

Fundamental Programming Techniques

**Queues Simulator**

Assignment 2

*Hunor Debreczeni*

*Group: 30423*

Table of Contents

[Objective 3](#_Toc36775135)

[Problem Analysis, Modeling, Scenarios, Use cases 3](#_Toc36775136)

[Design 5](#_Toc36775137)

[Packages 5](#_Toc36775138)

[Controller 5](#_Toc36775139)

[Models 5](#_Toc36775140)

[Exceptions 6](#_Toc36775141)

[Strategies 6](#_Toc36775142)

[Util 6](#_Toc36775143)

[Validators 6](#_Toc36775144)

[Implementation 7](#_Toc36775145)

[Results 8](#_Toc36775146)

[Conclusions 8](#_Toc36775147)

[Bibliography 9](#_Toc36775148)

# Objective

Design and implement a simulation application aiming to analyze queuing-based systems for determining and minimizing clients’ waiting time.

Queues are used in many places, including in real-world problems and also in software related ones also. The purpose of a queue is to provide place for a task/client while it is waiting for a given service, to be processed. This waiting time can be minimized if more queues are added, so we can part the clients to multiple servers where they can wait.

The software simulates in real time, the processing of clients, that are later put in queues. It always takes the clients according to their arriving time. We don’t know at simulation time N, that who will arrive at simulation time N + 1. Each client should be put in a queue, that finishes in the fastest time possible compared to other queues. By doing this, we aim to achieve the best average waiting/ processing time in the end.

For simulating the queues, the application must manage to run them at the same time but meanwhile also allowing to set/ get data between the top-level module and the queues. This is necessary because we need to decide somehow where to put the new clients and also to be able to add them in the queue.

# Problem Analysis, Modeling, Scenarios, Use cases

The application simulates the clients, that arrive in a specific time. At that time, it should decide that which queue suits the client the best. The main problem of the application is to simulate the queues, that should run at the same time. In a given second, the clients are entered in the ascending order of their processing time, by doing this, we increase the chance of processing more clients and also having a smaller waiting average.

The **Input file** of the program must look as follows:

*N (Integer) -* Number of clients that should be randomly generated

*Q (Integer)* - Number of queues that will be used

*Tnaxsimulation (Integer)* - The maximum simulation interval

*Tminarr Tmaxarr (Int, Int)* - Minimum and Maximum arrival time (0 < Min <= Max)

*Tminser Tmaxser (Int, Int)* - Minimum and Maximum service (processing) time (0 < Min <= Max)

The **Output file** of the program looks as follows:

*Time* ***{integer}****Waiting clients: (****ID****,* ***Tarr****,* ***Tproc****); ...  
Queue* ***1****: closed/ (****ID****,* ***Tarr****,* ***Tproc****); …  
….  
Queue* ***Q****: closed/ (****ID****,* ***Tarr****,* ***Tproc****); …*

*Average waiting time:* ***{double}***

The user can read from this report the client’s status in each time interval/ second, and their waiting time. The queues current load is also presented. The last line of the whole file contains the average waiting time of those clients that have been fully processed (entered a queue, but also exited it)

The **Arguments** of the program should be used as follows:

*Java -jar software.jar* ***input output second=1000***

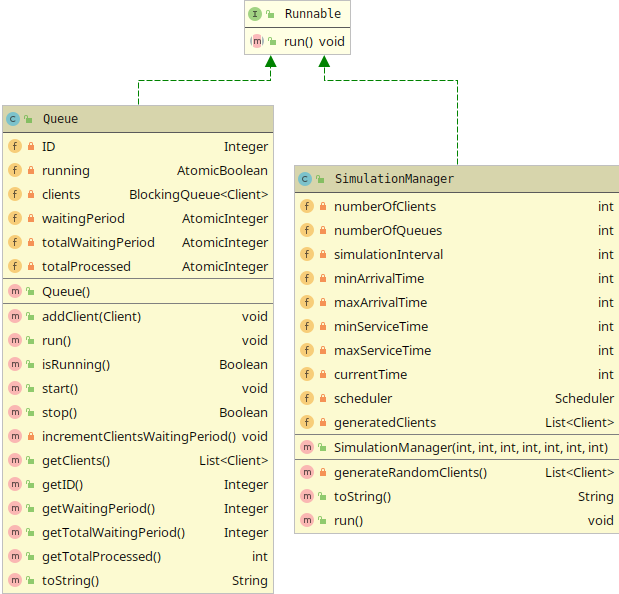
* **Input:** 
  + Specifies the name of the input file that **must exist and must be in the same folder as the .jar file**
  + **Must be a text a file**
  + **Must** **be the first argument**
* **Output:**
  + Specifies the name of the output file, where the report will be generated on program exit.
  + **Optional:** in case if it’s not entered, the output will appear in the console (STDOUT)
  + If entered, it **must be the second argument**
* **Second={integer}:**
  + Specifies how many milliseconds a second should be considered in the application (Can be used to run the program faster, if needed)
  + **Optional:** if not entered, then a second is considered as 1000 milliseconds
  + If entered, it **must be greater than 0**

