FACULTY: Faculty of Automation and Computer Science

SPECIALTY: Computer Science

DISCIPLINE: Programming Techniques

PROJECT: Queue Simulator

Student:

Valcauan Laura Maria

CONTENT

1. Objective

1.1 Main Objective

Design a simulation application in which the desired result aims to analyze queuing based systems for determining and minimizing clients' waiting time. The system is used to model real world domains.

1.2 Secondary Objectives

- Minimize clients' waiting time
- Handle multiple queues at a time (multithreading)
- Assure thread safety
- Open/close threads dynamically
- Compute average waiting time
- Keep track of simulation time
- Track time of clients while in queue
- Read data from input file
- Print results to an output file
- Application should be able to be run using java -jar command
- Multiple input tests should be able to run

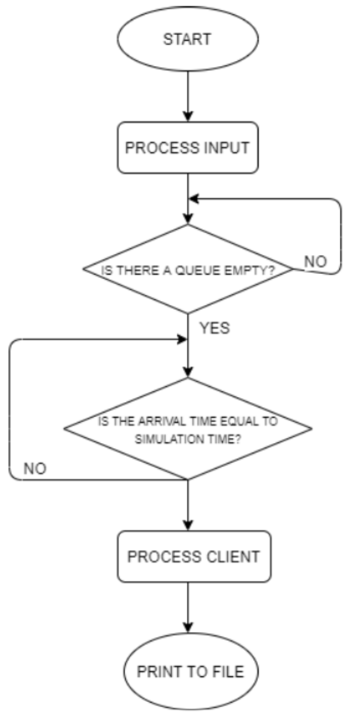
2. Problem Analysis

The use of the application implies the creation of a text file with metadata about the system. That data is processed by the system, generating randomized clients which are inserted in queues based on an algorithm. The output will be presented in a text file, giving on-clock updates with the queues and clients.

2.1 Modeling

Once the application is started, the program gathers the metadata about the system and generates clients and queues based on that initial information. The clients are order with respect to their arrival time, and then placed in the queues accordingly. While the queues are full, no client can enter, so their arrival time will be surpassed. Once a queue is free, the next client will be taken according to their arrival time; if the queues are all empty and the next client has a higher arrival time than the current time, that client will wait until the right time. The system stops once the time is over or is all the clients have been handled.

