Technical University of Cluj-Napoca

Year 2016-2017

Department of Computer Science

Programming Techniques

2nd Assignment

Student: Gyarmathy Tímea

Group: 30423

Table of Contents

[Specification of the problem 3](#_Toc479318735)

[Description 3](#_Toc479318736)

[The analysis of the problem 3](#_Toc479318737)

[Modeling 3](#_Toc479318738)

[Use cases 3](#_Toc479318739)

[Scenarios 4](#_Toc479318740)

[Design 5](#_Toc479318741)

[Design concepts 5](#_Toc479318742)

[Class diagram 6](#_Toc479318743)

[The Model 6](#_Toc479318744)

[The View 9](#_Toc479318745)

[The Controller 10](#_Toc479318746)

[The Main Class 12](#_Toc479318747)

[Sequence diagram 13](#_Toc479318748)

[Graphical User Interface 13](#_Toc479318749)

[Implementation and testing 14](#_Toc479318750)

[Results 14](#_Toc479318751)

[Conclusions 14](#_Toc479318752)

[What have I learned? 14](#_Toc479318753)

[Further developments 14](#_Toc479318754)

[Bibliography 15](#_Toc479318755)

# Specification of the problem

Design and implement a simulation application aiming to analyze queuing based systems for determining and minimizing clients’ waiting time.

# Description

Queues are commonly used to model real world domains. The main objective of a queue is to provide a place for a "client" to wait before receiving a "service". The management of queue based systems is interested in minimizing the time amount their "clients" are waiting in queues before they are served. One way to minimize the waiting time is to add more servers, i.e. more queues in the system (each queue is considered as having an associated processor) but this approach increases the costs of the service supplier. When a new server is added the waiting customers will be evenly distributed to all current available queues.

The application should simulate a series of clients arriving for service, entering queues, waiting, being served and finally leaving the queue. It tracks the time the customers spend waiting in queues and outputs the average waiting time. To calculate waiting time we need to know the arrival time, finish time and service time. The arrival time and the service time depend on the individual clients – when they show up and how much service they need. The finish time depends on the number of queues, the number of clients in the queue and their service needs.

# The analysis of the problem

As seen in the description, we will have to work with virtual models of real life structures, and so, they are associated with random events, that should not be predictable in any way. Thus, we will use threads and randomly generated data.

# Modeling

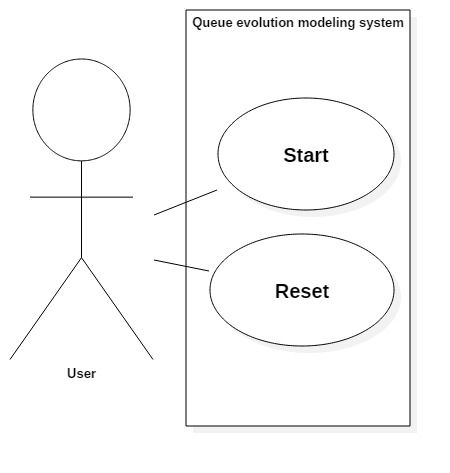
To be able to model a queue based system we have to decompose the problem into the basic building units: customer, queue and the market. Each queue will run its separate thread and the whole market will also have a thread.

Customers should retain the information about their arrival and service time, and each customer should be unique, hence we’ll also associate them an ID number.

Queues are dynamic, their data constantly changes in time as customers arrive and they are served. Queues should hold an array of customers which is constantly updated throughout the thread’s evolution. Besides this, information like the average of waiting time, service time and empty queue time is also stored and constantly updated. Constant data passed to the queues from the user are the minimum and the maximum service time, which define the interval of the randomly generated numbers.

Markets hold an array of queues, and adds customers to them, which arrive randomly. For this, we’ll pass the minimum and the maximum required arrival time from the user and generate random numbers within the specified interval. The next arriving customer will always be added to the shortest queue.

