**Homework 2**

1. **Objective**

The main objective of this homework is to design and implement a simulation application aiming to analyze queuing-based systems for determining and minimizing clients’ waiting time.

Input data:

- Minimum and maximum interval of arriving time between customers;

- Minimum and maximum service time;

- Number of queues;

- Simulation interval;

- Other information you may consider necessary (I choose the number of clients);

Minimal output:

- The average of waiting time, service time and empty queue time for 1, 2 and 3 queues for the simulation interval and for a specified interval (other useful information may be also considered);

- Log of events and main system data;

- Queue evolution;

- Peak hour for the simulation interval;

The secondary objectives are:

1. **Designing a Client.** This will be our “consumer” that is generated randomly and is queued at one of the servers at its arrival time and is taken out of the queue when its at the top and its processing time passes. This is detailed in cap 2 and the implementation can be found in cap 4.
2. **Designing a Server.** The server is used to queue and offer service to the Clients. A server uses its own Thread in synchronization with the other servers and the main Controller. This is detailed in cap 2 and the implementation can be found in cap 4.
3. **Designing a Scheduler.** This is used to “deploy” the Clients to one of the existing servers. This can be found in cap 3 and the implementation in cap 4;
4. **Designing strategies.** Some strategies are required in order to schedule the Clients in the best possible manner. For the purpose of simplicity, here I used only 2: by time (shortest waiting time in queue) and by queue (shortest number of Clients in queue). This is detailed in cap 2 and the implementation can be found in cap 4.
5. **Designing a Controller.** The controller is used to generate the Clients and to monitor all the operations done on them by the servers. This will be displayed on the Graphical User Interface (GUI). This is detailed in cap 2 and the implementation can be found in cap 4.
6. **Implementing the required output.** We are required to find different parameters like the average waiting time, the peak time, the empty queue time etc. The implementation can be found in the Controller class which can be seen in cap 4;
7. **Implementing the Graphic User Interface (GUI).** Final step where we make a simple GUI for the user to interact with. Can be found in cap 3.
8. **Problem analysis, modeling, scenarios, use cases**

There are several cases for this problem. The “actor” is the user who wishes to find the different parameters for the operations of a queue or simply wishes to see a queue in action. The system in this case is the Queue System (QS). Let us see the use cases:

1. Use case (simulation)

-The user introduces the simulation time;

-The user introduces the number of servers;

-The user introduces the number of clients;

-The user introduces the minimum processing time;

-The user introduces the maximum processing time;

-The user introduces the minimum arrival time;

-The user introduces the maximum arrival time;

-The user presses the “Start” button;

-The QS shows the evolution of the queues, the log and the generated Clients;

1. Use case (simulation + Parameters)

-The user introduces the simulation time;

-The user introduces the number of servers;

-The user introduces the number of clients;

-The user introduces the minimum processing time;

-The user introduces the maximum processing time;

-The user introduces the minimum arrival time;

-The user introduces the maximum arrival time;

-The user presses the “Start” button;

-The QS shows the evolution of the queues, the log and the generated Clients;

-The user chooses from which server he wants to see the parameters and presses the corresponding buttons;

-The user introduces the start time;

-The user introduces the finish time;

-The user presses the “Compute” button;

-The QS shows the results in the log;

1. Use case (reset)

-The user introduces the simulation time;

-The user introduces the number of servers;

-The user introduces the number of clients;

-The user introduces the minimum processing time;

-The user introduces the maximum processing time;

-The user introduces the minimum arrival time;

-The user introduces the maximum arrival time;

-The user realizes he introduced the wrong set of parameters and is deeply disappointed by his lack of attention;

-The user presses the “Reset” button;

-The QS resets;

1. Use case (simulation with queue)

-The user introduces the simulation time;

-The user introduces the number of servers;

-The user introduces the number of clients;

-The user introduces the minimum processing time;

-The user introduces the maximum processing time;

-The user introduces the minimum arrival time;

-The user introduces the maximum arrival time;

-The user presses the “Queue strategy” button;

-The user presses the “Start” button;

-The QS shows the evolution of the queues, the log and the generated Clients which are placed in servers by the Queue Strategy;

