Student: Gergő Papp-Szentannai
Group: 30421

Documentation

<Queue Simulator>

Homework number: 3
Due date: week 8

Contents

1. 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).

2. 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:

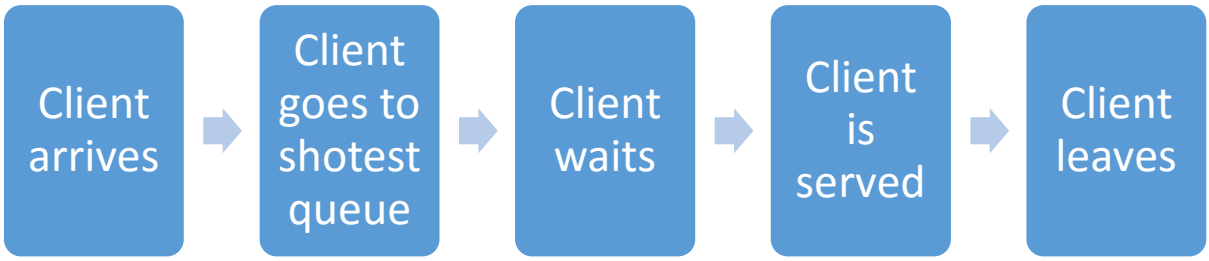

Figure 1

3. Design

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

a. 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, which has 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:

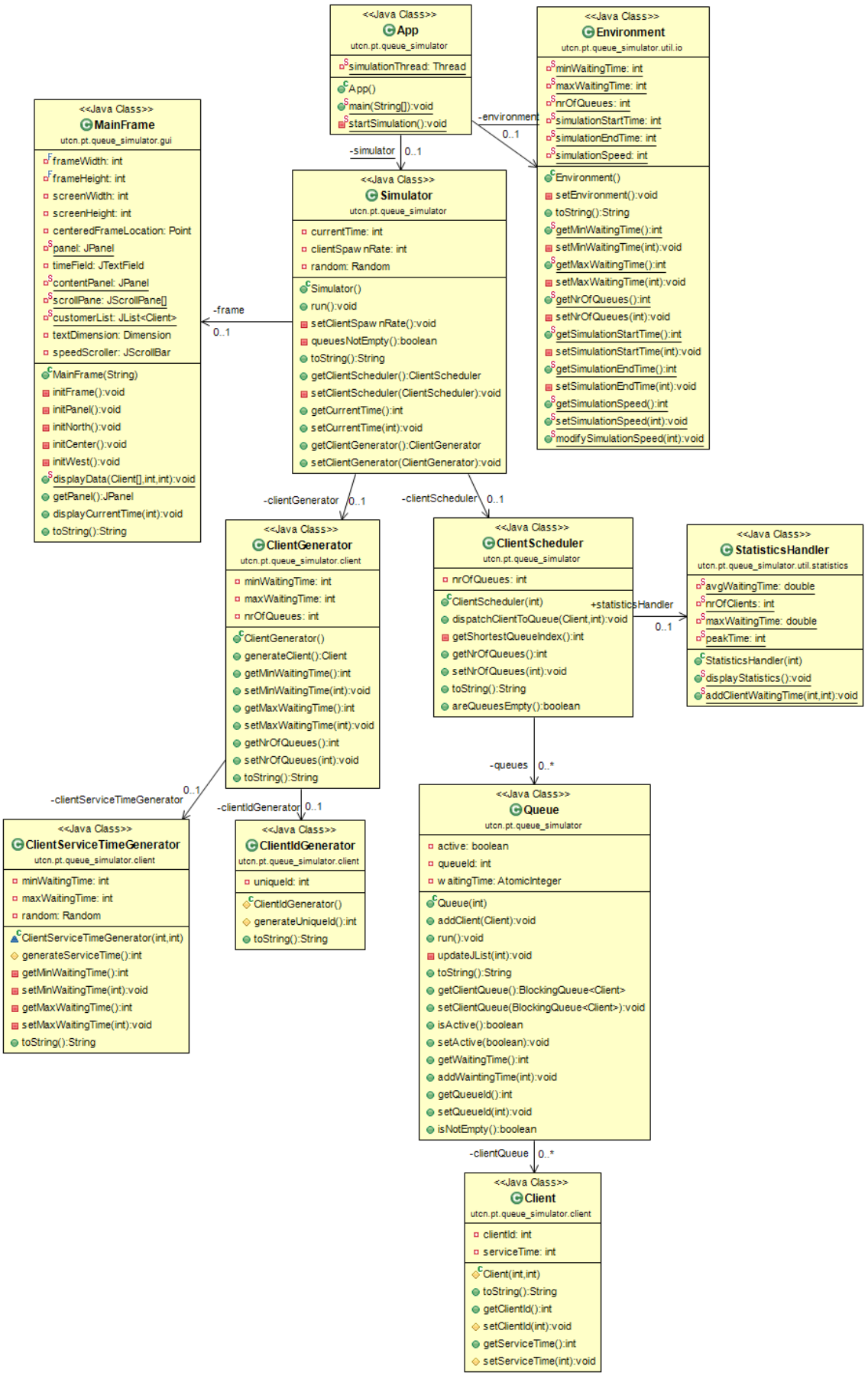
<<Java Class>>
App
utcn.pt.queue_simulator
simulationThread: Thread
App()
main(String[]):void
startSimulation():void
<<Java Class>>
Environment
utcn.pt.queue_simulator.util.io
minWaitingTime: int
maxWaitingTime: int
nrOfQueues: int
simulationStartTime: int
simulationEndTime: int
simulationSpeed: int
Environment()
setEnvironment():void
toString():String
getMinWaitingTime():int
setMinWaitingTime(int):void
getMaxWaitingTime():int
setMaxWaitingTime(int):void
getNrOfQueues():int
setNrOfQueues(int):void
getSimulationStartTime():int
setSimulationStartTime(int):void
getSimulationEndTime():int
setSimulationEndTime(int):void
getSimulationSpeed():int
setSimulationSpeed(int):void
modifySimulationSpeed(int):void
-environment
0..1
-simulator
0..1
<<Java Class>>
MainFrame
utcn.pt.queue_simulator.gui
frameWidth: int
frameHeight: int
screenWidth: int
screenHeight: int
centeredFrameLocation: Point
panel: JPanel
timeField: JTextField
contentPanel: JPanel
scrollPane: JScrollPane[]
customerList: JList<Client>
textDimension: Dimension
speedScroller: JScrollBar
MainFrame(String)
initFrame():void
initPanel():void
initNorth():void
initCenter():void
initWest():void
displayData(Client[],int,int):void
getPanel():JPanel
displayCurrentTime(int):void
toString():String
<<Java Class>>
Simulator
utcn.pt.queue_simulator
currentTime: int
clientSpawnRate: int
random: Random
Simulator()
run():void
setClientSpawnRate():void
queuesNotEmpty():boolean
toString():String
getClientScheduler():ClientScheduler
setClientScheduler(ClientScheduler):void
getCurrentTime():int
setCurrentTime(int):void
getClientGenerator():ClientGenerator
setClientGenerator(ClientGenerator):void
-frame
0..1
-clientGenerator
0..1
-clientScheduler
0..1
<<Java Class>>
ClientGenerator
utcn.pt.queue_simulator.client
minWaitingTime: int
maxWaitingTime: int
nrOfQueues: int
ClientGenerator()
generateClient():Client
getMinWaitingTime():int
setMinWaitingTime(int):void
getMaxWaitingTime():int
setMaxWaitingTime(int):void
getNrOfQueues():int
setNrOfQueues(int):void
toString():String
<<Java Class>>
ClientScheduler
utcn.pt.queue_simulator
nrOfQueues: int
ClientScheduler(int)
dispatchClientToQueue(Client,int):void
getShortestQueueIndex():int
getNrOfQueues():int
setNrOfQueues(int):void
toString():String
areQueuesEmpty():boolean
+statisticsHandler
0..1
<<Java Class>>
StatisticsHandler
utcn.pt.queue_simulator.util.statistics
avgWaitingTime: double
nrOfClients: int
maxWaitingTime: double
peakTime: int
StatisticsHandler(int)
displayStatistics():void
addClientWaitingTime(int,int):void
-queues
0..*
-clientServiceTimeGenerator
0..1
-clientIdGenerator
0..1
<<Java Class>>
ClientServiceTimeGenerator
utcn.pt.queue_simulator.client
minWaitingTime: int
maxWaitingTime: int
random: Random
ClientServiceTimeGenerator(int,int)
generateServiceTime():int
getMinWaitingTime():int
setMinWaitingTime(int):void
getMaxWaitingTime():int
setMaxWaitingTime(int):void
toString():String
<<Java Class>>
ClientIdGenerator
utcn.pt.queue_simulator.client
uniqueId: int
ClientIdGenerator()
generateUniqueId():int
toString():String
<<Java Class>>
Queue
utcn.pt.queue_simulator
active: boolean
queueId: int
waitingTime: AtomicInteger
Queue(int)
addClient(Client):void
run():void
updateJList(int):void
toString():String
getClientQueue():BlockingQueue<Client>
setClientQueue(BlockingQueue<Client>):void
isActive():boolean
setActive(boolean):void
getWaitingTime():int
addWaintingTime(int):void
getQueueId():int
setQueueId(int):void
isNotEmpty():boolean
-clientQueue
0..*
<<Java Class>>
Client
utcn.pt.queue_simulator.client
clientId: int
serviceTime: int
Client(int,int)
toString():String
getClientId():int
setClientId(int):void
getServiceTime():int
setServiceTime(int):void

