

**FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE**

QUEUES SIMULATOR

Documentație

Dorofte Andrei

Grupa 302210

Cuprins

1. Obiectivul temei

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

3. Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator)

4. Implementare

5. Rezultate

6. Concluzii

7. Bibliografie

1. **Obiectivul temei**

Proiectati si implementati o aplicatie de simulare care sa analizeze sistemele bazate pe cozi pentru determinarea si reducerea timpului de asteptare pentru clienti.

Cozile pot fi observate atat in lumea reala cat si ca model virtual. Obiectivul principal al unei cozi este de a furniza un loc de asteptare pentru un client inainte de a fi servit. Obiectivul unui sistem bazat pe cozi este de a reduce timpul de asteptare al clientilor care asteapta la coada pentru a fi serviti. Un mod de a reduce timpul de asteptare este de a adauga mai multe „servere”, dar aceasta lucru creste costul.

Cand este adaugat un server nou clientii vor fi distribuiti in mod egal la toate cozile care sunt disponibile. Sistemul trebuie sa fie capabil sa poata simula o serie de clienti care intra la coada, asteapta sa fie serviti, si in final sa fie serviti si scosi din coada. Pentru a calcula timpul de asteptare avem nevoie de timpul de sosire, timpul de terminare procesarii clientului si timpul de servire. Timpul de sosire si timpul de servire depinde de fiecare client cand acesta soseste si cat timp de servire are nevoie, de aceea se foloseste un generator aleator de clienti, astfel fiecare client fiind diferit. Timpul de plecare depinde de numarul de cozi, de numarul de clienti care asteapta la coada si de timpul de servire al acestora. Timpul mediu de asteptare va fi calculat si suprascris la fiecare procesare a unui client.

|  |
| --- |
| **Obiective secundare** |
| * Crearea claselor necesare |
| * Definirea metodelor de baza pentru fiecare clasa |
| * Implementarea algoritmilor de rulare |
| * Construirea generatorului si a cozilor |

1. **Analiza problemei, modelare, scenarii, cazuri de utilizare**

Cozile sunt de obicei utilizate pentru a modela domenii din lumea de zi cu zi. Obiectivul principal al unei cozi este de a oferi un loc de asteptare pentru un „client“ înainte de a primi un „serviciu“. Managementul colzilor este orientat spre minimizarea timpului pe care clientii il asteapta in coada inainte sa fie serviti.

Acest proiect are rolul de a simula o serie de clienti, generati aleator, care vor ajunge la cozi intr-un anumit interval de timp. Dupa sosire, se vor aseza la coada la care timpul de asteptare este cel mai mic, vor astepta sa le vina randul, dupa care vor fi serviti si eliminati din coada. Pentru a realiza acest lucru, trebuie sa cunoastem timpul de sosire si timpul necesar pentru servirea acestuia. De asemenea, o coada trebuie sa furnizeze timpul de asteptare al noului client, pana cand va ajunge sa fie servit, deoarece acesta doreste sa astepte cat mai putin. Acest timp depinde de numarul clientilor din coada si timpul de servire al fiecaruia.

* Sistemul trebuie sa realizeze generarea aleatoare a unui anumit numar de clienti dat si plasarea acestorea la cozi
* Sistemul trebuie sa simuleze servirea acestora si calcularea timpului mediu de asteptare

**Datele de intrare:**

Un fisier care sa contina: numarul clientilor, numarul de cozi, timpul de simulare ,intervalul de sosire al clientilor, intervalul de procesare al acestora.

**Datele de iesire:**

Un fisier de iesire care sa contina: ce se intampla la fiecare pas cu clientii, starea cozilor, timpul mediu de asteptare al clientilor la fiecare coada, timpul mediu de procesareal unei cozi, atfel spus un jurnal al evenimentelor. Sistemul trebuie sa preia datele de intrare din fisier si sa scrie datele de iesire in alt fisier.

