Technical University of Cluj-Napoca

Faculty of Automation and Computer Science

Semester 2021-2021

Fundamental Programming Techniques

- Laboratory Assignment no. 2 -

Queues management application using threads and synchronization mechanisms

Student: Maria Gliga-Hambeț

Group: 30423

Supervising teacher: Cristina Pop

# TABLE OF CONTENTS

[1. Objectives 3](#_Toc97399318)

[2. Problem analysis, modelling, use cases 4](#_Toc97399319)

[3. Design 5](#_Toc97399320)

[4. Implementation 8](#_Toc97399321)

[5. Results 15](#_Toc97399322)

[6. Conclusions 17](#_Toc97399323)

[7. Bibliography 17](#_Toc97399324)

1. **Objectives**

The main purpose of this project is to design and later on implement a queues management system with a user-friendly graphical interface, which could be easily used in order to simulate real life scenarios. The user can set all the details concerning the simulation: time limit, number of clients and queues, maximum and minimum arrival and service time, as well as the selection strategy (shortest time management and shortest queue management). What we are ideally trying to obtain is an application which can determine, based on all the inputs, the best decision for each client when choosing a queue, in order to obtain an optimal solution, a minimal waiting time.

At the same time, the application computes the average waiting, the average service time and the peak hour. Such outputs are useful in real life for making decisions relating the future – the number of queues that should be open at a certain moment in the day, for example.

In order to achieve the desired result, the whole project can be divided into the following, smaller tasks:

* Design a queue and create a thread for each one
* Design a client – in my project, they are called “tasks”
* Design a scheduler which assigns tasks based on the chosen selection strategy
* Design a simulation manager which properly connects the clients and the queues
* Design a graphical interface
* Design a controller which assures the real-time updates on our graphical interface with the data provided by our simulation manager

1. **Problem analysis, designing, scenarios, use cases**
   1. **Problem analysis**

The queues management application simulates and gives a solution to an issue that people are facing daily. The problem in our situation is the handling the flow of customers in order for them to receive their service in the smallest time possible. As mentioned before, in order to make a decision for each client, we must first know what out options and requirements are. This being said, we need to know:

* The number of clients
* The number of queues
* The simulation time (= for how long we want our simulation to run)

In order for us to simulate a real-life situation, the arrival time and service time of each client is generated randomly based on some minimum and maximum inputs for both categories.

* 1. **Designing**

For bringing a close to life experience to our simulation, the application is based on threads. Each queue representing a thread, the queues are updating at the same time, decrementing the services time of their current client or simply waiting for a client to enter the queue. Of course, in order to make use of the threads, we firstly need to make sure that we have the right classes, each implementing the desired functions. The details concerning the classes, package structure and implementation of each are going to be presented later on in the documentation.

* 1. **Scenarios and use cases**

When the simulation is started, the user inputs the desired numbers and options for the simulation. From than on, if he/she presses the “start” button, the generated clients are printed on the screen, with their real-life assignation to the queues. When the simulation is finishes, either by pressing the “stop” button or when the given time has passed, the user is also given the statistics of the simulation (average arrival and service time and peak hour). If the data is not entered correctly or is not complete, the simulation simply will not start.

**Use case description**

The customers are generated randomly, each having its own service time and arrival time, the number of clients depends on the input values and the simulation time is also decided by the user.

