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 hypotesis 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=20cm):
  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 equiment - 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 requriments 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 (starting from the same RBA of the robot inspector) should reach the bag in autonomous way and remove the bag from the area.

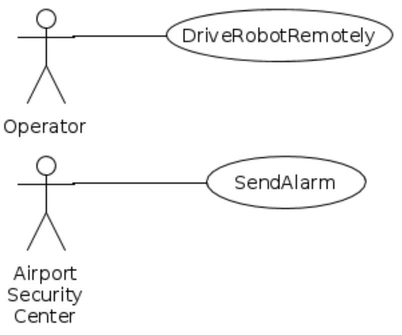
5 Requirement analysis

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

Title	DriveRobotRemotely
Description	The operator drives the robot to the suspicious bag
Relationships	
Actors	Operator
Preconditions	The robot is in the RBA, waiting for commands from the operator.
Postconditions	The robot starts the detection phase.
Main scenario	The operator uses the remote console to drive the robot. When the robot perceives the bag, it starts the detection phase.

Title	SendAlarm
Description	The Airport Security Center sends an alarm signal to the robot if needed
Relationships	
Actors	Airport Security Center
Preconditions	The Airport Security Center received the detection results
Postconditions	The robot blinks its led until it comes back to the RBA.
Main scenario	The Airport Security Center uses its interface to send the alarm to the robot. 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:

```
RobotSystem testCase2016Analysis

Dispatch drive : drive(X)
Dispatch detectionResults : detectionResults(X)
Event alarm : alarm
Event obstacle : obstacle(X)

Context ctxDriveRobot ip[host="localhost" port=8010]
Context ctxOperator ip[host="localhost" port=8015]
Context ctxASC ip[host="localhost" port=8020]

QActor operator context ctxOperator
{
  Plan init normal
    println("Operator starts");
    switchToPlan sendCommands

  Plan sendCommands
    println("Waiting for commands");
    delay time(3000);
    println("Sending command");
    forward driverobot -m drive : drive("driveCmdPayLoad");
    println("Command sent");
    delay time(2000);
    repeatPlan 0
}

QActor asc context ctxASC
{
  Plan init normal
    println("ASC starts");
    switchToPlan waitForResults

  Plan waitForResults
    receiveMsg time(600000);
    onMsg detectionResults : detectionResults(X) ->
    println(detectionResults(X));
    onMsg detectionResults : detectionResults(X) ->
    switchToPlan riskDecision

  Plan riskDecision
    println("Evaluating risks");
    delay time(3000);
    //It could emit the alarm signal
    emit alarm : alarm
}

Robot mock QActor driverobot context ctxDriveRobot
{
  Plan init normal
    println("driverobot starts");
    switchToPlan drive

  Plan drive
    //We'll have to make sure that the robot executes the commands from the first console only
    receiveMsg time(600000) react event obstacle -> detect;
    onMsg drive : drive(X) -> println(savingmove(X));
    onMsg drive : drive(X) -> println(driving(X));
    repeatPlan 0

  Plan detect
    println("Stopping...");
    delay time(1000);
    println("Start blinking the led");
    println("Starting detection Phase...");
    delay time(3000);
    println("Sending results");
    forward asc -m detectionResults : detectionResults("
       results");
    println("Detection Results Sent");
    println("Stop blinking the led");
    println("Back to base");
    switchToPlan backToBase

  Plan backToBase
    delay time(20000) react event alarm -> alarmReaction;
    switchToPlan finish

  Plan alarmReaction resumeLastPlan
    println("Alarm!");
    println("Start blinking the led")

  Plan finish
    println("DriveRobot ends")
}
```

