Documentation

<Queue Simulator>

Homework number: 3

Due date: week 8

# Purpose

The purpose of this assignment is to develop skills for solving real-life applications which have queues and to introduce the concept of multithreading in Java. The main goal is to learn how to handle programs which require multiple servers, which have to simultaneously serve a large number of clients. These clients can be modelled after real-life objects or they can represent parts of an embedded system which communicate with each other.

Another goal of this project is to introduce the concept of time in a software environment (for example simulating a weekday at a supermarket, in my case).

# Problem analysis

<Modelling, scenarios, usage>

The given client-server situation is the following: we are given a number of queues, each one working independently) and at undetermined time intervals, new customers arrive. Each customer spends some time at a queue. Customers try to go to the queue which is empty or has the least customers. If there are more customers at a queue, the customers that arrived later have to wait for the other customers in the front. This is a real-life situation, that you can find in any bigger store on hypermarket.

We are not interested in what the clients do, only in their processing time. Our main goal is to reduce the clients’ waiting time without introducing more queues (which is often imposed as a financial limitation). Instead we have to try to place each client in a que that has the minimal waiting time. This way we distribute the tasks between the “servers”. Not only the clients’ waiting time will decrease but we can also make sure that the load on the servers is balanced – in other words the average time a server “works” is nearly the same as for other servers.

The usage of the application is the following: at predefined time intervals, customers arrive. We have to assign each customer to a queue (which has the lowest waiting time). Each queue will be processed by a “server” which will serve/process only the first client at a time. All the other clients in the queue will be waiting as in a real-life queue (like at a shopping center). The number of such queues (each working in parallel) is specified at the beginning of the application. The simulation interval is also specified in advance. In my application, I simulated a day form 7:30 am to 9:30 pm (this interval can also be modified). After 9:30 pm, the application waits for all clients to exit the queue, however no new clients are allowed. At the end a report is generated showing data like the peak hour and the average waiting time.

An average scenario is presented below:

<flow chart of an average scenario>

# Design

<UML diagrams, data structures, class design, relationships, user interface>

## Class design

The structure of the classes follows the logical steps described in the Problem analysis (hint: each class is written with bold).

The entry point of the program is the **App** class, which has an Environment and a Simulator object (which is run in a thread).

The **Environment** Class has only static fields and it stores all the simulation-related information specified by the user (processing time of clients, simulation time interval, etc.). During execution, other classes are allowed to access the fields of this class, because these cannot change at runtime. After initializing the environment variables, the simulation is started.

The **Simulator** class keeps track of time – it has a variable, called *currentTime*, which is incremented at each step of the simulation. This class has the duty to instantiate the Frame class, which represents the graphical user interface. The main task is, however to generate Clients. This is done by the help of the *ClientGenerator* class. After generating the client, it is passed as an argument to another class, called *ClientScheduler*.

Each client is generated by the **ClientGenerator** class, whichhas a **ClientIdGenerator** (generates unique ID’s for clients – this is mostly for debugging purposes, but it helps by identifying each client) and a **ClientServiceTimeGenerator** (generates a random service time between the values specified in the environment).

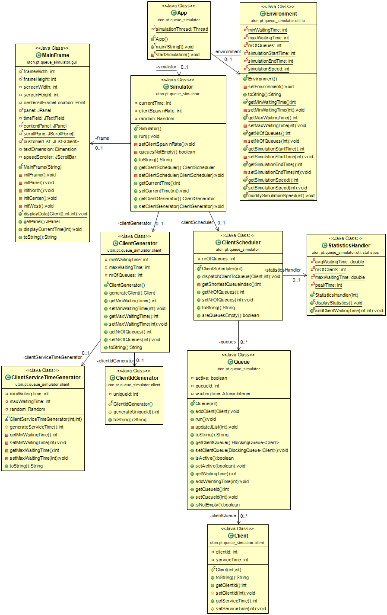
The **Client** objects will off course have the fields *clientId* and *serviceTime*.

The generated clients are processed by the **ClientScheduler**, which stores all the queues and can search for the queue having the shortest waiting time. The client will then be dispatched to this queue. This class also has to signal to the Simulator if none of the queues are active (this case is important when deciding when to end the simulation time).

A **Queue** is handled in a separate thread and does two tasks: adds a client to the end (when needed) and processes each client. The implementation of the queues (and also other classes) will be discussed later.

To process how the simulation behavers, I’ve included the **StatisticsHandler** class, which generates reports, for example the peak hour or the average waiting time of clients.

The Class diagram is presented below – please observe the clear hierarchy and the data-flow:



## Data structures

The first question, for me, of what data types to use was how to represent time. The java library already offers the Date class and many other implementations are already available, however I did not want to use what I do not necessarily need. I believe, when simulation a whole day, it is enough to count minutes (this is also how many real-time simulation games work). So I simply created an integer value, called *currentTime*. For example, 11:25 am is represented by the value 11\*60 + 25 = 685. The beginning and end time of the simulation also has to be specified in this format.

Each queue has its own field for storing the Clients – these are *BlockingQueue*s, implemented as *LinkedBlockingQueue*s. This is a thread-safe Collection which acts as a queue, which is exactly what my problem needs.

Each of these Queue objects is stored in the ClientScheduler class as an ArrayList<Queue>. The implementation here is not so important; we just need to access the queues by their index.

## User interface

The GUI is implemented in the **Frame** class having a Panel. I used the BorderLayout to manage the layout of the panel. The panel has the following regions:

* NORTH: displays the current time (implemented as a JTextField);
* WEST: lets the user adjust the simulation time (implemented as a JScrollBar);
* SOUTH: displays logging;
* CENTER: displays the contents of each queue in a JList. The header of each JList is the current waiting time at that queue; these values are dynamically updated.

<GUI>

At the end of the simulation the statistics are displayed in a different frame.

<Results>

# Implementation and testing

# Results

# Conclusions

<What have I learned, further improvements & development>

# Bibliography