Documentatie Tema 2
Queues Simulator

George Adam

Grupa 30225

Profesor Laborator: Dorin Moldovan

1. Obiectivul temei

Scopul principal al aceste aplicatii este de a simula comportamentul unor cozi de clienti. Poate fi utila si folosita in optimizarea sau automatizarea caselor de marcat din supermarket-uri astfel incat acestea sa fie deschise si inchise dinamic in functie de numarul de clienti care se asteapta, astfel fiind optimizat timpul de lucru al fiecarui angajat si reorientat. Simularea se face dinamic, in functie de nevoi (numarul de clienti), si concurent, simuland exact comportamentul din viata reala.

Obiectiv secundar	Descriere	Capitol
Viteza	Durata impartirii si servirii clientilor trebuie sa fie cat mai rapida, astfel se alege o solutie concurenta.	
Verificare pe pasi	Pentru a fi informat despre fiecare alegere facuta, la toate momentele de timp ale simularii, fiecare dintre acestea fiind detaliat intr-un fisier de iesire.	
Corectitudinea	Prin fisierul de iesire ce contine fiecare pas detaliat, se poate verifica corectitudinea. Se folosesc structuri de date care faciliteaza functionarea concurenta a aplicatiei.	

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

- Cerinte functionale.

  - Implementarea cozilor de clienti.
  - Generarea dinamica a cozilor de clienti.
  - Impartirea clientilor pe cozi.
  - Rezolvarea operatiilor concurent pentru a creste viteza de executie.
  - Folosirea structurilor de date care sa nu afecteze abordarea concurenta.
  - Alegerea unei strategii care prioritizeaza viteza servirii clientilor.
  - Salvarea fiecarui pas intr-un fisier de iesire.

- Use-cases

Use-case	Solutie
Crearea cozilor	Cozile sunt create dinamic dupa un numar introdus de utilizator, pentru a nu se irosi resursele.
Impartirea clientilor	Clientii sunt impartiti la cozi tinandu-se cont de anumite variabile proprii, totodata se urmareste minimizarea timpului de asteptare.
Strategia de timp	Se alege o strategie de timp care favorizeaza impartirea clientilor catre cozi astfel incat sa astepte cat mai putin pentru a fi serviti.
Alegerea structurilor de date	Se folosesc structuri de date predefinite pentru a facilita implementarea concurenta. Totodata, cozile si clientii se definesc in propriile lor clase/structuri, pentru a putea fi refolosit codul.
Afisarea in fisier	Pentru a verifica corectitudinea si modul de functionare, fiecare actiune facuta de aplicatie la un moment de timp este scrisa intr-un fisier de iesire, la finalul simularii, cu toate datele importante.
Dezvoltarea algoritmilor	Se folosesc algoritmi pentru sincronizarea Thread-urilor.
Impartirea pe clase	Se creeaza clase pentru a imparti problema in probleme mai mici. Se foloseste un Scheduled-Task pattern pentru lucrul cu Thread-uri si fisiere.
Implementarea solutiei	Se genereaza in faza initiala o diagrama UML pentru a avea structura proiectului, apoi se incepe implementarea propriu-zisa.
Testare	Sunt rulate mai multe fisiere .txt cu date de test, apoi se verifica output-ul scris in alt fisier test.

3. Proiectare

- Decizii de proiectare

Proiectul este impartit in urmatoarele clase: Person si RandomClientGenerator, FileControl si FileData, Queue, Scheduler si Simulation.

Totodata aceasta impartire separa codul in 3 mari categorii: operatii cu fisiere, I/O, entitati pentru client si generarea aleatoare a acestora si simularea si operatiile efectuate cu ajutorul Thread-urilor.

Se adopta un design pattern Scheduled-Task, pentru a implementa functionalitatea concurenta a programului si pentru a face lizibila si a avea control peste ce se intampla in coze (Thread-uri).

Clasa FileControl initializeaza variabile de tip FileWriter, FileReader, BufferedWriter, BufferedReader pentru un fisier de intrare si unul de iesire, date ca argument in linia de comanda la rularea programului. Este implementata o metoda care citeste anumite date din fisier, sub un anumit format. Cealalta metoda implementata este de a scrie un String primit ca argument in fisier pe o linie.

Clasele Person si RandomClientGenerator implementeaza structura de Persoana si genereaza pe baza datelor citite intr-o instanta a clasei File Data anumiti clienti cu atribute aleatoare, dar incadrate in intervale bine stabilite.

