



CentraleSupélec

Software Design Final Project

SimErgy - Simulation of an Emergency Department



EL AAMRANI Ahmed

LHOURTI Wassim

December 24th 2017
Gif-sur-Yvette, France

Abstract

This report describes SimErgy, a JAVA software framework to represent and simulate an Emergency Department operations. We will first depict this project's goals and background. Thereafter, we will discuss the project's specifications, simulation process and design patterns. Furthermore, we describe in this report our CLUI implementation and the relevant *Use Case Tests*. Finally, we depict the GUI platform that we developed for this project.

Keywords: Java, Software, Emergency Department, Simulation.

Contents

1	Introduction	2
1.1	Project's goals	2
1.2	Workload split	2
2	General context	3
2.1	Programming paradigm	3
2.2	Software hypotheses	3
3	Simulation's description	4
3.1	Creating a simulation	4
3.2	Running a simulation	4
3.2.1	Updating the events queue	5
3.2.2	KPI computation	6
4	Design Patterns	7
4.1	The Event-based approach	7
4.2	The Event class	8
4.3	The Composite pattern	9
4.4	The Mediator pattern	10
4.5	The Strategy pattern	11
4.6	The Singleton pattern	12
5	Command Line User Interface	13
5.1	CreateED	13
5.2	importInitED	16
6	Use Case Scenarios	18
7	Graphical User Interface	22
7.1	Frames Architecture	22
7.2	Support functionalities	25
8	JUnit tests	27
9	Conclusion	28
	Bibliography	29

1. Introduction

1.1 Project's goals

First of all, we must keep in mind that this project is part of an Object oriented software design course. For both of us, this was our first experience manipulating Object oriented programs and it gave us the opportunity to try out the power of OOP software ! A project was thereby the most suitable way to strengthen our knowledge of some important JAVA tools and techniques, as it confronted us with numerous specific issues that software developers have to face everyday.

Our final project intends to simulate an Emergency Department (ED), which assesses and treats people who present without prior appointment. This medical facility works often 24 hours a day and seven days a week, as ED workers have to deal with an unplanned patients' attendance. Furthermore, time is an essential factor in emergency treatment. Thus, it is important to provide a simulation platform to analyze the resources distribution and the patient's workflow, to ensure that the ED operates correctly and effectively.

Thereby, our goal is to develop a JAVA software framework that is able to:

- Represent an ED with all of its relevant resources (human and material) and workflow events;
- Simulate various test cases and automatically calculate suitable Key Performance Indicators (KPI).

1.2 Workload split

As we were seeking efficiency and flexibility in the coding part of the project, we planned to use GitHub. Unfortunately, we had a hard time setting up the GitHub platform in Papyrus and we decide to abandon this idea. Instead of that, we decided to use a *Google Drive* in which we shared the JAVA project after each modification. Moreover, we had a common *Google Sheet* document in which we could book time slots in order to modify the program. Furthermore, we split the specific tasks following the scheme below:

El Aamrani Ahmed	Lhourti Wassim
Common thinking about architectural design, with the realization of the first UML diagrams drafts	
Creation of the classes and methods of <code>fr.cs.simergy.events</code> and <code>fr.cs.simergy.coreed</code> packages, simultaneously with JUnit tests and Javadoc comments	Creation of the classes and methods of <code>fr.cs.simergy.humanresources</code> and <code>fr.csc.simergy.materialresources</code> packages, simultaneously with JUnit tests and Javadoc comments
Common thinking about the simulation part of the project.	
CLUI implementation	GUI implementation

2. General context

2.1 Programming paradigm

In order to understand the aims of such a project, we shall begin by asking ourselves: what is a simulation and why do we need one? One can define a simulation as the representation of the dynamic behavior of a system based on a model. Computer simulations are especially useful when the real experimentation on a system is either too expensive, too dangerous or just impossible. In our situation, experimenting with a real ED would take too much time and money and it would not be as modular as we want.

For this project, we can correctly represent an ED workflow with a *Discrete event simulation*, which means that the system's state changes only at discrete instances of time. Furthermore, we decided to adopt an *event-based approach* as a programming paradigm: this means that the model relies actually on a collection of *events*. Whenever an event occurs, specific actions take place and further events are scheduled.

2.2 Software hypotheses

In order to achieve this program, we had to set various hypotheses regarding the ED operations.

- First of all, we chose to **set a duration for the *Registration process* such as: $\Delta t_{\text{reg}} = 1$ instead of 0**. This assumption makes sense as it would be less realistic to consider an instantaneous operation. Furthermore, a model with $\Delta t_{\text{reg}} = 0$ might fail if we operate a simulation containing 5 nurses and at a certain time, 6 patients arrive to register simultaneously. However, such hypothesis brings up another issue: if a patient waits for registration while another one waits for installation, which one should be treated first by a unique idle nurse? Our current implementation gives the priority to *Installation*.
- An ED contains an **infinite number of sketchers**. Considering a finite number of these resources would just bring more irrelevant complexity to the project, as in reality, sketchers are not a limiting resource (you can easily buy more).
- **Each service has its own waiting room and each waiting room has an infinite capacity**. In reality, waiting rooms have a finite capacity and once they are filled, additional patients have to wait in "corridors". Making the waiting rooms with an infinite capacity does not affect the ED's performances and therefore, the KPI's calculations.
- **For a patient waiting to get installed (in a shock room or a box room), the only necessary resource for installation is the room (even if there is no idle physician)**. This helps to reduce the physician's idleness. Plus, for the sake of simplicity, we decided that **no operation can be interrupted by a more severe patient**. The combination of these two hypotheses raises a particular issue: if an L5 patient gets installed in a box room without a physician and an L1 patient comes for consultation in the meantime, the latter will have to wait even if the former has not started the consultation yet !

3. Simulation's description

Let's go through the simulation's process to explain each of its numerous steps.

3.1 Creating a simulation

Within the `fr.cs.simergy.clientinterface` package, the `MainSimulator` class is the one that allows to run a simulation through its main method. In this method, we first instantiate an object of the `EmergencySimulation` class, by indicating the following integer parameters that define an Emergency Department:

- The simulation period
- The number of physicians
- The number of nurses
- The number of transporters
- The number of box rooms
- The number of shock rooms
- The number of blood test rooms
- The number of MRI test rooms
- The number of radio rooms.

In our project, the simulation period corresponds actually to the duration in which patients can arrive to the ED. This means that if we put 10000 as a simulation period, no new patient is admitted into the ED after $t = 10000$. However, the simulation does not stop until all the patients are released.

Once those parameters are specified, we add each of the ED resources in dedicated *queues*, which are `Head` objects in our case (see section 4.3). For instance, if we create an `EmergencySimulation` object with 3 physicians, we instantiate 3 `Physician` objects that we add to the `idlePhysicians` queue. The same process is followed for material resources: if we specify 2 box rooms in our `EmergencySimulation`, we create two `BoxRoom` objects that we add to the `emptyBoxRoom` queue. Note that for simplicity reasons, when those objects are instantiated, they have a name and/or an ID which are automatically generated.

3.2 Running a simulation

Afterwards, we invoke the `setStochasticArrivals()` method from the `EmergencySimulation` class. The operations that are executed within this method are the scheduling of five events, which are the arrival of patients from the five different severity levels. The time stamp of each of these events is given as a sample of specific probability distributions. Let's consider the scheduling of the first L1 patient arrival. First, we create this event, which is a `PatientArrivalL1` object that is not executed yet. Thereafter, we get a sample from the predefined probability distribution for L1 patients (which is in this case a uniform distribution $U(60,70)$). Then, we schedule the arrival of the first L1 patient at this sample, using the `schedule()` method. Basically, when we call this method on an event, it accesses a list of all the events that have already been scheduled and compares their time stamp to the one of the event we want to schedule. In this manner, the event to be scheduled is placed at the right place of the chronological sequence.

