

# Software Systems Engineering

## Case Study 2016

Paolo Sarti and Marcello Ballanti

Alma Mater Studiorum – University of Bologna  
viale Risorgimento 2, 40136 Bologna, Italy  
`paolo.sarti2studio.unibo.it`, `marcello.ballantistudio.unibo.it`

### 1 Introduction

The following report describes the software development process for an IoT application. The whole process is divided in two steps: at first, the client will communicate some requirements for the application, then new features will be requested. The report will show the impact of client requirements changes on the project on both the design and implementation phase.

### 2 Vision

We want to discuss the process of software development in order to overcome the limits of a technology-based approach in heterogeneous distributed system application design. We try to adopt a model-driven software development taking into account the AGILE methods for cooperation and work management. In particular, we want to:

- define a formal, executable model of the application to receive feedback from the client and ensure that requirements are clearly defined as soon as possible
- minimize the abstraction gap between the development tools and the application domain entities
- delay any technological hypothesis as much as possible in order to improve application reusability
- create flexible applications to resist requirements changes and add new features easily

### 3 Goals

The goal is to solve the given problem following the principles described in the vision and determine if this approach is viable and convenient. We want to build a prototype quickly and test if we can add new features to the application with minimum effort.

## 4 Requirements

We have to solve the following problem:

The Security Department of an Airport intends to exploit a differential drive robot equipped with a sonar (and some other device) to inspect -in a safe way- unattended bags when they are found in some sensible area of the Airport.

The software working on the inspector-roobot should support the following behavior:

- an operator drives the robot from an initial point (robot base area, RBA) towards the bag. To drive the robot the operator makes use of a remote robot control interface running on a smart device or a PC. The robot must accept commands from a single source only;
- as soon as the robot sonar perceives the bag within a prefixed distance (e.g.  $d=20\text{cm}$ ):
  1. the robot automatically stops
  2. the robot starts blinking a led
  3. the robot starts a first detection phase ( e.g. it moves around and performs some action according to its equipment - for example it could take some photo of the bag)
  4. the robot sends the results of its detection phase to the Airport Security Center;
- at the end of its work, the robot turns the led off and automatically returns to its RBA. During this phase the Airport Security Center could emit an 'alarm' signal; in this case the robot must restart to blink.

### STEP 1

Design and build a working prototype of this inspector-robot.

#### Non functional requirements at step1

The goal is to build a software system able to evolve from an initial proptotype (defined as the result of a problem analysis phase) to a final, testable product, by 'mixing' in a proper (pragmatically useful) way agile (SCRUM) software development with modelling.

### STEP 2 (Implementation Optional)

Extend the last requirement as follows:

- If the bag is qualifed as "harmful", the Airport Security Center emits an 'alarm' signal and activates another (properly equipped) robot that (start-ing from the same RBA of the robot inspector) should reach the bag in autonomous way and remove the bag from the area.

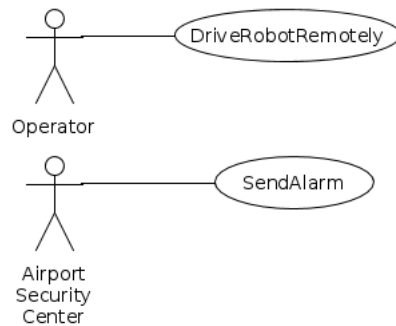
## 5 Requirement analysis

### 5.1 Use cases

The use cases describe how actors (UML actors i.e. the role played by a user or external system) interact with the system. In the requirements we can identify two external entities:

- **The operator** that drives the robot remotely from the initial point to the bag.
- **The Airport Security Center** that receives the results of the robot's detection phase and then it may emit an 'alarm' signal.

These interactions are shown by the UML below:



## 5.2 Scenarios

|                       |  |
|-----------------------|--|
| <b>Title</b>          | DriveRobotRemotely   |
| <b>Description</b>    | The operator drives the robot to the suspicious bag  |
| <b>Relationships</b>  |  |
| <b>Actors</b>         | Operator   |
| <b>Preconditions</b>  | The robot is in the RBA, waiting for commands from the operator.   |
| <b>Postconditions</b> | The robot starts the detection phase.  |
| <b>Main scenario</b>  | The operator uses the remote console to drive the robot.<br>When the robot perceives the bag, it starts the detection phase. |
| <b>Title</b>          | SendAlarm  |
| <b>Description</b>    | The Airport Security Center sends an alarm signal to the robot if needed   |
| <b>Relationships</b>  |  |
| <b>Actors</b>         | Airport Security Center  |
| <b>Preconditions</b>  | The Airport Security Center received the detection results   |
| <b>Postconditions</b> | The robot blinks its led until it comes back to the RBA.   |
| <b>Main scenario</b>  | The Airport Security Center uses its interface to send the alarm to the robot.<br>The robot blinks its led.                  |

## 5.3 (Domain) model

In this phase we try to find an agreement with the client on what the entities mentioned in the requirements are and what they have to do.

