Software Systems Engineering
process report template

Roberto Casadei Massimiliano Martella Roberto Reda

November 20, 2014

1 Introduction

This report represents a snapshot of the current status of the "ButtonLed" project (actor-based, new style).

2 Vision

We believe that a good software product is the natural result of a **mature, systematic process**.

We believe that there is no code without design, no design without analysis, no analysis without requirements. So, we recognize the causal progression from the *why* (vision/goals/requirements) to the *what* (analysis) and finally to the *how* (design/code).

We recognize the importance of analysis for a mature process, but we don't want to waste time by just producing informal knowledge. Instead, we would like to **capitalize the effort spent in analysis by producing something that has value, that is, working software (prototype) and reusable frameworks**. That is, we believe that *reuse* is prominent in what we do.

We also think that following a **top-down approach**, i.e., from the problem to the technology, is extremely valuable because it can adequately support our goals. In doing so, we would like to constantly **evaluate the abstraction gap** in order to gain a better vision of what we need to *effectively* build the kind of systems we are considering and also as a way to *innovate*.

Moreover, as we are dealing with (complex) software systems, we believe that a **systemic approach** fits more than an algorithmic approach and gives us access to more powerful conceptual tools.

We think that a software development process built upon the principles outlined above can maximize the value of the effort spent during analysis and development, effectively produce software architectures that are resistant to requirements change, and even turn those changes into an opportunity to increase the organizational know-how.

In this very case study, our emphasis is on the **relationships between the conceptual tools used in a project and the project itself** (i.e., processes and artifacts).

We firmly believe that, as we choose to follow a top-down approach, the results from (requirements and problem) analysis should NOT depend on the reference paradigm and, least of all, on the technological platform.

However, we are aware that paradigms (and, in general, the conceptual spaces projected by a given technology) are similar to *lens* that make certain aspects clearly visible while completely hiding others. This is not bad, because **complex systems** typically consists of multiple **(orthogonal) dimensions**, and each dimension might require different (conceptual and practical) tools to be adequately investigated.

As a consequence, we think that, in order to be able to effectively tackle the problems at hand, we should

1. be aware of the limitations of adopting a single point of view
2. be aware of the current reference conceptual space in use and
3. try to find and use the (conceptual) tools that most fit the problem

so that we can reduce both the cognitive load and the abstraction gap (if a platform exists or can be built).

In other words, we think that the use of the right tools can help minimizing the *accidental complexity*, thus allowing us to **focus on the essential complexity** of our systems.

Last but not least, we believe that the **trade-off between long-term and short-term productivity** should be guided (at least) by the potential for reuse and the lifetime of the software artifacts produced. In this view, addressing the abstraction gap is key to grow and optimize in a sustainable manner.

3 Goals

We use the ButtonLed system as an opportunity to actualize the vision. In this case study, the technology hypothesis is given by the QActor framework and conceptual space.

We are aware that the system we are building is just one of the countless applications that could be built in this domain. So, we generalise/abstract from the issues we encounter in a sustainable manner, so that we can **factor domain-knowledge out of the specific system**.

Also, by analyzing the requirements and the problem, we would like to produce a **working prototype by following an incremental (*piecemeal growth*) approach**.

In particular, we would like to evaluate the cost of passing from a homogeneous, concentrated system to a distributed, etherogeneous system.

Moreover, we would like to correlate what we do with the following concepts:

- Models ($\Rightarrow$ Structure, Interaction, Behavior) as a way to capture the essential characteristics of what we talk about
- Software reuse as the key element to deal with bounded resources
  - Design patterns as synthetic analyses and design ideas for recurring problems
  - Framework vs. pattern vs. middleware
- Technologies as specific ways to actualize designs but also as conceptual spaces (i.e., "representing" a paradigm)
- Logic architecture as the main result from analysis and possibly our main specification of the problem at hand
- Traceability (from requirements to architecture to code) as a way to support "informational consistency" within the project
- Specific vs. schematic vs. generic parts of software

In addition, here we shift the focus towards a **systematic comparison** of what we did in the two approaches: object-based ("old-style") vs. message-passing ("new-style").

