

System Software Engineering: final task 2018

Adriano Cardace - Filippo Mearini - Gabriele Sorrentino

Alma Mater Studiorum – University of Bologna
viale Risorgimento 2, 40136 Bologna, Italy
`adriano.cardace@studio.unibo.it`
`filippo.mearini@studio.unibo.it`
`gabriele.sorrentino@studio.unibo.it`

1 Introduction

The aim of this project is to build a distributed system which makes possible for a user to control a robot cleaner, which is also able to detect and react to fixed and moving obstacles. Furthermore we want to focus on the quality of the product process development, and not only on the quality of the final product, extending and integrating already available components (or microservices) or, if needed, realizing new ones which are extensible and reusable.

2 Vision

Our main goal is to offer to the client an executable prototype as soon as possible.

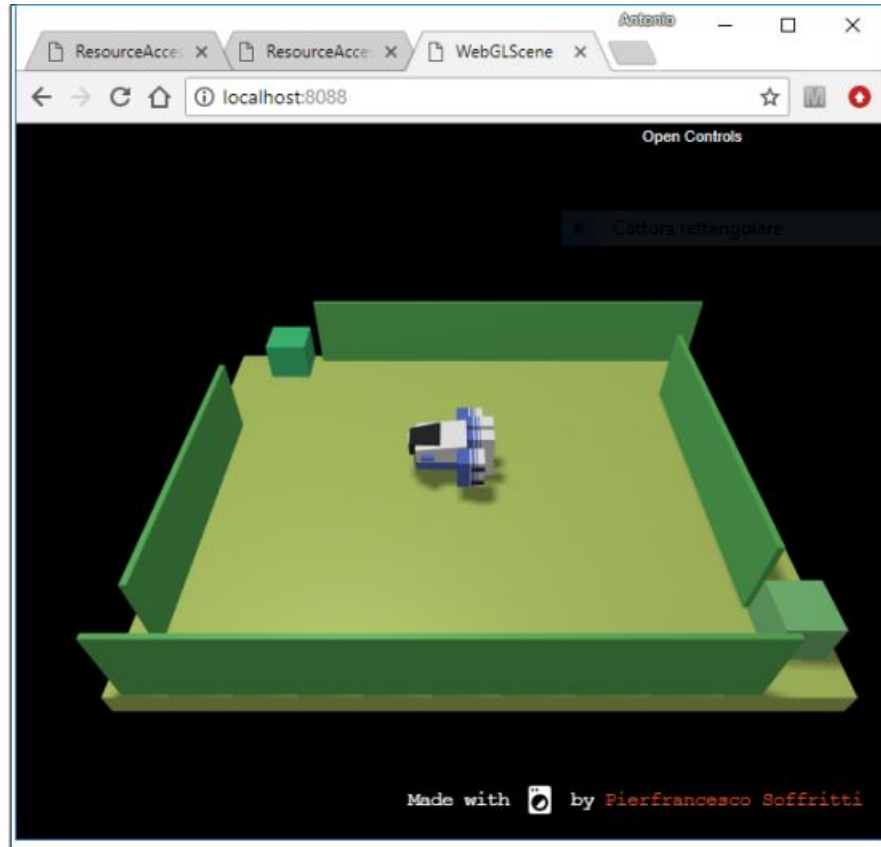
To achieve this, the choice of the language will be delayed as much as possible, using an iterative top-down technique; this means that starting from an higher abstraction level, we provide a formal and executable model in order to receive a constant feedback from the client. In this contest a model is *a set of concepts and properties aimed at capturing essential aspect of a system, in a precise context*.

We can summarize the way we will face the project with the following statement: *from technologies to the project analysis and from project to the technologies again*. In other words we are admitting to be influenced by the technologies that surround us, consequently we will start right from them to evaluate the project, but this will have to be done as a source of inspiration and not as a limitation.

However a careful requirement analysis could highlight a high *abstraction gap* derived from ours technologies. If this is the case, we will not be able to focus correctly on the problem solution. For this reason, if the appropriate resources are not found, we could be forced to build them ourselves, possibly with the effort of make these new technologies reusable for the future.

