

**Documentatie Tema 2**

**Aplicatie pentru gestionarea clientilor**

**Si minimizarea timpului de asteptare**

**Codrescu Razvan-Gabriel**

**Grupa 30224**

**Profesor Laborator Antal Marcel**

**Cuprins**

**1.Obiectivul temei**

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

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

**4. Implementare**

**5. Cazuri de testare**

**6.Use-case**

**7. Concluzii, ce s-a invatat din tema, dezvoltari ulterioare**

**1.Obiectivul temei**

Principalul scop al acestei teme este proiectarea si implemetarea unei aplicatii pentru gestionarea clientilor, in functie de numarul de cozi deschise si a timpului de asteptare a fiecarei coada.

Scopul aplicatiei este de a identifica timpul minim de asteptare pentru fiecare client, pentru a eficientiza un astfel de sistem. Prin gestionarea clientilor, se intelege a dispune clientii in cozi astfel incat sa sa astepte cat mai putin la coada si pentru a-si termina nevoile cat mai repede.

Pentru aceasta tema s-au efectuat urmatorii pasi pentru realizarea programului dorit:

1.crearea claselor necesare;

2.stabilirea fiecarei caracteristici pentru obiectele utilizate;

3.implementarea metodelor necesare;

4.crearea si gestionarea cozilor si a clientilor.

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

Principiile de baza sunt:

**1.Abstractizare** = Este posibilitatea ca un program să separe unele aspecte ale informației pe care o manipulează, adică posibilitatea de a se concentra asupra esențialului. Fiecare obiect în sistem are rolul unui “actor” abstract, care poate executa acțiuni, își poate modifica și comunica starea și poate comunica cu alte obiecte din sistem fără a dezvălui cum au fost implementate acele facilitați.Are loc dupa analiza problemei si specificarea cerintelor.

**2.Încapsularea** = numită și ascunderea de informații: Asigură faptul că obiectele nu pot schimba starea internă a altor obiecte în mod direct (ci doar prin metode puse la dispoziție de obiectul respectiv); doar metodele proprii ale obiectului pot accesa starea acestuia. Fiecare tip de obiect expune o interfață pentru celelalte obiecte care specifică modul cum acele obiecte pot interacționa cu el.

**3.Polimorfismul** = Este abilitatea de a procesa obiectele în mod diferit, în funcție de tipul sau de clasa lor. Mai exact, este abilitatea de a redefini metode pentru clasele derivate. De exemplu pentru o clasă Figura putem defini o metodă arie. Dacă Cerc, Dreptunghi, etc. ce vor extinde clasa Figura, acestea pot redefini metoda arie.

**4.Moștenirea** = Organizează și facilitează polimorfismul și încapsularea, permițând definirea și crearea unor clase specializate plecând de la clase (generale) deja definite - acestea pot împărtăși (și extinde) comportamentul lor, fără a fi nevoie de a-l redefini. Aceasta se face de obicei prin gruparea obiectelor în clase și prin definirea de clase ca extinderi ale unor clase existente. Conceptul de moștenire permite construirea unor clase noi, care păstrează caracteristicile și comportarea, deci datele și funcțiile membru, de la una sau mai multe clase definite anterior, numite clase de bază, fiind posibilă redefinirea sau adăugarea unor date și funcții noi.

Programul ce urmeaza sa fie creat, va avea ca date de intrare, date extrase dintr-un fisier(Ex In-Test1.txt) si pentru afisarea procesului prin care trece toata operatiunea de gestionare si procesare a cozilor si clientilor, vor fi scrise intr-un fisier(Ex. Out-Test1.txt). In urma consultarii cerintelor pentru aceasta tema, programul nu va dispune de o interfata grafica. Acesta va efectua operatii de creare a cozilor si clientilor, introducerea de clienti in coada daca conditiile necesare sunt indeplinite si scoaterea acestora din coada dupa procesarea lor. Programul trebuie sa fie cat mai clar si mai usor de urmarit si de asemenea sa prezinte un nivel de eficienta rezonabil.

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

Pentru rezolvarea cerintei, am implementat urmatoarele clase: Task, Server, Scheduler si SimulationManager. Clasa Strategy nu a fost implementata datorita faptului ca se va folosi o singura strategie prestabilita.

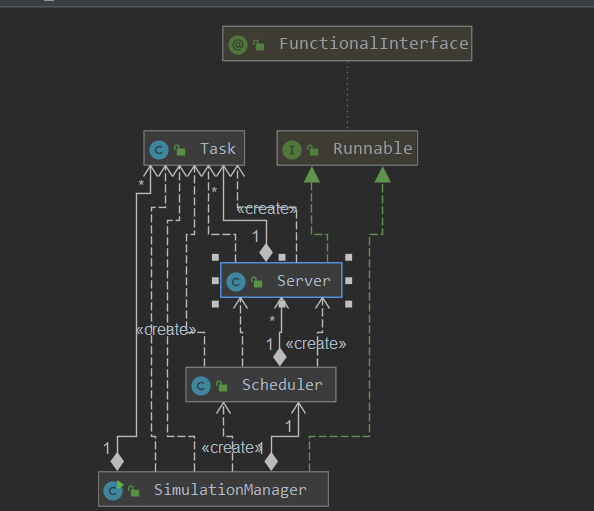
Fiecare metoda cu variabile declarate de tipul private prezinta gettere si settere pentru accesarea continutului acestora.

