

MINISTRY OF EDUCATION AND RESEARCH



**TECHNICAL UNIVERSITY**  
OF CLUJ-NAPOCA, ROMANIA

# **PROGRAMMING TECHNIQUES**

## **ASSIGNMENT 2**

### **QUEUES SIMULATOR**

**Reckerth Daniel-Peter**

**30444/2**

## Table of Contents

I.	Assignment Objective.....	3
II.	The Problem Analysis, Modelling, Scenarios, Use Cases.....	3
1.	Problem Analysis.....	3
2.	Modelling the Problem .....	3
3.	Use Case Scenarios .....	4
III.	Design.....	5
1.	Design Decisions .....	5
2.	UML Diagrams .....	6
2.1	Class Diagram .....	6
2.2	Package Diagram .....	7
3.	Data Structures.....	7
4.	Designing of Classes.....	7
5.	Algorithms .....	8
6.	Interface.....	9
IV.	Implementation.....	10
1.	Model.....	10
1.1	Task.....	10
1.2	Server .....	10
2.	Business.....	10
2.1	Scheduler .....	10
2.2	SimulationManager.....	10
2.3	Strategy – interface .....	11
2.4	ConcreteShortestQueueStrategy implements Strategy and the addTaskToServers() method.....	11
2.5	ConcreteShortestTimeStrategy implements Strategy and the addTaskToServers() method.....	11
3.	Controller.....	11
5.1.	TaskUtils .....	12
5.2.	SimulationInformation .....	12
5.3.	SimulationResults .....	12
V.	Results .....	13
VI.	Conclusions .....	14
VII.	Bibliography .....	14

## I. Assignment Objective

The second assignment is based on designing and implementing an application which aims to simulate and analyse queue-based systems for the final purpose of determining and minimizing client's waiting time.

Below are listed the secondary objectives which will help us in accomplishing the realisation of the system:

- analysis of the problem and identification of the requirements: determine the use-cases, modelling the problem
- designing the queue-based application: tasks (clients) generation, task dispatching, processing, waiting, servicing
- implementation: forward-engineering (based on the diagrams), thread creation, thread safety, GUI development

## II. The Problem Analysis, Modelling, Scenarios, Use Cases

### 1. Problem Analysis

The real-life problem which arises is how to efficiently manage some task's waiting time. As an analogy, we can suppose that a real client which goes shopping, and which needs a place to wait before being processed by the cashier. So, in our case the queue is used to model this real-world domain. It provides a place where a client can wait before being serviced. We wish to minimize however the time amount the client is waiting in the queue before being served.

Queue work based on the FIFO (First-In First-Out) principle. Thus, new clients will be added (enqueued) at the tail of the list while the first client which was added will be (dequeued) at the front. Such a queue simulates a waiting line for our clients (tasks) which wait to be processed.

### 2. Modelling the Problem

When we talk about modelling, we talk about high-level abstractions, devoid of the complexity and low-level details.

In order to achieve that we must define which are the clients (tasks), the queues, and the dynamics which happen (arriving, waiting, being serviced). The application simulates a series of  $N$  clients which arrive for service, which enter  $Q$  queues, which wait there in line, then they are being served and finally leaving the queues. Thus, the application's input data which will be entered by the user in the application's graphical user interface is defined as follows:

- $N$ : number of clients (tasks)
- $Q$ : number of queues (servers)
- $t_{simulation}^{MAMM}$ : maximum simulation timeout
- $t_{arrival}^{MIN} \leq t_{arrival} \leq t_{arrival}^{MAMM}$ : minimum and maximum arrival time
- $t_{service}^{MIN} \leq t_{service} \leq t_{service}^{MAMM}$ : minimum and maximum service time

A client or task is generated randomly from the data above and is characterized by the following three parameters (a tuple).

- ID: a number in the range  $[1, N]$
- $t_{arrival}$ : simulation time when they are ready to enter the queue (when they finished shopping)
- $t_{service}$ : time duration the client must be served (waiting time when he is in the front of the queue)

Each client is added to the queue with the minimum waiting time when its arrival time is greater than or equal to the simulation time.

Additionally, we have defined two strategies for the clients to enter the queue, using the Strategy design pattern. They are as follows:

- shortest time of waiting: described by the total waiting period of the clients in sequence (the least amount of waiting time)
- shortest queue size: number of clients in a queue (the smallest number)

The simulation ends if the simulation time has reached the time out limit (the shop closed) or if there are no more waiting clients and processed ones in the queue (they finished shopping).

