Differential Drive Robot System

Marco Carlo Cavalazzi
Matteo Corradin
Lorenzo Monti

Alma Mater Studiorum – University of Bologna
via Sacchi 3, 47023 Cesena, Italy
marcocarlo.cavalazzi@studio.unibo.it
matteo.corradin2@studio.unibo.it
lorenzo.monti5@studio.unibo.it

Abstract. Building a software system implies to realize something characterized by multiple properties, like: the observable operative behaviour described using "interaction stories" that specify the received and transmitted messages; the processing, often included in a contest of interaction between the system and the outside world; the system, which is open and operates online.
The software engineering presents, by nature, multiple difficulties. The realization of a complex system, heterogeneous and distributed, increases such difficulties introducing numerous questions that have to find an answer through the whole life cycle of a software system. How can the system be broke down? How can the sub-systems be decomposed? How do the entities interact with each other? These are just few of the many question that concern the three fundamental coordinates: structure, behaviour and interaction.
There are typically two main approaches to the development of the software system: Model-Driven Engineering (MDE), methodology characterized by the creation of models and several levels of abstraction of a problem, and Code-Driven, which is an approach without any model and guided by the code.
The first approach and the use of models presents some important advantages like, as an example, the possibility to define what is essential and the possibility to represent it in a clear and intuitive manner. In this context, UML often accompanies the analysis and project phases and offers a clear, unique and easy to read view. In a distributed context, nevertheless, this tool presents various ambiguities, especially regarding the description of the "interaction stories" (that in a distributed environment cover what becomes the most important dimension): the semantic of the interactions loses clarity and expressive power. This corresponds to an "explosion" in the abstraction gap, because it becomes difficult to realize what has been represented, in abstract, in the model. In these conditions emerges the need for a tool that allows to reduce the abstraction gap; we will see how, through many different transformations Model-To-Model (M2M), it will be possible to reach one last Model-To-Code (M2C) transformation which will produce the code for a prototype.
The more conventional approaches (as an object-oriented modeling rather

than the use of agents or actors) would have involved various drawbacks, such as the lack of software reusability and, moreover, the possible upheavals due to the small changes in the requests made by the customer.

Keywords: Software engineering, software development process, process representation, managed software development, drone, smart devices, model-driven development, code-driven development

1 Introduction

Building software means to realize a product, on the basis of a project, by adopting a specific development process performed by the cooperative work of a group of people who pose specific goals, including:

- To understand the nature and the purpose of the product;
- To define and analyze the prerequisites;
- To choose the developing process, the available technologies and to define a work plan;
- To plan and build the product;
- To test the product;
- To distribute and maintain the product.

The software's development process will be intended as an ordered set of steps including the activities, the constraints and the resources needed to produce a specific output able to satisfy a set of prerequisites. This paper will be intended and used from the team as a guide line and a "travel diary" during the development process. The aim of this report is to give an answer to the fundamental questions that will lead the team during the whole making:

1. Did we built the right product?
2. Has it been built the right way?

2 Vision

"There is no code without a project, no project without problem analysis and no problem without requirements. The problem is how to make them explicit, effective and reusable."

To realize a correct and effective solution that satisfies all the requirements from the client through cost reduction and the reduction of the abstraction gap. Facing, with an engineering approach, the development process of an heterogeneous and distributed system in order to be able to answer, with the same method, to future requests. Particularly, it is necessary to implement a prototype of the product in a little amount of time, useful to show to the customer an intermediate result that can then be iteratively improved. In relation to a "waterfall" approach, this allows to minimize the risks and at the same time make it easy

to introduce changes in the specifics during the development. To this end we selected SCRUM as development strategy. The reasons for this choice are the easy interpretation of the contents and the possibility to obtain prototypes in a small amount of time promoting, in the meanwhile, cooperative and democratic teamwork. More on SCRUM can be found at http://www.scrumguides.org/scrum-guide.html.

3 Goals

The main goal is to satisfy the requirements and the needs of the client. Specifically, the aim is the realization of a modular, extendible and re-configurable software system to check and monitor the movement of a robot. The initial task will be the definition of the customer's needs and, in this context, the requirements analysis will be crucial, where it is going to be defined, without ambiguities, the explicit and implicit requests posed by the client. The second objective is the minimization of the "abstraction gap", where for us the abstraction gap of a language is as high as how difficult it is for us to express the desired concepts with it.

4 Requirements

STEP 1
Design and build a (prototype of a) software system that, with reference to a differential drive robot (called from now on robot) allows a user to select between a 'learning phase' and an 'autonomous phase'. During the learning phase, the user can send a sequence of move commands (e.g. forward, backward, left right, stop) to the robot. The robot must not only execute each command but it must also record the whole sequence of commands until the user decides to terminate the learning phase.
After the termination of the learning phase, the user can tell the robot to enter the autonomous phase in a 'direct' or in a 'reverse' mode. During this phase the robot executes autonomously the sequence of moves it has learned, by complementing each move (e.g. forward->backward) if the selected mode is 'reverse'.
During the autonomous phase, the robot must be able to excute (as soon as possible) a stop command sent by the user.

Non-functional requirements at Step 1:
Remember to express in an explicit way the technological hypothesis assumed during the problem analysis phase, to define the abstraction gap (if any) and to explain how the software project can overcome (in a repeatable way) such a gap.

5 Requirements analysis

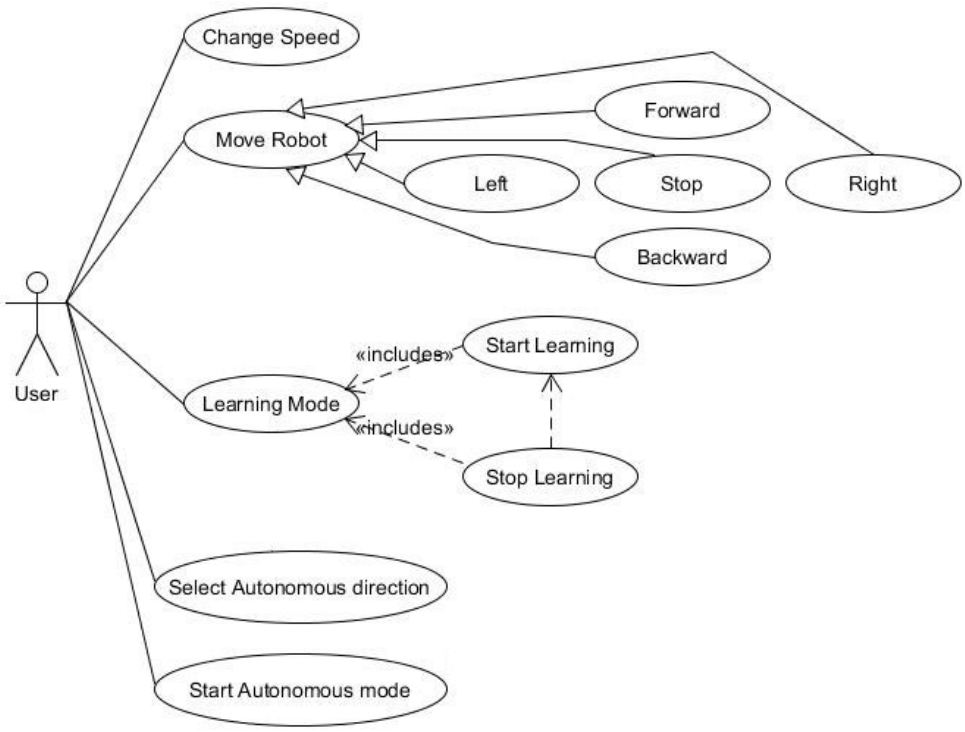
5.1 Glossary

The analysis of the nouns (highlighted in red) facilitates the writing of a glossary, necessary in order to use a common language, free of any ambiguities, with the customer and all the subjects involved.

Terms glossary

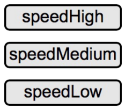
Term	Meaning
software system	a set of things working together as parts of a mechanism or an interconnecting network; a complex whole.
robot	a machine capable of carrying out a complex series of actions automatically, especially one programmable by a computer.
differential drive robot	a mobile robot whose movement is based on two separately driven wheels placed on either side of the robot body. It can thus change its direction by varying the relative rate of rotation of its wheels and hence does not require an additional steering motion.
user	any person or actor able to use the Graphical User Interface to interact with the robot.
phase	a distinct period or stage in a process of change or forming part of something's development or execution.
move	to go in a specified direction or manner; to change position.
stop command	a precise input given by the user in order to halt every move of the robot.

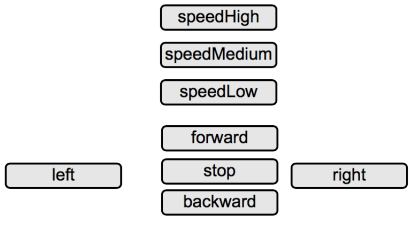
5.2 Use cases

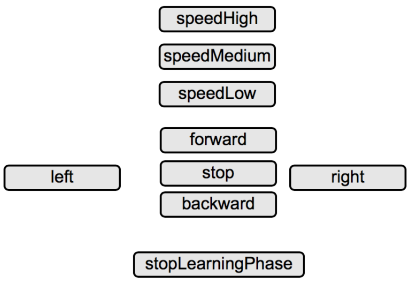


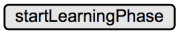
5.3 Scenarios

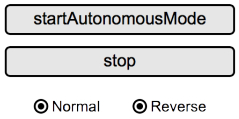
We will hereunder describe in more detail the many use cases of the project, explaining all the possible scenarios.

