Final Theme System
process report

Gruppo 3 Aimi Niccoló, Gallegati Mattia, Murgia Antonio, Zanotti Andrea

Alma Mater Studiorum – University of Bologna
viale Risorgimento 2, 40136 Bologna, Italy
niccolo.aimi@studio.unibo.it; mattia.gallegati2@studio.unibo.it;
antonio.murgia2@studio.unibo.it; andrea.zanotti9@studio.unibo.it

1 Introduction

In this document we'll describe and discuss the process that led to the development of a robotic software system. We'll discuss how traditional software/project development techniques can and should merge with the scrum project management system.
We'll face the limits of traditional analysis approaches and purpose novel ones that better match with new programming paradigms.

2 Vision

Here we're collecting some visions that will inspire the development of this project and of software in general:

- There's no code without project, there's no project without problem analysis and there's no problem without requirements;
- There's no code without tests. That means that tests must be developed before the development of the software product;
- There's no code without documentation;
- The team that develops the tests should be different from the one who will realize the project;
- We should use a top down approach during the development phase and bottom-up approach during the implementation phase;
- Zooming? Trying to get closer.. We should look at things initially from a far point of view like a black box and then trying to get closer to the details of the system, white box.
- Any sufficiently advanced technology is indistinguishable from magic.
- Develop the code in an IOT prospective using the ICT principles.
- Create valuable software using less time and resource possible in order to optimize software development process (factorization).
- A feature does not exist unless a test validates that it functions.
- Software entities should be open for extension but closed for modification.

- We should always check the existence of prior projects, trends or patterns regarding the technological domain we are facing. If does exist something we should study it, not only to take advantage of it during the development but also to recognize its limits and to purpose some innovative approach coming from different technological domains;
- The development of the project should be technology agnostic;
- We should find or create a formal language able to describe the results of the problem analysis. It should be similar to spoken language. This language could take advantage of different programming paradigms (functional, declarative and so on) and different programming models (actor model, message passing model and so on). This language should not be tied to any technology implementation. There should be one or more parsers/compilers for this language that will generate source code for a specific platform. This generated code should be used as the skeleton of the real product.

3 Goals

The first goal is to develop a robotic software system using the most advanced programming paradigms.
A secondary goal is to develop and to take advantage of a novel technique of requirements and problem analysis that will lead to a formal representation of the problem and to source code generation. Finally the main goal is discuss how some change (monotonic extensions) of the requirements impact on a product whose production is based on 'formal' and 'technology independent' artifacts rather than on ad-hoc code.

4 Requirements

Design and build a (prototype of a) software system that, with reference to a differential drive robot (called from now on robot):