In particular, we want to find an answer to the following question: **How do the project phases (analysis, design, ..) change when the reference paradigm/technology-hypothesis changes?**

Moreover, we also want to become aware (by seeing with our own eyes) of the **(long-term) advantage that results from building frameworks that adequately fill the abstraction gap that had been identified**.

4 Requirements

> " Design and build a ButtonLed software system in which a Led is turned on and off each time a Button is pressed. In particular, the system will have the button on an Android device, the controller on a RaspberryPi, and the led on Arduino. "

5 Requirement analysis

We follow a *systemic approach*. From the requirements, it follows that the system is structurally composed of three **subsystems** (for which we already know the deployment, see the Deployment section 11), as expressed by the following textual, formal model (specification):

```
context( ctxofbutton, "Address1",  "Protocol1", "Port1" ).
context( ctxofcontrol,   "Address2", "Protocol2",  "Port2" ).
context( ctxofled,   "Address3", "Protocol3",  "Port3" ).
```

where subsystems are modelled as *contexts*. The addresses, the protocols, and the ports will be specific to a particular system deployment (Note that other means for specifying the distribution configuration might be used – they are unessential at this level).

We recur on the structural dimension. We also know what **components** the system is composed of, and how they are distributed in the aforementioned subsystems.

```
qactor( qacontrol, ctxofcontrol ).
qactor( qabutton, ctxofbutton ).
qactor( qaled, ctxofled ).
```

By modelling the button, the led, and the controller as *qactors*, we say that they are entities which

- belong to a single context (subsystem)
- interact by message-passing

To see what a context, a message, a name is, please refer to the QActor model developed by our software house.

5.1 Use cases

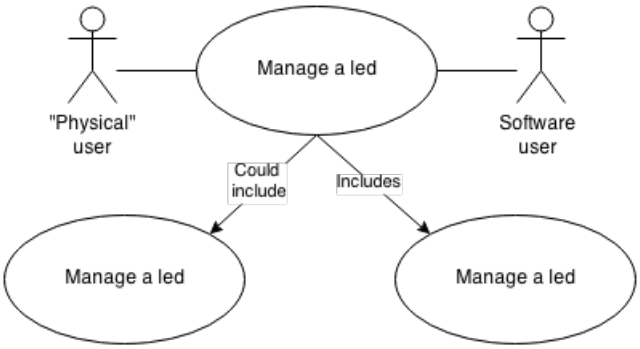


Figure 1: Use case 1

5.2 Scenarios

(Skipped) It's not a primary goal of this case study.

5.3 (Domain)model

From the requirements we know that:

- The button has a state (pressed and not pressed) and, as external entities are interested in that state, it follows that it should be an *observable* entity (see the *Observer* pattern)
- The led has a state as well (on, off) and properties such as the color.

The requirements explicitly give information about the structure of the system and the components, and express a cause-effect relationship between the press of the button and the update of the state of the led. Instead, the dimensions of behavior and interaction are only implictly and partially described by an *observational* point of view.

To explain what we mean by “led" and “button", we provide a **model** for them.

By following an internal convention, we express those models using Java **interfaces**. As interfaces are not sufficient, we also provide **test plans** in order to better define our intended semantics.

By analyzing our idea of “button" and “led", we also note that, in the domain of the *Internet of Things*, the led and the button are particular instance of the concept of **device**. Moreover, we know that a particular device can be realized (implemented) with different technologies (see the *Bridge pattern*)

See the following interfaces:

- it.unibo.buttonLed.interfaces.IDevice
- it.unibo.buttonLed.interfaces.IDeviceInput
- it.unibo.buttonLed.interfaces.ILed
- it.unibo.buttonLed.interfaces.IButton

5.3.1 Behavior

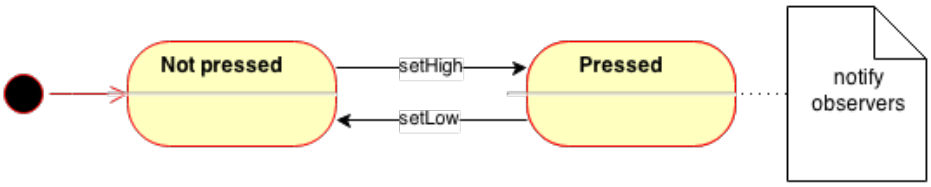


Figure 2: Button: behavior

Figure 3: Led: behavior

5.4 Test plan

See *it.unibo.group08.buttonled.test* project.

