DOCUMENTATION

ASSIGNMENT 2

STUDENT NAME: Bălan Ionela-Loredana
GROUP: 30223

CONTENTS

1. Assignment Objective

1.1. Main Objective

The main objective of the project is to develop a queues management application in Java which assigns clients to queues such that the waiting time is minimized.

1.2. Sub-objectives

Sub-objectives	**Description**	**Chapter**
Analyze the problem and identify requirements.	Identifying use-cases, functional and non-functional requirements	2
Design the queues management application	Division into sub-systems/packages, classes, sub-classes, routines.	3
Implement the queues management application	Each class and important method implementation will be described.	4
Test the queues management application	Different inputs will be put to test in order to demonstrate the functionality.	5

2. Problem Analysis, Modeling, Scenarios, Use Cases

2.1. Requirements

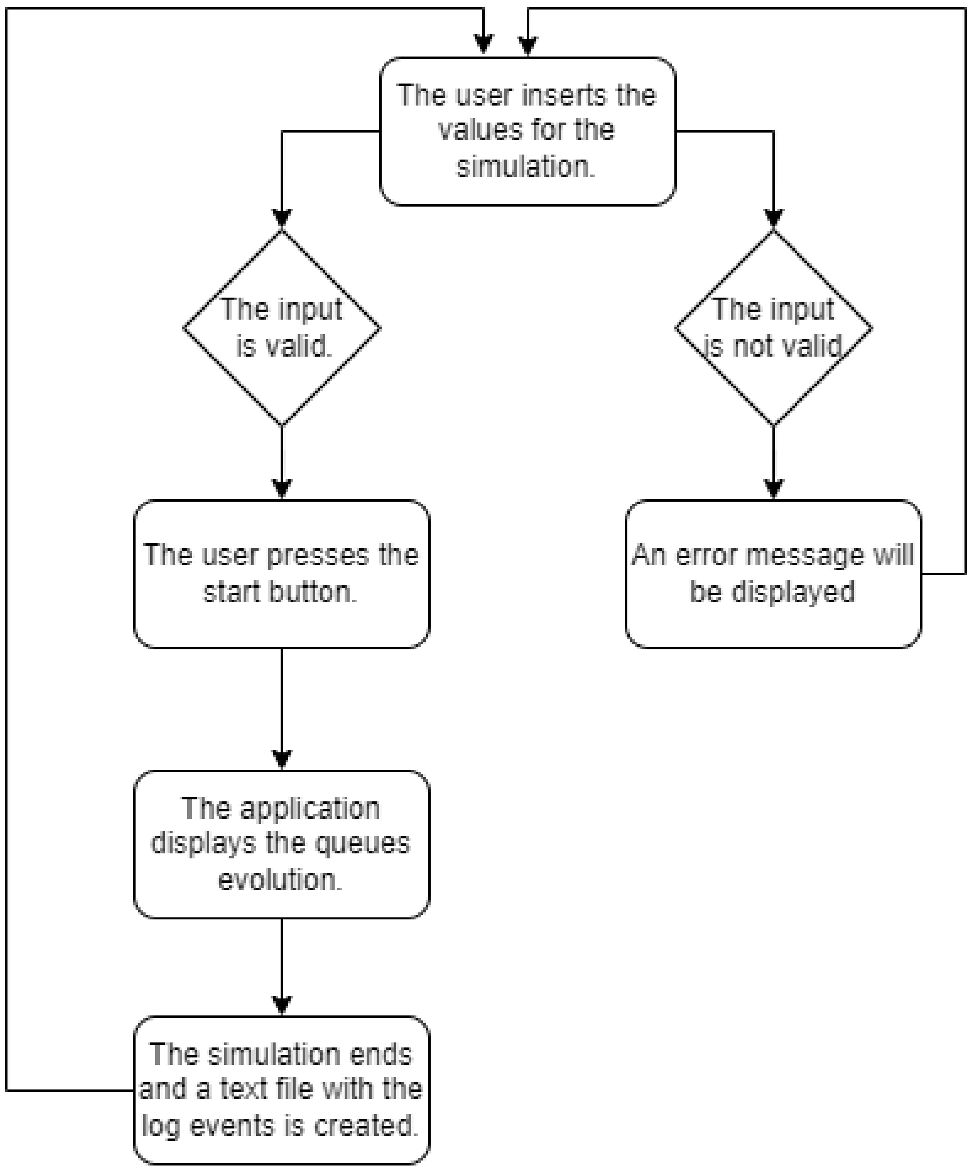
2.1.1. Functional requirements:

- The queues management app should allow users to setup the simulation.
- The queues management app should allow users to start the simulation.
- The queues management app should display the real-time queues evolution.
- The queues management app should allow users to save different simulation scenarios for future reference or comparison purposes.

