UNIVERSITÉ DE NANTES

# TER Report

## Refinement of communication protocols by models' transformation

---

Cousseau Axel · Haiti Aya · Juzdzewski Matthieu

Master 1 ALMA
Academic year 2019-2020

Supervisor : Pascal Andre
Team AeLoS, LS2N

Université de Nantes – faculté des Sciences et Techniques (FST)
2 Chemin de la Houssinière
BP 92208, 44322 Nantes Cedex 3

# Acknowledgements

We would like to express our sincere gratitude to our Supervisor Dr Pascal André from the LS2N(Laboratory of Digital Sciences) team for providing his valuable guidance, comments and suggestions throughout the course of this project. His assistance both theoretically and practically during our weekly meetings before lockdown allowed us to carry out our project and keep us on the right path during the hard period everyone was going through. We also would like to thank him for constantly motivating us to work harder and for being supportive while we encountered problems and difficulties.

We would also like to acknowledge the valuable scientific discussions with our colleagues from the "Refinement of statecharts in Java by model transformation" research group. Although our projects were complementary, these discussions helped us foster a scientific exchange at an early stage of the research.

Last but not least, we would like to thank Vivian Sicard for taking the time to manage the keys and giving us access to the LS2N facility.

The internship opportunity of working with a mentor, an experienced researcher in our case, was a great chance for us to hone our problem-solving skills. This project introduced us to the overall strategy of scientific work, from identifying the problem to designing the experiments and interpreting the results while planning a strategy to solve every problem that comes our way during the process.

# Table of contents

# Introduction

## 0.1 Introduction

During the second semester of our master's degree's first year (ALMA), we had to chose a subject among a given set and work as a 3-person team to get a first initiation of the world of research in computer science and to sharpen and test our skills.

We chose the following subject : "Refinement of communication protocols by models' transformation" which is a subpart of the much larger project "Automation in Model Driven Engineering". Supervised by Dr Pascal André, our goal was to study the transformation of models written in UML (mainly statechart and sequence diagrams) to usable Java code. More specifically, we focused on the communication aspect : the remote communication via wifi, bluetooth or USB and the exchange of messages within the system, like method calls for example. We take the UML message as an input and refine that message to the corresponding source code. A second team was tasked with transforming state machines into OOP models. As we need their work to complete our own, we will consider their part as already completed and we will manually code what we would have gotten from them.

So the goal of our team is to find a way to generate Java source code from the communication aspect of a given UML model. To get a physical support on which we can experiment and test our solutions, the LS2N has provided us with a complete LEGO Mindstorm kit and an EV3 controller. The path we have chosen to take, which will represent the three main chapters of this paper, can be resumed as follows : first we will try to understand and frame the problematic by building a simple garage door with the LEGO kit, remotely controlled through the EV3 connected to a computer via USB and wifi. Then we will analyse how the system works by experimenting different approaches and solutions for the refinement process. Finally, we will discuss our solutions and present our findings based on our experience.

# Getting started with the project

## 0.2 Project management

Below is the Gantt diagram presenting the various tasks we achieved throughout the semester. The diagram itself has been made after the facts, however we did have a general idea of what we needed to do. This diagram is just the representation of the informal timeline that we had in mind.

During the first part of the semester, we had team meetings every Friday to work on the project and set goals for the week after. We adapted our schedule during the lock-down with vocal meetings on Discord about twice a week and written communication for planning and details.

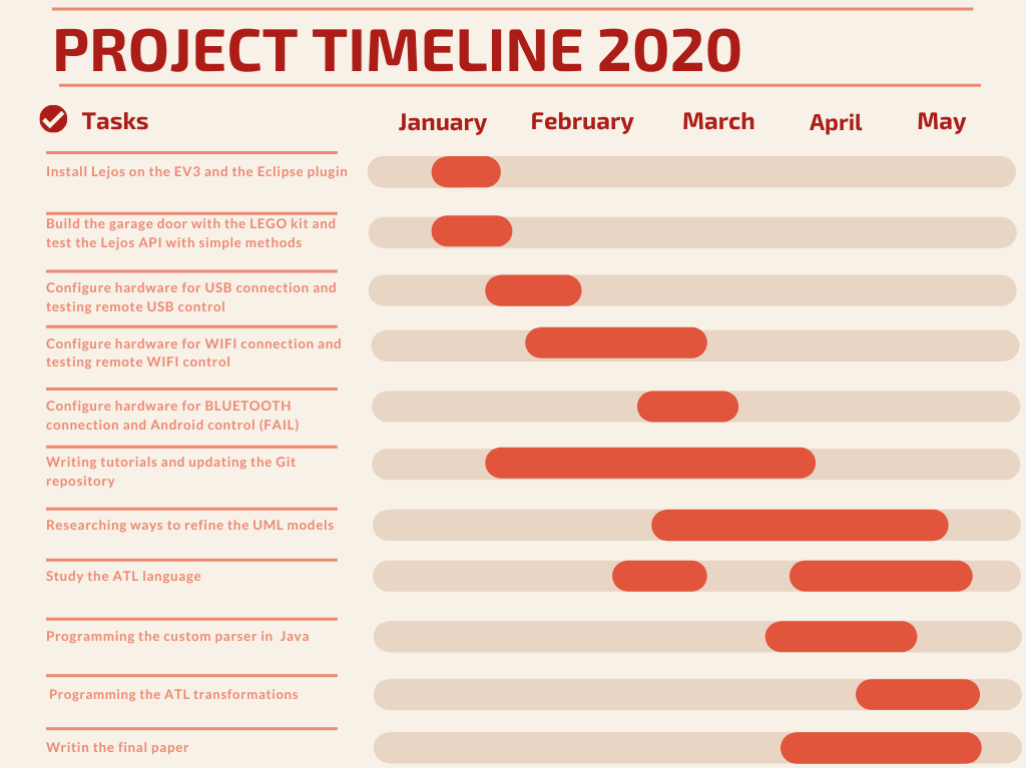



FIGURE 1 – Gantt diagram

## 0.3 Available tools

We have a LEGO EV3 controller and a full LEGO toolkit to start experimenting with a simple case study : a garage door. We will code with Java and Eclipse using the Lejos API. The EV3 needs a 2 to 32 GB SD card to install the Lejos virtual machine.


FIGURE 2 – EV3 Controler

## 0.4 Study case : garage door

In order to better understand the problem of automation and automatic code generation, we based our experiments on the concrete case of a garage door controlled by a remote control or application communicating with various possible means (Wifi, bluetooth, USB cable).

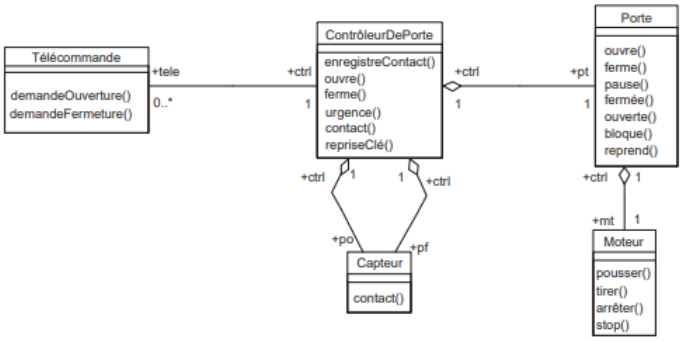



FIGURE 3 – Garage door class diagram

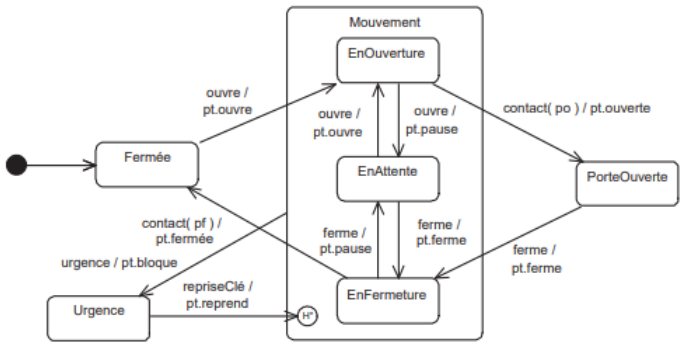



FIGURE 4 – Garage door state diagram

This garage door consists of a remote control, a sensor (open/close), a door and a motor. A controller manages these different parts. This example is perfect for our problem, because in addition to being very concrete and familiar, communications have a central place here.