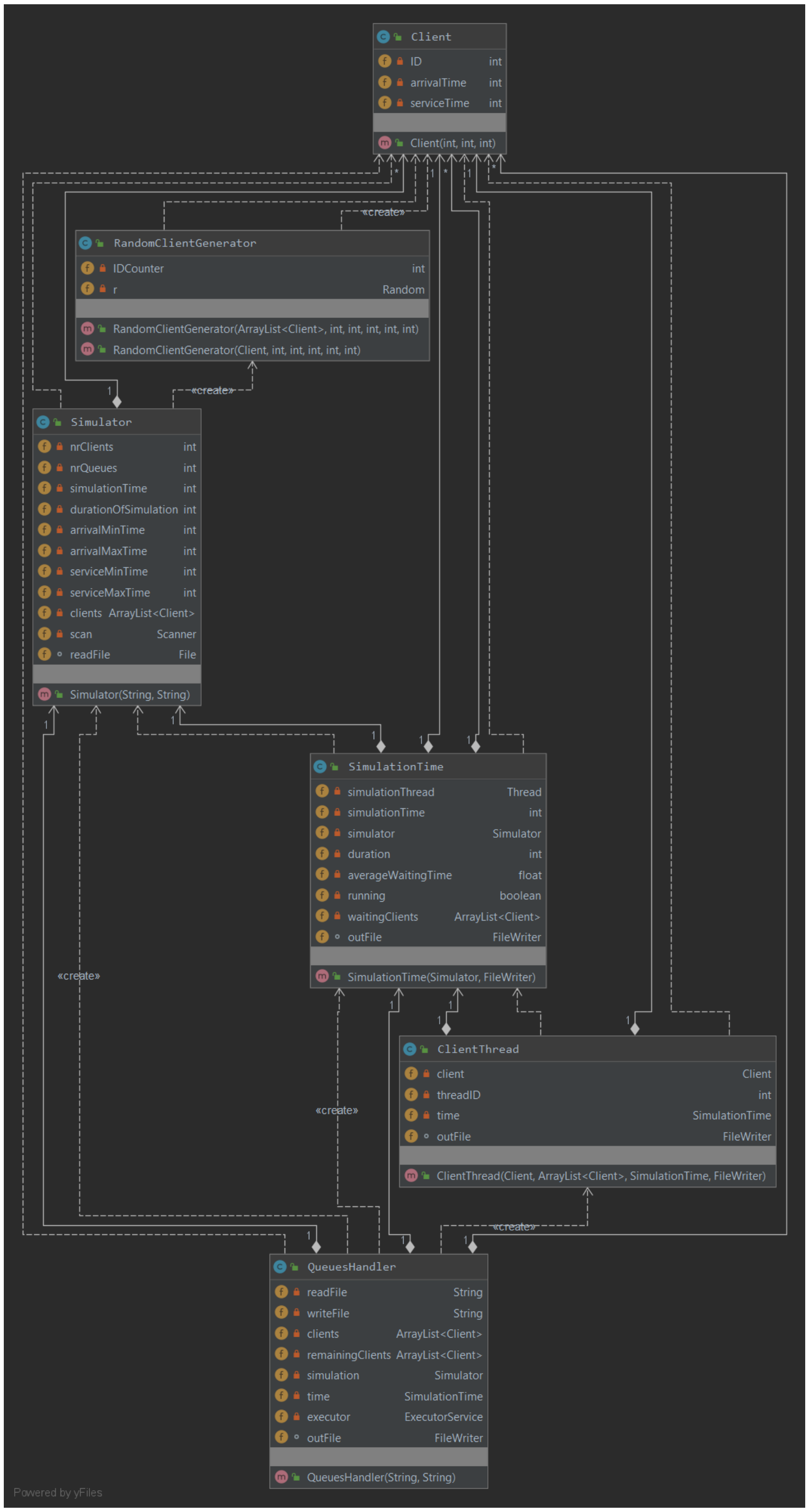
2.2 Flow Chart Diagram



3. Design

The system is designed using multiple classes, which each hold data to be used for the application, along with runnable classes to implement threads. All input metadata is inserted into a class Simulator, which processes it and creates the system accordingly to the requirements. This class communicates directly with the Queue Handler, which actually starts and stops all threads, with the use of the Simulation Time class, which is a Runnable class that keeps track of the time. The Queue Handler, after it generates the data and starts the clock, it starts all the threads, which are implemented in the Client Thread Runnable class. Once the client enters the Queue thread, it waits until the service time passes, only then the Queue is ready to accept another client. When all the clients are processed, the time thread is stopped, and the queue threads are all finished. Every time a second passes, the output file is updated, adding details about what is happening within the system; all of this is happening using a FileWriter.

3.1. UML Diagram



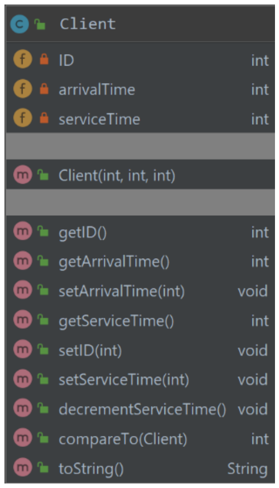
3.2. Data Structures

The data structure used was ArrayList, which was used to hold the array of Clients in order to traverse all of the clients waiting to be processed. The main benefits of using ArrayList is the time which it takes to make operations: get(), add() both are done in O(1). This allows fast random read access, and facilitates the operations needed to be performed on the polynomials.

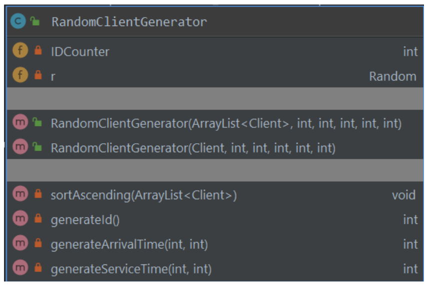
3.3. Class Design

Client

This class is meant to hold the data about the clients. It has a decrementServiceTime() method, which is meant to be used within the queue thread to better process the client. The compareTo() method is a helper function to sort the array of clients in order of their arrival time, as to insert each client in order.



