

Buttons: from models to implementations

Antonio Natali

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

Abstract. A software **system** is made of a set of **components** properly interconnected. In this work we start the design and development of a recurrent case study (*Input-Elaboration-Output*) according to a test-driven software development approach. This work is also a graceful introduction to model-driven software development and to the usage of **UML** diagrams built with proper tools (e.g. *Architect* of Sparx).

1 Introduction

Our first case study is related to the design and development of a distributed software system that enables an user to turn on some led by pressing a button:

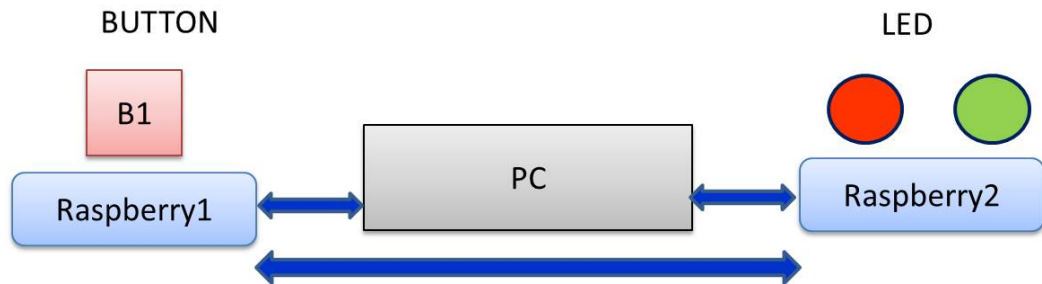


Fig. 1. Case study

Since "*there is no code without ... requirements*", let us start by defining in a precise way what the costumer intends with the words 'button' and 'led'. In fact these are the main entities that will compose (by a proper interaction) our software system and any other software system involving buttons and leds.

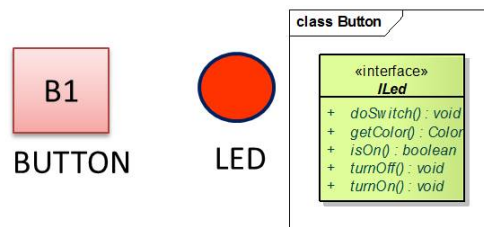


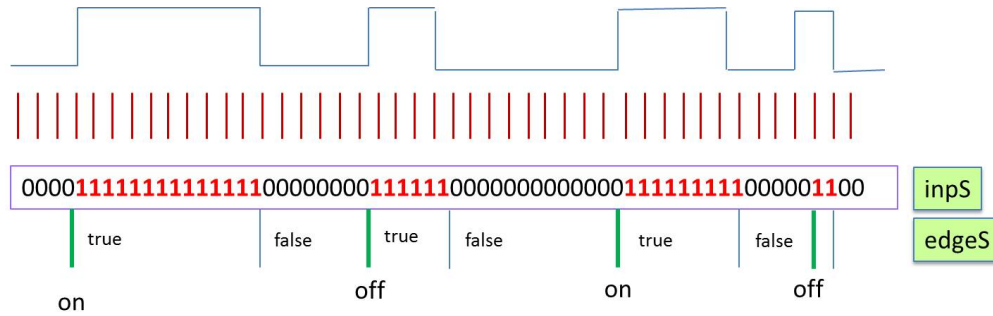
Fig. 2. System entities

The definition of the interface `ILed` according to a test-driven approach is left to the reader. In this work we will face the problem of defining the concept of 'button'.

2 Button: what is it?

2.1 A physical view

The button is a source of information that emits a wave that is sampled by some low-level entity:



The samples form a sequence of values in which each value can be modelled as a *boolean*, where **true** means "high" and **false** means "low". From this sequence of values ('*input sequence*' or **inpS**) we can find the edges that in their turn form a sequence of values called here *edge sequence* or **edgeS**. Each value of the **edgeS** sequence can be also modelled as a *boolean*, where **true** means "low to high" and **false** means "high to low". Since the button is supposed to be initially **unpressed** (the voltage level is low), the sequence **edgeS** is either empty or takes always the following form:

An output

true false true false ...

