DOCUMENTATION

ASSIGNMENT 2

STUDENT NAME: GRAD LAURENȚIU-CĂLIN
GROUP: 30425

Contents

1. Assignment Objective

Design and implement a queues management application which assigns clients to queues such that the waiting time is minimized. 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. The queues management application should simulate (by defining a simulation time $t\_simulation$) a series of N clients arriving for service, entering Q queues, waiting, being served, and finally leaving the queues. All clients are generated when the simulation is started and are characterized by three parameters: ID (a number between 1 and N), $t\_arrival$ (simulation time when they are ready to enter the queue) and $t\_service$ (time interval or duration needed to serve the client, i.e. waiting time when the client is in front of the queue). The application tracks the total time spent by every client in the queues and computes the average waiting time. Each client is added to the queue with the minimum waiting time when its $t\_arrival$ time is greater than or equal to the simulation time ($t\_arrival \geq t\_simulation$). The following data should be considered as input data for the application that should be inserted by the user in the application's user interface:

- Number of clients (N)
- Number of queues (Q)
- Simulation interval ($t\_simulation\_MAX$)
- Minimum and maximum arrival time ($t\_arrival\_MIN \leq t\_arrival \leq t\_arrival\_MAX$)
- Minimum and maximum service time ($t\_service\_MIN \leq t\_service \leq t\_service\_MAX$)

2. Problem Analysis, Modeling, Scenarios, Use Cases

The problem aims to model a real-life situation: choosing the optimal queue (out of a certain number) when waiting at a public counter, for example. One wants to finish what he/she has to do as quickly as possible so he/she tries to pick the optimal queue based on some sort of inherent strategy. The problem suggests that there are two main strategies: one picks the shortest waiting line (this is probably the way most of us would do) and one picks the queue with the smallest average waiting time (however, in real life, this means one should know for sure how much each task of each person in front of him takes, let alone the computations required to make an educated guess with such a strategy). The first strategy is of real interest because the results it provides should be similar to what should happen in a real-life scenario. However, the second strategy is the mathematically optimal one because it minimizes the overall working time. The problem statement underlines the requested parameters of the simulation we intend to create: a global time limit of the simulation (*simulation interval*), the number of clients (*N*) and the number of queues these people can go to (*Q*). More limiting parameters are provided in order to manage better the generation of

people with random tasks: arrival time constants (*MIN* and *MAX*) specify the bounds of the time interval when people may appear in the simulation (and should be distributed to the corresponding optimal queue) and the service time constants (*MIN* and *MAX*) define the interval of time a task may take. These additional parameters may prove useful especially for statistical analysis.

The Queue Simulator Application is designed to create the environment mentioned above. It generates *N* clients, each aiming to solve a single task that takes somewhere between *service_time_MIN* and *service_time_MAX*. The time stamp when any of these clients appear is randomly generated and is somewhere in the interval [*arrival_time_MIN*; *arrival_time_MAX*]. After the task generation occurs, the application launches *Q* queues (threads) that process in parallel the people that are corresponding to each one. The application uses this information to generate a window that simulates time stamp by time stamp how the queues evolve from time 0 until the time limit.

The application is intended to work under some assumptions:

a) The source files are run inside a Java Integrated Development Environment such as IntelliJ Idea, Eclipse IDE etc.
b) Once the project is opened with an IDE, the *.main()* method can be run.
c) A pop-up window should appear that asks for the mentioned input parameters. In addition to them a strategy to position in the queue should be chosen. The "pick the shortest queue at current time" strategy is the default chosen policy.
d) If all the parameters are inserted, the "START SIMMULATION" button can be pressed and the window changes into the simulation panel. If any parameter is not a valid number, the console will print an exception message and the application will not start. However, the application will not halt or close.
e) The application will start the simulation and wait indefinitely (even if the simulation finishes) until the close button is pressed.

NOTE: At any time, the application can be closed if the close button (X) in the top right corner is pressed.

The user diagram describes the dependencies between the user's interactions and the system represented by the Queue Simulator Application. The user tries to follow the simulation that he/she has described conceptually by the already fed input parameters.

