

A Button-Led systems: from objects to actors

DISI-Cesena// edited by Antonio Natali

Alma Mater Studiorum – University of Bologna
via Venezia 52, 47023 Cesena, Italy

Table of Contents

A	Button-Led systems: from objects to actors	1
	<i>DISI-Cesena// edited by Antonio Natali</i>	
1	Requirements	2
1.1	Software life cycle process	2
2	Technology-based design	3
2.1	Function-based software	3
2.2	Objects	3
2.3	Observable and Observers	4
2.4	Modelling a Button	4
3	An object-based design	5
3.1	From BLS Mock devices to physical devices	6
3.1.1	Dependency injection and Inversion of Control	6
4	Evolving the BLS into a distributed system	7
5	Devices as microservices	7

1 Requirements

Every **IOT** system usually performs a basic set of actions:

- Acquire data from sensor devices.
- Perform some control action.
- Send commands to actuator devices.

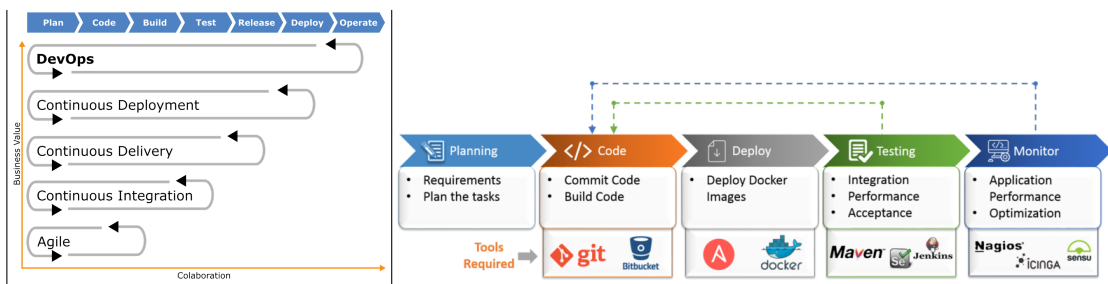
In this very basic demo, we use a **Button** as a **sensor** and a **Led** as an **actuator** and the control action represents our business logic. Examples of the business logic implemented by our Button-Led (**BLS**) system are:

1. **ROnOff**: the Led is **turned on/off** each time the Button is pressed.
2. **RBlink**: when the Button is pressed, the Led **starts blinking**. When the Button is pressed again, the Led **stop blinking**. And so on.
3. ...

Our goals can be summarized as follows:

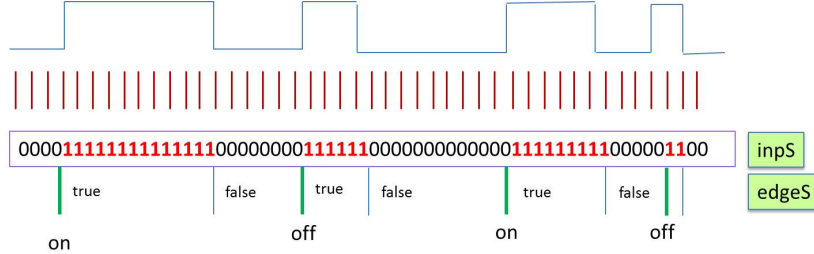
1. Define a 'technology-independent' architecture/prototype of the BLS in a local, **tightly coupled** environment.
2. Specialize the initial prototype according to different technologies. For example:
 - (a) The devices are implemented as **Mock** objects in a virtual environment.
 - (b) The devices are concrete things controlled by low-costs devices such as **Arduino/RaspberryPi**.
3. Modify the first working prototype into a **loosely-coupled** (distributed) system in which each device works within its own computational node.
4. Modify the distributed prototype by 'transforming' each device into a (**RESTful**) **service**.
5. ...

1.1 Software life cycle process



2 Technology-based design

The button is a source that emits a wave that can be sampled.

