DOCUMENTATIE

TEMA *2*

NUME STUDENT: …Kiraly Nicole Elena…
GRUPA: ….30227……

CUPRINS

1. Obiectivul temei

Obiectivul principal al temei 2 este implementarea unui sistem de management al cozilor, la care trebuie adaugati clientii generati random, cu ajutorul threadurilor si structurilor de date care ajuta la sincronizare. Vom utiliza si o interfata grafica din care vom extrage datele introduse de utilizator, si le vom folosi in proiect.

Obiectivele secundare pentru indeplinirea obiectivului principal sunt:

- Analiza problemei si intelegerea cerintelor
- Realizarea generatorului de clienti random
- Realizarea interfetei grafice pentru introducerea datelor problemelor
- Afisarea rezultatelor simularii intr-un fisier si in timp real intr-o interfata, precum si a timpului mediu de asteptare, a timpului mediu de procesare, si a timpului maxim cand se afla cei mai multi clienti in coada.

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

Pentru rezolvarea acestei teme, trebuie sa analizam urmatorii pasi principali:

- Definirea problemei
- Intelegerea problemei si domeniul sau
- Analiza problemei
- Gasirea solutiilor pentru sub-probleme individuale
- Construirea solutiei finale

In cazul sistemului de management al cozilor, este necesar sa intelegem functionarea unei cozi, cum putem adauga clientii dupa cel mai scurt timp sau cea mai scurta coada, cum se poate calcula timpul mediu de asteptare sau cel de procesare.

Pentru inceput, se construiesc clasele din pachetul model, Task si Server. Apoi se creeaza clasa Scheduler din pachetul BusinessLogic pentru organizarea sistemului de management, mai apoi se construieste clasa SimulationManager pentru simularea sistemului de gestiune a cozilor. Clasele ConcreteStrategyTime si ConcreteStrategyQueue pentru alegerea strategiei dupa care vom adauga clientii in coada. In final, vom crea clasele pentru view in pachetul GUI si anume, SimulationFrame si Events pentru introducerea datelor de intrare si pentru testarea functionalitatii in timp real.

Deci, sistemul de management va primi datele de intrare-numberOfClients, numberOfServers, maxProcessingTime, minProcessingTime, minArrivalTime, maxArrivalTime, timeLimit, selectionPolicy unde se allege strategia dupa care vor fi adaugati clientii.

Pachetele au fost imapartite in functie de rolul claselor:

1. Model: Task si Server
2. BusinessLogic: Scheduler, SimulationManager, ConcreteStrategyTime, ConcreteStrategyQueue, SelectionPolicy si interfata Strategy

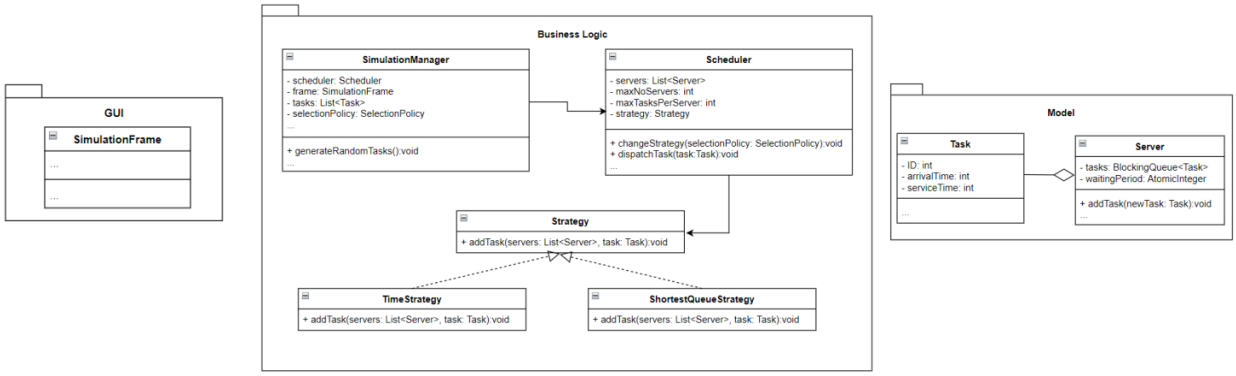
3. GUI: SimulationFrame si Events

Pentru utilizare, vom introduce datele de intrare in interfata grafica si vom apasa butonul start pentru inceperea simularii. Rezultatul va aparea intr-o noua fereastra, rulandu-se in tip real in fiecare secunda. In fisierul log_of_events,txt va aparea rezultatul final, precum si timpul mediu de asteptare, timpul mediu de procesare si peak hour. Se vor afisa clientii in cozile corespunzatoare, precum si cei care asteapta sa intre in cozi.

