The ButtonLed system in JavaScript and Node.js

Antonio Natali

Alma Mater Studiorum – University of Bologna
viale Risorgimento 2, 40136 Bologna, Italy
antonio.natali@unibo.it

Table of Contents

The ButtonLed system in JavaScript and Node.js 1
Antonio Natali
1 The ButtonLed system in JavaScript 2
1.1 Installation details 2
2 BLS naive 2
2.1 Led as object literal 3
2.2 Testing with nodeunit 4
3 Devices as objects 5
3.1 Led as object 5
3.2 Button as object 7
3.3 Button as observable 8
3.4 Button as event emitter 10
4 JavaScript on Raspberry 14
4.1 Led 14
4.2 Button 14
5 ButtonLed: an oop version 16
5.1 The Button 16
5.1.1 The High-level Button 16
5.2 The Led 17
5.2.1 The High-level Led 17
5.2.2 The Led implementation on a PC 18
5.2.3 The Led implementation on a Raspberry 18
5.3 A system 19
5.4 Web interface 20
5.5 The BLS as a qa model 21
6 A CRUD HTTP sever to handle a Led 23
6.1 The HTTP-CRUD server 23
6.2 Testing the server 24
6.3 An utility class for Java HTTP clients 25
6.4 A system that handles a Led 26
6.5 Work to do 27
55

1 The ButtonLed system in JavaScript

In this work, we report examples of different JavaScript/Node.js versions of a simple (but representative) software system: a button-led system (`BLS`) in which a Led is turned on/off (by software) when a Button is pressed.

1.1 Installation details

1. connect the Raspberry to the PC with an Ethernet cable
2. share the (`WIFI` on windows) Internet connection with the Ethernet-Raspberry
3. check that the system is working: `ping 8.8.8.8`
4. set the `DNS`:

   ```
   sudo bash
   echo "nameserver 8.8.8.8" > /etc/resolv.conf
   CRTRL D
   ```

   check that the `DNS` is working: `ping google.com`
5. update the system: `sudo apt-get update`
6. install an ARM-version of Node:

   ```
   pi@raspberrypi:~ $  sudo su -
   root@raspberrypi:~ # apt-get remove nodered -y
   root@raspberrypi:~ # apt-get remove nodejs nodejs-legacy -y
   root@raspberrypi:~ # apt-get remove npm  -y # if you installed npm
   root@raspberrypi:~ # curl -sL https://deb.nodesource.com/setup_5.x | sudo bash -
   root@raspberrypi:~ # apt-get install nodejs -y
   root@raspberrypi:~    node -v
   sudo apt-get install nodejs npm
   npm -v
   CRTL D
   ```

7.

2 BLS naive

Our first version is based on the idea of JavaScript as a scripting language to program the behaviour of web pages. We introduce a function (*buttonPress*) to be called when a Button of a HTML page is pressed:

```
/*
* ===================================
* bls0.js
* ===================================
*/
var ledOn = true; //global variable

buttonPress = function(ledId){
   ledOn = ! ledOn;
   if( ledOn ) document.getElementById(ledId).style.backgroundColor='#00FF33';
   else document.getElementById(ledId).style.backgroundColor='#FF0000';
   println( "The led is " + ledOn );
}
```

Listing 1.1. `bls0.js`

In this version, the Led is represented as a page-area with a background color that changes (from red to green) when the Button is pressed. The HTML page can be defined as follows :

```html
<html>
<head>
<script src="bls0.js"></script>
</head>
</html>
<body>
<h4>A ButtonLed system in Javascript and HTML5</h4>
<!-- A LED named ledId -->
<div id="ledId" style="height:20px; width:3%; position: absolute; background-color:#00FF33"></div>
<!-- BUTTON : a click calls buttonPress('ledId') where ledId is the name of the led -->
 <br/> <br/>
<div><button onclick="buttonPress('ledId');">BUTTON</button></div>
</body>
</htnl>
```

Listing 1.2. `bls0Simple.html`

This 'system' works, but it does not represent a good software product, for several reasons:

- it implements the business logic in a very 'technology dependent' way, by working on HTML elements;
- it includes the business logic in a function that works as a 'listener' of the 'onclick' event of a HTML button:
- it does not introduce any reusable entity related to the notion of Button and Led;
- ...

2.1 Led as object literal

Let us now represent a Led as an object literal that defines a state and property (`guiId`) to reference a HTML page-area.

```javascript
/*
* ====================================
* LedLiteral.js
* ====================================
*/
var led = {
       name:   "led1" ,
       guiId:  "ledGuiId",
       ledState: false ,
//Methods
turnOn: function(){
    this.ledState = true;
},
turnOff: function(){
    this.ledState = false;
},
switchState: function(){
    this.ledState = ! this.ledState;
},
getState: function(){
    return this.ledState;
},
getName: function(){
    return this.name;
},
getDefaultRep: function(){
    return this.name+"||"+ this.ledState
},
showGuiRep: function(){
    if( typeof document != "undefined"){
        if( this.ledState ) document.getElementById(this.guiId).style.backgroundColor='#00FF33';
        else document.getElementById(this.guiId).style.backgroundColor='#FF0000';
    }
    println( this.getDefaultRep() );
}, //trailing comma

};//led

/*
 * UTILITIES
 */
function showMsg(outView, txt){
    if( txt == null ){
        document.getElementById(outView).innerHTML="<font size='5' color='blue'><pre><br/></pre></font>";
        return;
    }
    cc = document.getElementById(outView).innerHTML; //current value to be accumulated
    cc = cc.replace('<font size="5" color="blue"><pre>',""); //DO NOT CHANGE !!!
    cc = cc.replace("</pre></font>","");
    document.getElementById(outView).innerHTML="<font size='5' color='blue'><pre>"+cc + txt+ "</pre></font>";
}
println = function ( v ){
    try{
        if( typeof document != "undefined" ) showMsg( 'outView', v+"<br/>" );
        else console.log( v );
    }catch(e){
        console.log( v );
    }
}
//EXPORTS
if(typeof document == "undefined") module.exports.led = led;
```

Listing 1.3. `LedLiteral.js`

The HTML page includes now also a virtual output area (`outView`):

```
<html>
<head>
<script type="text/javascript" src="LedLiteral.js"></script>
</head>
</html>

<body>
<!--
    OUTPUT Area
 -->

<h3>blsLedObj</h3>
<!-- A LED named ledGuiId -->
<div id="ledGuiId" style="height:20px; width:3%; position: absolute; background-color:#00FF33"></div>
 <br/> <br/>
<!-- OUTPUT Area -->
<div id="outView" ></div>

 <br/> <br/>
<!-- BUTTON : a click calls buttonPress('ledGuiId') where ledGuiId is the name of the led -->
<div>
<button onclick="buttonPress();">BUTTON</button>
</div>

<div id="outView" ></div>

 <script>
var buttonPress = function(){
    led.switchState();
    led.showGuiRep();
}
 </script>

</body>
</htnl>
```

Listing 1.4. `blsLedObjLiteral.html`

2.2 Testing with nodeunit

The introduction of a Led as an object allows us to test the Led without introducing the HTML page, e.g. by using a testing framework as `nodeunit`.

