FUNDAMENTAL PROGRAMMING TECHNIQUES

ASSIGNMENT 2
SOLUTION DESCRIPTION DOCUMENT

QUEUES SIMULATOR

Student: Mihai Cristina-Mădălina
Teaching assistant: Chifu Viorica

Faculty of Automation and Computer Science
Computer Science Department (En)
2nd year of study, gr. 30424

Content:

1. Objective

- **The main objective** of this laboratory work is to **design and implement a polynomial calculator** with a **dedicated graphical interface** through which the user can enter polynomials, select the operation to be performed (i.e. addition, subtraction, multiplication, division, differentiation, integration) and display the result.
- The project will implemented as a **Maven Project** using **Eclipse application**.

Secondary objective:

- to provide a place for a "client" to wait before receiving a "service";
- to set a strategy for sending “clients” to be served;
- to simulate a series of clients arriving for service, entering queues, waiting, being served and finally leaving the queue;
- to calculate the average waiting time;

2. Analysis of the laboratory work, scenarios, use-cases

Analysis:

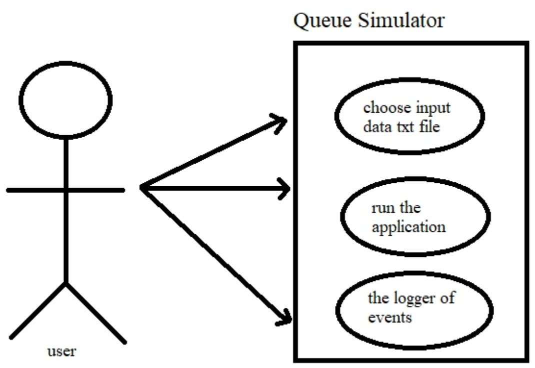
The application should simulate a series of N clients arriving for service, entering Q queues, waiting, being served and finally leaving the queues. All are generated when the simulation is started, and are characterized by three parameters: ID (a number between 1 and N), arrival time (simulation time when they are ready to go to the queue; i.e. time when the client finished shopping) and service time (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 “arrival time” is greater or equal to the simulation time.

The following data should be considered as **input data** read from a **text file** for the application:

- Number of clients (N);
- Number of queues (Q);
- Simulation interval ($t_{simulation}^{MAX}$);
- Minimum and maximum arrival time ($t_{arrival}^{MIN} \leq t_{arrival} \leq t_{arrival}^{MAX}$);
- Minimum and maximum service time ($t_{service}^{MIN} \leq t_{service} \leq t_{service}^{MAX}$);

The **output** of the application is a **text file** containing the **status of the pool of waiting clients and queues** as the simulation time goes from 0 to the maximum simulation time and the **average waiting time of the clients**.

Scenarios:



Title: Queue simulation execution

Actor: In all of the following use-cases, the actor is the user

Use-case:

