**QUEUES SIMULATOR**

1. Objectives
   1. Main Objective

Design and implement a simulation application aiming to analyze queuing based systems for

determining and minimizing clients’ waiting time. The application should simulate (by defining a simulation time Tsimulation) a series of N clients arriving for service, entering Q queues, waiting, being served and finally leaving the queues. The application tracks the time spent by every customer waiting in the queues and computes the average waiting time. Each client is added to the queue with minimum waiting time when its arrival time is equal to the simulation time.

* 1. Secondary Objectives
* Define a representation for a client. (Chapter 3)
* Design a thread class which will act as a queue. (Chapter 3, 4)
* Implement a strategy by which the clients are added to different queues in an efficient way. (Chapter 3)
* Design a scheduler that keeps track of its queues and manages the clients it receives.

(Chapter 3, 4)

* Design the simulator that will generate data to be tested on the queue mechanism.

(Chapter 3, 4)

* Implement the functionality of computing the average waiting time in the queues.

(Chapter 2, 3)

* Add the necessary features such that the simulation can be tracked by the user. (Chapter 4)

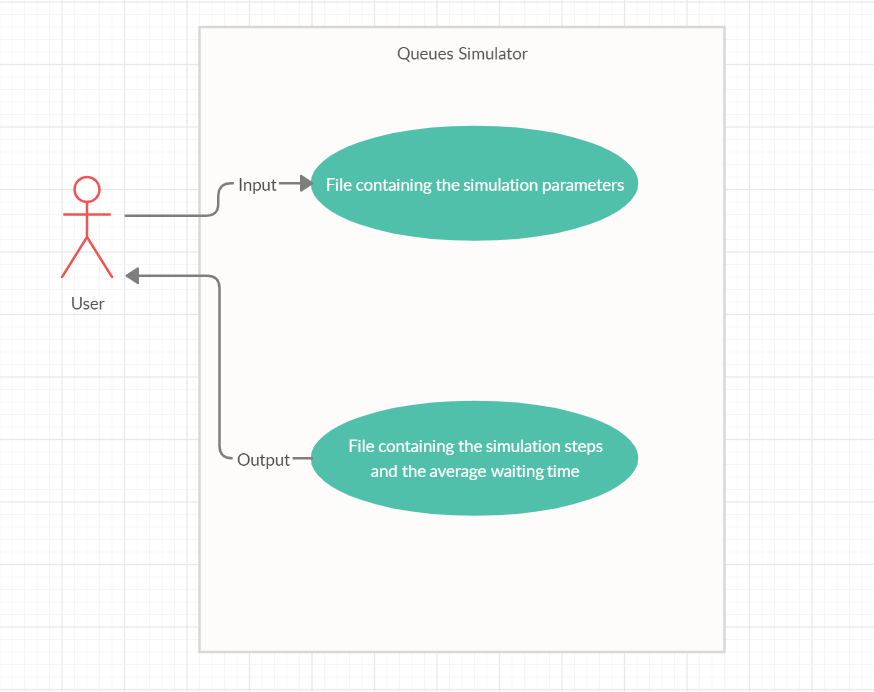
1. Task analysis, Modeling, Scenarios, Uses Cases

The application should receive as argument the name of the file that contains the simulation parameters. The file needs to have a specific format as shown below:

* 1. Number of clients (i.e. 4)
  2. Number of queues (i.e. 2)
  3. Max simulation time (i.e. 60)
  4. Minimum arrival time Maximum arrival time (i.e. 2 30)
  5. Minimum processing time Maximum processing time (i.e. 2 4)

🡪 each field is on a separate row and numbers are separated by a single space.