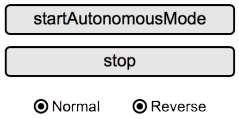
Use Case:	**Change Speed**
Description:	Allows the User to change the robot's Speed
Actors:	User
Preconditions:	A connection to the HTTP server
Main Scenario:	The system stores the speed to apply to the next movement commands (speedLow, speedMedium, speedHigh)
Secondary Scenarios:	
Post-conditions:	The system is ready to receive a new command
Mock-up:	

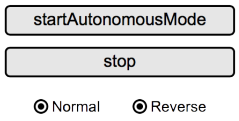
Use Case:	**Move Robot**
Description:	Sends the movement orders to the robot (with the relative speed)
Actors:	User
Preconditions:	A connection to the HTTP server
Main Scenario:	The system gives the input to the robot that carries the information regarding the movement (as a couple of elements: type of command / speed). The input may mean: forward, backward, left, right, stop. The speed can be: speedLow, speedMedium, speedHigh.
Secondary Scenarios:	If the robot is in the autonomous mode and the "stop" button is pressed the robot will halt and stop the execution
Post-conditions:	The system is ready to receive a new command
Mock-up:	

Use Case:	**Learning Mode**
Description:	Makes the robot store the sequence of executed commands that can later be automatically re-executed using the "autonomous mode"
Actors:	User
Preconditions:	A connection to the HTTP server
Main Scenario:	In order for the "learning phase" to take place the "start learning phase" button has to be pressed. During the learning phase the commands received are stored. In order to end the learning phase the "stop learning phase" button has to be pressed.
Secondary Scenarios:	
Post-conditions:	The system is ready to receive a new command and/or start the autonomous phase
Mock-up:	

Use Case:	**Start Learning**
Description:	Tells the robot to store each command received until the "stop learning" button is pressed.
Actors:	User
Preconditions:	A connection to the HTTP server.
Main Scenario:	The system initializes the structure that will hold the commands given to it and waits for them.
Secondary Scenarios:	
Post-conditions:	The "stop learning" button is enabled and so are the buttons regarding the movements.
Mock-up:	

Use Case:	**Stop Learning**
Description:	Tells the Robot to stop memorizing commands and exit the learning phase.
Actors:	User
Preconditions:	A connection to the HTTP server. The Robot must be in the learning phase.
Main Scenario:	1. The robot is stopped 2. The learning phase is interrupted (the robot won't store the next commands anymore) 3. The system awaits for other commands
Secondary Scenarios:	
Post-conditions:	The system is ready to receive a new command and start the autonomous phase
Mock-up:	

Use Case:	**Start Autonomous Mode**
Description:	Makes the robot execute the commands' sequence memorized during the learning phase. The User can choose between Normal or Reverse mode.
Actors:	User
Preconditions:	The learning phase must have been executed and terminated at least once.
Main Scenario:	1. The robot executes each command in the list in order: (a) if the current mode is "Normal", the commands will be executed in the same order they have been stored (b) if the current mode is "Reverse", the robot will execute the opposite of each command (e.g. forward -> backward) in the opposite order in which they have been stored 2. The system awaits for other commands
Secondary Scenarios:	If the stop button is pressed during the autonomous phase, the robot will halt the execution of the current mode and reach a stop.
Post-conditions:	The system is ready to receive a new command
Mock-up:	

Use Case:	**Select Autonomous Mode**
Description:	It defines the order in which the robot executes the commands during the autonomous phase
Actors:	User
Preconditions:	A connection to the HTTP server
Main Scenario:	The system memorizes the "mode" to be used from the robot while in the autonomous phase (Normal or Reverse)
Secondary Scenarios:	
Post-conditions:	The system is ready to receive a new command
Mock-up:	

5.4 (Domain)model

From the prerequisites analysis, the use cases and the scenarios presented emerges, once again, the distributed and heterogeneous nature of the system. A *distributed software system* is a computing system in which a number of software components cooperate by communicating over a **network**. It is clear that we can subdivide the system in two interacting software components:

1. Robot
2. Console

Considering the subsystems like black boxes, we are able to carry on the problem analysis and deduce the logic architecture of the system. For every subsystem we will describe the three dimensions: structure, interaction and behaviour.

The physical robot has been handed to us from the customer. Talking to him we acknowledged that it had a built-in system that allows the use of an API to control its basic movements. To further understand the robot's system, the customer pointed us to its model, which can be found here:
https://137.204.107.21/syskb/it.unibo.iss2015intro/docs/Robots/robotEntry.html
Our aim is to start modeling directly using the custom DSL used in the robot provided us by the customer. Using that we will implement our personalized solution for the problem at hand.

The QActors DSL exploits the *pattern bridge*, which allows us to decouple the abstraction of the system and its implementation so that they can be modified independently. This makes us able to define the main concepts of structure, interaction and behaviour without having to worry about the final implementation. Moreover, our DSL presents many advantages:

- high-level development allows us to be independent from the low-level environment;
- through the DSL we will be able to start the tests as soon as possible, using a mock object;
- through the use of high-level code we reduce the incoherence and inconsistency;
- robotics and IoT will be a main topic in the near future;
- self-explanatory, there is not need to further comment the code, since it shows a formal representation of the system.

The architecture of the system is shown hereunder. Using the DSL, we can simultaneously display the structure, behaviour and, most importantly, the interactions of the system's components in a formal manner:

```
RobotSystem rob -regeneratesrc

Event usercmd: usercmd(X)

Context ctxRobot ip[host="localhost" port=8079] -httpserver
```

```
Robot mock QActor robotq context ctxRobot{

   Plan init normal
      switchToPlan selectMode

   Plan selectMode
        sense time(300000) usercmd -> continue ;
        memoCurrentEvent;
         [ ?? msg(usercmd, EVENT, WSOCK, NONE,
              usercmd(robotgui(l(startLearning))), TIME) ] switchToPlan
              startLearning ;
        [ ?? msg(usercmd, EVENT, WSOCK, NONE,
            usercmd(robotgui(n(startAutonomousDirect))), TIME) ]
            switchToPlan autonomousDirect ;
        [ ?? msg(usercmd, EVENT, WSOCK, NONE,
            usercmd(robotgui(n(startAutonomousReverse))), TIME) ]
            switchToPlan autonomousReverse

   Plan startLearning
      sense time(300000) usercmd -> continue ;
      switchToPlan learningLoop

   Plan learningLoop
      memoCurrentEvent;
      [ ?? msg(usercmd, EVENT, WSOCK, NONE,
          usercmd(robotgui(k(stopLearning))), TIME) ] switchToPlan
          selectMode;
      /*
       * Save current move and execute it
       */
      sense time(300000) usercmd -> continue ;
        repeatPlan 0

   Plan autonomousStart
      /* Set moves lenght
       * Save moves in KB, if moves are ended switch to autonomousLoop */
      repeatPlan 0

   Plan autonomousDirect
      switchToPlan autonomousStart

   Plan autonomousReverse
      addRule reverse(true);
      switchToPlan autonomousStart

   Plan autonomousLoop
      /* execute every move
       * if moves are ended switch to endAutonomousPhase
       */
```

```
    repeatPlan 0

  Plan endAutonomousPhase
    switchToPlan selectMode

}
```

The console is then generated using the QActors framework through the *-httpserver* flag. It consists in a web interface for the user with buttons that allow the interaction between the user and the robot. The final result will be shown in the appropriate section of the paper (see sub-section 11.1).

With this we can say to have fully described the system and we are able to have a confirmation from the customer on the satisfaction of the prerequisites.

5.5 Test plan

At this point, the elements that compose the logic architecture can provide us enough data on what the different subsystems have to realize, without having to detail their internal behaviour yet. We talk here about test plan instead of simply "testing" because the step to the testing phase will take place only later, after the Project phase. The test plans allow us to comprehend and specify the expected behaviour of an entity before its design and implementation. The testing phase considering each feature implemented in every sprint is carried on a Mock-Object. This helps us to speed up the process to check all the features of the robot without having it physically.

```
RobotSystem rob -regeneratesrc

Event usercmd: usercmd(X)

Context ctxRobot ip[host="localhost" port=8079] -httpserver

QActor test context ctxRobot {

  Rules{
    loadTheory(FName) :-
      actorPrintln( loadTheory(FName) ),
      consult( FName ).
    cmd(usercmd(robotgui(l(startLearning)))).
    cmd(usercmd(robotgui(w(low)))).
    cmd(usercmd(robotgui(w(medium)))).
    cmd(usercmd(robotgui(w(high)))).
    cmd(usercmd(robotgui(a(low)))).
    cmd(usercmd(robotgui(a(medium)))).
    cmd(usercmd(robotgui(a(high)))).
    cmd(usercmd(robotgui(s(low)))).
    cmd(usercmd(robotgui(s(medium)))).
    cmd(usercmd(robotgui(s(high)))).
```

```
    cmd(usercmd(robotgui(d(low)))).
    cmd(usercmd(robotgui(d(medium)))).
    cmd(usercmd(robotgui(d(high)))).
    cmd(usercmd(robotgui(k(stopLearning)))).
    cmd(usercmd(robotgui(n(startAutonomousDirect)))).

    simulateCmd :-
       retract( cmd( CMD ) ),
       doemit( "usercmd", CMD ).
    simulate(no).
}

Plan init normal
    println("> Plan INIT") ;
    solve loadTheory("applTheory.pl") time(0) onFailSwitchTo
        prologFailure;
    [ !? simulate(yes) ] switchToPlan simulate

Plan simulate
    println(">> Plan Simulate") ;
    println(">> solve simulateCmd -> doemit usercmd ");
    solve simulateCmd time(0) onFailSwitchTo endSimulateCmd ;
    delay time(1000);
    repeatPlan 0

Plan endSimulateCmd
    println(">>> Plan endSimulateCmd" ) ;
    println(">>> Waiting a bit before TEST if robot react to arbitrary
        events") ;
    delay time(3000) ;
    println(">>> EMIT EVENT") ;
    emit usercmd : usercmd(stop)

Plan prologFailure
    println(">> Plan prologFailure" )

}
```

6 Problem analysis

In the problem analysis we will examine the development of the product, trying to delay as much as possible the choice of a technological hypothesis, in order to solve the problem in a platform-independent way. Primary objective during the analysis is the implementation of a formal model, avoiding any ambiguity. It will be, thus, necessary to pick an adequate representational method for the system.

6.1 Abstraction gap

In this section we discuss the many technological hypothesis considered for the realization of the product.

The first language considered is Java. Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. The latter is the peculiar feature that catched our attention. Through the Java programming language it is possible to develop a software able to work on any device, thanks to the Java Runtime Environment (JRE), which facilitates the software's compatibility and the communication between the entities of the system, through the sharing of standard data and communication protocols. Regarding the data exchange, Java allows to exploit various protocols, but it would not be easy to make changes later on, like modifying the protocols from UDP to TCP. About the communication between the entities of the distributed system it would be possible to take advantage of the message passing for example. Unfortunately, even in this case the problem of the hardship in carrying out changes to the software once completed stands out. The issue at hand would rise in case, for example, it would be necessary to change the implementation that instead of synchronous messages chooses asynchronous ones or vice-versa. It is important to notice that the concept itself of the messages is not built-in in Java. Thus, we spot an abstraction gap, because we wish to implement a model of interaction based on message passing instead of procedure calls. This is not expressed from the language, even if it could be built with it.
We need to remind that, as specified in the prerequisites, the goal of the project is the making of a modular, extendible and re-configurable software system. Choosing Java as technological hypothesis we know that the modification of the system, during or after the development, risks to imply a long re-engineering work and, consequently, a high waste of resources.
We remember, also, that in the prerequisites it is specified a request to make possible to switch from a system with a proactive behaviour to one with a reactive behaviour. This is possible in Java using particular extensions (like RxJava) which in turn would force us to modify part of the work done.
In conclusion, the development of the distributed applications in Internet has forced the object-oriented paradigm 'against the ropes', which reveals the lack of concepts like autonomy, orientation toward the duties, the placement in an environment and the ability to interact flexibly. As an example, object-based approaches does not capture the idea that the interaction may be founded on negotiation and does not provide any support on how to keep a balance between a reactive and a proactive behaviour in complex and dynamic situations.
The adoption of the object-oriented paradigm induces the building of applications using a perspective oriented toward functionality and carries to the development of static architectures or to the need to adopt complex infrastructures for the management of the dynamics, the ways to reconfigure the system and the support for the negotiations of resources and duties. The next evolutionary

step may be the agent-oriented paradigm.

