Handling (sensor) inputs: the ButtonCounterLed system

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*) as 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

This case study is related to the design and development of a distributed software system (called ***ButtonCounterLed***) that enables an user to increment a counter and turn on/off some led by pressing a button. For example (the *Raspberry* (sub)systems could be replaced by *Arduino*, *Beaglebone*, etc.):

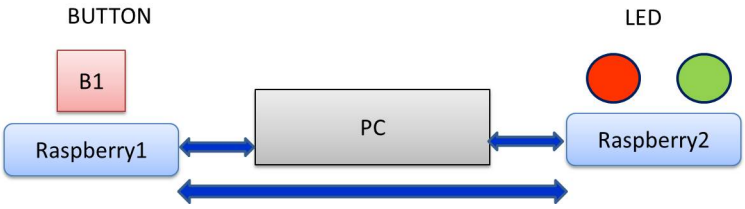


Fig. 1. Case study

The ButtonCounterLed represents a typical software system that includes three main functionalities

- *input* (one or more buttons)
- *elaboration* (a counter)
- *output* (leds, buzzer, etc.)

Since "*there is no code without project, no project without problem analysis and no problem 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 our software system and any other software system involving buttons, counters and leds.

1.1 A first domain model

The ILed interface captures the idea of a passive entity that provides methods to switch/turn a led light:

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

public interface ILed extends ILedGpio{
    public void doSwitch();//modifier
}
```

Listing 1.1. The interface `ILed.java`

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

import java.awt.Color;

public interface ILedGpio {
    public void turnOn(); // modifier

    public void turnOff(); // modifier

    public Color getColor(); // provider

    public boolean isOn(); // property
}
```

Listing 1.2. The interface `ILedGpio.java`

A more precise definition, according to a *test-driven approach*,is left to the reader.

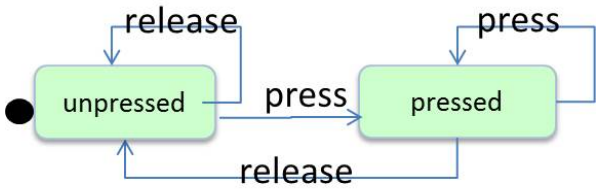
In the rest of this section we will concentrate our attention on the concept of 'button' as a special case of input device.

2 Modelling input devices

Our intent here 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.

In fact, software reusability is not only related to code, but it is also (mainly) related to concepts and logic design.

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 our 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 "*addObserver/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.)

Each of these interaction models could be appropriate in some software application.

2.1 Button as a passive, observable entity

The `IDevButton` interface aims at capturing the idea of a 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 IDevButton extends IDevInput{
4      public boolean isPressed() throws Exception;
5      public void setDevImpl(IDevInputImpl buttonImpl);
6  }
```

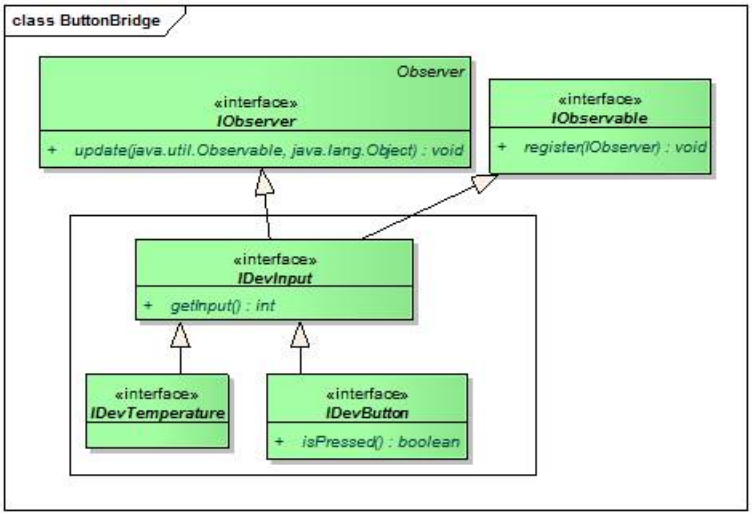
Listing 1.3. The interface `IDevButton.java`

The operation `setDevImpl` injects in an implementation in a logical device, according to the bridge pattern of [1] (see Subsection 3.1). The concept of logical button `IDevButton` is viewed as special case of a more general concept of "*Input Device*", as expressed by the interface `IDevInput`:

```
1  package it.unibo.domain.interfaces;
2  import it.unibo.is.interfaces.IObservable;
3  import it.unibo.is.interfaces.IObserver;
4
5  public interface IDevInput extends IObserver, IObservable{
6      public String getName();
7      public int getInput() throws Exception;
8      public String getDefaultRep() ;
9  }
```

Listing 1.4. The interface `IDevInput.java`

A device of class `IDevInput` is intended as an *observable* entity that can work also as an *observer* in the conventional object oriented (oo) semantics [1].



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

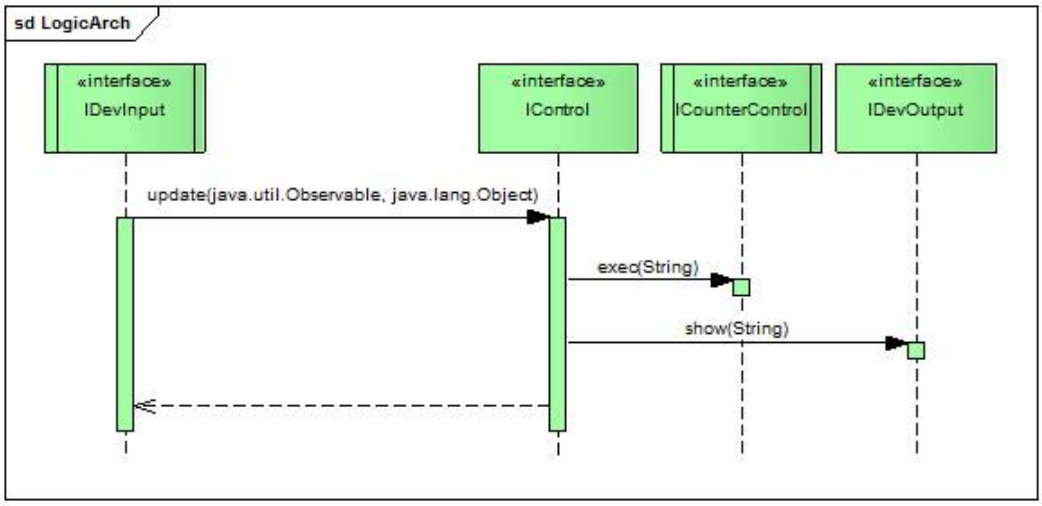
Listing 1.5. The interface `IObservable.java`

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

Listing 1.6. The interface `IObserver.java`

2.2 A first (oo) logic architecture

Starting form the model of a button as an observable input device, the logic architecture of our software system can be initially defined in the conceptual space of "classic" object oriented software development as shown in the following `UML` interaction diagram:



The input device is an (active) entity observed by the application control, that, when called by the input source, sends information on the output devices and commands to other controls (in our case the control of a counter). Since all the application logic is performed within the thread of the input device, the controller should return the control as soon as possible.

3 Concrete input devices

A concrete button can be realized in many different ways, each associated to a specific interface:

- as an element of a `GUI` (`IDevButtonGui`)
- as a device related to the standard input that could be an object based on `System.in` (`IDevButtonStdin`) or a virtual input device part of a `GUI` (`IDevButtonStdin`).
- as a physical button (`IDevButtonPi4J`)

All the concrete button types can be defined as a special kind of `IDevInputImpl` that defines the implementation of a device as an observable entity:

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

public interface IDevInputImpl extends IDevInput{
}
```

