TECHNICAL UNIVERSITY OF CLUJ-NAPOCA ROMANIA

Thread Processing

Student: Bețianu Miruna – Laura
Group: 30422

Table of Contents

1. Task

Design and implement a simulation application aiming to analyse queuing based systems for determining and minimizing clients' waiting time.

2. Project specifications

Queues are commonly used to model real world domains. The main objective of a queue is to provide a place for a "client" to wait before receiving a "service". The management of queue-based systems is interested in minimizing the time amount their "clients" are waiting in queues before they are served. One way to minimize the waiting time is to add more servers, i.e. more queues in the system (each queue is considered as having an associated processor) but this approach increases the costs of the service supplier. When a new server is added the waiting customers will be evenly distributed to all current available queues.

The application should simulate a series of clients arriving for service, entering queues, waiting, being served and finally leaving the queue. It tracks the time the customers spend waiting in queues and outputs the average waiting time. To calculate waiting time we need to know the arrival time, finish time and service time. The arrival time and the service time depend on the individual clients – when they show up and how much service they need. The finish time depends on the number of queues, the number of clients in the queue and their service needs.

3. Composition

1. Packages

- model
- view
- controller

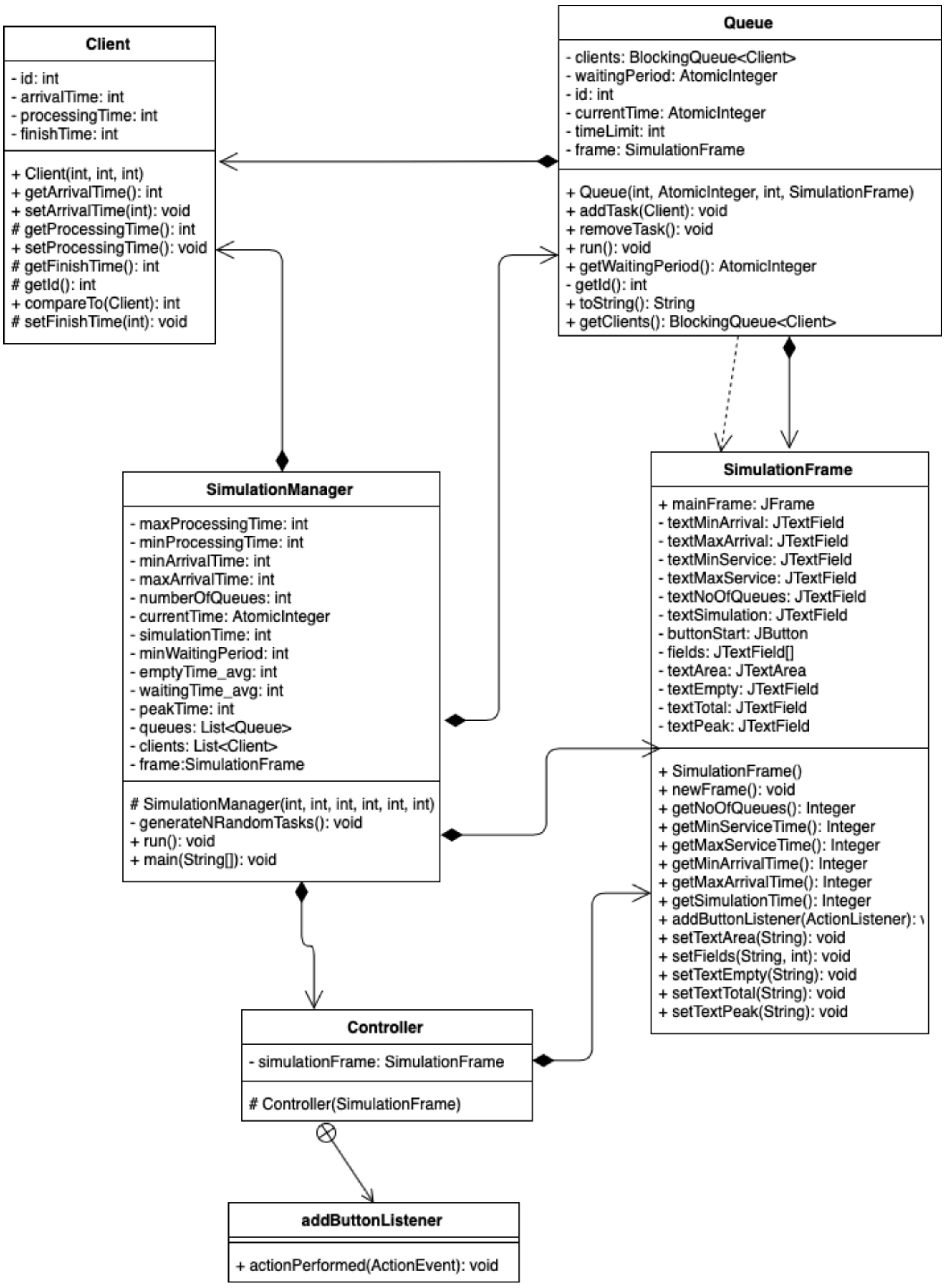
2. Classes

- Client
- Queue

- SimulationFrame

- Controller
- SimulationManager

The UML Class Diagram:


Client
- id: int
- arrivalTime: int
- processingTime: int
- finishTime: int
+ Client(int, int, int)
+ getArrivalTime(): int
+ setArrivalTime(int): void
# getProcessingTime(): int
+ setProcessingTime(): void
# getFinishTime(): int
# getId(): int
+ compareTo(Client): int
# setFinishTime(int): void
Queue
- clients: BlockingQueue<Client>
- waitingPeriod: AtomicInteger
- id: int
- currentTime: AtomicInteger
- timeLimit: int
- frame: SimulationFrame
+ Queue(int, AtomicInteger, int, SimulationFrame)
+ addTask(Client): void
