<u>Homework2 – Fundamental Programming Techniques</u>

Ramona-Alexandra Nechita

Contents

1.Task

The task for this assignment was to illustrate the evolution of multiple queues at a time (e.g. the queues at a supermarket) through the use of threads. Random task had to be generated, sent to the queues, processed and then removed.

Since each queue can be considered a separate process (taking clients in and fulfilling tasks), each queue (referred to as server) needed a new thread. The entire simulation was also a thread, since we had to simulate how time passed and how that affected the other processes.

The evolution of the queues had to be shown in GUI, and I chose to represent each server as a JList because it was more practical. The elements inserted in the list are clients, and they are sent to the shortest possible queue, stay in the list until they are at the front and after their task is processed they are removed.

2.Strategy

From this assignment's point of view a thread is nothing more than a process running in background until it is told to stop or it stops due to circumstances. My approach was quite straightforward : since a server needs to do his job until there are no more clients to serve, it can be considered a thread. Another thread-like concept was the fact that in the simulation clients had to be sent at servers for a specific period of time. Therefore, those classes implement the interface Runnable and have the method run().

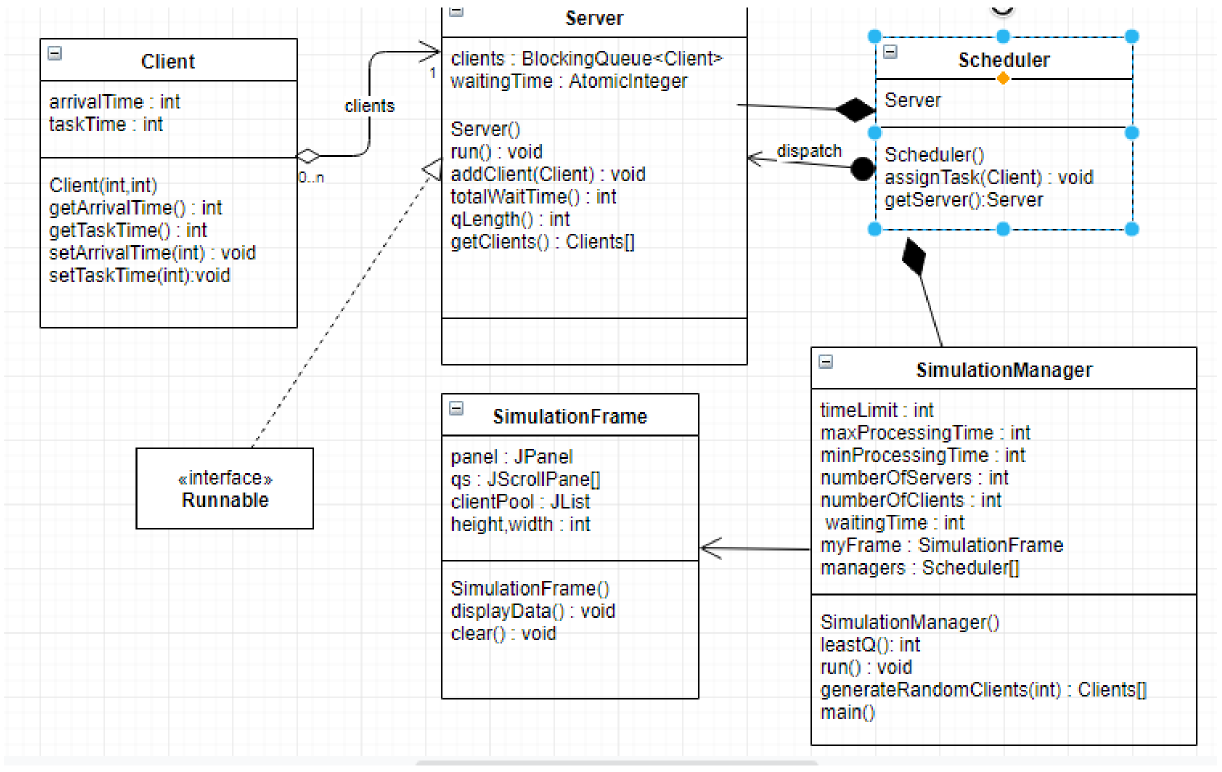
Threads gives an unmeasurable amount of capabilities, such as sleep(int millis) which allows the process to be idle for a given time, join() which waits for the completion of the process, and so on. Therefore, whenever a client is at a server, the server needs to sleep for amount of time the client is there in order to simulate the idea of processing the given task.

