# TEHNICI DE PROGRAMARE

## DOCUMENTATIA TEMEI 2

## SIMULATOR DE COZI

**Hulea Andrei-Florin**
**Grupa 30225**

**Cuprins:**

**1.Obiectivul temei**

-Se va prezenta obiectivul principal al temei printr-o fraza si un tabel sau o lista cu obiectivele secundare. Obiectivele secundare reprezinta pasii care trebuie urmati pentru indeplinirea obiectivului principal. Fiecare obiectiv secundar va fi descris si se va indica in care capitol al documentatiei va fi detaliat.

**2. Analiza problemei**

-Modelare, scenarii, cazuri de utilizare Se va prezenta cadrul de cerinte functionale formalizat si cazurile de utilizare ca si diagrame si descrieri de use-case. Descrierile use-case-urilor se vor face sub forma unui flow-chart ori sub forma unei liste continand pasii executiei fiecarui use-case.

**3. Proiectare**

-Se va prezenta proiectarea OOP a aplicatiei, diagramele UML de clase si de pachete, structurile de date folosite, interfetele definite si algoritmii folositi (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator)

**4. Implementare**

-Se va descrie fiecare clasa cu campurile si cu metodele importante. Se va descrie implemantarea interfetei utilizator.

**5. Rezultate**

-Se vor prezenta scenariile pentru testare cu Junit sau alt framework de testare.

**6. Concluzii**

-Se vor prezenta concluziile, ce s-a invatat din tema, dezvoltari ulterioare.

**7. Bibliografie**

-Se vor mentiona resursele bibliografice care au fost folosite pentru dezvoltarea temei

## 1. Obiectivul temei

Obiectivul acestei teme este implementarea unei aplicatii care simuleaza mai multe cozi, scopul acesteia fiind minimizarea timpului de asteptare pentru clienti si determinarea unui timp mediu de asteptare

Cozile sunt adeseori folosite pentru a reprezenta modele intalnite in viata reala. Scopul unei cozi este de a furniza un loc de asteptare pentru un "client" pana ca acesta sa poata fi "servit". Interesul gestiunii unui astfel de sistem este minimizarea timpului de asteptare al clientiilor inainte de a fi serviti. O metoda de minimizare a timpului de asteptare este adaugarea unei noi persoane care serveste, adica crearea unei noi cozi, fiecare avand propriul procesor in sistem. Cand o noua coada este adaugata, clientii vor fi distribuiti in mod egal la toate cozile care sunt disponibile.

Aplicatia ar trebui sa simuleze (prin definirea unui timp de simulare t_simulation) un numar de N clienti care asteapta sa fie serviti , intrand in Q cozi, asteptand pana sa fie serviti dupa care sa paraseasca coada. Toti clientii sunt generati la inceperea simularii , si sunt caracterizati de urmatorii 3 parametrii : ID(un numar intre 1 si N), t_arrival(timpul de simulare cand clientul este pregatit sa intre in coada sau cand clientul si-a terminat cumparaturile) si t_service ( timpul de simulare sau durata necesara pana clientul este servit de catre casier sau timpul cat clientul sta in fata cozii). Aplicatia contorizeaza timpul total petrecut de fiecare client la coada si calculeaza un timp mediu de asteptare la coada. Fiecare client este adaugat la coada cu un minim de asteptare cand timpul de ajungere al acestuia (t_arrival) este mai mare sau egal decat timpul de simulare (t_arrival >= t_simulation).

Urmatoarele campuri vor fi citite din fisier ca date de intrare pentru aplicatie:

- Numarul de clienti (N)
- Numarul de cozi (Q)
- Intervalul de simulare (tmax_simulation)
- Timpul minim si maxim de ajungere (tmin_arrival <= t_arrival <= tmax_arrival)
- Timpul minim si maxim de servire (tmin_service <= t_service <= tmax_service)

Obiective secundare:

- Dezvoltarea de use-case-uri
- Alegerea corecta a structurilor de date
- Impartirea pe clase
- Dezvoltarea algoritmilor
- Implementarea solutiilor
- Testarea programului

