Text

Description automatically generated

DOCUMENTAȚIE

TEMA 2-Queue Management

Buruian Cătălina

Grupa 30221

# 

**CUPRINS**

1. [Obiectivul temei 2](#_Toc97418559)

1. [Analiza problemei, modelare, scenarii, cazuri de utilizare 2](#_Toc97418560)
2. [Proiectare 2](#_Toc97418561)
3. [Implementare 2](#_Toc97418562)
4. [Rezultate 2](#_Toc97418563)
5. [Concluzii 2](#_Toc97418564)
6. [Bibliografie 2](#_Toc97418565)

**Capitolul 1:Obiectivul temei**

Obiectivul principal

Obiectivul principal al acestei teme a constat in implementarea unei aplicatii capabile sa analizeze un sistem bazat pe cozi prin simularea unei serii de N clienti care ajung pentru un serviciu, se asaza la coada, sunt serviti apoi, in final, parasesc coada. De asemenea, in cadrul acestei aplicatii, se va calcula si timpul mediu de asteptare, timpul mediu de servire si ora de varf.

Obiective secundare

1. Analiza problemei si identificarea cerintelor (Capitolul 2)
2. Proiectarea aplicatiei (Capitolul 3)
3. Implementarea aplicatiei (Capitolul 4)
4. Testarea (Capitolul 5)

**Capitolul 2:Analiza problemei, modelare, scenarii, cazuri de utilizare**

Cerinte functionale:

1. Aplicatia de simulare ar trebui sa permita utilizatorilor sa introduca datele pentru simulare
2. Aplicatia de simulare ar trebui sa permita utilizatorilor sa porneasca simularea
3. Aplicatia de simulare ar trebui sa afiseze cozile in timp real

Use case: realizarea setup-ului de catre utilizator

Actorul principal: Utilizatorul

Principalul scenariu de succes:

1. Utilizatorul introduce valori pentru numarul de clienti, numarul de cozi, timpul de simulare, minimul si maximul timpului de sosire si minimul si maximul timpului de servire.
2. Utilizatorul apasa pe butonul de „START” pentru a porni simularea
3. Aplicatia valideaza datele introduse de utilizator si afiseaza un mesaj care indica ca poate fi pornita simularea

Scenariu alternativ: Valori invalide pentru parametri de setup

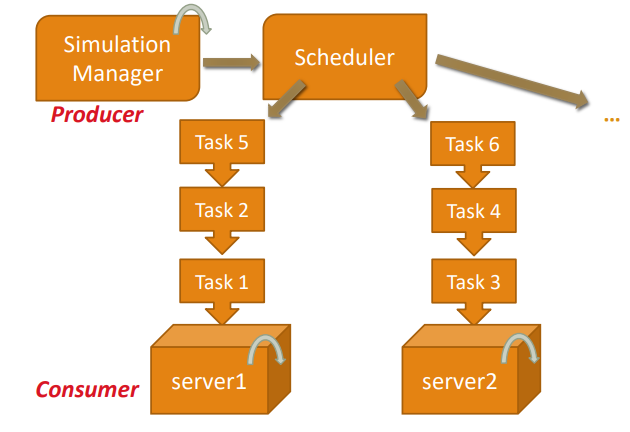
* Utilizatorul introduce valori invalide pentru parametri de setup
* Aplicatia afiseaza un mesaj de eroare si cere utilizatorului sa introduca valori valide
* Scenariul se intoarce la pasul 1.

