Introduction to UniboDISI DDR-Robots (2019)

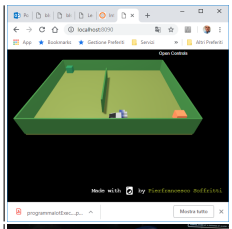
Antonio Natali

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

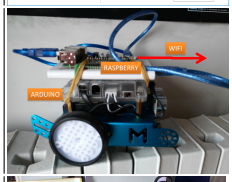
Table of Contents

1 Virtual and Real robots

Our company (`Unibo`) has developed:

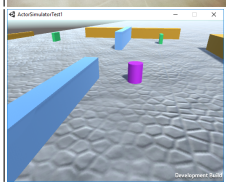


A virtual environment (named `W-Env`) built in `JavaScript` (see Subsection 2.1), that includes a virtual robot that accepts commands sent on a `TCP` connection on port `8999`.



A physical robot based upon a `mBot robot` (see Section 3).

A physical `ddr` robot built in-house (see Section 4), modelled as an observable `POJO`.



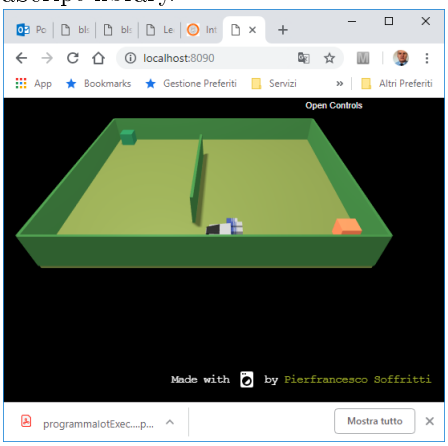
A virtual environment (named `U-Env`) based on the `Unity` system (not described here), that includes a virtual robot that accepts commands sent on a `TCP` connection on port `8090`.

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 JavaScript (`wenv`)

This virtual environment (called `wenv`) has been built by Pierfrancesco Soffritti[1] using the https://threejs.org/ JavaScript library.



The scene is described in a configuration file. For an example, see Lab9.html | Interacting with a virtual robot

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

```
moveForward :  ‘{ "type": "moveForward", "arg": 300 }‘
moveBackward :  ‘{ "type": "moveBackward", "arg": 300 }‘

turnRight :    ‘{ "type": "turnRight", "arg": 300 }‘
turnLeft : ‘{ "type": "turnLeft", "arg": 300 }‘

stop :     ‘{ "type": "alarm" }‘
```

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

```
webpage-ready : ‘{ "type": "webpage-ready, "arg": {} }‘
sonar-activated : ‘{
     "type:" "sonar-activated",
     "arg": { "sonarName": "sonarName", "distance": 1, "axis": "x" }
}‘
collision : ‘{
    "type": "collision",
    "arg": { "objectName": "obstacle-1" }
}‘
```

2.2 An example of usage in Java

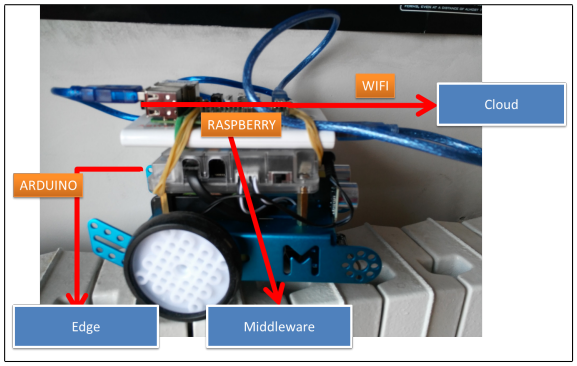
.

[1] See https://github.com/PierfrancescoSoffritti/ConfigurableThreejsApp

An example of usage at this basic level can be found in:
it.unibo.eclipse.qak.robot/src/resources/clientWenvObjTcp.kt.

3 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.



Our real robot is made of a RapberryPi 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.