3. Proiectare

Se va imparti rezolvarea acestui sistem de management a cozilor in pachete si clase. Deci, primul pachet pe care il voi descrie este Model. Aici, am implementat clasele Task si Server. Mai apoi, avem pachetul BusinessLogic cu clasele Scheduler, SimulationManager, interfata Strategy, ConcreteStrategyTime si ConcreteStrategyQueue care ne ajuta sa alegem strategia dupa care vor fi adaugati clientii in cozi. In final, ne vom ocupa de pachetul GUI, unde se afla interfata grafica pentru vizualizarea rezultatelor si introducerea datelor de intrare.

Diagrama de pachete o voi importa din laborator:



USE-CASE diagram:

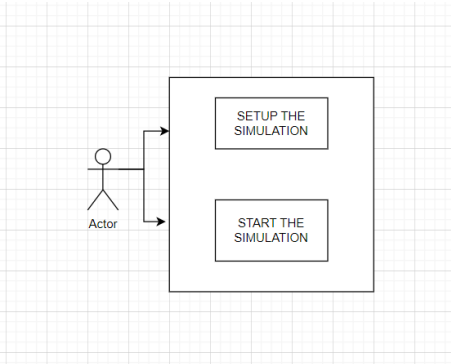
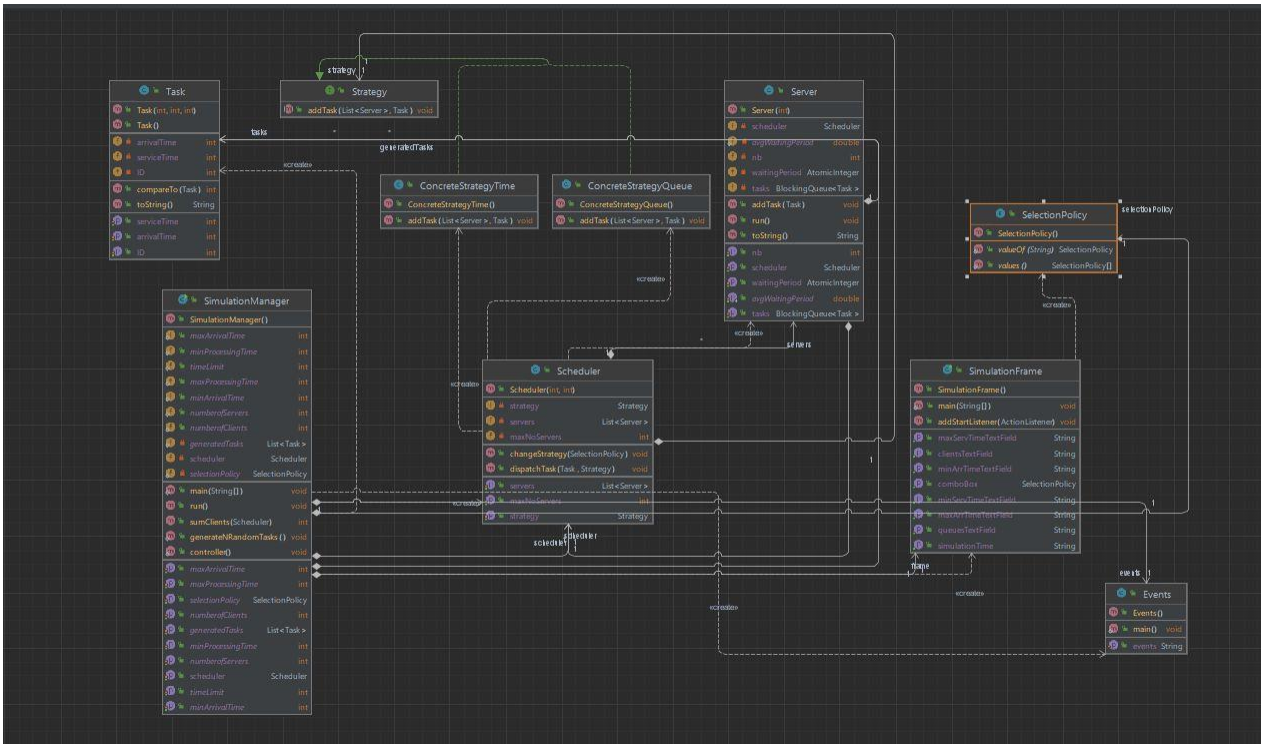
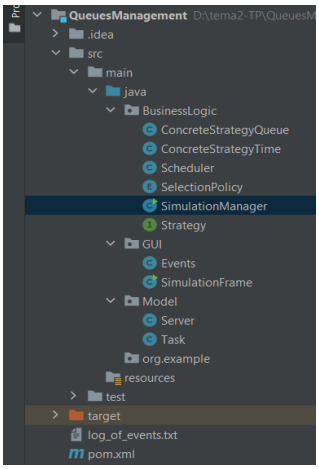