Nodeunit is a framework that allows for multiple test cases (running in parallel), and supports mocks and stubs. It is easy to use, and even allows you to run tests in the browser.

Each test function accepts one argument, which is the test object. The test object contains all the assert module methods plus two new methods: 'expect' and 'done'. The method *expect* can be used to tell nodeunit how many assertions you expect to run, while the *done* function tells nodeunit that the test has completed.

Let us test now our Led as object literal (command: `nodeunit testBlsLedObjLiteral.js`):

```
/*
 * testBlsLedObjLiteral.js for nodeunit
 */
/*
 * The test function accepts one argument, which is the test object.
 * The test object contains all the assert module methods plus two new methods:
 * 'expect' and 'done'.
 * Expect can be used to tell nodeunit how many assertions you expect to run,
 * and the done function tells nodeunit that the test has completed.
 */
var ledMod = require("../LedLiteral");

var l1 = ledMod.led;

exports.testLedInitial = function(test){
    var ledState = l1.getState();
    test.ok(  ! ledState, "initial");
    test.done();
}
exports.testLedTurnOn = function(test){
    l1.turnOn();
    test.ok( l1.getState(), "turnOn");
    test.done();
}
exports.testLedTurnOff=function(test){
    l1.turnOff();
    test.ok( ! l1.getState(), "turnOff");
    test.done();
}
exports.testTurnMany=function(test){
    test.expect(6);       //we expect 6 run
    for( i=1; i<=3; i++){
        l1.turnOn();
        test.ok( l1.getState(), "turnOn");
        l1.turnOff();
        test.ok( ! l1.getState(), "turnOff");
    }
    test.done();
}
```

Listing 1.5. `testBlsLedObjLiteral.js`

3 Devices as objects

A better way to deal with devices is to distinguish between their logical part and their implementation part, by following an oo programming style and some design pattern.

3.1 Led as object

In this version, the Led is represented as an object that receives a name and a `guiId` at construction time:

```
/*
* ====================================
* Led.js
* ====================================
*/

/*
 * *********************************************
 * Led as a conventional 'object'
 * *********************************************
 */
```

```
//State (specific)
var Led = function(name, guiId){
    //Led Constructor: instance data
    this.name     = name;
    this.guiId    = guiId;
    this.ledState = false;
}

//Methods (shared)
Led.prototype.turnOn = function(){
    this.ledState = true;
}
Led.prototype.turnOff = function(){
    this.ledState = false;
    }
Led.prototype.switchState = function(){
    this.ledState = ! this.ledState;
    }
Led.prototype.getState = function(){
    return this.ledState;
}
Led.prototype.getName = function(){
    return this.name;
}
Led.prototype.getDefaultRep = function(){
    return this.name+"||"+ this.ledState
}
Led.prototype.showGuiRep = function(){
    if( typeof document != "undefined"){
        if( this.ledState ) document.getElementById(this.guiId).style.backgroundColor='#00FF33';
        else document.getElementById(this.guiId).style.backgroundColor='#FF0000';
    }
    else println( this.getDefaultRep() );
}

println = function ( v ){
    try{
        if( typeof document != "undefined" ) showMsg( 'outView', v+"<br/>" );
        else console.log( v );
    }catch(e){
        console.log( v );
    }
}
// EXPORTS
if(typeof document == "undefined") module.exports.Led = Led;
//To work is a browser, do: browserify Led.js -o LedBro.js
```

Listing 1.6. `Led.js`

The Led object can be (re)used in other applications. For example:

```
/*
* ====================================
* blsLedObj.js
* ====================================
*/
var ledMod = require("./Led");

var buttonPress = function(led){
    led.switchState();
    led.showGuiRep();
}
/*
 * MAIN (for testing)
 */
function mainBlsLedObj(){
    println(" ---- mainBlsLedObj ---- ");
    l1= new ledMod.Led('l1', null);
    l1.turnOn();
    l1.showGuiRep();
    l1.turnOff();
    l1.showGuiRep();
}

```

```
if( process.argv[1].toString().includes("blsLedObj") ) mainBlsLedObj();
```

Listing 1.7. `blsLedObj.js`

3.2 Button as object

Let us introduce now also the Button as an object:

```
/*
* ====================================
* Button.js
* ====================================
*/
/*
 * ********************************************
 * Button as a conventional 'object'
 * ********************************************
 */
//Specific
function Button ( name, led ){
   //Button Constructor: instance data
   this.name    = name;
   this.led     = led;
}

//Shared
Button.prototype.press = function(){
   this.led.switchState();
   this.led.showGuiRep();
}

//EXPORTS
if(typeof document == "undefined") module.exports.Button = Button;
```

Listing 1.8. `Button.js`

In this version, the Button is directly associated (at construction time) with a Led.
A test in *nodeunit* is (command: `nodeunit testBlsObj.js`):

```
/*
 * testBlsObj.js for nodeunit
 */
//require("../blsUtils") ;
var LedMod   = require("../Led");
var ButtonMod = require("../Button");
var l1 = new LedMod.Led("l1",null);
var b1 = new ButtonMod.Button( 'b1', l1 );

exports.testInitial = function(test){
   var ledState = l1.getState();
   test.ok(  ! ledState, "initial");
   test.done();
}
exports.testTurnOn = function(test){
   l1.turnOn();
   test.ok( l1.getState(), "turnOn");
   test.done();
}
exports.testTurnOff=function(test){
   l1.turnOff();
   test.ok( ! l1.getState(), "turnOff");
   test.done();
}
exports.testTurnMany=function(test){
   test.expect(6);      //we expect 6 run
   for( i=1; i<=3; i++){
      l1.turnOn();
      test.ok( l1.getState(), "turnOn");
      l1.turnOff();
      test.ok( ! l1.getState(), "turnOff");
```

```
    }
    test.done();
}
exports.testButton=function(test){
    b1.press();
    test.ok( l1.getState(), "button press");
    test.done();
}
```

Listing 1.9. `testBlsObj.js`

3.3 Button as observable

The system defined in the previous section works, but is `logically wrong`, since ad entity that works as a Button is logically unrelated to a Led. Rather, we should better model a Button as an "observable" object that 'notifies' a set of possible 'registered' observers when it changes its state.