We have chosen to simplify the model to work on the communications between remote control and controller (client and EV3 in our case) which will be remote communications, Wifi or USB, and the communications internal to the controller, i.e. method calls to control the motors or the sensor.

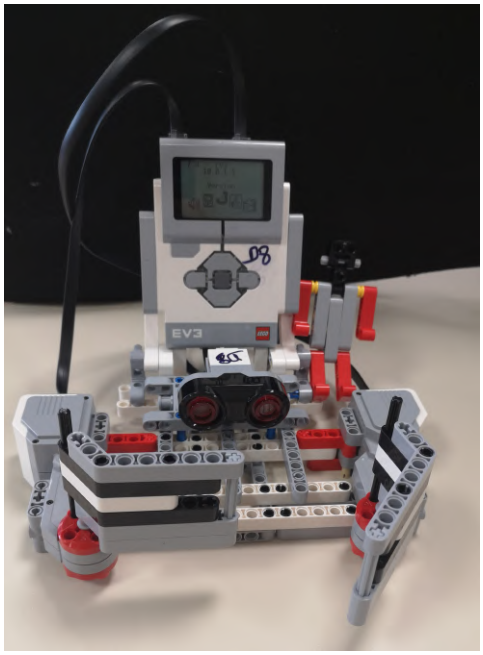
# Chapter 1

# Experimenting with the EV3 and Lejos

In this section, we will present how we built the garage door and the experiments we conducted using various communication protocols. Tutorials and advice on how to reproduce what we did step by step will be given, either in the next sections or in the annex part at the end of the paper.

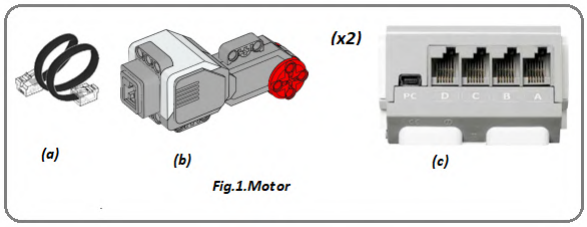
## 1.1 Building the LEGO robot

As it was the first time we encountered the Lejos framework and the LEGO hardware, we decided it would be best to simplify the study case and get going faster, with a simple but overall complete robot (enough for our purposes at least). Some decisions, specifically the building of the robot, are based solely on cosmetic, arbitrary reasons. This is why we decided to build a garage door with two moving panels instead of just one (as shown in the study case). This means that we have two motors, one for each door. We also decided to use a different kind of sensor to check whether the doors are open or closed. Below is a picture of the final garage door.

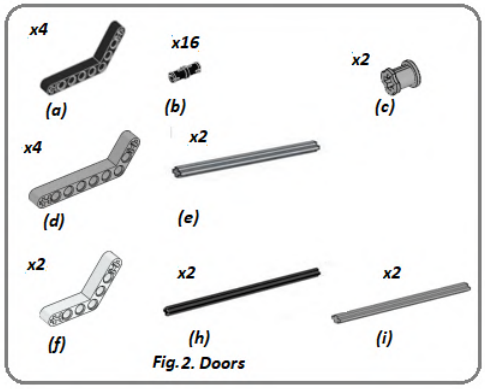



The robot consists of two motors, both of which incorporate a rotation sensor that is accurate to within a degree, optimized to serve as the motor base for the robot. They rotate with a rotational torque of 20 Ncm and a blocking torque of 40 Ncm, slow but powerful enough motors. It must also be taken into account that the action of the two motors is coordinated. The part « Fig.1.Motor (b) » is one of the two motors used. In order to control the forward or reverse rotation of a motor, the motor is connected to one of the four output ports of the controller « Fig.1.Motor (c) » using the flat cable « Fig.1.Motor (a)

». For this robot the motors have been connected to ports 'A' and 'D' as you can see in the code in the following sections.




Each of the doors is formed from two black angled beams with 4x6 modules « Fig.2.Doors (a) » and two grey angled beams with 3x6 modules « Fig.2.Doors (d) », and finally between the two pairs (grey, black), each is connected by two black friction pins « Fig.2.Doors (b) », a white beam with 4x4 modules « Fig.2.Doors (f) » is placed. Each door is fixed at its first end by a 10 black module pin « Fig.2.Doors (h) » and on the other by a 7 grey module pin « Fig.2.Doors (i) ». And finally with a grey 5-module shaft « Fig.2.Doors (e) » and a one-module ring « Fig.2.Doors (c) » each door is fixed to the motor's rotation axis. The final result of the doors is shown in « Fig.2.Doors (j) ».


The support that connects the motors to the controller and ensures the balance of the robot consists of the following parts. Firstly, by means of two red bent beams with 2x4 modules « Fig.3.Robot support (a) » fixed to each motor by means of a beige peg with 3 modules « Fig.3.Robot support (d) », the two motors are connected by means of 2 beams with 15 white modules « Fig.3.Robot support (e) », while the « Fig.3.Robot support (a) » is fixed to the motors by means of a red pin with 2 modules « Fig.3.Robot support (c) ». Thanks to the two 5x7 module frames, the grey « Fig.3.Robot support (f) » is connected to the rear part of the support with four black friction pins to the front part. The front part of the support is also fixed by two beams with 15 white modules using four black pins, two pins for each beam. In order to secure the front part that is closest to the rotation support of the two motors on the floor, two grey 13-module beams « Fig.3.Robot support (g) » are connected to each of the motors with 4 black pins on each side.

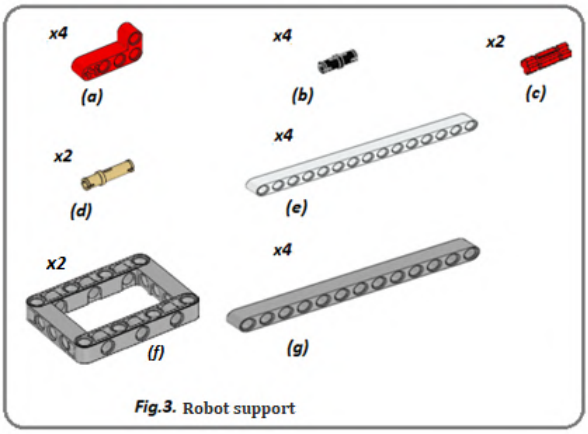


Fig.3. Robot support

In order to detect whether the doors are closed or open, an ultrasonic sensor has been installed to measure the distance between the sensor and the doors using reflected sound waves « Fig.4.Sensor (a) ». Like the engine, the sensor is also connected to the brick by the cable shown in « Fig.4.Sensor (b) » to one of the ports 1, 2, 3 or 4 « Fig.4.Sensor (c) ». The sensor is then attached to the bracket by the rest of the parts « Fig.4.Sensor (d)(e)(f)(g) ».