3 Requirements

In a home of a given city (e.g. Bologna), a DDR robot is used to clean the floor of a room ([R-FloorClean](#)). The floor in the room is a flat floor of solid material and is equipped with two sonars, named sonar1 and sonar2 as shown in the picture (sonar1 is that at the top). The initial position (start-point) of the robot is detected by sonar1, while the final position (end-point) is detected by sonar2.



The robot works under the following conditions:

1. [R-Start](#): an authorized user has sent a START command by using a human GUI interface (console) running on a conventional PC or on a smart device (Android).
2. [R-TempOk](#): the value temperature of the city is not higher than a prefixed value (e.g. 25 degrees Celsius).
3. [R-TimeOk](#): the current clock time is within a given interval (e.g. between 7 a.m and 10 a.m)

While the robot is working:

- it must blink a Led put on it, if the robot is a real robot ([R-BlinkLed](#)).
- it must blink a Led Hue Lamp available in the house, if the robot is a virtual robot ([R-BlinkHue](#)).
- it must avoid fixed obstacles (e.g. furniture) present in the room ([R-AvoidFix](#)) and/or mobile obstacles like balls, cats, etc. ([R-AvoidMobile](#)).

Moreover, the robot must stop its activity when one of the following conditions apply:

1. [R-Stop](#): an authorized user has sent a STOP command by using the console.
2. [R-TempKo](#): the value temperature of the city becomes higher than the prefixed value.
3. [R-TimeKo](#): the current clock time is beyond the given interval.
4. [R-Obstacle](#): the robot has found an obstacle that it is unable to avoid.
5. [R-End](#): the robot has finished its work.

During its work, the robot can optionally:

- [R-Map](#): build a map of the room floor with the position of the fixed obstacles. Once built, this map can be used to define a plan for an (optimal) path from the start-point to the end-point.

Other requirements:

1. The work can be done by a team composed of NT people, with $1 \leq NT \leq 4$.
2. If $NT > 1$, the team must explicitly indicate the work done by each component.
3. If $NT = 4$, the requirement R-Map is mandatory.

4 Requirement analysis

4.1 Glossary

Analyzing the text, the most important terms are showed in the following table:

Term	Meaning
<i>DDR (Dfferential Drive Robot)</i>	Entity that is responsible to clean the floor, could be virtual or real.
<i>Sonar</i>	Entities with a fixed position capable of emitting events when the DDR is close.
<i>Virtual environment</i>	It's a given application written in JavaScript that shows a virtual room made of a set of walls with fixed and mobile obstacle and sonars.
<i>Authorized user</i>	A user is an authorized person which is allowed to start the cleaning process and to stop it at any time.
<i>Console</i>	Interface used by the user to send commands to the DDR.
<i>Obstacle</i>	Fixed or moving block that should be avoided by the DDR.
<i>Map</i>	Symbolic representation of the floor of the virtual room that shows fixed obstacles and free floor.
<i>Start/End point</i>	Respectively the starting and the final position of the DDR. Starting from the initial position, after the DDR has completed its work, it should go to the final position.

4.2 Analysis

The first thing we should do is to analyze the requirements to obtain the logical architecture, so we have to think at the structure, the interaction and finally the behaviour of the system.

To model the system we use *QActor*, that is a custom meta-model inspired to the actors model (as can be found in the Akka library). The qa language is a custom language that can allow us to express in a concise way the structure, the interaction and also the behaviour of (distributed) software systems composed of a set of *QActor*[1]. From a very high point of view the system is composed by two main entities: a robot executor and by a console. The robot executor is a simple interpreter of commands (move forward/right/backward/left) .Through the console the user can trigger the activation of the DDR.

We can now start thinking about the interaction. Currently, we already have a player/executor [2], but we will discuss in the problem analysis other possibilities. From the requirements we understand that the console should have at least two button, the 'start' for triggering the cleaning process and the 'stop' button for stop it in any time. We decided also to add third way of interaction to send the

robot to the initial position, in such a way the user has the possibility to restart the cleaning process again.