**Use case-uri**

Programul va fi apelat din linia de comanda cu ajutorul fisierului .jar creat. In primul rand , trebuie sa ne aflam in directorul corect pentru a apela programul sau sa punem calea relativa a argumentelor fata de directorul in care ne aflam . De asemenea dupa ce introducem ‘ java –jar ‘ , numele jar-ului urmat de argumentul fisierului de intrare in care avem datele dupa care generam cozile si clientii si argumentul fisierului de iesire , unde vom scrie desfasurarea cozilor in functie de timp . De asemenea in fisierul de intrare vom avea urmatoarele valori: numarul de clienti, numarul de cozi, timpul maxim de simulare, valoarea minima si maxima de timp de arrival , valoarea minima si maxima de timp de service. In cazul in care user-ul nu introduce valori corecte pentru argumente programul nu va rula .

**Proiectare**

Pentru realizarea acestui proiect am considerat fiecare coada ca fiind un proces separat , adica un fir de executie , numit si thread. Conceptul de thread (fir de execuție) definește cea mai mică unitate de procesare ce poate fi programată spre execuție de către sistemul de operare. Este folosit în programare pentru a eficientiza execuția programelor, executând porțiuni distincte de cod în paralel în interiorul aceluiași proces. Câteodata însă, aceste portiuni de cod care constituie corpul threadurilor, nu sunt complet independente și în anumite momente ale execuției, se poate întampla ca un thread să trebuiască să aștepte execuția unor instructiuni din alt thread, pentru a putea continua execuția propriilor instrucțiuni. Această tehnică, prin care un thread asteaptă execuția altor threaduri înainte de a continua propria execuție, se numește sincronizarea threadurilor.

“Multithreading” înseamnă capacitatea unui program de a executa mai multe secvenţe de cod în acelaşi timp. O astfel de secvenţă de cod se numeşte fir de execuţie sau thread. Limbajul Java suportă multithreading prin clase disponibile în pachetul java.lang . În acest pachet există 2 clase Thread şi ThreadGroup, şi interfaţa Runnable. Clasa Thread şi interfaţa Runnable oferă suport pentru lucrul cu thread-uri ca entităţi separate, iar clasa ThreadGroup pentru crearea unor grupuri de thread-uri în vederea tratării acestora într-un mod unitar. Există 2 metode pentru crearea unui fir de execuţie: se creează o clasă derivată din clasa Thread, sau se creează o clasă care implementeză interfaţa Runnable. Crearea unui fir de execuţie prin extinderea clasei Thread Se urmează etapele: - se creează o clasă derivată din clasa Thread - se suprascrie metoda public void run() moştenită din clasa Thread - se instanţiază un obiect thread folosind new - se porneşte thread-ul instanţiat, prin apelul metodei start() moştenită din clasa Thread. Apelul acestei metode face ca maşina virtuală Java să creeze contextul de program necesar unui thread după care să apeleze metoda run().Principalele stări ale unui fir de execuție sunt : activ , pregatit si blocat .

Comparativ cu acestea, procesele au și starea specifică Repaus (Sleep). Daca un proces este eliminat din memoria RAM, toate firele de execuție ale acestuia vor fi oprite, deoarece se aflau în același spațiu de memorie ca și procesul respectiv.