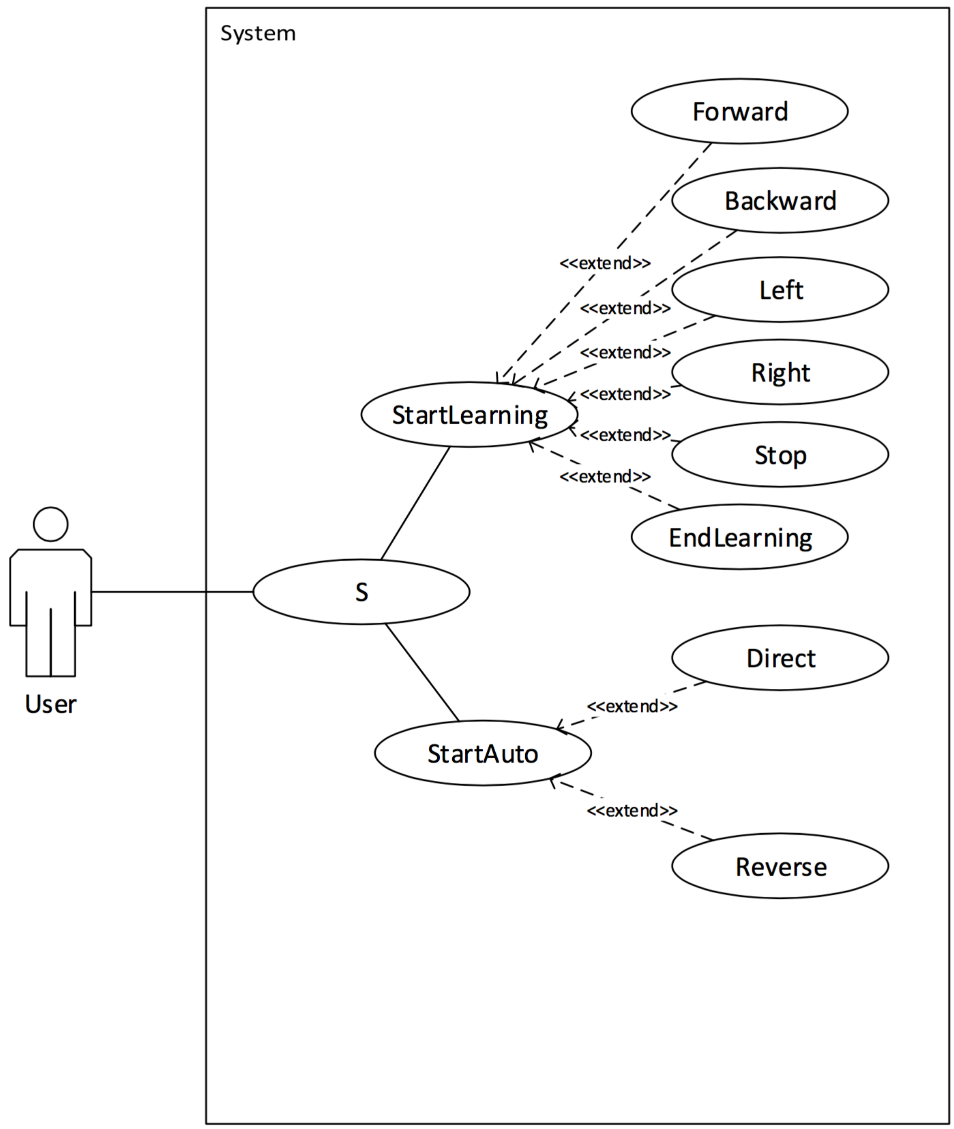
- allows a user to select between a 'learning phase' and a 'autonomous phase' during the learning phase, the user can send a sequence of move commands (e.g. forward, backward, left right, stop) to the robot. The robot must not only execute each command but it must also record the whole sequence of commands until the user decides to terminate the learning phase;
- after the termination of the learning phase, the user can tell the robot to enter the autonomous phase in a 'direct' or in a 'reverse' mode. During this phase the robot executes in autonomous way the sequence of moves it has learned, by complementing each move (e.g. forward->backward) if the selected mode is reverse;
- during the autonomous phase, the robot must be able to execute (as soon as possible) a stop command sent by the user.

After the development of this prototype, consider the possibility to enhance the functional capabilities of the robot, by allowing it:

– to perceive an obstacle during the autonomous phase and, once the obstacle is detected, to execute some alternative behaviour (in term of moves).

5 Requirement analysis

5.1 Use cases



5.2 Scenarios

Title	Execution of a sequence of actions by the robot
Description	The robot will repeat a sequence of actions previously recorded.
Relationships	“Reference to the previous use case”
Actors	User
Pre-Conditions	• User must be connected to the system via a user interface. • Robot must be “powered on” and connected to the network.
Post-Conditions	• The robot must be in an initial state where it can record new sequence of actions or repeat the recorded once.
Principle Scenario	1. The user can choose “StartLearning” to record a new sequence of actions on the robot, or “StartAuto” to let the robot execute the recorded sequence. This time we choose “StartLearning”. 2. The user commands the actions by the set of the allowed ones: Forward, Backward, Left, Right and Stop (Learning phase). When the user decides to finish this phase, he must choose “EndLearning”. 3. Now the user can choose again “StartLearning” to record a new sequence or “StartAuto”, this time we choose the last one (Autonomous phase). 4. The user can select “Direct”, so the robot will execute the actions in the same sequence of the user recorded ones. 5. At this point, the robot knows the sequence of actions saved in the learning phase, so in autonomous phase it replays those actions (in both directions) everytime the user wants, until a new learning phase is chosen.
Alternative Scenario	4. The user can choose “Reverse”: in this mode the robot will replay the sequence in a reversed way.
Open points	• Is it possible to add new actions in learning phase once it is finished?