Understood the requirements and after a conversation with the customer we can briefly summarize the behaviour of the system as follows: the DDR robot makes a first exploration round of the room. At the end of the first phase it should have defined the map of the floor (if all conditions remain respected during the execution), then it has to return to the initial point and makes a second round without colliding with the obstacles previously detected. The activity of the DDR is pointed out through a blinking led, at any time, if the conditions are not met, the execution must stop. It is important to note that only the authorized users can interact with the DDR using the appropriate console, this means that the console initially should provide the capability for getting the authorization, and only after the previous step should provide the buttons to command the DDR.

4.3 Abstraction gap

The reason why we use the *QActor* language is to overcome the abstraction gap between the conventional (object-oriented) programming language such as Java, C#, and C, and the implementation of distributed proactive/reactive systems. With these common languages, we don't have the capability of expressing directly the concept of events or messages, thus we prefer to use a custom software factory (*QActor*), which instead provides us this possibility. Moreover, through this language we are able to capture main architectural aspects of the system (structure, interaction, behaviour), it provides support for rapid software prototyping and a graceful transition from object-oriented programming to message-based and event-based computations. Furthermore, the use of qa rather than UML allows to have a model which is directly interpretable from machines, while UML (that is semi-graphic and semi-formal) is interpretable only by humans and cannot represent concepts such as *message* and *event* used for the realization of microservices.

5 Problem analysis

5.1 Logic Architecture

As soon as all the requirements are clarified, we can start the analysis of the problem. As said before, we already have available one virtual DDR. The advantage in this case is of course the cost, since this resource is open source, but we should analyze other alternatives. For example it could be interesting to consider the real case, in which we want to build our DDR. This operation has a cost, because we need to buy the right tools, for instance a Raspberry Pi or an Arduino board (or both), plus all the needed sensors and actuators, like the temperature sensor for getting the current temperature, and led for signaling the state of the DDR. The decision to use the virtual environment is very important, because it has a big impact on the process development and on the cost for supporting it, for example there is the possibility that none of us is able to build

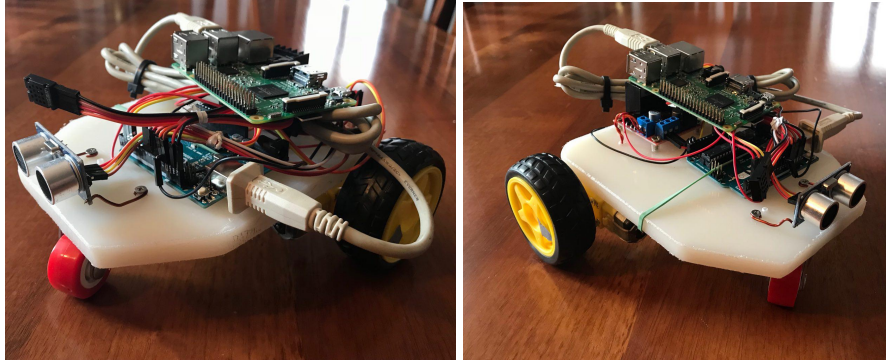


Fig. 1. Real DDR built with Raspberry Pi and Arduino

a physical robot. In such scenario, the amount of time required for completing the project will certainly be higher. In this academic context, we assume this cost affordable, for this reason we decided to use both the virtual DDR and the physical one, but this decision should be really taken by the buyer. In addition to this, we preferred to use both the approach in order to have an easy and quick way to test our software system.

Here we report some pictures of our real DDR built using both a Raspberry Pi that act as middleware, and an Arduino that is the interpreter of commands by a remote PC trough the Raspberry Pi.

In this context as a message we mean an information that is sent by someone to just one receiver; this information can never be lost, since is always enqueued on the receiver's queue. On the other hand by event we mean something that occurs in a precise time, and can be listened by anyone interested, but if the receiver is busy in another task this information is lost. We made an important decision concerning our system: all this interactions between the user and the robot should be implemented as messages instead of only events, since they are fundamental for the work flow of the system. In other words we don't want the user has to repeat two times the start or the stop due to the loose of events.