Once those five first events are scheduled, we invoke the `runSimulation()` from the `EmergencySimulation` class, which calls by itself the **`runSimulation(double period)`** method from `Event` class. The idea is that we consider our list of events and while this list is not empty and the current time is inferior to the specified period parameter, we take the events regarding their chronological sorting and we apply specific *actions* on them. At this level, two points need to be clarified:

- **What is the value of the period parameter ?** We want our simulator to run as long as there is a patient in the ED that has not been released yet. As we said above, the only time parameter that we specify to our `EmergencySimulation` object is the period during which patients can arrive to the ED (let's call it `arrivalPeriod`). This means that we cannot know in advance the time at which the last patient is released. However, if we give to our `runSimulation(double period)` a parameter that is "*big enough*", we can be sure that the simulation will completely treat all the patients. In our code, we have put as parameter `double period = arrivalPeriod + 10000000000000000000f`. Hence, as long as the `arrivalPeriod` is small compared to `10000000000000000000f`, all the patients that entered the ED will be treated in our simulation.
- **What actions are applied to each event ?** We touch here one of most important methods in our program, as it makes the simulation move forward through the whole process. First of all, you need to know that for each step of the ED workflow (except the patient's arrival and release), we created a *start* and an *end* event. For example, we developed the classes: `StartRegistration`, `EndRegistration`, `StartRadioExamination`, `EndRadioExamination`, etc. Let's explain what happens for the `RadioExamination` event for instance. When we invoke the method `actions()` for a `StartRadioExamination` object, the first thing that the program does is that it looks for the priority patient from those waiting for a radio examination and it takes him out from the `Head` object `patientsWaitingForRadioTest`. Then, it looks for the first empty `RadioRoom` and takes it out from the `Head` object `emptyRadioRoom`. Afterwards, the program reports on this event and updates the total cost of the patient. Finally, the `actions()` method creates a new `EndRadioExamination` event and schedules it with a probabilistic duration, given by a specific distribution. For the `EndRadioExamination`, the `actions()` method puts back the radio room in the `emptyRadioRoom` queue, reports and updates the event queue.

3.2.1 Updating the events queue

Now, let's figure out how the event queue is updated. We have actually two different ways of adding events to their queue:

- The first one happens whenever we *start* an event. Indeed, for each event we start, we schedule automatically its end. Hence, this operation leads to adding a new event to our event queue. For this kind of update, the event is added to the queue at a given time, with no regard to any priority rules but the chronological one.
- The second one happens whenever we *end* an event. Actually, once an event is over, some resources become available and we have to dispatch them on new events. In order to correctly allocate these resources, we invoke the `updateEventQueue()` method. The following paragraph explains this method's scheme.

Basically, this method browses all of the events that our program can schedule at a given time T . Our main concern was about the resources availability at T . Indeed, if two events end at the same time, the `updateEventQueue()` will be invoked twice simultaneously. Hence, it is important to ensure that resources are available for each of these invocations before scheduling any event. Let's

take an example of an ED that only has one nurse: a Consultation and a Registration end at the same time, which means that `updateEventQueue()` will be called twice at the same time. If there are two patients waiting for registration, the two method's calls will make the program crash as they both request the same resource - the nurse - at the same time. To avoid such issues, we created specific attributes such as `RegResourceReserverdAtT` (which stands for *Registration resources reserved at the time T*) that get incremented by one whenever a specific resource is requested at a given time T .

Actually, This problem can only occur if two events end at the exact same time, which is quite impossible as our events durations are given by probabilistic distributions. However, we implemented this solution to have a general frame. Plus, to test that our `updateEventQueue()` worked correctly, we made various simulations within which all events had fixed durations of $\Delta t = 2$.

Finally, this method browses all of the possible events in a linear way, which means that within our code order, we explicitly prioritize our events. For those that require the same resources (a nurse in this case), we chose the following priorities strategy:

1. Shock Room Installation
2. Box Room Installation
3. Registration

3.2.2 KPI computation

Our simulation allows to compute for each severity level category two different kinds of KPI's in order to evaluate the ED's performance:

- **Length of stay (LOS):** the total duration spent by a patient in the ED. First, for each severity level, we create an `ArrayList` that stores the LOS values for all of this severity level patients. Furthermore, whenever a patient arrives to the ED, we create a `Patient` object for which one of his attributes is the arrival time. When this patient is finally released, we just add the difference between his release time and his arrival time to the severity-corresponding `ArrayList`. Ultimately, we just have to compute the average LOS value.
- **Door to Doctor Time (DTDT):** the time interval between a patient arrival to the ED and his first consultation by a doctor. We also have here for each severity level an `ArrayList` to store all of the DTDT values. For this KPI, as patients can get consulted twice, we only compute the DTDT at the first consultation. To achieve this feature, each patient has an attribute called `numberOfTests`. As a patient cannot make a test before his consultations, we compute the DTDT value for each patient only if his `numberOfTests` equals zero. Finally, we just compute the average DTDT value for each severity level.

If the system prints "?" as a result for a given KPI, this means that a resource is missing in the ED (if there is zero box room or zero nurse for example). This situation indicates that there is a problem in the ED because some patients cannot get released, as necessary resources are missing.

4. Design Patterns

4.1 The Event-based approach

If we come back to our problem, the main issue is to design a discrete event simulator. After doing some research in this subject, we found out there are 3 main possible implementations of a discrete event simulator:

- **An event-based approach:** In this case, simulation is a sequence of events. Each event occurs at a specified time, and is performed for an elapsed time $\Delta t = 0$ corresponding to simulation time.
- **An activity-based approach** where simulation is a sequence of activities. Each activity is a task which has a specified duration, and modifies our system after its completion
- **A process-based simulation approach:** In this approach, we take in consideration a whole entity and its progression throughout our simulator

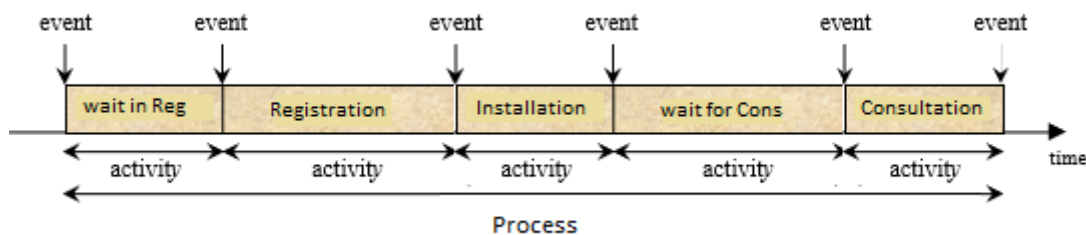


Figure 4.1: A comparison of the 3 different discrete event simulators

The diagram above depicts the differences between the three approaches for a patient throughout our simulator. As an example, we considered that he is only waiting for registration and consultation, and that he had not taken any examination test. We have represented particularly 3 key notions here which are event, activity and process.

But, these 3 different approaches have different advantages and drawbacks. Here is an overview table of characteristics, where we compare these 3 approaches among their basic implementations.

