MINISTERUL EDUCAŢIEI ŞI CERCETĂRII ŞTIINŢIFICE
UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Assignment 2

QUEUES SIMULATOR

Documentatie

MOLDOVAN ADELINA-STEFANIA

GRUPA 30226

CUPRINS

1. Obiectivul temei

Obiective principale

Proiectarea și implementarea unei aplicații de simulare care vizează analiza sistemelor bazate pe cozi pentru determinarea și minimizarea timpului de așteptare al clienților. Cozile sunt utilizate în mod obișnuit pentru modelarea domeniilor din lumea reală. Principalul obiectiv al unei cozi este să furnizeze un loc pentru ca un „client" să aștepte înainte de a primi un „serviciu". Gestionarea bazată pe cozi este interesata sa minimizeze timpul în care „clientii"asteapta la coada înainte sa fie serviti. O modalitate de a minimiza timpul de așteptare este de a adăuga mai multe servere, adică mai multe cozi în sistem (fiecare coadă este considerată ca având un procesor asociat), dar această abordare mărește costurile furnizorului.

Atunci când se adaugă un nou server, clienții care așteaptă vor fi distribuiți în mod egal pentru toate cozile curente disponibile. Sistemul ar trebui să simuleze o serie de clienți care sosesc pentru serviciu, introducând cozi, așteptând, servind și lăsând în sfârșit coada. Urmări timpul în care clienții petrec așteptarea în cozile de așteptare și scoate timpul mediu de așteptare. Pentru a calcula timpul de așteptare, trebuie să știm timpul de sosire, timpul de finalizare și timpul de serviciu. Timpul de sosire și timpul de serviciu depind de clienții individuali - când apar și de câte servicii au nevoie.

Timpul de terminare depinde de numărul de cozi, de numărul de alți clienți aflați în coada de așteptare și de nevoile de servicii ale celorlalți clienți.

Obiective secundare

1 . - dezvoltarea de use case-uri : folosim use case-uri (caz de utilizare) pentru a reprezenta cerinţe ale utilizatorilor. Sunt descrise acţiuni pe care un program le execută atunci când interacţionează cu actori şi care conduc la obţinerea unui rezultat .

2 . - dezvoltarea de diagrame UML : folosim diagrame UML pentru a înregistra relaţiile dintre clase .

3 . - implementarea unor clase in java : sunt descrise clasele folosite si rolul acestora .

4 . – dezvoltarea algoritmilor : procesul de dezvoltare al algoritmilor utilizati pentru acest proiect.

5 . – utilizarea threadurilor - Conceptul de thread (fir de executie) este folosit in programare pentru a eficientiza executia programelor, executand portiuni distincte de cod in paralel, in interiorul aceluiasi process.

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

Faza de rezolvare a problemei

1. *Analiza* înseamnă înţelegerea, definirea problemei şi a soluţiei ce trebuie dată;

2. *Algoritmul* presupune dezvoltarea unei secvenţe logice de paşi care trebuie urmaţi pentru rezolvarea problemei;

3. *Verificarea* este parcurgerea paşilor algoritmului pe mai multe exemple pentru a fi siguri că rezolvă problema pentru toate cazurile.

Faza de implementare

1. *Programul* reprezintă translatarea algoritmului într-un limbaj de programare

2. *Testarea* este etapa în care ne asigurăm că instrucţiunile din program sunt urmate corect de calculator. În situaţia în care constatăm că sunt erori, trebuie să revedem algoritmul şi programul pentru a determina sursa erorilor şi pentru a le corecta. Aceste două faze de dezvoltare a programului sunt urmate de faza de utilizare a programului care înseamnă folosirea acestuia pentru rezolvarea problemelor reale, cele pentru care a fost conceput. Ulterior pot interveni modificări ale programului fie pentru a răspunde noilor cerinţe ale utilizatorilor, fie pentru corectarea erorilor care apar în timpul utilizării şi care nu au putut fi găsite în faza de testare.

