

**Programming Techniques**

**Queues Simulator**

*Course Teacher*: prof. Ioan Salomie

*Laboratory Assistant*: ing. Cristian Stan

*Student*: Ivan Alexandru-Ionut

Contents:

1. Assignment objective
2. Problem analysis, scenario, modeling, use-cases
3. Design
4. Implementation
5. Results
6. Conclusions
7. Bibliography
8. Assignment objective

The objective of this assignment was to develop a Queue Simulator that will take as input some parameters and simulate a real-world queue, where the queues are ‘servers’ and clients are ‘tasks’ and will manipulate the generated data to provide the statistics necessary. The secondary objectives were:

* To create an UI interface that the user can interact with
* To correctly parse the input data
* To randomly generate the clients
* To make use of multithreading: one thread per queue
* To use appropriate synchronized data structures
* To clearly make use of good OOP concepts

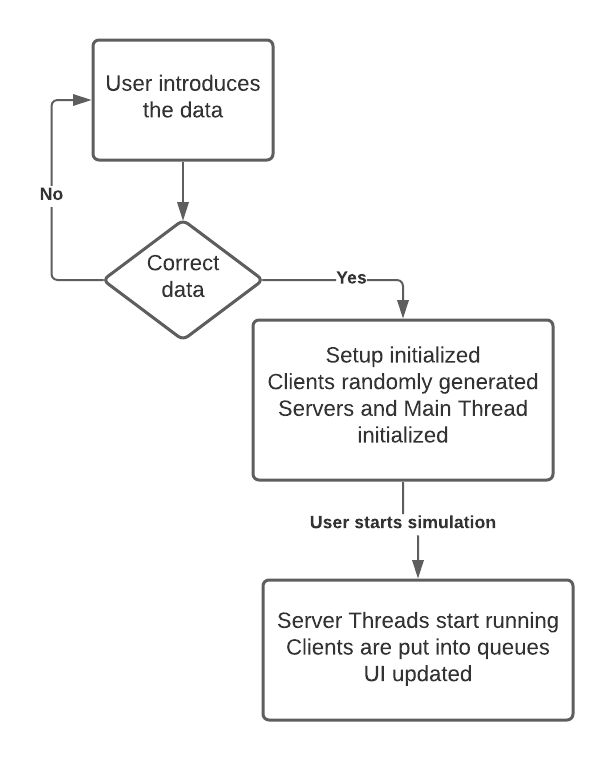
1. Problem analysis, modeling, scenario and use-cases
   1. Problem analysis

We are given a number of clients N, as well as a number of Queues/Servers Q, each client having the properties: ID, arrival time, processing time. The arrival time is the time when the client arrives at the queues, the processing time is the time needed for a client to be processed (how long he will stay at the queue), while the ID is the identification number.

We are required to manage the arrival and the processing of every client as they get to the queues and make use of multithreading – as we will use one thread per every queue and a main thread to manipulate them. As the clients arrive to the queues, they will automatically be put in the queue that has the current minimum waiting time, which is the time the client has to wait before it will be in the front of the queue.

As in real life, more clients can get to the queue at the same time and the differentiation method is the minimum ID will be the first one. Also, the queues are “open” only if there are clients in that queue and once a client is put into a queue, he cannot change it. But, due to the implementation, there cannot be a better possible queue a client can be changed to.

* 1. Modeling

We will use the properties of the Clients/Tasks mentioned above to create its class and we will use the Strategy stated in the Project Description. Every Server/Queue will have a queue, a waiting time and a thread that will be running. We will also use a Scheduler that will handle the starting of the servers, the Selection Policy and will also stop the servers. The Simulation Manager is the main Thread that will make everything work, from the Scheduler and the Servers to the Random Generation of the Clients and the updating of the UI interface.