**Title:** Queue management simulation

**Primary actor:** The user

**Main success steps:**

1. The user sets correctly all the inputs;

2. The user presses the “start” button;

3. The clients are generated and their evolution is displayed;

4. The time limit reaches its end and the simulation stops and prints out the statistics mentioned above.

**Alternative sequences:**

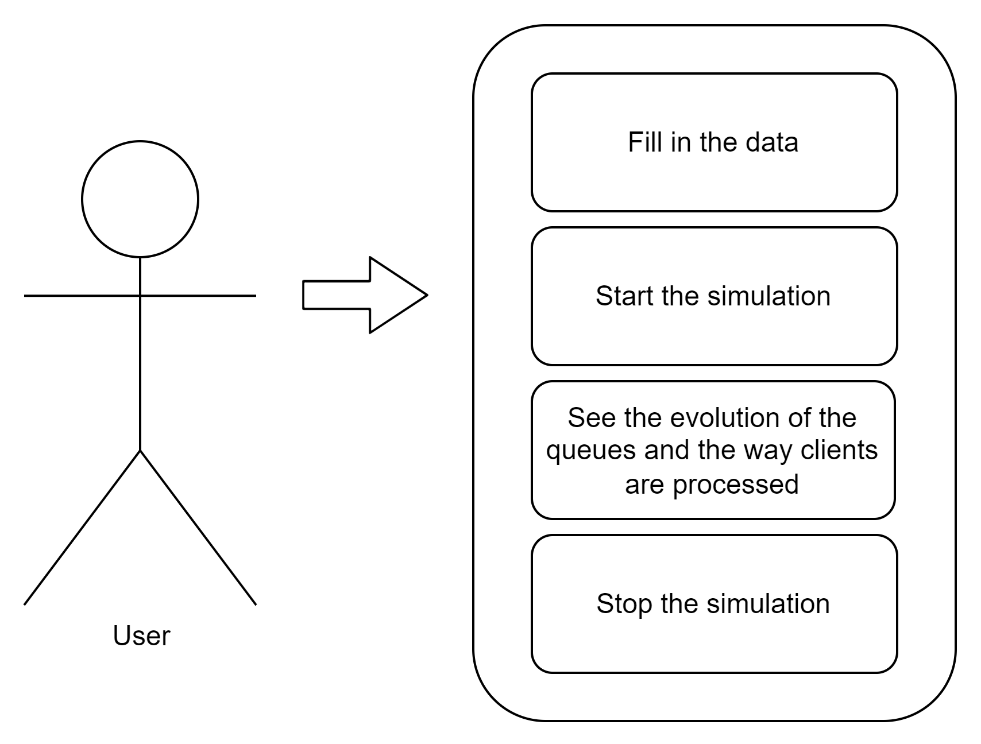
1.1. The user sets correctly all the inputs;

1.2. The user presses the “start” button;

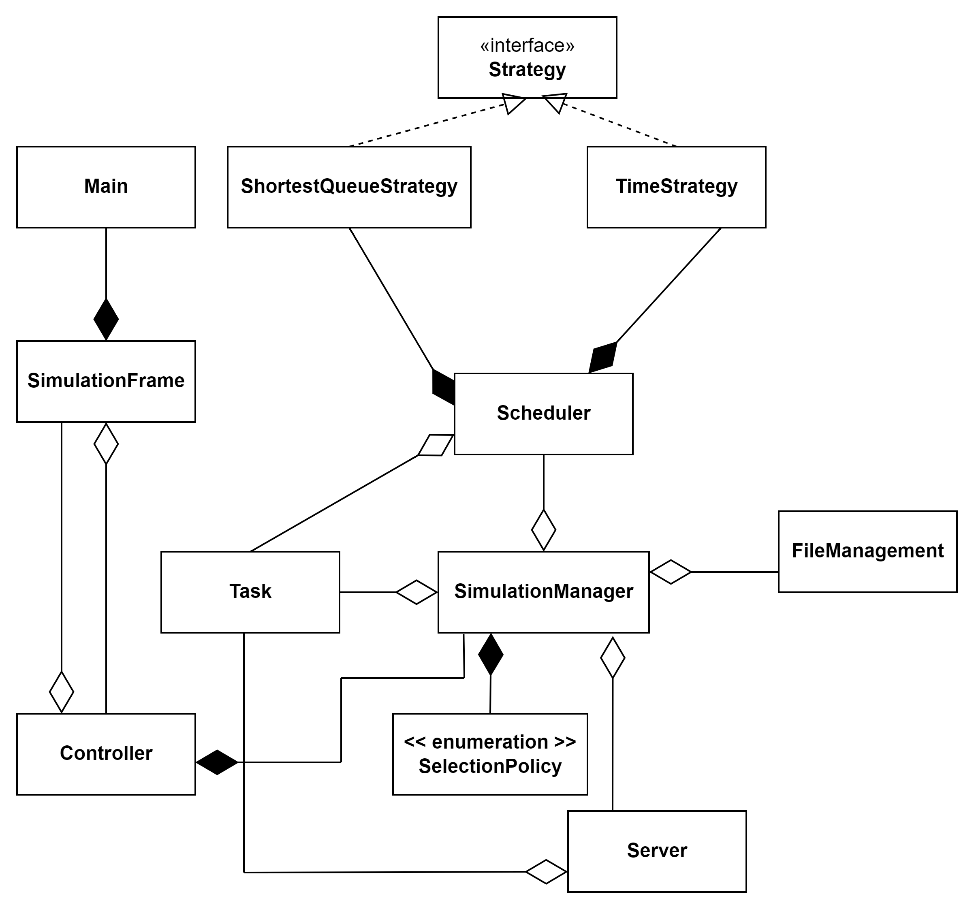
1.3. The clients are generated and their evolution is displayed;

1.4. The user presses the “stop” button before the time limit is reached and the simulation stops, with the statistics showing only the processed clients.

