MINISTRY OF EDUCATION AND RESEARCH
TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

TEHNICI DE PROGRAMARE

TEMA_2

SIMULARE DE COZI

Student: Tóth Szilveszter Zsolt

Gr.: 30224

1. Obiectivul temei

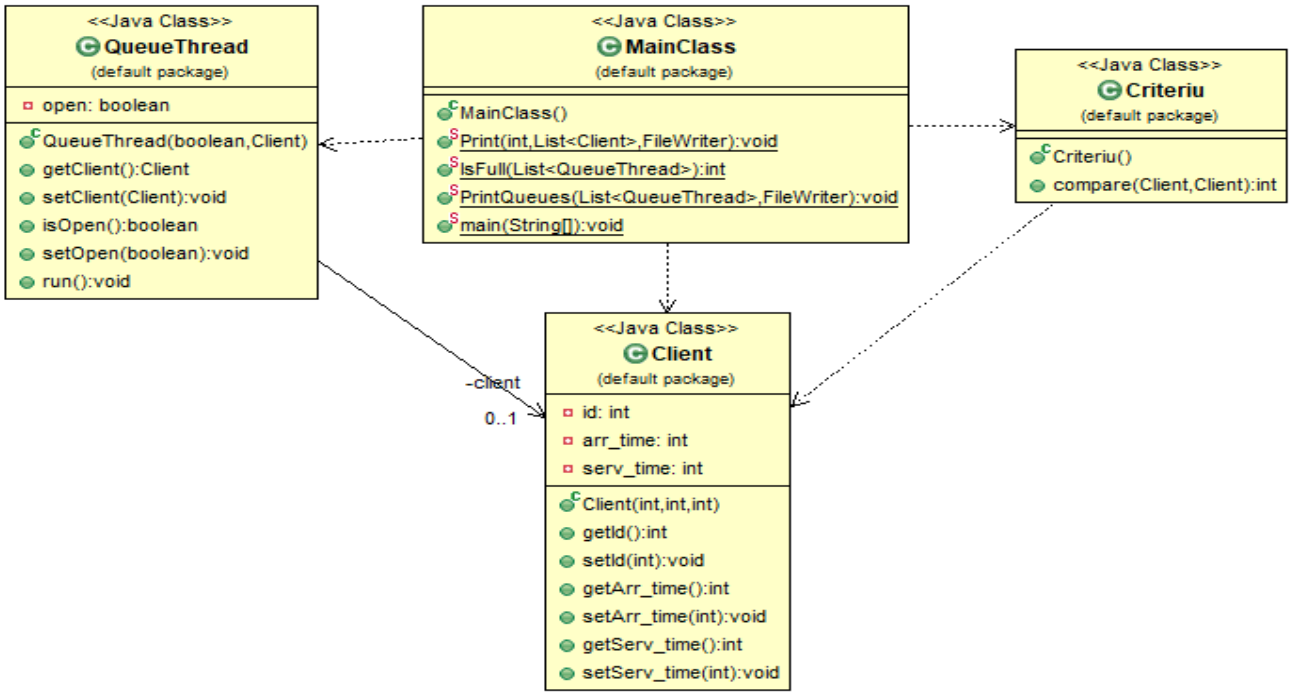
La aceasta tema am avut de implementat un simulator de cozi. De exemplu randurile de la un hipermarket, unde fiecare client intra in magazin (avand un numar unic), se plimba prin magazin iar apoi dupa ce a ales produsele dorite se ,,aseaza,, la un rand, iar mai apoi este servit. Timpul de servire depinde de cate produse are clientul.

2. Analiza problemei

La prima citire poate parea simpla problema, insa necesita mai multa gandire decat s-ar gandi omul. Trebuie sa avem in vedere trei parametri la clienti: ID-ul (care este unic pentru fiecare client care intra in magazin), Arrival Time (care reprezinta timpul la care si-a terminat cumparaturile si poate intra in rand), si Serving Time (care este timpul necesar pentru vanzator pentru a-i scana produsele). Cu aceste trei va trebui sa jonglam pentru a avea rezultatele dorite, de aceea este indicat sa facem o clasa ,,Client,, pentru a putea stoca aceste date pentru fiecare cumparator in parte. De asemenea trebuie sa ne definim cumva randurile (pentru ca o sa avem mai multe). De aceea este necesar sa ne definim o clasa ,,QueueThread,, in care vom trata si modifica randurile. Numarul lor poate varia, depinde de cat se cere din fisierul de intrare. Iar unificarea acestor clase se va realiza in ,,MainClass,, unde vom crea obiectele si vom scrie metodele necesare pentru realizarea acestei probleme.