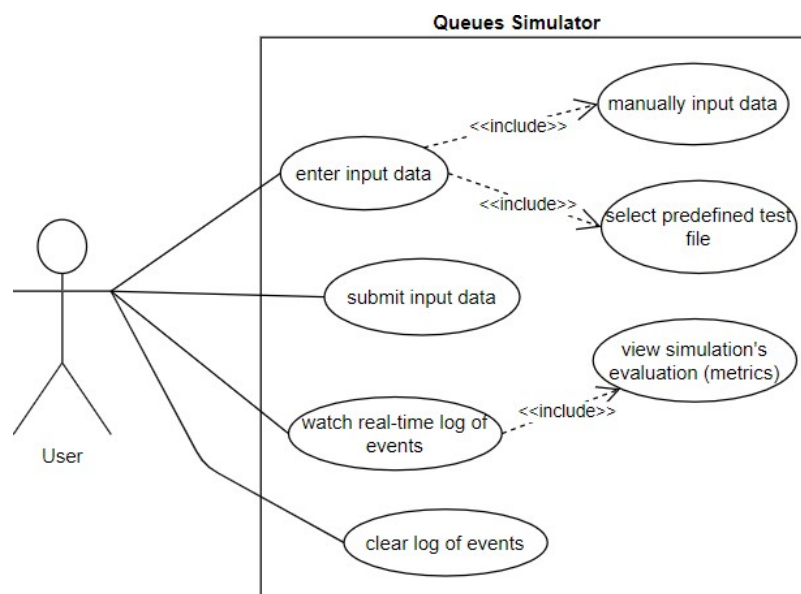
### 3. Use Case Scenarios

Scenarios represent different actions that can be taken by the user of the application. Together with a use case, they represent a list of actions or event steps typically defining the interactions between a role (known in UML as an actor) and the system in order to achieve a goal. [Wikipedia]

Therefore, a use case is an important and indispensable requirement analysis technique. Some actions the user can take which we talked before are:

- choose the input data (checkbox – manually enter data or choose a predefined input test file)
- enter input data (number of clients, queues, maximum simulation time, etc.)
- enter the output file name
- submit the input data
- clear the real-time log of events

Below the use case of our system is presented:

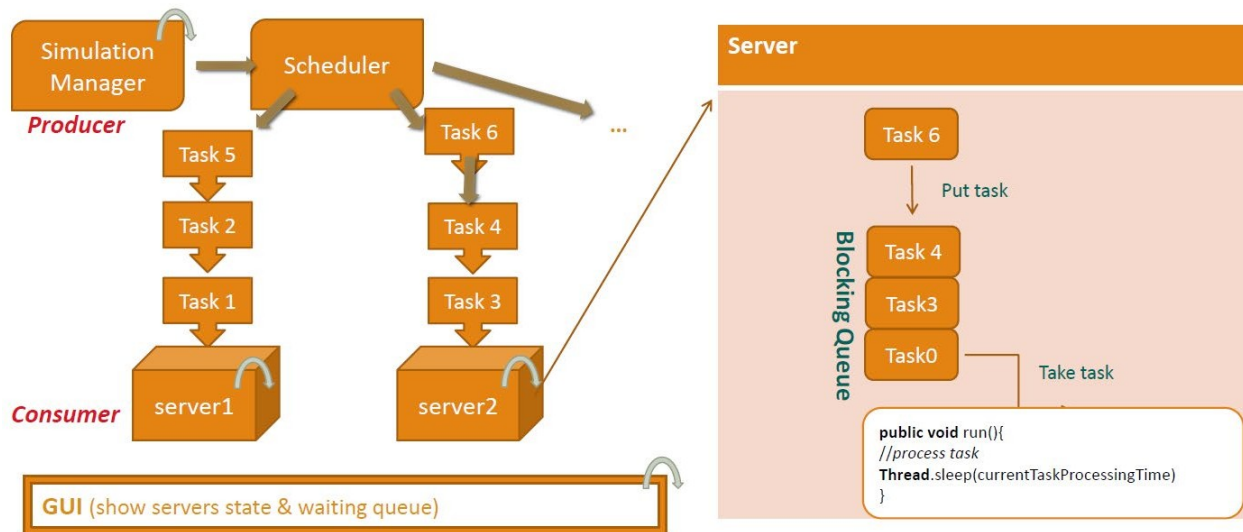


### III. Design

#### 1. Design Decisions

In order to work more organized, we have structured the application into packages. Although not specified, because we need a GUI for the user's interaction with the system, we have taken the decision to also use the MVC architectural pattern.

The model represents the main classes that needed to be implemented. It deals with many of the second secondary objective mentioned before, because we have specified here how a task and server should be modeled. Inbuilding the classes and the model per se we have followed the diagram presented in the support presentation.



[PT\_Assignment2\_Support\_Presentation – UTCN DSRL]

The model contains the two main abstractions that our application needs to use, namely **Server** and **Task**.

In the view we actually have our GUI created by using Swing. Its responsibility is to specify what the user's see. The view calls the appropriate listener method on the controller when an action is fired. The main class here is the **SimulationFrame**.