# Use cases

The user (actor) has to input the required data, the preferred simulation time and press the start button to start the simulation. From this point on, he will follow the evolution of the queues until the simulation interval is over. This is also displayed to the user, as well as the peak time (the moment when the total number of customers in the market was the largest).

## Scenarios

Precondition: The user successfully launches the application.

Success:

1. The user introduces the minimum arrival time for the customers in the upper field.
2. The user introduces the maximum arrival time for the customers in the field below.
3. The user introduces the minimum service time for the customers.
4. The user introduces the maximum service time for the customers.
5. The user introduces the number of queues.
6. The user introduces the preferred simulation time.
7. The user presses the “Start” button.
8. Input data is parsed, transformed and passed internally to the control of the system.
9. Simulation starts, threads are created and started, queues appear on the interface in their initial state.
10. Customers arrive and are served, evolution of the market can be followed on the screen, events are specified in the log; running time is also displayed.
11. Simulation time is over, customers stop arriving, queues stop serving, and all threads stop.
12. Final data remains on the screen.

Alternative scenarios:

**First:** Bad data is input into one of the text fields.

1. Error message displayed on a separate pop-up window that informs the user about the invalid input.
2. User corrects mistakes.
3. Return to step 7.

**Second:** Reset button is pressed anytime during the simulation.

1. Execution stops, threads are interrupted.
2. Input data can be modified by the user.
3. Return to step 7.

**Third:** Start button is re-pressed anytime during the simulation.

Do not. It has no effect, but a quick flicker on the user interface.

# Design

## D:\Desktop\Computer Science year 2 sem II\Java programs\PT2017_30423_Gyarmathy_Timea\Assignment2\packages.pngDesign concepts

The main concept of developing this queue processing system is based on Model – View – Controller design which means that firstly we will have three packages: Model, View and Controller, and to run the application a Main class was added besides these, in the default package.

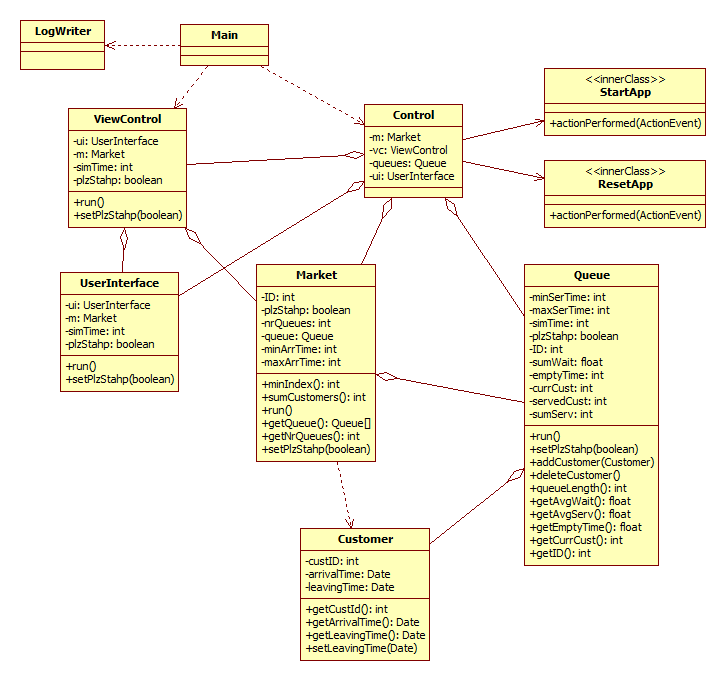
In the Model we will clearly have the Customer, Queue and Market classes.

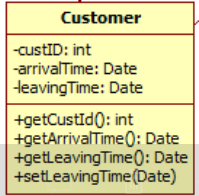
The View contains a single class, UserInterface, which implements the graphical user interface. It has methods which can be called by the control of the system to redraw the data that changed throughout the simulation time.

The Control package initially contained a single class Control, but I added a separate class, ViewControl to only periodically refresh the graphical user interface. The latter extends the Thread class, and runs until the simulation time is not over. The main control of the system remains the class Control, which takes the input data, parses it, creates the objects, threads, and starts the execution when start signal arrives from the user. To be able to create a log of events into the GUI, I also added a LogWriter class which redirects the output stream of the system into a text area in the interface.

