DOCUMENTATIE

Tema numarul 2

Student: Rad Vladut
Grupa: 30225

CUPRINS

1. Obiectivul Temei

Scopul principal al temei este de a crea o aplicație eficientă care să organizeze clienții în cozi, asemenea celor din magazine, iar rezultatele să poată fi vizualizate în timp real, atât în interfața grafică, cât și într-un fișier.

Obiectivele secundare includ analiza problemei, modelarea scenariilor și cazurilor de utilizare, proiectarea aplicației într-un mod orientat pe obiecte (MVC), crearea unei diagrame UML de clase și pachete, utilizarea structurilor de date și a interfețelor definite, precum și implementarea algoritmilor necesari. În plus, se va prezenta descrierea fiecărei clase, inclusiv a câmpurilor și metodelor, iar rezultatele vor fi prezentate în interfața grafică și într-un fișier de tip text.

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

Analiza problemei

-Aplicatia lasa utilizatorii sa introduca datele pentru simulare;
-Aplicatia foloseste o strategie de timp pentru a introduce cat mai eficient clientii in cozi;
-Aplicatia lasa clientii sa porneasca simularea prin intermediul unui buton;
-Aplicatia afiseaza in timp real timpul,clientii care urmeaza sa intre in coada si continutul
cozilor,atat cu clienti cat si goale;
-Aplicatia afiseaza la finalul simularii rezultatele obtiunte;

Modelare

Am folosit conceptul de thread pentru a putea observa aceste rezultate in timp real in
interfata noastra grafica.Am pornit un thread principal,iar dupa am folosit conceptul de multi
threading,unde am mai pornit cate un thread pentru fiecare coada

Scenarii

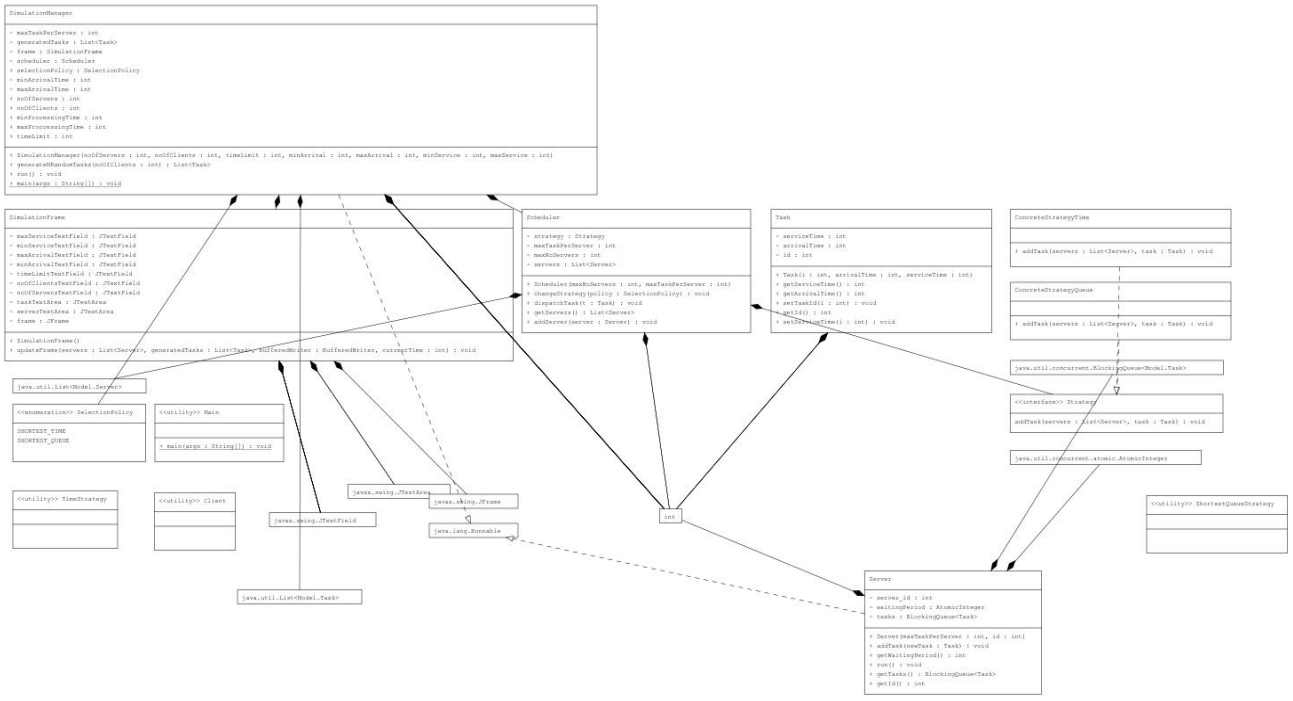
Fiecare scenariu de utilizare respecta,aproximativ,acelasi caz de utilizare,diferenta
reprezentand-o datele de intrare,ce pot sa difere de la o rulare la alt4
Utilizatorul trebuie sa introduca datele pentru numarul de clienti,numarul de cozi,intervalul
in care clientii generati aleatoriu se vor pune in coada,intervalul in care clientii generati aletoriu
vor fi serviti si timpul total de simulare al aplicatiei,in secunde.
In caz ca vor fi introduse date eronate de la tastatura,va aparea un mesaj de eroare.

