A Button-Led systems:
from local objects to distributed systems

DISI-Cesena
Antonio Natali

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

Table of Contents

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. ...

> Reference project: `it.unibo.bls.oo` on `https://github.com/anatali/IotUniboDemo`.

1.1 Goals

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: see Subsection **??**.
4. Modify the distributed prototype by 'transforming' each device into a (`REST`ful) `service`.
5. ...

1.2 Software life cycle process

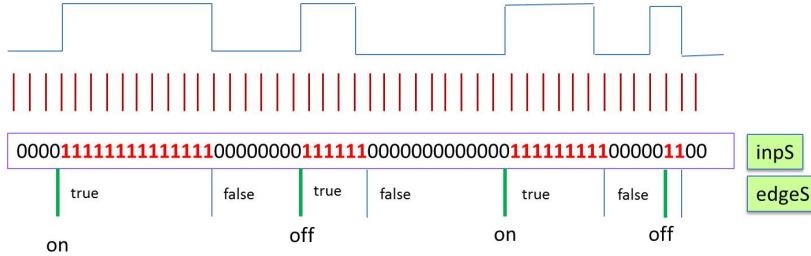> Discussed in `BBS-module5 : Software production`.
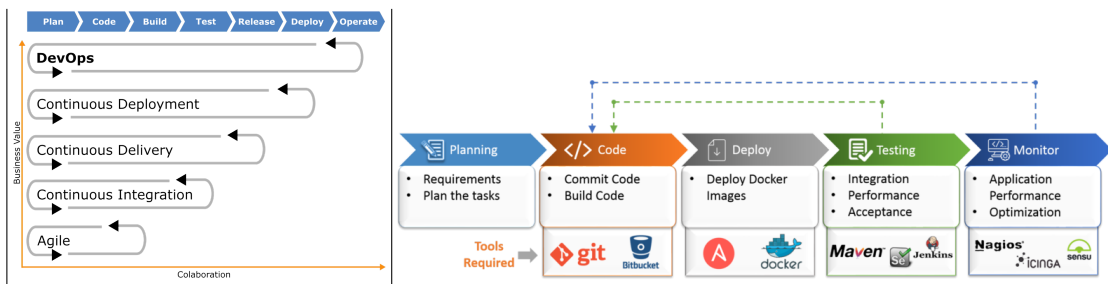
1.3 Bottom-up or Top-Down?
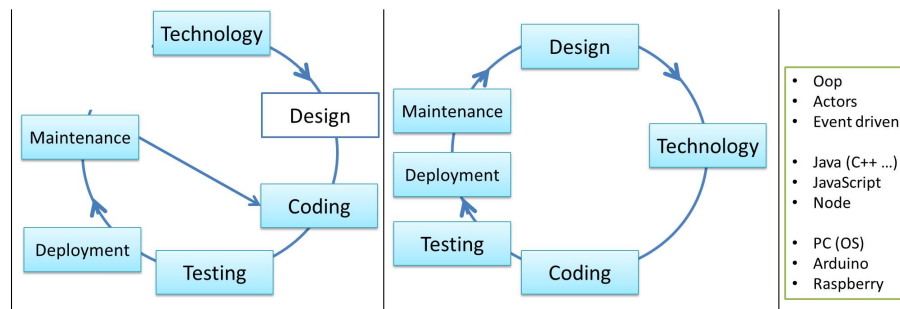
2   Technology-based design

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.
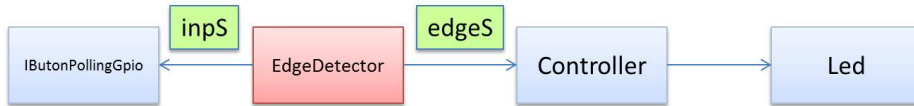

The code can be structured in imperative style, by using `functions` as a first kind of software component:
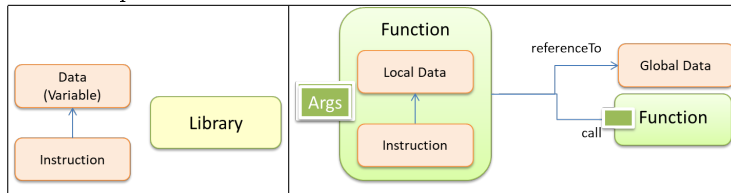
If an (application) function is called each time a new input becomes available, the system is 'reactive' or `event-driven`.

