The ButtonLed system

Antonio Natali

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

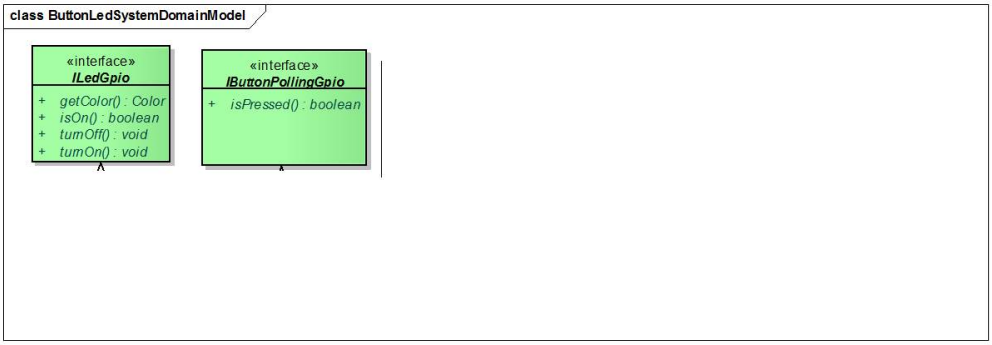
Abstract

1 Introduction

2 Requirements

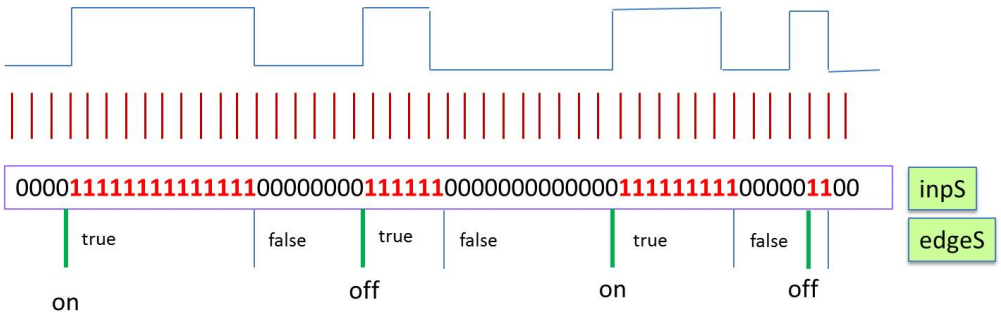
Design and build a *ButtonLed* software system in which a `Led` is turned on and off each time a `Button` is pressed (by an human user). The software system must be deployed as an executable `jar` file.

2.1 Domain model



3 Problem analysis

The button is a source that emits a wave that is sampled by an entity that implements the interface `IButtonPollingGpio`.



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 must 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 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 ...
```

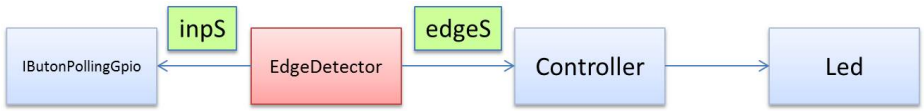
We can say that the `Led` is turned on `N` times, where `N` is the number of `true` in odd position in the `edgeS` sequence.

3.1 Elaboration components

The problem requires that the following elaborations on the basic input

- the detection of the edges in the input sequence
- the detection of edges of type "low to high" in order to switch the led

The responsibility of these functions can be given to two new different entities: an entity *EdgeDetector* and an entity *Controller* that realizes "business logic" of the system. In particular, the *Controller*) receives in input the sequence `edgeS` and performs a *switch* of the led for each `true` value found in the input sequence.



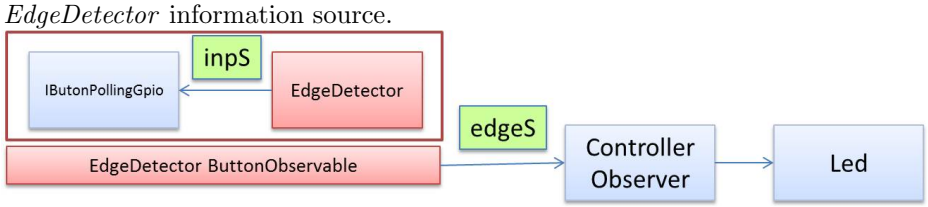
IController
```
public interface IController extends Observer{
        public void doJob(String cmd);
        public String getInfo();
```

The system should be *event-driven*, i.e. a computation should take place each time a new input becomes available.

Both the *EdgeDetector* and the *Controller* can be modelled as finite state machines (`FSM`) working as *transducers*. They can be viewed either as *objects* interacting via procedure-calls or *active entities* (e.g. processes, actors, agents, etc) interacting via message-passing.

3.2 Object-oriented architectures

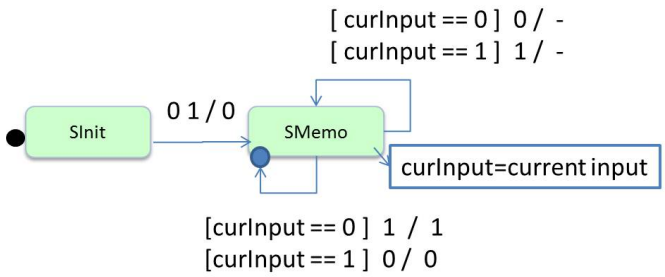
In any object-oriented model, all the computation usually takes place within a single thread. In our case the main thread could be the thread related to the component that performs the polling of the wave, i.e. the `EdgeDetector`. In this case, the *Controller* is called by the *EdgeDetector* that, must explicitly know the *Controller* in order to call it. However, a more flexible architecture an be obtained (without changing the run-time interaction pattern) by conceiving the *Controller* as an *observer* that can be registered to the *EdgeDetector* information source.



3.3 Message-passing architectures

To do

3.4 Edge detector behavior model



Edge detector FSM

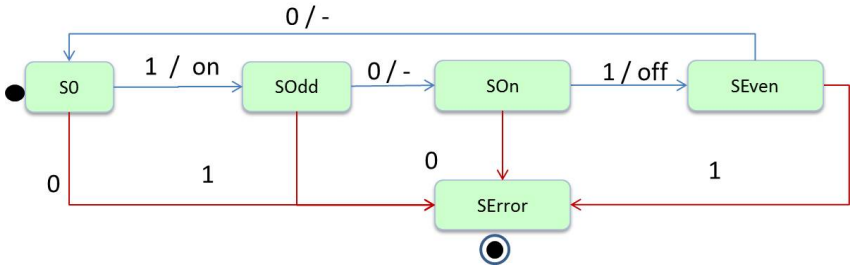
```
public class EdgeDetector implements IEdgeDetector{
        protected boolean curInput = false ;
        protected String curState="SInit";
        protected boolean output = false;
        protected IDevInputBoolean inputDev ;
        protected boolean pressed ;

