DOCUMENTATION

ASSIGNMENT 2

STUDENT NAME: BÁLINT ROLAND
GROUP: 30424

TABLE OF CONTENTS

1. Objective

The objective of this project is to deliver a Java application, which will simulate the real-time evolution of a queue. To do so, we must provide the user the ability to enter necessary data (number of queues, number of tasks, simulation time, and other task- specific information) through a graphical user interface. After validating the data, the simulation may begin.

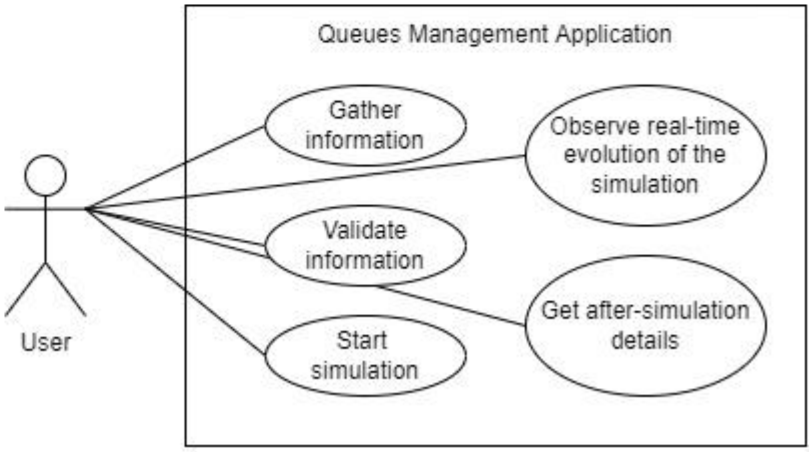
To understand it more clearly, I will take a real-life analogy for this task. Let's imagine we go to an office. There are a certain number of employees willing to help us- each of them representing the "head" of a certain queue. We, as a person with a problem to be solved, can be looked upon as "tasks". Usually, in such scenarios, there may be more tasks than office workers. In this case, we must stay in line. These lines represent the queues themselves. Only one question remains: which line should we choose?

This is where our application comes in handy: by choosing the appropriate selection policy, an efficient way to organize work is possible. This will save us time, and time is money.

As a secondary objective, the desired output of this assignment should be easy-to-use, and it should also display, at the end of the simulation, data about how everything went: average waiting time, peak hour, average service time. As a school project, another secondary goal could be to practice working with threads, to introduce new concepts, such as Java concurrency, thread-safe solutions, parallelism.

2. Analysis, modeling, scenes, use cases

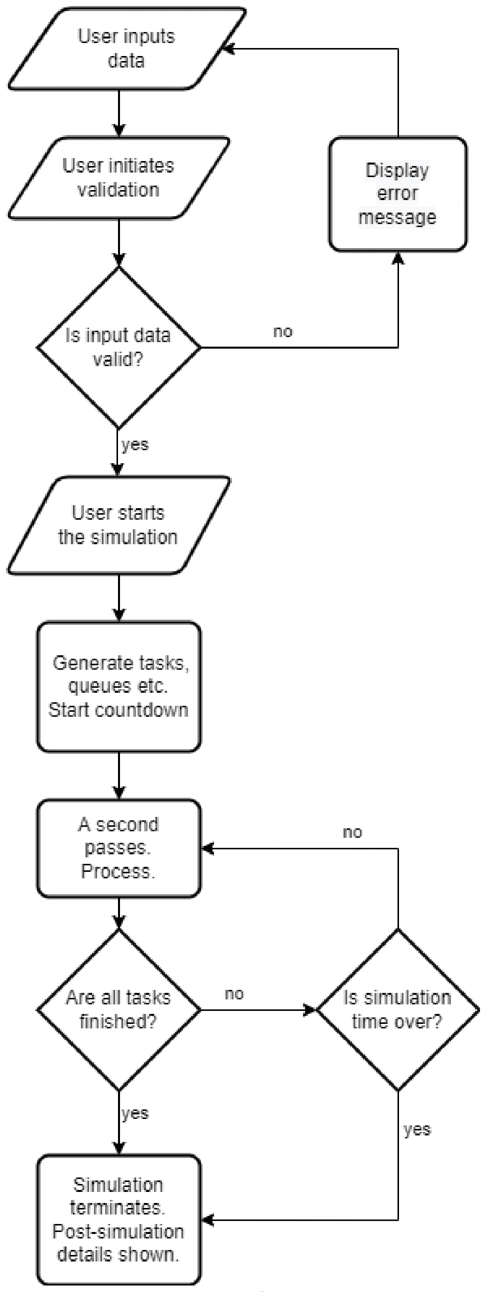
Even after the short introduction given in the previous section, one may say that things still aren't crystal clear- and that is completely true. In comparison with the first assignment of the course (Polynomial Calculator), this one is more subjective, leaving a lot of room for personalization and design choices. However, it is very important to underline the principal functionalities of this project, not only because it must be graded objectively, but also because as a programmer, I should have a clear goal ahead of me when coding. To achieve this, a use- case diagram may be an appropriate beginning:


Queues Management Application
Gather information
Observe real-time evolution of the simulation
Validate information
Get after-simulation details
Start simulation
User

Use-Case Diagram

Of course, to reach the use-cases presented above is only the main objective, and while this is crucial, other non-functional qualities are desirable as well. Just to mention some, think about the ones needed for any project of this kind: a clear, easily understandable and maintainable code, correct and deterministic logic. Since the project is written in Java, the Java naming conventions should not be violated, and an object-oriented approach should be utilized when writing the code. Apart of all this, the task itself is not super complicated, it's just very vaguely specified, and hence a lot of inconsistencies may appear throughout development- these should be avoided.

