**FUNDAMENTAL PROGRAMMING TECHNIQUES**

**ASSIGNMENT 2**

**QUEUES SIMULATOR**

Anghel Dan-Marian

Group 30422

Professor: Diana Balc

**Table of contents:**

1. **Objectives**
2. **Problem analysis and use-cases**
3. **Development**
4. **Implementation**
5. **Results**
6. **Conclusion**
7. **Bibliography**

**1. Objectives**

**Primary objectives:**

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

**Queues** are commonly used to model real world domains. The main objective of a queue is to provide a place for a "client" to wait before receiving a "service". The management of queue-based system is interested in minimizing the time amount their "clients" are waiting in queues before they are served. One way to minimize the waiting time is to add more servers, i.e. more queues in the system (each queue is considered as having an associated processor) but this approach increases the costs of the service supplier. When a new server is added the waiting customers will be evenly distributed to all current available queues.

The application should simulate (by defining a simulation time 𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛) a series of N clients arriving for service, entering Q **queues**, waiting, being served and finally leaving the queues. All clients are generated when the simulation is started, and are characterized by three parameters: ID (a number between 1 and N), 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 (simulation time when they are ready to go to the queue; i.e. time when the client finished shopping) and 𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 (time interval or duration needed to serve the client by the cashier; i.e. waiting time when the client is in front of the queue). The application tracks the total time spend by every customer in the queues and computes the average waiting time. Each client is added to the queue with minimum waiting time when its 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 time is greater than or equal to the simulation time (𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 ≥ 𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛).

Input data:

* Number of clients
* Number of queues
* Maximum simulation time
* Minimum and maximum arrival time
* Minimum and maximum processing time

Output data:

* Information about the data, the program is processing
* Evolution of the queues
* Average waiting time of the clients

Both the input and the output data are provided/generated inside text files, read from the command line arguments, using a “.jar” file.

**Secondary objectives:**

* **Use-cases** (Chapter 2):  a **use case** is a list of actions or event steps typically defining the interactions between a role and a system to achieve a goal, usually represented trough a flow-chart or an UML diagram.
* **Diagrams** (Chapter 3): they provide a mock-up of the project so that the developer can more easily grasp the concept and the small details of it.
* **Data structures** (Chapter 3): the types of data structures used in making the application.
* **The classes used** (Chapter 3): the project has been split in smaller classes that can accomplish certain tasks, accordingly to the MVC model.
* **Implementation** (Chapter 4): each class will be explained, along with the graphical interface.
* **Results** (Chapter 5): these are in the form of logs inside the output text file and they give information about the program’s execution at each step.

**2. Problem analysis and use-cases**

* **General overview**

This application’s purpose is to simulate customers waiting in a queue to receive a service (i.e. mall, gas-station, etc.), just like they would in the real world. Each queue can serve a certain number of clients at the same time and they have to wait a certain amount of time.

* **Input and output**

A number of N clients are generated randomly, each with a unique ID number and arrival and processing times inside the intervals given by the user in the text file.

**The user can change:**

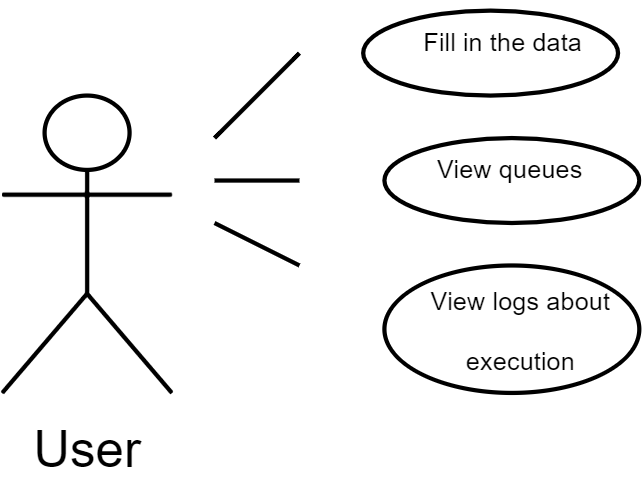
* The total number of clients to be processed.
* The maximum number of queues available.
* The maximum allowed time for the simulation.
* Minimum and maximum arrival time, i.e. the time from the start of the simulation and until the client is sent to a queue.
* Minimum and maximum processing time, i.e. the time that each client spends when they are at the head of the queue.