These parameters will define the simulation environment. The clients will be generated and their attributes will be set accordingly to the given parameters. The number of requested queues will also be created, each having the limit of Number of client / Number of queues + 1 clients. After the initialization is done, the simulation is started. The current simulation time is incremented by at every second until it reaches the limit specified in the input file. At every second the state of the simulation is written to an output file such that the user can keep track of how the clients are organize in the queues. If a queue has no clients waiting in it, then the queue should be closed and remain like this until the scheduler dispatches a client to it. The waiting time of a client is defined as the time period needed for him to reach the front position in the queue when he will be served. Thus, when a client enters a queue his waiting time is set to 0. At every second, if he is not in the front of the queue, its waiting time is incremented by one. If a client is added to a queue directly in the front position, the waiting time for this client will remain 0. At every second, all the clients that are in the front position in their queue (they are being processed) have their processing time decremented by one. The queues are processing clients one by one in order of arrival for a duration of time equal to their processing time in seconds. At the end of the simulation, the average waiting time of a client needs to be computed based on the data acquired during it. It is needed to be mentioned that only the clients that have been processed are taken into consideration because the clients that are still in the queues when the simulation is over have a partial waiting time and would not make sense counting them. Keeping these in mind, the formula for computing the average waiting time is given by a simple division between the sum of the waiting times of all the processed clients and their number.



* Use Case: Simulate queues
* Primary Actor: User
* Main Success Scenario:
  + The user provides the application the input file with the correct specified simulation parameters.
  + The application displays in the console the current time of the simulation.
  + When the simulation is done, the message “Done” is printed on the screen.
  + The user can check the generated output file to see the details of the simulation.
* Alternative Sequences:
  + The user does not provide an input file
    - The application will print in the console the usage information
  + The user provides a bad input file
    - The application will throw an error not being able to interpret the given simulation parameters.