We can say that the Led is turned on N times, where N is the number of **true** in odd position in the **edgeS** sequence.

2.2 Towards a (software) model

From the *structural* point of view, a button is intended by the customer as an *atomic* entity whose *behavior* can be modelled as a *state machine* composed of two states: '**pressed**' and '**unpressed**'. The transition from the state **unpressed** to the state **pressed** is performed by some agent *external* to the software system (an user, a program, a device, etc.).

From the *interaction* point of view, the button can expose its internal state in different ways:

- by providing a *property* operation (e.g. **boolean** `isPressed()`) that returns **true** when the button is in the **pressed** state. In this way the interaction is based on "polling";
- by providing a synchronizing operation (e.g. **void** `waitPressed()`) that blocks a caller until the button transits in the **pressed** state. In this way the interaction is based conventional "procedure-call";
- by working as an *observable* according to the *observer* design pattern [1]. In this way the interaction is based on "inversion of control" and involves observers (also called "*listeners*") that must be explicitly referenced (via a "*register*" operation) by the button.
- by emitting *events* handled by an event-based support. In this way the interaction is based on "inversion of control" that involves observers (usually known as "*callbacks*") referenced by the support and not by the button itself.
- by sending *messages* handled by a message-based support. In this way the interaction is based on message passing and can follow different "patterns" (in our internal terminology we distinguish between *dispatch*, *signal*, *invitation*, *request-response*, etc.)

All these "models" could be appropriate in some software application. Thus, a very useful exercise is to define in a formal way each of these models by adopting (at the moment) a test-driven approach.

2.3 What we are going to do

Before starting, we stress the fact that our intent is not to model some specific physical "button device", but to define a *logical entity* that will be used by our application code. Any "abstraction gap" between our logical models and any specific physical button will be overcome by some proper software layer.

3 Button as a passive entity

The `IButtonPolling` interface captures the idea of button as a passive entity that allows a caller to check if it pressed (*isPressed*):

```

1 package it.unibo.domain.interfaces;
2
3 public interface IButtonPolling {
4     public boolean isPressed() ; //property
5     /*
6      * Defined for simulation purposes
7      */
8     public void press(); //modifier
9     public void release(); //modifier
10 }

```

Listing 1.1. `IButtonPolling.java`

The "*press, release*" operations are introduced to allow our software to automatically execute the tests over a button implementation.

3.1 A first test plan

```

1 package it.unibo.domain.tests;
2 import static org.junit.Assert.*;
3 import org.junit.After;
4 import org.junit.Before;
5 import org.junit.Test;
6
7 import it.unibo.button.ButtonPollingSimulator;
8 import it.unibo.domain.interfaces.IButtonPolling;
9
10 public class ButtonPollingSimulatorTest {
11     protected IButtonPolling button;
12
13     @Before
14     public void setUp() throws Exception{
15         System.out.println(" *** setUp " );
16         button = new ButtonPollingSimulator(); //TODO new ...; or factory
17     }
18
19     @After
20     public void tearDown() throws Exception{
21         System.out.println(" *** tearDown " );
22     }
23
24     @Test
25     public void testCreation(){
26         System.out.println(" testCreation ... " );
27         try {
28             assertTrue("testCreation", ! button.isPressed() );
29         } catch (Exception e) {
30             fail("testCreation " + e.getMessage() );
31         }
32     }
33
34     @Test
35     public void testRelease(){
36         System.out.println(" testRelease ... " );
37         button.release();
38         try {
39             assertTrue("testReset", ! button.isPressed() );
40         }
41     }
42 }

