

STORE DATABASE MANAGEMENT APPLICATION

(ASSIGNMENT 3)

Student Name: Cristea Tudor

Group: 30423

CONTENT

1. ASSIGNMENT OBJECTIVE ………………………………………………………... 3
2. PROBLEM ANALYSIS, MODELING, SCENARIOS, USE CASES …………...….. 3
3. DESIGN AND IMPLEMENTATION ……………………………………………..… 4
4. RESULTS ……………………………………………………………………………. 7
5. CONCLUSIONS …………………………………………………………………….. 8
6. BIBLIOGRAPHY ……………………………………………………………….…… 8
7. ASSIGNMENT OBJECTIVE

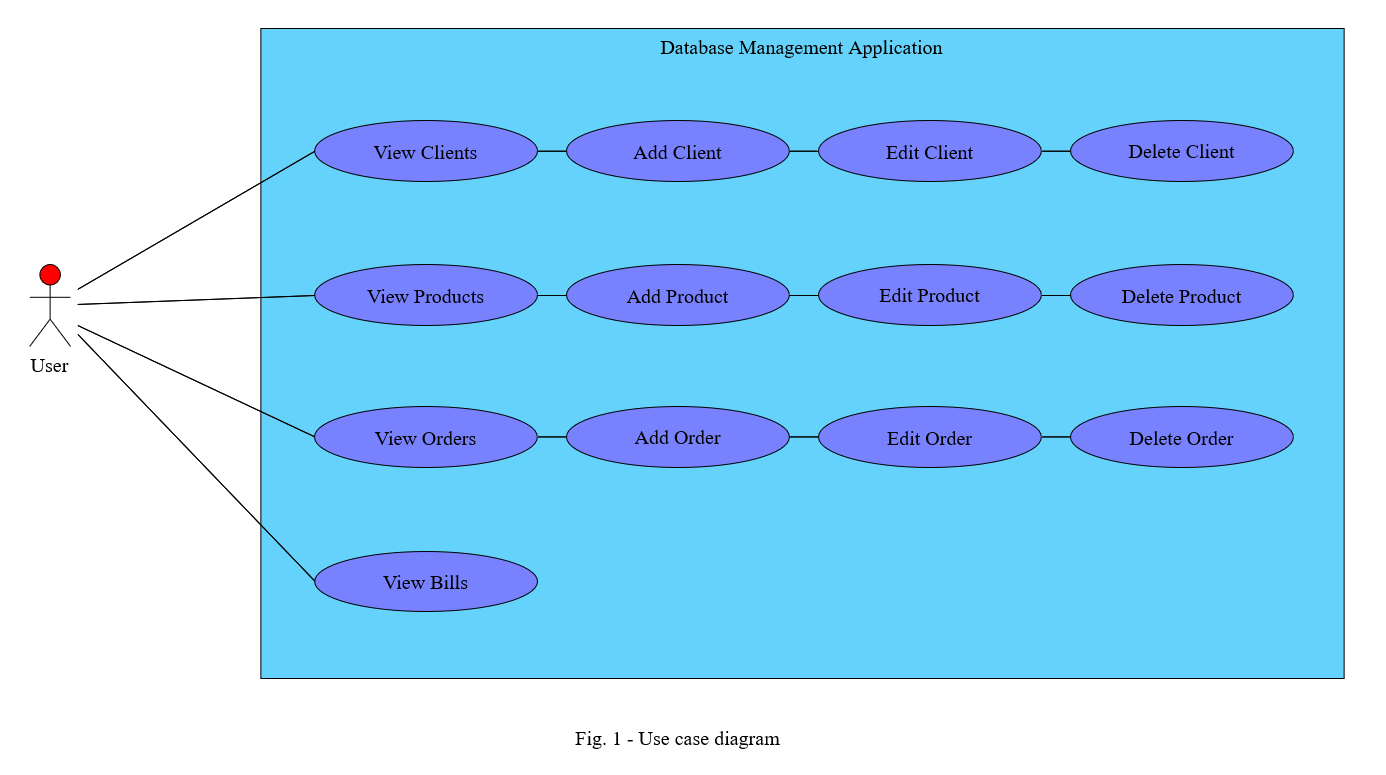
The objective of this assignment is to design and implement a store database management application which gives the user the possibility to manipulate, partially or completely, the tables found in the database. In this sense, the user can insert, update, read or delete entries from certain tables.