Listing 1.7. The interface `IDevInputImpl.java`

A class that can be used as the root of any concrete button implementation can be defined as follows:

```
package it.unibo.button.bridge.impl;
import java.util.Observable;
import it.unibo.button.bridge.SysKb;
import it.unibo.domain.interfaces.IDevInputImpl;
import it.unibo.is.interfaces.IActivityBase;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.system.SituatedPlainObject;

public class DevInputImpl extends SituatedPlainObject implements IDevInputImpl, IActivityBase {
protected boolean isPressed = false;
protected String name ;
    public DevInputImpl(String name, IOutputEnvView outView) {
        super(outView);
        this.name = name;
    }
    @Override
    public int getInput() throws Exception { return isPressed ? 1 : 0; }
    @Override
    public void execAction(String cmd) {
        isPressed = cmd.equals( SysKb.repHigh );
        println("DevButton isPressed=" + isPressed + " since upadatd by " + cmd);
        this.setChanged(); //!!!!
        this.notifyObservers(cmd);
    }
    public void update( boolean v){
        String cmd = v ? SysKb.repHigh : SysKb.repLow ;
        execAction( cmd );
    }
    @Override
    public void update(Observable arg0, Object arg1) {
        String vs = ""+arg1;
        isPressed = vs.equals( SysKb.repHigh );
        update( isPressed );
    }
    @Override
    public String getDefaultRep() {
        try {
            return "sensor("+this.name+","+this.getInput()+")";
        } catch (Exception e) { return "sensor("+this.name+",null)"; }
    }
    @Override
    public String getName() {
        return name;
    }
 }
```

Listing 1.8. `DevInputImpl.java`

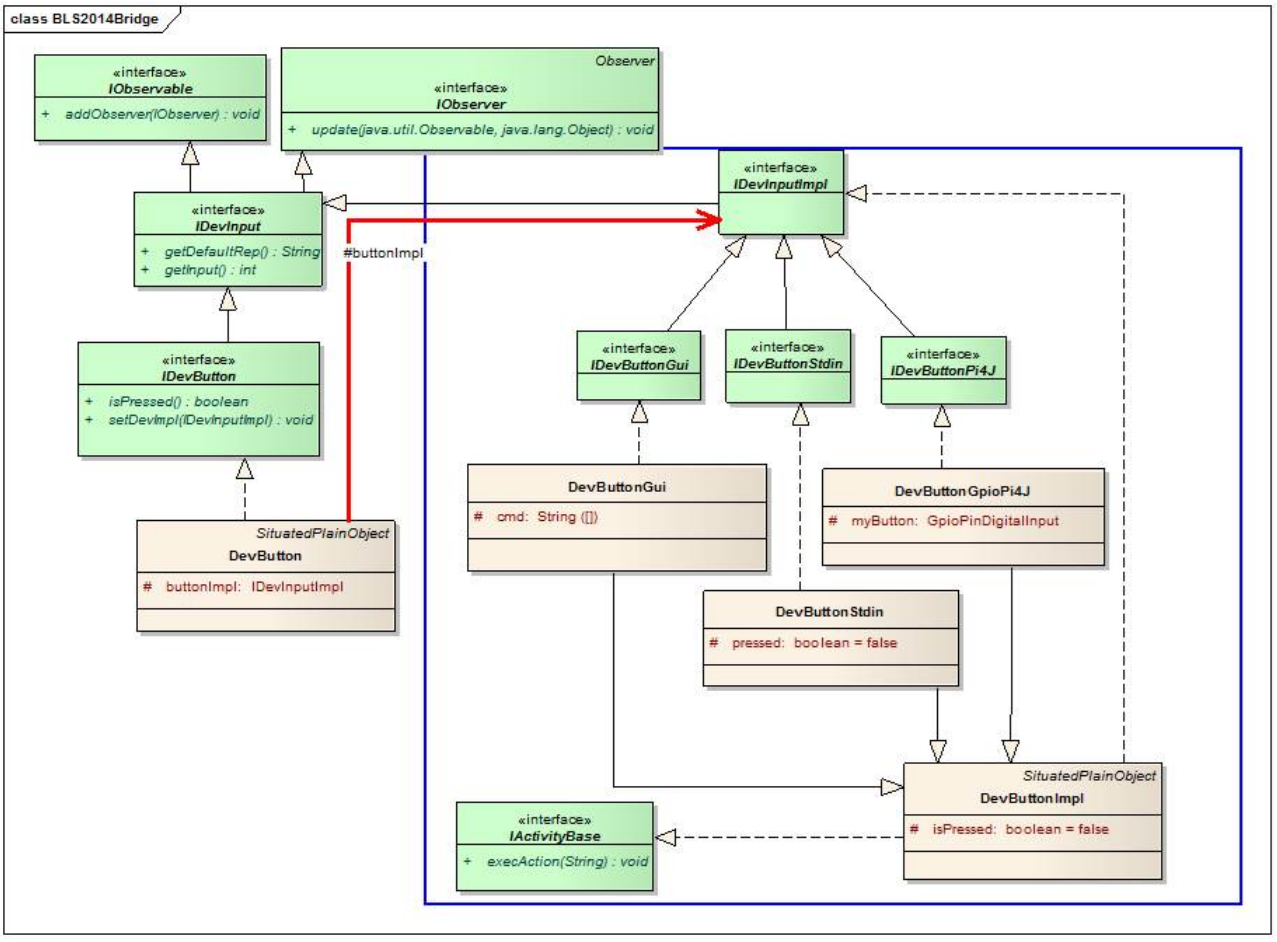
Note that the *SituatedPlainObject* has been (re)defined as an observable entity that extends `java.util.Observable` and implements the interface `it.unibo.is.interfaces.IObservable`. The class `DevImplButton` defines a passive entity that implements also the `IActivityBase` interface to provide a standard operation (`execAction`) that can be called by a run-time platform (see Subsection 3.3):

