

**FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE DEPARTAMENTUL CALCULATOARE**

DOCUMENTAŢIE TEMA 2

SIMULARE COZI CU CLIENŢI

la disciplina

Tehnici de programare

CĂRBUNE IOANA - ELENA

Grupa 30221

An academic : 2018 - 2019

Cuprins

[1. Obiectivul temei 3](#_Toc5131986)

[1. 1. Obiectivul principal 3](#_Toc5131987)

[1. 2. Obiective secundare 3](#_Toc5131988)

[2. Analiza problemei 3](#_Toc5131989)

[2. 1. Use-case 3](#_Toc5131990)

[2. 2. Descrierea paşilor 4](#_Toc5131991)

[3. Proiectare 4](#_Toc5131992)

[3. 1. Decizii de proiectare 4](#_Toc5131993)

[3. 2. Diagramă pachete 4](#_Toc5131994)

[3. 3. Diagrama de clase ( nivel conceptual ) 5](#_Toc5131995)

[3. 4. Structuri de date 5](#_Toc5131996)

[3. 5. Proiectare clase 5](#_Toc5131997)

[3. 6. Relaţii 6](#_Toc5131998)

[3. 7. Interfaţa utilizator 6](#_Toc5131999)

[4. Implementare 6](#_Toc5132000)

[4. 1. Implementare clase 6](#_Toc5132001)

[4. 1. 1. Customer 6](#_Toc5132002)

[4. 1. 2. SimulationQueue 6](#_Toc5132003)

[4. 1. 3. RemovingThread 6](#_Toc5132004)

[4. 1. 4. MainView 7](#_Toc5132005)

[4. 1. 5. MainController 7](#_Toc5132006)

[4. 1. 6. MessageController 8](#_Toc5132007)

[4. 1. 7. App 8](#_Toc5132008)

[4. 2. Implementarea interfeţei utilizator 8](#_Toc5132009)

[5. Concluzii 8](#_Toc5132010)

[6. Bibliografie 9](#_Toc5132011)

# 1. Obiectivul temei

## 1. 1. Obiectivul principal

Obiectivul principal al temei este acela de a dezvolta un sistem de simulare şi gestionare a unor cozi cu clienţi. Pe baza unor parametri introduşi de către utilizator este necesar ca fiecare coadă să poată surpinde venirea, respectiv plecarea clienţilor. Pe baza simulării se urmăreşte generarea unor rezultate la finalul acesteia.