2.2   Object-based software

As an alternative of the function-based code of Subsection 2.1, 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.
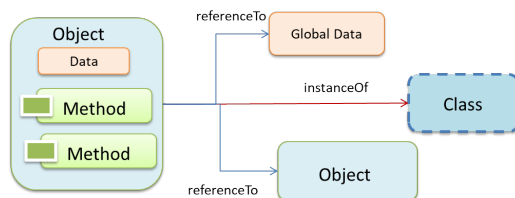
In any object-oriented model, all the computation usually takes place within a single thread. In our case the main thread could be the thread related to the component that performs the polling of the wave, i.e. the `EdgeDetector`. In this case, the *Controller* is called by the *EdgeDetector* that, must explicitly know the *Controller* in order to call it.

2.3 Observable and Observers

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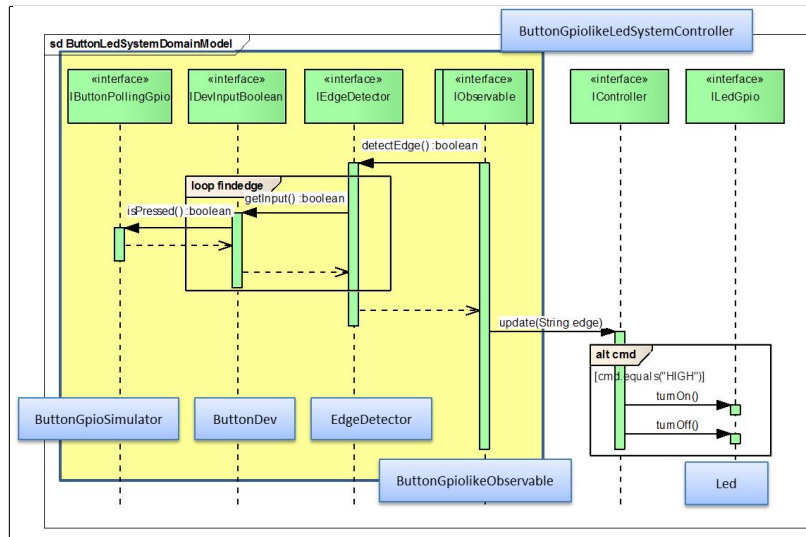
> However, staring from the idea that a Button is an 'edge detector' device is a too low-level approach for modern software applications. An effort has to be made to introduce a more appropriate `model` of the `Button` entity in terms of structure, interaction and behavior.