2.1.2. Non-functional requirements:

- The queues management app should be intuitive and easy to use by the user.
- The simulation should be fast.
- The interface should provide error-handling messages for bad input.

2.2. ***Use-cases flowchart***


The user inserts the values for the simulation.
The input is valid.
The input is not valid.
The user presses the start button.
An error message will be displayed
The application displays the queues evolution.
The simulation ends and a text file with the log events is created.

2.3. Example scenario

1. Use Case: Setup simulation
Primary Actor: user
Main Success Scenario:

1. The user inserts the values for the: number of clients, number of queues, simulation interval, minimum and maximum arrival time, and minimum and maximum service time.
2. The user starts the application.
3. The queues management application displays the real-time queues evolution.
4. The simulation ends and a text file with the log events is created.

Alternative Sequence: Incorrect input
- The user inserts incorrect values for the simulation.
- The application displays a window with an error message.
- The scenario returns to the first step.

2.4. Modeling

CashRegister class - this class represents the abstraction of a cash register and serves as a fundamental entity in the system, it has a run method used for clients processing simulation.

Client class - this class includes relevant attributes of a client for the queues management application.

ClientGenerator class - this class contains a method for generating clients with a random ID, arrival time and service time, based on the data from the interface.

Scheduler – this class is used to create CashRegister objects, assign and start threads for them, and to assign clients to cash registers using a strategy based on the current needs in the simulation.

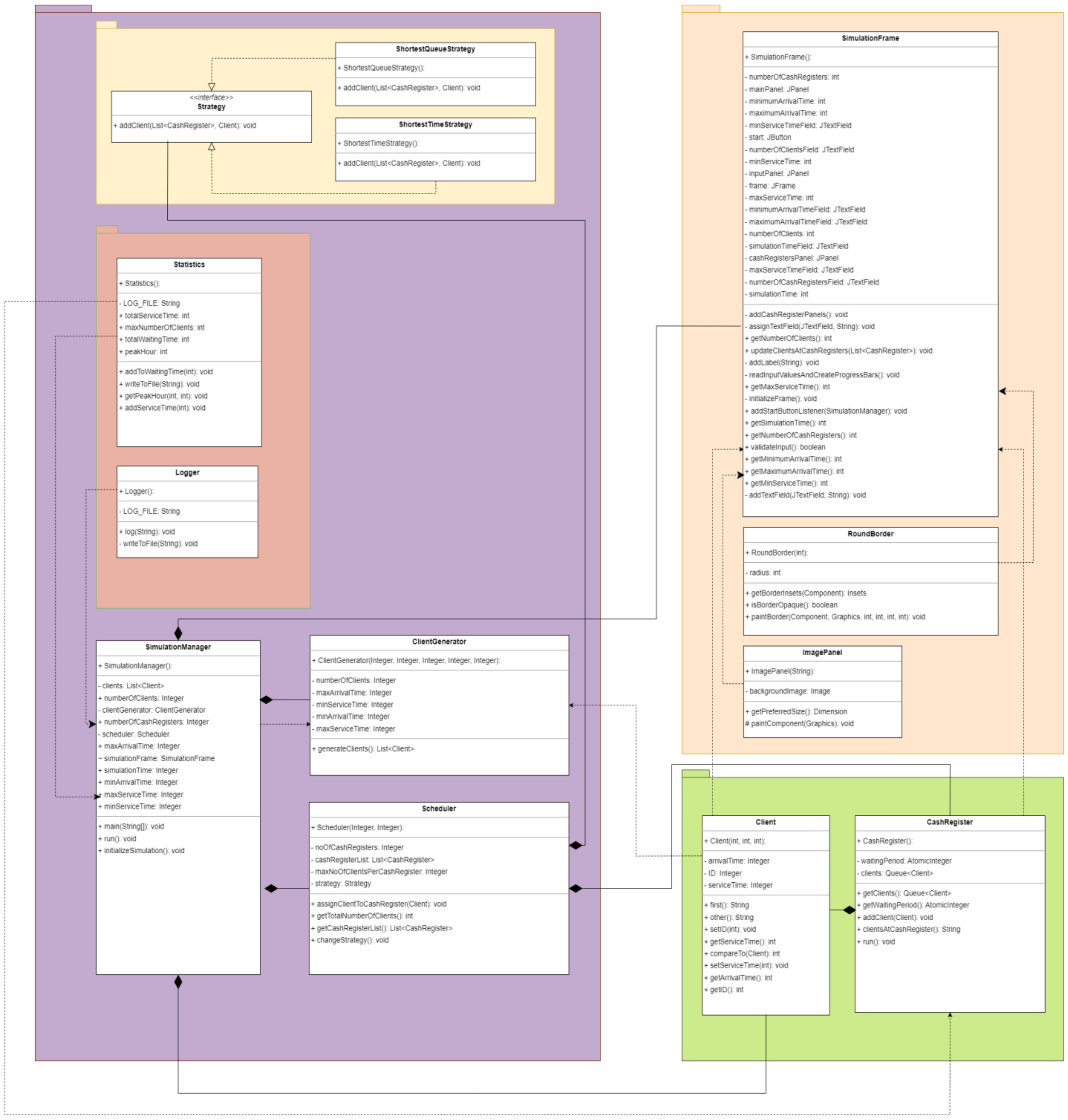
SimulationManager – this class is used to simulate the queues evolution by iterating the clients list and pick the ones that have the arrival time equal to the current time in the simulation.