Let us introduce an `Observable` that allows us to register two different types of computational entities:

– objects that implement the method *update*;
– functions (or better closures) that play the role of a "callbacks"

```
/*
* ===============================================
* ButtonAsObservable.js
* GOAL:
*   define a Button as a logical observable entity
*   that can call registered observers / functions
* ===============================================
*/
/*
 * ********************************************
 * Observable prototype
 * ********************************************
 */
Observable = function(){
    this.nobs       = 0;
    this.nobsfunc   = 0;
    this.observerFunc = [ ];
    this.observer   = [ ];
    this.register   = function(obs){
        //println(" Observable register " + this.nobs);
        this.observer[this.nobs++] = obs;
    }
    this.registerFunc = function(func){
        //println(" Observable registerFunc " + this.nobs + " " + func);
        this.observerFunc[this.nobsfunc++] = func;
    }
    this.notify = function(){
        for(var i=0;i < this.observer.length;i++){
            //console.log(" Observable update " + this.observer[i] );
            this.observer[i].update();
        }
        for(var i=0;i < this.observerFunc.length;i++){
            //console.log(" Observable calls " + this.observerFunc[i] );
            this.observerFunc[i]();
        }
    }
  }
/*
```

Listing 1.10. `ButtonAsObservable.js: the Observable`

A `Button` can now be introduced as an object that inherits from `Observable`:

```
 * Button as an 'object' that inherits from Observable
 * ********************************************
 */
//Specific
```

```
function Button ( name ){
   //Button Constructor: instance data
   this.name    = name;
}

//Shared
Button.prototype          = new Observable();
//Button.prototype.constructor = Button;
Button.prototype.press = function(){
   this.notify();
}

//EXPORTS
if(typeof document == "undefined") module.exports.Button = Button;
```

Listing 1.11. `ButtonAsObservable.js: the Button`

A test in *nodeunit* is (command: `nodeunit testBlsObservableObj.js`):

```
/*
 * testBlsObservableObj.js for nodeunit
 */
var LedMod   = require("../Led");
var ButtonMod = require("../ButtonAsObservable");

var l1 = new LedMod.Led("l1",null);
var b1 = new ButtonMod.Button( 'b1' );

exports.testObservableButton=function(test){
   test.expect(3);       //we expect 3 run
   test.ok(  ! l1.getState(), "testObservableButton initial");
   b1.registerFunc(
          function(){ l1.switchState(); });
   b1.press();
   test.ok( l1.getState(), "testObservableButton press 1");
   b1.press();
   test.ok( ! l1.getState(), "testObservableButton press 2");
   test.done();
}
```

Listing 1.12. `testBlsObservableObj.js`

A page HTML:

```
<html>
<head>
<script src="./ButtonAsObservable.js"></script>
</head>


<body>
<h3>buttonAsObservable</h3>
<!-- LED -->
<div id="ledGuiId" style="height:20px; width:3%; position: absolute; background-color:#00FF33"></div>
<br/><br/>
<!-- BUTTON -->
<div><button onclick="button.press()">BUTTON</button></div>
</body>
<script>
 var button;
 var LedImplGui = function( name ){
       this.name     = name;
       this.ledState = 0;
       document.getElementById( name ).style.backgroundColor='#FF0000';
 }
 LedImplGui.prototype.turnOn = function(){
    this.ledState = 1;
    document.getElementById(this.name).style.backgroundColor='#00FF33';
 }
 LedImplGui.prototype.turnOff = function(){
    this.ledState = 0;
    document.getElementById(this.name).style.backgroundColor='#FF0000';
 }
 LedImplGui.prototype.switchState = function(){
    if( this.ledState ) this.turnOff(); else this.turnOn() ;
 }

var configure = function (){
    //configure the system
    ledgui = new LedImplGui("ledGuiId");
    button = new Button( 'b1' );
    button.registerFunc(
        function(){ ledgui.switchState(); }
    );
    console.log("configuration done" );
}
configure();
</script>

</html>
```

Listing 1.13. `ButtonAsObservable.html`

3.4 Button as event emitter

A web browser that brings JavaScript to a client is `event-driven` (i.e. it uses an `event loop`) and non-blocking when handling I/O (it uses `asynchronous I/O`).

Much of the Node.js core API is built around an idiomatic asynchronous `event-driven architecture` in which certain kinds of objects (called "emitters") periodically emit named events that cause Function objects ("listeners") to be called. The Node.js framework does introduce the `EventEmitter` object, that provides *asynchronous event handling to objects.* With a *EventEmitter* instance we can do two essential tasks: attach an event handler to an event, and emit an event.

All objects that emit events are instances of the *EventEmitter* class. These objects expose an `eventEmitter.on()` function that allows one or more functions to be attached to named events emitted by the object. Typically, event names are camel-cased strings but any valid JavaScript property key can be used.

When the *EventEmitter* object emits an event, all of the functions attached to that specific event are called `synchronously`. Any values returned by the called listeners are ignored and will be discarded.

A possible definition of a Button as an emitter of events is:

```
/*
* ====================================
* ButtonEmitter.js
* ====================================
*/
var EventEmitter = require('events').EventEmitter;
/*
 * ********************************************
 * Button as an emitter of events
 * ********************************************
 */
Button = function ( name ){
    this.emitter = new EventEmitter();
    this.name    = name;
    this.evId    = "pressed";
    this.count   = 1;
}
Button.prototype.emitEvent = function(){
    println("  ButtonEmitter emits the event " + this.evId + " count=" + this.count++);
    this.emitter.emit(this.evId,'buttonPressed') ;
}
Button.prototype.getEmitter = function(){
    return this.emitter;
}
Button.prototype.setHandler = function( handler ){
    this.emitter.on(this.evId, handler );
}
Button.prototype.removeHandler = function( handler ){
    this.emitter.removeAllListeners(this.evId);
}
```

```
Button.prototype.press = function(){
    this.emitEvent() ;
}

if(typeof document == "undefined") module.exports.Button = Button
//To work is a browser, do: browserify ButtonEmitter.js -o ButtonEmitterBro.js
```

Listing 1.14. `ButtonEmitter.js`

A new event is emitted when the method `press` is called.

