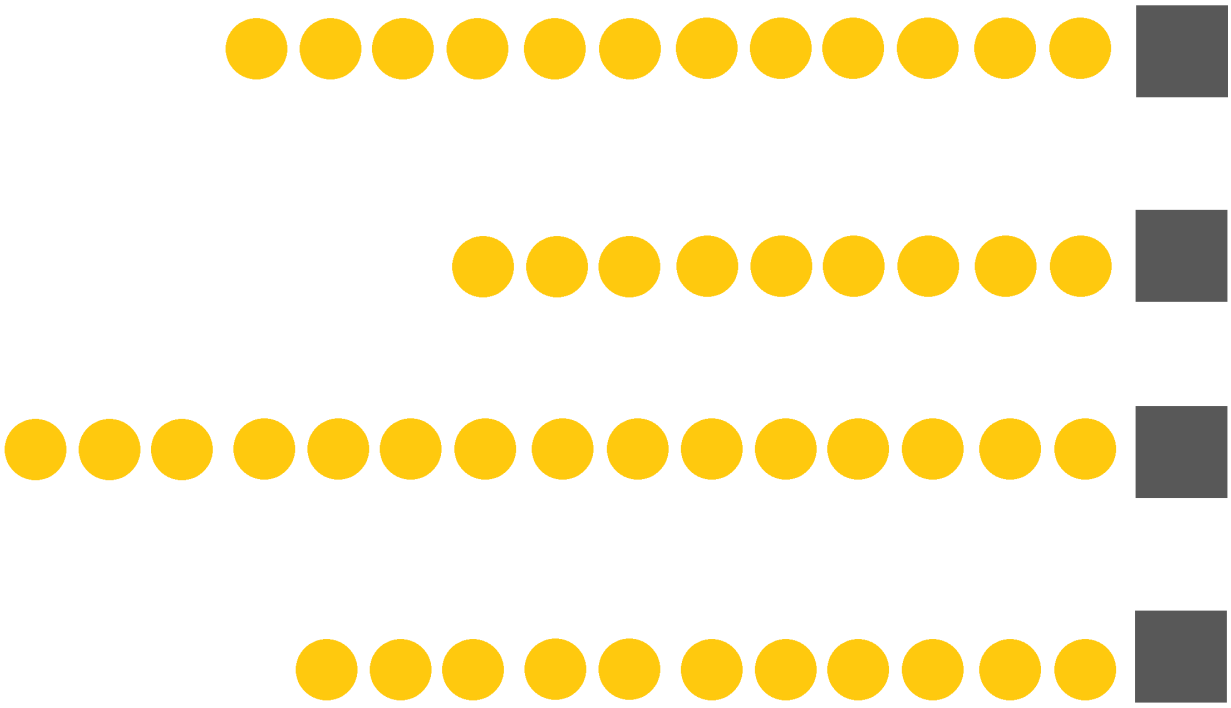
2019 | Fărcaș Alexandru



Supermarket Queue

Tehnici de programare

CUPRINS

OBIECTIVUL TEMEI

Obiectivul principal al celei de a doua tema de laborator este reprezentat de proiectarea si implementarea unei aplicații de simulare menita sa analizeze sistemele bazate pe cozi, pentru determinarea si reducerea timpului pierdut al clienților.

Descriere

Cozile sunt in general folosite pentru modelarea situațiilor din viața de zi cu zi. Obiectivul principal al unei cozi este furnizarea unui loc de așteptare pentru un client, înainte ca acestuia sa ii fie furnizat un serviciu. Scopul sistemelor de administrare al cozilor este de reducerea la minimum a timpului pe care clienții îl pierd așteptând în cozi înainte de a fi serviți. O modalitate de a minimiza timpul de așteptare este adăugarea mai multor servere, adică mai multe cozi în sistem (fiecare coadă este considerată ca având un procesor asociat), dar această abordare mărește costurile furnizorului de servicii. Atunci când se adaugă un nou server, clienții care așteaptă vor fi distribuiți în mod egal pentru toate cozile disponibile la momentul respectiv. Aplicația ar trebui să simuleze o serie de clienți care sosesc pentru un anumit serviciu, alăturându-se unei cozi, așteptând, servind și părăsind la sfârșit coada. In urma timpului pe care clienții îl petrec așteptând rezulta 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 servire depind de clienții individuali - când apar și de câte servicii au nevoie. Timpul de terminare depinde de numărul de cozi, de numărul de clienți din coadă și de nevoile lor de serviciu.

