

Introduction to UniboDISI DDR-Robots

(2018)

Antonio Natali

Alma Mater Studiorum – University of Bologna
via dell'Università 50,47521 Cesena, Italy,
Viale Risorgimento 2,40136 Bologna, Italy
antonio.natali@studio.unibo.it

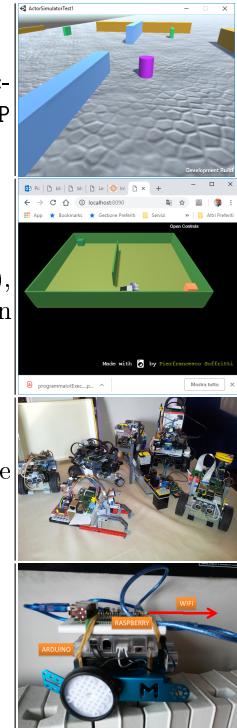
Table of Contents

Introduction to UniboDISI DDR-Robots (2018)	1
<i>Antonio Natali</i>	
1 Virtual and Real robots	3
2 The virtual environments	4
2.1 A virtual environment based on Unity (<code>uenv</code>)	4
2.2 A virtual environment based on JavaScript (<code>wenv</code>)	6
2.2.1 A virtual environment based on JavaScript (<code>wenv</code>)	7
2.2.2 The Application class <code>basicRobotExecutor</code>	9
3 The in-house (nano) robot	10
3.1 Beyond the hardware level	11
3.2 A model for the <code>BaseRobot</code>	12
3.3 The <code>BasicRobot</code> class	13
3.4 Using a <code>BaseRobot</code>	14
3.4.1 The project workspace	14
3.4.2 The code	14
3.5 The work of the Configurator	16
3.6 From mocks to real robots	17
3.7 Sensors and Sensor Data	18
3.7.1 Sensor data representation in Prolog (high level)	19
3.7.2 Sensor data representation in Json (low level)	19
3.8 Sensor model	19
3.9 Actuators and Executors	21
4 The mbot robot	23
4.0.1 IOT reference architecture	23
4.1 The low-level code on Arduino	24
4.1.1 Mblock software	27
4.2 <code>mbotConnArduinoObj</code>	29
4.2.1 Init	29
4.2.2 Output	29
4.2.3 Observer	30
4.2.4 Input	30
4.3 The adapter <code>basicRobotExecutor</code>	31
5 A robot-command executor	32
5.1 The adapter	33
5.2 Activating the <code>wenv</code>	34

1 Virtual and Real robots

Our company (**Unibo**) has developed:

A virtual environment (named **U-Env**) based on the **Unity** system (see Subsection 2.1), that includes a virtual robot that accepts commands sent on a TCP connection on port 8090.



A virtual environment (named **W-Env**) built in **JavaScript** (see Subsection 2.2.1), that includes a virtual robot that accepts commands sent on a TCP connection on port 8999.

A physical **ddr** robot built in-house (see Section 3), modelled as an observable POJO.

A physical robot based upon a **mBot robot** (see Section 4).

2 The virtual environments

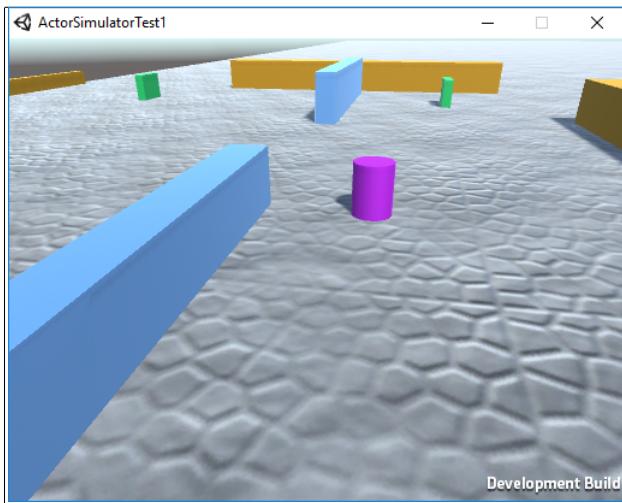
During software analysis and testing, it could be preferable to use virtual devices rather real, physical devices.

2.1 A virtual environment based on Unity (uenv)

Unity can be used as an advanced simulation environment, able to provide a virtual working environment and sensor data.

Our virtual environment built upon Unity (called **uenv**) is included in the file:¹
it.unibo.issMaterial/issdocs/Lab/virtualRobot.zip.

Let us *unzip* this file, and run **VirtualRobotE80.exe**. We obtain a scene showing an environment made of a set of walls and fixed obstacles, a mobile obstacle (the cylinder) and a sonar (the small boxes in green, named **sonar1** and **sonar2**):



The original Unity environment has been modified to interact with *QActor* systems.

The operation **createSimulatedActor** can be used in a *QActor* model to 'inject' into the virtual environment a qactor (to be called **rover**, at the moment) that can be moved in a Unity scene by using proper commands (**onward**, **backwards**, **left**, **right**, **stop**).

When the **rover** is intercepted by one of the sonar, the (modified) Unity system emits the event **sonar : sonar(SONAR, TARGET, DISTANCE)**.

Moreover, the virtual game object that represents the **rover** is equipped (in its front) with a sonar, that emits the event **sonarDetect : sonarDetect(X)** when detects an obstacle.

Both these events that can be handled by our applications. For example:

```
1 System testUnityRover
2 Event sonarDetect : sonarDetect(X) //From (virtual robot) sonar
3 Event sonar      : sonar(SONAR, TARGET, DISTANCE) //From (virtual) sonar
4
5 Context ctxRover ip [ host="localhost" port=8070 ]
6 EventHandler evh for sonarDetect , sonar -print ;
7
```

¹ You could clone the project [it.unibo.issMaterial](http://it.unibo.issMaterial/issdocs/Lab/virtualRobot.zip) by executing: git clone <https://github.com/anatali/iss2018.git>

```

8 | QActor rover context ctxRover {
9 |     Plan init normal [
10|         println("rover START");
11|         javaOp "workWithUnity(\"localhost\")" ;
12|         javaOp "createSimulatedActor(\"rover\", \"Prefabs/CustomActor\")" ;
13|         right 50 time ( 1000 ) //position
14|     ]
15|     switchTo moveVirtualRobot
16|
17|     Plan moveVirtualRobot [
18|         println("moveVirtualRobot")
19|     ]
20|     reactive onward 40 time ( 5000 )
21|         whenEnd -> endOfMove
22|         whenTout 30000 -> handleTout
23|         whenEvent sonarDetect -> handleObstacle
24|             or whenEvent sonar -> handleSonar
25|     finally repeatPlan
26|
27|     Plan handleSonar resumeLastPlan [
28|         onward 50 time ( 300 ) ; //out of sonar range
29|         stop 50 time ( 1000 )    //stop for a while ...
30|     ]
31|     Plan handleObstacle resumeLastPlan [ backwards 50 time ( 3500 ) ]
32|     Plan endOfMove resumeLastPlan [ println("endOfMove") ]
33|     Plan handleTout [ println("handleTout") ]
34|
35| }

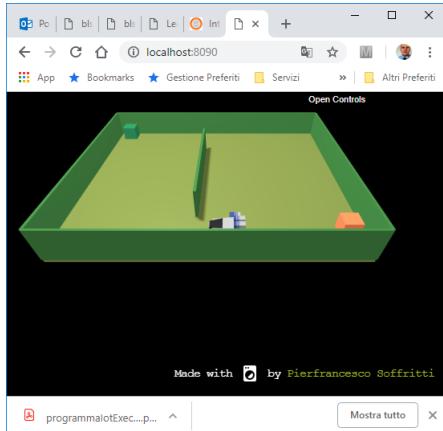
```

Listing 1.1. testUnityRover.qa

The point to highlight here is that, when the virtual robot (`rover`) is intercepted by a sonar, the modified Unity system emits the `QActor` event `sonar : sonar(SONARNAME, TARGET, DISTANCE)` where `SONARNAME` is `sonar1` or `sonar2`. Moreover, the sonar on the front of the `rover` emits the event `sonarDetect : sonarDetect(TARGET)` when it detects an obstacle.

2.2 A virtual environment based on JavaScript (wenv)

This virtual environment (called **wenv**) has been built by Pierfrancesco Soffritti² using the <https://threejs.org/> JavaScript library.