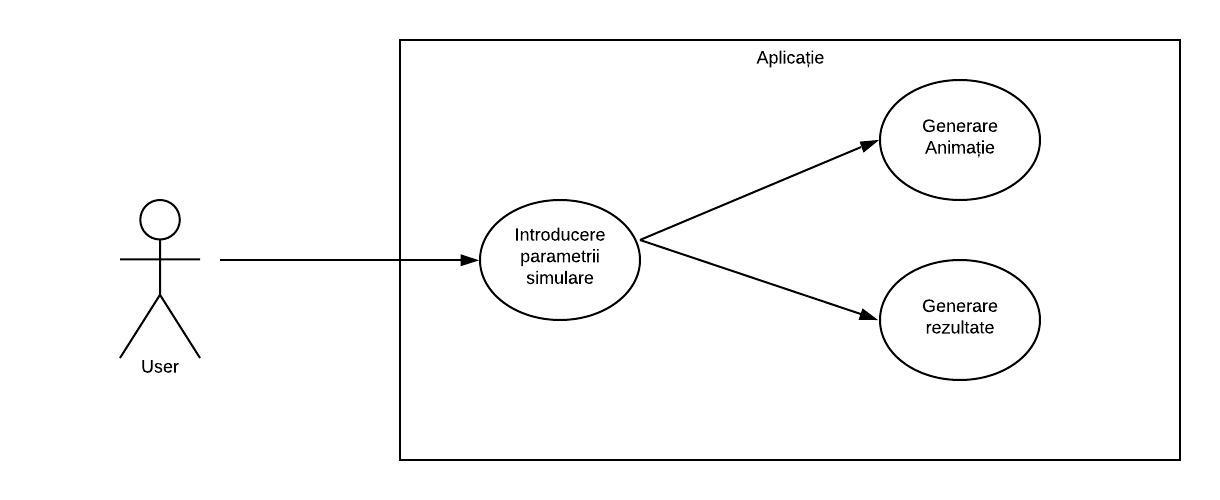
## 1. 2. Obiective secundare

|  |  |  |
| --- | --- | --- |
| Obiective secundare | Descriere | Capitol |
| Descriere use-case | Constă în descrierea sumară a funcţionalităţii aplicaţiei . | 2 |
| Structuri de date | Descrierea şi argumentarea structurilor de date alese . | 3 |
| Proiectarea claselor | Se prezintă clasele realizate. | 3 |
| Implementare interfaţă utilizator | Constă în prezentarea interfeţei GUI . | 4 |

2. Analiza problemei

2. 1. Use-case

Se observa că user-ul este responsabil doar de introducerea parametrilor necesari simulării.



## 2. 2. Descrierea paşilor

Primul pas şi cel mai important constă în introducerea parametrilor. În ordine, aceştia sunt: timpul minim de sosire, timpul maxim de sosire, timpul minim de serviciu, timpul maxim de serviciu, numărul de cozi şi de asemenea şi timpul de simulare.

După pornirea aplicaţiei va apărea o animaţie care va conţine atâtea cozi cât a fost introdus de user. Pe baza parametrilor de sosire va apărea câte un client, care odată ajuns la casă va sta acolo un număr de secunde ales întâmplător între intervalul minim şi maxim de serviciu. După terminarea serviciului lor, clienţii pleacă.

În tot acest timp se pot observa evenimentele afişate pe ecran. La finalul simulării, user-ului i se aduc la cunoştinţă patru rezultate pe baza simulării: timpii medii de aşteptare şi servire, timpul mediu în care cozile au fost goale dar şi ora de vârf la care clienţii au ajuns la un număr maxim.

