DOCUMENTATIE

TEMA *2*

QUEUES MANAGEMENT APPLICATION USING THREADS AND

SYNCHRONIZATION MECHANISMS

NUME STUDENT: CIU ADRIAN-VALENTIN
GRUPA: 30227

CUPRINS

1. Obiectivul temei

Obiectivul principal al temei a fost implementare unei aplicatii ce reuseste in mod efficient sa aranjeze "clientii" la cozi, rezultate putand fi vazute in timp real in interfata grafica sau analizate ulterior din logfile.

Obiectivele secundare sunt reprezentate de:

- Analiza problemei, modelare, scenarii, cazuri de utilizare → Se prezinta cadrul de cerinte functionale formalizat si cazurile, cazurile de utilizare → prezentat in capitolul 2
- Proiectare → Vor fi prezentate modul in care fost proiectata aplicatia in POO (MVC), diagrama UML de clase si pachete, structurile de date folsite, interfetele definite si algoritmii folositi → prezentat in capitolul 3
- Implemantare → Descrierea fiecarei clase cu campuri si metode→ prezentat in capitolul 4
- Rezultate → Prezenatarea scenariilor de testare cu Junit→ prezentat in capitolul 5

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

2.1 Analiza problemei

Cerinte functionale:

-Aplicatia lasa utilizatorii sa introduca datele pentru simulare;
-Aplicatia lasa utilizatorul ce strategie doreste sa aplice pentru a aranja clientii la cozi;
-Aplicatia lasa utilizatorii sa dea drumul la simulare;
-Aplicatia afiseaza in timp real lista de client ce nu au fost pusi la nici o coada;
-Aplicatia afiseaza in timp real continutul fiecarei cozi;
-Aplicatia afiseaza la finalul simularii rezultatele obtinute in urma simularii.

2.2 Modelare

Pentru a putea lucara cu thread-uri, a fost necesare extinderea clasei Thread sau a implementarii interfetei Runnable. Astfel, a fost posibila realizarea conceptului de multithreading pentru a putea simula aplicatia in timp real.

2.3 Scenarii

Fiecare scenariu de utilizare respceta aproximativ acelasi caz de utlizare, diferenta reprezentand-o datele de intrare ce pot sa difere de la efectuarea unei simulari la alta.

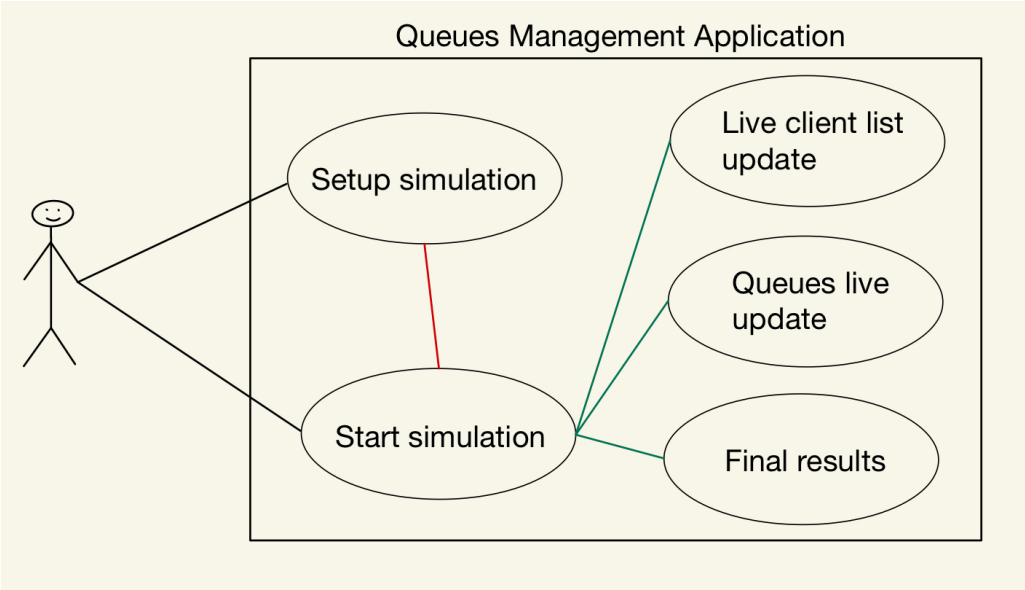
Scenariu de utilizare: simularea unei cozi;
Actor principal: utilizatorul
Scenariu de success:

- Utilizatorul introduce datele pentru numarul de client, numarul de cozi, intervalul de simulare, numarul maxim de client ce se pot afla la o coada, minimul si maximul timpului de sosire, minimul si maximul timpului de servire;
- Utilizaotrul apasa pe butonul de start ce porneste aplicatia;
- Aplicatia valideaza datele si porneste aplicatia.

Scenarii alternative:

- Utilizatoul introduce valori invalide pentru simualrea aplicatie;
- Aplicatia afiseaza un mesaj in care specifica eroarea si il pune pe utilizator sa introduca din nou datele;
- Scenariul se intoarce la primul pas.

2.4 Cazuri de utilizare

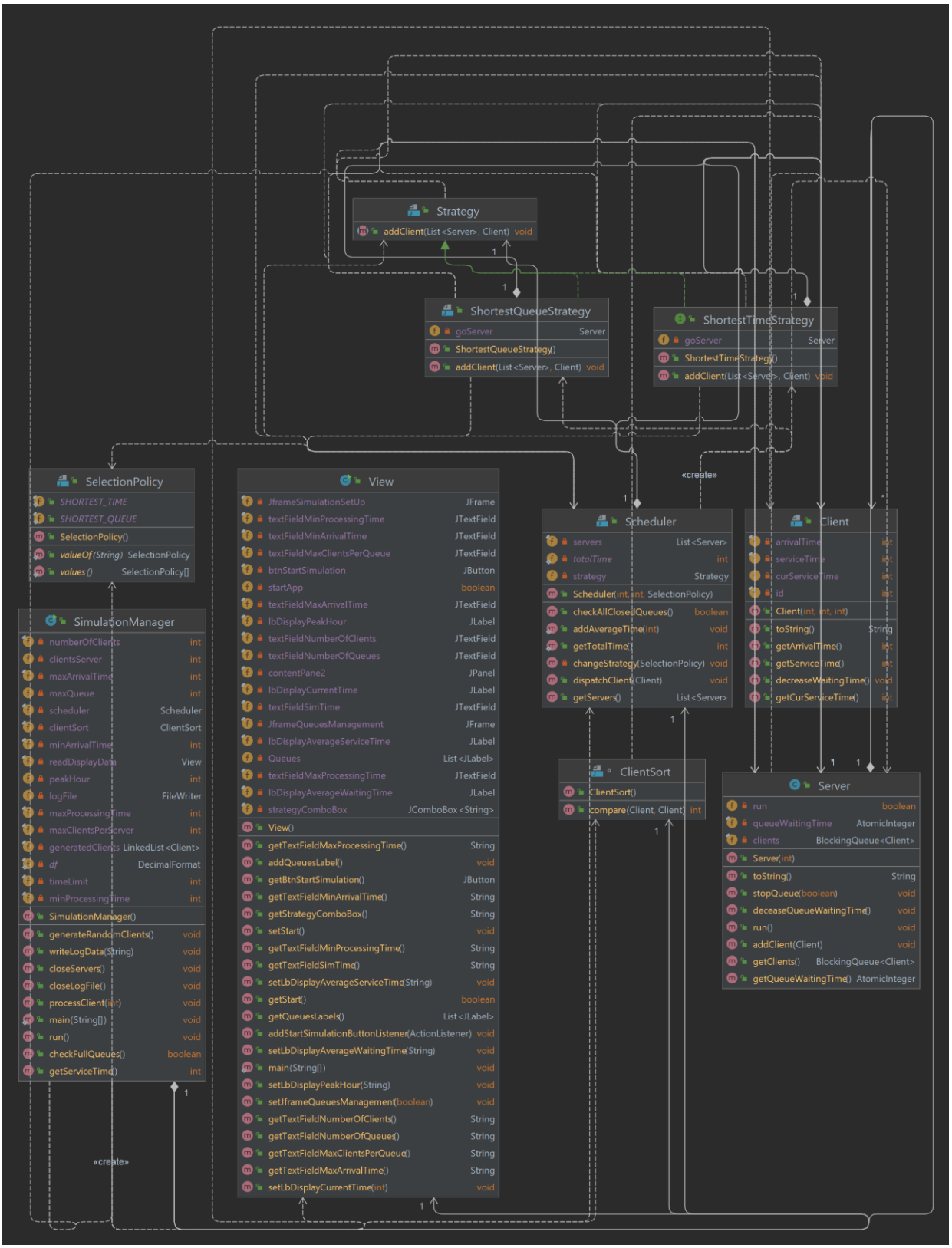


3. Proiectare

3.1 **Proiectarea aplicatiei**

Aplicatia a fost dezvoltata folosind abordarea de Model View Controller. In model au fost implementati algoritmii pentru a folosi un thread drept o coada de client. In View au fost implementatata interfata grafica pentru utilizator ce permite sa introducem datele pentru a simula managerierea unuei cozi de clienti din vaiata reala. De asemenea, se poate selecta strategia in functie de care fiecare client este pus la coada. Partea de Controler se ocupa cu de implementarea action listener-ului pentru butonul de start a aplicatiei. De asemenea, aici este este procesat scenariul de simulare introdus in interfata grafica.

3.2 **Diagrama UML → Unified Modeling Language**



3.3 **Structurile de date**

Fiecare client care care a fost generat random pentru simularea aplicatieie a fost pus intr-un LinkedList. Pentru coada noua create a fost pusa tot intr-un LinkedList. A fost necesara aceasta structura de date deoarece pentru a extrage un client am utilizat metoda **pop()**. De