+ removeTask(): void
+ run(): void
+ getWaitingPeriod(): AtomicInteger
- getId(): int
+ toString(): String
+ getClients(): BlockingQueue<Client>
SimulationManager
- maxProcessingTime: int
- minProcessingTime: int
- minArrivalTime: int
- maxArrivalTime: int
- numberOfQueues: int
- currentTime: AtomicInteger
- simulationTime: int
- minWaitingPeriod: int
- emptyTime_avg: int
- waitingTime_avg: int
- peakTime: int
- queues: List<Queue>
- clients: List<Client>
- frame:SimulationFrame
# SimulationManager(int, int, int, int, int, int)
- generateNRandomTasks(): void
+ run(): void
+ main(String[]): void
SimulationFrame
+ mainFrame: JFrame
- textMinArrival: JTextField
- textMaxArrival: JTextField
- textMinService: JTextField
- textMaxService: JTextField
- textNoOfQueues: JTextField
- textSimulation: JTextField
- buttonStart: JButton
- fields: JTextField[]
- textArea: JTextArea
- textEmpty: JTextField
- textTotal: JTextField
- textPeak: JTextField
+ SimulationFrame()
+ newFrame(): void
+ getNoOfQueues(): Integer
+ getMinServiceTime(): Integer
+ getMaxServiceTime(): Integer
+ getMinArrivalTime(): Integer
+ getMaxArrivalTime(): Integer
+ getSimulationTime(): Integer
+ addButtonListener(ActionListener):
+ setTextArea(String): void
+ setFields(String, int): void
+ setTextEmpty(String): void
+ setTextTotal(String): void
+ setTextPeak(String): void
Controller
- simulationFrame: SimulationFrame
# Controller(SimulationFrame)
addButtonListener
+ actionPerformed(ActionEvent): void

4. Implementation

Client (package model)

This class describes the object ‘Client’. Each client is identified by some particular attributes: id, arrival time, processing (service) time and finish time.

The ‘id’ is the unique feature of the clients (there will be no more than one client with a specific id).

The ‘arrival time’ is a variable that specifies the time the client arrives in the store one the simulation has started. The ‘processing time’ represents the time the client needs to be in the store (a service time). Finally, ‘finish time’ is a variable that specifies when the client leaves the store. This variable is calculated using the following formula: FinishTime = arrivalTime + processingPeriod+ waitingPeriodOnChosenServer.