Such agents can be found in JADE. JADE is a middleware built on Java that facilitates the development of multi-agent systems. In this case the use of agents in a distributed system seems to be a well-fitting option.
It is possible to exploit smart agents (through the use of Artificial Intelligence) able to decide when to use a TCP, UDP or different connection to then inform the target agent and together set up a precise protocol suitable for the communication. We would obtain a flexible system, but not a controllable one. Plus, it would be hard to adapt it to the use of new technologies or protocols. The exchange of data between agents, moreover, is possible only through an asynchronous communication. In conclusion, this approach does not considers a high level of control on the behaviour of the agents and this could be seen as a big concern in case a standard has to be selected for security reasons, a matter of prerequisites or, more in general, of network management. Despite the positive points, Jade does not allow to define a system configurable enough to be taken into serious consideration in this particular case.

The use of UML would allow us to define a model of the system, through its coordinates: structure, interaction and behaviour. Through this we would obtain exactly the goal of the analysis. Nevertheless, in our case UML represents a proper solution. First, it is not easy for UML to represent the taxonomies based on classes (base elements in the Object Oriented programming). Second, <in its current form, the Object Management Group's documents do not offer a rigorous definition of UML's true semantics, not even of the semantic domain. Rather, they concentrate on the abstract syntax, intermixed with informal natural language discussions of what the semantics should be. These discussions certainly contain much interesting information on the semantics, but they are a far cry from what developers, as well as tool vendors, really need. As recent research shows, they still lack many clarifying details and contain many inconsistencies. Actually, rigorously defining semantics for the full UML is a daunting task that would require a far more detailed mathematical analysis of UML's many loosely connected kinds of diagrams than has been done to date.> (from the paper <Meaningful Modeling: What's the Semantics of 'Semantics'?>, David Harel, Bernhard Rumpe).
This tool presents various ambiguities, especially regarding the description of the "interaction stories" (that in a distributed environment cover what becomes the most important dimension): the semantic of the interactions loses clarity and expressive power. This corresponds to an "explosion" in the abstraction gap, because it becomes difficult to realize what has been represented, in abstract, in the model.
For these reasons we could not consider UML as the best choice for the definition of the assignment's concepts.

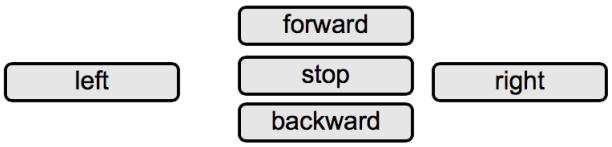
The next development tool considered is Xtext. Xtext is a framework for the development of programming languages and domain-specific languages (DSL). Using Xtext it is possible to specify a custom language and obtain, from it, an equivalent code in Java. This solution allows to specify well-defined entities having a unique semantic and, at the same time, grants us to exploit the strong points of the Java language.
Because of the abstraction gaps it is necessary the creation of a custom language that will bridge the distance between what we want to build and the technology used. We identified in Xtext the technology required for the realization of a DSL, and QActors DSL as the reference technology.
The QActors DSL, implemented using XText, is the perfect choice for the project's realization. QActors (Quasi-Actors) DSL is the name given to a custom framework built by Antonio Natali for the course 'Engineering of Software Systems' to show how application designers can face the analysis, the design and the implementation of (distributed heterogeneous) software systems whose components interact by adopting a message-passing or an event-driven or an event-based style rather than a traditional object-based style. Every single QActor is then an active software component that can perform both proactive and reactive behaviour expressed as a sequence of actions. A QActor always works within a Context and is able to send/receive messages.

6.2 Logic architecture

At this point it is possible to switch to the problem analysis, following the SCRUM model. The system's logic architecture constitutes the synthetical artifact of the analysis. Its purpose is to better define the subsystems of the problem, suggesting a solving system. The logic architecture itself will be the reference for the next steps of the project, thus it has to be as much as independent as possible in respect to the implementation environment. We will now expand every sprint of the SCRUM development process and examine them.

SPRINT 1:

- Sprint Planning:
  The purpose now is to design and build an application that allows users to command the movements of a robot by using a remote input device. We are particularly interested in two input devices:
  - a conventional keyboard with the chars: a,s,d,w,h, where the speed is fixed and 'h' means STOP, 'w' means FORWARD, 'a' means LEFT, 's' means BACKWARD and 'd' means RIGHT;
  - a GUI interface like the following one:



Whatever be the user interface, the remote input device can be viewed as a generator of messages or events. To achieve this purpose we've decided to choose the QActor (DSL) framework, which allows us to analyze, design and implement heterogeneous distributed systems, whose components interact using the message-passing / event-driven / event-based paradigm, rather than the more traditional object-oriented paradigm. Another reason why we chose this framework is that it helps removing the abstraction gap of handling different system behaviours (e.g. proactive/reactive).
The resulting model of this phase can be found here:

```
RobotSystem rob -regeneratesrc

Event usercmd: usercmd(X)

Context ctxRobot ip[host="localhost" port=8079] -httpserver

Robot mock QActor robotq context ctxRobot{
   Rules{
      loadTheory(File) :-
         actorPrintln( loadTheory(File) ),
         consult( File ).
   }

   Plan init normal
      println("Ready!");
      solve loadTheory("applTheory.pl") time(0) onFailSwitchTo
          prologFailure;
      switchToPlan waitCommand

   Plan waitCommand
      println("Plan waitCommand");
        sense time(300000) usercmd -> work

   Plan work
      println("Plan work");
      solve robotMoveFromUsercmdEvent(300000, usercmd, work) time(0)
          onFailSwitchTo prologFailure

   Plan prologFailure
      println(">>>>> [x] Plan prologFailure") ;
        println(">>>>> [x] failed to solve a Prolog goal")
}
```

We can take a look at Prolog rules contained in the "applTheory.pl" file here:

```
/*-----------------------------------------------------------------
Perform a reactive move related to the current usercmd event
-----------------------------------------------------------------*/
robotMoveFromUsercmdEvent( MoveTime, AlarmEvent, ReactionPlan ) :-
    actorPrintln( robotMoveFromUsercmdEvent( Robot, AlarmEvent,
        ReactionPlan ) ),
    myname( Robot ),
    Robot <- memoCurrentEvent,
    retract( msg( MSGID, event, SENDER, RECEIVER, CONTENT, SEQNUM )
        ),   %%retract always !!!
    actorPrintln( robotMoveFromUsercmdEvent( Robot, CONTENT ) ),
    Robot <- getCurrentPlan returns CurPlanName,
    robotMove(Robot, CurPlanName, CONTENT, MoveTime, AlarmEvent,
        ReactionPlan ) .

robotMove( Robot, CurPlanName, usercmd(CMD), MoveTime, AlarmEvent,
     ReactionPlan ):-
   actorPrintln( robotMove( CurPlanName, CMD , MoveTime, AlarmEvent,
       ReactionPlan ) ),
   moveCmdTable(CMD, RobotCmd, Speed ),
   actorPrintln( robotMove( CurPlanName, RobotCmd , Speed, MoveTime,
       AlarmEvent, ReactionPlan ) ),
   Robot <- execRobotMove(CurPlanName,RobotCmd,Speed,0, MoveTime,
       AlarmEvent, ReactionPlan) returns B,
   actorPrintln( robotMove( CurPlanName, RobotCmd , Speed,
       AlarmEvent, ReactionPlan, B ) ),
   B == true.

/*-----------------------------------------------------------------
Map a user command into a move and a speed
-----------------------------------------------------------------*/
moveCmdTable( robotgui(w(low)), forward, 40 ).
moveCmdTable( robotgui(w(medium)), forward, 70 ).
moveCmdTable( robotgui(w(high)), forward, 100 ).
moveCmdTable( robotgui(a(low)), left, 40 ).
moveCmdTable( robotgui(a(medium)), left, 70 ).
moveCmdTable( robotgui(a(high)), left, 100 ).
moveCmdTable( robotgui(s(low)), backward, 40 ).
moveCmdTable( robotgui(s(medium)), backward, 70 ).
moveCmdTable( robotgui(s(high)), backward, 100 ).
moveCmdTable( robotgui(d(low)), right, 40 ).
moveCmdTable( robotgui(d(medium)), right, 70 ).
moveCmdTable( robotgui(d(high)), right, 100 ).
moveCmdTable( robotgui(h(low)), stop, 40 ).
moveCmdTable( robotgui(h(medium)), stop, 70 ).
moveCmdTable( robotgui(h(high)), stop, 100 ).
```

– Sprint Review:
Now we have a working model handling the basic functionality of the Robot, seen as a "thing of the Internet", able to perceive events, to interact with remote "minds" and to execute other tasks.

– Sprint Retrospective:
After the first sprint we have talked to each other using e-mails and by phone. This has brough us slow conversations since not all of the team uses the e-mails very often during the day. In order to speed up the process and have a place where all our messages would be saved we need to exploit more common tools. We believe it would be best to create a Facebook and a Watsapp group as common places where a message can be seen from all the Development Team at the same time and be saved for later reviews.

SPRINT 2:

– Sprint Planning:
Starting from the previous model, we would like to enhance the capabilities of the robot, allowing a user to activate the "learning phase" (and end it). In order to develop the learning phase we encountered an issue related to the QActors DSL we are using. After analyzing the code of the DSL, we found out that the matter consisted in the "time" variable assigned to each event in the system, which was 0 (zero) for every generated event. We solved this replacing the representation of an event with our **custom representation**. The new specifications can be found in the files "EventItemVC.java" and "LocalTimeVC.java".
After building these improvements, we had to include the newly written entities in the file of the it.unibo.contactEvent.platform package: "ContactEventPlatform.java". We show hereunder the modifications done.