Clasa Queue implementeaza toate operatiile specifice cozi de clienti, o casa de marcat spre exemplu, si implementeaza interfata Runnable pentru a putea fi rulata pe un Thread stabilit.

Clasa Scheduler declara un anumit numar de cozi si Thread-uri pentru acestea, astfel incat acestea sa poate fi rulate. Totodata, exista metoda placeInQueue care cauta in ce coada sa fie introdus un client tinand cont de timpul de asteptare, incercand sa il minimizeze.

Clasa Simulation foloseste un Thread care numara din secunda in secunda, timp real, timpul simularii, si apeleaza functia placeInQueue dintr-o instanta a clasei Scheduler cu fiecare client care este pregatit de a fi trimis in coada.

- Structuri de date

Structurile de date folosite sunt:

- BlockingQueue: fiecare element de tip Person din clasa Queue este tinut in aceasta structura, deoarece mai multe thread-uri au acces la ea si este thread-safe.
- ArrayList: o lista de Person, Thread si Queue
- String: folosit la scrierea in fisier a informatiilor din fiecare pas de simulare.
- AtomicInteger: folosit la contorizarea timpului de asteptare pentru fiecare coada. S-a ales aceasta structura din aceleasi considerente ca si BlockingQueue.

- Proiectare clase

Clasele sunt:

- Person: structura de persoana ce salveaza datele despre aceasta.
- FileControl: implementeaza operatii de I/O cu fisiere.
- FileData: salveaza datele continute in fisierul de intrare.
- RandomClientGenerator: genereaza clienti aleator in functie de datele dintr-o instanta a clasei FileData.
- Queue: implementeaza functionalitatile unei cozi de clienti.
- Scheduler: implementeaza functii pentru controlul cozilor de clienti.
- Simulation: implementeaza un counter real time si manageriaza clientii. Contine si metoda “main”.

- Algoritmi
  - Adaugare in coada:

    Se parcurg toate cozile, se gaseste cea cu timpul cel mai mic de asteptare si se introduce respectivul client in ea.

  - Counter:

    Se numara cate o secunda, pe ceas, de la 0 pana la timpul de simulare citit in fisier.

  - Pornirea Thread-urilor:

    Se genereaza n thread-uri si queue-uri, n fiind variabil, iar apoi fiecarui thread I se atribuie cate un queue.

  - Verificarea clientilor:

La fiecare moment de timp se verifica daca sunt clienti care sunt eligibili pentru a fi trimisi la cozi, si sunt trimisi.

- Concurenta:

  Din moment ce fiecare thread are asignat cate un queue, atunci fiecare queue se executa concurent.

- Scrierea in fisier:

  La fiecare pas al simularii, se scriu in fisier datele de la momentul de timp respectiv preluate din fiecare coada si din clasa Simulation.

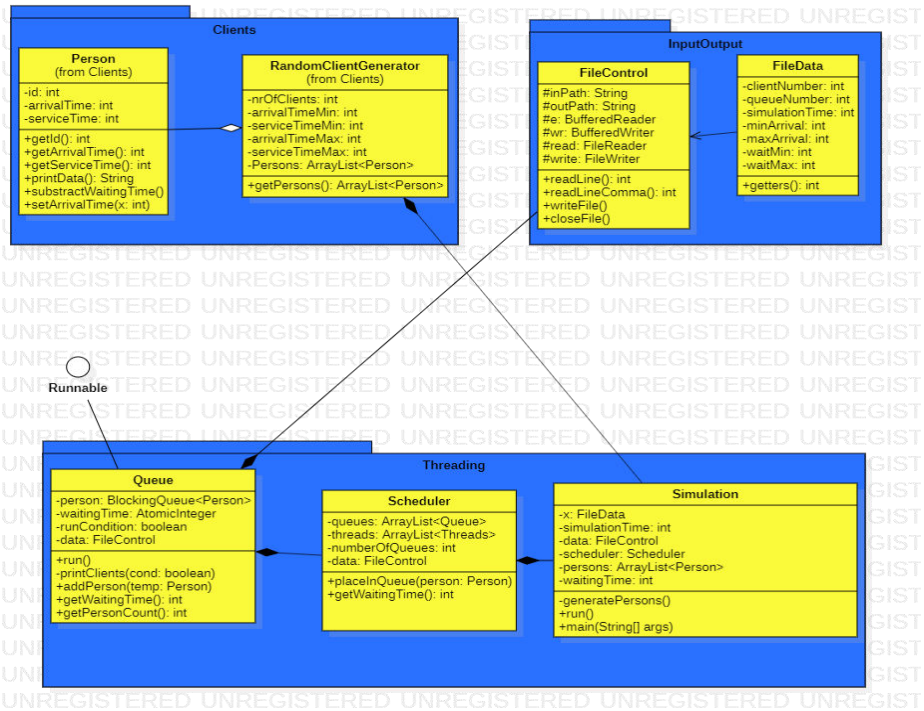
- Fisierul de iesire

  Fisierul de iesire va fi creat cu numele introdus ca al doilea parametru la rularea programului.