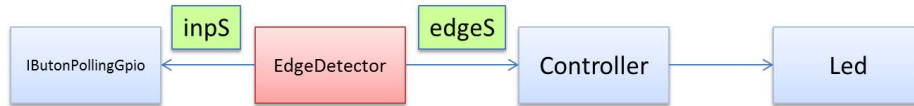


At this level, the problem requires that the following elaborations on the basic input

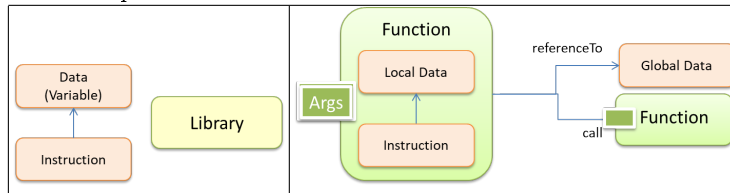
- the detection of the edges in the input sequence
- the detection of edges of type "low to high"

2.1 Function-based software

The responsibility of these functions can be given to two new different entities: an entity *EdgeDetector* and an entity *Controller* that realizes the "business logic" of the system.



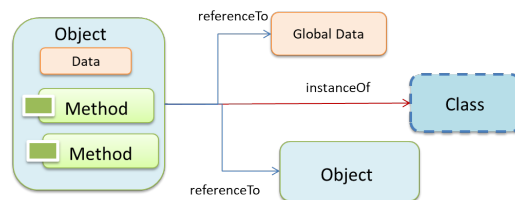
The code can be structured in imperative style, by using **functions** as a first kind of software component:



If an (application) function is called each time a new input becomes available, the system is 'reactive' or **event-driven**.

As an alternative, both the *EdgeDetector* and the *Controller* can be modelled as finite state machines (FSM) working as *transducers*. They can be viewed as *objects* interacting via procedure-calls.

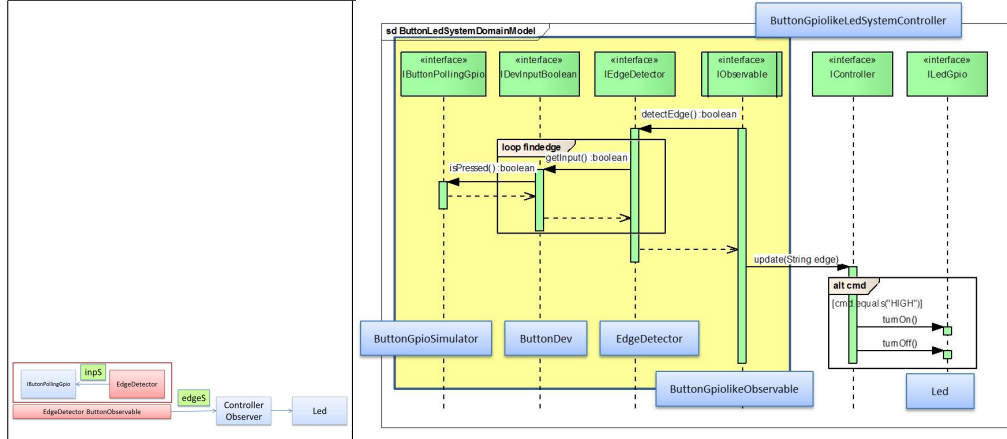
2.2 Objects



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.

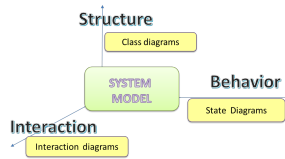
2.3 Observable and Observers

A more flexible architecture can 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.



However, starting from the idea that a Button is an 'edge detector' device is a too low-level approach for modern software applications. An effort has to be made to introduce a more appropriate **model** of the Button entity in terms of structure, interaction and behavior.

2.4 Modelling a Button



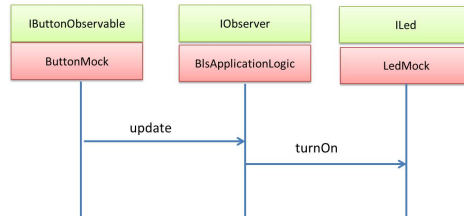
From the **structural** point of view, a button is intended by a customer as an **atomic** entity whose **behavior** can be modelled as a **finite state machine (FMS)** composed of two states ('pressed' and 'unpressed'). The state transition is performed by some agent *external* to the system (an user,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

3 An object-based design

Working in the conceptual space of 'classical' object oriented software development, the logic architecture of the BLS can be summarized by the following UML interaction diagram:



The computation starts from the observable device (*Button*), that calls a method of the object devoted to implement the application logic (that works as the *Observer*). An example is given in the project `it.unibo.bls.oo` that is based on the Java language. The working directory of the project is structured as follows:

- The package `it.unibo.bls.interfaces` includes the definition of the object interfaces.
- The package `it.unibo.bls.devices` includes the implementation of the object interfaces related to the devices (*Button* and *Led*). For each device, two different implementations are given: a *Mock* device and a 'virtual' object implemented with a GUI.
- The package `it.unibo.bls.applLogic` includes the definition of the object that implements the application logic.
- The package `it.unibo.bls.appl` includes the *Main* programs.
- The `test` directory includes examples of test units.