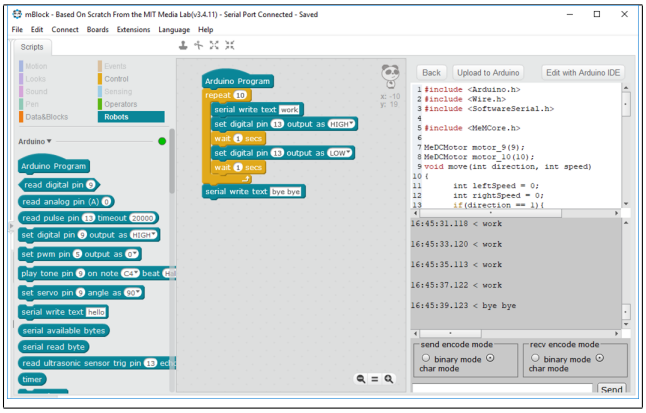
The software for the real robot is made of two main parts:

1. a low-level hardware-related part running on Arduino (see Subsection 3.2);
2. a high-level robot-control part running on the RaspberryPi (see Subsection ??).

3.1 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:



Of course, this is quite limited way to build low-level code for the mbot robot.

3.2 The low-level code on Arduino

The code can be built by working directly within a conventional Arduino IDE. We must 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 current code is composed of the following parts:

1. A set of declarations Subsection 3.2.1
2. A interpreter of commands to move the robot Subsection 3.2.2
3. An emitter of sonar data Subsection 3.2.3
4. A function that implements a *line-follower strategy* Subsection 3.2.4
5. The set-up and the main loop Subsection 3.2.5

3.2.1 Declarations. A set of declarations:

```
#include <Arduino.h>
#include <Wire.h>
#include <SoftwareSerial.h>
#include <MeMCore.h>

MeDCMotor motor_9(9);
MeDCMotor motor_10(10);
double angle_rad = PI/180.0;
double angle_deg = 180.0/PI;
void lookAtSonar();
void move(int direction, int speed);
double sonar;
int input;
int count;
MeUltrasonicSensor ultrasonic_3(3);
MeRGBLed rgbled_7(7, 7==7?2:4);
void remoteCmdExecutor();
```

Listing 1.1. `uniboControl.ino`: declarations

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

```
void remoteCmdExecutor()
{
    if((Serial.available()) > (0 )){
        input = Serial.read();
        //Serial.println(input);
        switch( input ){
          case 119 : move(1,150); break; //w
```

```
        case 115 : move(2,150); break; //s
        case 97 : move(3,150); break; //a
        case 100 : move(4,150); break; //d
        case 104 : move(1,0); stopFollow = true; break; //h
        case 102 : move(1,0); stopFollow = false; break; //f
        default : move(1,0); stopFollow = true;
      }
   }
}
/*
 * ---------------------------------
 * Moving
 * ---------------------------------
 */
void move(int direction, int speed)
{
     int leftSpeed = 0;
     int rightSpeed = 0;
     if(direction == 1){ //forward
          leftSpeed = speed;
          rightSpeed = speed;
     }else if(direction == 2){ //backward
          leftSpeed = -speed;
          rightSpeed = -speed;
     }else if(direction == 3){ //left
          leftSpeed = -speed;
          rightSpeed = speed;
     }else if(direction == 4){ //right
          leftSpeed = speed;
          rightSpeed = -speed;
     }
     motor_9.run((9)==M1?-(leftSpeed):(leftSpeed));
     motor_10.run((10)==M1?-(rightSpeed):(rightSpeed));
}
```

Listing 1.2. `uniboControl.ino`: the command interpreter

3.2.3 Sonar data. 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):

```
void lookAtSonar()
{
   sonar = ultrasonic_3.distanceCm();
   //emit sonar data but with a reduced frequency
   if( count++ > 50 ){ Serial.println(sonar); count = 0; }
   if((sonar) < (10)){ //very near
       if(((input)==(119))){
           move(1,0);
           rgbled_7.setColor(0,60,0,0);
           rgbled_7.show();
           //Serial.println("OBSTACLE FROM ARDUINO");
           _delay(0.3);
           move(2,100);
           _delay(1);
           move(2,0);
       }
   }
}
```

Listing 1.3. `uniboControl.ino`: sonar data

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

```
void _loop(){
}

void _delay(float seconds){
    long endTime = millis() + seconds * 1000;
    while(millis() < endTime)_loop();
}
```