Figure 2

b. 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.

c. 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 and the final statistics at the end of the simulation;
- 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.

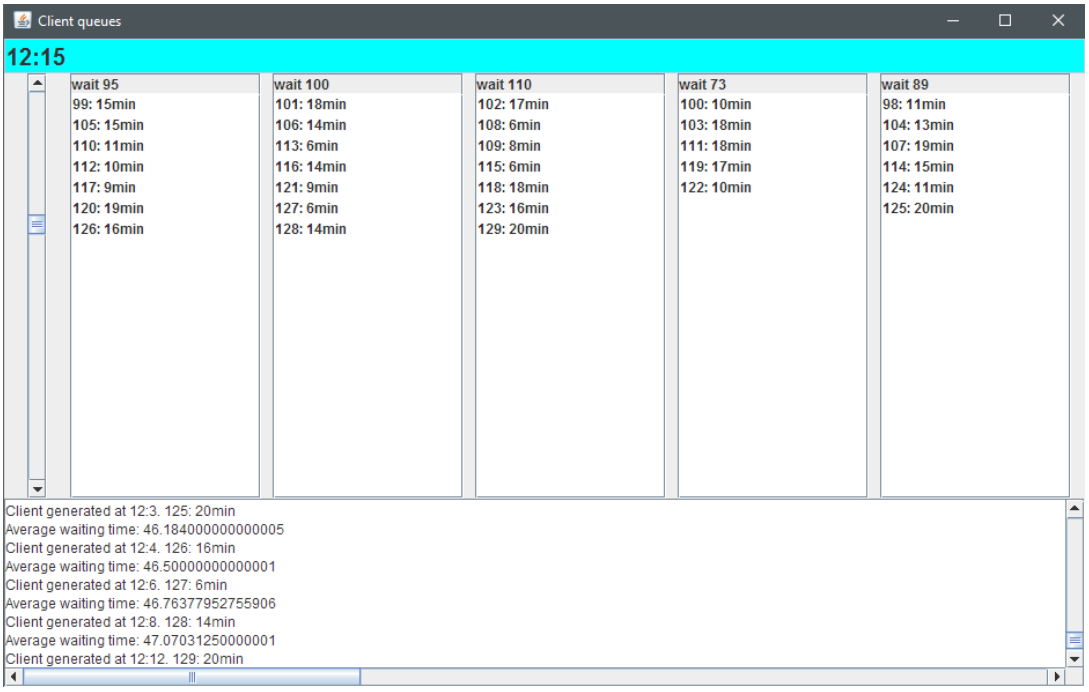

Figure 3

4. Implementation

When implementing the already discussed design, presented at chapter Class design, I took in consideration the following aspects:

A. The fields of the classes should have the most restrictive modifiers, such that they still fulfill their tasks.

The following example demonstrates this property:

```
private boolean active;
private int queueId;

private BlockingQueue<Client> clientQueue;
private AtomicInteger waitingTime;
```

To still be able to access some fields, getters and setters were generated, however these are also restrictive: getters are public, while setters are private or protected. Protected methods can only be accessed within the same package, this is one of the reasons why I have split my classes in many different packages – each representing one general tasks that its classes implement.

B. A method in a class should only be able to call methods which are right below the given class, presented in the class diagram (Figure 2).