  Acesta are o structura de forma:

  - Time:
  - Waiting Clients:
  - Queue X: where x is the number of the queue and afterwards, all clients in that queue are displayed.

- Diagrama UML



Clasa RandomClientGenerator are o relatia de agregare fata de clasa Person.

Clasele FileControl si Scheduler au relatii de compozitie cu clasa Queue.

Clasa Simulation are relatii de compozitie cu clasele Scheduler si Simulation.

Clasa FileData mosteneste FileControl.

- Packages

  Pachetele prezente sunt Clients, InputOutput si Threading.

  In Clients clasele sunt Person si RandomClientGenerator care stocheaza informatii despre clienti si ii genereaza aleator.

  In InputOutput clasa FileData salveaza datele citite din fisier, iar FileControl ofera functii pentru scriere si citire din fisier.

  Threading contine clasele Queue, Scheduler si Simulation, care implementeaza functionalitati de coada, imparte task-urile/clientii pe cozi si tine un counter global.

  S-a ales aceasta impartire pe clase pentru a tine separata lucrul cu fisiere si clientii de patter Scheduled-task folosit pentru a implementa functionarea concurenta a cozilor. Acest pattern a fost aleas deoarece usureaza munca cu Thread-uri, facand astfel usor de inteles si modificat codul specific acestei operatiuni.

4. Implementare

- Person

```java
private int id;
private int arrivalTime;
private int serviceTime;

public Person(int id, int arrivalTime, int serviceTime) {
    this.id = id;
    this.arrivalTime = arrivalTime;
    this.serviceTime = serviceTime;
}
```

Cel mai important motiv pentru care clasa Person a fost creata este nevoia de icnapsulare a mai multor detalii despre un client. Astfel, acesta este caracterizat de un ID (unic), un timp de sosire, mai exact, dupa ce simularea ajunge la acel timp, el poate fi introdus intr-un Queue. Service time-ul este timpul pe care acesta trebuie sa il petreaca in coada daca este primul.

De fiecare data, dupa cum se vede si in constructor, cand este creata o persoana noua, aceste trei detalii trebuie transmise.

- RandomClientGenerator

```java
public RandomClientGenerator(int nrOfClients, int arrivalTimeMin, int serviceTimeMin, int arrivalTimeMax, int serviceTimeMax) {
    this.arrivalTimeMin = arrivalTimeMin;
    this.arrivalTimeMax = arrivalTimeMax;
    this.nrOfClients = nrOfClients;
    this.serviceTimeMin = serviceTimeMin;
    this.serviceTimeMax = serviceTimeMax;

    Persons = new ArrayList<Person>();

    for (int i = 0; i < this.nrOfClients; i++) {
        Person temp = new Person(
                id: i + 1,
                arrivalTime: new Random().nextInt( bound: (this.arrivalTimeMax - this.arrivalTimeMin) + 1) + this.arrivalTimeMin,
                serviceTime: new Random().nextInt( bound: (this.serviceTimeMax - this.serviceTimeMin) + 1) + this.serviceTimeMin
        );

        Persons.add(temp);


    }
```

La instantierea unui obiect de tip RandomClientGenerator, se transmit parametrii in care sa se incadreze atributele unei persoane, apois e calculeaza cate o valoare aleatoare pentru fiecare.

- FileControl

```
public int[] readLineComma() {
    int[] val = new int[2];
    String[] vals = null;

    try {
        vals = e.readLine().split( regex: ",");
    } catch(IOException e) {
        e.printStackTrace();
        System.exit( status: -1);
    }

    int len = 0;
    for(String x : vals) {
        val[len++] = Integer.parseInt(x);
    }

    return val;
}
```

Atributul e este de tip BufferedReader. Aceasta functie citeste din fisier cate o linie, apoi, avand in vedere ca unele valori necesita un minim si un maxim, aceste vor fi scris pe o linie a fisierului, separate de o virgula. Functia desparte stringul in functie de cate virgule sunt, apoi converteste valoarea la un int si o salveaza intr-un vector pe care apoi il returneaza.