Cazuri de utilizare

La apasarea butonului de “START”,dupa introducerea datelor,se va deschide o noua interfata
grafica in care vom vedea rezultatele dorite,iar tot in acelasi timp,prin intermediul OOP-ului vom
executa si partea de back-end necesara.

3.Proiectare

Proiectarea aplicației urmează principiile programării orientate pe obiecte, prin crearea claselor care încapsulează funcționalitățile necesare. Aceste clase sunt organizate în pachete separate, pentru a asigura o structură coerentă și ușor de urmărit. În cadrul proiectului, se vor utiliza trei pachete distincte, respectiv GUI (interfața grafică), Logical (logica de gestionare a cozilor) și Model (reprezentarea obiectelor și entităților din aplicație). Această structură va facilita citirea și înțelegerea codului, și va ajuta la menținerea unei bune organizări a proiectului.



Structuri de date folosite

Pentru a simula activitatea aplicației, fiecare client este generat aleatoriu și adăugat într-o listă de clienți. Cozile sunt simulate prin utilizarea unei structuri de tip BlockingQueue, care asigură siguranța firurilor de execuție (thread-safe). În plus, pentru a asigura siguranța în ceea ce privește timpul de așteptare al clienților în coadă, s-a utilizat o structură de tip AtomicInteger. Pentru a stoca cozile utilizate în cadrul aplicației, s-a utilizat o listă de obiecte de tip coadă (List<Queue>).

Algoritmi utilizati

În cadrul modelului aplicației, s-a implementat un algoritm prin care coada de clienți este utilizată ca un fir de execuție. Fiecare client așteaptă în coadă până când este procesat, timp în care timpul de așteptare este actualizat de către coadă. După ce un client este procesat, acesta este eliminat din coada sa.

În cadrul controller-ului, s-au utilizat algoritmi similari pentru a pune un client în coadă în funcție de strategia aleasă. Pentru strategia în care este pus la coadă cu cei mai puțini clienți, algoritmul caută coada cu cei mai puțini clienți. În cazul strategiei în care timpul de așteptare este minim, algoritmul caută coada în care timpul de așteptare este cel mai mic. În ambele strategii, când această coadă este găsită, clientul este adăugat.

În cadrul simularii, pentru a verifica dacă mai sunt clienți în cozi după ce s-a terminat timpul de simulare, s-a utilizat un algoritm. Pentru a adăuga un client la una dintre cozi, se verifică dacă timpul de sosire este mai mic sau egal cu timpul curent. De asemenea, s-a utilizat un

algoritm care verifică dacă toate cozile sunt pline. Astfel, dacă se ajunge în acest caz, clientul așteaptă până când cel puțin una dintre cozi este eliberată. Dacă acest caz nu este tratat, o parte din clienți ar fi fost "pierduți" în cadrul simularii, deoarece ar fi fost extrasi din coadă, dar nu ar fi putut fi adăugați în nicio altă coadă.