2.1 The user inputs the data incorrectly or incompletely;

* 1. The simulation doesn’t start, but if given the right inputs and ordered to begin, it can be reset.

1. **Design**
   1. **Class diagram**

The following diagram mainly presents the relationship between classes. The individual variables and functions of each class are going to be presented as well, but later on when I will be describing the role of each class.

* 1. **Packages and relationship between them**

The architecture used in this project is a MVC (model-view-controller) one. Generally speaking, when talking about a MVC architecture, the roles of the three components are the following ones:

* The Model contains only application data and no data
* The View displays to our user the data given by the model. It, however, does not know what the data actually means or how the user can manipulate it
* The Controller is the one tying together the Model and the View. It is signaled by the view of certain events and executes certain actions, manipulating the data given by the Model. These actions are, in turn, displayed by the View.

In my particular case, the 3 packages consist of the following classes:

* The Model: classes Task(=Client) and Server(=Queue). Since the clients and the queues are the main “actors” of the simulation, it seemed natural for their data to be the foundation of everything that I was designing.
* The View: class SimulationFrame. It is a class that extends JFrame and holds all the components of my graphical interphase. As it can be seen from above, it communicates with the Controller class, through the actions performed on the 2 buttons.
* The Controller: here I have placed all the classes that manipulate data from the previously mentioned packages. It consists of: FileManagement, Scheduler, SelectionPolicy, ShortestQueueStrategy, TimeStrategy, Strategy, SimulationManager and the Controller class.

In addition to the 3 packages mentioned above, I also added a Main class which is used simply for running the simulation.

* 1. **Data Structures**
* A **BlockingQueue** is a Java queue which is thread safe to put elements into and take elements out of. To put it in a different way, multiple threads can operate concurrently on a Java BlockingQueue, without causing errors. I chose this structure for reasons of synchronization.
* An **AtomicInteger** is a class that protects an underlying int value by providing methods that perform atomic operations on the value. I chose to use it as it does the same as using the synchronize keyword, but it is more readable.
* An **ArrayList** is a resizable array. What makes it different from an array is that it does not have a given size which has to be altered at a certain moment. Internally, it uses a dynamic array to store the elements. I have chosen this data structures when working with data that did not need to be synchronized – the initially generated clients, for instance.
  1. **Interfaces**
     1. **The Runnable Interface**

The Runnable interface describes a class whose instances can be run as a thread. The interface itself is very simple, describing only one method (run) that is called automatically by Java when the thread is started. At the same time, the class implementing Runnable can extend another class and can implement other interfaces.

* + 1. **The Strategy Interface**

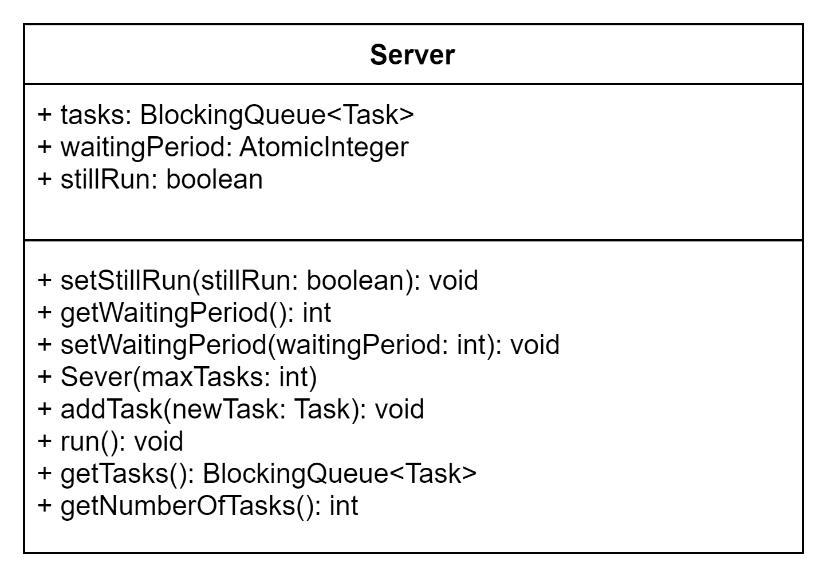
The Strategy Interface, present in the Controller package, is an interface later on implemented by 2 other classes (TimeStrategy and ShortestQueueStrategy). It consists of only one function, addTask. It is used to add clients in the queues, depending on the chosen selection policy, as the algorithms slightly differ from one to another.