6 Problem analysis

6.1 Logic architecture

The system is composed of three main components: the led, the button, and the controller.

From the requirements, we also know that they are on different subsystems (see Requirements Analysis in Section 5).

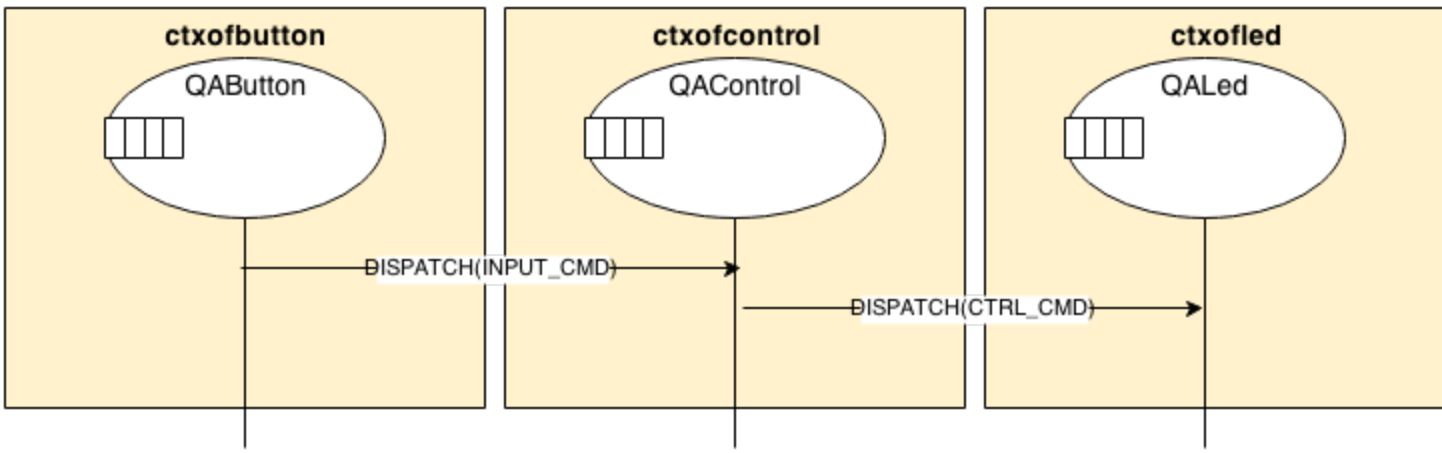


Figure 4: Logic Architecture – Structure/Interaction view

Please consult the software house's internal reference to get a description of the semantics of the "DISPATCH" messages.

For simplicity, at the moment we can think at the communication language as described by the following grammar:

```
INPUT_CMD = 0 | 1
CTRL_CMD = 0 | 1
```

The controller's behavior can be represented by the following FSM:

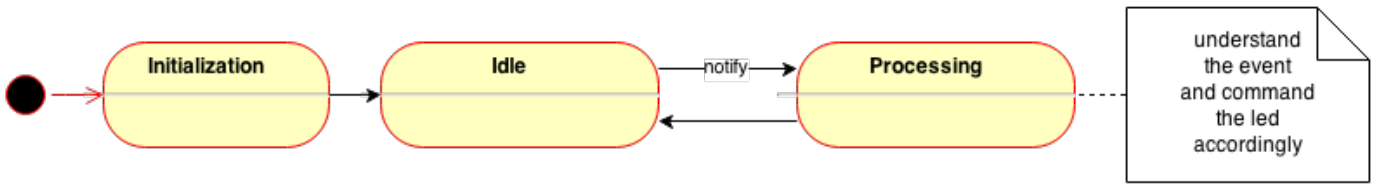


Figure 5: Controller: behavior

6.2 Abstraction gap

Our technology hypothesis is given by the QActor framework.