	Event scheduling	Activity scheduling	Process scheduling
Simplicity	+++	++	-
Execution efficiency	+	-	++
Overall	++++	+	+

Naturally, the comparison above leads us to **adopt the event-based approach!** The question that results is: what are the different events concerning a patient through our workflow ? According to the project's specifications, a patient might deal with several events (according to event-based simulation approach) that we represent below for a random patient.

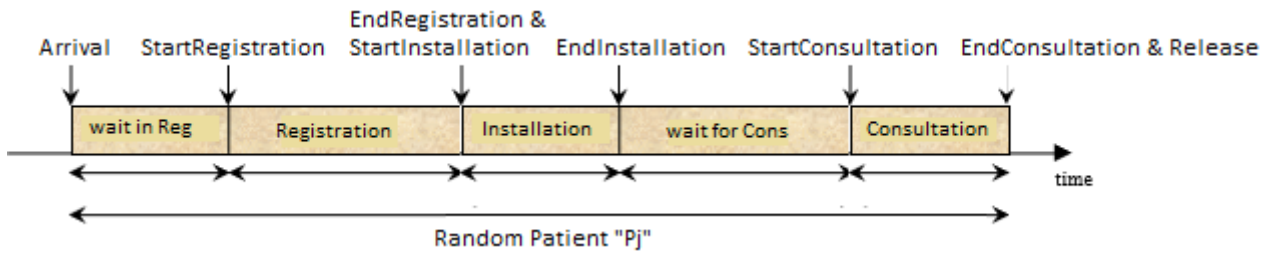


Figure 4.2: An example of the event based simulation with a random patient

In fact, each event is executed instantly relatively to our simulation. Naturally, we decided to define an event for each one of them, **extending a Superclass Event**.

4.2 The Event class

During the execution of simulation models, the simulator handles various events from our “event-queue”. Furthermore, operations performed on each event should be done independently of other events. Definitely, the execution of each event affects the state of the system, but it does not affect other events already scheduled. We could obtain these specifications across the **Command pattern**. An abstract class event is taken as an upper-class, and each of its subclasses defines an `actions()` method which defines the execution handler for each type of event. Hence, we could define several types of events in our system, and add them to our system simply by extending the class `Event`.

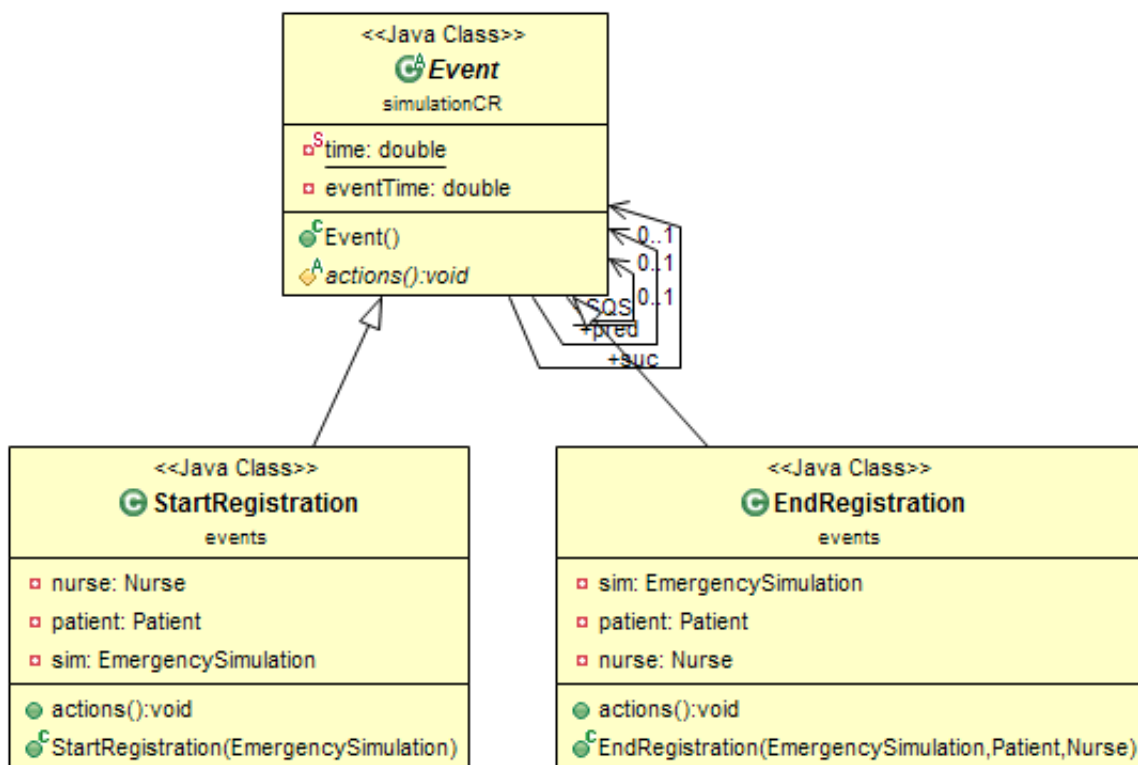


Figure 4.3: A simplified UML Diagram of the Command Pattern

The class event which is the core of our program, provides us with several methods:

- **actions()** is responsible of executing the actions of an event when it occurs.
- **schedule(double eventTime)** is responsible of adding the event to the list of events that should be executed later on.
- **cancel()** is responsible of retrieving an event from the list of events waiting to be executed
- **time()** returns the current simulation time
- **runSimulation(double period)** runs the simulation of all events for a given period
- **stopSimulation()** to stop Simulation
- **executeFirstEvent()** executes the first event in all the events queue

Note that this class was based on a report about *Discrete Event Simulation in JAVA* from the Computer Science Department of Roskilde University [1].

4.3 The Composite pattern

The other major design pattern is the Composite pattern. In fact, we considered in our design that:

- The **class Linkage** is a common Superclass for **class Link** and **Head**
- An object of **class Link**: List members are objects of subclasses of the **class Link**. Those list members can either be:
 - **Material resources** such as rooms
 - **Human resources** such as patient, physician, nurse or transporter
- An object of the **class Head** is used to represent a list. This list can either be:
 - **Waiting list** for each activity
 - **Idle resources** that we can use at each iteration

The following figure gives a UML representation of this design pattern.

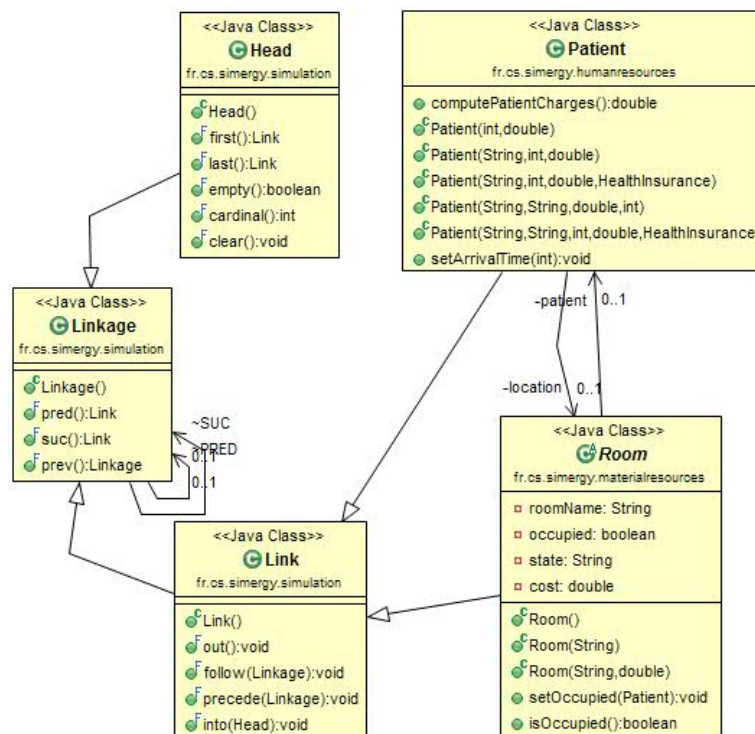


Figure 4.4: The Composite pattern UML Diagram

When dealing with a tree-structured data, we had the problem of differentiating leaf-nodes (patient for example) and branches (waiting queue for registration for example) each time. This makes code more complex, and therefore more error prone. This solution allows treating complex (head) and primitive objects (link) uniformly. The key here is that we manipulate a single instance of the object as we would manipulate a group of them.