Diagram

Description automatically generated

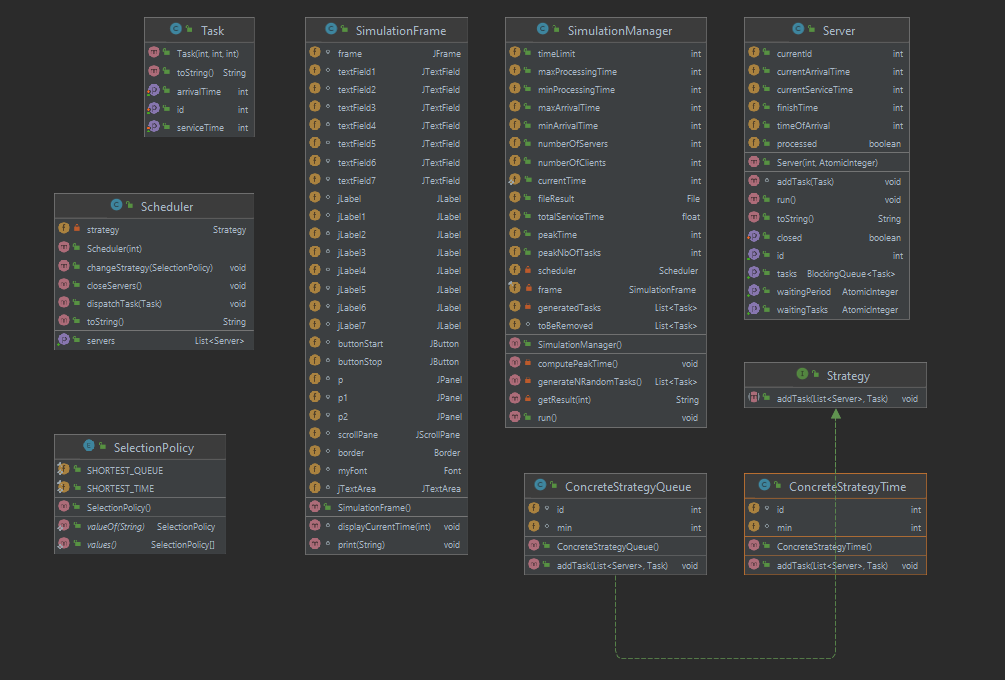
**Capitolul 3: Proiectarea**

**Design-ul conceptual**



**Diagrama UML de clase**

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. Așa se face că există aplicații ale UML-ului pentru management de proiecte, pentru business Process Design etc.



**Structuri de date folosite**

**Liste**

O lista reprezinta o colectie ordonata de elemente. Intr-o lista sunt permise elementele duplicate. Permite un control mult mai precis asupra pozitiei unui element prin intermediul indexului acelui element.

Interfata List descrie liste de elemente indexate si are disponibile implementari precum cea utilizata in cadrul acestei teme, si anume ArrayList.

ArrayList: permite acces aleator la elementele listei; obtinearea oricarui element din lista se va face in timp constant; in schimb, adaugarea sau stergerea unui element in interiorul listei necesita deplasarea elementelor care se gasesc dupa pozitia la care se executa operatia.

**BlockingQueue**

Interfața BlockingQueue acceptă controlul fluxului (în plus față de coadă) prin introducerea blocării dacă BlockingQueue este plina sau goala. Un fir de execuție care încearcă să pună în coadă un element într-o coadă completă este blocat până când un alt fir de execuție face spațiu în coadă, fie prin scoaterea din coadă a unuia sau mai multe elemente, fie prin ștergerea completă a cozii. În mod similar, blochează un fir de execuție care încearcă să ștergă dintr-o coadă goală până când alte fire de execuție introduc un element. BlockingQueue nu acceptă o valoare nulă. Dacă încercăm să punem în coadă elementul nul, atunci acesta aruncă NullPointerException.

Java oferă mai multe implementări BlockingQueue, cum ar fi LinkedBlockingQueue, ArrayBlockingQueue, PriorityBlockingQueue, SynchronousQueue etc. Implementările interfeței Java BlockingQueue sunt sigure pentru threaduri. Toate metodele de BlockingQueue sunt de natură atomică și folosesc blocări interne sau alte forme de control al concurenței. Java 5 vine cu implementări BlockingQueue în pachetul java.util.concurrent.

**Capitolul 4: Implementarea**

**Thread-uri**

Un thread este un fir de execuție într-un program. Mașina virtuală Java permite unei aplicații să aibă mai multe fire de execuție care rulează simultan.

Fiecare thread are o prioritate. Firele cu prioritate mai mare sunt executate de preferință față de firele cu prioritate mai mică. Fiecare fir poate fi sau nu marcat ca daemon. Când codul care rulează într-un fir de execuție creează un nou obiect Thread, noul fir de execuție are prioritatea setată inițial egală cu prioritatea firului de creare a execuției și este un fir de execuție demon dacă și numai dacă firul de execuție care creează este un daemon.

Când o mașină virtuală Java pornește, există de obicei un singur fir non-daemon (care numește de obicei metoda numită principal a unei clase desemnate). Mașina virtuală Java continuă să execute fire până când apare oricare dintre următoarele:

Metoda de ieșire a clasei Runtime a fost apelată și managerul de securitate a permis ca operația de ieșire să aibă loc.