2.4 Models
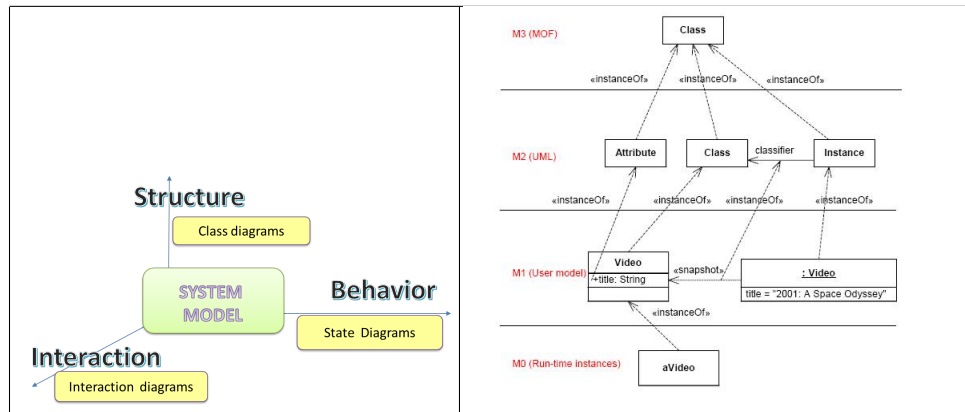
2.5 Modelling a Button

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:
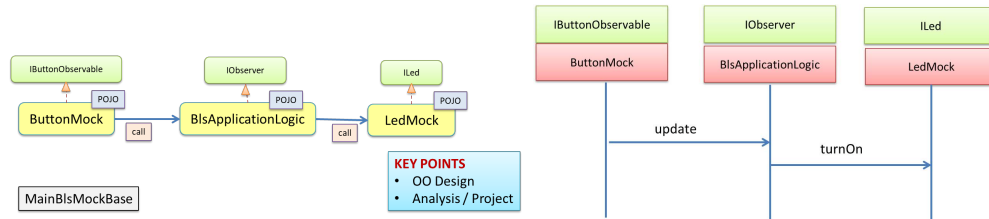
The computation starts from the observable device (*Button*), that calls a method of the object devoted to implement the application logic (that works as the `Observer`). An example is given in the project `it.unibo.bls.oo` that is based on the `Java` lanaguage. The working directory of the project is 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  private ILed led;
2  //Factory method
3      public static MainBlsMockBase createTheSystem(){
4          return new MainBlsMockBase();
5      }
6      protected MainBlsMockBase( ) {
7          createComponents();
8      }
9      protected void createComponents(){
10         led = LedMock.createLed( );
11         BlsApplicationLogic applLogic = new BlsApplicationLogic(led);
12         btn = ButtonMock.createButton( applLogic );
13         led.turnOff();
14     }
15     public void doSomeJob() {
16         System.out.println("doSomeJob starts" );
```

Listing 1.1. `MainBlsMockBase.java`: configuration

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

```
1          ((ButtonMock)btn).press();
2          UtilsBls.delay(1000);
3          ((ButtonMock)btn).press();
4          System.out.println("doSomeJob ends" );
5      }
6      public IButton getButton(){ //introduced for testing
7          return btn;
```

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:

```
    }
    public ILed getLed(){ //introduced for testing
        return led;
    }
public static void main(String[] args) {
    MainBlsMockBase sys = createTheSystem();
```
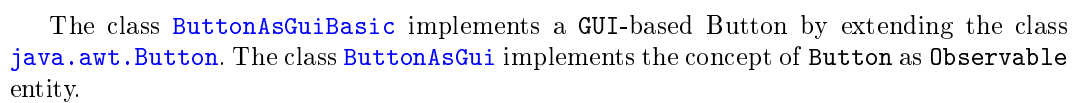
Listing 1.3. `MainBlsMockBase.java`: selectors

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

