The ButtonLed System: from Object Oriented programs to the Internet Of Things

Antonio Natali

Alma Mater Studiorum – University of Bologna
via Sacchi 3,47023 Cesena, Italy,
Viale Risorgimento 2,40136 Bologna, Italy
`antonio.natali@studio.unibo.it`

Table of Contents

1 The problem

We have to solve to following problem:

> Build a distributed system composed of a Button and a Led, each controlled by a different computational node.
> The system must initially provide a very basic function: a Led is turned on and off each time a Button is pressed (by an human user).
> In the future, the functionalities of the systems could be extended, e.g. by allowing a Button to blink a Led, to turn on/off many Led etc.

We read the text file that describes the requirements and immediately plan some work to:

– define the model of the Led (question: *what* is a Led in the user domain space?);
– define the model of the Button (question: *what* is a Button in the user domain space?);
– define the (model of the) Use Cases (i.e. define the functions/features of the software system).

1.1 The UseCases.

The main functions of the software systems can be summarized as done in |»site/BLS usecases.

1.2 From Requirement Analysis to the software development process

After the analysis related to the system components, we ask ourselves whether the models of the Led and of the Button defined in cooperation with the user can be also suited to face the problem of building the required distributed software system. Since it is not the case, we immediately advert a conceptual distance between the basic components and the needs of the application (i.e. we find an *abstraction gap*) that can be reduced by describing the single components and the whole system by means of a custom meta-model, no more based on classical objects but on *actors*.

In the following, we will show how a proper requirement analysis staring from the system components, properly related to the problem analysis and formally expressed by (executable) models can provide a solid guide for agile (`SCRUM`-based) project and implementation phases by reducing the risks and the costs for the software company.

2 The Led (as basic logical device)

In the application domain of the customer, the Led is a physical device (see |»site/LedEntry) that can be modelled as a *Plain Old Java Object* (see net/POJO).

2.1 The Led as a POJO

More precisely, we can state that a Led is an object with a modifiable state that can provide (implement) the following interface:

```java
public interface ILed {
  public String getName();         // property , primitive
  public void turnOn();            // modifier , primitive
  public void turnOff();           // modifier , primitive
  public java.awt.Color getLedColor(); // property , primitive
  public boolean isOn();           // property , primitive
  public void doSwitch();          // non-primitive
  public String getDefaultRep();   // mapping , non-primitive
}
```

The comments near each operation give some first indication on the intended semantics of the operation according with the terminology reported in book/ops)

2.2 The Led model test-plan

However, the usage of comments is not the best way to specify the intended semantics of an operations. By recalling the motto of Young et al. (1985):

> A design without specifications cannot be right or wrong, it can only be surprising!

our next step is to immediately introduce a test plan (see book/test plans), to better specify the expected behaviour (i.e. the meaning) of each operation.

The test plan that follows is directly expressed in JUnit (see net/JUnit tutorial) even if we do not have, at this moment, nothing to test. The idea is to use each test as a *specification* of the expected behaviour and as a way to express in a more formal way what each operation should do.

```java
package it.unibo.buttonLedSytem.tests;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import alice.tuprolog.Prolog;
import alice.tuprolog.Term;
import it.unibo.bls.highLevel.interfaces.IDevLed.LedColor;
import it.unibo.bls.lowLevel.interfaces.ILed;
import it.unibo.buttonLed.components.LedMock;
import it.unibo.system.SituatedSysKb;

public class TestLed {
protected ILed led;

@Before
    public void setUp() {
        System.out.println(" *** setUp " );
        try{
            //THIS SENTENCE IS INTRUDUCED AFTER PRJECT AND IMPLEMENTATION
            led = new LedMock("led1", SituatedSysKb.standardOutEnvView, LedColor.GREEN);
        }catch(Exception e){
            fail("setUp");
        }
    }
@After
    public void tearDown() throws Exception{
        System.out.println(" *** tearDown " );
    }
@Test
    public void testCreation(){
        System.out.println("  testCreation ... " );
        assertTrue("testCreation", ! led.isOn() );
    }
@Test
public void testTurnOn(){
    System.out.println("  testTurnOn ... " );
    led.turnOn();
    assertTrue("testTurnOn", led.isOn() );
 }
@Test
public void testTurnOff(){
    System.out.println("  testTurnOff ... " );
    led.turnOff();
    assertTrue("testTurnOf", ! led.isOn() );
}
@Test
public void testSwitch(){
    System.out.println("  testSwitch ... " );
    led.doSwitch();
    assertTrue("testSwitch", led.isOn() );
    led.doSwitch();
    assertTrue("testSwitch", ! led.isOn() );
}
@Test
public void testRep(){
    System.out.println("  testrep ... " );
    String rep  = led.getDefaultRep();
    String color = led.getLedColor()==LedColor.RED ? "RED" : "GREEN";
    String repExpected="device("+led.getName() +","+color+",false)";
    System.out.println("rep="+ rep + " repExpected=" + repExpected);
    assertTrue("testrep", rep.equals(repExpected) );
 }
@Test
public void testRepUnify(){
    System.out.println("  testRepUnify ... " );
    String color = led.getLedColor()==LedColor.RED ? "RED" : "GREEN";
    String repExpected="device("+led.getName() +","+color+",false)";
    Term te = Term.createTerm(repExpected);
    Term tr = Term.createTerm(led.getDefaultRep());
    boolean match = new Prolog().unify(tr, te);
    assertTrue("testRepUnify", match );
}
}
```

Listing 1.1. `A Test Plan for the Led`

2.3 The Led as an actor (problem analysis)

The current model of the Led as a `POJO` does not include any capability to interact via the network with the other components of the system. In other words, this model is not able to capture the distributed nature of the system.

In the context of a distributed button-led system, the Led could be conceived as a more advanced device, modelled as an *actor* able to receive and execute command messages. This new model can be expressed by using the `qa` metamodel as follows:

```
/*
 * ledMsg.qa
 * This is A MODEL defined during REQUIREMENT or PROBLEM ANALYSIS
 * by using the qa CUSTOM meta-model / language
 */
System ledMsg -regeneratesrc
Dispatch turnLed : turnLed(X)

Context ctxLedMsg ip [host="localhost" port=8010] -g cyan -httpserver

QActor ledmsg context ctxLedMsg {
    Plan init normal
        println( ledmsg(starts) );
        switchToPlan configure ;
        switchToPlan work
    Plan configure resumeLastPlan
        solve consult("ledTheory.pl") time(0) onFailSwitchTo prologFailure ;
        solve createLed("led",0) time(0) onFailSwitchTo prologFailure
    Plan work
        receiveMsg time(300000); //after 300 secs end the timeout
        [ !? tout(X,Y) ] switchToPlan toutExpired ;
        onMsg turnLed : turnLed(V) -> println( turnLed(V) );
        onMsg turnLed : turnLed(V) -> solve turnTheLed(V) time(0) onFailSwitchTo prologFailure;
        repeatPlan 0
    Plan prologFailure resumeLastPlan
        println("ledmsg has failed to solve a Prolog goal" )
    Plan toutExpired
        [ ?? tout(X,Y) ] println( timeout(X,Y) )
}
```

Listing 1.2. `ledMsg.qa`

The `ledmsg` actor is a state machine that first performs a configuration phase and then works in a message-based way. In the configuration phase the actor load a theory (`ledTheory`) and creates a (`POJO`) Led, that will be updated by calling tuProlog rules (`turnTheLed/1` or `turnOn/turnOff`) defined in the `ledTheory`.

Thus, the idea of a Led as a `POJO` is not abandoned; it is simply included (embedded) into the more advanced concept of `ledmsg` actor.

2.4 The Led (configuration) `ledTheory`

The main task of the `ledTheory` is done during its *initialization*: load a specific led-theory for each Led implementation:

```
/*
===============================================================
ledTheory.pl
Defines the rules to load a specific theory for each Led implementation
===============================================================
*/
%% ledImplementation( mock ).
ledImplementation( gui ).
%% ledImplementation( rasp ).
%% ledImplementation( serial ).

/*
* -----------------------------------------------------
* For each implementation get the proper theory
* -----------------------------------------------------
*/
ledImplementationFile( mock, "ledMockTheory.pl" ).
ledImplementationFile( gui, "ledGuiTheory.pl"   ).
ledImplementationFile( rasp, "ledPi4jTheory.pl" ).
ledImplementationFile( serial, "ledSerialTheory.pl" ).
/*
* -----------------------------------------------------
* initialize :
```

```
*   if there is a concrete implementation use it
*   otherwise use a simple rule-based led prototype
* ----------------------------------------------------
*/
initialize :-
    ledImplementation( I ),!,
    ledImplementationFile( I,F ),
    %% actorPrintln( ledImplementationFile( I,F ) ),
    consult( F ),
    consult("ledUsageTheory.pl").

initialize :-
    actorPrintln("initializing ledMsgTheory without any implementation"),
    consult( "ledNoImplTheory.pl" ).

:- initialization(initialize).
```

Listing 1.3. `ledTheory.pl`

If no `ledImplementation/1` fact is defined within the `ledTheory`, then we do not have any concrete Led implementation to use, as usually happens during the requirement or problem analysis phase. Nevertheless, we aim at introducing a working model even in this early phase of software development, in order to better interact with the user and to fix the requirements as soon as possible. For this reason the `ledTheory` loads a `ledNoImplTheory`.

2.5 A first Led (implementation) object

The `ledNoImplTheory` models the led state as a fact `ledState/1` and 'implements' the Led operations by introducing a proper set of rules:

```
/*
===============================================================
ledNoImplTheory.pl
Defines the rules to create a Led on the knowledge base
===============================================================
*/
/*
---------------------------------
Current state of the LED
---------------------------------
*/
ledState( off ).

/*
---------------------------------
Led construction rule
---------------------------------
*/
createLed(Name,Color) :-
    assert( ledName( Name ) ),
    assert( ledColor( Color ) ),
    turnTheLed( off ).
/*
------------------------------------------------
Led internal behavior (led implemented as a fact)
------------------------------------------------
*/
turnTheLed( on ):-
    retract( ledState( _ ) ),
    assert( ledState(on) ),
    show.
turnTheLed( off ):-
    retract( ledState( _ ) ),
    assert( ledState(off) ),
    show.
```

```
show:-
   ledName( N ),
   ledColor( C ),
   ledState( V ),
   actorPrintln( led( N,C,V ) ).
/*
-------------------------------------------------
Led 'public' operations (ILed interface)
-------------------------------------------------
*/
isOn  :-
   ledState(on).
turnOn :-
   turnTheLed( on ),
   isOn.      %%test
turnOff :-
   turnTheLed( off ),
   not isOn.  %%test
doSwitch :-
   isOn,!,
   turnOff.
doSwitch :-
   turnOn.
```

Listing 1.4. `ledNoImplTheory.pl`

This theory defines a very simple implementation of the Led as a `POJO`, by introducing rules related to the `ILed` interface of Subsection 2.2.

In this way, we are able to build a working prototype of a basic component (and later of the whole system) in a short time still during requirement analysis; even the unit test plans of Subsection 2.2 can be included in the component prototype.

The `createLed/2` rule is by now defined by asserting some facts in the knowledge base. This rule will be proprerly re-defined in each specific Led implementation theory, in order to introduce a specific concrete device, for example the `LedMock` used in the test plan of Subsection 2.2.

2.6 Testing the actor Led component

The Led component model can be executed by adding in the `qa` model a simple message generator (that will be later replaced by a button or by some other input device):

```
QActor ledtester context ctxLedMsg {
   Plan init normal
      delay time(800) ;
      forward ledmsg -m turnLed : turnLed(on) ;
      delay time(500) ;
      forward ledmsg -m turnLed : turnLed(off) ;
      repeatPlan 2
}
```

Listing 1.5. `ledMsg.qa extended with a Led command generator`

2.7 A Mock Led (project/implementation phase)

Our software company has already developed an implementation for a mock object[1] for the Led :

[1] In object-oriented programming, *mock objects* are simulated objects that mimic the behaviour of real objects in controlled ways.

```java
package it.unibo.buttonLed.components;
import it.unibo.bls.highLevel.interfaces.IDevLed.LedColor;
import it.unibo.is.interfaces.IOutputEnvView;

public class LedMock extends DeviceLedImpl{
   public LedMock( String name, IOutputEnvView outEnvView, LedColor color) throws Exception{
      super(name,outEnvView,color);
   }
   public LedMock(String defaltRep) throws Exception {
      super(defaltRep);
   }
   protected void show(){
      this.println( this.getDefaultRep() );
   }
}
```

Listing 1.6. `LedMock.java`

Since this class is the result of some project or implementation phase, a lot of work has been already done to enhance code reusability. In particular, the software team has introduced the class `DeviceLedImpl` (see Subsection 3.2) as a basic class for all the different types of Led implementations; in this way, for each specific Led implementation we have to redefine some operation only (in this case the internal operation `show`).

In order to 'inject' this new implementation of the Led into our logical component, we define the theory `ledMockTheory` with a specific `createLed/2` rule:

```prolog
/*
===============================================================
ledMockTheory.pl
Defines the rules to create a mock for the Led
===============================================================
*/

createLed(Name,Color) :-
   actorobj( Actor ),
   Actor <- getOutputEnvView returns OutView ,
   actorPrintln( createPojoMockLed( Name, Color) ),
   class("it.unibo.devices.qa.DeviceLedFactoryQa") <- createLedMock( Name, OutView, Color ) returns LED.
/* ------------------------------------------------------------------------------
   WARNING: assert( led(LED) ) cannot be used to store a reference to the LED
   We have to recur to the DeviceLedFactoryiQa.getTheLed operation
  ------------------------------------------------------------------------------
*/
```

Listing 1.7. `ledMockTheory.pl`

The next step is to introduce two new rules in the `ledTheory` of Subsection 2.4:

```prolog
ledImplementation( mock ).
...
ledImplementationFile( mock, "ledMockTheory.pl" ).
```

3 A Led factory

The Java class DeviceLedFactoryQa used in the LedMock of Subsection 2.7 is a *factory* that provides static methods to create a *singleton* Led object and to get a reference to such an object:

```java
package it.unibo.devices.qa;
import it.unibo.buttonLed.components.DeviceLedImpl;
import it.unibo.buttonLed.components.LedMock;
import it.unibo.buttonLedSystem.gui.DeviceLedGui;
import it.unibo.bls.highLevel.interfaces.IDevLed.LedColor;
import it.unibo.bls.raspberry.components.DeviceLedPi4j;
import it.unibo.is.interfaces.IOutputEnvView;

public class DeviceLedFactoryQa{
protected static DeviceLedImpl myself = null;


    public static DeviceLedImpl getTheLed( ){
//      System.out.println("getTheLed myself=" + myself);
        return myself;
    }

    public static DeviceLedImpl createLedMock(
            String name, IOutputEnvView outEnvView, int color ) throws Exception{
        if( myself == null ){
            LedColor ledcolor = (color == 0) ? LedColor.GREEN : LedColor.RED ;
            myself = new LedMock( name,outEnvView, ledcolor );
        }
        return myself;
    }
```

Listing 1.8. DeviceLedFactoryQa.java : createLedMock

Note that the creation method createLedMock returns (like any other creation method that we will introduce in the factory) an object of the class DeviceLedImpl (see Subsection 3.2).

3.1 The Led usage theory

Since we have introduced a factory that builds and gets objects of class DeviceLedImpl, we can write a set of Led-usage rules that do not depend on the specific Led implementation:

```prolog
/*
===============================================================
ledUsageTheory.pl
===============================================================
*/
turnTheLed( on ) :- turnOn.
turnTheLed( off ):- turnOff.

%% Led 'public' operations (ILed interface) %%

isOn  :-
    class("it.unibo.devices.qa.DeviceLedFactoryQa") <- getTheLed returns LED,
    LED <- isOn.
turnOn :-
    class("it.unibo.devices.qa.DeviceLedFactoryQa") <- getTheLed returns LED,
    LED <- turnOn.
turnOff :-
    class("it.unibo.devices.qa.DeviceLedFactoryQa") <- getTheLed returns LED,
    LED <- turnOff.
doSwitch :-
    class("it.unibo.devices.qa.DeviceLedFactoryQa") <- getTheLed returns LED,
    LED <- doSwitch.
```

Listing 1.9. ledUsageTheory.pl

3.2 A basic class for Led implementation

The `DeviceLedImpl` class is based on the custom framework |»site/uniboEnv) introduced for the rapid development of `GUI`-based prototypes.

```java
package it.unibo.buttonLed.components;
import it.unibo.bls.highLevel.interfaces.IDevLed.LedColor;
import it.unibo.bls.lowLevel.interfaces.ILed;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.system.SituatedPlainObject;

public class DeviceLedImpl extends SituatedPlainObject implements ILed{
private boolean on = false;
protected LedColor color;

    public DeviceLedImpl( String name, IOutputEnvView outEnvView, LedColor color) throws Exception{
        super(name, outEnvView);
        this.color=color;
        configure();
    }
    public DeviceLedImpl( String defaltRep ) throws Exception{
        throw new Exception("Not yet implemented");
    }
    protected void configure() throws Exception{
        if( color == LedColor.RED || color == LedColor.GREEN){
            turnOff();
        }else throw new Exception("a led can be only RED or GREEN");
        System.out.println("DeviceLedImpl configure done" );
    }
    @Override
    public void doSwitch() {
         if( on ) turnOff();
         else turnOn();
    }
    @Override
    public void turnOn() {
//      println("LED turnOn " + isOn() ) ;
        on = true;
        show();
    }
    @Override
    public void turnOff() {
//      println("LED turnOff " + isOn() ) ;
        on = false;
        show();
    }
    @Override
    public LedColor getLedColor() {
        return color;
    }
    @Override
    public boolean isOn() {
        return on;
    }
    @Override
    public String getDefaultRep() {
        String ledLedColor = getLedColor()==LedColor.RED ? "RED" : "GREEN";
        return "device("+this.name+"," + ledLedColor + ","+ isOn() +")";
    }
    @Override
    public String getName() {
        return name;
    }
    /*
     * Do nothing at the moment
     */
    protected void show(){ println("LED NEVER HERE " + isOn() ) ; }
}
```

Listing 1.10. `DeviceLedImpl.java`

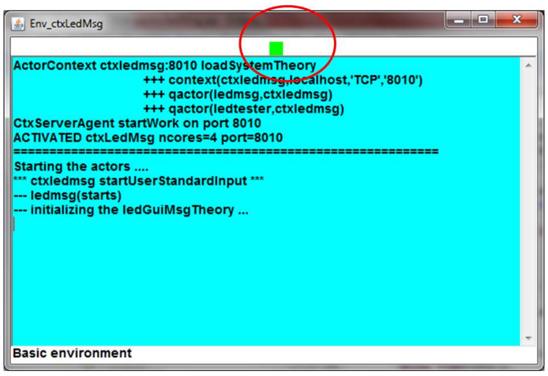
3.3 A Virtual Led (project/implementation phase)