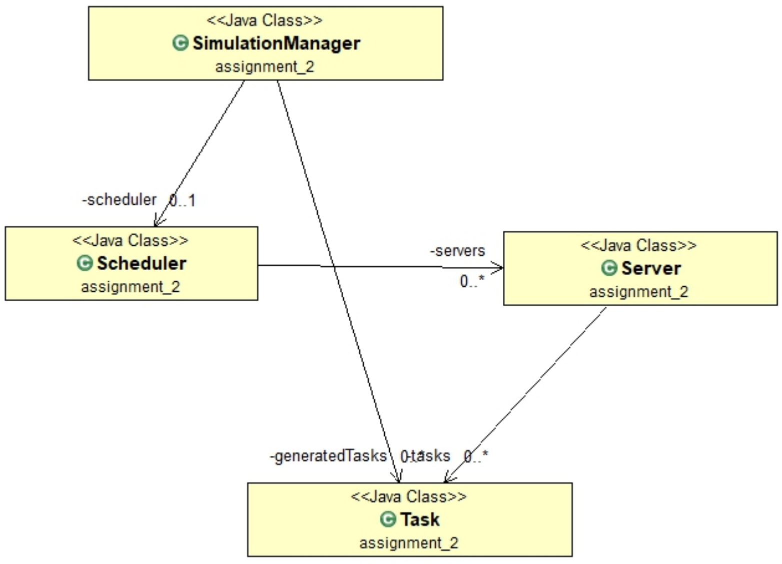
- Preconditions: The user has to choose one of the three text files for input (in-test-1.txt, in-test-2.txt, in-test-3.txt) and one the three text files for output (out-test-1.txt, out-test-2.txt, out-test-3.txt). The user has to introduce the following command line: java -jar PT2020_30424_Cristina_Mihai_Assignment_2.jar in-test-Z.txt out-test-Z.txt, where Z can has to be replaced conveniently by1, 2 or 3.
- Normal Scenario: The user has perfectly introduced has perfectly run the command. The end of execution is done when the average time of the simulation is displayed on the command prompt. After this he/she can look in the chosen output text file to the evolution of queues.
- Alternative Scenario: If the user does not introduce the command line as predefined, an error will occur.