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

public interface IActivityBase {
    public void execAction( String cmd );
}
```

Listing 1.9. `IActivityBase.java`

3.1 The bridge pattern

Usually, most of the problems found in software applications have been already analysed and solved. The `GOF` catalogue [1] is a good starting point for the best-practices in solving several application problems with object-based solutions. The ***bridge pattern*** [1] can be used to decouple the `IDevButton` abstraction from its implementation so that the two can vary independently:



The class `DevButton` is a logical device that owns a reference to the implementation device to which it delegates the work to do.

3.2 An implementation of the logical device `IDevButton`

The class `DevButton` is a first oo implementation of the logical device `IDevButton` that delegates most of the work to an object of class `IDevButtonImpl`:

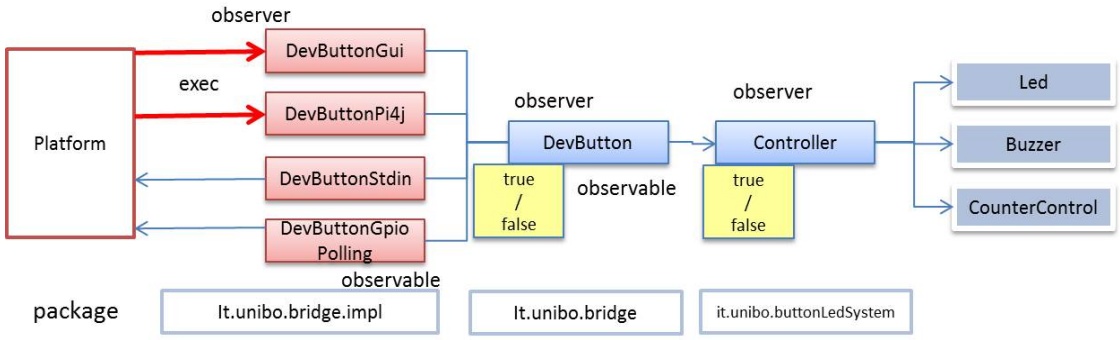
```
1 package it.unibo.button.bridge;
2
3 import java.util.Observable;
```

```
4
5  import it.unibo.domain.interfaces.IDevButton;
6  import it.unibo.domain.interfaces.IDevInput ;
7  import it.unibo.domain.interfaces.IDevInputImpl;
8  import it.unibo.is.interfaces.IOutputEnvView;
9  import it.unibo.system.SituatedPlainObject;
10
11 public class DevButton extends SituatedPlainObject implements IDevButton { //IObserver and IObservable
12 protected IDevInputImpl buttonImpl;
13 protected String name;
14 public DevButton( String name, IOutputEnvView outView ){
15     super(outView);
16     this.name = name;
17 }
18 public void setDevImpl(IDevInputImpl buttonImpl){
19     this.buttonImpl = buttonImpl;
20 }
21     @Override
22     public int getInput() throws Exception {
23         return buttonImpl.getInput();
24     }
25     @Override
26     public boolean isPressed() throws Exception {
27         return buttonImpl.getInput() == 1;
28     }
29     @Override
30     public synchronized void update(Observable arg0, Object arg1) {
31         buttonImpl.update(arg0, arg1);
32     }
33     @Override
34     public String getName() {
35         return name;
36     }
37     @Override
38     public String getDefaultRep() {
39         return buttonImpl.getDefaultRep();
40     }
41 }
```

Listing 1.10. `DevButton.java`

3.3 A project architecture

The bridge pattern can be implemented in several ways. The following picture gives an informal view (a sort of collaboration diagram) of a system in which there are different implementations for a logical device.



The *Platform* represents a basic computational support, e.g. the `JVM` or some other custom framework. In our case we suppose that some concrete button implementation can be directly called by the platform by exposing a proper interface (e.g the `IActivityBase` interface).