Near all these we also have a separate Main class which contains only the main function that runs the application: creates an instance of the GUI, asks the redirection of the system output, and creates a Control on the instantiated interface.

## Class diagram

Summarized class diagram (some getters, setters and helper methods were omitted for comprehension):

A complete class diagram is added to the end of this chapter.

### The Model

#### Class Customer

Models relevant information on a single customer: his ID, arrival time and leaving time. Besides its constructor, it only contains getters and setters for these data.

#### Class Queue

Extends Thread and represents a single queue which, once started, has its data constantly updated throughout the thread’s lifetime. It has a set of Customers stored in a structure of type Vector, because it eases addition and removal of objects. To meet the minimal output requirements, it calculates and retrieves the queue average waiting and service time, the queue empty time and the ID of the customer which is currently served.

Its run method, explained:

* The try-catch block surrounds the instructions, because, as being a thread, these cannot execute any further if their thread was interrupted.
* The variable ser is used to hold the randomly generated numbers for the customer being served, i. e. a service time.
* The variable plzStahp represents the control signal of the thread, when set true, the while from the run method exits, thus the threads stops its execution.
* While it is running, it verifies the length of the queue, and if not empty, takes action. Elsewise, it waits until a customer arrives and counts the queue empty time.
* When the queue is not empty, we try to serve the first customer in the line by generating a random service time for him and after this is over, we “delete” him by calling the deleteCustomer method from this class. This method, besides calculating and updating the information to be displayed later, diplays a message on the screen:

System.***out***.println("Customer " + currCust + " served in the queue: " + ID);

The code of the run method from class Queue:

@Override

**public** **void** run() {

**try** {

**int** ser = 0;

//Run it until stop signal arrives

**while** (plzStahp == **false**) {

**if** (customers.size() > 0) {

//Serve the first customer in the queue

Customer c = (Customer) customers.elementAt(0);

currCust = (**int**) c.getCustID();

//Generate a random service time for him

ser = (**int**) (minSerTime +Math.*random*()\*(maxSerTime -minSerTime));

*sleep*(ser);

deleteCustomer();

sumServ += ser;

} **else** {

//No customers in the queue, count empty time

**while** (customers.size() == 0) {

currCust = 0;

**int** startTime = (**int**) System.*currentTimeMillis*();

*sleep*(100);

emptyTime += (**int**) System.*currentTimeMillis*() - startTime;

}

}

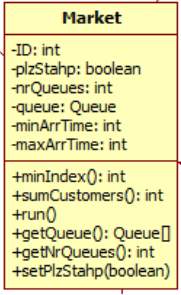
}

} **catch** (InterruptedException e) {

JOptionPane.*showMessageDialog*(**null**, e.toString(),"Error",

JOptionPane.***ERROR\_MESSAGE*** );

} **catch** (Exception excg){}

}

#### Class Market

Extends Thread and models an entire market, holding an array of queues, constantly generating randomly arriving customers and adding the next one to the shortest queue. When constructed, it gets the minimum and the maximum allowed arrival time of a customer, and of course, the number of queues.

Its run method, explained:

* Variable m will hold the index of the shortest queue, calculated by the method minIndex, implemented in this class.
* Variable arr is used to hold the randomly generated arrival time for each customer.
* The thread runs until signal plzStahp is set to true.
* While it is running, it sleeps for a randomly generated time period, then adds a new Customer to the shortest queue.
* It displays the following message if a customer arrived:

System.***out***.println("Customer " + Long.*toString*(ID) + " added to queue: " + Integer.*toString*(m + 1));

The code of the run method from class Market:

@Override

**public** **void** run() {

**try** {

**int** m, arr;

//Run the thread until stop command arrives

**while** (!plzStahp) {

//Set random arrival time for next customer

arr = (**int**) (minArrTime + Math.*random*() \* (maxArrTime - minArrTime));