Before digging too deep into the analysis of the problem and getting too political, let me present a flowchart containing the success scenario and other alternatives directions the flow of application may take:


User inputs data
User initiates validation
Is input data valid?
no
Display error message
yes
User starts the simulation
Generate tasks, queues etc. Start countdown
A second passes. Process.
no
Are all tasks finished?
no
Is simulation time over?
yes
yes
Simulation terminates. Post-simulation details shown.

Flow Chart of Scenarios

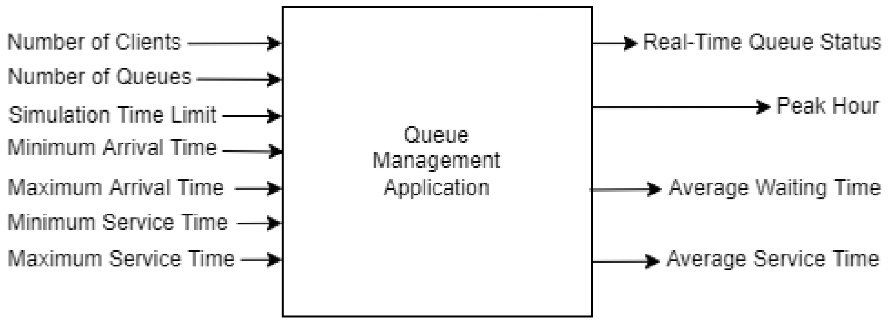
Now we can say that the abstract idea of the solution starts to get a physical shape. We can see that the user interaction is done at the setup level, while the simulation itself happens autonomously. The only occurrence of an error sequence may happen in case the user enters invalid data- the simulation cannot start like this; the user is requested to correct the data. We can also observe that there are two main possibilities of termination:

- *All tasks are finished before reaching the time limit given by the user*. In this case, there is no need to wait for the remaining time, since nothing will happen. No new customers will come, and the details after simulation (average waiting and service time, peak hour) will not be affected by the "dead-hours". Hence, the simulation may terminate early.

- *Time limit is reached*. Even if there are remaining tasks, the simulation must end.

3. Design

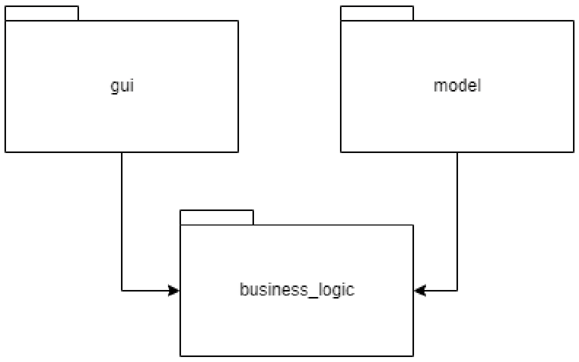
All in all, so far, the system's black box looks as following:



Overall System Design

When choosing the design pattern for this task, the support presentation was of great influence. I must admit that without the "skeleton" code provided, I would have spent a lot more time figuring out how to properly organize my classes. Even like this, there were some difficulties, for example, I ran into a dilemma when implementing the actionListener method- I wasn't sure in which package should it be. Overall, the pattern used is the "Business Logic- Model- GUI" one, but other design patterns may have their footprint on my project, such as the previously used MVC architecture, and the producer-consumer pattern.

Respecting the given idea, modifying it to make it my own and evolving it to fulfill further requirements, the final version of the UML diagrams will be presented. First, the package diagram:

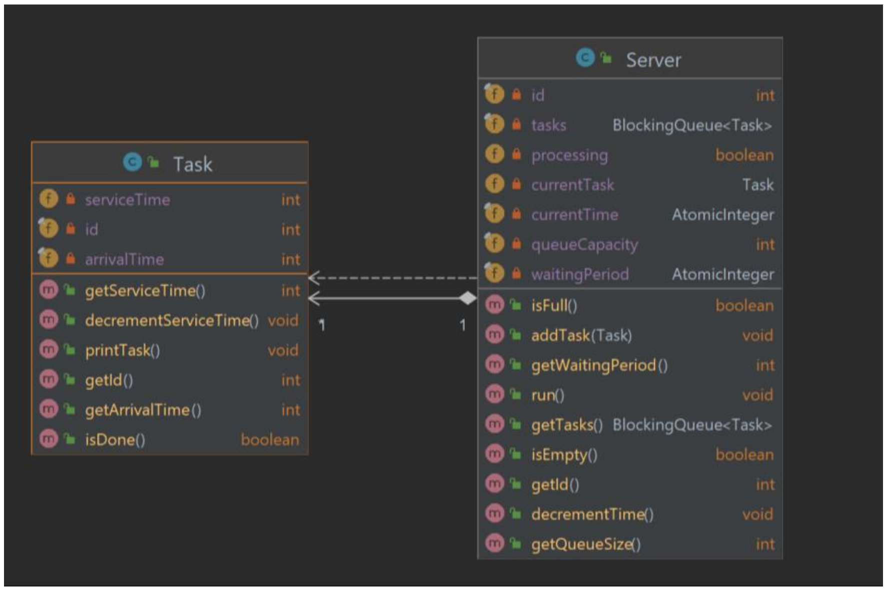


Package Diagram

To keep things organized, similarly to the approach used in the first documentation, I will take each package one after another, and analyze their content. I will provide a class diagram for each, and if it's the case, I will describe the logic used, the mathematical background, the thought process.