So, if we take a Head instance (hd), in our simulator it can be waiting queue for registration for example. Head objects support all the following methods:

Class Head

```
public class Head extends Linkage {
    public final Link first();
    public final Link last();
    public final boolean empty();
    public final int cardinal();
    public final void clear();
}
```

- **hd.first()** returns a reference to the first member of the list (null, if the list is empty).
- **hd.last()** returns a reference to the last member of the list (null, if the list is empty).
- **hd.cardinal()** returns the number of members in the list (null, if the list is empty).
- **hd.empty()** returns true if the list hd has no members; otherwise null.
- **hd.clear()** removes all members from the list.

Note, that in our simulator, when invoking each event, we don't pick the first patient in a queue, but the highest priority patient given its time and its priority level after calling the method 'getPriorityPatient(Head s)'.

Furthermore, a Link instance (lk) is an object that can feed a head, especially in our simulator it can be a human or a material resource. And, it has the following methods:

Class Link

```
public class Link extends Linkage {
    public final void out();
    ...
    public final void into(Head s);
}
```

- **lk.out()** removes lk from the list (if any) of which it is a member. The call has no effect if lk has no membership.
- **lk.into(hd)** removes lk from the list (if any) of which it is a member and inserts lk as the last member of the list hd.

These classes were also inspired from the 'Discrete Event Simulation in Java' report [1]. But we manage it in order to satisfy our needs, like for example by adding executeFirstEvent method, and getting the highest priority patient, not only the first patient in a queue.

4.4 The Mediator pattern

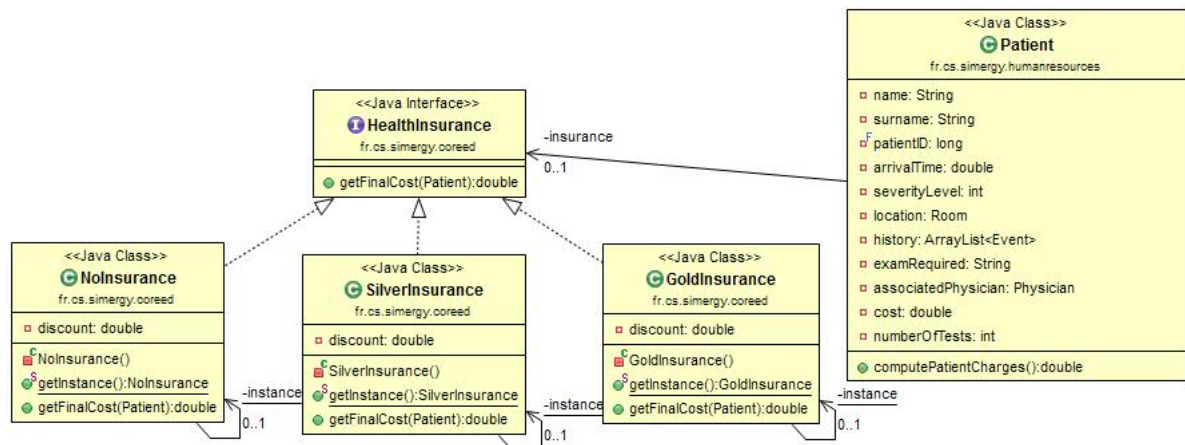
But when implementing our simulator, it happens that we face a big problem! In fact we had a time() static reference in our simulation, and at each discrete time(), different events can update the eventqueue, but if those different events don't communicate between them then they 'll have a problem of allocating more events than resources available, and it leads us naturally to choose implementing "mediator" design pattern. If we take the example of a queue, then we'll be aware of

the fact that our events shouldn't schedule other new entities at each fixed time if the entity amount in the queue reaches the top, and this depends on the state of our system. This pattern was possible by making a function 'updateEventQueue' that allows communication between all events, rather than having direct communication between events.

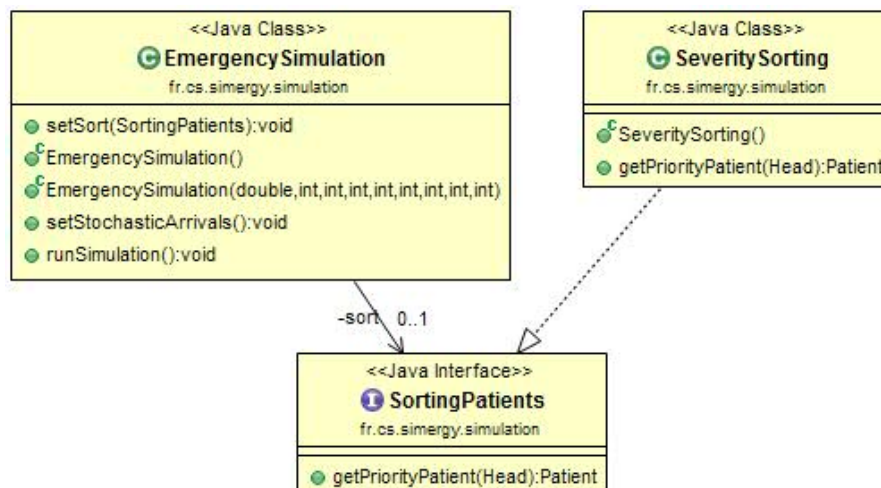
4.5 The Strategy pattern

From design patterns, we also implemented strategy pattern as we seen in the lectures. We implement this pattern for 2 specific tasks:

- Calculating patient charges depending on its health insurance, this computation depends indeed on either having 'no Insurance', 'silver Insurance', or 'gold Insurance'. In fact, the cost is calculated as the sum of all services where the patient has been through. Then we 'computePatientCharges()' for a given patient by calling the corresponding method for the subclass of the interface 'HealthInsurance'. HealthInsurance has 3 subclasses : NoInsurance, SilverInsurance and GoldInsurance. The first one has 0% discount on the total cost, the second 50% discount on the total cost, and the last 80% discount on the total cost. Note that all patients are given 'NoInsurance' by default, if no insurance was defined when registering the patient.



- Calculating the priority patient on a queue. In fact, we implemented a subclass of 'SortingPatients' interface which is 'SeveritySorting'. If we have other strategies, we could only add new subclasses of 'SortingPatients', and set the sorting strategy of the simulation in the emergency department. The 'SeveritySorting' method is a method which in a queue takes the most urgent patient, then if we have several patients in the same urgent category we choose the patient that first arrived at the queue.



4.6 The Singleton pattern

Last but not least, we also implemented a singleton pattern to ensure that only one instance of a class has been instantiated for a given class:

- We implemented a singleton pattern for subclasses of HealthInsurance interface, because there are only 3 general insurances, and we shouldn't create an instance for each patient. In order to achieve that, we defined the constructor as private, and defined also a static method called `getInstance()` which invokes the class constructor and is in charge for returning the only instance of the class.