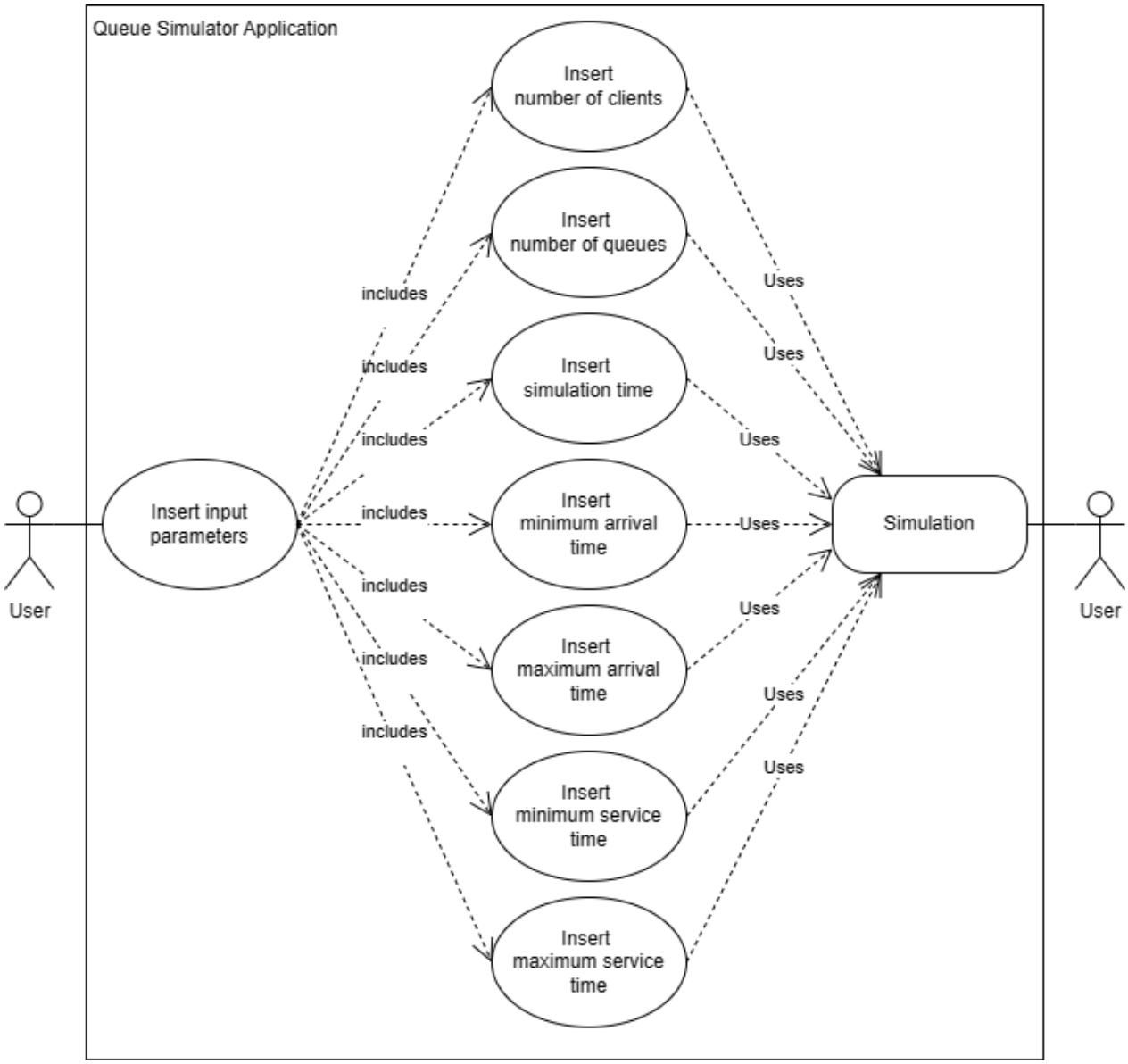


Figure 1. User diagram of the program

3. Design

I have chosen the canonical approach in solving a programming problem that involves a graphical user interface and a computational model: the application is divided conceptually into three directories (packages): “backend”, “frontend” and “utility” folders. I have followed partially the problem model described in the support presentation that was provided at the course. In the backend package all classes that model the queue, how the simulation works, how data evolves over time etc. are defined, while in the frontend package the focus is on the implementation of the GUI, how the displayed layers are connected to each other and what certain actions cause events that are handled correspondingly. The utility package stores enumerations and interfaces for better management and easier possible further extensions.

The fact that one of the problem requirements is that we must implement (at least) two queueing strategies means an enumeration is recommended to use. Moreover, because each strategy must also have a code body, I chose to create an interface that sets a standard. (These are great features that ease the further development process). All these functionalities are grouped in the utility package.

The backend package ("business logic") is composed of several classes that model the underlying mechanisms of the application. The *SimulationManager*, as its name suggests, manages the resources of the running simulation. It uses a *Scheduler* to decide to which queue (modeled by the *Server* class) a person (modeled by the *Task* class) should be sent (appended). The *ShortestQueueStrategy* and the *TimeStrategy* implement the *Strategy* interface, and each correspond to an entry in the *SelectionPolicy* enumeration. More details about what each class does will be discussed in the Implementation section.

The frontend package is centered around an *ApplicationFrame*. This class is what is displayed on the screen. It houses all the other GUI components such as smaller sections (panels), buttons, labels etc. Like any Java Swing GUI, its underlying architecture is layered meaning that more specific components are grouped into a panel that is glued somewhere in the main frame based on a Layout. *CardLayout* is used to manage 2 windows: the data insertion window (the front panel)and the simulation window (the back panel). The change of panel is triggered by the press of the *StartSimulation* button after all the required fields are filled correctly. The front panel (i.e. the panel that expects the user to input data) is composed of pairs of "input fields" (JLabel - JTextField) placed into a JPanel that is inserted into the main frame. Seven such fields are required and provided. In addition to them a group of buttons is provided to select which *SelectionStrategy* should be used by the simulation to be started. The back panel (i.e. where the simulation is happening) is computed only after the input data is fed to the program and the *StartSimulation* button is pressed.

The back panel follows the execution of the simulation time stamp by time stamp. In the top left corner, the current time of the simulation is displayed. The first row of yellow squares represents how many people are supposed to arrive at the queues during the simulation. Starting at the second row, each first circle (colored pink) represents a queue or a server where people will be routed. The green squares following each circle represent empty spaces at the queue. For a large number of queues/people the amount of circles and squares displayed is capped (20/30) so that the display area does not get too messy or cramped.

