Queueing Simulator

1. Objectives

The objective of this application is to create a system that simulates a scenario where clients arrive at an establishment, place themselves in one of several queues, wait until their turn comes up and are served.

To accomplish this objective, we identify three sub-objectives:

- to create a model that simulates the functionality of the queues and of the customers
- to engineer a solution permitting the real-time display of the above model
- to find a way to store a log of events for a simulation, so that the results can be reviewed afterwards

The first two sub-objectives are treated more generally in Chapter 3.2., while the exact implementation is touched upon in Chapter 4: 4.1, 4.2, 4.3, 4.4, 4.5, 4.6 touch upon the problem of modelling the queues, 4.7 and 4.8 upon real-time display, 4.7 and 4.9 upon logging and 4.10 connects all of the sub-goals into a final product.

2. Problem analysis

This chapter aims to describe the requirements of the problem as well as how the end-users will interact with the application, together with any assumptions made about the input of such end-users.

2.1. Functional requirements

The completed application must provide the following functionality:

- Read the simulation settings provided by the user

- Simulate a scenario according to the received settings, which involves:
    - Constructing a number of execution threads simulating a queue each
    - Distribute the customers among these queues according to a chosen strategy
    - Monitor relevant data about the customers for each queue, such as average waiting time and peak waiting time
- Present the simulation on the screen in real-time
- Log the events of the simulation and present them to the user in a way that can be viewed after the simulation has ended

2.2. Assumptions

For an easier implementation of the application, we assume that the user will input correct data into all of the required fields.

Moreover, we assume that the user will not input contradicting data, such as a maximum serving time shorter than the minimum serving time.

The end user will have to provide the following data:

- the total simulation time
- the number of queues
- the total number of clients
- the minimum and maximum serving time for clients
- the strategy used when choosing which queue to put clients in:
    - Shortest Queue - choose the queue with the fewest clients
    - Shortest Time - choose the queue with the lowest remaining time before the new client can be served

2.3. Use Cases

There is a single way of using the application; all that differs between several uses are the settings employed. The general way of using the application is as follows:

Actors: User
Preconditions: none
Main success scenario:

1. User correctly introduces the simulation settings
2. User clicks a button to start the simulation
3. The system reads the simulation settings
4. The system initialises the queues as threads and creates a set of simulated customers
5. The system performs the simulation, simultaneously displaying the process and logging the events
6. The system finishes the simulation

Alternative sequences:

a) Wrong input

1. User introduces data in a wrong format; e.g. 1A2 instead of an integer
2. User clicks a button to start the simulation
3. The system reads the simulation settings and notices the error
4. The system ignores the wrong fields, using instead the last-used value for those fields (or the default value, for the first run)
5. The scenario continues to the original step 4

Error sequences:

a) Minimum value higher than maximum value

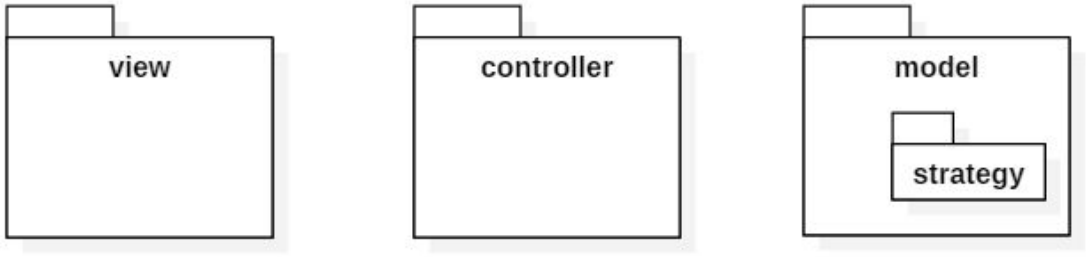
1. User introduces data where the minimum serving time is higher than the maximum serving time
2. User clicks a button to start the simulation
3. The system reads the simulation settings and detects the error. At this point, the use case fails.

3. Design

This chapter deals with how the the problem was approached from an object-oriented perspective, how it was split into packages, what interfaces and data structures were used and the algorithms that were employed.

3.1. Package distribution

Following the MVC guidelines, the application was split into three big packages. Furthermore, the *model* package contains the *strategy* sub-package, which was created in order to separate a functionally-independent part from the rest.