A software system working with the *Mock* devices should include a **configuration phase** to create the system components (objects) and properly connect them, according to the system architecture design:

```
1 public class MainBlsMockBase {
2     private IButton btn;
3     private ILed led;
4     //Factory method
5     public static MainBlsMockBase createTheSystem(){
6         return new MainBlsMockBase();
7     }
8     protected MainBlsMockBase( ) {
9         createComponents();
10    }
11    protected void createComponents(){
12        led = LedMock.createLed( );
13        BlsApplicationLogic applLogic = new BlsApplicationLogic(led);
14        btn = ButtonMock.createButton( applLogic );
15        led.turnOff();
16    }
}
```

Listing 1.1. MainBlsMockBase.java: configuration

The system defines also some working activity, to cause the change of the state in the *ButtonMock*:

```
1 public void doSomeJob() {
2     System.out.println("doSomeJob starts" );
3     ((ButtonMock)btn).press();
4     Utils.delay(1000);
5     ((ButtonMock)btn).press();
6     System.out.println("doSomeJob ends" );
7 }
```

Listing 1.2. MainBlsMockBase.java: simulated action

Since every system is a **composed** (i.e. it is a **non-atomic**) entity, it provides also **selector-methods** to get its components:

```

1 public IButton getButton(){ //introduced for testing
2     return btn;
3 }
4 public ILed getLed(){ //introduced for testing
5     return led;
6 }

```

Listing 1.3. MainBlsMockBase.java: selectors

The selectors can be useful during the testing, to access the state of the single devices. Finally, there is the **main** method:

```

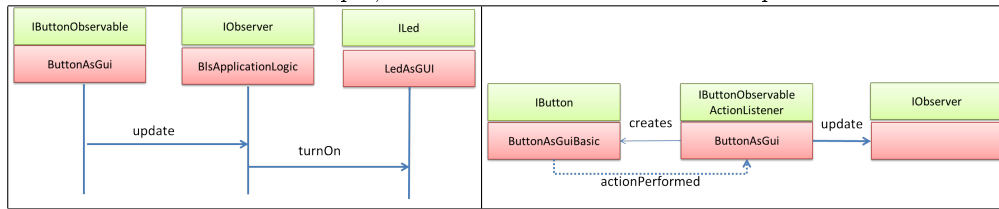
1 public static void main(String[] args) {
2     MainBlsMockBase sys = createTheSystem();
3     sys.doSomeJob();
4 }
5 }

```

Listing 1.4. MainBlsMockBase.java: the main method

3.1 From BLS Mock devices to physical devices

The logical architecture previously introduced does not change if we replace the **Mock** devices with concrete devices: For example, in the case of virtual devices implemented with a **GUI**:



The class **ButtonAsGuiBasic** implements a GUI-based Button by extending the class **java.awt.Button**. The class **ButtonAsGui** implements the concept of Button as Observable entity.

Note that **ButtonAsGui** re-uses the class **ButtonAsGuiBasic** but without exploiting inheritance. Rather, it creates an instance of **ButtonAsGuiBasic** and works as its listener. This behaviour is caused by the fact that Java does not support multiple inheritance for classes and **ButtonAsGui** already extends the class **java.util.Observable**.

In a more general perspective, we could build this part of our software system by exploiting the **Dependency Injection** pattern.

3.1.1 Dependency injection and Inversion of Control .

In software engineering, an **injection** is the passing of a dependency to a dependent object (a client) that would use it as a service. In the **dependency injection** pattern, passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement. Dependency injection is one form of the broader technique of **Inversion of Control**. The client delegates the responsibility of providing its dependencies to external code (the **injector**). The client is not allowed to call the injector code.

5 Devices as microservices

