

*Faculty of Automation and Computer Science*

**Queues Simulator**

Project made by  
Ștefan Ciuprina

Contents

[1. Objective 3](#_Toc36498832)

[2. Problem analysis, scenarios, use cases 3](#_Toc36498833)

[3. Project Design 4](#_Toc36498834)

[3.1 Approach 4](#_Toc36498835)

[3.2 UML Diagram 5](#_Toc36498836)

[3.3 Classes’ Design 6](#_Toc36498837)

[3.4 Data Structures 6](#_Toc36498838)

[4. Implementation 6](#_Toc36498839)

[5. Results 9](#_Toc36498840)

[6. Conclusions 9](#_Toc36498841)

[7. Bibliography 9](#_Toc36498842)

# Objective

The objective of this project is to create a program that will simulate queue based systems for determining and minimizing clients’ waiting time. The following list contains the secondary objectives which will make possible the creation of this project:

* Generating random clients to participate in the simulation, having as constrains a set of given parameters representing the limits for when a client should arrive and how much time the processing of the respective client will take (**chapter 4**);
* Creating multiple threads: a thread for each queue, a thread for managing the waiting clients and a thread for keeping the general time of the simulation (**chapter 2**);
* Displaying the average waiting time at the end of the simulation (**chapter 4**);
* Reading the required input from a text file and displaying it in another text file, as well as in the command prompt for seeing the simulation in real time; the input and output files will be given as command line arguments by running the .jar file (**chapter 4** and **chapter 5**).

# Problem analysis, scenarios, use cases

The application reads from the input file the simulation parameters: the number of clients, the number of queues, the maximum time of the simulation (the simulation could end earlier if all the clients have been processed) and the bounds of the arrival time and the service time. According to these parameters, the clients will be randomly generated, each having an unique ID (ranging from 1 to the total number of clients), an arrival time (at what time of the simulation, the client can enter a queue) and a service time (how many seconds of the simulation it takes for the client to be processed).

After reading the user input, the simulation will start in parallel a number of queues (the number of queues specified in the input file) and start processing clients as soon as the simulation time matches their arrival time. If there are no clients at a specific time for a queue, it will be closed until more clients will become available.

When a client’s time comes (its arrival time is equal to the simulation time), it will be sent to the queue which will serve it as soon as possible. This is done by sending the client to the queue with the smallest waiting time (the waiting time for each queue represents the sum of the service times of all the clients in the queue).

When a client is being processed, its service time is reduced by one at each second. After the processing of a client is done, it is removed from the queue it was in and the next client will be processed.

The simulation ends when either the maximum simulation time specified in the input file is reached or when there are no more waiting clients or clients in queues.