A Led as a virtual device can now be introduced as follows:

```java
package it.unibo.buttonLedSystem.gui;
import it.unibo.bls.highLevel.interfaces.IDevLed.LedColor;
import it.unibo.buttonLed.components.DeviceLedImpl;
import it.unibo.buttonLedSystem.gui.interfaces.IDeviceLedGui;
import it.unibo.is.interfaces.IBasicEnvAwt;
import it.unibo.is.interfaces.IOutputEnvView;
import java.awt.Color;
import java.awt.Panel;

public class DeviceLedGui extends DeviceLedImpl implements IDeviceLedGui{
protected Panel p ;

    public DeviceLedGui( String name, IOutputEnvView outEnvView, LedColor color ) throws Exception{
        super( name, outEnvView, color );
    }
    protected void configure() throws Exception{
        IBasicEnvAwt env = outEnvView.getEnv(); //get the environment from the given view
        if( env == null) throw new Exception("no gui environment found");
        p= new Panel();
        if( color == LedColor.GREEN) p.setBackground(Color.green);
        else p.setBackground(Color.red);
        env.addPanel(p);
        p.validate();
        turnOff();
    }
    @Override
    @SuppressWarnings("deprecation")
    protected void show( ){
        if( p == null ) return;
        if( this.isOn() ){
            p.resize(15, 15);
        }
        else p.resize(5, 5);
        p.validate();
    }
    @Override
    public Panel getPanel() {
        return p;
    }
}
```

Listing 1.11. `DeviceLedGui.java`

The class `DeviceLedImpl` redefines two internal operations: `configure` and `show`. The result is shown in the following picture:



```
ActorContext ctxledmsg:8010 loadSystemTheory
                              +++ context(ctxledmsg,localhost,'TCP','8010')
                              +++ qactor(ledmsg,ctxledmsg)
                              +++ qactor(ledtester,ctxledmsg)
CtxServerAgent startWork on port 8010
ACTIVATED ctxLedMsg ncores=4 port=8010
==========================================================
Starting the actors ....
*** ctxledmsg startUserStandardInput ***
--- ledmsg(starts)
--- initializing the ledGuiMsgTheory ...
```

Basic environment

In order to 'inject' this new implementation of the Led into our logical component, we define the theory `ledGuiTheory` with a specific `createLed/2` rule:

```
/*
===============================================================
ledGuiTheory.pl
Defines the rules to create a GUI-based virtual Led
===============================================================
*/

createLed(Name,Color) :-
    actorobj( Actor ),
    Actor <- getOutputEnvView returns OutView ,
    actorPrintln( createPojoGuiLed( Name, Color) ),
    class("it.unibo.devices.qa.DeviceLedFactoryQa") <- createLedGui( Name, OutView, Color ) returns LED.
```

Listing 1.12. `ledGuiTheory.pl`

The next step is to introduce two new rules in the `ledTheory` of Subsection 2.4:

```
ledImplementation( gui ).
...
ledImplementationFile( gui, "ledGuiTheory.pl" ).
```

Finally, we introduce a new Led creation operation in the entry in the `DeviceLedFactoryQa`:

```
    public static DeviceLedImpl createLedGui(
            String name, IOutputEnvView outEnvView, int color) throws Exception{
        if( myself == null ){
            LedColor ledcolor = (color == 0) ? LedColor.GREEN : LedColor.RED ;
            myself = new DeviceLedGui(name,outEnvView, ledcolor);
        }
        return myself;
    }
```

4 A Led on Raspberry (implementation phase)

The `GPIO` pins on a Raspberry Pi are a great way to interface physical devices like Buttons and Leds through some simple hardware connection, as in the example described in |»site/BLS low-level)). In this example the Led anode is connected to pin 25 in `BCM` code[2] while the Led cathode is connected to a `GND` pin.

However, a software design never interacts in direct way with the hardware level; at least some minimal support for the control of external devices must be provided by the operating system. In the case of Linux we can start from the shell or from a more advanced library.

4.1 Led control using files

The basic way provided by Linux to manage a device connected on a `GPIO` pin is reading/writing some (virtual) file associated with that pin.

```
# -------------------------------------------------------------
# led25OnOff.sh
# Key-point: we can manage a device connected on a GPIO pin by
# reading/writing some (virtual) file associated with that pin.
#   sudo bash led25OnOff.sh
# -------------------------------------------------------------

echo Unexporting.
echo 25 > /sys/class/gpio/unexport #
echo 25 > /sys/class/gpio/export #
cd /sys/class/gpio/gpio25 #

echo Setting direction to out.
echo out > direction #
echo Setting pin high.
echo 1 > value #
sleep 1 #
echo Setting pin low
echo 0 > value #
sleep 1 #
echo Setting pin high.
echo 1 > value #
sleep 1 #
echo Setting pin low
echo 0 > value #
#schmod +x ledOnOff.sh #
```

Listing 1.13. `led25OnOff.sh`

4.2 Led control using the `GPIO` shell library

WiringPi is a `GPIO` access library written in `C`[3] for the `BCM2835` used in the Raspberry Pi. *WiringPi* includes a command-line utility `gpio` which can be used to program and setup the `GPIO` pins. We can use this to read and write the pins and even use it to control them from shell scripts.

```
# -------------------------------------------------------------
# led25Gpio.sh
# Key-point: we can manage a device connected on a GPIO pin by
```

[2] `BCM` refers to the pin number of the `BCM2835` chip, and this is the pin number used when addressing the `GPIO` using the `/sys/class/gpio` interface

[3] The *WiringPi* library was written by Gordon Henderson to allow `GPIO` communication from `C,C++` in a style similar to the Arduino Wiring programming language

```
# using the GPIO shell library.
# The pin 25 is physical 22 and Wpi 6.
#   sudo bash led25Gpio.sh
# -------------------------------------------------------------
gpio readall #
echo Setting direction to out
gpio mode 6 out #
echo Write 1
gpio write 6 1 #
sleep 1 #
echo Write 0
gpio write 6 0 #
```

Listing 1.14. `led25Gpio.sh`

4.3 Led control using python

The Raspbian Linux operating system has the `RPi.GPIO` library pre-installed. It is a Python library that handles interfacing with the `GPIO` pins.

```
# -------------------------------------------------------------
# ledPython25.py
# Key-point: we can manage a Led connected on the GPIO pin 25
# using the python language.
#   sudo python ledPython25.py
# -------------------------------------------------------------
import RPi.GPIO as GPIO
import time

'''
----------------------------------
CONFIGURATION
----------------------------------
'''
GPIO.setmode(GPIO.BCM)
GPIO.setup(25,GPIO.OUT)

'''
----------------------------------
main activity
----------------------------------
'''
while True:
   GPIO.output(25,GPIO.HIGH)
   time.sleep(1)
   GPIO.output(25,GPIO.LOW)
   time.sleep(1)
```

Listing 1.15. `ledPython25.sh`

4.4 Led control using Pi4j

`Pi4J` is an open source project intended to provide a bridge between the native hardware and `Java` for full access to the Raspberry Pi. In addition to the basic raw hardware access functionality, this project also attempts to provide a set of advanced features that make working with the Raspberry Pi an easy to implement and more convenient experience for `Java` developers.

Thus, the `Pi4J` library (see |»site/Pi4j) can be used as our basic software layer (our *technology assumption*) for the implementation a `Java` class for the control of a Led connected to some pin of a Raspberry Pi:

```java
package it.unibo.bls.raspberry.components;
import com.pi4j.io.gpio.GpioPinDigitalOutput;
import com.pi4j.io.gpio.PinState;
import it.unibo.bls.highLevel.interfaces.IDevLed.LedColor;
import it.unibo.buttonLed.components.DeviceLedImpl;
import it.unibo.gpio.base.GpioOnPi4j;
import it.unibo.is.interfaces.IOutputEnvView;

/*
 * =====================================================================
 * The button is implemented as an observable by using the library pi4j
 * =====================================================================
 */
public class DeviceLedPi4j extends DeviceLedImpl {
protected GpioPinDigitalOutput ledpi4j;
    public DeviceLedPi4j( String name, IOutputEnvView outEnvView, LedColor ledColor, int pinNum ) throws Exception{
        super(name, outEnvView, ledColor);
        myconfigure(pinNum);
    }
    protected void myconfigure(int pinNum) throws Exception{
        /*
         * To access a GPIO pin with Pi4J, we must first provision the pin.
         * Provisioning configures the pin based on how we intend to use it.
         * Provisioning can automatically export the pin, set its direction,
         * and setup any edge detection for interrupt based events.
         */
        ledpi4j = GpioOnPi4j.controller.provisionDigitalOutputPin( GpioOnPi4j.getPin(pinNum) );
    }
    @Override
    public void turnOn() {
        ledpi4j.high();
        super.turnOn();
    }
    @Override
    public void turnOff() {
        if(ledpi4j!=null) ledpi4j.setState(PinState.LOW);
        super.turnOff();
    }
}
```

Listing 1.16. `DeviceLedPi4j.java`

In this case the class redefines two primitive operations of `DeviceLedImpl` (see Subsection 3.2): `turnOn` and `turnOff`. Moreover, it extends the configuration phase by 'provisioning' (in `Pi4j` terminology) the `GPIO` pin connected to the Led anode. The class `GpioOnPi4j` is an utility class that maps a `GPIO` pin number in `BCM` into a *wiring Pi* code:

```java
package it.unibo.gpio.base;
import com.pi4j.io.gpio.GpioController;
import com.pi4j.io.gpio.GpioFactory;
import com.pi4j.io.gpio.Pin;
import com.pi4j.io.gpio.RaspiPin;
public class GpioOnPi4j implements IGpioPi4j{
public static final GpioController controller = GpioFactory.getInstance();
    public static Pin getPin(int pinNum) {
        switch(pinNum){
        case 4  : return RaspiPin.GPIO_07;
        case 17 : return RaspiPin.GPIO_00;
        case 18 : return RaspiPin.GPIO_01;
        case 21 : return RaspiPin.GPIO_02;
        case 22 : return RaspiPin.GPIO_03;
        case 23 : return RaspiPin.GPIO_04;
        case 24 : return RaspiPin.GPIO_05;
        case 25 : return RaspiPin.GPIO_06;
        case 27 : return RaspiPin.GPIO_02;
        }
        return RaspiPin.GPIO_00;
    }
```

```
   @Override
   public GpioController getGpioPi4j() {
       return controller;
   }
}
```

Listing 1.17. `GpioOnPi4j.java`

4.5 Using the Led Pi4j in the actor

In order to 'inject' this new implementation of the Led into our logical component, we define the theory `ledPi4jTheory.pl`:

```
/*
===============================================================
ledPi4jTheory.pl
Defines the rules to create a Led on Raspberry
===============================================================
*/
pinledwpi( 25 ).

createLed(Name,Color) :-
   pinledwpi(Pin),
   createPi4jLed( Name,Color,Pin ).

createPi4jLed( Name,Color,PinNum ) :-
   actorobj( Actor ),
   Actor <- getOutputEnvView returns OutView ,
   actorPrintln( createPi4jLed( Name, Color,PinNum) ),
   class("it.unibo.devices.qa.DeviceLedFactoryQa") <- createLedPi4j( Name, OutView, Color, PinNum ).
```

Listing 1.18. `ledPi4jTheory.pl`

The next step is to introduce two new rules in the `ledTheory` of Subsection 2.4:

```
ledImplementation( rasp ).
...
ledImplementationFile( rasp, "ledPi4jTheory.pl" ).
```

Finally, we introduce a new Led creation operation in the entry in the `DeviceLedFactoryQa`:

```
   public static DeviceLedImpl createLedPi4j(
           String name, IOutputEnvView outEnvView, int color, int pinNum) throws Exception{
       if( myself == null ){
           LedColor ledcolor = (color == 0) ? LedColor.GREEN : LedColor.RED ;
           myself = new DeviceLedPi4j( name,outEnvView, ledcolor, pinNum);
       }
       return myself;
   }
```

4.6 Code deployment on the Raspberry Pi

In order to test our code on the Raspberry Pi we have to perform the following actions:

1. create a runnable `jar` file from the generated file `src-gen/it/unibo/ctxLedMsg/MainCtxLedMsg.java`[4];
2. copy the runnable `jar` and the library sub-folder into a directory (e.g. `ledTest`) the Raspberry Pi;
3. copy the generated `scrMore` directory into `ledTest`;
4. copy the into `ledTest` the theories `ledTheory.pl`, `ledMockTheory.pl`, `ledGuiTheory.pl` `ledPi4jTheory.pl` and `ledUsageTheory.pl`.

[4] We suggest to copy the required libraries in a sub-folder to keep the `jar` short and to reduce the time of file transfer to the Raspberry Pi.

4.7 Test the Led on the Raspberry Pi

The behaviour of the Led component on the Raspberry Pi can be tested by launching (within the directory `ledTest`):

Launch the Led test
```
sudo java -jar MainCtxLedMsg
```

We can select one of the different Led implementations by setting one of the `ledPi4jTheory.pl` implementation rules:

Select the Led implementation
```
%% ledImplementation( mock ).
%% ledImplementation( gui ).
ledImplementation( rasp ).
```

The system with the rule `ledImplementation( gui )` works only under a `X11` system[5] to enable remote graphical access to applications.

4.8 The Led as a standalone device

Since we now have a Led working on a Raspberry Pi, we can immediately switch from a local system to a distributed one. To this end, let us introduce an actor that, working on a conventional `PC`, sends command messages to the Led on the Raspberry Pi:

```
/*
 * ledSenderMsg.qa
 * A ledtester for a remote led
 */
System ledSenderMsg -regeneratesrc

Dispatch turnLed : turnLed(X)

Context ctxLedSenderMsg ip [host="192.168.43.229" port=8030] -g cyan
Context ctxLedMsg ip [host="192.168.137.2" port=8010] -standalone

QActor ledmsg context ctxLedMsg {
    Plan init normal
        println("never here: I am a placeholder ")
}

/*
 * A actor that sends commands to the ledmsg
 */
QActor ledtester context ctxLedSenderMsg {
    Plan init normal
        delay time(800) ;
        forward ledmsg -m turnLed : turnLed(on) ;
        delay time(500) ;
        forward ledmsg -m turnLed : turnLed(off) ;
        repeatPlan 2
}
```

Listing 1.19. `ledSenderMsg.qa`

Note that the `ledtester` is identical to the actor introduced in Subsection 2.3, with the difference that it works in its own Context `ctxLedSenderMsg`. Moreover, the behaviour of the `ledmsg` actor is **not** explicitly defined; rather a 'place holder' actor is introduced in the specification in order to:

[5] The *X Window System* (`X11`, or shortened to simply `X`, and sometimes informally `X-Windows`) is a windowing system for bitmap displays, common on `UNIX`-like computer operating systems

1. allow us to reference the `ledmsg` actor in a message send (`forward`) operation;
2. define the context (in practice, the `IP` of the Raspberry Pi) in which the `ledmsg` is working.

The flag `-standalone` indicates that the Led context `ctxLedMsg` must be considered an entity outside the system and that the actors defined in it are just 'place holders' that do not produce any new code. More on this in Section 12.

5 A Led on Arduino (implementation phase)

The basic way provided by Arduino to manage a device connected on a pin is to perform a read or a write operation on that pin. For example we can blink a Led connected on pin 13 as follows:

```
/*
==========================================================
 project it.unibo.arduino.intro Led13/Led13.ino ARDUINO UNO
 Pin 13 has a internal LED connected on ARDUINO UNO.
==========================================================
 */
int ledPin  = 13;
int count   = 1;
boolean on  = false;

void setup(){
  Serial.begin(9600);
  Serial.println( "--------------------------------------" );
  Serial.println( "project it.unibo.arduino.intro" );
  Serial.println( "Led13/Led13.ino" );
  Serial.println( "--------------------------------------" );
  configure();
}
void configure(){
  pinMode( ledPin, OUTPUT );
  turnOff();
}
/* ----------------------------
LED primitives
-------------------------------- */
void turnOn(){
  digitalWrite(ledPin, HIGH);
  on = true;
}
void turnOff(){
  digitalWrite(ledPin, LOW);
  on = false;
}
/* ----------------------------
Blinck the led (non primtive)
-------------------------------- */
void blinkTheLed(){
    turnOn();
    delay(500);
    turnOff();
    delay(500);
}
/* ----------------------------
Input handler
-------------------------------- */
void loop(){
  if( count <= 5 ){
    Serial.println("Blinking the led on 13 count=" + String(count) );
    blinkTheLed();
    count++;
  }
}
```

Listing 1.20. `Led13.ino`

5.1 Led with input output

The next *sketch* does introduce a `cmdHandler` operation that looks at the serial line as an input device and turns the Led `on/off` if the input value is `1/0`.

```
/*
============================================================
 project it.unibo.arduino.intro Led13Msg/Led13Msg.ino ARDUINO UNO
 Pin 13 has a internal LED connected on ARDUINO UNO.
============================================================
 */
int ledPin  = 13;
int count   = 1;
boolean on  = false;

void setup(){
  Serial.begin(9600);
  Serial.println( "---------------------------------------" );
  Serial.println( "project it.unibo.arduino.intro" );
  Serial.println( "Led13Msg/Led13Msg.ino" );
  Serial.println( "---------------------------------------" );
  configure();
}
void configure(){
  pinMode( ledPin, OUTPUT );
  turnOff();
}
/* -----------------------------
LED primitives
-------------------------------- */
void turnOn(){
  digitalWrite(ledPin, HIGH);
  on = true;
  forwardTheLedState();
}
void turnOff(){
  digitalWrite(ledPin, LOW);
  on = false;
  forwardTheLedState();
}
void forwardTheLedState(){
   //msg( MSGID, MSGTYPE, SENDER, RECEIVER, CONTENT, SEQNUM )
  Serial.println("msg( info, dispatch, arduino, ANY, ledstate(" + String(on) + "), " + String(count++) + " )" );
}
/* -----------------------------
Blinck the led (non primtive)
-------------------------------- */
void blinkTheLed(){
    turnOn();
    delay(500);
    turnOff();
    delay(500);
}
/* -----------------------------
Input handler
-------------------------------- */
void cmdHandler(){
   int v = Serial.read(); //NO BLOCKING
   //if( v > 0 ) Serial.println("arduinio input=" + String(v) );
   if( v != - 1 && v != 13 && v != 10 ){
     boolean isPressed = ( v == 49 ) ; //48 is '0' 49 is '1'
     if( isPressed ) turnOn();
     else turnOff();
   }
}
void loop(){
  cmdHandler();
}
```