The system is composed by three parts:

- **Operator's remote console**
- **Airport Security Center's remote console**

#### – Differential drive robot

A **console** is a physical or virtual device that allows communication between the system and an external entity. It can get user input data and send them to the system, show some system output data to the user or both. In this case, the operator's console can get input from the operator and the Airport Security Center's console can receive the detection results and emit an alarm signal.

A **differential drive robot** is a composed entity that is able to use some devices to perform actions and receive data from the environment. It can also communicate with other parts of the system. All differential drive robots must have a sonar and are able to move in the environment. In this case, the differential drive robot has DC motors and wheels to move, a sonar and a led. DC motors, wheels, led and sonar are the hardware components mounted on the robot.

A **DC motor** can spin the attached wheel clockwise or counter-clockwise.

A **led** can be turned on or off.

A **sonar** can send an ultrasonic signal (trigger) and generates a corresponding response waveform (echo). The waveform analysis allows to estimate the distance from an obstacle.

The system can be formally defined with a custom language / executable meta-model developed by our software house. It allows us to describe what are the parts of the system, how they interact with each other and their behaviour.

The following is a first description of the system obtained by the requirement analysis:

```
1 RobotSystem testCase2016Analysis
2
3 Dispatch drive : drive(X)
4 Dispatch detectionResults : detectionResults(X)
5 Event alarm : alarm
6 Event obstacle : obstacle(X)
7
8 Context ctxDriveRobot ip[host="localhost" port=8010]
9 Context ctxOperator ip[host="localhost" port=8015]
10 Context ctxASC ip[host="localhost" port=8020]
11
12 QActor operator context ctxOperator
13 {
14   Plan init normal
15     println("Operator starts");
16     switchToPlan sendCommands
17
18   Plan sendCommands
19     println("Waiting for commands");
20     delay time(3000);
21     println("Sending command");
```

```

22     forward driverobot -m drive : drive("driveCmdPayload"
23         );
24     println("Command sent");
25     delay time(2000);
26     repeatPlan 0
27 }
28 QActor asc context ctxASC
29 {
30     Plan init normal
31         println("ASC starts");
32         switchToPlan waitForResults
33
34     Plan waitForResults
35         receiveMsg time(600000);
36         onMsg detectionResults : detectionResults(X) =>
37             println(detectionResults(X));
38         onMsg detectionResults : detectionResults(X) =>
39             switchToPlan riskDecision
40
41     Plan riskDecision
42         println("Evaluating risks");
43         delay time(3000);
44         //It could emit the alarm signal
45         emit alarm : alarm
46 }
47
48 Robot mock QActor driverobot context ctxDriveRobot
49 {
50     Plan init normal
51         println("driverobot starts");
52         switchToPlan drive
53
54     Plan drive
55         //We'll have to make sure that the robot executes the
56         //commands from the first console only
57         receiveMsg time(600000) react event obstacle =>
58             detect;
59         onMsg drive : drive(X) => println(savingmove(X));
60         onMsg drive : drive(X) => println(driving(X));
61         repeatPlan 0
62
63     Plan detect
64         println("Stopping...");
65         delay time(1000);

```

```

64     println("Start blinking the led");
65     println("Starting detection Phase...");
66     delay time(3000);
67     println("Sending results");
68     forward asc -m detectionResults : detectionResults("
        results");
69     println("Detection Results Sent");
70     println("Stop blinking the led");
71     println("Back to base");
72     switchToPlan backToBase
73
74 Plan backToBase
75     delay time(20000) react event alarm => alarmReaction;
76     switchToPlan finish
77
78 Plan alarmReaction resumeLastPlan
79     println("Alarm!");
80     println("Start blinking the led")
81
82 Plan finish
83     println("DriveRobot ends")
84 }

```

The operator can only send commands to drive the robot. The Airport Security Center waits for the detection results and can emit the alarm only after the results have been sent.