Between these packages, *controller* is connected to both *view* and *model*, but *model* and *view* only have access to the interfaces defined in *controller*. Moreover, there is no connection between *view* and *model*.

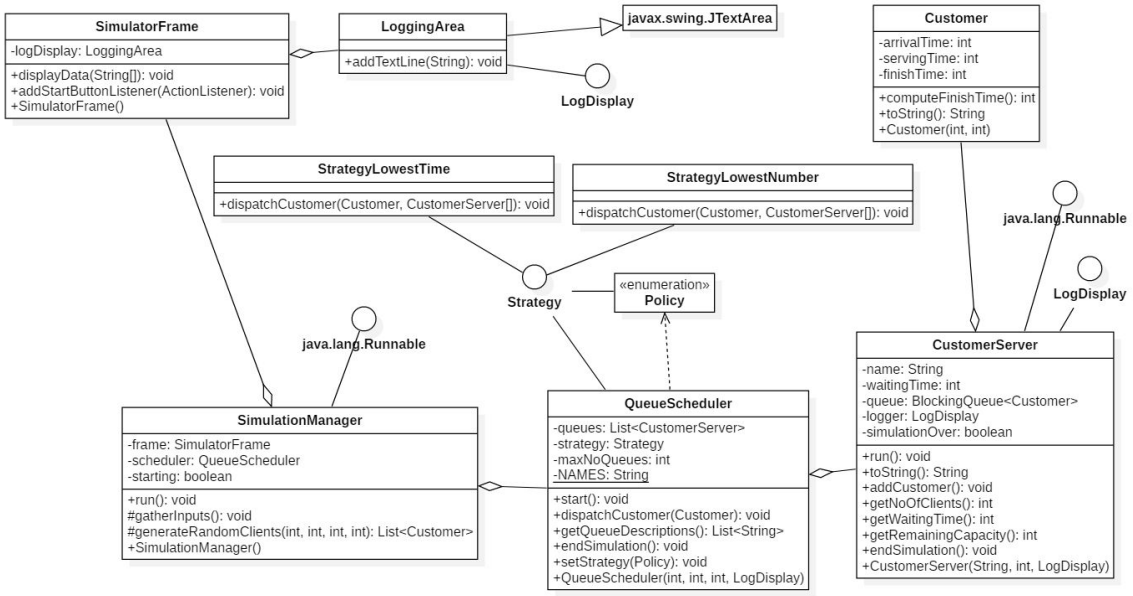
3.2. Class Design

The basic elements of this project are the clients, which are represented by the Customer class. Queues (and the clerks or cashiers they represent) are CustomerServer objects. A collection of queues is coordinated by a QueueScheduler, who is in charge of distributing clients to each queue.

The UI is comprised of a SimulationFrame class, augmented by a LoggingArea object - a special type of JTextArea that implements the LogDisplay interface and is used for logging purposes.

Finally, controlling and connecting these two is the SimulationManager - an object in charge of setting up the parameters of the simulation and bridging between the model and the UI.

All of these can be seen in the class diagram below (where, for brevity, we have omitted getters, setters and other straightforward and not very important methods):



4. Implementation

This chapter describes the classes and interfaces in more detail, covering the most important methods and aspects of each.

4.1. Customer

A Customer is defined by three parameters: when it arrives, how long it takes for it to get served and when it leaves:

```
private int arrivalTime;
private int servingTime;
private int finishTime;
```

The most important method of this class is the *toString()* method, because it is used extensively when logging.

```
/**
 * Returns the arrival and servicing time for this Customer, in the
 * (arrival time, serving time) format. E.g. for a Customer with arrival
 * time 7 and serving time 2, it will return "(7, 2)".
 */
@Override
public String toString() {
    return "(" + arrivalTime + ", " + servingTime + ")";
}
```

The Customer can also compute its own finish time, long as it is told how long it needs to wait:

```
/**
 * Computes the finish time for this Customer, given the time it had to wait in queue
 * @param waitingTime - the time the Customer waited before s/he started being served
 */
public void computeFinishTime(int waitingTime) {
    this.finishTime = arrivalTime + waitingTime + servingTime;
}
```

4.2. CustomerServer

The CustomerServer represents a single clerk / cashier / queue. Its most important attributes are the queue itself, which is a *BlockingQueue* of Customers and the current waiting time that any newly-arrived Customers would have to bear with, which is an *AtomicInteger*:

```
private BlockingQueue<Customer> queue;
private AtomicInteger waitingTime;
```

These representations were chosen because we will be working concurrently, and clients might be added and removed at the same time.