- FileData

```
public FileData(String inPath, String outPath) {
    super(inPath, outPath);

    clientNumber = readLine();
    queueNumber = readLine();
    simulationTime = readLine();

    int[] val = readLineComma();
    minArrival = val[0];
    maxArrival = val[1];

    val = readLineComma();
    waitMin = val[0];
    waitMax = val[1];

}
```

Clasa FileData, odata instantiata, se foloseste de readLineComma() din clasa FileControl, pe care o si mosteneste, pentru a salva toate datele din fisier folosite pentru generarea clientilor.

- Queue

```java
public void addPerson(Person temp) {
    this.person.add(temp);
    waitingTime.addAndGet(temp.getServiceTime());
    try {
        Thread.sleep( millis: 5);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Functia adauga parametrul de tip Person la coada curenta si incrementeaza timpul de asteptare al cozii cu service time-ul persoanei.

```java
@Override
public void run() {
    while (runCondition) {
        try {
            Person current = person.take();
            int i = 0;
            while (current.getServiceTime() > 0) {
                Thread.sleep( millis: 50);
                String toWrite = "Queue " + Thread.currentThread().getName() + ": "
                        + "(" + current.getId()
                        + ", " + current.getArrivalTime()
                        + ", " + current.getServiceTime()
                        + ")";
                data.writeFile(toWrite);
                if (person.isEmpty()) {
                    data.writeFile( toWrite: "\n");
                } else {
                    printClients( cond: false);
                }
                Thread.sleep( millis: 1000);
                person.remove(current);
                current.substractWaitingTime();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Functia ruleaza continuu de cand thread-ul care a primit obiectul Queue este pornit. Aceasta preia prima persoana din lista, iar daca nu exista, thread-ul se va pune in asteptare pana cand apare alta. Scrie in fisier informatiile despre persoana extrasa, toate astea intr-un while pentru a scadea timpul sau de asteptare odata cu counter-ul global.

- Scheduler

```
public Scheduler(int numberOfQueues, FileControl data) {
    this.numberOfQueues = numberOfQueues;
    queues = new ArrayList<Queue>();
    threads = new ArrayList<Thread>();
    this.data = data;

    for(int i = 0; i < numberOfQueues; i++) {
        Queue q = new Queue( runCondition: true, data);
        queues.add(q);
        threads.add(new Thread(q));
        threads.get(i).setName(String.valueOf(i + 1));
    }
    for(Thread t : threads) {
        t.start();
    }
}
```

Constructorul clasei Scheduler initializeaza un numar primit ca parametru de thread-uri si queue-uri. Fiecarul thread ii este setat numele pentru a fi mai usor de urmarit, apoi acesta este deja pornit, doar ca nu va face nimic pana nu se vor adauga elemente in cozi.

```
public void placeInQueue(Person person) {
    int minTime = Integer.MAX_VALUE;
    for(Queue q : queues) {
        if(q.getWaitingTime() < minTime) {
            minTime = q.getWaitingTime();
        }
    }
    for(Queue q : queues) {
        if(q.getWaitingTime() == minTime) {
            q.addPerson(person);
            break;
        }
    }
}
```

Functia placeInQueue primeste ca si argument un obiect de tip Person. Aceasta cauta obiectul de tip Queue cu cel mai mic timp de asteptare. Dupa ce l-a gasit, atunci obiectul primit ca si parametru se adauga in acel Queue. Pentru a nu fi adaugat in mai multe Queue-uri, atunci se adauga in prima gasita si se opreste functia cu ajutorul instructiunii “break”.

- Simulation

```java
@Override
public void run() {
    int currentTime = 0;
    int waitingTime = 0;
    while(currentTime <= simulationTime) {
        data.writeFile( toWrite: "Time: " + currentTime + "\n");
        Iterator<Person> iterator = persons.iterator();

        while(iterator.hasNext()) {
            Person temp = iterator.next();
            if(temp.getArrivalTime() == currentTime) {
                scheduler.placeInQueue(temp);
                waitingTime += temp.getServiceTime();
                iterator.remove();
            }
        }

        waitingTime += scheduler.getWaitingTime();

        data.writeFile( toWrite: "Waiting clients: ");
        for(Person person : persons) {
            data.writeFile(person.printData());
        }
        data.writeFile( toWrite: "\n");
        currentTime++;

        try {
            Thread.sleep( millis: 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    data.writeFile( toWrite: "Average waiting time: " + String.valueOf((waitingTime - 1.0) / x.getClientNumber()));
    data.closeFile();
    System.exit( status: 1);
}
```