1. **Implementation**

In this section, I will be discussing each individual class, its way of working and purpose in the whole application.

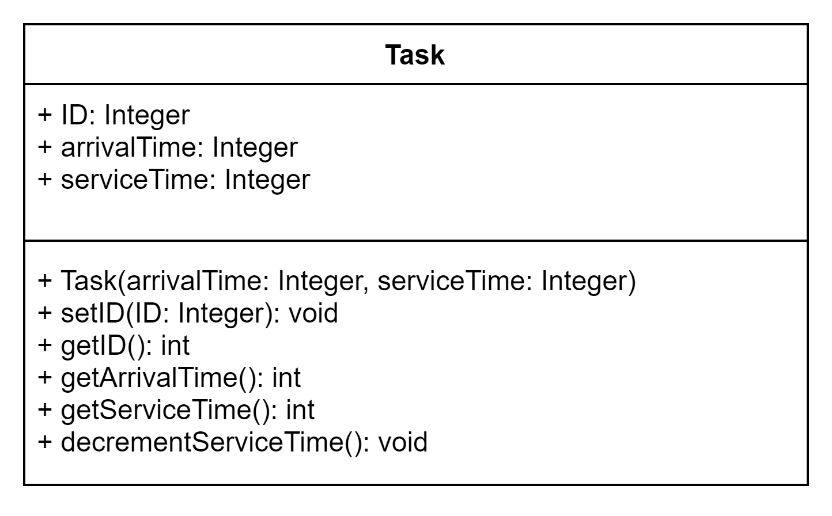
* 1. **Server Class** (part of Model):

The Server Class, representing the queues of my simulation, is one on which the application strongly relies on. As it can be seen below, the attributes describing it are 2 thread-safe ones and a third one, used for seeing when a queue is closed or open or, in other words, when the thread associated with that particular queue is running. The BlockingQueue is used for storing the clients who are currently waiting in line or are being processed at the moment of speaking, while the AtomicInteger tells us the number of those clients. The class implements the Runnable interface and, therefore, implements the method run(). Inside the function, the first client’s service time is decremented by one and the thread sleeps for one second. When a client’s service time is equal to 0, the client is removed from the queue. The class also has getters and setters for part of its variables, in order to maintain a certain level of safety to the code, but still be able to communicate with other classes of the project. The constructor has one parameter, the maximum number of clients per task, in order to initialize the queue with the correct and sufficient number of elements



* 1. **Task Class** (part of Model):

The Task Class, representing the clients of my application, holds a role as important as the one of the Server Class. The 3 attributes describing each client are: a unique ID, arrival (the moment the client should the processed) and service time (the amount of time needed for the service of the set client to be completed). The class also consists of getters for all the 3 attributes and a setter for the ID, while the other 2 are set with the help of the class constructor. There is also a decrementServiceTime method, later on used in the simulationManager.



* 1. **Strategy Interface** (part of Controller):

The Strategy Interface is later on implemented by two classes, which describe the way clients are added to the queues, depending on the type of selection strategy that has been chosen. Intuitively, the interface has only one method, addTask().



* 1. **TimeStrategy Class** (part of Controller):

The TimeStrategyClass is one of the above-mentioned classes implementing the Strategy Interface. The addTask method consists of adding a client to the queue having the shortest total service time. In that way, the amount of time spent by the client waiting is minimal.

