MINISTERUL EDUCAȚIEI NAȚIONALE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Assignment 2
Queue Simulation
- Documentation –

10.04.2018

Linca Paul Tudor

Group: 30422

1. Task

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

2. Brief description:

This assignment consists of developing an application which simulates a grocery store environment. It is mandatory that the app receive as input the arrival interval of time of the customers, the service interval of time for each customer, the maximum time the simulation should run for and the number of queues, i.e. the number of open cash registers the store should have. I also decided to add as an extra input the maximum amount of customers the store shall have during the simulation time for easier manipulation of the data.

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.

An extra requirement for the assignment is that a log of events containing simulation results should be computed during the simulation and displayed at the end of it.

With all this in mind, the assignment simply comes down to:

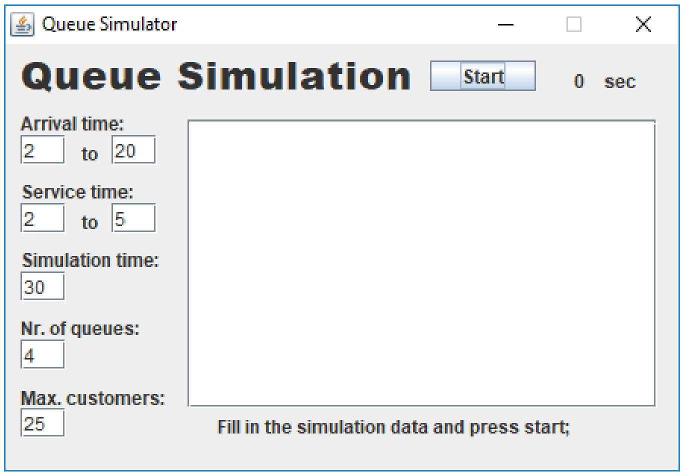
- Generating random customers at random times (although the times should be in conformity with the input);
- Generating a given amount of queues;
- Developing a way of distributing the clients to the queues;
- Documenting each step the simulation takes;
- Develop a timer system to easily keep track of the simulation;
- Multithreading;

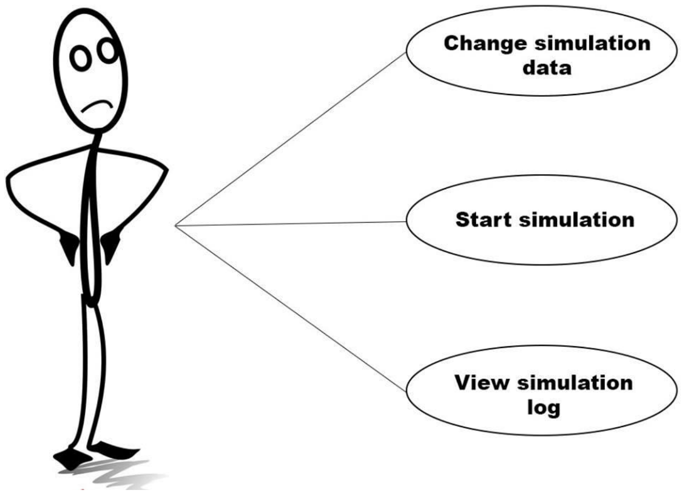
3. Queues

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.

I decided that my strategy would be to place each new customer to the queue with the least amount of customers already in it. I am aware that this may not be the most efficient way. A better approach would be to place each new customer to the queue with the least total service time but unfortunately, it is impossible to calculate that in the real world because a lot of foreign factors could intervene.