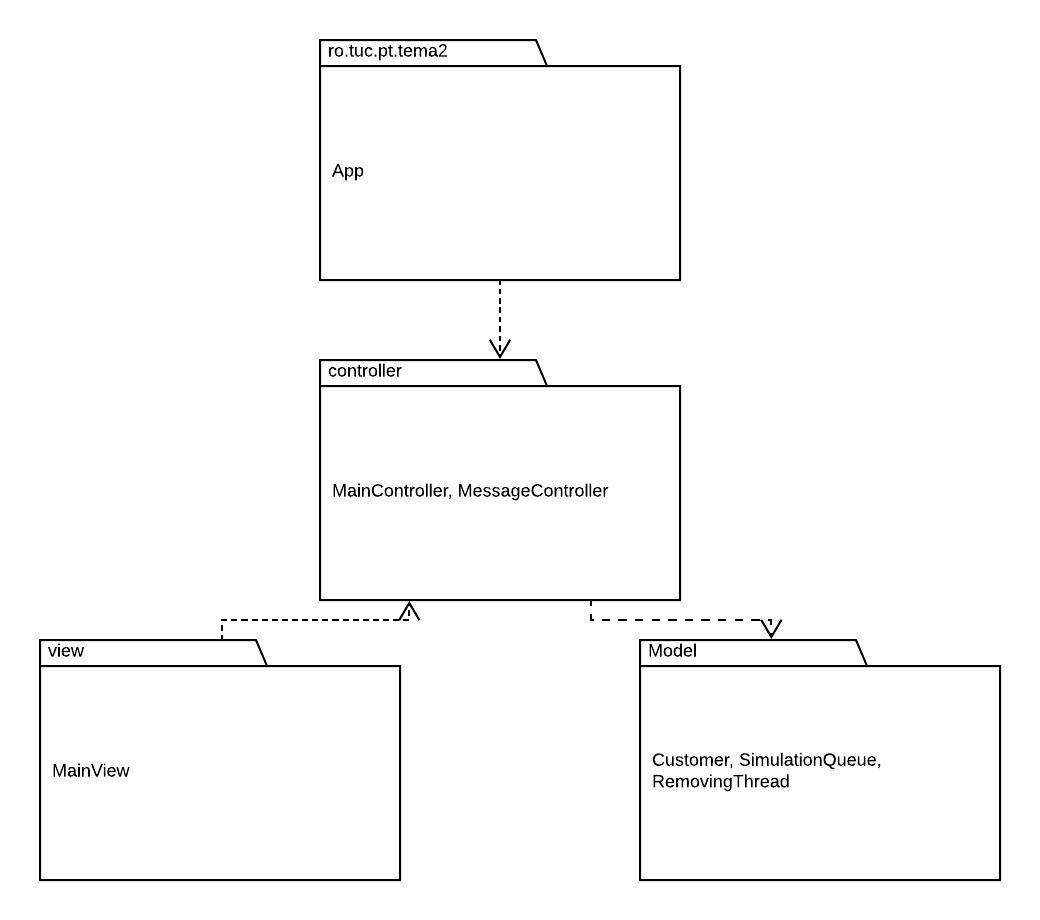
3. Proiectare

## 3. 1. Decizii de proiectare

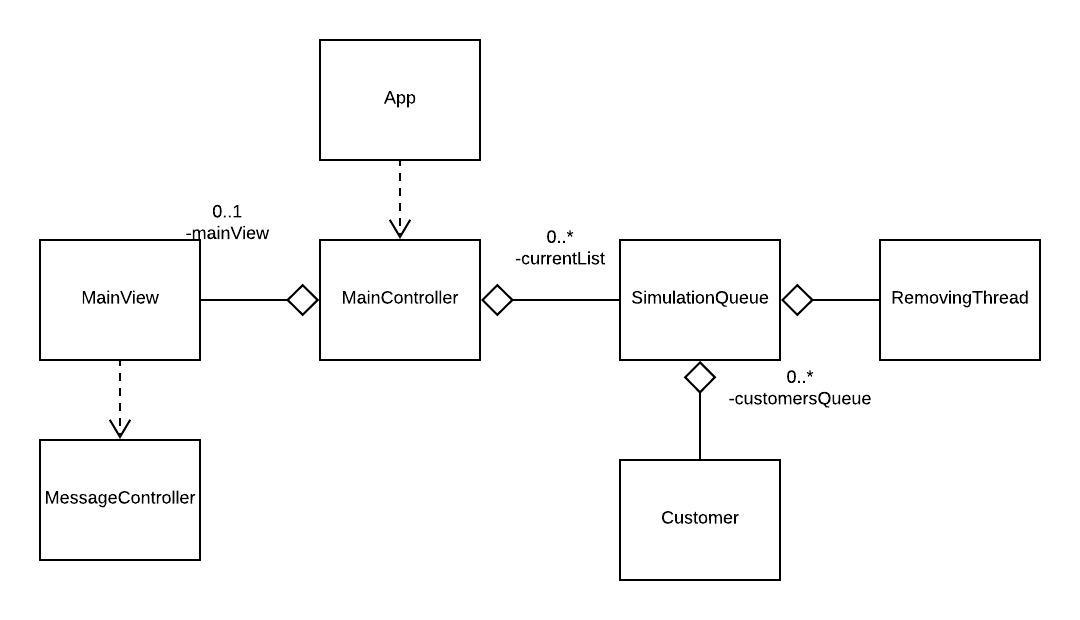
O decizie majoră de proiectare luată în realizarea acestei aplicaţii este cea de a urma arhitectura MVC ( Model-View-Controller ) cu scopul de a fi totul bine organizat şi structurat. Prin urmare, proiectul este structurat pe mai multe pachete în funcţie de rolul fiecărei clase. Din punct de vedere al relevanţei pentru această aplicaţie, cele mai importante clase sunt Customer şi SimulationQueue.

## 3. 2. Diagramă pachete

În diagrama de mai jos se poate observa modul în care aplicaţia este structurată pe pachete, relaţiile dintre acestea şi clasele componente.



## 3. 3. Diagrama de clase ( nivel conceptual )

În această diagramă se pot observa clasele şi relaţiile dintre acestea, care urmează a fi descrise într-un capitol următor..

## 3. 4. Structuri de date

În mod firesc o coada de clienţi poate fi reprezentată cel mai bine cu o structură de date de tip Queue. Datorită necesităţii de a lucra cu mai multe fire de lucru şi de a realiza operaţii de adăugare şi de ştergere pe respectivele cozi, am ales să folosesc ConcurrentLinkedQueue. Această alegere este motivată în special de faptul că în timp ce un fir de lucru ar putea adăuga un client în coadă, un altul ar putea acţiona asupra aceleaşi cozi încercând să şteargă respectivul client şi deoarece firele de lucru acţionează independent trebuie asigurat faptul că înainte de a fi şters elementul apare în coadă.