The sub-objectives are the following:

* + - * Design and implement several model classes, the objects of which will resemble precisely the structure of their counterpart entities from the tables that exist in the database
* Design and implement classes which will ensure the connection and manipulation of the database; these classes will use reflection in order to accomplish their task
* Design and implement the Graphical User Interface (GUI)
* Design and implement the controller that links the models and “database connectors” to the views (the interface)
* Testing and Debugging possible errors and/or edge-cases

1. PROBLEM ANALYSIS, MODELING, SCENARIOS, USE CASES

The use case diagram is depicted in Figure 1.



**Use case #1: View Clients/Products/Orders/Bills**

**Primary actor: User**

**Success Scenario Steps:**

1. The user presses the button corresponding to the table of objects he wants to see/analyze.
2. The corresponding view (window) opens and the table is shown to the user.

**Alternative Scenario:**

1. The corresponding window does not open
   * + The application informs the user through a pop-up about a potential error (bad database connection)
     + The scenario returns to step 1

**Use case #2: Add Client/Product/Order**

**Primary actor: User**

**Success Scenario Steps:**

1. The user presses the add client/product/order button and a second view (window) opens up
2. The user enters the values of the fields of a new client/product/order in the designated text fields (for an order, the client and product need to be selected using combo boxes)
3. The user presses the “Add” button
4. The data is converted internally from Strings to their actual type and inserted as a new entry into the Client/Product/Order table from the database

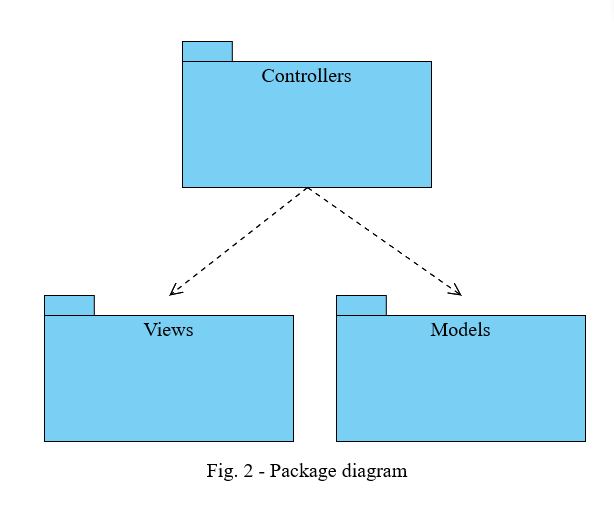
**Alternative Scenarios:**

1. The user enters anything other than a positive integer (for product/order quantity), a positive real number (for product price) or an appropriate String (for client phone number, client/product name)
   * + The application informs the user that the text he entered is invalid
     + The client/product/order cannot be added to the database
     + The scenario returns to step 2
2. The user leaves one of the text fields empty

* The application informs the user that there is at least one empty text field
  + - The client/product/order cannot be added to the database
* The scenario returns to step 2

1. DESIGN AND IMPLEMENTATION

The package diagram is depicted in Figure 2. I decided to group my classes based on their functionality, and thus, adopted the Layered Architecture (Business Logic Layer, Model Layer, Presentation Layer and Data Access Layer) pattern for developing this application. This way, the application has a high coherence.