4. User Diagram

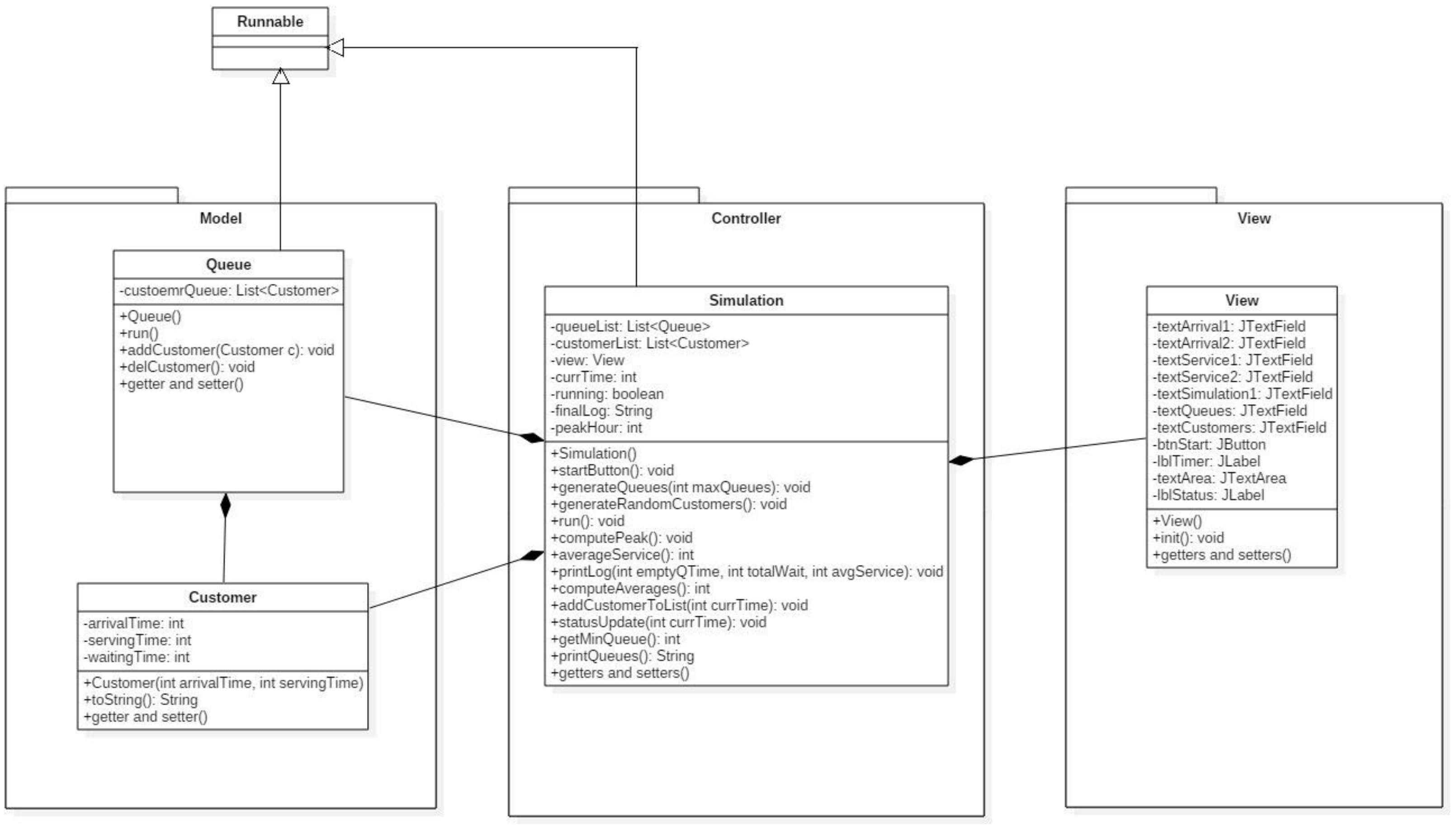




The user is presented with a simple and smooth graphical interface. It consists of 7 text fields in which the user will insert the required simulation data (the text fields are autocompleted at first for easier and faster testing on the same input), and a start button which will kickstart the simulation. There are numerous labels to help the user understand the GUI.

In my opinion, the graphical user interface is pretty straight forward in how the user is supposed to use the system.

5. Class diagram



A class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

I designed the problem based on the Model – Controller – View (MVC) architectural pattern. It is commonly used for developing user interfaces that divides an application into three interconnected parts. This is done to separate internal representations of information from the ways information is presented to and accepted from the user. The MVC design pattern decouples these major components allowing for efficient code reuse and parallel development.