File LocalTimeVC.java:

```
package it.unibo.contactEvent.platform;

import java.util.Calendar;

import it.unibo.contactEvent.interfaces.ILocalTime;

public class LocalTimeVC implements ILocalTime {
   protected long time = 0;

   public LocalTimeVC() {
      this(Calendar.getInstance().getTimeInMillis());
   }

   public LocalTimeVC(long time) {
      try {
         setTime(time);
```

```
      } catch (Exception e) {
         e.printStackTrace();
      }
   }

   protected void setTime(long time) throws Exception {
      if (time < 0)
         throw new Exception("[LocalTimeVC] negative time not
             allowed");
      else
         this.time = time;
   }

   @Override
   public long getTheTime() {
      return this.time;
   }

   @Override
   public String getTimeRep() {
      return "'" + Long.toString(this.time) + "'";
   }
}
```

File EventItemVC.java:

```
package it.unibo.contactEvent.platform;

import java.util.Hashtable;

import alice.tuprolog.Struct;
import alice.tuprolog.Term;
import it.unibo.contactEvent.interfaces.IEventItem;
import it.unibo.contactEvent.interfaces.ILocalTime;

public class EventItemVC implements IEventItem {
   protected String eventId;
   protected String subj;
   protected ILocalTime time;
   protected String msg;
   protected Hashtable<String, Object> at;
   protected Struct msgStruct;

   public EventItemVC(String rep) throws Exception {
      msgStruct = (Struct) Term.createTerm(rep);
      this.eventId = msgStruct.getArg(0).toString();
      this.msg = msgStruct.getArg(4).toString();
      this.subj = msgStruct.getArg(2).toString();
      String timeStr = msgStruct.getArg(5).toString().replace("'",
          "");
```

```
        try {
            this.time = new LocalTimeVC(Long.parseLong(timeStr, 16));
        } catch (Exception e) {
            this.time = new LocalTimeVC(0);
        }
    }

    public EventItemVC(String eventId, String msg, ILocalTime time,
            String subjId) throws Exception {
        this("msg( EVID, MSGTYPE, SENDER, RECEIVER, MSG, TIME )"
                .replace("EVID", eventId)
                .replace("MSGTYPE", "event")
                .replace("SENDER", subjId)
                .replace("RECEIVER", "none")
                .replace("MSG", msg)
                .replace("TIME", time.getTimeRep()));
    }

    @Override
    public String getEventId() {
        return this.eventId;
    }

    @Override
    public String getSubj() {
        return this.subj;
    }

    @Override
    public ILocalTime getTime() {
        return this.time;
    }

    @Override
    public String getMsg() {
        return this.msg;
    }

    @Override
    public String getPrologRep() {
        return msgStruct.toString();
    }

    @Override
    public String getDefaultRep() {
        Term t = Term.createTerm(getPrologRep());
        return t.toString();
    }

    @Override
```

```
    public Hashtable<String, Object> getArgTable() {
       return this.at;
    }

}
```

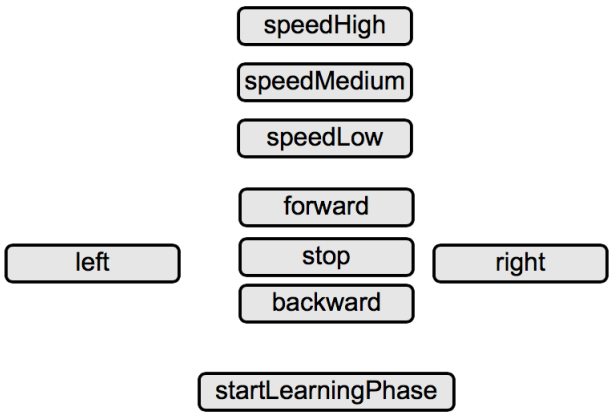
The new files have been used both in the package it.unibo.contactEvent.platform. In particular, in the file ContactEventPlatform.java:

```
.....
    protected IEventItem buildEvent( String evId, String evContent,
        String subj ){
       try {
          return new EventItemVC( evId, evContent, new LocalTimeVC() ,
              subj );
       } catch (Exception e) {
          e.printStackTrace();
          return null;
       }
    }
....
```

SPRINT 3:

– During this sprint our goal is to finally make the "learning phase" take place. During this phase the user is able to send a sequence of move commands (e.g. forward, backward, left, right, stop) to the robot. While the user does so, the robot must not only execute each command but also record the whole sequence, until the user decides to terminate the learning phase.
As no clear specifications are given us from the user in the requirements, the team decided that the best approach, for the moment, capable of both satisfy the requirements and be feasible, is to keep in memory the last learning phase's instructions, may they be one minute or one year old.

  Now the interface should look like the following:



The model of the system regarding this sprint is reported here:

```
RobotSystem rob -regeneratesrc

Event usercmd: usercmd(X)

Context ctxRobot ip[host="localhost" port=8079] -httpserver

Robot mock QActor robotq context ctxRobot{
   Rules{
      loadTheory(File) :-
         actorPrintln( loadTheory(File) ),
         consult( File ).
      amp("autonomous.pl"). //autonomous mode path
   }

   Plan init normal
      println("Ready!");
      solve loadTheory("applTheory.pl") time(0) onFailSwitchTo
          prologFailure;
      switchToPlan selectMode

   Plan selectMode
      println("Plan selectMode");
        sense time(300000) usercmd -> continue ;
        memoCurrentEvent;
         [ ?? msg(usercmd, EVENT, WSOCK, NONE,
              usercmd(robotgui(l(startLearning))), TIME) ]
              switchToPlan startLearning ;
        removeRule msg(usercmd, EVENT, WSOCK, NONE, usercmd,TIME);
        repeatPlan 0
```

```
   Plan startLearning
      println("Plan startLearning");
      solve cleanMemory time(0) onFailSwitchTo prologFailure;
      sense time(300000) usercmd -> continue ;
      switchToPlan checkStopLearning;
      switchToPlan learningLoop

    Plan checkStopLearning resumeLastPlan
      println("Plan checkStopLearning");
      memoCurrentEvent;
        [ ?? msg(usercmd, EVENT, WSOCK, NONE,
            usercmd(robotgui(k(stopLearning))), TIME) ] switchToPlan
            selectMode

   Plan learningLoop
        println("Plan learningLoop");
       solve addMove time(0) onFailSwitchTo prologFailure;
       solve robotMoveFromUsercmdEvent(300000, usercmd,
           moveCallback) time(0) onFailSwitchTo prologFailure;
       repeatPlan 0

    Plan moveCallback
      println("Plan moveCallback");
      memoCurrentEvent;
      [ !? msg(usercmd, EVENT, WSOCK, NONE,
          usercmd(robotgui(k(stopLearning))), TIME) ] switchToPlan
          saveLastMove;
      switchToPlan learningLoop

    Plan saveLastMove
      solve addMove time(0) onFailSwitchTo prologFailure;
        switchToPlan selectMode

   Plan prologFailure
      println(">>>>> [x] Plan prologFailure") ;
        println(">>>>> [x] failed to solve a Prolog goal" )
}
```

In order to store every move made by the robot during the learning mode we use these rules, defined in the file "applTheory.pl":

```
cleanMemory :-
   java_object("ApplSystem", [], Appl),
   amp(PATH),
   Appl <- reset(PATH).

addMove :-
   myname( Actor ),
```

```
   retract( msg( MSGID, MSGTYPE, SENDER, RECEIVER,
       usercmd(robotgui(CONTENT)), TIME ) ),
   actorPrintln( writeCurrentEV(Actor, CONTENT, TIME) ),
   java_object("ApplSystem", [], Appl),
   amp(PATH),
   Appl <- addMoveInFile(PATH, CONTENT, TIME).
```

The Prolog rules described above will be calling the following functions, implemented in "ApplSystem.java":

```
/* ApplSystem is written by the Application designer
 * in order to define operations that can be used
 * from the Prolog rules (foundable in prolog-files/applTheory.pl) */
import it.unibo.contactEvent.platform.EventPlatformKb;
import it.unibo.is.interfaces.IOutputView;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Iterator;

public class ApplSystem {
   private static IOutputView outView = EventPlatformKb.stdOutView;

   public static void setOutView(IOutputView outViewArg){
      outView = outViewArg;
   }

   public static ArrayList<String> readFile(String fName){
      ArrayList<String> result = new ArrayList<String>();
      try {
         outView.addOutput(" *** ApplSystem readFile " + fName);
         InputStream fs        = new java.io.FileInputStream(fName);
         InputStreamReader inpsr = new InputStreamReader(fs);
         BufferedReader br     = new BufferedReader(inpsr);
         Iterator<String> lsit = br.lines().iterator();
         while(lsit.hasNext()){
            result.add(lsit.next());
         }
         br.close();
      } catch (Exception e) {
         outView.addOutput(" *** ApplSystem ERROR " + e.getMessage());
      }
      return result ;
   }

   public static void clearFile(String fName) {
```

```
        try {
            FileOutputStream fsout = new FileOutputStream(new
                File(fName));
            fsout.write(("").getBytes());
            fsout.close();
        } catch (Exception e) {
            outView.addOutput(" *** ApplSystem ERROR " + e.getMessage());
        }
    }

    public static void writeInFile(String fName, String content) {
        writeInFile(fName, content, "true");
    }

    public static void writeInFile(String fName, String content,
         String append) {
        try {
            outView.addOutput(" [ApplSystem] ApplSystem writeInFile " +
                fName);
            FileOutputStream fsout = new FileOutputStream(new
                File(fName),
                 append.toLowerCase() == "true" ? true : false);
            fsout.write((content+"\n").getBytes());
            fsout.close();
        } catch (Exception e) {
            outView.addOutput(" [ApplSystem] ApplSystem ERROR " +
                e.getMessage());
        }
    }

    public void reset(String fName){
        clearFile(fName);
    }

    public void addMoveInFile(String fName, String cmd, String time){
        Move m = new Move(cmd, Long.parseLong(time));
        writeInFile(fName, m.getDefaultRep());
    }

}
```

The representation of every single "move" made from the robot has been defined by us in the "Move.java" file:

```
public class Move {
    private long time;
    private String command;

    public Move(String cmd, long time){
```

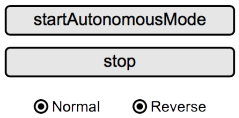
```
        this.command = cmd;
        this.time = time;
    }

    public long getTime() {
        return time;
    }

    public void setTime(long time) {
        this.time = time;
    }

    public String getCommand() {
        return command;
    }

    public void setCommand(String command) {
        this.command = command;
    }

    public String getDefaultRep(){
        return "move("+this.command+", '"+
             Long.toString(this.time)+"')." ;
    }

    public Move reversed(){
        char[] ccmd = command.toCharArray();
        switch (ccmd[0]) {
        case 'w':
            ccmd[0] = 's';
            break;
        case 'a':
            ccmd[0] = 'd';
            break;
        case 's':
            ccmd[0] = 'w';
            break;
        case 'd':
            ccmd[0] = 'a';
            break;
        }
        return new Move(String.valueOf(ccmd),time);
    }
}
```

– Sprint Review:

At this point we find the working model of the robot, able to learn and store each command received.