Listing 1.21. `Led13Msg.ino`

5.2 DeviceLedArduinoProxy

The possibility to send/receive information via the Serial Line can be used as our basic software layer (our *technology assumption*) for the implementation a Java class for the control of a Led connected to some pin of a Arduino device. To achieve the goal let us introduce a new specialized version of the DeviceLedImpl (see Subsection 3.2) class:

```
package it.unibo.devices.qa;
import jssc.SerialPortEvent;
import jssc.SerialPortEventListener;
import alice.tuprolog.Struct;
import alice.tuprolog.Term;
import it.unibo.bls.highLevel.interfaces.IDevLed.LedColor;
import it.unibo.buttonLed.components.DeviceLedImpl;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.is.interfaces.protocols.IConnInteraction;
import it.unibo.supports.FactoryProtocol;

public class DeviceLedArduinoProxy extends DeviceLedImpl implements SerialPortEventListener {
protected String PORT_NAME;
protected IConnInteraction conn; //the comm channel with Arduino is a "general" two-way IConnInteraction
protected FactoryProtocol factoryProtocol;

    public DeviceLedArduinoProxy( String name, LedColor ledColor, String portName, IOutputEnvView outView ) throws
          Exception{
        super( name, outView, ledColor ) ;
        this.PORT_NAME = portName;
        myconfigure();
    }
    protected void myconfigure() {
        try {
            println("DevLedArduinoProxy waiting for connection ... " + PORT_NAME );
            factoryProtocol = new FactoryProtocol(outView, "SERIAL", name+"pxy");
            conn = factoryProtocol.createSerialProtocolSupport(PORT_NAME,this,true);
        } catch (Exception e) { e.printStackTrace(); }
    }
    /* The proxy is a "mirror" of the state in order to reduce network traffic */
    @Override
    public void turnOn() {
        try {
            if( ! isOn() && conn != null ) {
                conn.sendALine("1");
            }
            super.turnOn();
        } catch (Exception e) {
            println("DevLedArduinoProxy turnOn ERROR " + e.getMessage() );
        }
    }
    @Override
    public void turnOff() {
        try {
            if( isOn() && conn != null ) {
                conn.sendALine( "0");
            }
            super.turnOff();
        } catch (Exception e) {
            println("DevLedArduinoProxy turnOff ERROR " + e.getMessage() );
        }
    }
/*
 * ====================================================
 */
    @Override
    public synchronized void serialEvent(SerialPortEvent oEvent) {
//      println("DeviceLedArduinoProxy serialEvent event type=" + oEvent.getEventType() );
        try {
             String input = conn.receiveALine();
             if( input.startsWith("msg")){
                println("DevLedArduinoProxy input=" + input );
            handleMsgContent(input);
          }
      } catch (Exception e) {
          println("DevLedArduinoProxy ERROR:"+e.getMessage());
      }
    }
    protected void handleMsgContent(String input){
        try {
          //HANDLE a input WRITTEN IN PROLOG SYNTAX
          //msg( MSGID, MSGTYPE, SENDER, RECEIVER, CONTENT, SEQNUM )
          Struct msg = (Struct) Term.createTerm(input);
          Struct ledState = (Struct) msg.getArg(4);
          String ledValue = ledState.getArg(0).toString();
//          println("DevLedArduinoProxy ledValue=" + ledValue );
          this.notifyObservers( ledValue.equals("1") );
        } catch (Exception e) {
          println("DevLedArduinoProxy handleMsgContent " + input +" ERROR:"+e.getMessage());
        }
    }
}
```

Listing 1.22. `DeviceLedArduinoProxy.java`

The class `DeviceLedArduinoProxy`[6] redefines two primitive operations: `turnOn` and `turnOff`. Moreover, it extends the configuration phase by creating an object of interface `IConnInteraction` in order to use the serial connection with Arduino.[7]

5.3 IConnInteraction interface

The interface `it.unibo.is.interfaces.protocols.IConnInteraction` defines operations to send/receive Strings over a network connection.

```
package it.unibo.is.interfaces.protocols;

public interface IConnInteraction {
    public void sendALine( String msg ) throws Exception;
    public void sendALine( String msg, boolean isAnswer ) throws Exception;
    public String receiveALine( ) throws Exception;
    public void closeConnection( ) throws Exception;
}
```

Listing 1.23. `IConnInteraction.java`

The project *it.unibo.noawtsupports* defines a framework for network communications based on connected, two-way protocols (like `TCP`, `UDP`) and over a serial line.

The factory `it.unibo.supports.FactoryProtocol` creates objects of type `IConnInteraction` that hide the technological details related to specific protocols. More information on this framework can be found in book/UniboSupports.pdf.

5.4 Using the serial line in the actor

In order to 'inject' this new implementation of the Led into our logical component, we define the theory `ledSerialTheory.pl`:

[6] In computer programming, the *proxy* pattern is a design pattern. A proxy, in its most general form, is a class functioning as an interface to something else.

[7] The *Java Simple Serial Connector* (`jscc`) library is required. It is an evolution of `RxTx`, see net/jSSC.

```
/*
===============================================================
ledGuiTheory.pl
Defines the rules to create a serial-proxy Led
===============================================================
*/
portName( "COM5" ).

createLed(Name,Color) :-
    portName( PORTNAME ),
    createSerialLedProxy( Name, Color, PORTNAME ).

createSerialLedProxy( Name, Color, PORTNAME ) :-
    actorobj( Actor ),
    Actor <- getOutputEnvView returns OutView ,
    actorPrintln( createSerialLedProxy( Name, Color, PORTNAME) ),
    class("it.unibo.devices.qa.DeviceLedFactoryQa") <- createLedSerialProxy(Name,OutView,Color,PORTNAME) returns LED.
```

Listing 1.24. `ledSerialTheory.pl`

The next step is to introduce two new rules in the `ledTheory` of Subsection 2.4:

```
ledImplementation( serial ).
...
ledImplementationFile( serial, "ledSerialTheory.pl" ).
```

Finally, we introduce a new Led creation operation in the entry in the `DeviceLedFactoryQa`:

```
    public static DeviceLedImpl createLedSerialProxy(
         String name, IOutputEnvView outEnvView, int color, String portName) throws Exception{
        if( myself == null ){
            LedColor ledcolor = (color == 0) ? LedColor.GREEN : LedColor.RED ;
            myself = new DeviceLedArduinoProxy( name, ledcolor, portName, outEnvView);
        }
        return myself;
    }
```

6 The Led controlled via a Web page

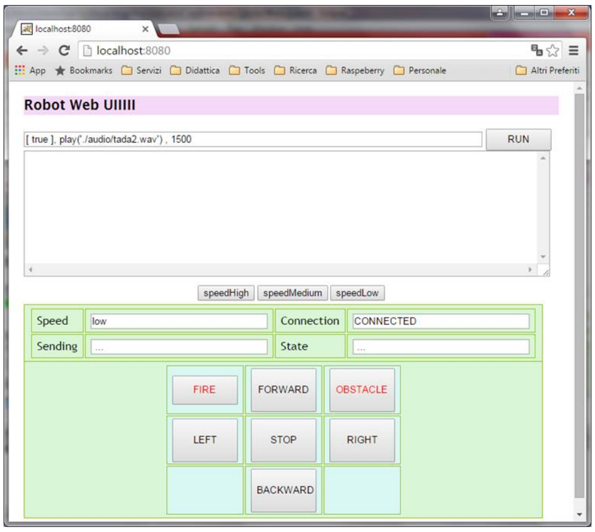
In the `qa` model of the Led, the flag `-httpserver` can be introduced in a Context declaration to specify that it must provide a built-in support for web-based interaction.

```
/*
 * ledMsg.qa
 * This is A MODEL defined during REQUIREMENT or PROBLEM ANALYSIS
 * by using the qa CUSTOM meta-model / language
 */
System ledMsg -regeneratesrc
Dispatch turnLed : turnLed(X)

Context ctxLedMsg ip [host="localhost" port=8010] -g cyan -httpserver
...
```

When the `-httpserver` flag is set, a simple `HTTPserver`[8] is created and started during the Context configuration phase. This server is based on the `WebSocket` technology [9] and answers to `HTTP` requests on port 8080 by returning the web page named `QActorWebUI.html` stored in the context package generated in the `srcMore` directory.

6.1 The default page

The built-in page has the following aspect:



Robot Web UIIIII

[true], play('./audio/tada2.wav') , 1500

Speed	low	Connection	CONNECTED
Sending	...	State	...

The application designer can define (and modify) the content of the `QActorWebUI.html` page in order to interact with a QActor by exploiting a conventional Web Browser.

[8] The class of the server is `it.unibo.qactors.web.QActorHttpServer`)
[9] The *WebSocket* specification defines an `API` establishing "socket" connections between a web browser and a server, i.e. a persistent client-server connection so that both parties can start sending data at any time.

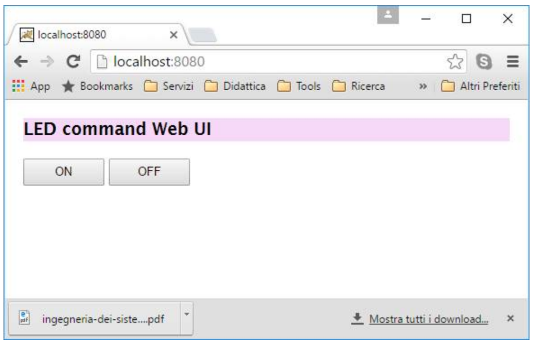
6.2 A HTML page for the Led

Let us define a simple page for Led control:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<link rel="stylesheet" type="text/css" href="QActorWebUI.css">
<script type="text/javascript" src="QActorWebUI.js"></script>
</head>
<body>
<h3 id="h3">LED command Web UI</h3>
<div>
<input style="height: 30px; width: 90px;" type="button"
    id="on" value="ON" onmousedown="send('turnLed(on)')"/>
<input style="height: 30px; width: 90px;" type="button"
    id="off" value="OFF" onmousedown="send('turnLed(off)')"/>
</div>
 </body>
</html>
```

Listing 1.25. QActorWebUI.html for the Led

The page shows itself as follows:



The new page for the Led defines two buttons that, once clicked, invoke the *send* operation defined in the JavaScript file QActorWebUI.js:

```
//QActorWebUI.js
    console.log("QActorWebUI.js : server IP= "+document.location.host);
 /*
 * WEBSOCKET
 */
    var sock = new WebSocket("ws://"+document.location.host, "protocolOne");
    sock.onopen = function (event) {
         //console.log("QActorWebUI.js : I am connected to server.....");
     };
    sock.onmessage = function (event) {
        //console.log("QActorWebUI.js : "+event.data);
       alert( "onmessage " + event);
    }
    sock.onerror = function (error) {
        //console.log('WebSocket Error %O', error);
        alert( "onerror " + error);
    };
    function send(message) {
        var msg = "msg( MSGID, MSGTYPE, SENDER, RECEIVER, CONTENT, 0 )"
        msg = "m-" + msg.replace("MSGID","turnLed").replace("MSGTYPE","dispatch").
            replace("SENDER","web").replace("RECEIVER","ledmsg").replace("CONTENT",message);
        //alert( "send " + msg);
        sock.send(msg);
    };
//  alert("loaded");
```

Listing 1.26. `QActorWebUI.js` for the Led

6.3 The work of the built-in QActorHttpServer.

The `QActorHttpServer` server handles the input string as follows:

```
    protected void handleInputMsg(String msg) throws Exception{
 //         outEnvView.addOutput("    === QActorHttpServer handleUserCmd "+cmd);
            if( msg.startsWith("m-")){
                handleMsg(msg.split("-")[1]);
            }else if( msg.startsWith("i-")){
                hendleInputCommand(msg.split("-")[1]);
            }else{
                handleCmd(msg);
            }
    }
```

Listing 1.27. `QActorHttpServer.java` for the Led

Thus, the server performs different actions according to the input string prefix.

6.4 Input string prefix `m-`

In this case the rest of the input string is assumed to be a `qa` message of the form:

```
msg( MSGID, MSGTYPE, SENDER, RECEIVER, CONTENT, SEQNUM )
```

The `QActorHttpServer` sends the message to the `RECEIVER` actor.
For the default page, there no case of this type, that is instead present in the Led web page.

6.5 Input string prefix `i-`

In this case the rest of the input string is assumed to be a user command (`CMD`) that is translated into a string of the form:

```
usercmd( executeInput( MOVE ) )
```

where `MOVE` can take one of the following forms:

```
[ GUARD ] , ACTION
[ GUARD ] , ACTION , DURATION
[ GUARD ] , ACTION , DURATION , ENDEVENT
[ GUARD ] , ACTION , DURATION , [EVENTLIST], [PLANLIST]
```

For the default page, this is the case of the user-command interface; the `QActorHttpServer` emits the following event:
`usercmd : usercmd(executeInput( CMD ))` (`RUN` button)

6.6 Input string without special prefix

In this case, the string is assumed to be written in Prolog syntax. For the default page, this is the case of the robot-console buttons, that generate input strings of the form:

```
e(alarm(fire))
e(alarm(obstacle))
w( high )
...
```

The `QActorHttpServer` emits the following events:

`usercmd : usercmd(robotgui(MOVE))` with `MOVE=w(low),...,s(high)` (`MOVE` button)
`alarm : alarm(fire)` (`FIRE` button)
`alarm : alarm(obstacle)` (`OBSTACLE` button)

7 The Button

As regards the Button, we will follow a workflow similar to that introduced for the Led in Section 2 :

1. model the basic idea of a Button as a `POJO` component;
2. model the idea of Button required by the problem as an actor;
3. introduce a set of different implementations for the basic button and use them in the proper concrete situation.

7.1 The Button as a `POJO`

In the application domain of the customer, the Button is a physical input device that can be modelled in several ways (a first discussion can be read in |»site/button.pdf).

In this section we start form the idea that a Button is an *observable* (in sense of `GOF` patterns) `POJO`.[10].

The Button is then modelled as an entity with some internal modifiable state that can be inspected by all the registered observers or by using the operation `isPressed`.

```
public interface IButton {
  public String getName();          // property , primitive
  public void ispressed();          // property , primitive
  public void addObserver(IObserver o); // modifier
  public String getDefaultRep();    // mapping , non-primitive
}
```

The argument of the operation `addObserver` must be an 'observer' that implements the following interface:

```
package it.unibo.is.interfaces;
import java.util.Observable;
import java.util.Observer;

public interface IObserver extends Observer {
    public void update(Observable source, Object state); //modifier
}
```

Listing 1.28. `IObserver.java`

The `update` operation is called when the state of the button changes.

The interface `IButton` does not provide any operation to change the internal state of the Button; in fact, the Button should change its state as consequence of some external action. However, it can be wise to introduce some explicit modifier in order to facilitate the testing:

```
public interface IButton {
    ...
    public void high(); //modifier
    public void low(); //modifier
}
```

[10] See |»site/buttonEntry

7.2 The Button model test-plan

As already done for the Led (see Subsection 2.2), we immediately introduce a test-plan related to the `IButton` interface as a way to express in a more formal way *what* each operation should do:

```java
package it.unibo.buttonLedSytem.tests;
import static org.junit.Assert.*;
import it.unibo.bls.lowLevel.interfaces.IButton;
import it.unibo.bls.lowLevel.interfaces.IDeviceButtonImpl;
import it.unibo.buttonLed.components.ButtonMock;
import it.unibo.system.SituatedSysKb;

import org.junit.Before;
import org.junit.Test;
/*
 * This test makes reference to ButtonMock
 * The test unit works like a user that calls the high/low methods
 */


import alice.tuprolog.Prolog;
import alice.tuprolog.Term;

public class TestButton {
private IButton button;
@Before
    public void setUp() {
        try{
            //TODO: replace with a real button or by a call to a factory
            button = new ButtonMock("button1",SituatedSysKb.standardOutEnvView);
        }catch(Exception e){
            fail("setUp");
        }
    }
@Test
    public void testCreation(){
        assertTrue("  testCreation", ! button.isPressed() );
    }
@Test
public void testPress(){
    button.high();
    assertTrue("testPress", button.isPressed() );
    button.high();
    assertTrue("testPress", button.isPressed() );
 }
@Test
public void testRelease(){
    button.low();
    assertTrue("testRelease", ! button.isPressed() );
    button.high();
    assertTrue("testPress", button.isPressed() );
    button.low();
    assertTrue("testPress", ! button.isPressed() );
}
@Test
public void testRepUnify(){
    System.out.println("  testRepUnify ... " );
    String repExpected="sensor("+button.getName() +",false)";
    Term te = Term.createTerm(repExpected);
    Term tr = Term.createTerm(button.getDefaultRep());
    boolean match = new Prolog().unify(tr, te);
    assertTrue("testRepUnify", match );
}
@Test
public void testObserver(){
    ButtonObserverNaive buttonObserver = new ButtonObserverNaive();
    button.addObserver(buttonObserver);
    button.high();
    System.out.println("getCurVal"+ buttonObserver.getCurVal());
    assertTrue("testObserver", buttonObserver.getCurVal().equals(IDeviceButtonImpl.repHigh) );
```

```
   button.low();
   assertTrue("testObserver", buttonObserver.getCurVal().equals(IDeviceButtonImpl.repLow) );
 }
}
```

Listing 1.29. `A Test Plan for the Button`

7.2.1 A first observer.

The observer used in the test-plan simply stores the current state of the button:

```
package it.unibo.buttonLedSytem.tests;
import java.util.Observable;
import it.unibo.is.interfaces.IObserver;

public class ButtonObserverNaive implements IObserver{
private String mirror = null;
   @Override
   public void update(Observable source, Object state) {
       System.out.println("ButtonObserverNaive update " + state);
       mirror = (String) state;
   }
   public String getCurVal(){
      return mirror;
   }
}
```

Listing 1.30. `ButtonObserverNaive`

7.3 The Button as an actor (problem analysis / project)

The current model of the Button as a `POJO` does not include any capability to interact via the network with the other components of the system. In the context of a distributed ButtonLed system, the Button could be conceived as a more advanced device, modelled as an *actor* able to:

- emit *events*. In this way any actor can sense or react to state changes of the button ;
- send *messages* to some other actor, that can be:
  - statically known by the button actor. For example, this is the case of a simple ButtonLed distributed system in which the Button knows the actor Led that must be turned on/off;
  - dynamically acquired by the button actor through some 'registration' message. This is the 'distributed version' of the `GOF` observer pattern, in which the observable (the Button) does not known a-priori its possible 'remote observers'.

As already done for the Led (see Subsection 2.3) the Button as a `POJO` introduced in Subsection 7.1 is not given up; it is still necessary and must be used as a basic support for the behaviour of the Button as an actor.

7.4 Fron the Button `POJO` to the Button actor

More precisely, the Button-`POJO` becomes the low-level (technology-dependent) part of our abstract (technology-independent) concept of Button actor. The 'bridge' between the low-level layer and the logical layer can be delegated to a Button-`POJO` observer (a *ButtonObserverForActors* - shortly `BOA` - like that introduced in Subsection 7.7) that can implement a set of different actions:

1. the `BOA` maps a button state change into a `qa` event. In this case the Button actor works as an active observer of the events emitted by the underlying level;

2. the BOA calls a 'callback' defined in the actor as a Plan. In this case the Button actor logically becomes a passive observer that works as a high-level extension of the underlying BOA[11]. The 'callback' Plan does include the 'business logic' related to a change of the button state (e.g. it could send a message to other actors declared in the system model);
3. the BOA immediately performs the 'business logic' by executing some 'script' of the application level. This 'script' could be a rule written in the *WorldTheory* of the Button actor that logically becomes a 'do-nothing' entity.

The last BOA strategy is mentioned here as one as a behaviour 'technically' possible but certainly not advisable, since the behaviour of the Button component cannot be understood at system level.

This problem does not occur with the second BOA strategy that does minimize computational overheads like the third one. But in this case we must avoid the risk to 'break the model' by mixing the event/message-based nature of the behaviour of an actor with something that looks like as event-driven. This mixing is quite harmful, since we could reintroduce all the problems of concurrent access to shared memory by independent processes that are actually excluded by the event/message-based behaviour of the actors.

The first BOA strategy is the most appropriate at conceptual level, since an actor is an active machine working as a event/message-based finite state machine. However it does introduce computational overhead since the system must generate a qa event and after resume a quiescent actor.

In the following we will show examples of all these kinds of behaviour by introducing rules that assure that, in cases of a BOA strategies 2 and 3 the Button actor immediately ends logically 'becoming' a conventional object.

7.5 The Button as a message sender

Let us introduce in this section a qa model of the Button that (once 'pressed') sends a message to a statically known actor. A model for a 'remotely observable' Button actor will be discussed in Section 11.