Now we are ready to present our first logical architecture. Here, we can see that we have two different context, one for the console, and another one for the robot. The console is responsible for starting the whole process by sending a `consoleCmdMsg(start)` message, that is listened by the robot. As soon as from the console we send the `consoleCmdMsg(halt)`, the robot stops its execution.

console.qa Model Analysis 0

```

1 System console
2
3 Dispatch consoleCmdMsg : consoleCmd ( X )
4
5 Context ctxConsole ip [ host="localhost" port=8029 ]

```

```

6 Context ctxRobot ip [ host="localhost" port=8030 ] --standalone
7
8 QActor mvcccontroller context ctxConsole {
9
10     Plan init normal[
11         println("MVCCONTROLLER_starts...");
12         delay 1000
13     ]
14     switchTo sendStart
15
16     Plan sendStart[
17         println("MVCCONTROLLER_send_start_to_robot...");
18         sendto player in ctxRobot -m consoleCmdMsg : consoleCmd(start);
19         delay 3000
20     ]
21     switchTo sendStop
22
23     Plan sendStop[
24         println("MVCCONTROLLER_send_stop_to_robot...");
25         sendto player in ctxRobot -m consoleCmdMsg : consoleCmd(halt)
26     ]
27     switchTo end
28
29     Plan end[
30         println("MVCCONTROLLER_ends...")
31     ]
32
33
34 }

```

robot.qa Model Analysis 0

```

1 System robot
2
3 Dispatch consoleCmdMsg : consoleCmd ( X )
4 Dispatch startMsg : start (X)
5 Dispatch stopMsg : halt (X)
6
7 Context ctxRobot ip [ host="localhost" port=8030]
8
9 QActor player context ctxRobot {
10
11     Plan init normal[
12         println("ROBOT_starts...")
13     ]
14     switchTo waitCommand
15
16     Plan waitCommand[]
17     transition stopAfter 60000
18     whenMsg consoleCmdMsg -> waitforMsg

```

```

19
20     Plan waitForMsg[
21         onMsg consoleCmdMsg : consoleCmd ( start ) -> {
22             println("ROBOT_received_START");
23             selfMsg startMsg : start (x)
24         };
25         onMsg consoleCmdMsg : consoleCmd ( halt ) -> {
26             println("ROBOT_received_STOP");
27             selfMsg stopMsg : halt (x)
28         }
29     ]
30     transition stopAfter 60000
31         whenMsg startMsg -> starts,
32         whenMsg stopMsg -> end
33
34     Plan starts resumeLastPlan [
35         println("ROBOT_execution_started")
36     ]
37     switchTo waitCommand
38
39     Plan end [
40         println("ROBOT_ends_due_to_STOP_message");
41         delay 4000
42     ]
43 }

```

As a first extension in our analysis we add a new actor in the *ctxConsole*, that is the *tempdetector*. This new component simulates the scenario in which the temperature exceeds the limit, consequently it sends a message to the *mvcccontroller*, which in turn shuts down the robot execution by sending a *consoleCmdMsg(halt)* to the player.

We can also extend whole system with a third context, which is the *ctxFrontend*. The idea here is that we have a *frontend* that can be in two different state: firstly it shows the login page to the user, and only after the authentication the *frontend* shows the command buttons.

console.qa Model Analysis 1

```

1  [...]
2
3  Dispatch frontendCmdMsg : frontendCmd (X)
4  Dispatch startTempMsg : startTemp (X)
5  Dispatch tempStateMsg : tempState (X)
6
7  [...]
8
9  QActor mvcccontroller context ctxConsole {
10
11     Plan init normal[
12         println("CONSOLE_starts...");

