PT

Queues

Homework 2

Chirodea Mihai – Cristian

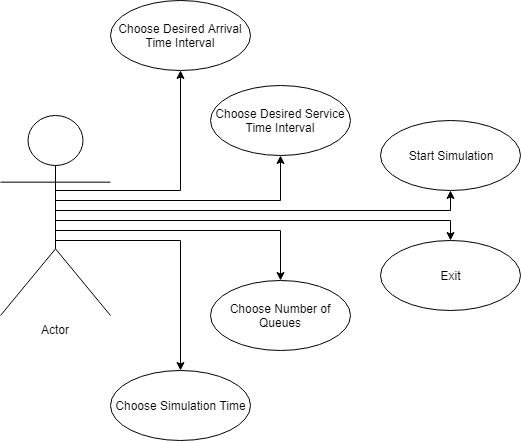
Group 30424

1. **Project Objective**

* The main objective of this project was to create a Java application capable of simulating a series of queues in which a series of clients come, enter queues, get served and then leave the queues. The program also required an easy to use interface, created using Java’s Swing package, for the user to interact with.
* The project was realized in Eclipse and it’s able to perform the operations listed above. In order for the program to give correct results during simulation, a few steps have to be followed:
  + - **First Step**: The user has to introduce the Min/Max in the first text field labeled accordingly, same for the 2-nd. Then, he needs to introduce the number of queues in the 3-rd text field, and lastly, in the 4-th text field he has to introduce the max simulation time which, when reached, the program stops.
    - **Second Step**: The user has to verify that the values introduced at step 1 have the following format: **NO** spaces between the values in the first 2 text fields, **NO** negative values, the program will crash, **IF** in the second field, a value smaller than 2 is introduced the program will default to 2, **ORDER** of the members matter, **LASTLY**, if a values greater than the max number of queues then the program defaults to 7.
    - **Third Step:** After Step 1 and Step 2 are completed, the user will now be presented with a new window containing a graphical representation of the current queue evolution.
* If the steps above have been followed correctly, then the program should give the correct result during simulation. 4