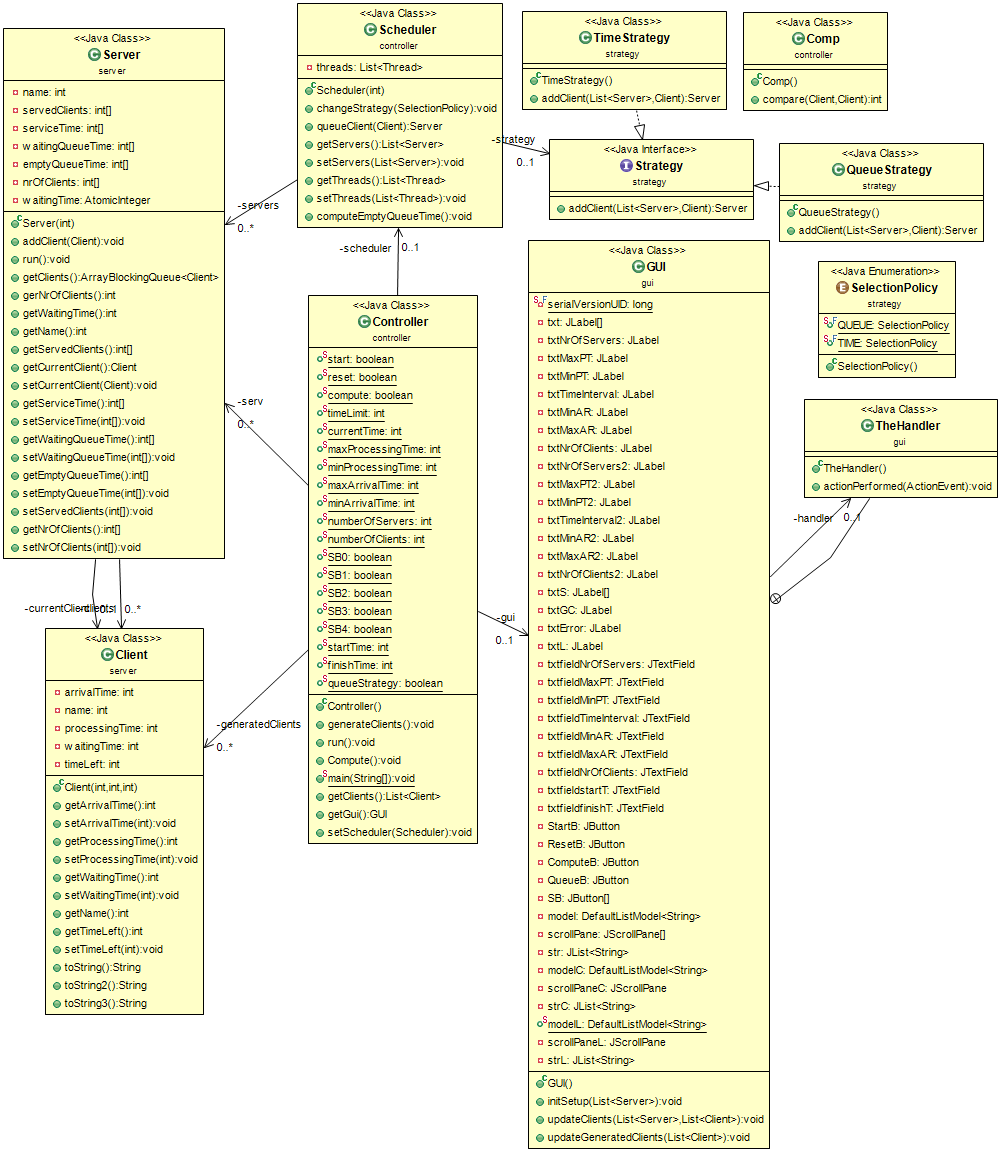
Of course, there can be several cases where the user tries to do an illegal action or an illogical one. One example is trying to introduce a bigger number of servers than what is allowed. Another one would be trying to introduce a bigger minimum processing time than a maximum processing time. For all of these cases a message is displayed warning the user of its illegal action.

1. **Design (design decisions, UML diagrams, data structures, class design, interfaces, relationships, packages, algorithms, user interface)**

Firstly, we must design a Client entity that can hold information like the arrival time of the Client and the processing time, both of which are given as input. Other information like a waiting time (to know how much it must wait until given service), the time left for service (to better illustrate how the servers work) and a name (easily identify the Client) I considered to be necessary. I also created different toStrings to help with the visualization.