```
/*
 * buttonObsMsg.qa
 * This is A MODEL written in the qa CUSTOM meta-model/language
 */
System  buttonObsMsg -regeneratesrc
Event   clicked : clicked(V) //emitted by the Button (implementation) device
Dispatch button : button(X)  //sent by the button actor to some destination

Context ctxButtonObsMsg ip [host="localhost" port=8010] -g cyan
//EventHandler evh for clicked -print ;

//Context ctxRemoteButtonObsMsg ip [host="192.168.43.229" port=8030] -g cyan
//EventHandler evh for clicked -print ;

QActor buttonobsmsg context ctxButtonObsMsg {
   Rules{
/*
Behavior configuration rules.
 */
       //actorPerceivesEvents.  //the actor works as an event-based machine
       //sendMessageImmediately. //the message is sent by the buttonTheory
       //NO configuration fact set => the buttonTheory calls entryAsObserver
/*
* Defines the button receiver and the details on the message to be sent
```

[11] We could also conceive the Button Actor as an event-driven machine rather than an event-based machine, since a Plan transition occurs without any explicit control.

```
* Used when sendMessageImmediately is eet
*/
      templateMsgToSend( buttontester,dispatch,button,button(V) ).

/*
* Defines the plan to be called when NO configuration is set
* Used when sendMessageImmediately is eet
*/
      callback(  entryAsObserver ).

   }
   Plan init normal
      println("buttonobsmsg STARTS" ) ;
      switchToPlan configure  ;
      [!? actorPerceivesEvents] switchToPlan senseButtonEvents;
      println("buttonobsmsg ENDS" )
   Plan configure resumeLastPlan
      solve consult("buttonTheory.pl") time(0) onFailSwitchTo prologFailure ;
      solve createButton( "click", clicked, clicked(V) ) time(0) onFailSwitchTo prologFailure
   Plan senseButtonEvents
```

Listing 1.31. `buttonObsMsg.qa`

The actor: *i)* loads an application-specific theory (`buttonTheory`), *ii)* creates a (`POJO`) Button (as an observable entity), and then *iii)* works according to the behaviour configuration rules written in the *Rules* section. More precisely:

1. `actorPerceivesEvents` rule: if set, the button works as an *event-based* state machine that sends a command message to its remote partner when a button event is perceived (Plan `senseButtonEvents`). This behaviour requires a `BOA` that implements the strategy `1` of Subsection 7.4.
2. no configuration rule set: the button actor `ENDS`.
   This behaviour requires a `BOA` that implements the strategy `2` of Subsection 7.4 by calling a rule of the `buttonTheory` that 'learns' from the fact `callback/1` defined in the *Rules* section the name of the Plan to be called each time the underlying `POJO` button changes its state.
3. `sendMessageImmediately` rule: if set, the button actor `ENDS`. It continues to works in a procedure-call based way with respect to the embedded `GOF` observable `POJO` button.
   This behaviour requires a `BOA` that implements the strategy `3` of Subsection 7.4 by calling a rule of the `buttonTheory` that 'learns' from the fact `templateMsgToSend/3` defined in the *Rules* section the details (type, destination, message-identifier, message-content structure) about the message to send.

The actor `buttontester` is an actor that we can add to the model to test the behaviour of our button:

```
      println("buttonobsmsg WAITS for an event" );
      sense time(300000) clicked -> continue;
      onEvent clicked : clicked(V) -> forward buttontester -m button : button(V) ;
      repeatPlan 0
   //switchToPlan entryAsObserver can be called by the button POJO observer
   Plan entryAsObserver
      println("buttonobsmsg entryAsObserver" );
      [ ?? buttonValue(V)] forward buttontester -m button : button(V)
   Plan prologFailure resumeLastPlan
      println("buttonobsmsg has failed to solve a Prolog goal" )
}

/*
 * A actor that receives the messages sent from the button
 */
QActor buttontester context ctxButtonObsMsg { //ctxRemoteButtonObsMsg
   Plan init normal
      println("buttontester WAITS" );
      receiveTheMsg m( button , dispatch, SENDER, buttontester, button(V) , N ) time(300000);
      [ !? tout(X,Y) ] switchToPlan toutExpired ;
      onMsg button : button(V) -> println( buttontester(button(V)) ) ;
      repeatPlan 0
   Plan toutExpired
      [ ?? tout(X,Y) ] println( timeout(X,Y) )
}
```

Listing 1.32. `buttonObsMsg.qa: the buttontester actor`

7.6 The `buttonTheory`

Besides the knowledge about the implementation of the button (as done for the Led in Subsection 2.4), the `buttonTheory` defines also a rule (`handleButtonInput/1`) that implements the actor behaviour policy specified by the configuration rules written in the model. In this way, the `BOA` strategy (see Subsection 7.4) in not implemented directly in the `Java BOA` code, but is written as 'script' in the actor theory. Thus, this `buttonTheory` can be viewed as a behavioural extension of both the of the `buttonObsMsg` model and of the `Java` button observer of Subsection 7.7.

```
/*
===========================================================================
buttonTheory.pl
Defines the rules to load a specific theory for each Button implementation
===========================================================================
*/
/*
handleButtonInput is called by a button observer (ButtonObserverForActors)
*/
%% ---------------------------------------------------------------
%% If actorPerceivesEvents set, the actor works in event-based way
%% We do not nothing here
%% ---------------------------------------------------------------
handleButtonInput( V ):-
   actorPerceivesEvents, !.
%% ---------------------------------------------------------------
%% If sendMessageImmediately set, we send a message as specified in templateMsgToSend/4
%% If no rule is set, we switch to the plan named entryAsObserver
%% ---------------------------------------------------------------

handleButtonInput( V ):-
   actorobj( Actor ),
   %% actorPrintln( handleButtonInput( Actor,RECEIVER,MSGTYPE,MSGID,V ) ),
   ( sendMessageImmediately, !, sendMsg(Actor,V) ;
     executeObserverPlan(Actor,V)
   ).
executeObserverPlan(Actor,V):-
   assert( buttonValue( V ) ),
   callback( CBACK) ,
   actorPrintln( entryAsObserver( Actor,V,CBACK) ),
   Actor <- CBACK returns Bool.
sendMsg(Actor, V ):-
   templateMsgToSend(RECEIVER,MSGTYPE,MSGID,PAYLOAD),  %%defined at model level
   bind( PAYLOAD, V , CONTENT ),
   Actor <- sendMsg( MSGID, RECEIVER, MSGTYPE, CONTENT ).

%% 'injects' the value V in the structure Term with arity 1
bind( Term , V, R ):-
   functor(Term, Functor, 1),
   R =.. [ Functor, V ].

/*
* -----------------------------------------------------
* For each implementation get the proper theory
* -----------------------------------------------------
*/
```

Listing 1.33. buttonTheory.pl

7.7 The ButtonObserverForActors

The handleButtonInput/1 rule is called by a Java button observer defined as follows:

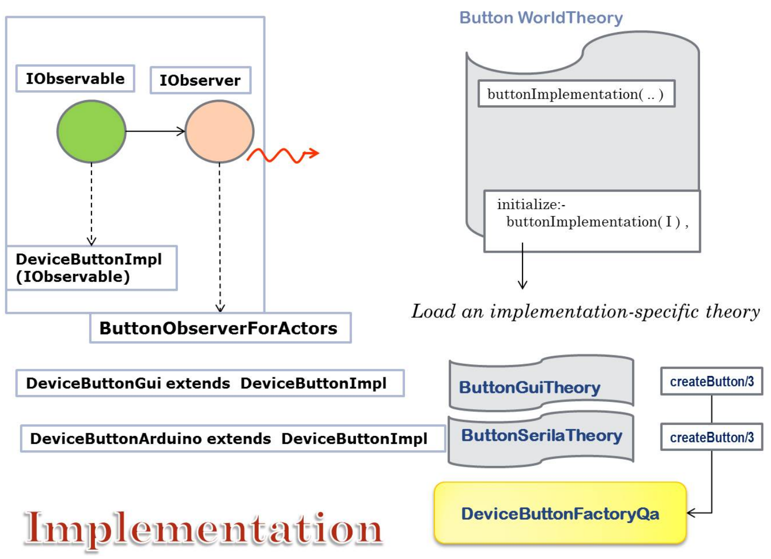
```java
package it.unibo.devices.qa;
import java.util.Observable;
import alice.tuprolog.Prolog;
import alice.tuprolog.SolveInfo;
import alice.tuprolog.Term;
import it.unibo.contactEvent.interfaces.IContactEventPlatform;
import it.unibo.contactEvent.platform.ContactEventPlatform;
import it.unibo.is.interfaces.IObserver;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.qactors.QActor;
import it.unibo.system.SituatedPlainObject;
/*
 * An Observer that calls the prolog rule handleButtonInput(V) and emits eventId:eventMsg(V)
 */
public class ButtonObserverForActors extends SituatedPlainObject implements IObserver{
 protected IContactEventPlatform platform ;
 protected String eventId = "";
 protected String eventMsg = "";
 protected QActor actor;
 protected Prolog pengine;
 protected int clickCount = 1;
    public ButtonObserverForActors(String name, IOutputEnvView outEnvView, QActor actor, String eventId, String
          eventMsg){
        super( name, outEnvView );
        this.actor   = actor;
        this.eventId = eventId;
        this.eventMsg = eventMsg;
        pengine      = actor.getPrologEngine();
        try { platform = ContactEventPlatform.getPlatform();
        } catch (Exception e) { println("ButtonObserverForActors WARNING: DeviceButtonGuiQa NO PLATFORM");}
    }
    @Override
    public void update(Observable source, Object cmd) {
        String input = ""+cmd;
        emitTheEvent(input);
        delegateHandling(input);
    }
    protected void delegateHandling(String cmd) {
        try {
            String handleGoal = "handleButtonInput(" + cmd + "_" + clickCount++ + ").";
            pengine.solve(handleGoal);
        } catch (Exception e) { println("ButtonObserverForActors WARNING: handleButtonInput does not work ");}
    }
    protected void emitTheEvent(String cmd) {
        if( eventId == null || eventId.length()==0 ) return;
        try { String evMsg = eventMsg ;
              if( ! Term.createTerm(eventMsg).isAtom() ){
                   String g1 = "bind("+eventMsg+","+cmd+",R).";
                   SolveInfo sol = pengine.solve(g1);
                   evMsg = ""+sol.getVarValue("R") ;
                   //println("ButtonObserverForActors emitTheEvent: " + evMsg );
              }
              platform.raiseEvent(this.getName(), eventId, evMsg );
        } catch (Exception e) { println("ButtonObserverForActors WARNING: emitTheEvent does not work ");}
    }
 }
```

Listing 1.34. ButtonObserverForActors.java

Note that the operation *emitTheEvent* of `ButtonObserverForActors` emits the event given at construction time with a content obtained by 'injecting' (via the `bind/3` rule) in the given `eventMsg` structure the value of the button `cmd`.

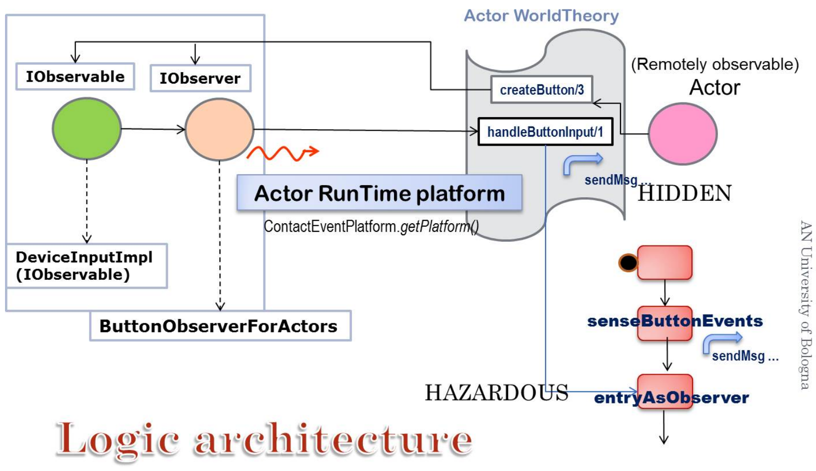
This observer should be the same for all the different implementations of the Button.

Let us report here a picture that show the implementation layer:



Button WorldTheory
IObservable
IObserver
buttonImplementation(..)
initialize:-
buttonImplementation(I) ,
Load an implementation-specific theory
DeviceButtonImpl (IObservable)
ButtonObserverForActors
DeviceButtonGui extends DeviceButtonImpl
ButtonGuiTheory
createButton/3
DeviceButtonArduino extends DeviceButtonImpl
ButtonSerilaTheory
createButton/3
DeviceButtonFactoryQa
Implementation

The following picture shows instead the relationship between the implementation layer and the model layer:



8 A Button factory

The Java class `DeviceButtonFactoryQa` is a *factory* that provides static methods to create a *singleton* Button object and to get a reference to such an object:

```java
package it.unibo.devices.qa;
import it.unibo.bls.raspberry.components.DeviceButtonPi4J;
import it.unibo.buttonLed.components.ButtonMock;
import it.unibo.buttonLed.components.DeviceButtonImpl;
import it.unibo.buttonLedSystem.gui.DeviceButtonGui;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.qactors.QActor;

public class DeviceButtonFactoryQa{
protected static DeviceButtonImpl myself = null;
/*
 * A singleton is not required, since we never need to acces the POJO Button
 * In fact, the POJO Button is a SOURCE of information
 */

    public static DeviceButtonImpl getTheButton( ){
        return myself;
    }

    public static DeviceButtonImpl createButtonMock(
            String name, IOutputEnvView outEnvView ) throws Exception{
        if( myself == null ){
            myself = new ButtonMock( name,outEnvView );
        }
        return myself;
```

Listing 1.35. `DeviceButtonFactoryQa.java : createLedMock`

Note that any creation method `createXXX` returns an object of the class `DeviceButtonImpl` (see Subsection 8.1).

8.1 A basic class for Button implementation

The `DeviceButtonImpl` class is based on the custom framework |»site/uniboEnv) introduced for the rapid development of `GUI`-based prototypes.