```
    sys.doSomeJob();
 }
}
```

Listing 1.4. `MainBlsMockBase.java`: the main method

4 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.

4.0.1 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.
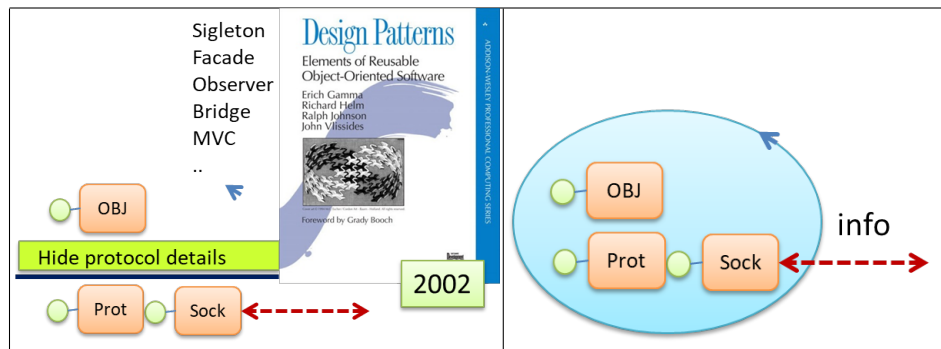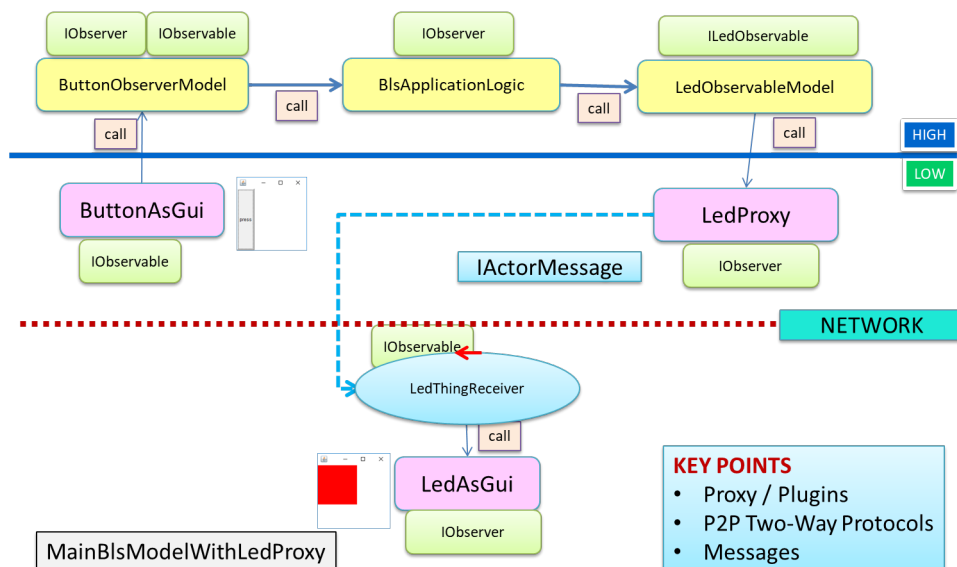
4.1 Devices on Arduino
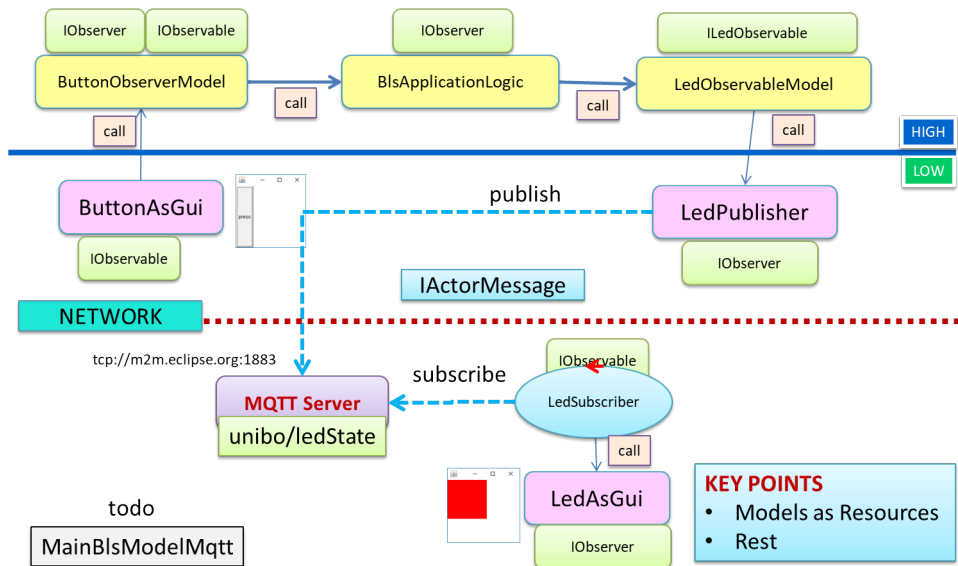