Another step is the creation of a server which “consumes” and queues Clients. To queue the clients, I used an ArrayBlockingQueue. This structure permits the safe dequeuing and queuing of elements by threads. There are also several vectors in which I store information needed to compute the required parameters. A current Client is used to know which Client currently and an Atomic Integer Waiting time is used to continuously store info about the waiting time of a newly added Client.

Next, we have the Scheduler. The Scheduler has a list of servers, a list of threads and a strategy. It simply takes newly created Clients and adds them to a server by a chosen strategy. The threads are used for the servers.

****

An interesting aspect of this homework it is the fact that it firstly presents a design pattern, that is, the strategy/policy pattern. This pattern allows us to choose the behavior of the Clients when they choose a server.

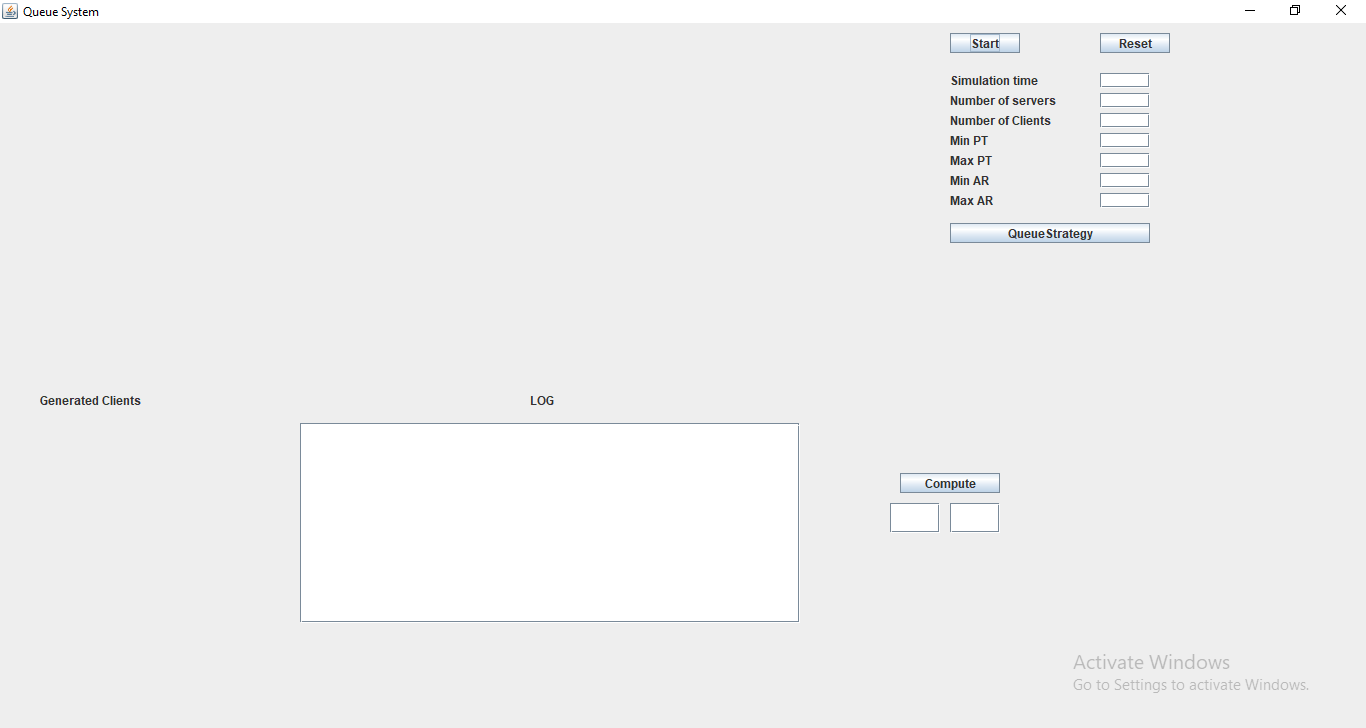
A Controller is used to generate new Clients, to compute the parameters required, to take data from the GUI and to update the GUI with new data.

The GUI simply displays the required information in different JScrollPanes for the servers and one for the generated Clients and one for the Log. The parameters can be computed after the simulation is done and they will be displayed in the Log.