**Structuri de date**

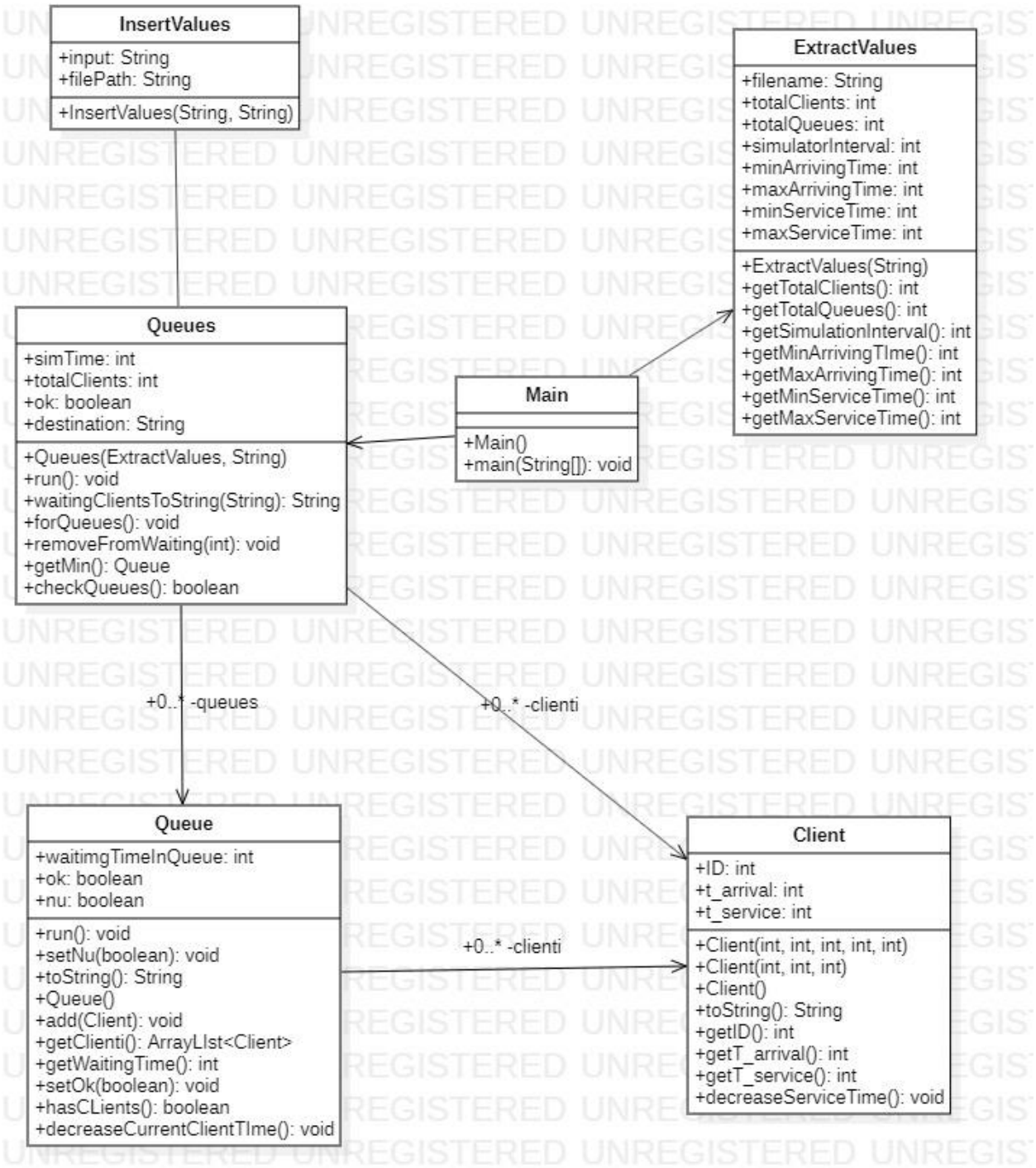
Structura de date principala pe care am folosit-o , cu ajutorul careia stocam clientii si cozile , este ArrayList-ul . In primul rand , acem un ArrayList in care stocam clientii , initial in acesta clientii fiind in asteptare . In al doilea rand , avem un ArrayList in care stocam ArrayList-urile de clienti , acesta reprezentand cozile . Client-ul este o clasa separata si un client este format din ID, timp de ajungere si timp de servire .

De asemenea ne vom mai folosi de clasa Iterator pentru a putea selecta clientii , a-i sterge din lista de waiting clients si pentru a-i introduce in cozi . In acest caz folosirea unui for each nu era convenabil deoarece in cazul stergerii unei valori programul ar fi afisat eroare de out of bounds sau in cazul folosirii unui auxiliar pentru stergerea unui client , daca ar fi doi clienti cu acelasi timp de arrival in waiting clients , programul ar adauga doar unul dintre ei , ceea ce face ca la fisierele cu multi clienti si liste programul nostru sa nu functioneze cum a fost asteptat .