*sleep*(arr);

//Add him to the shortest queue

Customer c = **new** Customer(++ID, **new** Date(), **new** Date());

m = minIndex();

**if** (!plzStahp)

System.***out***.println("Customer " + Long.*toString*(ID) + " added to

queue: " + Integer.*toString*(m + 1));

queue[m].addCustomer(c);

}

} **catch** (InterruptedException e) {

System.***out***.println(e.toString());

}

}

### The View

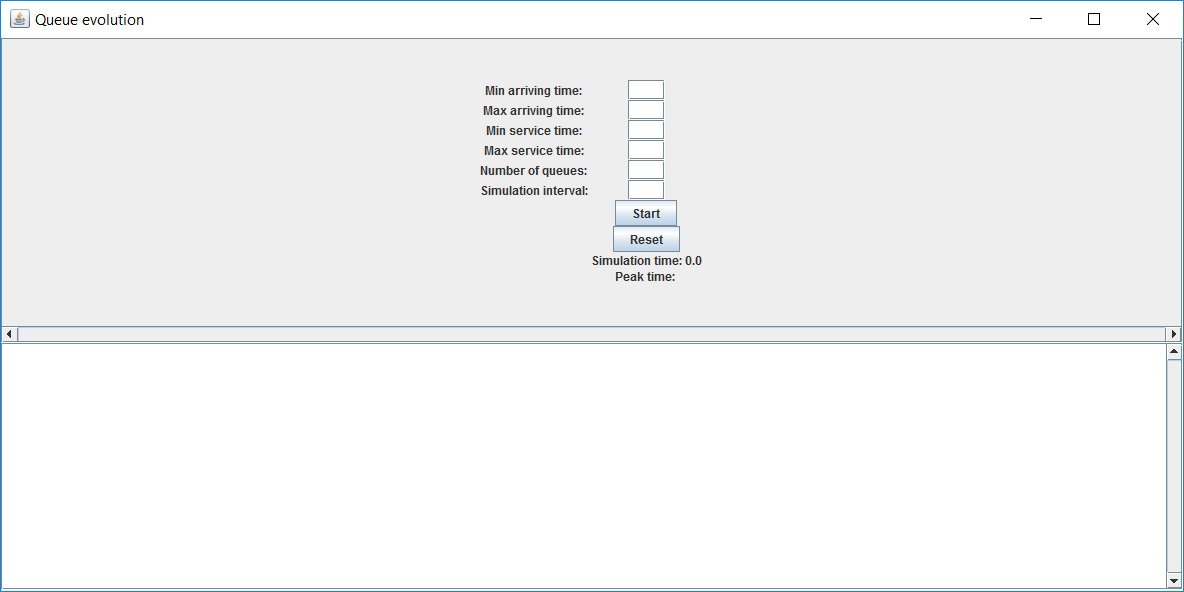
#### Class UserInterface

It contains the whole implementation of the graphical user interface, it extends JFrame and it has a list of components from which the constructor ‘constructs’ the user interface.

As it is constantly changing in time, when the application is opened, not all data is displayed at once. The boxes for the user input data are displayed and below this, the empty log of events. After correct data was input, and the user pressed the start button, the requested number of queues appear, their data and evolution is displayed, while events are listed in the log.

To be able to display queues of preferred number, in the upper panel a vertical scrollbar was added, while the log serves with a vertical one. These but become usable only if the size of the panel exceeds the one of the main frame, for example when many queues were added, or many events occurred already and the log listed more that can fit into its initial height.