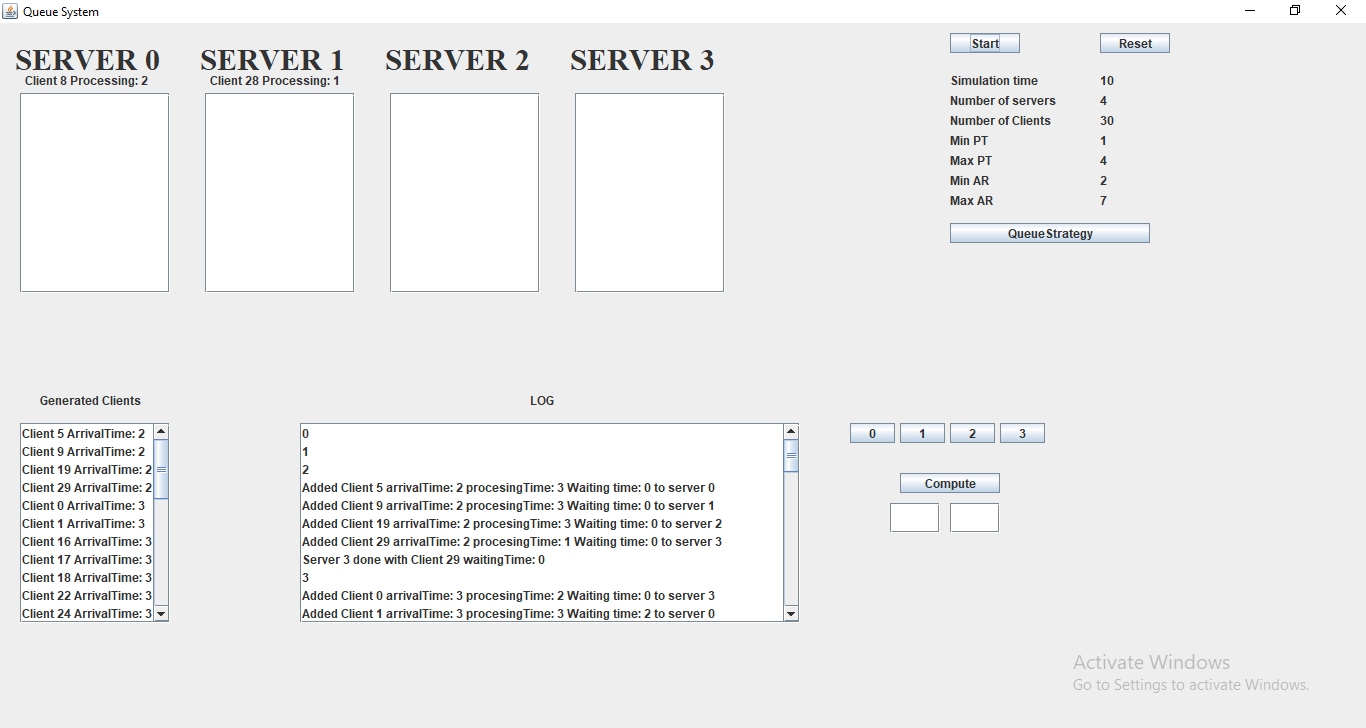
There are 4 packages used:

1. server: holds Server and Client classes;
2. strategy: holds everything related to the strategy pattern: Selection Policy, Strategy interface and Time Strategy and Queue strategy classes;
3. gui: Holds the GUI;
4. controller: holds the control systems: the Scheduler and the Controller (also a Comp class used to sort the clients after their arrival time);

Some images of the GUI:







1. **Implementation**

The most important classes used are:

1. Client

Here we have the basic “task” done by the servers.

**public** **class** Client {

**private** **int** arrivalTime;

**private** **int** name;

**private** **int** processingTime;

**private** **int** waitingTime;

**private** **int** timeLeft;

1. Server

As explained before, server uses the an ArrayBlockingQueue and several vectors to store required information for later.