De asemenea, deoarece nu se ştie din momentul implementării câte cozi o să fie, am ales să folosesc structura List pentru a le stoca.

## 3. 5. Proiectare clase

Aplicaţia este formată din următoarele clase:

* Customer;
* SimulationQueue;
* RemovingThread;
* MainView;
* MainController;
* MessageController;
* App;

În proiectarea claselor am apelat la încapsulare. Acestea conţin atribute private care pot fi accesate din alte clase prin metodele aferente fiecăreia, acestea fiind publice.

## 3. 6. Relaţii

După cum s-a putut observa în diagrama de mai sus între clasele aplicaţiei există mai multe relaţii. Majoritatea relaţiilor sunt de agregare. SimulationQueue are o lista de Customer, dar şi un RemovingThread. De asemenea, în MainController este instanţiată o listă de SimulationQueue dar şi un MainView. Celelalte relaţii sunt de dependenţă între App şi MainController, respectiv între MainView şi MessageController.

## 3. 7. Interfaţa utilizator

Interfaţa utilizator este proiectată în felul următor: utilizatorul dispune de posibiliatea de a introduce şase texte corespunzătoare celor şase parametri esenţiali simulării, iar sub acestea se află un buton pentru start. După apăsarea butonului deasupra va apărea o animaţie iar în dreapta acesteia se vor observa evenimentele în ordine cronologică. Datorită faptului că unul din scopurile acestei simulări este de a genera nişte rezultate, în partea dreaptă se vor putea observa patru texte needitabile corespunzatoare acestora. Dacă utilizatorul omite completarea unui câmp, după apăsarea butonului de start se va afişa un mesaj de eroare. De asemenea, sunt posibile simulări succesive datorită faptului ca la apăsarea butonului toate datele se resetează.

# 4. Implementare

### 4. 1. Implementare clase

### 4. 1. 1. Customer

* Atributele clasei Customer sunt waitingTime, serviceTime şi arrivingTime ;
* Ca şi metode, clasa Customer constă doar în setters şi getters importanţi pentru calculele pe coadă.

### 4. 1. 2. SimulationQueue

* Precum s-a specificat mai sus, clasa SimulationQueue conţine o listă de clienţi ( Customer ), un RemovingThread, un index şi nişte parametri importanţi în generarea rezultatelor medii şi anume totalServiceTime, totalWaitingTime şi numberOfClients;
* Adăugarea clienţilor -> addCustomer ( Customer ) : void -> această metodă are rolul de a adăuga un client nou primit ca parametru la lista de clienţi a cozii , creşte numărul de clienţi, dar în acelaşi timp modifică şi timpul total de aşteptare şi de servire a cozii adăugând datele cu care vine clientul curent;
* Calculul timpului curent de aşteptare -> getCurrentWaitingTime ( ) : Integer -> are rolul de a calcula timpul curent de aşteptare în vederea folosirii rezultatului la alegerea cozii în care va fi plasat următorul client. Metoda parcuge fiecare client aflat în acel moment în coadă şi adună la rezultat timpul necesar pentru servirea lor ;
* Celelalte metode sunt setters şi getters pentru atributele clasei, fiecare din acestea fiind foarte importante la calcularea rezultatelor finale.

### 4. 1. 3. RemovingThread

* Această clasă este concepută pentru a gestiona ştergerea clienţilor de pe coada la care aparţine. Are doar două metode, start şi stop;
* Metoda start ( SimulationQueue , MainView ) : void -> constă în pornirea unui nou fir de lucru care are ca şi scop principal ştergerea primului client din coada primită ca parametru şi a imaginii corespunzătoare respectivului client. De asemnea această metodă afişează şi un mesaj corespunzător în fereastra cu evenimente .
* Metoda stopCurrentThread ( ) : void -> are rolul, aşa cum e de aşteptat, de a opri un fir de lucru.