**The user can see:**

* The status of the waiting queue, i.e. the clients that have not finished their shopping yet (𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 > 𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛).
* The status of the service queues, i.e. each queue and the clients waiting there.
* The time of the simulation incrementally.
* The average waiting time in the queue for the clients.

(\* time is represented in seconds)

* **Use-cases**



(\* Use-case diagram of the program)

|  |  |
| --- | --- |
| Use Case 1 | Fill the data in |
| Actor | End-User |
| Basic Flow | The user will provide a text file containing the simulation parameters, which will be read from the command line. |
| Alternate Flow | In the case that the file cannot be opened or an error occurred during the processing of the file, the user will be notified and can make adjustments accordingly. |

|  |  |
| --- | --- |
| Use Case 2 | View the queues and the execution logs |
| Actor | End-User |
| Basic Flow | After the simulation has completed, the user can view the output text file that was filled in by the program (an empty text file, used for output, needs to be specified in the command line by the user as well) in order to access information such as: the status of the queues at each time of the execution or which clients gets sent to which queue. |
| Alternate Flow | In case that the output file cannot be written to, the user will be notified and he/she can verify that the specified path of the text file is valid. |

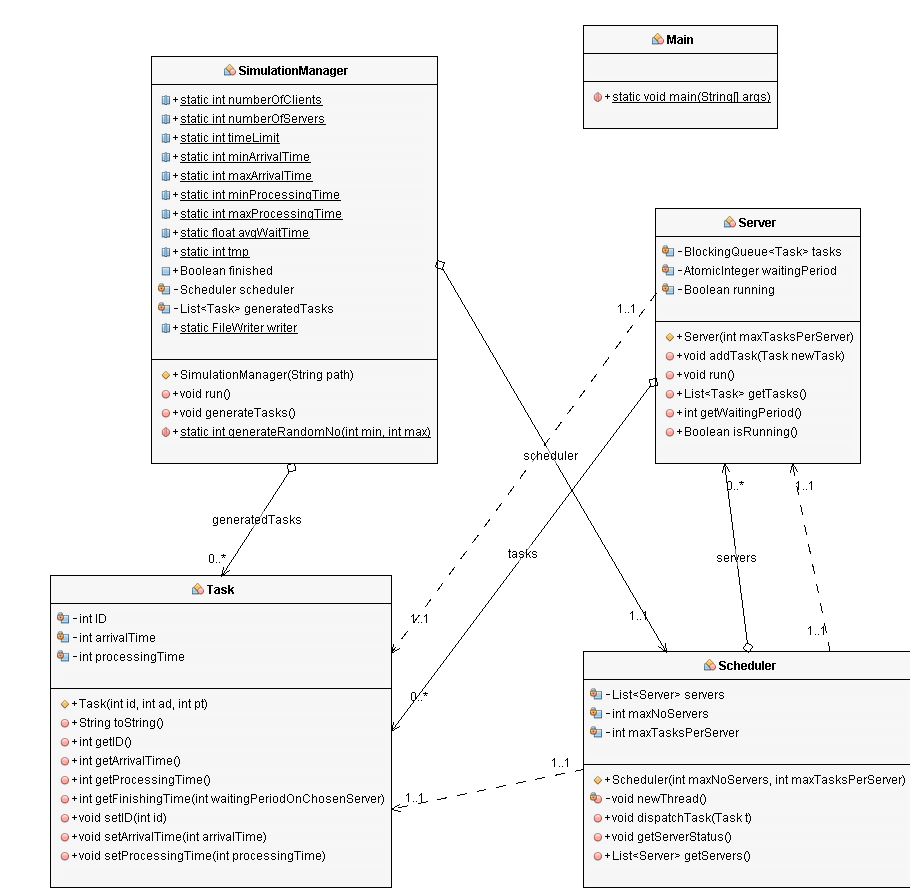
(\* Flow chart of use-case description)

**3. Development**

The project is split in 4 classes (plus a 5th main class that is responsible for the input and output and to start the simulation). These classes play the role of the client, the queue (each of these being an individual thread), a scheduler that is linked to the queues and is responsible to sending the clients to the queues and a simulation manager that holds the parameters of the simulation and synchronizes the execution of each queue under a common thread.

