Assignment2- Queue Simulator

# 1.Objectives

The purpose if this assignment was to write an application that can efficiently and accurately simulate the real-life working of a shopping mall-like environment, the queuing phase of it by analyzing and minimizing the queuing time of clients.

In order to implement the functionalities of such a system, the problem has to be decomposed into many sub-problems which have to be reunited to form the whole solution.

# 2.Analyzing the problem

Queues are frequently reappearing in our daily lives and we can efficiently model many aspects of the real world by them ( such as a shopping line ). The main functionality of it is to provide a “space” for an item (in our case a client) , place it at the end of the item list ( in our case the shopping line ) , store it for a period of time until it reaches the head of the line ( in our case it becomes the current client ) and remove it ( when the client is served, he/she leaves the queue ). Consequently is quite straightforward to use queues for simulating the functionality of a shopping line/shopping queue.

Since even in real life is quite an unpleasant aspect to queue up for a very long period of time, in this application we also have to take into account the fact, that this whole functionality has to be optimized. One way to do that is by adding queues to the system each time it is necessary (when the “population” of the shop reaches a pre-defined maximum). At each “addition” event the clients are equally distributed by the system ( just like in the real world – when a new queue is opened at any market, some of the clients from other queues change their minds and go to the previously opened one ) . After this redistribution the whole system continues its functioning just like previously.

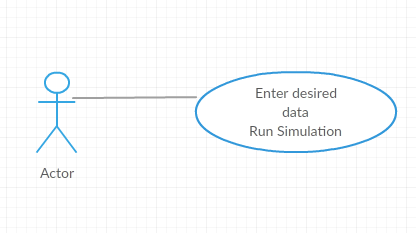
Another aspect here is to concurrently simulate the working of each queue. This could be done by the usage of multithreading ( for each queue -in case there are more than one – we start a new thread, each of them implementing the same functionality but working separately, independently from the others) . To obtain the desired behavior of the system, these threads have to be synchronized somehow, so there will be no noticeable delay between any of them.

Finally, what we should also take into account is the way the clients are generated. Since it would be extremely inefficient and not user-friendly at all to ask the user to generate during the simulation time a whole sequence of clients that are going to be placed in the queues, it is the developer’s responsibility to generate them. For this an algorithm has to be defined. It will generate a series of clients based on some user-defined data ( simulation time, minimum arrival time, maximum arrival time, minimum service time, maximum service time ) , having some random values as name, arrival time and service time.

# 3.Modelling

As mentioned before, the most important concept on which my solution of the problem is based is the functioning of a queue. In case of this system we are talking about multiple queues, each of them having a list of clients. Respecting the OOP paradigm, each component should enclose its properties ( incapsulation ). When working with clients, what we should consider in this case is the arrival time and the service time of them ( the waiting time being determined is dependent on other circumstances ), while in a case of a queue what is the most important is the list of the clients it holds. One level above these two is the shop which holds the queues, so these above mentioned classes ( Client, ShoppingLine, Shop ) are the basic elements of the system. Working with queues and simulating a real-life-like scenario is a more complex task. To perform it we have to build a whole logic above the mentioned base, which is mainly dealing with multiple threads and “transferring” data to/between them.

# 4.Use cases

The user only has to perform a given sequence of operations in order to reach the desired functionality. If entering the input data in a correct way ( corresponding the expectations ) the system should work correctly, so basically if he/she entered all the required data (simulation time, min. and max. arrival time, min. and max. service time) there is nothing else to do, just wait the end of the simulation to display the analyzation data.

# 5.Scenarios – Enter desired data, Run simulation

## Success Scenario

1. The user runs the application.
2. The graphical user interface is displayed and shows where to enter the different parameters of the simulation (simulation time, min. and max. arrival time, min. and max. service time).
3. If all the data is correct (each of the inputs is a number, the min. values are smaller than the max. values), the user presses the “Start simulation” button and he/she can follow the evaluation of the system ( clients arriving/leaving, new queues being added ) and the “real time” of the simulation Is constantly displayed.
4. The simulation is complete when a message dialog widow is displayed showing the final results of the analyzation, meaning the maximum waiting time, minimum waiting time, average waiting time, peak time start, peak time finish and the maximum number of clients during the peak “hours”.