It is important to note that, although I started my project with the given code, there were many inconsistencies, both in the logic and the naming conventions. Of course, the cause of this may be that everyone thinks a little differently, and I cannot fully relate to the writer of the presented code. Anyways, it was helpful as a starting point.

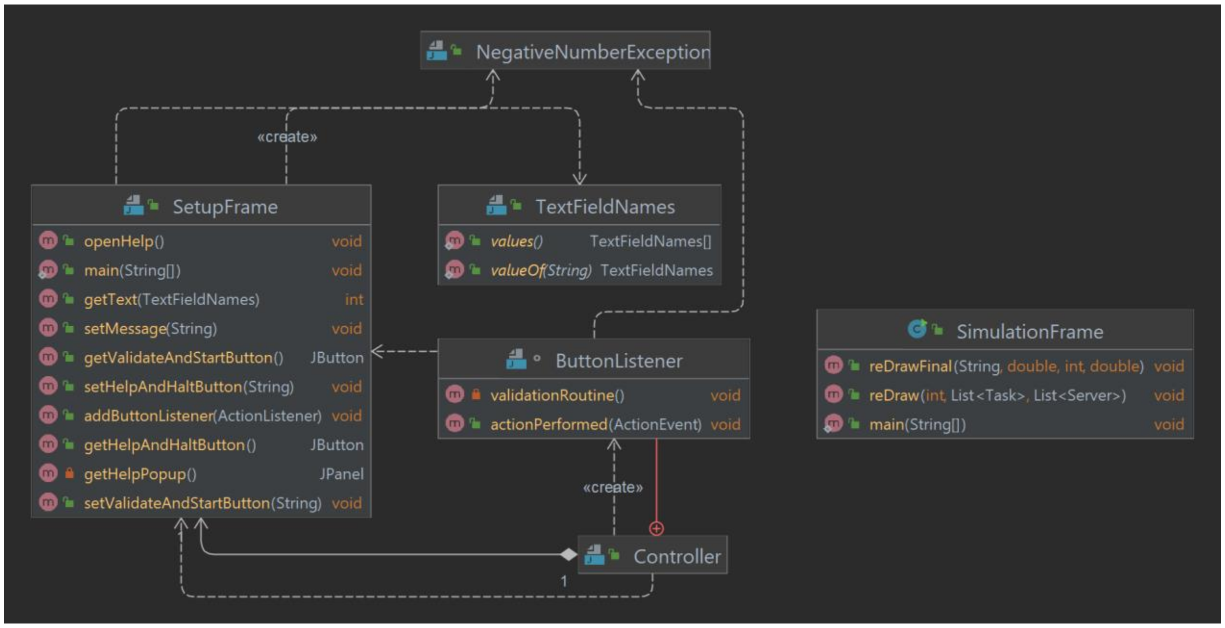
Let's start with the simplest package, *model*. This contains the logic, but unlike in the case of MVC, the model by itself does not provide a solution for the problem. In our case, the *model* package contains the two fundamental elements of the project: *Server* and *Task*. By server I mean a singleton queue, and by task, I mean a single person. By using the object-oriented approach, these classes' methods do provide their behavior, but the "magic", the simulation itself will happen in the *business_logic* package, more specifically, in the *SimulationManager* and *Scheduler* classes.



Class Diagram - Package "model"

One may observe that, with the goal of avoiding concurrency, I used thread-safe data types, such as *BlockingQueue* and *AtomicInteger*. While these types offer a great variety of usage for projects using threads, in the current context I think using them is over-exaggerating. However, I included them, given that they were used in the source code as well. Also, I don't think that I managed to use them at their full potential, because I did not fully understand them and the motivation behind their usage (I think that, in my case, a simple queue would have done the job). But let's not forget that I lack a lot of knowledge in the field of threads, and that I had to develop all this in a very short time, without many explanations or help.