* 1. Scenario and use-cases

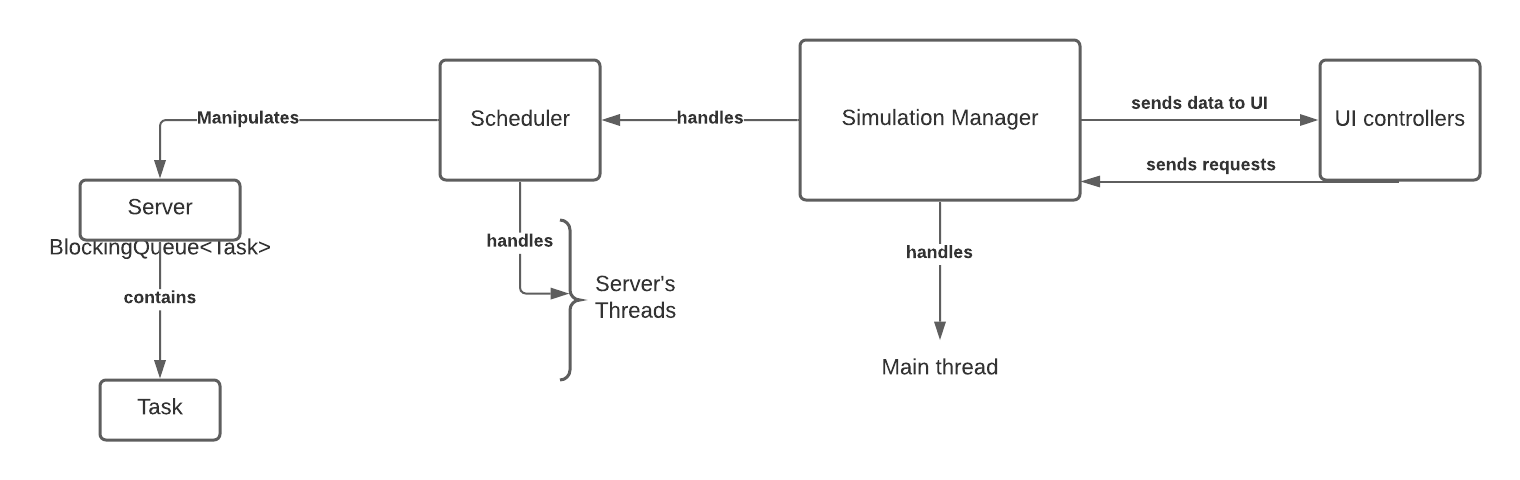
The following flow-chart resumes all the scenarios the user can go through.

The application is done in such a manner that if the input is wrong an error message will be displayed and the user has to introduce the correct input.

After the input is processed, the user is prompted with another window, where he has to press the Start button, which will start the simulation and therefore display the current data. After the simulation is finished, the statistics are displayed and all the Server’s threads are dead.

1. Design
   1. OOP design

Regarding the OOP design, the use of the Scheduler that manipulates the Server is the one that reflects it the most, as all the Servers/Queues are initiated in the Scheduler’s constructor and are managed by means of the Scheduler’s functions. In the main thread, the Simulation Managed, there is only one instance of a Scheduler, so the program is safe in terms of the creation and starting of the threads, as I assured that they are only created and started once. Moreover, in order to display the Tasks and Servers, I have overridden the toString method of the classes and changed them accordingly to my own needs. I have also used an Interface and an Enumerator, as well as using an MVC pattern for my application.

* 1.  Class diagram / Application diagram

Above is a simplified application diagram that shows all the relation between classes and how the application works. It is easier to see now hoe the Simulation Managed handles the Main thread as well as the other Threads by means of the Scheduler and how data is transmitter/requested from the UI.

We can also see the MVC pattern as the Model part is in the left side, the Controller in the middle and in the right side the UI controllers will send data to UI – to the View part of the Application.

* 1. Data Structures

Besides the simple data structures data I have used – integers, Booleans, floats or other primitives, I had to use thread-safe data structures and so the ones I used were BlockingQueue, AtomicBoolean and AtomicInteger and in this manner I was assured that my data will not be corrupted in some way by the calling of some functions from different threads.

