DOCUMENTATION

ASSIGNMENT NUMBER 2:

**— QUEUE MANAGEMENT —**

NAME: Chețe Doru-Gabriel

GROUP NUMBER: 30423

1. Objective of the assignment

The main object of the assignment is to build an application that simulates a series of N clients arriving for a service, entering a number Q of queues, possibly waiting for an amount of time and then finally being served and leaving the queue. The application should also compute the average waiting time(time waiting in a queue to be served), the average service time(time spent in the queue, both while waiting and while being serve), and the peak hour, that is, the hour at which the queue management system is at its most busiest.

The secondary objectives are:

1. Analyze the problem and identifying the requirements
2. Design the application that can perform the simulation
3. Implement the application
4. Test the built simulation application

With these objectives clearly set, the start of the analysis of the problem will be presented:

2. Analysis of the problem, modeling, scenarios and use cases

Analysis:

The problem can be viewed as a matter of assigning the best queue at a certain moment, that is, the moment a client gets into a queue, based on a certain criterion. This could be a different criterion, if using the Strategy pattern. For the purpose of the analysis we’ll however reference the shortest waiting time strategy.

Every queue object (server) has to keep track of its own load (number of clients waiting or waiting time required to get to the front, or others criteria). These servers have to essentially function in parallel. This means that the data will be updated concurrently. So managing the entire simulation means keeping track of these queues(servers) and the clients that will be “dispatched” to their corresponding queue”. For the shortest waiting time strategy, it will be essential that the model takes into account the fact that these servers are running in parallel and the updating of the waiting times is done correctly.

So the user has to be able to provide an input, set the simulation up and once ready, start it using some UI interaction and be able to see the real time progress of the queues and the clients being served. The UI should be straightforward, with simple elements that just set the parameters, run the simulation and a simple and perhaps large element for displaying the status of the queues, since they can get quite large and although they may not always fit onto the screen, especially for large parameters, it should still be able to fit enough text.

Modeling:

In order to model this problem, we can consider two big packages: Business Logic and Model. The model is rather simple. There are two entities that are needed to solve this problem: the queue or more abstractly, the server and the client or more abstractly, the task. So these two entities will be modeled as two classes. Within these two classes, we’ll keep track of: the activity on the server in the case of the Server class and we’ll provide a run() method since this object will be runnable – each server(queue) will run on a separate thread. The task object will just keep the data corresponding to each task(a client).

Within the Business Logic package, there are 2 main entities: the Simulation Manager and the Scheduler objects. The manager will take care of the generation of the random clients, the initialization of the simulation and the handling of the UI. Since JavaFX will be used for the purposes of providing a GUI, UI elements will be referenced within the Simulation Manager object. The scheduler will be initializing the Server objects, based on the parameters given at simulation start. The scheduler also has the role of directing the Strategy pattern. This brings us to the final part of the Business Logic: the Strategy. Strategy is just an interface that is to be implemented by the various solutions to the problem of assigning clients to servers. In this case, we’ll have two strategies, so two classes that implement a specific strategy. The time strategy class has to implement the dispatching of tasks by looking for the server with the overall lowest waiting time, regardless of the number of clients waiting, while the queue strategy class has to implement the dispatching based on the number of clients already in a queue, without taking into account the time it will take for a task to get to the front of the queue.

Scenarios:

The failure case for this application would be if the user chooses to insert non-integer characters as parameters and in this case, since the simulation wouldn’t make sense, the GUI log of events will display an error message, indicating the problem. This could either be non-integer input or invalid input, that is, if one the minimum parameters is greater than its corresponding maximum. In this case the user can continue and if a correct input is inserted, than the simulation will start after the “Simulation start” button is pressed.

In the case the input is correct, the simulation will start as follows: data from GUI will be fetched, clients will be generated based on the parameters and shown on the GUI log for a period of a couple of seconds. Then the simulation will start and every time unit (every second), an updated view of the queues, the clients waiting in the queues and the clients not yet arrived will be shown on the log. Once simulation maximum is reached, the simulation will finish, the results will be computed and will be displayed on the GUI log(peak hour, average wait time and average service time) for a couple of seconds before the application will close by itself.

Use cases:

Diagram

Description automatically generated

Diagram

Description automatically generated

The use case for this program is in conclusion very simple: user chooses all input parameters: the number of clients, the number of queues, maximum and minimum arrival time, minimum and maximum service time and finally the simulation time. Once done, user presses the “Simulation start” button and if the input data is valid the simulation will commence. The user can then view the evolution of the servers and of the clients being served in real time and the final results regarding averages and peak hour.