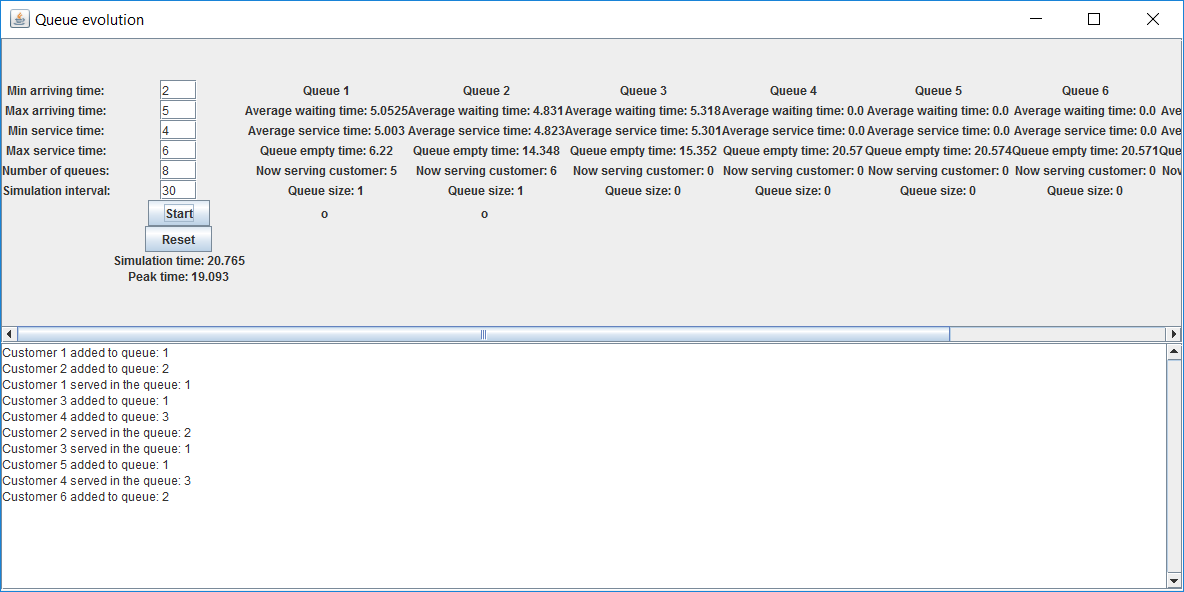
The initial appearance:



To display dynamic data, the labels are constantly refreshed from within the ViewController class in the Controller package. The header of the method from UserInterface class which is called to refresh queue data labels:

**public** **void** drawQueue(**int** nr, **float** avgWait,**float** avgServ, **float** emptyTime, **int** cust,**int** size)

Appearance while running:



To be able to display the events, the standard output was redirected into the white TextArea that can be seen on the lower part of the interface.

The class defines two methods for listening to the two functional buttons of the interface:

**public** **void** addStartListener(ActionListener start) {

button.addActionListener(start);

}

**public** **void** addResetListener(ActionListener reset) {

buttonR.addActionListener(reset);

}

Besides these the class only holds getters and setters for some labels, and a method which can be called to pop up a separate Error message window, used to announce the user that bad data was input.

### The Controller

#### Class LogWriter

This class extends the OutputStream and was created to be able to redirect the system output stream to the graphical user interface. It has a specific TextArea component, and this is passed to it when an instance of this class is created. It overrides the method **public** **void** write(**int** b) throughout which every byte of System.out commands is appended to the specified text area.

#### Class Control

The “brain” of the whole application, it links the interface and the entire model, parses input data, instantiates and starts threads. It implements two inner classes which both extend ActionListener, for the two functional buttons of the system. In its constructor it instantiates these classes onto the buttons.

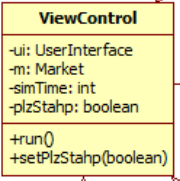
##### Class StartApp

Once the button was pressed, the execution starts according to the commands described within this class. The input data are parsed, and in case an exception occurs, it is properly handled: a popup window shows up with the error message. If the input data was valid, the queues are created accordingly and started one by one, and based on hem the market is also constructed and its thread execution starts. Besides these, the thread of the view controller is started from here. These are are contained in the implemented method actionPerformed from the interface ActionListener.

##### Class ResetApp

In order to reset the application, all the threads should be gracefully shut down, by setting their control signal plzStahp value to true. Thus, whenever the Reset button is pressed, all queues are stopped in order, then the market is shut down, and lastly, the view controller.