```

```

37     } catch (Exception e) {
38         fail("testRelease " + e.getMessage() );
39     }
40 }
41 @Test
42 public void testPressed(){
43     System.out.println(" testPressed ... " );
44     button.press();
45     try {
46         assertTrue("testReset", button.isPressed() );
47     } catch (Exception e) {
48         fail("testPressed " + e.getMessage() );
49     }
50 }
51 @Test
52 public void testPressedProtocol(){
53     try {
54         System.out.println(" testPressed again ... " );
55         button.press();
56         assertTrue("testReset", button.isPressed() );
57         button.release();
58         assertTrue("testReset", ! button.isPressed() );
59     } catch (Exception e) {
60         fail("testPressed " + e.getMessage());
61     }
62 }
63 }

```

Listing 1.2. ButtonPollingSimulatorTest.java

To make the test executable, we introduce a ButtonPollingSimulatorTest as a Mock button.

```

1 package it.unibo.button;
2 import it.unibo.domain.interfaces.IButtonPolling;
3
4 public class ButtonPollingSimulator implements IButtonPolling {
5     protected boolean pressed = false;
6
7     public ButtonPollingSimulator() {
8         System.out.println("ButtonPollingSimulator CREATED " );
9     }
10    @Override
11    public void release() {
12        pressed = false;
13    }
14    @Override
15    public void press() {
16        pressed = true;
17    }
18    @Override
19    public boolean isPressed() {
20        return pressed;
21    }
22 }

```

Listing 1.3. ButtonPollingSimulatorTest.java

3.2 Work TODO

1. Define the class `Button` that implements `IButton` by using the standard input device `System.in` in the following way: the button is pressed when the user hits the "carriage return" key.
2. Experiment the usage of some UML modelling tool (e.g. *Architect* of Sparx) to create a graphical representation of the interfaces and exploit source (reverse) engineering facilities.

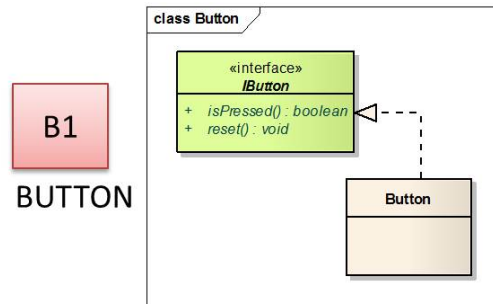


Fig. 3. IButton and Button

4 Button as a producer

Let us extend the interface as follows:

```

1 package it.unibo.domain.interfaces;
2
3 public interface IButtonSynch {
4     public boolean isPressed(); //property
5     public void waitPressed(); //synchronizer
6     public void reset(); //modifier
7 }
  
```

Listing 1.4. IButtonSynch.java

The operation *waitPressed* is introduced to avoid "polling": it blocks the caller until the button state transits in 'pressed' mode.

The test unit is extended with a new test-plan:

```

1 @Test
2 public void testWaitPressed(){
3     try {
4         // ButtonSynch.debug = true;
5         for( int i=1; i<=3; i++){
6             button.waitPressed();
7             System.out.println("testWaitPressed step= " + i );
8             assertTrue("testWaitPressed", button.isPressed() );
9         }
10    } catch (Exception e) {
11        fail("testWaitPressed " + e.getMessage());
12    }
13 }
14 }
  
```

Listing 1.5. ButtonSynchTest.java

```

      testWaitPressed
      public void testWaitPressed(){
          try {
              for( int i=1; i<=3; i++){
                  button.waitPressed();
                  System.out.println("testWaitPressed step= " + i );
                  assertTrue("testWaitPressed", button.isPressed() );
              }
          } catch (Exception e) {
              fail("testWaitPressed " + e.getMessage());
          }
      }
  