The problem and the practical issues encountered through the analysis phase point out that a gap exists with respect to the following themes/areas:

- **Model specification**: we have specified the models of the system and the components using Java interfaces, test plans, and (custom variants of) UML. However, our need of producing models that are both formal and expressed at the right level of abstraction is not adequately satisfied.
  Some work has been done with the QActor framework, but it is limited to the structural dimension (see Section 5).

- **"General" system concepts**: OOPLs do not (directly) support our system view (which we use instead of an algorithmic view) through high-level concept such as "environment", "situated entity", "named entity", ... ⇒ our effort in filling this gap has resulted in the *it.unibo.system* package (within *it.unibo.noawtsupports*)
  The QActor framework also introduces the notion of *context*, which is how we model a subsystem.

- **Rapid prototyping**: our goal to have analysis end up with a (graphical) prototype is not effectively supported by Java GUI libraries (AWT, Swing) ⇒ our effort in filling this gap has resulted in *it.unibo.envBaseAwt*

- ~~Communication infrastructure~~: The QActor framework adequately supports a *message-passing* interaction style. The infrastructure hides technology-specific details (such as connection setup, protocols, ...) and the application designer can just focus on the application logic (no need for proxies etc.).
- **Interaction semantics**: the high-level semantics of the interaction between the system's components is not directly captured by neither the tools we use to express models nor by our platform. We conventionally introduced some names (dispatch, request, signal, ...) to specify a vocabulary for (informally) talking about interaction at the level of abstraction we consider more appropriate. However, this semantics is not directly supported. However, as our infrastructure already supports the sending/receiving of messages, it should be easy to extend it with interaction forms of higher level.
- ~~Infrastructure configuration and setup~~: The QActor framework initializes and configures the system based on the system's structural specification. So, the related details/issues have been completely hidden to the application designers.

6.3 Risk analysis

If we don't adequately fill the abstraction gap, and if we don't abstract from technology-specific and application-specific details, we risk to have significant losses in case of change of (both functional and non-functional) requirements.

7 Work plan

The project team consists of three engineers. So, ideally, the work is subdivided on a subsystem-basis:

- One developer will design/develop the Android subsystem
- One developer will design/develop the RaspberryPi subsystem
- One developer will design/develop the Arduino subsystem

Here, it is important to clearly define the subsystems' interfaces (communication media, communication language, ...). An architectural design will be defined (see Section 8) to better specify the constraints and interactions.

Moreover, certain code/naming conventions should be defined in order to preserve the internal quality (coherence) of the system.