4.2 Devices on Raspberry
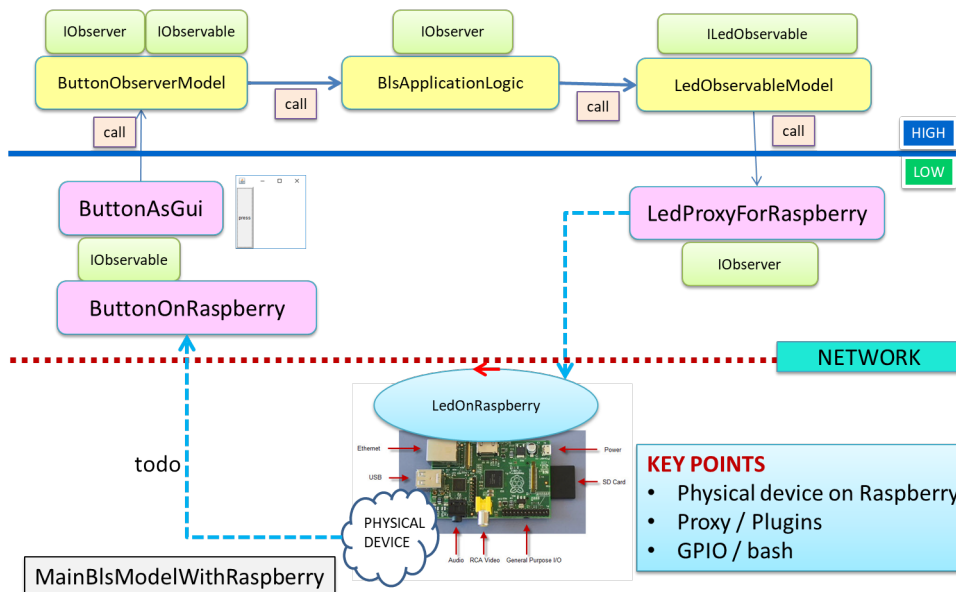
5 Evolving the BLS into a distributed system

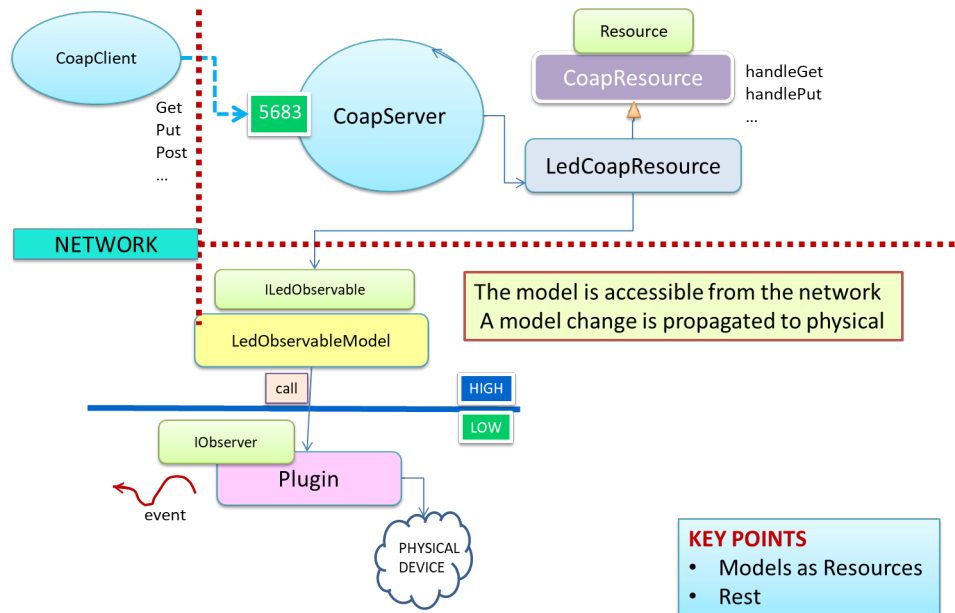## 5.1 Led as a remote devices: the proxy pattern
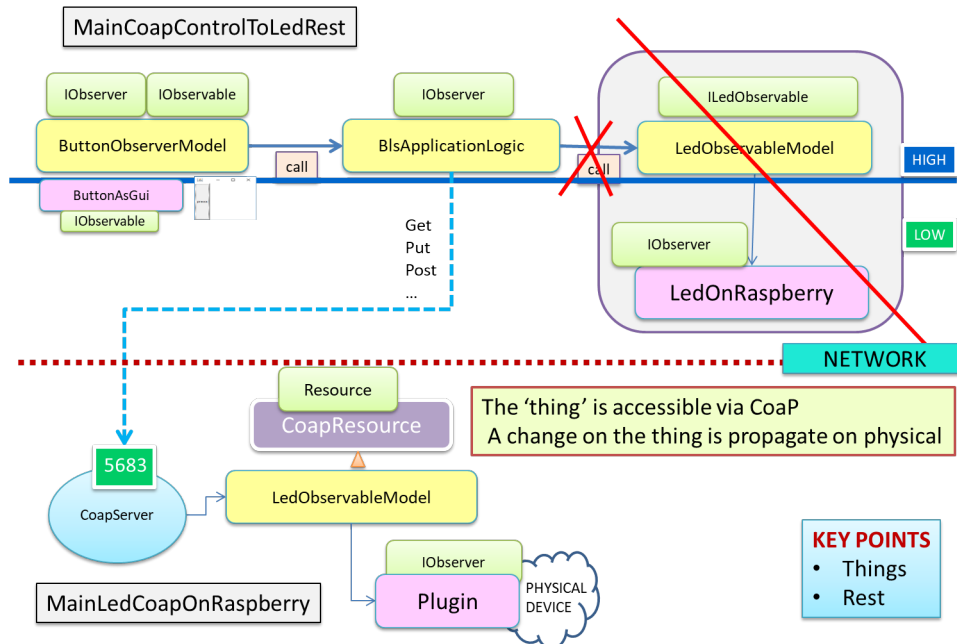
5.2 Led as a remote device. the publish-subscribe pattern

6 CoAP

6.1 Led as a (CoAP) thing

6.2 BLS front end