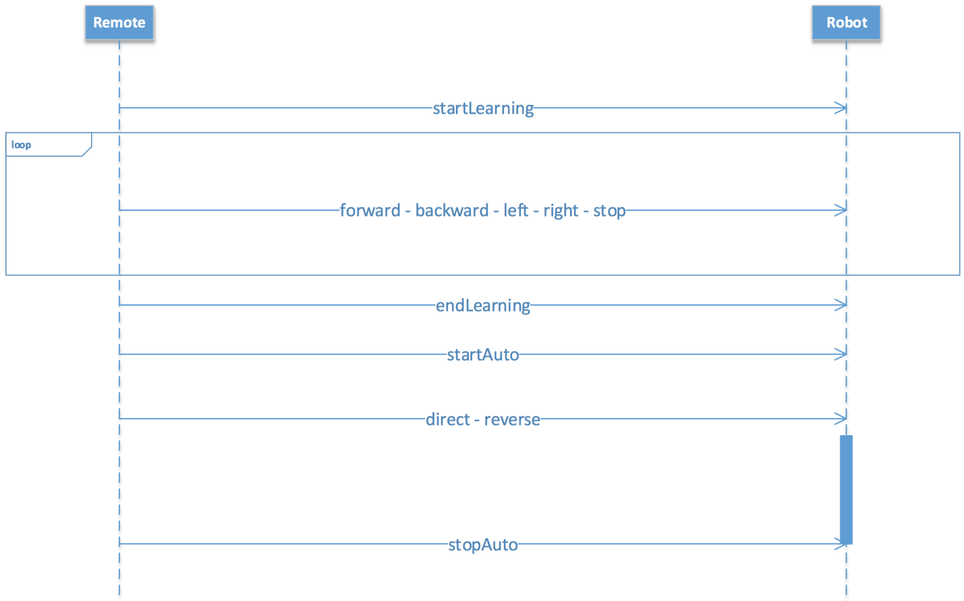
5.3 (Domain)model

Structure

ROBOT We consider the robot as a reactive and atomic entity.

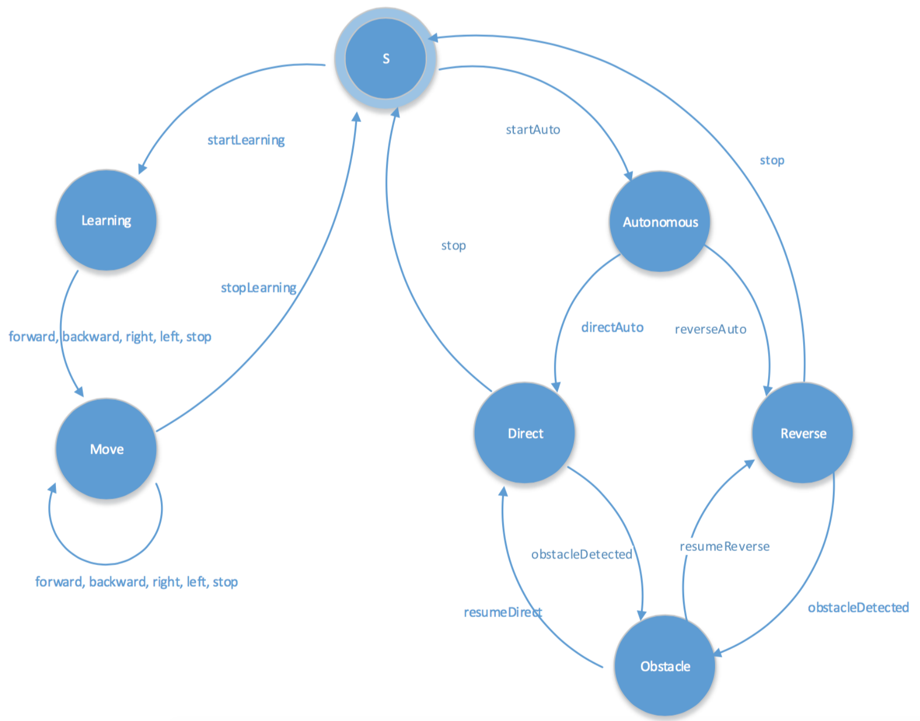
REMOTE We consider the remote as a proactive and atomic entity.

Interaction The interaction between the entities of the system is described with this sequence diagram:



Behaviour

ROBOT The behaviour of the robot is described with this FSM:



REMOTE The behaviour of the remote is described with this FSM:

5.4 Test plan

Test sui piani del robot e eventi remote

6 Problem analysis

Our technological hypothesis is JAVA and Object Oriented Programming. We will show that we need to develop a new infrastructure because the only features offered by the programming language chosen are not enough. To keep our code well developed, scalable, efficient and maintainable, we will use the most known development patterns. We must define an interaction language, that needs a communication standard. The communication standard will be essential during the Integration Test.