SPRINT 4:

– Sprint Planning:
We want to extend the model of the Robot so that he can enter the autonomous phase (only after the termination of the learning phase) in a "Normal" or in a "Reverse" mode. During this phase the robot autonomously executes the sequence of moves it has learned if the selected mode is "Normal" or complements them (e.g. "forward" -> "backward") and executes them in the opposite order if the selected mode is "Reverse".
Even in this case we are going to implement a proactive feature which does not require any particular changes to take place.
The piece of the interface considered will look like the following picture:



The model of the system regarding this sprint is reported hereunder:

```
RobotSystem rob -regeneratesrc

Event usercmd: usercmd(X)

Context ctxRobot ip[host="localhost" port=8079] -httpserver

Robot mock QActor robotq context ctxRobot{
   Rules{
      loadTheory(File) :-
         actorPrintln( loadTheory(File) ),
         consult( File ).
      amp("autonomous.pl"). //autonomous mode path
   }

   Plan init normal
      println("Ready!");
      solve loadTheory("applTheory.pl") time(0) onFailSwitchTo
          prologFailure;
      switchToPlan selectMode

   Plan selectMode
      println("Plan selectMode");
        sense time(300000) usercmd -> continue ;
        printCurrentEvent;
        memoCurrentEvent;
         [ ?? msg(usercmd, EVENT, WSOCK, NONE,
             usercmd(robotgui(l(startLearning))), TIME) ]
             switchToPlan startLearning ;
```

```
      [ ?? msg(usercmd, EVENT, WSOCK, NONE,
          usercmd(robotgui(n(startAutonomousDirect))), TIME) ]
          switchToPlan autonomousDirect ;
      [ ?? msg(usercmd, EVENT, WSOCK, NONE,
          usercmd(robotgui(n(startAutonomousReverse))), TIME) ]
          switchToPlan autonomousReverse ;
      removeRule msg(usercmd, EVENT, WSOCK, NONE, usercmd,TIME);
      repeatPlan 0

Plan startLearning
   println("Plan startLearning");
   solve cleanMemory time(0) onFailSwitchTo prologFailure;
   sense time(300000) usercmd -> continue ;
   switchToPlan checkStopLearning;
   switchToPlan learningLoop

 Plan checkStopLearning resumeLastPlan
   println("Plan checkStopLearning");
   memoCurrentEvent;
     [ ?? msg(usercmd, EVENT, WSOCK, NONE,
          usercmd(robotgui(k(stopLearning))), TIME) ] switchToPlan
          selectMode

Plan learningLoop
      println("Plan learningLoop");
     solve addMove time(0) onFailSwitchTo prologFailure;
     solve robotMoveFromUsercmdEvent(300000, usercmd,
          moveCallback) time(0) onFailSwitchTo prologFailure;
     repeatPlan 0

 Plan moveCallback
   println("Plan moveCallback");
   memoCurrentEvent;
   [ !? msg(usercmd, EVENT, WSOCK, NONE,
       usercmd(robotgui(k(stopLearning))), TIME) ] switchToPlan
       saveLastMove;
   switchToPlan learningLoop

 Plan saveLastMove
   solve addMove time(0) onFailSwitchTo prologFailure;
     switchToPlan selectMode

Plan autonomousStart
   println("autonomous");
   solve prepareForAutonomousLoop time(0) onFailSwitchTo
       prologFailure;
   switchToPlan loadMovesInProlog;
   switchToPlan loadMovesInJava

Plan autonomousDirect
```

```
   switchToPlan autonomousStart

Plan loadMovesInProlog resumeLastPlan
   println("loadMovesInProlog");
   solve loadMovesInProlog time(0)

Plan loadMovesInJava
   println("loadMovesInJava");
   solve loadTempMoveInJava time(0) onFailSwitchTo
       movesLoadedInJava;
   repeatPlan 0

Plan autonomousReverse
   println("autonomousReverse");
   addRule reverse(true);
   switchToPlan autonomousStart

Plan movesLoadedInJava
   println("movesLoadedInJava");
   switchToPlan autonomousPrepare

Plan autonomousPrepare
   println("autonomousPrepare");
   solve calculateTimes time(0) onFailSwitchTo prologFailure;
   [ ?? reverse(true) ] solve setReverseMode time(0)
       onFailSwitchTo prologFailure;
   switchToPlan loadFinalMovesInJava

Plan loadFinalMovesInJava
   println("loadFinalMovesInJava");
   solve isMovesEnded time(0) onFailSwitchTo prologFailure;
   [?? end] switchToPlan autonomousLoop;
   [?? notend]
   solve loadMoveInJava time(0) onFailSwitchTo prologFailure;
   repeatPlan 0

Plan autonomousLoop
   println("autonomousLoop");
   solve execMove(usercmd,doMoveCallback) time(0) onFailSwitchTo
       endAutonomousPhase;
   repeatPlan 0

Plan endAutonomousPhase
   println("endAutonomousPhase");
   switchToPlan selectMode

Plan doMoveCallback
   println("doMoveCallback");
   solve removeLeftAutonomousCommand time(0) onFailSwitchTo
       endAutonomousPhase;
```

```
      repeatPlan 0

   Plan prologFailure
      println(">>>>> [x] Plan prologFailure") ;
        println(">>>>> [x] failed to solve a Prolog goal" )
}
```

In order to store every move made by the robot during the learning mode and use them later during the autonomous phase we use these rules, defined in the file "applTheory.pl":

```
cleanMemory :-
   java_object("ApplSystem", [], Appl),
   amp(PATH),
   Appl <- reset(PATH).

addMove :-
   myname( Actor ),
   retract( msg( MSGID, MSGTYPE, SENDER, RECEIVER,
       usercmd(robotgui(CONTENT)), TIME ) ),
   actorPrintln( writeCurrentEV(Actor, CONTENT, TIME) ),
   java_object("ApplSystem", [], Appl),
   amp(PATH),
   Appl <- addMoveInFile(PATH, CONTENT, TIME).

loadMovesInProlog :-
   amp(PATH),
   consult(PATH).

loadTempMoveInJava :-
   retract(move(CMD, TIME)),
   java_object("ApplSystem", [], Appl),
   Appl <- addMove(CMD, TIME).

loadMoveInJava :-
   actorPrintln(loadMoveInJava),
   java_object("ApplSystem", [], Appl),
   Appl <- getCurrentCommand returns CMD,
   Appl <- getCurrentTime returns TIME,
   assertz(move(usercmd(CMD),TIME)),
   Appl <- setNextMove,
   actorPrintln(loadMoveInJavaEND).

setReverseMode :-
   java_object("ApplSystem", [], Appl),
   Appl <- setReverse.

calculateTimes :-
   java_object("ApplSystem", [], Appl),
```

```
   Appl <- calculateTimes.

prepareForAutonomousLoop :-
   java_object("ApplSystem", [], Appl),
   Appl <- prepareForAutonomousLoop.

isMovesEnded :-
   java_object("ApplSystem", [], Appl),
   Appl <- isMovesEnded returns END,
   actorPrintln("isMovesEnded"),
   asserta(END).

execMove(AlarmEvent, ReactionPlan) :-
   actorPrintln("[applTheory] execMove" ),
   actorPrintln(execMove(AlarmEvent, ReactionPlan)),
   retract( move(usercmd( CMD ), TIME )),
   actorPrintln( move(usercmd( CMD ), TIME ) ),
   doMove( AlarmEvent, CMD, TIME, ReactionPlan ).

doMove( AlarmEvent, EVMSG, EVTIME, RP ) :-
   actorPrintln(doMove( AlarmEvent, EVMSG, EVTIME, RP )),
   myname( R ),
   R <- getCurrentPlan returns CurPlanName,
   actorPrintln("sono qui"),
   robotMove(R, CurPlanName, EVMSG, EVTIME, AlarmEvent, RP ),
   actorPrintln( doMove( R, CurPlanName, AlarmEvent, EVMSG, EVTIME,
       RP ) ).

robotMove( Robot, CurPlanName, CONTENT, MoveTime, AlarmEvent,
    ReactionPlan ):-
   actorPrintln("[applTheory] Robot Move"),
   actorPrintln(robotMove( CurPlanName, CONTENT, MoveTime,
       AlarmEvent, ReactionPlan )),
   moveCmdTable(CONTENT, RobotCmd, Speed ),
   Robot <- execRobotMove(CurPlanName, RobotCmd, Speed, 0, MoveTime,
       AlarmEvent, ReactionPlan) returns B,
   actorPrintln( robotMove( CurPlanName, RobotCmd , Speed,
       AlarmEvent, ReactionPlan, B ) ),
   B == true.

removeLeftAutonomousCommand :-
   retract(move(CMD, TIME)).
```

The Prolog rules described above will be calling the following functions, implemented in "ApplSystem.java":

```
/* ApplSystem is written by the Application designer
 * in order to define operations that can be used
 * from Prolog rules (see prolog-files/applTheory.pl) */
```

```
import it.unibo.contactEvent.platform.EventPlatformKb;
import it.unibo.is.interfaces.IOutputView;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;

public class ApplSystem {
   private static IOutputView outView = EventPlatformKb.stdOutView;
   private static ArrayList<Move> moves = new ArrayList<Move>();
   private static int currentMove = 0;

   public static void setOutView(IOutputView outViewArg){
      outView = outViewArg;
   }

   public static ArrayList<String> readFile(String fName){
      ArrayList<String> result = new ArrayList<String>();
      try {
         outView.addOutput(" *** ApplSystem readFile " + fName);
         InputStream fs        = new java.io.FileInputStream(fName);
         InputStreamReader inpsr = new InputStreamReader(fs);
         BufferedReader br     = new BufferedReader(inpsr);
         Iterator<String> lsit = br.lines().iterator();
         while(lsit.hasNext()){
            result.add(lsit.next());
         }
         br.close();
      } catch (Exception e) {
         outView.addOutput(" *** ApplSystem ERROR " + e.getMessage());
      }
      return result ;
   }

   public static void clearFile(String fName) {
      try {
         FileOutputStream fsout = new FileOutputStream(new
             File(fName));
         fsout.write(("").getBytes());
         fsout.close();
      } catch (Exception e) {
         outView.addOutput(" *** ApplSystem ERROR " + e.getMessage());
      }
   }

   public static void writeInFile(String fName, String content) {
```

```
    writeInFile(fName, content, "true");
}

public static void writeInFile(String fName, String content,
     String append) {
    try {
        outView.addOutput(" [ApplSystem] ApplSystem writeInFile " +
            fName);
        FileOutputStream fsout = new FileOutputStream(new
            File(fName),
             append.toLowerCase() == "true" ? true : false);
        fsout.write((content+"\n").getBytes());
        fsout.close();
    } catch (Exception e) {
        outView.addOutput(" [ApplSystem] ApplSystem ERROR " +
            e.getMessage());
    }
}

public void reset(String fName){
    clearFile(fName);
}

public void addMove(String cmd, String time){
    Move m = new Move(cmd, Long.parseLong(time));
    moves.add(m);
}

public void addMoveInFile(String fName, String cmd, String time){
    Move m = new Move(cmd, Long.parseLong(time));
    writeInFile(fName, m.getDefaultRep());
}

public void setReverse(){
    ArrayList<Move> reverse = new ArrayList<Move>();
    for(Move m : moves){
        reverse.add(m.reversed());
    }
    Collections.reverse(reverse);
    moves = reverse;
}

public void calculateTimes(){
    for (int i = 0; i < moves.size() - 1; i++) {
        moves.get(i).setTime(moves.get(i + 1).getTime() -
            moves.get(i).getTime());
        outView.addOutput(" [DIFF] " + moves.get(i).getTime());
    }
    moves.remove(moves.size() - 1);
}
```

```
    public void prepareForAutonomousLoop(){
       moves = new ArrayList<Move>();
       currentMove = 0;
    }

    public String getCurrentCommand(){
       return moves.get(currentMove).getCommand();
    }

    public Long getCurrentTime(){
       return (moves.get(currentMove).getTime());
    }

    public void setNextMove(){
       currentMove++;
    }

    public String isMovesEnded(){
       if (currentMove >= moves.size()){
          return "end";
       }
       return "notend";
    }
}
```

– Sprint Review:
Into the Product Backlog of this sprint there is now the model of the Robot which is able to execute received commands, to save them and execute them again autonomously.