There are two very important methods for the CustomerServer class. The first of them is *addCustomer(Customer)*:

```java
/**
 * Adds the given Customer to the queue of this CustomerServer. This method
 * automatically increments the waiting time for the queue.
 * @param cust - the Customer to be added to queue
 */
public void addCustomer(Customer cust) {
    try {
        cust.computeFinishTime(waitingTime.get());
        queue.put(cust);

        waitingTime.addAndGet(cust.getServingTime());

        logger.addTextLine(name + " received customer " + cust.toString());
    } catch (InterruptedException e) {
        System.out.println("Interrupted in addCustomer!");
        e.printStackTrace();
    }
}
```

Notice how the waiting time needs to be incremented, even though this method will likely be called from within the controller package, from a separate thread. Also notice how the event of receiving a new Customer is logged for further inspection.

The second and more important method of this class is the *run()* method. This is because CustomerServer implements the *java.lang.Runnable* interface, so that it can operate concurrently with the main thread.

```
/**
 * Simulates the behaviour of a queue by serving clients in order and waiting
 * an appropriate amount for each client. New clients may be added while the
 * simulation runs by inserting them with the {@link #addCustomer(Customer)}
 * method of this class.
 */
@Override
public void run() {
    while (!simulationOver) {
        try {
            Customer currentlyServed = queue.take();

            // Wait in intervals of one second at a time, for better accuracy
            // of remaining waiting time for new-arriving Customers
            for(int i = currentlyServed.getServingTime(); i > 0; --i) {
                waitingTime.decrementAndGet();
                Thread.sleep(1000);
            }

            logger.addTextLine(name + " finished serving customer " +
                               currentlyServed.toString());
        } catch (InterruptedException e) {
            System.out.println("Interrupted in CustomerServer#run!");
            e.printStackTrace();
        }
    }
}
```

In this method, too, we log our events. However, another two important aspects are the infinite loop conditioned by *simulationOver* (it is a boolean that is externally set to *false* when the simulation should terminate) and the usage of *take()* to get an element out of the queue (it is a *BlockingQueue*-specific method that causes the thread to wait if it tries to take an element from an empty queue).

4.3. Queue Scheduler

A QueueScheduler controls several CustomerServer objects, deciding how to attribute new Customers to one of the queues. Because of this, its two most important attributes are the list of CustomerServer elements, and the Strategy which is used for choosing:

```
private List<CustomerServer> queues;

private Strategy strategy;
```

The most important method of this class is the *start()* method, whose role is to wrap the *Runnable* CustomerServer objects into *Thread* objects and start them. This is not

done in the constructor, because we may want to initialise data or make preparations (such as setting a specific Strategy) before starting the simulation.

```
/**
 * Starts the simulation as per the conditions of {@link #QueueScheduler(int, int, int)}
 * and returns a list of Threads containing the threads that were generated.
 * @return a list of Threads containing one thread for each queue generated
 */
public List<Thread> start() {
    List<Thread> threads = new ArrayList<Thread>();

    for (CustomerServer server : queues) {
        Thread t = new Thread(server);
        threads.add(t);
        t.start();
    }

    return threads;
}
```

The resulting *Thread* objects are returned by the *start()* method; although this is not currently used, future extensions of the application that may wish to do more thorough work on threads may find it useful to have access to the running threads.

4.4. Strategy

Strategy is an interface used for the strategy pattern. The aim of using this design pattern is to abstract the method of dispatching a Customer to a queue away from the class methods. This way, it is easy to change the way clients are distributed or even to create new ways of distributing them (all it takes is to create a new implementation of the interface).