3. Design of the simulator

For the design of the simulator, I started with the ideas presented in the modeling part of the documentation and followed it faithfully. The simulation manager and the scheduler will make use of the model classes to simulate the queues and the clients. Within the simulation manager object, there will be a list of Task objects, the “clients”. These are the objects that will be generated and “dispatched” to the scheduler to be dealt with at the level of the servers running on concurrent threads. In this class there will also be a field for the Scheduler object. It will be initialized at simulation start and then used as the handler. The scheduler class will contain the list of running servers, so the queues, and the Strategy object will be initialized here, satisfying the Strategy pattern, that is, choosing a strategy at runtime, when the scheduler is created. Obviously the strategy can be changed within the scheduler object. The concrete strategies contain just the implementation of the dispatch method. These were the objects within the Business Logic package.

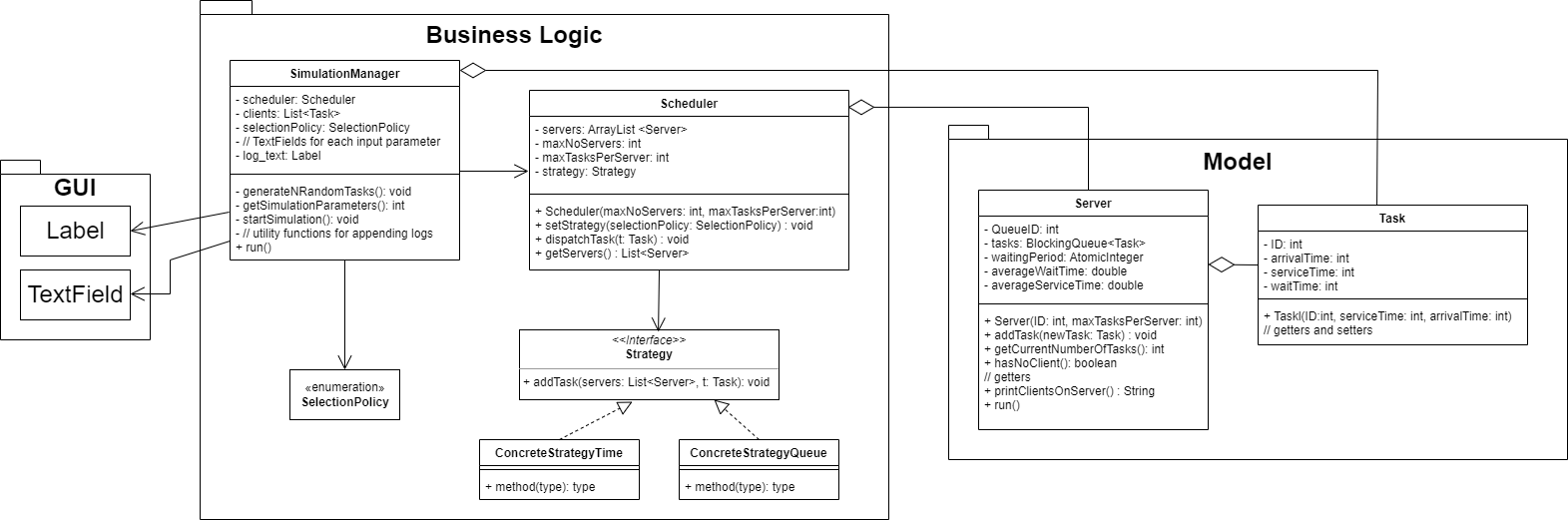
Within the Model package, the Server object contains the fields describing the queue and its status: the id, the waiting period, the average waiting time and the average service time. Waiting period is a field of Atomic Integer type, since its important to consider the concurrent nature of the program. Using this data type, operations with the waiting time will be computed atomically and we’ll be able to avoid synchronization issues.

Within the GUI package, there is the .fxml file, which in the case of our JavaFX application represents an object to be initialized, “loaded” that contains all of the UI elements and that runs on its own UI thread.