Diagrama UML a claselor:



4. Implementare

Implementarea acestei teme se bazeaza pe impartirea in pachetele corespuzatoare.



Clasa Task contine atributele: id(private int ID), arrivalTime(private int arrivalTime), serviceTime(private int serviceTime). ID reprezinta id-ul cozii, arrivalTime reprezinta timpul la care clientul ajunge la coada, serviceTime este timpul de procesare. Pentru toate aceste atribute am creat getteri si setteri pentru a ne ajuta sa le folosim pe parcurs. Aceasta clasa are doi constructori, unul fara parametri, iar altul cu parametrii, creand astfel un Task cu un unic id, arrivalTime si serviceTime generate random. Pe langa acestea, am creat metoda toString pentru afisarea clientilor intr-un mod interactiv (id, arrivalTime, serviceTime).

Clasa Server implementeaza clasa Runnable pentru crearea thread-urilor. Aceasta foloseste 4 atribute: nb-numarul cozii de tipul int, perioada de asteptare(waitingPeriod) de tipul AtomicInteger, coada pentru clienti(tasks) de tipul BlockingQueue<Task>, perioada medie de asteptare avgWaitingPeriod, scheduler de tipul Scheduler. Am folosit tipurile AtomicInteger si BlcokingQueue pentru a asigura sincronizarea, fiind necesare pentru corectitudine cand lucram cu fire de executie. Pe langa acestea am mai introdus o variabila statica si volatile pentru oprirea threadurilor, cand s-a ajuns la timpul maxim de simulare. De asemenea, si avgWaitingPeriod este tot static deodarece vom calcula timpul mediu de asteptare, aceasta modificandu-se constant. Clasa Server are un constructor cu parametrii pentru crearea Serverelor, dandu-i o valoare unica nb pentru a sti in ce coada ne aflam. Aici se initializeaza timpul de asteptare cu 0(new AtomicInteger(0) si new ArrayBlockingQueue<Task>() pentru initializarea cozilor). Am implementat si metoda toString pentru a afisa continutul cozii la rulare, In plus, fiecare atribut are metodele de getteri si setteri in cazul utilizarii pe parcurs a acestora, ele avand modificatorul de acces private. Metoda addTask va adauga un nou client in Server si va incrementa waitingPeriod-ul cu serviceTime-ul clientului dat ca si parametru. Tot aici se calculeaza suma fiecarui client care urmeaza sa intre in coada pentru calcularea timpului mediu de asteptare ca si suma dintre timpul de procesare al noului client si timpul de asteptare a celor dinaintea sa. Metoda run este folosita aici deoarece fiecare Server reprezinta un thread, iar clasa implementeaza interfata Runnable, iar tot ce se scrie in aceasta functie va fi executat de catre threadurile specifice. Se parcurge coada element cu element folosind metodele peek care ia elementul din capul cozii si poll care sterge elementul din varf. Metoda va pune in asteptare threadul prin metoda sleep pentru un timp egal cu timpul de procesare al clientului current.