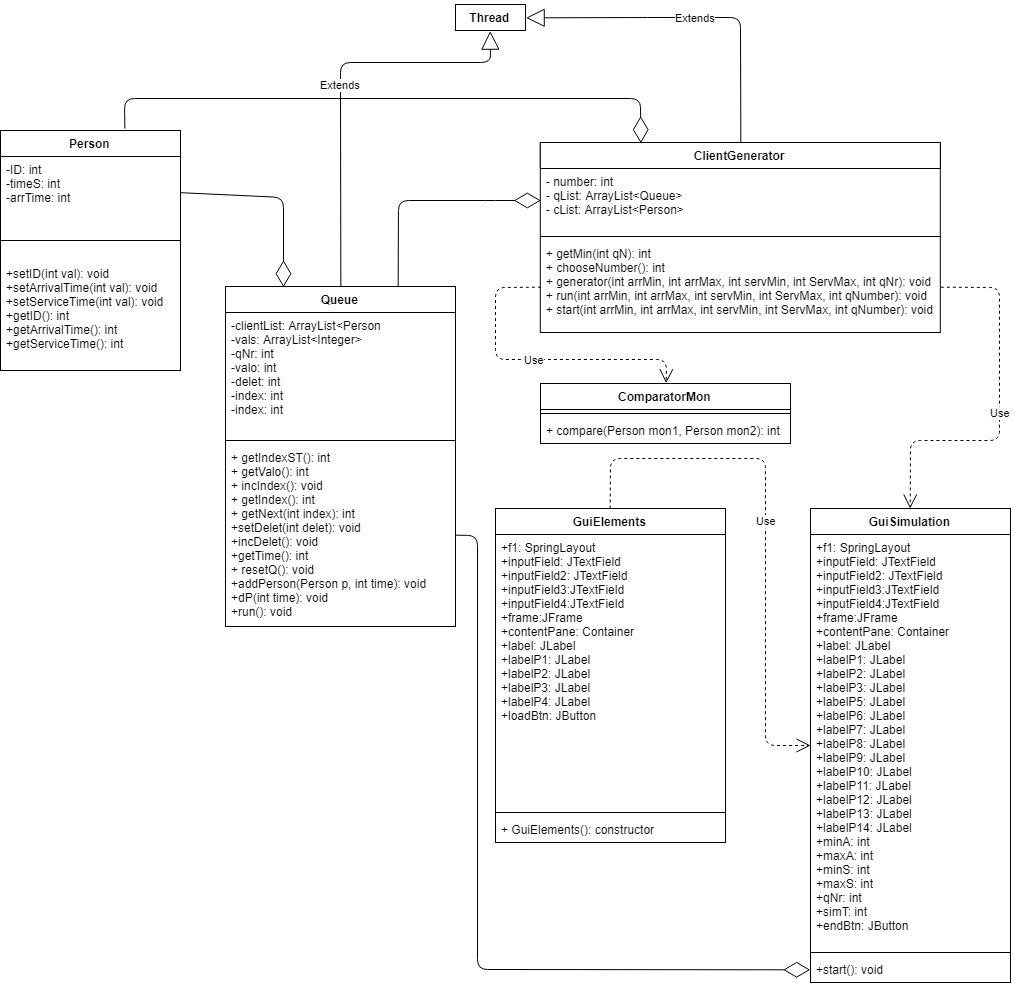
1. **Problem Analysis**

* **Functional Requirements:**
  + In order to create a program that satisfies all the needs for the operations to occur, it’s important to know how a Thread works, to be more precise, how threads synchronize with each other. Understanding that, we can decompose the problem into data structures and perform the needed operation by creating methods and classes to deal with said data structure.
  + So, first step is figuring what sort of structure to use, in order to store the clients. As the specification says, the simulation takes out and adds new clients when needed, so what the program has to do, is to comply and implement a structure that can do this efficiently.
  + As the users are generated with random values, the arrival times are unknown, so I implemented a comparator that sorts the list when a new client is introduced.
* **Use Case:**



1. **Design**

* **UML Diagram:**

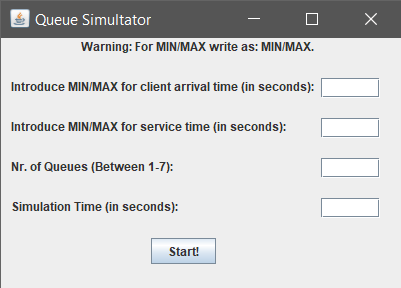
****

* **Data Structures:**
  + In order to best represent the clients, I chose to create a Class, named Person that contains the description of each individual, more precise (Arrival Time, Service Time and ID).
  + To best represent the 2 windows that the user will interact with I created 2 separate interfaces, one containing the first part where the user introduces the data and chooses when to start and the other containing the second part that displays the actual simulation.
* **Design Decisions:**
  + A big decision I had to make was to either use an ArrayList or a HashSet to represent the clients, in the end I settled with an ArrayList as it was easier to implement.
  + This decision made a huge impact on my program as it was a lot easier to do the client operations and it took a lot less lines of code to get it to work. But as a downside I had to pay more attention to the indexes, so it would not go out of bounds.
  + Another decision I made was to use one text field as input for Min/Max. In my mind, this would have been more user friendly than simply making the user introduce the values separately.
  + And, again, this decision had a significant impact on the program, as I had to extract the values from the text, which in turn placed a few constraints on the user (Not being able to write with spaces and having to put min and max where they need to be).
  + Lastly, I made the decision to use as a log, the console, as it seemed easier to implement that using a file.

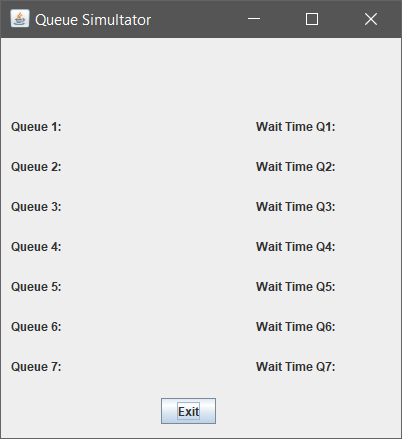
1. **Implementation**

* **Packages:**
  + In order to organize the classes, I split them into two packages:
    - **control:** Contains the Person, ClientGenerator, Queue, ComparatorMon.
    - **gui:** Contains the classes for the 2 intefaces.
* **Clients Class:**
  + Its name is Person, it contains 3 definitions of integers, named ArrivalTime, ServiceTime and ID. ArrivalTime stores the time when the client should be displayed and added to the queue, ServiceTime stores the time when the client should finish and get taken out of the queue and ID, stores the identification of the client so that the user could distinguish them.
  + This class contains the following methods:
    - **The Constructor:** It initializes the id, arrivalTime and serviceTime and grade.
    - **setArrivalTime(int val):** It’s an int method that sets the arrival time of the client by receiving an int value.
    - **setServiceTime(int val):** It’s an int method that sets the service time of the client by receiving an int value.
    - **getArrivalTime():** It’s a void method that returns the arrival time of the client .
    - **getServiceTime():** It’s a void method that returns the service time of the client.
    - **getID():** It’s a void method that returns the id of the client.
    - **setID(int val):** It’s an int method that sets the ID of the client by receiving an int value.