Listing 1.4. `uniboControl.ino`: delay

3.2.4 Line follower. A function that implements a line-follower strategy:

```
double stopFollow = true;
double sonarVal;
void lineFollow();
MeLineFollower linefollower_2(2);

void sonarDetect()
{
    sonarVal = ultrasonic_3.distanceCm();
    Serial.println(sonarVal);
    if((sonarVal) < (10)){
        move(1,0);
        stopFollow = true;
        //Serial.println("stopFollow line follow");
    }
}
void lineFollow()
{
    if( stopFollow == true ) return;
    if(((linefollower_2.readSensors())==(0))){
        move(1,200);
    }
    if(((linefollower_2.readSensors())==(1))){
        motor_9.run((9)==M1?-(0):(0));
        motor_10.run((10)==M1?-(150):(150));
    }
    if(((linefollower_2.readSensors())==(2))){
        motor_9.run((9)==M1?-(150):(150));
        motor_10.run((10)==M1?-(0):(0));
    }
    if(((linefollower_2.readSensors())==(3))){
        move(2,100);
    }
    //sonarDetect();
}
```

Listing 1.5. `uniboControl.ino`: the line-follower

3.2.5 Setup. The set-up and the main loop:

```
void setup(){
    Serial.begin(115200);
    //Serial.println("start");
}

void loop(){
    rgbled_7.setColor(0,0,60,0);
    rgbled_7.show();
    remoteCmdExecutor();
    lookAtSonar();
```

```
11      lineFollow();
12      _loop();
13  }
```

Listing 1.6. `uniboControl.ino`: the main loop

3.3 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.

The current code is composed of the following parts:

1. Initialization Subsection 3.3.1
2. Command sender Subsection 3.3.2
3. Observer Subsection 3.3.3
4. Data receiver Subsection 3.3.4

3.3.1 Initialization. The first part of `mbotConnArduinoObj` provides operations to initialize the serial connection between a PC or a Raspeberry 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 1.7. `MbotConnArduinoObj.java: init`

3.3.2 Command sender.

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

```
    public void executeTheCommand( char cmd) {
        System.out.println("mbotConnArduinoObj executeTheCommand " + cmd + " conn=" + conn);
        switch(cmd) {
            case 'h' : mbotStop(); break;
            case 'w' : mbotForward(); break;
            case 's' : mbotBackward(); break;
            case 'a' : mbotLeft(); break;
            case 'd' : mbotRight(); break;
            case 'f' : mbotLinefollow(); break;
        }
    }
    public  void mbotForward() {
        try { if( conn != null ) conn.sendCmd("w"); } catch (Exception e) {e.printStackTrace();}
    }
    public  void mbotBackward() {
        try { if( conn != null ) conn.sendCmd("s"); } catch (Exception e) {e.printStackTrace();}
    }
    public  void mbotLeft() {
        try { if( conn != null ) conn.sendCmd("a"); } catch (Exception e) {e.printStackTrace();}
    }
    public  void mbotRight( ) {
        try { if( conn != null ) conn.sendCmd("d"); } catch (Exception e) {e.printStackTrace();}
    }
    public  void mbotStop() {
        try { if( conn != null ) conn.sendCmd("h"); } catch (Exception e) {e.printStackTrace();}
    }
    public  void mbotLinefollow( ) {
        try { if( conn != null ) conn.sendCmd("f"); } catch (Exception e) {e.printStackTrace();}
    }
```

Listing 1.8. `MbotConnArduinoObj.java: output`

3.3.3 Observer.

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

```
    public void addObserverToSensors( ISensorObserverFromArduino observer ){
        this.observer = observer;
    }
```

Listing 1.9. `MbotConnArduinoObj.java: addObserver`

3.3.4 Data receiver.

Finally, there is an operation that waits for sonar values coming from Arduino (see Subsection 3.2.3) and maps these data into an event `realSonar:sonar(DISTANCE)`:

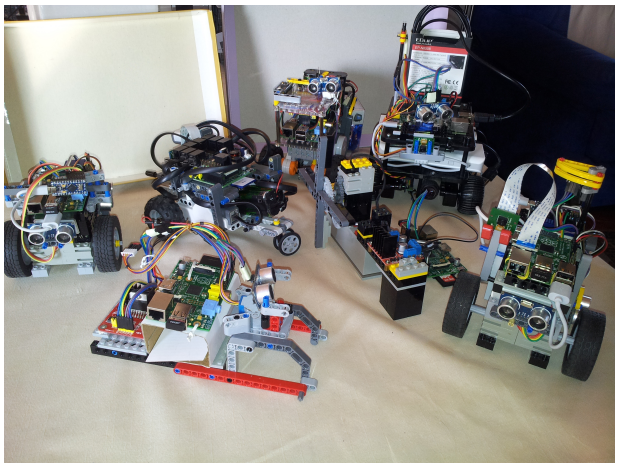
```
    private void getDataFromArduino() {
        new Thread() {
            public void run() {
                try {
                    System.out.println("mbotConnArduinoObj getDataFromArduino STARTED" );
                    while(true) {
                        try {
                            curDataFromArduino = conn.receiveALine();
//                          System.out.println("mbotConnArduinoObj received:" + curDataFromArduino );
                            double v = Double.parseDouble(curDataFromArduino);
                            //handle too fast change
                            double delta = Math.abs( v - dataSonar);
                            if( delta < 7 && delta > 0.5 ) {
                                dataSonar = v;
                                System.out.println("mbotConnArduinoObj sonar:" + dataSonar);
                                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 }
```