For instance, when starting the program, the following sequence of calling different methods occurs:

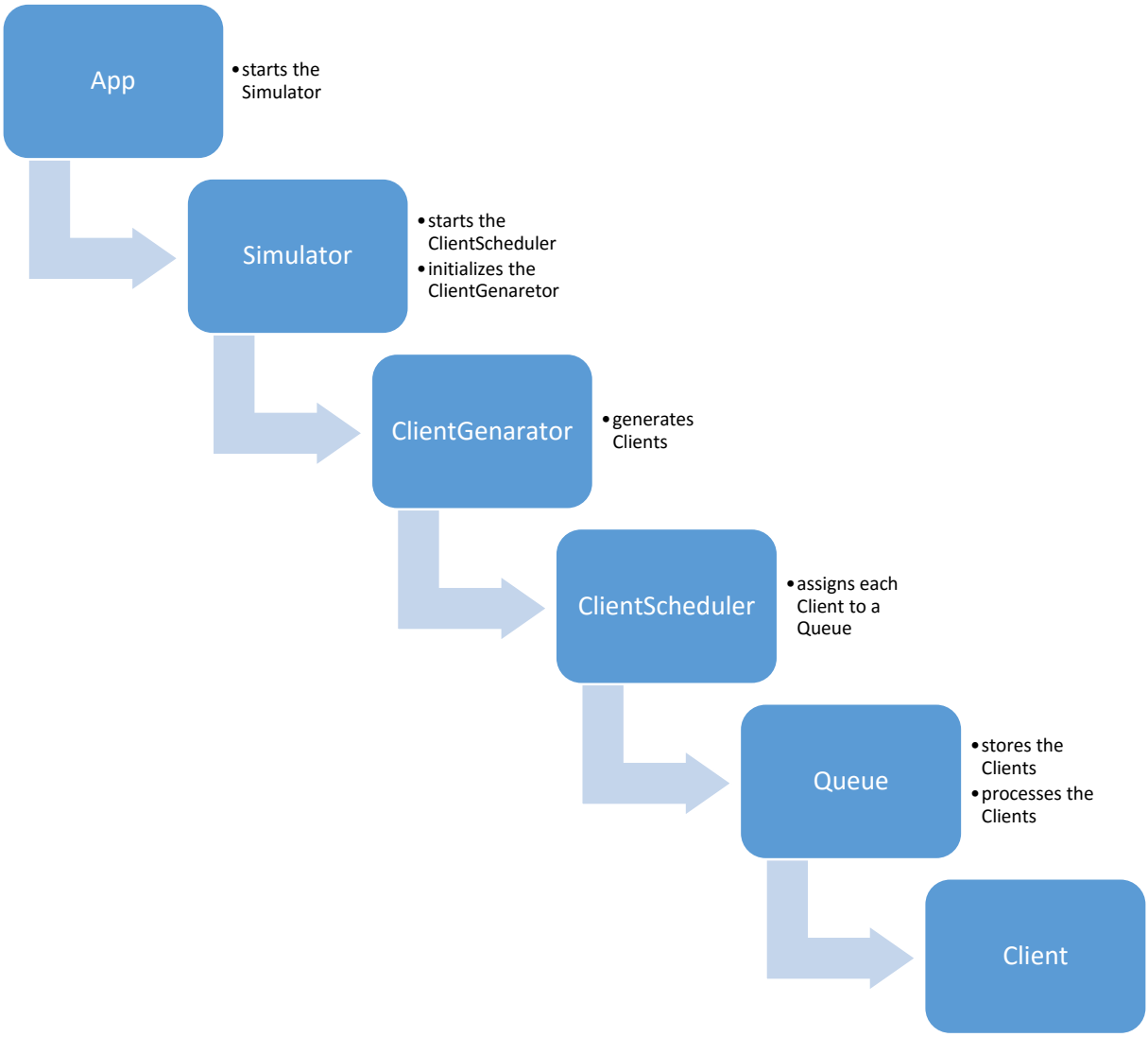

Figure 4

I have also identified some classes that do not fit into this hierarchy. These are the following classed:

- **Environment** – contains only static fields which can (and have to) be accessed by many classes. See the Class design chapter;
- **ConfigFileReadException** and **IllegalServiceTimeException** – these classes extend the Exception class and are thrown when the specified configuration file does not have the required format or some input values are not accepted (for example the service time is negative);
- **MainFrame** – this is the implementation of the graphical UI. Because more classes have to access and write to different panels, the MainFrame also has some static methods. These are the following:

```
public void displayCurrentTime(int currentTime)

public static void displayData(Client[] client, int queueId, int
waitingTime)

public static void printLogMessage(String message)
```

DisplayCurrentTime displays the current time as hours and minutes, displayData places all the current Clients in one of the lists representing a Queue and printLogMessage displays logging information. Only the first method is not static because the caller object already has a reference to the MainFrame (MainFrame was instantiated in the Simulator class, which also contains the current time).