Logger and Statistics classes – these classes are used to log the simulation events.

ShortestQueueStrategy and ShortestTimeStrategy classes – these classes are used by the Scheduler class in order to assign a client to a certain cash register based on the needed strategy.

3. Design

UML package and class diagrams


ShortestQueueStrategy
+ ShortestQueueStrategy():
+ addClient(List<CashRegister>, Client): void
<<interface>>
Strategy
+ addClient(List<CashRegister>, Client): void
ShortestTimeStrategy
+ ShortestTimeStrategy():
+ addClient(List<CashRegister>, Client): void
Statistics
+ Statistics():
- LOG_FILE: String
+ totalServiceTime: int
+ maxNumberOfClients: int
+ totalWaitingTime: int
+ peakHour: int
+ addToWaitingTime(int): void
+ writeToFile(String): void
+ getPeakHour(int, int): void
+ addServiceTime(int): void
Logger
+ Logger():
- LOG_FILE: String
+ log(String): void
- writeToFile(String): void
SimulationManager
+ SimulationManager():
- clients: List<Client>
+ numberOfClients: Integer
- clientGenerator: ClientGenerator
+ numberOfCashRegisters: Integer
- scheduler: Scheduler
+ maxArrivalTime: Integer
~ simulationFrame: SimulationFrame
+ simulationTime: Integer
+ minArrivalTime: Integer
+ maxServiceTime: Integer
+ minServiceTime: Integer
+ main(String[]): void
+ run(): void
+ initializeSimulation(): void
ClientGenerator
+ ClientGenerator(Integer, Integer, Integer, Integer, Integer):
- numberOfClients: Integer
- maxArrivalTime: Integer
- minServiceTime: Integer
- minArrivalTime: Integer
- maxServiceTime: Integer
+ generateClients(): List<Client>
Scheduler
+ Scheduler(Integer, Integer):
- noOfCashRegisters: Integer
- cashRegisterList: List<CashRegister>
- maxNoOfClientsPerCashRegister: Integer
- strategy: Strategy
+ assignClientToCashRegister(Client): void
+ getTotalNumberOfClients(): int
+ getCashRegisterList(): List<CashRegister>
+ changeStrategy(): void
SimulationFrame
+ SimulationFrame():
- numberOfCashRegisters: int
- mainPanel: JPanel
- minimumArrivalTime: int
- maximumArrivalTime: int
- minServiceTimeField: JTextField
- start: JButton
- numberOfClientsField: JTextField
- minServiceTime: int
- inputPanel: JPanel
- frame: JFrame
- maxServiceTime: int
- minimumArrivalTimeField: JTextField
- maximumArrivalTimeField: JTextField
- numberOfClients: int
- simulationTimeField: JTextField
- cashRegistersPanel: JPanel
- maxServiceTimeField: JTextField
- numberOfCashRegistersField: JTextField
- simulationTime: int
- addCashRegisterPanels(): void
- assignTextField(JTextField, String): void
+ getNumberOfClients(): int
+ updateClientsAtCashRegisters(List<CashRegister>): void
- addLabel(String): void
- readInputValuesAndCreateProgressBars(): void
+ getMaxServiceTime(): int
- initializeFrame(): void
+ addStartButtonListener(SimulationManager): void
+ getSimulationTime(): int
+ getNumberOfCashRegisters(): int
+ validateInput(): boolean
+ getMinimumArrivalTime(): int
+ getMaximumArrivalTime(): int
+ getMinServiceTime(): int
- addTextField(JTextField, String): void
RoundBorder
+ RoundBorder(int):
- radius: int
+ getBorderInsets(Component): Insets
+ isBorderOpaque(): boolean
+ paintBorder(Component, Graphics, int, int, int, int): void
ImagePanel
+ ImagePanel(String):
- backgroundImage: Image
+ getPreferredSize(): Dimension
# paintComponent(Graphics): void
Client
+ Client(int, int, int):
- arrivalTime: Integer
- ID: Integer
- serviceTime: Integer
+ first(): String
+ other(): String
+ setID(int): void
+ getServiceTime(): int
+ compareTo(Client): int
+ setServiceTime(int): void
+ getArrivalTime(): int
+ getID(): int
CashRegister
+ CashRegister():
- waitingPeriod: AtomicInteger
- clients: Queue<Client>
+ getClients(): Queue<Client>
+ getWaitingPeriod(): AtomicInteger
+ addClient(Client): void
+ clientsAtCashRegister(): String
+ run(): void