Methods:

- setArrivalTime(arrivalTime): sets the arrival time of the client;
- getArrivalTime(): returns the arrival time of the client;
- setProcessingTime(processingTime): sets the processing (service) time of the client;
- getProcessingTime(): returns the processing time of the client;
- getId(): returns the id of the client;
- setFinishTime(finishTime): sets the finish time of the client;
- compareTo(client): overridden method from the interface ‘Comparable’ used in order to sort the collection of clients by a specific feature (in this case, by the arrival time).
- Client(arrivalTime, processingTime): creates the object of type Client having the properties received as parameters;

Queue (package model)

This class describes the object ‘Queue’. Each object ‘queue’ represents one of registers of the store in which we want to simulate the queue processing. Each queue has its own list of clients and some auxiliary variables that help the implementation: a current period and a waiting period (each of these modifies during the simulation).

Methods:

- Queue (id, currentTime, timeLimit, simulationFrame): creates the object of type Queue. It initializes the list of clients and the waiting period. Also, it sets its properties to the ones received as parameters.
- addClient(newClient): adds a new client to the list of clients; in this method, the waiting time of the queue is increased with a value of the processing time of the new client. Also, each modification is going to be shown in the user interface, so we print it from this method.
- removeClient(): removes the client from the list; the list is treated like a queue, so the head of the list is removed. Also, the waiting time is decreased with a value of the processing time of the client to be removed; this modification is printed to the logs.

- run(): method from the interface 'Runnable'; it controls the threads.
- getWaitingPeriod(): return the waiting period of the queue;
- getId(): returns the id of the queue;
- toString(): returns a string that contains all the clients of the queue; if no clients, the returned string is 'Empty'.
- getClients(): returns the list of the clients.

SimulationManager (package controller)

This class controls the execution of the application. It contains all the variables needed from the interface in order to make the simulation to start. Basically, this class models also the objects of Queue and Client.

Methods:

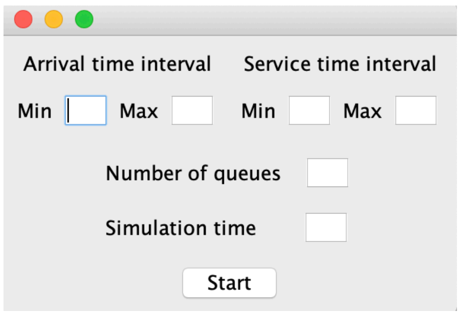
- SimulationManager(): constructor of the class; all the variables are initialized, and the objects are created and instantiated.
- generateNRandomClients(): this method generates a number of clients with respect to the simulation time.
- run(): from the interface 'Runnable', this method helps handling the threads.
- main(): a simulation frame is created which makes the application to start.

Controller (package controller)

This class implements the action listener designed for the button declared in the simulation frame. It has an inner class for the object needed to be used in the frame. The inner class has as method 'actionPerformed', method from the interface 'ActionListener'.

SimulationFrame (package view)

This class creates the graphical user interface.



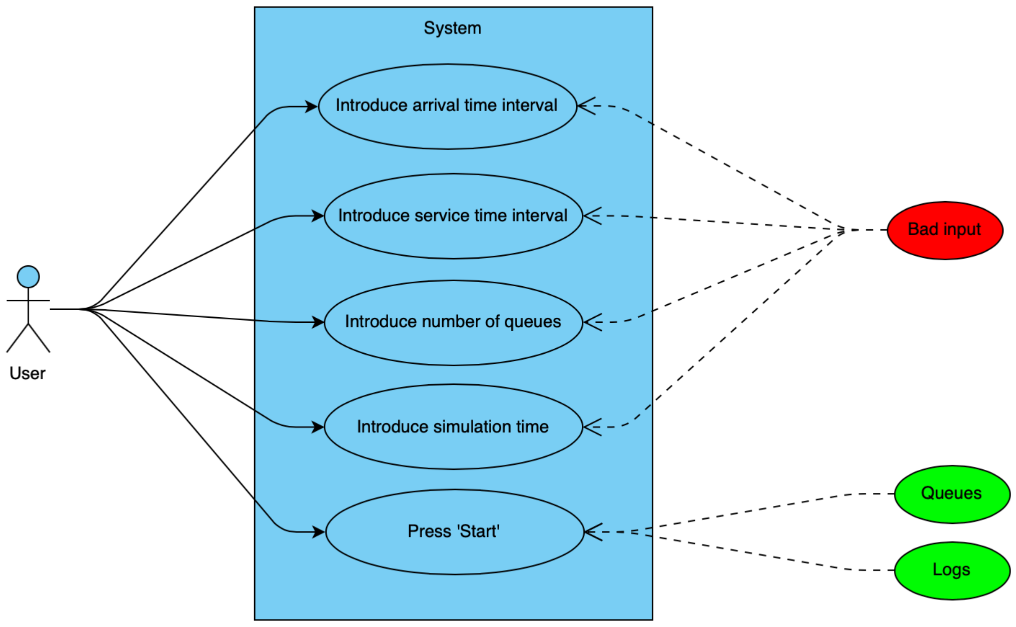
Each text field, label and the button needed in order to create this interface is created and instantiated in this class.