Queues Simulator

Male icon

Generate clients

Male icon

Input data from file

Place in queues

Output file

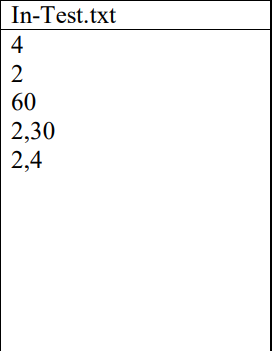
Simulate service

User

User

Caz de utilizare:

1. Utilizatorul introduce corespunzător datele de intrare in fisier.



Prima valoare reprezinta numarul de clienti.

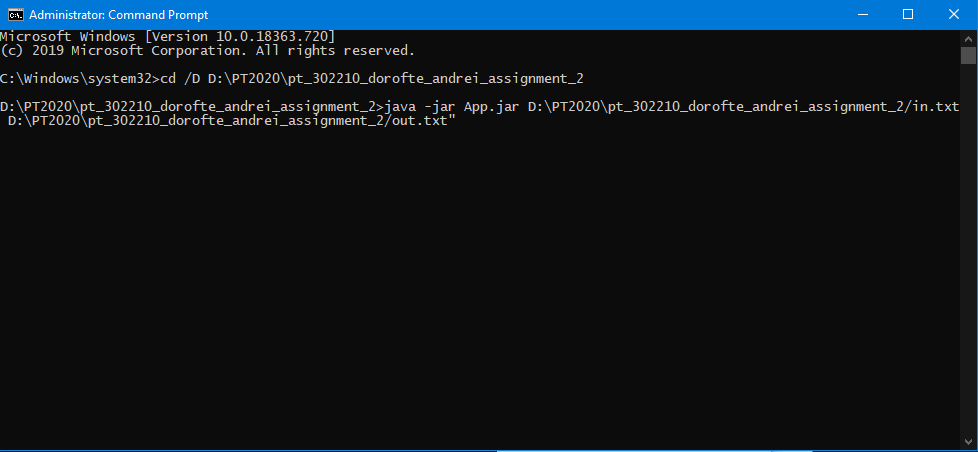
A doua valore reprezinta numarul de cozi.

A treia valoare reprezinta timpul maxim permis pentru rularea simularii.

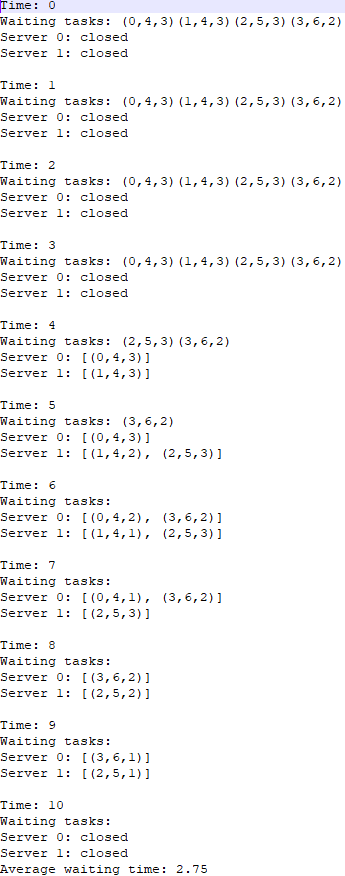
A patra si a cincea valoare reprezinta timpul minim si maxim al sosirii.

A sasea si a saptea valoare reprezinta timpul minim si maxim al servirii clientilor.

1. Utilizatorul ruleaza aplicatia dintr-o fereastra command prompt, deschisa din folder-ul in care se afla aplicatia, utilizand comanda „java -jar App.jar ‚path’/in.txt ‚path’/out.txt”.



1. Dupa terminarea rularii aplicatiei, se verifica fisierul de iesire pentru rezultate.



1. **Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfețe, relații, packages, algoritmi, interfață utilizator)**

Unified Modeling Language sau UML, pe scurt, este un limbaj standard pentru descrierea de modele si specificatii pentru software. UML a fost la bază dezvoltat pentru reprezentarea complexității programelor orientate pe obiect, al căror fundament este structurarea programelor pe clase, și instanțele acestora ( numite și obiecte ). Cu toate acestea, datorită eficienței și clarității în reprezentarea unor elemente abstracte, UML este utilizat dincolo de domeniul IT.