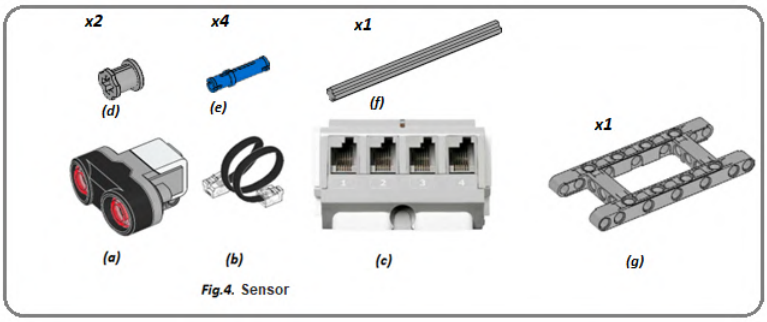
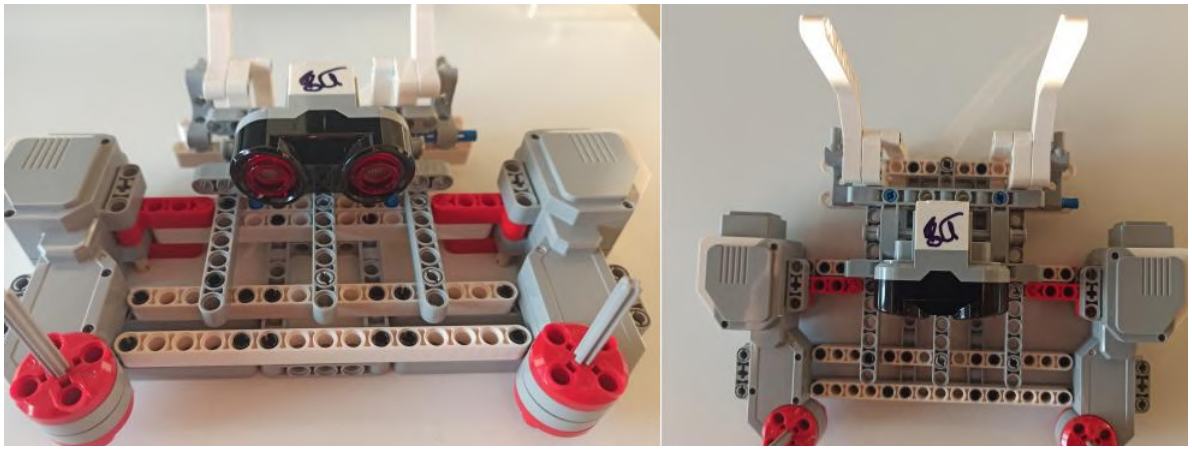


Fig.4. Sensor

Although the robot is functional with these components, we still chose to add a support in order to integrate the controller to the robot.



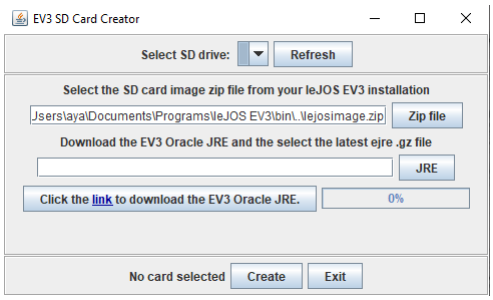
## 1.2 Installing Lejos and the eclipse plugin

Before you can start experimenting with the robot, you need to install Lejos on the EV3 controller as well as the Lejos plugin on Eclipse.

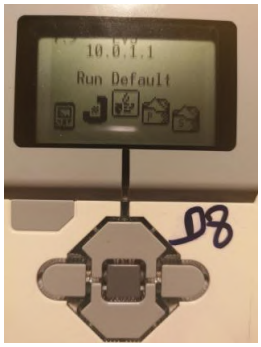
### 1.2.1 Installation of the LeJOS Framework on the EV3 brick

Originally, the EV3 brick boots on the Mindstorm OS. However, this one doesn't allow to program the behaviour of our robot in Java : so we have to install Lejos (which is a kind of Java virtual machine) on the EV3. This step can be tedious because of a sometimes ambiguous documentation and outdated tutorials. We have therefore described in detail the steps we followed ourselves. You will find this tutorial in « Appendix A » with solutions to possible problems.

First of all, in order to install the LeJOS Framework on the EV3 brick, you will need an SD card between 2 and 32 GB. We worked with a 2 GB card. Preferably, it should be empty and in FAT32 format. Then, you will have to download « leJOS_EV3 _0.9.0 beta » on the official website « lejos.org », more precisely by clicking on leJOS EV3 => Downloads. After launching the application and following the installation, the page on the figure below will be displayed.




You must also prepare your Java development space and the IDE that suits you, IntelliJ or eclipse for example. For the rest we chose eclipse. Once this is done, you will need to insert first a micro SD to the SD card adapter that you will then insert on your computer, and select your card on 'select SD drive'. Then, you just have to click on 'Click the link to download the EV3 Oracle JRE' which will redirect you directly to the oracle site for Java for LEGO Mindstorms EV3. The second download has been chosen : 'Oracle Java SE Embedded version 7'. And finally, before clicking on 'Create', don't forget to indicate the path to this last download for the JRE field. Now you just have to remove the adapter from the SD card and insert your card into the EV3 controller box and turn it on. A new default IP address will now be displayed : 10.0.1.1.




### 1.2.2 Installing the EV3 plugin on Eclipse

To be able to create a Lejos project on Eclipse, a Lejos project is not a traditional Java project, and manipulate the robot's features, you need to install the Lejos plugin on Eclipse. The tutorial is on the same file mentioned above.

#### 1.2.2.1 Méthodology

Generally speaking, the development process goes like this : In order to install the LeJOS plugin on eclipse to be able to run the upcoming code, it's simple, as for any other plugin, you go to the eclipse menu, Help and click on Install New Software. On the window that appears, enter in the first field the following URL : http://lejos.sourceforge.net/tools/eclipse/plugin/ev3. You then select

the LeJOS EV3 plugin and launch its installation, keeping the default settings afterwards. In order to connect to the EV3 brick later, simply go to Window then preferences and on LeJOS EV3, tick connect to named brick and enter the previous default IP address then Apply and OK. You can now create your first LeJOS EV3 Project by going to File, new Project, LeJOS EV3 Project and add a new library of the type « LeJOS Library Container ».

In order to quickly test if our installation is correct, we started by loading a simple program (see Appendix B) on the EV3 box. There was no user interaction, just an automatic activation of the motors.

## 1.3 Communication

In this part, we will discuss the remote communication aspect between the robot (here the garage door) and the controller (here a PC). In particular, we will see the technologies at our disposal : USB, Wifi and Bluetooth.

### 1.3.1 USB

After having prepared your working environment according to the previous chapter, it is time to get your EV3 controller and the USB cable provided in the Lego box. At first, we experimented with controlling the robot via USB. In order to establish any exchange between your PC and your EV3 controller, you will have to start by connecting the two of them with the USB cable. Once the controller is connected to your machine, unlike Linux where it finds and loads the driver, on Windows, you will have to do it manually. Once this is done, the controller is now connected to your computer and the driver takes care of managing the low level inputs and outputs to this device.

In order to load our program (Appendix D) on the EV3 box, we used Java sockets to implement the Server/Client behavior. A socket is a combination of an IP address and a port number that is used in the Layer 4 transport of the OSI model to establish the connection between the two devices. It is this connection that will allow the exchange of data packets using the USB (Universal serial Bus) communication protocol. When the connection is established, the client can then send orders (in the form of String) to the box which will interpret these Strings and call the corresponding methods.

However, it is once again necessary to configure certain parameters before the USB connection is recognized. A tutorial that explains in detail how to set up USB recognition is available in Appendix (see Appendix C).

