DOCUMENTATIE

TEMA 2

Queues Simulator

Vamvu Denisa-Elena

Grupa 30229

CUPRINS

1. Obiectivul temei

-Se va prezenta obiectivul principal al temei printr-o fraza si un tabel sau o lista cu obiectivele secundare. Obiectivele secundare reprezinta pasii care trebuie urmati

pentru indeplinirea obiectivului principal. Fiecare obiectiv secundar va fi descris si

se va indica in care capitol al documentatiei va fi detaliat.

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

-Modelare, scenarii, cazuri de utilizare . Se va prezenta cadrul de cerinte functionale formalizat si cazurile de utilizare ca si diagrame si descrieri de use-case.

Descrierile use-case-urilor se vor face sub forma unui flow-chart ori sub forma unei liste continand pasii executiei fiecarui use-case.

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

-Se va prezenta proiectarea OOP a aplicatiei, diagramele UML de clase si de pachete, structurile de date folosite, interfetele definite si algoritmii folositi (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator).

4. Implementare

-Se va descrie fiecare clasa cu campurile si cu metodele importante. Se va descrie implemantarea interfetei utilizator.

5. Rezultate

-Se vor prezenta scenariile pentru testare cu Junit sau alt framework de testare.

6. Concluzii

-Se vor prezenta concluziile, ce s-a invatat din tema, dezvoltari ulterioare.

7. Bibliografie

-Se vor mentiona resursele bibliografice care au fost folosite pentru dezvoltarea temei.

**1. Obiectivul temei**

Obiectivul acestei teme este implementarea unei aplicatii care simuleaza mai multe cozi, scopul acesteia fiind minimizarea timpului de asteptare pentru clienti si determinarea unui timp mediu de asteptare.

1.2 Obiective secundare

|  |  |  |
| --- | --- | --- |
| **Obiectiv Secundar** | **Descriere** | **Capitol** |
| Dezvoltarea de use case-uri si scenarii | Într-un sistem software un ‘use-case’ este o listă de acțiuni sau pași de eveniment care definesc în mod obișnuit interacțiunile dintre un rol ( cunoscut ca un actor în limbajul unificat de modelare ( UML ) ) și un sistem in atingerea unui obiectiv.  In computere, un scenariu este o naratiune a interactiunilor dintre roluri ( actori in UML ) si sistem. In cazul nostru rolul - un utilizator al aplicatiei. | 2 |
| Alegerea structurilor de date | Structurile de date folosite pentru a implementa cerinta propusa. | 3 |
| Impartirea pe pachete | Definirea pachetelor pentru impartirea claselor in functie de rolul acestora, asigurand organizarea si lizibilitatea codului. | 3 si 4 |
| Dezvoltarea algoritmilor | Vor fi descrise structurile de date necesare pentru atingerea obiectivului principal, schema UML ( diagrama de clase ) precum si algoritmii folositi pentru realizarea operatiilor. | 4 |
| Implementarea solutiei | Vor fi descrise fiecare clasa cu campurile si metodele importante . | 4 |
| Testarea programului | Exemplificarea unei rulari a programului | 5 |

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

Cozile sunt adeseori folosite pentru a reprezenta modele intalnite in viata reala. Scopul unei cozi este de a furniza un loc de asteptare pentru un “client” pana ca acesta sa poata fi “servit”. O metoda de minimizare a timpului de asteptare este adaugarea unei noi persoane care serveste, adica crearea unei noi cozi.

Aplicatia simuleaza (prin definirea unui timp de simulare t\_simulation) un numar de N clienti care asteapta sa fie serviti , intrand in Q cozi, asteptand pana sa fie serviti dupa care sa paraseasca coada. Toti clientii sunt generati la inceperea simularii , si sunt caracterizati de urmatorii 3 parametrii: ID(un numar intre 1 si N), t\_arrival(timpul de simulare cand clientul este pregatit sa intre in coada sau cand clientul si-a terminat cumparaturile) si t\_service ( timpul de simulare sau durata necesara pana clientul este servit de catre casier sau timpul cat clientul sta in fata cozii). Aplicatia contorizeaza timpul total petrecut de fiecare client la coada si calculeaza un timp mediu de asteptare la coada. Fiecare client este adaugat la coada cu un minim de asteptare cand timpul de ajungere al acestuia (t\_arrival) este mai mare sau egal decat timpul de simulare (t\_arrival >= t\_simulation).