## 5.4 Test plan

We can do a test plan even before starting to implement the application, as a means to specify the expected behaviour of the system in a precise way. We just need to check if the parts of the system behave and interact with each other as defined in the requirements. We can't express tests formally though, because we already described the entities as actors, so object oriented tests (e.g. JUnit tests) are inadequate. Furthermore, some tests check the interaction of the physical system with the environment and this can only be done observing the actual behaviour of the system. Thus, we'll describe these tests in natural language.

In the initial phase, the operator drives the robot. We have to check the following:

- the operator can send commands to the robot
- the robot executes the commands it receives
- the robot perceives the presence of an obstacle

In the detection phase, the robot inspects the bag. We'll test the following:

- the robot stops and ignores commands from the operator
- the robot starts blinking after it stopped

- the robot can send the results to the Airport Security Center
- the Airport Security Center can receive the results of the inspection
- the robot stops blinking at the end of this phase

In the final phase, the robot comes back to the RBA. These are the tests we'll do:

- the robot actually comes back autonomously
- the Airport Security Center can emit the alarm signal
- the robot blinks the led if it perceives the alarm

At this stage in the development process, we can't define more specific functional or integration tests, we'll add them as needed during the implementation phase. We still haven't decided what technology we will use to implement the application, so we can't write executable tests yet. However, at the end of the analysis phase, we'll have an executable logical architecture of the application and we'll be able to perform some of the tests on it.

## 6 Problem analysis

### 6.1 Logic architecture

Logic architecture can be expressed in 3 dimensions:

1. **Structure:** what parts the system is made of.
2. **Interaction:** how the parts of the system communicate with each other.
3. **Behaviour:** what the parts of the system do.

We can formally express these concepts with the DDR custom language:

```

1 RobotSystem testCase2016LogicArchitecture
2
3 Event local_inputDrive : local_inputDrive(X) //events
   from GUI/External Input
4 Dispatch drive : drive(X)
5 Dispatch detectionResults : detectionResults(X)
6 Event alarm : alarm
7 Event local_alarm : local_alarm //events from GUI
   /External Input
8 Event obstacle : obstacle(X)
9 Event bagFound : bagFound
10 Event endDetection : endDetection
11
12 Context ctxDriveRobot ip[host="192.168.1.69" port=8010]
13 Context ctxOperator ip[host="192.168.1.2" port=8015]
14 Context ctxASC ip[host="192.168.1.2" port=8020]
15
16 QActor led context ctxDriveRobot

```

```

17 {
18   Plan init normal
19     println("Led starts");
20     switchToPlan senseStartBlink
21
22   Plan senseStartBlink
23     println("Led Off");
24     sense time(60000) bagFound => startBlinking;
25     repeatPlan 0
26
27   Plan startBlinking
28     println("led On");
29     delay time(1000) react event endDetection =>
30       senseAlarm;
31     println("Led Off");
32     delay time(1000) react event endDetection =>
33       senseAlarm;
34     repeatPlan 0
35
36   Plan senseAlarm
37     println("Led Off");
38     sense time(60000) alarm=> startBlinking;
39     repeatPlan 0
40 }
41
42 QActor operator context ctxOperator -g cyan
43 {
44   Plan init normal
45     println("Operator starts");
46     switchToPlan senseInput
47
48   Plan senseInput
49     sense time(60000) local_inputDrive =>
50       sendDriveCommands;
51     repeatPlan 0
52
53   Plan sendDriveCommands
54     onEvent local_inputDrive : local_inputDrive(X) =>
55       forward driverobot -m drive : drive(X)
56 }
57
58 QActor asc context ctxASC -g green
59 {
60   Plan init normal
61     println("ASC starts");