```



```

13         delay 1000
14     ]
15     switchTo waitCommand
16
17     Plan waitCommand[]
18     transition stopAfter 600000
19         whenMsg frontendCmdMsg -> checkConditions,
20         whenMsg tempStateMsg -> sendStop
21
22     Plan checkConditions [
23         println("MVCCONTROLLER_conditions_valid_and_start_from_
24             FRONTEND");
25         forward tempdetector -m startTempMsg : startTemp (x) //
26             assuming that condition is already verified for the first time
27     ]
28     switchTo sendStart
29
30     Plan sendStart[
31         println("MVCCONTROLLER_send_start_to_robot...");
32         sendto player in ctxRobot -m consoleCmdMsg : consoleCmd(start);
33         delay 1000
34     ]
35     switchTo waitCommand
36
37     [...]
38 }
39
40 QActor tempdetector context ctxConsole {
41
42     Plan init normal[
43         println("TEMPDETECTOR_starts...");
44         delay 20000
45     ]
46     switchTo waitCommand
47
48     Plan waitCommand[]
49     transition stopAfter 600000
50         whenMsg startTempMsg -> sendTempKO
51
52     Plan sendTempKO[
53         println("TEMPDETECTOR_temperature_raises_over_the_limit!_
54             Event!");
55         forward mvcccontroller -m tempStateMsg : tempState(false)
56     ]
57     switchTo end
58
59     Plan end[
60         println("TEMPDETECTOR_ends...")
61     ]

```

60 }

frontend.qa Model Analysis 1

```
1 System frontend
2
3 Dispatch frontendCmdMsg : frontendCmd (X)
4
5 Context ctxFrontend ip [ host="localhost" port=8031 ]
6 Context ctxConsole ip [ host="localhost" port=8029 ] -standalone
7
8 QActor frontendmanager context ctxFrontend {
9
10     Plan init normal [
11         println("FRONTEND_starts...");
12         delay 1000
13     ]
14     switchTo userLogin
15
16     Plan userLogin [
17         println("FRONTEND_login_page");
18         delay 2000
19     ]
20     switchTo userGui
21
22     Plan userGui [
23         println("FRONTEND_gui_page");
24         delay 2000
25     ]
26     switchTo sendStart
27
28     Plan sendStart[
29         println("FRONTEND_send_start");
30         sendto mvcontroller in ctxConsole -m frontendCmdMsg :
            frontendCmd(x)
31     ]
32 }
```

Here we do a second refactoring, zooming only into the console. We add the *timedetector* that has a similar role of *tempdetector*. We add also the *blinker*, which is the actor responsible for make the led blinking during the execution of the robot. The last component we introduce here is the *mind* that now takes the role from the *mvcontroller* to start and stop the application logic, in such a way the latter is only responsible for checking the conditions and starting both the *mind* and the *blinker* in case these are respected.

console.qa Model Analysis 2

```
1 [...]
2
```

```

3 Dispatch startTimeMsg : startTime (X)
4 Dispatch timeStateMsg : timeState (X)
5 Dispatch startMsg : start (X)
6 Dispatch stopMsg : halt (X)
7
8 [...]
9
10 QActor mvcccontroller context ctxConsole {
11
12     [...]
13
14     Plan waitCommand[]
15     transition stopAfter 600000
16         whenMsg frontendCmdMsg -> checkConditions,
17         whenMsg tempStateMsg -> sendStop,
18         whenMsg timeStateMsg -> sendStop
19
20     Plan checkConditions [
21         println("MVCCONTROLLER_conditions_valid_and_start_from_
22             FRONTEND");
23         forward tempdetector -m startTempMsg : startTemp (x); //
24             assuming that condition is already verified for the first time
25         forward timedetector -m startTimeMsg : startTime (x) //assuming
26             that condition is already verified for the first time
27     ]
28     switchTo sendStart
29
30     Plan sendStart[
31         println("MVCCONTROLLER_received_start");
32         forward blinker -m consoleCmdMsg : consoleCmd(start);
33         forward mind -m consoleCmdMsg : consoleCmd(start);
34         delay 1000
35     ]
36     switchTo waitCommand
37
38     Plan sendStop[
39         println("MVCCONTROLLER_received_stop");
40         forward blinker -m consoleCmdMsg : consoleCmd(halt);
41         forward mind -m consoleCmdMsg : consoleCmd(halt)
42     ]
43     switchTo end
44
45     [...]
46 }
47
48 QActor mind context ctxConsole {
49     Plan init normal[
50         println("MIND_starts...")
51     ]
52     switchTo waitCommand

