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:

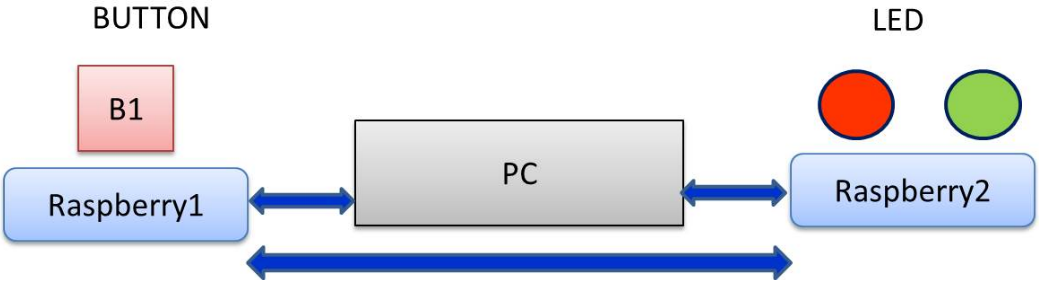


Fig. 1. Case study

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.

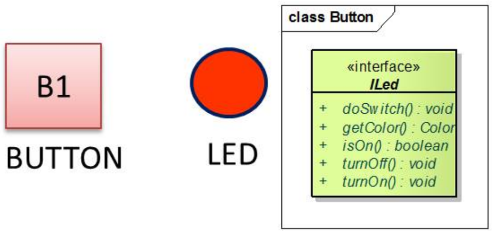


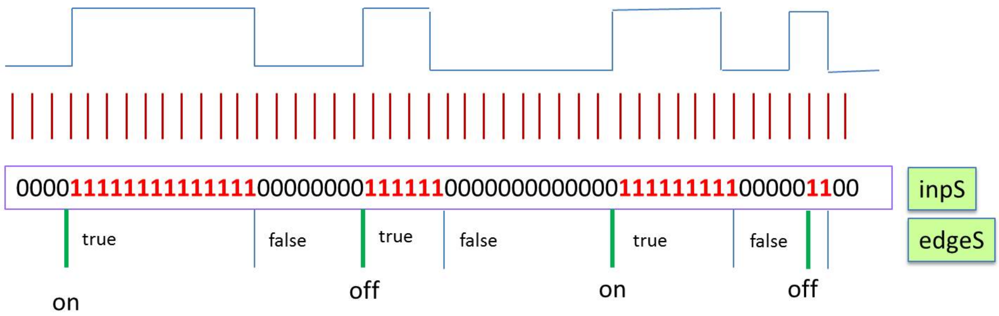
Fig. 2. System entitities

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*):

```java
package it.unibo.domain.interfaces;

public interface IButtonPolling {
    public boolean isPressed() ; //property
/*
 * Defined for simulation purposes
 */
    public void press();  //modifier
    public void release(); //modifier
}
```

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

```java
package it.unibo.domain.tests;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import it.unibo.button.ButtonPollingSimulator;
import it.unibo.domain.interfaces.IButtonPolling;

public class ButtonPollingSimulatorTest {
protected IButtonPolling button;

@Before
    public void setUp() throws Exception{
        System.out.println(" *** setUp " );
        button = new ButtonPollingSimulator(); //TODO new ...; or factory
    }
@After
    public void tearDown() throws Exception{
        System.out.println(" *** tearDown " );
    }
@Test
    public void testCreation(){
        System.out.println("  testCreation ... " );
            try {
                assertTrue("testCreation", ! button.isPressed() );
            } catch (Exception e) {
                fail("testCreation " + e.getMessage() );
            }
    }
@Test
public void testRelease(){
    System.out.println("  testRelease ... " );
    button.release();
    try {
        assertTrue("testReset", ! button.isPressed() );
```

```
    } catch (Exception e) {
        fail("testRelease " + e.getMessage() );
    }
}
@Test
public void testPressed(){
        System.out.println("  testPressed ... " );
        button.press();
        try {
            assertTrue("testReset", button.isPressed() );
        } catch (Exception e) {
            fail("testPressed " + e.getMessage() );
        }
}
@Test
public void testPressedProtocol(){
    try {
        System.out.println("  testPressed again ... " );
        button.press();
        assertTrue("testReset", button.isPressed() );
        button.release();
        assertTrue("testReset", ! button.isPressed() );
    } catch (Exception e) {
        fail("testPressed " + e.getMessage());
    }
}
}
```

Listing 1.2. ButtonPollingSimulatorTest.java

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

```
package it.unibo.button;
import it.unibo.domain.interfaces.IButtonPolling;

public class ButtonPollingSimulator implements IButtonPolling {
protected boolean pressed = false;

    public ButtonPollingSimulator() {
        System.out.println("ButtonPollingSimulator CREATED " );
    }
    @Override
    public void release() {
        pressed = false;
    }
    @Override
    public void press() {
        pressed = true;
    }
    @Override
    public boolean isPressed() {
        return pressed;
    }
}
```

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.