3. Design

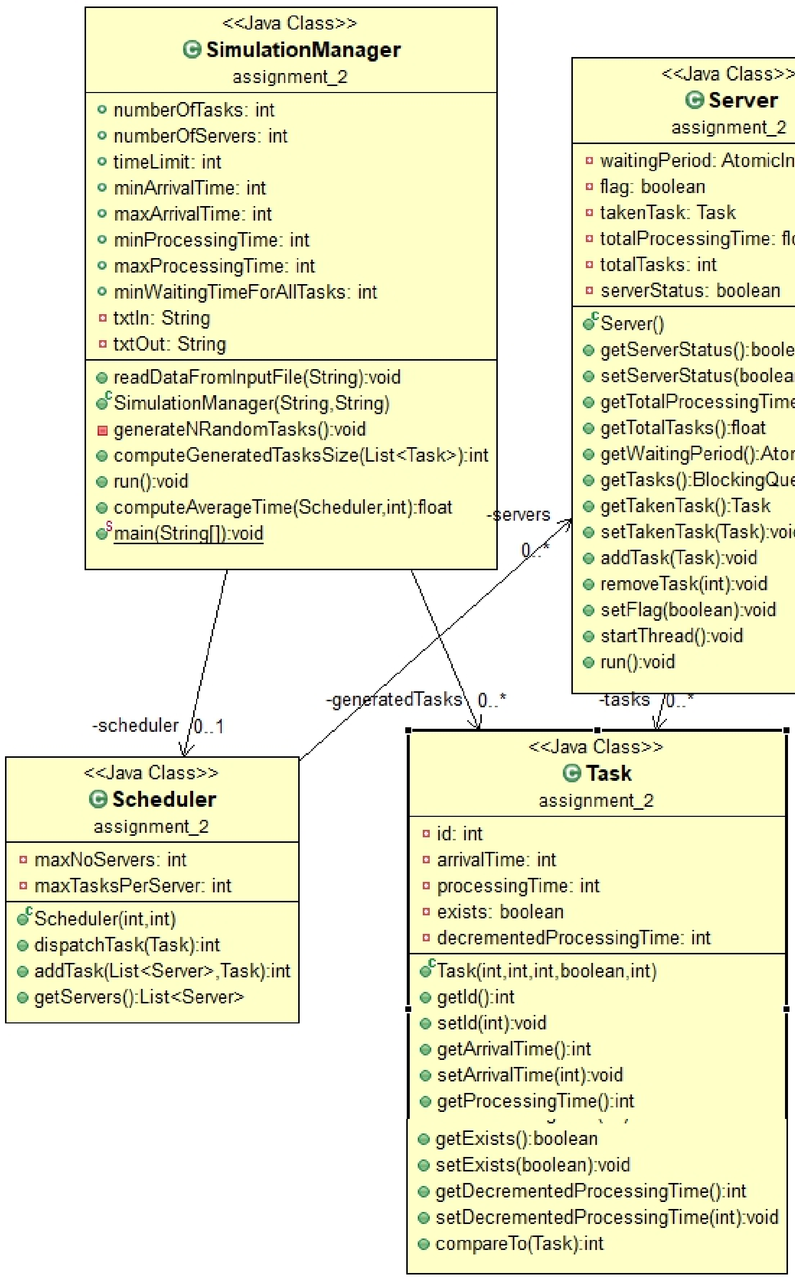
3.1 Design approach

The queue simulator in a single package due to the little number of classes. It contains the following classes:

1. Task: contains the characteristics of each client in the shop, i.e. ID, arrival time in the queue and processing time when they reach in front of the queue.
2. Server: contains the necessary information for dealing with the clients in queue and processes one client at a time in their order of arrival.
3. Scheduler: sends tasks to server according to the established strategy. The strategy was build inside this class and it is based on time the following way. Each client ready to go to the queue will be distributed to the queue with the minimum waiting time of all.
4. Simulation Manager: contains methods for generating random clients' characteristics, for reading data from the input text file, for displaying the expected output in a text file and, also, it contains the simulation loop. This simulation loop keeps track of the current time of the simulation, calls the Scheduler to dispatch one client per second and update the output.



3.1 Class diagram



3.1 Data structures

- BlockingQueue<E> : I used this structure to handle the queues of clients while waiting to be served. I chose to use this structure instead of the classical Queue because it additionally 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.
- ArrayList : I chose this structure to handle the servers and the generated clients.

4. Implementation

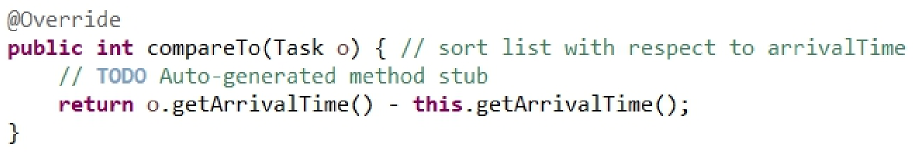
Class Task:

Instance variable:
- *id*: int
- *arrivalTime*: int
- *processingTime*: int
- *exists*: bool
- *decrementedProcessingTime*: int

The first three variables are the characteristics visibly accessible to the user, the boolean one is used as true when the client is generated and waiting to be dispatched to the corresponding server. When the client gets in the waiting line, its variable "exists" is set to false, meaning that it is no longer shopping. The last variable is an auxiliary variable to help decrementing the processing time while in front of server and not modifying the initial processing value because we will need it to compute the average waiting time.

It contains constructor to initialize each task, and getters and setters to access and change the value of the private variable above if needed.

The class also implements the **Comparable** interface and contains the method below which will help sort the clients in order of the arrival time when generated.



```
@Override
public int compareTo(Task o) { // sort list with respect to arrivalTime
    // TODO Auto-generated method stub
    return o.getArrivalTime() - this.getArrivalTime();
}
```

Class Server:

Instance variable:
- *tasks*: BlockingQueue<Task>
- *waitingPeriod*: AtomicInteger
- *flag*: boolean
- *takenTask*: Task
- *totalProcessingTime*: float
- *totalTasks*: int
- *serverStatus*: boolean

The "tasks" variable will keep track of the current clients waiting at the current server. The "waitingPeriod" one is just an auxiliary variable which will compute the time to finish the current number of clients and will help the Scheduler decide in which server to put the next client. The flag is used just to prepare the thread when the server will be initialized. The "takenTask" will take the next client in queue and process its characteristics in that variable. The "totalProcessingTime" and "totalTasks" will help compute the final average time and will be incremented when a client is introduced to server and processed. The "serverStatus" is set to false when the server is closed and to true otherwise.

