Assignment 2

Buzan Vlad-Sebastian

Gr: 30421 CTI

a. Scope of the project.

The main purpose of this assignment was to build a Java application having the following functionality: parsing from a text file (which is given as a command line argument) values describing the number of clients in a store, the number of queues available to that store, bounds for each clients arrival and waiting time and a maximum simulation time. The application is supposed to create threads for each queue

The secondary objectives (which have been derived from the main objective):

- Providing the user the option to give 2 text files as command line arguments
- Parsing the text files in order to obtain the desired information about the persons entering the store
- Creating a representative class for the "person" concept
- Generating instances of the Person class with random values within the bounds given in the input text file
- Creating the class equivalent of a queue
- Prove thread-functionality for the queue class
- Have a scheduler class for dynamically opening/closing threads, taking into account the number of clients needing to be processed
- Generating an output file providing information about each queue's status at each moment during the execution

b. Analyzing the problem, modelling, scenarios, use-cases.

First of all, as stated in the previous section, we can deduce some of the classes that need to be designed further for the project. In my implementation, I decided to implement a class having the sole purpose of handling the input (***PersonsReader***). Another obvious decision was to create a class responsible for holding information about the clients. The key data needed for processing the clients are their time of arrival, the time spent at front of the queue(i.e. the time it takes to process the given client), an ID and the total time spent in the queue. Next, another crucial class would be the actual queue which also needs to implement the

thread functionality. In my approach, the queue class offers the following functionality: adding a person to the queue and elapsing one unit of time(i.e. decrementing the wait time of the person at the front of the queue and increasing the wait time of all the persons in the queue). The queue also automatically removes the persons whom have been processed. Another important class in my implementation is the Scheduler class. Its main purpose is to handle the threads running the queues.

As far as use-cases go, we have the following scenarios:

- The user inputs two text files, the first one containing the data about the persons in the store, the second one being used to output the steps of execution
- The program executes successfully, writing the steps of execution in the output file
- The program doesn't execute properly, writing an error message to standard output

In our case, the primary actor is the user, and the execution of the program is highly dependent on the input file given by the user.

Main success scenario:

(a) The user inputs one existing input file and the name of the output file which may or may not exist
(b) Application successfully opens the files given as command line arguments
(c) The data inside the first file is successfully parsed
(d) Instances of the person class are generated with random values within the bounds read from the input file
(e) Scheduler class is instantiated, and the thread specific to this class is started
(f) Queues are instantiated
(g) Inside the scheduler thread, threads are generated for each queue
(h) The program is executed one step at a time
(i) Output file is written

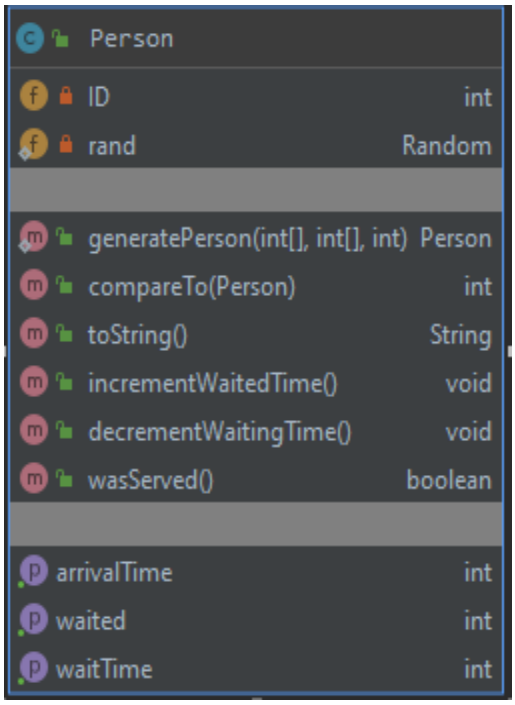
Alternate scenario:

i. The user inputs the command line arguments
    1. The input file doesn't exist -> error is thrown and program execution is halted
ii. The application successfully opens the files given as command line arguments.

iii. The program tries to parse the data in the first command line argument
    1. Data isn't specified as expected -> error is thrown and program execution is halted
iv. Instances of the person class are generated with random values within the bounds read from the input file
v. Queues are instantiated
vi. Inside the scheduler thread, threads are generated for each queue
vii. The program is executed one step at a time
viii. Output file is written

c. Design choices, data structures, class design, packages and algorithms, implementation

For this particular application I went for an object-oriented design, as this approach made it possible to have abstract data structures representing the queues, which were easy to implement, and allowed me to focus more on thread synchronization, scheduling and managing.