As the main data structure used, the blocking queue is chosen due to its implementation that supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space

to become available in the queue when storing an element. Blocking queue methods come in four forms, with different ways of handling operations that cannot be satisfied immediately, but may be satisfied at some point in the future: one throws an exception, the second returns a special value (either null or false, depending on the operation), the third blocks the current thread indefinitely until the operation can succeed, and the fourth blocks for only a given maximum time limit before giving up.



(\* Class diagram of the project)

This image has been generated using the easyUML plugin of the NetBeans IDE.

A better-quality representation can be observed at the link: <https://imgur.com/a/Q5wlaig>

**4. Implementation**

(\* describing each class and their functionality)

**4.1 Task**

This class defines the client, each with a unique ID number and arrival and processing times. It overrides the “toString()” method in order to have a nicer print of the clients’ information. It also contains a few getter/setter methods.

**4.2 Server**

This class implements the queue structure using a thread and synchronized data structures. It uses a blocking queue in order to hold the tasks and an atomic integer as a counter for the waiting period of each queue. It has a Boolean value that determines if the thread is still running or not.

The “addTask” method adds a specified task to the queue, increases the waiting period and contributes to the average wait time. The “run” method goes trough each task in the queue and puts the thread to sleep for the according processing time. Lastly some getter and setter methods.

**4.3 Scheduler**

This class holds a list of queues and is responsible for sending each client to the appropriate queue. The constructor initializes the instance variables and creates a number of queues by repeatedly calling the “newThread” method. This method creates a new queue and starts the thread corresponding to it.

The “dispatchTask” methods searches the list for the queue with the minimal waiting time and proceeds to add a task to that queue.

**4.4 SimulationManager**

  This class is controlling the process of simulation. It holds the parameters of the simulation which are made static and public since accessing them from other classes is necessary, a Boolean that tells if the simulation is finished or not, a scheduler-type object, a list of clients that are generated randomly according with the parameters and a “FileWriter” object used to generate the output file.

The constructor generates the random clients and initializes the scheduler. The “run” method uses a global counter to keep track of the time of the program. It iterates trough the list of clients and when the arrival time of a client is equal to the running time, it is sent to the scheduler to place the client in the appropriate queue. It is also responsible to write the information about the queue and clients inside the output file and to generate the average waiting time.

A static method for generating random numbers inside and interval is provided which is used to generate the clients with random attributes.

**4.5 Main**

  This class contains the main method which reads the arguments of the simulation from the input text file, and sets them inside the simulation manager. It creates the simulation manager object and starts the thread thus begging the execution of the program.

**5. Results**

The results are present in output text file in the form of information about the waiting/service queues, the average wait time and about each client at each moment of the execution of the program. A jar file is provided for easier use of the program.

For the input file “in.txt”:

4  
2  
60  
2,30  
2,4

We get the following output file “out.txt”:

Time 0  
Waiting clients: (0,9,3);(1,28,4);(2,8,3);(3,13,3);  
Queue 0: closed  
Queue 1: closed  
  
Time 1  
Waiting clients: (0,9,3);(1,28,4);(2,8,3);(3,13,3);  
Queue 0: closed  
Queue 1: closed  
  
Time 2  
Waiting clients: (0,9,3);(1,28,4);(2,8,3);(3,13,3);  
Queue 0: closed  
Queue 1: closed  
  
Time 3  
Waiting clients: (0,9,3);(1,28,4);(2,8,3);(3,13,3);  
Queue 0: closed  
Queue 1: closed  
  
Time 4  
Waiting clients: (0,9,3);(1,28,4);(2,8,3);(3,13,3);  
Queue 0: closed  
Queue 1: closed  
  