This class contains constructor, getters & setters with the same purpose as above. Two additional important methods are described below:

```java
public void addTask(Task newTask) {
    // 1. add task to queue
    // 2. increment the total number of tasks
    // 3. increment the waitingPeriod
    this.tasks.add(newTask);
    this.waitingPeriod.addAndGet(newTask.getProcessingTime());
    totalTasks++;
}

public void removeTask(int initialProcessingTime) {
    // 1. remove task from queue
    // 2. decrement the waitingPeriod
    // 3. increment the totalProcessingTime
    this.tasks.remove();
    this.waitingPeriod.addAndGet((-1) * initialProcessingTime);
    totalProcessingTime += initialProcessingTime;
}
```

As it can be noticed, the *totalTasks* is incremented when clients are added to server and the *totalProcessingTime* is incremented when processing of the current client is done.

Since the class implements **Runnable**, it overrides the *run()* method described below:

```java
@Override
public void run() {
    while (flag == true ) {

        if (tasks.size() > 0) { // --> server has tasks
            setServerStatus(true); // --> open server

            this.takenTask = tasks.peek(); // 1. peek next task from queue
            int currentTaskProcessingTime = takenTask.getProcessingTime();
            while (currentTaskProcessingTime > 0) { // 2. stop the thread for one second and decrement processingTime of that Task
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    System.out.println("\nserver can't sleep 1 sec");
                    e.printStackTrace();
                }
                --currentTaskProcessingTime;
                takenTask.setDecrementedProcessingTime(currentTaskProcessingTime);
            }

            removeTask(takenTask.getProcessingTime()); // 3. decrement the waitingPeriod + remove task from queue

        } else { // --> server has NO tasks
            setServerStatus(false); // --> closed server
        }
    }
}
```

While there are clients in the queue, the server is open. Otherwise, it is closed. In case it is open, the method peeks the next task from queue and decrements the current client processing time at each second. When decrementing is finished, the task is deleted from queue and server processes the next client if they exist or closes.

Class Scheduler:

Instance variables:
- *servers*: List<Server>
- *maxNoServers*: int
- *maxTasksPerServer*: int

This class uses the list of servers to easily access them and their status. The "maxNumberOfServers" contains the value of the maximum numbers of servers on which the simulation runs and it its value is taken from the file input. The "maxTasksPerServer" is desired to be as large as possible since clients can vary from 4 to 1000.

It also contains a constructor and a method *getServers()* shown below to access clients from each server when output needs to be done:

```
public List<Server> getServers(){
    return servers;
}
```

The main method is *dispatchTask(Task task)*, which just calls the method *addTask(List<Server> servers, Task task)* which indeed iterates through all servers and find the one with minimum waiting time so that the client who has to be dispatched will be sent to the queue where he/she will have to wait the shortest time possible. The methods return the minimum waiting time found which will help compute the average time. The implementation is shown below:

```
public int dispatchTask(Task task) {
    // time-based strategy
    return addTask(this.servers, task);
}
```

```
public int addTask(List<Server> servers, Task task) {

    int minWaitingTime = servers.get(0).getWaitingPeriod().get(); // waitingPeriod of the first Server
    for (Server i : servers) { // iterate through the list of Servers
        if (i.getWaitingPeriod().get() <= minWaitingTime) { // find minimum waitingPeriod
            minWaitingTime = i.getWaitingPeriod().get();
        }
    }
    for (Server i : servers) { // iterate again
        if (i.getWaitingPeriod().get() == minWaitingTime) { // when Server with minWaitingPeriod found
            i.addTask(task); // add task to that particular Server
            i.setServerStatus(false);
            break;
        }
    }
    return minWaitingTime;
}
```

Class SimulationManager:

Public instance variable:
- *numberOfTasks*: int
- *numberOfServers*: int
- *timeLimit*: int
- *minArrivalTime*: int
- *maxArrivalTime*: int
- *minProcessingTime*: int
- *maxProcessingTime*: int
- *minWaitingTimeForAllTasks:* int