The application may be used to simulate a restaurant, bank, supermarket, etc. setup. The owner may test a queue system with an additional line added to the store. He needs to:

1. Specify all the data that is needed in the input file (described above):
2. Place it next to the jar file
3. Run it through the terminal with *Java -jar software.jar input output second=1000* (JDK must be installed on the operating system (and added to the PATH))
   1. ‘java’ is not recognized / unknown command
      1. Download JDK and install it
      2. Retry
   2. InvalidArgumentsException
      1. Error will be printed out
      2. Fix Arguments (Check above)
      3. Retry
   3. InvalidInputException
      1. Error will be printed out
      2. Fix errors in the input file (Check above)
      3. Retry
4. Wait for the result to appear in the output file, or in the console if no output file was specified

# Design

For implementing the problem described above, the best approach is to use Threads. This helps us to run simultaneously the queues and also access their data, by using synchronized functions and volatile variables. It helps us, so that we can easily use one another methods or variables from different threads and at the same time without any problem. In java, one may easily implement a Thread, a class must implement the Runnable interface or extend the Thread class and afterwards implement the run() method.

In our case, we had 2 kinds of classes that implements these methods.

The **SimulationManager** class is used as a Thread, because it’s the main controller of the whole application, that simulates the time passing by. It’s achieved by making the thread sleep for a given amount of time, by default 1 second. After each sleep, it checks if there are any clients that should be added to a Queue.

The **Queue** also runs the same way, with the same sleeping amount, because we need synchronous client processing. The only difference compared to the SimulationManager class is that there can be any number of Queues, that are implicitly influenced by the SimulationManager’s Thread through the Scheduler class.

In order to make the Threads Safe, a synchronized collection is used in the Queue, most exactly the LinkedBlockingQueue. It’s a Queue that is optimized for concurrent programming in Java (working with multiple threads). The variables are also pre-defined Atomic variables, that are optimized for the same reason. The volatile variables are stored in the main memory, not in the CPU cache in order to maintain Thread safety. Beside that the Atomicity also solves the problem of bad timing between Threads. By using this, we will never happen to update a value, but get the old result back.

The Blocking Queue also helps us with the blocking of the thread. In case there are no more elements left in the list, or the maximum limit is exceeded, the current Thread will be blocked. It can be un-blocked by adding a new element or taking out one.

## Packages

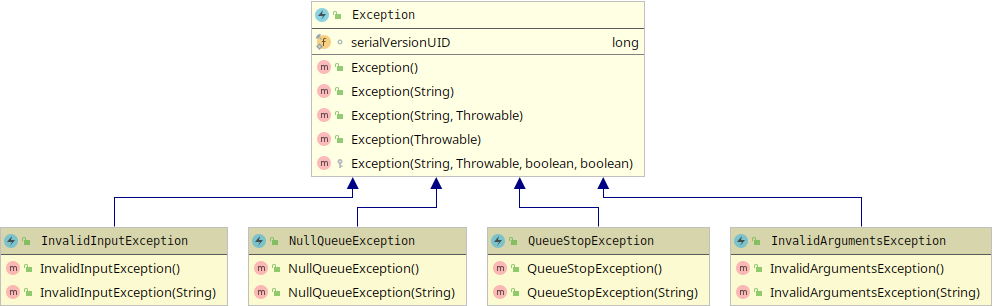
### Controller

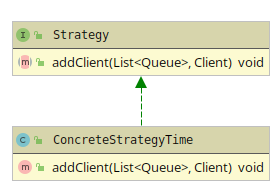
The Controller package contains three classes: Queue, Scheduler, SimulationManager. These 3 are the so-called brain of the whole application, the core of the software. It’s what connects together the front-end, which in our case should be the console/ file and the models.

### Models

The only custom model in this application is the Client that holds basic information about the given customer, that waits in a queue. Each one has a unique ID, arrival time to tell us when it will enter the queue. The processing time is used to simulate the processing of its task and a waiting time that is used to calculate the average waiting time at the end.