```java
package it.unibo.buttonLed.components;
import java.util.Observable;
import it.unibo.bls.lowLevel.interfaces.IButton;
import it.unibo.bls.lowLevel.interfaces.IDeviceButtonImpl;
import it.unibo.is.interfaces.IActivityBase;
import it.unibo.is.interfaces.IOutputView;
import it.unibo.system.SituatedPlainObject;
/*
 * A SituatedPlainObject that defines a common behavior for all button implementations
 */
public class DeviceButtonImpl extends SituatedPlainObject implements IButton,IActivityBase {
protected String input = null;
protected boolean buttonPressed = false;

    public DeviceButtonImpl(String name, IOutputView outView ) {
        super("button("+name+")",outView);
    }
    /*
     * This is the entry point for a "polling interaction"
     */
    @Override
    public int getInput() throws Exception { return input.length(); }
    /*
     * Entry point after a button-click (button implements IActivityBase)
     */
    @Override
    public void execAction(String cmd) {
//      println( "DevButtonImpl execAction:" + cmd );
        this.setChanged(); //!!!!
        this.notifyObservers( cmd );
    }
    /*
     * This update method is used by some class (e.g. DevButtonStdin) that generates a boolean
     */
    public void update( boolean v){
        input = v ? IDeviceButtonImpl.repHigh : IDeviceButtonImpl.repLow ;
        execAction( input );
    }
    /*
     * This update method is used by some class (e.g.Arduino?) that generates a string 'LOW' or 'HIGH'
     */
    @Override
    public void update(Observable source, Object value) {
        String vs = ""+value;
        if( vs.equals( IDeviceButtonImpl.repHigh ) || vs.equals(IDeviceButtonImpl.repLow) ){
            execAction( ""+vs.equals( IDeviceButtonImpl.repHigh ) );
        }else{
            execAction( vs );
        }
    }
    @Override
    public String getDefaultRep() {
        String s = "sensor("+this.name+","+ buttonPressed +")";
        return s;
    }
    @Override
    public String getName() {
        return name;
    }
    @Override
    public boolean isPressed() {
        return buttonPressed ;
    }
    /*
     * API for debugging
     */
    @Override
    public void high() {
        buttonPressed = true;
        execAction(IDeviceButtonImpl.repHigh);
    }
    @Override
    public void low() {
        buttonPressed = false;
        execAction(IDeviceButtonImpl.repLow);
    }
/*
 * Remove all the observers
 */
    protected void terminate(){
         this.deleteObservers();
    }
 }
```

Listing 1.36. DeviceButtonImpl.java

8.2 A Virtual Button (implementation phase)

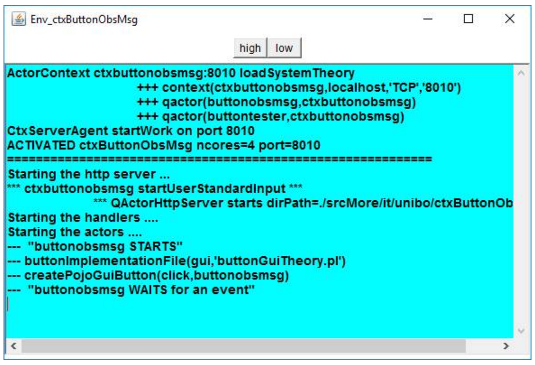
A Button as a virtual device can now be introduced as follows:

```java
package it.unibo.buttonLedSystem.gui;
```

```java
import java.awt.Color;
import java.awt.Panel;
import it.unibo.baseEnv.basicFrame.EnvFrame;
import it.unibo.buttonLed.components.DeviceButtonImpl;
import it.unibo.buttonLedSystem.gui.interfaces.IDeviceButtonGui;
import it.unibo.is.interfaces.IBasicEnvAwt;
import it.unibo.is.interfaces.IOutputEnvView;

public class DeviceButtonGui extends DeviceButtonImpl implements IDeviceButtonGui {
protected IBasicEnvAwt env;
protected String[] cmd;
protected Panel cmdPanel;
    public DeviceButtonGui(String name, IOutputEnvView outEnvView, String[] cmd) {
        super( name, outEnvView);
        this.env = outEnvView.getEnv();
        this.cmd = cmd;
        configure();
    }
    protected void configure() {
        cmdPanel = env.addCmdPanel("", cmd, this);
    }
    @Override
    public Panel getPanel(){
        return cmdPanel;
    }
    /*
     * ---------------------------------------
     * Main (rapid check)
     * ---------------------------------------
     */
    public static void main(String args[]) throws Exception {
        IBasicEnvAwt env = new EnvFrame("DevButtonGui", Color.white, Color.BLACK);
        env.init();
//          new DeviceButtonGui("b0", env.getOutputEnvView() , new String[] { repLow, repHigh });
        new DeviceButtonGui("b0", env.getOutputEnvView() , new String[] { eventName });
    }

}
```

Listing 1.37. `DeviceButtonGui.java`

The result is shown in the following picture:



```
ActorContext ctxbuttonobsmsg:8010 loadSystemTheory
                    +++ context(ctxbuttonobsmsg,localhost,'TCP','8010')
                    +++ qactor(buttonobsmsg,ctxbuttonobsmsg)
                    +++ qactor(buttontester,ctxbuttonobsmsg)
CtxServerAgent startWork on port 8010
ACTIVATED ctxButtonObsMsg ncores=4 port=8010
==========================================================
Starting the http server ...
*** ctxbuttonobsmsg startUserStandardInput ***
             *** QActorHttpServer starts dirPath=./srcMore/it/unibo/ctxButtonOb
Starting the handlers ....
Starting the actors ....
---  "buttonobsmsg STARTS"
--- buttonImplementationFile(gui,'buttonGuiTheory.pl')
--- createPojoGuiButton(click,buttonobsmsg)
---  "buttonobsmsg WAITS for an event"
```

In order to 'inject' this new implementation of the Button into our logical component, we define the theory `buttonGuiTheory` with a specific `createButton/3` rule:

```
/*
===============================================================
buttonGuiTheory.pl
Defines the rules to create and use a GUI-based virtual Button
===============================================================
*/

createButton( Name, Event, EventMsg ) :-
    actorobj( Actor ),
    Actor <- getOutputEnvView returns OutView ,
    actorPrintln( createButton( Name,Actor, Event,EventMsg ) ),
    class("it.unibo.devices.qa.DeviceButtonFactoryQa") <-
            createButtonGui( Name, OutView , Actor, Event,EventMsg ) returns Button.
```

Listing 1.38. `buttonGuiTheory.pl`

The next step is to introduce two new rules in the `buttonTheory` of Subsection 7.6:

```
buttonImplementation( gui ).
...
buttonImplementationFile( gui, "buttonGuiTheory.pl" ).
```

Finally, we introduce a new button creation operation in the entry in the `DeviceButtonFactoryQa`:

```

    public static DeviceButtonImpl createButtonGui(
             String name, IOutputEnvView outEnvView, QActor actor, String eventId, String eventMsg ) throws Exception{
//      if( myself == null ){
            // outEnvView, ActorContext ctx, String eventId, String eventMsg
            myself = new DeviceButtonGui( name,outEnvView, new String[]{name} );
            myself.addObserver( new ButtonObserverForActors( name+"obs", outEnvView, actor, eventId, eventMsg) );
//          }
        return myself;
    }
```

Listing 1.39. `DeviceButtonFactoryQa.java : createButtonGui`

9 A Button on Arduino (implementation phase)

An input device such as the Button can be managed in Arduino in two ways: by polling or via interrupt.

9.1 Managing a Button by polling.

```
/*
=======================================
 buttonPolling.ino
 project it.unibo.arduino.intro
 Input from Pin 3 (Button)
 ARDUINO UNO
 =======================================
 Arduino has internal PULLUP resistors to tie the input high.
 Hook one end of the switch to the input (5V) and the other end to ground:
 when we PUSH the switch the input goes LOW.
 */

int pinButton = 3;
int pinLed   = 13; //internal led

void initLed(){
   pinMode(pinLed, OUTPUT);
}
void initButton(){
  pinMode(pinButton, INPUT);
  digitalWrite(pinButton, HIGH); //set PULLUP : PUSH=>0
}

void setup(){
  Serial.begin(9600);
  Serial.println( "----------------------------------------" );
  Serial.println( "project it.unibo.arduino.intro" );
  Serial.println( "buttonPolling/buttonPolling.ino" );
  Serial.println( "----------------------------------------" );
  initButton();
  initLed();
}

/*
 * When the button is PUSHED, the led is ON
 */
void loop(){
  int v = digitalRead( pinButton );
  digitalWrite( pinLed, ! v );
  delay(100);
}
```

Listing 1.40. `buttonPolling.ino`

9.2 Managing a Button via interrupt.

```
/*
=======================================
 buttonInterrupt.ino
 project it.unibo.arduino.intro
 Input from Pin 3 (Button)
 ARDUINO UNO
    pin 3 maps to interrupt 1,
    pin 2 is interrupt 0,
 ======================================
 Arduino has internal PULLUP resistors to tie the input high.
 Hook one end of the switch to the input (5V) and the other end to ground:
 when we PUSH the switch the input goes LOW.
 */

int pinButton = 3;
int pinLed   = 13;

void initLed(){
   pinMode(pinLed, OUTPUT);
}
void initButton(){
  pinMode(pinButton, INPUT);
  digitalWrite(pinButton, HIGH); //set PULLUP : PUSH=>0
  attachInterrupt(1, buttonInterruptHandler, CHANGE); // RISING CHANGE FOLLING LOW
}
/*
-----------------------------
INTERRUPT HANDLER function
-----------------------------
*/
void buttonInterruptHandler(){
  boolean ok = debouncing();
  if( ok ){
    int v = digitalRead(pinButton);
    Serial.println( "interrupt v=" + String(v));
    digitalWrite( pinLed, !v );
   }
}
boolean debouncing(){
static unsigned long lastInterruptTime = 0;
static int lastVal = 0;
unsigned long interruptTime = millis();
int v = digitalRead(pinButton);
  if( (interruptTime - lastInterruptTime) > 100 ){
    if( (v != lastVal) ) lastVal = v;
    return true;
  }
  lastInterruptTime = interruptTime;
  return false;
}
void setup(){
  Serial.begin(9600);
  Serial.println( "----------------------------------------" );
  Serial.println( "project it.unibo.arduino.intro" );
  Serial.println( "buttonInterrupt/buttonInterrupt.ino" );
  Serial.println( "----------------------------------------" );
  initButton();
  initLed();
}
void loop(){
  //do nothing
}
```

Listing 1.41. `buttonInterrupt.ino`

9.3 Managing a Button for qa message-interaction

```
/*
======================================
 button3Msg.ino
 project it.unibo.arduino.intro
 Input from Pin 3 (Button)
 ARDUINO UNO
```

```
   pin 3 maps to interrupt 1,
   pin 2 is interrupt 0,
 =====================================
 Arduino has internal PULLUP resistors to tie the input high.
 Hook one end of the switch to the input (5V) and the other end to ground:
 when we PUSH the switch the input goes LOW.
 */
 /*
-----------------------------
 SUPPORT
-----------------------------
*/
boolean debouncing(int pinButton){
static unsigned long lastInterruptTime = 0;
static int lastVal = 0;
unsigned long interruptTime = millis();
int v = digitalRead(pinButton);
  if( (interruptTime - lastInterruptTime) > 100 ){
    if( (v != lastVal) ) lastVal = v;
    return true;
  }
  lastInterruptTime = interruptTime;
  return false;
}

void setup(){
  Serial.begin(9600);
  Serial.println( "--------------------------------------" );
  Serial.println( "project it.unibo.arduino.intro" );
  Serial.println( "button3Msg/button3Msg.ino" );
  Serial.println( "--------------------------------------" );
  initButton();
  initLed();
}
void loop(){
  //do nothing
}
/*
-----------------------------
 * APPLICATION
-----------------------------
 */
int pinButton = 3;
int pinLed   = 13;
String lastMsgSent="";

void initLed(){
   pinMode(pinLed, OUTPUT);
}
void initButton(){
  pinMode(pinButton, INPUT);
  digitalWrite(pinButton, HIGH); //set PULLUP : PUSH=>0
  attachInterrupt(1, buttonInterruptHandler, CHANGE); // RISING CHANGE FOLLING LOW
}
/*
INTERRUPT HANDLER function
*/
void buttonInterruptHandler(){
  boolean ok = debouncing(pinButton);
  if( ok ){
    int v = digitalRead(pinButton);
    //Serial.println( "interrupt v=" + String(v));
    digitalWrite( pinLed, !v );
    forwardTheButtonState( !v );
   }
}
void forwardTheButtonState( boolean v){
       /*
      * Send the dispatcj
      * msg( button, dispatch, arduino, connectedpc, buton(V),0 )
```

```
     */
     String msgOut = "msg( button, dispatch, arduino, connectedpc, ";
     msgOut = msgOut +"button("+String( v ) + "),0)" ;
     if( msgOut != lastMsgSent ){
        Serial.println(msgOut);
        lastMsgSent = msgOut;
     }
}
```

Listing 1.42. `button3Msg.ino`

9.4 DeviceButtonArduinoProxy

The class `DeviceButtonArduinoProxy` creates an object that converts the observed messages coming from an `Arduino` port into `qa` *events* according to the arguments given at creation time:

```
package it.unibo.devices.qa;
import jssc.SerialPort;
import jssc.SerialPortEvent;
import jssc.SerialPortEventListener;
import alice.tuprolog.Struct;
import alice.tuprolog.Term;
import it.unibo.buttonLed.components.DeviceButtonImpl;
import it.unibo.is.interfaces.IOutputView;
import it.unibo.is.interfaces.protocols.IConnInteraction;
import it.unibo.supports.FactoryProtocol;

/*
 * This class implements a button that handles Arduino events
 * on the serial port by looking at messages of the form
 *   msg( MSGID, MSGTYPE, SENDER, RECEIVER, CONTENT, SEQNUM )
 * with CONTENT = button( V )
 */
public class DeviceButtonArduinoProxy extends DeviceButtonImpl implements SerialPortEventListener {
protected String PORT_NAME ;
protected IConnInteraction portConn; //the comm channel with Arduino is a "general" two-way IConnInteraction
protected SerialPort serialPort;
protected int n = 0;
protected FactoryProtocol factoryProtocol;

    public DeviceButtonArduinoProxy(String name, IOutputView outView, String PORT_NAME) {
        super(name, outView);
        this.PORT_NAME = PORT_NAME;
        factoryProtocol = new FactoryProtocol(outView,"SERIAL","deviceButtonArduino");
        configure();
    }
    protected void configure() {
        try {
            /* Create a SerialPortSupport and connect */
            portConn = factoryProtocol.createSerialProtocolSupport(PORT_NAME,this,true); //this is the
                SerialPortEventListener
        } catch (Exception e) { e.printStackTrace(); }
    }
    /*
     * A portEvent event is generated each time a line is written by Arduino
     */
    @Override
    public synchronized void serialEvent(SerialPortEvent portEvent) {
        String input="noInput";
          //println("DevButtonArduino serialEvent event type=" + oEvent.getEventType() );
          try {
             input = portConn.receiveALine();
             if( input == null || input.length() == 0 ) return;
             //println("DevButtonArduino input=" + input );
             if( input.startsWith("msg")){
                handleMsgContent(input);
             }
        } catch (Exception e) {
            println("DevButtonArduino WARNING:"+e.getMessage() );
        }
    }

    protected void handleMsgContent(String input){
        try {
            //HANDLE MSG WRITTEN IN PROLOG SYNTAX
            //msg( MSGID, MSGTYPE, SENDER, RECEIVER, CONTENT, SEQNUM )
            Struct msg = (Struct) Term.createTerm(input);
//          println("DevButtonArduino msg=" + msg );
            Struct stateStruct = (Struct) msg.getArg(4);
            String state      = stateStruct.getArg(0).toString();
            this.update( state.equals("1") );
        } catch (Exception e) {
            println("DevButtonArduino handleMsgContent ERROR:"+e.getMessage());
        }
    }

    protected void close() {
        try {
            portConn.closeConnection();
        } catch (Exception e) {
            println("DevButtonArduino close ERROR:"+e.getMessage());
        }
    }
  }
```

Listing 1.43. `DeviceButtonArduinoProxy.java`

9.5 Using the serial line in the actor

In order to 'inject' this new implementation of the Button into our logical component, we define the theory `buttonSerialTheory` with a specific `createButton/3` rule:

```
/*
===============================================================
buttonSerialTheory.pl
Defines the rules to create and use a serial-proxy Button
===============================================================
*/
portName( "COM5" ).

createButton( Name, Event, EventMsg ) :-
    actorobj( Actor ),
    portName( Port ),
    Actor <- getOutputEnvView returns OutView ,
    actorPrintln( createButtonSerialProxy( Name,Actor ) ),
    class("it.unibo.devices.qa.DeviceButtonFactoryQa") <-
            createButtonSerialProxy( Name, OutView ,Port, Actor, Event,EventMsg ) returns Button.
```

Listing 1.44. `buttonGuiTheory.pl`

The next step is to introduce two new rules in the `buttonTheory` of Subsection 7.6:

```
buttonImplementation( arduino ).
...
buttonImplementationFile( arduino, "buttonSerialTheory.pl" ).
```

Finally, we introduce a new button creation operation in the entry in the `DeviceButtonFactoryQa`:

```
    public static DeviceButtonImpl createButtonSerialProxy(
         String name, IOutputEnvView outEnvView, String portName, QActor actor, String eventId, String eventMsg)
              throws Exception{
//          if( myself == null ){
```

```
            myself = new DeviceButtonArduinoProxy( name, outEnvView, portName);
            myself.addObserver( new ButtonObserverForActors( name+"obs", outEnvView, actor, eventId, eventMsg) );
//      }
        return myself;
   }
```
Listing 1.45. `DeviceButtonFactoryQa.java : createButtonSerialProxy`

10 A Button on Raspberry (implementation phase)

In this example, one terminal of the Button is connected to the `5V` pin (physical pin 2) while the other one (button-input pin) to pin `24` in `BCM` code. A resistor of `10K` ohm[12]) is also inserted between the input pin and a `GND` pin (e.g. physical pin 9) to assure a connection to the ground when the button is not pressed.

10.1 Button control using files

The basic way provided by Linux to manage a device connected on a `GPIO` pin is reading/writing some (virtual) file associated with that pin.