Clasa Scheduler, care face parte din pachetul BusinessLogic, ne ajuta la organizarea cozilor si programarea lor. Aceasta contine o lista de servere (private List<Server> servers), numarul maxim de servere pe care le vom prelua din interfata SimulationFrame, si strategia dupa care vom adauga clientii pe care o vom prelua tot din interfata si numarul maxim de taskuri din server, variabila statica care va fi preluata din interfata. Constructorul clasei Scheduler are 2 parametrii numarul de servere si numarul maxim de taskuri pe server. Aici vom initializa lista de servere, vom atribui numarul maxim de servere si de taskuri si vom parcurge cu un for toate serverele pentru a crea threaduri pentru acestea si a le da start. Metoda changeStrategy ne ajuta la schimbarea strategiei dupa care vom adauga clientii in cozi. Metoda dispatchTask ajuta la adaugarea clientilor in coada in functie de strategia aleasa. Ultimele metode sunt getteri si setteri pentru atributele private ale metodei.

Urmeaza sa implementam clasele ConcreteStrategyTime si ConcreteStrategyQueue care implementeaza strategia Strategy unde gasim o singura metoda addTask pentru adaugarea clientilor. In ConcreteStrategyTime vom alege timpul minim dupa care adaugam clientii in coada. Parcurgem cu un for serverele si luam un minim la Integer.MAX_VALUE apoi il actualizam de fiecare data cand waitingPeriod-ul e mai mic, iar coada minima va fi stocata in minQueue. Daca minQueue se modifica atunci adaugam clientul in coada, altfel aruncam o exceptie deoarece coada e plina. La fel vom proceda pentru ConcreteStrategyQueue, doar ca in loc de timpul de asteptare vom pune numarul de taskuri existente in Server si vom cauta minumul intre size-urile cozilor si acolo vom adauga noul client.

Ultima clasa a pachetului BusinessLogic este SimulationManager care implementeaza si ea interfata Runnable. De aici se incepe simularea prin pornirea threadului principal. Acesta are atribute statice unde vom stoca datele de intrare introduse de la tastatura in interfata, acestea fiind timeLimit, maxProcessingTime, minProcessingTime, minArrivalTime, maxArrivalTime, selectionPolicy unde se afla strategia dupa care vom adauga clientii in coada, numberOfServers si numberOfClients. Alte atribute necesare sunt scheduler, avgServiceTime, interfetele frame si events, in events se vor afisa rezultatele si lista random de taskuri. In constructorul clasei SimulationManager vom initializa scheduler-ul dandu-i ca parametrii numberOfServers si numberOfClients. Aici avem si metoda generateNRandomTasks care va crea doua elemente random pe care le vom atribui timpului de sosire si timpului de procesare. Daca maxProcessingTime e mai mic ca minProcessingTime aruncam o exceptie deoarece intervalele sunt inversate. Daca maxArrivalTime < minArrivalTime aruncam de asemenea o exceptie pentru ca intervalele sunt