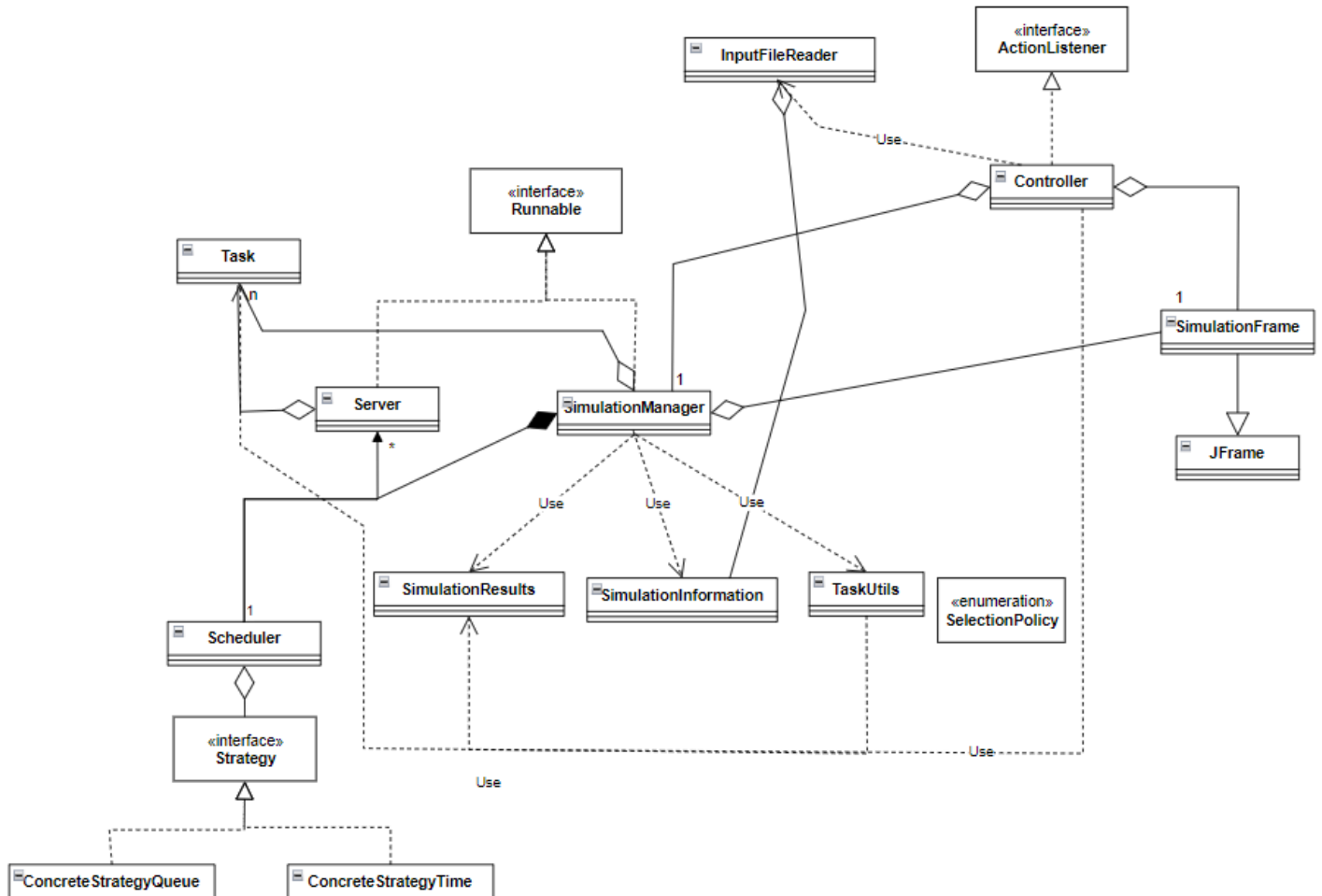
In the controller part we bind the view with the model, to facilitate communication between the two. It accesses both the model and the view and updates the notification for both. The main class here is the **Controller**.

Regarding the business logic we do this in a separate package in which we have both the strategies, shortest queue and shortest time and also the scheduler and the simulation manager.

## 2. UML Diagrams

The use case diagram was presented before. In this section we will present the class and package diagrams.