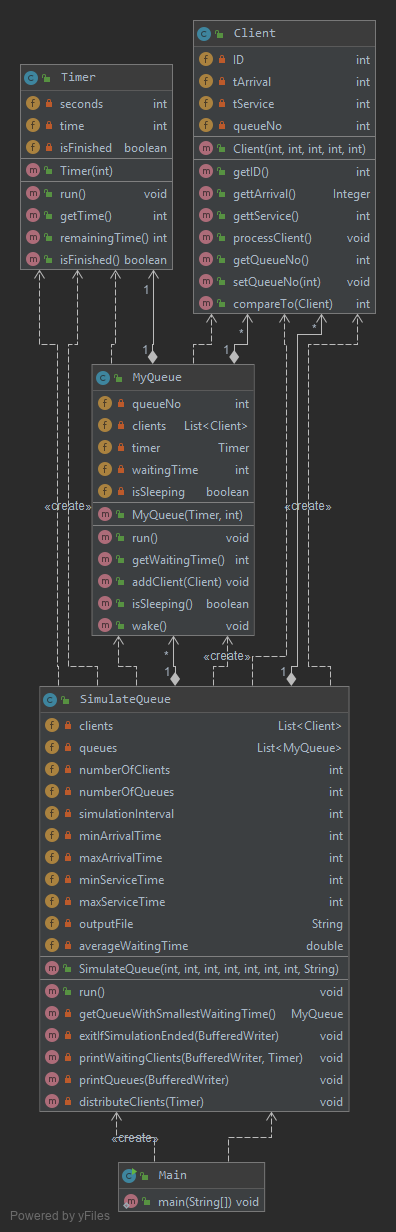
# Project Design

## Approach

The program will begin by reading the user input from the file specified as command line argument. Then, it will create a thread, namely *SimulateQueue*, which will take the data which was read from the input file and create a list of all customers. Then, the number of queues specified in the input file will be created as threads, as well as a timer thread which will keep the time of the simulation. After this, a loop will be executed each second, deciding when a customer can enter a queue based on its arrival time. It will be sent accordingly to the queue with the smallest waiting time. Also, still being in the “one-second” loop, it will print to the console and to the output file the current situation of the program, namely the time of the simulation, the waiting clients, and the queues (with their respective clients).

Each client thread will process, at each second, the client at the front. The process consists of reducing the client’s service time at each second and eliminating it after it has finished. When there are no clients in the queue, the thread will sleep until the end of the simulation, when it will end, unless it is interrupted by another client entering the queue.

## UML Diagram



## 3.3 Classes’ Design

The project is made up of 5 classes: *Main, SimulateQueue, MyQueue, Client, Timer*. As presented before, the *Main* class reads the user input, *SimulateQueue* creates the clients, manages them while they are waiting and displays the output, *MyQueue* processes the queue, *Client* holds information regarding a client and *Timer* holds the simulation time. More information regarding each class is presented in **section 4**.

## 3.4 Data Structures

All the clients generated first are stored in an *ArrayList*. Also, each *MyQueue* thread has an *ArrayList* of the clients being present in the respective queue. The client processed is always the first client in the list, the new clients being added at the end of the list.

# Implementation

In this section I will present each class in the project. The classes which have no access modifier written are considered *public*.