Algoritmul reprezinta o succesiune de pasi care duc la rezolvarea unei probleme. In cazul problemei noastre avem urmatoarea succesiune de pasi:

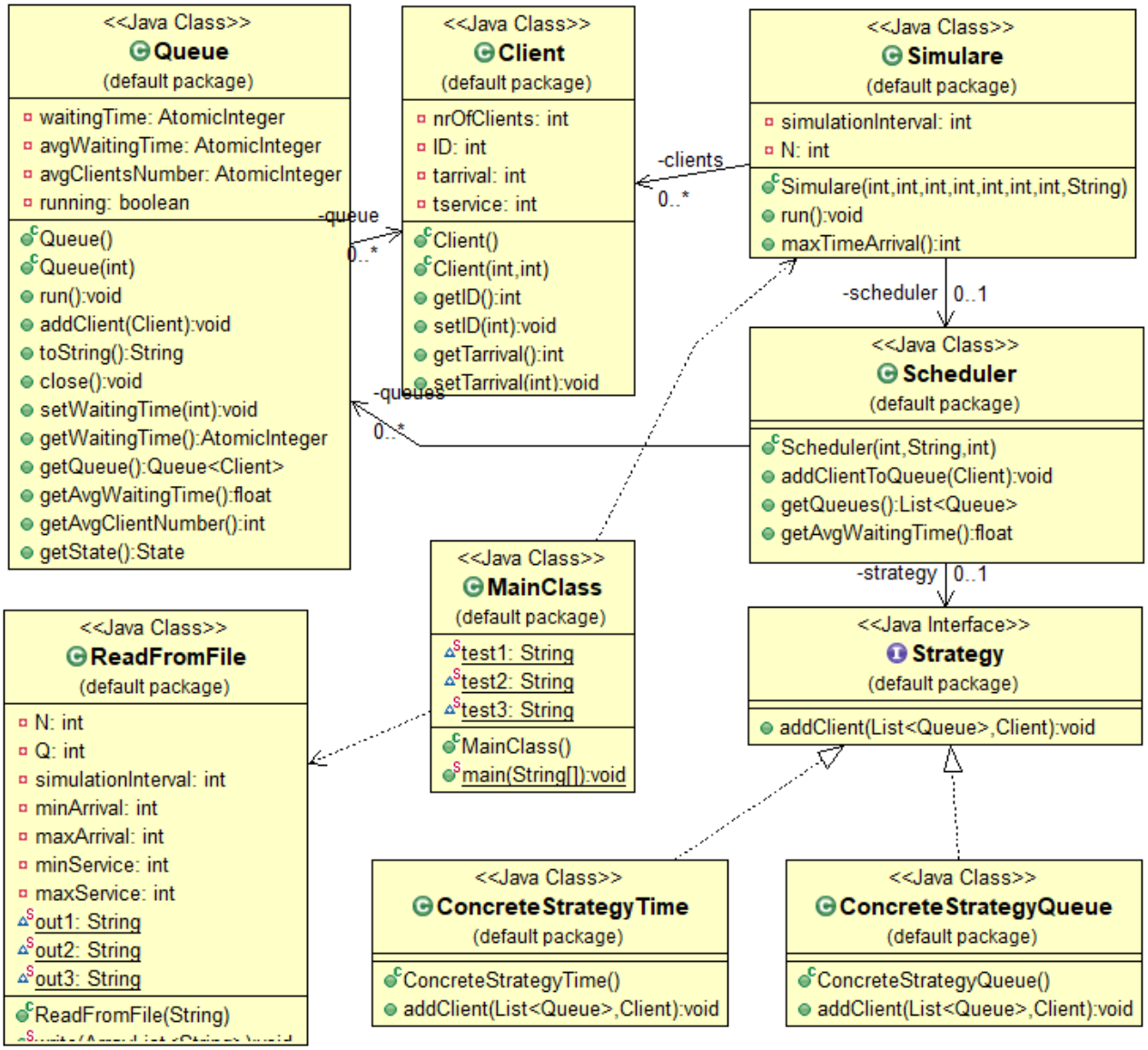
1. Citirea datelor din fisier
2. Generearea aleatoare a N clienti
3. Creare cozi si distribuire clienti la cozi
4. Iesirea clientilor si incchiderea cozilor
5. Calcularea timpului mediu de asteptare

Use case:

Nume use case: preluarea datelor dintr-un fișier
Actori: clientii din cozi
Scenariu:
- preluarea datelor
- clientii intra in cozi
- clientii sunt serviti si ies din cozi
- cozile raman goale si se incheie rularea
- calculare timp mediu de asteptare
- scriere rezultate in fisier

3.Proiectare

Diagrama UML



Structuri de date

Pentru implementarea solutiei s-a folosit conceptul de thread-uri.

In Java putem crea threaduri fie extinzand (*mostenind, derivand din*) clasa Thread, fie implementand interfata Runnable.