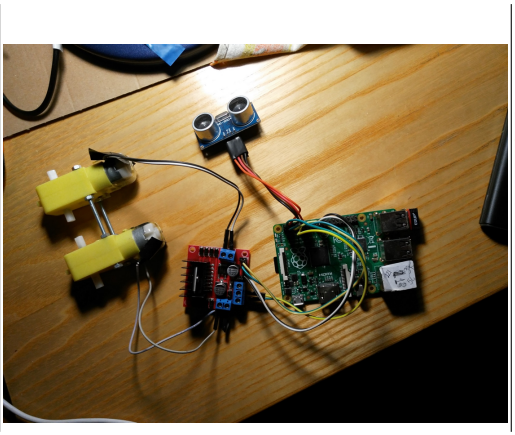
Listing 1.10. `MbotConnArduinoObj.java: input`

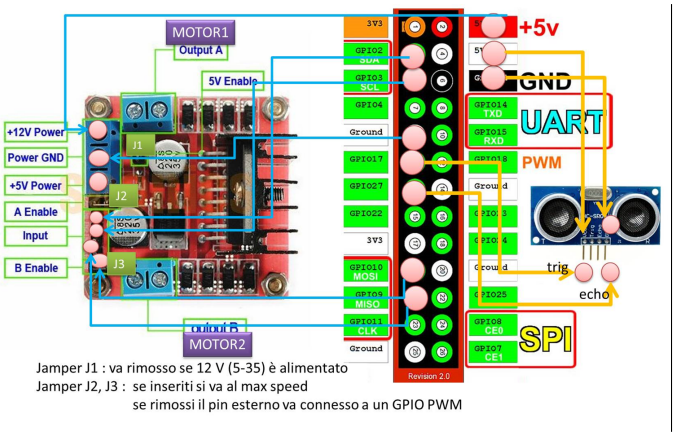
4 The in-house (nano) robot

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 :

```
#!/bin/bash
# ------------------------------------------------------------------
# nanoMotorDriveA.sh
# test for nano0
# Key-point: we can manage a GPIO pin by using the GPIO library.
# On a PC, edit this file as UNIX
# ------------------------------------------------------------------
```

```
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 1.11. `nanoMotorDriveA.sh`

4.1 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.

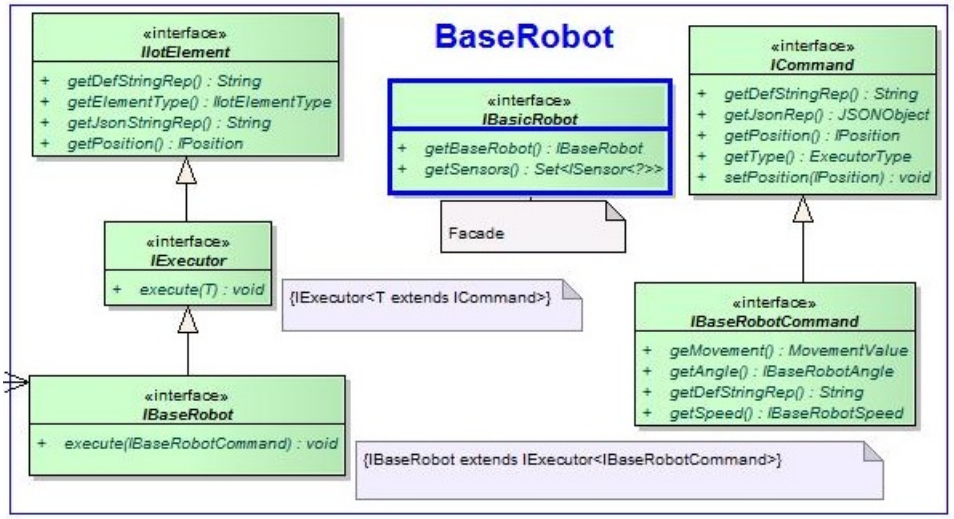
A software layer of this kind is provided by the library `labbaseRobotSam.jar`. More specifically:

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

4.2 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 1.12. IBasicRobot.java

4.3 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
```

```java
public class BasicRobot implements IBasicRobot{
private static IBasicRobot myself = null;
public static IBasicRobot getRobot(){
    if( myself == null ) myself = new BasicRobot();
    return myself;
}
public static IBaseRobot getTheBaseRobot(){
    if( myself == null ) myself = new BasicRobot();
    return myself.getBaseRobot();
}
//-------------------------------------------
private Configurator configurator;
private IBaseRobot robot ;
    //Hidden constructor
    protected BasicRobot() {
        init();
    }
    protected void init(){
        configurator = Configurator.getInstance();
        robot       = configurator.getBaseRobot();
    }
    public IBaseRobot getBaseRobot(){
        return robot;
    }
    public Set<ISensor<?>> getSensors(){
        Set<ISensor<?>> sensors = configurator.getSensors();
        return sensors;
    }
}
```