Datele de intrare

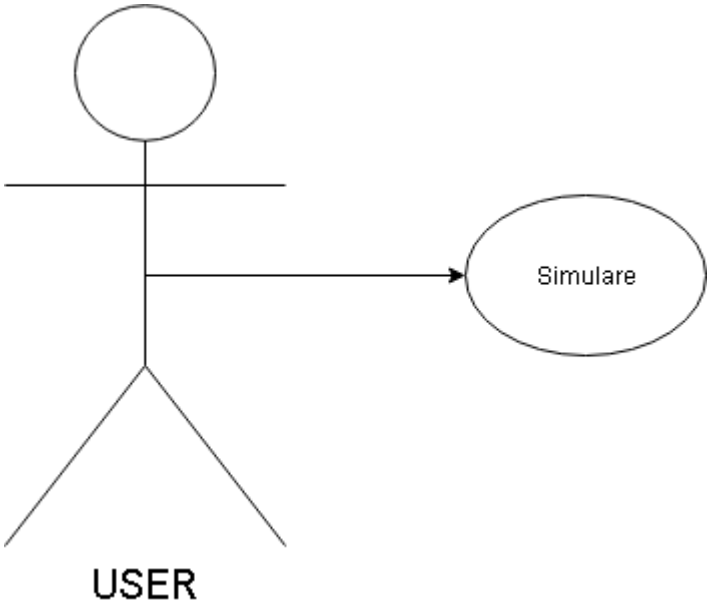
- Intervalul de timp in care pot sosi noi clienți;
- Durata minima si maxima a unui serviciu;
- Numărul de cozi utilizate;
- Intervalul de simulare;
- Alte informații care sunt considerate necesare;

Datele de ieșire

- Timpul mediu de așteptare, de serviciu si timpul in care cozile au fost goale pentru 1, 2 sau 3 cozi in intervalul de simulare sau într-un interval specificat;
- Datele principale de program, respectiv șirul evenimentelor;
- Evoluția cozilor;
- Ora de vârf pentru intervalul simulat;

ANALIZA PROBLEMEI, MODELARE, SCENARII, CAZURI DE UTILIZARE

După cum este specificat încă din cerința, tema abordata are la baza structura de dat de tipul coada („queue”). Coada, prin definiție, este o structura dedate la care adăugarea elementelor se realizează la sfârșit (PUSH) iar eliminarea unui element se realizează de la început. Conform definiției se poate afirma ca o coada este de tip FIFO (first input, first output = primul venit, primul servit). Evident, modelul unui sir la un ghișeu este cea mai nimerita reprezentare pentru o coada. Astfel, apare in discuție problema de implementat. Situația cea mai reprezentativa pentru un sistem de cozi similar cu cel cerut, este cea de la un Supermarket. Aici, clienții vin in mod aleator la casele de marcat, fiecare având un timp diferit de serviciu, care nu poate fi anticipat dinainte ca clientul sa își manifeste intenția de a se alătura unei cozi existente. Scopul urmărit in situațiile de acest gen este reducerea timpului pe care clientul îl petrece in așteptarea furnizării serviciului pentru care sa așezat la o anumita coada. In situația prezentata, toate cozile furnizează același serviciu, si anume posibilitatea de achitare a cumpărăturilor. Cel mai simplu mod de reducere a timpului pierdut este distribuirea clienților la o anumita coada de așteptare in funcție de timpul total de serviciu pe care clienții, deja așezați la coada respectiva, îl însumează. Intr-o aplicație reala, acest lucru este realizat de fiecare client in mod individual, aceștia fiind capabili sa aproximeze durata serviciului celorlalți clienți in funcție de produsele pe care aceștia le au in coșul de cumpărături.