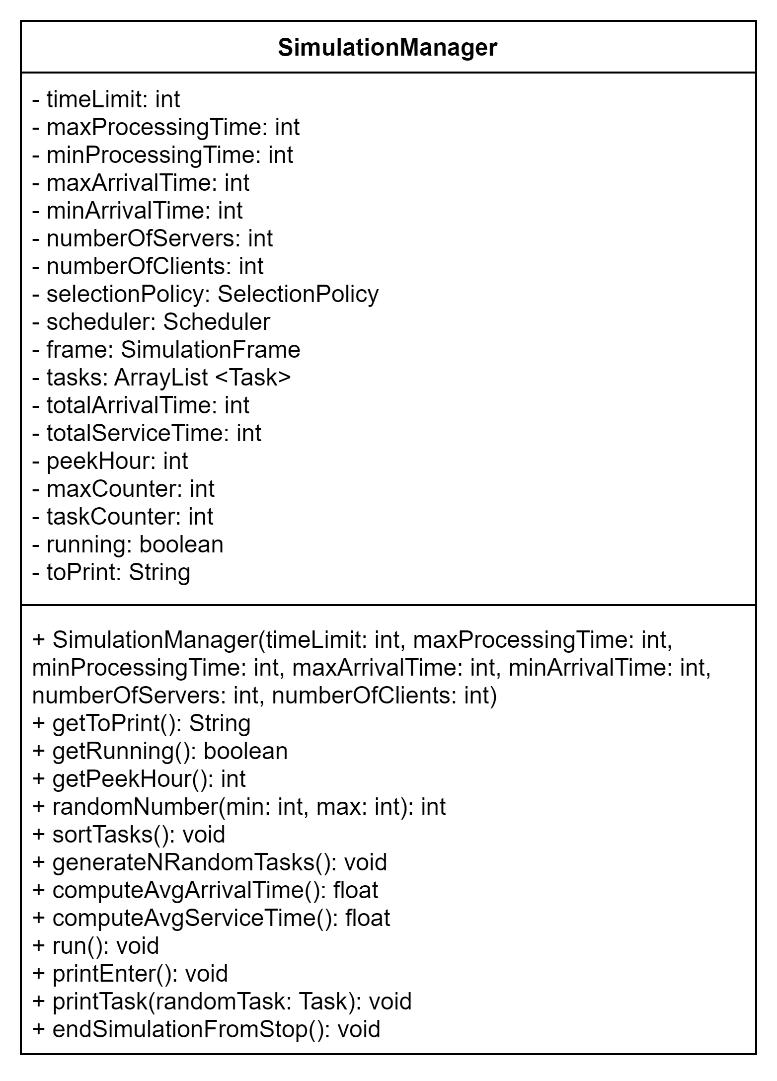


* 1. **ShortestQueueStrategy Class** (part of Controller):

The ShortestQueueStrategyClass is the other class implementing the Strategy Interface. The addTask method consists of adding a client to the queue having the smallest number of people waiting in line. Even thought logically speaking, the previously mentioned selection strategy is more efficient, this one could also be preferred in a real life scenario – when the provider only has a limited amount of space for each queue in which the clients can wait for their turn, for example.



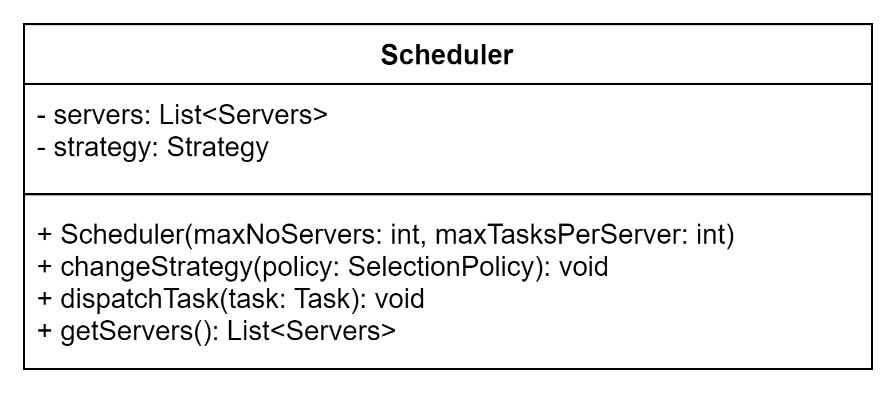
* 1. **SimulationManager Class** (part of Controller):