**Diagrama UML:**

A close up of a map

Description automatically generated

(c)

(b)

(a)

Am încercat sa modularizez cerința cat de mult posibil, astfel rezultând clasele din diagrama de mai sus: Main, FileRead, SimulationManager, Task, Scheduler, Server, Strategy, SelectionPolicy, ConcreteStrategyTime, ConcreteStrategyQueue, SortByArrival. Clasele au fost proiectate intuitiv, pentru a fi ușor de citit si de înțeles. Interfata FileRead putea fi implementata si ca o simpla clasa abstracta, la fel si interfata Strategy.

Pachetele create sunt fileRead, SimulationManager, Strategy, model.

1. **Implementare**

**Clasa Task**

Aceasta clasa modeleaza un client. Inmagazineaza pe langa toate caracteristicile unui client, timp servire, timp sosire, timpul de iesire din coada, si metode prin care aceste informatii sunt trasmise si mai apoi procesate.

**Clasa Server**

Clasa aceasta este utilizata pentru a modela o coada si a pune la dispozitie metode prin care adaugarea clientilor din coada sa fie posibila. Se utilizeaza o lista pentru stocarea clientilor prezenti in coada. Cu ajutorul metodei run() se proceseaza cate un client. Fiecare server dispune de o variabila booleana „active” cu ajutorul careia se verifica daca serverul este activ sau nu. De asemenea ajuta la pornirea si oprirea thread-urilor asociate server-ului, in caz ca acesta e activ sau nu.

**Interfata FileRead**

In aceasta interfata are loc citirea datelor de intrare din fisier si stocarea lor intr-o lista de string-uri. Citirea se face cu ajutorul unei expresii regulate.

O expresie regulată, regex (numită uneori expresie rațională) este, în informatica teoretică⁠ și în teoria limbajelor formale, un șir de caractere⁠ care definesc un șablon⁠ de căutare. De obicei, acest șablon este apoi utilizat de către algoritmii de căutare pe șiruri⁠ pentru operațiuni de „căutare” sau „căutare și înlocuire” operațiuni pe șiruri de caractere⁠.

Astfel datele de intrare sunt transmise mai departe catre clasa SimulationManager.

**Clasa SimulationManager**

Aceasta este clasa care face legatura intre toate celelalte clase. Aici se executa simularea propriu zisa: se atribuie datele de intrare citite din fisier, care sunt mai departe utilizate la generarea a N clienti cu timpi aleatorii de sosire si procesare. Apoi, metoda run() ia cate un client din varful cozii, il trimite la procesare, iar cand acesta e gata, il scoate din coada si preia urmatorului client. De asemenea, aici se calculeaza timpul mediu de asteptare.

**Clasa Scheduler**

Aceasta clasa preia si aloca cate un thread pentru fiecare server. Thread-ul este pornit atunci cand la coada se afla cel putin un client. De asemenea, aceasta clasa aloca strategia potrivita, in functie de strategia aleasa in SimulationManager.

**Clasa ConcreteStrategyTime**

Aceasta clasa implementeaza alegerea, de catre client, a cozii cu timpul de asteptare cel mai mic, daca se opteaza pentru strategia aceasta.

**Clasa ConcreteStrategyQueue**

Aceasta clasa implementeaza alegerea, de catre client, a cozii cu cei mai putini clienti, daca se opteaza pentru strategia aceasta.

**Clasa SortByArrival**

Aceasta clasa implemeteza un comparator ce ajuta la sortarea clientilor generati aleator, in functie de timpul de sosire.

**Interfata Strategy**

Aceasta interfata contine doar metoda neimplementata addTask(), care este ulterior implemetata in ConcreteStrategyTime si ConcreteStrategyQueue

**Enumeratia SelecionPolicy**

Aceasta enumeratie contine doar cele 2 tipuri de strategii din care se poate alege: SHORTEST\_TIME si SHORTEST\_QUEUE

**Clasa Main**

Aceasta clasa contine doar instantierea unui obiect de tip SimulationManager, ce are ca paramentri in constructor path-ul pentru fisierul de intrare si fisierul de iesire. Se creeaza un thread cu acesti fisier si se porneste.