The interface defines only one method, *dispatchCustomer()*, but it also defines an enumeration type, Policy, that can be used to select Strategy implementations.

```
public interface Strategy {

    public enum Policy {
        /**
         * Customers are assigned to the queue with the fewest people
         */
        SHORTEST_QUEUE,
        /**
         * Customers are assigned to the queue with the shortest waiting time
         */
        SHORTEST_TIME
    }

    /**
     * Assigns the given Customer to one of the CustomerServers given.
     * The exact CustomerServer chosen depends on the implementation.
     * @param cust - the Customer to be added to a queue
     * @param queues - the CustomerServer queues to which to add the Customer
     * @see StrategyLowestNumber#dispatchCustomer(Customer, List)
     * @see StrategyLowestTime#dispatchCustomer(Customer, List)
     */
    public void dispatchCustomer(Customer cust, List<CustomerServer> queues);
}
```

4.5. StrategyLowestNumber

An implementation of Strategy that always places new clients into the queue with the fewest people.

```
public class StrategyLowestNumber implements Strategy {

    /**
     * Assigns the given Customer to the CustomerServer with the lowest number
     * of clients in the given list. This is a more natural representation of
     * customer behaviour, who often judge based on "rules of thumb" such as
     * the number of people in each queue.
     * @param cust - the Customer to be added to a queue
     * @param queues - the CustomerServer queues to which to add the Customer
     */
    @Override
    public void dispatchCustomer(Customer cust, List<CustomerServer> queues) {
        // Find the CustomerServer with the lowest number of clients
        CustomerServer target = queues.get(0);
        for(CustomerServer server : queues) {
            if(server.getNoOfClients() < target.getNoOfClients()
                    && server.getRemainingCapacity() > 0) target = server;
        }

        // Assign the Customer there
        target.addCustomer(cust);
    }

}
```

4.6. StrategyLowestTime

An implementation of Strategy that places clients in the queue where they will have the lowest time to wait before being served.

```
public class StrategyLowestTime implements Strategy {

    /**
     * Assigns the given Customer to the CustomerServer with the lowest
     * waiting time from the given list. This guarantees that each client
     * will be served in the lowest possible time.
     * @param cust - the Customer to be added to a queue
     * @param queues - the CustomerServer queues to which to add the Customer
     */
    @Override
    public void dispatchCustomer(Customer cust, List<CustomerServer> queues) {
        // Find the CustomerServer with the lowest waiting time
        CustomerServer target = queues.get(0);
        for(CustomerServer server : queues) {
            if(server.getWaitingTime() < target.getWaitingTime()
                    && server.getRemainingCapacity() > 0) target = server;
        }

        // Assign the Customer there
        target.addCustomer(cust);
    }

}
```

4.7. LoggingArea

A LoggingArea is a type of *JTextArea* that has been slightly modified to implement the LogDisplay interface from the *controller* package. It only differs from a regular *JTextArea* through one method - *addTextLine(String)*, which adds on top of the existing text of the text area instead of overwriting it:

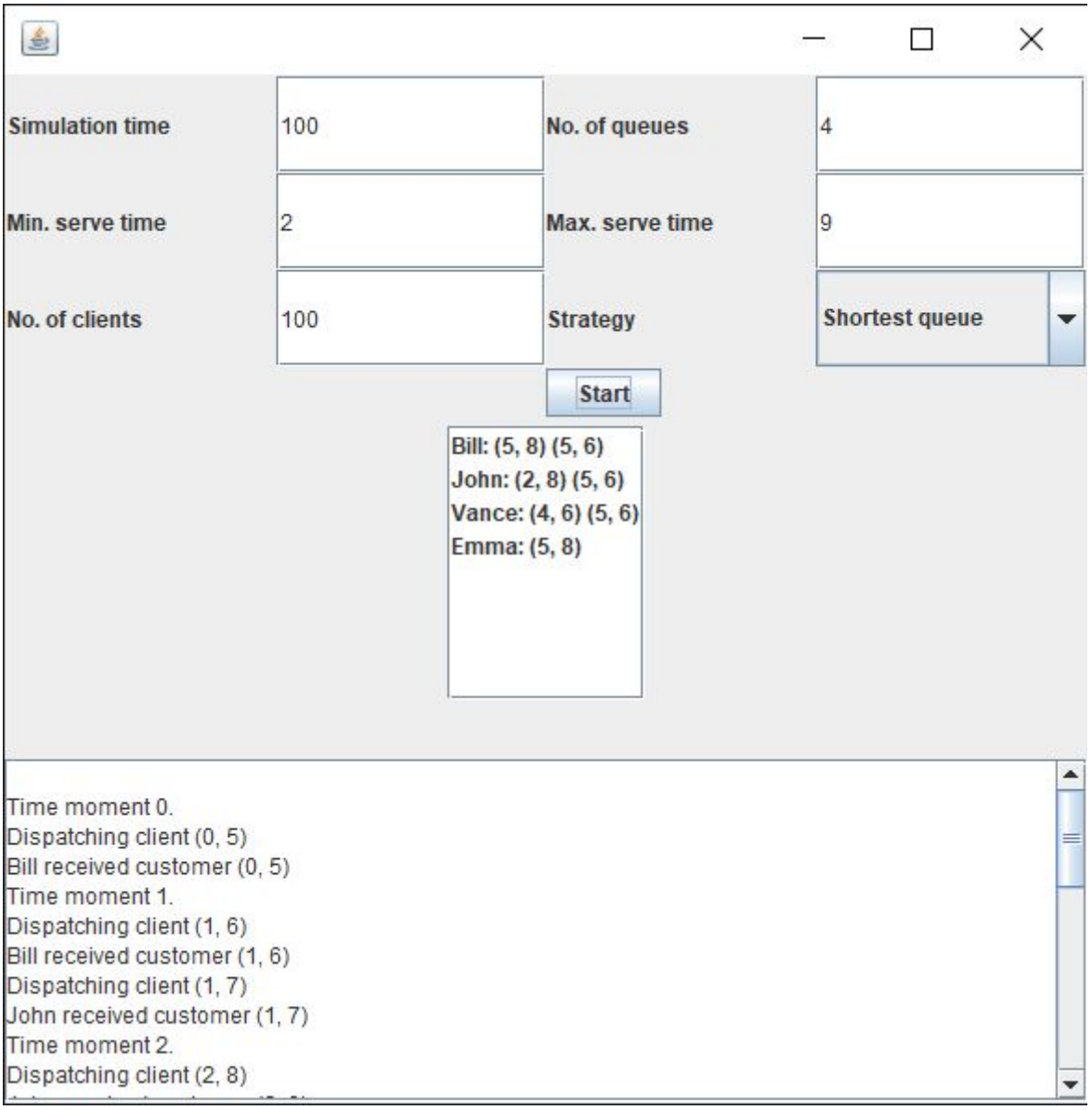
```
public void addTextLine(String text) {
    setText(getText() + "\n" + text);
}
```

Note that although this is a *view* class, since it only deals with the UI, it never becomes visible to the *model* package. Instead, it is masked with the aid of an interface, as we will see later.

4.8. SimulatorFrame

The SimulatorFrame is a heavy class that contains all the elements needed for displaying the simulation. It is also advantageously a subclass of *JFrame*, so that merely instantiating one and setting it to be visible creates an application window.

When the application is running, the result is as follows:



The UI is split into three parts, separated using a *GridLayout*: the top part is where the input happens, the central part is dedicated to real-time simulation of the queues, while the bottom part is a log of events modelled with a LoggerArea.

Internally, the SimulatorFrame mostly deals with setting or getting data or otherwise with setting up its structure. The one method that stands out is *displayData(List<String>)* with its helper method *createDataScrollPane(List<String>)*, whose purpose is to convert the raw input data (in String format) into UI-ready data (a *JScrollPane* placed on a *JPanel* in the main pane).

```
/**
 * Displays the given data pertaining to the queues in the simulation
 * window. The data set is expected to contain one String representing
 * the current state for each queue.
 * @param queuesData - the data to be displayed
 */
public void displayData(List<String> queuesData) {
    simulationPanel.removeAll();

    JScrollPane dataPane = createDataScrollPane(queuesData);

    simulationPanel.add(dataPane);
    simulationPanel.repaint();
    simulationPanel.validate();
}

/**
 * Creates a new JScrollPane that holds the data pertaining to the queues
 * that has been passed as a parameter.
 * @param queuesData - the data to be displayed
 * @return a JScrollPane that contains the given data
 */
private JScrollPane createDataScrollPane(List<String> queuesData) {
    // We use toArray because JList does not support
    // getting initial data from a List, but only from an array
    JList<String> queueList = new JList<String>(queuesData.toArray(new String[0]));
    return new JScrollPane(queueList);
}
```