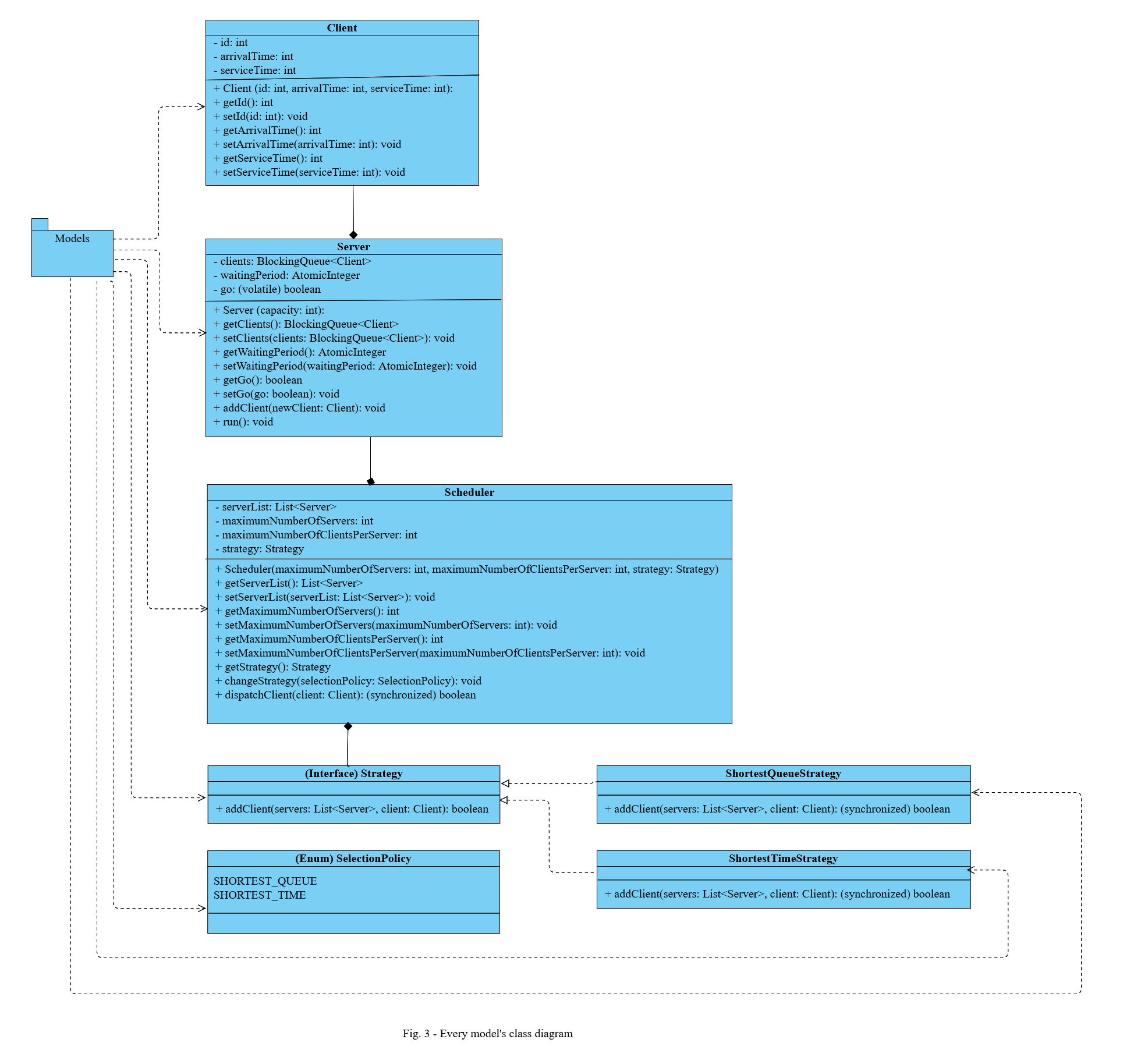


The *Client* class diagram is depicted in Figure 3. Each client will have a unique id, an arrival time (this is supposed to replicate the scenario in which clients request a server in real-time, at various time stamps) and a service time, which represents the time it takes for the server to fulfill the client’s request.

The *Server* class diagram is depicted in Figure 3. Each server contains a *BlockingQueue* of clients, as well as a waiting period which will be declared as an *AtomicInteger* in order to ensure the thread-safety of the *run()* method, where this waiting period will be primarily used. The *run()* method, for each thread of the *Server* class will run as long as the *volatile boolean* variable *go* is set to true. The first client of the queue is taken and the thread will “sleep” for a number of seconds equal to the amount of the service time of that particular client. After the thread “wakes up” again, the client will be removed from the queue and the waiting period will be decreased by an amount equal to the service time of the removed client.