The scene is described in a configuration file. For example:

```
1 const config = {
2   floor: {
3     size: { x: 34, y: 30 }
4   },
5   player: {
6     position: { x: 0.065, y: 0.075 },
7     speed: 0.2
8   },
9   sonars: [
10   {
11     name: "sonar2",
12     position: { x: 0.94, y: 0.88},
13     senseAxis: { x: true, y: false }
14   },
15   ],
16   movingObstacles: [
17   ],
18   staticObstacles: [
19   {
20     name: "wallUp",
21     centerPosition: { x: 0.5, y: 1},
22     size: { x: 1, y: 0.01}
23   },
24   {
25     name: "wallDown",
26     centerPosition: { x: 0.5, y: 0},
27     size: { x: 1, y: 0.01}
28   },
29   {
30     name: "wallLeft",
31     centerPosition: { x: 0, y: 0.5},
32     size: { x: 0.01, y: 1}
33   },
34   {
35     name: "wallRight",
36     centerPosition: { x: 1, y: 0.5},
37     size: { x: 0.01, y: 1}
38   },
39   {
40     name: "obstacle",
```

² See <https://github.com/PierfrancescoSoffritti/ConfigurableThreejsApp>

```

41         centerPosition: {x: 0.45, y: 0.49},
42         size: { x: 0.01, y: 0.57}
43     ]
44 }
45
46
47 export default config;

```

Listing 1.2. sceneConfigSimple.js

The virtual robot accepts commands sent on a TCP connection on port 8999.

```

1 moveForward : '{ "type": "moveForward", "arg": 300 }'
2 moveBackward : '{ "type": "moveBackward", "arg": 300 }'
3
4 turnRight : '{ "type": "turnRight", "arg": 300 }'
5 turnLeft : '{ "type": "turnLeft", "arg": 300 }'
6
7 stop : '{ "type": "alarm" }'

```

Moreover, the virtual environment sends to the connected client (via TCP on the port 8999) the following data:

```

1 webpage-ready : '{ "type": "webpage-ready", "arg": {} }'
2 sonar-activated : '{'
3     "type": "sonar-activated",
4     "arg": { "sonarName": "sonarName", "distance": 1, "axis": "x" }
5 }
6 collision : '{'
7     "type": "collision",
8     "arg": { "objectName": "obstacle-1" }
9 }

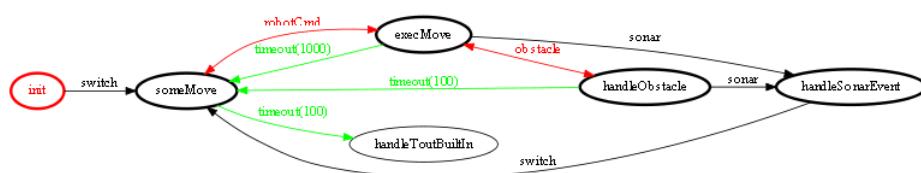
```

2.2.1 An example of usage in Java)

An example of usage at this basic level can be found in: [it.unibo.mbot.virtual.clientTcp.java](#).

2.2.2 An example of usage with *QActor*

An example of usage of the [wenv](#) from a *QActor* model represented by the following (generated) FSM diagram



can be written as follows:

```

1 System demoRobotInWenv
2 Dispatch obstacle : obstacle(TARGET)
3 Dispatch robotCmd : cmd(X)           //X=w|a|s|d|h
4
5 Event sonarDetect : obstacle(X)      //From wenv-robot sonar
6 Event sonar      : sonar(SONAR, TARGET, DISTANCE) //From wenv sonar
7
8 Context ctxRobotInWenv ip [ host="localhost" port=8098 ] -httpserver
9 EventHandler evobstacle for sonarDetect {

```

```

10     forwardEvent robotdemoinwenv -m obstacle
11 };
12
13 QActor robotdemoinwenv context ctxRobotInWenv {
14 Rules{
15   move(w). move(d). move(w). move(s). move(a).
16   move(d). move(w). move(a). move(s). move(h).
17 }
18 State init normal [
19   println("robotdemoinwenv STARTS");
20   javaRun it.unibo.robotVirtual.basicRobotExecutor.setUp("localhost")
21 ]
22 switchTo someMove
23
24 State someMove[
25   [ !? move(X) ] println( doingMove(X) );
26   [ ?? move(X) ] selfMsg robotCmd : cmd(X)
27 ]
28 transition stopAfter 100
29   whenMsg robotCmd -> execMove
30
31 State execMove resumeLastPlan[
32   onMsg robotCmd : cmd(MOVE) ->
33     javaRun it.unibo.robotVirtual.basicRobotExecutor.doMove( MOVE )
34 ]
35 transition whenTime 1000 -> someMove
36   whenMsg obstacle -> handleObstacle,
37   whenEvent sonar -> handleSonarEvent
38
39 State handleObstacle resumeLastPlan [ printCurrentMessage ]
40 transition whenTime 100 -> someMove
41   whenEvent sonar -> handleSonarEvent
42
43 State handleSonarEvent [ printCurrentEvent ]
44 switchTo someMove
45 }

```

Listing 1.3. demoRobotInWenv.qa

Note that:

- The moves to do are specified as a sequence of facts in tuProlog-like syntax.
- The Application designer has written the class `it.unibo.robotVirtual.basicRobotExecutor` that provides static methods to `setUp` the connection with the `wenv` and to send commands (`doMove`) to the virtual robot.
- The state `execMove` is ‘reactive’ to the event `sonar` and to the message `obstacle`.
- The message `obstacle` is generated by an event handler related to the event `sonarDetect`.

In fact, the `wenv` generates a `stream` of `sonar` events as long as the virtual robot is under the action area of a sonar, while the `obstacle` event is generated `only once` (when the virtual robot hits an obstacle).

2.2.3 The Application class basicRobotExecutor .

The Application class `basicRobotExecutor` exploits the basic `clientTcp` to implement the logical operations required at model level. The name of the class starts with a lower-case letter for a constraint imposed by the current implementation of the `QActor` software factory. Moreover, each operation provided by the Java class must have as its first argument a variable of type `QActor`.

```

1 package it.unibo.robotVirtual;
2 import it.unibo.qactors.akk.QActor;
3 import it.unibo.utils.clientTcpForVirtualRobot;

```

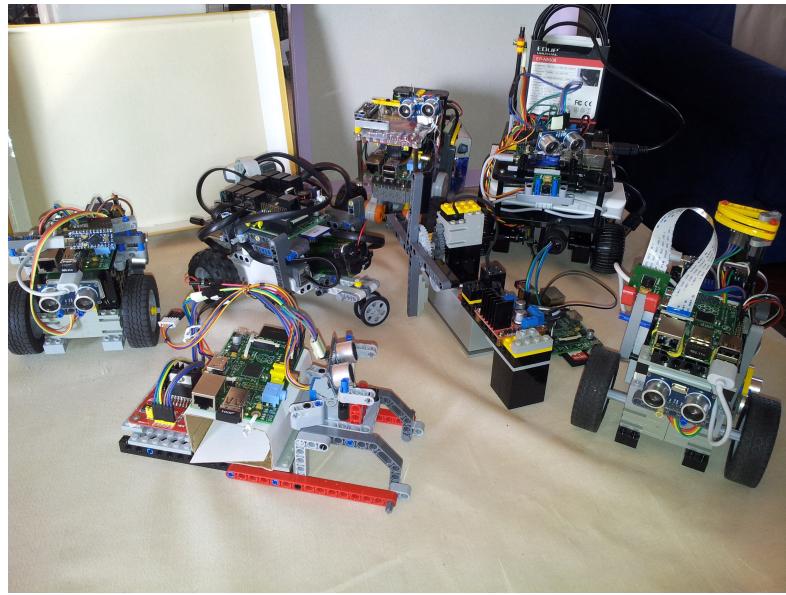
```

4
5  public class basicRobotExecutor {
6      public static void setUp(QActor qa, String hostNameStr) {
7          try {
8              qa.println("robotVirtual setUp " + hostNameStr );
9              clientTcpForVirtualRobot.initClientConn(qa, hostNameStr, "8999");
10         } catch (Exception e) {
11             e.printStackTrace();
12         }
13     }
14     public static void doMove( QActor qa, String cmd ) { //Args MUST be String
15     //    qa.println("robotVirtual doMove " + cmd );
16     switch( cmd ) {
17         case "h" : clientTcpForVirtualRobot.sendMsg(qa,"{'type': 'alarm', 'arg': 0 }");break;
18         case "w" : clientTcpForVirtualRobot.sendMsg(qa,"{'type': 'moveForward', 'arg': -1 }");break;
19         case "a" : clientTcpForVirtualRobot.sendMsg(qa,"{'type': 'turnLeft', 'arg': 400 }");break;
20         case "d" : clientTcpForVirtualRobot.sendMsg(qa,"{'type': 'turnRight', 'arg': 400 }");break;
21         case "s" : clientTcpForVirtualRobot.sendMsg(qa,"{'type': 'moveBackward', 'arg': -1 }");break;
22     }
23 }
24 }
```

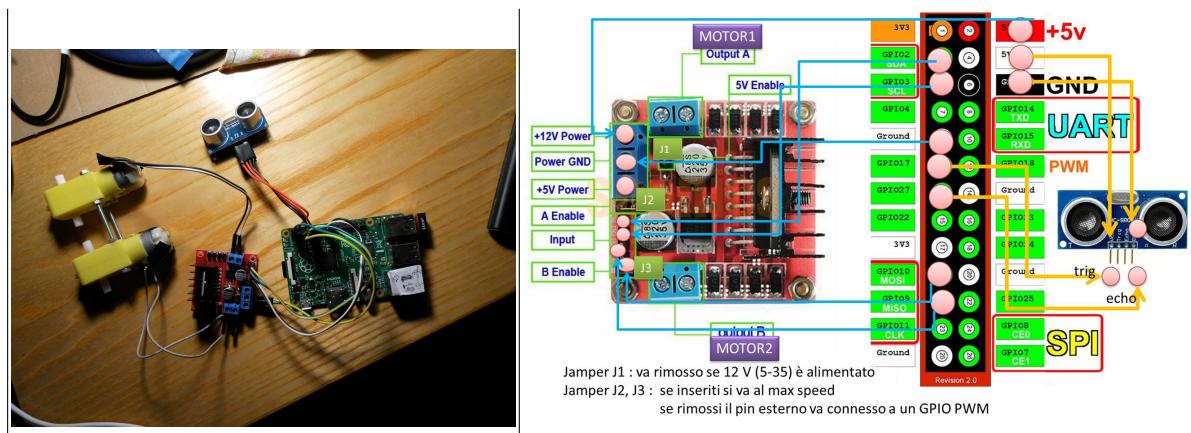
Listing 1.4. basicRobotExecutor.java

3 The in-house (nano) robot

From the physical point of view, a **BaseRobot** is a custom device built with low-costs components including sensors, actuators and processing units like *RaspberryPi* and *Arduino*. Examples are given in the following picture:



The hardware components of a **BaseRobot** and their configuration can be quite different from robot to robot. For example, a DDR-robot built around a RaspberryPi could have the basic hardware structure shown hereunder:



In this case, the correct configuration of the hardware connections can be tested by some simple bash code :

```

1 #!/bin/bash
2 #
3 # nanoMotorDriveA.sh
4 # test for nano0
5 # Key-point: we can manage a GPIO pin by using the GPIO library.
6 # On a PC, edit this file as UNIX
7 #
8
9 in1=2 #WPI 8 BCM 2 PHYSICAL 3
10 in2=3 #WPI 9 BCM 3 PHYSICAL 5
11 inwp1=8
12 inwp2=9
13
14 if [ -d /sys/class/gpio/gpio2 ]
15 then
16 echo "in1 gpio${in1} exist"
17 gpio export ${in1} out
18 else
19 echo "creating in1 gpio${in1}"
20 gpio export ${in1} out
21 fi
22
23 if [ -d /sys/class/gpio/gpio3 ]
24 then
25 echo "in2 gpio${in2} exist"
26 gpio export ${in2} out
27 else
28 echo "creating in2 gpio${in2}"
29 gpio export ${in2} out
30 fi
31
32 gpio readall
33
34 echo "run 1"
35 gpio write ${inwp1} 0
36 gpio write ${inwp2} 1
37 sleep 1.5
38
39 echo "run 2"
40 gpio write ${inwp1} 1
41 gpio write ${inwp2} 0
42 sleep 1.5
43
44 echo "stop"
45 gpio write ${inwp1} 0
46 gpio write ${inwp2} 0
47
48 gpio readall

```

Listing 1.5. nanoMotorDriveA.sh

3.1 Beyond the hardware level

However, some software layer is required to hide configuration differences as much as possible and to build a 'technology independent' layer, to be used by application designers.

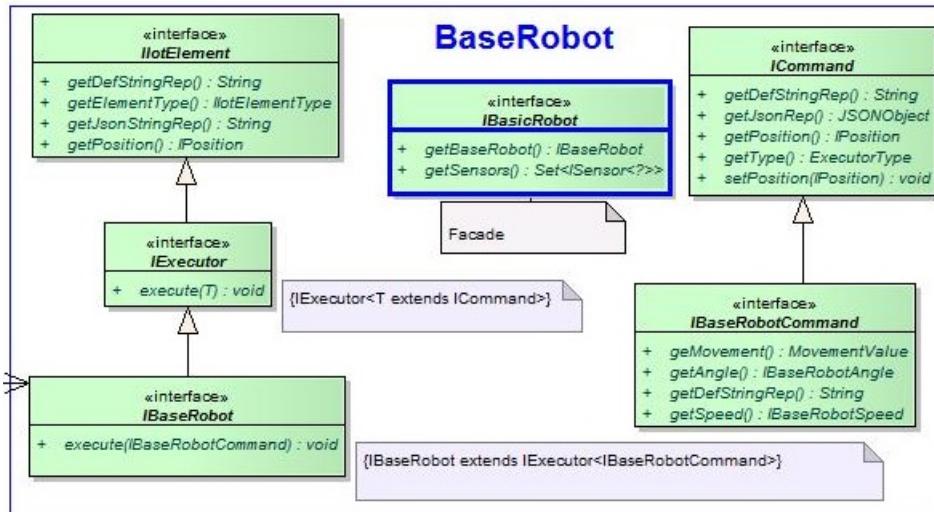
A software layer of this kind is provided by the library [labbaseRobotSam.jar](#). More specifically:

<i>it.unibo.lab.baseRobot</i>	The basic software for differential drive robots that are able to move and to acquire sensor data. Library: <i>labbaseRobotSam.jar</i> .
<i>it.unibo.robotRaspOnly.BasicRobotUsageNaive</i> in project <i>it.unibo.mbot2018</i>	Example of the usage of the API of a BaseRobot.

3.2 A model for the BaseRobot

The main goal of the *labbaseRobotSam.jar* library is to simplify the work of an application designer by exposing at application level a very simple model of a **BaseRobot**:

- As regards the **structure**, a **BaseRobot** can be viewed as a entity composed of two main parts:
 - An executor (with interface **IBaseRobot**), able to move the robot according to a a prefixed set of movement commands (**IBaseRobotCommand**)
 - A set of GOF -observable sensors (each with interface **ISensor**), each working as an active source of data.
- As regards the **interaction**, a **BaseRobot** can be viewed as a POJO that implements the interface **IBaseRobot**, while providing a (possibly empty) set of observable sensors;
- As regards the **behavior**, a **BaseRobot** is an object able to execute **IBaseRobotCommand** and able to update sensor observers defined by the application designer.



The interface **IBasicRobot** is introduced as a (GOF) *Facade* for the model.

```

1 package it.unibo.iot.baseRobot.hlmodel;
2 import java.util.Set;
3 import it.unibo.iot.executors.baseRobot.IBaseRobot;
4 import it.unibo.iot.sensors.ISensor;
5
6 public interface IBasicRobot {
7     public IBaseRobot getBaseRobot(); //selector
8     public Set<ISensor<?>> getSensors(); //selector
9 }
  
```

Listing 1.6. *IBasicRobot.java*

3.3 The BasicRobot class

The class **BasicRobot** provides a factory method to create a **BaseRobot** and to select its main components.

```

1 package it.unibo.iot.baseRobot.hlmodel;
2 import java.util.Set;
3 import it.unibo.iot.configurator.Configurator;
4 import it.unibo.iot.executors.baseRobot.IBaseRobot;
5 import it.unibo.iot.sensors.ISensor;
6
7 public class BasicRobot implements IBaseRobot{
8     private static IBaseRobot myself = null;
9     public static IBaseRobot getRobot(){
10         if( myself == null ) myself = new BasicRobot();
11         return myself;
12     }
13     public static IBaseRobot getTheBaseRobot(){
14         if( myself == null ) myself = new BasicRobot();
15         return myself.getBaseRobot();
16     }
17     //-----
18     private Configurator configurator;
19     private IBaseRobot robot ;
20     //Hidden constructor
21     protected BasicRobot() {
22         init();
23     }
24     protected void init(){
25         configurator = Configurator.getInstance();
26         robot      = configurator.getBaseRobot();
27     }
28     public IBaseRobot getBaseRobot(){
29         return robot;
30     }
31     public Set<ISensor<?>> getSensors(){
32         Set<ISensor<?>> sensors = configurator.getSensors();
33         return sensors;
34     }
35 }
```

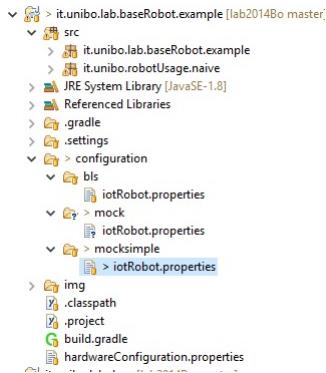
Listing 1.7. BasicRobot.java

This class shows that the work to set-up and access to the internal structure of a BaseRobot is delegated to a **Configurator**. This Configurator reads the specification of the robot structure written in a file named *iotRobot.properties* (see Subsection 3.5).

3.4 Using a BaseRobot

Let us introduce a robot with configuration named **mocksimple** (see Subsection 3.5), equipped with two motors and a distance sensor, all simulated.

3.4.1 The project workspace The application designer must organize its project workspace as shown in the following snapshot:



3.4.2 The code The application: *i)* first creates a sensor observer and adds it to all the sensors; *ii)* then it tells the robot to execute the commands (sent from an user console) it is able to understand.

```

1  private final IBaseRobotSpeed SPEED_LOW = new BaseRobotSpeed(BaseRobotSpeedValue.ROBOT_SPEED_LOW);
2  private final IBaseRobotSpeed SPEED_MEDIUM = new BaseRobotSpeed(BaseRobotSpeedValue.ROBOT_SPEED_MEDIUM);
3  private final IBaseRobotSpeed SPEED_HIGH = new BaseRobotSpeed(BaseRobotSpeedValue.ROBOT_SPEED_HIGH);
4  private IBaseRobot basicRobot;
5  private IBaseRobot robot ;
6
7  public BasicRobotUsageNaive() {
8      basicRobot = BasicRobot.getRobot();
9      robot     = basicRobot.getBaseRobot();
10 //    addObserverToSensors(basicRobot);
11 }
12 public void handleUserCommands() {
13     try {
14         while(true) {
15             int v   = System.in.read();
16             if( v == 13 || v == 10 ) continue;
17             char cmd = (char)v;
18             System.out.println( "INPUT= " + cmd + "(" + v +")" );
19             executeTheCommand( cmd );
20         }
21     } catch (IOException e) {
22         e.printStackTrace();
23     }
24 }
25 public void executeTheCommand( char cmd ) {
26     IBaseRobotCommand command = null;
27     switch( cmd ) {
28         case 'h' : command = new BaseRobotStop(SPEED_LOW );break;
29         case 'w' : command = new BaseRobotForward(SPEED_HIGH );break;
30         case 's' : command = new BaseRobotBackward(SPEED_HIGH );break;
31         case 'a' : command = new BaseRobotLeft(SPEED_MEDIUM );break;
32         case 'd' : command = new BaseRobotRight(SPEED_MEDIUM );break;
33         default: System.out.println( "Sorry, command not found");
34     }
35     if( command != null ) robot.execute(command);
36 }
37 protected void addObserverToSensors( ){
38     ISensorObserver observer = new SensorObserver();
39 //    for (ISensor<?> sensor : basicRobot.getSensors()) {
40 //        System.out.println( "doJob sensor= " + sensor.getDefStringRep() + " class= " + sensor.getClass().getName()
41 //    );
42 //        sensor.addObserver(observer);
43 //    }
44 //    addObserverToSensors(observer);
45 }
46 public void addObserverToSensors( ISensorObserver observer ){

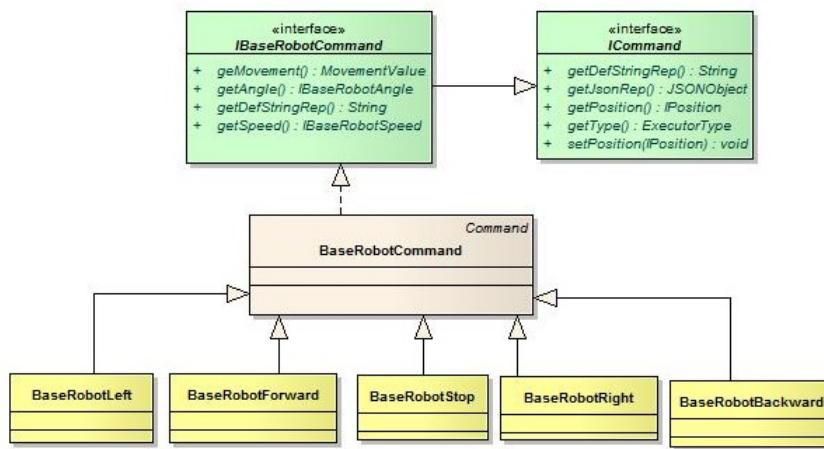
```

```

47     for (ISensor<?> sensor : basicRobot.getSensors()) {
48         System.out.println( "adding observer to sensor: " + sensor.getDefStringRep() );
49         sensor.addObserver(observer);
50     }
51 }
52
53 public static void main(String[] args) throws Exception{
54     new BasicRobotUsageNaive().handleUserCommands();
55 }
56

```

Listing 1.8. BasicRobotUsageNaive.java



3.5 The work of the Configurator

An object of class *it.unibo.iot.configurator.Configurator*:

1. first looks at the file `hardwareConfiguration.properties` to get the name of the robot (e.g. `mock`)
2. then, it consults the file `iotRobot.properties` into the directory `configuration/mock`

For each specification line, the `Configurator` calls (by using Java reflection) a factory method of the specific `DeviceConfigurator` class associated to the name of the robot.

For example, let us consider a Mock robot equipped with two motors and a distance sensor, all simulated: (file `configuration/mocksimple/iotRobot.properties`):

```

1 # =====
2 # ioRobot.properties for mocksimple robot
3 # Pay attention to the spaces
4 # =====
5 # ----- MOTORS -----
6 motor.left=mock
7 motor.left.private=false
8 #
9 motor.right=mock
10 motor.right.private=false
11 # ----- SENSORS -----
12 distance.front=mock
13 distance.front.private=false
14 # ----- COMPOSED COMPONENT -----

```

```

15 actuators.bottom=ddmotorbased
16 actuators.bottom.name=motors
17 actuators.bottom.comp=motor.left,motor.right
18 actuators.bottom.private=true
19 # ----- MAIN ROBOT -----
20 baserobot.bottom=differentialdrive
21 baserobot.bottom.name=mocksimple
22 baserobot.bottom.comp=actuators.bottom
23 baserobot.bottom.private=false

```

Listing 1.9. configuration/mocksimple/iotRobot.properties

The Configurator calls (using an object of class `IotComponentsFromConfiguration`):

- `getMotorDevice` of `it.unibo.iot.device.mock.DeviceConfigurator`
(for `motor.left=mock`)
- `getBaseRobotDevice` of `it.unibo.iot.device.differentialdrive.DeviceConfigurator`
(for `baserobot.bottom=differentialdrive`)
- `getDistanceSensorDevice` of `it.unibo.iot.device.mock.DeviceConfigurator`
(for `distance.front=mock`)
- `getMotorDevice` of `it.unibo.iot.device.mock.DeviceConfigurator`
(for `motor.right=mock`)
- `getActuatorsDevice` of `it.unibo.iot.device.ddmotorbased.DeviceConfigurator`
(for `actuators.bottom=ddmotorbased`)

3.6 From mocks to real robots

In order to use a physical robot rather than a Mock robot, the software designer must simply change the specification of the robot configuration; the application code is unaffected. For example, to use our standard '`nano`' robots, we have to include into the `configuration/nano` directory the following configuration file:

```

1 # =====
2 # ioRobot.properties for nano robot
3 # Pay attention to the spaces
4 # =====
5 # ----- MOTORS -----
6 motor.left=gpio.motor
7 motor.left.pin.cw=8
8 motor.left.pin.ccw=9
9 motor.left.private=false
10 #
11 motor.right=gpio.motor
12 motor.right.pin.cw=12
13 motor.right.pin.ccw=13
14 motor.right.private=false
15 # ----- SENSORS -----
16 distance.front_top=hcsr04
17 distance.front_top.trig=0
18 distance.front_top.echo=2
19 distance.front_top.private=false
20 # ----- COMPOSED COMPONENT -----
21 actuators.bottom=ddmotorbased
22 actuators.bottom.name=motors
23 actuators.bottom.comp=motor.left,motor.right
24 actuators.bottom.private=true
25 # ----- MAIN ROBOT -----
26 baserobot.bottom=differentialdrive
27 baserobot.bottom.name=nano
28 baserobot.bottom.comp=actuators.bottom
29 baserobot.bottom.private=false

```

Listing 1.10. configuration/nano/iotRobot.properties

3.7 Sensors and Sensor Data

The current version of the **BaseRobot** system implements the following sensors:

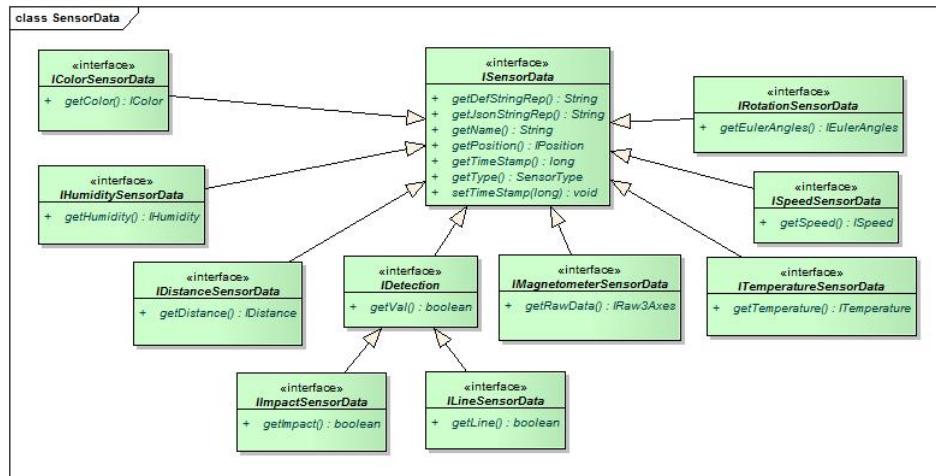
RobotSensorType: Line | Distance | Impact | Color | Magnetometer

Each sensor:

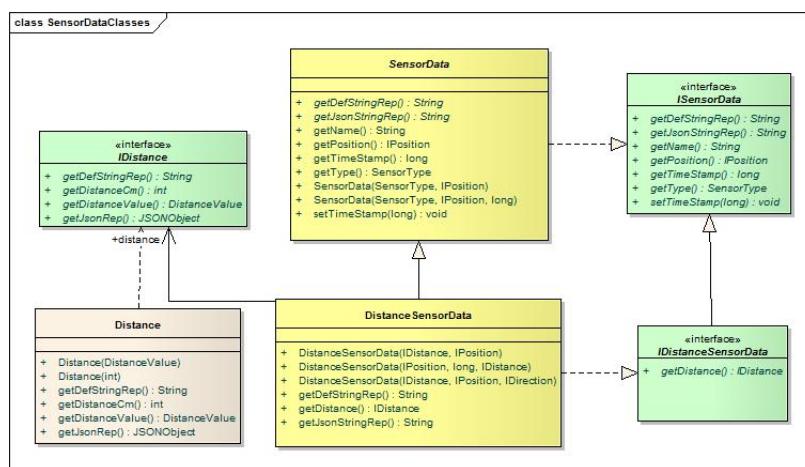
- is associated to a position that can assume one of the following values:

```
DONT CARE |
FRONT | RIGHT | LEFT | BACK | TOP | BOTTOM |
FRONT_RIGHT | FRONT_LEFT | BACK_RIGHT | BACK_LEFT |
TOP_RIGHT | TOP_LEFT | BOTTOM_RIGHT | BOTTOM_LEFT |
FRONT_TOP | BACK_TOP | FRONT_TOP_LEFT | FRONT_TOP_RIGHT |
FRONT_RIGHT_TOP | FRONT_LEFT_TOP | BACK_RIGHT_TOP | BACK_LEFT_TOP
```

