**Assignment nr. 2**

Lungoci Luca, 30424

1. **Assignment object.**

The scope of this assignment is to simulate how clients are placed in a queue according to two strategies. The simulation happens in real time and the results are displayed every second on the user interface. With this assignment, threads were introduced and processing multiple tasks in parallel was made possible. Clients are coming to be served at a random time, and they can be placed in the queue where they will wait as little as possible, or in the queue with the smallest number of customers in it. The number of costumers, queues, the minimum and maximum values for the time a client can arrive, the minimum and maximum values for the time a client is processed, simulation time, strategy to place the clients in a queue, all those things can be chosen by the user via the graphical interface.

In order to make this simulation possible, some secondary tasks had to be solved. The values introduced by the user in the text field had to be actual positive integers. Some connections between some values had to be checked, for example, the minimum time at which a client arrives cannot be greater than the maximum time of his arrival. Also, synchronized data structures had to be used in threads, to avoid thread interference.

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

The models I used in these assignment are: tasks (to represent the users), servers(to

represent the queues).

* Functional assignments: user can enter the intervals for the value of a simulation parameter, can choose the strategy for adding clients to the queues, can see the current status of the queues and the waiting clients.
* Non-functional requirements: the application should be easy to use, provide helpful messages in cases of errors, display the result in a friendly form.

Use cases.

1. User does not insert any values.

Success scenario:

* + User just clicks the “Start” button.
  + Some default values are chosen for the parameters and the simulation begins.
  + The results appear on the screen.

1. User chooses values for the parameters.

Success scenario:

* User sets values for the parameters.
* Presses “Start” button.
* Simulation starts and results are displayed.

Alternative scenario:

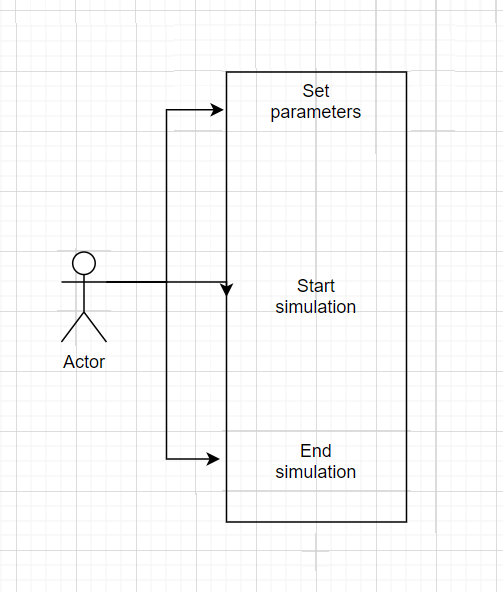
* User inserts some values.
* Simulation can’t run with the values provided by the user and displays an observation text, in red.
* User sets some other values for the parameters.
* User presses “Start” button.
* Simulation begins and results are displayed on the interface.

Alternative scenario:

* Users inserts values.
* Users starts simulation.
* User changes the strategy during simulation.
* Results are displayed and respect the current strategy.

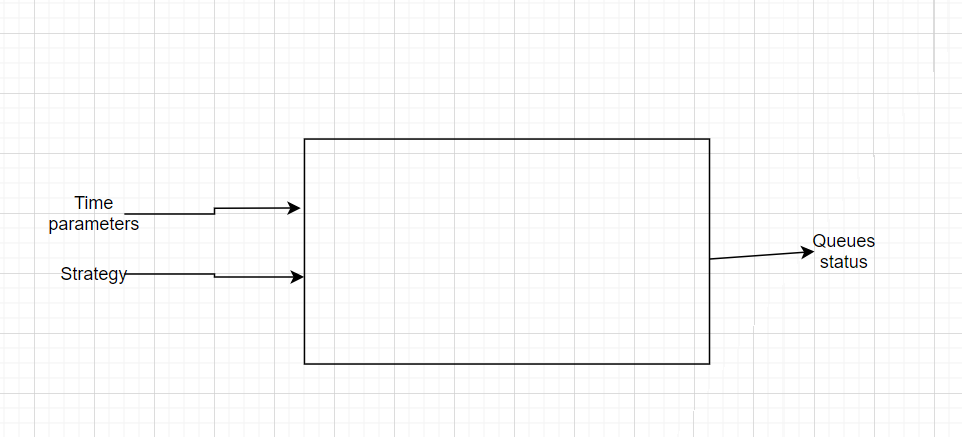
Alternative scenario:

* User inserts the values.
* User starts the simulation.
* Results begin showing up.
* User stops all threads by clicking the “Stop” button.