        public void setInputDevice(IDevInputBoolean inputDev){
                this.inputDev = inputDev;
        }
        public boolean detectEdge() throws Exception{
                return edgeDetectorFsm();
                //return edgeDetectorSmartFsm();
        }
        protected boolean edgeDetectorFsm() throws Exception{
                output = false;
                while( ! output  ){
                        if( curState.equals("SInit") ){
                                SInit();
                        }
                        if( curState.equals("SMemo") ){
                                pressed = inputDev.getInput() ;
                                SMemo(pressed);
                        }
                }//while
                return curInput; //true means lowToHigh
         }
        /*
         * Initially the button is low-level
         */
        protected void SInit(){
                output = false;
                curInput = false;
                curState="SMemo";
        }
        protected void SMemo(boolean input){
                if( curInput != input  ){
                        output = true;
                        curInput = input;
                        curState="SMemo";
                }else output = false; //same state
        }
 }
```

Since the `SInit` state is just an initialization, thee state machine (i.e. the method *edgeDetectorFsm*) can be optimised as follows:

Edge detector optimized

```
protected boolean edgeDetectorSmartFsm() throws Exception{
        boolean output = false;
        while( ! output ){
                boolean input = inputDev.getInput() ;
                output = curInput != input;
                curInput = input;
         }
         return curInput;
}
```

3.5 Controller behavior model



Controller FSM

```
protected int nOfLowHighEdges = 0;
protected String curstate = "S0";
protected boolean on = false;
       protected String controllerFsm(String inp){
               if( curstate.equals("S0")){
                       if( inp.equals("true")){
                               nOfLowHighEdges++;
                               curstate = "SOdd";
                               return "on";
                       }else return null; //Exception
               }
               if( curstate.equals("SOdd")){
                       if( inp.equals("false")){
                               curstate = "SOn";
                               return "";
                       }else return null; //Exception
               }
               if( curstate.equals("SOn")){
                       if( inp.equals("true")){
                               nOfLowHighEdges++;
                               curstate = "SEven";
                               return "off";
                       }else return null; //Exception
               }
               if( curstate.equals("SEven")){
                       if( inp.equals("false")){
                               curstate = "S0";
                               return "";
                       }else return null; //Exception
               }
               return null;
        }
```

The *controllerFsm* can be called by any entity that receives in input a `edgeS` sequence, for example by an observer that propagates the edges:

Controller as an observer

```
       public void update(Observable arg0, Object isOn) {
                doJob(""+isOn); //TODO: define better the data model
       }
       public void doJob(String cmd) {
               String result = controllerFsm(cmd);        //controllerSmartFsm(cmd);
               if( result != null ){
```

```
                        if( result.equals("on")){
                                if(led !=null) led.turnOn();
                                if(buzzer!=null) buzzer.turnOn();
                        }
                        if( result.equals("off")){
                                if(led !=null) led.turnOff();
                                if(buzzer!=null) buzzer.turnOff();
                        }
                }
        }
```

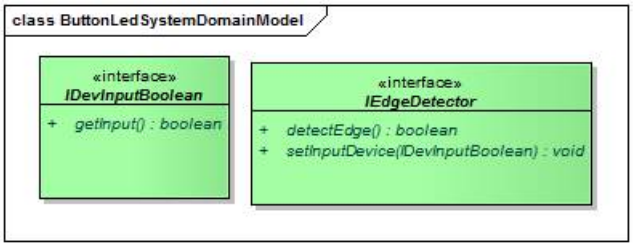
An optimized (but less model-based) version:

Controller as an observer

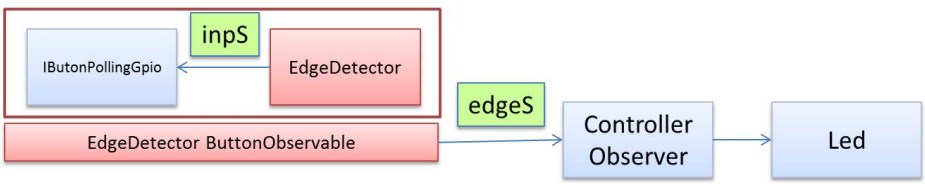
```
protected String controllerSmartFsm(String inp){
        if( inp.equals("true")){
                nOfLowHighEdges++;
                if( nOfLowHighEdges % 2 != 0 ) return "on";
                else return "off";
        }else return "";
}
```

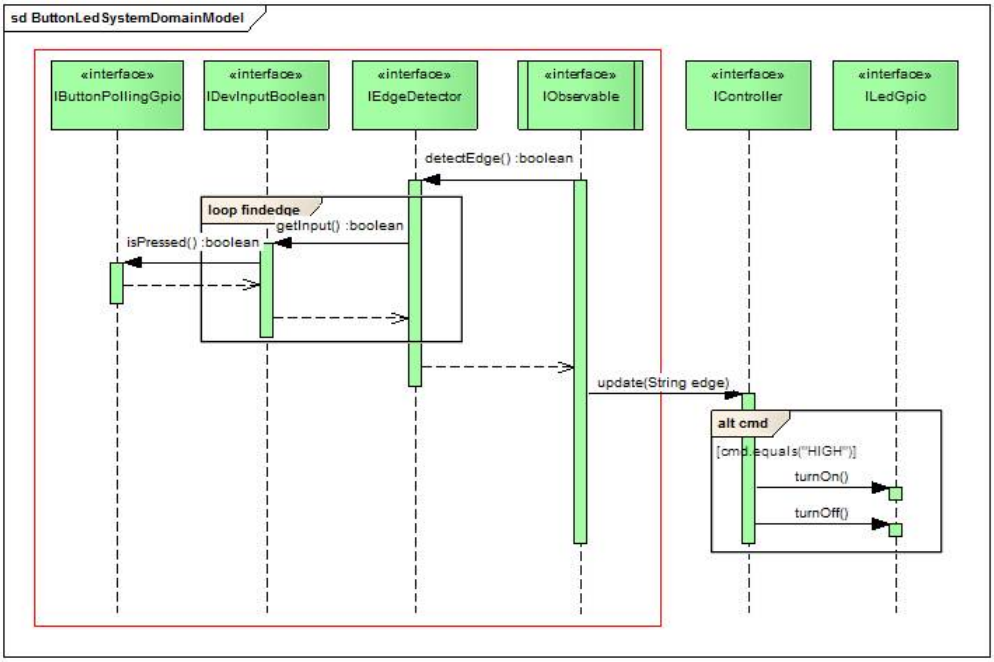
4 A first prototype

Let us introduce general device models:



4.1 Interaction model





4.2 Test plans

Test plans