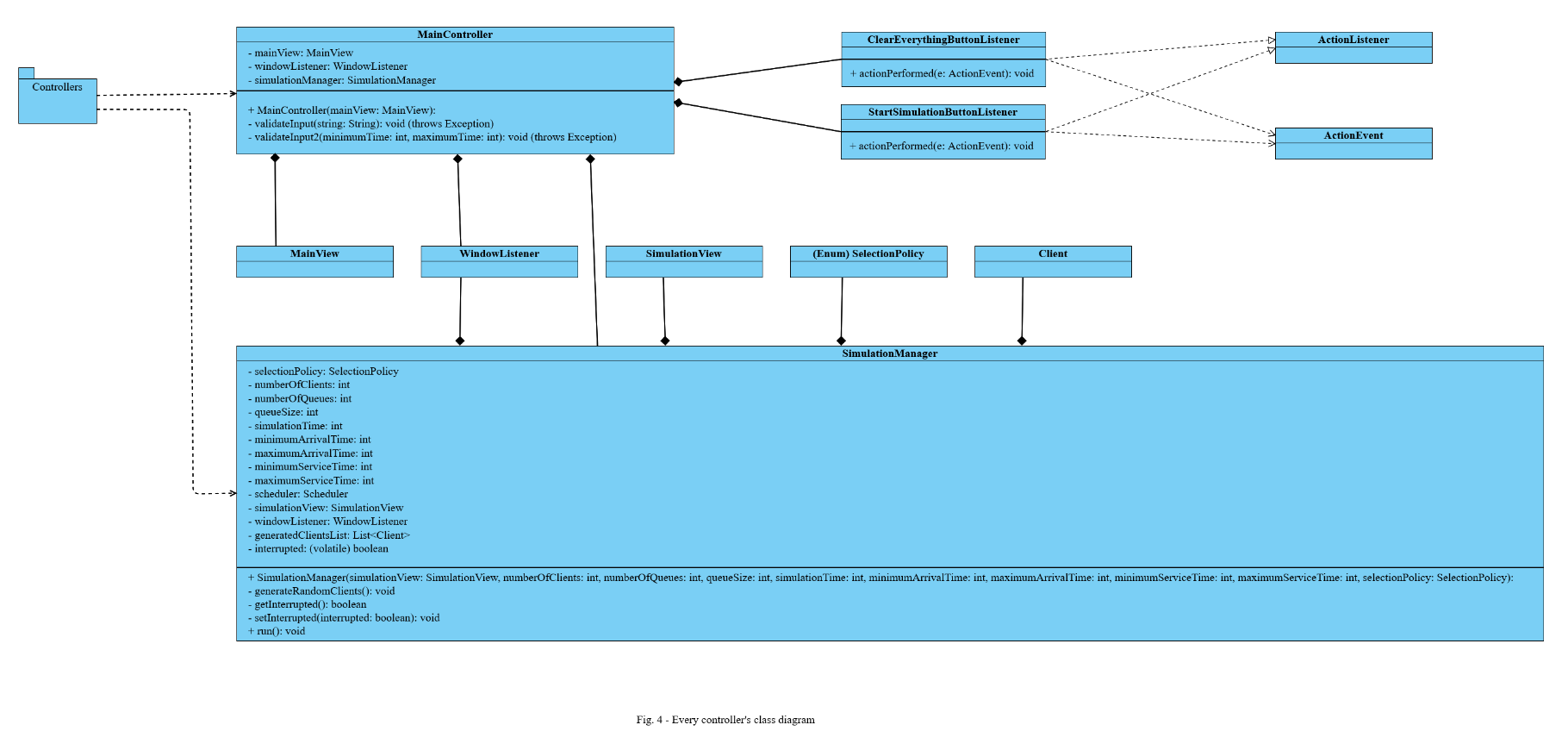
The *Scheduler* class diagram is depicted in Figure 3. The scheduler is responsible with creating and starting the servers, operations which are performed in the constructor of this class. The *changeStrategy()* method takes the enum as a parameter and changes the strategy accordingly. Therefore, the *dispatchClient()* method adds the client to one of the servers depending on the strategy type.

The *Strategy* interface is depicted in Figure 3. It is a functional interface, since it only contains the *addClient()* abstract method, is implemented by two classes: the *ShortestQueueStrategy* class and the *ShortestTimeStrategy* class. Each of them, implement the *addClient()* method according to their name. The former will place the client into the server with the least number of clients and the latter will place the client into the server with the smallest waiting period.

The *SelectionPolicy* enum is depicted in Figure 3. It is only used to help with the selection of the type of strategy that needs to be applied when assigning clients to servers.