### 4. 1. 4. MainView

* Clasa MainView conţine tot ce ţine de interfaţa grafică: 1 x Button pentru start ce porneşte simularea,6 TextField-uri pentru introducerea parametrilor, 4 label-uri pentru TextField urile rezultatelor , 4 TextField-uri needitabil pentru afişarea rezultatelor, un Label de eroare şi nu în ultimul rând un TextArea pentru afişarea evenimentelor . Acestea sunt structurate în funcţie de rolul lor în VBox-uri şi HBox-uri pentru o aliniere egală. De asemenea, animaţia este plasată într-un VBox. Acesta conţine pe fiecare linie un HBox în care urmează să fie plasate cozile. Numărul de liniile alea VBox-ului depinde de numărul de cozi introdus de utilizator.
* Ca şi metode importante se pot aminti createAnimationSpace ( Integer ) : void , aceasta fiind responsabilă pentru crearea propriu zisă a spaţiului în care va fi aplasată animaţia. Cum am specificat mai sus, în funcţie de numărul de cozi aceasta va segmenta spaţiul disponibil pentru fiecare coadă în parte. Iniţial aceasta va afişa câte o imagine la fiecare început de coadă reprezentând o casierie.
* Metoda addCustomerIcon ( Integer) : void -> are rolul de a afişa câte un icon la finalul HBox-ului care conţine indexul cozii primite ca parametru .
* Similar metodei de mai sus, metoda removeCustomerIcon ( Integer) : void -> are rolul de a şterge câte primul icon ( următorul după cel cu casieria) a cozii primite ca parametru .
* Celelalte metode au doar rol de gestionare a conţinutului TextField-urilor, TextArea şi Label.

### 4. 1. 5. MainController

* Start ( Stage ): void -> apelează metoda show din MainView care are rolul de a arăta pe ecran interfaţa grafică.
* initializeButtonListeners ( ) : void ->apelează pentru butonul de start metoda de ActionListener din MainView. Aici se întâmplă un lucru vital pentru funcţionarea corectă a programului. Se creează un fir de lucru nou care are ca şi scop adăugarea de clienţi la momentul potrivit. În primul rând, preia parametri simulării cu ajutorul cărora creează un client nou. Cu ajutorul unei metode explicate mai jos, adaugă clientul curent la coada la care o sa aibă cel mai scurt timp de aşteptare. Totodată , actualizează mesajele cu evenimente din text area, stabileşte timpul mediu în care cozile au fost goale şi ora de vârf. De asemenea, şi afişează rezultatele la finalul simulării.
* emptyQ ( List ) : Boolean -> are rolul de a parcurge şi de a verifica dacă există cel puţin o coadă goala la momentul curent.
* stopRemovingThreads ( List ) : void -> parcurge fiecare coadă şi îi opreşte firul de lucru destinat ştergerii clienţilor în momentul în care timpul de simulare se termină.
* getShortestQueue ( List ) : int -> are rolul de a stabili ce coadă o să aibă cel mai scurt timp de aşteptare pentru următorul client. Aceasta parcurge fiecare coadă şi află minimul , returnând mai apoi indexul acesteia.
* averageWaitingTime ( List ) : Double -> are rolul de a calcula timpul mediu de aşteptare calculat ca timpul de aşteptare pentru fiecare client de pe fiecare coadă în raport cu numărul total de clienţi.
* averageServiceTime ( List ) : Double -> are rolul de a calcula timpul mediu de servire calculat ca timpul de servire pentru fiecare client de pe fiecare coadă în raport cu numărul total de clienţi.
* getCurrentCustomerNumber ( List ) : int -> calculează şi returnează numărul curent de clienţi.