Used data structures

Lists

```
List<Client> clients = new ArrayList<>();
```

Queues, BlockingQueues

```
Queue<Client> clients = new LinkedBlockingQueue<>();
```

4. Implementation

1. Class “CashRegister”

- is a model class with a list of clients, and a waiting period of type AtomicInteger
- it implements Runnable and has a run method used for the clients processing simulation by decrementing their service time until becoming 1, after that removing them from the list in order to process the next clients.

```
@Override
    public void run() {
        while (true) {
            synchronized (this) {
                if (!clients.isEmpty()) {
                    try {
                        Client client = clients.peek();
                        Thread.sleep(1000);
                        while (client.getServiceTime() > 1) {
                            client.setServiceTime(client.getServiceTime() - 1);
                            waitingPeriod.decrementAndGet();
                            Thread.sleep(1000);
                        }
                        clients.poll();
                        waitingPeriod.decrementAndGet();
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
```

2. **Class ”Client”**

- is a model class that contains relevant attributes as ID, arrivalTime and serviceTime and getter/setter methods.

3. **Class “ClientGenerator”**

- this class contains a method used to generate clients with random attributes and sorts them by their arrival time, these clients are going to be used in the simulation.

```
  public List<Client> generateClients() {
        List<Client> clients = new ArrayList<>();

        for (int i = 0; i < numberOfClients; i++) {
            int id = i + 1;
            int arrival = ThreadLocalRandom.current().nextInt(minArrivalTime, 
maxArrivalTime + 1);
            int service = ThreadLocalRandom.current().nextInt(minServiceTime, 
maxServiceTime + 1);
            Client client = new Client(id, arrival, service);
            clients.add(client);
            Statistics.addServiceTime(client.getServiceTime());
        }
        Collections.sort(clients);
        return clients;
    }
```

4. **Class “Scheduler”**

- this class is used to create cash registers and threads for them, then using a strategy chosen by the changeStrategy method to assign clients to them in the assignClientToCashRegister method.

```
public void changeStrategy() {
        int shortestWaitingTime = Integer.MAX_VALUE;
        for (CashRegister cashRegister : cashRegisterList) {
            int currentWaitingTime = cashRegister.getWaitingPeriod().intValue();
            if (currentWaitingTime < shortestWaitingTime) {
                shortestWaitingTime = currentWaitingTime;
            }
        }
        if (shortestWaitingTime == Integer.MAX_VALUE) {
            strategy = new ShortestQueueStrategy();

        } else {
            strategy = new ShortestTimeStrategy();
        }
    }
```

```
public void assignClientToCashRegister(Client client) {
        if (strategy != null) {
            changeStrategy();
            strategy.addClient(cashRegisterList, client);
        }
    }
```