Data structures that are used:

With respect to the data structures that are used, we can limit to talking about Lists. The way the various classes keep track of the objects they contain is through lists. In the case of Simulation Manager, we have a list for keeping track of the clients that we generate and dispatch to the Scheduler. In the case of the Scheduler, another List for keeping track of its running queues. For the Server however, we must again consider the concurrent nature of the program. There are multiple threads running. Every server is running on its own thread. Unavoidably there will be cases when threads will want to access a shared resource. Therefore, we’ll make use of the Blocking Queue. This data structure is a synchronized one, which again ensures that we can avoid concurrency issues while processing clients to different threads. While processing, obviously, we also need to check all of the queues for the best server a client can be dispatched to. With blocking queue, we can rest assured that concurrency problems are kept at bay.

UML diagram:



For the UML diagram, the packages described above are illustrated. The relationships between the different classes are also made obvious. Concrete strategy objects implement the Strategy interface, that is used as a field in the Scheduler. Here we can consider an association relationship, since Scheduler has a Strategy object. The same applies to the Simulation Manager object, that has a Scheduler object and is also associated with the Scheduler class. In the diagram, we also consider the simple enumeration that is the Selection Policy. This can be widely expanded in case more strategies are to be added and is also associated with the Simulation Manager since it is one of its fields. The manager also contains references to all the UI elements, which are really just text fields and a label for the log. Finally, the Server and the Task are in a Aggregation type relationship, since we have an array of Tasks in each Server object, that is though not entirely composed of Task objects. Aggregation is also the case for Manager and Task as well as for Server and Scheduler. In these cases, the contained object doesn’t quite own the containing class. A server object is an aggregation of Tasks, but still logically exists without any tasks in it.

Algorithms:

For algorithms, we can consider the different implementations of the Strategy interface. Obviously, there will be a different algorithm for each of the concrete strategies. The used and implemented strategy, shortest time, is implemented with a very simple algorithm. Every time a task is to be added, a maximum value is initialized and an iteration through the servers list at that particular moment is started. Every server keeps track of its waiting time in real time so we can expect the current waiting times of the queues. By comparison, if there is a smaller waiting time the maximum will be set to that value and the best server will be set to that particular object. At the end of the algorithm, if we did find a server with the best(smallest) waiting time, we added the dispatched task to that server. This algorithm would work for any type of strategy, since really the only thing that changes is the criterion by which we make the selection of the best server to be used.

There is also an algorithm for Server’s run() method. At each time step, each Server thread checks if it has any tasks and if it does have any, it updates the number of clients it served, the time it took for them both to be serviced and the whole time they spent waiting and then updates the Server average of wait times. Then the thread goes to sleep sequentially for 1 second at a time until the current client is served. The client is then considered served and removed. These average times are then used in the main Simulation thread for computing the final average times.