1. Design

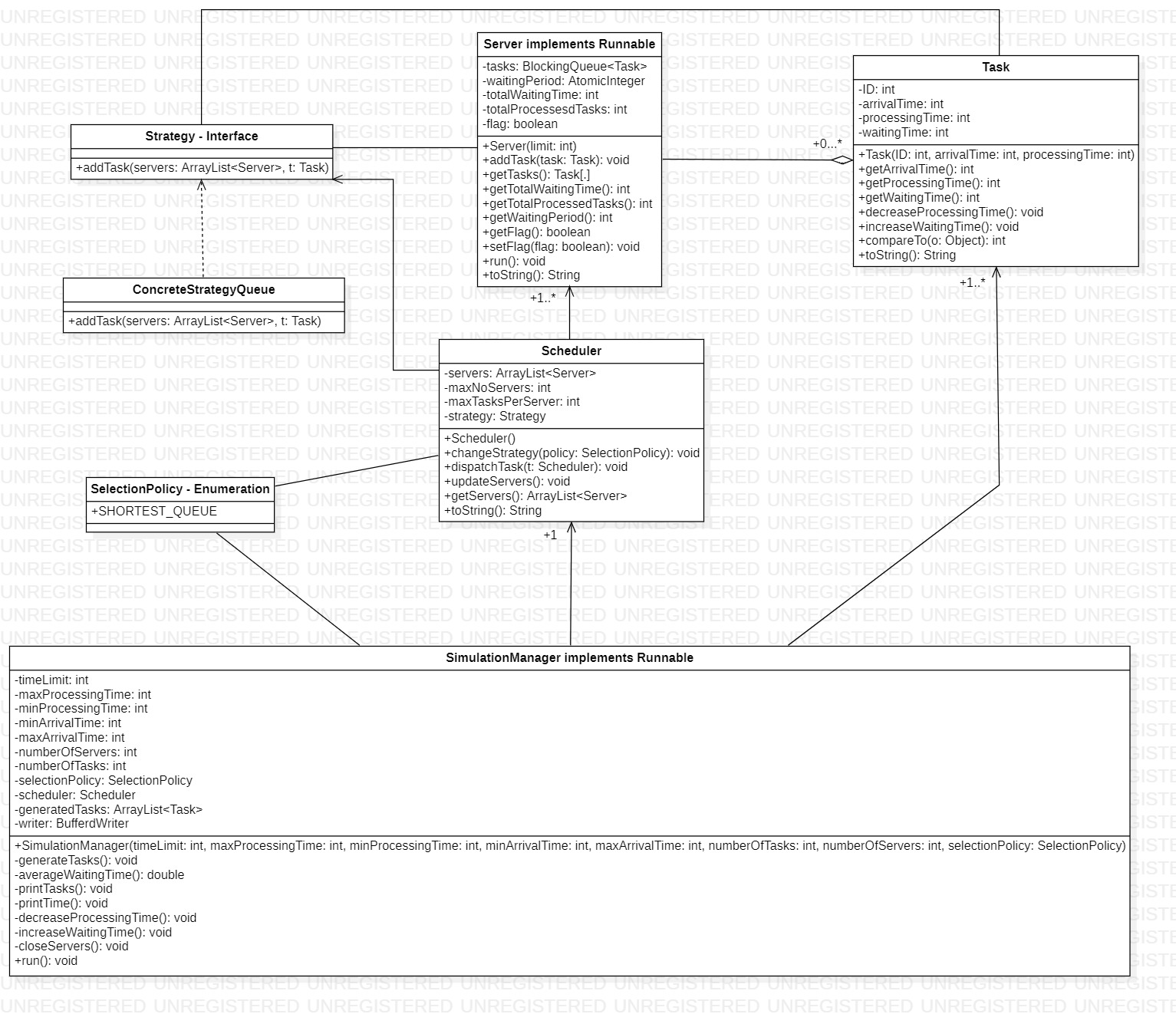
The first objective of the implementation process is to obtain objects based on which the simulation will take place. That being said, we start off by taking a look at what a queue should do and how. Since a queue needs something to be added to it we should define now the implementation of a client or in other word the task for our server which is the queue. The task is identified by an ID such that it will be easy to differentiate between them during the simulation. There are other three important attributes that should characterize a task: its arrival time (the timestamp when the task should be added to a server), its processing time (the time needed for the server to process the task) and the waiting time which will define the time period that a task waits in a queue before it is processed. Coming back to the main component of the application, the server should have a way of storing task that are waiting for it to processed them. The solution for this is the usage of an arrayBlockingQueue. The sum of all the task’s processing times from a server result in the waiting period that a new task has to wait if it is added to that server and this is stored as an atomicInteger. Two more attributes used to keep track of the number of clients processed by the server and the total waiting time of these clients. The last thing needed is a flag through which the scheduler can control the server activity.

Another thing that we need is a way to coordinate one or more servers and a set of tasks that should be distributed among these servers. This is achieve by the use of a scheduler that holds a list of N servers (N is given by the maxNoServers attribute) each with a limit of M tasks (M is given by noOfTasks attribute). One more important attribute is the strategy that is used to dispatch a given task to one of the servers controlled by the scheduler. The scheduler provides methods for adding a new task to one of its servers according to the chosen strategy and changing the strategy dynamically. An important method is the update one which turns on a server if it received tasks while it was closed.

The strategy patter is composed from its defining interface strategy on which are built all the classes that implement a specific strategy of adding a task to a server. An enumeration Strategy Policy is used to distinguish between all the implemented strategy and acts an identifier for selecting a specific strategy implementation in the scheduler class.

Finally, the only class that needs to be implemented in order for the application to have its specified functionality is the simulation environment. The simulation class will generate the data based on which the simulation will take place. The parameters for the simulation will be read from a file given as input and the results will be written in an output file by the running application. The output file will contain the steps of the simulation displaying the state of the tasks and servers at every second of the simulation time interval.

The relations between the classes and their details are illustrated in the below UML diagram.



1. Implementation

* Task Class

Represents a client that has to be processed in a queue. It is characterized by the four private int fields: ID, arrivalTime, processingTime and waitingTime. Along the speficic getters and setters two more methods are implemented for decrementing the processingTime and incrementing the waitingTime. It also overrides the compareTo method used for sorting task by their arrivalTime.