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 tough, 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 physiscal 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:

```
RobotSystem testCase2016LogicArchitecture

Event local_inputDrive : local_inputDrive(X)   //events from GUI/External Input
Dispatch drive : drive(X)
Dispatch detectionResults : detectionResults(X)
Event alarm : alarm
Event local_alarm : local_alarm          //events from GUI/External Input
Event obstacle : obstacle(X)
Event bagFound : bagFound
Event endDetection : endDetection

Context ctxDriveRobot ip[host="192.168.1.69" port=8010]
Context ctxOperator ip[host="192.168.1.2" port=8015]
Context ctxASC ip[host="192.168.1.2" port=8020]

QActor led context ctxDriveRobot
```

```
{
  Plan init normal
    println("Led starts");
    switchToPlan senseStartBlink

  Plan senseStartBlink
    println("Led Off");
    sense time(60000) bagFound -> startBlinking;
    repeatPlan 0

  Plan startBlinking
    println("led On");
    delay time(1000) react event endDetection ->
       senseAlarm;
    println("Led Off");
    delay time(1000) react event endDetection ->
       senseAlarm;
    repeatPlan 0

  Plan senseAlarm
    println("Led Off");
    sense time(60000) alarm-> startBlinking;
    repeatPlan 0
}

QActor operator context ctxOperator -g cyan
{
  Plan init normal
    println("Operator starts");
    switchToPlan senseInput

  Plan senseInput
    sense time(60000) local_inputDrive ->
       sendDriveCommands;
    repeatPlan 0

  Plan sendDriveCommands
    onEvent local_inputDrive : local_inputDrive(X) ->
       forward driverobot -m drive : drive(X)
}

QActor asc context ctxASC -g green
{
  Plan init normal
    println("ASC starts");
    switchToPlan work

  Plan work
    receiveMsg time(600000);
    onMsg detectionResults : detectionResults(X) ->
        println(detectionResults(X));
    switchToPlan senseAlarm

  Plan senseAlarm
    sense time(100000) local_alarm -> continue;
    onEvent local_alarm : local_alarm -> emit alarm :
        alarm
}

Robot mock QActor driverobot context ctxDriveRobot
{
  Plan init normal
    println("driverobot starts");
    solve consult("talkTheory.pl") time(0) onFailSwitchTo
         prologFailure;
    switchToPlan drive

  Plan drive
    //We'll have to make sure that the robot executes the
         commands from the first console only
    receiveMsg time(600000) react event obstacle ->
        detect;
    onMsg drive : drive(X) -> println(savingmove(X));
    onMsg drive : drive(X) -> solve X time(0);
    repeatPlan 0

  Plan detect
    println("Stopping...");
    robotStop speed(100) time(1000);
    emit bagFound : bagFound;
    println("Starting detection Phase...");
    [?? detection(X) ] forward asc -m detectionResults :
        detectionResults(X);
    println("Detection Results Sent");
    emit endDetection : endDetection;
    println("Back to base");
    switchToPlan backToBase

  Plan backToBase
```

```
    solve backToBase time(0); //It doesn't need to react,
        as the qactor led handles that
    switchToPlan finish

  Plan finish
    println("DriveRobot ends")

  Plan prologFailure resumeLastPlan
    println("Failed to load talkTheory")
}
```