4. Implementare

Fiecare pachet contine cel putin cate o clasa, astfel, pachetul Bussines_logic este alcatuit din urmatoarele clase:

Clasa **ConcreteStrategyQueue** implementează interfața Strategy și este una dintre strategiile folosite pentru a adăuga o sarcină (Task) la un server. Această strategie alege serverul cu cel mai mic timp de așteptare (waitingPeriod) dintre toate serverele disponibile și adaugă sarcina la coada acestuia. Clasa include o metodă addTask care primește lista de servere și sarcina de adăugat și determină serverul cu cel mai mic timp de așteptare, apoi adaugă sarcina la coada acelui server. De asemenea, în cadrul metodei, se afișează un mesaj care indică faptul că a fost aleasă această strategie ("Queue").

Clasa **ConcreteStrategyTime** implementează interfața Strategy și definește o strategie de a adăuga o sarcină în coada unui server în funcție de timpul estimat. Mai exact, metoda addTask primește o listă de servere și o sarcină și alege serverul care are timpul estimat minim pentru a procesa sarcina. Timpul estimat pentru fiecare server este calculat prin adăugarea timpului de așteptare actual și timpul de serviciu al sarcinii. Dacă un server nu are nicio sarcină în așteptare, atunci timpul estimat pentru acesta este egal cu timpul de așteptare inițial al sarcinii. După găsirea serverului cu timpul estimat minim,

sarcina este adăugată în coada acelui server. Această clasă poate fi utilizată împreună cu un controller pentru a gestiona sarcinile într-un sistem cu mai multe servere și cozi.

Clasa **Scheduler** este o clasă care se ocupă de planificarea și coordonarea sarcinilor pe mai multe servere. Are o listă de obiecte Server, unde sunt stocate informații despre fiecare server și poate adăuga servere noi prin intermediul metodei addServer(). De asemenea, clasa are o strategie de selectare a serverului pentru fiecare sarcină nouă care vine prin intermediul metodei dispatchTask(). Strategia poate fi schimbată prin intermediul metodei changeStrategy(), care primește un enum SelectionPolicy care specifică tipul de strategie

In ceea ce privește implementarea, clasa Scheduler utilizează obiecte ale interfeței Strategy pentru a implementa strategiile specifice. Aceste obiecte sunt inițializate cu o strategie implicită (ConcreteStrategyTime) în constructorul clasei. Metoda dispatchTime() utilizează strategia curentă pentru a selecta serverul cel mai potrivit pentru sarcina curentă, pe baza listei de servere stocate.

Clasa **SimulationManager** este o clasă care gestionează simularea procesării de sarcini în cadrul unui sistem de servere. Aceasta implementează interfața Runnable și este folosită pentru a executa simularea într-un fir de execuție separat.

Clasa conține un număr de variabile de stare și metode de inițializare care permit configurarea parametrilor de simulare (numărul de servere, numărul de clienți, limitele de timp pentru sosirea și procesarea

sarcinilor etc.). De asemenea, clasa conține o instanță a clasei Scheduler și o instanță a clasei SimulationFrame.

Metoda generateNRandomTasks generează o listă de sarcini aleatorii, în funcție de parametrii de simulare specificați anterior. Aceste sarcini sunt sortate după timpul de sosire și sunt asociate unui ID unic.

Metoda run implementează bucla principală a simulării, care se execută până la atingerea timpului limită. În fiecare iterație, se verifică dacă există sarcini care trebuie să fie procesate la momentul curent și, în caz afirmativ, se trimite fiecare sarcină la un server disponibil, folosind obiectul Scheduler. De asemenea, metoda actualizează interfața grafică și scrie rezultatele într-un fișier de ieșire.

Metoda main este folosită pentru a crea și afișa interfața grafică.

Clasa **Strategy** este o interfață pentru strategiile de selectare a serverelor în cadrul unui sistem de simulare a unei cozi. Ea definește o singură metodă addTask(List<Server> servers, Task task) care trebuie implementată de toate clasele care implementează această interfață. Scopul acestei metode este de a adăuga o sarcină (Task) în coada (sau serverul) cu cel mai mic număr de sarcini, în funcție de strategia specifică. Implementarea acestei metode va varia în funcție de strategia aleasă (de exemplu, adăugarea sarcinilor în funcție de timpul de procesare sau lungimea cozii).