1. **Rezultate**

Pentru cazurile de testare am testat aplicatia pentru 3 cazuri: avem aceeasi timpi pentru primele 2 cazuri(60), dar ceea ce va fi diferit: numarul de clienit, numarul de cozi, iar pentru a arata ca fiecare client stie sa se pozitioneze la coada la care este cel mai mic timp de asteptare. Timpii de sosire ai unui client la coada vor fi exprimate in secundele citite din fisier, timpul de procesare al unei comenzi, la fel, citite fiecare din fisierul corespunzator, numarul de cozi va fi diferit in functie de caz, iar intervalul de simulare va fi de 60 de secunde, dar si 200. De asemenea, am realizat si un fisier .jar, pe care l-am atasat in proiect, si care poate fi rulat cu parametrii cailor pentru fiserul de intrare si cel de iesire.

* Pentru primul caz, datele de intrare sunt (fisierul de intrare: in-test-1.txt; iesire: out-test-1.txt):

4 - numarul de clienti

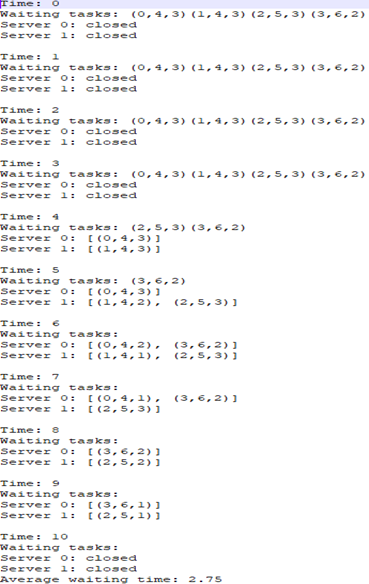
2 - numarul de cozi

60 - timpul de simulare

2,30 - arrivalTime

2,4 - serviceTime

Rezultatul simularii pentru primul caz cu durata de 1 minut:



* Pentru al doilea caz, datele de intrare sunt (fisierul de intrare: in-test-2.txt; iesire: out-test-2.txt):

