mai are de stat, apeland functia decrementQueues, apoi scoatem clientii ce au terminat sa fie procesati, apeland metoda removeTasks. La final dupa ce se termina simularea, va trebui sa printam, atat in fisier cat si pe interfata, timpul mediu de servire si de asteptare. Nu uitam inchiderea fisierului de log la finalul simularii.

Metoda decrementQueues(): se ocupa de decrementarea serviceTime-ului fiecarui client la din cozi la fiecare moment de timp. Pentru fiecare server din lista de servere, vom parcurge lista de task-uri a acestuia, si decrementam serviceTime-ul, timpul global de asteptare si timpul de asteptare pt fiecare q.

Metoda removeTasks(): cand un client termina de a fi procesat, acesta trebuie sa fie scos din coada. Ne dam seama ca este terminat, uitandu-ne la serviceTime-ul clientului, daca acesta este 0, stim ca acesta a fost terminat, deci trebuie scos. Am parcurs lista de servere si am folosit metoda removeIf si o functie lambda pentru a da remove la client. (acest lucru rezolva problema concurentei si stergerea concurenta a task-urilor)

Metoda printQueuesF(int currentTime, FileOutputStream file): este folosita pentru a afisa in fisierul de log datele de simulare. Parcurgem fiecare server din lista de servere din schaduler, iar pentru fiecare server (coada), vom parcurge lista de task-uri si vom afisa datele acestora utilizand metoda „toString”.

Metoda printQueues(int currentTime): este folosita pentru a afisa in consola datele de simulare. Parcurgem fiecare server din lista de servere din schaduler, iar pentru fiecare server (coada), vom parcurge lista de task-uri si vom afisa datele acestora utilizand metoda „toString”.

Metode din pachetul ControllerPackage:

1. Clasa HomePage:

Graphical user interface, application, Word

Description automatically generatedIn clasa HomePage, avem doar metoda ce adauga un listener pe butonul de startSimulation si o metoda ce seteaza continutul comboBox-ului.

Este folosita pentru introducerea datelor de intrare pentru simulare, si arata astfel:

1. Clasa DisplayData:

Interfata grafica arata in felul urmator: Graphical user interface, text, application, email

Description automatically generated

Am creat un tabel (JTabel), ce are ca si linii coziile efective. In coloana A va fi afisat numarul cozii (exemplu: Q1, Q2, etc.), iar in coloana B vor fi afisati pe rand, clientii ce intra sau ies din coada.

Jos, dupa cum se poate observa, se afiseaza timpul curent, numarul de clienti ramasi in coada, timpul mediu de asteptare si de servire.

Metoda setNumColumns(int numOfServers): care seteaza de fapt numarul de randuri a tabelului nostru. Initial am vrut sa reprezint serverele ca fiind coloane, dar nu am putut redenumi coloanele, asa ca am schimbat abordarea, iar numele metodei am uitat sa il modific. De asemenea metoda scrie in celulele primei coloane numele si numarul cozii.

Metoda printData: are 2 implementari.

Una care primeste ca parametru un singur double, si este folosita pentru printarea numarului mediu de asteptare.

Una care primeste scheduler-ul (pentru lista de cozi), timpul curent, clinetii ramasi si timpul mediu de servire, pentru a le afisa pe interfata grafica. Afisarea se face asemanator cu afisarea in fisier sau consola, doar ca vom afisa in celula ce are linia egala cu numarul cozii si coloana 2.

# Rezultate

Am afisat rezultatele celor 3 teste specificate in cerinte in 3 fisiere de log: log1, log2 si log3.

Rezultatele generate de simulare, sunt corecte. Clientii sunt adaugati in coada la momentul corect si sunt scoti de asemenea odata ce termina a fi servit. De asemenea merg ambele strategii, algoritmul alege corect coada in care trebuie sa insereze clientul.

Calculul timpiilor medii (servire si asteptare) este corect.

# Concluzii

Tema aceasta a fost mai dificila din cauza faptului ca a trebuit sa ne documentam mai mult despre Threduri si despre rezolvarea problemei concurentelor, dar de asemenea cred ca am avut un micut avantaj, deoarece am primit in materialele de laborator niste ghidari.

Consider ca a fost foarte de ajutor aceasta tema, deoarece am putut sa intelegem mai bine conceptul de Thread si conceptul de concurenta. Este un aspect foarte important in programare, un lucru de viitor si din acest motiv trebuie stapanit foarte bine.

# Bibliografie

stackoverflow.com