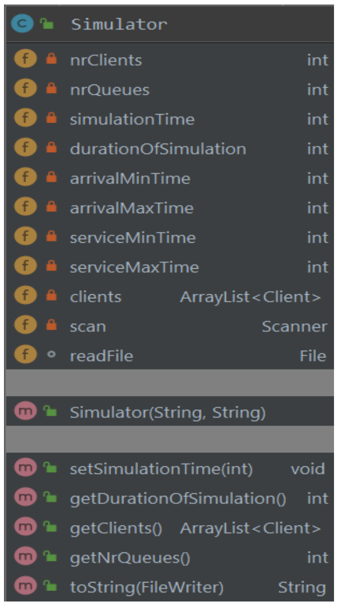
```
Client
ID                       int
arrivalTime              int
serviceTime              int

Client(int, int, int)

getID()                  int
getArrivalTime()         int
setArrivalTime(int)      void
getServiceTime()         int
setID(int)               void
setServiceTime(int)      void
decrementServiceTime()   void
compareTo(Client)        int
toString()               String
```



```
RandomClientGenerator
IDCounter                                                int
r                                                        Random

RandomClientGenerator(ArrayList<Client>, int, int, int, int, int)
RandomClientGenerator(Client, int, int, int, int, int)

sortAscending(ArrayList<Client>)                         void
generateId()                                             int
generateArrivalTime(int, int)                            int
generateServiceTime(int, int)                            int
```

Random Client Generator

The RandomClientGenerator class creates the array of clients based on the information received from the input file. It generates the arrival and service time random, in the interval read from the input file. It then sorts ascending the clients according to their arrival time.



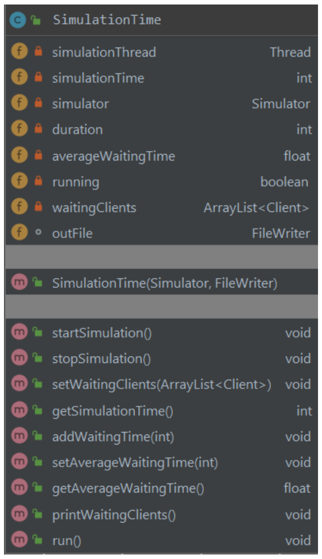
```
Simulator
nrClients                    int
nrQueues                     int
simulationTime               int
durationOfSimulation         int
arrivalMinTime               int
arrivalMaxTime               int
serviceMinTime               int
serviceMaxTime               int
clients          ArrayList<Client>
scan                     Scanner
readFile                    File

Simulator(String, String)

setSimulationTime(int)      void
getDurationOfSimulation()   int
getClients()   ArrayList<Client>
getNrQueues()                int
toString(FileWriter)      String
```

Simulator

The Simulator class is the one where all the data is read from the input file, in order to generate accordingly the clients and the number of queues. It uses a scanner which reads data from the file

Simulation Time

This class creates the time instance, which is the object every action takes after. It also holds the remaining clients that have to be processed, in order to print them to the output file. This thread can be stopped by calling the stopSimulation() method, which brings the system to an end no matter what the current time is at the moment.

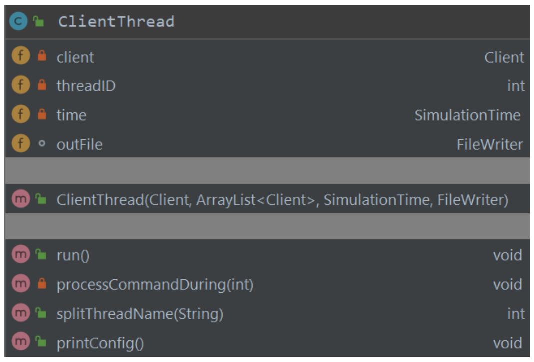


SimulationTime

simulationThread	Thread
simulationTime	int
simulator	Simulator
duration	int
averageWaitingTime	float
running	boolean
waitingClients	ArrayList<Client>
outFile	FileWriter
SimulationTime(Simulator, FileWriter)	
startSimulation()	void
stopSimulation()	void
setWaitingClients(ArrayList<Client>)	void
getSimulationTime()	int
addWaitingTime(int)	void
setAverageWaitingTime(int)	void
getAverageWaitingTime()	float
printWaitingClients()	void
run()	void

Client Thread

This is the runnable class which handles one queue. It receives a client at the appropriate time, together with the time instance, and hold the client while decrementing the service time, in order to release it when it is done.