**public** **class** Server **implements** Runnable {

**private** **int** name;

**private** **int**[] servedClients = **new** **int**[100];

**private** **int**[] serviceTime = **new** **int**[100];

**private** **int**[] waitingQueueTime = **new** **int**[100];

**private** **int**[] emptyQueueTime = **new** **int**[100];

**private** **int**[] nrOfClients = **new** **int**[100];

**private** Client currentClient;

**private** ArrayBlockingQueue<Client> clients;

**private** AtomicInteger waitingTime;

It also implements the Runnable interface which means the run method must be defined:

**public** **void** run() {

**while** (Controller.*currentTime* < Controller.*timeLimit*) {

**try** {

**if** (clients.isEmpty())

**this**.emptyQueueTime[Controller.*currentTime*] = 1;

**else** {

**this**.emptyQueueTime[Controller.*currentTime*] = 0;

currentClient = clients.poll();

**for** (**int** i = 1; i <= currentClient.getProcessingTime(); i++) {

Thread.*sleep*(1000);

currentClient.setTimeLeft(currentClient.getTimeLeft() - 1);

waitingTime.decrementAndGet();

nrOfClients[Controller.*currentTime*] = clients.size();

**if** (currentClient.getTimeLeft() == 0) {

**this**.servedClients[Controller.*currentTime*] = 1;

waitingQueueTime[Controller.*currentTime*] = currentClient.getWaitingTime();

**this**.serviceTime[Controller.*currentTime*] = currentClient.getProcessingTime();

GUI.*modelL*.addElement("Server " + **this**.name + " done with Client " + currentClient.getName()

+ " waitingTime: " + currentClient.getWaitingTime());

}

}

}

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

}

}

}

Here, although is pretty hard to see, the thread used by the server is put to sleep for the amount of processing time the current Client has. If no client is present, I increment the empty queue time and wait for a client until the simulation time is done. I also count the served clients and the waiting time for those that have been served.

1. Scheduler

The Scheduler is used to queue the coming clients and to start the servers.

**public** Scheduler(**int** NrOfServers) {

**this**.threads = **new** ArrayList<Thread>();

servers = **new** ArrayList<Server>();

**for** (**int** i = 0; i < NrOfServers; i++) {

Server s = **new** Server(i);

servers.add(s);

threads.add(**new** Thread(s));

}

// default strategy

strategy = **new** TimeStrategy();

}

**public** **void** changeStrategy(SelectionPolicy policy) {

**if** (policy == SelectionPolicy.***QUEUE***)

strategy = **new** QueueStrategy();

**if** (policy == SelectionPolicy.***TIME***)

strategy = **new** TimeStrategy();

}

**public** Server queueClient(Client c) {

**return** strategy.addClient(servers, c);

}

As can be seen from the code, we have a different thread for each server, also in changeStrategy, we use the policy pattern to choose between time and queue.

1. Controller

Here the magic happens. The controller uses some static fields like current time and the other parameters required to start the simulation. Also a scheduler, a Gui, a list of generated Clients, and a list of servers to make the computation on.

**public** **class** Controller **implements** Runnable {

**public** **static** **boolean** *start* = **false**;

**public** **static** **boolean** *reset* = **false**;

**public** **static** **boolean** *compute* = **false**;

**public** **static** **int** *timeLimit* = -1;

**public** **static** **int** *currentTime*;

**public** **static** **int** *maxProcessingTime* = -1;

**public** **static** **int** *minProcessingTime* = -1;

**public** **static** **int** *maxArrivalTime* = -1;

**public** **static** **int** *minArrivalTime* = -1;

**public** **static** **int** *numberOfServers* = -1;

**public** **static** **int** *numberOfClients* = -1;

**public** **static** **boolean** *SB0* = **false**;

**public** **static** **boolean** *SB1* = **false**;

**public** **static** **boolean** *SB2* = **false**;

**public** **static** **boolean** *SB3* = **false**;

**public** **static** **boolean** *SB4* = **false**;

**public** **static** **int** *startTime* = -1;

**public** **static** **int** *finishTime* = -1;

**public** **static** **boolean** *queueStrategy* = **false**;

**private** GUI gui;

**private** Scheduler scheduler;

**private** List<Client> generatedClients;

**private** List<Server> serv;

**public** **void** generateClients() {

Random random = **new** Random();

**int** procTime;

**int** arrivalTime;

**for** (**int** i = 0; i < *numberOfClients*; i++) {

procTime = random.nextInt(*maxProcessingTime* - *minProcessingTime*) + *minProcessingTime*;

arrivalTime = random.nextInt(*maxArrivalTime* - *minArrivalTime*) + *minArrivalTime*;

Client c = **new** Client(arrivalTime, procTime, i);

**this**.generatedClients.add(c);

}

generatedClients.sort(**new** Comp());

}

In generate Clients, I generate randomly parameters for the Clients, taking in account the maximum and minimum processing time and arrival time. I also sort the list in regard to the arrival time.