3. Proiectare si implementare

Mai jos este diagrama UML a acestor clase pe care se pot vedea si relatiile dintre clase. Mai apare o clasa numita ,,Criteriu,, care, dupa cum ii zice si numele, contine doar un criteriu de sortare pe care o vom folosi pentru a aseza clientii in functie de Arrival Time-ul lor.



Am incercat sa abordez o metoda cat mai object oriented pentru a face cat mai usor de inteles si de rulat acest proiect. In ceea ce urmeaza am sa prezint pe scurt toate clasele folosite.

Prima clasa care este si cea mai simpla este clasa Criteriu. Cum am mentionat si mai devreme, acesta contine doar un return, care este totodata un criteriu pentru sort-ul din MainClass a clientilor. Dupa apelarea acestei metode, clientii vor fi aranjati in ordine crescatoare in functie de ArrivalTime-ul lor, pentru a fi procesati.

```
import java.util.Comparator;

public class Criteriu implements Comparator<Client> {

	public int compare(Client a, Client b) {

		return a.getArr_time() - b.getArr_time();

	}

}
```

Si este apelata in MainClass:

```
Collections.sort(clients, new Criteriu());
```

Urmatoarea clasa este a clientilor, daca tot am mentionat pana acum de ei. Acesta are trei variabile, cele trei atribute mentionate mai sus: id, arr_time si serv_time, care toate sunt intregi. Se creaza indivizii prin constructor, mai apoi sunt puse cateva getter-e si setter-e, pentru a putea afla sau eventual modifica datele clientilor pe parcursul programului.

```
public class Client {

	private int id;

	private int arr_time;

	private int serv_time;

	public Client(int id, int arr_time, int serv_time) {

		super();

		this.id = id;

		this.arr_time = arr_time;

		this.serv_time = serv_time;

	}
```

```
	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public int getArr_time() {
		return arr_time;
	}
	public void setArr_time(int arr_time) {
		this.arr_time = arr_time;
	}
	public int getServ_time() {
		return serv_time;
	}
	public void setServ_time(int serv_time) {
		this.serv_time = serv_time;
	}
}
```

Dupa asta, urmeaza sa vorbim despre clasa QueueThread, care dupa cum ii zice si numele, implementeaza cozile cu ajutorul threadurilor din Java. Pe scurt, threadurile in Java se pot percepe ca niste fire de lucru, care se pot rula in parallel cu programul principal, si se pot crea in numar foarte mare. Acestea dispun de cateva functii fara care nu s-ar putea lucra cu ele, de exemplu:

- sleep(x): care pune threadul intr-o stare de hibernare ,,x,, milisecunde , adica nu face nimic, dar exista ;
- run(): efectiv ruleaza firul de executie (trebuie sa avem insa grija, pentru ca in momentul in care se termina aceasta functie, threadul moare, se sterge din program si nu se mai poate apela sau refolosi);
- interrupt(): care intrerupe executarea threadului, adica practic intrerupe functie run(), pentru a face ca threadul sa existe in continuare;
- printre multe altele se enumara si functiile: start(), wait() etc.