4.9 LogDisplay

This is an interface whose sole purpose is to mask the underlying implementation (a LoggerArea) from its actual functionality, which is not related directly to the UI (being a logger of events).

```
public interface LogDisplay {
    public String getText();
    public void setText(String text);
    public void addTextLine(String text);
}
```

By acting this way, we can avoid creating a connection between the *model* and the *view*, because all the *model* needs to know about is one interface located in the *controller*. Even if the implementation were to change in the future (for instance, using a *java.util.logging.Logger* instead of this *JTextArea* approach), we only need to map that new implementation to this interface and the code can remain unchanged.

4.10. SimulationManager

This is the class in charge of controlling the simulations. It collects data from the UI and sets the parameters of the simulation, then tasks the QueueScheduler to start the required threads and to dispatch to them the Customers.

The most important attributes of the SimulationManager are its connection to the *view* (a SimulatorFrame), its connection to the *model* (a QueueScheduler) and its data (a randomly-generated list of Customers).

```
private SimulatorFrame frame;
private QueueScheduler scheduler;
private List<Customer> generatedClients;
```

In terms of methods, the most important method of the SimulationManager is its *run()* method. Other important methods are *gatherInputs()* and *generateRandomClients(int,int,int,int)*.

```
@Override
public void run() {
    while(true) {
        // Hacky way to not start until the button is pressed
        while(!starting) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e1) {
                System.out.println("Interrupted while waiting to start");
                e1.printStackTrace(); // TODO: same as always
            }
        }

        // Preparatory phase
        gatherInputs();
        frame.getLoggingArea().setText("");

        // Initialise values
        scheduler = new QueueScheduler(timeLimit, numberOfQueues, maxClientsPerQueue, frame.getLoggingArea());
        scheduler.setStrategy(selectionPolicy);

        generatedClients = generateRandomClients(numberOfClients, minProcesingTime, maxProcessingTime, timeLimit);

        // Begin simulation
        scheduler.start();
        for(int currentTime = 0; currentTime <= timeLimit; ++currentTime) {
            frame.getLoggingArea().addTextLine("Time moment " + currentTime + ".");
```

```
            // Have Customers arrive
            for(Customer c : generatedClients) {
                if (c.getArrivalTime() == currentTime) {
                    frame.getLoggingArea().addTextLine("Dispatching client " + c.toString());
                    scheduler.dispatchCustomer(c);
                }
            }

            // Update UI
            frame.displayData(scheduler.getQueueDescriptions());

            // Skip one second
            try {
                Thread.sleep(1000);
                //this.wait(1000);
            } catch (InterruptedException e) {
                System.out.println("Interrupted in SimulationManager#run!");
                e.printStackTrace();
            }
        }

        scheduler.endSimulation();
        starting = false; // Set this value to false at the end, so the button must be re-pressed
    }
}
```

The operation of this method is as follows: in an infinite loop (so that the user can do multiple simulations in a row), we start by waiting for the signal *starting* (this is a boolean flag that is set to *true* when the “Start” button is clicked).

Once the flag has been set, we begin by clearing previous output and taking in the data from the input. Following that, it becomes possible to set up the QueueScheduler and prompt it to start.

Following this step, the method acts like a clock, checking each second for clients that need to be dispatched. These clients are sent to the QueueScheduler who places them in queues as it sees fit. With every loop, we update the UI, to obtain a real-time simulation of events (within one-second accuracy, at least).