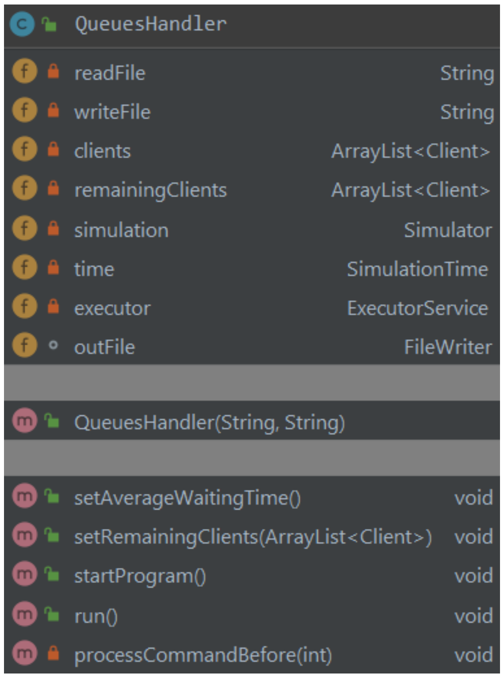


ClientThread

client	Client
threadID	int
time	SimulationTime
outFile	FileWriter
ClientThread(Client, ArrayList<Client>, SimulationTime, FileWriter)	
run()	void
processCommandDuring(int)	void
splitThreadName(String)	int
printConfig()	void

Queues Handler

This is the main controller of the system, this is where the input file is called to be processed, where the time instance is started and where the queue thread are handled. This uses a Thread Pool Executor to start and stop the threads, according to the arrival time of the clients. At the end of the application, if either the time has stopped or the clients are all processed, the executor of the threads is closed, the time is stopped and the system exits the program.



QueuesHandler

readFile	String
writeFile	String
clients	ArrayList<Client>
remainingClients	ArrayList<Client>
simulation	Simulator
time	SimulationTime
executor	ExecutorService
outFile	FileWriter
QueuesHandler(String, String)	
setAverageWaitingTime()	void
setRemainingClients(ArrayList<Client>)	void
startProgram()	void
run()	void
processCommandBefore(int)	void

3.4. Algorithms

The application uses a variety of methods and functions in order to enhance the functionality of the system, to ensure thread safety and client serving. The next algorithms presented belong to those elements of the application which require a higher level of diligence:

- *Thread Pool Executor*

  The thread pool is a collection of pre-initialized threads, which facilitates the execution of N number of tasks using the same threads, in our case, the same queues. If there are more clients than queues, or the queues are all occupied, the clients (tasks) await in a queue like structure (FIFO). When one of the queues finish its execution, it takes a client from the queue and handles it. The executor is responsible for the execution of the Runnable objects; while waiting for a client, the threads (queues) are stopped, this ensures the thread safety. When the executor is stopped, all threads are destroyed.

- *Starting the Queues Handler*

  This method ensures the initialization of every needed component of the application; this is where the input file is read and distributed accordingly. After the reading, the clients are generated based on the requirements read from the file, together with the start of the clock thread, which has a limited running time, also read from the input file. The Thread Pool Executor is started, having as parameter a known number, which starts the queues allocated for one particular system; one by one, each client, which is ordered based on their arrival times, will enter one of the queues. If there is an empty queue, the client must wait until his arrival time is due to enter the queue; while the client waits, the queue thread will be stopped.

- *Average Waiting Time*

  This is calculated in two parts: the first one, where before the simulation thread starts, it adds from each client its service time; the second part, which happens during the *ClientThread,* where the clients are processed, it checks if the client got there in time (arrivalTime = simulationTime), if not, it adds to the total waiting time the difference between those two. At the end, once the queue threads are all executed, the average waiting time is set, by dividing the total waiting time to the number of clients.

4. Implementation

In this chapter I will describe the functionality of each class and each method within that class in order to facilitate the readers comprehensibility. I will not mention the getters and the setters of these classes.