- The model is the central component of the pattern. It expresses the application's behavior in terms of the problem domain, independent of the user interface. It directly manages the data, logic and rules of the application. Thus, my model consists of two classes: the Queue class and the Customer class.
- The view consists of the Graphical User Interface (GUI) which will be described later in this document.
- The controller accepts input and converts it to commands for the model or view. So it basically links the model and the view.

I chose the MVC architectural pattern because it enables the logical grouping of related actions on a controller together, it creates components that are independent of one another, thus enabling the reusability of the code, it has low coupling meaning that its components has, or makes use of, little or no knowledge of the definitions of other separate components and it is good practice for future group projects I will be a part of because multiple developers will be able to work simultaneously on the mode, controller and views. Also, even though I have not used it for this project, the MVC architecture can have multiple views which can be very useful in a lot of projects.

Although it has many advantages, I noticed that the MVC has disadvantages too. The framework navigation can be complex because it introduces new layers of abstraction and requires users to adapt to the decomposition criteria. It also has a pronounced learning curve although I would consider this useful for developing my skills as a programmer.

6. Class description

- ### View (GUI)

This class represents the GUI (Graphical User Interface). The private attributes are the GUI's elements which are needed for the correct execution of the simulation. They also have the getters and setters needed to perform the required task.

The init() method is a pretty straight forward window building methid. It uses JTextFields, JButtons, JLabels, a JFrame, a JScrollPane and a JTextArea.

- The input text fields expect to receive integers that would correspond the labels just above them.
- The label near the title represents the timer which will increment every second.
- The output will be displayed on the scroll pane that dominates almost the entire window. The scroll pane has a text area inside of it, thus even if a string

larger than the available space is to be displayed, the user can still access every part of it by scrolling.

The constructor of the View class simply calls the init() method.

This class is contained in a package with the same name.

- Customer

This class, as the name would suggest, represents each customer. It has three private attributes: arrivalTime, servingTime and waitingTime. The names of the attributes are pretty descriptive so a detailed explanation is not necessary for them. The waitingTime is initialized with 0 so that it can be incremented every second the customer is in the queue.

In the class constructor the arrival and serving time are attributed the values given in the arguments.

The most important method of this class is the toString() method. It prints the customer with the following format: "C(arrivalTime, servingTime) -". The '-' is just a delimiter between the customers.

This class is contained in the model package.

- Queue

This class, is the actual queue. It has a single argument, customerQueue, which is a list of Customer entities. Thus, just like in real life, a queue is made up of a collection of customers. In the class constructor the customerQueue is initialized as an ArrayList.

The Queue class implements the Runnable interface so it can create a thread. Runnable abstracts a unit of executable code. One can construct a thread on any object that implements Runnable. To implement Runnable, a class need only implement a single method called run( ). Inside run( ), we will define the code that constitutes the new thread.

The run() method builds a loop that will decrement every second the servingTime of the first customer in the customerQueue list. For the program to wait a single second between decrements, the sleep() method is called and 1000 (miliseconds) will be the argument. If the servingTime of the customer reaches 0 then the customer will be deleted using the delCustomer() method.

The methods delCustomer() and addCustomer() are simple methods that add and delete a customer from the customerQueue respectively.

The last methods implemented are the getter and the setter for the customerQueue.

This class is contained in the model package.

- Simulation

The Simulation class is the controller of the application. This class accepts input and converts it to commands for the model or view. It has as attributes a list of Queues, a list of Customers, a View, an integer representing the current time of the simulation, a boolean variable, a string in which the final log will be built in and a int array of two values.

- QueueList – is an ArrayList containing the generated queues

- customerList – is an ArrayList containing all the customers that will be simulated
- View – is an instance of the View class, thus building the application window
- currTime – is a variable that keeps track of the current time in seconds of the simulation
- running – a Boolean variable that tells if the simulation is running or not
- finalLog – in this string the final log containing the simulation data which will be built during the simulation
- peakHour – an int array that has on it's [0] position the time in which the peak hour happens and on it's [1] the number of customers at that peak hour.