```
inpS                num of edges
000000                0
011111                1
011110                1
010101                3
011010                2
```

4.3 System-integration test model

The ButtonLedSystem

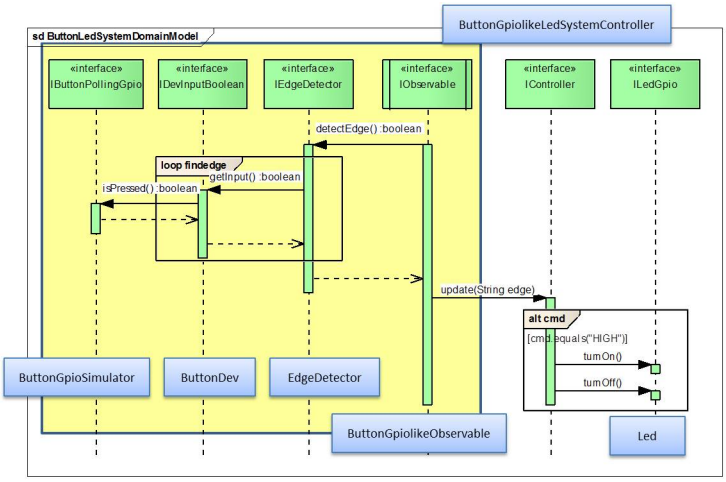
```
public class ButtonGpioLikeObervableLedSystem {
protected IObservable button ;
protected ILedGpio ledGreen ;
protected ButtonGpiolikeLedSystemController controller;
protected IButtonPollingGpio basicButton ;

        public void doJob( String inpS ) throws Exception{
                System.out.println("ButtonGpioLikeObervableLedSystem STARTS");
                init();
                configure(inpS);
                start();
        }
        protected void init(){
                 basicButton = new ButtonGpioSimulator();
                controller = new ButtonGpiolikeLedSystemController();
                button = new ButtonGpiolikeObservable(basicButton);
                ledGreen = new Led(Color.green);
        }
        protected void configure(String inputS){
                 controller.setButton(button);
                 if( inputS != null ) ((ButtonGpioSimulator)basicButton).configure(inputS);
                 controller.setLed(ledGreen);
                button.register(controller);
        }
        protected void start(){
/*
 * The ButtonGpiolikeObservable includes a simulator
```

```
 */
        }
}
```

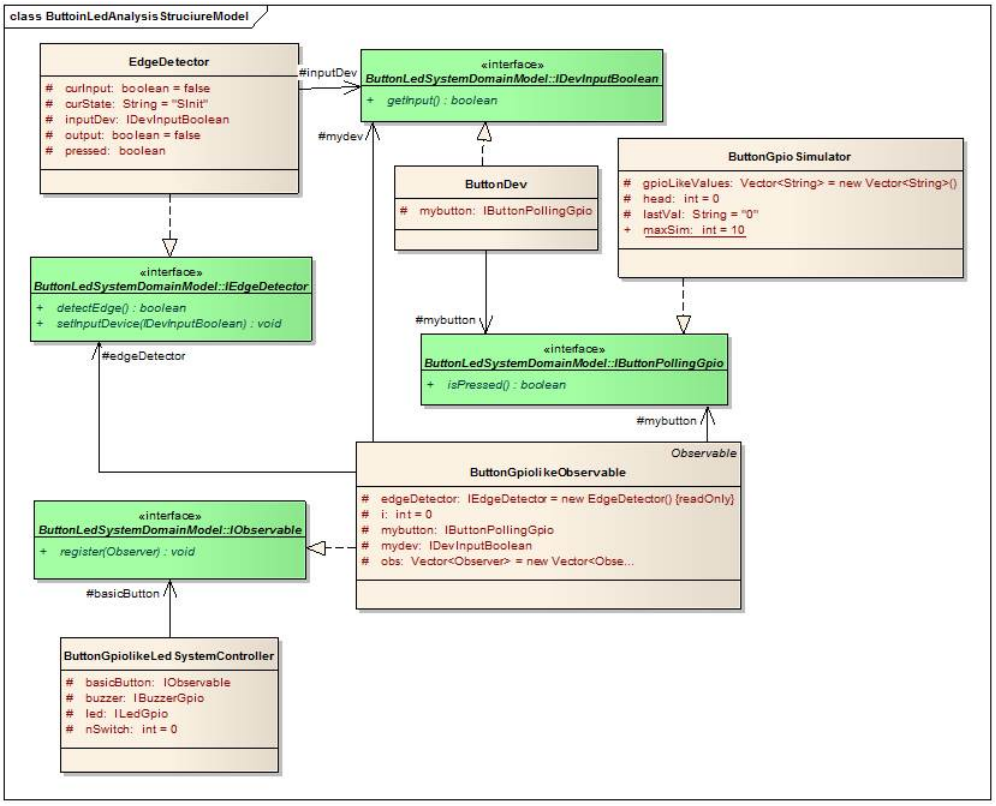
The main program