După cum se observa din diagrama ,,*Use Case*”, utilizatorul are la dispoziție o singura funcționalitate din partea aplicației implementate, si anume simularea situației cozilor de la Supermarket, mai sus prezentata. Astfel, utilizatorul trebuie sa introducă un număr de parametrii pentru simulare care urmează sa se efectueze. Acești parametri reprezintă timpul minim de sosire al clienților, timpul maxim de sosire al clienților, timpul minim de serviciu al unui client, timpul maxim de serviciu al unui client si durata simulării măsurată in secunde.

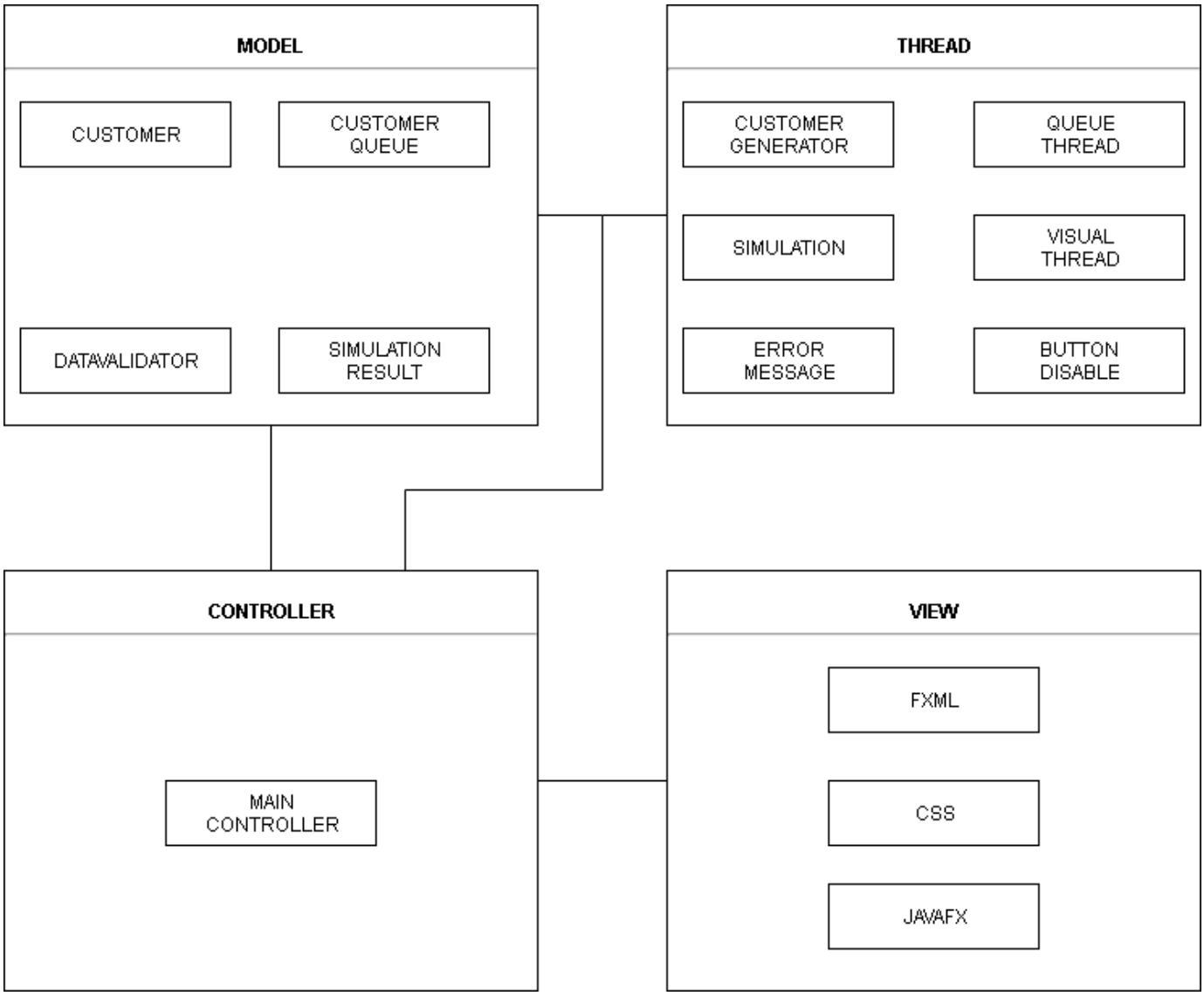
Odată introduși corect acești parametrii, utilizatorul poate cere începerea simulării. După ce simularea a început, utilizatorul are 2 opțiuni: sa aștepte finalizarea acesteia conform numărului de secunde introdus, sau sa ceara întreruperea simulării. In ambele cazuri, la finalul simulării statisticile obținute sunt puse la dispoziția utilizatorului prin intermediul interfeței grafice.