asemenea, pentru numarul total de cozi puse intr-o lista nu conta daca folosim insatantierea de la ArrayList sau LinkedList deoarece mereu parcurgeam cu un for each toate cozile din lista pentru a extrage statusul in care se afla clientii. Pentru a fi thread safe, am folosit strucutra de date AtomicInteger pentru a transmite timpul pe care trebuie sa-l astepte un client care intra in coada. Tot pentru a fi thread safe am folosit structura de date BlockingQueue pentru a pune clientii la coada fiecarui server.

3.4 **Interfetele folosite**

In aplicatia s-au implementat doua strategii pentru a determina la care dintre cozi un client este pus. Prima strategie consta in a pune mereu clientul la coada cea mai mica, iar cea de-a doua in a pune clientul la coada in care are cel mai putin timp de astept pana la servire. Aceste doua strategii au fost implementate folosing interfata Strategy. De asemenea, pentru a sorta lista de clienti crescator in functie de timpul de sosire, a fost utilizata interfata Comparator. Avand in vedere ca este o aplicatie multithreading, a fost utilizata interfata Runnable.

3.5 **Pachetele utilizate**

Proiectul realizat a fost implementat utilizant trei pachete: GUI (reprezinta View-ul), BusinessLogic (reorezinta Controller-ul) si Model. Aceste pachete au fost alese astfel incat sa se respecte implemenatre Model View Controller.

3.6 **Algoritmii utilizati**

In cadrul modelului am implementat algoritmul pentru a folosi o coada ca un thread. Fiecare client asteapta la coada pana cand ii vine randul. In tot acest interval, timpul de asteptare este updatat de catre coada. Dupa ce un client a fost procesat, acesta este eliminat din coada sa.