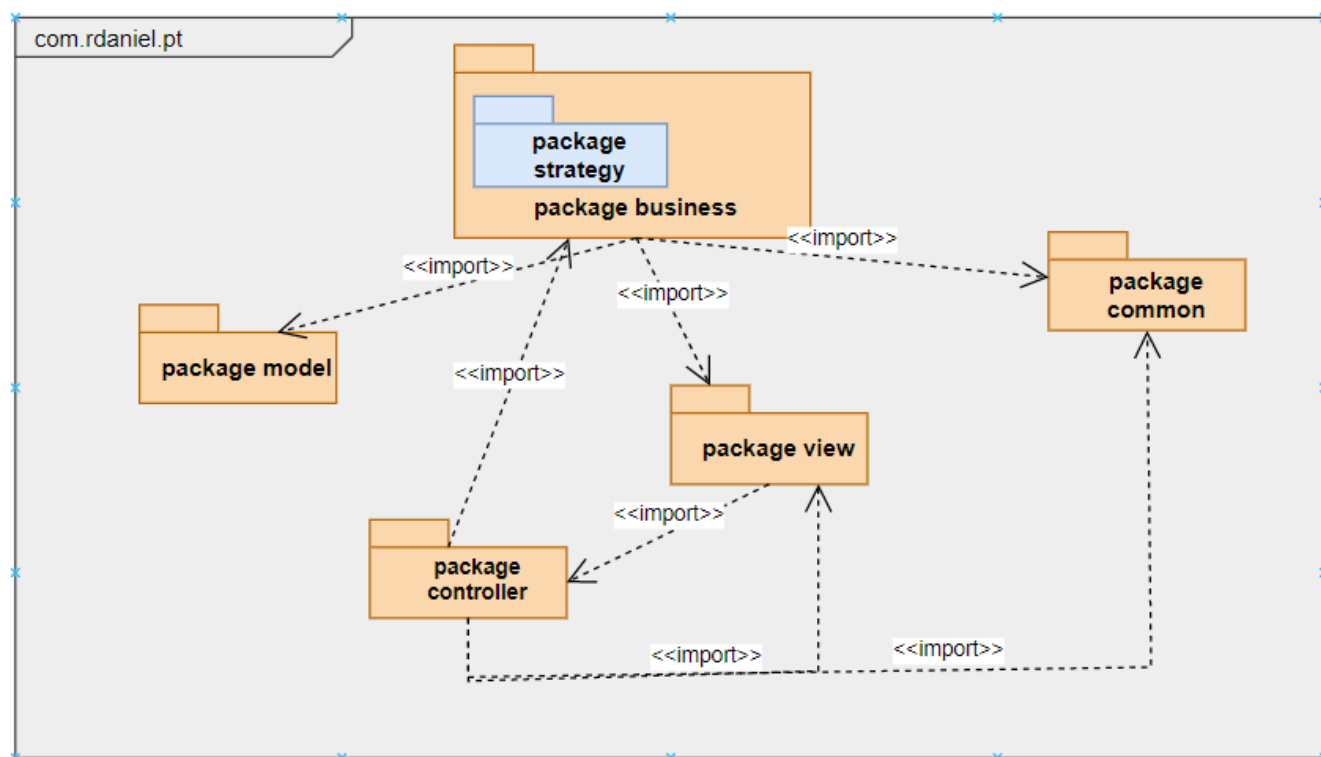
### 2.1 Class Diagram



This diagram represents the whole class diagram of our system. The application is run from the **Main** class. The controller implements all the action listeners for our GUI and contains the **SimulationFrame** and uses the **SimulationManager** classes, depicted by a composition relationship. The **SimulationFrame** is the view part of our project extending the Java **JFrame** class.

## 2.2 Package Diagram

Our system is divided in four main packages: model, view, controller, business and one common package.



## 3. Data Structures

Regarding the data structures used, we have both used already-implemented Java ones and also we have created our own. The used one from the Java library are: `List`, namely `ArrayList` for the servers, `BlockingQueue`, more specifically a `LinkedBlockingQueue` in which we place our tasks and which is a special queue in which operations wait for it to become non-empty when retrieving an element, and a `CopyOnWriteArrayList` used in the `SimulationManager` class, in which we hold the generated task. This kind of array list is a thread-safe variant of `ArrayList` which makes a fresh copy of the array each time a mutative operation is performed.

The `Server` class represents actually a queue-based data structure in which tasks arrive and are enqueued and dequeued.

## 4. Designing of Classes

When designing the classes we have followed the normal process, i.e. we have our class fields, constructors, and methods, which include setters and getters.

In the model part we have defined the `Task` class which models a task or a client arriving in our "shop". It has an ID, an arrival time, service time and waiting time. All of those are integers. Besides the constructor, the usual setters and getter we have included an overridden to string method and a method which decrements the service time.

Regarding our strategy for inserting tasks in the queues, we have used the strategy pattern in which we choose the policy in order to distribute clients. The context class is represented by the `Scheduler`. We have defined a functional interface called `Strategy` which defines only a method, namely the one to add a task. In our enumeration class we specified the two possible policies: shortest queue size or shortest waiting time on the queue. We then have two concrete classes which implement the two policies and the interface. In the context, namely the scheduler we have our servers list and the specified strategy. In the constructor we are initializing the servers and for the number of servers received as an argument we are also creating and starting a thread by providing a runnable object, and that is the instance of the `Server` class which implements the runnable interface. Also, in the `Scheduler` class we are dispatching the task in the servers by calling the specified strategy adding task method. We can also change the policy, returning the servers and printing them by providing an overridden to string method.

The `Server` class implements the `Runnable` interface and represents our consumer. It defines a `BlockingQueue` of tasks and different atomic data types like the waiting period on server and a boolean for the running of the thread. The `addTask()` method is called in the concrete strategy classes when we add a task to our server. We have also defined a `stop()` function which sets the atomic boolean flag to false. We have overridden the `run()` method of the `Runnable` interface in which we specify the thread's execution. In a while loop of the flag, we are working with the blocking queue. We retrieve (without removing) the first client in the queue, we decrement its service time and also, we decrement the waiting period and if the service time is zero, then that client was processed and is removed from the queue. All of these actions happen if the tasks queue is not empty. After we process the task or if the queue was empty the thread sleeps for 1000 ms.

In the `SimulationManager` class we are also implementing the `Runnable` interface, needed by the main thread. Here we have many fields which are the inputs to be read from the GUI. Additionally, we have defined a `FileWriter` object for writing the log of events in an output .txt file. The constructor initializes all of the fields, creates a new file writer object, a new scheduler and calls the `generateNRandomTask()` method from the `TaskUtils` which will be presented in the next chapter. The `run` method of this class is used to specify the thread's execution.

The `SimulationFrame` is our view (a picture of it will be provided shortly). It is simple and does not include the action listeners logic.

Our `Controller` class which is also the one in which the main method is found, implements the `ActionListener` interface and binds the GUI with the model.

## 5. Algorithms

We do not have any special algorithms which were used however, we care to mention some interesting method in which we have used some kind of algorithms.