Functia run va rula pana cand timpul curent atinge timpul citit din fisier. Dupa fiecare executie a instructiunilor din bucla while, se asteapta o secunda pentru ca programul sa ruleze real time.
Se trece prin toata lista de clienti care exista, iar cei care arrival time-ul egal cu timpul curent vor fi repartizati la cozi cu ajutorul functiei placeInQueue() din clasa Scheduler. Am ales ca fiecare client care corespunde cerintelor sa fie repartiat intr-un moment de timp, nu doar unul per moment, deoarece usureaza impartirea lor ulterioara.

Totodata se scrie in fisier timpul curent alaturi de celelalte informatii care sunt scrise din functii din clasa Queue pentru a se observa evolutia programului pe o durata de timp si modul de impartire.

Pentru a se calcula timpul mediu de asteptare, care este format din timpul asteptat de client pentru servire adunat cu timpul asteptat in coada pana ii soseste randul, am folosit urmatoarea metoda: La fiecare pas se aduna numarul de clienti prezenti in toate cozile si la final se imparte la numarul de clienti. Este mult mai usor de inteles, pentru ca dupa ce un client va disparea din coada, in acest fel se aduna cat a stat pentru servire si cat a stat pentru a isi primi randul.

5. Rezultate

Toate rezultatele sunt salvate in fisiere care insotesc .jar-ul creat, dar si fisierele folosite pentru a crea aceste rezultate.

Asa ar trebui sa arate un fisier cu rezultate:

```
Time: 0
Waiting clients: (1, 1, 6) (2, 2, 3) (20, 2, 6) (34, 2, 4)
Time: 1
Waiting clients: (2, 2, 3) (20, 2, 6) (34, 2, 4) (37, 2, 3)
Queue 1: (1, 1, 6)
Time: 2
Waiting clients: (3, 3, 6) (47, 3, 1) (14, 5, 1) (21, 5, 4)
Queue 2: (2, 2, 3)
Queue 3: (20, 2, 6)
Queue 4: (34, 2, 4)
Queue 5: (37, 2, 3)
Queue 1: (1, 1, 5)
Time: 3
Waiting clients: (14, 5, 1) (21, 5, 4) (22, 5, 6) (50, 5, 4)
Queue 2: (2, 2, 2) (3, 3, 6)
Queue 3: (20, 2, 5)
Queue 4: (34, 2, 3)
Queue 5: (37, 2, 2) (47, 3, 1)
Queue 1: (1, 1, 4)
Time: 4
Waiting clients: (14, 5, 1) (21, 5, 4) (22, 5, 6) (50, 5, 4)
Queue 2: (2, 2, 1) (3, 3, 6)
Queue 3: (20, 2, 4)
Queue 4: (34, 2, 2)
Queue 5: (37, 2, 1) (47, 3, 1)
Queue 1: (1, 1, 3)
```

Bineinteles, aceasta este doar o parte din fisier, dar este prea mare sa incapa intr-un document.

6. Concluzii

Ideile cea mai importanta care a rezultat din realizarea acestei teme este, cu siguranta, importanta folosirii Thread-urilor. De asemenea, asigurarea conceptului de Thread-safety este imperativa, deoarece pot aparea multe erori din cauza nesincronizarii.

Este prezenta si nevoia de realizare a unui cod “curat”. Necesitatea de a realiza un cod atat de inteligibil incat nu are nevoie de a fi comentat este imperativa, deoarece reintoarcerea la el si/ sau modificarea acestuia devine mult mai usoara. Totodata, impartirea codului in clase si pachete specifice unei anumite probleme permite reutilizarea acestuia si evita scrierea unor bucati largi de cod care vor fi folosite doar o singura data.

Elementul care necesita dezvoltare ulterioara este afisarea rezultatelor dupa executie. Acesta poate fi rezolvat ori prin gasirea unei structuri mai inteligibile de a le scrie in fisier, ori prin implementarea unei interfete grafice pentru afisarea lor real-time.

7. Bibliografie

Threads: https://beginnersbook.com/2013/03/multithreading-in-java/

Data Structures: https://medium.com/elp-2018/thread-safe-collections-8f1f17c283e7

File I/O: https://stackabuse.com/reading-and-writing-files-in-java/

Scheduling: https://www.tutorialspoint.com/java/util/timer_schedule_period.htm