The program creates two text files: “stateOfServers.txt” and “tasks.txt”. The first file represents the log of events: it stores the state of each server at each time stamp starting at time 0 up to the time limit. The second file saves the randomly generated people: their id, arrival time and how much their task takes.

```
Current time: <number>
Waiting people:
>><id> → (<arrival time>: <number>; <service time>: <number>)
...

State of servers:
server: <name> has amount of work: <integer>
        <id> → (<arrival time>: <number>; <service time>: <number>)
        ... *integer fields | no fields*
server: <name> has amount of work: <integer> | is closed
        <id> → (<arrival time>: <number>; <service time>: <number>)
        ... *integer fields | no fields*
...
*the number of total queues*

Average overall waiting time: <double>
```

File format 1. The format of the "stateOfServers.txt" file

```
<integer> generated tasks:
<id> → (<arrival time>: <number>; <service time>: <number>)
<id> → (<arrival time>: <number>; <service time>: <number>)
...
*integer tasks*
```

File format 2. The format of the "tasks.txt" file

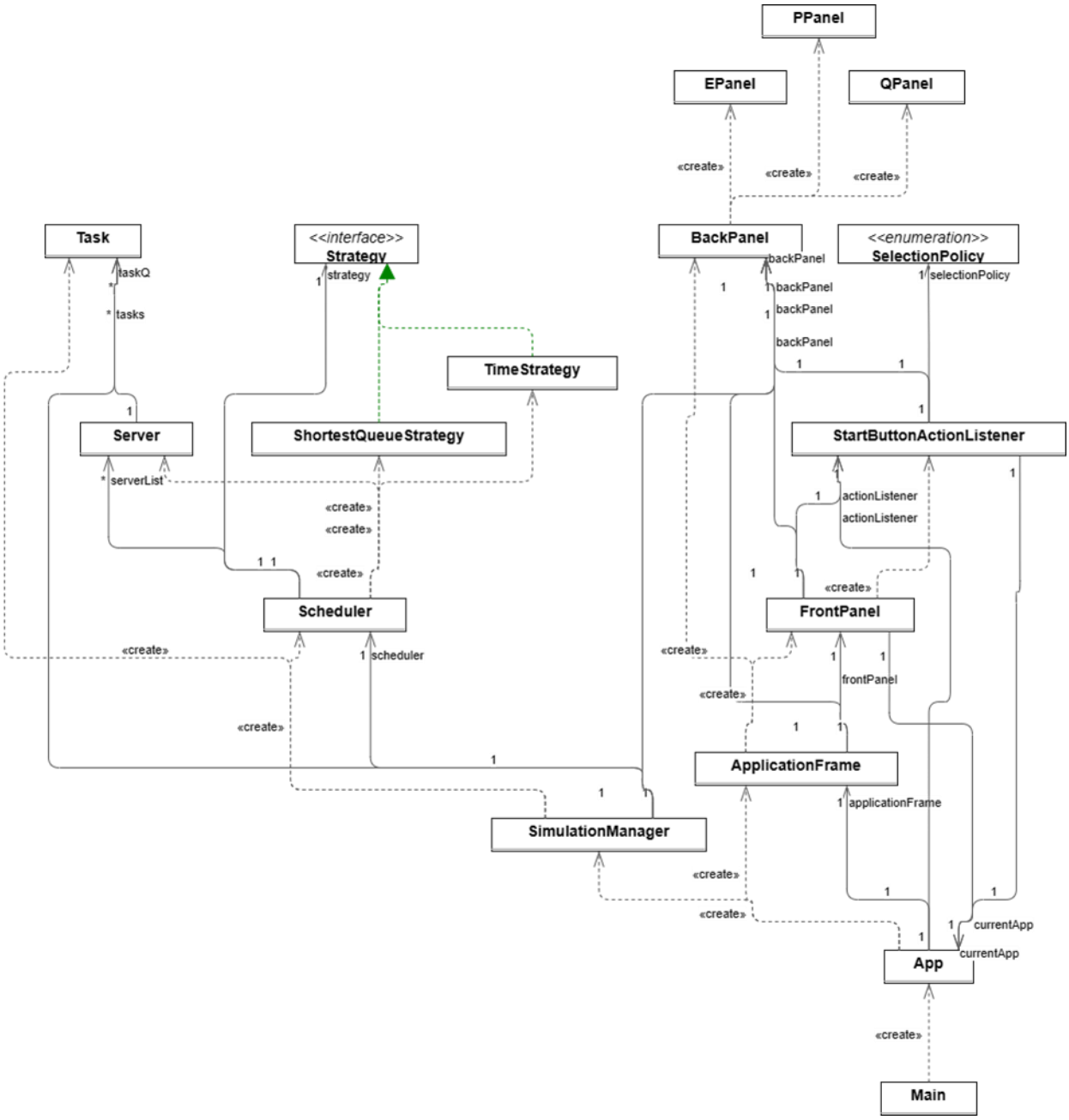
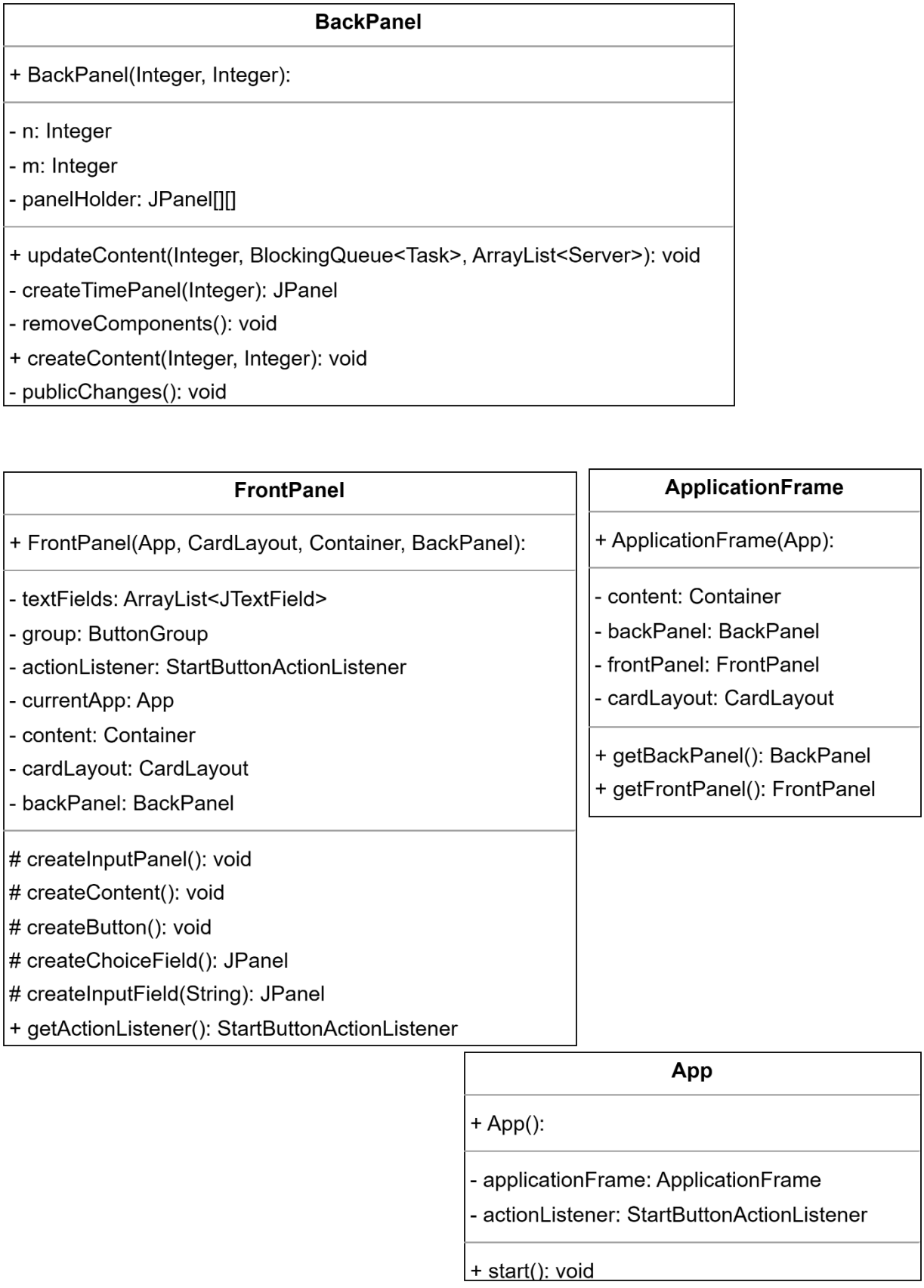
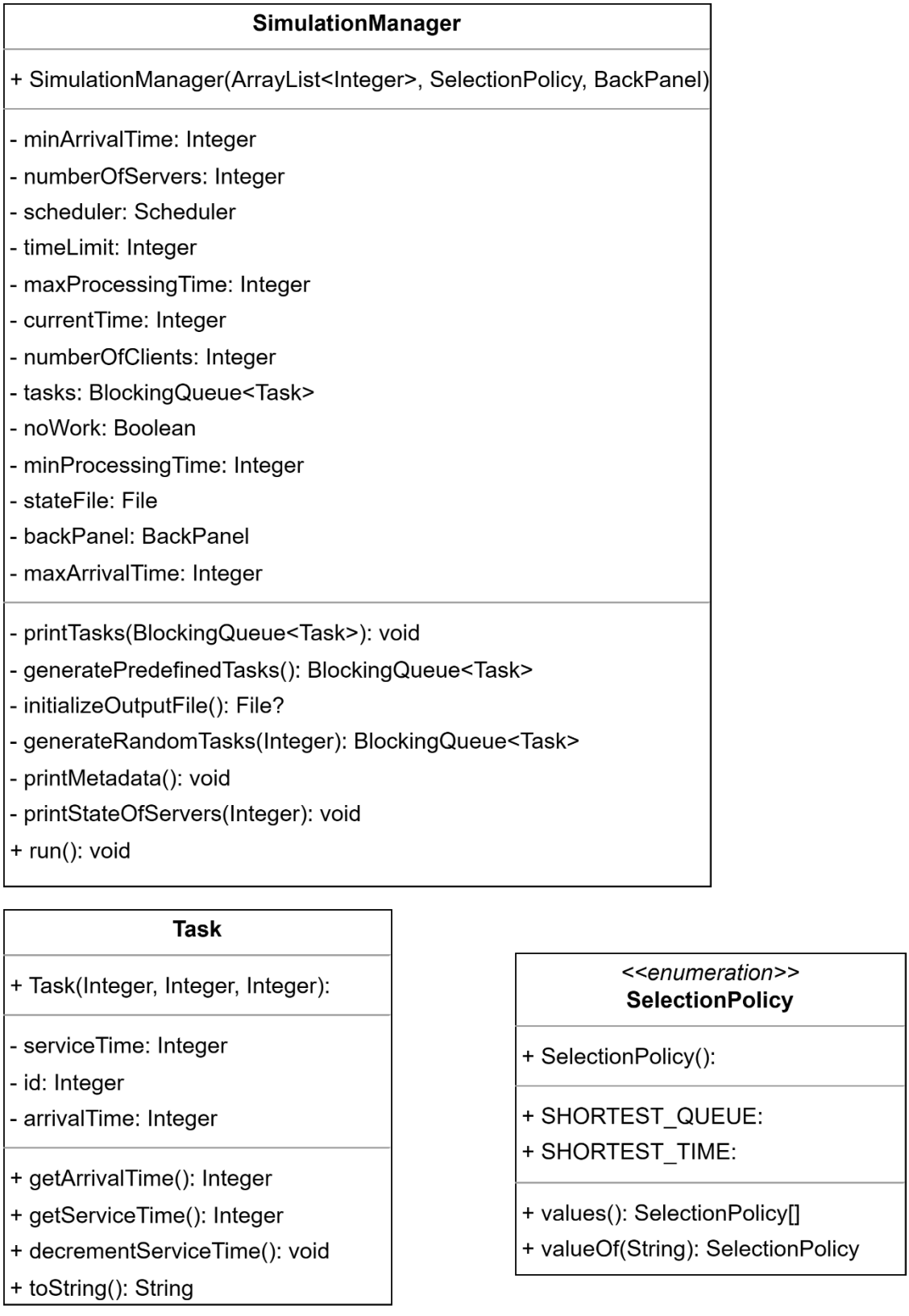
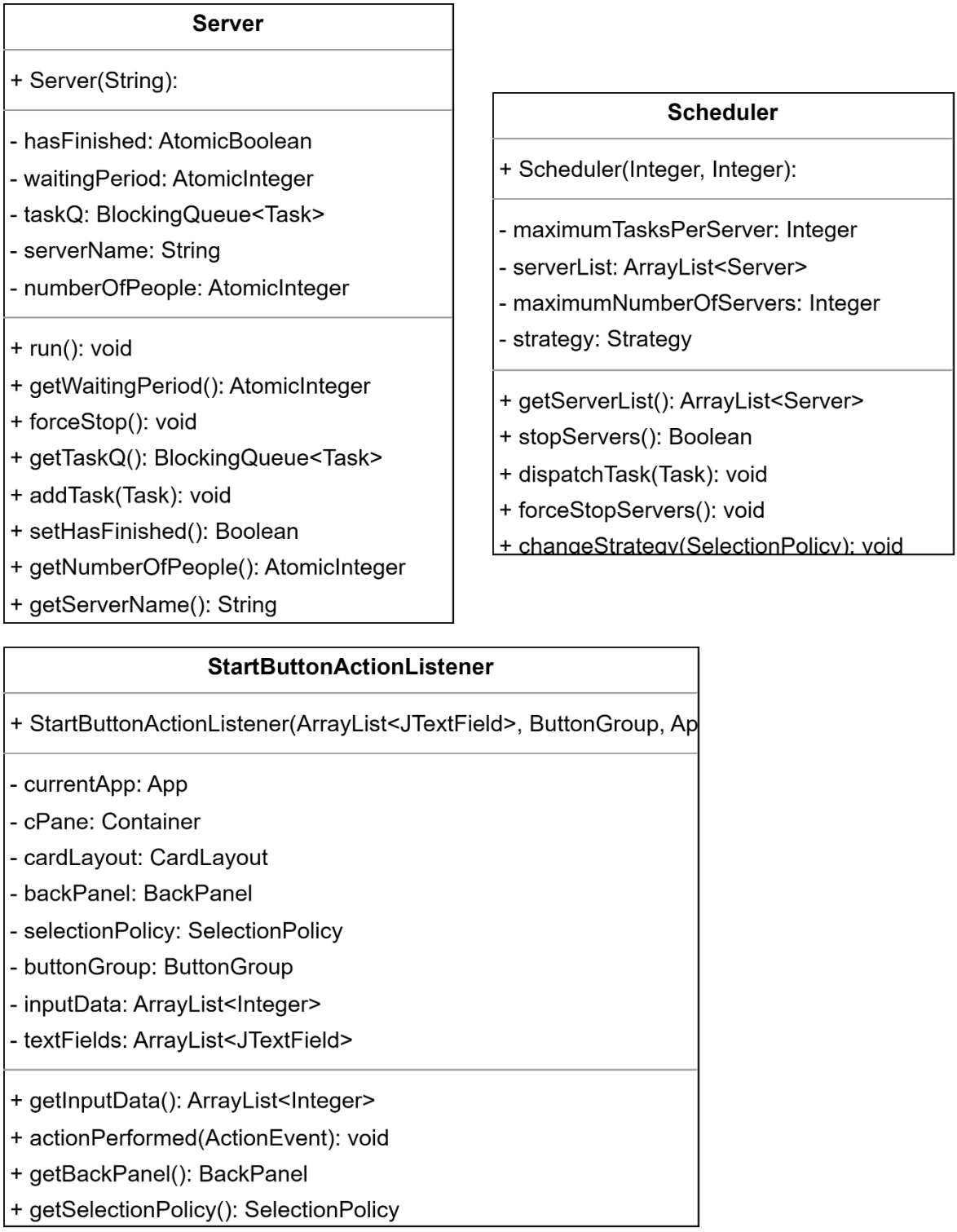
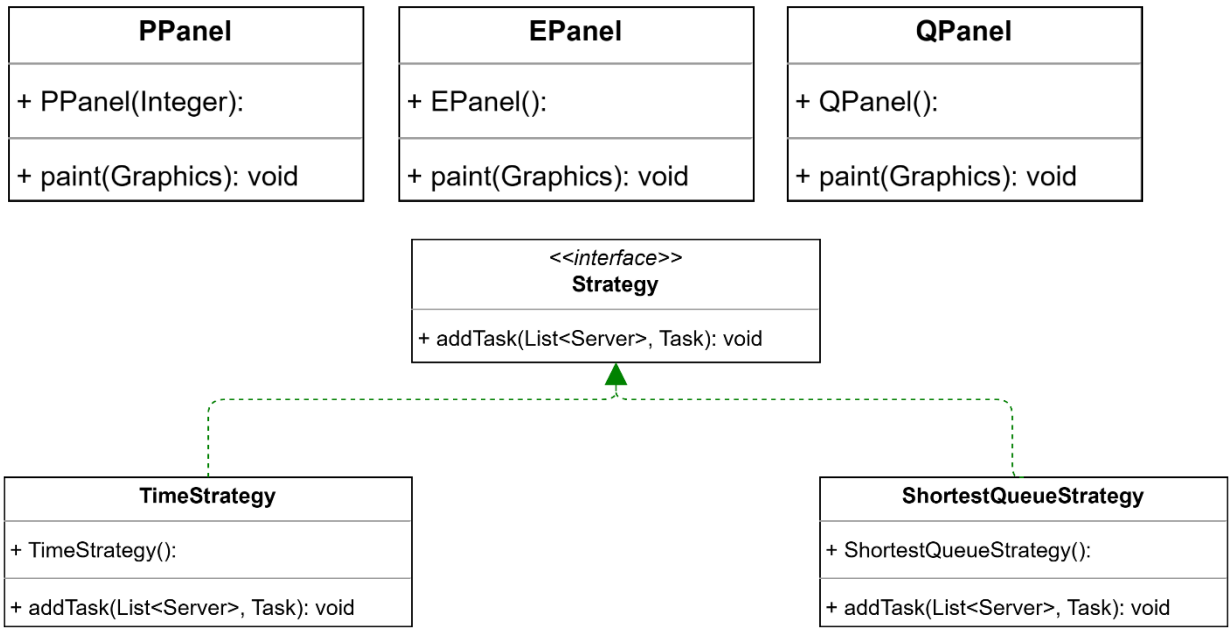