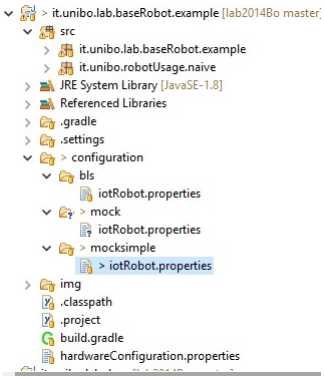
Listing 1.13. `BasicRobot.java`

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 4.5).

4.4 Using a BaseRobot

Let us introduce a robot with configuration named `mocksimple` (see Subsection 4.5), equipped with two motors and a distance sensor, all simulated.

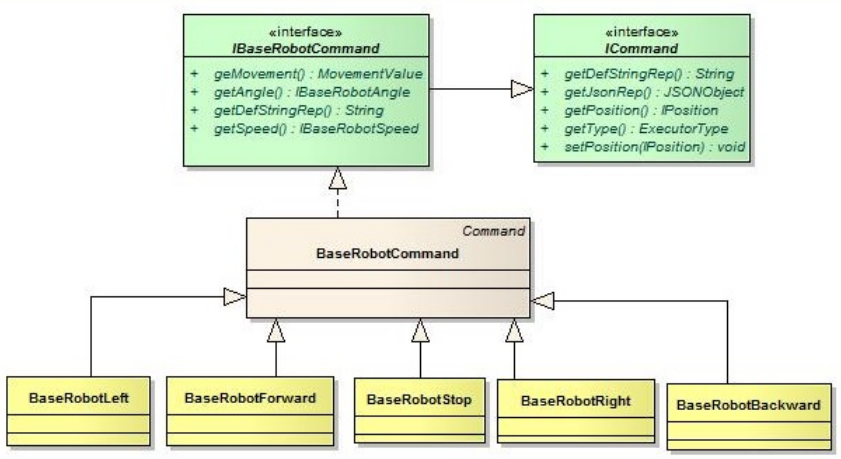
4.4.1 The project workspace The application designer must organize its project workspace as shown in the following snapshot:



4.4.2 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 IBasicRobot 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 //          sensor.addObserver(observer);
42 //      }
43         addObserverToSensors(observer);
44     }
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 1.14. `BasicRobotUsageNaive.java`



4.5 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*):

```
# ==================================
# ioRobot.properties for mocksimple robot
# Pay attention to the spaces
# ==================================
# --------- MOTORS ---------
motor.left=mock
motor.left.private=false
#
motor.right=mock
motor.right.private=false
# --------- SENSORS ---------
distance.front=mock
distance.front.private=false
# --------- COMPOSED COMPONENT ---------
actuators.bottom=ddmotorbased
actuators.bottom.name=motors
actuators.bottom.comp=motor.left,motor.right
actuators.bottom.private=true
# --------- MAIN ROBOT ---------
baserobot.bottom=differentialdrive
baserobot.bottom.name=mocksimple
baserobot.bottom.comp=actuators.bottom
baserobot.bottom.private=false
```

Listing 1.15. `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**)