The SimulationManager class is the one in which a big part of the application is happening. The constructor of the class has as parameters all the data that the user inputs in the graphical interface. Based on those attributes, the entire simulation is created. The clients are generated randomly, using the generatedNRandomTasks, with an arrival and service time inside the two intervals given by the user. The class implements the Runnable interface, having to be synchronized with the other threads of the project. For this reason, the run() method is implemented. Inside it, while the current time is smaller than the given time, the task list keeps on being checked and its elements distributed when the current time equals the arrival time of that client. Also, inside this method, the printing part (in the console, for the graphical user interface and in the file) is being handled. In order to do this, the client queue of each server is being looked at and the algorithm acts in concordance with the result. For instance, is the queue has no clients, it is labeled as closed. The totalArrivalTime, totalServiceTime and peekHour are also being updated whilst going through the initial generated clients. These are going to be useful when computing the statistics that need to be printed at the end of the simulation. In order to have the average of the arrival and service time, as those are the desired characteristics, I also added methods to compute those. Besides the getters of the class, there is also a method sorting the clients in order of their arrival time and some methods that help with the simulation (one that ends it on command, not when the given time has passed and 2 that help with the printing of the results, in order not to repeat code without purpose).

* 1. **SelectionPolicy Enumeration** (part of Controller):

The SelectionPolicy Enumeration presents the options when talking about the way in which we want our clients to be processed. At the moment, the 2 strategies are shortest queue and shortest time, but there could be other options added for further development.

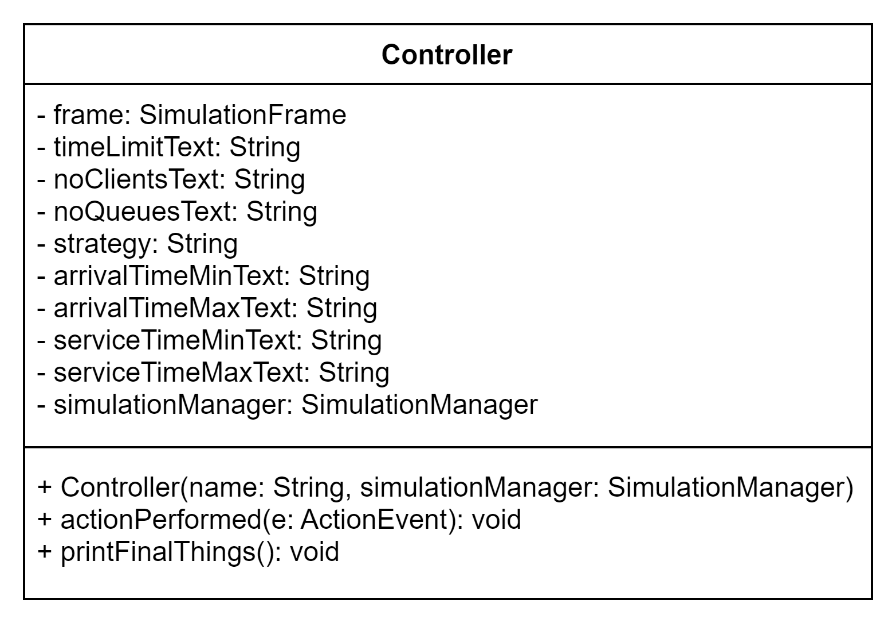
* 1. **Scheduler Class** (part of Controller):

The Scheduler Class is the one helping with the distribution of clients in queues. The 2 attributes are the list of servers (=our queues) and the strategy. The constructed is given 2 parameters, maximum number of clients and maximum numbers of tasks. Here is the point where the queues list is actually initialized and the threads associated with them are started. The changeStrategy and dispatchTask methods work for dispatching the clients in the correct queue, depending on the given inputs.

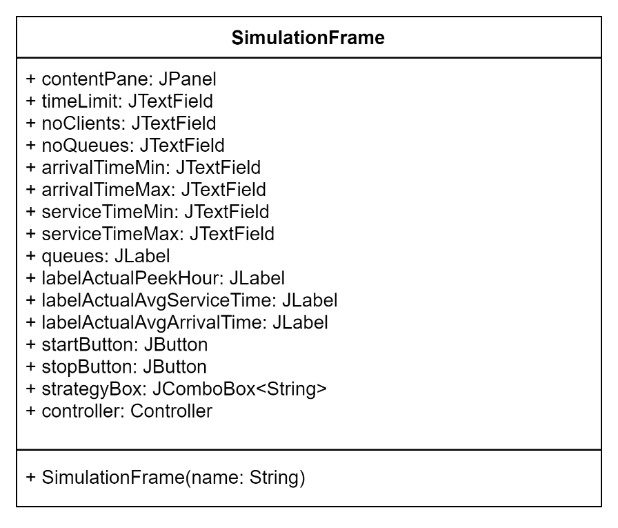
* 1. **FileManagement Class** (part of Controller):

The FileManagement Class consists of only 2 attributes and 3 methods (setFile, write and closeFile), which are all declaring public and static, in order to help with the printing inside the files task. The methods are rather intuitive and simply describe the actions of creating, and closing a file and also writing in it a certain string.