A test in *nodeunit* shows that a ButtonLed system can be built by delegating to an `event handler` the task of switching the Led each time the Button is pressed:

```
/*
 * testBlsEmitterObj.js
 * USAGE: nodeunit testBlsEmitterObj.js
 */
var LedMod   = require("../Led");
var ButtonMod = require("../ButtonEmitter");

var l1 = new LedMod.Led("l1",null);
var b1 = new ButtonMod.Button( 'b1' );

/*
    It is possible to have multiple groups of tests in a module,
    each group with its own setUp and tearDown functions
*/
module.exports = {
        setUp: function (callback) {
            l1.turnOff();
            b1.setHandler( function( evMsg ){
                console.log("     event handler: evMsg="+ evMsg);
                l1.switchState();}
            );
            b1.emitter.on( b1.evId, function( evMsg ){
                console.log("     log:"+ evMsg);
            });
            console.log("SETUP" );
            callback();
        },
        tearDown: function (callback) {
            console.log("TEARDOWN" );
            // clean up
            b1.removeHandler();
            callback();
        },
        testEmitterButton : function(test){
            test.ok(  ! l1.getState(), "testEmitterButton initial");
            console.log("PRESS");
            b1.press();
            test.ok( l1.getState(), "testEmitterButton press 1");
            console.log("PRESS");
            b1.press();
            test.ok( ! l1.getState(), "testEmitterButton press 2");
            test.done();
        },
        testEmitterButtonAgain : function(test){
            test.ok(  ! l1.getState(), "testEmitterButton initial");
            console.log("PRESS AGAIN " );
            b1.press();
            test.ok( l1.getState(), "testEmitterButton press 1");
            console.log("PRESS AGAIN");
            b1.press();
            test.ok( ! l1.getState(), "testEmitterButton press 2");
            test.done();
        }
};
```

Listing 1.15. `testBlsEmitterObj.js`

A possible application is:

```
/*
* ====================================
* blsObjEmitter.js
*
* ====================================
*/
var LedMod   = require("./Led");
var ButtonMod = require("./ButtonEmitter");

/*
 * A trick to wait for ms milleseconds anf finaly execute a callaback cb (if any)
 */
function waitFor(ms, callback) {
    var waitTill = new Date(new Date().getTime() + ms);
    while(waitTill > new Date()){};
    if (callback) callback(); else return true
}
/*
 * APPLICATION
 */
var eventCount = 0;

handler1 = function( evMsg ){
    println("  handler1 " + evMsg + " when led=" + l1.getState());
    l1.switchState();
    println( l1.getDefaultRep() );
}
handler2 = function( evMsg ){
    println("  handler2 " + evMsg + " when led=" + l1.getState());
    waitFor(1000);
    println("  handler2 ENDS" );
}
/*
 * ********************************************
 * button --- event --> handler --> led
 * ********************************************
 */
configure = function(){
    b1    = new ButtonMod.Button('b1');
    l1    = new LedMod.Led("l1");
    /*
     * We set two handlers to show that they are executed atomically
     * The button pressed only when all the handlers are terminated
     */
    b1.setHandler( handler1 );
    b1.emitter.on( "pressed", handler2 );
}
/*
 * -------------------------------------------------------
 * MAIN:
 * 1) configure a system with one button and one led
 * 2) press the button three times
 * -------------------------------------------------------
 */
mainBlsObjEmitter = function(){
    configure();
    for( var i=1; i<=3; i++){
        console.log("PRESS");
        b1.press();
    }
}

if( process.argv[1].toString().includes("blsObjEmitter") ) mainBlsObjEmitter();
```

Listing 1.16. `blsObjEmitter.js`

The output is:

```
/*
PRESS
       ButtonEmitter emits the event pressed count=1
       handler1 buttonPressed when led=false
l1||true
       handler2 buttonPressed when led=true
       handler2 ENDS
PRESS
       ButtonEmitter emits the event pressed count=2
       handler1 buttonPressed when led=true
l1||false
       handler2 buttonPressed when led=false
       handler2 ENDS
PRESS
       ButtonEmitter emits the event pressed count=3
       handler1 buttonPressed when led=false
l1||true
       handler2 buttonPressed when led=true
       handler2 ENDS
 */
```

Listing 1.17. `blsObjEmitter.js`

To run this code within a HTML page, we should run a Node.js application within a browser. Since it is not currently possible, we can overcome the problem by using the |browserify| utility.

```
<html>
<head>
<script src="./Led.js"></script>
<script src="./ButtonEmitterBro.js"></script>
</head>
</html>

<body>
<h3>blsObjEmitter</h3>
<!-- A LED -->
<div id="l1GuiId" style="height:20px; width:3%; position: absolute; background-color:#00FF33"></div>
<!-- OUTPUT Area
<div id="outView" ></div>
 -->
 <br/><br/>
<!-- BUTTON : a click calls b1.press() -->
<div>
<button onclick="b1.press()">BUTTON</button>
</div>

<script>
var hc = 1;
var n = 1;
var handler2 = function( evMsg ){
    println("  handler2 " + evMsg );
}
var control = function(msg){
    println("control msg=" + msg +" when led=" + l1.getState() + " n=" + n++);
    l1.switchState();
    l1.showGuiRep();
}
var configure = function (){
    //configure the system
    l1 = new Led( "l1","l1GuiId" );
    b1 = new Button( 'b1' );
    l1.showGuiRep();
    b1.setHandler( control );
    b1.setHandler( handler2 );
}
configure();
</script>

</body>
</htnl>
```

Listing 1.18. `blsObjEmitter.html`

4 JavaScript on Raspberry

The library onoff provides GPIO access and interrupt detection with Node.js on Linux boards like the Raspberry.

```
sudo npm install -g onoff
```