-it also has a method to get the total number of clients at all the cash registers called in the SimulationManager at every time of the simulation and used in the statistics class to determine the peak hour.

```
public int getTotalNumberOfClients(){
        int size = 0;
        for(CashRegister cashRegister : cashRegisterList){
            size += cashRegister.getClients().size();
        }
       return size;
    }
```

5. **Class "SimulationManager"**
   -this class is used to simulate the queues and clients activity.
   -it has a list of clients, a Scheduler object, a ClientGenerator object used for the initialization of the clients' list, a SimulationFrame object and some variables that hold the inputs from the interface.
   -it implements Runnable and has a run method where it calls the Scheduler's assignClientToCashRegister method for clients in the clients list where their arrivalTime is equal to the time of the simulation.
   -it also calls the Logger and Statistics methods for writing to file the events on every time of the simulation.
   -it is used to display real-time evolution of the simulation in the graphical interface by calling the SimulationFrame's updateClientsAtCashRegisters method.

```
    @Override
    public void run() {
        int currentTime = 0;
        while (currentTime <= simulationTime) {
            Logger.log("\nTime " + currentTime + "\nWaiting clients: ");
            for (Iterator<Client> iterator = clients.iterator(); iterator.hasNext(); ) {
                Client client = iterator.next();
                if (client.getArrivalTime() == currentTime) {
                    scheduler.assignClientToCashRegister(client);
                    iterator.remove();
                } else {
                        Logger.log("(" + client.getID() + "," + client.getArrivalTime() +
                        "," + client.getServiceTime() + "); ");
                }
            }
            for (CashRegister cashRegister : scheduler.getCashRegisterList()) {
                Logger.log("\nQueue " + ": ");
                if (cashRegister.getClients().isEmpty()) {
                    Logger.log("closed");
                } else {
                    for (Client client : cashRegister.getClients()) {
                        Logger.log("(" + client.getID() + "," + client.getArrivalTime() +
                        "," + client.getServiceTime() + "); ");
                    }
                }
            }
```

```
simulationFrame.updateClientsAtCashRegisters(scheduler.getCashRegisterList());

            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            Statistics.getPeakHour(scheduler.getTotalNumberOfClients(), currentTime);
            currentTime++;
        }
       Statistics.writeToFile(String.format("Average waiting
       time%.2f",(Statistics.totalWaitingTime/(double)numberOfClients)));

       Statistics.writeToFile(String.format("Average service ]
       time%.2f",(Statistics.totalServiceTime/(double)numberOfClients)));

      Statistics.writeToFile("Peak hour: "+ Statistics.peakHour);

      System.out.println("\nEnded");

    }
```

6. **Interface “Strategy”**
   -this interface is used in the Scheduler class and has an addClient method that is overridden in the ShortestQueueStrategy and ShortestTimeStrategy classes.

7. **Class “ShortestQueueStrategy”**
   -it implements the Strategy interface
   -this class chooses the queue that has the minimum number of clients and adds the client given as parameter to it.
   -it is used in the Scheduler class in the assignClientToCashRegister method.

```
    @Override
     public void addClient(List<CashRegister> cashRegisterList, Client client) {
         CashRegister minCashRegisterQueue = cashRegisterList.get(0);
         for (CashRegister cashReg : cashRegisterList) {
             if (cashReg.getClients().size() < minCashRegisterQueue.getClients().size()) {
                 minCashRegisterQueue = cashReg;
             }
         }
         minCashRegisterQueue.addClient(client);
     }
```

8. **Class “ShortestQueueStrategy”**
   -it implements the Strategy interface
   - this class chooses the queue that has the minimum waiting period and adds the client given as parameter to it.
   -it is used in the Scheduler class in the assignClientToCashRegister method.

```
   @Override
     public void addClient(List<CashRegister> cashRegisterList, Client client) {
         CashRegister minTimeCashRegister = cashRegisterList.get(0);
         for (CashRegister cashReg : cashRegisterList) {
             if (cashReg.getWaitingPeriod().intValue() <
minTimeCashRegister.getWaitingPeriod().intValue()) {
                 minTimeCashRegister = cashReg;
             }
         }
         minTimeCashRegister.addClient(client);
     }
```