* 1. **Controller Class** (part of Controller):

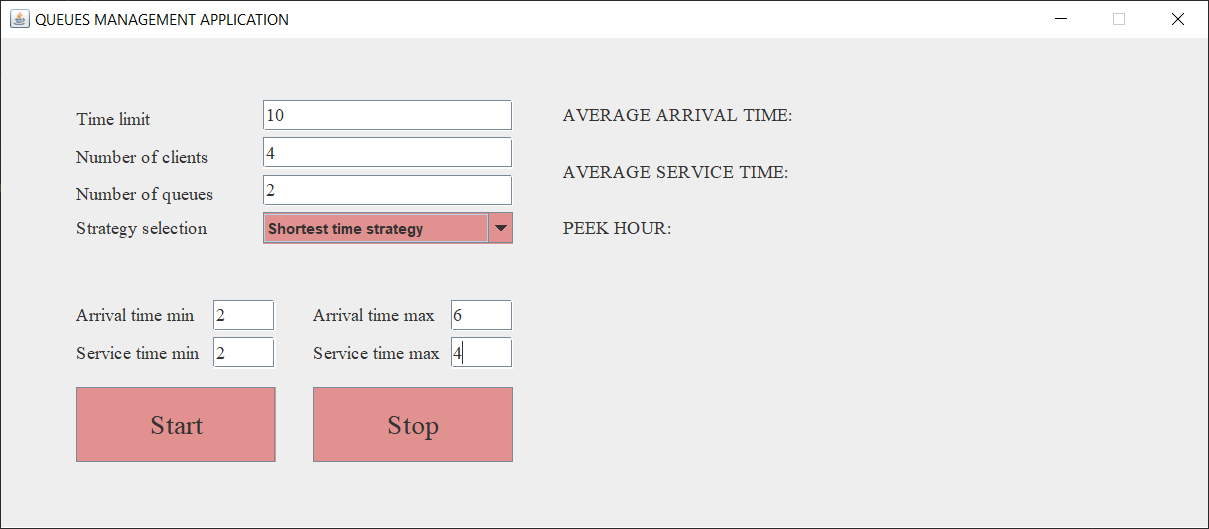
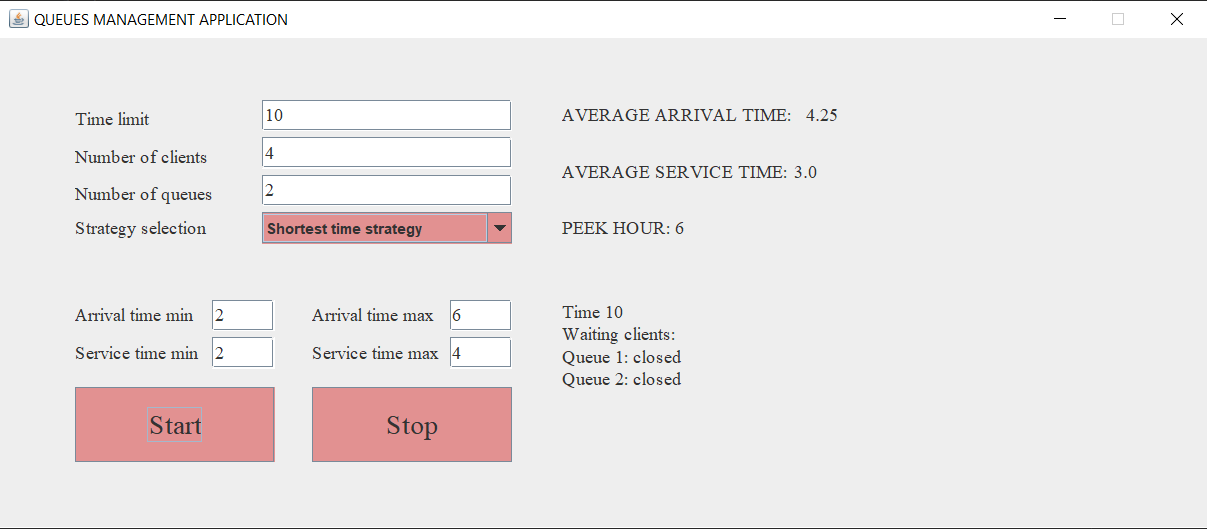
****The Controller Class is the one connecting our simulation to our graphical interface and the other way around. Its constructor has the parameters name and simulationFrame, both used in order to instantiate our frame, which is also set to visible and given the default close operation. The main method of the class is the actionPerformed one, which is being looked at from the simulationFrame, when the 2 buttons are being pushed. If the command is given by the “start” button, while the thread of the simulationManager is still running, the method prints live updates to what happens with our clients at queues. Otherwise, if the “stop” button is being pushed and if the simulationManager was previously instantiated (a thing which happens inside the “start” command part of the method), the final statistics are also printed, with the help of the printFinalThings() method