Clasa Thread ca și interfața Runnable definesc o metodă numită *run()*. Această metodă este metoda de start pentru thread-ul nou, analog metodei *public static void main(String[])* pentru thread-ul principal. Această metodă *run()* poate fi apelată în două moduri:

- apelând direct metoda *run()*, în care caz execuția se face ca un apel obișnuit de metodă;
- apelând metoda *start()*, care pornește un thread nou care va începe execuția cu metoda *run()*.
-

Eu am ales pentru implementare varianta in care extindem interfata Runnable.

Conceptul de thread (fir de executie) este folosit in programare pentru a eficientiza executia programelor, executand portiuni distincte de cod in paralel, in interiorul aceluiasi proces. Cateodata insa, aceste portiuni de cod care constituie corpul threadurilor, nu sunt complet independente si in anumite momente ale executiei, se poate intampla ca un thread sa trebuiasca sa astepte executia unor instructiuni din alt thread, pentru a putea continua executia propriilor instructiuni. Aceasta tehnica prin care un thread asteapta executia altor threaduri inainte de a continua propria executie, se numeste *sincronizarea threadurilor*. Java ofera urmatoarele facilitati pentru sincronizarea threadurilor:

- mecanismul *synchronizes*
- si metodele: *wait*, *notify* si *notifyAll*.

Pentru a intelege problematica sincronizarii threadurilor, vom considera problema producatorilor si a consumatorilor. Aceasta spune ca avem mai multi producatori care "produc" in paralel obiecte si le depoziteaza intr-un container comun si avem mai multi consumatori care "consuma" in acelasi timp obiectele depozitate in container de catre producatori. Toti producatorii si consumatorii vor partaja acelasi container. Astfel ca producatorii si consumatorii vor produce, respectiv consuma.

4.Implementare

Clasa Client

- clasa Client contine trei atribute ID, tarrival, tservice;

- atributele reprezinta ID-ul unui client, un numar intre 1 si N, tarrival = timpul de sosire sit service = timpul de servire al clientului respective

- clasa contine gettere si settere + o metoda toString care ne va ajuta la scrierea datelor

- este clasa cu ajutorul careia vom stoca date relevante pentru fiecare client

Interfata Strategy

- contine o singura metoda numita addClient

- interfata este implementata de 2 clase: ConcreteStrategyQueue, ConcreteStrateyTime

Clasele ConcreteStrategyQueue, ConcreteStrateyTime

- clasa ConcreteStrategyQueue adauga clientul la coada cu cei mai putini client

```
public class ConcreteStrategyQueue implements Strategy{
  public void addClient(List<Queue> queue, Client client) {
        int minSize = Integer.MAX_VALUE;
           Queue minQueue = null;
           for (Queue q : queue) {
                if (q.getQueue().size() < minSize) {
                   minSize = q.getQueue().size();
                   minQueue = q;
                }
           }
           if (minQueue != null) {
             minQueue.addClient(client);
           }}
}
```

- clasa ConcreteStrategyTime adauga clientul la coada cu cel mai mic timp de asteptare

```
public class ConcreteStrategyTime implements Strategy {
        public void addClient(List<Queue> queue, Client client) {
                int minTime = Integer.MAX_VALUE;
                Queue minQueue = null;
                for (Queue q : queue) {
                        if (q.getWaitingTime().intValue() < minTime) {
                                minTime = q.getWaitingTime().intValue();
                                minQueue = q;
                        }
                }
                if (minQueue != null) {
                        minQueue.addClient(client);
                }
        }
}
```

Clasa ReadFromFile

- este clasa cu ajutorul careia vom citi date din fisier si vom scrie rezultatele in fisier;

- datele care se citesc din fisier sunt: numarul de Clienti, numarul de cozi care se vor crea in total, intervalul de simulare, timpul minim, respective maxim, pentru ca un client sa ajunga la magazine si timpul minim si maxim pana cand un client este servit;

- clasa contine gettere si settere pentru atributele mentionate mai sus;

- deasemenea avem implementat o functie cu ajutorul careia vom scrie date in fisier:

```
public static void write(ArrayList<String> args) {
	try {
		File outFile = new File(out3);
		outFile.createNewFile();
	} catch (IOException e) {
		e.printStackTrace();
	}
	try {
		FileWriter output = new FileWriter(out3);
		for (String s : args) {
			output.write(s);
		}
		output.close();
	} catch (IOException e) {
		e.printStackTrace();
	}
}
```

Clasa Scheduler

- contine un obiect de tipul Strategy si o lista a cozilor;

- clasa creeaza cate un thread pentru fiecare coada si stabileste care este “strategia” de lucru: coada mai goala sau timpul mai scurt;