The model and the view are linked through the controller by using a multitude of methods.

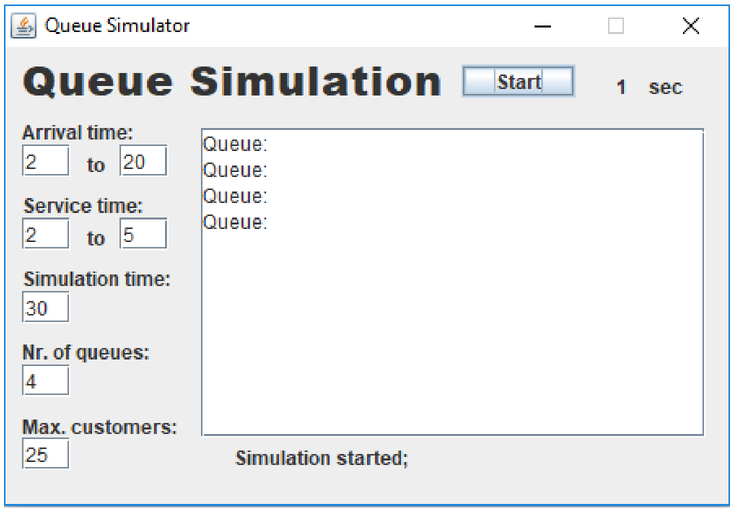
- The constructor
  - ~ In the constructor the two ArrayLists containing the queues and the customers are initialized and the startButton() method aswell.
- startButton()
  - ~ this method implements an action listener for the start button in the view class. Thus, if the button is pressed the simulation will start: the queue and customerList will be filled, the queue threads will be started and the running attribute will be set to 'true'.
- generateQueues(int maxQueues)
  - ~ a simple method which generates maxQueues queues and adds them to the queueList

- generateRandomCustomers()
  - ~ this method generates a random number of customers between 1 and the number of customers that were inputted in the application window. For each customer a random arrival time and a random service time will be computed. Finally, the customers are added to the customerList.
- run()
  - ~ the run method is the fundamental methods for creating a thread. Firstly, a continuous loop is created. This loop keeps running infinitely until the start button is pressed. When the start button is pressed the program accesses a secondary loop which runs the actual simulation. The simulation works like this:
    - the addCustomerToList() is called
    - the currTime is incremented
    - the average is computed using the computeAverages(…) method
    - the peakTime is computed by using the computePeak() method
    - the thread waits 1 second (1000 miliseconds)
    - the process is repeated until the maximum time for the simulation is reached
    - after that threshold is reached, the loop is closed and printLog() is called
- computePeak()
  - ~ this is a simple method for finding a maximum. The list of queues and customers are iterated and the customers at the currTime are counted. If the amount of customer is the

largest up until that point the peakHour array is updated accordingly.

- averageService()
  - ~ this method computes the average service time of all the customers
- computeAverages(int totalWait, int emptyQTime)
  - ~ this method computes the total wait time of the customers and the total time the queues are empty
- addCustomerToList(int currTime)
  - ~ this method simply adds a customer from the customerList to the queueList if their arrivalTime is matches the current time currTime and updates the status of the application by using the statusUpdate(…) method.
- statusUpdate(int currTime)
  - ~ this method updates the final log and the status label of the view depending on the input data.
- getMinQueue()
  - ~ this simply returns the index of the queue that has the minimum amount of customers
- printQueues()
  - ~ this builds a string which contains all the queues and their respective customers and then it returns it
- getters and setters