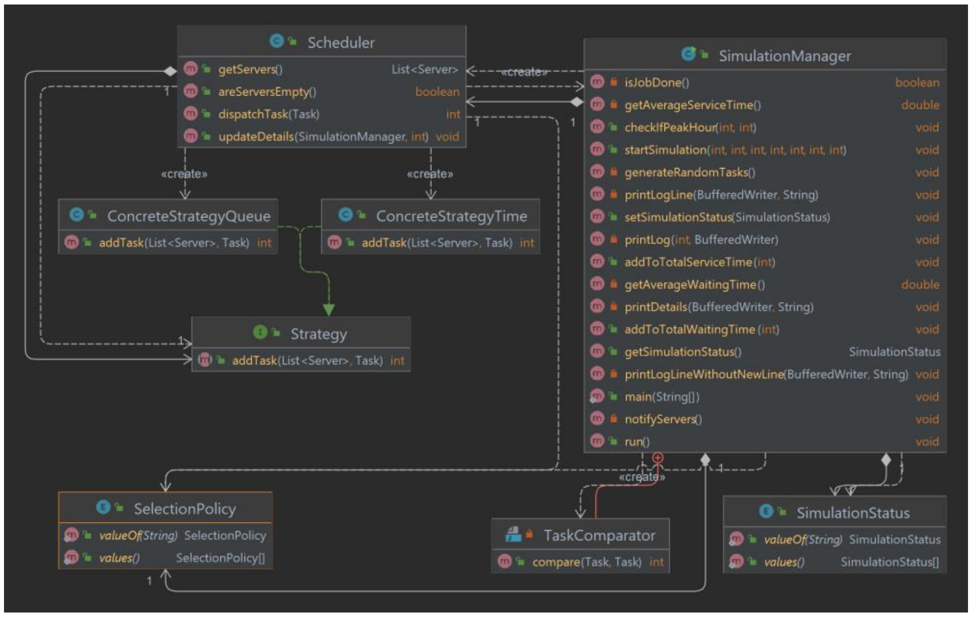
Talking about the classes themselves, I think that the methods are self-explanatory. They are mostly small methods which may seem like extra fillings, but they will make the code much more readable and compact. It is also important to note that the *Server* class implements *Runnable*, hence each queue will represent a separate thread. Therefore they also have a run method, where they will "wait" until the task is done.


NegativeNumberException
«create»
SetupFrame
openHelp() void
main(String[]) void
getText(TextFieldNames) int
setMessage(String) void
getValidateAndStartButton() JButton
setHelpAndHaltButton(String) void
addButtonListener(ActionListener) void
getHelpAndHaltButton() JButton
getHelpPopup() JPanel
setValidateAndStartButton(String) void
TextFieldNames
values() TextFieldNames[]
valueOf(String) TextFieldNames
ButtonListener
validationRoutine() void
actionPerformed(ActionEvent) void
SimulationFrame
reDrawFinal(String, double, int, double) void
reDraw(int, List<Task>, List<Server>) void
main(String[]) void
«create»
Controller
1
Class Diagram - Package "gui"

Moving on, the class diagram for classes of package *gui* is presented above. The first thing that can be seen is that, for the sake of simplicity and encapsulation, I divided the graphical user interface into two frames: the one accountable for the setup, and the one showing the simulation itself. I also left the main methods inside them- they were used to instantiate the frames alone, to avoid running the whole code every time I checked the result of a little change in the code of them. The look of these will be presented in the Implementation section.

Another detail is that I included *Controller*, a.k.a. the front-end – back-end connection here since this package seemed the most suitable for it. The business logic package was already overwhelmed, and by putting this class here, I divided the responsibility between the packages in a more balanced manner. Anyways, the only source of events are the two buttons, the *help*, and the *validate/ start/ halt* button. There are other cool things I did, but more on this in the Implementation section.

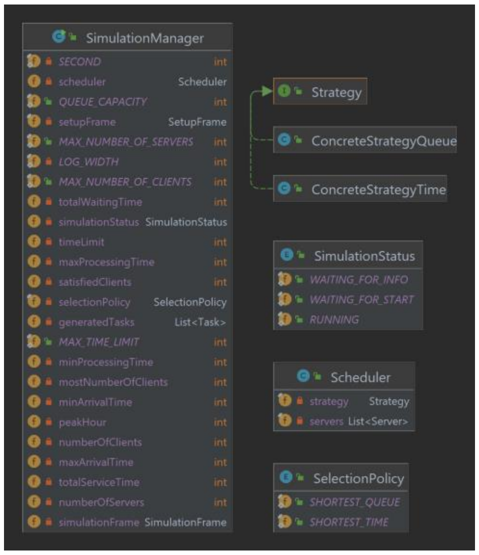
The class diagram of classes from business logic package looks as follows:



Scheduler
getServers() List<Server>
areServersEmpty() boolean
dispatchTask(Task) int
updateDetails(SimulationManager, int) void
«create»
«create»
ConcreteStrategyQueue
addTask(List<Server>, Task) int
ConcreteStrategyTime
addTask(List<Server>, Task) int
Strategy
addTask(List<Server>, Task) int
SimulationManager
isJobDone() boolean
getAverageServiceTime() double
checkIfPeakHour(int, int) void
startSimulation(int, int, int, int, int, int, int) void
generateRandomTasks() void
printLogLine(BufferedWriter, String) void
setSimulationStatus(SimulationStatus) void
printLog(int, BufferedWriter) void
addToTotalServiceTime(int) void
getAverageWaitingTime() double
printDetails(BufferedWriter, String) void
addToTotalWaitingTime(int) void
getSimulationStatus() SimulationStatus
printLogLineWithoutNewLine(BufferedWriter, String) void
main(String[]) void
notifyServers() void
run() void
«create»
SelectionPolicy
valueOf(String) SelectionPolicy
values() SelectionPolicy[]
TaskComparator
compare(Task, Task) int
SimulationStatus
valueOf(String) SimulationStatus
values() SimulationStatus[]
1
1
1
1
1
1

Class Diagram - Package "business_logic" - no fields

Since the diagram was too big with fields included, I made a separate one with the fields only (they are important when analyzing the design choices).