- is a source of data, each associated to a specific class and interface:



Each class related to sensor data inherits from a base class **SensorData**. For example:



The class `SensorData` provides operations to represent data as strings in two main formats: *i*) in Prolog syntax and *ii*) in Json syntax. For example:

3.7.1 Sensor data representation in Prolog (high level)

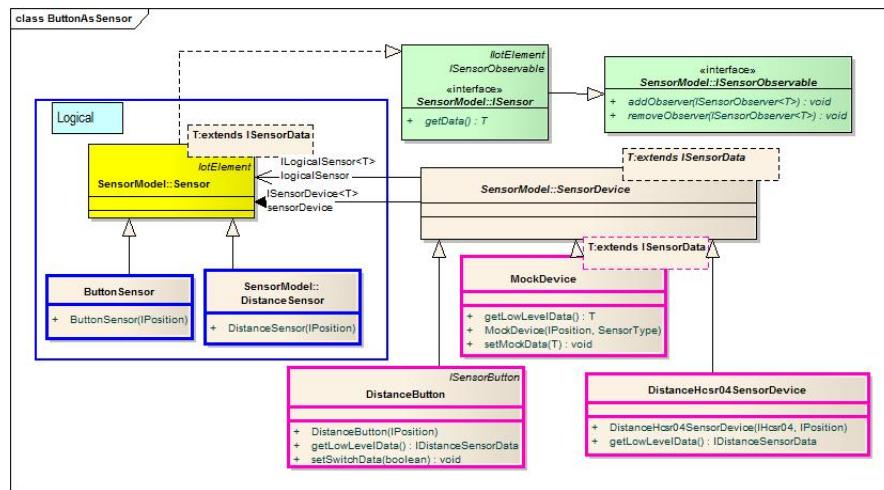
COLOR	<code>color(255 255 255, front)</code>
DISTANCE	<code>distance(43,forward, front)</code>
IMPACT	<code>impact(touch/loss, front)</code>
LINE	<code>line(lineLeft/lineDetected, bottom)</code>
MAGNETOMETER	<code>magnetometer(x(50),y(100),z(0), front)</code>

3.7.2 Sensor data representation in Json (low level)

COLOR	<code>"p":"f","t":"c","d":"color":"r":255,"b":255,"g":255,"tm":148...</code>
DISTANCE	<code>"p":"f","t":"d","d":"cm":43,"tm":14...</code>
IMPACT	<code>"p":"f","t":"i","d":"detection":"touch","tm":14...</code>
LINE	<code>"p":"b","t":"l","d":"detection":"lineDetected","tm":14...</code>
MAGNETOMETER	<code>"p":"f","t":"m","d":"raw3axes":"x":50,"y":100,"z":0,"tm":14...</code>

3.8 Sensor model

The sensor subsystem of the `BaseRobot` is based on the class `Sensor` that represents a sensor from the logical point of view. Each sensor is associated to a class that inherits form `Sensor`.



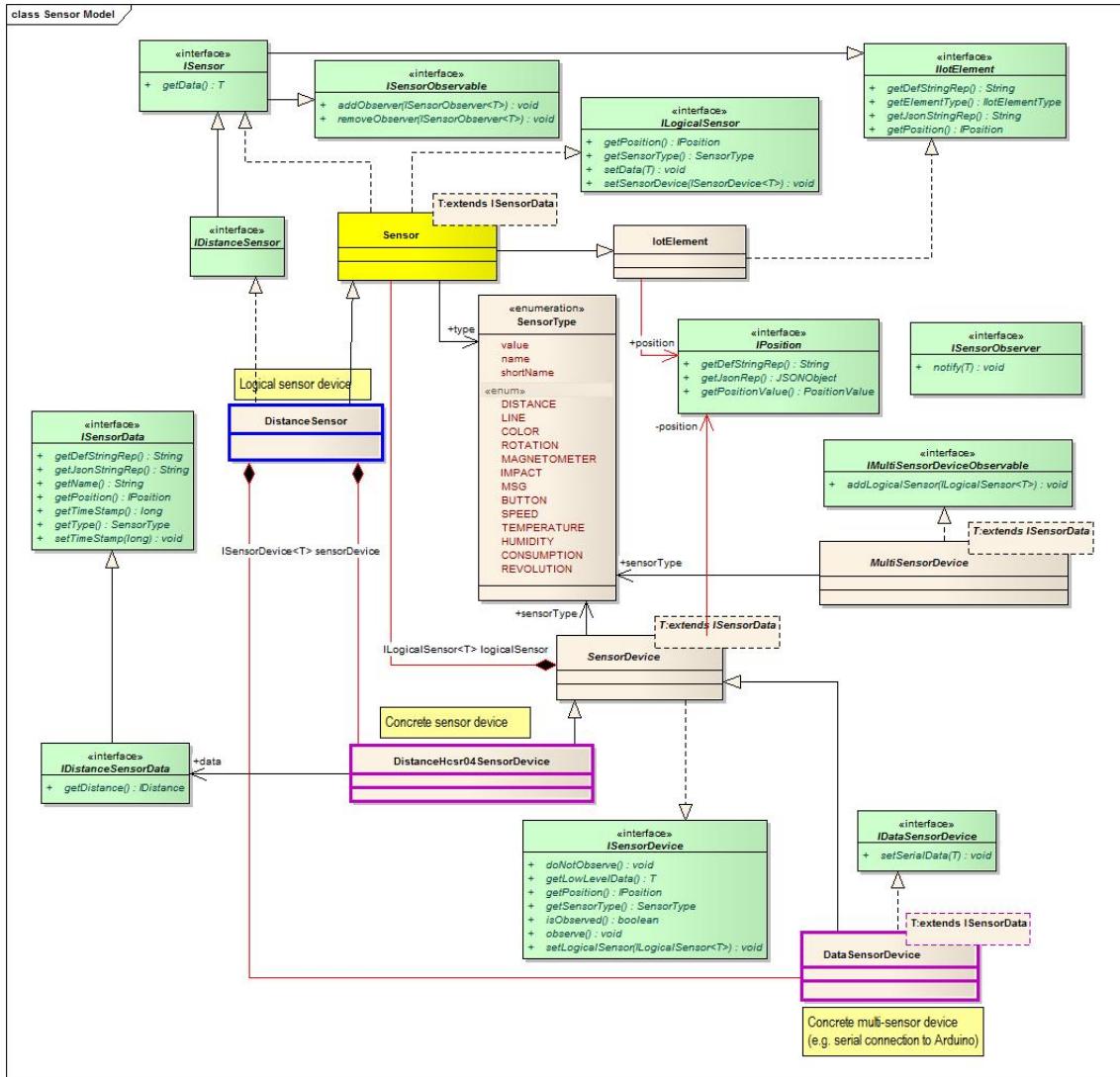
The model reported in the picture above shows that:

- A `DistanceSensor` is a logical `Sensor` associated (by the *Configurator*) to a concrete device (e.g. `DistanceHCSR04SensorDevice`). The same is true for a `ButtonSensor` (impact).
- A `DistanceHCSR04SensorDevice` is a concrete `SensorDevice` that updates its logical sensor when it produces a value. The same is true for a `DistanceButton` (impact). The diagram shows also a `MockDevice` that can be used to simulate the behavior of the supported sensors.

- Any **Sensor** is an observable entity that, when updated from its concrete device, updates the registered application observers.