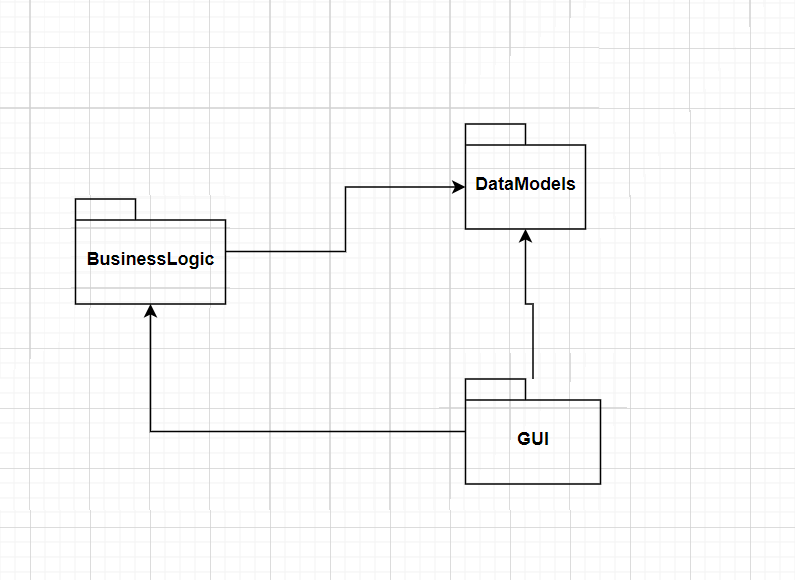
**3.Design.**

* Application

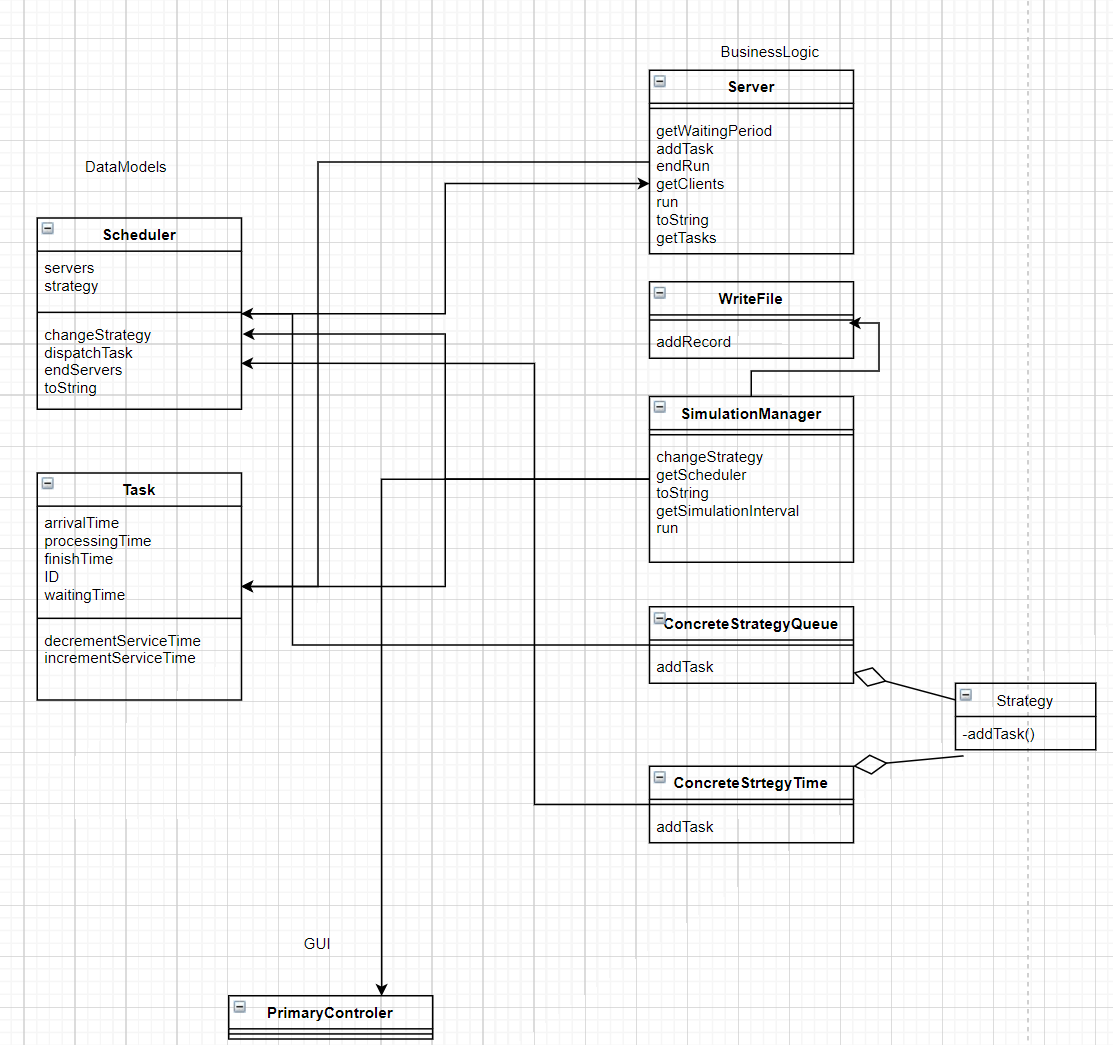


* Packages.

I considered the “Scheduler” and “Task” classes to be part of the the “DataModels” package because there are mainly used for storing the representations of clients and queues. The methods of those two classes are used for adding elements, getting and setting some fiels, just like you will do for a typical data structure. “Strategy” ( contains a selection method ), “Server” (processes clases, has a “run” method), “SimulationManager” (has a run method, commands the other classes) and “WriteFile” (appends a text file), all those method I considered as being fit for the “BusinessLogic” package.



* Classes



1. **Implementation.**

After choosing the parameters, the strategy and after clicking the start button, the constructor of the “SimulationManager” class created random tasks ( tasks representing the clients), assigns some values to the tasks respecting the intervals inserted by the user ( or the default parameters if the text fields are empty). A “Scheduler” object is also created and its constructor creates a list with “Server” objects (representing the queues) . Now, after just instantiating an object of type “SimulationManager”, we already have the queues and the clients that need to be served, stored in lists. By calling the “run()” method of the “SimulationManager” , time time of the simulation starts increasing, the queues start filling up and emptied. The threads representing the queues are stopped when the current time of the simulation has reached the value for the total simulation time. During the simulation, a sequence of “toString” methods are called, which build a string representation of the tasks not yet processed and the ones from each queue. This string is passed on to a method in the “GUI” package and displayed on the user interface. Also, the same string is appended to a text file so that the user can see the state of each step of the simulation.

The “Server” objects are responsible of removing the clients that have been served from the queue. To store the clients, a “BlockingQueue” was used, because operations of adding clients and removing clients are done in parallel, a synchronized data structure had to be used. The “waitingPeriod” field and the “clients” field is used by the classes implementing “Strategy” as a comparation criteria between queues. Those fields must be updated every time a client is added to the queue, leaves the queue, or another second of the simulation has passed. This class also needs a stop method, that breaks the loop in the run method. The “endRun” method will be called when the simulation time and the current time have reached the same value. The “toString” of this method comes in play when displaying the clients for each queue. The blocking queue with the costumers is traversed, each “Task” is accessed and a string with information of every client in the queue is obtained. This string is passed latter on to the “Scheduler” class.

“ConcreteStrategyQueue” and “ConcreteStrategyTime” are both in charge of sending clients to the server. A client can eighter be send to the queue where he will be processed sooner, or to the queue where less clients are standing in line. The application can swap between those two strategy any time, at the request of the user. Those classes call the “addTask” method for best server according to the chosen strategy.