8 Project

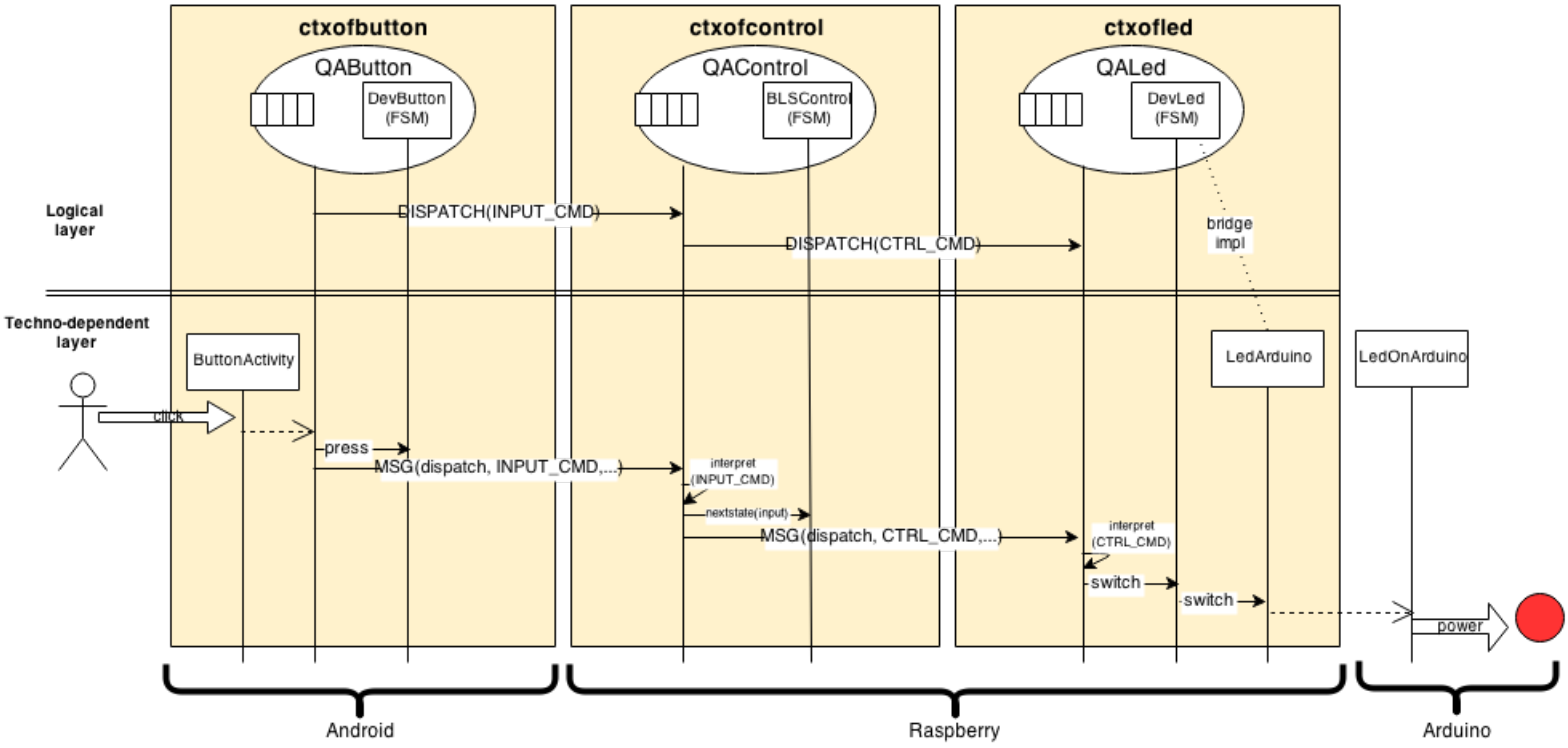


Figure 6: Design architecture

Notes:

- Logical subsystems (contexts) vs. physical subsystems
- Point-to-point communications require the sender to know the recipient's name

9 Implementation

See *it.unibo.group08.buttonled.logical* and *it.unibo.group08.buttonled.actorsystem* projects.

10 Testing

See *it.unibo.group08.buttonled.test* project.

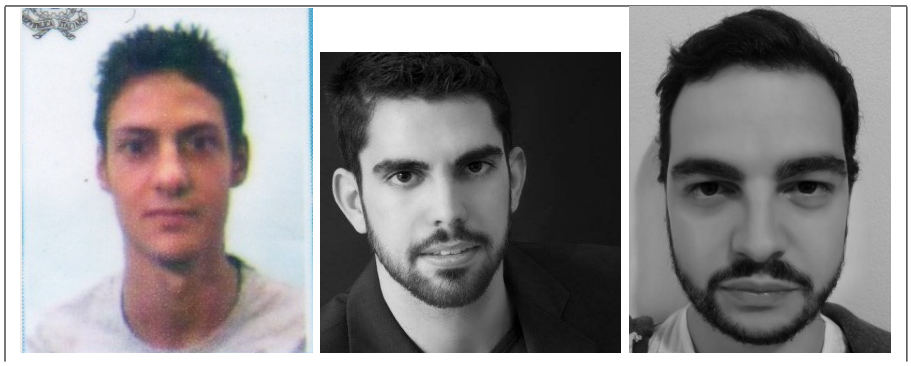
11 Deployment

Directly from the requirements, the following deployment configuration follows:

- An APK package for the Android subsystem
- A JAR package for the RaspberryPi subsystem
- A INO file for the Arduino subsystem

12 Maintenance

13 Information about the authors



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.