6.3 Risk analysis

In this phase there are highlighted the main risks that could occur during the development process or as effects on the system. The definition of the logic architecture allows us to highlight the most critical parts for the project and development steps, in relation to resources, competences and available technologies.

The effects on the system can be of different nature and can lead to setbacks in the short term; in some cases, however, the failure of a software system can lead to economic losses, defining the so-called critical systems.

The system built following the logic architecture defined will surely be safe, reliable and available any time the user wants. Nevertheless, some risks during the designing and the development of the project could take place:

– the development could take more time than expected;

- there could be a lack of resources/knowledge, but this would only increase the amount of time needed to complete the assignment;
- the fault tolerance will have to be considered as many operations will have to be carried out while others are still being computed. There is a chance for an operation to cause exceptions during the execution of the program.
- regarding the security of the system, it is necessary to point out that the software will comply with the specifics. This said, the prerequisites do not include (as an example) protocols to prevent intruders to control the robot instead of the customer. It could be necessary the adoption of a more advanced level of security, in order to grant the integrity and consistency of the data exchanged. In any case, this would be an important improvement for the future.

Considering the explained risks the project can be considered having a mid-low level of risk, without any critical risk on the long term.

7 Work plan

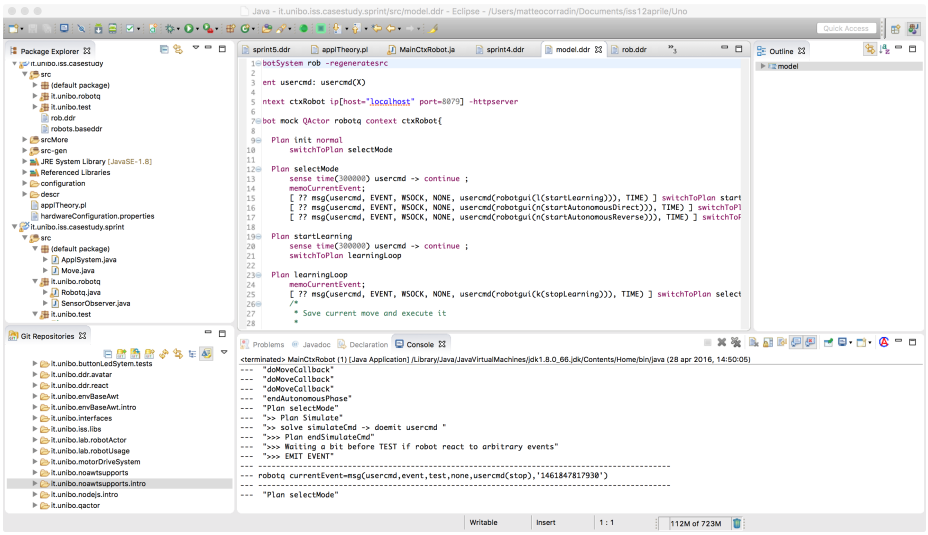
The work plan is structured in an iterative development of progressively more advanced prototypes. The analysis phase becomes yet again paramount in order to correctly handle the work plan definition phase and the distribution of the duties, while keeping a constant attention to the risks highlighted during the problem analysis. As specified before, the SCRUM model will cover an important role during the whole development process. The prototypes will be built following its framework. This said, while the project is the result of the efforts of all the members of the team together, we have identified some areas of expertise in which each member excels and given them priority in order to speed up the process. Marco Cavalazzi has taken the lead writing down and reviewing the ideas of the team in the project report, Matteo Corradin has committed to the development of the model and the code regarding the virtual environment and Lorenzo Monti has chosen to realize the real-life robot and carry out the testing phase.

8 Development tools

The main development tool necessary for the project is the Eclipse IDE. Through Eclipse we are able to use the XText framework and the QActors DSL described before.

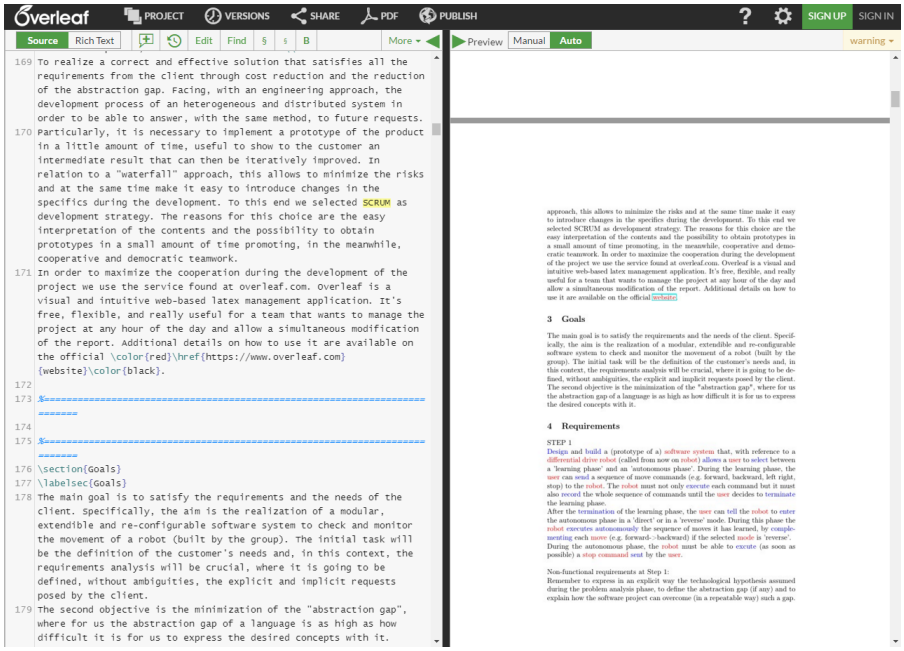
Beyond that, in order to have a real-time collaboration during the development of the project the team has chosen to use WhatsApp® and Skype®. This way it is possible for the SCRUM team to have a high level of interactions any time of the day, exploiting both **synchronous** and **asynchronous** communication. Through Skype the development is made easy even when some member of the team cannot physically reach the workplace, allowing messaging, audio and video conversations. Through the "screen sharing" function it is possible to

operate a distant version of the software development methodology known as *Extreme programming*.



Example of screen sharing through Skype®

In the interest of getting a real-time preview of the outcome we chose to use Overleaf. Overleaf is a visual and intuitive web-based latex management application. It's free, safe, and really useful for a team that wants to manage the project at any hour of the day and allow a simultaneous modification of the report. Additional details are available on the official website.



Example of an Overleaf's interface

9 Project & Implementation

The whole system, with its *structure, interaction* and *behaviour* components, has already been shown in the previous sections. In order to avoid redoundancy we prefer to forward you to the **section 6.2** of this paper.
The whole project can be found in the package "it.unibo.iss.casestudy.sprint".

10 Testing

Software testing involves the execution of a software component or system component to evaluate one or more properties of interest. In general, these properties indicate the extent to which the component or system under test:

- meets the requirements that guided its design and development,
- responds correctly to all kinds of inputs,
- performs its functions within an acceptable time,
- is sufficiently usable,
- can be installed and run in its intended environments, and
- achieves the general result its stakeholders desire.

Purpose of the testing is to find "failures" so that, once the software failed a test, a solution for those faults (responsible for the failure) can be found.

In order to test the system a **mock object** has been created and run. We developed many Prolog rules and through those tested the outcome of every input that may come from the user.

After this preliminary test we also tested the code after the deployment, loading the code built so far on a differential-drive robot. This way, we have been able to test the system in a physical environment, where we have been able to check its behaviour thoroughly.

11 Deployment

The release of the software system to the client will take place through the distribution and installation of the various files on the robot:

- rob.jar: The main application.
- rob_lib/*.jar: Used libraries.
- applTheory.pl: Application level prolog theory
- srcMore/it/unibo/ctxRobot/QActorWebUi2.*: Web user interface.
- srcMore/it/unibo/robotq/*: Generated files.

An SSH connection to the robot will provide us the bash necessary to launch the project. Using the bash, the software can be run with this command:
sudo java -jar rob.jar
The GUI is then available at: http://robot_ip:8080
where "robot_ip" is the IP given to the robot.

11.1 GUI

In this paragraph we show the Graphical User Interface (GUI) built for the project.

The GUI shown is the graphical interface that is going to be shown to the user when connected to the system. Through the GUI the user can interact with the robot, following the behaviour explained during the domain model's definition phase.

12 Maintenance

We know that systems continue to evolve over time. Maintenance is the evolutionary development done in order to keep the software aligned with customer's priorities.
There are many types of maintenance that we, as project managers and developers, have to cover:

– preventive maintenance: increasing software maintainability or reliability to prevent problems in the future

- corrective maintenance: it involves all the corrections that will have to be made in order for the system to pass the testing phase
- evolutionary maintenance:
  - adaptive maintenance: the target is to modify the system to cope with changes in the software environment
  - perfective maintenance: implementing new or changed user requirements which concern functional enhancements to the software

Thanks to our model based on the QActors DSL we have an environment that gives us an auto-generated code. This will make the product updates and maintenance much easier because the cost of a change in the logic architecture is really low. The new source code will be auto-generated in order to match the improvements that take place.

A Project upgrade

At this point, we want to improve the project discussed in the previous chapters through the STEP 2 of the requirements.

B Introduction

The robot needs now to perceive the world around it, reacting to it in a human-like fashion. Through this project upgrade we will demonstrate that modifying the concepts defined in the previous project it is possible to create a product with a new kind of capabilities with a minimal effort.

C Vision

A redesigned software is significantly altered by the new requirements, that affect previous analysis like the logic architecture. This said, the code of the software must change in order to meet the new needs of the customer. With this project we aim to discuss how requirement shifts affect the software redesign when artifacts like the QActors DSL are available. Our Vision is to be able to realize the modifications requested as a monotonous extension of the system using as little effort as possible.

D Goals

The aim of this project is to demonstrate that the QActors DSL is an effective technology that manages to shrink the gap between the model and the implementation. In other words, it decreases the amount of work to be done in order to satisfy the requirements change. Thus, it allows the rapid prototyping of a new technology-independent software with a minimal effort.

E Requirements

STEP 2 (Optional)
After the development of the project, consider the possibility to enhance the functional capabilities of the robot, by allowing it:
to perceive an obstacle during the autonomous phase and, once the obstacle is detected, to execute some alternative behavior (in term of moves).

Non-functional requirements at Step 2:
The main goal is to discuss how some change (monotonic extensions) of the requirements impact on a product whose production is based on 'formal', 'technology independent' artifacts rather than on ad-hoc code.
During this phase, the software could optionally define some modification/extension to the QA/DDR DSL in order to fulfill a specific goal of the team.