```
public class ButtonLedSystemMain {
public static void main(String args[]) throws Exception {
 ButtonGpioLikeObervableLedSystem system = new ButtonGpioLikeObervableLedSystem( );
 system.doJob("01010101010");
 }
}
```

4.4 Interaction model (project)



4.5 Implementation classes



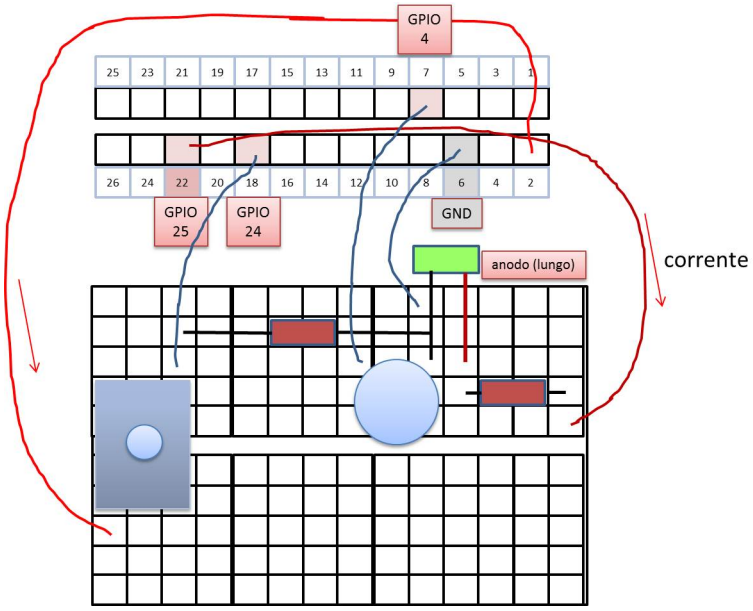
4.6 An output of the prototype

An output

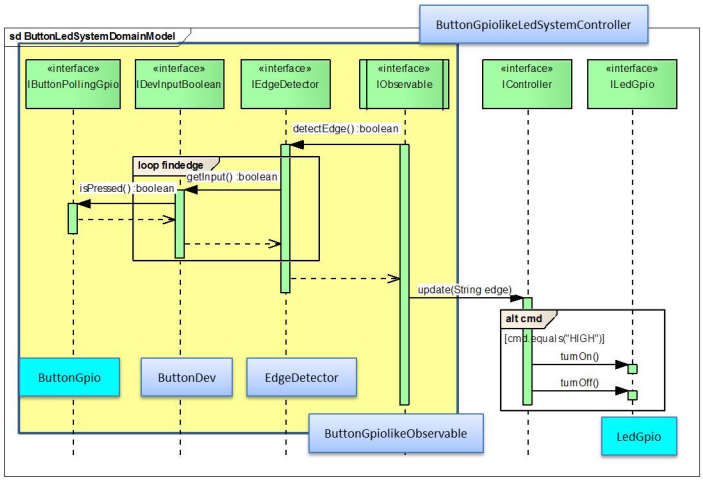
```
ButtonGpioLikeObervableLedSystem STARTS
ButtonPollingActiveSimulator CREATED
        *** ButtonGpio configure done 0101010101
 ---------- ButtonGpiolikeLedSystemController update ----------- true 0
%%% Led ON green %%%
buttonObservableLoop:0
 ---------- ButtonGpiolikeLedSystemController update ----------- false 1
buttonObservableLoop:1
 ---------- ButtonGpiolikeLedSystemController update ----------- true 1
%%% Led OFF green %%%
buttonObservableLoop:2
 ---------- ButtonGpiolikeLedSystemController update ----------- false 2
buttonObservableLoop:3
 ---------- ButtonGpiolikeLedSystemController update ----------- true 2
%%% Led ON green %%%
buttonObservableLoop:4
 ---------- ButtonGpiolikeLedSystemController update ----------- false 3
buttonObservableLoop:5
 ---------- ButtonGpiolikeLedSystemController update ----------- true 3
%%% Led OFF green %%%
buttonObservableLoop:6
 ---------- ButtonGpiolikeLedSystemController update ----------- false 4
buttonObservableLoop:7
 ---------- ButtonGpiolikeLedSystemController update ----------- true 4
%%% Led ON green %%%
buttonObservableLoop:8
 *** input terminated
 ---------- ButtonGpiolikeLedSystemController update ----------- false 5
}
```

5 The prototype on the Raspberry Pi

5.1 The circuit



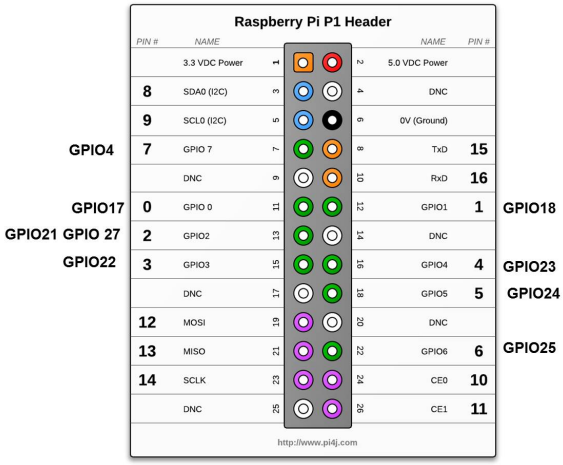
5.2 Interaction model (GPIO basic project)



5.3 The library Pi4J

The library `Pi4J` provides a bridge between the native libraries and `Java` for full access to the Raspberry Pi. The functionality is reasonable stable but some documentation and development tasks have not yet been completed.

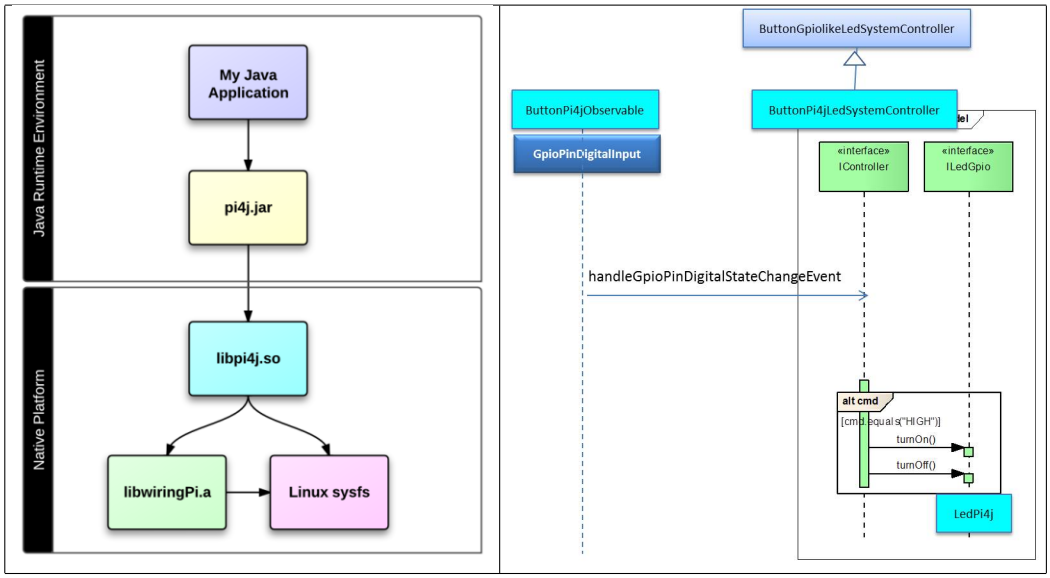
`Pi4J` uses an abstract pin numbering scheme to help insulate software from hardware changes. `Pi4J` implements the same pin number scheme as the *Wiring Pi* project as shown in the following picture, in which we report also the "basic" `GPIO` names.



To access a `GPIO` pin with `Pi4J`, we must first *provision* the pin. Provisioning configures the pin based on how we intend to use it. Provisioning can automatically export the pin, set its direction, and setup any edge detection for interrupt based events.

Thus, by using the `Pi4J` library, the *EdgeDetector* component is already provided by the library according to the pattern observer [1] model.

5.4 Interaction model (GPIO Pi4J project)



ButtonPi4jObservable