PROIECTARE

După cum am precizat deja, problema abordata presupune o structura de date. Totodată, necesitatea de a folosi mai multe fire de lucru, a dus la utilizarea pachetului *java.util.concurrent,* respectiv a clasei *ConcurrentLinkedQueue<E>.* Aceasta, conform documentației oficiale de la Oracle, este:

An unbounded thread-safe queue based on linked nodes. This queue orders elements FIFO (first-in-first-out). The head of the queue is that element that has been on the queue the longest time. The tail of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue. A ConcurrentLinkedQueue is an appropriate choice when many threads will share access to a common collection. Like most other concurrent collection implementations, this class does not permit the use of null elements. This implementation employs an efficient "wait-free" algorithm based on one described in Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms by Maged M. Michael and Michael L. Scott.

Astfel, structura menționată a permis implementarea fără probleme de sincronizare a unui număr nelimitat de cozi, fiecare cu propriul fir de lucru. Totodată, prin aceasta abordare au fost evitate erorile de programare datorate scrierilor sau citirilor simultane de către diferite fire de lucru, ale aceiași variabile de memorie.



Pe plan global proiectul este realizat după modelul de date MVC (model-view-controller). Clasele utilizate au fost organizate in pachetele corespunzătoare acestui tip de structura, cu excepția claselor responsabile de extinderea celei *THREAD*. Pentru interfața grafica am ales utilizarea JAVAFX.

Astfel, am utilizat următoarele clase Java:

→ Model:
- Customer;
- CustomerQueue;
- DataValidator;
- SimulationResult;

→ Thread:
- CustomerGenerator;
- Simuation;
- ErrorMessage;
- QueueThread;
- VisualThread;
- ButtonDisable;

→ Controller:
- MainController;

IMPLEMENTARE

Clasa Customer.java

```
package model;

public class Customer {
    private int timeOfArrival;
    private int serviceDuration;
    private int waitingTime;

    public Customer(int timeOfArrival, int serviceDuration) {. . .}

    public void setWaitingTime(int waitingTime) {. . .}

    public int getWaitingTime() {. . .}

    public int getServiceDuration() {. . .}
}
```

Aceasta este clasa care sta la baza proiectului. Este folosita ca structura de baza pentru celelalte cozi implementate si stochează datele corespunzătoare fiecărui client, precum timpul la care acesta a ajuns într-una dintre cozi, durata serviciului i respectiv a timpului de așteptare, timp care este calculat in momentul in care acesta se alătură unei cozi, ca suma a timpului clienților deja așezați la coada respectiva.