4.1 Led

The following system blinks a led by calling a function that turns on/off the every 500 msecs.

```
/*
 * ====================================
 * ledGpio.js
 * ====================================
 */
// (1) Import the onoff library
// (2) Initialize pin 25 to be an output pin
// (3) This interval will be called every 2 seconds
// (4) Synchronously read the value of pin 25 and transform 1 to 0 or 0 to 1
// (5) Asynchronously write the new value to pin 25
// (6) Listen to the event triggered on CTRL+C
// (7) Cleanly close the GPIO pin before exiting

var onoff = require('onoff');       //(1)

var Gpio = onoff.Gpio;
var led = new Gpio(25, 'out');      //(2)


interval = setInterval(function () {  //(3)
  var value = (led.readSync() + 1) % 2; //(4)
  led.write(value, function() {     //(5)
    console.log("Changed LED state to: " + value);
  });
}, 500);

process.on('SIGINT', function () {   //(6)
  clearInterval(interval);
  led.writeSync(0);                 //(7)
  led.unexport();
  console.log('Bye, bye!');
  process.exit();
});

interval;
```

Listing 1.19. `ledGpio.js`

4.2 Button

Let us introduce now a Button that 'reacts' to interrupts sent when the related pin (24) changes its value:

```
/*
 * ====================================
 * gpioOnOffbutton.js
 * See https://github.com/fivdi/onoff
 * ====================================
*/
var onoff = require( 'onoff' );
var Gpio     = onoff.Gpio;
var buttonPin = 24;
var ledPin   = 25;

var button = new Gpio(buttonPin, 'in', 'both');
var led   = new Gpio(ledPin, 'out');

/*
Watch for hardware interrupts on the GPIO.
```

```
The edge argument that was passed to the constructor
determines which hardware interrupts to watch for.
*/
button.watch(function (err, level) {
  if (err) {
    throw err;
  }
  console.log('Interrupt level =' + level);
  led.writeSync(level);
});

//Read GPIO value asynchronously.
button.read( function(value){
   console.log('read value= ' + value);
});

process.on('SIGINT', function () {
  led.writeSync(0);
  led.unexport();
  button.unexport();
  console.log('Bye, bye!');
  process.exit();
});

//=======================================================
function showButton() {
    console.log("button=" + button.readSync() );
}

for( i=1; i<=10; i++){
    setTimeout(showButton, 1000*i);
}

console.log('Waiting for clicks on pin=' + buttonPin);
```

Listing 1.20. `gpioOnOffbutton.js`

The interrupt callback implements the business logic, since it directly handles the led.

Note that there is also a function(`showButton`) that reads the current value of the button pin. This function is executed 10 times, every second, by using `setTimeout`.

5 ButtonLed: an oop version

In the following version, we implement the ButtonLed system by starting from an `project architecture` in which:

- The Led is an object that can be implemented in two basic ways: as a mock object or using a physical device (on a RaspberryPi).
- To reduce technology-dependecy we suppose that any *concrete* led must provide two basic operations: `turnOn()` and `turnOff()`.
- The Button is a GOF-observable object that can also emit NodeJs events. The business logic is delegated to a Button operation (`press()`) that notifies the observers and emits an event with `id=pressed`.
- The Led can be handled by objects working as observers, or by any event-listener registered at the Button as an emitter.

5.1 The Button

A Button is a device that can be implemented in several ways on different platforms. In a `oo` design we introduce first of all a technology-independent definition of a Button as an observable device.

5.1.1 The High-level Button .

The code that follows defines a Button as a technology-independent observable object able to notify registered observers and to emit events:

```
/*
* ===============================================
* ButtonObservable.js
* GOAL:
*   define a Button as a logical observable entity
*   that can call registered observers / functions
*   and that can rise (NodeJs) events
* ===============================================
*/
/*
 * ********************************************
 * Observable prototype
 * ********************************************
 */
Observable = function(){
    this.nobs       = 0;
    this.nobsfunc   = 0;
    this.observerFunc = [ ];
    this.observer   = [ ];
    this.register   = function(obs){
        //println(" Observable register " + this.nobs);
        this.observer[this.nobs++] = obs;
    }
    this.registerFunc = function(func){
        //println(" Observable registerFunc " + this.nobs + " " + func);
        this.observerFunc[this.nobsfunc++] = func;
    }
    this.notify = function(){
        for(var i=0;i < this.observer.length;i++){
            //console.log(" Observable update " + this.observer[i] );
            this.observer[i].update();
        }
        for(var i=0;i < this.observerFunc.length;i++){
            //console.log(" Observable calls " + this.observerFunc[i] );
            this.observerFunc[i]();
        }
    }
  }
/*
 * ***********************************************************
 * Button as an 'object' that inherits from Observable
 * and works also as an event emitter.
 * ************************************************************
 */
var EventEmitter = require('events').EventEmitter;

function Button ( name ){
    this.emitter = new EventEmitter();
    this.name    = name;
    this.evId    = "pressed";
    this.count   = 0;
}

//Shared
Button.prototype     = new Observable();
Button.prototype.press = function(level){
    //console.log(" Button press " + level );
    this.notify();
    this.emitEvent() ;
}
Button.prototype.emitEvent = function(){
    console.log(" Button emits " + this.evId + " count=" + this.count++);
    this.emitter.emit(this.evId,'buttonPressed') ;
}
Button.prototype.getEmitter = function(){
    return this.emitter;
}
Button.prototype.setHandler = function( handler ){
    this.emitter.on(this.evId, handler );
}
Button.prototype.removeHandler = function( handler ){
    this.emitter.removeAllListeners(this.evId);
}


//EXPORTS
module.exports.Button = Button;
//To work is a browser, run: browserify ButtonObservable.js -o ButtonObservableBro.js
```

Listing 1.21. `ButtonObservable.js`

This version 'merges' the implementation of Button introduced in Subsection 3.3 and in Subsection 3.4.

If we call the operation `press()` of this resource, all the registered observers (objects and functions) will be updated and a Node.js event will be emitted. This operation can be directly invoked by the different concrete implementations of a Button like the physical Button of Subsection 4.2) or the clickable HTML entity of Subsection 3.3. For an example, see Section 5.3.

5.2 The Led

A Led is a device that can be implemented in several ways on different platforms. In a oo design we introduce a technology-independent definition of a Led and several implementation versions that we will 'inject' in the technology-independent code.