```
public class ButtonPi4jObservable extends Observable implements IObservable {
protected GpioPinDigitalInput myButton;

        public ButtonPi4jObservable( int pinNum ) {
                myButton = GpioOnPi4j.controller.provisionDigitalInputPin(
                        RaspiPin.GPIO_05, PinPullResistance.PULL_DOWN);
        }
        @Override
        public void register(Observer arg0) {
                myButton.addListener((GpioPinListenerDigital)arg0);
        }
```

```
}
```

ButtonPi4jLedSystemController (adapter)

```
public class ButtonPi4jLedSystemController extends ButtonGpiolikeLedSystemController implements GpioPinListenerDigital{
         @Override
        public void handleGpioPinDigitalStateChangeEvent(
                        GpioPinDigitalStateChangeEvent event) {
                boolean on = event.getState().isHigh();
                System.out.println(" --> ButtonLedSystemController GPIO PIN STATE CHANGE: "
                                + event.getPin() + " = " + on);
                update(null, on );
        }
}
```

6 Code (on Raspberry Pi)

A simple *buttonLed* system that looks at a button (on `GPIO24`) and turns on (off) a led (on `GPIO25`) if the button is pressed (unpressed) can be defined in shell Linux as follows (the reader could modify the code to switch the led each time the button is pressed):

6.1 Bash code

Bash code

```
#!/bin/bash
led=25
but=24
if [ -d /sys/class/gpio/gpio25 ]
then
        echo "led gpio${led} exist"
        echo out > /sys/class/gpio/gpio${led}/direction
else
        echo "creating led gpio${led}"
        echo ${led} > /sys/class/gpio/export
        echo out > /sys/class/gpio/gpio${led}/direction

fi
if [ -d /sys/class/gpio/gpio24 ]
then
        echo "button gpio${but} exist"
        echo in > /sys/class/gpio/gpio${but}/direction
else
        echo "creating button gpio${but}"
        echo ${but} > /sys/class/gpio/export
        echo in > /sys/class/gpio/gpio${but}/direction
fi

while true
do
        b=‘cat /sys/class/gpio/gpio${but}/value‘
        echo $b > /sys/class/gpio/gpio25/value
        sleep 0.1
done
```

The important point is that the programmer can manage a device connected on a `GPIO` pin by reading/writing some (virtual) file associated with that pin.

6.2 Java: GPIO basic

Here is a version of the *buttonLed* system written in `Java` that exploits the same files used in the bash code. The class GpioSys is an utility class introduced to help application designers in using

GPIO basic Java code

```
package it.unibo.gpio.basic.test;
import it.unibo.gpio.base.GpioOnSys;
import it.unibo.gpio.base.IGpio;
import it.unibo.gpio.base.IGpioConfig;
import java.io.FileWriter;
import java.io.IOException;

public class TestButtonBuzzerLed {
protected GpioOnSys gpio = new GpioOnSys();
protected FileWriter fwrLed;
protected FileWriter fwrBuzzer;

        public void doJob() throws Exception{
                prepareGpioButton();
                prepareGpioLed();
                prepareGpioBuzzer();
                doCmdBlink();
                System.out.println("END, bye bye");
        }
        protected void prepareGpioBuzzer() throws IOException{
                gpio.prepareGpio(IGpioConfig.gpioOutBuzzer);
                fwrBuzzer = gpio.openOutputDirection(IGpioConfig.gpioOutBuzzer);
        }
        protected void prepareGpioButton() throws IOException{
                gpio.prepareGpio(IGpioConfig.gpioInButton);
                gpio.openInputDirection(IGpioConfig.gpioInButton);
        }
        protected void prepareGpioLed() throws IOException{
                gpio.prepareGpio(IGpioConfig.gpioOutLed);
                fwrLed = gpio.openOutputDirection(IGpioConfig.gpioOutLed);
        }
        protected void doCmdBlink( ) throws Exception{
                String inps = "";
                 for( int i = 1; i<=15; i++) {
                        System.out.println("TestLedBlink reading ...");
                        inps = gpio.readGPio(IGpioConfig.gpioInButton );
                        System.out.println("TestLedBlink -> " + inps);
                        if( inps.equals("0") ){
                                gpio.writeGPio( fwrLed, IGpio.GPIO_OFF);
                                gpio.writeGPio( fwrBuzzer, IGpio.GPIO_OFF);
                        }
                        if( inps.equals("1") ){
                                gpio.writeGPio( fwrLed, IGpio.GPIO_ON);
                                gpio.writeGPio( fwrBuzzer, IGpio.GPIO_ON);
                        }
                        // Wait for a while
                        java.lang.Thread.sleep(400);
                 }
        }
        public static void main(String[] args) {
                try {
                        TestButtonBuzzerLed sys = new TestButtonBuzzerLed();
                        sys.doJob();
                 } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

The interface `IGpio` is defined as follows:

The IGpio interface

```
public interface IGpio {
        public  final String GPIO_OUT = "out";
        public  final String GPIO_IN = "in";
        public  final String GPIO_ON = "1";
        public  final String GPIO_OFF = "0";

        public void prepareGpio(String gpioChannel) throws  Exception;
        public FileWriter openOutputDirection( String gpioChannel ) throws Exception;
        public void openInputDirection( String gpioChannel ) throws Exception;
        public void writeGPio(FileWriter commandFile, String value) throws Exception;
```

```
        public void writeGPio(String gpioChannel, String value) throws Exception;
        public String readGPio( String gpioChannel ) throws Exception;

        public GpioController getGpioPi4j();
}
```

6.3 Java: Pi4J

The following `Java` version of the *buttonLed* system exploits the `Pi4J` library and the class ButtonPi4jObservable introduced in the Subsection 5.4.