Clasa CustomerQueue.java

```
package model;

import java.util.concurrent.ConcurrentLinkedQueue;

public class CustomerQueue {
```

```
    private ConcurrentLinkedQueue<Customer> queue = new ConcurrentLinkedQueue<>();
    private int numberOfProcessedCustomers = 0;
    private int totalServiceTime = 0;
    private float averageWaitingTime = 0;
    private float averageServiceTime = 0;

    public CustomerQueue() {}

    public ConcurrentLinkedQueue<Customer> getQueue() {. . .}

    public int getTotalServiceTime() {. . .}

    public float getAverageWaitingTime() {. . .}

    public float getAverageServiceTime() {. . .}

    public void updateAverageData(Customer customer){. . .}

    public void updateTotalServiceTime(){. . .}
}
```

Clasa CustomerQueue.java, are ca variabila de baza o coada din pachetul java.util.concurrent (ConcurrentLinkedQueue<E>), care rezolva problemele de sincronizare, probleme care pot apărea in urma accesării acesteia de doua thread-uri diferite, precum thread-ul de generare de clienți (care vrea sa adauge un nou *Customer* in coda) si thread-ul unic al cozii (care încearcă sa șteargă un *Customer* din coada). Clasa mai are implementate metode de actualizare, responsabile cu recalcularea timpului total de servire, precum si al mediei de așteptare si de servire. Aceste date sunt recalculate la fiecare actualizare a cozii, astfel, informațiile sunt disponibile si in momentul in care simularea a fost întreruptă de către utilizator, valorile afișate fiind cele obținute pana la acel moment.

Parametrul care retine numărul de clienți procesați este folosit la calcularea valorilor de medie in timp real, lucru fiind realizat după formula: *valoare medie = (valoare precedenta * nr + valoare curenta)/++nr;*

Clasa DataValidator.java

```
package model;

public class DataValidator {

    private int minArrival;
    private int maxArrival;
    private int minService;
    private int maxService;
    private int simulationInterval;

    public DataValidator(String minArrival, String maxArrival, String minService,
String maxService, String simulationInterval) throws Exception {. . .}

    public int getMinArrival() {. . .}

    public int getMaxArrival() {. . .}

    public int getMinService() {. . .}

    public int getMaxService() {. . .}

    public int getSimulationInterval() {. . .}
}
```

Validarea datelor introduse de către utilizator se face prin clasa DataValidator.java. Aceasta verifică ca fiecare string introdus reprezintă un număr întreg strict pozitiv folosind funcțiile regex.

Clasa SimulationResult.java

Datele obținute pe durata simulării sunt stocate într-un obiect de tipul SimulationResult pentru fiecare coada individual. Acestea sunt calculate la secunda, după cum am mai precizat, iar după finalizarea forțata sau nu a simulării sunt prezentate utilizatorului prin intermediul interfeței grafice.

```
package model;

public class SimulationResult {
    private float averageWaitingTime, averageServiceTime = 0;
    private int emptyTime, maxServiceTime, peekHour = 0;

    {. . . getters and setters . . .}
}
```

Clasa ButtonDisaableThread.java

Aceasta clasa primește ca parametrii prin constructor butoanele si intervalul de simulare. Scopul urmărit este de a dezactiva butonul responsabil cu începerea simulării pe durata acesteia si de al activa pe cel de anulare al acesteia, respectiv invers.

Clasa CustomerGenerator.java

```
package thread;

import {…};

public class CustomerGenerator extends Thread implements Runnable {

    private ArrayList<CustomerQueue> queues;
    private int time;
    private int minArrival;
    private int maxArrival;
    private int minService;
    private int maxService;
    private boolean stop = false;
    private AtomicReference<String> logView;
    public CustomerGenerator(ArrayList<CustomerQueue> queues, int minArrival, int
maxArrival, int minService, int maxService, AtomicReference<String> logView){. .. .}

    public void setTime(int time) {. . .}

    public void stopSimulation(){. . .}

    public void sendToQueue(Customer customer){. . .}

    @Override
    public void run(){. . .}
}
```