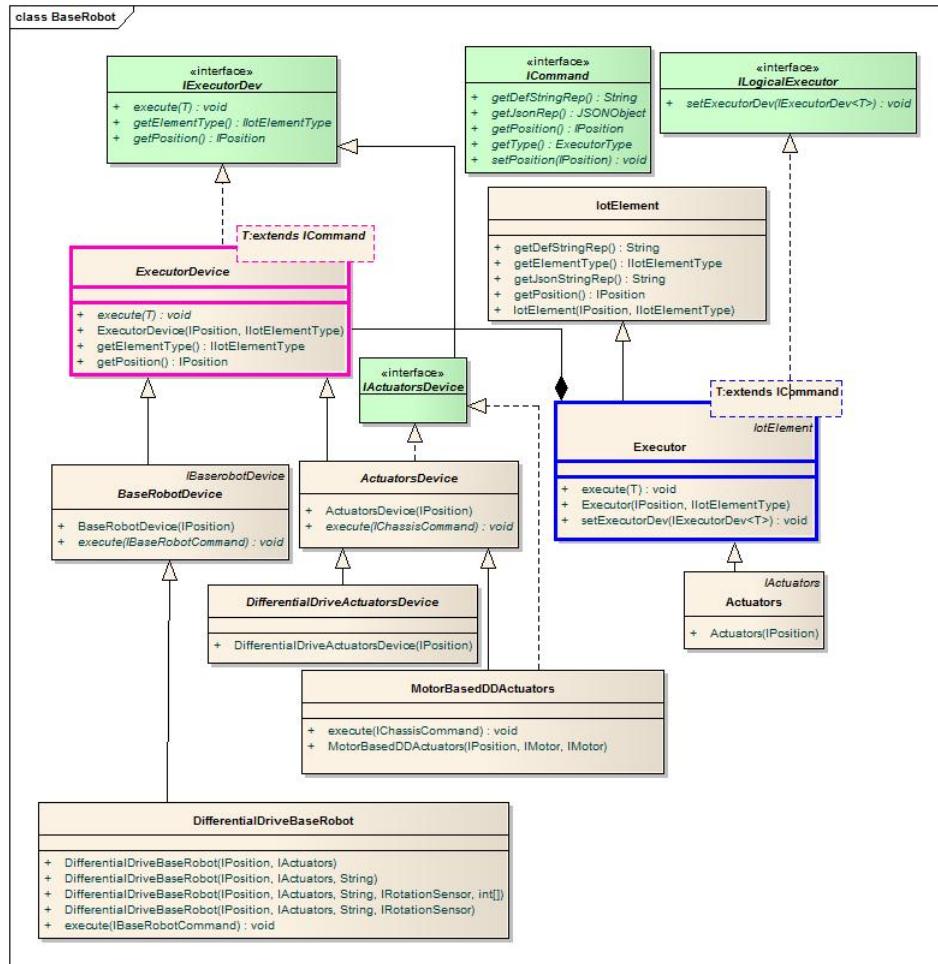
In this way, according to the GOF pattern *Bridge*, the **Sensor** abstraction hierarchy is decoupled from the hierarchy of **SensorDevice** implementation.

A more detailed picture is reported hereunder:



3.9 Actuators and Executors

The GOF Bridge pattern has been adopted also to model the 'motors' and the more general concept of 'executor'.



However this part of the `BaseRobot` can be ignored by the application designer and it is no more discussed here.

3.10 An example of usage with *QActor*

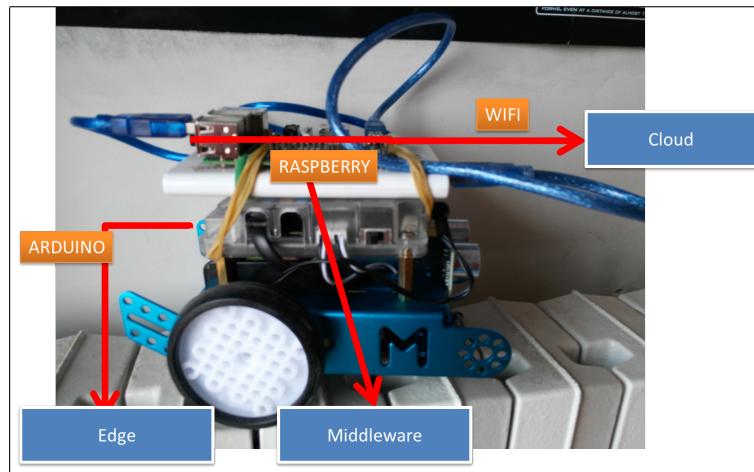
```
1 System nanoControl
2
3 Event sonarEvent : distance(D, DIR, POS) //from SonarInCObserver
4 Context ctxNanoControl ip [ host="localhost" port=8098 ] -httpserver
5
6 QActor robotnano context ctxNanoControl {
7     State init normal [
8         println("robotdemo STARTS");
9         javaRun it.unibo.robotOnRaspOnly.basicRobotExecutor.setUp()
10    ]
11    switchTo someMove
12
13    State someMove[
14        delay 1000;
15        javaRun it.unibo.robotOnRaspOnly.basicRobotExecutor.doMove( "w" );
16        delay 1000;
```

```
17     javaRun it.unibo.robotOnRaspOnly.basicRobotExecutor.doMove( "s");
18     delay 1000;
19     javaRun it.unibo.robotOnRaspOnly.basicRobotExecutor.doMove( "h")
20 ]
21 transition stopAfter 600000
22   whenEvent sonarEvent -> handleSonarEvent
23   finally repeatPlan
24
25 State handleSonarEvent resumeLastPlan [
26   printCurrentEvent
27 ]
28
29 }
```

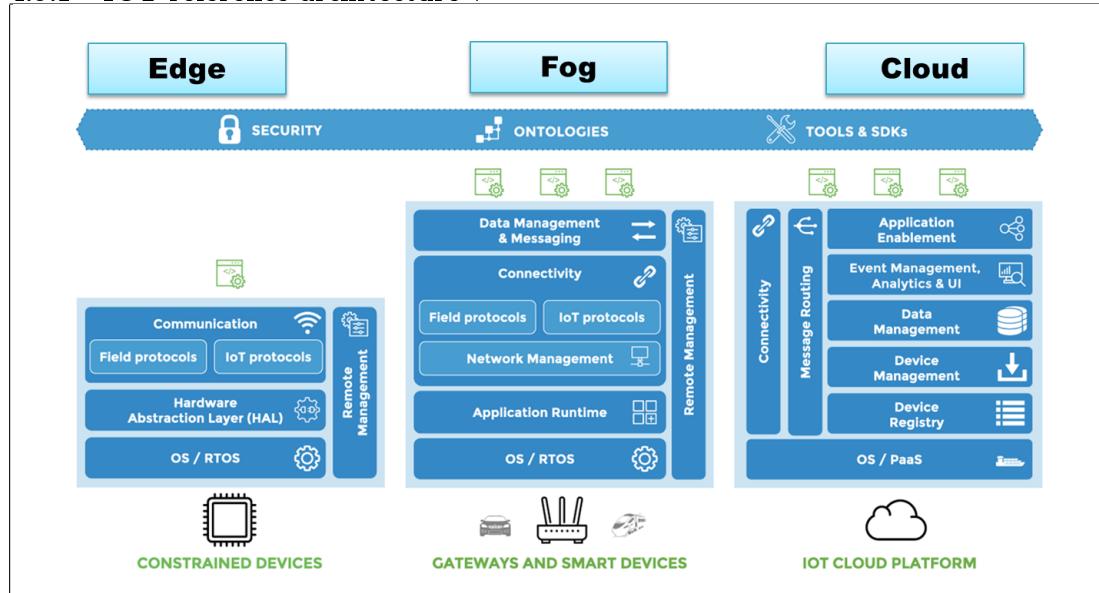
Listing 1.11. nanoControl.qa

4 The mbot robot

The mbot architecture can be viewed as a simple example of a IoT architecture in which the **edge** part is implemented on Arduino, the **middleware** part is implemented on a RaspberryPi and the **cloud** part is implemented on a conventional PC.



4.0.1 IoT reference architecture



Our real robot is made of a RaspberryPi connected via the serial USB cable with the Arduino included in the Makeblock Mbot device:

Arduino handles physical devices such as motors and sensors, while RaspberryPi provides support for interaction with a remote node.

The software for the real robot is made of two main parts:

-
1. a low-level hardware-related part running on Arduino (see Subsection 4.1);
 2. a high-level robot-control part running on the RaspberryPi (see Subsection 4.2).

4.1 The low-level code on Arduino

The code running on Arduino has been built as reported in Subsection 4.1.1. It is composed of the following parts:

1. A set of declarations:

```

1 #include <Arduino.h>
2 #include <Wire.h>
3 #include <SoftwareSerial.h>
4 #include <MeMCore.h>
5
6 MeDCMotor motor_9(9);
7 MeDCMotor motor_10(10);
8 double angle_rad = PI/180.0;
9 double angle_deg = 180.0/PI;
10 void lookAtSonar();
11 void move(int direction, int speed);
12 double sonar;
13 int input;
14 int count;
15 MeUltrasonicSensor ultrasonic_3(3);
16 MeRGBLed rgbled_7(7, 7==7?2:4);
17 void remoteCmdExecutor();
```

Listing 1.12. uniboControl.ino: the main loop

2. A function that works as an interpreter of commands to move the robot:

```

1 void remoteCmdExecutor()
2 {
3     if((Serial.available() > (0 ))){
4         input = Serial.read();
5         //Serial.println(input);
6         switch( input ){
7             case 119 : move(1,150); break; //w
8             case 115 : move(2,150); break; //s
9             case 97 : move(3,150); break; //a
10            case 100 : move(4,150); break; //d
11            case 104 : move(1,0); stopFollow = true; break; //h
12            case 102 : move(1,0); stopFollow = false; break; //f
13            default : move(1,0); stopFollow = true;
14        }
15    }
16 }
17 /*
18 * -----
19 * Moving
20 * -----
21 */
22 void move(int direction, int speed)
23 {
24     int leftSpeed = 0;
25     int rightSpeed = 0;
26     if(direction == 1){ //forward
27         leftSpeed = speed;
28         rightSpeed = speed;
29     }else if(direction == 2){ //backward
30         leftSpeed = -speed;
31         rightSpeed = -speed;
32     }else if(direction == 3){ //left
33         leftSpeed = -speed;
34         rightSpeed = speed;
```

```

35     }else if(direction == 4){ //right
36         leftSpeed = speed;
37         rightSpeed = -speed;
38     }
39     motor_9.run((9)==M1?-leftSpeed):(leftSpeed));
40     motor_10.run((10)==M1?-(rightSpeed):(rightSpeed));
41 }
```

