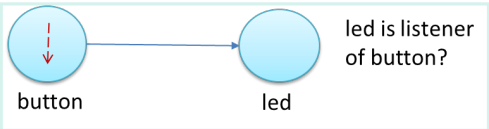
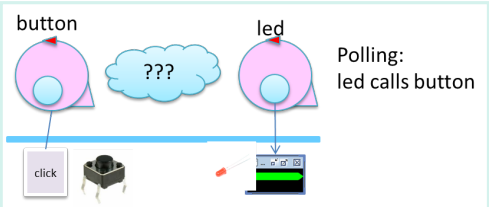
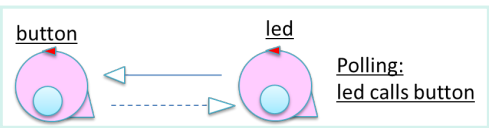
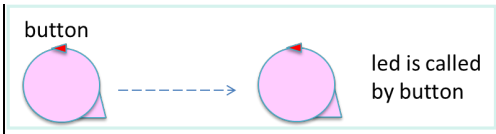
BLS2020: The button-led system

Introduction

Case-studies based on the usage of one or more *button* and one or more *led* are simple but not simplistic, since they include the main aspects of information-processing systems: **input**, **output** and **elaboration**.

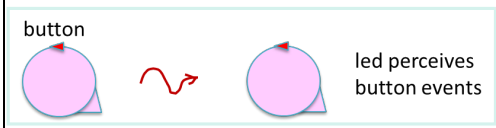
Since **there is no code without a project, no project without problem analysis and no problem without requirements**, let us present here a possible scenario related to the case-study **led-blink** proposed in: ApplProposalo.html | A set of tasks to do .

Concrete devices  	Technology dependent layer. See for example: LedMock.java LedAsGui.java ButtonAsGui.java
0 - Object oriented  	The *button* is an 'active' object and the *led* is a **'event listener'**. See for example a definition of event listener that requires some critical analysis. **Analysis** **pros**: very efficient **cons**: the system solves only the problem at hand, no effort for a reusable architecture is made; also, the classes represent both the component and the control policy and, with this kind of interaction, the system is only runnable locally. See also the (horrible) system: • ButtonAsGui.java • LedListenerAsGui.java • MainBlsGuiNaive.java
1 - Actor-based  	The *button* and the *led* are (qak) actors able to interact. Technological details are hidden into a proper adapter' object. The problem is to define the **'logical interaction'** between them. At the moment, we can introduce a model for the led: ledalone.qak **Analysis** **pros**: abstraction between the component and its software representation easy to model. **cons**: if only the button and the led actor are present, the control policies have to implemented in the button and/or the led.
2 - Actor-based: polling  	The *button* and the *led* are (qak) actors able to interact. The *led* sends a **request** to the *button* in order to know its state. **Analysis** **pros**: the led can percieve the button's change of state readily. **cons**: the polling system is very expensive in terms of communication. Most of the times there will be a negative answer from the button, so many useless messages will be exchanged between the actors. Also, the led has to know the button to be able to ask for its state.
3 - Actor-based: direct call	The *button* forwards a **dispatch** to the *led* each time it changes its state. **Analysis**



pros: the messages exchanged are reliable and only sent when the button changes state.
cons: the button has to know the led to be able to send a message to it.

4 - Actor-based: events

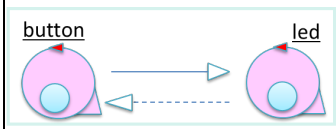


The *button* emits an **event** each time it changes its state, hoping that the *led* can perceive it.

Analysis

pros: the button and led don't have to know each other in order to exchange messages.
cons: an event is not a reliable message, so the button actor can't be sure that the led has actually received the message.

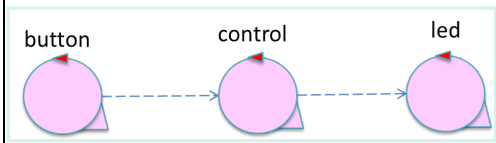
5 - Actor-based: request



Analysis

pros: the communication is reliable and status messages cand be exchanged to be able to adjust the state according to the other component's state.
cons: the two actors have to know each other to be able to establish this kind of connection.

6 - Introducing another component

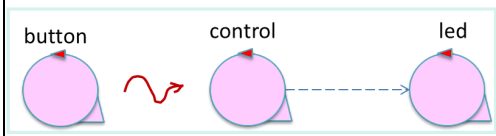


The *button* does not interact with the *led*, but with another component (*control*) that embeds the **'business logic'** of the system.

Analysis

pros: loose coupling makes for a better software architecture; this way if the button or the led become some other kind of I/O components it's easy to implement. Also, the control policies can be grouped inside the control actor.
cons: none comes to mind.

7 - The *control* as button-observer

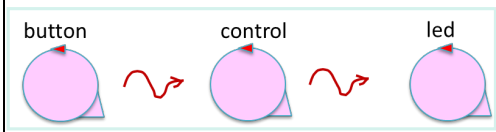


The *control* is an 'observer' of the *button* since it is able to **perceive events** emitted by the *button* when it changes its state.

Analysis

pros: the button and the control don't have to know each other.
cons: as per [4], the only con that I can think of is the non-reliability of events.

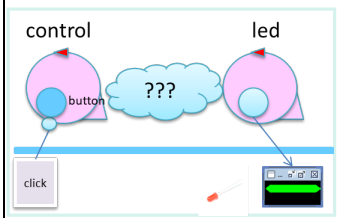
8 - All event-based observers



Analysis

pros: very light, no direct messages exchanged between any actors.
cons: not a realiable communication in a distributed system.

9 - Embedding the *button*



The *button* is just a resource embedded into the *control* to allow human interaction.

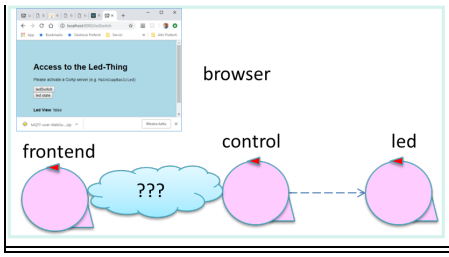
Analysis

pros: there is no need for message exchange between the control and the button.
cons: embedding a specific source of input in the control makes for a tightly coupled system, so if the button becomes a lever in the future the control actor has to be changed.

10 - The *button* in a browser

Human interaction is based on the usage of a browser provided by a *frontend* component that interacts in some way (to be designed) with the *control*.

Analysis

	**pros**: no need to distributed client side application and the webside can be designed using standard technologies, such as HTML. **cons**: the web communication model isn't good for all applications and, since it's a general purpose technology, it can't be optimized for a particular application.

Other considerations

Regarding my idea for the system, I just wanted to clear up why I decided to implement it with the reqest-reply kind of communication: I had imagined the system would work as [10], but I ended up designing it as [5], hence the r-r communication.
I was wrong implementing the control for the led inside it and not making another "middle man" actor to act as intermediary between them.
The other reason why I designed the system that way was that I could be sure to match the states of the button with the state of the led, so that if the button was pressed it would actually change state if the led answered with an ack. I see now how this solution is expensive and not very realistic, since the button should be independent from the actual state of the led and a single button can control more than one led eventually.