* *queuesim* package:
  + *Client* class
    - fields: ID (int), tArrival(int), tService (int), queueNo (int);
    - constructor: Client(int ID, int minArr, int maxArr, int minSrv, int maxSrv) – sets the ID of the client with the given ID, tArrival with a random value between minArr and maxArr, tService with a random value between minSrv and maxSrv, and queueNo to 0 (this field represents the queue where the client is – or will be – processed; if the client’s process by a queue is finished, this value will be set to -1);
    - methods:
      * double getID(), int gettArrival(), int gettService(), int getQueueNo() – getters;
      * void processClient(double value) – decrements the *tService* by one;
      * int compareTo(Client o) – is used for sorting the clients by their service time; this feature is used by the *SimulateQueue* class;
  + *Timer* class – extends *Thread*
    - fields: seconds (int), time (int), isFinished (boolean);
    - constructor: Timer(int seconds) – sets the number of seconds the timer has to count to; sets the *isFinished* filed to false (this field turns true when the timer counted to the number of seconds it requires) and the *time* filed to 0 (this represent the actual count of the timer, starts and 0 and ends when it reaches the value of *seconds*);
    - methods:
      * void run () – executes the code that the thread must do when it is started; loops the increment of *time* field by one each second until it reaches the value of *seconds*; when the loop is finished, the value of the *isFinished* field is set to *true*;
      * int getTime() – getter for *time* field;
      * int remainingTime() – returns the difference between *seconds* and *time* fields (i.e. the remaining number of seconds until the timer has finished counting);
      * boolean isFinished() – returns the value of the *isFinished* field;
  + *MyQueue* class – extends *Thread*
    - fields: queueNo (int); clients (List<Client); timer (Timer); waitingTime (int); isSleeping (boolean);
    - constructor: MyQueue(Timer timer, int queueNo) – sets the *timer* and the *queueNo* fields with the values given as parameters; initializes the *clients* list with an empty *ArrayList*, the *waitingTime* to 0 and the *isSleeping* field to *true* (this field marks that the current thread is currently sleeping);
    - methods:
      * void run() – executes the code that the thread must do when it is started; sleeps for 0.2 seconds the first time it is called to produce a small delay between itself and the timer to prevent out of sync problems (it reads the value the timer has and if the timer is just switching its value, problems may arise); then it loops each second as long as there are clients to process (the clients are stored in an *ArrayList*, the new clients are added at the end of the list and the client at index 0 is the once processed) and processes them by reducing their service time by 1 or by eliminating them (setting them as finished by removing them from the queue *ArrayList* and setting their *queueNo* to -1) if their service time is 1; also, each time when processing a client, decrements its waiting time by one; if there are no more clients to process, the thread will be put to sleep until the end of the simulation (unless it is interrupted by a new client being added to the queue), when it will finish completely; when the thread is sleeping, the value of the *isSleeping* field will be set to *true*;
      * int getWaitingTime() – getter of the *waitingTime* field;
      * void addClient(Client client)– adds a client to the list of clients of the queue; it also sets the value of the *queueNo* field of the client to the value of the *queueNo* of the current queue; it also adds the service time of the client to the total waiting time of the queue;
      * boolean isSleeping() – returns the value of the *isSleeping* field;
      * void wake() – interrupts the sleeping thread to make it continue process its clients;
  + *SimulateQueue* class – implements *Runnable*
    - fields: clients (List<Client), queues (List<MyQueue), numberOfClients (int), numberOfQueues (int), simulationInterval (int), minArrivalTime (int), maxArrivalTime(int), minServiceTime (int), maxServiceTime (int), outputFile (String), averageWaitingTime (double);
    - constructor: initializes the lists *clients* and *queues* with empty *ArrayLists*, the *averageWaitingTime* with 0, the rest of its fields being set with the values given as parameters to the constructor;
    - methods:
      * void run() – executes the code that the thread must do when it is started; creates the number of clients given by the *numberOfClients* field and sorts them according to their arrival time (this will help when giving the output to the user); creates the number of queues given by the *numberOfQueues* field, creating and starting a *MyQueue* thread for each one of them; creates and starts a *Timer* thread; as long as the timer hasn’t finished, it will distribute the clients to their specific queues using the *distributeClients* method, it will stop the simulation if there are no more waiting clients using the *exitIfSimulationEnded* method and will print the output to the user (using *printWaitingClients* and *printQueues* methods) in the output file as well as to the command window; then it will put the current thread to sleep, making the current loop execute itself only once at each second; after finishing the simulation, it will output the average waiting time;
      * private MyQueue getQueueWithSmallestWaitingTime() – returns the queue with the smallest waiting time; this method is used by the *distributeClients* method when deciding where to send a client;
      * private void exitIfSimulationEnded(BufferedWriter writer) – checks if all the clients have been completely processed (they all have their *queueNo* field -1) and if so, prints the average waiting time and exits the program, stopping the simulation;
      * private void printWaitingClients(BufferedWriter writer, Timer timer), printQueues(BufferedWriter writer) – these methods display the waiting clients and the ones present in each queue to the output file and to the command window;
      * private void distributeClients(Timer timer) – goes through all the waiting clients and if the simulation time is greater than a client’s arrival time, the respective client is sent to the queue with the smallest waiting time; if the respective queue is sleeping it will be woken up so it can start working;
  + *Main* class – has the ***main*** method, which gets the input from the input text file given as argument and parses it, then it starts a *SimulateQueue* thread, giving its necessary parameters (as well as the output file string which will be used by the thread to output to the file; it will be saved in the *outputFile* field of the *SimulateQueue* class);

# Results

The program has been tested with the *in-test-1.txt*, *in-test-2.txt*, *in-test-3.txt* input fields and behaved as expected. To test with any input text file, the .jar file of the program was created, making possible the run of the program through the terminal with the following command:

*java -jar QueuesSimulator.jar INPUT\_FILE.txt OUTPUT\_FILE.txt*

*INPUT\_FILE.txt* and *OUTPUT\_FILE.txt* can be any text files, as long as they are given in this order, otherwise an error will be displayed on the screen (if the input file doesn’t exist or it doesn’t provide the necessary data). The output file doesn’t have to be created before as the program will automatically create it under the name given in the command line.

# Conclusions

All of the objectives presented in **section 1** have been achieved, the final form of the project being the wanted queue simulator capable of simulating the management of clients with a given number of queues. By working on this project, I familiarized myself more with the java threads and I improved my overall java knowledge. Further improvements could be done to this project in the future, such as user interface and change of the simulation time step (instead of one second).

# Bibliography

For learning to work with threads:

* <http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_2/Java_Concurrency.pdf>
* <https://www.youtube.com/watch?v=TCd8QIS-2KI>