```

```

50
51     Plan waitCommand[]
52     transition stopAfter 600000
53         whenMsg consoleCmdMsg -> handleConsoleCmd,
54         whenMsg startMsg -> startMind,
55         whenMsg stopMsg -> stopMind
56
57     Plan handleConsoleCmd[
58         onMsg consoleCmdMsg : consoleCmd(halt) -> selfMsg stopMsg :
59             halt (x);
60         onMsg consoleCmdMsg : consoleCmd(start) -> selfMsg startMsg :
61             start (x)
62     ]
63     switchTo waitCommand
64
65     Plan startMind[
66         println("MIND_send_start_to_robot...");
67         sendto player in ctxRobot -m consoleCmdMsg : consoleCmd(start)
68     ]
69     switchTo waitCommand
70
71     Plan stopMind[
72         println("MIND_send_stop_to_robot...");
73         sendto player in ctxRobot -m consoleCmdMsg : consoleCmd(halt);
74         println("MIND_ends...")
75     ]
76 }
77
78 QActor blinker context ctxConsole {
79
80     Plan init normal[
81         println("BLINKER_starts...")
82     ]
83     switchTo waitCommand
84
85     Plan waitCommand[]
86     transition stopAfter 600000
87         whenMsg consoleCmdMsg -> handleConsoleCmd,
88         whenMsg startMsg -> startBlink,
89         whenMsg stopMsg -> stopBlink
90
91     Plan handleConsoleCmd[
92         onMsg consoleCmdMsg : consoleCmd(halt) -> selfMsg stopMsg :
93             halt (x);
94         onMsg consoleCmdMsg : consoleCmd(start) -> selfMsg startMsg :
95             start (x)
96     ]
97     switchTo waitCommand
98
99     Plan startBlink[

```

```

96         println("BLINKER_start_blinking")
97     ]
98     switchTo waitCommand
99
100     Plan stopBlink[
101         println("BLINKER_stop_blinking");
102         println("BLINKER_ends...")
103     ]
104 }
105
106 QActor tempdetector context ctxConsole {
107     [...]
108 }
109
110 QActor timedetector context ctxConsole {
111
112     Plan init normal[
113         println("TIMEDETECTOR_starts...");
114         delay 20000
115     ]
116     switchTo waitCommand
117
118     Plan waitCommand[]
119     transition stopAfter 600000
120         whenMsg startTimeMsg -> sendTimeKO
121
122     Plan sendTimeKO[
123         println("TIMEDETECTOR_time_is_no_more_in_range!_Event!");
124         forward mvcontroller -m timeStateMsg : timeState(false)
125     ]
126     switchTo end
127
128     Plan end[
129         println("TIMEDETECTOR_ends...")
130     ]
131
132 }

```

6 Project

6.1 Project Analysis

In the project analysis we can start thinking about the technologies to use for implementing our system. Regarding the frontend we already developed during the course a server that implements a login with user and password, and only after the authentication, the commands buttons are showed to the user. This fronted server is written in Node, and it uses MongoDB for storing the information about the users. We want to highlight the fact that our goal wasn't

mainly a secure system, thus we focused more in other aspects.

About the interaction we have previously seen that we prefer messages. For this reason we must use an EventHandler for mapping the two. Again we have several degrees of freedom, for example we can exchange messages exploiting the *QActor* infrastructure or we can base our system to a MQTT server for dispatching the messages. The problem is that at the moment the EventHandler is not supported yet if we use an MQTT server, hence we are forced to use the *QActor* infrastructure.

At this point we can zoom in a deeper level of details and introduce new components into the system: the mind and the controller. To sum up our system is now composed by these four components:

- **Robot executor:** in our system the robot executor is called *player* and has only the purpose to receive commands from the application logic and act as an interpreter for the DDR. The executor is able to receive any commands in two different ways: we can send a low level message (a string formatted in a proper way) using a tcp connection, or we can work at a higher level, exploiting the *QActor* infrastructure, that is indeed capable of delivering all the messages and events to the components of the system.
- **Mind and Controller:** this represents the application logic. To have a less complexity during the development and guarantee a better flexibility for the future we opted for splitting the mind in two different components: the "real" *mind* which decides the actions that the DDR has to perform by calculating the next step to explore the map using the *A* algorithm* and the *mvcontroller* that maintains the consistency of the model and handles incoming events. This division allows to change in a very simple way the logic used to explore the room and build the map, if needed one day.
- **Console:** it is not the same console seen during model analysis, is instead represented by our frontend server, is a component that interacts with the *player*, communicating to it the commands given by the authenticated users. The commands screen can be seen by a user only after that he completed the sign up and login procedure.