In cadrul controller-ului am folosit algoritmi asemanatori pentru a pune un client in coada in functie de strategia aleasa. Pentru strategia in care este pus la coada cu clienti cei mai putini, algoritmul cauta coada cu cei mai putin client. In cazul strategie in care timpul de asteptare este minim, algorimul cauta coada in care timpul de asteptat este cel mai mic. In ambele strategii, atunci cand aceasta coada este gasita, clientul este adugat. In cadrul simularii, pentru a ne da seama daca mai sunt clienti la cozi dupa ce s-a terminat timpul de simulare, am un utilizat un algoritm ce verifica acest lucru. Pentru a aduga un client la una dintre cozi se verifica daca timpul sa de sosire este mai mic sau egal cu timpul current (mai mic sau egal deoarece poate exitsa cazul in care pot sa vina mai multi clienti in acelasi timp). De asemenea, am folosit un algoritm ce verfica daca toate cozile sunt pline. Astfel, daca ne aflam in acest caz, clientul asteapata pana cand cel putin una dintre cozi este eliberata (daca nu as fi tratat acest caz, in cadru simularii “as fi pierdut” o parte din clientini deoarece i-as fi extras din coada, dar acesti nu ar fi putut fi pusi la nicinua).

4. Implementare

In fiecare pachet se afla cel putin o clasa.

Pentru pachetul **Model** am impplemenata urmatoarele clase:

- **Client**

Clasa Client are drept scop crearea unui nou client ce va fi ulterior pus la una dintre cozi. Acesta are trei attribute: id → reprezinta id-ul generat pentru clientul respective, serviceTime → reprezinta timpul de servire atunci cand o sa-i vina randul si curServiceTime → este folosit pentru a face update la cat timp mai are de asteptat la coada atunci cand este servit. Metoda toString() este suprascrisa pentru a afisa in mod corespunzator datele despre un client : “(id, arrival time, service time);”.

- **Server**

Aceasta clasa este utilizata pentru a implementa un thread ce va reprezenta o coada de clienti in metoda **run()**. Aceasta utilizeaza un BlockingQueue in care sunt clientii pusi in asteptare. Metoda **stopQueue()** este utilizata pentru a opri thread-ul atunci cand simularea s-a terminat. Metoda **addClient()** are derept scop sa adauge in coada un client si sa creasca timpul de asteptare in functie de timpul de servire al clientului. Metoda **toString()** este suprascrisa pentru a afisa clientii ce se afla in coada sau, in cazul in care coada este goala (metoda **size()** returneaza 0), “closed”.

Pentru pachetul **BusinessLogic** am implementat urmatoarele clase:

- **ShortestTimeStrategy**

Aceasta clasa este utilizata pentru a determina la care din cozile din simulare un client trebuie adaugat ca sa ateptepte pana in momentul servirii, in functie de timpul cel mai scurt de asteptare. Algotimul de functionare ste descries la punctul 3.6. Implementaza doar metoda din interfata Strategy.

- **ShortestQueueStrategy**

Aceasta clasa este utilizata pentru a determina la care din cozile din simulare un client trebuie adaugat ca sa ateptepte pana in momentul servirii, in functie de coada cea mai scurta. Algotimul de functionare ste descries la punctul 3.6. Implementaza doar metoda din interfata Strategy.

- **Scheduler**

Aceasta clasa este utilizata pentru a crea thread-urile pentru cozi si a le pune intr-o lista. Acest lucru este realizat in constructor. Metoda **changeStrategy()** este utilizata pnetru a schimba strategia in functie de care un lcinet este adaugat. In functie de parametrii dati constructorului se stabileste strategi ace urmeaza a fi folosita. Metoda **dispatchClient()** are drept scop sa adauge un client la o coada, folosind strategia instantiate. Metoda **checkAllClosedQueues()** este folosita pentru a determina daca mai sunt clienti la cel putin una dintre cozi.