Even though the strategy is plain and straightforward, the implementation might come up with a certain number of issues that need to be taken care of. For instance, when the two different threads are to access the same data (say, an array or a list) due to concurrency they might try to access it at the same time or at very short intervals and cause it to deliver inadequate values. Knowing this, any variable, variable array or list that is known to be shared by two different threads is declared as volatile. This way, it is safe to send both threads to the same variable because it will behave accordingly for the concurrent processes. Moreover, the Collection of BlockingQueue is also thread-safe since its methods : take and put , will patiently wait for elements inside the queue instead of returning null or throwing any exceptions. In this manner, thread execution is not interrupted when the queue is empty.

3.Project Diagram

The following image presents the overall diagram of the project, connections between classes, means of communication between classes, etc.

More detailed diagrams and more explanations will be given in the section number 4 (Classes used explained).



4. Classes used

Client

The client class is simple and typical. It only has the arrival time of the Client(in the simulation) and the task time duration (for how long it will stay once it is in the head of the queue).

Its only methods are setters and getters since no other modifications with its data is needed.

Server

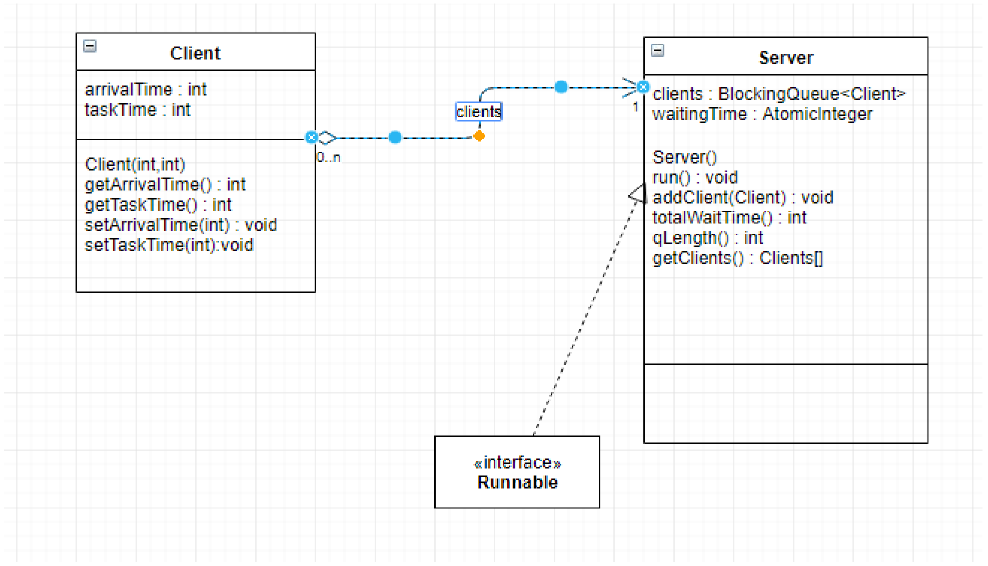
The server class presents more complexity since it implements the Runnable interface and it performs concurrent tasks. Its connection with the Client class is done through the BlockingQueue, in which we store the clients waiting at the server.

In the method run(), the server extracts the head of the queue and it “processes given task” meaning the thread sleeps for as many seconds as the client needs in order to leave. Then, it removes the client from the queue to allow the next one in line to come forward, and so on.

The use of BlockingQueue is crucial in this task due to the fact that the thread is running for a given amount of time and the queue might not have clients always.

The other methods of the Server class are mostly utility methods, in order to compare them with other servers or to set/get values or sizes.

The client and Server class are strongly connected :

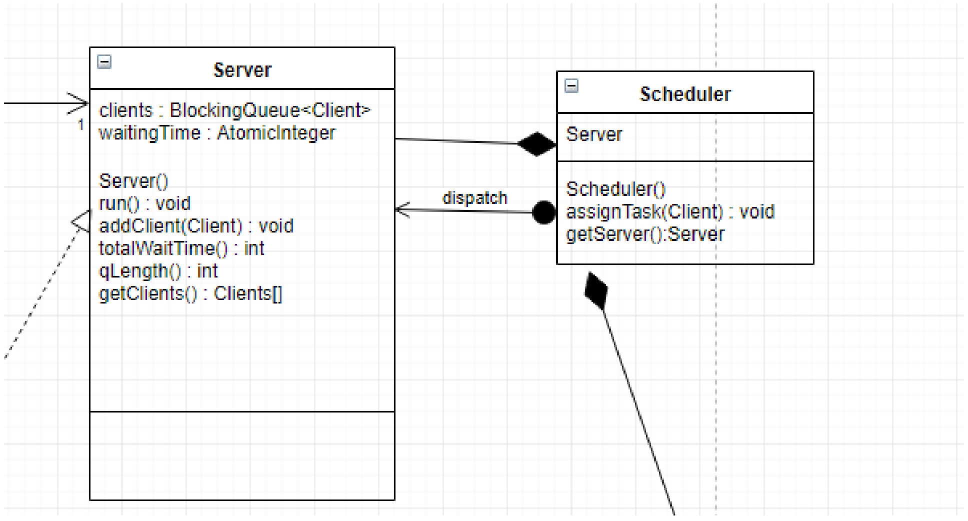


Scheduler

The scheduler is entirely based on the Server. Although not essential, this class offers transparency to the code, helps to a better understanding of the functioning of the program and provides stability for the multi-threading strategy.