The “Scheduler” class switches between strategies and stored the list corresponding to the queues. It is also responsible for starting those queue threads and ending them when the time comes. Because this classes stores every queue of the simulation, its “toString” method brings together the strings describing the clients of every queue. It creates new lines and puts some text to identify each queue (“Queue 1”, “Queue 2”). This is string is passed later on to the “Simulation manager” class. This class is where the “peakHour” is computed. Having all the references to the queues, the for “forPeakHour” method traverses the list with the queues to see if the we have a new biggest number of clients at the queues. It takes as parameter the current time, given by the “SimulatuionManager” class (calling class), to update the time when there where the most costumers in the queues.

The “Task” class is a container for the parameters describing a client. Its fields are: id, arrival time, processing time, waiting time, most of them assigned in the “SimulationManager”. Each task corresponds to a costumer. The methods in this calls are mostly getters and setters, but also a method decrementing the processing time, in order to simulate the passing of time in the simulation.

The “WriteFile” class is responsible of copying simulations results, at each step, to a text file and rewrite the file when another simulation takes place. In this class there is a path to the specific text file, open, close methods and one for adding another simulation record.

The “PrimaryController” class handles the graphical user interface. It keeps the parameters that are extracted from the text fields in an inner class: “Struct”. These parameters are extracted form the seven text fields as stings, are passed to the “toInt” method so that their integer representation is obtained. A method to check, if the values inserted by the user can actually be used in the simulation, was needed. This task is assigned to the “checkInputValues” method with checks if those values would be good to use when simulating a queue. Inserting characters in the text fields or negative numbers instead of positive integers will cause a lot of errors while running the simulation, so the “checkIfInts” method prevents such event from happening. Those to methods for checking also change a label in the graphical user interface, colored in red, to inform the user what mistake was done when inserting the simulation parameters. There are three buttons in the graphical user interface which, when pressed, execute the code in the following methods: “endSimulation”, “changeStrategy” and “Start”. This class also has a method for filling up the text fields that the user left empty. Some default values will be added in order to avoid crashing the application. The user can just click the “Start” button and the application will execute without any problem. By pressing buttons, the user can change the strategy which decides how costumers are added to the queues, start and end the simulation process. In the down side of the graphical user interface is a text area, where the current status of the queues, the clients that have not yet been served and the current time of the simulation are displayed. These values change every second when the “addText” method is called from the “SimulationManeger” class. In the “start” method of this “PrimaryController” class starts the chain instantiation of classes with the creation of an object of type “SimulationManager”. A reference to the “PrimaryController” classed is passed on to the constructor so that the string describing the current status of the simulation can be passed on.

“SimulationManager” is the class generating the tasks, managing the scheduler, deciding which task to send for processing, at which time. Because in this class are stored all the references to all the tasks, here are computed the average waiting time and the average processing time. The average processing time can be computed after generating the lists. The average waiting time uses the “waitingTime” field stored in the “Task” class. This field was incremented by a “Server” class every time a task was waiting in the queue but was not yet its turn to be processed. In the “toString” method of this class, the strings returned by “Scheduler”, with information about the status of each queue and each task in the queue, those strings are concatenated with some new ones created in this class. These new strings store information about the tasks not yet send to queues, the current time, and the peak hour, average waiting time and average processing time. After the tasks are generated and random values between the chosen parameters are set, in the “generateNRandomClasses” method, those tasks are sorted in ascending order. For this, the sort method of the “ArrayList” class is used, but its “compare” method is being overridden, so that the tasks are sorted taking into account their “arrivalTime” field. The “while” loop of the run method ends when the current time is equal to the set time for the simulation. After that, the file in which the output was written is closed, and the “Scheduler” objects method, to end all threads representing queues, is called. This marks the end of the simulation and the very last step remains displayed on the user interface.

1. **Conclusions.**

This assignment was very useful, working on it made me understand how threads work, why there are useful and the problems that arise when working with them. I also learned how to develop applications that allow me to see the evolution of events in real time. Putting threads to sleep and viewing the status of the simulation seems to open many possibilities for future programming applications. I also found useful the way of changing strategies form the support presentation, extending from an interface and switching between subclasses when needed.

1. **Links used.**

<https://stackoverflow.com/questions/16252269/how-to-sort-an-arraylist>

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html>