5.2.1 The High-level Led .

The Led as a technology-independent object can be defined as follows:

```
/*
* ====================================
* Led.js
* Led as a conventional 'object'
* ********************************************
*/
var Led = function(name, ledImpl){
    this.name     = name;
    this.ledImpl  = ledImpl;
    this.ledState = 0;
    this.turnOff();
}
Led.prototype.turnOn = function(){
    this.ledState = 1;
    this.ledImpl.turnOn();
}
Led.prototype.turnOff = function(){
    this.ledState = 0;
    this.ledImpl.turnOff();
}
Led.prototype.switchState = function(){
    this.ledState = (this.ledState + 1) % 2;
    if( this.ledState == 0 ) this.ledImpl.turnOff();
    else this.ledImpl.turnOn();
}
Led.prototype.getState = function(){
    return this.ledState;
}
Led.prototype.getName = function(){
    return this.name;
}
Led.prototype.getDefaultRep = function(){
    return this.name+"||"+ this.ledState
}
// EXPORTS
module.exports.Led = Led;
//To work is a browser, do: browserify Led.js -o LedBro.js
```

Listing 1.22. `Led.js`

This definition assumes a precise interface for any Led implementation object (`ledImpl`). The `ledImpl` object is 'injected' in the technology-independent Led at construction time.

5.2.2 The Led implementation on a PC .

The Led implementation on a PC can be defined as follows:

```
/*
* ====================================
* LedImplPc.js
* Led implementation on a PC
* ====================================
*/
var LedImplPc = function( name ){
    this.name     = name;
    this.ledState = 0;
}
LedImplPc.prototype.turnOn = function(){
    this.ledState = 1;
    console.log("LED " + this.name + " ON");
}
LedImplPc.prototype.turnOff = function(){
    this.ledState = 0;
    console.log("LED " + this.name + " OFF");
}
// EXPORTS
if(typeof document == "undefined") module.exports.LedImplPc = LedImplPc;
```

Listing 1.23. `LedImplPc.js`

5.2.3 The Led implementation on a Raspberry .

The Led implementation on a Raspberry can be defined as follows:

```
/*
* ====================================
* LedImplGpio.js
*
* Led implmentation on a RaspberryPi
* ====================================
*/
var onoff = require('onoff');
var Gpio = onoff.Gpio;

var LedImplGpio = function(name,ledpin){
    this.name     = name;
    this.ledGpio  = new Gpio(ledpin, 'out');
    this.ledState = 0;
}

LedImplGpio.prototype.turnOn = function(){
    this.ledState = 1;
    this.ledGpio.writeSync(1);
}
LedImplGpio.prototype.turnOff = function(){
    this.ledState = 0;
    this.ledGpio.writeSync(0);
}


// EXPORTS
if(typeof document == "undefined") module.exports.LedImplGpio = LedImplGpio;
```

Listing 1.24. `LedImplGpio.js`

5.3 A system

Here is the definition of a Button-Led system that can be configured to work on a PC and on a RaspberryPi.

```
/*
 * ======================================================
 * blsOop.js
 * VISION:
 *   A logic architecture should guide any software development
 * GOAL :
 *  1) build a prototype working on a pc
 *  2) test the prototype on the pc
 *  3) refactor (extend) the code so to work on a RaspberryPi
 * ======================================================
*/
var ButtonHL = require("./ButtonObservable");
var LedHL   = require("./Led");
/*
 * ----------------------------------------------------
 * BUTTON HIGH-level
 * ----------------------------------------------------
 */
var button  = new ButtonHL.Button( "b1" );
/*
 * ----------------------------------
 * CONFIGURATION ON RASPBERRYPI
 * ----------------------------------
*/
configureForRasp = function(){
    var LedOnRasp = require("./LedImplGpio");
    var onoff     = require( 'onoff' );
    var Gpio      = onoff.Gpio;
    var l1gpio    = new LedOnRasp.LedImplGpio("l1rasp", 25);
    var buttonGpio = new Gpio(24, 'in', 'both');
    var l1rasp    = new LedHL.Led("l1rasp",l1gpio);

//  button.registerFunc( function(){ l1rasp.switchState(); });   //callback
//Set an interrupt handler that calls the business logic
 buttonGpio.watch(function (err, level) {
      if (err) {   throw err;  }
      //console.log('Interrupt level=' + level + " on " + button);
      if( level == 1 ) button.press(level); //activates the observers
  });
  //Set another event handler (for the HL button as an emitter)
  button.getEmitter().on( button.evId, function(v){console.log(" %%% HL RASP event handler: event content=" + v); } )
}
/*
 * ----------------------------------
 * CONFIGURATION ON PC
 * ----------------------------------
*/
configureForPc = function(){
    var LedOnPc  = require("./LedImplPc");
    var l1pc     = new LedOnPc.LedImplPc("l1pc");
    l1           = new LedHL.Led("l1pc",l1pc);
    /*
     * handler of the event 'pressed' emitted by the button
     */
    button.setHandler(
        function(msg){
            console.log(" configureForPc handler msg=" + msg +" when led=" + l1.getState());
            l1.switchState();
    } );
     //Set another event handler (for the HL button as an emitter)
    button.getEmitter().on( button.evId,
        function(v){console.log(" %%% HL PC event handler: event content=" + v); } )
}

/*
 * CTRL-C handling
 */
process.on('SIGINT', function () {
  ledGpio.writeSync(0);
  ledGpio.unexport();
  buttonGpio.unexport();
  console.log('Bye, bye!');
  process.exit();
});

//Main
console.log('blsOop STARTS' );
configureForPc();
//configureForRasp();
//RAPID CHECK (working also on a RaspberryPi)
for( i=1; i<=5; i++ ) button.press();
console.log('blsOop ENDS' );

//EXPORTS
if(typeof document == "undefined") module.exports.button = button; //USED BY HttpServerBls.js
```

Listing 1.25. `blsOop.js`

5.4 Web interface

Let us create a Node HTTP server that does a 'button press' each time a client connects to it:

```
/*
* ====================================
* HttpServerBls.js
* ====================================
*/
var http   = require("http");
var blsMod = require("./blsOop");

//blsMod.configureForPc();

http.createServer(function(request, response) {
    //The request object is an instance of IncomingMessage (a ReadableStream and it's also an EventEmitter)
    var method = request.method;
    var url    = request.url;
    console.log("Server request method=" + method + " url="+ url);
    if (request.method === 'GET' && request.url === '/') {
        response.writeHead(200, {"Content-Type": "text/plain"});
        response.write("The server calls the operation press of button");
        blsMod.button.press();
        response.end();
    }
}).listen( 8080, function(){ console.log('bound to port 8080');} );
```