GPIO Pi4J Java code

```
package it.unibo.buttonLedMD.gpio.p4j.components;
import it.unibo.gpio.base.GpioOnSys;
import it.unibo.gpio.base.IGpio;
import it.unibo.gpio.base.IGpioConfig;

import java.awt.Color;
import java.io.FileWriter;
import java.io.IOException;

public class ExamplePi4j0ButtonBuzzerLed {
protected FileWriter fwrBuzzer;
protected GpioOnSys gpio = new GpioOnSys();

protected  ButtonPi4jObservable button;
protected  ButtonPi4jObserverNaive buttonObserver;
protected  LedPi4j led;
        public void doJob() throws Exception{
                preparePi4jButton();
                prepareGpioLed();
                prepareGpioBuzzer();
                doCmdBlink();
                System.out.println("END, bye bye");
        }
        protected void prepareGpioBuzzer() throws IOException{
                gpio.prepareGpio(IGpioConfig.gpioOutBuzzer);
                fwrBuzzer = gpio.openOutputDirection(IGpioConfig.gpioOutBuzzer);
        }
        protected void preparePi4jButton() throws IOException{
                button = new ButtonPi4jObservable(IGpioConfig.pinInButton);
                buttonObserver = new ButtonPi4jObserverNaive();
                button.register(buttonObserver);
        }
        protected void prepareGpioLed() throws IOException{
                led = new LedPi4j( Color.green, IGpioConfig.pinOutLed );
        }
        protected void doCmdBlink( ) throws Exception{
                 for( int i = 1; i<=5; i++) {
                                  led.turnOn();;
                                 System.out.println("LED on " + led.isOn() );
                                 gpio.writeGPio( IGpioConfig.gpioOutBuzzer, IGpio.GPIO_ON);
                                // Wait for a while
                                java.lang.Thread.sleep(1000);
                                led.turnOff();
                                System.out.println("LED off " + led.isOn() );
                                gpio.writeGPio( IGpioConfig.gpioOutBuzzer, IGpio.GPIO_OFF);
                                java.lang.Thread.sleep(1000);                                                    }
        }
        public static void main(String[] args) {
                try {
                        ExamplePi4j0ButtonBuzzerLed sys = new ExamplePi4i0ButtonBuzzerLed();
                        sys.doJob();
                 } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

6.4 Java: model based

The *buttonLed* system that results from our model-based software development approach is configured and started as follows:

Model-based code (using Pi4J)

```
public class ButtonPi4jLedSystemMD {
protected IObservable button ;
protected ILedGpio ledGreen ;
protected IBuzzerGpio buzzer ;
protected ButtonLedSystemController controller;
protected IButtonObserver buttonObserver;
protected IButtonPollingGpio basicButton ;
        public void doJob( ) throws Exception{
                System.out.println("ButtonGpioLikeObervableLedSystem STARTS");
                init();
                configure();
                start();
        }
        protected void init(){
                 buzzer = new BuzzerPi4j(IGpioConfig.pinOutBuzzer);
                ledGreen = new LedPi4j( Color.green, IGpioConfig.pinOutLed );
                button = new ButtonPi4jObservable(IGpioConfig.pinInButton);
                buttonObserver = new ButtonPi4jObserver();
                controller = new ButtonLedSystemController();
        }
        protected void configure(){
                buttonObserver.setControl(controller);
                controller.setButton(button);
                 controller.setLed(ledGreen);
                 controller.setBuzzer(buzzer);
                button.register(buttonObserver);
                System.out.println("ButtonGpioLikeObervableLedSystem observer registered");
         }
        protected void start(){
                try {
                        System.out.println("Please press the button ...");
                        Thread.sleep(5000);
                } catch (InterruptedException e) {
                         e.printStackTrace();
                }
          }
}
```

6.5 Java: meta-model based

Here we report the specification of the *buttonLed* system written in a custom language that allows us to express *event-driven/event-based* software systems. From the technical point of view, the following code is a *model*, instance of a *metamodel* (named `ECSL`, brother of `UML` and instance of of `MOF`) defined by our research group by using the `Xtext` [2] framework:

buttonLed in ECSL (*Event Contact Specification Language*)

```
EventSystem buttonLed
        /* --- Declaration of the events --- */
Event click ;
Event cmdLed ;
        /* --- Declaration of the components --- */
Task led ;
Task controller ;
        /* --- Declaration of the external components --- */
External buttonObserver raising click ;
        /* --- Declaration of the behavior of the tasks --- */
BehaviorOf controller{
var String msg = ""
var it.unibo.contact.buttonLed.ButtonObserver bobs = null
        state controllerInit initial
                showMsg("STARTS")
                activateExternal buttonObserver("") withname bobs
                onEvent click goToState controllerWork
        endstate
```

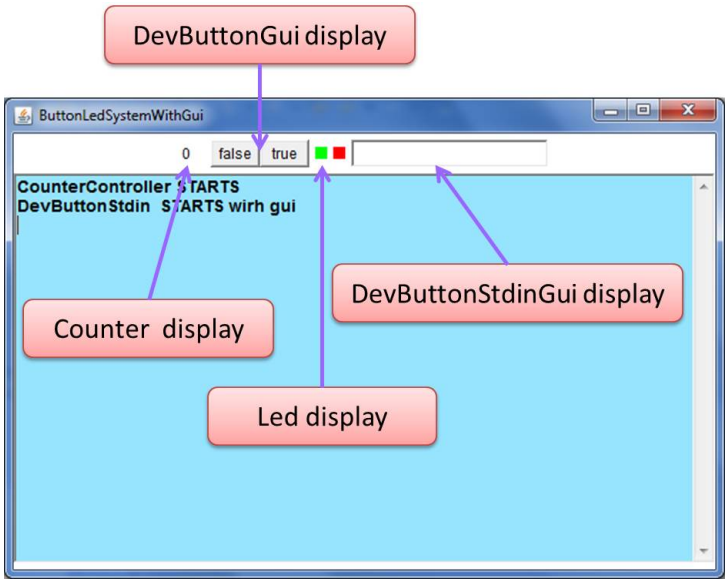
```
         state controllerWork
                 set msg = call curEventItem.getMsg()
                 showMsg("controllerWork " + %msg)
                 raiseEvent cmdLed(%msg)
                onEvent click goToState controllerWork
        endstate
}
BehaviorOf led{
var String msg = ""
        action void turn( String cmd )
        state ledInit initial
                showMsg("STARTS")
                onEvent cmdLed goToState ledWork
          endstate
         state ledWork
                 set msg = call curEventItem.getMsg()
                 showMsg("ledWork " + %msg)
                 exec turn(%msg)
                 onEvent cmdLed goToState ledWork
        endstate
}
```

Most of the application code (in particular the event-based interaction support) is automatically generated by the custom `Event-ide` associated to the `ECBSL` language and developed within the Eclipse ecosystem.

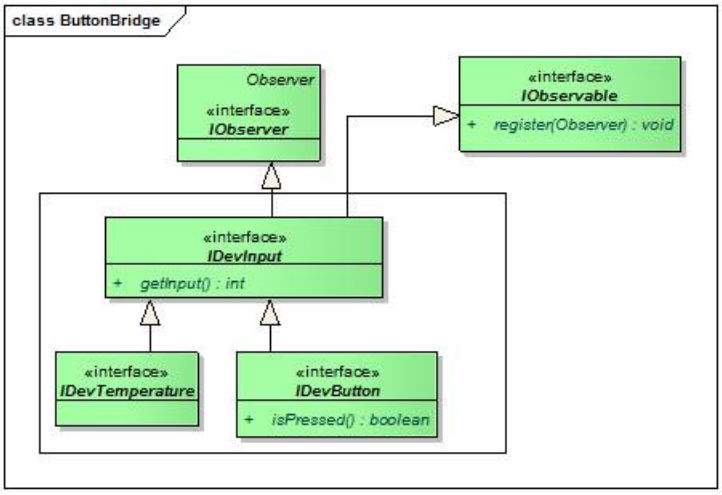
7 Overview next

Reusability is not only related to code, but it is also (mainly) related to concepts and logic design. Most of the application problems have been already analysed and solved. The `GOF` [1] catalogue is a good starting point for the best-practices in solving several application problems with object-based solutions.

Our next goal is to extend the *ButtonLedCounter* system by introducing several kinds of buttons. Besides physical devices, we aim at introducing a virtual button via some `GUI` component and a command button related to standard input.



1. A generic button (`IDevButton`) is introduced as a special `IDevInput` intended as an observable entity that works also as an observer:



IDevInput