Finally, it should be noted that the risk of USB communication is at the heart of its operating mode. Any application within the computer that can read the exchanges and damage the integrity of the exchanged data represents a major risk since there is no principle of rights as for applications on Android. There are some mechanisms that may protect from a malware attack and ensure the security of data being transferred outside the system's environment. Encryption is a part of these mechanisms ; it is used to encrypt files before transferring them to USB device as a part of the data loss prevention policy.

### 1.3.2 WiFi

The methodology for establishing a Wifi connection is essentially the same as for the USB part, i.e. a server program is loaded on the EV3 box (see Appendix D). It waits for a client to connect. A program is then executed on the PC (see Appendix G). It connects to the server and can then send commands. The only technical difference between Wifi and USB is the value of the IP address used.

After putting the three devices (Phone, PC and EV3 controller) on the same network by sharing the 4G connection on the phone, the connection between the PC and the box which is in server position on the network is then established, using the "Telnet" protocol. The Telnet protocol is a bidirectional exchange protocol of the application layer of the OSI model based on the TCP protocol that allows

communication with a remote server for an exchange in text format.
Since any communication using the Telnet protocol is plaintext over the network, multiple interceptors on the network can affect the security of network exchanges. It is for this reason that encrypted protocols such as SSH have been developed to replace Telnet to encrypt the data exchange and thus avoid any data breach.

Once again, it is necessary to first configure the hardware before you can claim to use wifi. In Appendix F you will find a tutorial that allows you to follow this configuration step by step. We have worked a lot to make the wifi work, and tried many different methods. Therefore, our tutorial is a mixture of all our discoveries, and it may not be *strictly* accurate. However, it does give a general idea of how to proceed.

### 1.3.3 Bluetooth

The latest remote communication technology at our disposal is Bluetooth. We started by developing a basic Android application that contains basic commands from the following buttons : connect, open, close and quit. This was done in order to test another control tool but several configurations were necessary. We were not able to get Bluetooth communication working in this time before confinement (February 2020) and the person who currently owns the robot does not have the necessary hardware to move forward on that side. Therefore, we leave this part open for future groups.

## 1.4 Results

To conclude the experimentation part with the EV3 box and the LEGO robot, we can notice that the biggest difficulty is to install and configure correctly all the necessary parameters and software. We hope that the tutorials we have written will be useful for the groups ofthe following years to get started more quickly. To sum up, we managed to control the robot remotely via a USB and Wifi connection. We also experimented with the basic methods of the Lejos libraries, including the use of motors and the ultrasonic distance sensor.

Our implementation of the garage door differs from the UML model proposed by our supervisor Pascal ANDRE, in the sense that we have a double-leaf door that does not use a pressure sensor but an ultrasonic distance sensor. The rest of this report is based on our implementation of the door, but it is quite possible - we think - to easily migrate to a model closer to the initial UML scheme with pressure sensors.

# Chapter 2

# Automation

After experimenting with the robot and the Lejos API, we were ready to start thinking about the automation of code generation from UML models. We did try to approach the problem from a purely theoretical perspective, but at the time we were still unclear about how to do it and where to start. So we decided to try an other approach : on one hand, we worked backward from the final Java code to the initial UML model while on the other hand, we still tried to refine the UML models.

## 2.1 Methodology

Before we begin, here is a reminder of our goal : we want to be able to generate automatically complete or partially complete Java code from one or more UML models. It is important to note that the goal is not necessarily to generate **all** the code but as much as possible, especially the repetitive or common sections we often find from one project to the other. Here is a representation of the work methodology we discussed above.

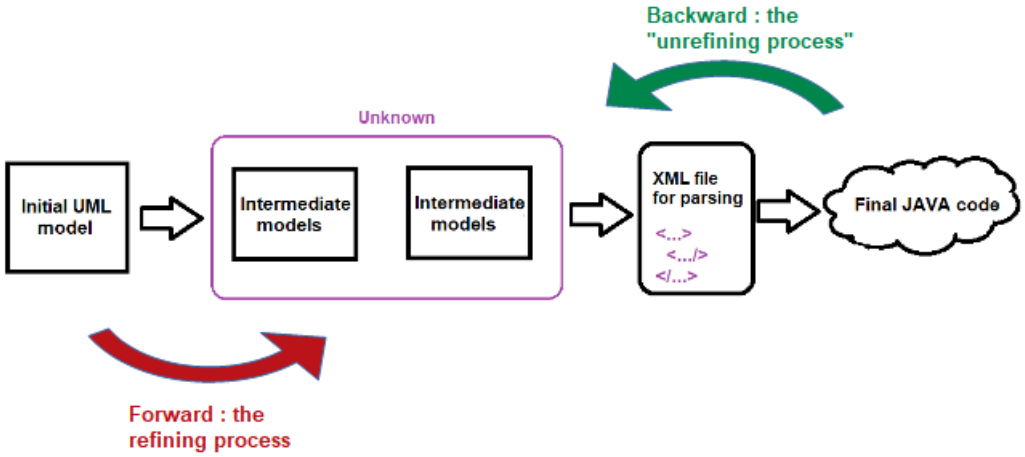



FIGURE 2.1 – Research workflow

The idea behind this methodology is to have a more practical minded approach that going from finish to start allows. This way, we can find out the different constraints and devise solutions as we go. It is also less confusing because we start from a position of strength (we are familiar with the final Java code and we can explain it because we wrote it ourselves) and we progressively get to higher levels of abstraction.

The first idea is to have a link between the Java code and the UML models that we understand so that we can quickly start working towards our goal. We chose to use an XML file : it is flexible, yet powerful, and last but not least it is how UML models are represented internally. So from the backward perspective, we want to have a representation of the final Java code as an XML file that contains all the data necessary. Then we want to abstract this XML file until we have the initial UML model. Basically,

it is a matter of "*unrefining*" : we look at the information that we currently have that the initial model doesn't, and we cut it. By finding out in which order we can do this, we get layers of refinement.

This means that from the forward approach, we should be able to go through these layers, refining and adding information to finally get to the XML link. At this point, a simple parser is enough to generate the Java code by mapping tags and attributes with pre-written code. The process has two main parts :

— refining the initial UML model until we have the XML link

— parsing the XML file to get the Java code

Following is the presentation of what we managed to achieve from a technical point of view. Obviously, there is still a lot more to do and we will discuss our findings in the next and last chapter. We will see what remains to be done and improved, what are the layers of refinement that we identified and other considerations for future development. But first we present here what we currently have which is a simple, but functional, transformation method from an initial UML model to an executable Java code.

## 2.2 UML to XML

For this part, we decided to use sequence diagrams as source files, believing that it suggested a maximum of communications between the different actors.The goal here is therefore to transform this source file into an XML file created to be simple and composed of the only information needed to generate Java code, based on the information found in a UML diagram. To do this, we used ATLAS Transformation Language (ATL), a model transformation language available as an Eclipse plugin. To carry out the complete transformation process we proceeded in several steps : first the creation of the UML model (sequence diagram) and its XML correspondence, then the creation of the metamodels representing our source and output files, and finally the writing of the ATL transformation rules.

### 2.2.1 Creation of the UML file

To model our UML diagram we used the Eclipse Papyrus plugin, a graphical editor for UML. Here we have modeled a very simple sequence diagram based on the model used in our experiments. We can see the client's connection message, and the different possible method calls. Once this diagram is finished we retrieve the .uml file generated by Papyrus in the same directory, it is a file describing our diagram in XML format, it will be used as source file during the ATL transformation.