### Exceptions

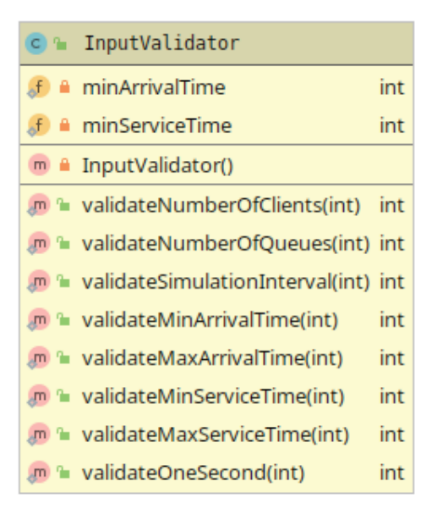
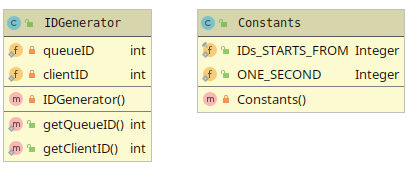
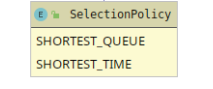
Exceptions were used for validation, that comes in handy mostly at the start of the application, but are used also at a later point in the program, to check if everything works just fine. If something goes wrong, an exception is thrown and the program alerts most of the time the developer, that something went wrong.



### Strategies

As the strategy package, it is used to implement the strategy Design Pattern. Unfortunately, this application only includes one type of strategy, and it cannot be changed. It makes further development much more easier and also easier to see and understand the project’s structure.

### Util

Mostly declares static variables and implements static functions that are used globally in the project, for generating unique IDs and to make further changes or inside changes much easier.

### Validators

They are being used the most at the start of the project. Each argument and every data from the input must be validated in order for the software to work as intended. These classes contain only static functions, in order to make their access easier from any part of the program.

# Implementation

As already been said, the core of the program is in the Controller package, that contains the *Scheduler, SimulationManager, Queue*.

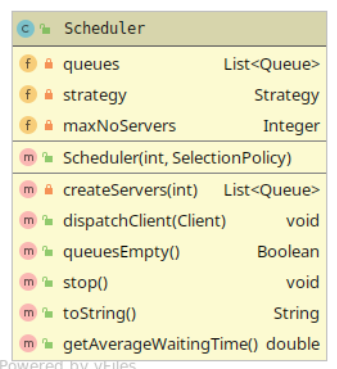
The **SimulationManager** contains the restrictions in which the simulation should be run (numberOfClients, min/ max Arrival/ Service times, etc.). With these variables, there will be generated some random Clients, which the program will use to run the test and calculate the average waiting time at the end.

But it’s most important method is the run() method, which keeps the thread running until 2 conditions are true at the same time:

1. The simulation time limit hasn’t been reached
2. One of the following is true:
   1. There are clients waiting
   2. There are clients in any of the queues

Inside the while loop, it forwards every client whose arrival time corresponds to the current time to the scheduler’s add() method, which then sorts pushes the client to a specific queue.

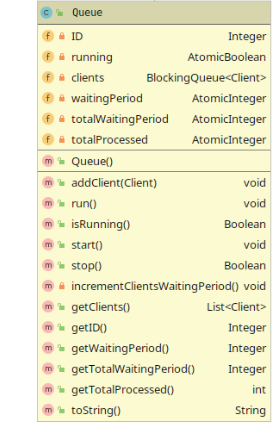
After the clients were checked, the method sleeps for one second (the second is taken from the Constants class) and increments the current time with one, that corresponds to the slept second.

If the while exists, it means that the clients were processed or it ran out of time. In this case the queue threads are also stopped.

The **Scheduler** is the only controller package class that is not a thread. It maintains the connection between the SimulationManager and each one of the Queues. It’s signature and most used function is the *dispatchClient()*.

This function sends the Client object, that must be added to a Queue for processing, to the strategy’s addClient() method, that finds the Queue with the least waitingTime, and pushes the given object to it’s queue for it to wait to get processed.

The constructor of the Scheduler creates a given no. of Queues, but won’t start them and neither has the permission to do that, it is done internally by the Queue itself.



The **Queue** actually counts down on each iteration in the run() method and decrements the processing time of the current Client if there exists one. Otherwise it just sleeps until stopped. The start of the thread is being done automatically by itself in the addClient() method. It checks if the variable isRunning is true, if not, then calls the start() method, that creates a thread from itself and starts it at the same time.

The *run()* method functions as follows:

* Checks if the variable running is true, that is done using the isRunning() method.
* If the clients list is empty, then it skips the current while iteration (enters into an infinite loop until a Client is added or the thread is stopped).
* Sleeps for a second
* Decrements the waiting period with one to process the elapsed second
* In case the current Clients waiting period gets to 0 (it means that the client has been processed), it get’s removed from the queue and the total amount of clients processed incremented and the total waiting period incremented with the clients total waiting period

These three classes each contain a toString() method, to help out in the printing. It converts the values of the object to a string, so that it’s a lot easier to print out an object.

Another useful class that is worth the mention is the Constants class, where one can change variables very easily that were used almost globally, in more than one class.

# Results

As results, the user of the application can get a very well detailed simulation on steps for queueing up N random clients. It’s easy to calculate also the average and to estimate a real-life situation also with that.