Codul pentru aceasta clasa, care totodata extinde clasa Thread predefinita in java, se poate vedea mai jos:

```
public class QueueThread extends Thread {
	private boolean open;
	private Client client;
	public QueueThread(boolean open, Client client) {
		super();
		this.open = open;
		this.client = client;
	}
	public Client getClient() {
		return client;
	}
	public void setClient(Client client) {
		this.client = client;
	}
	public boolean isOpen() {
		return open;
	}
	public void setOpen(boolean open) {
		this.open = open;
	}
	public void run() {
		if (client.getServ_time() != 0) {
			int p = client.getServ_time();
			p--;
			client.setServ_time(p);
```

```
                this.interrupt();
            } else {
                client = null;
                open = false;
            }
        }
}
```

Acest obiect are doi parametrii: un obiect de tip Client in care vor fi ,,stocate,, informatiile clientului actual care se afla in coada respective, si o variabila de tip Boolean, care spune defapt programului daca coada este deschisa sau nu (altfel spus daca se afla client in ea la momentul actual sau nu). Ca si clasa mentionata mai devreme, si clasa QueueThread are getter-e si setter-e, si cel mai important lucru are un constructor care practice creeaza coada in care vor fi asezati clientii magazinului.

Am ajuns la clasa care uneste toate aceste clase, care totodata este cea mai mare si care contine cele mai multe instructiuni. In primul rand, in MainClass sunt 4 metode: main, 2 metode de print si o metoda care verifica daca coada respectiva este plina sau nu. Cele doua metoda de afisare ne ajuta la scrierea informatiile despre timpul simularii si despre cozi in fisierele output, pentru a putea verifica corectitudinea programului. Codul pentru acestea va fi pusa in codul per ansamblu a clasei MainClass. Metoda isFull este o simpla parcurgere a cozilor, in care se verifica mereu daca coada actuala este inchisa sau sunt clienti in aceasta.

Codul pentru MainClass:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Random;
import java.util.Scanner;


public class MainClass {
    public static void Print(int time, List<Client> clients, FileWriter toFile) {
        try {
            toFile.write("Time=" + time + "\n");
```

```
		} catch (IOException e1) {
			e1.printStackTrace();
		}

		//if (clients.isEmpty()==false)
		for (Client x : clients) {
			if (time != x.getArr_time())
				try {
			toFile.write("(" + x.getId() + "," + x.getArr_time() + "," + x.getServ_time() + ")\n");
				} catch (IOException e) {
					e.printStackTrace();
				}
		}

	}

	public static int IsFull(List<QueueThread> threads) {
		for (QueueThread x : threads) {
			if (x.isOpen()==false)
				return 0;
		}
		return 1;
	}
	public static void PrintQueues(List<QueueThread> threads, FileWriter toFile) {
		try {
			toFile.write("Queues:\n");
		} catch (IOException e1) {
			e1.printStackTrace();
		}
```

```
        for (QueueThread t : threads) {
                if (t.getClient() == null || t.getClient().getServ_time() == 0)
                        try {
                                toFile.write("closed\n");
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                else
                        try {
        toFile.write("(" + t.getClient().getId() + "," + t.getClient().getArr_time() + ","
                                                        + t.getClient().getServ_time() + ")\n");
                        } catch (IOException e) {

                                e.printStackTrace();
                        }
        }
        try {
                toFile.write("\n");
        } catch (IOException e) {
                e.printStackTrace();
        }
}

public static void main(String[] args) throws InterruptedException, IOException {
        List<Client> clients = new ArrayList<Client>();
        int clientNumber = 0;
        int queueNumber = 0;
        int timeMax = 0;
        String arrTime = null;
```

```
String servTime = null;

try {
        File myObj = new File(args[0]);
        Scanner myReader = new Scanner(myObj);
        while (myReader.hasNextLine()) {

                String data = myReader.nextLine();
                clientNumber = Integer.parseInt(data);
                data = myReader.nextLine();
                queueNumber = Integer.parseInt(data);
                data = myReader.nextLine();
                timeMax = Integer.parseInt(data);
                arrTime = myReader.nextLine();
                servTime = myReader.nextLine();

        }
        myReader.close();
} catch (FileNotFoundException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
}
Random rn = new Random();

String[] arr = arrTime.split(",", -2);
int lowa = Integer.parseInt(arr[0]), higha = Integer.parseInt(arr[1]);

String[] serv = servTime.split(",", -2);
int lows = Integer.parseInt(serv[0]), highs = Integer.parseInt(serv[1]);
```