```



```

58     switchToPlan work
59
60 Plan work
61     receiveMsg time(600000);
62     onMsg detectionResults : detectionResults(X) =>
        println(detectionResults(X));
63     switchToPlan senseAlarm
64
65 Plan senseAlarm
66     sense time(100000) local_alarm => continue;
67     onEvent local_alarm : local_alarm => emit alarm :
        alarm
68 }
69
70 Robot mock QActor driverobot context ctxDriveRobot
71 {
72     Plan init normal
73         println("driverobot starts");
74         solve consult("talkTheory.pl") time(0) onFailSwitchTo
            prologFailure;
75         switchToPlan drive
76
77 Plan drive
78     //We'll have to make sure that the robot executes the
        commands from the first console only
79     receiveMsg time(600000) react event obstacle =>
        detect;
80     onMsg drive : drive(X) => println(savingmove(X));
81     onMsg drive : drive(X) => solve X time(0);
82     repeatPlan 0
83
84 Plan detect
85     println("Stopping...");
86     robotStop speed(100) time(1000);
87     emit bagFound : bagFound;
88     println("Starting detection Phase...");
89     [?? detection(X) ] forward asc -m detectionResults :
        detectionResults(X);
90     println("Detection Results Sent");
91     emit endDetection : endDetection;
92     println("Back to base");
93     switchToPlan backToBase
94
95 Plan backToBase

```

```

96 |     solve backToBase time(0); //It doesn't need to react,
    |         as the qactor led handles that
97 |     switchToPlan finish
98 |
99 | Plan finish
100 |     println("DriveRobot ends")
101 |
102 | Plan prologFailure resumeLastPlan
103 |     println("Failed to load talkTheory")
104 | }

```

This describes the whole logic architecture of our application. It can also be executed so that the client can confirm that the analysis defined a system that behaves as required.

This architecture derives from the one obtained in the domain model and introduces new interactions and a new entity.

The **DriveRobot** receives commands from the Operator Interface in the first phase, executes its automatic operations during the detection phase, it sends results to the ASCConsole and comes back to the RBA in the end. It has to react to obstacles to begin the detection phase.

The **Operator Interface** receives commands from the operator as events and sends the corresponding commands to the robot.

The **ASCConsole** receives the detection results from the detection phase and then enables the Airport Security Center to emit the alarm.

We decided to introduce the **led** as an active entity separated from the robot because it is an active entity that has to interact with other entities and has its own behaviour, modeling it as a passive object managed by the robot is inappropriate. The led starts to blink when the detection phase begins, stops to blink when the detection phase ends and it starts to blink if the alarm is emitted when the robot is coming back.

The interaction with external entities (the operator and the ASC) have been modeled as local events.

## 6.2 Abstraction gap

The abstraction gap is the distance between the concepts used to model the problem and those implied by the technology of choice. Thanks to the framework provided, executable code is generated from the model defined in the ddr meta-model. Thus, adopting this framework allows the application designers to use an extremely high-level description of the problem, closer to the application domain, reducing considerably the abstraction gap. The specific technology to be used can be decided later, in a configuration phase. The advantage of using a meta-model and a code generator is also that it can be extended to support more advanced concepts.

### 6.3 Risk analysis

Using the framework code generators, we can write most of the code independently from the specific implementation technology. Although the qa/ddr meta-model is technology independent, the code generated automatically may require some kind of environment on the computational nodes where it will be deployed (e.g. the JVM, the .NET runtime environment, a specific operating system etc).

## 7 Work plan

After the analysis phase, we decided to develop the application using the ddr framework, so that we don't start from scratch. We can reuse the executable logic architecture and enhance it. The framework already offers the implementation logic for some parts of the system and it offers high level abstractions that allow the developers to focus on business logic and not to worry too much about boilerplate code.

We'll use the following features offered by the framework:

- A communication system that allows the parts of the system to send and receive messages and events
- Reactive actions
- Timed actions
- The robot configuration
- DC motors driver
- sonar driver (and management of its data)

We need to implement these features:

- the interfaces that generate the external entities interaction events
- the led blinking logic
- the detection phase logic
- an algorithm that allows the robot to come back to the RBA

We also need to decide which platforms will be used by the operator and the Airport Security Center.

## 8 Project

### 8.1 Structure

The structure is essentially the same as the logic architecture.

### 8.2 Interaction

There are no significant changes from the logic architecture.

### 8.3 Behavior

More details have been added to implement the missing features described in the work plan.