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 intecract 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 metamodel 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.

```
RobotSystem testCase2016Project

Event local_inputDrive : local_inputDrive(X)  //events
    from GUI/External Input
Dispatch drive : drive(X)
Dispatch detectionResults : detectionResults(X)
Event alarm : alarm
Event local_alarm : local_alarm          //events from GUI
    /External Input
Event obstacle : obstacle(X)
Event bagFound : bagFound
Event endDetection : endDetection

Context ctxDriveRobot ip[host="localhost" port=8010]
Context ctxOperator ip[host="localhost" port=8015]
Context ctxASC ip[host="localhost" port=8020]

QActor led context ctxDriveRobot
{
  Plan init normal
    println("Led starts");
    solve consult("ledTheory.pl") time(0);
    switchToPlan senseStartBlink

  Plan senseStartBlink
    println("Led Off");
    solve turnTheLed(off) time(0);
    sense time(60000) bagFound -> startBlinking;
    repeatPlan 0

  Plan startBlinking
    println("led On");
    solve turnTheLed(on) time(0);
    delay time(1000) react event endDetection ->
        senseAlarm;
    println("Led Off");
    solve turnTheLed(off) time(0);
    delay time(1000) react event endDetection ->
       senseAlarm;
    repeatPlan 0

  Plan senseAlarm
    println("Led Off");
    solve turnTheLed(off) time(0);
    sense time(60000) alarm-> startBlinking;
    repeatPlan 0
}

QActor operator context ctxOperator -g cyan
{
  Plan init normal
    println("Operator starts");
    switchToPlan senseInput

  Plan senseInput
    sense time(60000) local_inputDrive ->
       sendDriveCommands;
    repeatPlan 0

  Plan sendDriveCommands resumeLastPlan
    onEvent local_inputDrive : local_inputDrive(X) ->
       forward driverobot -m drive : drive(X)
}

QActor asc context ctxASC -g green
{
  Plan init normal
    println("ASC starts");
    switchToPlan work

  Plan work
    receiveMsg time(600000);
    onMsg detectionResults : detectionResults(X) ->
       println(detectionResults(X));
    switchToPlan senseAlarm

  Plan senseAlarm
    sense time(100000) local_alarm -> continue;
    onEvent local_alarm : local_alarm -> emit alarm :
       alarm
}
```

```

Robot plexiBox QActor driverobot context ctxDriveRobot
{
  Plan init normal
    println("driverobot starts");
    solve consult("talkTheory.pl") time(0) onFailSwitchTo
         prologFailure;
    solve consult("driveRobotTheory.pl") time(0)
        onFailSwitchTo prologFailure;
    switchToPlan receiveFirstCommand

  Plan receiveFirstCommand
    receiveMsg time(600000) react event obstacle ->
        detect;
    //Save first sender
    [?? msg(drive,dispatch, S, R, drive(X), N)] solve
        assert(firstSender(S)) time(0);
    onMsg drive : drive(X) -> solve savemove(X) time(0)
        onFailSwitchTo prologFailure;
    onMsg drive : drive(X) -> solve X time(0)
        onFailSwitchTo prologFailure;
    onMsg drive : drive(X) -> switchToPlan drive;
    repeatPlan 0

  Plan drive
    receiveMsg time(600000) react event obstacle ->
        detect;
    //To make sure that the sender is the same as the
        first one
    [?? msg(drive,dispatch, S, R, drive(X), N)] solve
        firstSender(S) time(0) onFailSwitchTo drive;
    onMsg drive : drive(X) -> solve savemove(X) time(0)
        onFailSwitchTo prologFailure;
    onMsg drive : drive(X) -> solve X time(0)
        onFailSwitchTo prologFailure;
    repeatPlan 0

  Plan detect
    println("Stopping...");
    solve endSavemoves time(0);
    robotStop speed(100) time(1000);
    emit bagFound : bagFound;
    println("Starting detection Phase...");
    [?? detection(X) ] forward asc -m detectionResults :
        detectionResults(X);
    println("Detection Results Sent");
    emit endDetection : endDetection;
    println("Back to base");
    switchToPlan backToBase

  Plan backToBase
    solve backToBase time(0); //It doesn't need to react,
        as the qactor led handles that
    switchToPlan finish

  Plan finish
    println("DriveRobot ends")

  Plan prologFailure resumeLastPlan
    println("Failed to load prolog theories")
}
```

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.

```

createPi4jLed( PinNum)  :-
  actorobj( Actor ),
    Actor <- getOutputEnvView returns OutView ,
  class("it.unibo.devices.qa.DeviceLedPi4jQa") <-
      createLed( OutView, PinNum ) returns LED.


turnTheLed( on ):-
  class("it.unibo.devices.qa.DeviceLedPi4jQa") <-
      getTheLed  returns LED,
  LED <- turnOn .

turnTheLed( off ):-
  class("it.unibo.devices.qa.DeviceLedPi4jQa") <-
      getTheLed  returns LED,
  LED <- turnOff .

/*
----------------------------------------------------------------

initialize
```

```
----------------------------------------------------------------------------------

*/
initialize  :-   createPi4jLed(24).
:- initialization(initialize).
```