```

To make the test executable, we introduce a `ButtonSynch` that implements `IButtonSynch` by using the standard input device `System.in` in the following way: the button is pressed when the user hits the "carriage return" key..

```

1 package it.unibo.button;
2 import it.unibo.domain.interfaces.IButtonSynch;
3 import java.io.IOException;
4
5 public class ButtonSynch implements IButtonSynch {
6     public static boolean debug = false;
7     protected boolean pressed = false;
8
9     @Override
10    public void reset() {
11        pressed = false;
12    }
13
14    @Override
15    public boolean isPressed() {
16        return pressed;
17    }
18
19    protected void lookAtPressed() throws IOException {
20        pressed = false;
21        System.out.println("Button PRESS "); // ... 13 10
22        int n = System.in.read();
23        // consume until the end of line
24        while (n != 10) {
25            n = System.in.read();
26        }
27        // n = System.in.read(); // consume 10
28    }
29
30    public void waitPressed() {
31        try {
32            reset();
33            if (debug) {
34                pressed = true;
35                return;
36            }
37            reset();
38            this.lookAtPressed();
39            pressed = true;
40            // Thread.sleep( IConstants.PRESSTIME );
41            // pressed = false;
42        } catch (Exception e) {
43            e.printStackTrace();
44        }
45    }
46 }

```

Listing 1.6. `ButtonSynch.java`

Note: The class `Button` is supposed to have a boolean property called `debug` that, when set, allows us to run the tests without the presence of an user.

5 Button as an observable entity

A button is defined here by the `IButtonActiveObservable` interface as an *active* entity that updates all its registered observers each time it is pressed. The entity starts its job as soon as it is created (*isRunning* return **true**) and terminates when the *stop* operation is called (*isRunning* return **false**).

```

1 package it.unibo.domain.interfaces;
2 import it.unibo.is.interfaces.IObservable;
3 /*
4  * An object that implements IButtonObservable is an active entity that updates
5  * all its registered observers * each time it is pressed.
6  * The entity starts its job as soon as it is created (isRunning return true)

```

```

7  * and terminates when the stop operation is called (isRunning return false).
8  */
9  public interface IButtonActiveObservable extends IObservable{
10     public void start(); //modifier
11     public void stop(); //modifier
12     public boolean isRunning(); //property
13 }

```

Listing 1.7. IButtonActiveObservable.java

```

1  package it.unibo.is.interfaces;
2
3  public interface IObservable {
4      public void addObserver(IObserver arg0); //modifier
5  }

```

Listing 1.8. IObservable.java

```

1  package it.unibo.is.interfaces;
2  import java.util.Observable;
3  import java.util.Observer;
4
5  public interface IObserver extends Observer {
6      public void update(Observable arg0, Object arg1); //modifier
7  }

```

Listing 1.9. IObserver.java

The interface *java.util.Observer* is defined as follows:

```

Interface java.util.Observer
public interface java.util.Observer {
    public void update(java.util.Observable arg0, java.lang.Object arg1);
}

```

The following test unit better defines the behavior of each operation:

```

public class ButtonObservableTest {
    protected IButtonObservable button;
    protected IButtonObserver buttonObserver;
    @Before
    public void setUp() throws Exception{
        System.out.println(" *** setUp " );
        button = null; //TODO new ...;           or factory
        buttonObserver = null; //TODO new ...;       or factory
        button.register( buttonObserver );
        button.start(); //starts the active object
    }
    @After
    public void tearDown() throws Exception{
        System.out.println(" *** tearDown " );
        button.stop(); //stops the active object
    }
    @Test
    public void testCreation(){
        try {
            assertTrue("testCreation", button.isRunning() );
        } catch (Exception e) {
            fail(" " + e.getMessage());
        }
    }
    @Test
    public void testPressed(){
        try {
            ButtonObservable.debug = true;
            Thread.sleep(1000);
            assertTrue("testPressed", buttonObserver.getNumOfUpdate() == ButtonObservable.MAXNUMOFPRESS );
        } catch (Exception e) {

```



```

        fail("testPressed " + e.getMessage());
    }
}

```

Note: The class `ButtonObservable` is supposed to have a boolean property called `debug` that, when set, allows us to run the button for a fixed number of times (`ButtonObservable.MAXNUMOFPRESS`) without the presence of an user.