## Alternative Scenario

In the previously described scenario there may be some modifications if the user introduces some incorrect data

5.The user introduces some value as parameter which is not a number.

6.A message dialog window is displayed after pressing the “Start simulation” button, that some of the parameters ( namely telling the user ) is not a number. In this case we skip back to the 3rd step.

7.The user introduces a min. parameter greater that the max. parameter ( in case of arrival time and service time ).

8.He/she is told that something went wrong ( displaying a message in a dialog window ) by namely telling the case too. In this case we skip back to the 3rd step.

## Error Sequence

In case of this application all the unexpected events that can happen are mentioned in the previous part ( Alternative Scenario ) and they are also handled, so hopefully nothing else can go wrong.

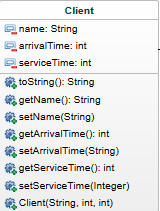
# 6.Design

The system implementation is composed of a Model, View, Controller combined in the MVC Class.

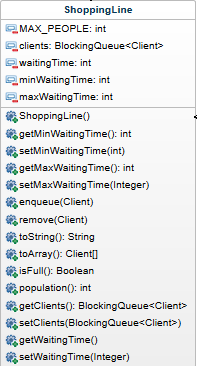
## Model

The model is built of numerous classes, since this one is that module which implements the complex logic of the whole queuing system.

### Client class

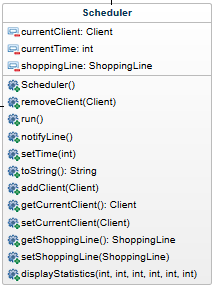
The client class basically implements the functionality of a single client. Each client has a name, an arrival time ( the moment it arrives to a queue ) and a service time ( the time needed to leave the queue after it becomes the current client in a queue ). The methods corresponding to this class are mainly **getters and setters**. The class also overrides the **toString()** method, so the attributes of each client will have a printable form in a single string.

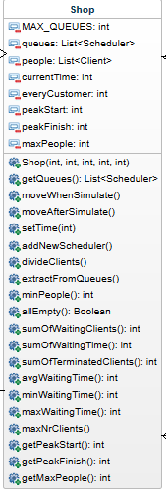
### ShoppingLine class

The ShoppingLine class implements the functionality of a single queue. One of its attributes is a **BlockingQueue** of Clients (I used a BlockingQueue, because it was easier to work with them when many threads are running and their behavior has to be synchronized – this way I don’t have to deal with those special cases when some of the queues are empty and some are not, the BlockingQueue just doesn’t take any client as current client when it is empty, just “waits” for the simulation to terminate), whose size has to be smaller than the **MAX\_PEOPLE** ( in case there are available ques to add to the system). Another attribute is the **waitingTime** which refers to the sum of all the customers’ waiting time that were served at the respective queue. It also holds the **minWaitingTime** and the **maxWaitingTime** ( the minimum and maximum waiting time out of all the clients’ waiting times ).

This class includes a series of getters and setters for its attributes, an **enque()** method ( for adding clients to the queue ), a **remove()** method ( used when redistributing the clients in case of addition of a queue to the system – in this case we have to take the clients from the end of the queues ), a **toString()** method ( basically “adds up” all the client strings ); **isFull ()** and **population()** both refer to the population of a queue (isFull() tells if the number of people in a queue reached the maximum number or not while population returns the number of clients in a queue). The last method to mention is the **toArray()** method, which converts the blocking queue into an array of type Client, making it possible to have a sequential access to the elements of the queue.

### Scheduler class

The Scheduler class’ most important “characteristic” is probably that it implements the **Runnable** interface so it is necessary to implement even the unimplemented **run()** method of it (when the thread is constructed this method is called). What it basically does is to take the first element (client) from the queue as currentClient. While the currentTime is smaller than the moment the client became the current client + the client’s service time, in each cycle it waits to be notified. After being notified by calling the **notifyLine()** method and synchronizing its current time by the **setTime()** method, it verifies whether the current time exceeded the mentioned value or not. If not it waits again and the whole “procedure” is repeated until the current time reaches the addition + service time, the client’s waiting time ( the time spent in the queue from being added to it until leaving the queue) is added to the queues waiting time and if the queue is not empty, the next clients is set as current client. This class has as its attributes the current client, the current time and a shopping line, on which the thread “operates”.