Methods:

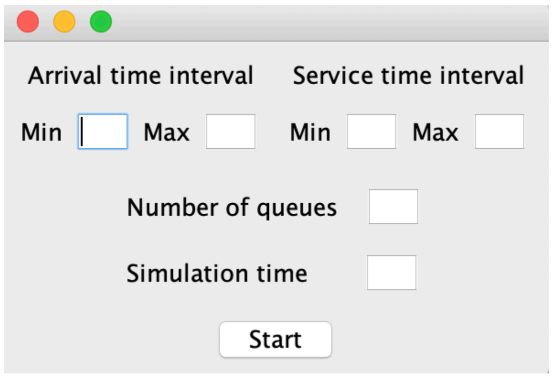
- SimulationFrame(): constructor of the class;

- newFrame(): method that opens a new frame once the button 'Start' is pressed.
- getNoOfQueues(): returns the number of queues specified by the user in the interface;
- getMinArrivalTime(): returns the minimum arrival time of a client specified by the user in the interface;
- getMaxArrivalTime(): returns the maximum arrival time of a client specified by the user in the interface;
- getMinServiceTime(): returns the minimum service time of a client specified by the user in the interface;
- getMaxServiceTime(): returns the maximum service time of a client specified by the user in the interface;
- getSimulationTime(): returns the simulation time;
- addButtonListener(actionEvent): sets the action listener to the 'Start' button.
- setTextArea(string): sets the text of the text area;
- setFields(string, i): sets the text of a text field where 'i' represents the position of the text to be introduced;
- setTextEmpty(string): sets the text of the average empty time;
- setTextTotal(string): sets the text of the average waiting time;
- setTextPeak(string): sets the text of the peak momentum.

5. User Interface (how to use the application)


System
Introduce arrival time interval
Introduce service time interval
Introduce number of queues
Introduce simulation time
Press 'Start'
User
Bad input
Queues
Logs