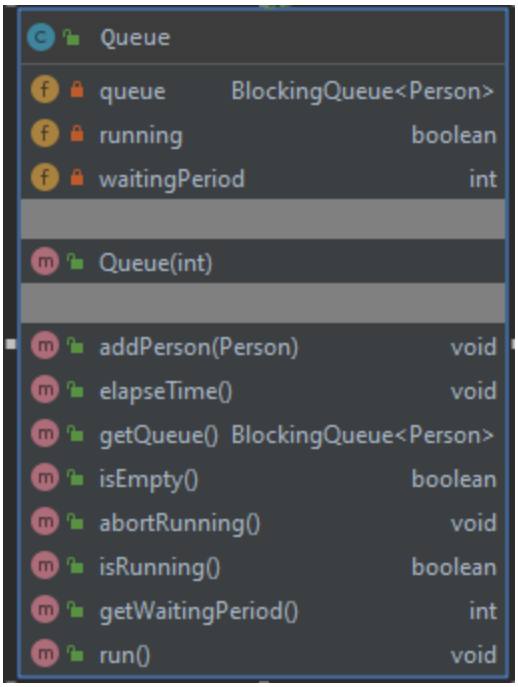
The data-structure used to hold the information about each individual client is a simple Person class having the following fields: arrivalTime, waitTime, ID, and waited. The arrivalTime and waitTime fields are generated randomly, but within the bounds read from the input file.

The IDs for each Person are given in the order the persons are generated (ID=1 for first person generated, 2 for second etc.). The methods found in this class are:

- generatePerson() – used to randomly generate the arrivalTime and waitTime of a person, and to return that person. Takes two arrays of integers containing the bounds for each value (arrival and wait times)
- compareTo() – used later for sorting the persons with respect to their arrival times. When equality is found when comparing the arrival times, the method compares the wait times.
- incrementWaitedTime() – used to increment the "waited" value, when a person belongs to a queue and the thread specific to that queue is running, this method is called upon all persons in that queue

- decrementWaitingTime() -similar to the previously discussed function, used to decrement the waiting time of the person at the front of the queue
- wasServed() – returns boolean type : true when the waitingTime of the person is 0, false otherwise

For representing the actual queue which is used to manage the clients, I designed another class, which also implements Runnable. In order to have the Queue class as abstract as possible, when it comes to clients, its interface offers only the functionality of adding an instance of the Person class, elapsing time for each member of the queue, getting the information about the status of the queue (whether there are still clients in the queue or the total waiting time of the queue).



The fields of this class are the queue field(BlockingQueue<Person> is used in order to ensure thread safety), the boolean running (which is used for controlling the thread corresponding to the Queue instance) and waitingPeriod(which is the total waiting period one would have to wait if one were to enter the queue).

Since the BlockingQueue class requires a size when instatiated, the Queue class also requires a size. This size (as will later be discussed) is given as the number of persons read from the input file. The running flag is set as false.

The addPerson() method, as its name implies, adds the person given as argument at the end of the BlockingQueue structure. This methods also increments the waitingPeriod field with the waitingTime of the added person.