**public** **void** run() {

gui.initSetup(scheduler.getServers());

**for** (Thread t : **this**.scheduler.getThreads()) {

t.start();

}

*currentTime* = 0;

**while** (*currentTime* < *timeLimit*) {

GUI.*modelL*.addElement(Integer.*toString*(*currentTime*));

gui.updateGeneratedClients(generatedClients);

**for** (**int** i = 0; i <= generatedClients.size() - 1; i++) {

**if** (generatedClients.get(i).getArrivalTime() <= *currentTime*) {

scheduler.queueClient(generatedClients.get(i));

generatedClients.remove(i);

i--;

}

gui.updateClients(scheduler.getServers(), generatedClients);

}

gui.updateClients(scheduler.getServers(), generatedClients);

*currentTime*++;

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

}

}

gui.updateClients(scheduler.getServers(), generatedClients);

generatedClients.clear();

gui.updateGeneratedClients(generatedClients);

// Compute();

**for** (Thread t : **this**.scheduler.getThreads()) {

t.interrupt();

}

Thread.*currentThread*().interrupt();

}

In run I update the interface and run the threads. At each second, we take out some clients that have that arrival time and put the in the servers. In the end I close the threads.

**public** **void** Compute() {

serv = **new** ArrayList<Server>();

**if** (*SB0*)

serv.add(scheduler.getServers().get(0));

**if** (*SB1*)

serv.add(scheduler.getServers().get(1));

**if** (*SB2*)

serv.add(scheduler.getServers().get(2));

**if** (*SB3*)

serv.add(scheduler.getServers().get(3));

**if** (*SB4*)

serv.add(scheduler.getServers().get(4));

**float** averageWaitingTime = 0;

**float** EmptyQueueTime = 0;

**float** averageServiceTime = 0;

**float** servedClients = 0;

**int** peakTime = 0;

**int** maxClients = 0;

**int**[] numberOfClients = **new** **int**[100];

**for** (Server s : serv) {

**for** (**int** i = *startTime*; i < *finishTime*; i++) {

numberOfClients[i] += s.getNrOfClients()[i];

averageWaitingTime += s.getWaitingQueueTime()[i];

EmptyQueueTime += s.getEmptyQueueTime()[i];

averageServiceTime += s.getServiceTime()[i];

servedClients += s.getServedClients()[i];

}

}

**for** (**int** i = *startTime*; i < *finishTime*; i++) {

**if** (numberOfClients[i] > maxClients) {

maxClients = numberOfClients[i];

peakTime = i;

}

}

**if** (servedClients == 0)

GUI.*modelL*.addElement("Average waiting time: " + 0 + " Empty queue time: " + EmptyQueueTime

+ " Average service time: " + 0 + " Peak time: " + peakTime + " with " + maxClients + " clients");

**else**

GUI.*modelL*

.addElement("Average waiting time: " + averageWaitingTime / (servedClients) + " Empty queue time: "

+ EmptyQueueTime + " Average service time: "

+ averageServiceTime / (servedClients) + " Peak time: " + peakTime + " with " + maxClients

+ " clients");

}

I use the compute method to compute the parameters required. Here, by taking in account the selected servers, the start time and the finish time of the desired interval, the average waiting time, the empty queue time, the peak time and the average service time, are computed. In the main we wait for the right input to do the simulation and the computation.

1. Gui

The gui updates the clients, the servers and the generated Clients. Also takes and displays data from and to the user.

1. **Results**

There were no tests done on this Project as it is impossible to predict the generated Clients parameters and to check it against some values.

1. **Conclusions**

This homework brought a nice view of the Multithreading “domain” of java. I believe I learned more about this and about the strategy pattern by doing this practical homework than only by reading on the subjects. Further improvements of the code can be done making a cleaner GUI and by “cleaning” the code a little bit. Also, more strategies can be added, and more info can be extracted from the simulation.

1. **Biography**

Mostly what was presented in PT2019\_tema2\_Vio at the lab and latter reading at home.