Clasa de generare de clienți funcționează după următorul principiu:

- intervalele introduse de către utilizator sunt primite ca parametru;
- se generează doua numere aliatoare in intervalele specificate pentru durata serviciului si respectiv a sosirii următorului client
- atâta timp cat thread-ul nu este oprit de către thread-ul principal responsabil cu simularea, acesta furnizează clienți cu numere aliatoare, iar după fiecare noua generare, acesta ,,doarme" pana la furnizarea următorului client

Clasa ErrorMessage.java

Aceasta clasa are ca scop afișarea unui mesaj de eroare pentru câteva secunde in momentul introducerii unei valori eronate de către utilizator. Acest lucru este realizat prin primirea ca parametru in constructor al unei variabile de tipul *Label*, pe care o face vizibile, urmând ca după intervalul de timp dorit sa fie readusa la starea de invizibilă.

Clasa QueueThread.java

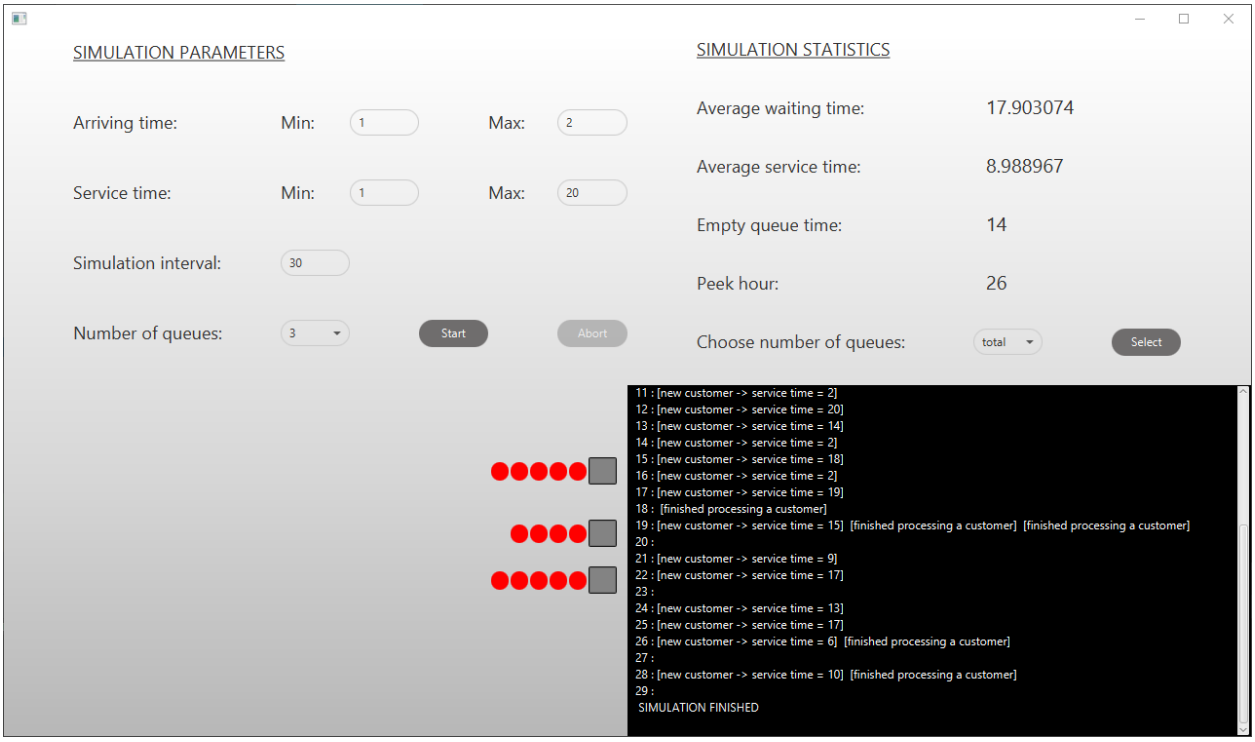
Fiecare clasa are asociat un thread implementat prin clasa QueueThread.java. Aceasta funcționează pana când este oprita de către thread-ul principal. Pe durata servirii unui client aceasta ,,doarme", urmând ca imediat după aceea sa îl elimine din coada si sa reia bucla de funcționare.