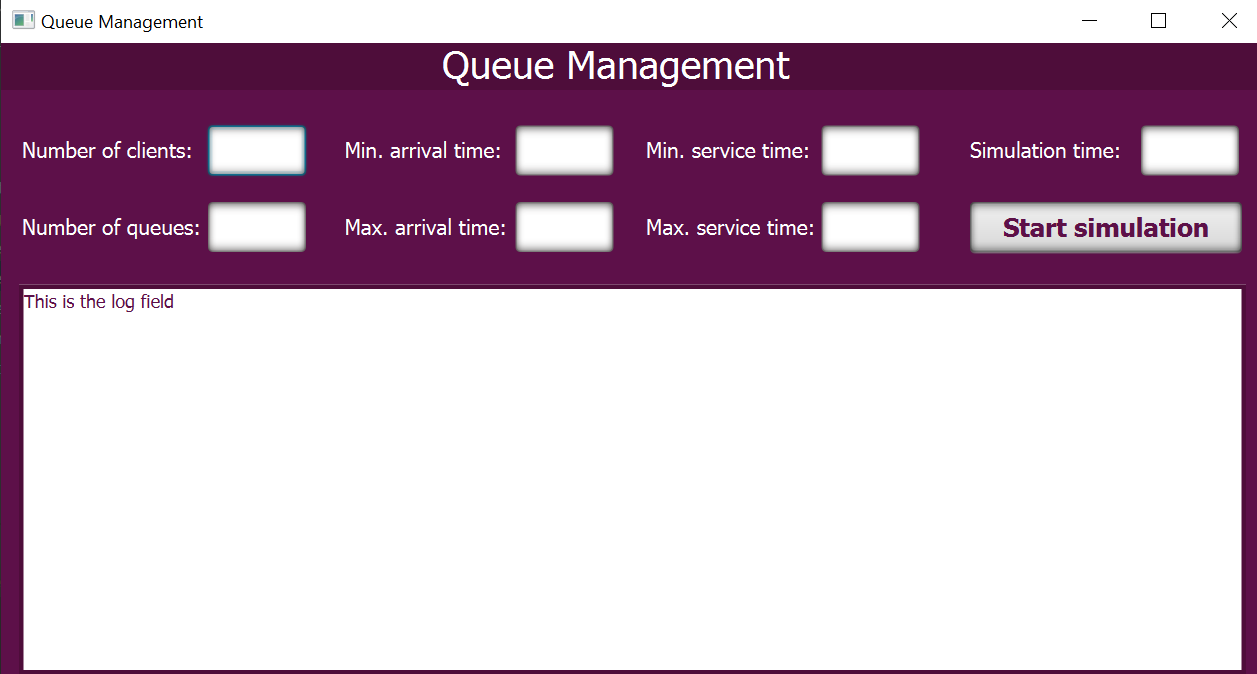
4. Implementation

The implementation, considering the constraints of the design is quite straightforward. The modeled structures can be easily implemented in Java. Every UI element (text fields and a label for the text in the log) are referenced in the Simulation Manager class. They are fetched with the getSimulationParameters() method at simulation start from the GUI and used in the generateNRandomTasks() method. Here every client is generated using random bounded values for each of the given parameters. These methods are called from the startSimulation() method. If we have a success in processing the input, the simulation properly commences by initializing the scheduler object and by starting the main thread. In the run() method of the manager, we just check if any client arrives at current time and if so, it is dispatched to the scheduler. Then printing to UI and to the log file is handled for each step and once simulation is over results are also printed to both UI and log file.

The scheduler has the fields described in the chapters above and has within its in constructor, the initialization of the strategy as well as the initialization of server list. Each server is created and its thread is started. The strategy can be initialized using the setStrategy() method.

The server and the task also have the fields described in the design chapter. For the server, besides setters and getters, there is a printing method for printing the status of the server and the addTask() method that creates a new Task object using the waiting period field and then adds it to tasks list of the object. The run() method in this class is implemented as described previously. Within the Task object, there are only the describing fields and getters and setter

Finally GUI.fxml contains the user interface.



This was made using JavaFX library. As aimed in the beginning chapter, it is very simple and straightforward. User input comes from the keyboard: integers for every field and then the Start simulation button can be pressed. Then the status and the progress of the simulation is displayed in the label below.

5. Results

For testing the implementation of the program I used log files. Every time the GUI is updated in the Simulation Manager, a FileWriter object is used to write to a log file that is generated every time a simulation is ran. In this log file we have all the steps that were taken by the program at each time step and we can carefully follow the progress. I checked in particular that the waiting times are computed correctly and updated at each step. When a waiting time is computed for a queue in order to possibly add a client to it, the program should consider the current waiting time, that is, a client may already be served for a long time and despite its long service time, the overall waiting time in the queue may be rather small and be the best option for a new client.

6. Conclusions

During the analysis and development of this project I learned a lot about concurrency. I consider this a very complex problem, however Java provides the programmer with a lot of tools that can be easily and quickly used in order to be able to use threads without encountering serious problems. Using synchronized data structures such as the Blocking Queue and an AtomicInteger type for the waiting period were essential parts. I also learned how to simply start threads in Java and how to provide a run method for them and I also learned to consider the fact that for a JavaFX application, there is an “Application thread” that runs the UI and obviously, modifications to elements from this Thread, the UI for example cannot be done carelessly. I used a Platform.runlater wrapper to fix this and I learned to take this into consideration when working with threads and UI in the future.

7. Bibliography

This article helped me learn more about the problems I had with updating the UI from another thread: <https://stackoverflow.com/questions/58567654/i-am-running-a-javafx-program-and-i-am-getting-an-illegalstateexception-in-a-sit>

This helped me learn more about Java concurrency in general: <https://jenkov.com/tutorials/javafx/concurrency.html>

Helped me with finding RGB colour codes for the UI: <https://www.color-hex.com/color/5d1049>

Helped me with understanding how to set the label to go to newline when space isn’t enough: <https://stackoverflow.com/questions/15977295/control-for-displaying-multiline-text>

Documentation for Platform.runlater(): <https://www.demo2s.com/java/javafx-platform-runlater-runnable-runnable.html>

Documentation for the Writer object I used for writing to the log file: <https://www.javatpoint.com/java-filewriter-class>