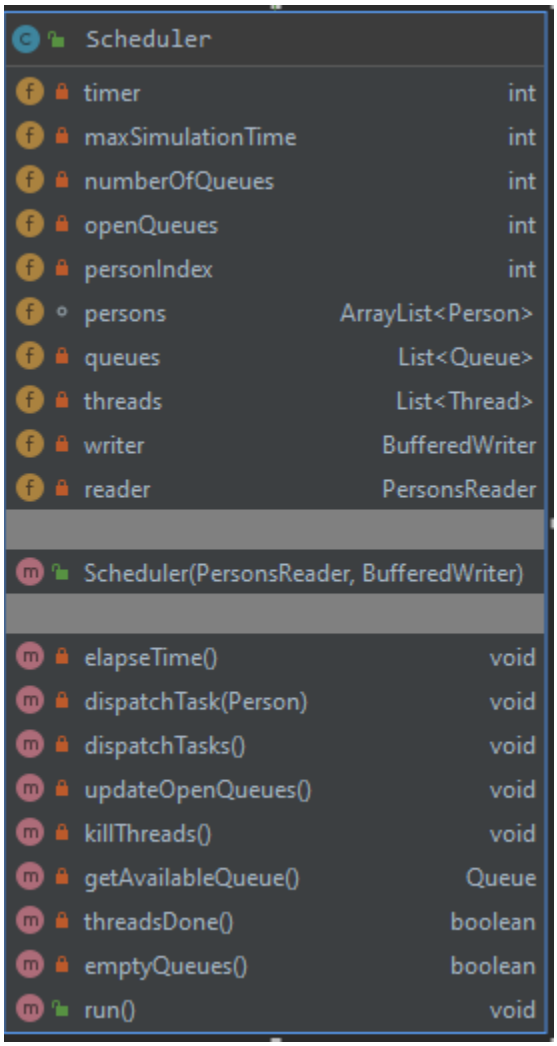
The elapseTime() method does the following: firstly, it increments the "waited" field of each person in the queue, secondly, decreases the waitTime of the person at the front of the queue. If the person at the front of the queue was completely processed, it is removed from the queue.

```
public void elapseTime() {
    queue.forEach((p)-> p.incrementWaitedTime());
    queue.peek().decrementWaitingTime();
    if(queue.peek().wasServed()) queue.remove();
    waitingPeriod --;
}
```

The thread implementation for the Queue class serve the following purpose: advance the queue one unit of time, then, decide whether or not the thread should be left to complete execution (i.e. to terminate). If the queue is empty, the thread is left to terminate,

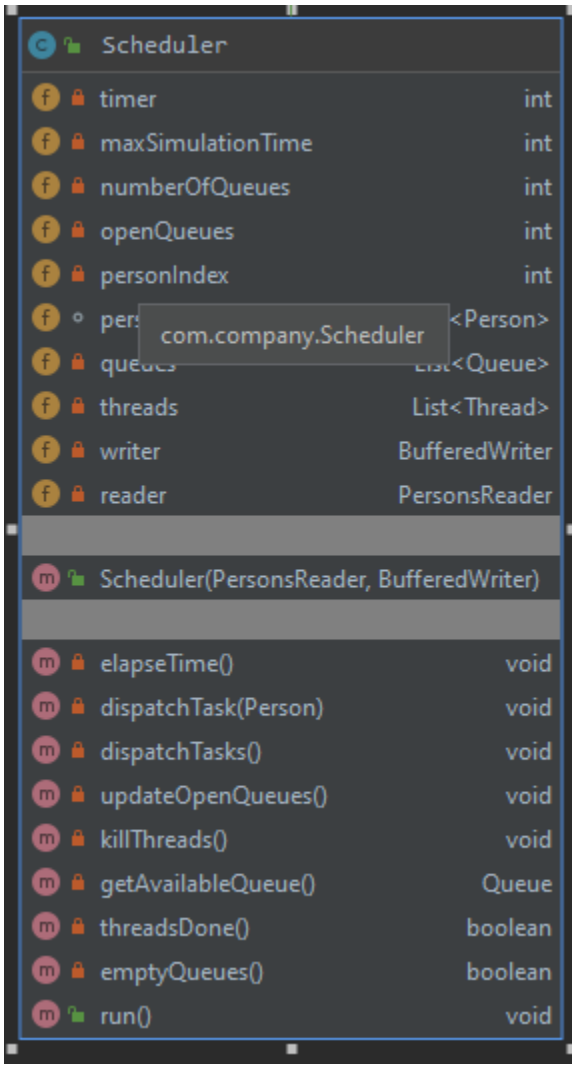
```java
@Override
public void run() {
    while(true) {
        elapseTime();
        if(isEmpty()) {
            running = false; //kill thread
            return;
        } else {
            try {
                synchronized (this) {
                    wait();
                    if(!running) {
                        return;
                    }
                }
            } catch (InterruptedException ex) {
                System.out.println("Thread interrupted");
                System.out.println(ex.getMessage());
            }
        }
    }
}
```