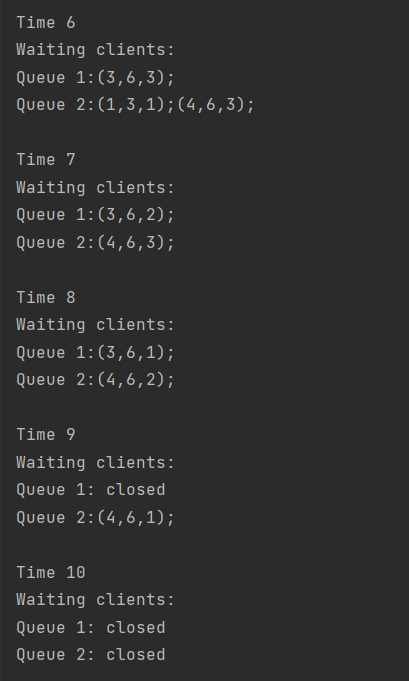
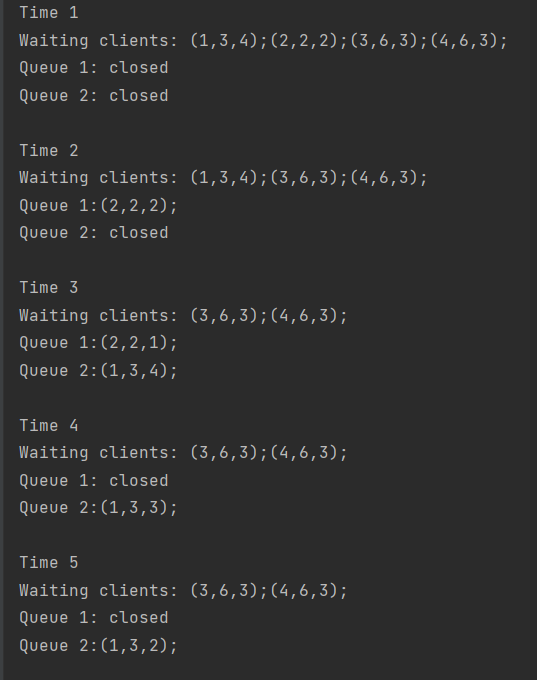
* 1. **SimulationFrame** (part of View):

****As previously said, the SimulationFrame Class holds all the components of the graphical user interface and, basically, describes it. It communicates with the Controller through the actions performed on the buttons, in order to synchronize it with the rest of the application and be able to see live updates on it.

1. **Results**

The simulation of the application can be done in various ways. Of course, the most user-friendly is the one in which we use the graphical interface. Nonetheless, the simulation works as well if we write the inputs manually in the SimulationManager case.

As previously mentioned, if the inserted results are not correct or if they are incomplete, the simulation simply will not start. In the example below, I have used the graphical interface and the results are the one from GUI when the simulation is completed and all the steps can be seen, as well, from the console.



1. **Conclusions**

The whole project has enabled me to learn more about parallel processing and actually have some hands-on experience when working with threads. I find that by using Java Swing, I was also forced to learn the way to synchronize a graphical interface with the rest of the application, something that does not come as naturally as maybe the other parts of the process.

What is more, I have deepened my knowledge on how to work with the model-view-controller architecture.

Future developments for the project:

* add some more statistics which could be seen as useful by the provider
* keep track of previously computed results in order to compare them and find the best solutions

1. **Bibliography**

* <https://stackabuse.com/reading-and-writing-files-in-java/>
* <https://dsrl.eu/courses/pt/>
* <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html>
* <https://stackoverflow.com>
* <https://www.javacodegeeks.com/2012/12/multi-threading-in-java-swing-with-swingworker.html>