The consoles used by the ASC and the operator will be GUIs that allow them to interact with the system. The robot behaviour has slightly changed in the first phase: to ensure it receives messages from a single source, it memorizes the sender of the first received drive message and accepts new drive commands from that source only. The detection logic will be simulated, because the robot we built can't collect meaningful information about the bag.

```
1 RobotSystem testCase2016Project
2
3 Event local_inputDrive : local_inputDrive(X) //events
   from GUI/External Input
4 Dispatch drive : drive(X)
5 Dispatch detectionResults : detectionResults(X)
6 Event alarm : alarm
7 Event local_alarm : local_alarm //events from GUI
   /External Input
8 Event obstacle : obstacle(X)
9 Event bagFound : bagFound
10 Event endDetection : endDetection
11
12 Context ctxDriveRobot ip[host="192.168.1.69" port=8010]
13 Context ctxOperator ip[host="192.168.1.2" port=8015]
14 Context ctxASC ip[host="192.168.1.2" port=8020]
15
16 QActor led context ctxDriveRobot
17 {
18   Plan init normal
19     println("Led starts");
20     solve consult("ledTheory.pl") time(0);
21     switchToPlan senseStartBlink
22
23   Plan senseStartBlink
24     println("Led Off");
25     solve turnTheLed(off) time(0);
26     sense time(60000) bagFound -> startBlinking;
27     repeatPlan 0
28
29   Plan startBlinking
30     println("led On");
31     solve turnTheLed(on) time(0);
32     delay time(1000) react event endDetection ->
       senseAlarm;
33     println("Led Off");
```

```

34     solve turnTheLed(off) time(0);
35     delay time(1000) react event endDetection ->
        senseAlarm;
36     repeatPlan 0
37
38     Plan senseAlarm
39         println("Led Off");
40         solve turnTheLed(off) time(0);
41         sense time(60000) alarm-> startBlinking;
42         repeatPlan 0
43     }
44
45     QActor operator context ctxOperator -g cyan
46     {
47         Plan init normal
48             println("Operator starts");
49             switchToPlan senseInput
50
51         Plan senseInput
52             sense time(60000) local_inputDrive ->
                sendDriveCommands;
53             repeatPlan 0
54
55         Plan sendDriveCommands resumeLastPlan
56             onEvent local_inputDrive : local_inputDrive(X) ->
                forward driverobot -m drive : drive(X)
57     }
58
59     QActor asc context ctxASC -g green
60     {
61         Plan init normal
62             println("ASC starts");
63             switchToPlan work
64
65         Plan work
66             receiveMsg time(600000);
67             onMsg detectionResults : detectionResults(X) ->
                println(detectionResults(X));
68             switchToPlan senseAlarm
69
70         Plan senseAlarm
71             sense time(100000) local_alarm -> continue;
72             onEvent local_alarm : local_alarm -> emit alarm :
                alarm
73     }

```

```

74
75 Robot mock QActor driverobot context ctxDriveRobot
76 {
77     Plan init normal
78         println("driverobot starts");
79         solve consult("talkTheory.pl") time(0) onFailSwitchTo
            prologFailure;
80         solve consult("driveRobotTheory.pl") time(0)
            onFailSwitchTo prologFailure;
81         switchToPlan receiveFirstCommand
82
83     Plan receiveFirstCommand
84         receiveMsg time(600000) react event obstacle ->
            detect;
85         memoCurrentMessage;
86         //Save first sender
87         [!? msg(drive,dispatch, S, R, drive(X), N)] solve
            assert(firstSender(S)) time(0);
88         onMsg drive : drive(X) -> solve savemove(X) time(0);
89         onMsg drive : drive(X) -> solve X time(0);
90         onMsg drive : drive(X) -> switchToPlan drive;
91         repeatPlan 0
92
93     Plan drive
94         receiveMsg time(600000) react event obstacle ->
            detect;
95         memoCurrentMessage;
96         //To make sure that the sender is the same as the
            first one
97         [!? msg(drive,dispatch, S, R, drive(X), N)] solve
            firstSender(S) time(0) onFailSwitchTo drive;
98         onMsg drive : drive(X) -> solve savemove(X) time(0);
99         onMsg drive : drive(X) -> solve X time(0);
100        repeatPlan 0
101
102    Plan detect
103        println("Stopping...");
104        solve endSavemoves time(0);
105        robotStop speed(100) time(1000);
106        emit bagFound : bagFound;
107        println("Starting detection Phase...");
108        [?? detection(X) ] forward asc -m detectionResults :
            detectionResults(X);
109        println("Detection Results Sent");
110        emit endDetection : endDetection;

```

```

111     println("Back to base");
112     switchToPlan backToBase
113
114 Plan backToBase
115     solve backToBase time(0); //It doesn't need to react,
        as the qactor led handles that
116     switchToPlan finish
117
118 Plan finish
119     println("DriveRobot ends")
120
121 Plan prologFailure resumeLastPlan
122     println("Failed to load prolog theories")
123 }