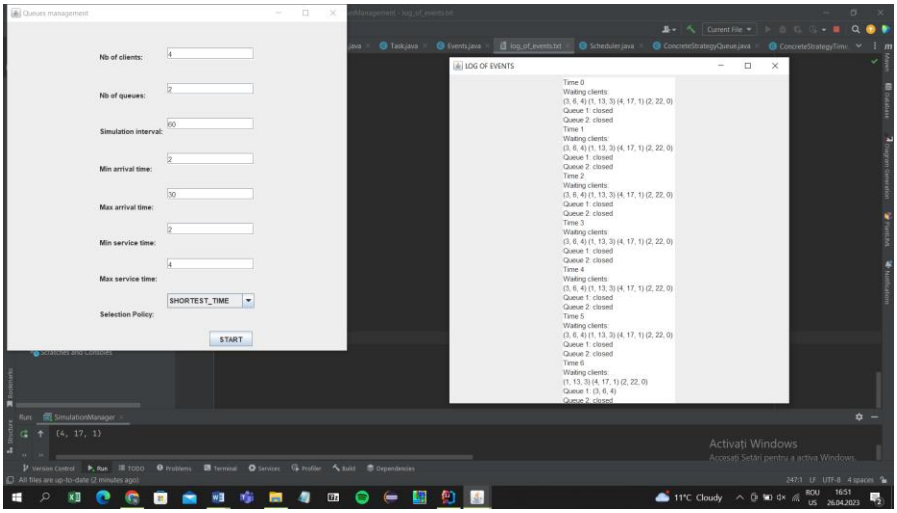
inversate. Apoi parcurgem fiecare client si atribuim valorile random generate. Aici vom calcula avgServiceTime ca si suma intre service time-urile tuturor clientilor adunate, iar la final se imparte la nuarul de clienti. Adaugam toata taskurile generate in generatedTasks, creand un nou task de fiecare data. Vom sorta lista de generatedTasks dupa timpul de sosire astfel incat timpul de sosire cel mai mic va fi primul. Metoda controller functioneaza intr-un mod asemanator cu un controller, aceasta va extrage informatiile din interfata, dupa apasrea butonului start. Varibilele statice vor primi datele din interfata si apoi se vor genera random clientii. Aici vom prinde exceptia si vom afisa mesajul in cazul in care intervalele au fost gresite. In caz de succes, se va crea un nou obiect de tipul SimulationManager si se va porni threadul principal. Threadul principal va apela functia run, unde se va crea rezultatul pe care il va scrie in fisier, precum si in fereastra a doua dupa apasarea butonului start. Aici vom calcula si peak Hour, adica la ce timp sunt cei mai multi clienti in cozi. Vom parcurge clientii cat timp timpul curent este mai mic dcat timpul de simulare. Daca timpul de sosire al unui client este egal cu timpul curent de simulare vom apela changeStrategy cu strategia din combo boxul din interfata, vom extrage strategia si vom adauga clientul in coada in functie de strategia rezultata. Apoi vom scoate din coada clientul. Vom calcula si peak hour cu ajutorul functie sumClients din scheduler care aduna numarul de clienti aflati in cozi la timpul curent, iar daca timpul curent are maximul mai mare decat cel de dinainte vom actualiza peakHour cu timpul curent. Se va afisa in fisier rezultatul simularii, precum si peak hour, avgWaititngTime si avgServiceTime, iar in a doua fereastra generata dupa apasarea pe start se va afisa rezultatul in timp real la fiecare secunda.

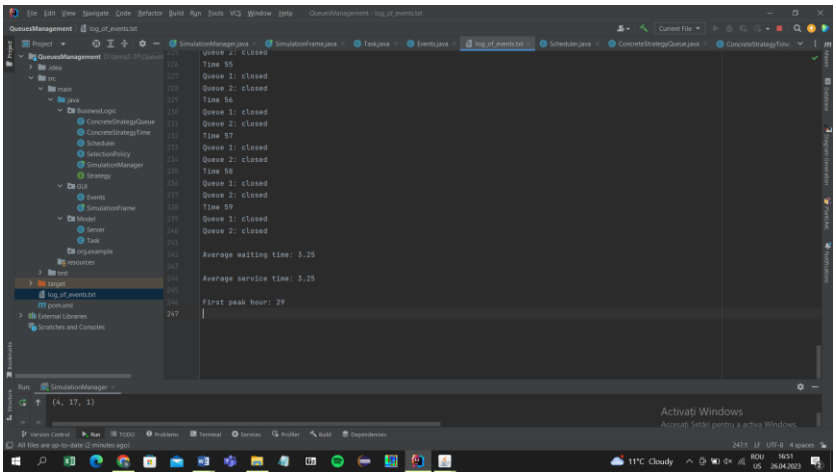
Clasa SimualtionFrame va contine 8 label-uri, 7 text fielduri, un buton pentru pornirea sitemului de management al cozilor si un combo-box pentru alegerea strategiei dupa care vor fi adaugati clientii in cozi. Clasa aceasta mai contine si metodele necesare pentru returnarea informatiilor in fiecare texField precum si din comboBox. Avem o metoda si pentru buton care foloseste actionListener. In metoda main corespunzatoare clasei vom folosi EventQueue.invokeLater pentru ca rularea sa nu se interfereze cu alte actiuni ce se petrec in fundal. De asemenea, dupa apasarea butonului de start se va deschide o noua fereastra Events unde se va afisa rezultatul in timp real, si aceasta folosindu-se de eventQueue.invokeLater pentru protectia threadului.