50 - numarul de clienti

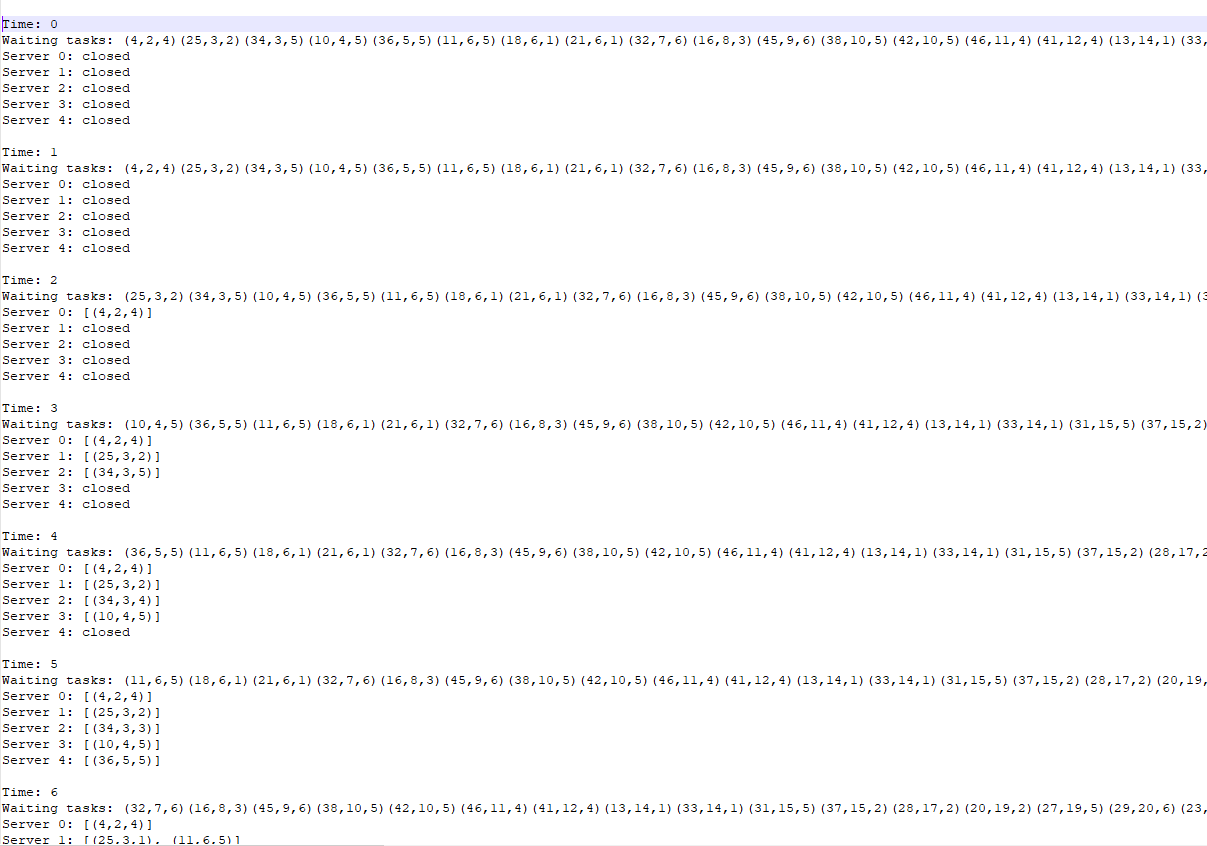
5 - numarul de cozi

60 - timpul de simulare

2, 40 - arrivalTime

1, 7 - serviceTime

Rezultatul simularii pentru al doilea caz cu durata de 1 minut:



* Pentru al treilea caz, datele de intrare sunt (fisierul de intrare: in-test-2.txt; iesire: out-test-2.txt):