6.1 Abstraction gap

During the problem analysis we considered Java as our technological hypotesis. We realized that Java's unique communication tool is procedure calls, with OO paradigm it becomes really difficult to implement modular communication where entities are completely independent and loosely coupled. We need a new way to let our components interact. In agreement with our vision, we understand that the best way to overcome this problem is to develop a new software infrastructure that will map more strictly to our model representation than Java paradigms. This infrastructure will be used not only in this project, but every future developed process that will have the same needs because, according to our visions, we want to develop reusable code.
Our platform must offer some functionalities. First of all we need a formal definition of a System:
a System is one or more active entities that interact to reach some goal or provide some functionalities. A system is composed by one or more Contexts, a Contexts identifies a logical location at deployment time. The System enables the communication between the entities and synchronizes them so Entities must interact using the System only. In this way a System can be concentrated or distributed seamlessly for the application designer. Defined this general perspective we can introduce two paradigms:

– Message Based Programming
– Event Based Programming

Message Based Programming To introduce message based programming we need to define the entities that can exchange messages. We will call these entities QActors in reference to Erlang's Actors.
QActors in our platform are not message-driven but message-based. The core difference between message driven and message based is that message driven systems can be only reactive to messages instead of being able to decide when messages and which messages are functional to evaluate. In this way we can model not only reactive entities but also proactive ones. QActors live in a context and they can communicate with other QActors using different communication primitives:

– dispatch - an asynchronous message without returning information;
– request - a message with returning information, can be executed as synchronous or asynchronous.

QActors can receive messages using the receive message primitives:

– receiveMessage - extracts the first message from the actor message queue;
– receiveTheMessage - extracts the first message that matches a pattern given by the application designer.

Event Based Programming We needed to define the concept of events as pervasive messages that are spread in the system. The message passing paradigm lacks of this feature because all messages are point to point. Events are a form of asynchronous communication that is not point to point. In our system our entities can declare to be interested in an event, when that event will be emitted the entities can react in some way.
When we introduced proactive QActors we realized that we needed the concept of Plan. A Plan is a sequence of actions (you can see actions as programming language instructions), every plan has its own logic and can switch to a different plan. When the execution of a plan reaches its end it can specify if the previous plan must continue its execution or suspend it. This abstraction was lacking of the better part of message/event driven programming: reactivity. This gap is filled by Asynchronous Actions. Asynchronous Actions are tipically time consuming actions that can be executed in blocking or non-blocking way. When executing an asynchronous action its execution can be interrupted by specified events and the QActor must react executing the associated plan. The logic of plan switching is defined as above. QActors are able to execute actions in two ways:

– executing compiled code
– interpreting meta code on the fly

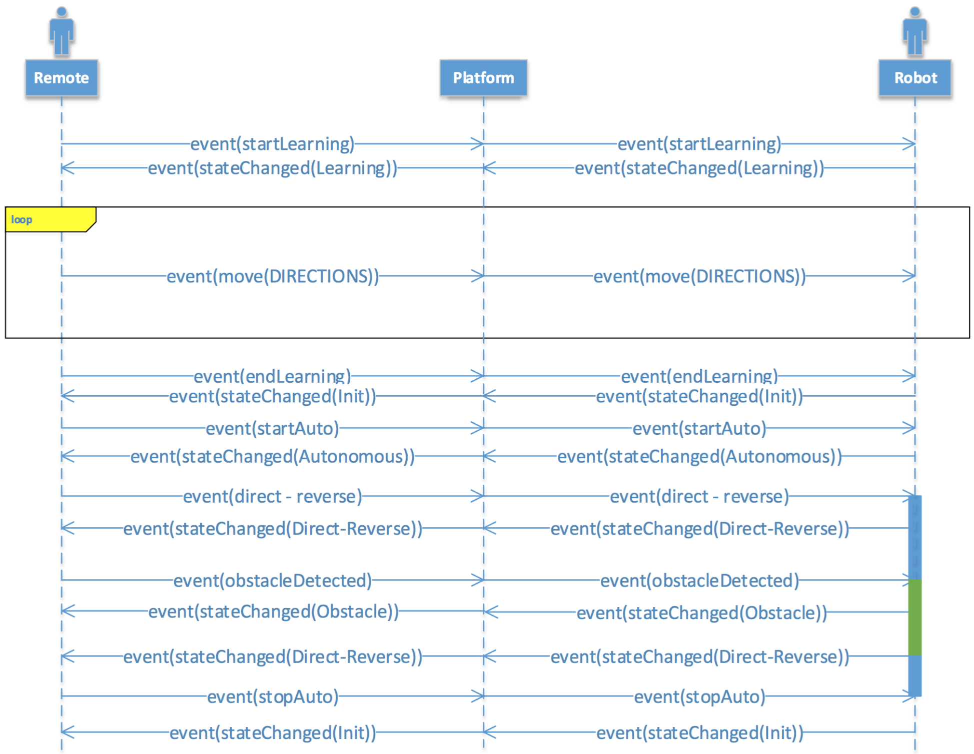
Adding the possibility to interpret code on the fly permits to the QActor to enrich its behaviour during runtime. The just described platform will be realized with Java Programming Language (our technologic assumption) but will be made to be interoperable with other technologies by the extensive use of text-based messages using the standard socket technology.
On top of the API offered by the platform we'll realize a dsl using the xtext framework. With the use of such tool we'll generate a declarative language that will provide two main benefits:

– a formal and human readable language to describe the problem analysis, entities interactions and system topology;
– code generation to be able to customize and execute the artifacts generated by the dsl interpretation.

With such tools in mind we can reimagine the traditionals problem analysis techniques and tackle the problems in a formal and more convenient way.

6.2 Logic architecture

L'architettura logica è esplicitata in figura.



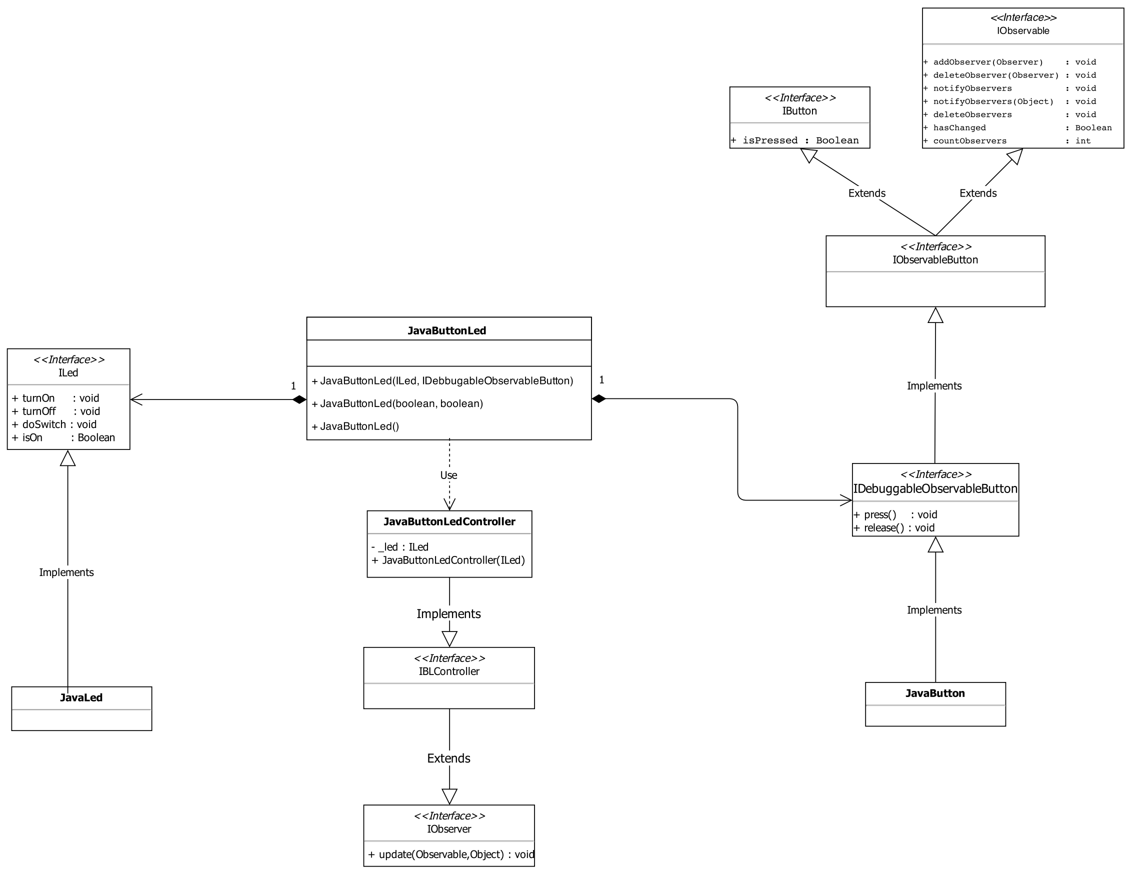
6.3 Risk analysis

Nell'interfaccia del Button, oltre al metodo accessor `isPressed` deve essere presente anche il metodo `addObserver`, ciò è risultato necessario dai colloqui con il committente (durante l'analisi dei requisiti).