5. Command Line User Interface

We implemented a client-command user interface that takes basic functionalities in our simulator. Before talking about functionalities, and in order to enhance your user experience and ameliorate visualization of all outputs in the Java console, you may uncheck 'Limit console output' following these steps:

Papyrus > Windows > Preferences > Run/Debug > Limit console output

Then, to use our CLUI, you should run the **CLUIDisplay** class in the `fr.cs.simergy.clientinterface` package.

First of all, when running the `CLUIDisplay.java` class, the console displays:

```
-----  
SIMERGY COMMAND-LINE USER INTERFACE  
-----  
Please select an option:  
  Create a new ED: createED <EDname>  
  Import an initial ED: importInitED <Filename>  
  Run a test case scenario: runTest <Filename>  
-----  
Please enter your command : Here you can type whatever you want
```

Hence, the client has 3 options at the beginning that we will display in the sections above.

5.1 CreateED

Here we can create an Emergency Department (ED) by giving its name. Once we do this, you could apply one of the list commands for Command-Line User Interface:

LIST OF COMMANDS FOR CLIENT-COMMAND USER INTERFACE

These are list of commands allowed by the Command-Line user interface :

```
createED <EDname>
addNurse <NurseName, NurseSurname>
addNurse <>
addPhysi <PhysiName, PhysiSurname>
addPhysi <>
addTransporter <>
addRoom <RoomType, RoomName>
addRadioService <DistType, DistParams>
addMRI <DistType, DistParams>
addBloodTest <DistType, DistParams>
setL1arrivalDist <DistType, DistParams>
setL2arrivalDist <DistType, DistParams>
setL3arrivalDist <DistType, DistParams>
setL4arrivalDist <DistType, DistParams>
setL5arrivalDist <DistType, DistParams>
addPatient <PatientName, PatientSurname, SeverityLevel, HealthInsurance>
addPatient <SeverityLevel>
registerPatient <PatientID>
setPatientInsurance <PatientID, HealthInsurance>
executeEvent <>
display <>
restart <>
quit <>
```

Parameters follow strict rules, and we can precise some of them:

SOME PRECISIONS ABOUT PARAMETERS

DistType is either
 'exponential' with a single positive real value parameter
 'deterministic' with a single positive real value parameter
 'uniform' with a pair (a,b) of real value parameters such as $0 < a < b$

RoomType is either 'boxRoom', or 'shockRoom'

HealthInsurance is either 'noInsurance', 'silverInsurance', or 'goldInsurance'

Once you created your Emergency Department, you could add human resources such as nurses, physicians and transporters:

```
Please enter your command : addNurse
Nurse Nurse475207 added

Please enter your command : addPhysi
Physician Physician42042952 added

Please enter your command : addTransporter
Transporter Transporter80564076 added
```

You could also add material resources such as box rooms, shock rooms, and medical service rooms as you can see above :


```

Please enter your command : addRoom boxRoom box1
Box Room added
-----
Please enter your command : addRoom shockRoom shock1
Shock Room added
-----
Please enter your command : addRadioService uniform 20 60
Radio room added
-----
Please enter your command : addMRI exponential 4
MRI room added
-----
Please enter your command : addBloodTest deterministic 40
BloodTest room added

```

After initializing the system, you could add patients to the workflow:

```

Please enter your command : addPatient 1
Patient Patient98033305 added

```

Here, '1' is the severity level of the patient and '82570323' is the unique ID of the patient. Implicitly, the patient arrives at the current time of the simulation. One could also display the state of the simulator:

```

Please enter your command : display
Number of idle physicians : 2
Number of idle nurses : 2
Number of idle transporters : 1
Number of idle Box Rooms : 2
Number of idle Shock Rooms : 2
Number of idle Blood Rooms : 1
Number of idle Radio Rooms : 1
Number of idle MRI Rooms : 1
-----
Patients Waiting For Registration : 0
Patients Waiting For Shock Installation : 0
Patients Waiting For Box Installation : 0
Patients Waiting For Consultation : 0
Patients Waiting For Blood : 0
Patients Waiting For Radio : 0
Patients Waiting For MRI : 0

```

Note that it is normal to have empty queues for services, because no event has been executed.

```

Please enter your command : executeEvent
t = 0 ____ Arrival ____ Patient ID : 98033305 ____ SeverityLevel : L1
-----
Please enter your command : executeEvent
t = 0 ____ Start Registration ____ Patient ID : 98033305
-----
Please enter your command : executeEvent
t = 1 ____ End Registration ____ Patient ID : 98033305
-----
Please enter your command : display
Number of idle physicians : 2
Number of idle nurses : 2
Number of idle transporters : 1
Number of idle Box Rooms : 2
Number of idle Shock Rooms : 2
Number of idle Blood Rooms : 1
Number of idle Radio Rooms : 1
Number of idle MRI Rooms : 1
-----
Patients Waiting For Registration : 0
Patients Waiting For Shock Installation : 1
Patients Waiting For Box Installation : 0
Patients Waiting For Consultation : 0
Patients Waiting For Blood : 0
Patients Waiting For Radio : 0
Patients Waiting For MRI : 0

```

After finishing registration, this patient is L1 severity, so he would be waiting for shock installation, in order to be installed in a shock room. And you can continue manipulating manually the simulator in our system. Note that all kind of errors are handled, and we show here some errors that can occur :

```

Please enter your command : addRoom box box1
Invalid syntax --- Please for RoomType choose either 'boxRoom' or 'shockRoom'
-----
Please enter your command : addPatient Lhourti Wassim 1 silver
Invalid syntax --- Please for HealthInsurance choose either 'noInsurance' or 'silverInsurance' or 'goldInsurance'
-----
Please enter your command : setL1arrivalDist uniform -4 20
Your distribution parameters should be strictly positive

```

5.2 importInitED

After finishing this manual simulation, we can also import an emergency department based on a file initialization. But first, we should type 'restart' in the command line, in order to have the possibility to choose between the 3 commands : 'createED', 'importInitED', 'runTest'. Here, we choose to 'importInitED', if no arguments are given then the system imports 'my_simergy.ini', else it imports the file given on input argument.

```

Please select an option:
    Create a new ED: createED <EDname>
    Import an initial ED: importInitED <Filename>
    Run a test case scenario: runTest <Filename>
-----
Please enter your command : importInitED
ED imported
-----
You could apply one of the commands defined above for this specific EmergencyDepartment
If you want to work on another Emergency Department please type the command 'restart'
-----
Please enter your command : display
Number of idle physicians : 3
Number of idle nurses : 5
Number of idle transporters : 5
Number of idle Box Rooms : 4
Number of idle Shock Rooms : 1
Number of idle Blood Rooms : 1
Number of idle Radio Rooms : 1
Number of idle MRI Rooms : 1
-----
Patients Waiting For Registration : 0
Patients Waiting For Shock Installation : 0
Patients Waiting For Box Installation : 0
Patients Waiting For Consultation : 0
Patients Waiting For Blood : 0
Patients Waiting For Radio : 0
Patients Waiting For MRI : 0

```

Especially, here we will import an emergency department from ‘my_simergy.ini’, where the department contains 3 physicians, 5 nurses, 5 transporters, 4 box rooms, 1 shock room, 1 blood test room, 1 radio service room and 1 MRI room. Once we import our emergency department, we could apply the same commands as when we create an ED ourselves.

When finishing with simulating this imported department, you could type ‘restart’ in order to build other simulations.

```

Please enter your command : restart

```

Finally, we explain the runTest option in the following chapter, as it deals with Use Case Scenarios.