The scheduler takes one server, offering it a thread and it starts the said thread. Since the simulation needs a thread of its own it is much more easier to put the runnable object server in a thread through a different class rather than just doing it in the simulation. Basically, it avoids confusion between threads and provides cleaner code.

This class is in a strong relation with the Server that it operates on, but it is also a component to be initialized in the final simulation.

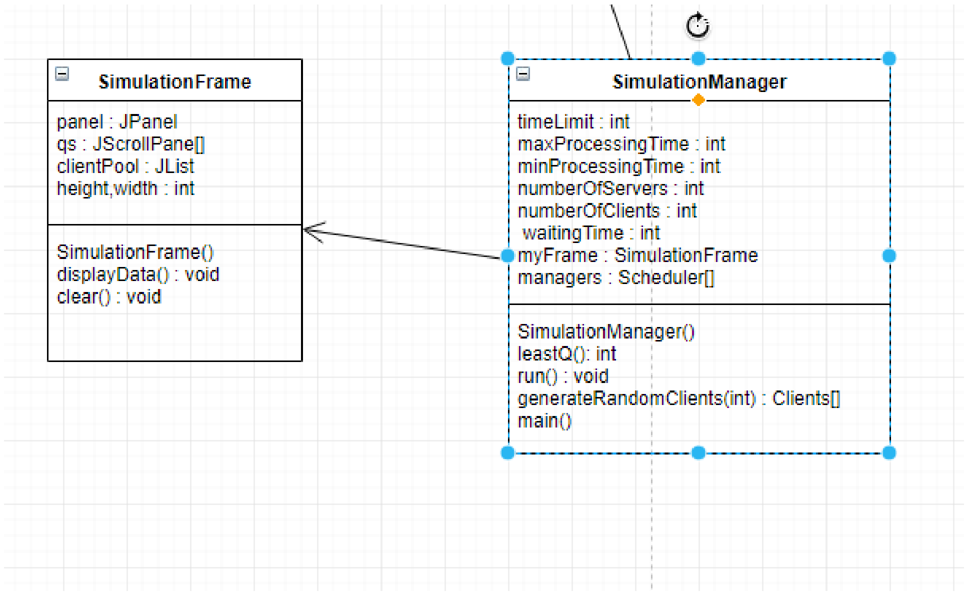


SimulationFrame

This class represents just the UI for the simulation. As I mentioned before in this documentation, I use a JList for the randomly generated clients and a JScrollPane for the queues. The number of servers is given, but the simulation time and the minimum and maximum processing number can be chosen from the UI.

Since the threads are constantly running, the UI is updated in the run() method in order to repaint the whole panel and display the current situation.

Moreover, the average waiting time is also displayed on the frame, giving the user enough control over the simulation but also the useful information that can be obtained and analyzed through these running tests.



SimulationManager

The simulation manager “paints the big picture” in the entire program. The simulation manager gets the scheduler of the servers performing the tasks, generates the clients randomly and updates the UI.

It also has utility methods such as leastQ(), determining which queue is the shortest such that the client will wait as little as possible. LeastQ() sorts the queue by waiting time, not by queue size.

The simulation manager is also run in a thread because it needs to simulate how each second passes and how clients are assigned to desks only when they arrive (I.e. if the arrival time = = current time).

This class contains the main method and orchestrates all the other classes in the project in order to create the final product, which is the final simulation observed on the GUI

<u>5.Main methods</u>

The most important methods of this projects are the run() methods, both from the Server class and the Simulation Manager class. They operate on the threads, simulate the real life interaction by delaying and waiting, and in the Simulation class it also updates the GUI permanently such that the data displayed is accurate as humanly possible. Moreover, the run() methods represent the logic behind the whole concept of each thread. The thread does not exist without the run method and therefore its outstanding importance is visible in the code as well. The run methods contain the most complex and well written operations, such that the simulation is done properly.