6 An implementation

Let us report here a possible implementation of the class *ButtonObservableSimulator*. The code is written by introducing a set of internal operations, in order to improve code readability and modifiability. Let us start from the *public* operations:

```

1  package it.unibo.button;
2  import java.io.IOException;
3  import java.util.Iterator;
4  import java.util.Observable;
5  import java.util.Vector;
6  import it.unibo.domain.interfaces.IButtonActiveObservable;
7  import it.unibo.domain.interfaces.IConstants;
8  import it.unibo.is.interfaces.IObserver;
9
10
11  /*
12   * The ButtonActiveObservable generates the calls to its registered listeners.
13   */
14  public class ButtonObservableSimulator extends Observable implements IButtonActiveObservable {
15      public static boolean debug = false;
16      protected Vector<IObserver> obs = new Vector<IObserver>();
17      protected boolean running = false;
18      protected Thread myThread = null;
19
20      public ButtonObservableSimulator() {
21          myThread = createThread();
22      }
23
24      @Override
25      public void addObserver(IObserver arg0) {
26          obs.add(arg0);
27      }
28
29      @Override
30      public boolean isRunning() {
31          return running;
32      }
33
34      @Override
35      public void start() {
36          myThread.start();
37          running = true;
38      }
39
40      @Override
41      public void stop() {
42          running = false;
43      }

```

Listing 1.10. `ButtonObservableSimulator.java`

The constructor binds the `myThread` variable to a new `Thread` that performs the button's job:

```

1  protected Thread createThread(){
2      return new Thread() {
3          public void run() {
4              for (int i = 1; i <= IConstants.MAXNUMOFPRESS; i++) {

```

```

5         if (!running) break;
6         buttonPressed();
7         updateObservers(true);
8         buttonUnPressed();
9         updateObservers(false);
10    } //for
11    running = false;
12  }
13 };
14 }

```

Listing 1.11. The createThread operation

The internal thread works for a number of times defined by the constant `IConstants.MAXNUMOFPRESS`; it delegates to two other operations the task to check when the button is first "pressed" and then "unpressed":

```

1  protected void buttonPressed(){
2      if (!debug) lookAtInput();
3  }
4  protected void buttonUnPressed(){
5      delay( IConstants.PRESSTIME );
6  }
7  protected void delay( int dt){
8      try {
9          Thread.sleep( dt );
10     } catch (InterruptedException e) {
11         e.printStackTrace();
12     }
13 }

```

Listing 1.12. The button(Un)Pressed operation

From the code we see that at each state modification of the button the *updateObservers* operation is called with argument `true` when the button is pressed and `false` when it is unpressed. Moreover, the button becomes unpressed after fixed amount of time defined by the constant `PRESSTIME` defined in the interface `IConstants`:

```

1  protected void updateObservers(boolean on) {
2      Iterator<IObserver> itObs = obs.iterator();
3      while (itObs.hasNext()) {
4          IObserver observer = itObs.next();
5          observer.update(this, on);
6      }
7  }

```

Listing 1.13. The updateObservers operation

Finally, let us define the *lookAtInput* operation that checks if the button is pressed by reading the standard input device:

```

1  public void lookAtInput() {
2      try {
3          System.out.println("Button PRESS "); // ... 13 10
4          int n = System.in.read();
5          // consume until the end of line
6          while ( n != 10 ) {
7              n = System.in.read();
8          }
9      } catch (IOException e) {
10         e.printStackTrace();
11     }
12 }

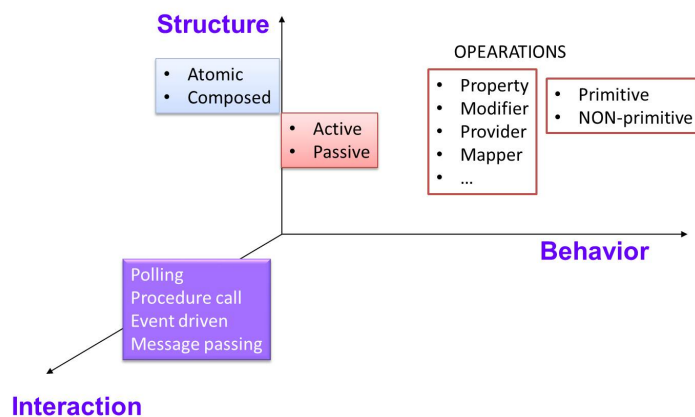
```

Listing 1.14. The lookAtInput operation

The operations *buttonPressed* and *buttonUnPressed* is the only parts of our code that is "technology dependent": Thus these operations must be changed according to the concrete nature of the button.