```
1  package it.unibo.is.interfaces;
2
```

```
3  public interface IActivityBase {
4      public void execAction( String cmd );
5  }
```

Listing 1.11. `IActivityBase.java`

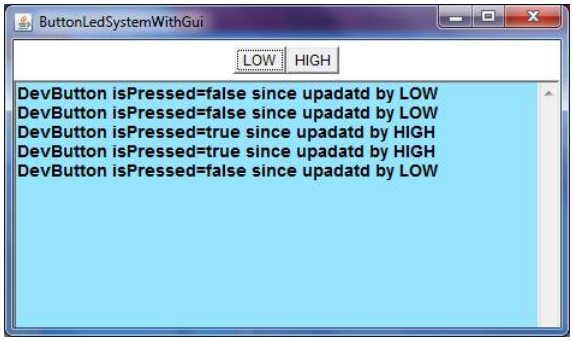
3.4 GUI-based buttons (mock objects)

Before facing the problem of managing real buttons (see Section **??**), let us introduce here a possible implementation of a (mock) button based on a `GUI`:

```
1  package it.unibo.button.bridge.impl;
2  import java.awt.Color;
3  import it.unibo.baseEnv.basicFrame.EnvFrame;
4  import it.unibo.button.bridge.SysKb;
5  import it.unibo.domain.interfaces.IDevButtonGui;
6  import it.unibo.is.interfaces.IBasicEnvAwt;
7  import it.unibo.is.interfaces.IOutputEnvView;
8
9  public class DevButtonGui extends DevInputImpl implements IDevButtonGui {
10     protected String[] cmd;
11
12     public DevButtonGui(String name, IOutputEnvView outView, String[] cmd) {
13         super(name, outView);
14         this.cmd = cmd;
15         configure();
16     }
17
18     protected void configure() {
19         //Panel cmdPanel =
20             outEnvView.getEnv().addCmdPanel("", cmd, this);
21     }
22
23     /*
24      * Main (rapid check)
25      */
26     public static void main(String args[]) throws Exception {
27         IBasicEnvAwt env = new EnvFrame("DevButtonGui", Color.lightGray, Color.BLACK);
28         env.init();
29         new DevButtonGui("b0", env.getOutputEnvView(), new String[] {
30                 SysKb.repLow, SysKb.repHigh });
31     }
32
33 }
```

Listing 1.12. `DevButtonGui.java`

A click on a button of the `GUI` calls the *update* operation that in its turn updates the registred observers (e.g. a controller). The button works as shown in the following picture (input sequence: `LOW-LOW-HIGH-HIGH-LOW`):

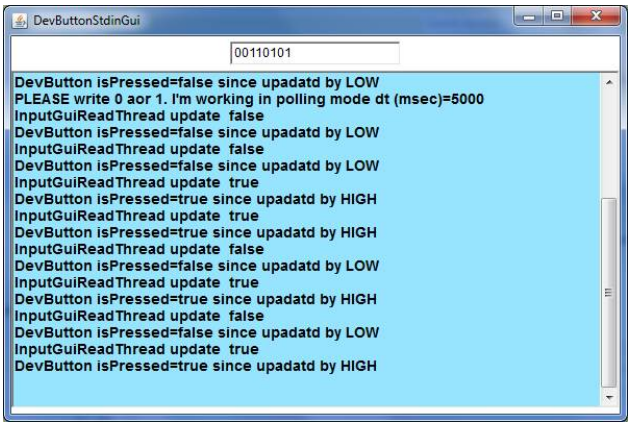


Another implementation, based on a standard input virtual device, could be:

```java
package it.unibo.button.bridge.impl;
import java.awt.Color;

import it.unibo.baseEnv.basicFrame.EnvFrame;
import it.unibo.domain.interfaces.IDevButtonStdinGui;
import it.unibo.is.interfaces.IBasicEnvAwt;
import it.unibo.is.interfaces.IOutputEnvView;

public class DevButtonStdinGui extends DevInputImpl implements IDevButtonStdinGui {
    protected boolean pressed = false;

    public DevButtonStdinGui(String name,IOutputEnvView outView) {
        super(name,outView);
        println("STARTS wirh gui ");
        new InputGuiReadThread(outView.getEnv(), this).start();
    }
    /*
     * Main (rapid check)
     */
    public static void main(String args[]) throws Exception {
        IBasicEnvAwt env = new EnvFrame("DevButtonStdinGui", Color.lightGray, Color.BLACK);
        env.init();
        env.addInputPanel();
        new DevButtonStdinGui("b0",env.getOutputEnvView());
    }
/*
* InputGuiReadThread
*/
private class InputGuiReadThread extends Thread {
protected IBasicEnvAwt env;
protected DevButtonStdinGui dev;
protected boolean running = true;
protected int dt = 5000;
protected String curS = "";

        public InputGuiReadThread(IBasicEnvAwt env, DevButtonStdinGui dev) {
            this.env = env;
            this.dev = dev;
        }
        public void run() {
            try {
                while (running) {
                    env.println("PLEASE write 0 aor 1. I'm working in polling mode dt (msec)="
                            + dt);
                    String inpS = env.readln(); // NON Blocking
                    if (inpS.length() > 0)
                        elab(inpS);
                    // POLLING
                    Thread.sleep(dt);
                }
            } catch (Exception e) {
                dev.update(false);
            }
        }// run

        protected void elab(String inpS) {
            if (curS.equals(inpS)) {
                // env.println("as before ... ");
                return;
            }
            for( int i=0; i<inpS.length(); i++ ){
                if( inpS.charAt(i) == '1' ){
                    env.println("InputGuiReadThread update true " );
                    dev.update(true);
                }else if( inpS.charAt(i) == '0' ){
                    env.println("InputGuiReadThread update false " );
                    dev.update(false);
                }
            }
            curS = inpS;
        }
    }
}
```

Listing 1.13. `DevButtonStdinGui.java`

The button works as shown in the following picture (input sequence: `00110101`):



3.5 A factory of buttons

Another recurrent strategy for designing and building quality code is to delegate the creation of object to proper resources according to on the following *creational patterns*:

- *Factory Method*: creation through inheritance.
- *Prototype*: creation through delegation.
- *Abstract Factory*: interface for creating families of related or dependent objects without specifying their concrete classes.

Factory Method Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. Factory Methods are usually called within *Template Methods* that return a type that is an interface (or an abstract class).

Prototype Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Abstract Factory Provide a level of indirection that abstracts the creation of families of related or dependent objects without directly specifying their concrete classes. The "factory" object has the responsibility for providing creation services for the entire platform family. Clients never create platform objects directly, they ask the factory to do that for them.

The `DevButtonFactory` reported hereunder provides a set of static factory methods to create concrete buttons and logical buttons:

```
1  package it.unibo.button.bridge;
2  import it.unibo.button.bridge.impl.DevButtonGpioPi4J;
3  import it.unibo.button.bridge.impl.DevButtonGpioPolling;
4  import it.unibo.button.bridge.impl.DevButtonGui;
5  import it.unibo.button.bridge.impl.DevButtonStdin;
6  import it.unibo.button.bridge.impl.DevButtonStdinGui;
7  import it.unibo.domain.interfaces.IDevButton;
8  import it.unibo.domain.interfaces.IDevInput;
9  import it.unibo.domain.interfaces.IDevInputImpl;
10 import it.unibo.is.interfaces.IOutputEnvView;
```