Clasa Simulation.java

```
package thread;

import javafx.scene.control.Button;
import javafx.scene.control.TextArea;
import javafx.scene.layout.GridPane;
import model.CustomerQueue;
import model.SimulationResult;

import java.util.ArrayList;
import java.util.concurrent.atomic.AtomicReference;

public class Simulation  extends Thread implements Runnable{
    private int time = 0;
    private int minArrival, maxArrival, minService, maxService, simulationDuration,
numberOfQueues,
                firstQueueSize, secondQueueSize, thirdQueueSize, fourthQueueSize,
fifthQueueSize;
    private Button[] firstQueue, secondQueue, thirdQueue, fourthQueue, fifthQueue;
    private boolean stop = false;
    private ArrayList<CustomerQueue> queues = new ArrayList<>();
    private TextArea logView;
    private AtomicReference<String> logViewString = new AtomicReference<>();
    private GridPane resultGrid;
    private SimulationResult firstQueueResult = new SimulationResult();
    private SimulationResult secondQueueResult = new SimulationResult();
    private SimulationResult thirdQueueResult = new SimulationResult();
    private SimulationResult fourthQueueResult = new SimulationResult();
    private SimulationResult fifthQueueResult = new SimulationResult();

    public Simulation(int minArrival, int maxArrival, int minService, int maxService,
int numberOfQueues, int simulationDuration,
                      Button[] firstQueue, Button[] secondQueue, Button[] thirdQueue,
Button[] fourthQueue, Button[] fifthQueue, TextArea logView,
                      SimulationResult firstQueueResult, SimulationResult
secondQueueResult, SimulationResult thirdQueueResult,
                      SimulationResult fourthQueueResult, SimulationResult
fifthQueueResult, GridPane resultGrid){. . .}

    public Simulation() { }

    public void updateQueues(){. . .}

    public void stopSimulation(){. . .}

    public TextArea getLogView() {. . .}

    public ArrayList<CustomerQueue> getQueues() {. . .}

    @Override
    public void run(){. . .}
}
```

Aceasta este clasa care controlează impropriu-zis întregul proces de simulare. Aici se pornesc thread-urile individuale in funcție de numărul de cozi primit ca parametru, se recalculează rezultatele simulării la secunda si la nevoie se oprește întregul proces.

Interfața grafică



Interfața grafica este realizata folosind JAVAFX si CSS pentru un aspect cat mai prietenos cu utilizatorul. Inițial acesta poate introduce parametrii de simulare in partea din stânga sus a ecranului, iar rezultatele rezultate vor putea fi vizualizate imediat lângă acestea. Procesul de simulare este reprezentat grafic prin cercuțe rotunde de

culoare roșie, reprezentând clienții, si prin pătratele care ilustrează casele de marcat. Totodată evenimentele sunt prezentate si in căsuța de terminal din partea dreapta jos a ecranului.

REZULTATE

Rezultatele simulării nu pot fi evaluate exact folosindu-se JUnit, însă am testat individual anumite funcții pe durata implementării proiectului folosind Framework-ul precizat.

CONCLUZII

Aceasta tema a condus la dobândirea unor cunoștințe care nu pot fi însușite decât prin exercițiu, iar tema efectiv abordata este de importanta redusa comparativ cu scopul educativ al cerinței. In alta ordine de idei, implementarea in viața de zi cu zi a unui sistem de distribuire al persoanelor la diferite cozi de așteptare ar reprezenta o inovație binevenita.

BIBLIOGRAFIE

→ https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html

→ https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html

→ https://en.wikipedia.org/wiki/Queue_(abstract_data_type)