Time 5  
Waiting clients: (0,9,3);(1,28,4);(2,8,3);(3,13,3);  
Queue 0: closed  
Queue 1: closed  
  
Time 6  
Waiting clients: (0,9,3);(1,28,4);(2,8,3);(3,13,3);  
Queue 0: closed  
Queue 1: closed  
  
Time 7  
Waiting clients: (0,9,3);(1,28,4);(2,8,3);(3,13,3);  
Queue 0: closed  
Queue 1: closed  
  
Time 8  
Waiting clients: (0,9,3);(1,28,4);(3,13,3);  
Queue 0:(2,8,3);  
Queue 1: closed  
  
Time 9  
Waiting clients: (1,28,4);(3,13,3);  
Queue 0:(2,8,3);  
Queue 1:(0,9,3);  
  
Time 10  
Waiting clients: (1,28,4);(3,13,3);  
Queue 0:(2,8,3);  
Queue 1:(0,9,3);

Time 11  
Waiting clients: (1,28,4);(3,13,3);  
Queue 0: closed  
Queue 1:(0,9,3);  
  
Time 12  
Waiting clients: (1,28,4);(3,13,3);  
Queue 0: closed  
Queue 1: closed  
  
Time 13  
Waiting clients: (1,28,4);  
Queue 0:(3,13,3);  
Queue 1: closed  
  
Time 14  
Waiting clients: (1,28,4);  
Queue 0:(3,13,3);  
Queue 1: closed  
  
Time 15  
Waiting clients: (1,28,4);  
Queue 0:(3,13,3);  
Queue 1: closed  
  
Time 16  
Waiting clients: (1,28,4);  
Queue 0: closed  
Queue 1: closed  
  
Time 17  
Waiting clients: (1,28,4);  
Queue 0: closed  
Queue 1: closed  
  
Time 18  
Waiting clients: (1,28,4);  
Queue 0: closed  
Queue 1: closed  
  
Time 19  
Waiting clients: (1,28,4);  
Queue 0: closed  
Queue 1: closed  
  
Time 20  
Waiting clients: (1,28,4);  
Queue 0: closed  
Queue 1: closed  
  
Time 21  
Waiting clients: (1,28,4);

Queue 0: closed  
Queue 1: closed

Time 22  
Waiting clients: (1,28,4);  
Queue 0: closed  
Queue 1: closed  
  
Time 23  
Waiting clients: (1,28,4);  
Queue 0: closed  
Queue 1: closed  
  
Time 24  
Waiting clients: (1,28,4);  
Queue 0: closed  
Queue 1: closed  
  
Time 25  
Waiting clients: (1,28,4);  
Queue 0: closed  
Queue 1: closed  
  
Time 26  
Waiting clients: (1,28,4);  
Queue 0: closed  
Queue 1: closed  
  
Time 27  
Waiting clients: (1,28,4);  
Queue 0: closed  
Queue 1: closed  
  
Time 28  
Waiting clients:   
Queue 0:(1,28,4);  
Queue 1: closed

Time 29  
Waiting clients:   
Queue 0:(1,28,4);  
Queue 1: closed  
  
Time 30  
Waiting clients:   
Queue 0:(1,28,4);  
Queue 1: closed  
  
Time 31  
Waiting clients:   
Queue 0:(1,28,4);  
Queue 1: closed

Time 32  
Waiting clients:   
Queue 0: closed  
Queue 1: closed  
  
Simulation complete!  
Average waiting time: 3.25

**6. Conclusions**

The queue simulator is a great project in order to learn the nuances of programming with threads. Synchronization is crucial if we want to keep a smooth running of the program. Such, appropriate structures should be used to ensure this feature.

Further development could see the implementation of multiple strategies used to send the clients to the appropriate queue.

**7. Bibliography**

* Class diagram of the project: <https://imgur.com/a/Q5wlaig>
* EasyUML plugin for generating class diagrams inside NetBeans: <http://plugins.netbeans.org/plugin/55435/easyuml>
* Easy to use website to generate UML diagrams, flowcharts, etc.: <https://app.diagrams.net/>