Listing 1.13. uniboControl.ino: the command interpreter

3. A function that works as an emitter of sonar data (a value of type `double`) that implements also a prefixed obstacle-avoidance policy (retrogress the robot when it is very near to an obstacle):

```

1 void lookAtSonar()
2 {
3     sonar = ultrasonic_3.distanceCm();
4     //emit sonar data but with a reduced frequency
5     if( count++ > 50 ){ Serial.println(sonar); count = 0; }
6     if((sonar) < (10)){ //very near
7         if(((input)==(119)) ){
8             move(1,0);
9             rgbled_7.setColor(0,60,0,0);
10            rgbled_7.show();
11            //Serial.println("OBSTACLE FROM ARDUINO");
12            _delay(0.3);
13            move(2,100);
14            _delay(1);
15            move(2,0);
16        }
17    }
18 }
```

Listing 1.14. uniboControl.ino: sonar data

The `delay` function is a loop to lose time:

```

1 void _loop(){
2 }
3
4 void _delay(float seconds){
5     long endTime = millis() + seconds * 1000;
6     while(millis() < endTime)_loop();
7 }
```

Listing 1.15. uniboControl.ino: delay

4. A function that implements a line-follower strategy:

```

1 double stopFollow = true;
2 double sonarVal;
3 void lineFollow();
4 MeLineFollower linefollower_2(2);
5
6 void sonarDetect()
7 {
8     sonarVal = ultrasonic_3.distanceCm();
9     Serial.println(sonarVal);
10    if((sonarVal) < (10)){
11        move(1,0);
12        stopFollow = true;
13        //Serial.println("stopFollow line follow");
14    }
15 }
16 void lineFollow()
17 {
18     if( stopFollow == true ) return;
19     if(((linefollower_2.readSensors())==(0))){
20         move(1,200);
21     }
22 }
```

```

21     }
22     if(((linefollower_2.readSensors())==(1))){
23         motor_9.run((9)==M1?-(0):(0));
24         motor_10.run((10)==M1?-(150):(150));
25     }
26     if(((linefollower_2.readSensors())==(2))){
27         motor_9.run((9)==M1?-(150):(150));
28         motor_10.run((10)==M1?-(0):(0));
29     }
30     if(((linefollower_2.readSensors())==(3))){
31         move(2,100);
32     }
33     //sonarDetect();
34 }
```

Listing 1.16. uniboControl.ino: the line-follower

5. The set-up and the main loop:

```

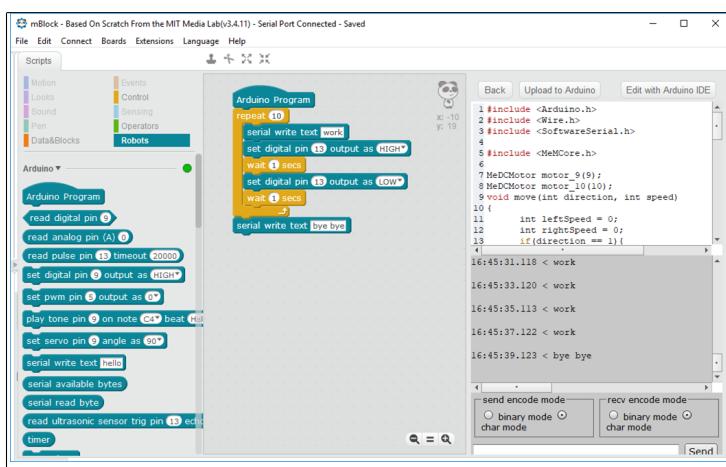
1 void setup(){
2     Serial.begin(115200);
3     //Serial.println("start");
4 }
5
6 void loop(){
7     rgbled_7.setRGB(0,0,60,0);
8     rgbled_7.show();
9     remoteCmdExecutor();
10    lookAtSonar();
11    lineFollow();
12    _loop();
13 }
```

Listing 1.17. uniboControl.ino: the main loop

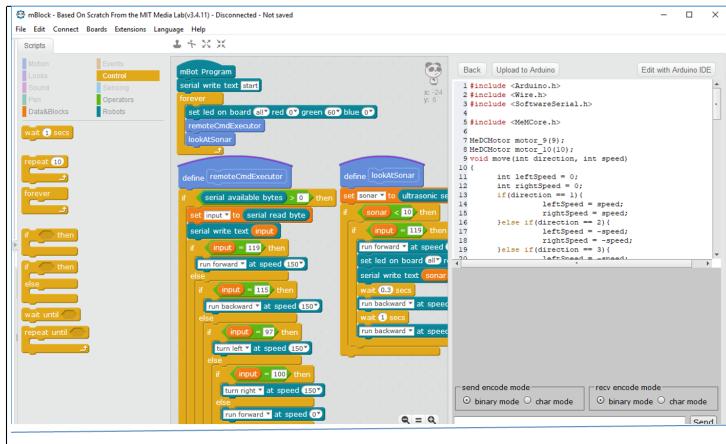
4.1.1 Mblock software .

This code on the mbot can be built with the help of the mblock IDE. mBlock is a graphical programming software which is designed based on Scratch 2.0 and compatible with Arduino UNO, mega 2560, leonardo, etc.

The following picture shows a [mblock](#) project for the ArduinoUno board that blinks the built-in led:



The following picture shows a **mBlock** project for the mCore, a control board specially designed for mBot. Based on Arduino Uno, mCore integrates various onboard sensors, such as buzzer, light sensor, RGB LED, etc.



Of course, this is quite limited way to build low-level code for the mbot robot. In order to work directly within a conventional Arduino IDE, we can operate as follows:

- Download the Makeblock-Official library from Github at <https://github.com/Makeblock-official/Makeblock-Libraries>
- Unzip the file into a temporary folder.
- Change to the temporary folder and zip the "makeblock" directory content.
- In the Arduino IDE, click Sketch->Include Library->Add .ZIP library and select the **makeblock.zip** file you've created.
- In the Arduino IDE, click file/Esempi/MakeBlockDrive/Firmware_For_mBlock/mbot_factory_firmware to download the firmware
- In the Arduino IDE, open C:/Didattica/mBot/uniboControl/uniboControl.ino

The code of Subsection 4.1 has been built in this way.

4.2 An example of usage with *QActor*

todo

4.3 mbotConnArduinoObj

The user-defined class `mbotConnArduinoObj` is the 'adapter' between high-level and the low-level worlds. It is written in Java and is based on another software layer (the class `SerialPortConnSupport`) that provides operations to send commands to the low-level executor on Arduino.

4.3.1 Init .

The first part of `mbotConnArduinoObj` provides operations to initialize the serial connection between a PC or a Raspberry and Arduino:

```
1 package it.unibo.mbot;
2 import it.unibo.mbot.serial.JSSCSerialComm;
3 import it.unibo.mbot.serial.SerialPortConnSupport;
4
5 public class MbotConnArduinoObj {
6     private SerialPortConnSupport conn = null;
7     private JSSCSerialComm serialConn;
8     private double dataSonar = 0;
9     private String curDataFromArduino;
10    private ISensorObserverFromArduino observer;
11
12    public void initRasp( String port ) { //"/dev/ttyUSB0"
13        init( port );
14    }
15    public void initPc(String port) { //"COM6"
16        init( port );
17    }
18    private void init(String port) {
19        try {
20            System.out.println("mbotConnArduinoObj starts");
21            serialConn = new JSSCSerialComm(null);
22            conn = serialConn.connect(port); //returns a SerialPortConnSupport
23            if( conn == null ) return;
24            curDataFromArduino = conn.receiveALine();
25            System.out.println("mbotConnArduinoObj received:" + dataSonar);
26            getDataFromArduino();
27        }catch( Exception e ) {
28            System.out.println("mbotConnArduinoObj ERROR" + e.getMessage());
29        }
30    }
}
```

Listing 1.18. `MbotConnArduinoObj.java: init`

4.3.2 Output .

Afterwards, the class defines operations to map high-level commands into low-level commands to be sent to Arduino:

```
1 public void executeTheCommand( char cmd ) {
2     System.out.println("mbotConnArduinoObj executeTheCommand " + cmd + " conn=" + conn);
3     switch(cmd) {
4         case 'h' : mbotStop(); break;
5         case 'w' : mbotForward(); break;
6         case 's' : mbotBackward(); break;
7         case 'a' : mbotLeft(); break;
8         case 'd' : mbotRight(); break;
9         case 'f' : mbotLinefollow(); break;
}
```

```

10     }
11 }
12 public void mbotForward() {
13     try { if( conn != null ) conn.sendCmd("w"); } catch (Exception e) {e.printStackTrace();}
14 }
15 public void mbotBackward() {
16     try { if( conn != null ) conn.sendCmd("s"); } catch (Exception e) {e.printStackTrace();}
17 }
18 public void mbotLeft() {
19     try { if( conn != null ) conn.sendCmd("a"); } catch (Exception e) {e.printStackTrace();}
20 }
21 public void mbotRight() {
22     try { if( conn != null ) conn.sendCmd("d"); } catch (Exception e) {e.printStackTrace();}
23 }
24 public void mbotStop() {
25     try { if( conn != null ) conn.sendCmd("h"); } catch (Exception e) {e.printStackTrace();}
26 }
27 public void mbotLinefollow() {
28     try { if( conn != null ) conn.sendCmd("f"); } catch (Exception e) {e.printStackTrace();}
29 }