Apart from the last one, all the public variables will be read from the input data text file. The "numberOfTasks" and "numberOfServers" defines the number of clients and servers available in the shop on the simulation period of time. The "timeLimit" variables means the time (measured in seconds) when the simulation will stop, regardless of the number of clients remaining in the shop. The minimum arrival time and maximum arrival time defines the limit when clients are ready with shopping and want to go to one of the queues. The minimum and maximum processing time means the limit of a client to wait in front of the queue while their goodies are processed. The minimum waiting time for all tasks is the amount of time each client has to wait in the queue until it is their turn to be processed by the server. This computation will be discussed later in the corresponding method.

Private instance variables:
- *scheduler:* Scheduler
- *generatedTasks*: List<Task>
- *txtIn*: String
- *txtOut:* String

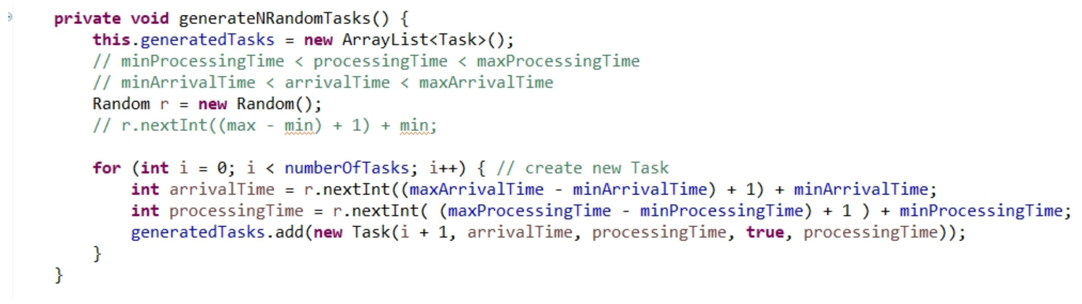
The "scheduler" variable is the entity responsible for queue management and client distribution. The "generatedTasks" are the tasks shopping in the store and the last two strings contain the name of the input and output text files which will be given as arguments to the class as hand.

The following section contains the method implemented along with the necessary explanations:

- *readDataFromInputFile(String txtFile)*: void
This method reads data from the input file and assigns the public variables described above the corresponding values. For this method, there has been used and imported the **java.util.File** and **java.util.Scanner** packages.

- *SimulationManager(String txtIn, String txtOut)*: <<constructor>>
It does the following:
1. reads data calling the method above and store it into the public variables;
2. initializes the Scheduler
3. creates and starts the servers
4. calls the method *generateNRandomTasks()* (described below) to generate clients and store them in the "generatedTasks" list.

- *generateNRandomTasks():* void



```
private void generateNRandomTasks() {
    this.generatedTasks = new ArrayList<Task>();
    // minProcessingTime < processingTime < maxProcessingTime
    // minArrivalTime < arrivalTime < maxArrivalTime
    Random r = new Random();
    // r.nextInt((max - min) + 1) + min;

    for (int i = 0; i < numberOfTasks; i++) { // create new Task
        int arrivalTime = r.nextInt((maxArrivalTime - minArrivalTime) + 1) + minArrivalTime;
        int processingTime = r.nextInt( (maxProcessingTime - minProcessingTime) + 1 ) + minProcessingTime;
        generatedTasks.add(new Task(i + 1, arrivalTime, processingTime, true, processingTime));
    }
}
```

Each client is given an ID (from 1 to numberOfTasks), and a random arrival and processing time given by the formula: random time = random.nextInt(maximum limit - minimum limit + 1) - minimum limit.