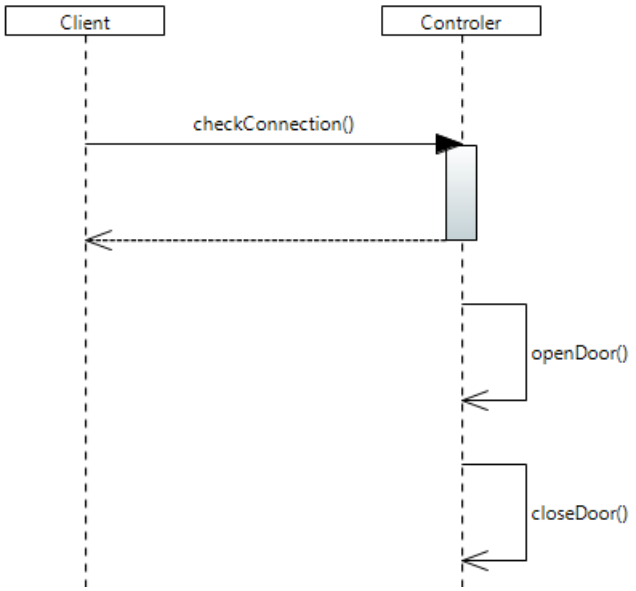



FIGURE 2.2 – Sequence diagram representing the study case

### 2.2.2 Creation of metamodels

A metamodel is an abstraction to describe the structure of a model. To perform a transformation with ATL we need the metamodels of the input and output models to describe exactly what it will be possible to have as input and how to transform it into an output. Here for example for our .uml file representing the sequence diagram, each tag in the file is represented by a class in the metamodel, and if a tag contained another one, its class will contain an attribute corresponding to the class of this tag in the metamodel. To realize these metamodels we use .ecore files, an EMF model. The metamodel thus made must be able to match any input sequence diagram without modifications.

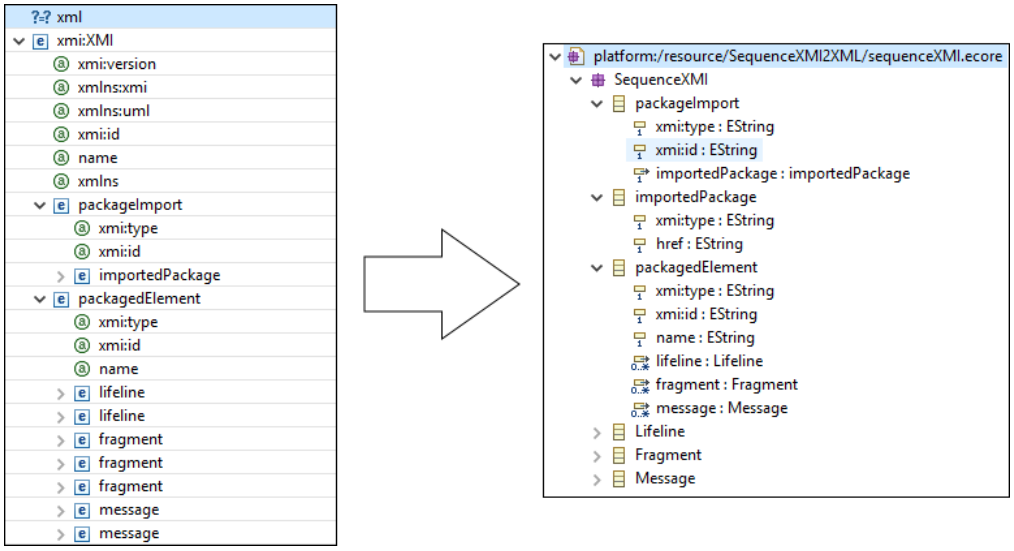



FIGURE 2.3 – xml model to ecore model

In the same way, we produce a metamodel representing the XML file used to generate the code.

### 2.2.3 Writing transformation rules

The ATL transformation part was the most complicated, as it required a lot of research and documentation concerning the use and syntax of the ATL language. In our ATL file we specify which are the input and output models, using the metamodels created in the previous step to inform ATL about what we have at the beginning and what we want as a result, and then we write the transformation rules. The goal is to link each part of the output model to a part of the source model, and to "explain" how to create them.

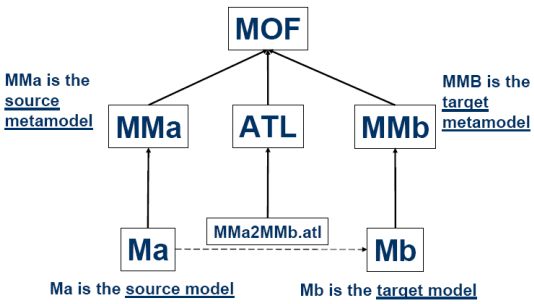



FIGURE 2.4 – ATL Context [1]

A rule is first composed of a "source" section, with the keyword *from*. This is used to inform the type of the element used and its model, we can also add conditions to restrict the set of elements affected by the rule, and we can declare local variables to the rule.

The rule then consists of the "target" section, with the keyword '*to*, the type of target element and its model are filled in. Note that several targets can be created for the same source.

```
lazy rule lifeline2class{
    from
        s : SequenceXMI!Lifeline
    to
        t1 : XML!Class(
            name <- s.name,
            hardware <- '',
            Fonctions <- t2,
            ConnectionInfo <- t3
        ),
        t2 : XML!Fonctions(
            fonction <- SequenceXMI!packagedElement.allInstances().first().message
                ->select(m | m.isContained(s.coveredBy))
                ->collect(m | thisModule.message2fonction(m))
        ),
        t3 : XML!ConnectionInfo(
            type <- '',
            role <- '',
            ip <- '',
            port <- '',
            inputType <- ''
        )
}
```

FIGURE 2.5 – ATL Rule Transforming a Lifeline element into a Class element in the output XML

Here for example, for each Lifeline type element of the SequenceXMI source model, three elements will be created in the target model : Class, Functions, ConnectionInfo. The attributes of these elements are then filled in either by the content of other targets as seen in "Class", or by elements already present in the source model, or by the result of other rules or Helper as in "Functions". A Helper is a method which can be defined and then called in ATL.

Note that in order for ATL to recognize the source file, and for the transformation to work, it was necessary to replace the *<uml :Model>* tag by a tag *<xmi :XMI>*, it was also necessary to add the "xmlns=[MetaModelSourceName]" attribute. These are the only modifications made on the source file after its creation in Papyrus.

### 2.2.4 Result

We were able to generate an XML file close to the expected result, nevertheless some parts are missing because no information about them is filled in the source UML diagram. This is the case, for example, of the connection information (IP, port, type), which will probably have to be added by hand or using a configuration file, but also of the name of the action assigned to each function, and the content of the Mapper specific to the connection and the function call. To go further, one may wonder if it is possible to add this missing information using other transformations.

Concerning the realized transformation, it works on sequence diagrams, another possible opening is to search if it is possible to generalize the ATL transformation so that it can take as input any UML diagram, or if it is mandatory to realize different transformations for each diagram in order to reach the more global XML model we have defined.

## 2.3 XML to java

As discussed previously, this is the last step of the code generation process. In the previous step, we refined the UML model to the XML link using the Atlas Transformation Language. This last XML file now contains all the data necessary to generate the code. The only thing left to do is to use the data contained in the XML file to generate the code. To do so, we decided to write our own custom program : we will feed the XML file as an input and the program will generate the code (in the form of a .txt file at the moment). Our program is heavily **component-oriented** : each tag from the XML file

is processed by a dedicated specialized class. Each class only cares about its role and nothing else. The result is progressively stored in a List<String> where each element is composed of similar data :

— At the index 0, we find all the import statements. Whenever a class needs an import, it will append it to this String.

— At the index 1, we find the declaration of the class.

— At the index 2, we find the *main* function.

— At the index 3, we find all the necessary functions (called *actions* in the XML file) as well as the declarations of all the global variables.

— At the index 4, we find the closing curly bracket of the class.