Class Diagram - Package "business_logic" - just fields

4. Implementation

Now I will enter in more details regarding the concrete implementation. Using the same order as in the previous section, I will briefly go through classes of each package.

a. Classes of package *model*

The two classes of package model, as mentioned before, are *Server* and *Task*. They define the fundamental elements of our simulation, their properties and (most of) their behavior.

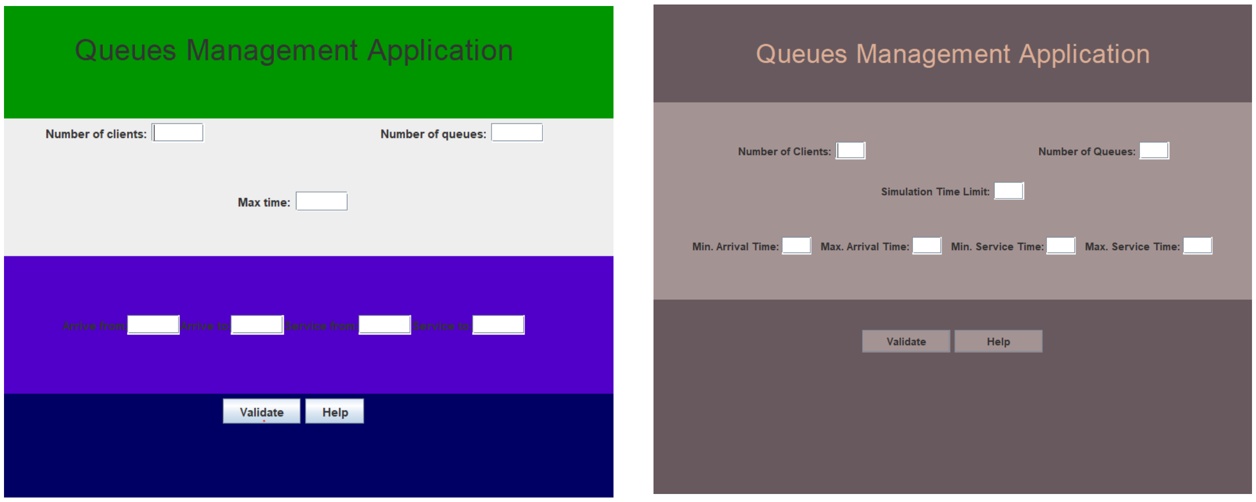
Let's start with *Task*, since it is the most basic entity of our project. Tasks resemble people, and by following the problem statement, they are described by the tuple $(ID^i, t^i_{arrivall}, t^i_{service})$ – they will be represented like this in the logs as well. So, obviously, these are the fields of class *Task*. Other than that, there is nothing sophisticated to be found here: getters, constructor, and a method to decrement the service time of a task, as it will be processed second by second. Getters will help is in displaying each task on the GUI.

Class *Server* is a bit more complex. It is a depicture of a queue, hence it has a collection of tasks, a capacity, an id, a waiting period. Other than that, for (hopefully) making my life easier, I added the fields *currentTime*, *currentTask*, a boolean value *processing* etc. In this class I used some thread-safe types, such as *BlockingQueue* for tasks, *AtomicInteger* for waiting period and current time. Aside of this, in this class and throughout the whole project, I tried to make every possible field final, for the same reason (avoiding concurrency).

b. Classes of package *gui*

The two classes responsible for the outlook of the project are *SetupFrame* and *SimulationFrame.*

SetupFrame provides the possibility of easily putting in all necessary data for the simulation. After writing the inputs in the corresponding fields, the user presses *Validate*. If everything is right, this button will transform into a *Start* button, with which the user may start the simulation itself, making the *SimulationFrame* appear.



SetupFrame prototype vs final version

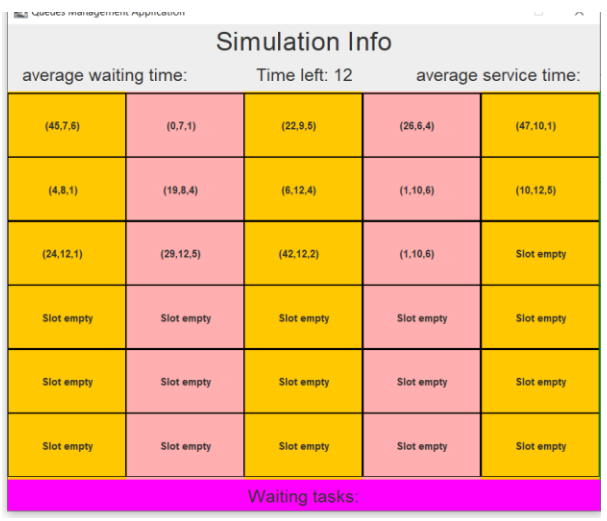
Error messages include:

- *Invalid input*: everything which throws a *NumberFormatException* will trigger this, so empty or illegal outputs (anything that is not a number, hence cannot be parsed into integer) will cause this message to show up
- *Not a valid interval*: there are two time intervals defined by the user: the minimum and maximum times of arrival, respectively of tasks' length. If these numbers do not describe a valid interval i.e., the supposed minimum is bigger than the maximum, this warning will show up
- *Negative value not allowed*: in this case the custom exception *NegativeNumberException* was thrown

- *Value exceeds maximum allowed value*: there are some limitations defined inside the code as constants, to avoid the abuse of the program

These error messages cover all the possible ways inputs can go wrong. Therefore, if the input does not fall in any of these categories, it is considered validated. Even if this is the case, the user needs to explicitly specify the beginning moment of the simulation, by pressing the same button again, which by now holds the text *Start*. The *Help* button is just a little addition, to keep up with the tradition I came up with during my Swing projects.