5. Rezultate

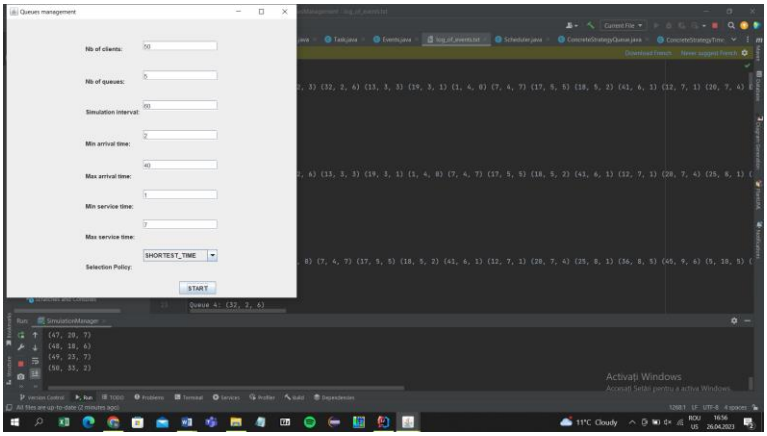
Pentru testarea aplicatiei am folosit datele de intrare prezentate in laborator, si am verificat manual timpul mediu de asteptare, timpul mediu de procesare si peak hour, si daca clientii au fost adaugati la momentul in care timpul curent este egal cu timpul de sosire al acestora, si cat timp au stat in coada. Am folosit o variabila volatile runnable pentru oprirea threadurile pornite pentru fiecare Server, astfel ca threadurile se opresc dupa terminarea timpului de simulare.

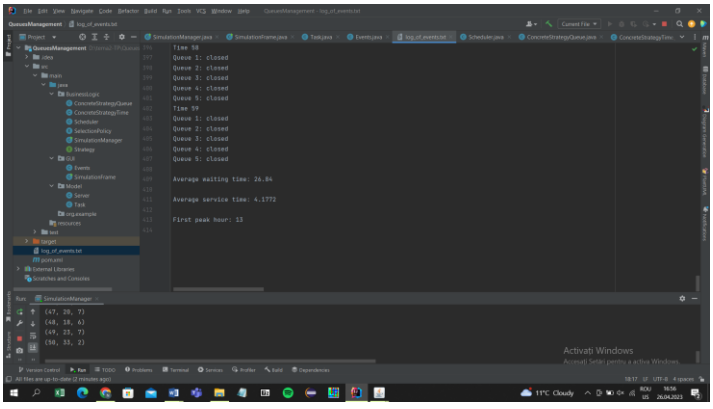
Pentru primul set de valori am obtinut urmatoarele valori:





Pentru al doilea set de valori am obtinut urmatoarele valori:





6. Concluzii

In cocluzie, cu ajutorul acestui sistem de gestiune al cozilor am reusit sa inteleg threadurile, cum functioneaza, pentru ce se utilizeaza, precum si cum se poate face sincronizarea cu ajutorul unei structure de date speciale pe care nu le-am mai intalnit pana acum. De asemenea, conceptul de multi-threading nu mai e strain. Am invatat cum se pot opri threadurile. Pe langa acestea, am fixat notiunile de generare random a clientilor. Consider util acest proiect si in viata de zi cu zi. Posibile dezvoltari ulterioare:

- Adaugarea unei interfete care sa arate ca o coada propriu-zisa, iar clientii se vor pune in coada
- Adaugarea mai multor strategii dupa care se vor adauga clientii
- Adaugarea unui serviciu specific dupa care se va adauga clientul in coada

7. Bibliografie

- https://www.tutorialspoint.com/java/java_multithreading.htm
- https://www.javatpoint.com/blockingqueue-in-java
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html
- https://docs.oracle.com/javase/7/docs/api/java/io/BufferedWriter.html