In the `TaskUtils` class, we have a method called `generateNRandomTasks()` which actually generates the clients for our application randomly. So inside a for from  $[1, N]$  ( $N$  given) we are creating a `Random` object. The service time and arrival time are generated with the help of the `nextInt()` function with the following bound:  $t_{arrival/service}^{MAX} - t_{arrival/service}^{MIN} + 1$ , to which we add  $t_{arrival/service}^{MIN}$ . For example:

- service time range:  $[2, 4] \Rightarrow \text{nextInt}(4-2+1) + 2 \rightarrow \text{nextInt}(3)$ : random from 0 to 2, to which we add 2  $\Rightarrow$  random from 2, 4

We also compute the average service time here by the following formula:  $t_{service_{avg}} = \frac{\sum^N t_{service}}{N}$ . We then also sort list by the arrival time. The code is presented below.

```
public static List<Task> generateRandomTasks(SimulationInformation info) {
    Random random = new Random();
    List<Task> tasks = new CopyOnWriteArrayList<>();
    for(int i = 1; i <= info.getNumberOfTasks(); i++) {
        int serviceTime = random.nextInt( bound: info.getMaxServiceTime() -
            info.getMinServiceTime() + 1) + info.getMinServiceTime();

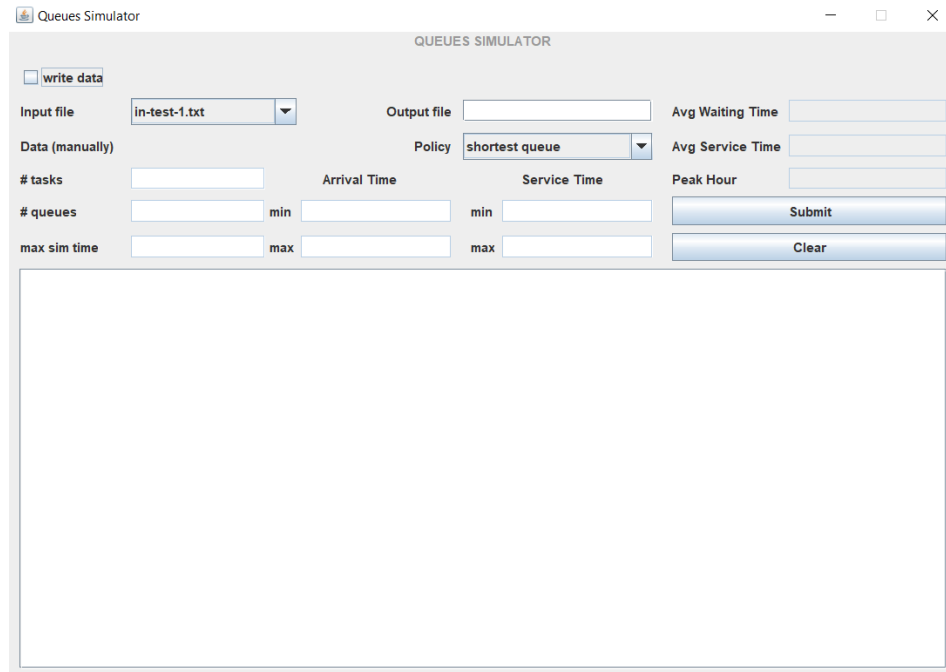
        int arrivalTime = random.nextInt( bound: info.getMaxArrivalTime()
            - info.getMinArrivalTime() + 1) + info.getMinArrivalTime();
        tasks.add(new Task(i, arrivalTime, serviceTime));
    }
    tasks.sort(Comparator.comparingInt(Task::getArrivalTime));
    return tasks;
}
```

The peak hour, and the average waiting time are computed inside the `run` method of the simulation manager's class. The peak hour represents the simulation time in which the waiting clients in the queues is the maximum. The average waiting time is actually the waiting time of the task in the queue and also the waiting time from the moment it is allocated to a queue until it is removed from it (the service time actually). Then it is summed up and divided by the client's number. This can also be achieved by retrieving from each server (queue) the size of it for each simulation time and then dividing all this sum by the number of clients.