1000 - numarul de clienti

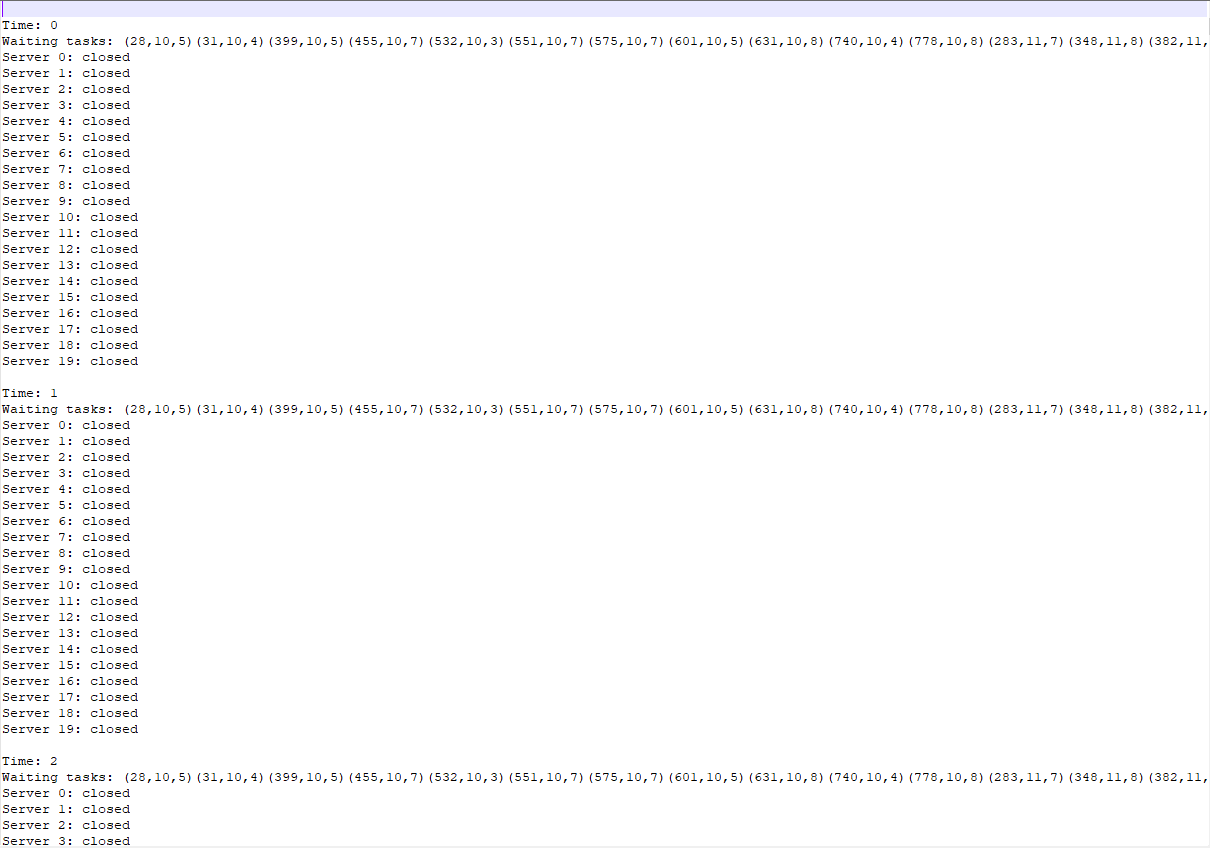
20 - numarul de cozi

200 - timpul de simulare

10, 100 - arrivalTime

3, 9 - serviceTime

Rezultatul simularii pentru al treilea caz cu durata de aproximativ 3 minute si jumatate:



Rezultatele din etapa de testare arata ca aplicatia este modelata corect, iar algoritmul de simulare a cozilor este functional. Rezultatele obtinute demonstreaza ca pentru diferite cazuri avem timpi de asteptare mai mici sau mai mari. Daca avem timpi mai mici de aparitie a clientilor si timpi mai mari de procesare a comenzilor atunci pentru a micsora timpul de asteptare a clientilor este indicat sa avem mai multe cozi deschise. Programul permite utilizatorului sa vada cum ar trebui organizate cozile pentru a reduce timpul de asteptare.

**Concluzii**

In concluzie, sunt de părere ca acest proiect m-a ajutat sa îmi aprofundez cunoștințele in tot ce înseamnă limbajul Java, implementarea paradigmelor OOP, si dezvoltarea unui algoritm eficient, care sa satisfacă obiectivul unui proiect.

Aceasta tema simuleaza situatii din viata de zi cu zi, intrucat, oriunde avem nevoie sa mergem, deseori suntem nevoiti sa stam la cozi mai mult sau mai putin mari, pentru a obtine ce ne dorim. Daca si in realitate, am avea posibilitatea de a ne aseza la coada cu cel mai mic timp de asteptare, lucrurile s-ar fluidiza foarte mult.

Chiar daca simularea este corecta, ea nu poate prevedea toate situatiile neasteptate din realitate. In aceasta tema am invatat sa lucrez cu threaduri, executia lor in acelasi timp si implementarea mai multor metode pentru a putea simula sistemul bazat pe cozi. De asemenea, am invatat cum sa lucrez cu un LinkedBlockingQueue, care este foarte util din cauza metodelor peek() sau poll(), si de acum, oriunde voi avea ocazia, voi utiliza aceasta structura. De asemenea, am invatat sa generez, sa configurez un fisier .jar sa preia parametri si sa il rulez din linia de comanda, folosind parametrii rezpectivi.

In ceea ce privește dezvoltarea ulterioara a programului:

* Se poate incerca implementarea unor situatii neprevazute, se mareste timpul de procesare al unei comenzi la

un moment aleator

* Implementarea mai concisa a main-ului, pentru o interactionare mai buna a thread-urilor
* Utilizarea unui executor pentru a elimina comanda de thread.sleep()
* Se poate imbunatati performanta algoritmilor
* Se poate configura fisierul .jar astfel incat parametrii fisierelor de intrare si de iesire sa fie doar in.txt si out.txt si sa se poata rula doar cu ajutorul comenzii „java -jar App.jar in.txt out.txt”

1. **Bibliografie**

* <https://stackoverflow.com/>
* <http://docs.oracle.com/>
* <https://www.geeksforgeeks.org/>
* youtube.com - tutoriale