4.6 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:

```
# ================================
# ioRobot.properties for nano robot
# Pay attention to the spaces
# ================================
# --------- MOTORS ---------
motor.left=gpio.motor
motor.left.pin.cw=8
motor.left.pin.ccw=9
motor.left.private=false
#
motor.right=gpio.motor
motor.right.pin.cw=12
motor.right.pin.ccw=13
motor.right.private=false
# --------- SENSORS ---------
distance.front_top=hcsr04
distance.front_top.trig=0
distance.front_top.echo=2
distance.front_top.private=false
# --------- COMPOSED COMPONENT ---------
actuators.bottom=ddmotorbased
actuators.bottom.name=motors
actuators.bottom.comp=motor.left,motor.right
actuators.bottom.private=true
# --------- MAIN ROBOT ---------
baserobot.bottom=differentialdrive
baserobot.bottom.name=nano
baserobot.bottom.comp=actuators.bottom
baserobot.bottom.private=false
```

Listing 1.16. `configuration/nano/iotRobot.properties`

4.7 Sensors and Sensor Data

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

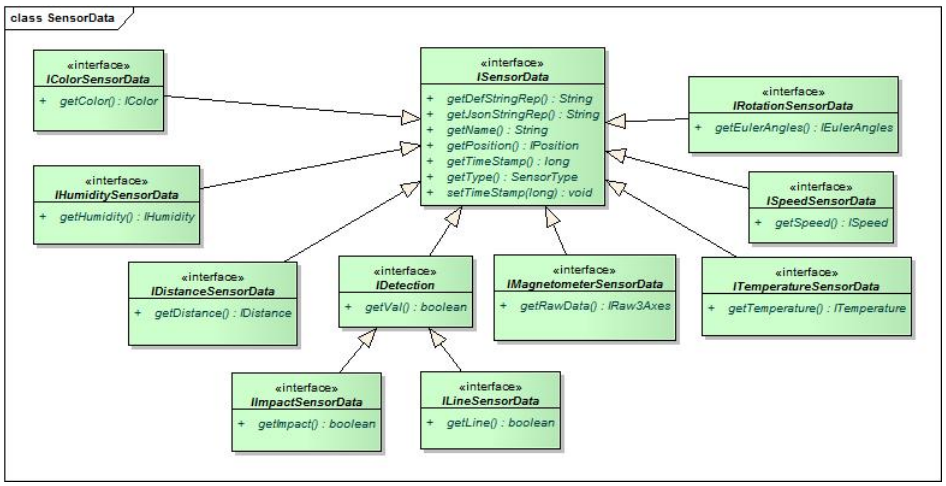
```
RobotSensorType: Line | Distance | Impact | Color | Magnetometer
```

Each sensor:

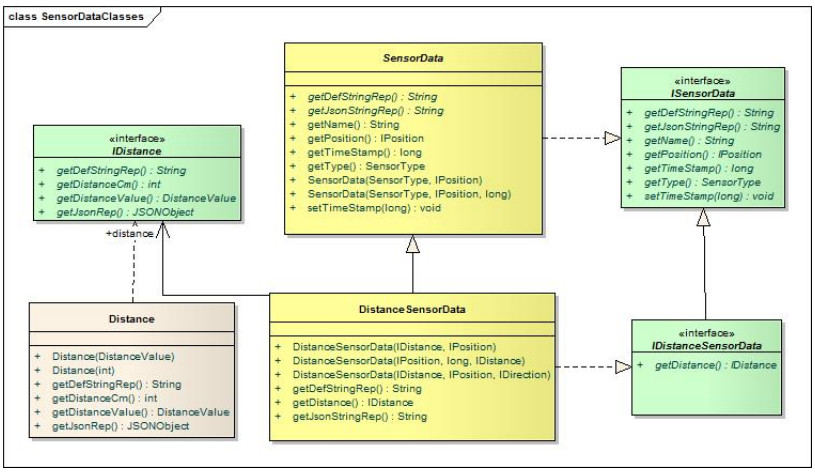
– is associated to a position that can assume one of the following values:

```
DONTCARE|
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:

4.7.1 Sensor data representation in Prolog (high level)

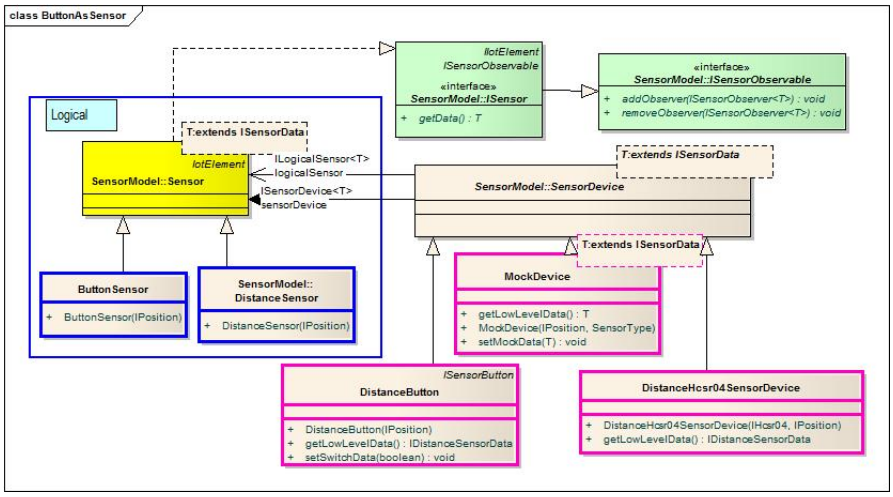
COLOR	`color(255 255 255, front)`
DISTANCE	`distance(43,forward, front)`
IMPACT	`impact(touch/loss, front)`
LINE	`line(lineLeft/lineDetected, bottom)`
MAGNETOMETER	`magnetometer(x(50),y(100),z(0), front)`

4.7.2 Sensor data representation in Json (low level)

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