otherwise, the thread enters the WAITING state. If entered the wait() state, the thread is left in this state until the Scheduler thread (discussed later in this section) will call notify on the Queue instance corresponding to the thread. The return statement after wait() is called is used to force the termination of the thread execution when the queue is not yet empty. This occurs when the maximum simulation time is reached and threads need to be terminated to finish the execution of the application.



Another class of utmost importance in the design process of the application is the Scheduler class. This class serves the purpose of managing the running threads and assigning task to each thread. The Scheduler class also implements the Runnable interface as its functionality is mostly attained by running in its own thread. Some of the fields of the Scheduler class are: the timer (which is used for keeping track of the current step of execution), the maxSimulationTime (read from the input file), numberOfQueues (also read from the input file), openQueues (the current number of queues that are open at each execution step), the personIndex (the index in the ArrayList<Person> that currently needs to be processed), persons (ArrayList containing all the randomly generated instances of the Person class), queues (List of Queue class instances), threads (List of threads created using the Queue class), writer and reader (used for input/output).

The constructor takes as arguments a PersonReader (discussed later in this section) and a BufferedWriter (used for writing output in the file).



Using the information obtained from the PersonsReader argument, it instantiates all the fields.

After the Scheduler class is instantiated, a thread is created using that instance, and it performs all the desired operations.

The run() method in the Scheduler class instantiated the queues and threads fields, then enters a while loop (while (timer <= maxSimulationTime)). In the while loop, each iteration represents one step in the execution of the program. The Scheduler thread is responsible with the following: assigning tasks to each queue (accomplished using the dispatchTasks() method), outputting the current status of the queues and the persons yet to be processed, starting the threads corresponding to each queue when such action is required and then waiting for the threads to finish execution in order to proceed to the next step of execution. As one can observe, the Scheduler thread never assigns new tasks before the Queue threads are not done elapsing one unit of time. This is done in order to be able to output the state of the queues at each step, otherwise if the scheduler thread would be left to run independently or the queue threads would not be suspended or terminated, synchronization issues would occur, such as the scheduler thread outputting before the queue thread managed to process the clients, or the scheduler threads processing the clients more steps before the scheduler thread gets to output the state of the queues.

The dispatchTasks() methods calls the dispatchTask() method on all the Person instances that have the arrivalTime field equal with the timer field. The dispatchTask() method firstlly checks whether there are any unopened queues and if so, adds the person in a empty queue. If there are no empty queues, it calls the getFastestQueue() method and adds the person in the returned queue.

The elapseTime() method works by checking each queue to see whether they are running, then checks whether the corresponding thread is in the waititing state or in the terminated state. If the thread is in the waiting state, it gets notified and continues execution. If

```
private void elapseTime() {
    Queue queue;
    Thread thread;
    for(int i = 0; i < numberOfQueues; i ++) {
        queue = queues.get(i);
        thread = threads.get(i);
        if(queue.isRunning()) {
            if(thread.getState() == Thread.State.WAITING) {
                synchronized (queue) {
                    queue.notify();
                }
            } else {
                threads.remove(i);
                thread = new Thread(queue);
                threads.add(i, thread);
                thread.start();
            }
        }
    }
}
```

the thread is in the terminated state, a new thread is created and the method start() is called.

The threadsDone() method is used to check whether threads finished execution in order to notify the scheduler thread that it can move on to the next step.

The killThreads() method is called when the program finished execution and as its name suggests, forces all threads to finish execution.

The Server(improperly named) class is simply used to instantiate the reader and writer for the i/o operations. It's constructor takes as arguments the paths to the input/output file. The start() method creates the Scheduler thread.

4. Conclusions, results.

After testing the application using the input files provided in the assigment specification, I can conclude that the application runs as expected, producing the right results, which can be observed in the ouput file. If there are no more clients to process, yet the maximum simulation times was not reached, the application stops.