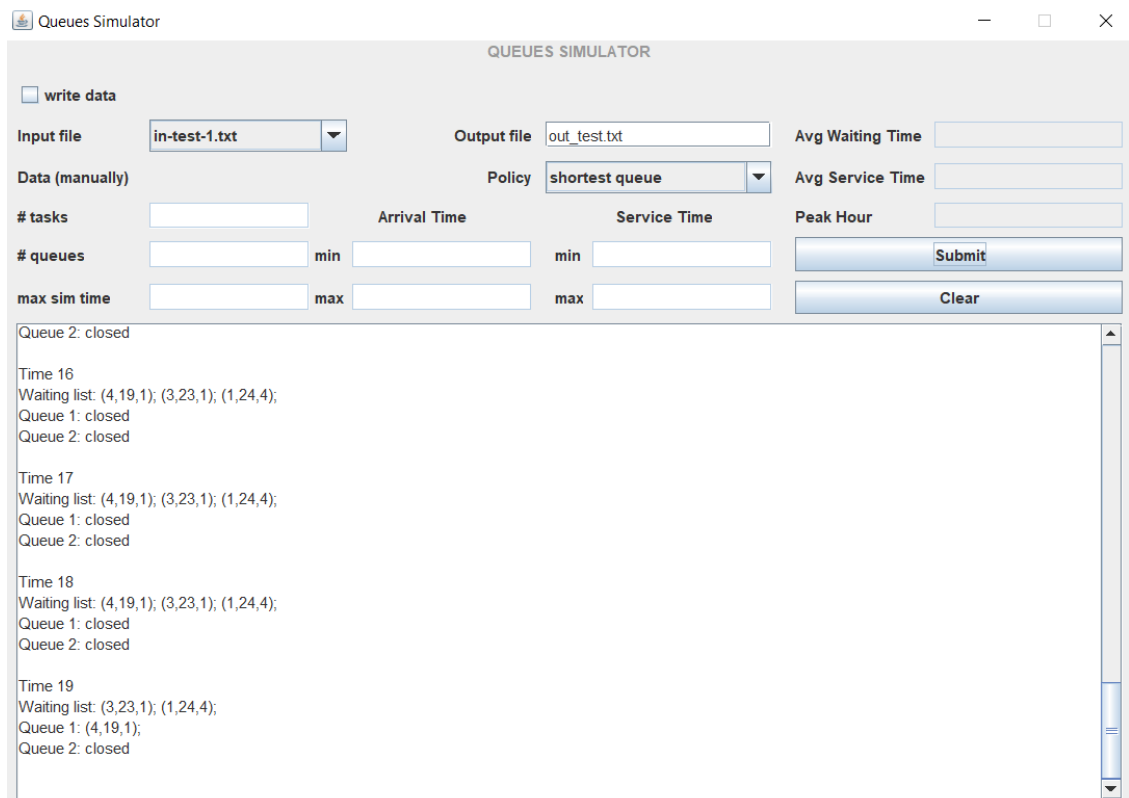
## 6. Interface

Below a picture of our GUI is shown.



The user has two choices: either to enter the data manually or choose a predefined data input file from a combo box. We have defined four input files (corresponding to the three test cases given in the requirements and a custom one). If the check box is not selected all the manually fields are disabled, if it is selected, they are enabled and the input file is disabled. However, for both of these options an output file name must be specified in order to generate the log of events (.txt file). In the right part, there are non-editable text boxes in which we display the system's evaluation. We have two buttons: clear log, which deletes the real-time text displayed in the white scrollable text area. It is important to mention that we have not implemented any input validation so we expect correct input from the user. Also we have a combo box allowing the user to choose the selection policy, between shortest queue and shortest time.

This picture is from executing the first predefined test case.



## IV. Implementation

We will present the implementation of our classes with their fields and most important methods described. We will divide this section by the packages.

### 1. Model

#### 1.1 Task

Fields:

- **ID, arrivalTime, serviceTime, waitingTime: int**

Main methods:

- constructor
- **decrementServiceTime()**: decrements by 1 the task's service time
- **toString()**: overridden method; returns the task as a string of the form "(ID, arrival time, service time)"



Task		
field	ID	int
field	arrivalTime	int
field	serviceTime	int
field	waitingTime	int
method	Task(int, int, int)	
method	getArrivalTime()	int
method	getServiceTime()	int
method	getWaitingTime()	int
method	setWaitingTime(int)	void
method	decrementServiceTime()	void
method	toString()	String

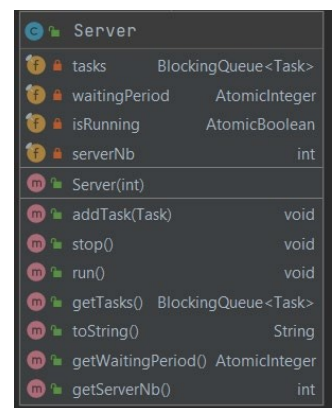
#### 1.2 Server

Fields:

- **tasks: BlockingQueue<Task>** - internal queue used for enqueueing and dequeueing clients
- **waitingPeriod: AtomicInteger** – decremented when the task is processed
- **isRunning: AtomicBoolean** – used as the while's loop condition
- **serverNb: int** – used for printing

Main methods:

- **addTask(Task)** – add a task in the blocking queue, set its waiting time to the sum of the waiting period and its service time; set the waiting period to its waiting time.
- **stop()** – set the **isRunning** flag to false
- **run()** – while **isRunning** is true, if the tasks queue is empty we retrieve without removing the first tasks; we decrement its service time, the waiting period and if the service time is zero we remove it from the queue, after these operations or if the tasks queue is empty we put the thread to sleep for 1000ms.



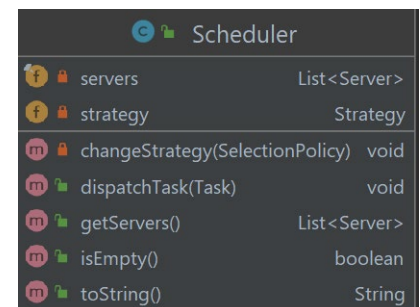
Server		
field	tasks	BlockingQueue<Task>
field	waitingPeriod	AtomicInteger
field	isRunning	AtomicBoolean
field	serverNb	int
method	Server(int)	
method	addTask(Task)	void
method	stop()	void
method	run()	void
method	getTasks()	BlockingQueue<Task>
method	toString()	String
method	getWaitingPeriod()	AtomicInteger
method	getServerNb()	int

### 2. Business

#### 2.1 Scheduler

Fields: servers, strategy

Main methods: dispatchTask(), changeStrategy(), isEmpty()



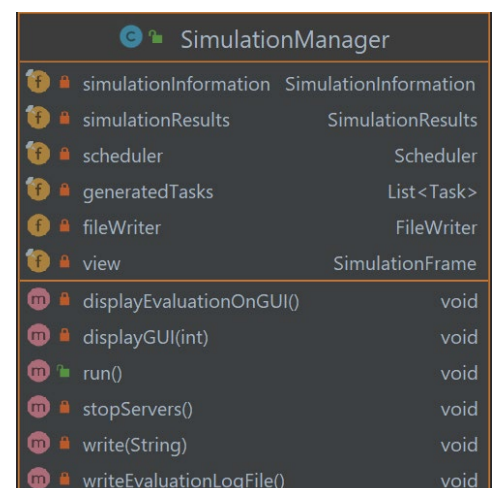
Scheduler		
field	servers	List<Server>
field	strategy	Strategy
method	changeStrategy(SelectionPolicy)	void
method	dispatchTask(Task)	void
method	getServers()	List<Server>
method	isEmpty()	boolean
method	toString()	String