- *run():* void
Once the simulation starts, a thread starts and stay alive as long as a currentTime variable initialized with 0 reaches the value of the timeLimit (i.e. 60 in first and second case and 200 in the third case). First thing to do is to check if there are still clients in the shop, so they can be dispatched and the variable "minWaitingTimeForAllTasks" can be increased. Next step is to display the waiting clients in the shop and the clients at the queue. Next step is to increment the currentTime variable and call the *sleep(1000)* method to make the simulation thread stop for one second. Once the currentTime reaches the timeLimit, it is time to calculate and display the average waiting time in the shop. This computation is don in the following method.

- *computeAverageTime(Scheduler scheduler, int currentTime)*: float
This method compute a fraction with the following values: the nominator is the sum of minimum waiting time of all clients + the processing time of each client who has been served, and the denominator is the number of all the clients who have been served while the simulation was running. The implementation is shown in the screenshot below:

```
public float computeAverageTime(Scheduler scheduler, int currentTime) {
    float nominator = minWaitingTimeForAllTasks;
    float demonimator = 0;
    for (Server i : scheduler.getServers()) {
        nominator += i.getTotalProcessingTime();
        demonimator += i.getTotalTasks();
    }
    return nominator / demonimator;
}
```

- *<u>main(String[] args)</u>*<u>:</u> void

```
public static void main(String[] args) {
    SimulationManager gen = new SimulationManager(args[0], args[1]);
    Thread simulationThread = new Thread(gen);
    simulationThread.start();
}
```

A simulation generator is initialized and a thread with it stats.

5. Results

For checking some results, we will run 2 test2 and extract some result parts:
Test 1:
Input file : in-test-1.txt
- 4 clients
- 2 queues
- 60 seconds simulation time
- 2 sec < arrival time < 30 sec
- 2 sec < processing time < 4 sec

Output file: out-test-1.txt

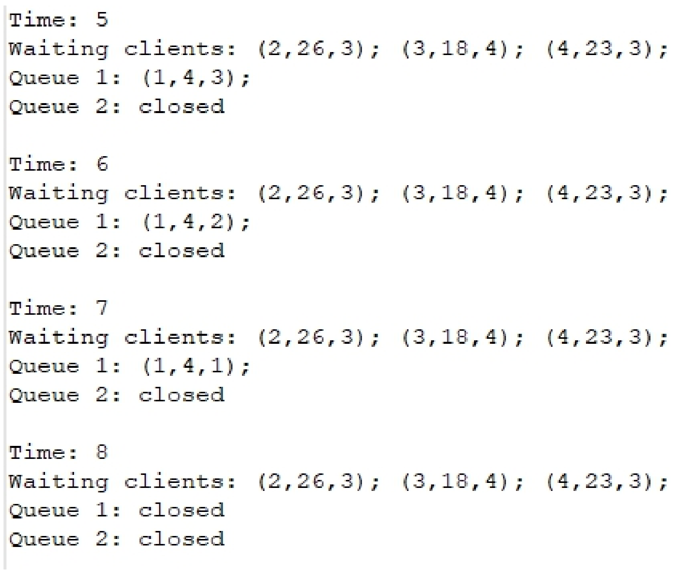
```
Time: 0
Waiting clients: (1,4,4); (2,26,3); (3,18,4); (4,23,3);
Queue 1: closed
Queue 2: closed

Time: 1
Waiting clients: (1,4,4); (2,26,3); (3,18,4); (4,23,3);
Queue 1: closed
Queue 2: closed

Time: 2
Waiting clients: (1,4,4); (2,26,3); (3,18,4); (4,23,3);
Queue 1: closed
Queue 2: closed

Time: 3
Waiting clients: (1,4,4); (2,26,3); (3,18,4); (4,23,3);
Queue 1: closed
Queue 2: closed

Time: 4
Waiting clients: (2,26,3); (3,18,4); (4,23,3);
Queue 1: (1,4,4);
Queue 2: closed
```