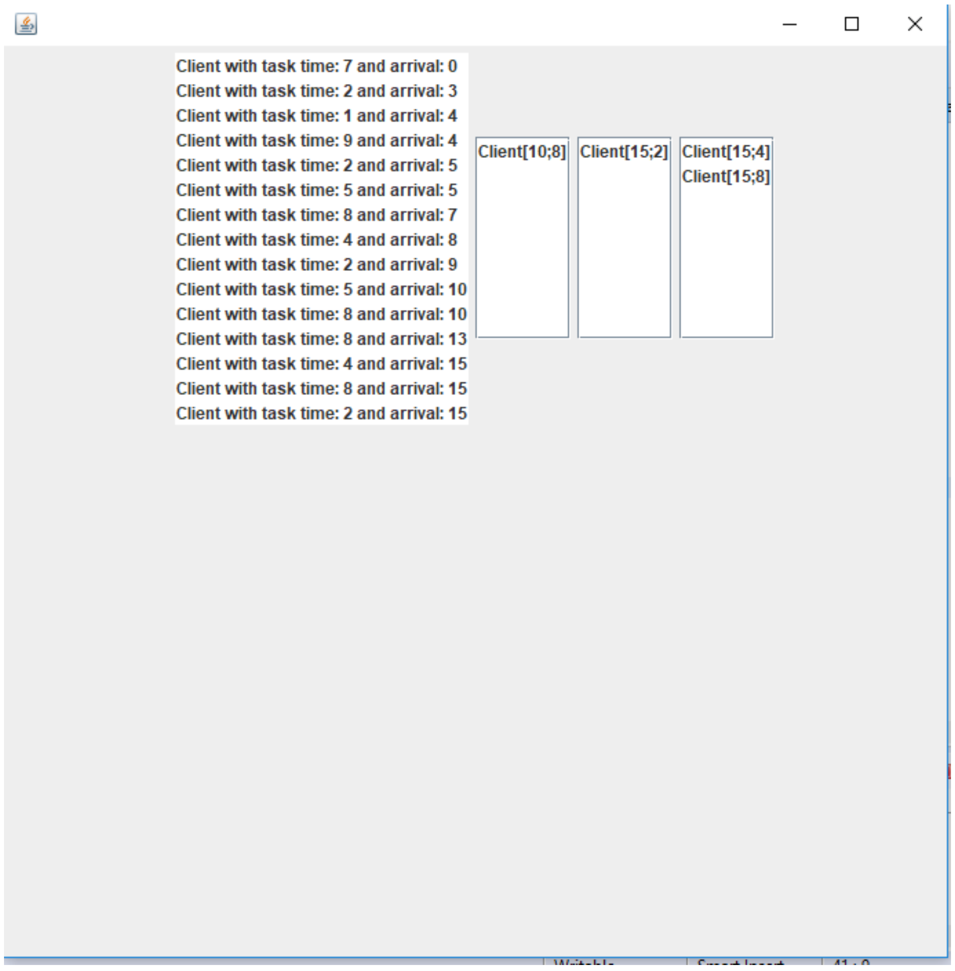
In the run() of the server, the client is popped from the head of the q and dealt with, meaning the q waits for its task to be finished.

In the run() of the simulation the generated clients are dispatched to the appropriate server(considering time) whilst keeping track of the current time.

Another important method to the project is the generate random clients method, where the class Random from java.util is used. In the method the first thing generated are integers, bounded by the limits imposed by the user. The arrival time generated is strictly smaller than the simulation time and the task time is anywhere between the minimum processing time and the maximum processing time, also imposed by the user. The two integers generated are then used in the constructor of a new Client and thus a new client is basically “generated” by this method.

6. GUI

This is how the GUI currently looks.(there is room for improvement)



```
Client with task time: 7 and arrival: 0
Client with task time: 2 and arrival: 3
Client with task time: 1 and arrival: 4
Client with task time: 9 and arrival: 4
Client with task time: 2 and arrival: 5
Client with task time: 5 and arrival: 5
Client with task time: 8 and arrival: 7
Client with task time: 4 and arrival: 8
Client with task time: 2 and arrival: 9
Client with task time: 5 and arrival: 10
Client with task time: 8 and arrival: 10
Client with task time: 8 and arrival: 13
Client with task time: 4 and arrival: 15
Client with task time: 8 and arrival: 15
Client with task time: 2 and arrival: 15
```

```
Client[10;8]    Client[15;2]    Client[15;4]
                                Client[15;8]
```

7. Possible future improvements

First and foremost, a great improvement and wonderful strategy would be to have two ways to decide which queue to assign the client to. First of all, the main focus should be on which q has the smallest waiting time, but in the case all the queues have the same waiting time then the client would go to the queue q\with the smallest size, such that the clients are distributed as evenly as possible. Implementing a strategy pattern that does this and deciding at run time which of the manners to adopt would be a wonderful way to treat such cases and would provide clean code. Moreover, there should be a settable maximum number of clients at a queue (to avoid overflow, even if graphically speaking), and to check if the queue is at full capacity and in case it is to assign the client to the next appropriate queue.