* Server Class

Could be interpreted as the queue where the clients are enqueued for getting processed. The attributes defining this class are the task that are stored in a BlockingQueue, the waitingPeriod an atomicInteger referring to the time need to process all the current clients in the queue, the totalWaitingTime and totalProcessedTasks use for computing the average waiting time and a boolean flag that controls the while loop in the method run(). It offers all the getters and setters needed as well as a method for adding a new task. The run method takes tasks one by one from the queue and processes them by calling the Thread.sleep method with the value of task’s processingTime \* 1000.

* Scheduler Class

It is design to coordinate the list of servers by dispatching task to them using a strategy. The servers are stored in an ArrayList<Server> which is a private attribute. The two attributes maxNoServers and maxTasksPerServer define the limits for servers. The strategy field hold the instance of a class that implements the Strategy interface. The usage of this attribute is for adding a task to one of the class servers through the method addTask implemented in the specific class that is instantiated. The method initialize() creates the servers and adds them to the tasks attribute. The method changeStrategy() gives the possibility of selecting the strategy implementation form a set using it’s identifier enum value. The method updateServers() iterates through the servers and makes sure that if one of the servers is closed but it received at least one task, it starts the thread such that the task will be processed.

* Strategy Interface

Defines the blueprint for the classes that will implement a strategy for the required method requested in it. The method is addTask() which receives as parameters a set of servers and a task object which has to be added to one of the servers selected by a specific logic.

* SelectionPolicy Enum

Defines the identifiers for the classes that implement the Strategy interface in order for the scheduler to differentiate between them and be able to choose one in particular.

* ConcreteStrategyQueue Class

The class implements the Strategy interface and defines the method addTask using logic of choosing the server as the server with the lowest waiting time.

* SimulationManager Class

The attributes for this class are mainly the boundaries for the simulation environment and an object of type Scheduler that is used for coordinating the clients and the queues on which the simulation is based. The class provides a method (generateTasks) for generating N different clients with all their attributes set accordingly to the given simulation parameters. The methods decreaseProcessingTime() and increaseWaitingTime() are used to update the fields of the task objects that were dispatched to servers by incrementing their waiting time or decrementing their processing time by one if needed. It also provides two methods for printing the tasks printTasks() and printTime() which prints the current simulation time on the screen such that the user has a visual feedback of the application while it is running. The averageWaitingTime() method computes the average waiting for all the servers in the scheduler. The run() method contains a while loop that iterates for the number of times specified by the simulation time, each iteration meaning a time step of one second. At each second all the tasks that have the arrivalTime equal to the current simulation time are dispatched by the scheduler to a server in which it waits to be processed. Also, in every iteration of the loop a new state is printed to the output file.

1. Results

The testing of the application is done on different sets of simulation parameters. We make sure that the clients are dispatched to a server when their time has come as well as checking if they are deleted from the queue after being processed. Another thing to be checked is the waiting time of each client. Assuring this is correctly calculated we can move on to check if the final result is right.

Some of the scenarios for which the application was tested are:

* Adding all the clients at once to test the dispatch strategy.
* Chaining clients such that there is no waiting time.

1. Conclusions

The main objective of this project was to get familiar with threads, to understand the way they works as well as understanding the benefits of using them. Moreover, the project provides a good example of working with strategy patters that proves how powerful it can be because it is so generic.

In terms of future improvements, considering the generic structure of the strategy patter it is clear that many other strategies can be implemented thus resulting in other ways of controlling the queues. Another big improvement would be adding the possibility of a client to leave the queue. This may result in a gap that should be dealt with dynamically in order to maintain the efficiency.

1. Bibliography

* <https://stackoverflow.com/>
* <https://docs.oracle.com/javase/tutorial/>
* Ppt presentation: Java Concurrency