* **Client Generator Class:**
  + Its name is ClientGenerator, it contains 2 structures. First one is named qList/cList, and it’s an ArrayList and the second one is named comparatorMon and it’s of ComparatorMon type.
  + cList stores the list of clients in ascending order, this is done by sorting using comparatorMon object.
  + qList stores the number of active queues.
  + This Class contains the following methods:
    - **The Constuctor:** Initializes cList, qList and the other values.
    - **generator():** generates the clients.
    - **run():** custom method to start the thread.
    - **start():** custom method to start the thread.
    - **chooseNumber():** chooses the queue in which the client wil be added.
    - **getMin():** returns an integer that is the min waiting time of the queues.
* **Comparator Class:**
  + Named ComparatorMon, it’s an auxiliary class that is used by ClientGenerator to sort its cList. It implements the Comparator<Person> interface.
  + It has only one method:
    - **compare(Person mon1, Person mon2):** Compares the arrival times of the 2 clients, and returns which one is bigger.
* **Queues Class:**
  + Named Queues, it serves as group of methods and data structures that other classes rely on. It has methods that helps in displaying the client on the graphical user interface, another that removes the client from the graphical user interface and so on.
  + It has the following methods:
    - **The constructor():** The constructor of the class that initializes the variables such as cList, vals, qNumber and valo.
    - **getIndexST():** returns the index that is used in going through the list of clients in the queue.
    - **getValo():** returns the waiting time of the queue that is displayed later on the graphical user interface.
    - **incIndex():** increments the index that goes through serviceTime of the clients.
    - **getIndex():** returns the index that goes through the serviceTime.
    - **getNext():** returns the value of the next serviceTime from the vals list.
    - **setDelet():** sets the value for delet that is used in checking if it’s time to delete a client from the graphical user interface.
    - **getTime():** gets the value for the waiting time that is used to display the waiting time.
    - **resetQ():** resets the waiting time.
    - **addPerson():** Adds a person to the List in the queue, and displays it on the Grapical User Interface.
    - **dP():** Deletes a person from the queue and graphical user interface and decreases the service time.
    - **run():** Used to start the thread.
* **GUI Class 1:**
  + Named GuiElements, it contains the definitions of Java.Swing elements such as buttons, labels, textFields, containers and frames.
  + This class is the one responsible for creating the graphical user interface with which the user interacts.
  + As methods it has the constructor that initializes the interface and its elements and that adds listeners to the buttons, if they exist.
  + It uses Spring Layout which, because of it’s constraints, can be modified however the programmer wants and can arrange elements in a way that other layouts have a hard time copying.
  + It has 1 button (Start) that, after the user introduces the values and presses the button, opens up the next interface, in which the simulation starts running.
  + This class is also where the program is executed from.
  + The Interface for this class looks as follows:

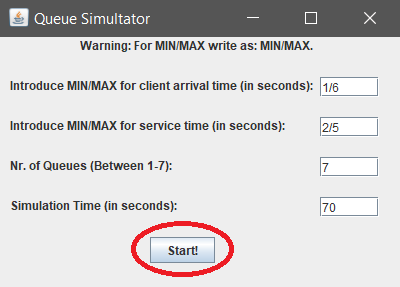
****

* **The Listener Classes:**
  + Each of those classes is used as a listener for each of the buttons, and when a button is pressed, the class corresponding to each button is called, executing the code in them.
  + As example, when pressing the start button, the program jumps to the StartButtonListener class, which in turn calls the second interface. But before that, it takes the values from text fields and converts them into integers checking for the restrictions applied.
* **GUI Class 2:**
  + Named GuiNew, it’s the interface where the user sees the result and gets to choose what to do with it.
  + It contains definitions of Java.Swing elements (JButton, JLabel, JFrame, Container and so on).
  + As methods it contains the constructor that initializes the interface and add the listeners to all the buttons.
  + It also contains the method start() which implements the Swing Worker class that starts Client Generator.
  + It contains 1 buttons (Exit) which has the following functions:
    - **Exit Button:** closes the first and second interface by using dispose().
    - **Caution! Closing any window with the ‘x’ button will close the program, not the window.**
  + The interface for this class looks as follows:

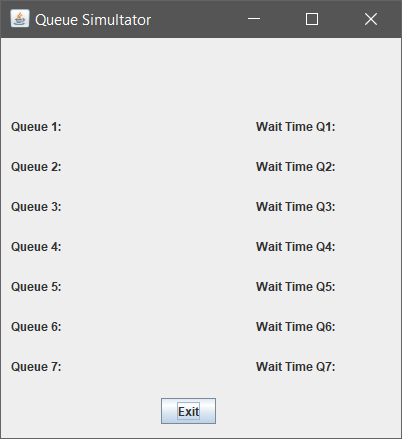
****

1. **Testing & Results**

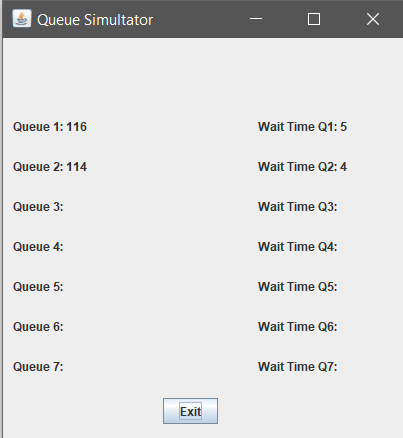
* In order to prove the correctness of the program, I have chosen the following cases and results( Note that being a simulation, results may differ from each iteration as the clients are randomly generated).
  + **Test Input:**
    - * **Arrival Time Min/Max: 1/6**
      * **Service Time Min/Max: 2/5**
      * **Number of Queues: 7**
      * **Simulation Time: 70**
  + **Simulation Results:**



* **Simulation Starts:** At first we are presented with the second interface, which is blank.

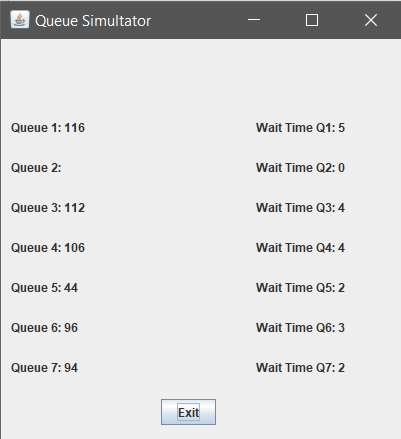


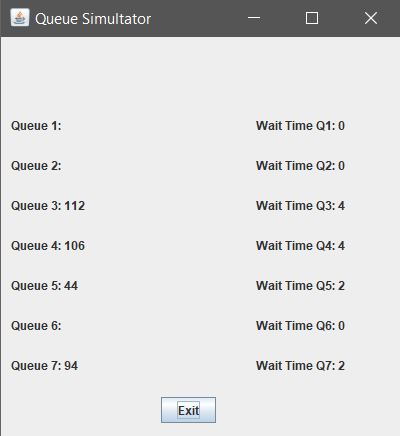
* **Next:** The queues start to be filled with clients, each client going to the shortest queue.



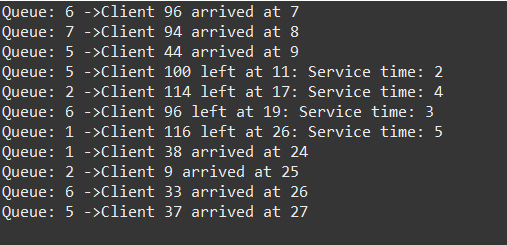
* **Next 2:** After the queues get filled, the clients start piling up by going to the shortest queue. This is pretty much the final step before the simulation reaches it’s end. (**Note:** this step is heavily dependent on the service time of each client)







* + - **Queue Log:** Finally, while the program runs and also when it ends, we can check the log to see exactly at what time a client entered or left.



1. **Conclusions**

* To sum it all up, I had to create a program that simulated a number of queues and how clients choose the queue and get serviced. I did this using Threads and methods that helped in displaying the values in a somewhat real-time.
* **What I learned:**
  + I learned to create GUI using only code, not drag & drop like I used to do.
  + I better familiarized myself with Hash Sets and OOP use of Objects.
  + I improved on working with the GUI.
  + I learned to extract information from a line of text using the split method in java.
  + I learned to create and use Button Listeners.
  + I learned how to link multiple interfaces.
  + I learned how to use Threads.
  + I learned how a GUI updates(Hint: It’s a Thread)
  + I learned how to use Array Lists
* **Future improvements:**
  + Adding option to save results.
  + Displaying results in a different window.
  + Making the program more flexible to user input errors.
  + Removing user constraints on the input.
  + Adding Peak Time.
  + Adding average waiting time.
  + Fully synching the threads.

1. **Bibliography**

* <https://stackoverflow.com/questions/3481828/how-to-split-a-string-in-java>
* https://en.wikipedia.org/wiki/Thread\_(computing)
* <https://docs.oracle.com/javase/tutorial/uiswing/components/panel.html>
* <https://www.ntu.edu.sg/home/ehchua/programming/java/J4a_GUI.html>
* <https://docs.oracle.com/javase/tutorial/uiswing/layout/spring.html>
* <https://stackoverflow.com/questions/5993779/use-string-split-with-multiple-delimiters>
* <https://stackoverflow.com/questions/22212412/sort-a-set-in-reverse-order>
* <https://stackoverflow.com/questions/17370909/sorting-a-linkedhashset>