- *Client:* this class represent the data required for a client

  compareTo(): helper function to order the clients according to their arrival time
  decrementServiceTime(): when the user is processed in the queue, its service time decreses, in order to update the period of time it needs to remain in the queue
  toString(): helper function to get the configuration of each client of the system

- *Random Client Generator:* this class generates the clients in a randomized fashion, having as restrictions the values given in the input file
  - *generateId()*: gives to each of the clients created an ID, keeping track of how many are supposed to be, and how many are already allocated;
  - *generateArrivalTime():* has two values (min, max) as being the bounderies, and it generates randomly a value in that interval to be the arrival time of the new client;
  - *generateServiceTime():* has two values (min, max) as being the bounderies, and it generates randomly a value in that interval to be the service time of the new client;
  - *sortAscending():* with the help of the *compareTo()* function from the Client class, it orders each client from the generated array of clients with respect to their arrival time;

- *Simulator:* this class represents the transition of the data read from the input file to an instance of a class which holds all the metadata of the specific system
  - *Simulator():* the constructor is where the file is opened and read from; each line contains data about the system, which is extracted and placed in the right field; the clients are generated using the Random Client Generator, having as parameters the newly extracted information;

- *Simulation Time:* this Runnable class is where the clock is created
  - *SimulationTime():* the constructor receives as parameters the Simulator, which holds all the data, and the output file, in order to print at each step the current time of the system
  - *startSimulation():* calling this function means that the thread will start running; it sets the Boolean variable *running* as true, because this is a condition for the thread to function continuously
  - *stopSimulation():* calling this function sets the *running* to false, which stops the thread, ending the whole system, no matter if the time is finished or not
  - *printWaitingClients()*: this class receives the information about what other clients are still waiting to go in the queue, and this function prints them accordingly to the output file
  - *run()***:** the defining method for the thread; to function, the *running* variable must be true (this means someone must start the thread), then it goes to another check: the *simulationTime* (current time) must not surpass the duration of the application (the duration is the preset value which tells how long the system is supposed to run); each time the simulationTime increases, its value, together with the *waitingClients* is printed to the output file; after that, the thread sleeps for one second; if the system is not stopped before the simulationTime reaches the final duration of the program, it will exist the nested while, and the variable *running* will become false, stopping the whole thread;

- *Client Thread:* this Runnable class is where the clients are processed in queues
  - *splitThreeName():* it splits the Thread name, in order to get the ID of the current queue
  - *printConfig():* when called, it prints the configuration of the queue (the current client which is processed), together with the ID of it
  - *processCommandDuring():* it receives as parameter a value which is meant to be used as the number of seconds the thread must wait until releasing the client; in this method, the service time of the client is decremented, to

lower its waiting time, the configuration is printed to file, and the thread sleeps for one second; this happens a finite number of times

- *run():* the thread ID is set, and the total waiting time of the system is incremented, only if the client reaches the queue after its arrival time is supposed to be; it processes the client, and then it releases it

- *Queues Handler:* this Runnable class is where every component of the application is set, and where the queue threads are handled
  - *startProgram():* once this method is called, the thread is started, but before that, multiple actions take place before; the *Simulator*() is instantiated, the *time* is started, by creating a SimulationTime object; we create an *executor* which uses the Thread Pool Executor to start as many queue threads as required by the input data; once the thread ends, the time is stopped and the Average Waiting Time is set;
  - *processCommandBefore():* it receives an integer value, representing the number of seconds the current thread must wait before performing an action
  - *run():* processes each client (the array is ordered after their arrival time); before it places them in a queue, it checks that their arrival time is suitable(either the current time equals their arrival time, or it's bigger that their arrival time, which means the client did not have a place to go in at their right time), if it is, then the *remainingClients* array removes this client from the ones which are waiting; it creates a new *ClientThread*, which is then executed by the executor. When all the clients are finished, the executor shuts down;

5. Conclusions

The queue simulator implemented has all the required objectives (the main and the secondary ones) achieved. The application is able to safe operations on threads and to minimize clients' waiting time successfully. Further improvements can be done to this project by supporting a user interface or by additional functionalities of the queues and more others.

6. Bibliography

[1]. https://www.journaldev.com/1069/threadpoolexecutor-java-thread-pool-example-executorservice

[2]. https://www.baeldung.com/thread-pool-java-and-guava

[3]. https://howtodoinjava.com/java/multi-threading/java-thread-pool-executor-example/

7. Instruction of use

In order to be able to run the program with the .jar configuration, the command must be:

Must repeat this line multiple times, with different arguments: input files (in-test-1.txt, in-test-2.txt, in-test-3.txt), output files(out-test-1.txt, out-test-2.txt, out-test-3.txt) in order to obtain all the possible tests.