```
11
12 public class DevButtonFactory {
13 //IMPLEMENTATION
14     public static IDevInputImpl createButtonGui(String name,IOutputEnvView outView,String[] cmd){
15         return new DevButtonGui( name,outView, cmd);
16     }
17     public static IDevInputImpl createButtonGpioPolling( String name,IOutputEnvView outView ){
18         return new DevButtonGpioPolling( name, outView );
19     }
20     public static IDevInputImpl createButtonPi4j(String name,IOutputEnvView outView ){
21         return new DevButtonGpioPi4J( name,outView );
22     }
23     public static IDevInputImpl createButtonStdin( String name ){
24         return new DevButtonStdin( name );
25     }
26     public static IDevInputImpl createButtonStdin(String name,IOutputEnvView outView ){
27         return new DevButtonStdinGui( name,outView );
28     }
29 //HIGH LEVEL
30     /*
31      * The button is implemented by a GUI
32      */
33     public static IDevButton create(String name,IOutputEnvView outView,String[] cmd){
34         IDevInputImpl devImpl = createButtonGui(name, outView,cmd );
35         return  createLogicalButton( name, outView, devImpl);
36     }
37     /*
38      * The button is implemented by the standard input
39      */
40     public static IDevButton createStdin( String name ){
41         IDevInputImpl devImpl = createButtonStdin( name );
42         return  createLogicalButton(name, null, devImpl);
43     }
44     public static IDevButton createStdin( String name,IOutputEnvView outView ){
45         IDevInputImpl devImpl = createButtonStdin( name,outView );
46         return  createLogicalButton( name, outView, devImpl);
47     }
48     /*
49      * The button is implemented by a GPIO
50      */
51     public static IDevButton createGpioPolling(String name,IOutputEnvView outView,String[] cmd){
52         IDevInputImpl devImpl = createButtonGpioPolling( name,outView );
53         return  createLogicalButton( name, outView, devImpl);
54     }
55     public static IDevButton createGpioPi4j( String name,IOutputEnvView outView ){
56         IDevInputImpl devImpl = createButtonPi4j( name,outView );
57         return  createLogicalButton( name, outView, devImpl);
58     }
59 //-----------------------------------------
60     protected static IDevButton createLogicalButton(String name,IOutputEnvView outView, IDevInputImpl devImpl){
61         IDevButton button = new DevButton(name, outView );
62         //Injection of the implementation
63         button.setDevImpl(devImpl);
64         return button;
65     }
66  }
```

Listing 1.14. `DevButtonFactory.java`

3.6 A system prototype

The capabilities of the software so far developed can be shown by building in a short time a prototype that includes three different type of concrete buttons: a button (`DevButtonGui`) based on a `GUI` with two value commands (`LOW` and `HIGH`), a button (`DevButtonStdin`) based on information (some line ended with `CR`) inserted by the user in the standard input device (`System.in`) and a button (`DevButtonStdinGui`) based on a standard virtual input device:

```
1 package it.unibo.buttonLedSystem;
2
```

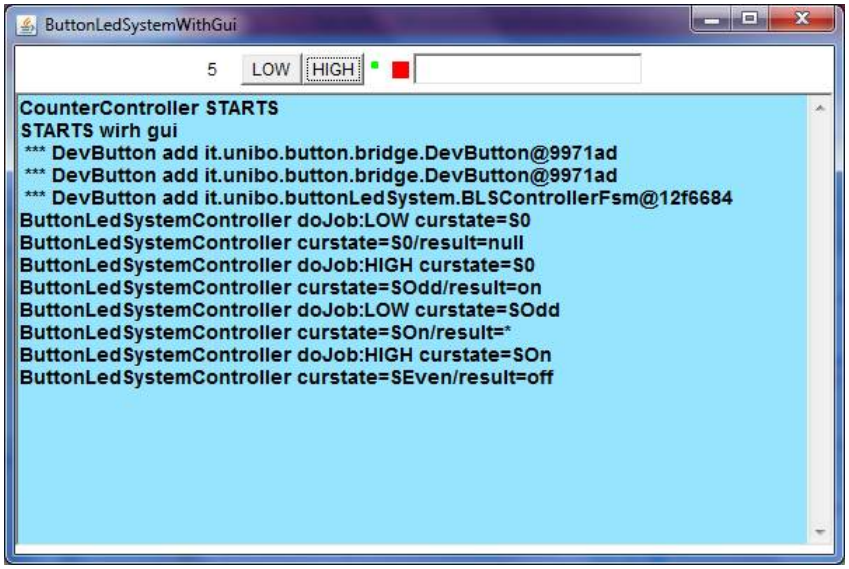
```
3  import it.unibo.baseEnv.basicFrame.EnvFrame;
4  import it.unibo.button.bridge.DevButtonFactory;
5  import it.unibo.button.bridge.SysKb;
6  import it.unibo.counter.CounterController;
7  import it.unibo.counter.CounterWithGui;
8  import it.unibo.domain.interfaces.IController;
9  import it.unibo.domain.interfaces.ICounterController;
10 import it.unibo.domain.interfaces.ICounterUp;
11 import it.unibo.domain.interfaces.ILedGpio;
12 import it.unibo.domain.interfaces.IDevInput;
13 import it.unibo.is.interfaces.IBasicEnvAwt;
14 import it.unibo.led.LedWithGui;
15 import java.awt.Color;
16
17 public class BLSWithGuiAndFactoryMain {
18     protected ILedGpio ledGreen;
19     protected ILedGpio ledRed;
20     protected IController controller;
21     protected IDevInput[] devs;
22     protected ICounterUp counter;
23     protected ICounterController counterCtrl;
24     protected IBasicEnvAwt env;
25
26     public void doJob() throws Exception {
27         System.out.println("ButtonLedSystemWithGui STARTS");
28         init();
29         configure();
30         start();
31     }
32
33     protected void init() {
34         env = new EnvFrame("ButtonLedSystemWithGui");
35         env.init();
36         counter = new CounterWithGui(env);
37         counterCtrl = new CounterController(env, counter);
38         devs = new IDevInput[3];
39         // Buttons
40         devs[0] = DevButtonFactory.createButtonGui("bGui",
41                 env.getOutputEnvView(), new String[] { SysKb.repLow,
42                         SysKb.repHigh });
43         devs[1] = DevButtonFactory.createButtonStdin("bStdIn");
44         devs[2] = DevButtonFactory.createButtonStdin("bStdIn",
45                 env.getOutputEnvView());
46         // Led
47         ledGreen = new LedWithGui(env, Color.green);
48         ledRed = new LedWithGui(env, Color.red);
49         // Controller
50         controller = new BLSControllerFsm(env);
51         for (int i = 0; i < devs.length; i++)
52             devs[i].addObserver(controller);
53     }
54
55     protected void configure() {
56         env.addInputPanel();
57         controller.setLedGreen(ledGreen);
58         controller.setLedRed(ledRed);
59         controller.setCounter(counterCtrl);
60     }
61
62     protected void start() {
63     }
64
65     public static void main(String args[]) throws Exception {
66         BLSWithGuiAndFactoryMain system = new BLSWithGuiAndFactoryMain();
67         system.doJob();
68     }
69 }
```

Listing 1.15. `BLSWithGuiAndFactoryMain.java`

All the buttons arre connected to the same controller, that provides to turn on/off the leds and to start/stop the increment of a counter:

```java
package it.unibo.buttonLedSystem;

import it.unibo.button.bridge.SysKb;
import it.unibo.is.interfaces.IBasicEnvAwt;

public class BLSControllerSimple extends
       BLSControllerBase {
protected int nSwitch = 0;
protected boolean isHigh = false;
    public BLSControllerSimple(IBasicEnvAwt env) {
        super(env);
        init();
    }

    @Override
    public void doJob(String cmd) {
        println("BLSControllerSimple " + cmd);
        if (cmd.equals(SysKb.repHigh) ){
            if( ! isHigh ){
                isHigh = true;
                ledSwitch();
            }
        }
        else isHigh = false;
    }

    protected void init(){
        if (ledGreen != null )
            ledGreen.turnOff();
        if (ledRed != null)
            ledRed.turnOn();
    }
    protected void ledSwitch() {
        if (ledGreen == null ) return;
        nSwitch++;
        if ( ledGreen.isOn() ) {
            ledGreen.turnOff();
            if (ledRed != null)
                ledRed.turnOn();
            if (counterCtrl != null)
                counterCtrl.stop();
        } else {
            ledGreen.turnOn();
            if (ledRed != null)
                ledRed.turnOff();
            if (counterCtrl != null)
                counterCtrl.start();
        }
    }

    @Override
    public String getInfo() {
        return "" + nSwitch;
    }
}
```

Listing 1.16. `BLSControllerSimple.java`



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.