6. Use Case Scenarios

In the command-line user interface, we have defined the possibility to run the simulation of some use case scenarios, so one could run simulation that the file contains only by typing 'runTest <filename>'.

testScenario1

```
-----  
SIMERGY COMMAND-LINE USER INTERFACE  
-----  
Please select an option:  
    Create a new ED: createED <EDname>  
    Import an initial ED: importInitED <Filename>  
    Run a test case scenario: runTest <Filename>  
-----  
Please enter your command : runTest testScenario1.txt
```

In this testScenario1, we defined very limited resources: 1 physician, 1 nurse, 1 transporter, 1 box room, 1 shock room, 1 blood room, 1 mri room, 1 radio room. And we ran the simulation for 10000 minutes. For one given simulation, we have:

```
-----  
Simulation completed  
-----  
Average Door To Doctor Time for L1 Patients : 13,65  
Average Door To Doctor Time for L2 Patients : 16,15  
Average Door To Doctor Time for L3 Patients : 1219,63  
Average Door To Doctor Time for L4 Patients : 1385,62  
Average Door To Doctor Time for L5 Patients : 1993,55  
-----  
Average Length Of Stay for L1 Patients : 149,3  
Average Length Of Stay for L2 Patients : 264,19  
Average Length Of Stay for L3 Patients : 1679,42  
Average Length Of Stay for L4 Patients : 1996,3  
Average Length Of Stay for L5 Patients : 2831,23  
-----
```

Here, we can see that in an asymptotic initialization, L1 patients have higher priority than L2 than L3 and so on. And it's coherent with the basic sorting priority where we took first L1 patients waiting for a service, then L2 ...

testScenario2

```
-----  
SIMERGY COMMAND-LINE USER INTERFACE  
-----  
Please select an option:  
    Create a new ED: createED <EDname>  
    Import an initial ED: importInitED <Filename>  
    Run a test case scenario: runTest <Filename>  
-----  
Please enter your command : runTest testScenario2.txt|
```

Here, we studied the other asymptotic behavior of our system where there are much more human and material resources resource. Indeed, we choose here 10 physicians, 10 nurses, 10 transporters, 6 box rooms, 6 shock rooms, 5 blood rooms, 5 mri rooms, 5 radio rooms. And we ran the simulation for 20000 minutes.

```
-----
Simulation completed
-----
Average Door To Doctor Time for L1 Patients : 3
Average Door To Doctor Time for L2 Patients : 3
Average Door To Doctor Time for L3 Patients : 3
Average Door To Doctor Time for L4 Patients : 3
Average Door To Doctor Time for L5 Patients : 3
-----
Average Length Of Stay for L1 Patients : 55,43
Average Length Of Stay for L2 Patients : 57,21
Average Length Of Stay for L3 Patients : 49,37
Average Length Of Stay for L4 Patients : 53,1
Average Length Of Stay for L5 Patients : 54,9
-----
```

So, the idea here is that all patients are well served and don't wait too much or even don't wait for a given service. Note that fortunately, the average Door To Door Time (DTDT) is 3 because patients spend one minute on registration and 2 minutes on installation. However, the average length of stay (LOS) has the same average value for all types of patients, but is randomly distributed because patients have similar verdicts after consultation.

testScenario3

```
-----
SIMERGY COMMAND-LINE USER INTERFACE
-----
Please select an option:
    Create a new ED: createED <EDname>
    Import an initial ED: importInitED <Filename>
    Run a test case scenario: runTest <Filename>
-----
Please enter your command : runTest testScenario3.txt
```

Here, we took more or less the same use case scenario 2, but we just limited resources for test rooms. Indeed, we choose here 10 physicians, 10 nurses, 10 transporters, 6 box rooms, 6 shock rooms, 3 blood rooms, 3 mri rooms, 3 radio rooms. And we ran the simulation for 20000 minutes.

```
-----
Simulation completed
-----
Average Door To Doctor Time for L1 Patients : 3
Average Door To Doctor Time for L2 Patients : 3
Average Door To Doctor Time for L3 Patients : 3
Average Door To Doctor Time for L4 Patients : 3
Average Door To Doctor Time for L5 Patients : 3
-----
Average Length Of Stay for L1 Patients : 69,62
Average Length Of Stay for L2 Patients : 137,49
Average Length Of Stay for L3 Patients : 144,51
Average Length Of Stay for L4 Patients : 153,49
Average Length Of Stay for L5 Patients : 188,02
-----
```

We can deduce, that DTDt is not affected and remains equal to 3, because there are always enough resources for registration, and installation. But the LOS change because test rooms are limited, and priority is given to higher priority levels, and the average LOS increases for all patients, because majority of them wait for the service.

testScenario4

```
-----
SIMERGY COMMAND-LINE USER INTERFACE
-----
Please select an option:
    Create a new ED: createED <EDname>
    Import an initial ED: importInitED <Filename>
    Run a test case scenario: runTest <Filename>
-----
Please enter your command : runTest testScenario4.txt
```

In order to optimize hospital revenue, we tried several values of each parameter and choose the lowest values (because we want always to pay less) of each resource guaranteeing reasonable DTDt and LOS averages. And we found out that we should take 3 physicians, 3 nurses, 3 transporters, 3 box rooms, 2 shock rooms, 4 blood rooms, 2 mri rooms, 2 radio rooms. Note that blood test rooms are much cheaper than other rooms. The KPI for a given simulation are as below:

```
-----
Simulation completed
-----
Average Door To Doctor Time for L1 Patients : 6,18
Average Door To Doctor Time for L2 Patients : 6,97
Average Door To Doctor Time for L3 Patients : 6,33
Average Door To Doctor Time for L4 Patients : 7,39
Average Door To Doctor Time for L5 Patients : 10,64
-----
Average Length Of Stay for L1 Patients : 115,68
Average Length Of Stay for L2 Patients : 120,81
Average Length Of Stay for L3 Patients : 123,87
Average Length Of Stay for L4 Patients : 131,83
Average Length Of Stay for L5 Patients : 168,95
-----
```

Here DTDt are always very low, and LOS are bearable. Note, that if we decrease each parameter of 1, the performance goes very bad.

testScenario5

```
-----
SIMERGY COMMAND-LINE USER INTERFACE
-----
Please select an option:
    Create a new ED: createED <EDname>
    Import an initial ED: importInitED <Filename>
    Run a test case scenario: runTest <Filename>
-----
Please enter your command : runTest testScenario5.txt
```

In this last test case scenario, we wanted to show that from an optimized solution, decreasing a resource by 1 affects very strongly simulation results. In fact, we take the same input parameters as the case scenario 4 but we only change the number of physicians to 3 instead of 2. Here is the results of a simulation during 20000 minutes.

Simulation completed

Average Door To Doctor Time for L1 Patients : 10,95
Average Door To Doctor Time for L2 Patients : 11,82
Average Door To Doctor Time for L3 Patients : 1719,44
Average Door To Doctor Time for L4 Patients : 1931,99
Average Door To Doctor Time for L5 Patients : 2579,83

Average Length Of Stay for L1 Patients : 56,11
Average Length Of Stay for L2 Patients : 60,03
Average Length Of Stay for L3 Patients : 1770,18
Average Length Of Stay for L4 Patients : 1983,51
Average Length Of Stay for L5 Patients : 2631,37

Here we can say that L1 and L2 are not much affected, but lower priority patients wait too much for a physician, because we can see that for the latter the difference between DTDT and LOS is very small. So, the bottleneck here is for consultations.