#### Class ViewControl

By extending the Thread class, it is also a dynamically evaluating class, its run method works like the following:

* This is the run method which is depending on the simulation time and it runs until it is not over.
* Variable elapsedTime counts the current time, startTime holds the moment of the start of the execution, and peakTime is used to memorize the moment when the market had the largest number of customers.
* While it runs, it constantly refreshes the user interface according to the data which may have changed over time.
* This refresh rate is 100 ms, because we put the thread to sleep 100 ms in each iteration.

@Override

**public** **void** run(){

**long** startTime = System.*currentTimeMillis*();

**long** elapsedTime = 0L;

**long** peakTime = 0L;

//While simulation time is not over yet

**while** (elapsedTime < simTime && !plzStahp)

{

**int** nrQueues = m.getNrQueues();

Queue[] queues = m.getQueue();

**int** sumMax = m.sumCustomers();

**int** i;

**try** {

//Refresh the data on the queues

**for** (i = 0; i < nrQueues; i++)

ui.drawQueue(queues[i].getID(), queues[i].getAvgWait(),

queues[i].getAvgServ(), queues[i].getEmptyTime(),

queues[i].getCurrCust(), queues[i].queueLength());

*sleep*(100);

**if** (m.sumCustomers() > sumMax){

peakTime = elapsedTime;

sumMax = m.sumCustomers();

}

} **catch** (InterruptedException e) {

System.***out***.println(e.toString());

} **catch** (Exception e){}

elapsedTime = (**new** Date()).getTime() - startTime;

ui.setPeak((**float**) peakTime / 1000);

ui.setCounter((**float**) elapsedTime / 1000);

}

//Stop threads

Queue[] queues = m.getQueue();

**for** (**int** i = 0; i < m.getNrQueues(); i++)

{

queues[i].setPlzStahp(**true**);

}

m.setPlzStahp(**true**);

m = **null**;

} }

### The Main Class

It contains only the method main, indispensable to be able to run the application:

**public** **static** **void** main(String[] args) {

UserInterface ui = **new** UserInterface();