When the simulation starts, the *SimulationFrame* pops up.



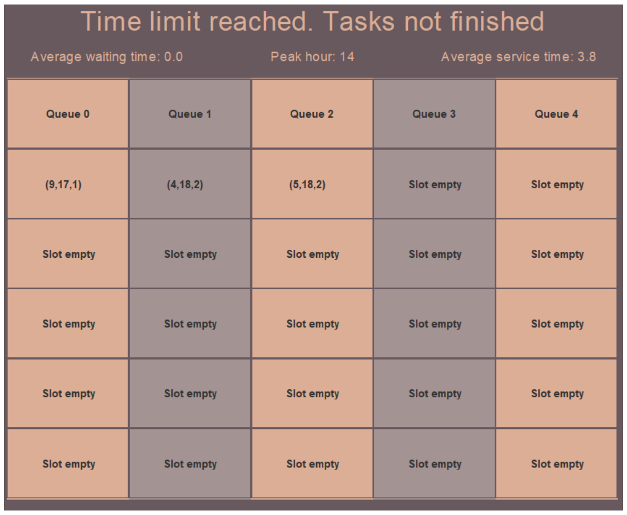
Queues Management Application

Simulation Info

average waiting time: Time left: 12 average service time:

(45,7,6)	(0,7,1)	(22,9,5)	(26,6,4)	(47,10,1)
(4,8,1)	(19,8,4)	(6,12,4)	(1,10,6)	(10,12,5)
(24,12,1)	(29,12,5)	(42,12,2)	(1,10,6)	Slot empty
Slot empty	Slot empty	Slot empty	Slot empty	Slot empty
Slot empty	Slot empty	Slot empty	Slot empty	Slot empty
Slot empty	Slot empty	Slot empty	Slot empty	Slot empty

Waiting tasks:



Time limit reached. Tasks not finished

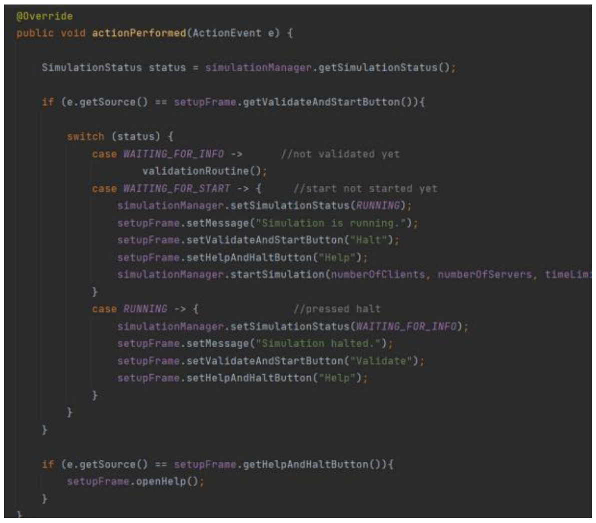
Average waiting time: 0.0 Peak hour: 14 Average service time: 3.8

Queue 0	Queue 1	Queue 2	Queue 3	Queue 4
(9,17,1)	(4,18,2)	(5,18,2)	Slot empty	Slot empty
Slot empty	Slot empty	Slot empty	Slot empty	Slot empty
Slot empty	Slot empty	Slot empty	Slot empty	Slot empty
Slot empty	Slot empty	Slot empty	Slot empty	Slot empty
Slot empty	Slot empty	Slot empty	Slot empty	Slot empty

SimulationFrame prototype vs final version

This is where the nice, visual things happen. Every queue gets a column, and every column has five available slots (this represents the capacity, and since it wasn't specified, it is hard-coded). Tasks may occupy the empty slots, based on the active selection policy. If their arrival time is not reached, or if each slot is full, they must wait. Since there may be a very large number of clients waiting, I chose to limit the displayed waiting clients to the five "closest" customers.

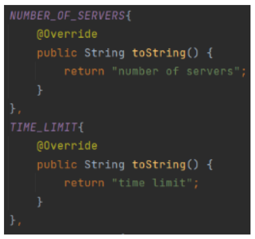
The class which makes the connection between the front-end and back-end possible is Controller. One may think that since only two buttons are "interactive", *Controller*'s job is limited- but this is not the case. This is mostly due to two main reasons. First, I added some more responsibility to this class by making the validations here- after all, this is where the data is read, why not validate it at the same time? Secondly, the button presses cause the state of the simulation juggle from one state to another. At first, I had many states, since I wanted to implement the possibility to pause a simulation, then to continue or halt it. This caused to much headache, so at the end I managed to simplify the action listener method to the following form:



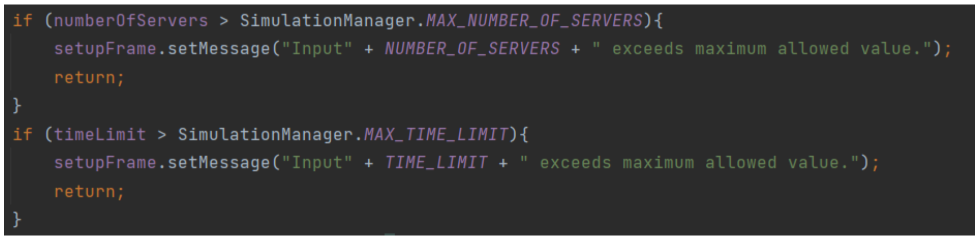
```java
@Override
public void actionPerformed(ActionEvent e) {

    SimulationStatus status = simulationManager.getSimulationStatus();

    if (e.getSource() == setupFrame.getValidateAndStartButton()){

        switch (status) {
            case WAITING_FOR_INFO ->      //not validated yet
                    validationRoutine();
            case WAITING_FOR_START -> {     //start not started yet
                simulationManager.setSimulationStatus(RUNNING);
                setupFrame.setMessage("Simulation is running.");
                setupFrame.setValidateAndStartButton("Halt");
                setupFrame.setHelpAndHaltButton("Help");
                simulationManager.startSimulation(numberOfClients, numberOfServers, timeLim
            }
            case RUNNING -> {               //pressed halt
                simulationManager.setSimulationStatus(WAITING_FOR_INFO);
                setupFrame.setMessage("Simulation halted.");
                setupFrame.setValidateAndStartButton("Validate");
                setupFrame.setHelpAndHaltButton("Help");
            }
        }
    }

    if (e.getSource() == setupFrame.getHelpAndHaltButton()){
        setupFrame.openHelp();
    }
}
```

Snippet from Controller Class

You may observe that in the previous switch, I use some status field as filter, and that the switch case has no default branch. This is because I thought that making the enumeration type *TextFieldNames*, holding every possible type of text field is a good practice. This may be true, since it removes the possibility of wrong inputs, and makes the code more flexible, easier to change, but maybe I'm just overthinking as usual. In any case, through this I learned and used a new thing. To print the source of error (which input is incorrect), I needed a nice form of *TextFieldNames*. I learned that this can be easily done through overriding the *toString* method for every possible value for the given enumeration class.



```java
NUMBER_OF_SERVERS{
    @Override
    public String toString() {
        return "number of servers";
    }
},
TIME_LIMIT{
    @Override
    public String toString() {
        return "time limit";
    }
},
```



```java
if (numberOfServers > SimulationManager.MAX_NUMBER_OF_SERVERS){
    setupFrame.setMessage("Input" + NUMBER_OF_SERVERS + " exceeds maximum allowed value.");
    return;
}
if (timeLimit > SimulationManager.MAX_TIME_LIMIT){
    setupFrame.setMessage("Input" + TIME_LIMIT + " exceeds maximum allowed value.");
    return;
}
```

Overriding toSring in enum and its usage

c. Classes of package *business_logic*

This package and its classes are by far the most complicated ones from the project. They went through a lot of modifications until they reached their current form.

To begin, there is an interface called *Strategy* with a single method: *addTask.* There are two ways (in my app) in which customers may choose which queue to join. One is the "primitive" method of looking at the number of people in each queue and choosing the shortest one. This approach is called *SHORTEST_QUEUE*, as defined in the enumeration *SelesctionPolicy*. The other approach is a smarter one, more specifically, to look at the time needed to complete each task in the queue, and choose the queue which, in current setup, at the moment of arrival, would finish the fastest. This is called *SHORTEST_TIME*. As you may have guessed, the implementation for these two strategies will be done by implementing the *addTask* method in two different, concrete classes. They are called *ConcreteStrategyQueue* and *ConcreteStrategyTime*.

Even if the two classes may sound very different, in practice they are 80% identical, if not more. The only difference is what kind of "minimum queue" are we waiting for.

A special case I stumbled upon and took a lot of time to figure out is the situation in which every slot is occupied. I did not think of this at the beginning, and because of it some of my tests failed. But after discovering it, it was not so hard to fix. Also, the averages and the peak hour details were added very late in the project, so it may feel like it's not the best way they are implemented, but hey, it somewhat works.

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

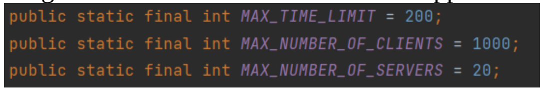
Rigorous Mathematical Background

The only equation needed to compute such things is the well-know formula of simple average: add each value and divide it by the number of values.

Class *Scheduler* has a confusing name since it does a little more than just scheduling tasks. Its main purpose indeed is to dispatch tasks, but it also handles other server-specific operations: checking if servers are empty (used to see if simulation ends early), returns the servers through a getter, updates/ computes the averages, and checks if new peak hour is reached. Note that, if the same number of maximum clients is reached multiple times, the first such time is considered peak hour.