- clasa mai contine si o metoda care adauga un client la o coada

- tot aici se afla si functia care calculeaza timpul mediu de asteptare:

```
public float timpulMediuDeAsteptare() {
	 int a = 0;
	 float t = 0;
	 for (Queue q : queues) {
	     t = t+ q.getAvgWaitingTime();
	    a = a+  q.getAvgClientNumber();
	 }
	 return t/a;
}
```

Clasa Queue

- contine o structura de tipul BlockingQueue<Client> si trei atribute de tipul Atomic Integer cu ajutorul carora vom stoca date despre timpul current de asteptare, timpul mediu de asteptare si numarul mediu al clientilor

- clasa contine gettere si settere pentru aceste atribute;

- clasa extinde interfata Runnable, implicit implementeaza functia run(), care v-a activa un thread doar daca exista client la coada; daca exista client la coada acestia vor fi eliminati pe rand, in caz contrar, coada va fi in asteptare;

Clasa Simulare

- contine atributele: simulationInterval, o lista de clienti si un obiect de tipul scheduler

- clasa implementeaza interfata Runnable, iar rolul acesteia este de a initializa threadul principal in care clientii generate random vor fi trimisi catre scheduler

```
public class Simulare implements Runnable {
	private int simulationInterval;
	private List<Client> clients;
	private Scheduler scheduler;
	private int N;

	public Simulare(int N, int Q, int minArrival, int maxArrival, int minService, int maxService,
			int simulationInterval, String strategyType) {
		// initialize queue and waitingPeriod
		this.setN(N);
		this.simulationInterval = simulationInterval;
		scheduler = new Scheduler( N, strategyType,Q);
		clients = new ArrayList<Client>();
		Random r = new Random();
		for (int i = 1; i <= N; i++) {
			Client client = new Client(r.nextInt(maxArrival - minArrival) + minArrival,
					r.nextInt(maxService - minService) + minService);
			clients.add(client);
		}
		Collections.sort(clients);
	}
```

Clasa MainClass

- este clasa in care vom porni threadul implementat de clasa Simulare

```
public static void main(String[] args) throws IOException {
	ReadFromFile read = new ReadFromFile(test3);
	String strategyType = "SHORTEST_TIME";  //sau SHORTEST_QUEUE
	Simulare sm = new Simulare(read.getN(), read.getQ(), read.getMinArrival(),
	read.getMaxArrival(), read.getMinService(), read.getMaxService(),
	read.getSimulationInterval(), strategyType);
	Thread start = new Thread(sm);
	start.start();
}
```

5 Rezultate

Pentru fisierul de intrare care contine datele:

N = 4; Q = 2; simulationInterval = 60;

tMinArrival = 2; tMaxArrival = 30; tMinService = 2; tMaxService = 30;

Rezultate:

```
Time: 0
Waiting Clients:(0, 7, 2)(0, 8, 3)(0, 13, 2)(0, 16, 2)
Queue 0: closed
Queue 1: closed

Time: 1
Waiting Clients:(0, 7, 2)(0, 8, 3)(0, 13, 2)(0, 16, 2)
Queue 0: closed
Queue 1: closed
....
Time: 7
Waiting Clients:(0, 8, 3)(0, 13, 2)(0, 16, 2)
Queue 0: (0, 7, 2)
Queue 1: closed

Time: 8
Waiting Clients:(0, 13, 2)(0, 16, 2)
Queue 0: (0, 7, 1)
Queue 1: (0, 8, 3)

Time: 9
Waiting Clients:(0, 13, 2)(0, 16, 2)
Queue 0: closed
Queue 1: (0, 8, 2)

Time: 10
Waiting Clients:(0, 13, 2)(0, 16, 2)
Queue 0: closed
Queue 1: (0, 8, 1)
....
Time: 13
Waiting Clients:(0, 16, 2)
Queue 0: (0, 13, 2)
Queue 1: closed

Time: 14
Waiting Clients:(0, 16, 2)
Queue 0: (0, 13, 1)
Queue 1: closed

Time: 15
Waiting Clients:(0, 16, 2)
Queue 0: closed
Queue 1: closed

Time: 16
Waiting Clients:
Queue 0: (0, 16, 2)
Queue 1: closed
```

7.Bibliografie

- http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_2/Java_Concurrency.pdf

-https://stackoverflow.com/

- https://www.javatpoint.com/

- https://www.w3schools.com/

- https://www.tutorialspoint.com/java/java_multithreading.html