```
# -------------------------------------------------------------
# buttonOn24Click.sh
# Key-point: we can manage a device connected on a GPIO pin by
# reading/writing some (virtual) file associated with that pin.
#   sudo bash buttonOn24Click.sh
# -------------------------------------------------------------

echo Unexporting.
echo 24 > /sys/class/gpio/unexport #
echo 24 > /sys/class/gpio/export #
cd /sys/class/gpio/gpio24 #
echo Setting direction to in.
echo in > direction #
echo Reading pin.
cat value
#schmod +x ledOnOff.sh #
```

Listing 1.46. `buttonOn24Click.sh`

10.2 Button control using the `GPIO` shell library

The `gpio` library allows us to write a program (bash file) that reads the button-input pin and writes a `0/1` value on the Led pin:

```
#!/bin/bash
# WARNING: document class text UNIX
# --------------------------------------------------------------------------
# button24Gpio.sh
# Key-point: we can manage a device connected on a GPIO pin by using the GPIO shell library.
# The pin 24 (button) is physical 18 and Wpi 5.
# The pin 25 (led)  is physical 22 and Wpi 6.
#   sudo bash button24Gpio.sh
# --------------------------------------------------------------------------
But=5
Led=6
LedBCM=25

gpio mode $But in
gpio -g mode $LedBCM out
gpio readall
echo "start"

while true
do
    gpio read $But > vgpio24.txt
    V1=`cat vgpio24.txt`
    if [ $V1 == "1" ] ; then echo "on" ; fi
```

[12] 10K colors: brown, black, orange and argent/gold

```
    #echo $V1
    gpio write $Led $V1 #command the led
    sleep 0.1
done
```

Listing 1.47. `button24Gpio.sh`

10.3 Button control using python

The Python library that handles interfacing with the `GPIO` pins allows us to write:

```
# -------------------------------------------------------------
# buttonPython24.py
# Key-point: manage in python a Button connected on the GPIO pin 24
#  using polling.
#   sudo python buttonPython24.py
# -------------------------------------------------------------
import RPi.GPIO as GPIO
import time

'''
----------------------------------
CONFIGURATION
----------------------------------
'''
GPIO.setmode(GPIO.BCM) #BCM or BOARD
GPIO.setup(24, GPIO.IN, pull_up_down=GPIO.PUD_DOWN )


'''
----------------------------------
main activity
----------------------------------
'''
while True:
    if GPIO.input(24):
        print('Button input HIGH')
    else:
        print('Button input LOW')
    time.sleep(1)  #wait 1sec

#while GPIO.input(24) == GPIO.LOW:
#     time.sleep(0.01) #wait 10msec
```

Listing 1.48. `buttonPython24.py`

10.4 Button control using Pi4j

The `Pi4J` library (see |»site/Pi4j) can be used as our *technology assumption* for the implementation a Java class for the control of a Button connected to some pin of a Raspberry Pi:

```
package it.unibo.bls.raspberry.components;
import it.unibo.buttonLed.components.DeviceButtonImpl;
import it.unibo.gpio.base.GpioOnPi4j;
import it.unibo.is.interfaces.IOutputView;
import it.unibo.system.SituatedSysKb;
import com.pi4j.io.gpio.GpioPinDigitalInput;
import com.pi4j.io.gpio.PinPullResistance;
import com.pi4j.io.gpio.PinState;
import com.pi4j.io.gpio.event.GpioPinDigitalStateChangeEvent;
import com.pi4j.io.gpio.event.GpioPinListenerDigital;

/*
```

```
 * ===========================================================================
 * The button is implemented as an observable by using the library pi4j
 * ===========================================================================
 */
public class DeviceButtonPi4J extends DeviceButtonImpl {
protected GpioPinDigitalInput device;
    public DeviceButtonPi4J(String name, IOutputView outView , int pinNum ) {
        super(name,outView);
        com.pi4j.io.gpio.Pin gpioPinNum = GpioOnPi4j.getPin(pinNum);
        println("DevButtonGpioPi4J pinNum=" + pinNum + " gpioPinNum=" + gpioPinNum);
        /*
         * To access a GPIO pin with Pi4J, we must first provision the pin.
         * Provisioning configures the pin based on how we intend to use it.
         * Provisioning can automatically export the pin, set its direction,
         * and setup any edge detection for interrupt based events.
         * We provision the gpio pin as an input pin with its
         * internal pull down resistor enabled
         */
        device = GpioOnPi4j.controller.provisionDigitalInputPin(gpioPinNum, PinPullResistance.PULL_DOWN);
        println("DevButtonGpioPi4J register Pi4jHandler " );
        device.addListener( new Pi4jHandler( ) );
    }
    /*
     * ------------------------------------------------------------------
     * Pi4jHandler class
     * An adapter from a pi4j GpioPinListenerDigital to a DeviceButtonImpl (execAction)
     * ------------------------------------------------------------------
     */
    protected class Pi4jHandler implements GpioPinListenerDigital{
        @Override
        public void handleGpioPinDigitalStateChangeEvent(GpioPinDigitalStateChangeEvent event) {
            PinState input = event.getState(); //HIGH or LOW
            println("DevButtonGpioPi4J Pi4jHandler= " + ""+input );
            if( input.isLow() ) return;
            String s = input.toString().toLowerCase();
            execAction( s );
        }
    }

    public void terminate(){
        super.terminate();
        device.removeAllListeners();
    }
```

Listing 1.49. `DeviceLedPi4j.java`

10.5 Using the Button Pi4j in the actor

In order to 'inject' this new implementation of the Led into our logical component, we define the theory `buttonPi4jTheory.pl`:

```
/*
===================================================================
buttonPi4jTheory.pl
Defines the rules to create a Button on Raspberry
===================================================================
*/
pinbuttonwpi( 24 ).

createButton( Name, Event, EventMsg ) :-
    pinbuttonwpi(PinNum) ,
    actorobj( Actor ),
    Actor <- getOutputEnvView returns OutView ,
    actorPrintln( createPi4jButton( Name, PinNum) ),
    class("it.unibo.devices.qa.DeviceButtonFactoryQa") <- createButtonPi4j( Name, OutView, PinNum, Actor, Event,
          EventMsg ).
```

Listing 1.50. `ledPi4jTheory.pl`

The next step is to introduce two new rules in the `ledTheory` of Subsection 2.4:

```
buttonImplementation( rasp ).
...
buttonImplementationFile( rasp, "buttonPi4jTheory.pl" ).
```

Finally, we introduce a new button creation operation in the entry in the `DeviceButtonFactoryQa`:

```
    public static DeviceButtonImpl createButtonPi4j(
            String name, IOutputEnvView outEnvView , int pinNum, QActor actor, String eventId, String eventMsg)
                 throws Exception{
//         if( myself == null ){
               myself = new DeviceButtonPi4J( name,outEnvView, pinNum);
               myself.addObserver( new ButtonObserverForActors( name+"obs", outEnvView, actor, eventId, eventMsg) );
//         }
        return myself;
    }
```

Listing 1.51. `DeviceButtonFactoryQa.java : createButtonPi4j`

11 A Button as an observable

In our first actor-based model (see Subsection 7.3) the button sends a message to a single (remote) actor. In that case, our effort was focussed on the relationship between the button-actor and the embedded button-POJO. In particular, the button-POJO has been conceived as an observable (GOF) object, and the ButtonObserverForActors defined in Subsection 7.7 has been introduced to perform (when the button changes its state) two basic actions:

1. emit an event `eventId:eventMsg` (values given at creation time) ;
2. solve the goal `handleButtonInput/1` in order to delegate to the button-actor the handling of the current button state.

Therefore, our button-actor `buttonobsmsg` is already able to interact with any other local or remote actor that decides to sense the `eventId:eventMsg`. In other words, the button-actor is already 'observable' by other actors, thanks to the underlying `qa` event support.

In the following model we want modify the behaviour of our button so that is becomes also able to interact with P2P messages with other actors that explicitly manifest (via a 'registration') their intention to receive updating massages.

```
/*
 * buttonObservable.qa
 */
System  buttonObservable -regeneratesrc
Event   clicked  : clicked(V)   //emitted by the Button (implementation) device
Dispatch register : register(X)  //sent by an observer
Dispatch button  : button(X)    //sent by the button actor to some destination

Context ctxButtonObservable ip [host="localhost" port=8010] -g yellow -httpserver
//EventHandler evh for clicked -print ;

QActor buttonobservable context ctxButtonObservable {
 Rules{
    //actorPerceivesEvents. //event-based (senseButtonEvents)
    //sendMessageImmediately. //POJO sends
    buttonImplementation( gui ).
 }
     Plan init normal
        println("buttonobservable STARTS" ) ;
        switchToPlan configure  ;
        switchToPlan waitForRegister;
        println("buttonobservable ENDS" )
    Plan configure resumeLastPlan
        solve consult("buttonTheory.pl") time(0) onFailSwitchTo prologFailure ;
        solve consult("observableTheory.pl") time(0) onFailSwitchTo prologFailure ;
        solve createButton( "click", clicked, clicked(V) ) time(0) onFailSwitchTo prologFailure
    Plan waitForRegister
//      println("buttonobservable WAITS for a registration" );
        receiveTheMsg m( register, dispatch, SENDER, R, register(SENDER), N ) time(600000) ;
        [ !? tout(X,Y) ] switchToPlan toutExpired ;
        onMsg register : register(SENDER) -> solve addObserver(SENDER) time(0) onFailSwitchTo prologFailure;
        [ !? observer(N,Observer) ] println( registered(N,Observer) );
        solve showSystemConfiguration time(0) onFailSwitchTo prologFailure ;
        repeatPlan 0
    //switchToPlan entryAsOobservable can be called by the button POJO oobservable
    Plan entryAsObserver
        println("buttonobservable entryAsObserver" ) ;
        [ ?? buttonValue(V) ] solve updateObs( button(V), button, dispatch ) time(0) onFailSwitchTo prologFailure
    Plan prologFailure resumeLastPlan
        println("buttonobservable has failed to solve a Prolog goal" )
    Plan toutExpired
        [ ?? tout(X,Y) ] println( timeout(X,Y) )
}
```

Listing 1.52. buttonObservable.qa

The buttonobservable does not need to know its observers. The observers can be dynamically added to a system(see Section 12) that initially starts with the Button as unique component. When the Button registers an observer, the knowledge-base about the current system configuration is shown by the operation showSystemConfiguration introduced in Subsection 12.2.

When the (embedded POJO) button changes its state, the entryAsObserver Plan is called and the Button sends an asynchronous message to each registered observer as specified in the updateOps rule of the observableTheory:

```
/*
===========================================================================
observableTheory.pl
Defines the rules to handle observers
===========================================================================
*/
value( nobs,0 ).
addObserver( Obs ):-
    inc( nobs,1,N ),
    actorPrintln( added( observer(N,Obs) ) ),
    asserta( observer(N,Obs) ). %%insert as first

updateObs( V, MSGID, MSGTYPE ):-
    actorobj(A),
    value( nobs,N ),
    updateObs( MSGID, MSGTYPE, V, A,1,N ).

updateObs( MSGID, MSGTYPE, CONTENT, A,I,N ) :- I>N,!.
updateObs( MSGID, MSGTYPE, CONTENT, A,I,N ) :-
    observer(I,Obs),
    %% actorPrintln( updateObs( MSGID, MSGTYPE, CONTENT, A,Obs,N) ),
    A <-sendMsg( MSGID, Obs, MSGTYPE, CONTENT ),
    I1 is I + 1,
    updateObs( MSGID, MSGTYPE, CONTENT, A,I1,N ).
```

Listing 1.53. observableTheory.pl

To test the buttonObservable we can introduce some local observer actor:

```
QActor btnobserver1 context ctxButtonObservable {
    Plan init normal
        println("btnobserver1 STARTS" ) ;
        forward buttonobservable -m register : register(btnobserver1);
        switchToPlan observe
    Plan observe
        receiveMsg time(300000);
        printCurrentMessage ;
        repeatPlan 0
}
```

Listing 1.54. buttonObservable.qa: an observer

11.1 A Led as an observer

Now that we have extended the GOF observer pattern to a distributed environment, we can dynamically introduce (see Section 12) a remote Led that works as a button observer, by looking either at messages or at events.

```
/*
 * ledAsObserver.qa
 * A led that works as an observer of the buttonObservable
 */
System ledAsObserver -regeneratesrc
Event    clicked   : clicked(V)    //emitted by the Button (implementation) device
Dispatch register : register(X)  //sent by an observer
Dispatch button  : click        //sent by the button actor to some destination
//Dispatch turnLed : turnLed(X)

Context ctxButtonObservable ip [host="localhost" port=8010] -standalone
Context ctxLedAsObserver ip [host="localhost" port=8043] -g white
EventHandler evh for clicked -print ;

QActor ledobserver context ctxLedAsObserver {
Rules{
    // ledImplementation( gui ).
}
    Plan init normal
        println( ledmsg(starts) );
        switchToPlan configure ;
        println("ledobserver registers to the button" ) ;
        forward buttonobservable -m register : register(ledobserver) ;
//          switchToPlan observeMsg
        switchToPlan observeEvents
    Plan configure resumeLastPlan
        solve consult("ledTheory.pl") time(0) onFailSwitchTo prologFailure ;
        solve createLed("led",0) time(0) onFailSwitchTo prologFailure
    Plan observeMsg
        receiveMsg time(300000); //after 300 secs end the timeout
        [ !? tout(X,Y) ] switchToPlan toutExpired ;
//          printCurrentMessage ;
        onMsg button : click -> println("clicked");
        solve doSwitch time(0) onFailSwitchTo prologFailure;
        repeatPlan 0
    Plan observeEvents
        sense time(600000) clicked -> continue ;
//          printCurrentEvent ;
        onEvent clicked : clicked(V) -> solve doSwitch time(0) onFailSwitchTo prologFailure;
        repeatPlan 0
    Plan prologFailure resumeLastPlan
        println("ledmsg has failed to solve a Prolog goal" )
    Plan toutExpired
        [ ?? tout(X,Y) ] println( timeout(X,Y) )
}

QActor buttonobservable context ctxButtonObservable {
     Plan init normal
        println("Never HERE. I am just a place holder" )
}
```

Listing 1.55. `ledAsObserver.qa`

12 Towards dynamic systems

The flag `-standalone` in the declaration of a Context indicates that all the actors defined in that context must be considered external to the current system and (perhaps) already in existence. Thus, the actors defined in a `-standalone` Context are 'place holders' for reference purposes (e.g. message passing, that requires a reference to an actor name).

The ButtonLed introduced in Section 11 is an example of a dynamic system that can be incrementally expanded. The first logical step consists in activating the Button as a (standalone) observable entity. In the next step, a number of Leds can be activated, each working (in its own computational) node as an observer that must 'register' itself to the Button.

12.1 Updating the system knowledge base

Each time that a new component is dynamically added to the system, the system knowledge-base stored in each Context[13] is dynamically extended by the `qa` run-time support with the configuration related to the new component. In this way, each actor in the system becomes able to send messages to other remote actors (dynamically introduced in the system), if it is able to know their names[14]

With reference to the ButtonLed, the initial knowledge-base related to the observable Button[15] is:

```
%====================================================================================
% Context ctxButtonObservable SYSTEM-configuration: file it.unibo.ctxButtonObservable.buttonObservable.pl
%====================================================================================
context(ctxbuttonobservable, "localhost", "TCP", "8010" ).
%%% -------------------------------------------
qactor( buttonobservable , ctxbuttonobservable ).
qactor( btnobserver1 , ctxbuttonobservable ).
%%% -------------------------------------------
```

Listing 1.56. `buttonobservable.pl of ctxButtonObservable`

The initial knowledge-base related to a Led[16] is:

```
%====================================================================================
% Context ctxLedAsObserver SYSTEM-configuration: file it.unibo.ctxLedAsObserver.ledAsObserver.pl
%====================================================================================
context(ctxbuttonobservable, "localhost", "TCP", "8010" ).
context(ctxledasobserver, "localhost", "TCP", "8043" ).
%%% -------------------------------------------
qactor( ledobserver , ctxledasobserver ).
qactor( buttonobservable , ctxbuttonobservable ).
%%% -------------------------------------------
```

Listing 1.57. `ledasobserver.pl of ctxLedAsObserver`

[13] The static part of the system knowledge is stored in a file `projectname,pl` generated in the Context package of the directory `srcMore`.

[14] The `register` message sent by a Led to the Button is the way used to make the Button aware of the name of the Led.

[15] The initial knowledge-base of the Button is generated in the file *srcMore/it/unibo/ctxButtonObservable/buttonobservable.pl*

[16] The initial knowledge-base of the observer Led is generated in the file *srcMore/it/unibo/ctxLedAsObserver/ledasobserver.pl*

12.2 Getting the system-configuration at application level

The knowledge about a context configuration can be acquired at application level by rules like the following ones:

```
 /*
 ----------------------------------
Find system configuration
 ----------------------------------
 */
getTheContexts(Ctx,CTXS) :-
    Ctx <- solvegoal("getTheContexts(X)","X") returns CTXS .
getTheActors(Ctx,ACTORS) :-
    Ctx <- solvegoal("getTheActors(X)","X") returns ACTORS.

 /*
 ----------------------------------
Show system configuration
 ----------------------------------
 */
showSystemConfiguration :-
    actorobj(A),
    A <- getContext returns Ctx,
    actorPrintln('CURRENT CONTEXTS:'),
    getTheContexts(Ctx,CTXS),
    showElements(CTXS),
    actorPrintln('CURRENT ACTORS:'),
    getTheActors(Ctx,ACTORS),
    showElements(ACTORS).

showElements(ElementListString):-
    text_term( ElementListString, ElementList ),
    %% actorPrintln( list(ElementList) ),
    showListOfElements(ElementList).
showListOfElements([]).
showListOfElements([C|R]):-
    actorPrintln( C ),
    showElements(R).
```

Listing 1.58. observableTheory.pl

When the Led is added to the system, the knowledge-base of the Button is shown as follows[17] :

```
--- CURRENT CONTEXTS:
--- context(ctxbuttonobservable,localhost,'TCP','8010')
--- context(ctxledasobserver,localhost,'TCP','8043')
--- CURRENT ACTORS:
--- qactor(buttonobservable,ctxbuttonobservable)
--- qactor(btnobserver1,ctxbuttonobservable)
--- qactor(evlpactxbuttonobservable,ctxbuttonobservable)
--- qactor(ledobserver,ctxledasobserver)
--- qactor(evlpactxledasobserver,ctxledasobserver)
```

The same knowledge-base is updated in the Led site (and all in the other nodes dynamically added to the system).

[17] The actors whose names stars with **evlpa** are internal actors of class **EventLoopActors** that implement event-handling including system events.

13 A system based on events

The button-POJO embedded in the Button actor is an observable (GOF) object associated with a 'listener' (ButtonObserverForActors, see Subsection 7.7) that emits an event eventId:eventMsg (values given at creation time). Therefore, our button-actor is already an 'observable' entity and any other local or remote actor that decides to sense the eventId:eventMsg can be work as a button observer.

Thus, the logical architecture of the ButtonLed system can be redefined as follows.

13.1 The Button as an event emitter

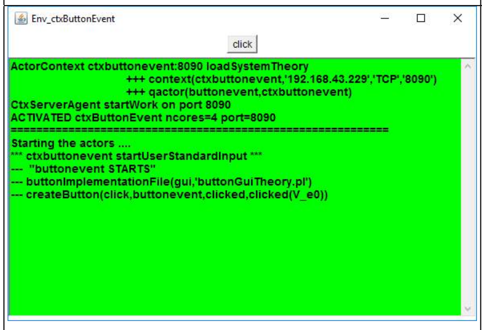
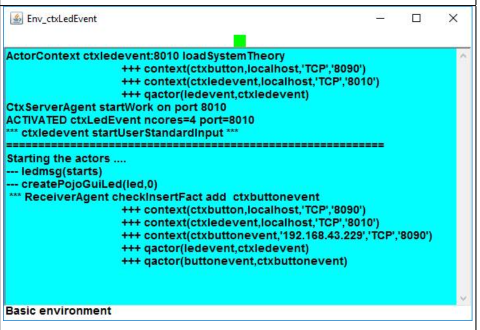
The Button can be defined as an actor that emits the event clicked:clicked(click). A first prototype (useful in our first sprint review) can be introduces by selecting the GUI implementation of the Button:

```
/*
 * buttonEvent.qa     project it.unibo.bls2016.button
 */
System  buttonMsg -regeneratesrc
Event   clicked : clicked(V) //emitted by the Button (implementation) device
Context ctxButtonEvent ip [host="192.168.43.229" port=8090] -g green
//EventHandler evh for clicked -print ;