Ex: ArrayList <Client>  clienti = new ArrayList <Client> (N) ;

ArrayList <Queue> queues = new ArrayList <Queue> (Q) ;

## Diagrama de clase



**Algoritmi**

In acest proiect nu avem nevoie de algoritmi complicati. Avem nevoie doar de parcurgerea normala cu for-uri, cu for each si iterator .

```
Iterator<MyClass> it = myCollection.iterator();
while(it.hasNext()) {
  MyClass myClass = it.next():
  // do something with myClass
}


for (MyClass myClass:myCollection) {
  // do something with myClass
}
```

## 2. Implementare

**Clasa Main**

In clasa main avem metoda de extragere a valorilor din fisier si pe baza acestor valori extrase , avem metoda principala Queues care formeaza cozile si le scrie in fisierul de iesire , ambele metode fiind apelate cu argumente .

De asemenea , cu ajutorul .jar-ului creat putem apela programul din linia de comanda. Prima oara trebuie sa ne asiguram ca ne aflam in directorul corect sau inseram argumentele cu tot cu calea lor relativa. Daca ne aflam in fisierul in care se afla si jar-ul si fiserele, urmatoarea linie trebuie scrisa in linia de comanda :

Ex : java –jar JarName.jar input_file.txt output_file.txt

**Clasa InsertValues** – este folosita pentru a scrierea valorilor in fisier .

Are ca variabile doua stringuri, unul pentru ce va fi scris in fisier si unul pentru adresa fisierului. Scrierea in fisier se face cu ajutorul clasei FileWriter. Al doilea parametru din constructorul lui FileWriter este pentru append , care permite continuarea scrierii in fisier. Deci daca acesta este setat pe true valorile scrise vor ramane in fisier si nu vor fi suprascrise de cele scrise ulterior . O problema pe care o prezinta aceasta implementare este ca daca rulam programul de mai multe ori in acelasi fiser de iesire , noile valori vor aparea in continuarea celor vechi

Aceasta clasa o vom folosi in clasa Queues pentru a printa starile cozilor si clientii la fiecare iteratie a timpului ( firelor de executie ) .

Exemplu FileWriter:

```
FileWriter g = new FileWriter (filePath , true) ;
g.write (this.input  /* + " \n " */) ;
```

**Clasa ExtractValues**

Aceasta clasa o vom folosi pentru a extrage valorile necesare rularii programului din fisierul text dat ca si argument . Aceasta clasa o vom folosi si in constructorul clasei Queues .

Extragerea valorilor se face cu ajutorul clasei File pentru a selecta fisierul dorit si clasei Scanner care ne ajuta sa separam liniile din fisier pentru a le imparti in valorile dorite si atribui variabilelor dorite . De asemenea avem settere si gettere pentru toate valorile extrase , pentru a usura atribuirea valorilor.

Exemplu extragere:

```
File f = new File(this.filename);
Scanner sc = new Scanner(f);

while(sc.hasNextLine()){
   data = data + " " + sc.nextLine();
}
```

**Clasa Client**

Aceasta clasa este clasa de baza a acestui proiect deoarece toate operatiile se fac in functie de clienti . Aceasta clasa are 3 campuri : ID , timp de arrival si timp de servire . Timpul de arrival si timpul de servire le vom atribui cate o valoare random intre tmin_arrival si tmax_arrival , respectiv tmin_service si tmax_service . De asemenea vom avea si o metoda decreaseServiceTime care decremeanteaza cu 1 timpul de servire al unui client data ca argument . Aceasta metoda va fi folosita in clasa Queue cand un client se afla in capul cozii si timer-ul trece la urmatoarea etapa . Valoarea random o vom determina cu ajutorul clasei Random .

```
public Client(int id,int ta1,int ta2,int ts1,int ts2){
 this.ID = id;
 Random random = new Random();
 this.t_arrival = random.nextInt(ta2 - ta1 + 1) + ta1 ;
 this.t_service = random.nextInt(ts2 - ts1 + 1) + ts1;
}
```

## Clasa Queue

Dupa cum se poate deduce din informatiile precizate anterior , clasa Queue reprezinta o coada , aceasta fiind formata dint-o lista de clienti . Clientii vor fi stocati intr-un ArrayList . De asemenea aceasta clasa extinde clasa Thread , deci va trebui sa implementam metoda run() care descrie cum se executa firul de executie . Metoda run este controlata de 2 variabile ok si nu care vor fi schimbate in clasa Queues cand executia trebuie sa se opreasca pentru o anumita coada .