The **operator console** includes a GUI that translates external input events in messages to drive the robot.

```
/* Generated by AN DISI Unibo */
/*
This code is generated only ONCE
*/
package it.unibo.operator;
import java.awt.Button;
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.Label;
import java.awt.Panel;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import it.unibo.baseEnv.basicFrame.EnvFrame;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.qactors.ActorContext;

public class Operator extends AbstractOperator {

  protected Map<String, String> driveCmdMap;

  protected List<String> driveCmdsSorted;

  public final static String Forward="Forward";
  public final static String Backward="Backward";
  public final static String Right="Right";
  public final static String Left="left";
  public final static String Halt="Halt";

```

```java
  public Operator(String actorId, ActorContext myCtx, IOutputEnvView outEnvView )  throws Exception{
    super(actorId, myCtx, outEnvView);
  }

  protected void initCmdMap(){
    driveCmdMap=new HashMap<>();
    driveCmdMap.put(Forward, "executeInput(move(mf,100,0))");
    driveCmdMap.put(Backward, "executeInput(move(mb,100,0))");
    driveCmdMap.put(Right, "executeInput(move(mr,100,0))");
    driveCmdMap.put(Left, "executeInput(move(ml,100,0))");
    driveCmdMap.put(Halt, "executeInput(move(h,100,0))");
//    driveCmdsSorted=new ArrayList<>();
//    driveCmdsSorted.add(Forward);
//    driveCmdsSorted.add(Backward);
//    driveCmdsSorted.add(Left);
//    driveCmdsSorted.add(Right);
//    driveCmdsSorted.add(Halt);
  }

  @Override
  protected void addInputPanel(int size) {
  }

  @Override
  protected void addCmdPanels(){
    //super.addCmdPanels();
    initCmdMap();
    //((EnvFrame) env).removeAll();
    //((EnvFrame) env).setLayout(new FlowLayout());
    ((EnvFrame) env).setSize(800,700);
    Panel p = new Panel();
    GridLayout l =  new GridLayout();
    l.setVgap(10);
    l.setHgap(10);
    l.setColumns(3);
    l.setRows(3);
    p.setLayout(l);

    MouseListener ml =new MouseListener() {
      @Override
      public void mouseReleased(MouseEvent e) {
        execAction(Halt);
        System.out.println("DEBUG: UNPRESSED");
      }
      @Override
      public void mousePressed(MouseEvent e) {
        Button b = (Button)e.getSource();
        execAction(b.getLabel());
        System.out.println("DEBUG: PRESSED" + b.getLabel());
      }
      @Override
      public void mouseExited(MouseEvent e) {
        // TODO Auto-generated method stub
      }
      @Override
      public void mouseEntered(MouseEvent e) {
        // TODO Auto-generated method stub
      }
      @Override
      public void mouseClicked(MouseEvent e) {
        // TODO Auto-generated method stub
      }
    };

    Button forward = new Button(Forward);
    forward.addMouseListener(ml);
    Button backward = new Button(Backward);
    backward.addMouseListener(ml);
    Button right = new Button(Right);
    right.addMouseListener(ml);
    Button left = new Button(Left);
    left.addMouseListener(ml);
    Button halt = new Button(Halt);
    halt.addMouseListener(ml);
    p.add(new Label(""));
    p.add(forward);
    p.add(new Label(""));
    p.add(left);
    p.add(halt);
    p.add(right);
    p.add(new Label(""));
    p.add(backward);
    p.add(new Label(""));
    ((EnvFrame) env).add(p);
    ((EnvFrame) env).validate();
  }



  @Override
  public void execAction(String cmd) {
    super.execAction(cmd);

    if(driveCmdMap.containsKey(cmd)){
      String actualCmd = driveCmdMap.get(cmd);
      platform.raiseEvent("input", "local_inputDrive", "
          local_inputDrive("+actualCmd+")");
      return;
    }
  }

}
```