Urmatoarele campuri vor fi citite din fisier ca date de intrare pentru aplicatie:

* Numarul de clienti (N)
* Numarul de cozi (Q)
* Intervalul de simulare (tmax\_simulation)
* Timpul minim si maxim de ajungere (tmin\_arrival <= t\_arrival <= tmax\_arrival)
* Timpul minim si maxim de servire (tmin\_service <= t\_service <= tmax\_service)

**Use case**

Programul va fi apelat din linia de comanda cu ajutorul unui fisier .jar ce se afla in folderol proiectului. Se executa comanda “java -jar” urmata de numele fisierului .jar si numele fisierelor de intrare si iesire (cel care va fi creat automat de catre aplicatie) in unul dintre cele doua cazuri:

* Cmd-ul este deschis din folderul proiectului, deci se va apela cu numele normal al argumentelor
* Cmd-ul este dechis in alt folder, deci se va apela folosind calea absoluta pentru toate cele trei argumente

Se presupune ca elementele din fisierul de intrare, cat si fisierul, sunt valide si reprezinta cerintele temei.

A screenshot of a cell phone

Description automatically generated

1. Proiectare

* Decizii de proiectare:

Fiecare coada este reprezentata de un thread, un fir de executie, unitate de procesare ce permite unui program sa se imparta pe el insusi in mai multe task-uri rulabile simultane sau pseudo-simultane. De asemenea, un thread principal comanda executia cozilor.

* Structuri de date folosite

In clasa **Server** am folosit o structura de tip BlockingQueue pentru a retine lista de client a fiecarei cozi. Termenul blocking queue vine de la faptul ca aceasta structura este capabila de a bloca threadurile care incearca ssa insereze sau sa acceseze elemente din aceata coada.



Alocarea se face astfel: 

Folosim aici ArrayBlockingQueue pentru a seta o capacitate maxima a cozii.

In clasele **Scheduler** si **SimulationManager** am folosit cate o structura de tip ArrayList, una pentru a tine toate cozile pe care le folosim iar cealalta pentru a stoca clientii care sunt generati aleator si sunt in asteptare pentru a intra la o coada (cazul in care timpul de simulare nu este inca suficient de mare ca sa le permita intrarea intr-o coada).

Diagrama use-case:

A close up of text on a white background

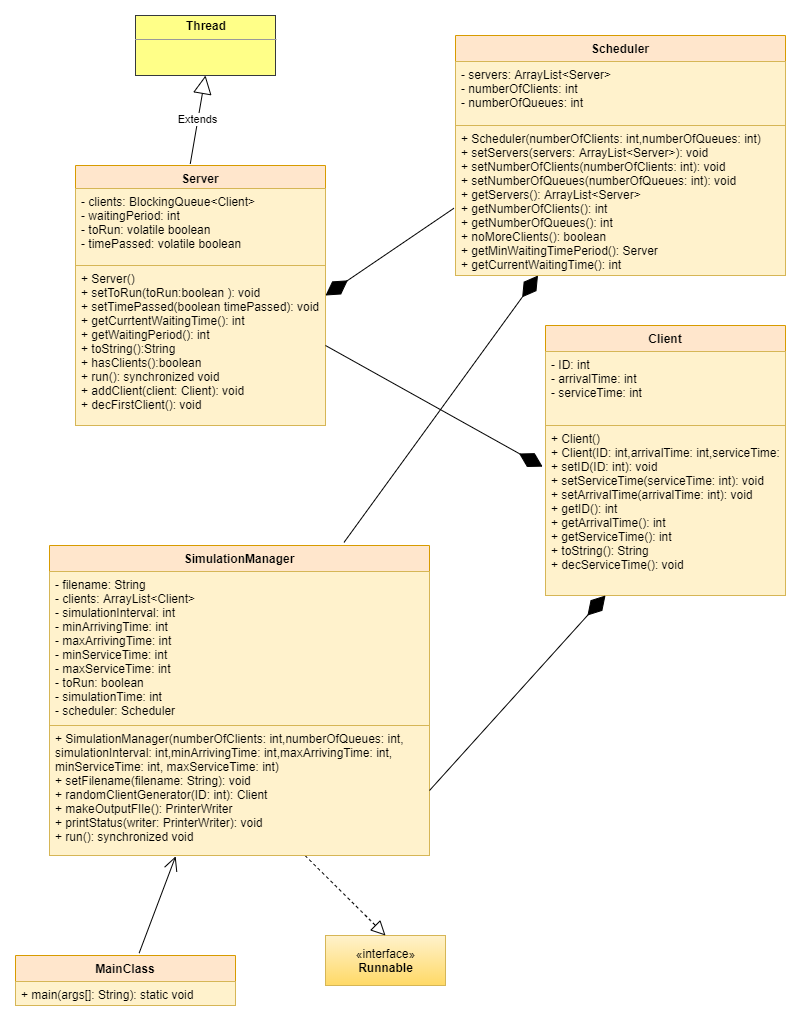
Description automatically generated

Diagrama UML de pachete:

A screenshot of a cell phone

Description automatically generated

Diagrama UML de clase:



1. Implementare

In continuare voi prezenta clasele pe care le-am folosit in acest proiect si metodele cele mai importante din fiecare.

* **Client**

Clasa de baza, de aici se porneste. Clientul reprezinta un Task pentru aplicatie, ce trebuie procesat. Ca si atribute de clasa enumeram: ID, arrivalTime, serviceTime. Pe langa cei doi constructori, cu si fara parametrii, setters si getters, avem metoda decServiceTime() care decrementeaza timpul de servire si va fi folosita mai tarziu pentru procesarea propriu-zisa a clientului, atunci cand acesta este la coada.

* **Server**

Clasa **Server** contine o “coada de clienti” , o perioada de asteptare si doua variabile mai speciale, de tipul **volatile boolean** care se asigura ca thread-ul ruleaza sau ca thread-ul stie cand sa proceseze primul client. Keyword-ul volatile este folosit pentru a modifica valoarea unei variable de catre multiple threaduri, adica aceasta sa fie stocata in memoria comuna a lor. **Server** extinde clasa **Thread** si suprascrie metoda **run**().

Functionarea metodei **run**() este urmatoarea: cat timp variablia **toRun** este adevarata (va fi setata pe fals in clasa SimulationManager in anumite conditii) thread-ul ruleaza iar daca am trecut la urmatorul pas in simulare, se apeleaza metoda decFirstClient(). Altfel thredul va trebui sa astepte pana cand alt thread va apela metoda notifyAll() pentru acelasi obiect.

Metoda **decfirstClient**(): daca in momentul in care este apelata metoda coada de clienti nu este goala, se scade cu 1 timpul de servire al primului client. Functia peek() returneaza primul element dintr-un BlockingQueue. In cazul in care timpul de servire devine 0, clientul a fost servit si este eliminat din coada: la pasul urmator nu va mai fi afisat. Perioada de asteptare de la coada respectiva ce reprezinta suma timpilor de servire a tuturor clientilor aflati in coada va fi decrementata.

* **Scheduler**

Clasa Scheduler are ca si atribute un ArrayList de Servere, adica toate cozile ce vor fi deschise la un moment dat in simulare, numarul de clienti si numarul de cozi. Clasa are un constructor, setters si getters pentru campuri si o metoda care returneaza true in caz ca nu mai sunt client in coada la un moment dat.

Metoda **getMinWaitingTimePeriod**() are o importanta deosebita pentru ca de ea depinde logica aplicatiei: in ce coada va fi introdus noul client. Metoda calculeaza minimul dintre perioadele de asteptare corespunzatoare fiecarei cozi, prin doua parcurgeri cu foreach a listei: prima salveaza valoarea minima, iar cea de-a doua gaseste prima coada cu timpul minim, iar aceasta se returneaza.

Metoda **getCurrentWaitingTime**() returneaza timpul de asteptare total ca suma a timpilor de asteptare de la fiecare coada pentru a putea fi folosit la calcularea timpului mediu de asteptare.

* **SimulationManager**

Clasa **SimulationManager** are ca si atribute pe langa intervalul de timp in care se executa simularea si limitele pentru timpul de sosire si de servire ale clientilor ce vor fi citite din fisier si un ArrayList de clienti are va stoca toti clientii generate aleator inainte ca acestia sa fie introdusi in cozi, un String ce reprezinta numele fisierului de iesire,un timp de simulare ce se va incrementa succesiv pana va ajunge la maximul citit din fisier,o variablia toRun ce va tine threadul pornit, un obiect de tip Scheduler ce permite atat comunicarea cu clasa Server cat si folosirea metodelor **getMinWaitingTimePeriod()** si **getCurrentWaitingTime()** pentru realizarea logicii de control.