4.8 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`.

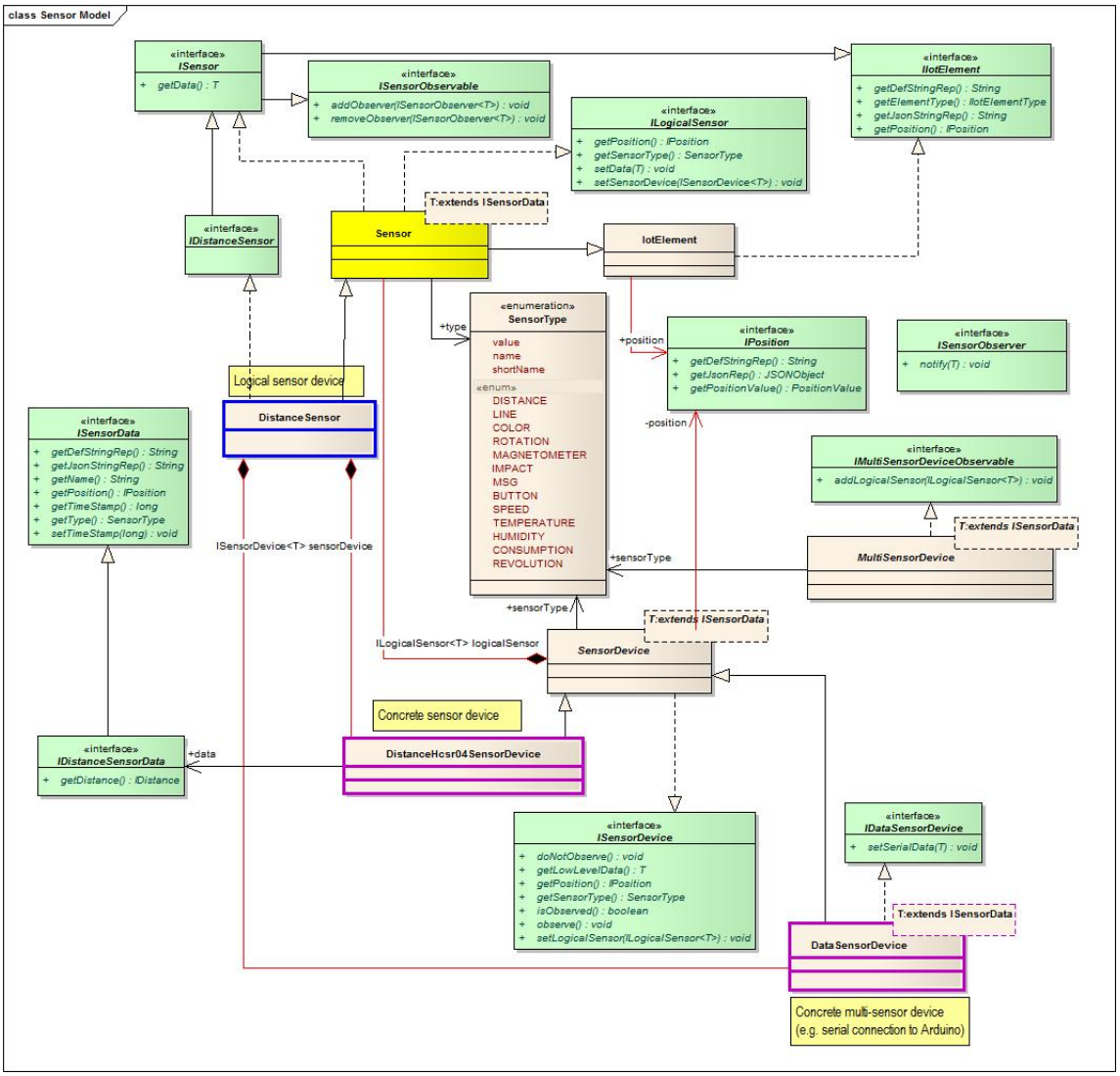


The model reported in the picture above shows that:

- 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 diagrma 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.

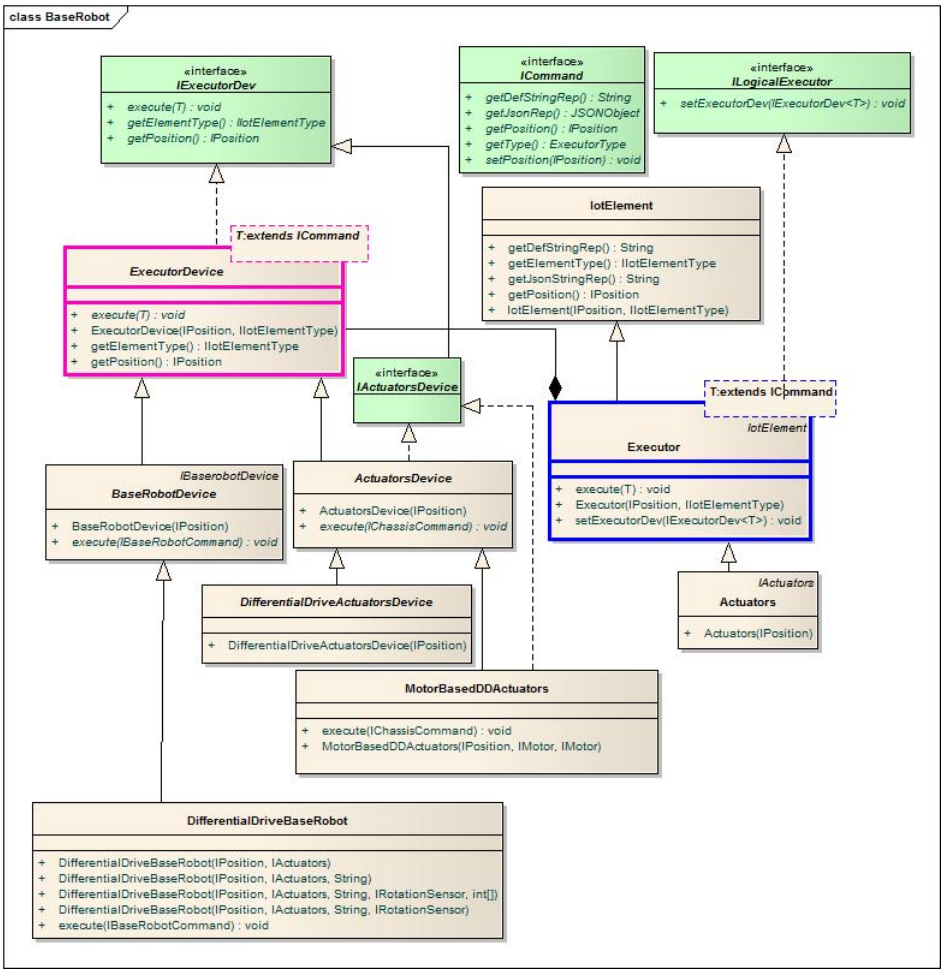
In this way, according to the GOF pattern *Bridge*, the `Sensor` abstraction hierarchy is decoupled from the hierarchy of `SensorDevice` implementation.

A more detailed picture is reported hereunder:



4.9 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.