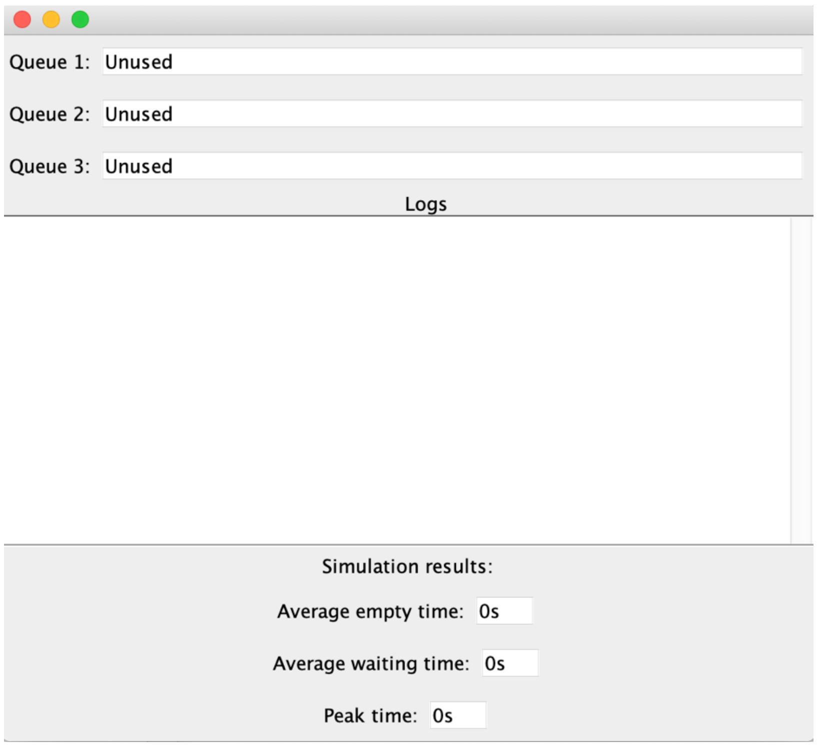
Use-Case Diagram



⇒ At a first glance, we observe that the graphical interface has some labels, some text fields and some buttons;

⇒ Attention: It is very important to introduce valid data in all the text boxes!

⇒ Once the user introduced valid data and the 'Start' button is pressed, the current frame closes and a new one is open as below:



Queue 1: Unused

Queue 2: Unused

Queue 3: Unused

Logs

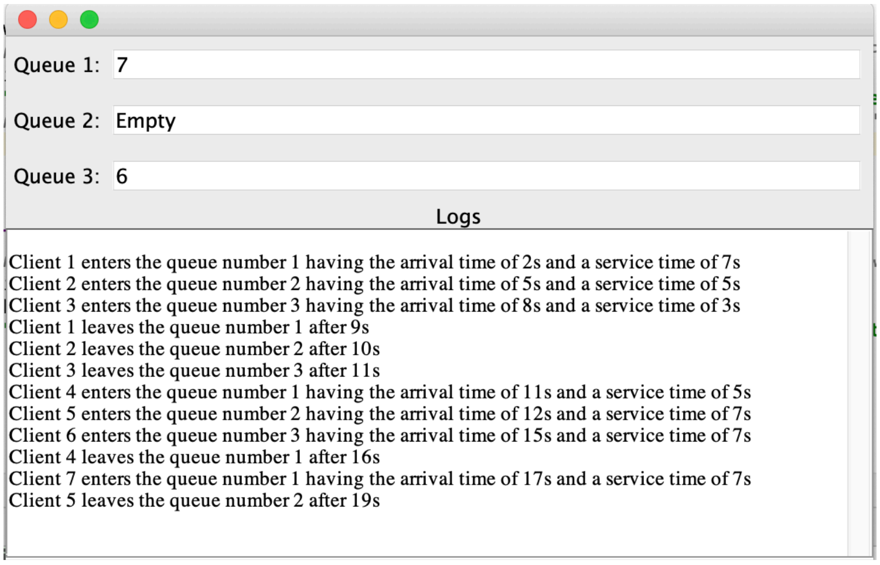
Simulation results:

Average empty time: 0s

Average waiting time: 0s

Peak time: 0s

⇒ This form opens and the application start to generate clients, to put them in queues and to process their orders. The example from above has 3 queues which wait for the clients to be generated and then proceeded by the registers (queues). Once the first client is generated, the application begins to proceed the costumers and the queues and logs will look like below.