#### 2.2 SimulationManager

Fields: simulationInformation, simulationResults, scheduler, generatedTasks, fileWriter, view

Main methods:

- displayEvaluationOnGUI()
- displayGUI()
- **run()** – main thread where we also compute the peak hour and the average waiting time
- stopServers()
- writeEvaluationLogFile()
- write()



SimulationManager		
field	simulationInformation	SimulationInformation
field	simulationResults	SimulationResults
field	scheduler	Scheduler
field	generatedTasks	List<Task>
field	fileWriter	FileWriter
field	view	SimulationFrame
method	displayEvaluationOnGUI()	void
method	displayGUI(int)	void
method	run()	void
method	stopServers()	void
method	write(String)	void
method	writeEvaluationLogFile()	void

## 2.3 Strategy – interface

## 2.4 ConcreteShortestQueueStrategy implements Strategy and the addTaskToServers() method

```
public class ConcreteShortestQueueStrategy implements Strategy {  
  
    @Override  
    public void addTaskToServers(List<Server> servers, Task task) {  
        Server serverWithShortestQueue = servers.get(0);  
  
        for(Server server : servers) {  
            if(server.getTasks().size() < serverWithShortestQueue.getTasks().size()) {  
                serverWithShortestQueue = server;  
            }  
        }  
  
        serverWithShortestQueue.addTask(task);  
    }  
}
```




## 2.5 ConcreteShortestTimeStrategy implements Strategy and the addTaskToServers() method

```
public class ConcreteShortestTimeStrategy implements Strategy {  
  
    @Override  
    public void addTaskToServers(List<Server> servers, Task task) {  
        Server serverWithShortestTime = servers.get(0);  
  
        for(Server server : servers) {  
            int currentServerTime = server.getWaitingPeriod().get();  
            int shortestTime = serverWithShortestTime.getWaitingPeriod().get();  
            if(currentServerTime < shortestTime) {  
                serverWithShortestTime = server;  
            }  
        }  
  
        serverWithShortestTime.addTask(task);  
    }  
}
```

## 3. Controller

Fields: view

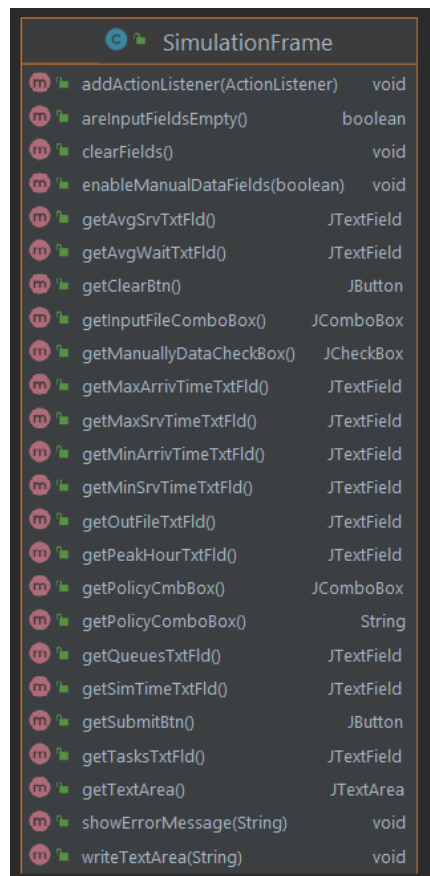
Main method: actionPerformed(), parseInputFields()

Controller		
	view	SimulationFrame
	actionPerformed(ActionEvent)	void
	parseInputFields(SimulationInformation)	void

#### 4. View

Fields: all the swing components

Main methods: getters, setters, enableManualFields(), addActionListeners(),



SimulationFrame		
m	addActionListener(ActionListener)	void
m	areInputFieldsEmpty()	boolean
m	clearFields()	void
m	enableManualDataFields(boolean)	void
m	getAvgSrvTxtFld()	JTextField
m	getAvgWaitTxtFld()	JTextField
m	getClearBtn()	JButton
m	getInpFileComboBox()	JComboBox
m	getManuallyDataCheckBox()	JCheckBox
m	getMaxArrivTimeTxtFld()	JTextField
m	getMaxSrvTimeTxtFld()	JTextField
m	getMinArrivTimeTxtFld()	JTextField
m	getMinSrvTimeTxtFld()	JTextField
m	getOutFileTxtFld()	JTextField
m	getPeakHourTxtFld()	JTextField
m	getPolicyCmbBox()	JComboBox
m	getPolicyComboBox()	String
m	getQueuesTxtFld()	JTextField
m	getSimTimeTxtFld()	JTextField
m	getSubmitBtn()	JButton
m	getTasksTxtFld()	JTextField
m	getTextArea()	JTextArea
m	showErrorMessage(String)	void
m	writeTextArea(String)	void