C. Each queue should work in parallel

I used the already given Thread implementation in Java. Beside the automatically generated Main thread I created a minimum of two threads: one for the GUI and on for the Simulator. The simulator, then, depending on the number of required queues, created new/different threads for each Queue.

The hierarchy of threads (based on the parent-child relationship) is the following:

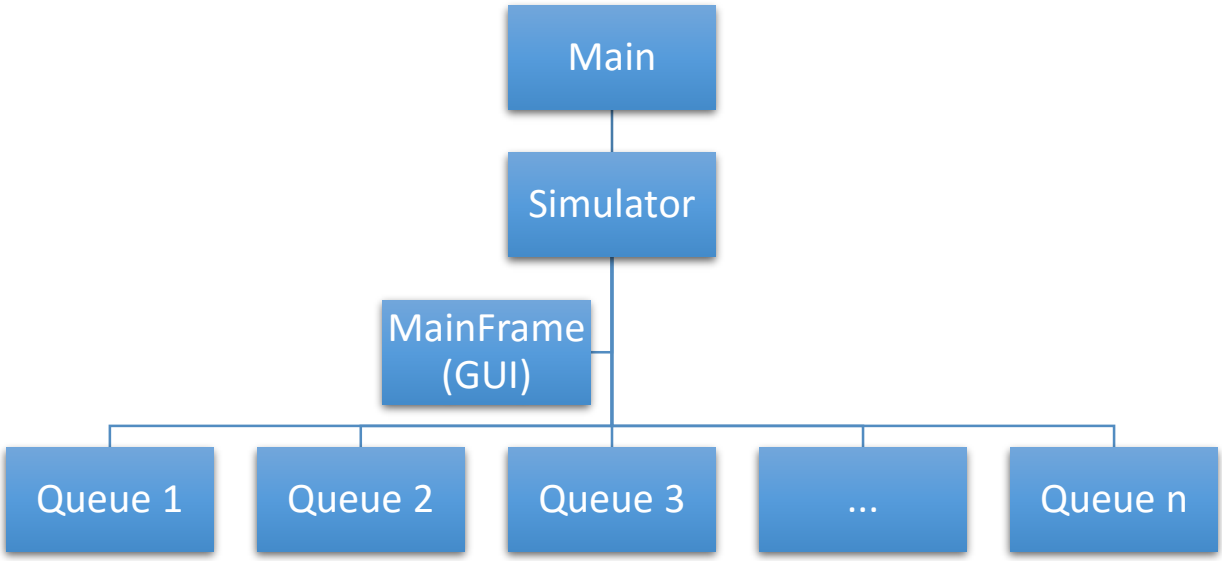

Figure 5

Each queue sleeps for its current client's processing time – this way I simulate the processing time and also let other threads to use the CPU resources. The simulator thread is set to high importance – if this thread would be blocked or would starve, the whole simulation might freeze/block.

At the end of the program, the queues remain blocked and the Simulator asks the StatisticsHandler to display the results on the logging panel. At this point the queues can be stopped. All the other threads will stop when exiting the Main thread (when closing the application).

5. Results

As a result, my program does the following tasks:

- Reads the simulation environment variables from an .xml file
- Generates clients
- Uses multiple threads to process the clients – multithreading!
- Logs events
- Shows the queue evolution in a GUI
- Computes the following:
    - Average waiting time at each moment (including total average waiting time/client)
    - Processing time for each client (generated randomly between boundary values)
    - Service time – the simulation interval is also specified
    - Computation of the peak hour
    - Waiting time at each queue at any given moment (updated automatically)
- And many other side-tasks that are required…

6. Conclusions

<What have I learned, further improvements & development>

I have learned how to design and implement server-client situations using object oriented programming. I have also learned the main concepts of multithreading and the benefits of it, but also the synchronization issues that may occur (and how to solve these). I have also learned how to use some new Java Swing components that I haven't used so far.

Based on the sketch of this program, many other client-server problems could be modelled, while keeping the same logical structure.

7. Bibliography

I only used fast google searches and stackoverflow.com for solving most of my problems. I also inspected the *JavaDoc*s of many already implemented classes. Finally, I also took in consideration the good advices my teaching assistant gave us.