Figure 2. The UML diagram of the application


BackPanel
+ BackPanel(Integer, Integer):
- n: Integer
- m: Integer
- panelHolder: JPanel[][]
+ updateContent(Integer, BlockingQueue<Task>, ArrayList<Server>): void
- createTimePanel(Integer): JPanel
- removeComponents(): void
+ createContent(Integer, Integer): void
- publicChanges(): void
FrontPanel
+ FrontPanel(App, CardLayout, Container, BackPanel):
- textFields: ArrayList<JTextField>
- group: ButtonGroup
- actionListener: StartButtonActionListener
- currentApp: App
- content: Container
- cardLayout: CardLayout
- backPanel: BackPanel
createInputPanel(): void
createContent(): void
createButton(): void
createChoiceField(): JPanel
createInputField(String): JPanel
+ getActionListener(): StartButtonActionListener
ApplicationFrame
+ ApplicationFrame(App):
- content: Container
- backPanel: BackPanel
- frontPanel: FrontPanel
- cardLayout: CardLayout
+ getBackPanel(): BackPanel
+ getFrontPanel(): FrontPanel
App
+ App():
- applicationFrame: ApplicationFrame
- actionListener: StartButtonActionListener
+ start(): void


SimulationManager
+ SimulationManager(ArrayList<Integer>, SelectionPolicy, BackPanel)
- minArrivalTime: Integer
- numberOfServers: Integer
- scheduler: Scheduler
- timeLimit: Integer
- maxProcessingTime: Integer
- currentTime: Integer
- numberOfClients: Integer
- tasks: BlockingQueue<Task>
- noWork: Boolean
- minProcessingTime: Integer
- stateFile: File
- backPanel: BackPanel
- maxArrivalTime: Integer
- printTasks(BlockingQueue<Task>): void
- generatePredefinedTasks(): BlockingQueue<Task>
- initializeOutputFile(): File?
- generateRandomTasks(Integer): BlockingQueue<Task>
- printMetadata(): void
- printStateOfServers(Integer): void
+ run(): void
Task
+ Task(Integer, Integer, Integer):
- serviceTime: Integer
- id: Integer
- arrivalTime: Integer
+ getArrivalTime(): Integer
+ getServiceTime(): Integer
+ decrementServiceTime(): void
+ toString(): String
<<enumeration>>
SelectionPolicy
+ SelectionPolicy():
+ SHORTEST_QUEUE:
+ SHORTEST_TIME:
+ values(): SelectionPolicy[]
+ valueOf(String): SelectionPolicy


Server
+ Server(String):
- hasFinished: AtomicBoolean
- waitingPeriod: AtomicInteger
- taskQ: BlockingQueue<Task>
- serverName: String
- numberOfPeople: AtomicInteger
+ run(): void
+ getWaitingPeriod(): AtomicInteger
+ forceStop(): void
+ getTaskQ(): BlockingQueue<Task>
+ addTask(Task): void
+ setHasFinished(): Boolean
+ getNumberOfPeople(): AtomicInteger
+ getServerName(): String
Scheduler
+ Scheduler(Integer, Integer):
- maximumTasksPerServer: Integer
- serverList: ArrayList<Server>
- maximumNumberOfServers: Integer
- strategy: Strategy
+ getServerList(): ArrayList<Server>
+ stopServers(): Boolean
+ dispatchTask(Task): void
+ forceStopServers(): void
+ changeStrategy(SelectionPolicy): void
StartButtonActionListener
+ StartButtonActionListener(ArrayList<JTextField>, ButtonGroup, Ap
- currentApp: App
- cPane: Container
- cardLayout: CardLayout
- backPanel: BackPanel
- selectionPolicy: SelectionPolicy
- buttonGroup: ButtonGroup
- inputData: ArrayList<Integer>
- textFields: ArrayList<JTextField>
+ getInputData(): ArrayList<Integer>
+ actionPerformed(ActionEvent): void
+ getBackPanel(): BackPanel
+ getSelectionPolicy(): SelectionPolicy



4. Implementation

The implementation of the classes and of the main methods is discussed in this section.

- **Server**
  - Implements the Runnable interface.
  - *.run()* while the simulation is not finished the waiting period of the queue is incremented with the number of people that are waiting (i.e. size - 1); the person in front of the queue is checked for elimination (if enough time to finish its task has passed).
  - *.addTask()* appends a new person to the end of the queue.
  - *.setHasFinished()* if there are no tasks in the queue the thread may stop.
  - *.forceStop()* stops the thread regardless of the number of people there are inside.
- **SimulationManager**
  - Implements the Runnable interface.
  - *.initializeOutputFile()* creates a new file where the log of events is saved.
  - *.run()* manages the main thread of the application; loops through the tasks, updates the current time of the simulation, sends the corresponding tasks to the queues based on the chosen strategy; when the time of the simulation is exceeded or there are no people waiting or in any queue.
  - *.printMetadata()* computes and appends the overall waiting time to the log file.
  - *printTasks()* creates a new file where the people id, their arrival time and their service time is stored for debugging.

- *.printStateOfServers()* prints inside the log file the current state of the queues at each time stamp.
- *.generateRandomTasks()* creates *N* people based on the input data that is provided in the Graphical User Interface.
- *.generatePredefinedTasks()* hardcoded creation of people; used for debugging.

- **Scheduler**
  - *Scheduler()* starts *Q* queues (servers).
  - *.changeStrategy()* switches the strategy based on the input data.
  - *.dispatchTask()* calls the method corresponding to the selection policy that is used.
  - *.forceStopServers()* forcefully stops all servers.
  - *.stopServers()* used to find if there are servers that are working; empty queue servers are stopped.
- **Task**
  - The conceptual model of a person trying to get into queue to solve a problem that takes a certain amount of time.
  - *.toString()* creates a more intuitive printing format for the class.
- **TimeStrategy**
  - *.addTask()* implementation of the method in the interface Strategy; loops through the queue, searches for the server with the smallest waiting time at the current time stamp and returns the best choice.
- **ShortestQueueStrategy**
  - *.addTask()* implementation of the method in the interface Strategy; loops through the queue, searches for the emptiest server and returns the best choice.
- **ApplicationFrame**
  - *ApplicationFrame()* creates the required GUI objects, sets the layout and the dimension of the window.
- **BackPanel**
  - uses a matrix of JPanels to create a matrix grid where the corresponding panels (representing queues, people or blank/empty spaces) are stored; the housing panel is designed based on the layout of this matrix.
- **FrontPanel**
  - *.createButton()* creates the "Start Simulation" button and links it to the action listener
  - *.createInputPanel()* puts together the input section (composed of six input fields) together with the choice field.
  - *.createChoiceField()* creates the choice field composed of a group of two radio buttons and their labels.
  - *.createInputField()* creates the structure of a text field and its label.
  - *.creteContext()* fixes all the elements of the front panel.
- **EPanel**
  - paints a green square inside a panel, used to represent a free space at a queue.