The *MainController* class diagram is depicted in Figure 4. The *validateInput()* method is responsible with validating the strings of text that are entered by the user in the text fields of the GUI. If these strings are either empty or do not represent strictly positive numbers, then the user will be notified of their mistakes and the simulation will not start. The *validateInput2()* method is responsible with verifying the fact that the minimum times are smaller or equal to their respective maximum times.

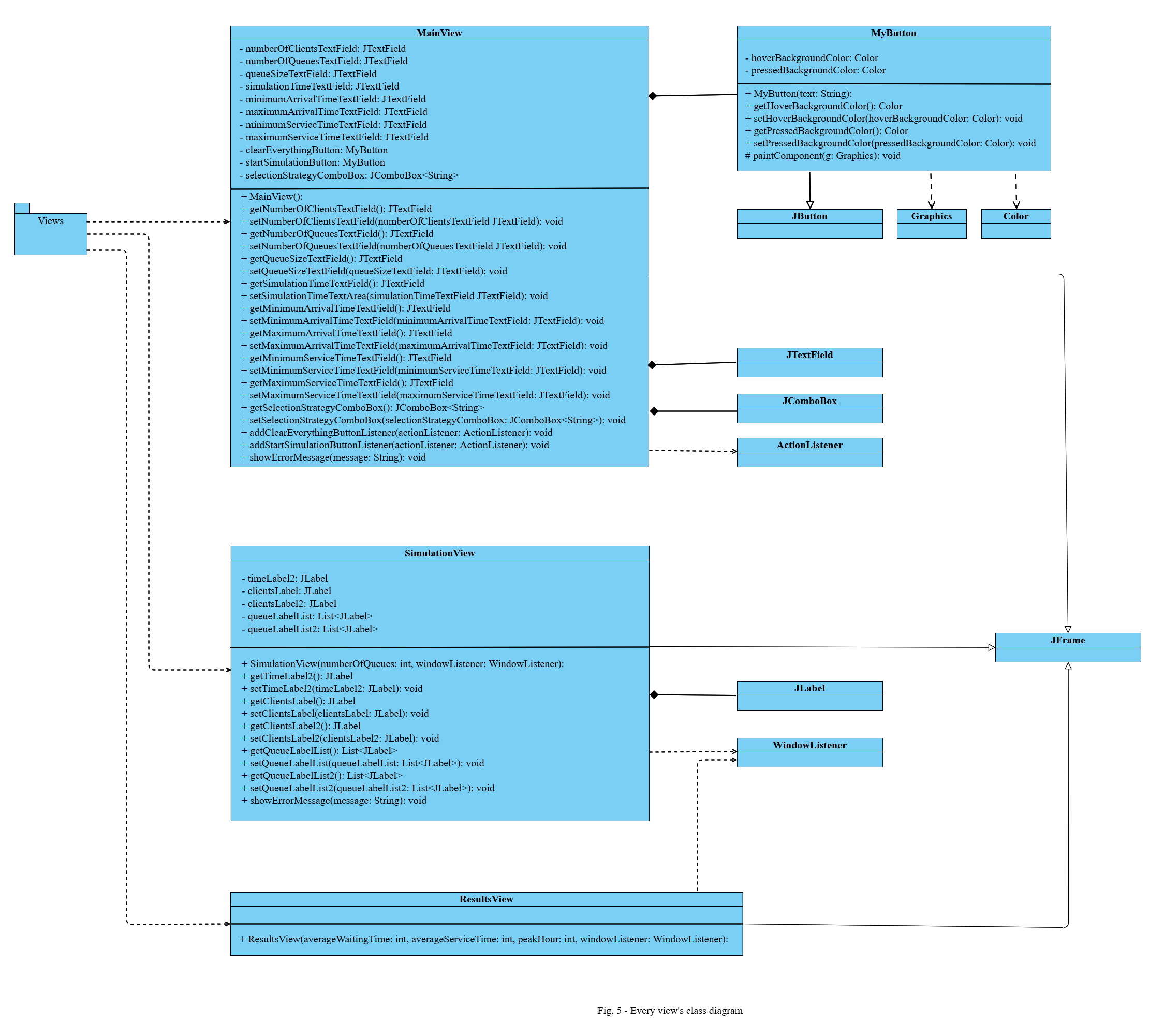
The *SimulationManager* class diagram is depicted in Figure 4. This represents the main thread of the application. The *generateRandomClients()* method, as the name suggests, will generate clients with random arrival times and service times, both of which are situated in a certain intervals. The list of generated clients is then sorted by their arrival times, since all of the them are created at the beginning of the simulation. Thus, they will be assigned one of the servers at the appropriate moment of the simulation, or later if all of the servers are full. However, even if they are assigned a certain server, each client will still wait its turn, until the server can process them (they reach the front of the queue). The *run()* method is responsible with many things that happen throughout the simulation. Firstly, it creates the log text file in which each step of the simulation as well as the results of the simulation are saved for later use. Secondly, it is responsible with the entire simulation, meaning that it dispatches clients to servers, displays the evolution of the simulation time, server queues and waiting queue, in real time, through the GUI. Additionally, it keeps track of all queues (including the waiting one) and stops the simulation prematurely if there are no more clients in any queue. At the end of the simulation, it stops all servers and displays, using a separate window, the results of the simulation including the average waiting time, average service time and peak hour.