Here is an overview of how the program acts :

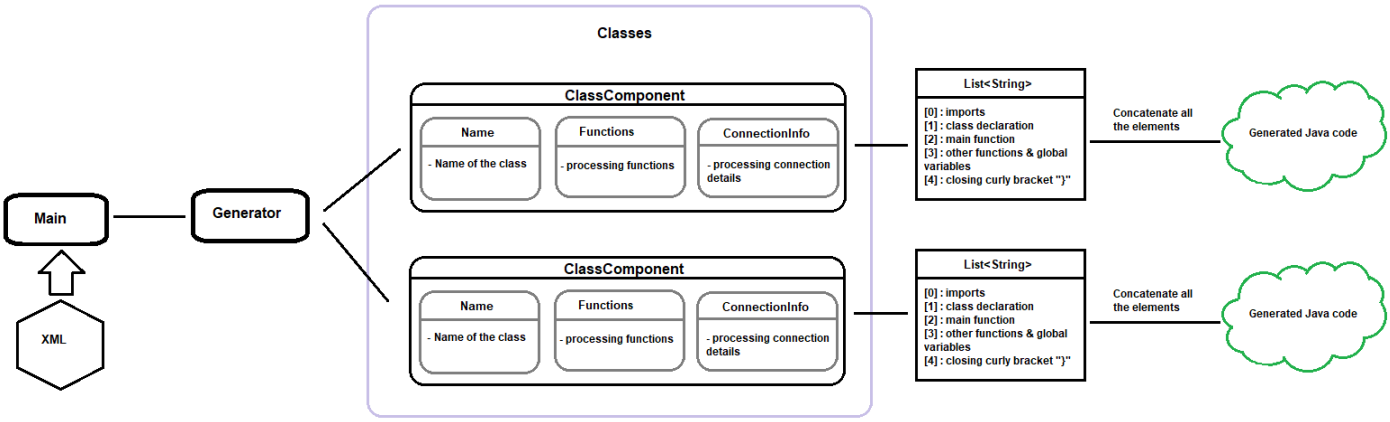



FIGURE 2.6 – Overview of the program's logic

The component-oriented workflow allows for an easier understanding of the program as well as an easier maintenance and perspectives of evolution. Each component contains already written portions of code that will be extracted and imported into the result List when the matching tag or attribute is found. The *FunctionDefinition* class acts as a kind of database containing already written basic functions. Those functions can be accessed using an identifier like A100 or A101 for example (at the moment, those identifiers represent the methods that pull and hire the motors of the garage door). They are stored in a HashMap<String,FunctionDefinition>. Adding new functions or components is just a matter of creating a new class and defining the required behaviour. This means that the program could be considerably extended over time, giving more and more possibilities as we feed it new components.

When all the tags from the XML file have been processed, each element of the List<String> containing the result is concatenated in a single String. That String contains the final code we want. At this point, we are only printing the result in the console, but we could just as well write to a .java file for example.

# Chapter 3

# Results and perspectives

In the previous chapter, we presented a complete transformation from a simple UML model to the final executable code. However, the work is far from complete. The goal was to identify the different steps of refinement while having a functional system to rely on. It is not meant to be the foundations of the complete project but rather to maybe give an insight on what has to be done. Keeping this fact in mind, we will now present the advantages and problems of our method. Finally, we will try to paint a general picture of the different refinement steps we have identified.

## 3.1 Discussing our method

As we said in the previous chapter, the program we wrote can be extended to allow for more possibilities. However, adding new components or tags/attributes will imply that the meta-models and rules we defined for ATL will have to be updated. As the complexity grows, so will the management of the ATL transformations. Unless a purely compartmentalized solution is found, this can become a tedious task. That being said, we can think of our method as unstable at the moment, but it would probably reach a point in time when no more heavy changes are necessary, therefore reaching its "mature", stable form. Moreover, we believe that this tedious "*growing process*" is inevitable, whichever solution is used in the future.

There is also a problem of flexibility with our method. The List<String> allows easy text insertion at the end and the beginning of the String, but there might come a time when one will need to insert text in between specific lines. At this point, an other data structure should be considered, like a tree for example. Or the parser could be abandoned entirely. Indeed, the final transformation could be realized through ATL. We used a custom java program to better understand the problematic and start faster, be this is in no way the only solution (and most likely not the best).

## 3.2 Refinement of UML models

We will now present the refinement steps that we have identified throughout the project. First of all, and we didn't explicitly mentioned it until now, everything can't be fully automated. Or rather it would go against the primary goal of this project which is the simplification of the development and maintenance process of a software using model driven engineering (especially distributed systems). Indeed, some data require user input, like the IP address or the port number in the case of TCP/Ip communication for example. We could feed it somehow into the model itself, but we think it is much faster to just provide that information manually, either in the final model, or in the produced code. However, there is some data that must be provided during the refinement process, decisions must be made, and this brings us to the presentation of the main steps. It is important to note that we only worked with sequence diagrams. The use of other complementary UML models could provide the data needed with minimal user input :

— **Identification of the environment** : the idea is to define which classes work together on the same hardware (meaning they don't communicate remotely, we will call them a *cluster*), which class is the main class, and which class will manage the remote communications (if any), as well

as its role : server or client. This is an important step because we need to know where to create the *main* function, the global variables referencing the other classes as well as the *ConnectionInfo* component.

— **Specification of the environment** : Now that we have defined the overall structure of the clusters, we can add data such as the protocols used for remote communication, the type of messages sent over the network (will it be text, integers, bytes, etc) and what do those messages map to. In terms of code, this step will mostly populate the *main* functions and the *InputMapper* tag of the final XML file.

— **General tasks** : by this point, most of the communication aspects have been dealt with. This step will mainly create the functions themselves.

This is how we conducted the "removing" process from the backward perspective : going from the last point to the first, the information is more and more abstract.

# Conclusion

The automation of code generation from UML models is a complex problem. There are many factors to consider and some of them fight the purpose of the others. If the solution is flexible and as complete as possible, the complexity of the refinement process is increased, as well as the number of inputs and decisions the user has to make during the transformation process. Spending that much effort (from the user point of view) into the refinement process might go against the primary goal which is speed and simplicity. On the other hand, if the solution is too general and rigid (say we only generate the skeleton of the classes, methods and some very basic code) then the user will still have a lot of programming to do. The complexity stems from the balancing act we have to do, finding the "sweet spot" between these two extreme cases. The work that we have presented in this paper is far from complete. We have chosen to take a step in a direction that we don't know where it leads to, or if it is even worth pursuing as is. Nevertheless, we have tried to make our solution fully reproducible (through this paper and our Git) so that future teams will be able, we hope, to start working faster and from a higher point of view.