The **ASC console** includes 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.

```
/* Generated by AN DISI Unibo */
/*
This code is generated only ONCE
*/
package it.unibo.asc;
import java.awt.Button;
import java.awt.FlowLayout;
import java.awt.Frame;
import java.awt.Panel;

import it.unibo.baseEnv.basicFrame.EnvFrame;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.qactors.ActorContext;

public class Asc extends AbstractAsc {
  public Asc(String actorId, ActorContext myCtx,
      IOutputEnvView outEnvView )  throws Exception{
    super(actorId, myCtx, outEnvView);
  }


  @Override
  protected void addCmdPanels(){
    //super.addCmdPanels();
    //photo panel
    Panel results = new Panel();
    results.setSize(300, 400);
    ((Frame) env).setLayout(new FlowLayout());
    ((Frame) env).add(results);
    ((EnvFrame) env).addCmdPanel("Alarm", new String[]{"
       Alarm"}, this);
  }

  @Override
  public void execAction(String cmd) {
    super.execAction(cmd);
    if( cmd.equals("Alarm") ){
      platform.raiseEvent("input", "local_alarm", "
         local_alarm");
      return;
    }
  }
}
```

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.
This is the Prolog theory that implements the robot functionalities:

```
%drivecommand example
%executeInput(move(mf,100,1000,0))

%lastmove is the next move to save,
%I can get the starting time, but I can't insert it into
    the moveList until it ends.

%The savemove/1 rule uses the knowledge base to store and
     update the information,
%but it uses savemove/5 to get the updated lastMove and
    moveList.
savemove(executeInput(CUR)):-
  moveList(L),
  savemove(CUR, lastMove(LASTMOVE,MVTIME),L,NEWLASTMOVE,
      NEWL),
  retract(lastMove(LASTMOVE,MVTIME)),
  assert(NEWLASTMOVE),
  retract(moveList(L)),
  assert(moveList(NEWL)).

%Here the savemove rule is implemented without assert and
    retract.
%savemove(CUR,LAST,LIST,NEWLAST,NEWLIST)
savemove(CUR,lastMove(none,0),[],lastMove(CUR,M),[]):-
  getCurrentMillis(M).

savemove(CUR,lastMove(move(MV,SPEED,0),FIRSTM),L,lastMove
    (CUR,M),[move(MV,SPEED,DIFF,0)|L]):-
  getCurrentMillis(M),
  DIFF is M - FIRSTM.

endSavemoves:-
  savemove(executeInput(move(h,100,1000))).

backToBase:-
  moveList(L),
  backToBase(L).

backToBase([]):-!.

backToBase([H,T]):-
  revMove(H,RH),
  executeInput(RH),
  backToBase(T).

getCurrentMillis(M):-
  class("it.unibo.sartiballanti.utils.Utils") <-
      getCurrentTimeMillis returns M.

revMove(move(mf,X,Y,Z),move(mb,X,Y,Z)).
revMove(move(mb,X,Y,Z),move(mf,X,Y,Z)).
revMove(move(mr,X,Y,Z),move(ml,X,Y,Z)).
revMove(move(ml,X,Y,Z),move(mr,X,Y,Z)).
revMove(move(h,X,Y,Z),move(h,X,Y,Z)).

initDriveRobotTheory  :-
  actorobj(Actor),
  assert(moveList([])),
  assert(lastMove(none,0)),
  ( Actor <- isSimpleActor returns R, R=true, !,
    actorPrintln(" *** driveRobotTheory Loaded  ***  ");
  ).

:- initialization(initDriveRobotTheory).
```

10 Testing

11 Deployment

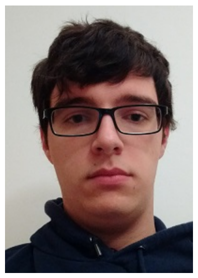
12 Maintenance

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

13 Information about the author

Paolo Sarti

Marcello Ballanti



References

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