FUNDAMENTAL PROGRAMMING TECHNIQUES

Assignment 2

Queues Simulator

Student: Halmai Erik

Group: 30422

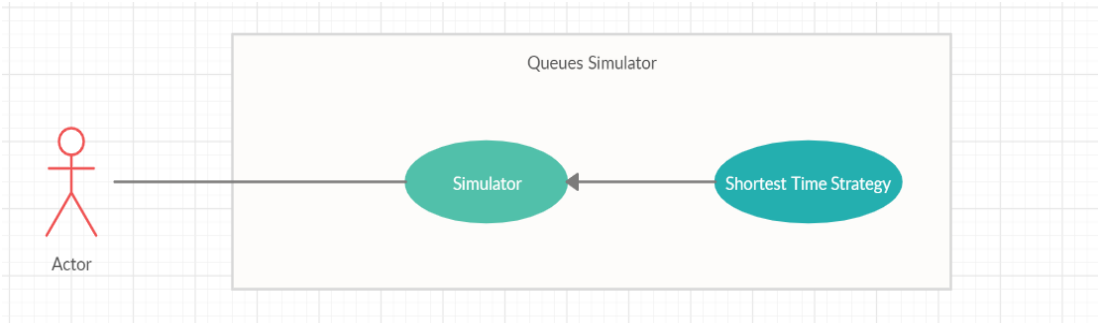
1. Objective

The main objective of the assignment is the creation of a simulation application, aiming to analyse queuing based systems for determining and minimizing clients' waiting time. In this implementation, the used strategy compares the waiting times of all queues and chooses the minimal one. The project intends to recreate a real-life shop, in which clients come in, do their shopping for a given amount of time and then go to the queue where they wait to be served. The management of such systems, in the real life, interested in the reduction of time that each client needs to wait before being served.

The main objective of the assignment can be divided into several secondary objectives, each achieved separately. These secondary objectives represent the steps that are taken in order to achieve the main objective of the assignment. These steps are the following:

1) **Creation of an object that can simulate the queue systems (Chapter 4)**: Queues provide a place for a "client" to wait before receiving a "service". In Java, the simulation of time-based systems, can be done using threads.
2) **Concurrency of queues (Chapter 4)**: The application should be able to handle appropriately multiple such queue simulating objects. This feature of multiple threads running in the same time is called Multithreading.
3) **Generation of Client objects (Chapter 4)**: Clients are characterized by three parameters, the id, the arrival time to the queue and the service time. These parameters need to be generated using the minimum and maximum values given in the input file specified in the arguments.
4) **Computation of the average waiting time (Chapter 4)**: At the end of the simulation, the average waiting time needs to be computed i.e. the sum of the total time waited in queue by each processed client divided by the number of processed clients.
5) **Output generation (Chapter 5):** The log of the simulation at every time frame and the average waiting time need to be written to an output file given as an argument.

2. Problem analysis, modeling, scenarios, use cases



1. **Introducing the input:** The input data is read from a file given as the first parameter when running the jar file from the command prompt. This data consists of 7 numbers given in 5 consecutive lines of the input file having the following format:
    1. Number of clients
    2. Number of queues
    3. Simulation interval
    4. Minimum arrival time, Maximum arrival time
    5. Minimum service time, Maximum service time

    Note that Minimum arrival time and Maximum arrival time need to be separated by a comma. The same applies for Minimum service time and Maximum service time.

The input data can generate two types of unwanted scenarios. One in which there are logic errors in the data, such as the minimum values being bigger than the maximum ones, or the number of clients and queues being zero. In such case, a WrongInputException is thrown and the simulation terminates. The second scenario is when the input file doesn't follow the given format (ex. The file contains non-digit characters that are read as input data) in which case, the application will crash. Example of input file:

In-Test.txt

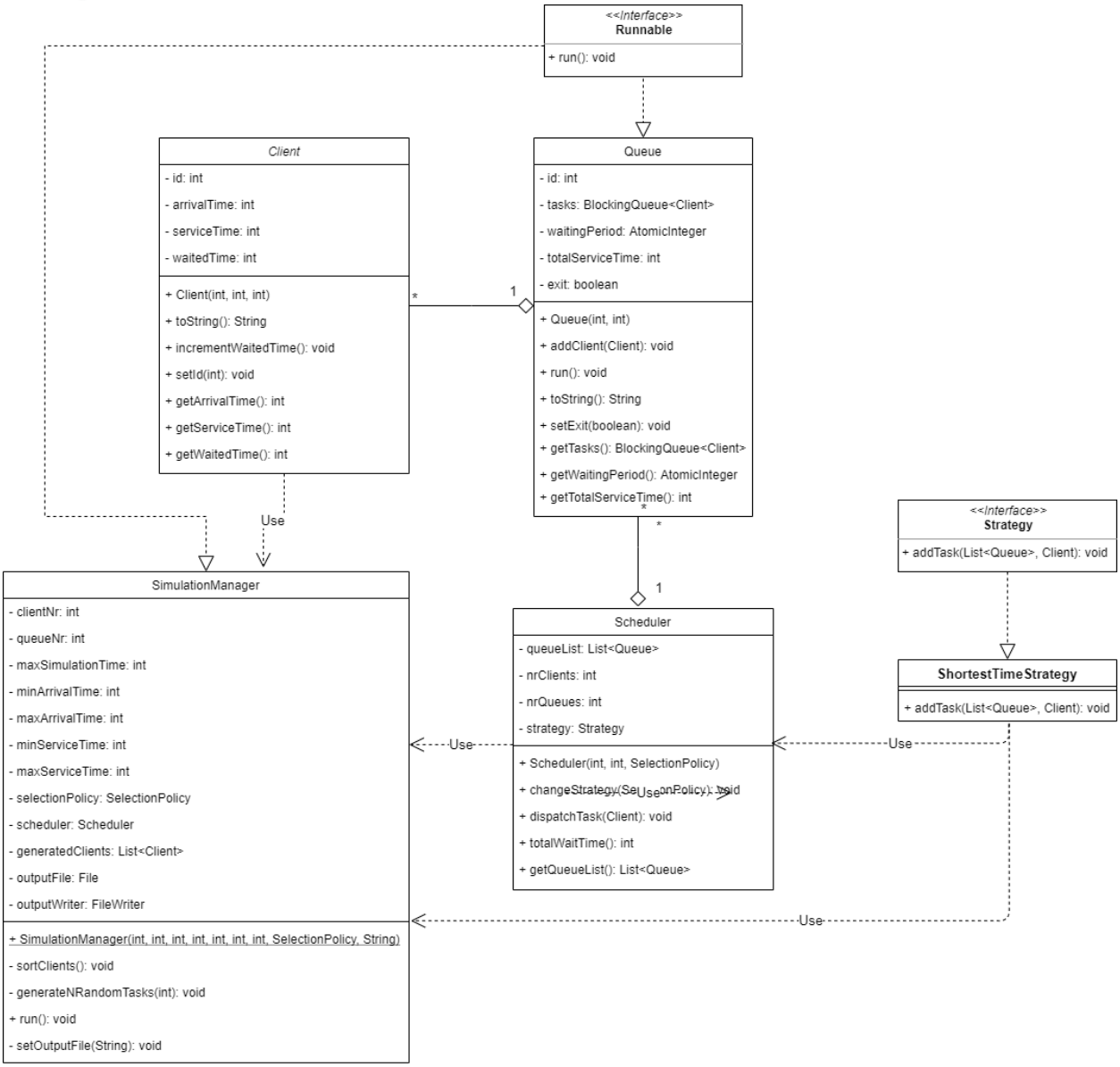
```
4
2
60
2,30
2,4
```

2. **Getting the simulation result:** The created output of the simulation will be in the file specified as the second parameter when running the jar file from the command prompt or the IDE's terminal. The output file will contain the simulation's log, specifying the state of the queuing system at each time frame (second) and the calculated average waiting time. Example of output file:

Out-Test.txt

```
Time 0
Waiting clients: (1,2,2); (2,3,3); (3,4,3); (4,10,2)
Queue 1: closed
Queue 2: closed

Time 1
Waiting clients: (1,2,2); (2,3,3); (3,4,3); (4,10,2)
Queue 1: closed
Queue 2: closed

Time 2
Waiting clients: (2,3,3); (3,4,3); (4,10,2)
Queue 1: (1,2,2);
Queue 2: closed

Time 3
Waiting clients: (3,4,3); (4,10,2)
Queue 1: (1,2,1);
Queue 2: (2,3,3);

Time 4
Waiting clients: (4,10,2)
Queue 1: (3,4,3);
Queue 2: (2,3,2);

…
Average waiting time: 2.5
```

3. Design



The application uses multithreading, i.e. multiple Thread objects are running in the same time. Two classes define a thread, Queue and SimulationManager, by implementing the Runnable interface and its run() method. The Scheduler class is responsible with setting up the Queues. It sends the Client objects to the Queues according to the established strategy. There is only one possible strategy in this implementation, the one concerned with the waitingTime of the queues. The Client is distributed to the Queue with minimal waiting time.

Packages: The application is divided in 4 packages:

- Data package: Contains the Client and Queue classes.

- Schedulers package: Contains the Scheduler and SimulationManager classes
- Util package: Contains an exception specific to the queue simulator application: WrongInputException, that extends RunTimeException.

4. Implementation

1) **Client**: Represents the clients in the simulation. It has four private fields: id (integer), arrivalTime (integer), serviceTime (integer) and waitedTime (integer).
   a) Client(int, int, int): Constructor of Client class. Sets id, arrivalTime and serviceTime of the Client object to the values given as parameter and sets the waitedTime to 0.
   b) toString(): Constructs the printable string representation of a client. This will be of the form: "(id,arrivalTime,serviceTime)".
   c) incrementWaitedTime(): Increments the waitedTime of a client. Waited time is equal to the total time spent by the client in a queue.
2) **Queue**: Simulates a real life queue with a BlockingQueue of clients, by implementing the Runnable interface. It uses an Atomic Integer (thread safety) to calculate the period a client would have to wait if he/she were to enter this queue. The class also has a nrProcessedClients (int) field, which calculates the number of Clients that have successfully gone through the queue in the given simulation time.
   a) Queue(int, int): Constructor of Queue class. Initializes the queue of tasks, waitingPeriod and totalServiceTime and sets the id of the Queue.
   b) addClient(Client): Method to add a new client to the queue. It adds the new client to the end of the BlockingQueue and sets the waitingPeriod to the service time of the new client.
   c) run(): Overridden run method from Runnable interface.
   d) toString(): Constructs the printable string representation of a queue, enumerating all clients. This will be of the form: "Queue id: (client1Id, client1ArrivalTime, client1ServiceTime); ...".
   e) setExit(boolean): Sets exit boolean variable of a queue to stop the thread.
3) **Scheduler:** Controls the list of Queue objects in the simulation. It creates the queues and sets them up by adding clients to the queues with a chosen strategy.
   a) Scheduler(int, int, SelectionPolicy): Constructor of Scheduler class. Initializes it's field, creates queue objects in queueList and starts the thread with each such object.
   b) changeStrategy(SelectionPolicy): Method to change the strategy of the simulation.
   c) dispatchTask(Client): Dispatches a task (i.e. a Client) using the set strategy's addTask method.
   d) totalWaitTime(): Returns total waited time of each finished client from every queue in the queue list.
   e) totalProcessedClients(): Return the number of successfully processed clients, i.e. the number of clients that managed to do their “shopping” and get through a queue within the given maximum simulation time
4) **SimulationManager:** Responsible with the management of the whole simulation. It gets the input data and using the scheduler, it constructs the output log of the simulation and writes it to the output file. This class is a Thread object and it is running during the whole simulation time, being paused for only 1 second at each iteration of the simulation time.