Analysis

For a problem that can be represented by a stochastic event simulator, the department's manager should really put strong attention on resources, because we can have optimal resources that set a trade-off between clients' satisfaction and department investment. However, the small perturbation of our system (for example a human resource gets off duty) has a cascading perturbation that weakens very badly performance indicators. Thereby, attention should be focused on keeping the number of available resources under control.

7. Graphical User Interface

Our project's Graphical User Interface (GUI) goal is to give to the user a simple and clear way to interact with the SimErgy core to process simulations with given parameters. We will present in this chapter the way we designed our GUI. To use our GUI, you should run the **GuiDisplayer** class within the `fr.cs.simergy.clientinterface` package.

7.1 Frames Architecture

First of all, we decided to use the Swing GUI toolkit because it was the one taught in the course [3]. Furthermore, the Swing uses lightweight components that do not require allocation of native resources in the operation system.

Our GUI is made of three different JFrame containers, that appear one after another:

- **WelcomeFrame:** the first one that appears and displays a welcome message to the user. The client is then invited to go further in the GUI to parameter the simulation.
- **ParametersFrame:** the second frame that appears allows the user to choose various simulation's parameters.
- **OutputFrame:** the last frame displays the simulation result, by reporting all of the actions that occurred in the ED and giving all the relevant KPI's.

WelcomeFrame

When a user displays our GUI, the first frame that comes up states "Welcome to our SimErgy Simulator". The client has then two choices. If he clicks "Next", a ParametersFrame is displayed and he can enter the simulation parameters. Conversely, if he clicks "Cancel", the simulation process simply stops and the current WelcomeFrame is simply closed.



Figure 7.1: The WelcomeFrame graphical appearance

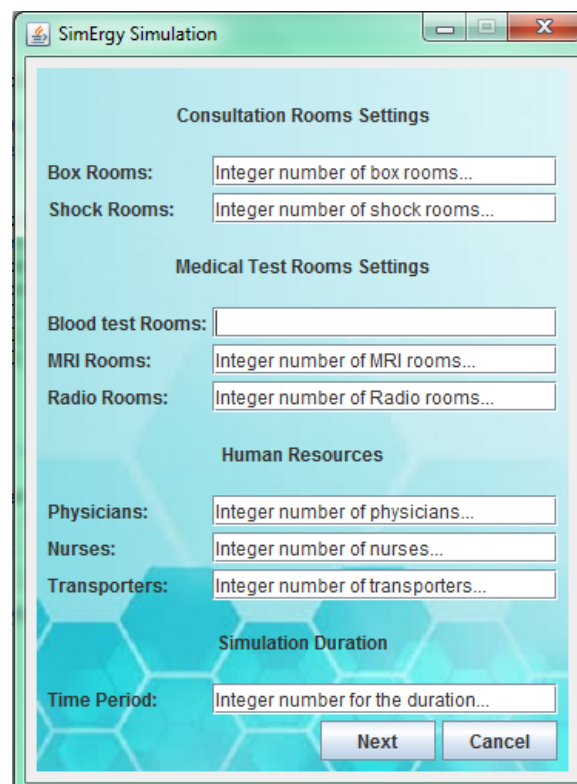
ParametersFrame

In this frame, the user can enter the parameters that define a simulation, which are the following:

- Number of box-rooms / Number of shock-rooms
- Number of Blood Test rooms / Number of MRI rooms / Number of Radio rooms
- Number of physicians / Number of nurses / Number of transporters
- The simulation duration

Even if the last parameter is actually a double, we decided that all the parameters have to be entered as integers. Each row of the frame's panel consists in a JLabel with the parameter's name and a JTextField where the user can type the chosen integer. These couples of components are organized vertically in the JFrame to emphasize them. Furthermore, we added in each of the JTextFields an initial text message stating for example: *"Integer number of box rooms..."*.

In order to make the user experience easier, we added a feature that erases the text contained in a JTextField whenever it gains focus. To do so, we created a method `setVisibleTextField(JTextField field)` that adds a `FocusListener` to the `JTextField` and overrides the `FocusListener` method `focusGained(FocusEvent arg0)`. This idea led us to another issue: when the user finished entering all the required data, if he decides to put the focus back on a JTextField, the value typed in this one is erased. We might just assume that is a user focuses back on a JTextField, his goal is to change the value typed in it.



The screenshot shows a Java Swing window titled "SimErgy Simulation". The window contains a light blue panel with a hexagonal pattern at the bottom. The panel is organized into sections with labels in bold. The "Consultation Rooms Settings" section has two rows: "Box Rooms:" and "Shock Rooms:", each followed by a text field containing the placeholder text "Integer number of box rooms...". The "Medical Test Rooms Settings" section has three rows: "Blood test Rooms:" followed by an empty text field, "MRI Rooms:" followed by a text field with "Integer number of MRI rooms...", and "Radio Rooms:" followed by a text field with "Integer number of Radio rooms...". The "Human Resources" section has three rows: "Physicians:" followed by a text field with "Integer number of physicians...", "Nurses:" followed by a text field with "Integer number of nurses...", and "Transporters:" followed by a text field with "Integer number of transporters...". The "Simulation Duration" section has one row: "Time Period:" followed by a text field with "Integer number for the duration...". At the bottom right of the panel are two buttons: "Next" and "Cancel".

Figure 7.2: The ParametersFrame graphical appearance

After entering the required parameters, the user can click "Next" to process the simulation. However, we must check that the given data types are correct (which means that they are all integers). For that, we created within the `actionPerformed` method a try-catch block that tries to convert the typed strings into Integers. If the program finds a type mismatch, it catches a `NumberFormatException` and shows a `JOptionPane` message dialog stating that an error was encountered. The user can then correct his mistake(s). We decided that, for simplicity, only one message dialog should appear even if many errors are found, which means that only one try-catch block is defined for all of the parameters.

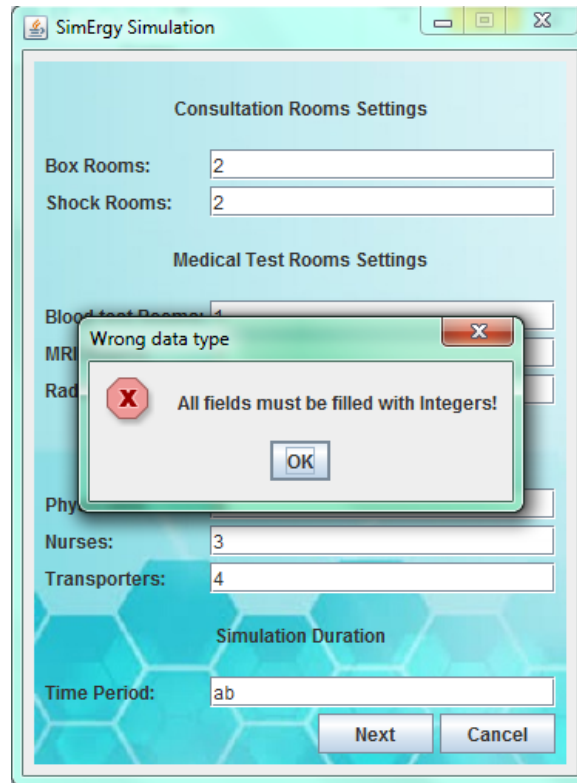


Figure 7.3: The error message that appears when there is a type mismatch

OutputFrame

The final frame that we designed allows to print the simulation report. It contains a non-editable `JTextArea` in which the results are displayed. We encountered an issue for which we didn't find the source: whenever we tried to instantiate an object of the `OutputFrame` class, we had two frames that was displayed one after another ! We trying to figure out why this was happening but we could not come up with the problem's reason. Hence, we just decided to fix the problem by creating a *Singleton pattern* that allows to create only a unique instance of the `OutputFrame` class. For that, we made the constructor private and we created a static method called `getOutputFrame()` and a static attribute of type `OutputFrame` called `instance`.