Queue 1: 7

Queue 2: Empty

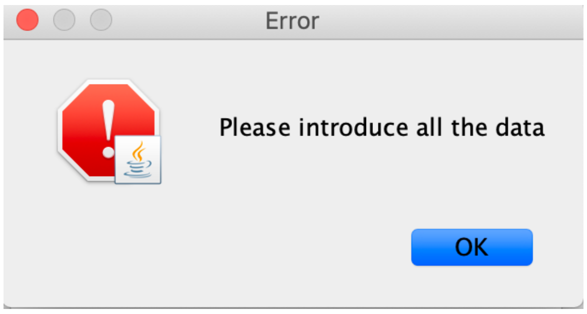
Queue 3: 6

Logs

```
Client 1 enters the queue number 1 having the arrival time of 2s and a service time of 7s
Client 2 enters the queue number 2 having the arrival time of 5s and a service time of 5s
Client 3 enters the queue number 3 having the arrival time of 8s and a service time of 3s
Client 1 leaves the queue number 1 after 9s
Client 2 leaves the queue number 2 after 10s
Client 3 leaves the queue number 3 after 11s
Client 4 enters the queue number 1 having the arrival time of 11s and a service time of 5s
Client 5 enters the queue number 2 having the arrival time of 12s and a service time of 7s
Client 6 enters the queue number 3 having the arrival time of 15s and a service time of 7s
Client 4 leaves the queue number 1 after 16s
Client 7 enters the queue number 1 having the arrival time of 17s and a service time of 7s
Client 5 leaves the queue number 2 after 19s
```

These are the logs and real queue evolution after a certain time of simulation.

What happens if the user introduces an invalid input?
This:


Error

Please introduce all the data

Also, some constant are shown in the second form: average waiting time per queues, average, average empty queue time for all of the queues and a peak time(when the queues are at the fullest capacity of the processing time).

6. Multithreading

In order to implement the application, it was needed to use the concept of multithreading. Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms: extending the Thread class or implementing the Runnable Interface. Particularly, I used the second method: implementing the Runnable interface. In order to use this, we create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

Example:

```java
public void run()
{
    while(true) {
        if (clients.size() > 0) {
            Client t = clients.element();
            try {
                Thread.sleep(t.getProcessingTime() * 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            removeClient();
            frame.setFields(this.toString(),this.id-1 );
        }
        else if(currentTime.intValue() > timeLimit) break;
    }
}
```

Also, in order to implement multiple threads, it is needed to synchronise the threads. This can be done using atomic variables.

The whole reason for atomic variables is for synchronization purposes during multi-threaded operations. Multi-threading is a lot like multi-tasking, parts of a Java program can be running at the same time, performing different updates or calculations. But what happens when two or more threads try to access the same variable? In that case, you can wind up with a lock on the data. That is, no updates can be made until the lock is released. Thread A might lock the counter variable while it's running, then release to Thread B.

Java provides a utility for keeping things in sync. The atomic variables ensure synchronization. With multi-threading, we need to have the ability to ensure that our threads don't crash into each other or cause issues with data changes.
Example: the waiting time for each queue is an AtomicInteger.

Also used for synchronization: BlockingQueue instead of a classic list or ArrayList. The main advantage is that a BlockingQueue provides a correct, thread-safe implementation. The "blocking" nature of the queue has a couple of advantages. First, on adding elements, if the queue capacity is limited, memory consumption is limited as well. Also, if the queue consumers get too far behind producers, the producers are naturally throttled since they have to wait to add elements. When taking elements from the queue, the main advantage is simplicity; waiting forever is trivial, and correctly waiting for a specified time-out is only a little more complicated.
Example: each queue has a list of clients declared as BlockingQueue.

7. Further developments

Some of the functionalities of this application can be extended further in order to be more complex and to give the user a real – life store simulation. For instance, we can develop the application so that it could have more strategies of distributing the clients (not only by the waiting time; the clients could be distributed at the shortest queue). Also, if a queue becomes empty and a new client is already in the store, it would be a nice implementation so that the client will swap the queue, getting to the empty one. Also, I think that an interesting improvement would be generating clients with a certain priority and when that client arrives to the store, him or she would go to the registers even if there are some people waiting in the line. This resembles the idea of having a reservation.

8. Conclusion

To conclude, creating this application developed my knowledge about java programming language and how to work with the graphical interface (java.swing). Also, it helped me understanding how exceptions work by making me doing research about how to use them and how to mould them in order to be useful for my project. I consider that this project was a little bit challenging because I had never worked with threads, but after doing research and starting to understand how they work, I realised that this application of simulating threads was not that complicated. I learned about atomic variables as well and how they make my job easier when implementing multithreading.

9. Bibliography

http://stackoverflow.com (learned about atomic variables and helped me to fix some bugs)
https://www.geeksforgeeks.org (learned about multithreading and how to work with the Runnable interface)
https://study.com/academy/lesson/atomic-variables-in-java-definition-example.html
https://www.techopedia.com/definition/16246/atomic-java