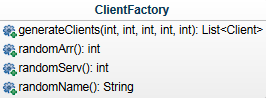


### Shop class

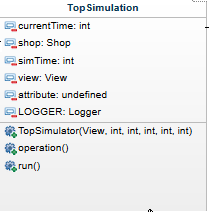
The shop class hold as its attributes a list of queues, whose size cannot exceed the maximum number of available queues ( MAX\_QUEUES=4 ), a list of pre-genereated clients with random data, the current time, the total number of customers, the start of the peak hour, the end of the peak hour and the number of people during that period. Here the two methods that have to be mentioned at first are the **moveWhenSimulate()** and the **moveAfterSimulate().** These two implement the base of the system’s behavior during simulation time. moveWhenSimulate() is used when the current time didn’t exceed the simulation time set by the user. While this condition is satisfied, after the time is synchronized for each active queue and they are all notified, it is evaluated whether the active queues are full or not; if they are and the number of active queues is under the allowed maximum, a new queue is activated by creating a new thread and starting it. After this, the clients that are queuing up are evenly distributed between the active ques and the simulation continues by checking if in the pre-generated client list the first one has to be added or not (its arrival time is equal with the current time). If this is the case, we are looking for the queue with the minimum number of clients and the client is added to that one. If there are active queues that are not full, the queue addition part is left out the others remain the same.

If the current time exceeds the simulation time, it is likely that there will still be non-empty queues which should not be left in this state. In this case the method moveAfterSimulate() is called which only sets the time and notifies the queues, but no extra queue is added.

The **divideClients()** and **extractFromQueues()** methods take part in the client redistribution; **minPeople()** returns the index of the queue with the least clients; **allEmpty()** returns true if all the queues are empty; the rest of the methods returns integer values referring to the aspect being analyzed (waiting clients, waiting time, terminated clients, average waiting time, minimum waiting time, maximum waiting time, maximum number of clients at a time, peak hour start, peak hour end).

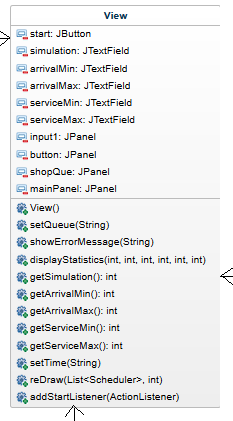
What is also really important to mention here is the way the clients are generated at the beginning of the simulation. For this I use the **ClientFactory** class, which has 4 methods. **randomArr()** generates an arrival time between the given parameters, taking into account the simulation time and the arrival time of the previous client. **randomServ()** generates a service time between the given parameters and **randomName()** generates a random string. generateClients() returns a list of clients with random attributes.

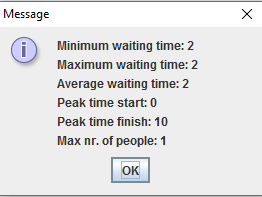
### TopSimulator class

The TopSimulator class implements the **Runnable** interface, consequently it was necessary to implement also the **run()** method. This is the top level of the Model. What it does is to call the Shop’s moveWhenSimulate() or moveAfterSimulate() method ,depending on the current time and the state of the queues; redraws the graphical user interface’s view corresponding to the state of the queues and the current time, continuously logs the current average waiting time and the thread **sleeps for a second** in each cycle ; at the end displays in a message dialog block the result of the analysis (minimum waiting time, maximum waiting time, average waiting time, peak hour start, peak hour end and maximum number of clients at that time).