7 Work plan

8 Project

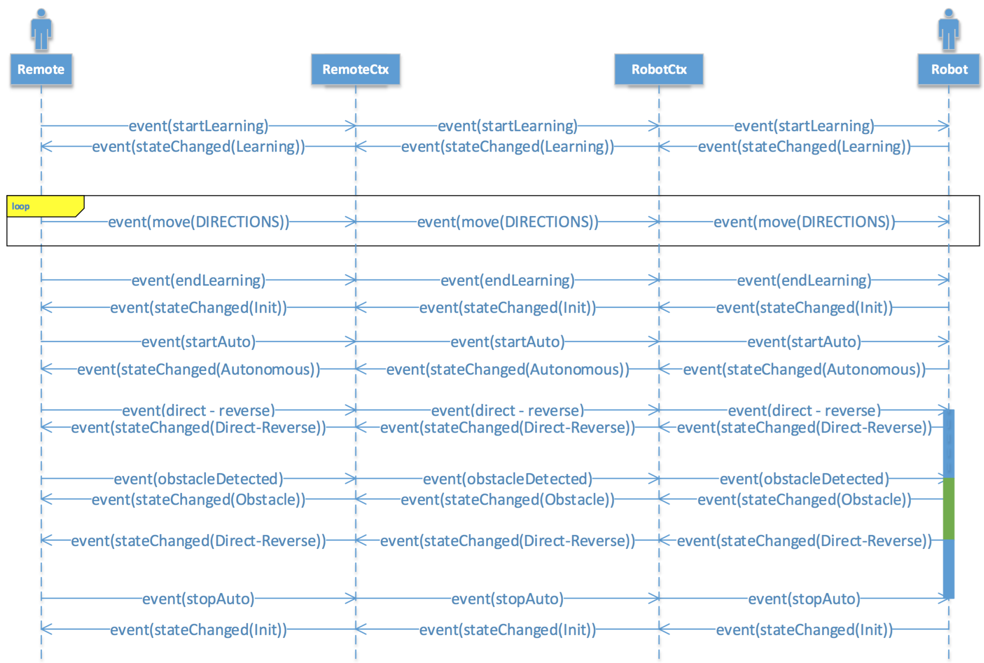
L'architettura progettuale è esplicitata in figura.
`https://github.com/iBelliDiISS/ButtonLedJava`



<<Interface>>
IObservable
+ addObserver(Observer) : void
+ deleteObserver(Observer) : void
+ notifyObservers : void
+ notifyObservers(Object) : void
+ deleteObservers : void
+ hasChanged : Boolean
+ countObservers : int
<<Interface>>
IButton
+ isPressed : Boolean
Extends
Extends
<<Interface>>
IObservableButton
JavaButtonLed
+ JavaButtonLed(ILed, IDebbugableObservableButton)
+ JavaButtonLed(boolean, boolean)
+ JavaButtonLed()
<<Interface>>
ILed
+ turnOn : void
+ turnOff : void
+ doSwitch : void
+ isOn : Boolean
1
1
Implements
Use
JavaButtonLedController
- _led : ILed
+ JavaButtonLedController(ILed)
<<Interface>>
IDebuggableObservableButton
+ press() : void
+ release() : void
Implements
Implements
Implements
JavaButton
<<Interface>>
IBLController
JavaLed
Extends
<<Interface>>
IObserver
+ update(Observable,Object) : void

8.1 Structure

8.2 Interaction



8.3 Behavior

9 Implementation

10 Testing

11 Deployment

12 Maintenance

13 Discussion

Requirement analysis discutere dell'evento nel caso sia generato dal nulla o da altro dire come rappresentarlo visto che non possiamo farlo in uml introduzione del sistema ad attori rispetto a java problema se in analisi dei requisiti parlare dell'indipendenza dalla tecnologia cambiamento dell'fsm del remote in fase successiva

A Codice

B Information about the author

Photo of the Author