```

## 9 Implementation

The led blinking logic is implemented directly as QActor behaviour. The Prolog theory turnTheLed(X) allows the QActor to manage the led and actually turn it on and off. The operator console consists of a GUI that translates external input events in messages to drive the robot. The ASC console is a GUI that shows the image received from the robot at the end of the detection phase and then shows a button that emits the alarm if pressed. The robot uses a Prolog theory to execute the detection phase. It takes a picture of the bag (in our implementation, it simulates this behaviour) and sends it to the ASC. Some Java library has to be called to convert the image as a string and vice versa.

Another Prolog theory is used to implement a simple algorithm to come back autonomously: the robot memorizes every move it makes in the first phase, so it can come back executing the same moves backwards.

```

1 %drivecommand example
2 %executeInput(move(mf,100,1000,0))
3
4 %lastmove is the next move to save,
5 %I can get the starting time, but I can't insert it into
    the moveList until it ends.
6
7 %The savemove/1 rule uses the knowledge base to store and
    update the information,
8 %but it uses savemove/5 to get the updated lastMove and
    moveList.
9 savemove(executeInput(CUR)):-
10     moveList(L),
11     savemove(CUR, lastMove(LASTMOVE,MVTIME),L,NEWLASTMOVE,
        NEWL),

```

```

12     retract (lastMove (LASTMOVE,MVTIME)) ,
13     assert (NEWLASTMOVE) ,
14     retract (moveList (L)) ,
15     assert (moveList (NEWL)) .
16
17 %Here the savemove rule is implemented without assert and
    retract .
18 %savemove (CUR, LAST, LIST, NEWLAST, NEWLIST)
19 savemove (CUR, lastMove (none, 0) , [] , lastMove (CUR, M) , [] ) :-
20     getCurrentMillis (M) .
21
22 savemove (CUR, lastMove (move (MV, SPEED, 0) , FIRSTM) , L, lastMove
    (CUR, M) , [move (MV, SPEED, DIFF, 0) | L] ) :-
23     getCurrentMillis (M) ,
24     DIFF is M - FIRSTM.
25
26 endSavemoves:-
27     savemove (executeInput (move (h, 100, 1000))) .
28
29 backToBase:-
30     moveList (L) ,
31     backToBase (L) .
32
33 backToBase ([]) :- !.
34
35 backToBase ([H, T]) :-
36     revMove (H, RH) ,
37     executeInput (RH) ,
38     backToBase (T) .
39
40 getCurrentMillis (M) :-
41     class ("it.unibo.sartiballanti.utils.Utils") <-
        getCurrentTimeMillis returns M.
42
43 revMove (move (mf, X, Y, Z) , move (mb, X, Y, Z)) .
44 revMove (move (mb, X, Y, Z) , move (mf, X, Y, Z)) .
45 revMove (move (mr, X, Y, Z) , move (ml, X, Y, Z)) .
46 revMove (move (ml, X, Y, Z) , move (mr, X, Y, Z)) .
47 revMove (move (h, X, Y, Z) , move (h, X, Y, Z)) .
48
49 initDriveRobotTheory :-
50     actorobj (Actor) ,
51     assert (moveList ([]) ) ,
52     assert (lastMove (none, 0)) ,
53     ( Actor <- isSimpleActor returns R, R=true , ! ,

```



```
54     actorPrintln(" *** driveRobotTheory Loaded *** ");
55   ).
56
57 :- initialization (initDriveRobotTheory).
```

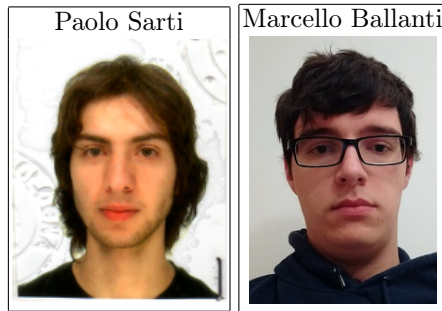
## 10 Testing

## 11 Deployment

## 12 Maintenance

See [1] until page 11 (CMM) and pages 96-105.

### 13 Information about the author



### References

1. A. Natali and A. Molesini. *Costruire sistemi software: dai modelli al codice*. Esculapio, 2009.