# Glossary

— **ATL** : ATLAS Transformation Language, a model transformation language available as an Eclipse plugin
— **LS2N** : Laboratoire des Sciences et du Numérique de Nantes
— **EV3** : the hardware used for controlling the robot
— **EMF** : Eclipse Modeling Framework
— **JRE** : Java Runtime Environment
— **Ncm** : Newton centimeter
— **OOP** : Object-oriented programming
— **SD** : Secure digital
— **UML** : Unified Modeling Language
— **XML** : Extensible Markup Language

# Bibliographie


[1] ATL User Guide, The ATL Language
https://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language

[2] ATL User Guide, Tutorial ATL Transformation
https://wiki.eclipse.org/ATL/Tutorials_-_Create_a_simple_ATL_transformation

[3] Document Java - Interface Document https ://docs.oracle.com/javase/7/docs/api/org/w3c/dom/Document.html

[4] LeJos EV3 - Documentation
http://www.lejos.org/ev3/docs/

[5] Wikimedia Commons - ATL Context
https://commons.wikimedia.org/w/index.php?curid=4307249

# Annexes

## 3.3 Annexe A

**Tutoriel d'installation du LEJOS firmware et de LeJOS plugin sur eclipse**

Le premier pas consiste à installer le Lejos firmware sur le boitier EV3. Pour cela, munissez-vous d'une carte SD entre 2 et 32 Go. De préférence, elle doit être vide et au format .FAT32. Suivez ensuite le tutoriel (vraiment, soyez attentif même si vous pensez avoir compris.. ne résinstallez pas Eclipse non plus, ya des limites) sur cette vidéo : https://www.youtube.com/watch?v=ycOA4wnvAyM Vous devriez maintenant avoir le boitier EV3 qui boot sur LEJOS au lieu de mindstorm et le plugin Eclipse installé. Si c'est le cas, bravo, c'est la fin de ce tutoriel, passer directement à la section "FIN". Sinon, peut-être la suite pourra vous aider :

Problème pour installer LEJOS sur le boitier :

Nous avons aussi eu des problème au début, et nous avons testé avec une autre carte SD, et tout fonctionnait après ça. Je ne peux que vous conseiller de suivre le tutoriel vidéo à la lettre (si ça ne fonctionne toujours pas, regardez la section suivante sur le PLUGIN ECLIPSE).

Problème pour installer le plugin LEJOS sur Eclipse :

Si vous avez des problèmes avec cette étape, encore une fois, suivez le tutoriel à la lettre, DEPUIS LE DÉBUT. Lorsque vous avez installé le plugin via le menu Eclipse Help => Install New SoftWare, il faut absolument aller voir dans le menu : Window => Preferences => Lejos EV3 Regardez le champ EV3_HOME. S'il est vide, indiquez le chemin du dossier où vous avez installer LEJOS EV3 (en suivant le tutoriel).

FIN

Normalement, vous devriez maintenant avoir le boitier qui boot sur le firmware LEJOS et le plugin LEJOS sur Eclipse. Pour créer un projet LEJOS sur Eclipse, faites File => New => Projet => Lejos EV3 => Lejos EV3 Project (Si une erreur du style "Build path problem" arrive, vous n'avez pas correctement lié EV3_HOME comme vu précédemment.) Allez dans Windows => Preference => LEjos EV3 et cochez Connect To names Brick et entrez dans le champ l'addresse 10.0.1.1. Validez. C'est terminé, vous êtes prêt pour la suite !

## 3.4 Annexe B

**Code de Test**

```
package org.testings;
import lejos.hardware.motor.Motor;
import lejos.hardware.port.SensorPort;
import lejos.hardware.sensor.EV3UltrasonicSensor;
```

```
import lejos.robotics.SampleProvider;
import lejos.utility.Delay;
public class Test1{

        public static void main(String[] args) {
                EV3UltrasonicSensor uSensor = new
                   EV3UltrasonicSensor(SensorPort.S2);
                SampleProvider sampleProvider = uSensor.getDistanceMode();
                float[] sample = new float[sampleProvider.sampleSize()];
                sampleProvider.fetchSample(sample, 0);
                System.out.println(sample[0]);
                while(true){
                        if(sample[0]>8){
                                System.out.println("OUVERT");
                                break;
                        }else{
                                System.out.println("FERME");

                        }
                        sampleProvider = uSensor.getDistanceMode();
                        sample = new float[sampleProvider.sampleSize()];
                        sampleProvider.fetchSample(sample, 0);
                        Delay.msDelay(1000);
                }
                Delay.msDelay(20000);
                Motor.A.setSpeed(90);
                Motor.D.setSpeed(90);
                Motor.A.forward();
                Motor.D.backward();
                Motor.A.rotateTo(-90);
                Motor.D.stop();
                Motor.A.stop();

        }

}
```

## 3.5 Annexe C

**Tutoriel de connexion du boitier au PC via USB**

Ce tuto suppose que vous avez correctement installé le plugin Eclipse, que vous avez créer un Projet Eclipse LEJOS et que votre boitier boot sur le LEJOS firmware. Si ça n'est pas le cas, voir tuto "Installation LEJOS firmware + plugin EClipse".

Munissez vous de votre boitier Ev3 et du cable USB fourni dans la boite LEGO. **BRANCHEZ VOTRE EV3 à votre PC pendant tout ce tuto.**

Ce tuto suppose également que vous opérez sous Windows 10, si ça n'est pas le cas, dommage !

Tout d'abord, il faut savoir que lorsque vous connectez votre boitier Ev3 à votre PC, Windows 10 semble le reconnaitre et installer des drivers, mais c'est faux... Complétement faux ! Il va vous falloir installer un driver particulier par vous-même : le RNDIS !!. Pour ce faire, lisez d'abord les lignes "A lire" qui suivent , vous pouvez téléchareger le DRIVER à partir du dossier DRIVER sur le git ter-ir-2020 Transfo-

protocolesTransfo-protocoles si vous y avez accès.

**A lire** : Créez un nouveau dossier "InsertRelevantNameForU" à la racine de votre disque dur. Extrayez-y les fichiers se trouvant dans le dossier DRIVER du Git. Cliquez droit sur le fichier d'extension .inf et selectionnez "Installer". Soyez consentant aux éventuelles popup. C'est fait ? Okay, alors suivez maintenant ceci :

- Windows + R
- devmgmt.msc
- Etendez "Carte réseau"
  - vous devriez voir quelque chose du genre "USB Ethernet/RNDIS Gadget"
  - Clique droit dessus
  - Propriétés => Avancé
  - Sélectionnez "Valeur" puis entrez dans le champ "10.0.1.1" et validez moi tout ça

Bravo, c'est terminé ! Maintenant il est temps de tester la connexion :

- Windows + R
- cmd
- ping 10.0.1.1

Vous devriez avoir une réponse avec 0

Si c'est le cas, vous avez correctement connecté votre boitier à votre PC via USB. Reste maintenant à upload un programme sur ce boitier Ev3.

J'ai suppososé au début de cet article que vous aviez déjà un projet Eclipse correctement Setup en mode Lejos project. Si ça n'est pas le cas, voir le tutoriel "Installation LEJOS firmware + plugin EClipse". C'est bon ? Créez maintenant une classe en cochant la case "public static void main" . Écrivez quelque chose de simple comme LCD.drawsString("BONJOUR EV3",0,0) (Importez ce qu'ecplise vous indique, soit import lejos.hardware.lcd.LCD ; dans ce cas). Maintenant nous sommes fin prêt pour importer ce beau programme sur le boitier. Pour ce faire, cliquez droit sur votre classe => Run As ==> lejos EV3 Program.

Votre boitier devrait vous implorer d'attendre 1 seconde. Puis vous devriez voir apparaitre "BONJOUR EV3" sur l'écran du boitier.

**Pas de panique si ça n'est pas le cas !** Moi non plus ça n'a pas fonctionné du premier coup...

Votre boitier fait un joli bruit avec une sympathique erreur qui ne fait pas sens pour vous ? Plongeons dans le magnifique monde des JRE ! Car là se trouve le problème. (Appuyez sur le bouton retour sur le boitier pour quitter l'écran d'erreur)

J'espère pour vous que avez la version 1.7 du JRE java. Sinon il va falloir l'installer. Cliquez droit sur votre projet LEJOS. Selectionnez Properties => java compiler

Puis Changer le JRE pour la version 1.7, et validez. Maintenant réessayez d'upload le programme.