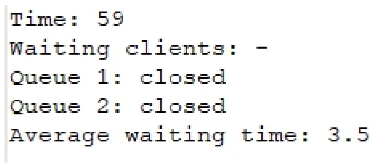
```
Time: 5
Waiting clients: (2,26,3); (3,18,4); (4,23,3);
Queue 1: (1,4,3);
Queue 2: closed

Time: 6
Waiting clients: (2,26,3); (3,18,4); (4,23,3);
Queue 1: (1,4,2);
Queue 2: closed

Time: 7
Waiting clients: (2,26,3); (3,18,4); (4,23,3);
Queue 1: (1,4,1);
Queue 2: closed

Time: 8
Waiting clients: (2,26,3); (3,18,4); (4,23,3);
Queue 1: closed
Queue 2: closed
```

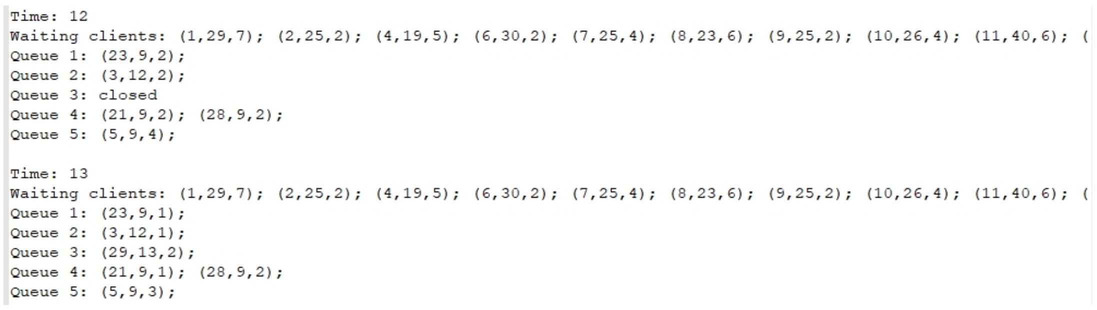
...



```
Time: 59
Waiting clients: -
Queue 1: closed
Queue 2: closed
Average waiting time: 3.5
```

Test 2:
Input file : in-test-2.txt
- 50 clients
- 5 queues
- 60 seconds simulation time
- 2 sec < arrival time < 40 sec
- 1 sec < processing time < 7 sec

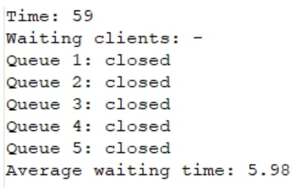
Output file: out-test-2.txt
. . .



```
Time: 12
Waiting clients: (1,29,7); (2,25,2); (4,19,5); (6,30,2); (7,25,4); (8,23,6); (9,25,2); (10,26,4); (11,40,6); (
Queue 1: (23,9,2);
Queue 2: (3,12,2);
Queue 3: closed
Queue 4: (21,9,2); (28,9,2);
Queue 5: (5,9,4);

Time: 13
Waiting clients: (1,29,7); (2,25,2); (4,19,5); (6,30,2); (7,25,4); (8,23,6); (9,25,2); (10,26,4); (11,40,6); (
Queue 1: (23,9,1);
Queue 2: (3,12,1);
Queue 3: (29,13,2);
Queue 4: (21,9,1); (28,9,2);
Queue 5: (5,9,3);
```

. . .



```
Time: 59
Waiting clients: -
Queue 1: closed
Queue 2: closed
Queue 3: closed
Queue 4: closed
Queue 5: closed
Average waiting time: 5.98
```

6. Conclusions

Lessons learned:
- never mess up with threads
- the more I try to fix things, the more occur
- threads are sensitive and working with them requires a high level of awareness and patience.

Possible future improvements:
- maybe clients could be alerted when shop is close to get closed so they can decrease their processing time and not get stuck in the shop and the end of simulation
- maybe the simulation time could be increased or decreased according to the clients left in the shop.

7. Bibliography

Use-case diagrams:
https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/

Blocking queue:
https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html

Atomic Integer:
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html

Approach:
http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_2/Java_Concurrency.pdf