Toate firele de execuție care nu sunt fire daemon au murit, fie prin întoarcerea de la apelul la metoda run, fie prin lansarea unei excepții care se propagă dincolo de metoda run.

Creare thread-uri:

Clasa Thread în sine implementează Runnable - metoda sa de rulare nu face nimic. O clasă poate extinde clasa Thread și poate oferi o implementare a metodei run.

Sincronizarea Threadurilor:

Probleme posibile: interferenta, inconsistenta memoriei

Solutii: declararea variabilelor instanta de tip final, atomicitate, metode sincronizate etc.

**Descrierea claselor si metodelor din cadrul claselor**

Clasa Task: -defineste task-ul (clientul) care se aseaza la coada si urmeaza a fi procesat, dupa care paraseste coada. Aceasta clasa are ca variabile instanta timpul de sosire (arrivalTime), timpul de servire (serviceTime) si un id prin care este identificat. De asemena, in cadrul acestei clase sunt definite metode de tip setter si getter, precum si o metoda de afisare.

Clasa Service: -defineste coada (BlockingQueue) de task-uri (clienti) si un waitingPeriod care indica durata cozii. Prin metoda addTask, se adauga task-urile la coada si se incrementeaza waitingPeriod cu serviceTime-ul task-ului. Aceasta clasa implementeaza interfata Runnable cu metoda asociata run, unde se ia primul client din coada, se asteapta un timp prin apelarea Thread.sleep(), apoi se decrementeaza waitingTime cu serviceTime-ul task-ului. Interfața Runnable ar trebui să fie implementată de orice clasă ale cărei instanțe sunt destinate să fie executate de un thread, avand metoda aferenta run.

Această interfață este concepută pentru a oferi un protocol comun pentru obiectele care doresc să execute cod în timp ce sunt active. De exemplu, Runnable este implementat de clasa Thread. A fi activ înseamnă pur și simplu că un fir a fost început și nu a fost încă oprit.

Pentru actualizarea timpului de asteptare am folosit waitingPeriod de tip AtomicInteger. O clasă java.util.concurrent.atomic.AtomicInteger oferă operații asupra valorii intregi care poate fi citita și scrisa atomic și conține, de asemenea, operații atomice avansate. AtomicInteger acceptă operații atomice asupra variabilei intregi. Are metode get și set care funcționează ca citirea și scrierea pe variabile volatile. Pentru incrementarea lui waitingPeriod am folosit metoda getAndAdd() care este o metodă încorporată în java care adaugă valoarea dată la valoarea curentă și returnează valoarea înainte de actualizare, care este de tipul de date int.

Clasa include si o metoda de afisare a cozii cu task-urile din ea, ori cu mesajul „closed” in cazul in care nu exista clienti la coada.

Clasa Scheduler: -are un constructor in care initializeaza un anumit numar de cozi, creeaza thread-uri pentru acestea si le activeaza cu metoda start(). Prin metoda changeStrategy se poate schimba strategia de adaugare a clientilor la cozi, optiunile valabile fiind: timpul minim de asteptare sau lungimea cozii minima. Metoda dispatchTask schimba strategia si apeleaza metoda addTask din server implementata in cadrul interfetei Strategy. Metoda de afisare din cadrul acestei clase este utila pentru a vedea clientul curent care este procesat si cum i se modifica timpul de service, pana cand acesta este zero, moment in care clientul este scos din coada.

Clasele ConcreteStrategyQueue si ConcreteStrategyTime: -care implementeaza interfata Strategy cu metoda addTask si aplica un model de distributie a clientilor in cadrul cozilor.

Clasa SimulationManager: -contine o metoda de generare a N clienti random, folosind metoda Math.random() si ii adauga la o lista de task-uri care va fi sortata in functie de timpul de sosire a clientilor. In constructor, creeaza interfata si adauga actionListener pe butonul de start care va lua datele necesare simularii din interfata. Prin activarea threadului cu thread.start(), se vor afisa rezultatele simularii atat in interfata, cat si intr-un fisier text.

In cadrul metodei run se verifica pentru fiecare task din cadrul listei generatedTasks daca are timpul de sosire egal cu timpul curent, caz in care este adaugat la coada si sters din lista. Pentru a calcula timpul de servire mediu se calculeaza timpul total de servire pentru fiecare task care este adaugat in coada la un moment dat si se imparte la numarul de taskuri. De asemenea, tot in cadrul acestei clase se va calcula peakTime care reprezinta timpul la care sunt cei mai multi clienti procesati. Pentru timpul curent am folosit keyword-ul „volatile”, ceea ce asigura folosirea in siguranta a unei variabile instanta de catre mai multe thread-uri.