At this point we have to face the a problem: where do we put the model containing all the information regarding the state of the DDR (state, current temperature, currents time, state of the led) ? We have different opportunities, but our decision is to replicate the model between the application logic and the console. More precisely, the model in the console is a mirror of the model contained in the application logic, in such a way the controller is still in charge of changing its own model, but since all the information are replicated, these are also available to the final user through the console. This approach allows us to decouple this two main components of our application, so that we can update the two independently.

6.2 Workplan

For the process development we will work using the **SCRUM** method, i.e. an agile framework for software development based on the division of the work in *sprint*. Each sprint is a prefixed time box during which the team works on a determined requirement. In the first sprint we will implement the first and easier requirements, and only after the testing we will start to tackle the optional requirement [R-Map](#), since it is the most difficult one. Of course this is not a long term project, so we dimension one sprint in three days, so we estimated that within 8 sprints at most we should complete the project.

7 Implementation

As we said in the project analysis the console model is a copy of the one contained in the *mvcontroller*. To maintain the consistency between the two we inserted four adapter to the frontend: one for the temperature, a second for the time, a third for the led state, and finally an adapter for the status of the DDR (started/stopped/restarted). This four components could be inserted in just one *QActor*, but we used four different *QActors*, in such a way we can easily update or extend one without touching the others working piece of code. The adapters use a *restfull* API offered by the frontend. We decide to send, update and map dates from controller to frontend manually, but exist other mechanisms to automatize these operations.

The temperature sensor, the clock and the led are simulated by three qactors. This three components are just wrappers, hence if in the future we want to change the technology we only need to update this actors, and we don't need to change anything in both the *mvcontroller* and the *mind*.

The time and temperature checking are implemented exploiting the PROLOG language, in fact thanks to its declarative properties, it's very easy to express rules that should be respected. For example, in the case of the the temperature, when the user press the start button, the `initialCheck` rule is executed and it retrieves the current temperature from the model, it compares the current value with the limit, and if all the other requirements are respected it emits directly event that starts the cleaning process. On the other hand, the model used in the frontend is express in JSON, since it is a very simple format that can be updated in handy way exploiting the RESTFUL API offered by the frontend.

Periodically there is an actor that acts as a wrapper for Node script, which is able to retrieve the current temperature in Bologna, and to emits an events, which has the value as payload. Every time that this occurs, we check again the condition, in such a way we are able to respect [R-TimeOk](#). The process is similar when we want to check the time limit, but in this case we simulated the clock with a Java class, to which we ask the current hour periodically.

The Node script and Java class generate their corresponding event every 30 seconds, and consequently the mind check the new conditions and whether to

stop or not the application.

For **R-Map** we exploited the working way of the virtual environment, in fact when we send a command to the server, we have to specify as argument the amount of time for that the correspondent operation lasts. In this way we have seen that in the virtual environment, if we fix the time at 300 ms, in order to go from the left side of the room to the right side, five steps are required, while for traversing the room from south to north four steps are needed. Hence the mind, before the execution initializes an empty 5x6 matrix, that represents the entire map. When the DDR is able to move to an adjacent cell, it marks that cell with **1**, to express the fact that the corresponding location is clean.

During the cleaning (exploration) of the floor the robot could find on its path two different kinds of obstacles, fixed and movable. They are identified by the virtual sonar on the robot, which is however not able to recognize their type. For this reason, after that an obstacle is detected by the sonar, the robot moves backward and waits for three seconds. Then it tries to move forward, if an obstacle is found again we are assuming that it should be a fixed one and a **X** is set in the corresponding matrix by the mind, other case it is a moving obstacle and a **1** is put in matrix. Doing this way, we can easily solve the problem of the second cycle, in fact in the second phase, the A* algorithm will know that the cells marked with 'X' aren't available due to the presence of a fixed obstacle, so it will avoid to the action that will result in a collision.