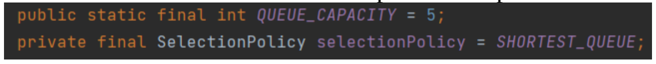
Last, but not least, the *SimulationManager* class is the home of all kinds of magic. This is where the simulation is performed. I will start describing it by first presenting its fields. They can be categorized as follows:

- Programmer-defined limitations of the application



```
public static final int MAX_TIME_LIMIT = 200;
public static final int MAX_NUMBER_OF_CLIENTS = 1000;
public static final int MAX_NUMBER_OF_SERVERS = 20;
```

- Hard-coded values of metrics not specified in problem statement

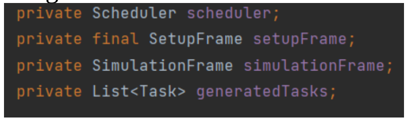


```
public static final int QUEUE_CAPACITY = 5;
private final SelectionPolicy selectionPolicy = SHORTEST_QUEUE;
```

- Simulation status

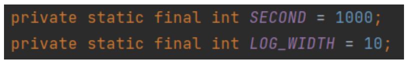
- Data read from GUI

- Usage of other classes



```
private Scheduler scheduler;
private final SetupFrame setupFrame;
private SimulationFrame simulationFrame;
private List<Task> generatedTasks;
```

- Other constants for customization

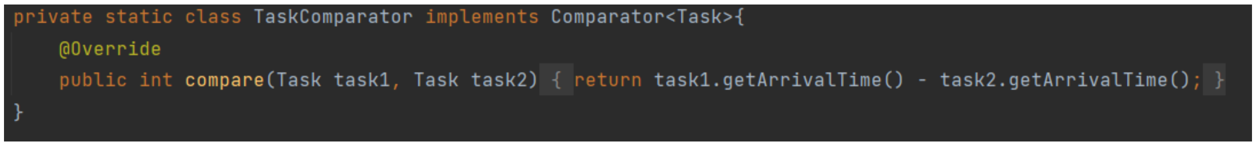


```
private static final int SECOND = 1000;
private static final int LOG_WIDTH = 10;
```

- Post-simulation data

The constructor sets the applcation in *WAITING_FOR_INFO* state, and initializes some of the fields. The simulation itself is started through calling this class' *startSimulation* method. This initializes every other field, calls the *generateRandomTasks* method, and starts the thread.

The *generateRandomTasks* method does what its name suggests. Only thing I would like to highlight is another peak moment of creativity of mine: to sort the tasks I defined my own *Comparator* class, like a boss.



```
private static class TaskComparator implements Comparator<Task>{
    @Override
    public int compare(Task task1, Task task2) { return task1.getArrivalTime() - task2.getArrivalTime(); }
}
```

The run method is the home of the simulation. This is where the "one second" sleep happens. In each iteration, the data is processed, the log is written, the GUI is updated/ redrawn, and the servers are notified, so each task's waiting time can decrement. One final thing to brag about is that logging is done through either *printLogLine* or *printLogLineWithoutNewLine* methods, and in this way by commenting out two lines we can toggle if we want to see the log in the console or not, and exceptions are handled locally here.

5. Results

The testing of this application was done by hand, during each step of implementation. When the application's functionality reached its "peak", I ran the tests given in the problem statement. The app seems to be working correctly. I know that "seems" is not enough. As Edsger Dijkstra would have said, *"Program testing can be used to show the presence of bugs, but never to show their absence!"*. But, for this project, I consider that there is no need to further deep into bugs since it is already a not-so-organized project created in hurry and with a lack of needed knowledge. Also, testing if correct output is given, when every generated simulation differs due to randomness from task creation looks very hard to me.

My main concern about the application is that it takes a lot of computational power to run tests on large values even if the updating GUI is disabled. This looks like a serious code smell to me, something went wrong somewhere in the code, but I could not find the source.

Even like this, without reaching the quality which would make me totally pleased, the project does pass all the requirements of the problem. And to finish this section of with a high note, let's not forget to appreciate the graphical user interface's beauty.

6. Conclusions

In conclusion, I have to say that in the given context, this project does stand its place. It did not teach me as much as the previous one, and it was less fun to implement it, but I did wander around, as usual, and I did find new things, even if they are not related to threads. I do understand the importance of threads, and I do hope that someday I will be able to master them, and use them properly, if needed.

As further development, I do not wish to continue this application's development, but there is, of course, a lot of space for its advancement. It could be optimized, it could be organized better, it could be documented better, with more comments. But it (kind of) does the job.

7. Bibliography

Course materials

https://dsrl.eu/courses/pt/materials/PT2021-2022_Assignment_2.pdf
https://dsrl.eu/courses/pt/materials/A2_Support_Presentation.pdf
https://dsrl.eu/courses/pt/materials/PT2021-2022_Documentation_Template.doc

For drawing

https://app.diagrams.net/

YouTube tutorials

https://www.youtube.com/watch?v=qwhfN8NwW_E
https://www.youtube.com/watch?v=ScUJx4aWRi0&t=183s
https://www.youtube.com/watch?v=gx_YUORX5vk
https://www.youtube.com/watch?v=wm1O8EE0X8k&t=79s
https://www.youtube.com/watch?v=orP1nBes9bE&t=316s
https://www.youtube.com/watch?v=JqZ3vAW_8ss&t=3s
https://www.youtube.com/watch?v=r_MbozD32eo
https://www.youtube.com/watch?v=V2hC-g6FoGc

Other links

https://www.jetbrains.com/help/idea/class-diagram.html#analyze_class

https://mkyong.com/swing/java-swing-how-to-make-a-simple-dialog/
https://docs.oracle.com/javase/7/docs/api/javax/swing/UIManager.html
https://www.canva.com/colors/color-palettes/rosy-flamingo/
https://www.programiz.com/java-programming/enum-string
https://www.geeksforgeeks.org/formatted-output-in-java/
https://www.baeldung.com/java-generating-random-numbers-in-range
https://www.inspiringquotes.us/author/8692-edsger-dijkstra