```
for (int i = 1; i <= clientNumber; i++) {
        int a = rn.nextInt(higha - lowa) + lowa;
        int s = rn.nextInt(highs - lows) + lows;
        Client c = new Client(i, a, s);
        clients.add(c);
}
Collections.sort(clients, new Criteriu());
List<QueueThread> threads = new ArrayList<QueueThread>(queueNumber);
for (int i = 0; i < queueNumber; i++) {
        threads.add(new QueueThread(false, null));
}
int time = 0;
float averageTime=0;
Client x1 = null;
Client x = null;
int ok=1;
int endingTime=clients.get(clientNumber-1).getArr_time()+highs;

FileWriter toFile=new FileWriter(args[1]);

while (time != timeMax && time!=endingTime) {
        if (ok==1)
                {
                        ok=0;
                        Print(time, clients,toFile);
                }
        if (x1 == null) {
```

```
            if (clients.isEmpty() == false)
                        x = clients.get(0);
} else {
            x = x1;
}
if (time >= x.getArr_time()) {
            if (IsFull(threads)==1) {
                        time++;
                        //Print(time, clients);
                        PrintQueues(threads,toFile);
                        for (QueueThread t : threads)
                                    if (t.isOpen() == true) {
                                                t.run();
                                                if (!(t.getClient() == null))
                                                averageTime++;
                                    }
            }
            else {
            for (QueueThread t : threads) {
                        if (t.isOpen() == false) {
                                    t.setClient(x);
                                    if (clients.isEmpty() == false)
                                                clients.remove(0);
                                    t.setOpen(true);
                                    break;
                        }
            }
            }
}
```

```
                if (clients.isEmpty() == false)
                        x1 = clients.get(0);

                if (clients.isEmpty() == true) {
                        time++;
                } else if (time < x1.getArr_time()) {
                        time++;
                }
                else
                                continue;

                PrintQueues(threads,toFile);
                ok=1;
                for (QueueThread t : threads)
                        if (t.isOpen() == true) {
                                t.run();
                                if (!(t.getClient() == null))
                                averageTime++;
                        }

        }
        toFile.write("Si tot asa va fi, pana cand time ajunge la: " + timeMax + "\n");
        toFile.write("Average waiting time: " + averageTime/clientNumber);
        toFile.close();
    }