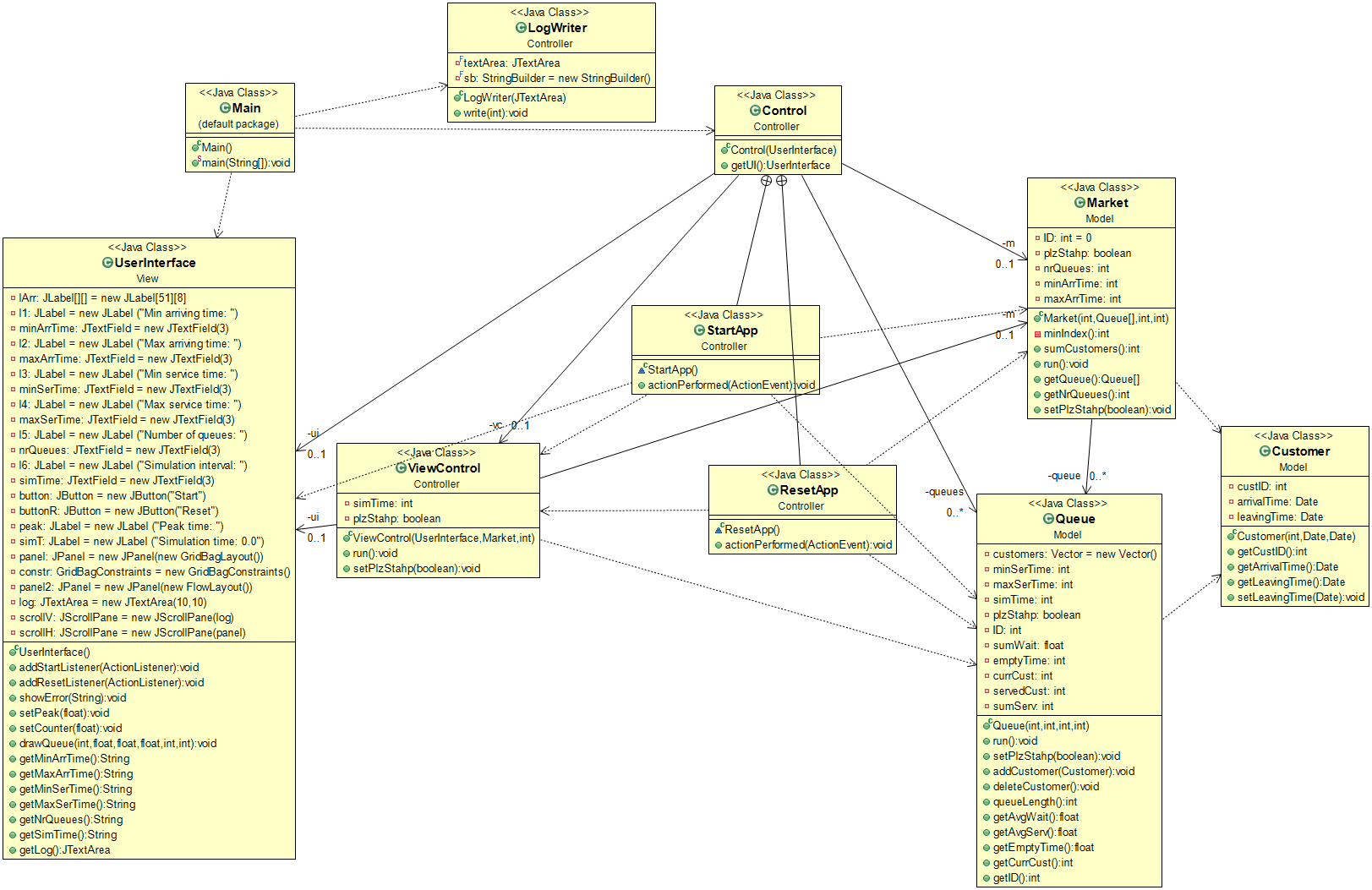
//redirecting the system output to the user interface

System.*setOut*(**new** PrintStream(**new** LogWriter(ui.getLog())));

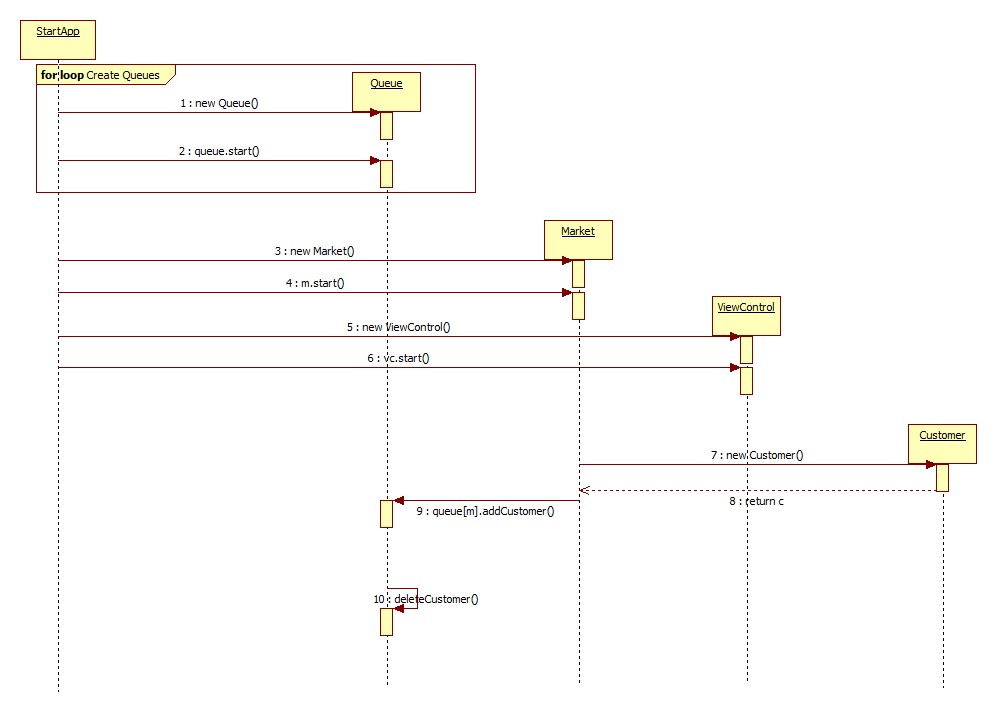
Control ctrl = **new** Control(ui);