F Requirements analysis

F.1 Glossary

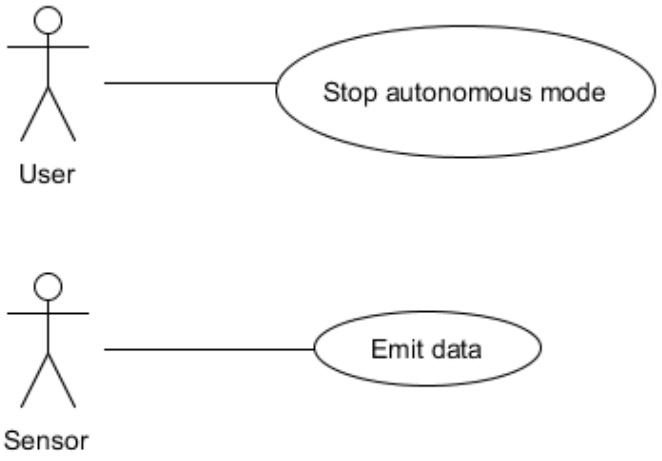
The analysis of the nouns (highlighted in red) facilitates the writing of a glossary, necessary in order to use a common language, free of any ambiguities, with the customer and all the subjects involved.

Terms glossary

Term	Meaning
robot	a machine capable of carrying out a complex series of actions automatically, especially one programmable by a computer.
obstacle	a thing that blocks one's way or prevents or hinders progress.
behaviour	the way in which the robot behaves in response to a particular situation or stimulus.
move	to go in a specified direction or manner; to change position.

F.2 Use cases

We will here show the new use cases introduced.



Autonomous Phase Use Case Diagram

F.3 Scenarios

We will hereunder describe in more detail the use cases of the project, explaining all the possible scenarios.

Use Case:	**Autonomous Scenario**
Description:	The user decides to press the autonomous phase button. The robot will follow the commands that are already in its memory.
Actors:	User, Sensor
Preconditions:	The robot must be ready to receive the commands. The robot interface must be accessible by the user.
Main Scenario:	The user chooses the "direct" or "reverse" mode and presses the Autonomous phase button. The robot will do the commands stored in its memory. During their execution it will react to the distance sensor's data. The user can press anytime the Stop button to arrest the robot.
Secondary Scenarios:	While in autonomous mode, if the distance sensor reads the presence of an object in front of the robot less than 20cm far the robot halts.
Post-conditions:	The system is ready to receive a new command

F.4 (Domain)model

Using the DSL, we can simultaneously display the structure, behaviour and, most importantly, the interactions of the system's components in a formal manner:

```
RobotSystem rob -regeneratesrc

Event usercmd: usercmd(X)
Event obstacle : obstacle(X)

Context ctxRobot ip[host="localhost" port=8079] -httpserver

Robot mock QActor robotq context ctxRobot{

   Plan init normal
      switchToPlan selectMode

   Plan selectMode
        sense time(300000) usercmd -> continue ;
        memoCurrentEvent;
         [ ?? msg(usercmd, EVENT, WSOCK, NONE,
              usercmd(robotgui(l(startLearning))), TIME) ] switchToPlan
              startLearning ;
        [ ?? msg(usercmd, EVENT, WSOCK, NONE,
            usercmd(robotgui(n(startAutonomousDirect))), TIME) ]
            switchToPlan autonomousDirect ;
        [ ?? msg(usercmd, EVENT, WSOCK, NONE,
            usercmd(robotgui(n(startAutonomousReverse))), TIME) ]
            switchToPlan autonomousReverse

   Plan startLearning
      sense time(300000) usercmd -> continue ;
      switchToPlan learningLoop

   Plan learningLoop
      memoCurrentEvent;
      [ ?? msg(usercmd, EVENT, WSOCK, NONE,
          usercmd(robotgui(k(stopLearning))), TIME) ] switchToPlan
          selectMode;
      /*
       * Save current move and execute it
       */
      sense time(300000) usercmd -> continue ;
        repeatPlan 0

   Plan autonomousStart
      /* Set moves lenght
       * Save moves in KB, if moves are ended switch to autonomousLoop */
      repeatPlan 0
```

```
   Plan autonomousDirect
      switchToPlan autonomousStart

   Plan autonomousReverse
      addRule reverse(true);
      switchToPlan autonomousStart

   Plan autonomousLoop
      /* execute every move
       * if moves are ended switch to endAutonomousPhase
       * Added another event (obstacle) to handle
       *
       */
       solve execMove("obstacle,usercmd", "evhSensor,evhCmd") time(0)
            onFailSwitchTo endAutonomousPhase ;
      repeatPlan 0

   Plan evhCmd
      switchToPlan endAutonomousPhase

    Plan evhSensor
      switchToPlan endAutonomousPhase

   Plan endAutonomousPhase
      switchToPlan selectMode

}
```

F.5 Test plan

At this point, the elements that compose the logic architecture can provide us enough data on what the different subsystems have to realize, without having to detail their internal behaviour yet. The test plan here described allows us to comprehend and specify the expected behaviour of an entity before its design and implementation.

```
RobotSystem rob -regeneratesrc

Event usercmd: usercmd(X)
Event sensordata : sensordata(X)
Event obstacle : obstacle(X)

Context ctxRobot ip[host="localhost" port=8079] -httpserver

QActor test context ctxRobot {

   Rules{
      loadTheory(FName) :-
```

```
        actorPrintln( loadTheory(FName) ),
        consult( FName ).
    cmd(usercmd(robotgui(l(startLearning)))).
    cmd(usercmd(robotgui(w(high)))).
    cmd(usercmd(robotgui(a(high)))).
    cmd(usercmd(robotgui(w(high)))).
    cmd(usercmd(robotgui(k(stopLearning)))).
    cmd(usercmd(robotgui(n(startAutonomousDirect)))).
    cmd(obstacle(test)).

    simulateCmd :-
        retract( cmd( CMD ) ),
        doemit( "usercmd", CMD ).
    simulate(yes).
}

Plan init normal
    delay time(1000);
    println("> Plan INIT") ;
    solve loadTheory("applTheory.pl") time(0) onFailSwitchTo
        prologFailure;
    [ !? simulate(yes) ] switchToPlan simulate

Plan simulate
    println(">> Plan Simulate") ;
    println(">> solve simulateCmd -> doemit usercmd ");
    solve simulateCmd time(0) onFailSwitchTo endSimulateCmd ;
    delay time(1000);
    repeatPlan 0

Plan endSimulateCmd
    println(">>> Plan endSimulateCmd" ) ;
    println(">>> Waiting a bit before TEST if robot react to arbitrary
        events") ;
    delay time(3000) ;
    println(">>> EMIT EVENT") ;
    emit usercmd : usercmd(stop)

Plan prologFailure
    println(">> Plan prologFailure" )

}
```

G Problem analysis

G.1 Logic architecture

At this point it is possible to switch to the problem analysis, following the SCRUM model. The system's logic architecture constitutes the synthetical artifact of the analysis. Its purpose is to better define the subsystems of the problem, suggesting a solving system. The logic architecture itself will be the reference for the next steps of the project, thus it has to be as much as independent as possible in respect to the implementation environment. We will now expand every sprint of the SCRUM development process and examine them.

SPRINT 1:

– Sprint Planning:
We want to further improve the functional capabilities of the robot, by allowing it to perceive an obstacle during the autonomous phase and, once the obstacle is detected, to execute some alternative behavior (in this case we want to stop the Robot).
To check the presence of an obstacle in front of the Robot it uses a distance sensor. This sensor can be modeled, following the pattern Observer, as an Observable that will notify the data taken from the physical sensor to the robot.
We want to generate an event when the distance perceived by the sensor changes and, in particular, is lower than a certain threshold (in this case set to 20cm).
As in the previous sprint, it is necessary to introduce a new functionality with a reactive behaviour.
The model that describes the robot after this phase is here shown:

```
RobotSystem rob -regeneratesrc

Event usercmd: usercmd(X)
Event sensordata : sensordata(X)
Event obstacle : obstacle(X)

Context ctxRobot ip[host="localhost" port=8079] -httpserver

Robot mock QActor robotq context ctxRobot{
   Rules{
      loadTheory(File) :-
         actorPrintln( loadTheory(File) ),
         consult( File ).
      amp("autonomous.pl"). //autonomous mode path
   }

   Plan init normal
      println("Ready!");
```

```
   solve loadTheory("applTheory.pl") time(0) onFailSwitchTo
       prologFailure;
   switchToPlan selectMode

Plan selectMode
   println("Plan selectMode");
     sense time(300000) usercmd -> continue ;
     printCurrentEvent;
     memoCurrentEvent;
      [ ?? msg(usercmd, EVENT, WSOCK, NONE,
          usercmd(robotgui(l(startLearning))), TIME) ]
          switchToPlan startLearning ;
     [ ?? msg(usercmd, EVENT, WSOCK, NONE,
         usercmd(robotgui(n(startAutonomousDirect))), TIME) ]
         switchToPlan autonomousDirect ;
     [ ?? msg(usercmd, EVENT, WSOCK, NONE,
         usercmd(robotgui(n(startAutonomousReverse))), TIME) ]
         switchToPlan autonomousReverse ;
     removeRule msg(usercmd, EVENT, WSOCK, NONE, usercmd,TIME);
     repeatPlan 0

Plan startLearning
   println("Plan startLearning");
   solve cleanMemory time(0) onFailSwitchTo prologFailure;
   sense time(300000) usercmd -> continue ;
   switchToPlan checkStopLearning;
   switchToPlan learningLoop

 Plan checkStopLearning resumeLastPlan
   println("Plan checkStopLearning");
   memoCurrentEvent;
     [ ?? msg(usercmd, EVENT, WSOCK, NONE,
         usercmd(robotgui(k(stopLearning))), TIME) ] switchToPlan
         selectMode

Plan learningLoop
      println("Plan learningLoop");
     solve addMove time(0) onFailSwitchTo prologFailure;
     solve robotMoveFromUsercmdEvent(300000, usercmd,
         moveCallback) time(0) onFailSwitchTo prologFailure;
     repeatPlan 0

 Plan moveCallback
   println("Plan moveCallback");
   memoCurrentEvent;
   [ !? msg(usercmd, EVENT, WSOCK, NONE,
       usercmd(robotgui(k(stopLearning))), TIME) ] switchToPlan
       saveLastMove;
   switchToPlan learningLoop
```

```
 Plan saveLastMove
   solve addMove time(0) onFailSwitchTo prologFailure;
     switchToPlan selectMode

Plan autonomousStart
   println("autonomous");
   solve prepareForAutonomousLoop time(0) onFailSwitchTo
       prologFailure;
   switchToPlan loadMovesInProlog;
   switchToPlan loadMovesInJava

Plan autonomousDirect
   switchToPlan autonomousStart

Plan loadMovesInProlog resumeLastPlan
   println("loadMovesInProlog");
   solve loadMovesInProlog time(0)

Plan loadMovesInJava
   println("loadMovesInJava");
   solve loadTempMoveInJava time(0) onFailSwitchTo
       movesLoadedInJava;
   repeatPlan 0

Plan autonomousReverse
   println("autonomousReverse");
   addRule reverse(true);
   switchToPlan autonomousStart

Plan movesLoadedInJava
   println("movesLoadedInJava");
   switchToPlan autonomousPrepare

Plan autonomousPrepare
   println("autonomousPrepare");
   solve calculateTimes time(0) onFailSwitchTo prologFailure;
   [ ?? reverse(true) ] solve setReverseMode time(0)
       onFailSwitchTo prologFailure;
   switchToPlan loadFinalMovesInJava

Plan loadFinalMovesInJava
   println("loadFinalMovesInJava");
   solve isMovesEnded time(0) onFailSwitchTo prologFailure;
   [?? end] switchToPlan autonomousLoop;
   [?? notend]
   solve loadMoveInJava time(0) onFailSwitchTo prologFailure;
   repeatPlan 0

Plan autonomousLoop
   println("autonomousLoop");
```

```
      solve execMove("obstacle,usercmd", "evhSensor,evhCmd") time(0)
          onFailSwitchTo endAutonomousPhase ;
      repeatPlan 0

   Plan evhCmd
      println("evhCmd");
      printCurrentEvent;
      switchToPlan doMoveCallback

    Plan evhSensor
      println("evhSensor");
      printCurrentEvent;
      switchToPlan doMoveCallback

   Plan endAutonomousPhase
      println("endAutonomousPhase");
      switchToPlan selectMode

   Plan doMoveCallback
      println("doMoveCallback");
      solve removeLeftAutonomousCommand time(0) onFailSwitchTo
          endAutonomousPhase;
      repeatPlan 0

   Plan prologFailure
      println(">>>>> [x] Plan prologFailure") ;
        println(">>>>> [x] failed to solve a Prolog goal" )
}
```