# Conclusions

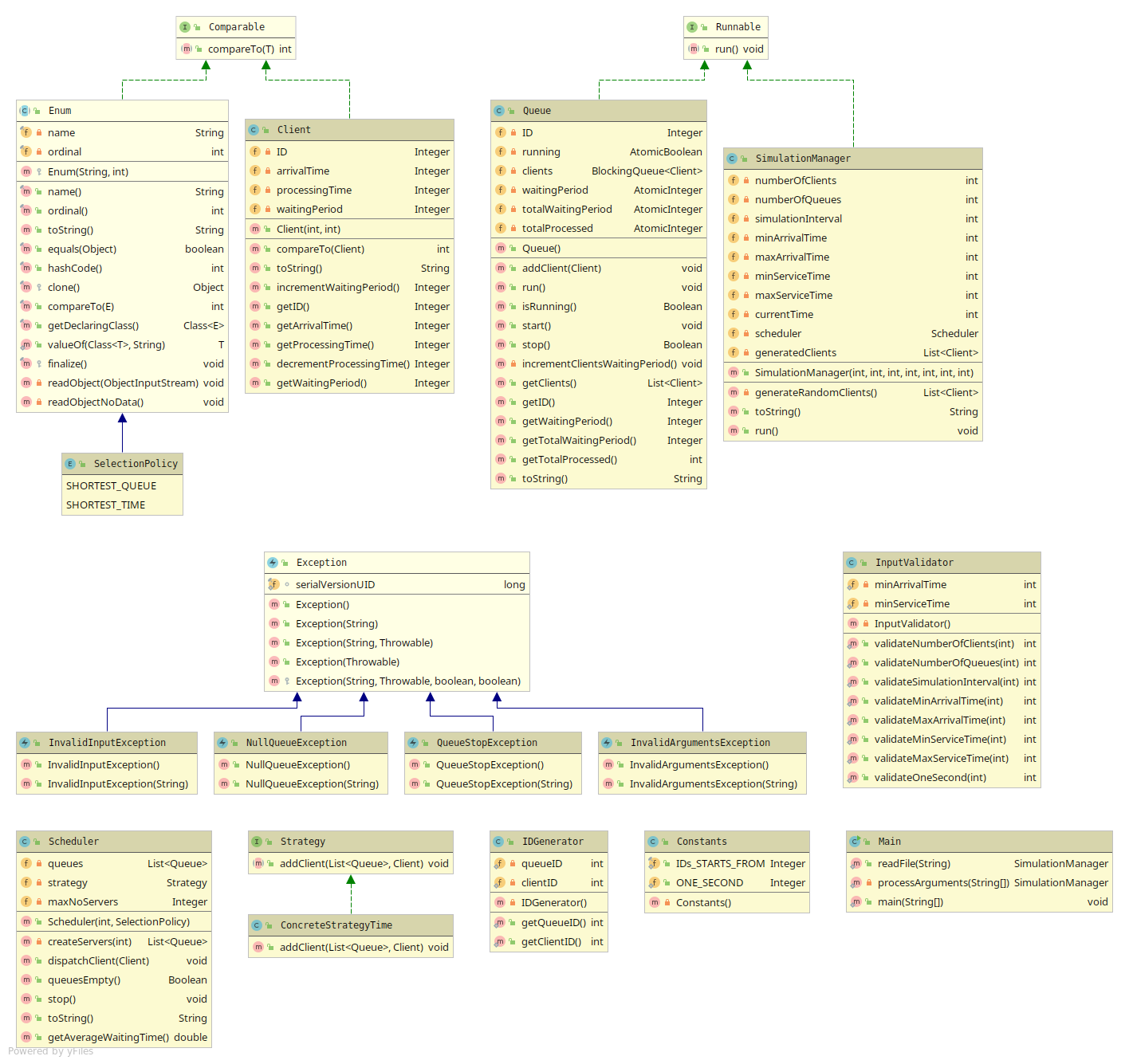
As conclusion, I would say that this software would come in handy for some smaller restaurants, hotels, motels where the owner can’t decide if there’s a worth of upgrading his/ her business.

The project for myself as the developer of it, was a good exercise and helped me understand the concept of concurrency in Java, by that I mean the use of Threads and their advantages. I learnt that time management is an important aspect of almost anything and even a small error could cause a big failure in bigger projects. This is why paying attention to thread safety is important.

Although I didn’t know much about the subject of threads, I learnt a lot. Understanding and making it work was a lot easier when I had to do the research and try out the given examples and make the connections between each element.

Lastly, there could be made some further developments in the application. As an example, by implementing new Queue selecting strategies, which at the moment is locked to the fastest time strategy.

# Bibliography

* <https://www.geeksforgeeks.org/blockingqueue-interface-in-java/>
* Java\_Concurrency.pdf