#### 5. Common

##### 5.1. TaskUtils

Static methods: generateNRandomClients(), computeAverageServiceTime()

##### 5.2. SimulationInformation

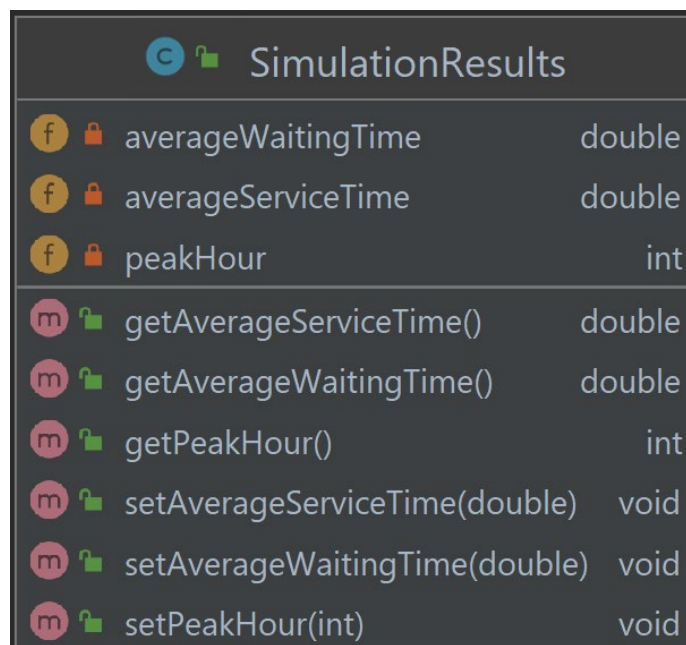
Fields: maxSimulationTime, maxArrivalTime, maxServiceTime, minArrivalTime, ...

Main methods: getters, setters

##### 5.3. SimulationResults

Fields: avgWaitingTime, avgServiceTime, peakHour

Main methods: getters, setters



SimulationResults		
f	averageWaitingTime	double
f	averageServiceTime	double
f	peakHour	int
m	getAverageServiceTime()	double
m	getAverageWaitingTime()	double
m	getPeakHour()	int
m	setAverageServiceTime(double)	void
m	setAverageWaitingTime(double)	void
m	setPeakHour(int)	void

## V. Results

The program performs very well for the three test cases given in the requirements. The output log of events is also created correctly. However, there is a huge lagging for the real-time display on the GUI for the third test. Without using the SwingWorker I have observed that for the third case the output txt would also be incorrect because some steps were actually neglected. We will present below a random, manually user entered test case similar to the first one given but the maximum arrival time is not 30 but 7, and the simulation maximum time is 30. This is for a shorter range in order to not have lengthy txt file. The example is given below:

$N = 4$

$Q = 2$

$t_{simulation}^{MAMM} = 30$   
 $\Phi t_{MIN}^{arrival}, t_{MAMM}^{arrival} \Phi = [2, 7]$   
 $\Phi t_{MIN}^{service}, t_{MAMM}^{service} \Phi = [2, 4]$

Below is the log of events:

```
Time 0
Waiting list: (1,3,3); (4,3,3); (2,4,2); (3,6,2);
Queue 1: closed
Queue 2: closed

Time 1
Waiting list: (1,3,3); (4,3,3); (2,4,2); (3,6,2);
Queue 1: closed
Queue 2: closed

Time 2
Waiting list: (1,3,3); (4,3,3); (2,4,2); (3,6,2);
Queue 1: closed
Queue 2: closed

Time 3
Waiting list: (1,3,3); (4,3,3); (2,4,2); (3,6,2);
Queue 1: (1,3,3);
Queue 2: (4,3,3);

Time 4
Waiting list: (2,4,2); (3,6,2);
Queue 1: (1,3,2); (2,4,2);
Queue 2: (4,3,2);

Time 5
Waiting list: (3,6,2);
Queue 1: (1,3,1); (2,4,2);
Queue 2: (4,3,1);

Time 6
Waiting list: (3,6,2);
Queue 1: (2,4,2);
Queue 2: (3,6,2);

Time 7
Waiting list:
Queue 1: (2,4,1);
Queue 2: (3,6,1);

Time 8
Waiting list:
Queue 1: closed
Queue 2: closed

Average waiting time: 3.00
Average service time: 2.50
Peak hour: 4
```

## VI. Conclusions

This project was very helpful in understanding how threads perform and what is a waiting queue and a processing-based system. It systematized what concurrency and concepts like thread safety mean. I have also worked with `SwingWorker` which proposes a concurrent solution for swing, but also with the strategy design pattern.

The system could help for further developments such as:

- using an executor service instead of own server for thread management
- more complex statistic showcase
- simulation interruption and resuming
- better integration and code organization

## VII. Bibliography

Contributors, W., 2019. *Strategy Pattern*. [Online]

Available at: [https://en.wikipedia.org/wiki/Strategy\\_pattern](https://en.wikipedia.org/wiki/Strategy_pattern)

Jenkov, J., 2020. *Java Concurrency and Multithreading*. [Online]

Available at: <https://www.youtube.com/watch?v=mTGdtC9f4EU&list=PLL8woMHwr36EDxjUoCzboZjedsnhLP1j4>

2022. *Programming Techniques - Assignment 2 Support Presentation*, Cluj-Napoca: UTCN, AC.

*Programming Techniques Course*, Cluj-Napoca: UTCN,