De asemenea mai avem si metodele :

public void setNu(boolean aux) - > schimba valoarea lui nu ;

public String toString() - > parcurge sirul de clienti si returneaza valorile tuturor acestora intr-un String ;

public Queue() - > constructorul clasei care initializeaza lista de clienti si seteaza timpul de asteptare al cozii la 0 ;

public void add(Client aux) - > adauga un client nou in lista de clienti curenta si ii adauga timpul de serviciu la timpul de asteptare al cozii ;

public gArrayList<Client> getClienti() - > returneaza lista curenta de clienti ;

public void setOk (boolean ok) - > seteaza valoarea lui ok ;

public boolean hasClients() - > verifica daca lista de clienti este goala sau nu . In caz afirmativ , metoda va returna fals , iar in celalatl caz adevarat ;

public void decreaseCurrentClientTime() - > daca lista nu e goala verifica daca timpul de asteptare al primului client este mai mare decat 0 . Daca este acesta va fi decrementat cu 1 . Daca timpul de asteptare al primului client devine 0 , acesta va fi scos din lista .

## Clasa Queues

In mod evident , aceasta clasa va contine mai multe Queue-uri . De asemenea , deoarece are mai multe Queue-uri care extind clasa Thread , si aceasta clasa va trebui sa o extinda , deci , evident , sa implementeze si metoda run() . Inauntrul metodei run() vom forma cate queue-uri avem nevoie , le vom popula cu clienti si vom scoate sau introduce in cozi clientii , dupa cum este necesar . Totodata , vom parcurge cozile si clientii din ele pentru a face operatiile necesare pentru o buna functionare a proiectului . Pentru calcularea timpului mediu initializam o variabila de tip double in care adunam la fiecare iteratie numarul de clienti prezenti in cozi , dupa care la sfarsitul metodei run() vom imparti valoarea rezultata a variabilei de average la numarul de

clienti pentru a obtine rezultatul dorit . Aceasta metoda de calculare a timpului mediu de asteptare este mai eficienta decat calcularea timpului petrecut la coada al fiecarui client deoarece ne scuteste de implementarea unor campuri sau metode suplimentare . De asemenea vom mai avea si urmatoarele metode :

public Queues(ExtractValues aux , String destination) - > constructorul clasei care ia valorile necesare din fisierul aux , initializeaza variabilele si listele necesare cu valorile respective si scrie rezultatele obtinute in fisierul destination ;

public void forQueues() - > seteaza variabila de control nu ca fiind true pentru toate queue-urile si aplica sleep de 1ms acestora pentru a nu se suprapune . Functia sleep trebuie pusa intr-un bloc de try-catch ;

public void removeFromWaiting(int counter) - > verifica cu ajutorul clasei Iterator ce clienti trebuie scosi din lista de asteptare si adaugati in cozi ;

public Queue getMin() - > returneaza queue-ul cu timpul de asteptare minim ;

public boolean checkQueues() - > verifica daca queue-urile mai au clienti in ele ;

## 3. Testare

Testarea a fost realizata in mod practic cu ajutorul exemplelor date .

## 4. Concluzii si dezvoltari ulterioare

In urma acestui proiect am reusit sa imi dezvolt intelegerea si abilitatile de a lucra cu fire de executie.

O posibila imbunatatire poate fi adusa la capitolul eficientei , folosing o alta structura mai eficienta decat ArrayList-ul.

## Bibliografie

https://ro.wikipedia.org/wiki/Fir_de_execuție
https://en.wikipedia.org/wiki/Thread_(computing)
https://en.wikipedia.org/wiki/Multithreading_(computer_architecture)