7. Results


Queue Simulator

Queue Simulation Start 1 sec

Arrival time: 2 to 20

Service time: 2 to 5

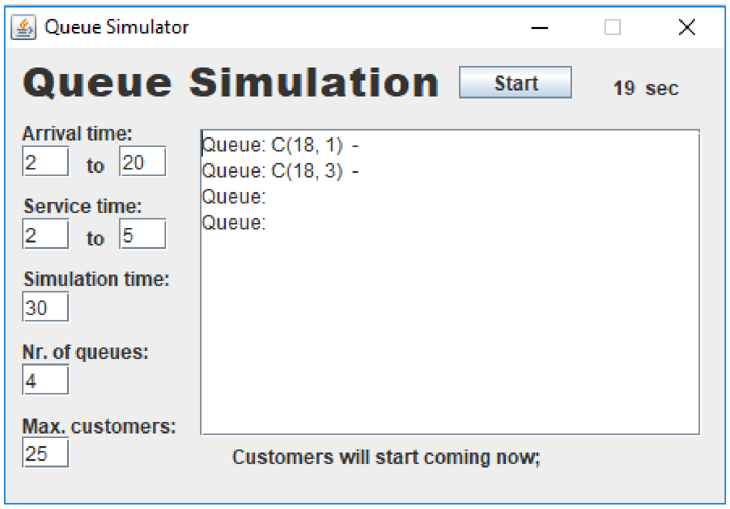
Simulation time: 30

Nr. of queues: 4

Max. customers: 25

```
Queue:
Queue:
Queue:
Queue:
```

Simulation started;


Queue Simulator

Queue Simulation Start 19 sec

Arrival time: 2 to 20

Service time: 2 to 5

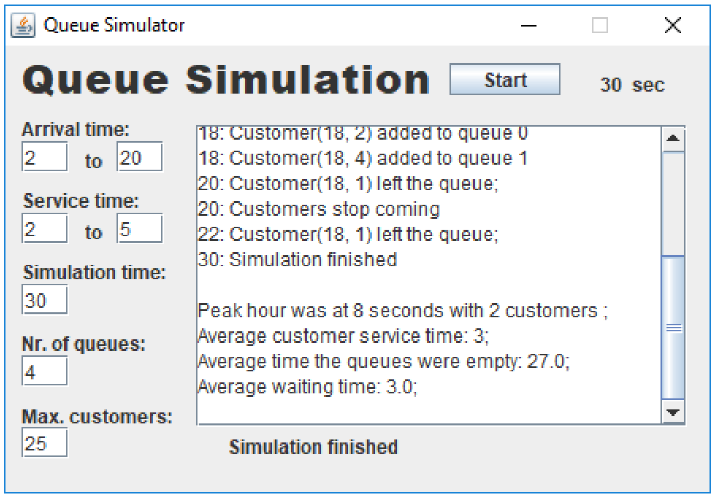
Simulation time: 30

Nr. of queues: 4

Max. customers: 25

```
Queue: C(18, 1) -
Queue: C(18, 3) -
Queue:
Queue:
```

Customers will start coming now;


Queue Simulator

Queue Simulation Start 30 sec

Arrival time: 2 to 20

Service time: 2 to 5

Simulation time: 30

Nr. of queues: 4

Max. customers: 25

```
18: Customer(18, 2) added to queue 0
18: Customer(18, 4) added to queue 1
20: Customer(18, 1) left the queue;
20: Customers stop coming
22: Customer(18, 1) left the queue;
30: Simulation finished

Peak hour was at 8 seconds with 2 customers ;
Average customer service time: 3;
Average time the queues were empty: 27.0;
Average waiting time: 3.0;
```

Simulation finished

It can be observed that the results are correct based on the input data.

8. Conclusion

In my opinion, this second project was chosen very wisely. It is a very good way of polishing our OOP skills and, at least for me, a good project for learning the basics of working with threads and multithreading .

Further improvements to the programs would be:

- ~ Implement a better method for choosing on which queue a customer should be added. An idea was mentioned in section 3 of this document.
- ~ Use less methods in the Simulation class
- ~ Have the Simulation class have less attributes
- ~ Use another class to manage the models

9. Bibliography

https://en.wikipedia.org

http://www.coned.utcluj.ro/~salomie/PT_Lic/

https://softwareengineering.stackexchange.com

https://www.google.ro