Si ça marche, bravo, vous avez correctement lié votre boitier à votre PC ! Sinon, suivez bien les étapes dans l'ordre et bonne chance

## 3.6 Annexe D

**Code source à exécuter sur le boîtier pour une connexion USB et WIFI**

```
package org.client;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

import lejos.hardware.lcd.LCD;
import lejos.hardware.motor.Motor;
import lejos.hardware.port.SensorPort;
import lejos.hardware.sensor.EV3UltrasonicSensor;
import lejos.robotics.SampleProvider;


public class TestUSBcontrol {
        private static EV3UltrasonicSensor uSensor;

        public static boolean estOuvert(){
                SampleProvider sampleProvider = uSensor.getDistanceMode();
                float[] sample = new float[sampleProvider.sampleSize()];
                sampleProvider.fetchSample(sample, 0);
                if(sample[0]>8){
                        return true;
                }
                return false;
        }

        public static void main(String[] args) {
                String clientRequest;
                String serverResponse;
                uSensor = new EV3UltrasonicSensor(SensorPort.S2);
                try(ServerSocket socket = new ServerSocket(5555))
                {
                        while(true)
                        {
                                LCD.drawString("Waiting", 0, 0);
                                Socket connectionSocket = socket.accept();
                                LCD.clear();
                                LCD.drawString("Connected client", 0, 0);
                                BufferedReader reader = new
                                    BufferedReader(new
                                    InputStreamReader(connectionSocket.getInputStream())

                                OutputStream output =
                                    connectionSocket.getOutputStream();
                                PrintWriter writer = new
                                    PrintWriter(output,true);
                                String text;

                                do {
                                        text = reader.readLine();
                                        if(text.equals("1") &&
                                            !estOuvert())
```

```
                                        {
                                                // commenter ces deux
                                                   lignes
                                                LCD.clear();
                                                OpenDoor();
                                        }
                                        else if(text.equals("2") &&
                                           estOuvert())
                                        {
                                                //commenter ces deux lignes
                                                LCD.clear();
                                                CloseDoor();
                                        }

                                }while(!text.equals("leave"));

                                socket.close();
                        }
                }
                catch(IOException e1)
                {
                        e1.printStackTrace();
                }
        }
        public static void OpenDoor()
        {
                Motor.A.setSpeed(90);
                Motor.D.setSpeed(90);
                Motor.A.forward();
                Motor.D.backward();
                Motor.A.rotateTo(-90);
                Motor.D.stop();
                Motor.A.stop();
        }

        public static void CloseDoor()
        {
                Motor.A.setSpeed(90);
                Motor.D.setSpeed(90);
                Motor.A.forward();
                Motor.D.forward();
                Motor.A.rotateTo(0);
                Motor.D.stop();
                Motor.A.stop();
        }


        }
```

## 3.7 Annexe E

**Code source de contrôle à lancer sur le PC pour une connexion USB**

```
import java.io.IOException;
```

```
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Scanner;
import java.io.OutputStream;
import java.io.PrintWriter;

public class ClientUSB {

        public static void main(String[] args) {
                try(Socket socket = new Socket("10.0.1.1",5555))
                {
                        OutputStream output = socket.getOutputStream();
                        PrintWriter writer = new PrintWriter(output,true);
                        Scanner sc = new Scanner(System.in);
                        String text;

                        do
                        {
                                text = sc.nextLine();
                                writer.println(text);
                        }while(!text.equals("leave"));

                        socket.close();
                }
                catch(UnknownHostException e)
                {
                        System.out.println("Server not found");
                }
                catch(IOException e)
                {
                        System.out.println("I/O error");
                }




        }

}
```

## 3.8 Annexe F

**Tutoriel de connexion du boitier au PC via WIFI**

Ce tuto aussi suppose que vous avez correctement installé le plugin Lejos EV3 sur Eclipse, et que vous avez créé un Projet LEJOS et que votre boitier boot sur le LEJOS firmware. Si ça n'est pas le cas, voir tuto "Installation LEJOS firmware + plugin EClipse".

Dans un premier temps, munissez-vous uniquement de votre boitier Ev3 et de votre PC et veuillez suivre les étapes suivantes :

ETAPE 1) Brancher l'adaptateur Wifi dans le port USB Host de la brique EV3.

ETAPE 2) Allumer la brique EV3.

ETAPE 3) - dérouler le menu "outils"

- sélectionner "Wifi" ;
- cocher la case "Wifi" ;
- sélectionner "Connections" ;
- sélectionner le point d'accès Wifi dans la liste, puis attendre ;
- saisir la clé WEP/WPA si nécessaire ;
- sélectionner "Connect" ;
- dérouler le menu "outils" ;
- sélectionner "Brick info" ;
- conserver l'adresse IP de la brique EV3 (par exemple 192.168.1.2) pour un usage ultérieur.

ETAPE 4) Il faut établir un partage de connexion 4G entre notre Android, PC et le contrôleur afin que ces trois appartiennent au même réseau. Une nouvelle adresse IP s'affichera à l'écran du contrôleur de la brick Lejos.

ETAPE 5) Définir le contrôleur Lejos comme serveur plutôt que client en déroulant et cliquant sur l'icône Wifi ensuite sur Access point pt+ et entrez l'adresse IP de votre brick et validez.

ETAPE 6) Dans la ligne de commande de votre PC lancez l'instruction : telnet suivie de la nouvelle adresse IP affichée sur la brick et rentrez comme login 'root' et sans mot de passe.

ETAPE 6bis) Si la commande Telnet n'est pas reconnu : C :>telnet google.com 80

'Telnet' is not recognized as an internal or external command, operable program or batch file.

**Cause** : Le client Telnet dans un Système d'exploitation Windows est désactivé par défaut.

**Solution-ligne-de-commande** : Lancez la commande suivante en disposant d'autorisations de niveau Administrateur :

dism /online /Enable-Feature/FeatureName :TelnetClient

**Solution-configuration-manuelle** : commencez par rechercher et cliquer sur 'Turn Windows features on or off' à partir du domaine de recherche près du Start button. Ensuite, sur la page qui s'affiche recherchez et sélectionnez Telnet Client et pour finir cliquez sur Ok.

Vous aurez par la suite la page de chargement des changements en titre : 'Apply Changes'. Une fois que les changements sont réalisés, vous pouvez enfin cliquer sur 'Close'.

ETAPE 7)root@EV3 : # dropbear

root@EV3 : # hcitool dev

Devices :

hci0 11 :22 :33 :44 :55 :

ETAPE 8) Lancer la Classe 'TestUSBControl', voir l'Annexe D

ETAPE 9) Attendre l'affichage de 'Waiting' sur l'écran du contrôleur qui marque l'attente d'une connexion.

ETAPE 10 ) Lancer la Classe 'Client WIFI', voir l'Annexe G

ETAPE 11) Pour Ouvrir le portail, entrez 1 sur votre Consol et pour le fermer entrez 2 et afin terminer la connexion entrez Leave.

## 3.9 Annexe G

**Code source de contrôle à lancer sur le PC pour une connexion WIFI**

```
package org.client;
import java.io.IOException;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Scanner;
import java.io.OutputStream;
import java.io.PrintWriter;

public class ClientWIFI {

        private static Scanner sc;

        public static void checkConnection(){
                String text ="";
                sc = new Scanner(System.in);
                do
                {
                        text = sc.nextLine();
                }while(!text.equals("connect"));

        }

        public static void main(String[] args) {
                checkConnection();
                try(Socket socket = new Socket("192.168.43.37",5555))
                {
                        OutputStream output = socket.getOutputStream();
                        PrintWriter writer = new PrintWriter(output,true);
                        String text;

                        do
                        {
                                text = sc.nextLine();
                                writer.println(text);
                        }while(!text.equals("leave"));

                        socket.close();
                }
                catch(UnknownHostException e)
                {
                        System.out.println("Server not found");
                }
                catch(IOException e)
                {
                        System.out.println("I/O error");
                }
        }
}
```