```
public interface IDevInput extends Observer, IObservable{
        public int getInput() throws Exception;
}
```

The *getInput* operation is assumed to be available for all the input devices.

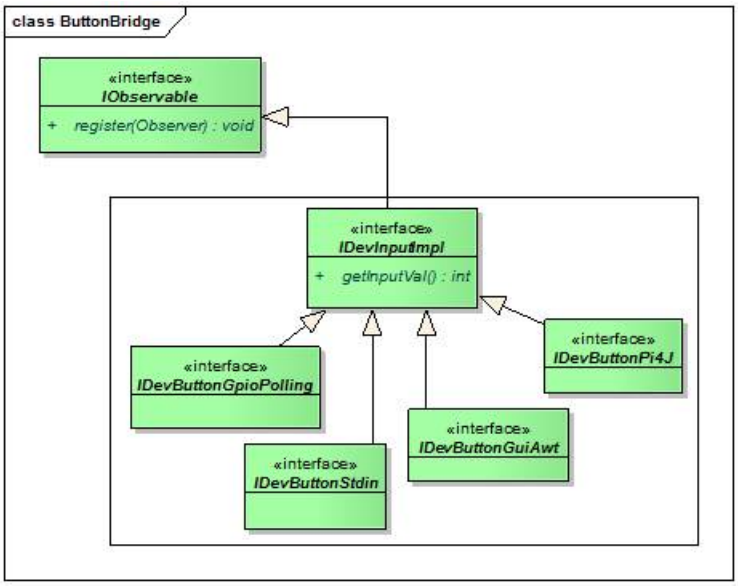
IDevButton

```
public interface IDevButton extends IDevInput{
        public boolean isPressed() throws Exception;
}
```

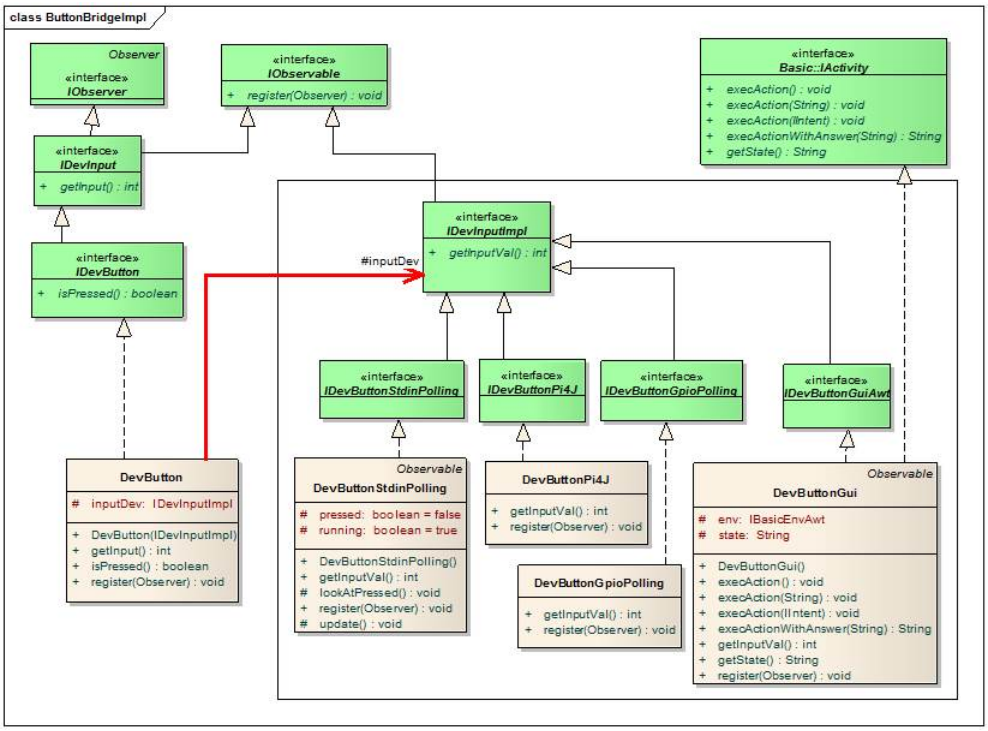
The *isPressed* operation is introduced fro *legacy* reasons.

2. A concrete button can be realized in many different ways, each associated to some specific interface:
   - as a physical button (`IDevButtonGpioPolling`, `IDevButtonPi4J`)
   - as an element of a `GUI` (`IDevButtonGuiAwt`)
   - as a device related to the standard input (`IDevButtonStdin`)

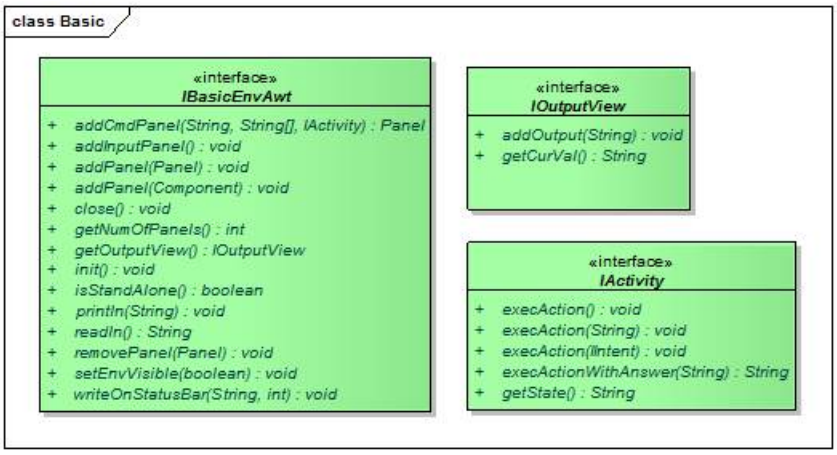
   All the concrete button types can be defined as a special kind of `IDevInputImpl` that defines the implementation of a device as an observable entity:



3. The ***bridge pattern*** [1] can be used to decouple the `IDevButton` abstraction from its implementation so that the two can vary independently:



4. The project *it.unibo.is.envBaseAwt* defines a custom framework for the rapid development of `GUI`-based applications prototypes. The framework is based on the class `it.unibo.baseEnv.basicFrame.EnvFrame` that implements the `IBasicEnvAwt` interface:



A test unit can better show the usage of some basic operation:

test