- **SimulationManager**

Aceasta clasa este utilizata pentru a creea cadrul de simulare pentru aplicatie. In constructor este implementat, folosind o lamba expression, butonul de start din interfata grafica. Tot in constructor sunt initializate toate atributele cu valori introduse de catre utilizator in interfata grafica. Metoda **generateRandomClients()** are drept scop genrarea random a clientilor in fucntie de atributele numberOfClients, minProcessingTime, maxProcessingTime, minArrivalTime si maxArrivalTime. Aceasta metoda este utilizata in constructor. Tot in constructor fiecare thread (coada) este pornit si ii este pus un nume corespunzator pentru partea de interfata grafica. Metoda getdata() are drept scop validarea input-ului de la tastatura. In cazul in care input-ul este invalid, utilizatorul trebuie sa introduca datele pana cand acestea sunt corecte.Metoda **writeLogData()** are drept scop sa scrie in fisierul log.txt si sa afiseze pe ecran mesajul trimis (String) trimis ca parametru. Metoda **getServiceTime()** are drept scop sa returneza timpul total de asteptatre al tuturor clientilor. Metoda **processClient()** are drept sa scop sa proceseze un client: verifica daca are arrivalTime corespunzator si daca exista cel putin o coada care nu este plina. Daca aceste condintii sunt indeplinite, atunci clientul este trimis la coada corespunzatpare prin metoda **dispathClient()** si este scos din lista de asteptare a clientilor. Metoda **closeServer()** are drept scop sc inchide thread-urile (cozile) dupa ce simulare a luat sfarsit. Metoda **checkFullQueues()** are drept scop sa afle daca exista cel putin o coada ce mai are un loc liber pentru un client. In metoda **run()** sunt folosite meotdele descries mai sus, tinandu-se cont si de timpul current de simulare.

Pentru pachetul **GUI** am implementat urmatoarea clasa:

- **View**

In construcotul acestei clase se realizeaza cele doua ferestre ale interfetei grafice: prima este folosita pentru introducerea datelor de simulare. Dupa ce se apasa butonul de start, prima fereastra se inchide si se deschide fereastra in care se poate observa in timp real modul de functionare al aplicatiei. Acest tranzitia de la o fereastra la alta este realizata de metoda **setJframeQueuesManagement()**. Pentru a putea genera label-uti pentru cozi in functie de numarul introdus de la tastatura am implementat metoda addQueuesLabel().

5. Rezultate

Pentru a testa aplicatia am utilizat mai multe cazuri de input de la tastatura, dar si testele prezentatae in eneutul temei. Astfel, am obtinut urmatoarele rezultate in urma testelor:

- **Test1:**

N = 4
Q = 2
$tsimulationMAX$ = 60 seconds
[$tarrivalMIN$,$tarrivalMAX$] = [2, 30]
[$t_{serviceMIN}$,$t_{serviceMAX}$]= [2, 4]

Rezultate:

Average waiting time: 3.00
Average service time: 2.25
Peak Hour: 3

- **Test2:**

N = 50
Q = 5
$tsimulationMAX$ = 60 seconds
[$tarrivalMIN$,$tarrivalMAX$] = [2, 40]
[$tserviceMIN$,$tserviceMAX$]= [1, 7]

Rezultate:

Average waiting time: 8.62
Average service time: 3.10
Peak Hour: 4

- **Test3:**

N = 1000
Q = 20
$tsimulationMAX$ = 200 seconds
[$tarrivalMIN$,$tarrivalMAX$] = [10, 100]
[$tserviceMIN$,$tserviceMAX$]= [3, 9]

Rezultate:

Average waiting time: 17.77
Average service time: 28.96
Peak Hour: 12

6. Concluzii

Acesta a fost unul dintre cele mai complexe proiecte la care am lucrat deoarece a trebuit mai intai sa inteleg conceptul de multithreading si cum functioneaza. Dupa ce am studiat sufficient acest aspect, partea de implemenetare a fost relative usoara. Am reusit ca maximul de linii pe care o are o clasa sa fie de 110 linii si am respectat ca fiecare metoda sa nu aiba mai mul de 30 de linii. Din aceasta tema am invata cum plot sa fac o aplicatie ce utiliezeaza un numar variabil de thread-uri.

Ca dezvoltari ulterioare, acest proiect ar putea sa fie conectata la o baza de date ce continue client adevarti. De asemenea, se poate extinde spre o aplicatie ce aranjeaza clientii in mod optim la mese intr-un restaurant, de exemplu.

7. Bibliografie

-http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

- http://www.tutorialspoint.com/java/util/timer_schedule_period.htm

- http://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html