Clasa SimulationFrame: - contine declaratii de butoane, text field-uri, labels, panels si alte componente necesare pentru realizarea unei interfete care sa imbunatateasca experienta utilizatorului.

**Descrierea GUI**

**Graphical user interface, application

Description automatically generated**

Interfata contine textField-uri in care utilizatorul poate introduce datele pentru simulare, precum timpul de simulare, numarul de clienti, numarul de cozi, timpul de sosire, respectiv timpul de servire minim si maxim. Prin apasarea butonului de START, utilizatorul va vedea rezultatele afisate in partea de jos a ferestrei in timp real. Astfel, va fi afisata evolutia procesului de asezare a clientilor la cozi, iar in partea de sus a ferestrei se poate observa timpul curent.

**Capitolul 5: Rezultate**

Exemple de rulare:

Test 1: Pentru N=4 clienti, 2 cozi, timp maxim de simulare=60 secunde, timpul minim de sosire=2, timpul maxim de sosire=30, timpul minim de servire=2 si timpul maxim de servire=4 se vor afisa urmatoarele rezultate:

**Graphical user interface

Description automatically generated**

La timpul 0 ambele cozi sunt inchise, se genereaza random 4 clienti, iar primul va intra in prima coada dupa 3 secunde si va sta in varful cozii 4 secunde pana va fi servit. Dupa ce acesta este procesat va iesi din coada, iar urmatorul client va fi procesat.

Graphical user interface

Description automatically generated

La timpul 6 urmatorul client din lista de astepare va fi adaugat in coada 2 si va fi procesat timp de 2 secunde dupa care va iesi din coada. La timpul 7 se poate observa cum coada nr 1 se inchide deoarece primul client a fost servit. Urmatorii 2 clienti vor fi adaugati in coada la timpul 9, respectiv 24, avand 3, respectiv 2 secunde in care vor fi in varful cozii si vor fi procesati. Mecanismul de adaugare la cozi respecta strategia cu timpul minim de asteptare la coada.

Graphical user interface

Description automatically generated

La final am afisat average service time-ul si peak time-ul.

Test 2: Pentru N=50 clienti, 5 cozi, timp maxim de simulare=60 secunde, timpul minim de sosire=2, timpul maxim de sosire=40, timpul minim de servire=1 si timpul maxim de servire=7 se vor afisa urmatoarele rezultate:

Graphical user interface

Description automatically generated

La fel ca la prima simulare, clientii sunt adaugati la coada in functie de timpul de sosire, sunt procesati, apoi parasesc coada. La final am afisat average service time-ul si peak time-ul.

Graphical user interface

Description automatically generated

Test 3: Pentru N=1000 clienti, 20 cozi, timp maxim de simulare=200 secunde, timpul minim de sosire=10, timpul maxim de sosire=100, timpul minim de servire=3 si timpul maxim de servire=9 se vor afisa urmatoarele rezultate:

Graphical user interface

Description automatically generated

La fel ca la prima simulare, clientii sunt adaugati la coada in functie de timpul de sosire, sunt procesati, apoi parasesc coada. La final am afisat average service time-ul si peak time-ul.

Graphical user interface, text, application

Description automatically generated

De asemenea, pentru ca utilizatorul sa poata accesa cu usurinta rezultatele simularii le-am afisat si intr-un fisier text.

Text

Description automatically generated

**Capitolul 6: Concluzii**

In concluzie, aceasta tema m-a ajutat sa aprofundez anumite aspect legate de OOP, cat sa-mi si reamintesc notiunile invatate semestrul trecut. A fost o buna oportunitate pentru mine sa ma familizarizez cu limbajul de programare Java, in special cu folosirea threadurilor.

Dezvoltari ulterioare: Crearea unei interfete mai interactive si adaugarea unui buton de STOP

**Capitolul 7: Bibliografie**

<https://docs.oracle.com/javase/8/docs/api/java/util/Timer.html>

<https://www.baeldung.com/java-runnable-callable>

<https://docs.oracle.com/javase/8/docs/api/java/util/Timer.html>

<https://www.javatpoint.com/multithreading-in-java>

<https://www.geeksforgeeks.org/volatile-keyword-in-java/>