Clasa **Task** reprezinta clasa “Client”. Fiecare client este procesat in functie de ID-ul sau(numarul sau de ordine), **arrivalTime** care reprezinta timpul acestuia de sosire in “Institutie” si **processingTime** ce reprezinta timpul de procesare al acestuia(timpul in care clientul a asteptat la coada si a fost procesat).

Clasa **Server** reprezinta clasa “Coada”. Aici fiecare client va fi dispus in functie de timpul la care a sosit. Aceasta este reprezentata prin urmatorii parametri : **tasks** care este o listade clienti si **waitingPeriod** ce reprezinta perioada de asteptare la fiecare coada. waitingPeriod este de tipul **AtomicInteger**, un tip de Integer folosit de obicei cu doua scopuri: ca si numerator care e folosit de thread-uri multiple in acelasi timp sau in operatii de comparare si interschimbare. Sunt folosite metode speciale ale acestui tip precum decrementAndGet care este echivalentul unui -- executat pe o variabila de tip Integer. **Tasks** este un **BlockingQueue**, care de asemenea este avantajos in implementarea cu thread-uri. Aceasta natura de “blocare” are cateva avantaje. Un prim avantaj este reprezentat de faptul ca daca capacitatea sirului este limitata, si consumul de memorie va fi limitat de asemenea. Un alt avantaj este acela ca extragerea elementelor din coada este simpla.

Clasa **Scheduler** se ocupa cu asezarea Clientilor in Cozi dupa un anumit model stabilit de noi si calcularea timpului mediu de asteptare. Ca si parametrii prezinta o lista de Cozi – **servers** – in care se va tine evidenta tuturor cozilor, **maxNoServers** – numarul de cozi existente , **maxTasksPerServer** – cati client poti fi asezati la o coada si **avgWaitingPer** – timpul de asteptare mediu pentru fiecare coada in parte.

In final, clasa **SimulatitonManager** va prezenta cateva amanunte despre implementarea finala a problemei. **timeLimit** – timpul de rulare maxim al aplicatiei, **maxProcessingTime** si **minProcessingTime** – timpul minim si maxim de procesare admis, **numberOfServers** si **numberOfTasks** – numarul de client si de cozi admise, **scheduler** – organizatorul pentru cozi si **generatedTasks** – care reprezinta clientii totali generati, cu tot cu datele despre ei.



**4. Implementare**

In clasa **Task** vom avea doar constructorul clasei si gettere si settere pentru fiecare variabila declarata ca parametru al clasei.

In clasa **Server** avem un constructor pentru initializare a perioadei de asteptare si a listei de client. O prima metoda a acestei clase este **public void addTask(Task newTask),** cu ajutorul careia se adauga un client in lista si se incrementeaza **waitingPeriod** cu ajutorul metodei incrementAndGet(). Datorita faptului ca Server implementeaza **Runnable** vom avea o metoda run() unde, cat timp aceasta ruleaza si cat timp lista de client nu este goala, se vor extrage elemente, se va decrementa perioada de asteptare cu metoda decrementAndGet() deoarece un client s-a terminat de procesat si va elibera coada si se va face sleep() la thread-ul in interiorul unor metode de try{}catch. De asemenea, pe langa gettere si settere exista o metoda **public void toStr()** folosita pentru afisarea tuturor clientilor existenti in coada accesata.

In clasa **Scheduler** mai intai se vor crea cozi in functie de numarul maxim de cozi admis, fiecare coada reprezentand cate un thread si fiecare client cate un proces. Aceste thread-uri vor fi create si pornite. In urma lipsei clasei Strategy, metoda de distribuire a fost implementata aici **public void distributionMethod(Task task)**. Aceasta metoda primeste un client si dupa o parcurgere a tuturor cozilor care nu au atins numarul maxim de clienti, se va stabili care coada are cel mai bun timp de procesare la momentul actual si clientul va fi dispus la acea coada cu ajutorul metodei addTask(). Tot in aceasta clasa, se va calcula perioada medie de asteptare totala.