QActor buttonevent context ctxButtonEvent {
Rules{
    actorPerceivesEvents. //event-based (senseButtonEvents)
    //sendMessageImmediately. //POJO sends
    buttonImplementation( gui ).
    callback( entryAsObserver ).
 }
    Plan init normal
        println("buttonevent STARTS" ) ;
        switchToPlan configure  ;
        [!? actorPerceivesEvents] switchToPlan senseButtonEvents;
        println("buttonevent ENDS" )
    Plan configure resumeLastPlan
        solve consult("buttonTheory.pl") time(0) onFailSwitchTo prologFailure ;
        solve createButton( "click", clicked, clicked(V) ) time(0) onFailSwitchTo prologFailure
    Plan senseButtonEvents
        //println("buttonevent WAITS for an event" );
        sense time(600000) clicked -> continue;
        onEvent clicked : clicked(V) -> println( senseButtonEvents(V) ) ;
        repeatPlan 0
    //switchToPlan entryAsObserver can be called by the button POJO observer
    Plan entryAsObserver
        println("buttonevent entryAsObserver" );
        [ ?? buttonValue(V)] println( buttonValue(V) )
    Plan prologFailure resumeLastPlan
        println("buttonevent has failed to solve a Prolog goal" )
}
```

Listing 1.59. The Button as an event emitter buttonEvent.qa

The Button Plan (state) senseButtonEvents does not perform here any interesting action and could be completely omitted. However, a more significant Button behaviour is introduced in Section 15.

Since the Button can be observed by any new component that decides to perceive the clicked event we can immediately starts to run the button without any observer:

The Button	The Led
	

```
ActorContext ctxbuttonevent:8090 loadSystemTheory
                         +++ context(ctxbuttonevent,'192.168.43.229','TCP','8090')
                         +++ qactor(buttonevent,ctxbuttonevent)
CtxServerAgent startWork on port 8090
ACTIVATED ctxButtonEvent ncores=4 port=8090
=================================================================
Starting the actors ....
*** ctxbuttonevent startUserStandardInput ***
---  "buttonevent STARTS"
--- buttonImplementationFile(gui,'buttonGuiTheory.pl')
--- createButton(click,buttonevent,clicked,clicked(V_e0))
```

```
ActorContext ctxledevent:8010 loadSystemTheory
                         +++ context(ctxbutton,localhost,'TCP','8090')
                         +++ context(ctxledevent,localhost,'TCP','8010')
                         +++ qactor(ledevent,ctxledevent)
CtxServerAgent startWork on port 8010
ACTIVATED ctxLedEvent ncores=4 port=8010
*** ctxledevent startUserStandardInput ***
=================================================================
Starting the actors ....
--- ledmsg(starts)
--- createPojoGuiLed(led,0)
 *** ReceiverAgent checkInsertFact add  ctxbuttonevent
                         +++ context(ctxbutton,localhost,'TCP','8090')
                         +++ context(ctxledevent,localhost,'TCP','8010')
                         +++ context(ctxbuttonevent,'192.168.43.229','TCP','8090')
                         +++ qactor(ledevent,ctxledevent)
                         +++ qactor(buttonevent,ctxbuttonevent)
Basic environment
```

13.2 The Led as an event perceiver

The Led can be defined as actor that waits for a clicked event and then, when the event is perceived, switches its state. Also in this case our first prototype is based on the `GUI` implementation of the Led:

```
/*
 * ledEvent.qa    project it.unibo.bls2016.led
 */
System ledEvent -regeneratesrc
Event   clicked : clicked(V) //emitted by the Button (implementation) device
Context ctxButton ip [host="localhost" port=8090] -standalone
Context ctxLedEvent ip [host="localhost" port=8010] -g cyan

QActor ledevent context ctxLedEvent {
    Plan init normal
        println( ledevent(starts) );
        switchToPlan configure ;
        switchToPlan work
    Plan configure resumeLastPlan
        solve consult("ledTheory.pl") time(0) onFailSwitchTo prologFailure ;
        solve createLed("led",0) time(0) onFailSwitchTo prologFailure
    Plan work
        sense time(300000) clicked -> continue;
        onEvent clicked : clicked(click) -> solve doSwitch time(0) onFailSwitchTo prologFailure;
        repeatPlan 0
    Plan prologFailure resumeLastPlan
        println("ledevent has failed to solve a Prolog goal" )
    Plan toutExpired
        [ ?? tout(X,Y) ] println( timeout(X,Y) )
}
```

Listing 1.60. `ledEvent.qa`

Note that the button context is declared as `-standalone`. In this case we do not introduce any place holder (for the Button) since we don't have any direct interaction with it.

14 A Button on Android

The ButtonObserverForActors of Subsection 7.7 is a 'listener' that we can add also to a Button-POJO implemented on an Android system. According to our work-plan, we can introduce the following (in SCRUM terminology) *product-backlog*:

– update the buttonTheory of Subsection 7.6 by introducing new facts related to the implementation of a Button-POJO in a Android environment:

```
buttonImplementation( android ).
...
buttonImplementationFile( gui, "buttonAndroidTheory.pl" ).
```

– define the buttonAndroidTheory:

```
/*
================================================================
buttonAndroidTheory.pl
Defines the rules to create and use a Button
================================================================
*/
createButton( Name, Event, EventMsg ) :-
    actorobj( Actor ),
    Actor <- createButton( Name, Event, EventMsg ).
```

Listing 1.61. buttonAndroidTheory.pl

The rule createButton/3 does not use any more the class("...") <- op(...) mechanism[18] , but explicitly delegates the creation of the button to an operation (createButton) to be defined by the application designer within the actor so to adapt this task to the rules of the Android environment.

– define an implementation for the button for Android as a specialization of the class DeviceButtonImpl of Subsection 8.1.

14.1 The implementation

For the sake of simplicity, we do not use any more the DeviceButtonFactoryQa of Subsection 8. Rather, we do introduce a factory method (createButtonAndroid) in the class *DeviceButtonAndroid*.

```
package it.unibo.android.button;
import it.unibo.buttonLed.components.DeviceButtonImpl;
import it.unibo.buttonOnAndrod.ButtonOnAndrodActivity;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.is.interfaces.IOutputView;
import it.unibo.qactors.QActor;
import android.view.View;

public class DeviceButtonAndroid extends DeviceButtonImpl{
private static DeviceButtonImpl buttonAndroid;
    public DeviceButtonAndroid(String name, IOutputView outView, View view) {
        super(name, null);
        this.outView = outView;
        if( view != null ) config(view, this);
    }
    protected void config(View ba, final DeviceButtonAndroid b){
        ba.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
```

[18] The tuProlog requires some modification of usage (see tuProlog manual, section 3.4) in an Android environment

```
                try {
                    //println("buttonAction onClick " + View.class.getName());
                    b.update(true);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
    }
    /*
     * Factory part
     */
    public static DeviceButtonImpl getTheButton( ){
        return buttonAndroid;
    }
    public static DeviceButtonImpl createButtonAndroid(
             String name, IOutputEnvView outEnvView, QActor actor, String eventId, String eventMsg) throws Exception{
//      outEnvView.addOutput( " +++ createButtonAndroid" );
        View androidView = ButtonOnAndrodActivity.getAndroidView();
        buttonAndroid   = new DeviceButtonAndroid( name, outEnvView, androidView);
        buttonAndroid.addObserver(new ButtonObserverForActors( name+"obs",outEnvView,actor,eventId,eventMsg));
        return  buttonAndroid;
    }
 }
```

Listing 1.62. `DeviceButtonAndroid.java`

This factory method is called by the `createButton` operation that should be defined by the Application designer in the generated button implementation class `Buttonqaandroid`:

```
/* Generated by AN DISI Unibo */
/*
This code is generated only ONCE
*/
package it.unibo.buttonqaandroid;
import it.unibo.android.button.DeviceButtonAndroid;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.qactors.ActorContext;

public class Buttonqaandroid extends AbstractButtonqaandroid {
    public Buttonqaandroid(String actorId, ActorContext myCtx, IOutputEnvView outEnvView ) throws Exception{
        super(actorId, myCtx, outEnvView);
    }
    public void createButton( String name, String eventId, String eventMsg){
        try {
            DeviceButtonAndroid.createButtonAndroid(name,outEnvView,this,eventId,eventMsg);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Listing 1.63. `Buttonqaandroid.java`

The class `Buttonqaandroid` is generated only once, as a specialized version of an abstract class that implements the behaviour defined in the model.

14.2 A model

Let us introduce here model similar to that of Section 11:

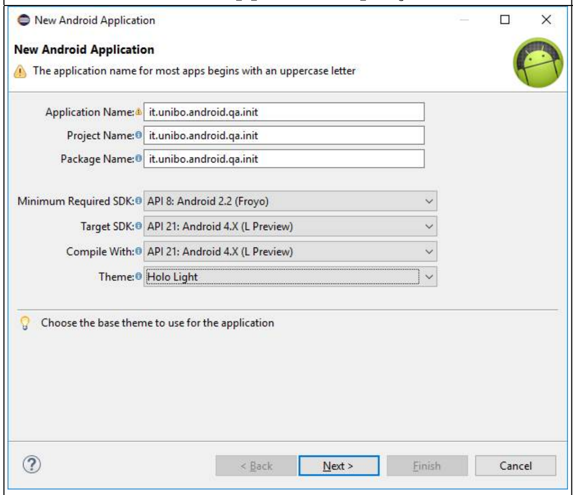
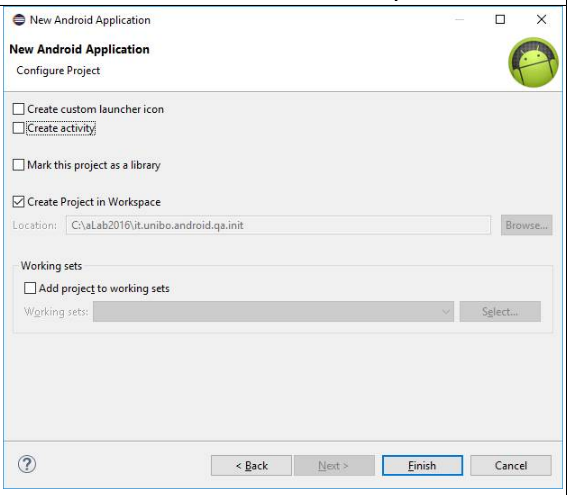
```
/*
 * buttonAndroid.qa
 */
System  buttonOnAndrod -regeneratesrc
Event   clicked  : clicked(V)   //emitted by the Button (implementation) device
```

```
Dispatch register : register(X)  //sent by an observer
Dispatch button  : button(X)    //sent by the button actor to some destination

Context ctxButtonOnAndrod ip [host="192.168.43.71" port=8010]
//EventHandler evh for clicked -print ;

QActor buttonqaandroid context ctxButtonOnAndrod {
 Rules{
    //actorPerceivesEvents. //event-based (senseButtonEvents)
    //sendMessageImmediately. //POJO sends
    //buttonImplementation( gui ).
    callback( entryAsObserver ).
 }
     Plan init normal
        println("button STARTS" ) ;
        switchToPlan configure  ;
        switchToPlan waitForRegister;
        println("button ENDS" )
    Plan configure resumeLastPlan
        solve consult("buttonTheory.pl") time(0) onFailSwitchTo prologFailure ;
        solve consult("observableTheory.pl") time(0) onFailSwitchTo prologFailure ;
        solve createButton( "click", clicked, clicked(V) ) time(0) onFailSwitchTo prologFailure
    Plan waitForRegister
//      println("button WAITS for a registration" );
        receiveTheMsg m( register, dispatch, SENDER, R, register(SENDER), N ) time(600000) ;
        [ !? tout(X,Y) ] switchToPlan toutExpired ;
        onMsg register : register(SENDER) -> solve addObserver(SENDER) time(0) onFailSwitchTo prologFailure;
        [ !? observer(N,Observer) ] println( registered(N,Observer) );
        solve showSystemConfiguration time(0) onFailSwitchTo prologFailure ;
        repeatPlan 0
    //switchToPlan entryAsOobservable can be called by the button POJO oobservable
    Plan entryAsObserver
        println("buttonqaandroid entryAsObserver" ) ;
        [ ?? buttonValue(V) ] solve updateObs( button(V), button, dispatch ) time(0) onFailSwitchTo prologFailure
    Plan prologFailure resumeLastPlan
        println("buttonqaandroid has failed to solve a Prolog goal" )
    Plan toutExpired
        [ ?? tout(X,Y) ] println( timeout(X,Y) )
}
/*
 * A local tester
 */
QActor btnobserver1 context ctxButtonOnAndrod {
    Plan init normal
        println("btnobserver1 STARTS" ) ;
//          println("btnobserver1 ENDS" ) ;
//          forward buttonqaandroid -m register : register(btnobserver1);
//      switchToPlan observe
        switchToPlan perceive
    Plan perceive
        sense time(300000) clicked -> continue;
        printCurrentEvent ;
        repeatPlan 0
    Plan observe
        receiveMsg time(300000);
        printCurrentMessage ;
        repeatPlan 0
    Plan prologFailure resumeLastPlan
        println("btnobserver1 has failed to solve a Prolog goal" )
}
```

Listing 1.64. buttonAndroid.qa

14.3 The software factory under Android

Let us start by creating a new Android Application Project[19].

Android Application project 1	Android Application project 2
	

New Android Application

New Android Application

The application name for most apps begins with an uppercase letter

Application Name: it.unibo.android.qa.init
Project Name: it.unibo.android.qa.init
Package Name: it.unibo.android.qa.init

Minimum Required SDK: API 8: Android 2.2 (Froyo)
Target SDK: API 21: Android 4.X (L Preview)
Compile With: API 21: Android 4.X (L Preview)
Theme: Holo Light

Choose the base theme to use for the application

New Android Application

New Android Application

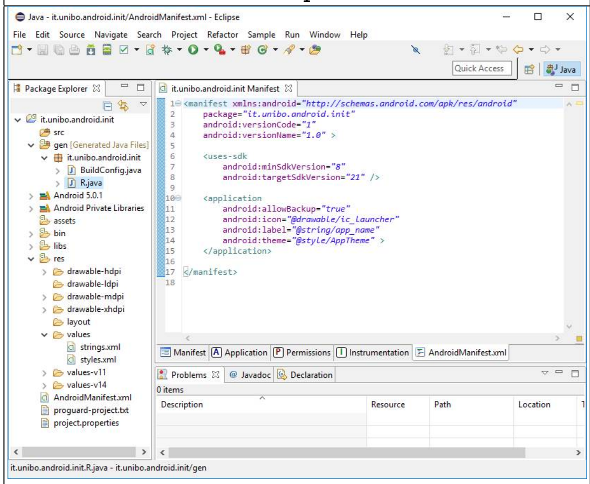
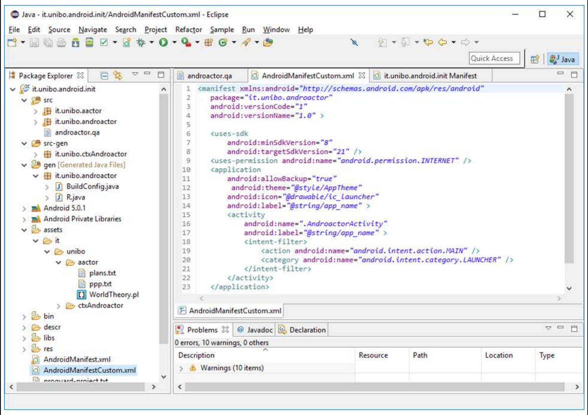
Configure Project

- [ ] Create custom launcher icon
- [ ] Create activity
- [ ] Mark this project as a library
- [✓] Create Project in Workspace

Location: C:\aLab2016\it.unibo.android.qa.init

Working sets
- [ ] Add project to working sets

If we introduce a qa model, a set of resources is generated, including a file *AndroidManifestCustom.xml* whose content must replace the content of the *AndroidManifest.xml* generated by the AIDE (Android IDE):

Define a qa model	Code generated
	

Java - it.unibo.android.init/AndroidManifest.xml - Eclipse

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="it.unibo.android.init"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="21" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
    </application>

</manifest>
```

Problems: 0 items

Java - it.unibo.android.init/AndroidManifestCustom.xml - Eclipse

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="it.unibo.androactor"
   android:versionCode="1"
   android:versionName="1.0" >

   <uses-sdk
       android:minSdkVersion="8"
       android:targetSdkVersion="21" />
   <uses-permission android:name="android.permission.INTERNET" />
   <application
       android:allowBackup="true"
        android:theme="@style/AppTheme"
       android:icon="@drawable/ic_launcher"
       android:label="@string/app_name" >
       <activity
           android:name=".AndroactorActivity"
           android:label="@string/app_name" >
           <intent-filter>
               <action android:name="android.intent.action.MAIN" />
               <category android:name="android.intent.category.LAUNCHER" />
           </intent-filter>
       </activity>
   </application>
```

Problems: 0 errors, 10 warnings, 0 others — Warnings (10 items)

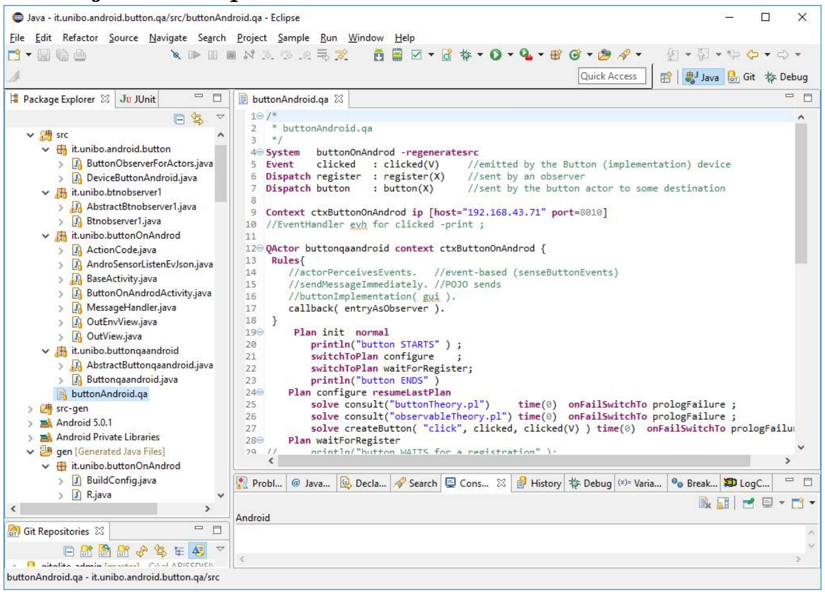
Note that the srcMore directory now does not exist any more; its content is now generated in the assets Android directory. Moreover, a xml file (whose name ***must*** be written in lower-case letters) is now generated in the Android directory res/layout to include the specification of a 'standard' GUI for the qa application, that forces an updating (by the AIDE)of the R.java file in the gen directory.