The *MainView* class diagram is depicted in Figure 5. It contains text fields and a combo box for the setup elements of the simulation, as well as two buttons: one for clearing every text field and another for launching the simulation. Supposing that the data entered is valid, a second window appears in which the real-time evolution of the queues can be observed, while the initial window will become inactive (non-interactable).

The *SimulationView* class diagram is depicted in Figure 5. It contains real-time information about certain aspects of the simulation including the current time, the list of waiting clients and the lists of clients for each server. The simulation will stop either when the current time will reach the maximum simulation time (which was chosen during the setup phase of the application) or when all queues (including the waiting queue) are empty. After the simulation is done, a third window will appear containing the results of the simulation. If, however, the user decides to stop the simulation abruptly by closing the simulation window, then the application will not compute and display any results, and all servers including the simulation manager will be halted. The user will have to wait for about a second until he can modify the parameters of the simulation and start a new one.

The *ResultsView* class diagram is depicted in Figure 5. It includes the concrete values of the average waiting time, average service time and peak hour (the busiest hour). This data might be useful to the user. Once the results window and the simulation window are both closed, the user can, once again, modify the parameters of the simulation and start a new one. Here is how I interpreted and computed these results:

* **average waiting time**: the sum of the waiting times of the clients that were in the “waiting queue” at some point during the simulation divided by their number added with the sum of the waiting times of the clients that have been assigned a server but have not been served yet divided by the number of servers; the latter sum is computed for every unit of time (in this case, every second) of the simulation and thus, at the end, these averages are summed up and divided by the total time of the simulation
* **average service time**: the sum of the service times of the clients that were served completely (their service time reached zero) or partially (the simulation ended while they were being served) divided by the total number of clients of the simulation
* **peak hour**: the unit of time (in this case, the second) in which the number of clients in every server, as well as in the waiting queue was maximum

1. RESULTS

The application was tested by manipulating the database in several different ways. Every operation performed on each table was tested individually as well as together with other operations which might influence the obtained result. No issues were detected during the testing. However, unless the application is properly and fully used in a real manner, by a company for example, further maintenance cannot be performed.

1. CONCLUSIONS

In conclusion, a database management application for a store or a warehouse is really useful because it becomes marginally easier to manage the accountability part of the business. Although data is not saved in a safer manner than if it was written on paper, it is easier

Working on this assignment, personally I have learned how to implement thread-safe classes and thread-safe methods (using the “*synchronized*” keyword and the “*volatile*” keyword). Additionally, I have learned how to work with threads (implement their *run()* method, instantiate them, start them, make them wait) and how to implement an application from a concurrent point of view as opposed to a sequential point of view (which was the primary case up until now).

Obviously, the application can be developed further with additional features, such as:

* letting the user choose a different size for each queue
* implementing a functionality such that the clients are randomly generated and added to the simulation in real-time (to make it more realistic and unpredictable)
* making the application able to add or remove queues based on the workload of the simulation

1. BIBLIOGRAPHY

* [online.visual-paradigm.com](https://online.visual-paradigm.com) (for diagrams)
* *Fundamental Programming Techniques* course lecture slides and laboratory slides
* [stackoverflow.com](https://stackoverflow.com)
* [chat.openai.com/chat](https://chat.openai.com/chat)
* [geeksforgeeks.org/](http://www.geeksforgeeks.org/)
* [youtube.com](https://www.youtube.com) (for tutorials)