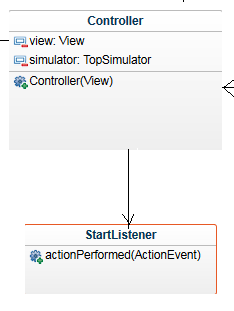
## View

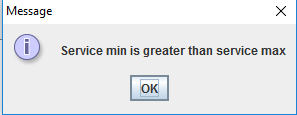
The View class implements **JFrame** interface and it is composed of a series of component. First of all, there is a series of **text fields** in which the user can enter the preferred data (simulation time, minimum arrival period, maximum arrival period, minimum service time, maximum service time). There is also a **button,** by pressing it the simulation begins. Under the button the **real time** of the simulation appears in a label. The last part of the view is the section (panel) kept for displaying the queue evaluation, each queue is held by a **scroll panel.**

Besides the getters of the fields and the setter of the queue panel there is a method called **reDraw().** This is the one called by the TopSimulator when displaying the current state of the queues. What it basically does is to remove every content of the queue panel, traverses the list of queues, creates a new scroll panel for each of them and adds it to the queue panel, containing the string referring to the respective queue. The **displayStatistics()** method is called at the end of the simulation, in a message dialog block it displays the analysis results.



## Controller

The Controller class combines the TopController and the View. Inside this class another one is implemented, namely the **StartListener,** which has one single method. Basically it implements the listener for the “Start simulation” button from the view. When the button is pressed, it is verified whether the received valid ( have to be integer numbers ) inputs satisfy further conditions, meaning the minimum value has to be smaller than the maximum value both for the arrival and service time.



If no such error appeared, then a new TopSimulator object is instantiated and the thread of it is constructed, the simulation starts.

## Additional Information

Another aspect here that may be mentioned is the way I organized the mentioned classes in different packages, having three in total.

* Factories: it holds the ClientFactory class;
* Model: it holds the classes that model the behavior of a shop – Client, ShoppingLine, Scheduler, Shop, TopSimulator
* GUI: it holds the rest of the classes that are part of the graphical user interface – View, Controller, MVC

# 7.Implementation and testing

## Client, ClientFactory

The first class that I implemented was the Client, since the further levels cannot be developed without it. It was followed by the implementation of the ClientFactory class. Calling the generateClients() method for a several times with a diversity of input parameters I checked whether the results returned (the returned Client list) was correct or not.

## ShoppingLine, Scheduler, Shop, TopSimulator

After I was done testing the client generation, I started to implements the ShoppingLine class and basically built the other classes one above the other, reaching the top level represented by the TopSimulator. First the methods were only written to simulate the behavior of a shop with the queues and clients, without analysis. To test if the whole until this point was working or not, I also had to implement the graphical user interface. Running my application a few times I noticed which are those cases which may cause errors and which are those parts that doesn’t work correctly (I obtained these results by using the GUI and the console too by logging and printing at different points in the program) – one example for an incorrect functionality was the distribution of clients (it was corrected later, but at the beginning the first added client was held only for a second as the current client, because the time at that point was not yet synchronized).

When the model itself worked correctly, I started to add methods for analyzing the problem during simulation – I added methods for the calculation of the waiting time, peak time, served clients etc.

## GUI

When implementing the graphical user interface I was continuously testing to obtain a more or less user friendly solution.

# 8.Results

After many hours of coding and debugging I obtained a solution for the problem which allows the user to model a real-life aspect in smaller dimensions, namely the queuing in nearly every shop. Using this application he/she can not only observe the clients coming and leaving but also can analyze based on the provided data how much time does a client spend with queuing and which are those hours (seconds in our case), when the shop is the most crowded.

I consider that the application is quite straightforward for every user and it is not difficult at all to use it. After a few minutes anyone can easily familiarize himself/herself with it.

# 9.Conclusions

## What I learnt

This assignment was very useful from my point of view, since I have never worked with threads before. By solving the proposed problem I had to read about threads and use them in a way to complete the requied functionality. I helped me to clearify a little how the synchronization of them has to be obtained but also I had a little overview about them, where they are used and why they are really useful in different situations.

## Further implementations

There are several further possibilities to impove and develop this project. For example, the service time of each client could be calculated based on the number of items they would like to purchase or on the speed of the cashier. Also the number of queues could be increased. I would also add a functionality in order that the user to be able to close a queue in the middle of the simulation.

# 10.Bibliography

* Many questions answered - <http://stackoverflow.com/>
* Java reference - <https://docs.oracle.com/javase/8/>
* Class Diagram - <https://api.genmymodel.com/>
* Use case Diagram - <https://creately.com/>
* <https://www.tutorialspoint.com/java/>
* Threads - <https://www.caveofprogramming.com/java-multithreading/>