a) SimulationManager(int, int, int, int, int, int, int, SelectionPolicy, String): Constructor method, that sets the corresponding fields of the SimulationManager to the ones given as parameter. It also creates a new scheduler object with the selected strategy and the output file with the name given as parameter and a FileWriter associated with the file.
b) sortClients(): Sorts the generated clients with respect to their arrivalTime.
c) generateNRandomTasks(int): Generates N number of clients having their arrivalTime and serviceTime in the given min-max interval, then adds these clients to the generatedClients list, sorts the list and sets the client's ids in the order that they will be served.
d) run(): Overriden run method.
e) setOutputFile(String): Creates and opens the output file with the name given as parameter.

5) **Strategy:** This is an interface that has a single implementable method. It is used to choose between the possible strategies for adding a new Client to a Queue. In this simulation, only one strategy is implemented.
   a) addTask(List<Queue>, Client): Method to be implemented by a class that implements the Strategy interface. It is used to add a client to one of the queues, in the queueList, using a desired strategy.
6) **ShortestTimeStrategy:** This is the used strategy. It implements the Strategy interface and its addTask method.
   a) addTask(List<Queue>, Client): Implementation of the Strategy interface's addTask method. The method adds a client to the queue from the queue list, which has minimal waiting period.

5. Results

The application yields the log of the simulation. This log represents the state of the queueing system at each second. The application shows the current simulation time, the Client objects that are still “shopping” or aren’t placed in any Queue and each Queue with its clients. If a Queue is empty, the output will show that it is closed.

The output will be written in a file. This file’s name is given as a parameter when running the simulator’s jar file in the command prompt or the IDE’s terminal. When the simulator is ran, a file is created by the SimulatorManager class, in the setOutputFile method.

At the end of the output file, the average waiting time is printed. This is calculated during the simulation and it is equal to the sum of the time waited by each successfully processed clients, divided by the number of such clients. It is important to note that only the clients that have gotten through the whole “shopping” process are taken into consideration when computing this average.

6. Conclusions

By following the secondary objectives, the created queue simulator achieves its main objective. The created application is still basic and it could use some feature development, but it does the job it should.

The strategy of using concurrency, or multi-threading proved to be useful in the implementation. This way a single queue is simulated by the Queue thread and the whole “shop” and elapsed time is simulated by the SimulationManager. Multiple Queue threads are running in the same time and they

are paused each time a new client comes to the Queue's first position. It is the first time I used Threads and they proved to be very useful for this assignment. This approach of using concurrency made the implementation process a lot simpler and easier to code.

I have also learned a lot from the strategy approach. This is mostly useful in cases when multiple possible strategies need to be simulated. In this project I have only implemented one, so it is not very useful, but it provides a more abstract way to implement the simulator. The user can choose from the implemented strategies, that are all present in the SelectionPolicy enum class.

The computation of the average waiting time proved to be trickier than I first thought. This is because the clients that have to be taken into consideration are the ones that have gone through the whole shopping process and were also served in a queue.

In conclusion, this was an interesting project that helped me understand Java concurrency and the use of multithreading in general. As a secondary lesson, I have learned that real life simulations are hard very hard to implement. They are very unpredictable, and unpredictability is hard to write and manage in programming languages. This is the reason why there aren't many good simulation applications of real-life scenarios. In real life scenarios, not everything has to happen for a reason, while in programming it is the exact opposite. The statistics of the queueing system, given as output, could be used to create more effective ways of placing clients in a shop's queue, for example.

For future development, I have 3 ideas:

1) Implementation of a well-designed GUI that shows the live feed of the queuing-based system.
2) A graphical interface that shows the clients doing their shopping and waiting for their service.
3) Implementation of new strategies to simulate.
4) Better representation of clients and queues, by adding additional data and every queue having different speeds, according to the cashier's experience.

7. Bibliography

- http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_2/Java_Concurrency.pdf
- https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
- https://stackoverflow.com/questions/1082580/how-to-build-jars-from-intellij-properly