Every time virtual robot moves forward or backward it should do it for a certain amount of time, but when an obstacle is encountered before the completion of the movement, the DDR could end up in inconsistent state, and not in one of the cells of the grid. For this reason every time a move command is sent to the robot a timer is reset and started, so that it can be stopped when an obstacle is detected. At this point the mind can retrieve real duration of robot's move to send a moveBackward with a more precise duration. In this way robot return at the exact start point after a collision with an obstacle.

Regarding the way of working of the mind, we already said it uses mainly the A* algorithm, but we should point out some important details. The matrix has dimension 5x6, hence the complexity of the problem (finding the best path from an initial point to the final point traversing all the cells) has an exponential complexity. For this reason we split the problem in two: first we use A* for finding the best path to cover all the cells, and then, from the current position, we run a BFS over the available cells to make the DDR to go the final position. Still, the problem of finding the best path traversing all the cells is too hard, so we implemented an approximation.

More precisely, when the mind want to compute the next move to execute, it runs again A*, but with a simpler task: it doesn't ask for the best path for going to the final position, but it asks for the next move for going to the first cell still not visited. Keep in mind that meanwhile the DDR is able to mark with 'X' the cells not available due to the presence of fixed obstacles.

8 Testing

temptest Actor

```
1 Plan tempadapter[
2     emit currentTemperature : currentTemperature(20);
3     delay 20000;
4     println("Temperature_is_going_above_the_limit");
5     emit currentTemperature : currentTemperature(35)
6 ]
```

The use of the previous Plan inside the *QActortemperaturesimulator* allows to test the condition "R-TempKo", because assuming a limit of 30 degree Celsius, the threshold will be exceeded, and this event will cause the stop of the whole system.

timetest Actor

```
1 Plan gettime[
2     emit isTime : isTime(2);
3     delay 20000;
4     println("Time_is_going_out_of_ranges");
5     emit isTime : isTime(21)
6 ]
```

In the same way, using the instructions above in the *QActortimesimulator*, it's possible to simulate correctly the flow of time and to test the condition "R-TimeKo", in fact the system should stop when time changes and goes out of the given ranges.

For the condition "R-Stop" the test can be easily effectuated sending at any time the corresponding event "usercmd(h(x))" that cause the stop of the whole system, by clicking on the Stop button on the console.

Concerning the condition "R-Obstacle" it can be tested by substituting the map of the virtual environment (i.e. the file named *sceneConfig.js* of the *WebGLScene* directory) with a different one in which the DDR cannot find a path to explore the entire floor, after a specific timeout the system stops.




Regarding the last condition "R-End" it can be simulated at the end of the second round of the room sending the event "stop2exploration" which finishes the system.

9 Deployment

The entire system can be deployed on a single PC that executes the frontend server, the mind of the robot and the DDR. The other possibility would be to execute the mind and the frontend on a dedicated pc, while the player for the DDR runs over a Raspberry PI. In this last case we should change the commands executed by the player, that are not meant anymore for the virtual environment, but rather commands sent to Arduino (that has to move the motors), In addition to this we have to handle somehow the information sent by the board sensors, such as real-time clock, temperature, and eventually obstacles. There is also a third possibility, we can deploy the whole system into a Raspberry Pi, and connect to it using the Web GUI from a smartphone.

It is fundamental to note that when we start the distributed application we MUST have care of launching firstly the virtual environment (if needed), then the context containing the mind and lastly the frontend server. This is needed because virtual environments opens a TCP socket port, which is used by the mind to send the commands to execute. In turn the mind opens another tcp socket used for the communication with the frontend server.

10 Authors

Photos of the authors		
		
Adriano Cardace	Filippo Mearini	Gabriele Sorrentino

References

1. Introduction to *QActors* and *QRobots*, Antonio Natali, 2017.
2. <https://github.com/PierfrancescoSoffritti/configurable-threejs-app>