```
public class EnvBaseAwtTest {
protected IBasicEnvAwt env ;
@Before
        public void setUp() throws Exception{
                System.out.println(" *** setUp "  );
                env = new EnvFrame();
         }
@After
        public void tearDown() throws Exception{
```

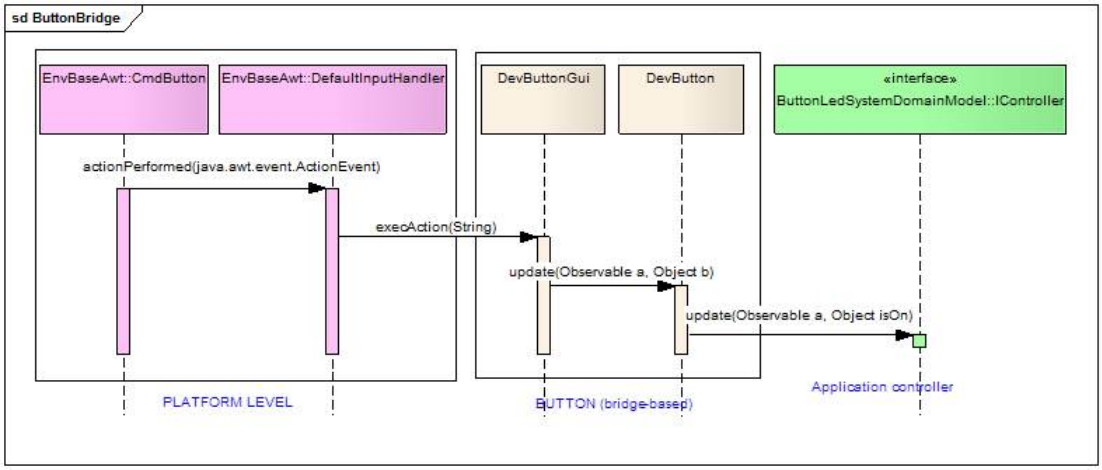
```
                System.out.println(" *** tearDown "  );
        }
@Test
        public  void testCreation(){
                System.out.println("        testCreation ... " );
                try {
                        env.init();
                        Thread.sleep(1000);
                         assertTrue("testCreation", env.isStandAlone() && env.getNumOfPanels() == 0  );
                } catch (Exception e) {
                        fail("" + e.getMessage());
                }
        }
@Test
        public  void testOutputDev(){
                System.out.println("        testOutputDev ... " );
                try {
                        env.init();
                        assertTrue("testOutputDev", env.getOutputView() != null  );
                        env.println("Hello world");
                        env.getOutputView().addOutput( "Hello world again ");
                         Thread.sleep(2000);
                 } catch (Exception e) {
                        fail("" + e.getMessage());
                }
        }
@Test
public  void testAddButton(){
        System.out.println("        testAddButton ... " );
        try {
                env.init();
                env.println("Hello world");
                IActivity activity = new ActivityDebug();
                Panel cmdPanel = env.addCmdPanel("", new String[]{"Click"}, activity);
                assertTrue("testAddButton", env.getNumOfPanels() == 1  );
                Thread.sleep(2000);
                env.println(""+activity.getState());
                Thread.sleep(2000);
                assertTrue("testAddButton", cmdPanel != null  && activity.getState().contains("Click") );
         } catch (Exception e) {
                fail("" + e.getMessage());
        }
}
@Test
public  void testInputDev(){
         try {
                env.init();
                env.addInputPanel();
                assertTrue("testInputDev", env.getNumOfPanels() == 1  );
                 Thread.sleep(4000);
                env.println("read ->" +  env.readln());
                System.out.println("        testInputDev ... " + env.readln());
                Thread.sleep(2000);
        } catch (Exception e) {
                fail("" + e.getMessage());
        }
}
}
```

5. A button as a `GUI` component can be easily introduced by using the `addCmdPanel` operation of `EnvFrame`:

IDevButton

```
public Panel addCmdPanel(String name, String[] commands, IActivity activity);
```

   The class `EnvFrame` implements the `addCmdPanel` operation by building a new *Panel* (returned as result of the operation) that implements as many buttons as the elements of the given `commands` array. Each of these buttons is an observable entity that, once 'clicked', calls the `execAction(String cmd)` operation of the given activity by passing its name (an element of the `commands` array) as argument.
6. The following diagram shows the interaction with reference to the *DevButtonGui* implementation class:



The `GUI` button is observed by a built-in handler (DefaultInputHandler) that calls the `execAction` of the `IActivity` implemented by the `DevButtonGui` button that is observed by a `DevButton`.

7. Counter controller
8. System controller: centralization point

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. Xtext. Xtext home page.
   http://www.eclipse.org/Xtext/.