Listing 1.26. *HttpServerBls.js*

5.5 The BLS as a qa model

Let us conclude this set of examples with a model of the BLS system expressed in the qa language/metamodel.

```
System bls0
Event fileChanged : fileChanged(FNAME,CONTENT) //emitted by watchFileInDir
Context ctxBls0 ip [ host="localhost" port=8029 ] //-g cyan
//ASSUMPTION: A Node.js HttpServerCrud writes 'click' on the file sharedFiles/cmd.txt
QActor qabls0led context ctxBls0 -g yellow{
    Plan init normal [
        println( "qabls0led STARTS" ) ;
        actorOp createLedGui
    ]
    switchTo work

    Plan work [
        println("qabls0led waits")
    ]
    transition stopAfter 600000
        whenEvent fileChanged : fileChanged(F,press) do actorOp ledSwitch
        finally repeatPlan
}
QActor qafilewatcher context ctxBls0 {
    Plan init normal [
        actorOp watchFileInDir("./sharedFiles") //C:/repoGitHub/it.unibo.blsNode.qa/sharedFiles
    ]
    finally repeatPlan
```

Listing 1.27. *bls0.qa*

The system is composed by the actor qabls0led that:

1. creates a Led implementation as a GUI on a PC;
2. activates (in asynchronous way) a watcher for the file sharedFiles/cmd.txt;
3. waits for the fileChanged event emitted when an external (Node.js) program writes (the string 'press') on the file sharedFiles/cmd.txt
4. calls the operation ledSwitch written by the application designer to change the color of the LED-GUI.

This is not the best way to deal with devices in a oop style (see the Section 3). However, our main goal here is to show how our Led (GUI) can be turned on-off by commands sent via a browser and handled by a Node.js HTTP server:

```
/*
* =====================================
* HttpServerWriteOnFile.js
* =====================================
*/
var http   = require("http");
var fs     = require('fs');
var path   = require('path');
var file   = path.join(process.cwd(), '../sharedFiles/cmd.txt');

http.createServer(function(request, response) {
    //The request object is an instance of IncomingMessage (a ReadableStream and it's also an EventEmitter)
    var method = request.method;
    var url    = request.url;
    console.log("Server request method=" + method + " url="+ url);
    if (request.method === 'GET' && request.url === '/') {
        response.writeHead(200, {"Content-Type": "text/plain"});
        response.write("The server calls the operation press of button");
        storeData(file,"press");
        response.end();
    }
}).listen( 8080, function(){ console.log('bound to port 8080');} );

function storeData(file, newData) {
     fs.writeFile(file, newData, 'utf8', function(err) {
       if (err) throw err;
       console.log('Saved: ' + newData);
     });
    }
```

Listing 1.28. *HttpServerWriteOnFile.js*

6 A CRUD HTTP sever to handle a Led

In this section, we introduce a `Node.js` `HTTP` server that works on the port 8080 and executes typical *create, read, update, delete* (`CRDD`) actions.

Our workflow can be summarized as follows:

1. define the `CRUD` server. In handling a `PUT` request we will write information (e.g. the string `click`) in a local file (say `cmd.txt`) (Subsection 6.1);
2. test the server by using a command-line HTTP client (e.g. `curl` (https://curl.haxx.se/)) or a Java HTTP client that performs PUT operation by using an utility (the class RestClientHttp, reported in (Subsection 6.3)) defined at application level (Subsection 6.2);
3. define a QActor model of a system that perform the switch of a Led each time the file `cmd.txt` is updated (Subsection 6.4).

6.1 The HTTP-CRUD server

The server simply stores data sent with a `POST` or a `PUT` request and returns the list of stored data by answering to a `GET` request.

```
/*
* ====================================
* HttpServerCrud.js
* ====================================
*/
var http = require("http");
var dataStore = []; //Array of Buffers
var fs     = require('fs');
var path   = require('path');
var file   = path.join(process.cwd(), '../sharedFiles/cmd.txt');

http.createServer(function(request, response) {
    //The request object is an instance of IncomingMessage (a ReadableStream; it's also an EventEmitter)
    var headers  = request.headers;
    var method   = request.method;
    var url      = request.url;
    console.log('method=' + method );
    if( method == 'GET'){
        dataStore.forEach( function(v,i){
            response.write( i + ")" + v + "\n")
        });
        response.end();
    }//if GET
    if( method == 'POST' || method == 'PUT'){
        var item = '';
        request.setEncoding("utf8"); //a chunk is a utf8 string instead of a Buffer
        request.on('error', function(err) {
            console.error(err);
          });
        request.on('data', function(chunk) { //a chunck is a byte array
            item = item + chunk;
          });
        request.on('end', function() {
                //dataStore = Buffer.concat(dataStore).toString();
                dataStore.push(item);
                console.log("dataStore=" + dataStore);
                buildResponse( url, method, response );
          });
    }//if POST PUT
}).listen( 8080, function(){ console.log('bound to port 8080');} );

function buildResponse( url, method, response ){
        response.on('error', function(err) { console.error(err); });
        response.statusCode = 200;
        response.setHeader('Content-Type', 'application/json');
        //response.writeHead(200, {'Content-Type': 'application/json'}) //compact form
        var responseBody = {
          //headers: headers, //comment, so to reduce output
          method: method,
        url:   url,
        dataStore:  dataStore
      };
      response.write( JSON.stringify(responseBody) );
      storeData(file,"press");
      response.end();
      //response.end(JSON.stringify(responseBody)) //compact form
}
console.log('Server running on 8080');


function storeData(file, newData) {
     fs.writeFile(file, newData, 'utf8', function(err) {
       if (err) throw err;
       console.log('Saved: ' + newData);
     });
   }
```

Listing 1.29. *HttpServerCrud.js*

6.2 Testing the server

We can interact with this server in several ways. One way is to use a powerful command-line HTTP-client that can be used to send requests in place of a web browser. Thus, we have to download curl (curl download win64) and then execute some command; for example:

```
curl -X PUT -d itemA http://localhost:8080
curl -X POST -d itemB http://localhost:8080
curl localhost:8080 //GET
```

Another way to interact with this server is to define a qactor application. In the model that follows, the actor qahttpclient performs a sequence of PUT operations, each followed by a GET:

```
System httpClient
Event httpinfo : httpinfo(X)  //emitted by PUT GET
Context ctxHttpClient ip [ host="localhost" port=8049 ]

QActor qahttpclient context ctxHttpClient {
Rules{
   data( a ). data( b ).  data( c ). data( d ). data( e ). data( f ).
}
   Plan init normal [
       println( "qahttpclient STARTS" ) ;
       [ ?? data(X) ] actorOp sendPut(X, 8080) else endPlan "no more data";
       actorOp sendGet( 8080 ) ;
       delay 500
   ]
   finally repeatPlan
}

QActor qahttpanswerhandler context ctxHttpClient {
```