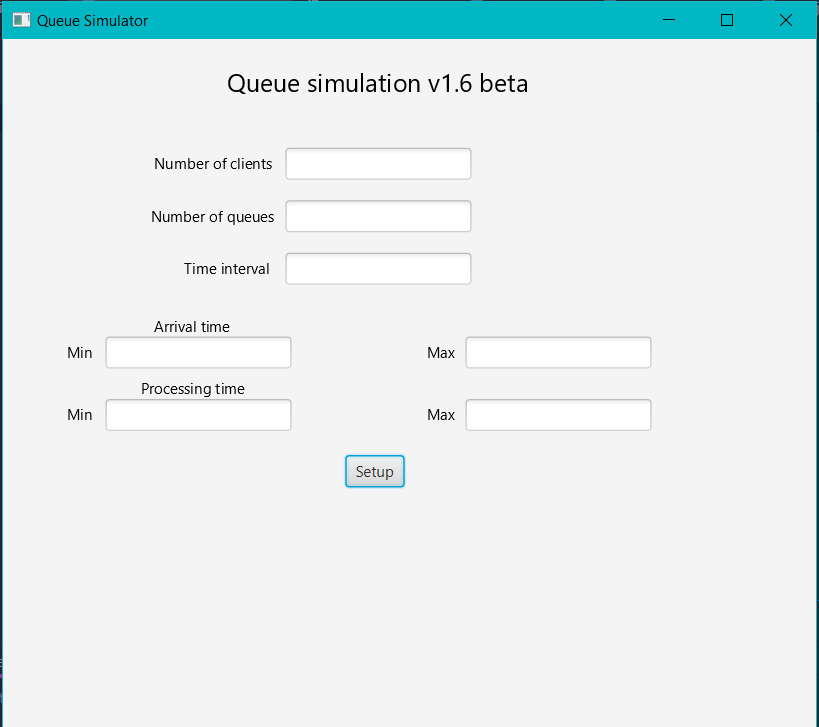
* 1. Class design

In this part I will briefly explain the class design regarding the division in the packages.

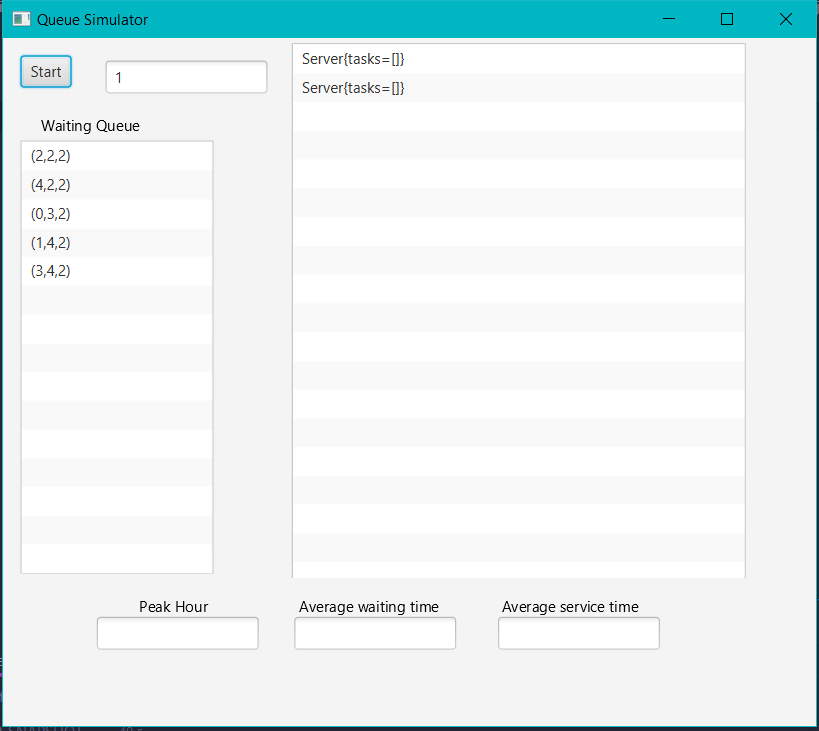
* **model**
  + - Task
      * This is simply the Task Model class
    - Strategy
      * The Strategy interface, which defines the Strategy used
* **controller**
  + - Scheduler
      * The Scheduler class, that manipulates the Server’s Threads
    - Server
      * The Server class
    - Simulation Manager
      * The Main Thread of the application, which controls everything, from Servers to UI
    - Main Menu & Simulation Controllers
      * These classes represent the controllers for the .fxml files
* **view**
  + - mainMenu.fxml and simulation.fxml
      * These are the UI files that the user will be able to see, they were created using SceneBuilder.

There also exists a **Main** class that contains the main function and it has the responsibility of starting the application.

* 1. UI part

On the left is a screenshot of the MainMenu controller, where the used can introduce the data for the queueSimulation. If the data introduced is not correct, and error message will be displayed on the screed.

After pressing the Setup button, the window from below will be displayed and the used simply has to press the “Start” button in order to start the simulation.



1. Implementation

* **Task**
  + This is simply the Task Model class, which contains the fields (the ID, the arriving time and the processing Time) and the toString method.
  + The **toString** *method* simply outputs all its fields.
* **Server**
  + The Server class has, other than the fields (**BlockingQueue**, **waiting Period** and Boolean **working**) and the required methods(**toString**, **addTask** and **stop**), the run method required for its Thread to start(**the class implements Runnable**).
  + The **toString** *method* iterated through the Blocking Queue and outputs all the Tasks that are currently in the queue.
  + The **addTask** *method* adds a Task in the BlockingQueue and increments the waitingTime of the Task**.**
  + The **stop** method sets the AtomicBoolean waiting to false in order for the Thread to stop running.
  + The **run** method keeps the Thread alive as long as **waiting** is set to true and it gets the first taks in the list and processes it, making the Thread sleep for that amount of time.