Constructorul clasei seteaza toate atributele si insereaza in lista de client toti clientii generate.

Metoda **randomClientGenerator**( ID ) creeaza clientul cu ID-ul dat ( ID-urile vor fi primite in ordine crescatoare de la 1 la N ) si cu un arrivalTime si serviceTime generate aleator cu ajutorul unui obiect al clasei Random si cu metoda **nextInt** care primeste ca si parametru limita max-min+1 iar numarului generat i se va adauga min pentru a genera numere in range-ul ( min, max ) pentru ambele campuri.

Metoda **makeOutputFile**() returneaza un obiect de tip PrintWriter care va reprezenta fisierul in care se va scrie output-ul. Numele acestuia va fi dat de atributul de clasa filename.

Metoda **printstatus**(writer) scrie in fisierul de iesire apeland metoda **println**() pentru obiectul primit ca si parametru.

Clasa implementeaza interfata **Runnable** si suprascrie metoda **run**().

Metoda **run**() deschide fisierul de output apeland metoda makeOutputFile() si denumeste toate cozile “Queue i” unde i este un numar de la 1 la N, cu o parcurgere cu foreach a cozilor. De asemenea, pe logica asemanatoare cu threadul ce il va reprezenta un obiect de tip Server, cat timp variabila toRun (initializata true) este adevarata, se parcurge cu o interfata de tip **Iterator**<Client> lista de clienti care asteapta sa fie procesati si pentru fiecare din ei se intampla urmatoarele: se ia serverul cu cel mai mic timp de asteptare si daca este indeplinita conditia de inserare in coada, anume arrivalTime-ul clientului <= cu timpul de simulare, acesta este introdus in coada corespunzatoare si scos din lista de clienti in asteptare. Se printeaza timpul de simulare si statusul clientilor si al tuturor cozilor, timpul de simulare se incrementeaza iar la timpul de asteptare se adauga suma tuturor timpilor clientilor aflati in cozi la momentul actual cu ajutorul functiei **getCurrentWaitingTime**(). Pentru fiecare thread ce reprezinta o coada se va seta variabila care transmite faptul ca a mai trecut o secunda din simulare pe true si se foloseste metoda **interrupt**() pentru a le intrerupe executia in caz ca sunt in starea indusa de metoda wait(). Threadul principal este pus pe pauza 100 de milisecunde. Conditiile de oprire a executiei threadului principal sunt: nu mai avem client in asteptare si toate cozile sunt goale, atunci se da un break, o iesire fortata; cand timpul de simulare, initializat la 0 in aceasta metoda a ajuns egal cu intervalul de simulare citit din fisier, conditia din while va devein falsa si executia se va opri. La final, toate cozile vor fi oprite prin setarea variabliei toRun la valoarea false si de asemenea se va folosi metoda **interrupt**(). Timpul mediu de asteptare va fi calculat ca si timpul de asteptare impartit la numarul de clienti. Se printeaza in fisier. Se inchide fisierul.

* **MainClass**

In clasa **MainClass** se citesc din fisierul primit ca si argument in linia de comanda valorile necesare simularii, se face un obiect de tip SimulationManager, threadul principal, se foloseste metoda **setFilename**(outputFilename) pentru a trimite numele fisierului din args[1], cel de output care functia **makeOutputFile(),** se face un obiect de tip **Thread** cu argumentul s, pentru ca SimulationManager implementeaza interfata Runnable si se porneste threadul apeland metoda start().

1. Rezultate

Testarea a fost realizata in mod practic cu ajutorul exemplelor date.

A screenshot of a social media post

Description automatically generated

1. Concluzii

In urma realizarii acestei aplicatii mi-am imbunatatit abilitatile de programare pe obiecte si am invatat sa folosesc fire se executie.O posibila imbunatatire poate fi adusa la capitolul eficientei.

1. Bibliografie

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

<https://howtodoinjava.com/java/multi-threading/wait-notify-and-notifyall-methods/>

<https://www.javatpoint.com/java-thread-interrupt-method>

<https://www.journaldev.com/1020/thread-sleep-java>