Ultima clasa, **SimulationManager**, se ocupa cu simularea a 60 de secunde de procesare a clientilor si trecerea lor prin cozi. In prima faza va fi stabilita metoda dupa care acestia vor fi introdusi in cozi, aceasta fiind una realizata in functie de timpul de asteptare. Se va crea coada noua – thread – si se vor citi datele dintr-un fisier cu ajutorul unui Scanner incapsulat intr-un try{]catch, in caz ca deschiderea fisierului nu a fost posibila. Fiecare parametru al functiei va lua dupa aceea valoarea prestabilita in fisier. O urmatoare metoda prezentata este **private void generateRandomTasks()** care se va ocupa cu generarea unui numar stabilit de clienti si adaugarea ei intr-o structura de tip lista, fiecare element avand timp de sosire si timp de procesare aleatoare si cu ID-ul de creare. Ca si **Server**, aceasta clasa implementeaza **Runnable** pentru a utiliza functia run. In aceasta functie se va face scrierea in fisier a datelor. Se va lua un timp curent care initial va fi 0 si cat timp acel timp nu a ajuns la limita maxima de timp prestabilita, va continua sa scrie in fisier sis a proceseze clientii daca timpul curent s-a ajuns la timpul de sosire al acestuia. In cazul in care s-a ajuns la timpul de sosire al acestuia, acesta se distribuie la coada dupa metoda aleasa mai sus si se va recalcula timpul mediu de asteptare. In momentul executarii unei iteratii, timpul curent va creste iar perioada de asteptare va scadea.

Exista conditia ca daca toate cozile sunt goale si lista de asteptare este goala, atunci programul se va opri. dataRdWr() se ocupa cu citirea din fisier a datelor, folosindiu-se de o variabila de tip scanner cu ajutor careia se va parcurge fisierul. Fisierul va fi citit linie cu linie cu ajutorul metodei next.Line() si se va transforma in inturi cu ajutorul metodei parseInt. Toata aceasta procedura este pusa intru-un try{}catch pentru a determina daca fisierul a putut fi deschis cu succes. In final, programul se va incheia cand toate cozile sunt goale si nu mai exista niciun client care sa astepte. La incheierea sa, se va afisa timpul mediu de asteptare si se va inchide fisierul de scriere. Fiecare sleep() care este apelat in program va avea sleepTime-ul egal cu timpul de procesare al fiecarui client, inmultit cu 1000, datorita faptului ca sleep (1) va face sleep timp de o milisecunda.

**5. Cazuri de testare**

Pentru cazuri de testare vor fi generate 3 fisiere cu dimensiuni diferite. Aceste fisere trebuiesc denumite si inlocuite in structura programlui, pentru a putea fi procesate. Datele in fisier sunt scrise cu ajutorul unei variabile writer de tip PrintWriter, aceasta apeland metoda de .write(), care poate scrie stringuri sau inturi in fisier.

**6.Use-case**

1.Utilizatorul va crea un fisier dupa detaliile prezentate in cerinta problemei:

-pe prima linie se va afla numarul de client

-pe a doua linie se va afla numarul de cozi

-pe a treia linie va fi perioada maxima de simulare

-pe a patra linie va fi intervalul de timp in care clientii vor merge in cozi

-pe ultima linie va fi intervalul de asteptare maxima a unui client.

2.Utilizatorul va rula programul.

3.Utilizatorul va verifica fisierul de accesare al proiectul, deoarece in urma rularii sale, au fost create fisiere cu date de iesire pentru simularea fiecarei situatii introduse in primul fisier.

**7. Concluzii, ce s-a invatat din tema, dezvoltari ulterioare**

In urma realizarii acestei teme am reusit sa invat metode noi de implementare, mi-am reactualizat cunostintele despre thread-uri si am invatat sa lucrez cu variabile de tip AtomicInteger si BlockedQueue, cat si rolul lor si nivelul de eficienta pe care il ofera. De asemenea, am invatat cum pot fi procesate thread-urile intr-o ordine stabilita de catre programator.

Ca si dezvoltari ulterioare se pot define:

1.Crearea unei interfete grafice pentru program, facand astfel mai usor de inteles pentru utilizator si mai interactiv. Aceasta aplicatie ar putea contine urmatoarele optiuni: setarea fiecarui parametru necesar rularii programului(numarul de clienti, de cozi, timpul de procesare si timpul de sosire), o schema de vizualizare mai buna a metodei de asociere a fiecarul client unei cozi.

2.Utilizarea mai multor metode de asezare a clientilor in cozi(dupa cea mai scurta coada spre exemplu). Realizand mai multe strategii, eficienta programului va creste datorita calcularii timpului de procesare al fiecarui client. Un alt exemplu de strategie ar putea fi includerea unui interval de timp in care clientii sa nu fie procesati, ca un fel de pauza de masa spre exemplu.

3.O sincronizare mai buna a thread-urilor programului, avand in vedere faptul ca programul nu este la capacitatea sa maxima.

4.Adaugarea unei metode in care daca se depaseste numarul maxim de clienti, sa se deschida o noua coada pentru procesare mai rapida a acestora.

.5.Un interval maxim de clienti care poate incapea intr-o coada.