For both classes I have used the *Lombok* tool to generate the Constructor and Getters and I have *overridden* the **toString** method according to my needs.

* **Scheduler**
  + The Scheduler class manipulates all the Servers, creating them in its constructor, as well as creating one Thread for each Server and starting it afterwards. The Scheduler has a SelectionPolicy, which states how a client will choose in which queue to go when it arrives.
  + The **stopServers** *method* iterates through the ArrayList of Servers and stops each of them. This method is going to be called from the SimulationManager, when the timeLimit has been reached or there is no other client in the waitingQueue.
  + There is also a **dispatch** *method*,which, according to the SelectionPolicy, will dispatch the Task given as a parameter to the Server that will be chosen. Themethod uses the addTask method from the Strategy interface.
* **Strategy (interface)**
  + This interface has a single method -> **addTask** that will be called by the dispatch method from the Scheduler according to the SelectionPolicy
* **SimulationManager**
  + This class is the most important one, as it manipulates the whole application. It is the class that keeps the Main Thread alive until all the Servers are closed. It has multiple fields: the Scheduler, the simulationController given through the controller, the generatedTasks, etc.
  + The generateNRandomTasks *method* generates randomly N tasks and sorts them in ascending order with regard to the arrival time.
  + The simulationSetup *method* receives the parameters passed from UI when the user introduced them in the MainMenu and sets the environment ready for the simulation.
  + The **run** *method* is the most important one, as it keeps everything under control. At every step of the while method, the current time is increased by one. All the Tasks that have the arrival time equal to the currentTime will be removed from the waitingQueue and dispatched to the correct Servers. After that, the UI is updated and the statistics are calculated. The sleep method is invoked, as we want the application to wait for one second. When the currentTime reaches the timeLimit or there are no Tasks in the waitingQueue or in the queues of the Servers, the while is terminated and the servers are stopped. In the same time, the statistics are displayed on the UI interface
* **MainMenu Controller & Simulation Controller**
  + These classes contain the FXML fields of the UI interface and the methods that updates them. In the MainMenu, the User will introduce the fields that are necessary for the simulation (N, Q, time, etc) and when he presses the Setup button, the simulationManager and simulationController are initiated and the data is transmitted to the Manager.
* **ConcreteStrategyTime & ConcreteStrategyQueue**
  + These strategies comply with the SelectionPolicy and they selec the queue according to it as following: If the first one is chosed, the newly arrived client will go to the queue that has the minimum waitingTime in the current second. If the second one is selected, then the client/task will go to the queue that has the minimum number of clients in its queue. For this project, I have used the Time SelectionPolicy, as it gives much better results.

1. Results

It was not a requirement to test our methods and it was truly not necessary to test them, as we worked with Threads and the most important part here was the Thread-Safety and the synchronization of the Application.

1. Conclusions and further implementation

The conclusion of the project is that we make use of strong OOP concepts and clearly divide it into small sub-problems, the project becomes simpler in no time. Another conclusion is that, when you start documenting your project, you realize that you might have not understood some ideas or your might have not been so clear in your implementation, so it is always better to start your project documenting and stating your ideas and your plan.

I have also learnt new things, like using Threads and Thread-Safety structures and methods, as well as using Lombok to generate methods that I need without wasting time, especially if my models have a lot of fields. I have also made my third project using the MCV pattern, so I have consolidated a more in-depth understanding of the pattern. I have learnt how to create a good documentation and good code documentation comments, while IntelliJ helped me with the Collections and Arrays functions. The other important thing I have got better at is working with git, git bash and GitLab, which is very important for the future. Also, I have learnt how to use streams, which seem to be very important and useful.

A further implementation for the project might be to extend the UI interface to show a more immersive simulation.

1. Bibliography

* **https://www.baeldung.com/java-thread-safety**
* [**https://app.diagrams.net/**](https://app.diagrams.net/)
* **https://lucid.app/**
* **https://en.wikipedia.org/wiki/Strategy\_pattern**
* [**https://www.atlassian.com/git/tutorials/setting-up-a-repository**](https://www.atlassian.com/git/tutorials/setting-up-a-repository)
* **https://riptutorial.com/javafx/example/7291/updating-the-ui-using-platform-runlater**