Furthermore, to print the results directly into the `JTextArea`, we had to redirect the `OutputStream`. We will present the way we handled this problem in the next section.

After reading the simulation report, the user can either click on the "New Simulation" button, which creates a `ParametersFrame` where he can enter new parameters to process a new simulation,

or click "Finish" to end the whole GUI experience.

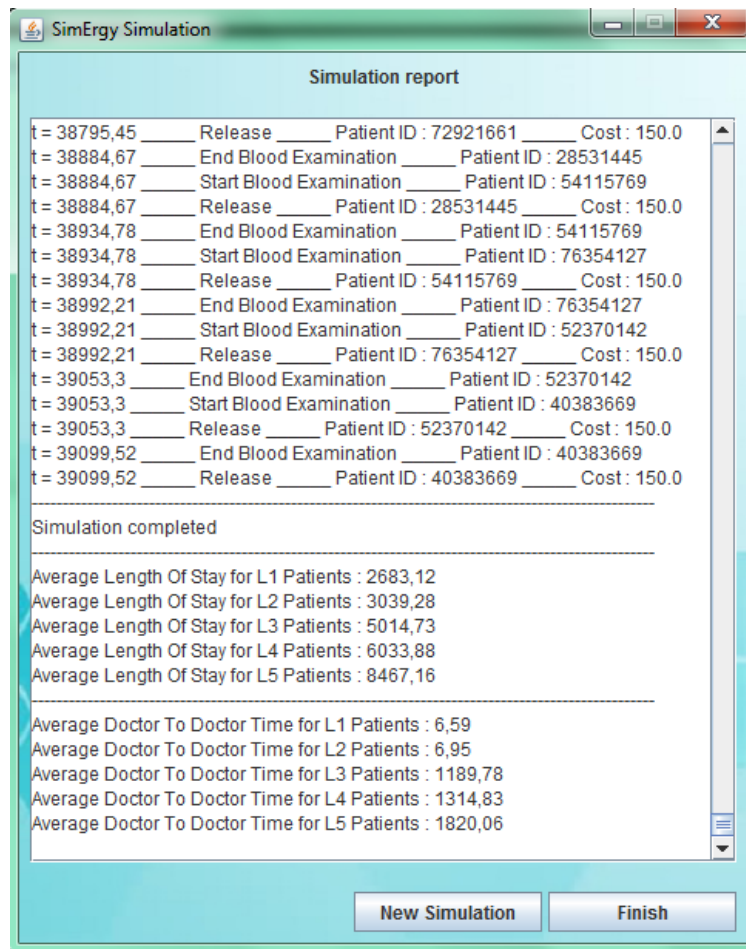


Figure 7.4: The OutputFrame graphical appearance

7.2 Support functionalities

We had to develop and use several classes to build our GUI platform. This sections aims to present these different classes and the functions they convey.

JTextAreaOutputStream

As we needed to print the simulation results directly in a Swing component (a JTextArea in our case) rather in the standard output console, we had to redirect the output streams. The basic idea is to create a subclass of the `java.io.OutputStream` class and to override its `write` methods. We did not know how to correctly provide such class and we used the code that we found in a website¹, after verifying that it matched our requirements.

BackgroundPanel

¹<https://java.developpez.com/telecharger/detail/id/1586/OutputStream-vers-un-JTextArea>

This class's only purpose is to improve the aesthetic appearance of our GUI. Indeed, we wanted to be able to put a given picture² as a panel's background. Therefore, we had to extend the `javax.swing.JPanel` class to override the `paintComponent()` method.

GuiDisplayer

This class makes it possible to fully test the Graphical User Interface. It contains a `main()` method that creates a `WelcomeFrame` when it is ran. Thereafter, you only have to follow the steps that are displayed to process the simulation or to end it.

To launch the GUI from `main()` method, we launched a separate thread in charge for running the GUI, as learned in class. This approach avoids that main thread gets stuck in case the GUI thread contains a deadlock.

²<https://ak4.picdn.net/shutterstock/videos/25969574/thumb/1.jpg>

8. JUnit tests

JUnit is a framework that allows testing in Java. Thanks to its simple implementation, this framework provides a useful platform to test our program's features. Indeed, JUnit's atomic definition enables to test each method of our software independently.

We did not use a *Test Driven Development* methodology as we started coding before mastering this useful feature. However, we planned to use the JUnit Framework to test as much code lines as possible in order to enhance our program's stability. Unfortunately, it was quite tedious to test some of our methods. The main reasons of these difficulties are the following:

- Our simulation process relies on probabilistic distributions for the events creation. Hence, our methods results were often random, which made it almost impossible to test.
- To test some features, we could either interrupt the simulation while it was processing or update the event queue manually. The former solution did not correspond to the way in which we implemented our code because once we run a simulation, it does not stop until every patient is released. For the latter solution, it was again very difficult to add events and to manage them manually to make a test. Hence, we could not develop relevant JUnit tests for some of our methods, especially the `updateEventQueue()`. Hence, we made some manual tests by printing intermediary results to see that these methods were running correctly.

However, we could test many other classes, such as all of the events classes. These operations allowed us to detect many bugs and to correct them. You can find all of the tests in the `fr.cs.simergy.junittest` package.

9. Conclusion

This project was for both of us the greatest and most complete programming experience that we ever had. Actually, we had a lot of fun implementing our code and progressing step by step. Obviously, our coding process was not as linear as one could imagine: we had many difficulties and we had to go back and forth trying to make the code functional and as practical as possible. We used to hear the following quote about programming: *“Measuring programming progress by lines of code is like measuring aircraft building progress by weight.”*. We finally could figure out why people were saying that, as we realized that making the code more lean was quite helpful in our progression.

The lectures and the tutorials that we had gave us a great insight of what Object Orient Software development was like, with many of its theoretical and practical principles. However, we faced numerous coding-specific challenges while developing our program and we felt that the Internet was like a gold mine for developers! Indeed, whenever we went to look for help to add new features to our program (especially the ones related to the IDE), we were happy to see that many other developers had the same issues and that they shared their solutions.

We are both quite satisfied with our project for two main reasons. The first one is that we feel that our Java and OOSD knowledge and reflexes improved sensibly thanks to this project. Plus, we consider that we delivered a code that functions correctly, meaning that it allows to run logically coherent simulations with user-friendly interface (CLUI and GUI). However, our project can evidently be improved in many points and we identified some of them:

- Adding waiting rooms with limited size for a more realistic implementation
- Adding new sorting strategies to the `severitySorting` that we used in order to compare their performances
- Ensure that the second consultation of a patient is made but the doctor who consulted him first (which could have been possible if we implemented the messaging service for physicians).

Bibliography

- [1] Helsgaun Keld, *Discrete event simulation in java*. Roskilde Universitetscenter, Datalogisk afdeling 2000.
- [2] Ballarini Polo, Lapitre Arnault, *SimErgy - Simulator for Emergency Department - v1*. OOSD, CentraleSupelec 2017.
- [3] Ballarini Paolo, *Object Oriented Software Design - Lecture Notes*. Laboratoire de MICS, CentraleSupelec 2016.
- [4] Eckel Bruce, *Thinking in Java - Third Edition*. 2010.