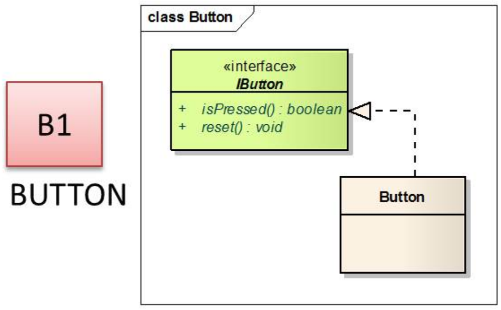


Fig. 3. IButton and Button

4 Button as a producer

Let us extend the interface as follows:

```java
package it.unibo.domain.interfaces;

public interface IButtonSynch {
    public boolean isPressed(); //property
    public void waitPressed(); //synchronizer
    public void reset();  //modifier
}
```

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:

```java
@Test
    public void testWaitPressed(){
        try {
//          ButtonSynch.debug = true;
            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());
        }
    }
}
```

Listing 1.5. `ButtonSynchTest.java`

testWaitPressed

```java
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..

```java
package it.unibo.button;
import it.unibo.domain.interfaces.IButtonSynch;
import java.io.IOException;

public class ButtonSynch implements IButtonSynch {
public static boolean debug = false;
    protected boolean pressed = false;

    @Override
    public void reset() {
        pressed = false;
    }

    @Override
    public boolean isPressed() {
        return pressed;
    }

    protected void lookAtPressed() throws IOException {
        pressed = false;
        System.out.println("Button PRESS "); // ... 13 10
        int n = System.in.read();
        // consume until the end of line
        while (n != 10) {
            n = System.in.read();
        }
//      n = System.in.read(); // consume 10
    }

    public void waitPressed() {
        try {
            reset();
            if (debug) {
                pressed = true;
                return;
            }
            reset();
            this.lookAtPressed();
            pressed = true;
//          Thread.sleep( IConstants.PRESSTIME );
//          pressed = false;
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

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`).

```java
package it.unibo.domain.interfaces;
import it.unibo.is.interfaces.IObservable;
/*
 * An object that implements IButtonObservable is 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)
```

```java
 * and terminates when the stop operation is called (isRunning return false).
 */
public interface IButtonActiveObservable extends IObservable{
    public void start(); //modifier
    public void stop(); //modifier
    public boolean isRunning(); //property
}
```

Listing 1.7. `IButtonActiveObservable.java`

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

public interface IObservable {
    public void addObserver(IObserver arg0); //modifier
}
```

Listing 1.8. `IObservable.java`

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

public interface IObserver extends Observer {
    public void update(Observable arg0, Object arg1); //modifier
}
```

Listing 1.9. `IObserver.java`

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

Interface java.util.Observer

```java
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:

```java
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:

```java
package it.unibo.button;
import java.io.IOException;
import java.util.Iterator;
import java.util.Observable;
import java.util.Vector;
import it.unibo.domain.interfaces.IButtonActiveObservable;
import it.unibo.domain.interfaces.IConstants;
import it.unibo.is.interfaces.IObserver;


/*
 * The ButtonActiveObservable generates the calls to its registered listeners.
 */
public class ButtonObservableSimulator extends Observable implements IButtonActiveObservable {
public static boolean debug = false;
protected Vector<IObserver> obs = new Vector<IObserver>();
protected boolean running = false;
protected Thread myThread = null;

    public ButtonObservableSimulator() {
         myThread = createThread();
    }

    @Override
    public void addObserver(IObserver arg0) {
        obs.add(arg0);
    }

    @Override
    public boolean isRunning() {
        return running;
    }

    @Override
    public void start() {
        myThread.start();
        running = true;
    }

    @Override
    public void stop() {
        running = false;
    }
```

Listing 1.10. `ButtonObservableSimulator.java`

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

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

```java
                if (!running) break;
                buttonPressed();
                updateObservers(true);
                buttonUnPressed();
                updateObservers(false);
            }//for
            running = false;
        }
    };
}
```

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":

```java
protected void buttonPressed(){
    if (!debug) lookAtInput();
}
protected void buttonUnPressed(){
    delay( IConstants.PRESSTIME );
}
protected void delay( int dt){
    try {
        Thread.sleep( dt );
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

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`:

```java
protected void updateObservers(boolean on) {
    Iterator<IObserver> itObs = obs.iterator();
    while (itObs.hasNext()) {
        IObserver observer = itObs.next();
        observer.update(this, on);
    }
}
```

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:

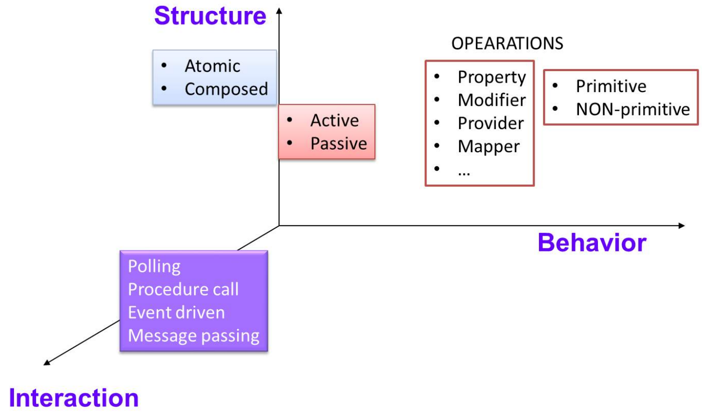
```java
public void lookAtInput() {
    try {
        System.out.println("Button PRESS "); // ... 13 10
        int n = System.in.read();
        // consume until the end of line
        while ( n != 10 ) {
            n = System.in.read();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

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.

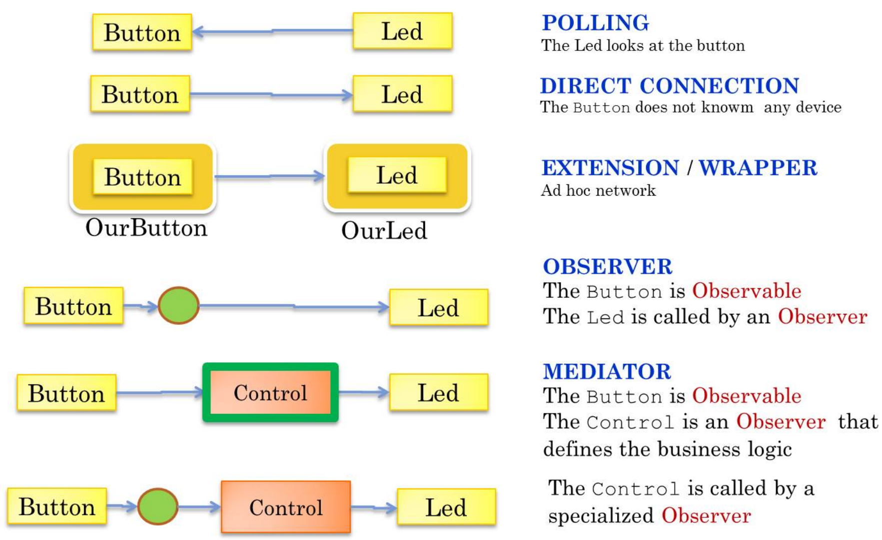
6.1 Overview

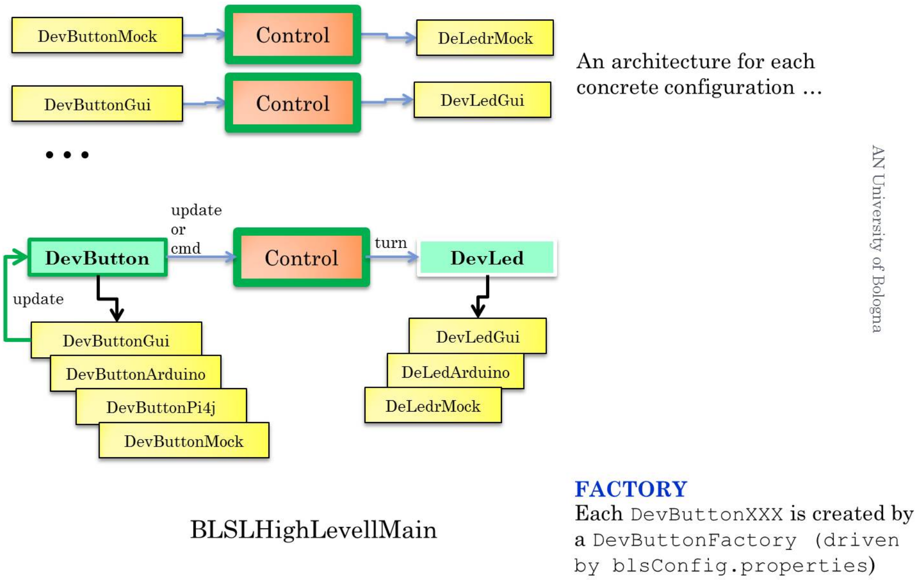
Let us report here a picture to recall the general conceptual working space so far introduced:



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.