Pachetul **Model** este alcatuit din urmatoarele clase:

Clasa **Server** implementează logica unui server în simularea unei cozi de așteptare. Aceasta are un identificator, un BlockingQueue de obiecte Task care reprezintă sarcinile care așteaptă să fie preluate și un

AtomicInteger care ține evidența timpului de așteptare total al tuturor sarcinilor din coadă.

Metoda addTask() adaugă o nouă sarcină în coadă și crește timpul de așteptare total.

Metoda getWaitingPeriod() returnează timpul total de așteptare al tuturor sarcinilor din coadă.

Metoda run() reprezintă execuția unui fir de execuție care preia sarcinile din coadă și le procesează. Dacă există cel puțin o sarcină în coadă, se preia prima sarcină din coadă, se așteaptă un interval de timp de o secundă simulând procesarea sarcinii și se scade timpul de procesare rămas din sarcină. Dacă timpul de procesare a devenit 0, sarcina este eliminată din coadă.

Metoda getTasks() returnează coada de sarcini a serverului.

Metoda getId() returnează identificatorul serverului.

Clasa **Task** este utilizata pentru a modela sarcinile care trebuie indeplinite in contextul unui sistem de coada cu mai multe servere. Aceasta clasa are trei atribute: un identificator unic (id), momentul de sosire al sarcinii in sistem (arrivalTime) si timpul necesar pentru a indeplini sarcina (serviceTime). Clasa ofera metode pentru a accesa si actualiza aceste atribute.

Am atasat si un fisier, output.txt, care ma ajuta sa vad afisate datele generate dupa o rulare a unui program si introducerea unor date pentru a face asta. Acesta ar trebui sa arate astfel:

```
Time 4
Server 1: (1,2,2)
Server 2: (2,3,14)

Task 3: arrival time=7, service time=11
Task 4: arrival time=10, service time=1
Task 5: arrival time=12, service time=3
Task 6: arrival time=12, service time=16

Time 5
Server 1: (1,2,1)
Server 2: (2,3,13)

Task 3: arrival time=7, service time=11
Task 4: arrival time=10, service time=1
Task 5: arrival time=12, service time=3
Task 6: arrival time=12, service time=16
```

5. Rezultate

Pentru a testa rezultatele am folosit mai multe exemlpe,aleatorii,pe care le-am analizat cu atentie,dar si exemple aflate in cerinta proiectului:

Test 1	Test 2	Test 3
N = 4 Q = 2 $t_{simulation}^{MAX}$ = 60 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}]$ = [2, 30] $[t_{service}^{MIN}, t_{service}^{MAX}]$= [2, 4]	N = 50 Q = 5 $t_{simulation}^{MAX}$ = 60 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}]$ = [2, 40] $[t_{service}^{MIN}, t_{service}^{MAX}]$= [1, 7]	N = 1000 Q = 20 $t_{simulation}^{MAX}$ = 200 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}]$ = [10, 100] $[t_{service}^{MIN}, t_{service}^{MAX}]$= [3, 9]

6. Concluzii

In concluzie, proiectul la care am lucrat a fost unul dintre cele mai complexe, deoarece implică utilizarea notiunilor de thread și multithreading și cum acestea sunt combinate. Totodată, acest proiect a fost o oportunitate excelentă de a învăța lucruri noi și de a le pune în practică într-un mod inedit.

Pentru dezvoltarea ulterioară a proiectului, se poate conecta programul Java la o bază de date pentru a stoca informații despre clienți și cozile noastre. De asemenea, putem modifica cozile noastre pentru a fi folosite în magazine cu intervale orare specifice și angajați care să servească clienții.

7. Bibliografie

-https://dsrl.eu/courses/pt/materials/PT2023_A2_S1.pdf
-https://dsrl.eu/courses/pt/materials/PT2023_A2_S2.pdf
- https://www.javatpoint.com/multithreading-in-java
- https://www.geeksforgeeks.org/runnable-interface-in-java/