- **QPanel**
  - paints a pink circle inside a panel, used to represent a queue.
- **PPanel**
  - Paints a yellow square inside a panel, used to represent a person.
- **StartButtonActionListener**
  - *.actionPerformed()* converts the string values that are taken from the input panel into integers that are transmitted to the internal model.
- **SelectionPolicy**
  - The enumeration that stores the possible strategy identifiers.
- **Strategy**
  - *.addTask()* the standard method used by each implementation of this interface.
- **App**
  - *.start()* instantiates the resources of the application, in particular the display window, the manager of the simulation and its thread.
  - *App()* creates the simulation frame and links it to the current instance.
- **Main**
  - *.main()* is called when the program begins its execution.

5. Results

Multiple simulation runs have been conducted. Some of them have been manually checked and the results were correct. Some of them are listed below. The output files are found in the directory "Documentation resources".

Test results:

a) Input parameters:

Number of People $= 4$
Number of Queues $= 2$
Time limit $= 60\ seconds$
Minimum arrival time $= 2$
Maximum arrival time $= 30$
Minimum service time $= 2$
Maximum service time $= 4$

i. Append to the shortest queue strategy results are stored in the files:
"TEST1_ShortestQueue_tasks.txt"
"TEST1_ShortestQueue_stateOfServers.txt"

ii. Append to the queue with the smallest average waiting time are stored in the files:
"TEST1_ShortestWaitingTime_tasks.txt"
"TEST1_ShortestWaitingTime_stateOfServers.txt"

b) Input parameters:

Number of People $= 50$
Number of Queues $= 5$
Time limit $= 60\ seconds$
Minimum arrival time $= 2$
Maximum arrival time $= 40$
Minimum service time $= 1$
Maximum service time $= 7$

i. Append to the shortest queue strategy are stored in the files:
"TEST2_ShortestQueue_tasks.txt"
"TEST2_ShortestQueue_stateOfServers.txt"

ii. Append to the queue with the smallest average waiting time are stored in the files:
"TEST2_ShortestQueue_tasks.txt"
"TEST2_ShortestQueue_stateOfServers.txt"

c) Input parameters:

Number of People $= 1000$
Number of Queues $= 20$
Time limit $= 200\ seconds$
Minimum arrival time $= 10$
Maximum arrival time $= 100$
Minimum service time $= 3$
Maximum service time $= 9$

i. Append to the shortest queue strategy are stored in the files:
"TEST3_ShortestQueue_tasks.txt"
"TEST3_ShortestQueue_stateOfServers.txt"

ii. Append to the queue with the smallest average waiting time are stored in the files:
"TEST3_ShortestQueue_tasks.txt"
"TEST3_ShortestQueue_stateOfServers.txt"

Based on more iterations of these tests I could draw no certain answer: both strategies perform almost the same in terms of overall waiting time. The peak hour and the average service time are not that useful in such a rough interpretation of the results. With more advanced tools, which may be provided by statistics, these two output results may characterize the population of people of a certain test. Similar populations can be used to analyze the two queueing strategies and so an efficiency-related comparison between them may come out on top.

6. Conclusions

This assignment proved to be a bit challenging for me because I have never worked before with threads before. The main issue I had was to keep track of the independently running threads especially because I had to adapt the classical way to debug (in console) I am used to so that I could handle and understand the parallel evolution of the program. Another challenge was to break down the application itself into independent modules that can perform more general tasks or are (more or less) open to extension while keeping it simple and easy to understand. I got more familiar to layouts, panels, how they interact and how a more complex window can be designed. I have used group buttons for the first time and tried frame by frame generation. I practiced the decomposition of classes into subclasses and aimed the design short and effective methods that are more suitable to the Java style.

The final application is not the best it can be for sure. I consider its main defects to be the graphical user interface implementation (the code is a bit chaotic). However, the GUI looks decent, is practical and does its job. The backend structure could be optimized as well.

Other further developments:

- Improved class design for the GUI.
- Improved and simplified inter-class connections.
- Improved, simplified, and more specialized intra-class connections.
- New possible strategies for implementation.
- Cleaner, more explicit code style.

7. Bibliography

1. Programming Techniques lectures, laboratories, and support materials
2. Paul Deitel, Harvey Deitel, Java How to program (10th edition), Publisher: Pearson Education, Inc., Upper Saddle River, NJ, United States, ISBN: 978-0-13-380780-6, Published: 2015
3. Ștefan Tanasă, Ștefan Andrei, Cristian Olaru, Java: de la 0 la expert (2nd edition), Publisher: Polirom, Iași, Romania, ISBN: 978-973-46-2405-8, Published: 2011
4. Java Thread Pool Example using Executors and ThreadPoolExecutor (javacodegeeks.com)
5. java.util.Timer.schedule() Method (tutorialspoint.com)
6. Lesson: Concurrency (The Java™ Tutorials > Essential Java Classes) (oracle.com)
7. Runnable vs. Callable in Java | Baeldung
8. Use Case Diagrams | Unified Modeling Language (UML) - GeeksforGeeks
9. Unified Modeling Language - Wikipedia
10. draw.io (diagrams.net)