Finally, once the simulation time has elapsed, we announce all sub-threads that their work is over with *endSimulation()*, so that they may stop.

```
/**
 * Collects the input data from the UI. Any incorrectly formatted
 * data will be ignored, using instead the last used values (or the default)
 * Currently gathered data:
 * <ul>
 *  <li> Total simulation time </li>
 *  <li> Number of queues </li>
 *  <li> Number of customers </li>
 *  <li> Minimum and maximum serving time per customer </li>
 *  <li> Customer allocation strategy </li>
 * </ul>
 */
protected void gatherInputs() {
    try { timeLimit = Integer.parseInt( frame.getTime() ); } catch (NumberFormatException e) {}
    try { numberOfQueues = Integer.parseInt( frame.getNoQueues() ); } catch (NumberFormatException e) {}
    try { numberOfClients = Integer.parseInt( frame.getClients() ); } catch (NumberFormatException e) {}
    try { minProcesingTime = Integer.parseInt( frame.getMinServeTime() ); } catch (NumberFormatException e) {}
    try { maxProcessingTime = Integer.parseInt( frame.getMaxServeTime() ); } catch (NumberFormatException e) {}
    selectionPolicy = frame.getStrategy() == 0 ? Policy.SHORTEST_QUEUE : Policy.SHORTEST_TIME;
}
```

The *gatherInputs()* method queries the SimulationFrame for data. If the data is intact, then it is stored. If, however, the data is incorrect, this method detects it and re-uses the previous set of data (by simply not changing it). It's worth noting that there is an error that this method does not detect, namely a user inputting a minimum service time greater than the maximum service time.

```
/**
 * Generates a list of randomised Customers with the given parameters. The
 * list is guaranteed to be sorted in ascended order of arrival times. It is guaranteed
 * that the ideal finishing time of each client (that is, the finish time assuming they
 * are served as soon as they arrive) is no greater than <b>maxTime</b>.
 * @param number - the number of clients to generate
 * @param minProcTime - the minimum processing time for each client
 * @param maxProcTime - the maximum processing time for each client
 * @param maxTime - a maximum bound on the client's time as per the description
 * @return a list of randomised Customers
 */
protected List<Customer> generateRandomClients(int number, int minProcTime, int maxProcTime, int maxTime) {
    List<Customer> customers = new ArrayList<Customer>(numberOfClients);

    Random rand = new Random();
    for(int i = 0; i < number; ++i) {
        int procTime = minProcTime + rand.nextInt(maxProcTime - minProcTime);
        int arrivTime = rand.nextInt(maxTime - procTime);

        customers.add(new Customer(arrivTime, procTime));
    }

    // Sort customers in ascending order of arrival, for commodity
    customers.sort( (Customer a, Customer b) -> Integer.compare(a.getArrivalTime(), b.getArrivalTime()) );

    return customers;
}
```

Finally, the *generateRandomClients(int, int, int, int)* method is in charge of creating the randomised clients that are distributed into the queues during the simulation. The method sorts the resulting list by arrival time, so that selecting the correct users for each time is simpler.

Note that although the method guarantees that, **in ideal conditions**, all clients would finish being served by the time the simulation ends, it is easy to see that in real conditions, that is not always true (for example, 100 clients to be served by 1 queue, in only 30 seconds). For this reason, it is possible for the simulation to end while there still are active clients.

5. Results

Via rigorous testing and especially via inspection of the application logs, it has been concluded that the program is functioning properly.

However, a number of small details have been observed to allow room for improvement:

- Inputting a minimum and maximum value for the processing time so that max ≤ min leads to a runtime error and a failure of the use-case
- Because of how the program is implemented, it is possible to have clients still in queue when the simulation is over
- Inputting so many clients that the queues fill up (about 15 clients per queue) will cause unpredictable behaviour, since *BlockingQueue* works with a fixed size. (We expect that, since using *put()*, it should cause the main thread to wait until the queue becomes emptier, essentially “freezing the clock” for a few moments).
- CustomerServers behave in an initially confusing (but ultimately logical) way: if a CustomerServer was free and is assigned a task in a given time unit, it will count that time unit towards completing the task; if a CustomerServer was busy, finished its current task in a given time unit and started a new one, it will not count that time unit towards completing the task (since it had already worked on the completed task.

This final observation also produces the interesting result that a client with a processing time of 2 assigned at moment 0 will be “finished” at some point in moment 1, rather than in moment 2 as one might expect.

6. Conclusions

By working on this task, I believe I have gained a better understanding of both threading / concurrency in general, as well as of the specifics of threading in Java.

As future expansion of this application, it could be adapted with a better algorithm, so that the number of queues open can vary dynamically with the demand. Another possible future task would be to create an improved algorithm for generating random clients - or for simulating - so that by the end of the simulation, all clients have been served.

7. Bibliography