ui.setVisible(**true**);

}



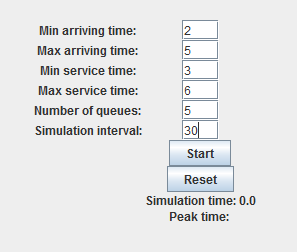
## Sequence diagram

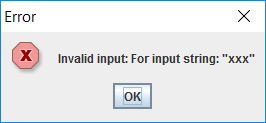
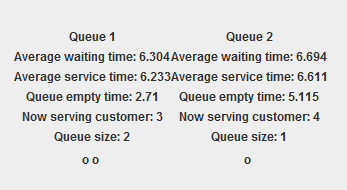


As it is illustrated, when an instance of class StartApp was created, it creates a chain reaction and all the components of the system are instantiated accordingly.

## Graphical User Interface

Numerous references were already made to the graphical user interface, and as already shown, it has a concrete and pleasant style with common and easy to use elements as the input boxes and the scrollbars. Each required input is labeled, and an example of inputs could be:

These inputs mean that customers will arrive in the market periodically in an arrival time bigger than 2 seconds and smaller than 5, will be served in the queue (when it is their turn) in a time greater than 3 seconds and smaller than 6. 5 queues will be displayed and simulated for 30 seconds of simulation time. Whenever these inputs do not conform, for example, non-numeric characters are input or min > max, an error pop-up window appears:



The queues are represented by their data and also, a little animation is made below them to represent the customers standing in line at them.

# Implementation and testing

Implementation started from modeling the problem and drawing an approximate of the class diagram, from where I first implemented the structure of the Model package. When it was done, I wrote the user interface so I could build on it, display and verify my results.

During implementation I have verified the correctness of my code by analyzing the log events that I have displayed in the console.

After my Model and my View were implemented, I connected them by creating the Control class into its corresponding package. Later on I had to figure out how to solve the problem with the dynamic user interface, thus I added the ViewControl class. My final steps consisted of redirecting the standard output onto the user interface and implementing the reset button.

Whenever an unexpected error occurred, I have used Eclipse’s debugger to find the source of the problem and correct it.

# Results

The development of this assignment resulted in a decent, user friendly application that illustrates queue evolution in a very simple manner. Not only can the user configure the input data of this system, but all relevant information is retrieved regarding the simulation, and if by mistake wrong inputs were given, the application can be reset by clicking a single button.

# Conclusions

## What have I learned?

With this assignment I have certainly learned the usage of threads, which I have failed to understand before. By developing the application, I have also enhanced my skills in Java programming language and in developing a proper Model-View-Controller based application.

## Further developments

This application can firstly serve for tracking and analyzing random data, which can be used to calculate probabilities and such. Further developments may include adding more constraints to the evolution in time of the system or calculating and displaying more information about the queues. A more sophisticated and colorful animation of the arriving and leaving customers could also be added in the future.

# Bibliography

1. For programming issues: <https://docs.oracle.com/javase/7/docs/api/overview-summary.html>
2. This pdf: <http://inf.ucv.ro/documents/tudori/laborator8_53.pdf>
3. Class diagram description: <https://en.wikipedia.org/wiki/Class_diagram>
4. Last semester’s OOP code on how to use Swing to create user interface and how to establish the connections in an MVC model