```

Listing 1.19. MbotConnArduinoObj.java: output

4.3.3 Observer .

Then, the class defines an operation to add an observer to the data sent on the serial line by Arduino.

```

1 public void addObserverToSensors( ISensorObserverFromArduino observer ){
2     this.observer = observer;
3 }

```

Listing 1.20. MbotConnArduinoObj.java: addObserver

4.3.4 Input .

Finally, there is an operation that waits for sonar values coming from Arduino (see Subsection 4.1, point 3) and maps these data into an event **realSonar:sonar(DISTANCE)**:

```

1 private void getDataFromArduino() {
2     new Thread() {
3         public void run() {
4             try {
5                 System.out.println("mbotConnArduinoObj getDataFromArduino STARTED" );
6                 while(true) {
7                     try {
8                         curDataFromArduino = conn.receiveALine();
9                         System.out.println("mbotConnArduinoObj received:" + curDataFromArduino );
10                        double v = Double.parseDouble(curDataFromArduino);
11                        //handle too fast change
12                        double delta = Math.abs( v - dataSonar );
13                        if( delta < 7 && delta > 0.5 ) {
14                            dataSonar = v;
15                            System.out.println("mbotConnArduinoObj sonar:" + dataSonar);
16                            observer.notify(""+dataSonar);
17                            QActorUtils.raiseEvent(curActor, curActor.getName(), "realSonar",
18                                "sonar( DISTANCE )".replace("DISTANCE", ""+dataSonar));
19                        }
20                     } catch (Exception e) {
21                         System.out.println("mbotConnArduinoObj ERROR:" + e.getMessage());
22                     }
23                 }
24             } catch (Exception e) {
25                 e.printStackTrace();
26             }
27         }

```

```
28     }.start();
29 }
```

Listing 1.21. MbotConnArduinoObj.java: input

4.4 The adapter basicRobotExecutor

The Application class **basicRobotExecutor** exploits the basic **MbotConnArduinoObj** to implement the logical operations required at model level:

The name of the class starts with a lower-case letter for a constraint imposed by the current implementation of the *QActor* software factory. Moreover, each operation provided by the Java class must have as its first argument a variable of type *QActor*.

```
1 package it.unibo.robotMBot;
2 import it.unibo.mbot.MbotConnArduinoObj;
3 import it.unibo.qactors.akka.QActor;
4
5 public class basicRobotExecutor {
6     private static MbotConnArduinoObj robotSupport ; //singleton
7
8     public static void setUp(QActor qa, String port) {
9         if( robotSupport == null ) {
10             robotSupport = new MbotConnArduinoObj();
11             robotSupport.initPc(port);
12             robotSupport.addObserverToSensors( new SensorObserverFromArduino(qa) );
13         }
14     }
15     public static void doMove( QActor qa, String cmd ) { //Args MUST be String
16         robotSupport.executeTheCommand(cmd.charAt(0));
17     }
18 }
```

Listing 1.22. basicRobotExecutor.java

5 A robot-command executor

In this section we introduce a system that implements the following set of features:

1. The system handles commands sent to a robot by an human user via a Web-GUI interface.
2. The system can be configured to work with:
 - the virtual robot running in the `wenv` of Subsection 2.2.1;
 - the physical robot of Section 3;
 - the physical robot of Section 4;
3. The system is able to react to events emitted by a `sonar` (both virtual and real).

The model shows that:

- The configuration is expressed as a fact (written in the `actorKb` of the robot) of the form:

```
1 robotType( ROBOTYPE, setuparg(ARGs) ).
```

- The robot modelled by the actor `robotdemo` initially performs some move, to show that it is working.
- The `robotdemo` waits for an message command sent by another actor (`cmdrobotconverter`) that maps a user command sent from the built-in `QActor` Web-GUI into a `dispatch robotCmd : cmd(X)`.
- The `robotdemo` assumes that the implementation of a move command (see state `doJob` and Sub-section 5.1) is performed in *asynchronous* way.

```
1 System demoRobot
2
3 Dispatch robotCmd : cmd(X)           //X=w|a|s|d|h
4 Event sonarEvent : distance(D, DIR, POS) //from SensorObserver
5
6 Event usercmd   : usercmd(X)    //from robot GUI; X=robotgui(CMD) CMD=s(low)
7 Event inputcmd  : usercmd(X)    //from input GUI; X=executeInput( do(G,M) )
8 Event alarm     : alarm(X)      //from red button
9 Event obstacle  : obstacle(X)   //from red button and from SensorObserver
10 Event sonarDetect : sonarDetect(X) //From (virtual robot) sonar
11 Event sonar     : sonar(SONAR, TARGET, DISTANCE) //From (virtual) sonar
12
13 //Context ctxConsole ip [ host="localhost" port=8078 ] -g cyan -httpserver
14 Context ctxRobot   ip [ host="localhost" port=8098 ] -httpsserver
15
16 QActor cmdrobotconverter context ctxRobot{
17   State init normal [ ]
18   transition stopAfter 600000
19   whenEvent usercmd -> handleUserCmd
20   finally repeatPlan
21
22   State handleUserCmd resumeLastPlan[//MAPPING AFTER EXPERIMENTATION
23     onEvent usercmd : usercmd( robotgui(h(X)) ) ->
24       forward robotdemo -m robotCmd : cmd("h");
25     onEvent usercmd : usercmd( robotgui(w(X)) ) ->
26       forward robotdemo -m robotCmd : cmd("w");
27     onEvent usercmd : usercmd( robotgui(a(X)) ) ->
28       forward robotdemo -m robotCmd : cmd("a");
29     onEvent usercmd : usercmd( robotgui(d(X)) ) ->
30       forward robotdemo -m robotCmd : cmd("d");
31     onEvent usercmd : usercmd( robotgui(s(X)) ) ->
32       forward robotdemo -m robotCmd : cmd("s")
33   ]
34 }
35
36 QActor robotdemo context ctxRobot {
37   Rules{
```

```

38     //robotType( "robotVirtual", setuparg("localhost") ) .
39     robotType( "robotRealMbot", setuparg("COM6") ) .
40     //robotType( "robotRealRaspOnly", setuparg("") ) .
41 }
42 State init normal [
43     println("robotdemo STARTS");
44     [ !? robotType(T,setuparg(A)) ]
45         javaRun it.unibo.utils.allRobots.setUp(T,A)
46     ]
47 switchTo someMove
48
49 State someMove[
50     delay 1000;
51     javaRun it.unibo.utils.allRobots.doMove( "w" );
52     delay 1000;
53     javaRun it.unibo.utils.allRobots.doMove( "s" );
54     delay 1000;
55     javaRun it.unibo.utils.allRobots.doMove( "h" )
56 ]
57 switchTo doJob
58
59 State doJob [
60     println("waiting for an user command ...")
61 ]
62 transition stopAfter 600000
63     whenMsg robotCmd:cmd(M) do javaRun it.unibo.utils.allRobots.doMove(M),
64     whenEvent sonarEvent -> handleSonarEvent,
65     whenEvent sonar -> handleSonarEvent
66     finally repeatPlan
67
68 State handleSonarEvent resumeLastPlan [
69     printCurrentEvent
70 ]
71
72 }

```

Listing 1.23. demoRobot.qa

5.1 The adapter

In order to achieve its goals, the system-model exploits an utility class written by the Application designer:

```

1 package it.unibo.utils;
2
3 import alice.tuprolog.SolveInfo;
4 import it.unibo.qactors.akka.QActor;
5
6 public class allRobots {
7
8     public static void setUp( QActor qa, String robotType, String args ) {
9         try {
10             qa.println("allRobots " + robotType + " setUp args=" + args);
11             switch( robotType ) {
12                 case "robotRealMbot" : it.unibo.robotMBot.basicRobotExecutor.setUp(qa,args);break;
13                 case "robotRealRaspOnly" : it.unibo.robotOnRaspOnly.basicRobotExecutor.setUp(qa);break;
14                 case "robotVirtual" : it.unibo.robotVirtual.basicRobotExecutor.setUp(qa,args);break;
15             }
16         }catch(Exception e) {
17             e.printStackTrace();
18         }
19     }
20
21     public static void doMove( QActor qa, String cmd ) { //Args MUST be String
22         try {
23             SolveInfo sol = qa.solveGoal("robotType( T, _ )");
24             String robotType = sol.getVarValue("T").toString();

```

```

25     qa.println("allRobots " + robotType + " doMove cmd=" + cmd);
26     switch( robotType ){
27         case "robotRealMBot" : it.unibo.robotMBot.basicRobotExecutor.doMove(qa, cmd);break;
28         case "robotRealRaspOnly" : it.unibo.robotOnRaspOnly.basicRobotExecutor.doMove(qa, cmd);break;
29         case "robotVirtual" : it.unibo.robotVirtual.basicRobotExecutor.doMove(qa,cmd);break;
30         default: qa.println("Sorry, robot type unknown");
31     }
32     }catch(Exception e) {
33         e.printStackTrace();
34     }
35 }
36 }
37 }
```

Listing 1.24. allRobots.java

5.2 Activating the wenv

The assumption in the model is that the virtual environment **wenv** has been already activated. However, the **wenv** can be activated by the model itself. For example:

```

1 QActor startscene context ctxRobot {
2     Plan init normal[
3         println("startscene starts....");
4         nodeOp "C:/Didattica2018Work/iss2018Lab/it.unibo.mbot2018/Soffritti/server/src/main.js 8999" -o
5     ]
6 }
```