Listing 1.30. SYSTEM *httpClient.qa*: the qahttpclient

The application code is:

```
/* Generated by AN DISI Unibo */
/*
This code is generated only ONCE
*/
package it.unibo.qahttpclient;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.qactors.QActorContext;
import it.unibo.rest.clienthttp.RestClientHttp;

public class Qahttpclient extends AbstractQahttpclient {
   public Qahttpclient(String actorId, QActorContext myCtx, IOutputEnvView outEnvView ) throws Exception{
       super(actorId, myCtx, outEnvView);
   }
```

```java
/*
 * ADDED BY THE APPLICATION DESIGNER
 */
    public void sendPut(String msg, int port) {
        println("sendPut:" + msg);
        RestClientHttp.setCtx( this.getQActorContext() );
        RestClientHttp.sendPut(msg, "http://localhost:"+port);
        RestClientHttp.setCtx( null );

    }
    public void sendGet( int port) {
        println("sendGet:" );
        RestClientHttp.setCtx( this.getQActorContext() );
        RestClientHttp.sendGet( port );
        RestClientHttp.setCtx( null );
    }
}
```

Listing 1.31. *Qahttpclient.java*

6.3 An utility class for Java HTTP clients

The 'adapter' that connects our application to the `Node.js` server is written in a utility Java class:

```java
public class RestClientHttp {

private static QActorContext ctx = null;

    public static void setCtx(QActorContext curctx) {
        ctx = curctx;
    }
    public static void sendGet( int port) {
         try {
           CloseableHttpClient httpclient = HttpClients.createDefault();
           HttpGet httpGet = new HttpGet("http://localhost:"+port);
           CloseableHttpResponse response = httpclient.execute(httpGet);
//         System.out.println("RestClientHttp response=" + response);
           if (response.getStatusLine().getStatusCode() != 200) {
               throw new RuntimeException("Failed : HTTP error code : "
                  + response.getStatusLine().getStatusCode());
           }
           BufferedReader br = new BufferedReader(
                           new InputStreamReader((response.getEntity().getContent())));
           String output;
           String info = "";
           while ((output = br.readLine()) != null) {
               info = info + output;
           }
               String msg = "httpinfo('"+info+"')";
//             System.out.println("raise event" + msg);
               QActorUtils.raiseEvent( ctx, "clienthttp", "httpinfo",msg);
          } catch ( Exception e) {
           e.printStackTrace();
         }
    }

    public static int sendPut(String data, String url) {
//      System.out.println("sendPut " + url);
        int responseCode = -1;
        CloseableHttpClient httpclient = HttpClients.createDefault();
           HttpPut request = new HttpPut(url);
           StringEntity params =new StringEntity(data,"UTF-8");
           params.setContentType("application/json");
           request.addHeader("content-type", "application/json");
           request.addHeader("Accept", "*/*");
           request.addHeader("Accept-Encoding", "gzip,deflate,sdch");
           request.addHeader("Accept-Language", "en-US,en;q=0.8");
           request.setEntity(params);
           CloseableHttpResponse response;
           try {
               response     = httpclient.execute(request);
```

```
            responseCode = response.getStatusLine().getStatusCode();
            handleResponse(response);
        } catch (ClientProtocolException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return responseCode;
    }
```

Listing 1.32. UTILITY class *RestClientHttp.java*

Each operation waits for the answer from the server and transforms such an answer into an event of the form: `httpinfo : httpinfo(X)`.

These events cannot be handled by the `qahttpclient` itself, since they are always raised before that the actor begins its transition phase. Thus, in order to handle the answer events, we introduce another actor (`qahttpanswerhandler`) that handles `httpinfo` events.

```
        actorOp noOp
    ]
    transition stopAfter 10000
        whenEvent httpinfo : httpinfo(X) do println( httpinfo(X) )
        finally repeatPlan
}
```

Listing 1.33. SYSTEM *httpClient.qa*: the `qahttpanswerhandler`

6.4 A system that handles a Led

We want to create a system that perform the switch of a Led each time the *HTTP* server of Subsection 6.1 receives a `PUT` request. The server is written in `Node.js`, but we want to write first of all a QActor model. Thus, we exploit the *file watching* mechanism of the QActors that generate the event `fileChanged : fileChanged(FNAME,CONTENT)` each time the file `cmd.txt` is updated by the server.

```
System bls0
Event fileChanged : fileChanged(FNAME,CONTENT) //emitted by watchFileInDir
Context ctxBls0 ip [ host="localhost" port=8029 ] //-g cyan
//ASSUMPTION: A Node.js HttpServerCrud writes 'click' on the file sharedFiles/cmd.txt
QActor qabls0led context ctxBls0 -g yellow{
    Plan init normal [
        println( "qabls0led STARTS" ) ;
        actorOp createLedGui
    ]
    switchTo work

    Plan work [
        println("qabls0led waits")
    ]
    transition stopAfter 600000
        whenEvent fileChanged : fileChanged(F,press) do actorOp ledSwitch
        finally repeatPlan
}
QActor qafilewatcher context ctxBls0 {
    Plan init normal [
        actorOp watchFileInDir("./sharedFiles") //C:/repoGitHub/it.unibo.blsNode.qa/sharedFiles
    ]
    finally repeatPlan
}
QActor qabls0client context ctxBls0 {
    Plan init normal [
        println( "qabls0client sendPut" ) ;
        actorOp sendPut("click", 8080) ;
        delay 500
    ]
    finally repeatPlan 5
}
```

Listing 1.34. SYSTEM *bls0.qa*: the `qahttpanswerhandler`

The system can be tested as follows:

1. launch the *HTTP* server
2. run the application (`bls0.qa`)
3. run the `httpClient.qa` of Subsection 6.2

6.5 Work to do

In this system, we have uses a virtual Led represented by a GUI picture. Now we should perform the following work:

1. modify the model with reference to a logical Led
2. inject in the logical Led one of the possible Led implementations: Led as a GUI, Led as a physical device connected to Raspberry or to Arduino, Led as a remote virtual device (in Unity), etc.
3. write a testing procedure to be launched in automatic way
4. build the system and look at the `jacoco` reports