9. Class “Logger”

-this class is used to write in a text file the events of the simulation sent as a string from the SimulationManager class and has a LOG_FILE string that represent the name of the file where the data is going to be written

```
    public  static void log(String message) {
        System.out.print(message);
        writeToFile(message);
    }

    private static void writeToFile(String message) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(LOG_FILE, true)))
{
            writer.write(message);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

10. Class “Statistics”

-this is use to write in a text file the average waiting time, average service time and the peak hour in the simulation.

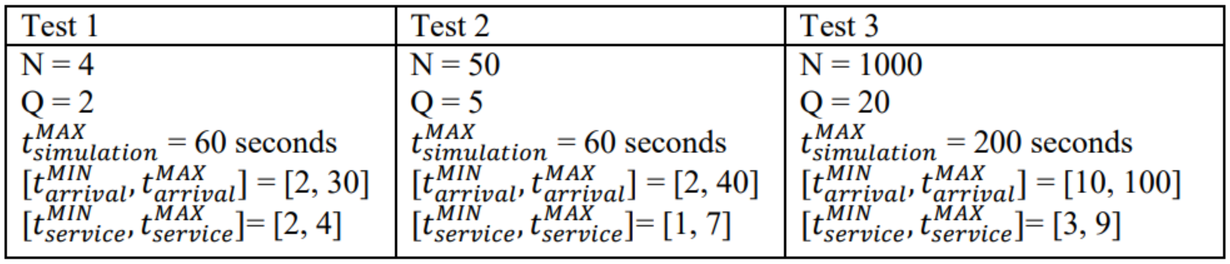
```
public static void addToWaitingTime(int waitingTimePerCashRegister){
        totalWaitingTime += waitingTimePerCashRegister;
    }

    public static void addServiceTime(int serviceTimePerClient){
        totalServiceTime += serviceTimePerClient;
    }

    public static void getPeakHour(int totalNumberOfClients, int time){
        if(totalNumberOfClients > maxNumberOfClients){
            maxNumberOfClients = totalNumberOfClients;
            peakHour = time;
        }
    }
    public static void writeToFile(String averageWaitingTime) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(LOG_FILE, true)))
{
            writer.newLine();
            writer.write(String.valueOf(averageWaitingTime));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

5. Results

For the application testing there were used the input data sets from the table below:



Test 1	Test 2	Test 3
N = 4 Q = 2 $t_{simulation}^{MAX}$ = 60 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}]$ = [2, 30] $[t_{service}^{MIN}, t_{service}^{MAX}]$= [2, 4]	N = 50 Q = 5 $t_{simulation}^{MAX}$ = 60 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}]$ = [2, 40] $[t_{service}^{MIN}, t_{service}^{MAX}]$= [1, 7]	N = 1000 Q = 20 $t_{simulation}^{MAX}$ = 200 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}]$ = [10, 100] $[t_{service}^{MIN}, t_{service}^{MAX}]$= [3, 9]

-the results are provided in the attached files.

6. Conclusion

In conclusion, the queues management project has been successfully developed to provide users a reliable tool for simulating a system of waiting queues. I have learned to apply strategy patterns, use Threads, how to write in text files in Java. Future development and upgrades might include a restart button for the simulation, a pause button, a clock to display the simulation time at each step.

7. Bibliography

1. *https://www.geeksforgeeks.org/arrayblockingqueue-peek-method-in-java/*
2. *https://www.baeldung.com/java-queue-linkedblocking-concurrentlinked*
3. *https://dsrl.eu/courses/pt/materials/PT_2024_A2_S1.pdf*
4. *https://dsrl.eu/courses/pt/materials/PT_2024_A2_S2.pdf*
5. *https://www.geeksforgeeks.org/how-to-create-a-thread-safe-queue-in-java/*
6. *https://jenkov.com/tutorials/java-util-concurrent/atomicinteger.html*
7. *https://medium.com/@greekykhs/all-about-blockingqueue-cb57350c9793*
8. *https://www.w3schools.com/java/java_threads.asp*
9. *https://www.youtube.com/watch?v=d3xb1Nj88pw*