The pattern observer is used to handle sensor's data.

```
package it.unibo.robotq;
import org.json.JSONException;
import org.json.JSONObject;

import it.unibo.contactEvent.interfaces.IContactEventPlatform;
import it.unibo.contactEvent.platform.ContactEventPlatform;
import it.unibo.iot.models.sensorData.ISensorData;
import it.unibo.iot.models.sensorData.SensorType;
import it.unibo.iot.sensors.ISensorObserver;
import it.unibo.is.interfaces.IOutputView;
import it.unibo.system.SituatedPlainObject;

public class SensorObserver<T extends ISensorData> extends
    SituatedPlainObject implements ISensorObserver<T>{
protected  IContactEventPlatform platform;
   public SensorObserver(IOutputView outView) {
      super(outView);
      try {
```

```
            platform = ContactEventPlatform.getPlatform( );
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    @Override
    public void notify(T data) {
        if (data.getType() == SensorType.DISTANCE) {
            double threshold = 20.0;
            JSONObject dataJson;
            try {
                dataJson = new JSONObject(data.getJsonStringRep());
                double dist = dataJson.getJSONObject("d").getDouble("cm");
                if (dist < threshold) {
                    platform.raiseEvent("sensor", "obstacle",
                        data.getDefStringRep());
                }
            } catch (JSONException an){
                an.printStackTrace();
            }
        }else{
            platform.raiseEvent("sensor", "sensordata",
                data.getDefStringRep() );
        }
    }

}
```

– Sprint Review:
Once this sprint is completed, we have the model of the Robot which satisfies all the customer requirements.

G.2 Risk analysis

In this phase there are highlighted the main risks that could occur during the development process or as effects on the system. The definition of the logic architecture allows us to highlight the most critical parts for the project and development steps, in relation to resources, competences and available technologies.

The effects on the system can be of different nature and can lead to setbacks in the short term; in some cases, however, the failure of a software system can lead to economic losses, defining the so-called critical systems.

The system built following the logic architecture defined will surely be safe, reliable and available any time the user wants. Nevertheless, some risks during the designing and the development of the project could take place:

- the development could take more time than expected;
- the fault tolerance will have to be considered, as many operations will have to be carried out while others are still being computed. There is a chance for an operation to cause exceptions during the execution of the program.
- regarding the security of the system, it is necessary to point out that the software will comply with the specifics. This said, the prerequisites do not include (as an example) protocols to prevent intruders to control the robot instead of the customer. It could be necessary the adoption of a more advanced level of security, in order to grant the integrity and consistency of the data exchanged. In any case, this would be an important improvement for the future.

Considering the explained risks the project can be considered having a low level of risk, without any critical risk on the long term.

H Work plan

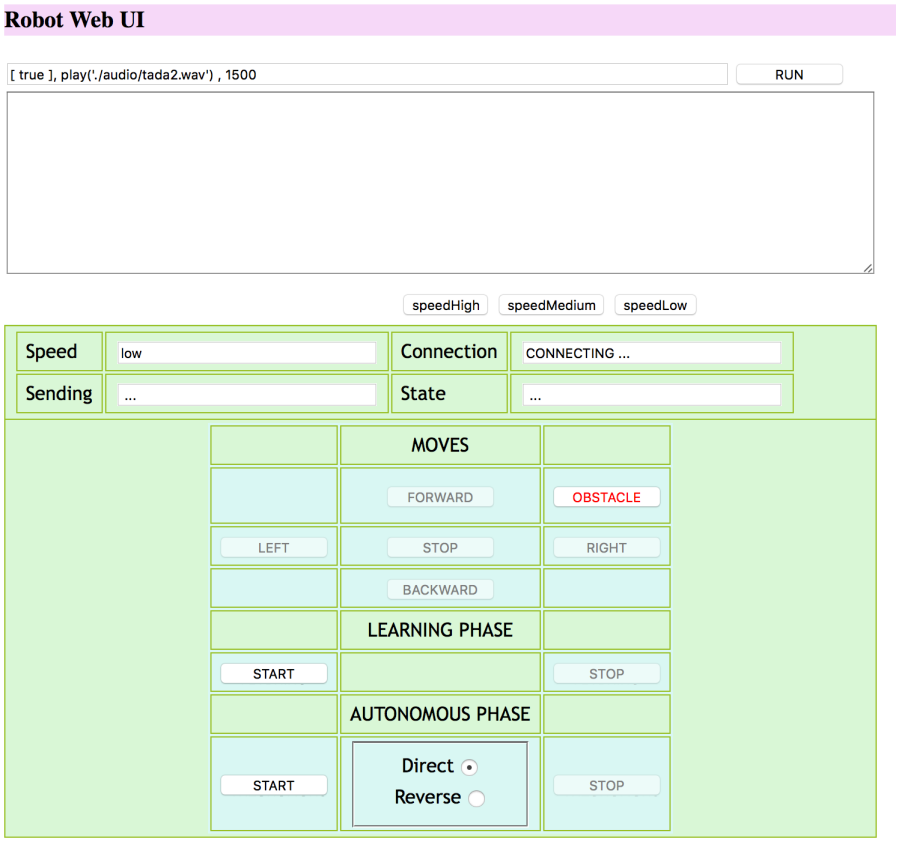
The work plan is structured in an iterative development of progressively more advanced prototypes. The analysis phase becomes yet again paramount in order to correctly handle the work plan definition phase and the distribution of the duties, while keeping a constant attention to the risks highlighted during the problem analysis. As specified before, the SCRUM model will cover an important role during the whole development process. The prototypes will be built following its framework. This said, while the project is the result of the efforts of all the members of the team together, we have identified some areas of expertise in which each member excels and given them priority in order to speed up the process. Marco Cavalazzi has taken the lead writing down and reviewing the ideas of the team in the project report, Matteo Corradin has committed to the development of the model and the code regarding the virtual environment and Lorenzo Monti has taken care of the real-life robot and the testing phase.

I Project & Implementation

The whole system, with its *structure, interaction* and *behaviour* components, has already been shown in the previous sections. In order to avoid redundancy we prefer to forward you to the **section G** of this paper.
The whole project can be found in the package "it.unibo.iss.casestudy.sprint".

J GUI

In this paragraph we show the Graphical User Interface (GUI) built for the project.



The new GUI introduces a new button that allows the user to simulate the presence of an obstacle in the robot's path.

K Information about the authors

Marco Carlo Cavalazzi

- Qualification: Bachelor Degree in Computer Sciences and Technologies gained at the University of Bologna
- Interests: Computer Science, Arts, Architecture, Energy, Management, Space
- He has passed to date the following number of exams: 12
- It expects to graduate in the session of: June-July 2016
- He intends to continue his studies in order to achieve: Master in Business Administration at the university of: Bocconi, Milano
- Worked at : C.E.R.N.
- Intends to work at : European Space Agency, C.E.R.N., SpaceX or Tesla.

Matteo Corradin

- Qualification: Bachelor Degree in Computer Sciences and Technologies gained at the University of Bologna
- Interests: Computer Science, Music, Travel, Games.

Lorenzo Monti

- Qualification: Bachelor Degree in Computer Sciences and Technologies gained at the University of Bologna
- Interests: Computer Science, Music, Art, Astronomy, Travel.

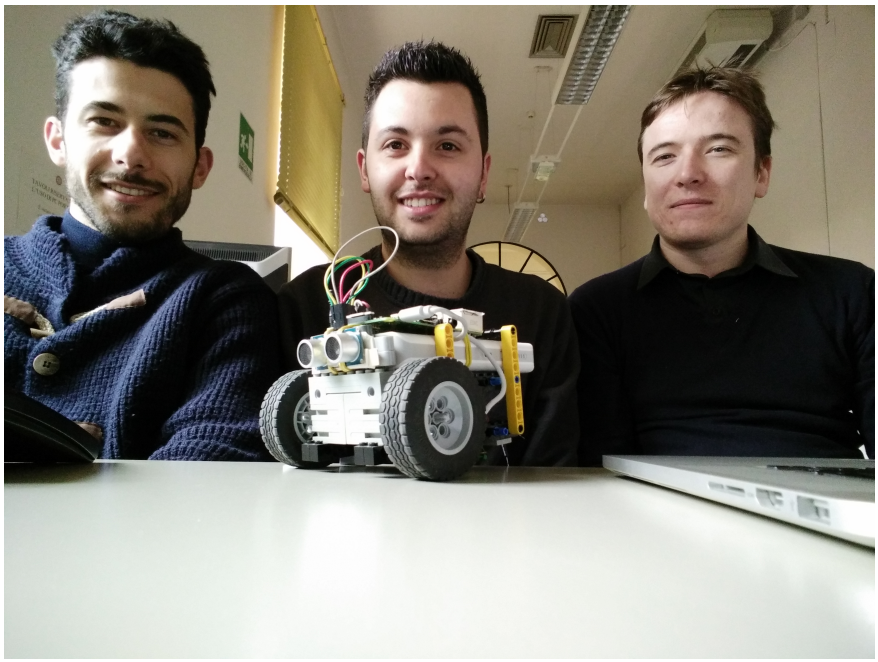


Photo of the authors.
From left to right: Marco Carlo Cavalazzi, Lorenzo Monti, Matteo Corradin