[19] We suppose to work in *Eclipse/Xtext* extended with the qa plugins.

In the `src` directory we can find an Activity (that extends the generated `BaseActivity` class), that defines the skeleton of the main application activity. This activity is generated only once; the task of the Application designer is to complete this code according to application needs.

Here is the structure of the project workspace for the model of Subsection 14.2.

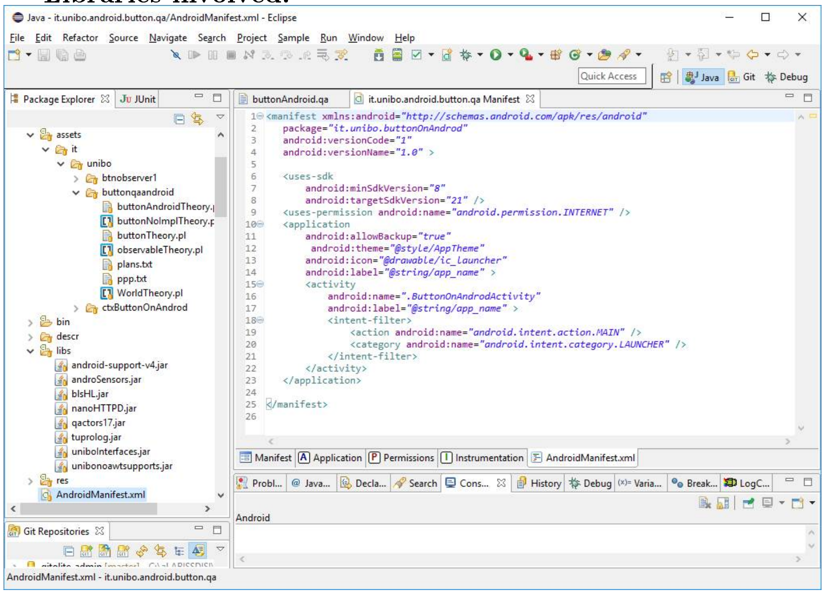
Project workspace.



```
/*
 * buttonAndroid.qa
 */
System   buttonOnAndrod -regeneratesrc
Event    clicked   : clicked(V)     //emitted by the Button (implementation) device
Dispatch register  : register(X)    //sent by an observer
Dispatch button    : button(X)      //sent by the button actor to some destination

Context ctxButtonOnAndrod ip [host="192.168.43.71" port=8010]
//EventHandler evh for clicked -print ;

QActor buttonqaandroid context ctxButtonOnAndrod {
 Rules{
    //actorPerceivesEvents.   //event-based (senseButtonEvents)
    //sendMessageImmediately. //POJO sends
    //buttonImplementation( gui ).
    callback( entryAsObserver ).
 }
     Plan init  normal
       println("button STARTS" ) ;
       switchToPlan configure    ;
       switchToPlan waitForRegister;
       println("button ENDS" )
    Plan configure resumeLastPlan
       solve consult("buttonTheory.pl")     time(0)  onFailSwitchTo prologFailure ;
       solve consult("observableTheory.pl") time(0)  onFailSwitchTo prologFailure ;
       solve createButton( "click", clicked, clicked(V) ) time(0)  onFailSwitchTo prologFailu
    Plan waitForRegister
//     println("button WAITS for a registration" );
```

Android

Libraries involved.



```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="it.unibo.buttonOnAndrod"
   android:versionCode="1"
   android:versionName="1.0" >

   <uses-sdk
       android:minSdkVersion="8"
       android:targetSdkVersion="21" />
   <uses-permission android:name="android.permission.INTERNET" />
   <application
       android:allowBackup="true"
        android:theme="@style/AppTheme"
       android:icon="@drawable/ic_launcher"
       android:label="@string/app_name" >
       <activity
           android:name=".ButtonOnAndrodActivity"
           android:label="@string/app_name" >
           <intent-filter>
               <action android:name="android.intent.action.MAIN" />
               <category android:name="android.intent.category.LAUNCHER" />
           </intent-filter>
       </activity>
   </application>

</manifest>

```

Android

15 Interactions using MQTT

In this section we will redesign the remotely-observable Button of Section 11 as an actor that works also as a *publisher* of information by using the MQTT protocol.

The MQ *Telemetry Transport* (MQTT) is an ISO standard (ISO/IEC PRF 20922) publish-subscribe based "light weight" messaging protocol for use on top of the TCP/IP protocol. It is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium.

The *Eclipse Paho* project provides open-source client implementations of MQTT and MQTT-SN (*MQTT For Sensor Networks*[20]) messaging protocols aimed at new, existing, and emerging applications for *Machine-to-Machine* (M2M) and *Internet of Things* (IoT). There are already MQTT C and Java libraries with Lua, Python, C++ and JavaScript at various stages of development.

15.1 The Button as a MQTT publisher

The remotely-observable Button introduced in Section 11 is here extended (see the Plan configure and the Plan entryAsObserver) to become (when it changes its state) a MQTT publisher of information on the topic "unibo/button/qa".

```
/*
 * buttonMqtt.qa
 */
System  buttonMqtt -regeneratesrc
Event   clicked  : clicked(V)   //emitted by the Button (implementation) device
Event   mqtt     : mqtt(TOPIC,PAYLOAD)
Dispatch register : register(X)  //sent by an observer
Dispatch button  : button(X)    //sent by the button actor to some destination
Dispatch mqttmsg : mqttmsg( TOPIC,PAYLOAD )

Context ctxButtonMqtt ip [host="localhost" port=8078] -g cyan
//EventHandler evh for clicked -print ;

QActor buttonmqtt context ctxButtonMqtt {
 Rules{
   //actorPerceivesEvents. //event-based (senseButtonEvents)
   //sendMessageImmediately. //POJO sends
   buttonImplementation( gui ).

 }
     Plan init normal
        println("buttonmqtt STARTS" ) ;
        switchToPlan configure  ;
        switchToPlan waitForRegister;
        println("buttonmqtt ENDS" )
    Plan configure resumeLastPlan
        solve consult("buttonTheory.pl") time(0) onFailSwitchTo prologFailure ;
        solve consult("observableTheory.pl") time(0) onFailSwitchTo prologFailure ;
        solve consult("mqttTheory.pl") time(0) onFailSwitchTo prologFailure ;
        solve createButton( "click", clicked, clicked(V) ) time(0) onFailSwitchTo prologFailure ;
        //ADDITION: Connect to the MQTT server
        switchToPlan connect
    Plan connect resumeLastPlan
        println(buttonmqtt( connect ) ) ;
        //WARNING: Do not use the name of the actor as clientid
        solve connect( "button_mqtt", "tcp://m2m.eclipse.org:1883", "unibo/button/qa" ) time(0) onFailSwitchTo
              prologFailure ;
        println(buttonmqtt( connected ) ) ;
        [ ?? tout(X,Y) ] switchToPlan toutExpired
```

[20] MQTT-SN is a protocol derived from MQTT, designed for connectionless underlying network transports such as UDP

```
   Plan disconnect resumeLastPlan
      println(prod_paho( disconnect ) ) ;
      solve disconnect time(0) onFailSwitchTo prologFailure
   Plan waitForRegister
      println("buttonmqtt WAITS for a registration" );
      receiveTheMsg m( register, dispatch, SENDER, R, register(SENDER), N ) time(600000) ;
      [ !? tout(X,Y) ] switchToPlan toutExpired ;
      onMsg register : register(SENDER) -> solve addObserver(SENDER) time(0) onFailSwitchTo prologFailure;
      [ !? observer(N,Observer) ] println( registered(N,Observer) );
      solve showSystemConfiguration time(0) onFailSwitchTo prologFailure ;
      repeatPlan 0
   //switchToPlan entryAsOobservable can be called by the button POJO oobservable
   Plan entryAsObserver
      println("buttonmqtt entryAsObserver" ) ;
      //ADDITION: publish informsation
      [ !? buttonValue(V) ] solve publish( "button_mqtt",
       "tcp://m2m.eclipse.org:1883", "unibo/button/qa", button(V), 1, true) time(0) onFailSwitchTo prologFailure ;
      [ ?? buttonValue(V) ] solve updateObs( button(V), button, dispatch ) time(0) onFailSwitchTo prologFailure
   Plan prologFailure resumeLastPlan
      println("buttonmqtt has failed to solve a Prolog goal" )
   Plan toutExpired
      [ ?? tout(X,Y) ] println( timeout(X,Y) )
}
```

Listing 1.65. The Button as a MQTT publisher `buttonMqtt.qa`

The usage of the MQTT protocol is delegated to rules defined by the application designer in the `mqttTheory`. These rules in their turn make use of a Java utility object of class `MqttUtils`.

15.2 The MqttUtils

The Java utility class to be used as a support for MQTT interaction can be defined as follows:

```
package it.unibo.mqtt.utils;
import it.unibo.contactEvent.interfaces.IContactEventPlatform;
import it.unibo.contactEvent.platform.ContactEventPlatform;
import it.unibo.qactors.QActor;
import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.eclipse.paho.client.mqttv3.MqttMessage;

public class MqttUtils implements MqttCallback{
private static MqttUtils myself = null;
   protected IContactEventPlatform platform ;
   protected String clientid = null;
   protected String eventId = "mqtt";
   protected String eventMsg = "";
   protected QActor actor = null;
   protected MqttClient client = null;

   public MqttUtils(){
      try {
         platform = ContactEventPlatform.getPlatform();
         myself  = this;
         System.out.println("MqttUtils created "+ myself );
      } catch (Exception e) {
         System.out.println("MqttUtils error "+ e.getMessage() );
      }
   }
   public static MqttUtils getMqttSupport( ){
      return myself ;
   }
   public void test( QActor actor, String name, String brokerAddr, String topic ) throws MqttException{
      System.out.println("test " + topic);
```

```
    }
    public void connect(QActor actor, String brokerAddr, String topic ) throws MqttException{
        System.out.println("connect/3 " );
        clientid = MqttClient.generateClientId();
        connect(actor,  clientid,  brokerAddr,  topic);
    }
    public void connect(QActor actor, String clientid, String brokerAddr, String topic ) throws MqttException{
        System.out.println("connect/4 "+ clientid );
        this.actor = actor;
        client = new MqttClient(brokerAddr, clientid);
        MqttConnectOptions options = new MqttConnectOptions();
        options.setWill("unibo/clienterrors", "crashed".getBytes(), 2, true);
        client.connect(options);
    }
    public void disconnect( ) throws MqttException{
        System.out.println("disconnect "+ client );
        if( client != null ) client.disconnect();
    }
    public void publish(QActor actor, String clientid, String brokerAddr, String topic, String msg, int qos, boolean
         retain) throws MqttException{
        MqttMessage message = new MqttMessage();
        message.setRetained(retain);
        if( qos == 0 || qos == 1 || qos == 2){//qos=0 fire and forget; qos=1 at least once(default);qos=2 exactly once
            message.setQos(0);
        }
        message.setPayload(msg.getBytes());
        client.publish(topic, message);
    }
    public void subscribe(QActor actor, String clientid, String brokerAddr, String topic) throws Exception {
        try{
            this.actor = actor;
            client.setCallback(this);
            client.subscribe(topic);
        }catch(Exception e){
                System.out.println("subscribe error "+ e.getMessage() );
                eventMsg = "mqtt(" + eventId +", failure)";
                System.out.println("subscribe error "+ eventMsg );
                //platform.raiseEvent("mqttutil", eventId, eventMsg );
                if( actor != null ) actor.sendMsg("mqttmsg", actor.getName(), "dispatch", "error");
                throw e;
        }
    }
    @Override
    public  void connectionLost(Throwable cause) {
        System.out.println("connectionLost = "+ cause.getMessage() );
    }
    @Override
    public  void deliveryComplete(IMqttDeliveryToken token) {
        System.out.println("deliveryComplete token= "+ token );
    }
    @Override
    public void messageArrived(String topic, MqttMessage msg) throws Exception {
        System.out.println("messageArrived on "+ topic + "="+msg.toString());
        String mqttmsg = "mqttmsg(" + topic +"," + msg.toString() +")";
        System.out.println("messageArrived mqttmsg "+ mqttmsg);
//      platform.raiseEvent("mqttutil", eventId, mqttmsg ); //events are here 'equivalent' to messages
        if( actor != null ) actor.sendMsg("mqttmsg", actor.getName(), "dispatch", mqttmsg);
    }
}
```

Listing 1.66. The utility class `MqttUtils.java`

An object of class `MqttUtils` is used as a *singleton* and works as the support for the actor that calls the operation `connect` (that creates a `MqttClient`).

The `subscribe` operation sets this singleton support as the object that provides the callback (`messageArrived`) to be called when the `MqttClient` is a subscriber. The callback is defined so to map a `MqttMessage` into a dispatch of the form:

```
mqttmsg : mqttmsg( TOPIC,PAYLOAD )
```

This dispatch is then sent to the actor that uses the singleton support, i.e. that works as a `MqttClient` (subscriber).

15.3 The mqttTheory

The `mqttTheory` that 'extends' the `qa` action-set with new operations for the usage of the `MQTT` protocol can be defined as follows:

```
/*
==============================================================
mqttTheory.pl
==============================================================
*/
connect( Name, BrokerAddr, Topic ):-
    java_object("it.unibo.mqtt.utils.MqttUtils", [], UMQTT),
    actorobj(A),
    actorPrintln( connect(UMQTT, A, Name, BrokerAddr, Topic ) ),
    UMQTT <- connect(A, Name, BrokerAddr, Topic ).
disconnect :-
    actorPrintln( disconnect ),
    class("it.unibo.mqtt.utils.MqttUtils") <- getMqttSupport returns UMQTT,
    %% actorPrintln( disconnect( UMQTT ) ),
    UMQTT <- disconnect.

publish( Name, BrokerAddr, Topic, Msg, Qos, Retain ):-
    actorPrintln( publish( Name, BrokerAddr, Topic, Msg, Qos, Retain ) ),
    actorobj(A),
    %% java_object("it.unibo.mqtt.utils.MqttUtils", [], UMQTT),
    class("it.unibo.mqtt.utils.MqttUtils") <- getMqttSupport returns UMQTT,
    %% actorPrintln( publish( UMQTT ) ),
    UMQTT <- publish(A, Name, BrokerAddr, Topic, Msg, Qos, Retain).
subscribe( Name, BrokerAddr, Topic ):-
    actorPrintln( subscribe( BrokerAddr, Topic ) ),
    actorobj(A),
    %% java_object("it.unibo.paho.utils.MqttUtils", [], UMQTT),
    class("it.unibo.mqtt.utils.MqttUtils") <- getMqttSupport returns UMQTT,
    %% actorPrintln( subscribe( UMQTT ) ),
    UMQTT <- subscribe(A, Name, BrokerAddr, Topic ).

mqttinit :-  actorPrintln("mqttTheory started ...") .
:- initialization(mqttinit).
```

Listing 1.67. The `mqttTheory.pl`

15.4 A Button observer as a MQTT subscriber

To test the behaviour of our new Button-publisher, let us introduce an observer as a `MQTT` subscriber:

```
/*
 * observerMqtt.qa
 */
System  buttonMqtt -regeneratesrc
Event   clicked  : clicked(V)   //emitted by the Button (implementation) device
Event   mqtt     : mqtt(TOPIC,PAYLOAD)
Dispatch register : register(X)  //sent by an observer
Dispatch button   : button(X)    //sent by the button actor to some destination
Dispatch mqttmsg : mqttmsg( TOPIC,PAYLOAD )

Context ctxObserverMqtt ip [host="localhost" port=8062]
//EventHandler evh for clicked -print ;
```

```
QActor mqttobs1 context ctxObserverMqtt -g green {
   Plan init normal
      println("btnobserver1 STARTS" ) ;
      solve consult("mqttTheory.pl") time(0) onFailSwitchTo prologFailure ;
      switchToPlan connect;
      switchToPlan subscribe;
      switchToPlan observe
   Plan connect resumeLastPlan
      println(mqttobs1( connect ) ) ;
      solve connect( "obs1", "tcp://m2m.eclipse.org:1883", "unibo/button/qa" ) time(0) onFailSwitchTo prologFailure ;
      [ ?? tout(X,Y) ] switchToPlan toutExpired
   Plan disconnect resumeLastPlan
      println(mqttobs1( disconnect ) ) ;
      solve disconnect time(0) onFailSwitchTo prologFailure
   Plan subscribe resumeLastPlan
      println(mqttobs1( subscribe) ) ;
      solve subscribe( "obs1", "tcp://m2m.eclipse.org:1883", "unibo/button/qa") time(0) onFailSwitchTo prologFailure
   Plan observe
      receiveMsg time(300000);
      [ ?? tout(X,Y) ] switchToPlan toutExpired ;
      printCurrentMessage ;
      onMsg mqttmsg : mqttmsg(TOPIC,PAYLOAD) -> println(mqttmsg(TOPIC,PAYLOAD) ) ;
      repeatPlan 0
   Plan prologFailure
      println( failure(prolog) )
   Plan toutExpired
      [ ?? tout(X,Y) ] println( timeout(X,Y) )
}
```

Listing 1.68. `observerMqtt.qa`

15.5 A Button observable actor

Of course we can introduce also a conventional actor that works as a Button observer via the 'registration' mechanism of Section 11:

```
/*
 * observerQa.qa
 */
System  observerQa -regeneratesrc
Event   clicked  : clicked(V)    //emitted by the Button (implementation) device
Dispatch register : register(X)  //sent by an observer
Dispatch button   : button(X)    //sent by the button actor to some destination

Context ctxButtonMqtt ip [host="localhost" port=8078] -standalone
Context ctxObserverQa ip [host="localhost" port=8056]
//EventHandler evh for clicked -print ;
QActor btnobserverqa1 context ctxObserverQa -g white {
   Plan init normal
      println("btnobserverqa1 STARTS" ) ;
      forward buttonmqtt -m register : register(btnobserverqa1);
      switchToPlan observe
   Plan observe
      receiveMsg time(300000);
      printCurrentMessage ;
      repeatPlan 0
}
QActor buttonmqtt context ctxButtonMqtt {
    Plan init normal
      println("Never HERE. I am just a place holder" )
}
```

Listing 1.69. `observerQa.qa`

16 From the components to the systems