### 4. 1. 6. MessageController

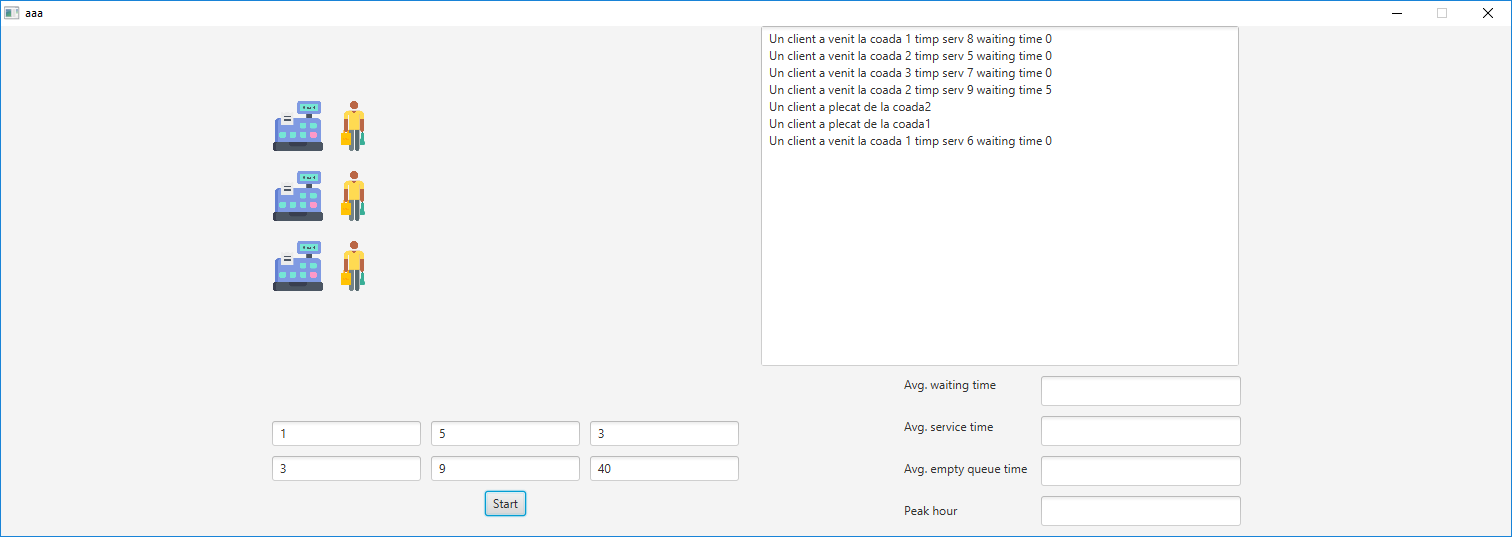
* Constă într-o singură metoda şi anume displayMessage ( String, Label , Color ) : void care are rolul de a crea un fir nou de lucru cu scopul de a afişa în label-ul primit ca paramentru un mesaj timp de 3 secunde după care acesta va dispărea.

### 4. 1. 7. App

* Această clasă are rolul de a lansa aplicaţia.

## 4. 2. Implementarea interfeţei utilizator

Pentru implementarea intefeţei utilizator am ales să utilizez JavaFx deoarce în opinia mea este mai accesibil decât alte opţiuni, de exemplu Swing. Cum a fost precizat mai sus ca şi elemente au fost folosite butoane, TextField-uri, lable-uri , TextArea dar şi VBox-uri şi HBox-uri. Toate acestea au fost poziţionate într-un BorderPane care are rol de layout. De asemenea, butonului îi corespunde o metodă addStartButtonActionListener ( EventHandler<ActionEvent> ) care are rolul de a-i atribui o acţiune.



# 5. Concluzii

Datorită acestei teme am reuşit o mai bună familiarizare cu ideea de fire de lucru multiple şi am avut ocazia să folosesc o structură de date nouă , şi anume ConcurrentLinkedQueue.

Consider că o posibilă dezvoltare a temei poate fi adăugarea unui sistem de gestionare mai dinamic al cozilor ca de exemplu atunci când numărul de clienţi depăşeşte o anumită valoare să se “ deschidă ” automat o casă nouă.

O dezvoltare mai mult grafică decât funcţională ar fi diversificarea clienţilor, adică mai multe icone-uri pentru reprezentarea acestora.

# 6. Bibliografie

* Pentru afişarea mesajelor de eroare :

<https://stackoverflow.com/questions/19968012/javafx-update-ui-label-asynchronously-with-messages-while-application-different>

* Pentru folosirea ConcurrentLinkedQueue :

<https://developer.android.com/reference/java/util/concurrent/ConcurrentLinkedQueue>

* Pentru metode JavaFX :

<https://docs.oracle.com/javafx/2/ui_controls/jfxpub-ui_controls.htm>