}
```

Pentru a putea intelege ce se intampla in main, o voi prezenta amanuntit, luand pas cu pas fiecare instructiune. In prima este definit doar o lista cu clienti, dupa care sunt citite datele din fisierul de intrare in care sunt

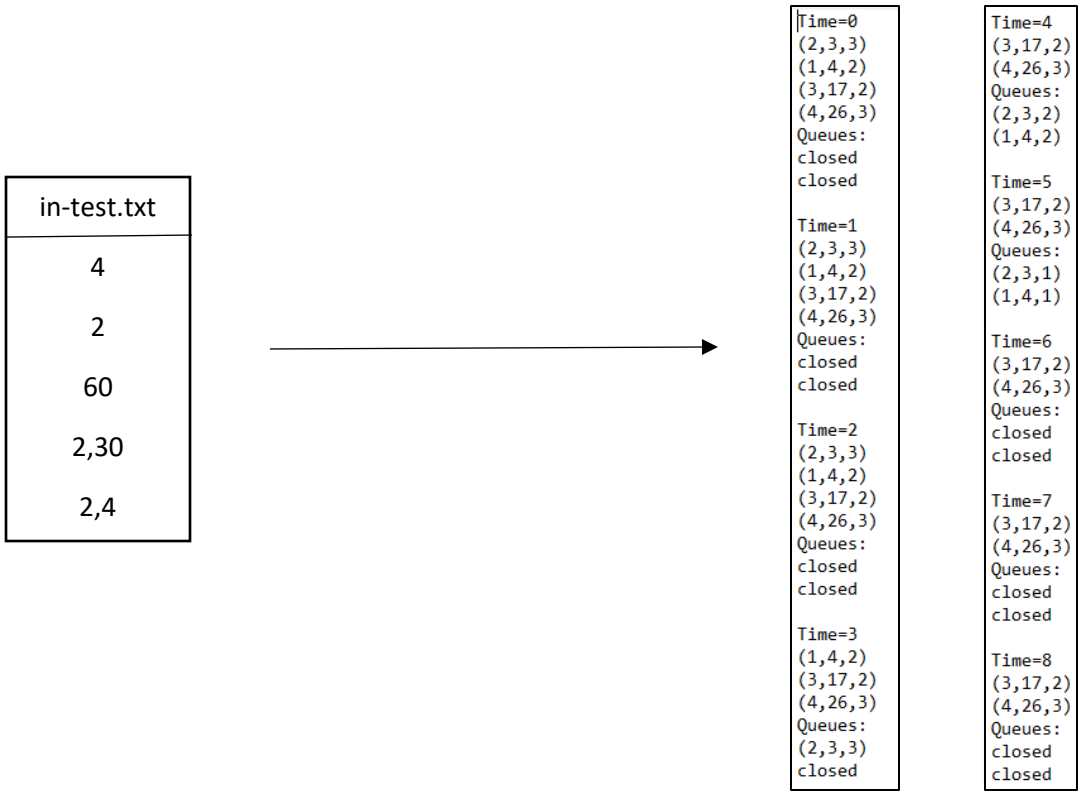
precizate urmatoarele: numarul de clienti, numarul de cozi, timpul simularii (adica dupa cat timp se va sfarsi simularea), iar la final 2 valori minime si maxime, intre care vor fi timpii de sosire si timpii de servire a clientilor, care urmeaza sa fie generate random. Urmatorul pas este, cum am si mentionat, creare clientilor. Acest lucru se face cu un for simplu, unde se creeaza timpii mentionati pentru clienti, folosind obiectul predefinit Random (nota: intervalele de timp sunt citite ca niste stringuri, deci necesita sa le spargem pentru a afla concret numerele care vor fi transformate in numere intregi). Apoi se face sortarea, in care aranjam clientii in functie de timpul de sosire. Dupa asta este definit timpul de simulare, care este cumva timerul proiectului. Simularea tine atat timp, cat timpul ajunge la valoare maxima citita din fisier.

Dupa ce s-a terminat partea de pregatire, urmeaza o bucla while care este ,,sufletul,, proiectului,,. In aceasta bucla vor fi cititi clientii din lista lor si inserati in coada, care este posibil doar daca coada actuala este inchisa. Se verifica si cazul in care toate cozile sunt pline, in cazul care nu se face inserare ci se prelucreaza datele clientilor. Dupa asta, se mai citeste un client, pentru a verifica daca timpul lui de sosire este aceeasi cu a clientului din fata, si daca da, se va face inserarea intr-o coada diferita. Daca nu indeplinesc conditiile, pur si simplu se mareste timpul de simulare si se prelucreaza datele clientilor aflati in cozi. Pe parcurs este calculat timpul de asteptare a fiecarui client, iar la final daca impartim aceasta valoare la numarul clientilor, vom afla timpul mediu de asteptare, adica in medie, cat a stat un client in coada pana ii s-au scanat produsele.

Toate datele vor fi scrise in fisierul output, atat timpul mediu de asteptare cat si starea cozilor pe parcursul simularii.

4. Rezultate

Cum am mai mentionat, se da un fisier cu metadate necesare si apoi se ruleaza simularea cozilor, care vor fi scrise intr-un fisier output, care va arata aproximativ in urmatorul fel:



Iar la final se va afisa timpul de asteptare mediu. In multe cazuri simularea nu va dura pana la timpul maxim alocat, asa ca in loc sa se afiseze de multe ori acelasi lucru, am pus o conditie care nu lasa programul sa afiseze dupa terminarea procesului propriu-zis:

```
Si tot asa va fi, pana cand time ajunge la: 60
Average waiting time: 2.5
```

5. Concluzii

La aceasta tema cea mai importanta lectie pe care am invatat-o este ca nu ne apucam de scris cod inainte de a ne gandi si a face un schelet pentru program. Lectie pe care n-o sa uit prea curand. M-a ajutat foarte mult experienta de la tema anterioara, mai ales la partea de prelucrat stringuri, dar si la crearea unui program cat mai object oriented. Daca e cazul sa vorbim despre posibile dezvoltari, ar fi o problema de rezolvat ce tine strict de afisaj (sunt cazuri in care se afiseaza altfel decat in situatia ideala).

6. Bibliografie

https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

https://www.w3schools.com/java/java_files_create.asp