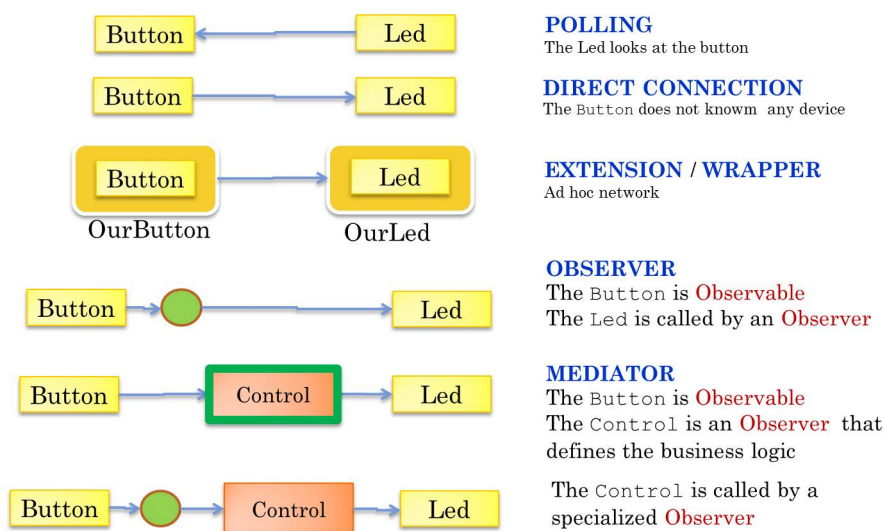
6.1 Overview

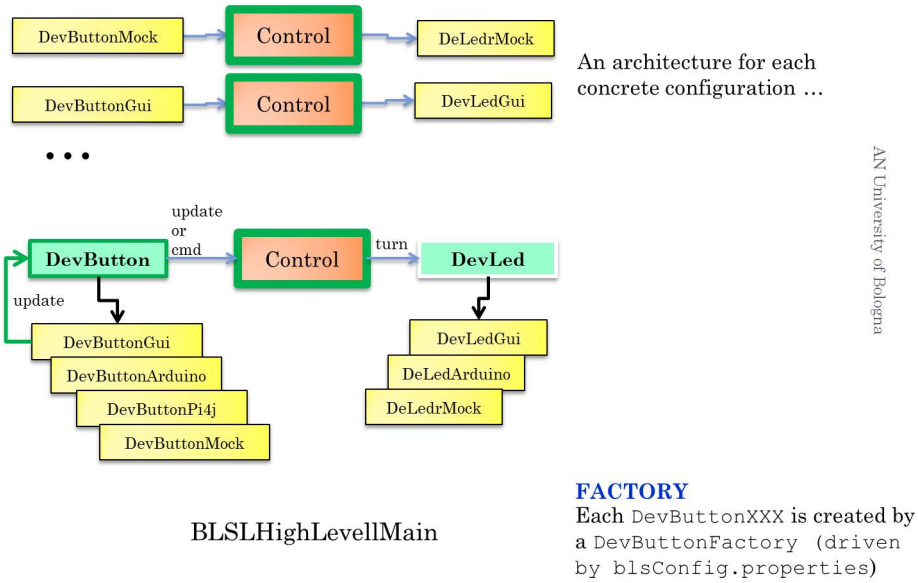
Let us report here a picture to recall the general conceptual working space so far introduced:



7 Towards a ButtonLed system

The following picture is an informal representation of possible architectural scenarios;





References

1. E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Computing Series. Addison-Wesley Professional, november 1994.
2. A. Natali. Introduction to the contact system. <http://edu222.deis.unibo.it/contact>.