

Software Engineering process

Luca Bonfiglioli, Nicola Fava, Antonio Grasso

Alma Mater Studiorum – University of Bologna
viale Risorgimento 2, 40136 Bologna, Italy
`luca.bonfiglioli10@studio.unibo.it`
`nicola.fava@studio.unibo.it`
`antonio.grasso5@studio.unibo.it`

Table of Contents

Software Engineering process	1
<i>Luca Bonfiglioli, Nicola Fava, Antonio Grasso</i>	
1 Introduzione	3
2 Vision	3
3 Requisiti	4
4 Analisi dei requisiti	5
5 Analisi del problema	11
6 Progettazione	18
7 Implementazione	32
8 Autori	33

1 Introduzione

L'ingegneria diversifica le fasi di produzione del software delineando un flusso di lavoro (**workflow**) costituito da un insieme di passi: definizione dei requisiti, analisi dei requisiti, analisi del problema, progettazione della soluzione, implementazione della soluzione e collaudo.

La progettazione del software può seguire due approcci:

- **Approccio top-down**: si considera l'intero sistema software come un'unica entità e lo si scompone per ottenere più di un sotto-sistema o componente. Ogni sotto-sistema o componente viene considerato come un sistema e ulteriormente decomposto;
- **Approccio bottom-up**: si compongono componenti di più alto livello utilizzando componenti base o di più basso livello. Si continua a creare componenti di più alto livello finché il sistema desiderato non si evolve come un singolo componente.

I problemi possono essere affrontati utilizzando due differenti approcci:

- **Approccio olistico**: un sistema viene visto come un insieme che va oltre i sotto-sistemi o i componenti di cui è costituito;
- **Approccio riduzionistico**: non può essere sviluppato nessun sistema a meno che non si conoscano informazioni su di esso e sui componenti di cui si compone.

Occorre chiedersi se sia meglio tentare di risolvere un problema partendo dalle ipotesi tecnologiche (come possono essere ad esempio gli oggetti Java) o piuttosto seguire un approccio in cui l'analisi del problema precede la scelta della tecnologia più appropriata. Dopo aver completato l'analisi del problema è possibile imbattersi in un cosiddetto **abstraction gap**, che evidenzia un gap tra le tecnologie disponibili ed il problema che si deve risolvere.

2 Vision

La visione adottata è quella per cui non si possa cominciare a scrivere codice prima di aver completato la fase di progettazione, che a sua volta deve seguire la fase di analisi del problema, preceduta da quella di analisi dei requisiti.

Si utilizza una metodologia top-down che consiste nell'aggregare il problema posto dai requisiti ad un livello generale, lasciando in ultima istanza il trattamento dei dettagli, ben distinguendo la fase di analisi, strategica nel processo di sviluppo del software, da quella di progettazione.

L'obiettivo dell'analisi dei requisiti è quello di capire cosa voglia il committente, al fine di produrre, al termine dell'analisi, uno o più modelli delle entità descritte dai requisiti, nel modo più formale e pratico possibile, catturandone gli aspetti essenziali in termini di struttura, interazione e comportamento.

Lo scopo della fase di analisi del problema è quello di capire il problema posto dai requisiti, le problematiche riguardanti il problema e i vincoli imposti

dal problema o dal contesto. L'analisi non ha come obiettivo la descrizione delle proprietà strutturali e comportamentali del sistema che risolverà il problema, in quanto questo è l'obiettivo della progettazione. Il risultato dell'analisi del problema è l'architettura logica implicata dai requisiti e dalle problematiche individuate.

L'obiettivo della fase di progettazione è quello di raffinare l'architettura logica del sistema, considerando tutti gli aspetti vincolanti che si sono trascurati nelle fasi precedenti, per arrivare a delineare e descrivere non solo la soluzione al problema ma anche e soprattutto i motivi che hanno condotto a questa soluzione. L'architettura del sistema scaturita dalla progettazione dovrebbe essere il più possibile indipendente dalle tecnologie realizzative. La progettazione dovrebbe procedere dal generale al particolare, sviluppando per primi i sottosistemi più critici individuati dall'analisi.

All'inizio del processo di sviluppo del software non si considera nessuna ipotesi tecnologica (come ad esempio il paradigma di programmazione ad oggetti o il paradigma di programmazione funzionale).

3 Requisiti

Nella casa di una determinata città (per esempio Bologna), viene usato un `ddr` robot per pulire il pavimento di una stanza ([R-FloorClean](#)).

Il pavimento della stanza è un pavimento piatto di materiale solido ed è equipaggiato con due *sonars*, chiamati `sonar1` e `sonar2`, come mostrato in Figura 1 (`sonar1` è quello in alto). La posizione iniziale (`start-point`) del robot è rilevata da `sonar1`, mentre la posizione finale (`end-point`) da `sonar2`.

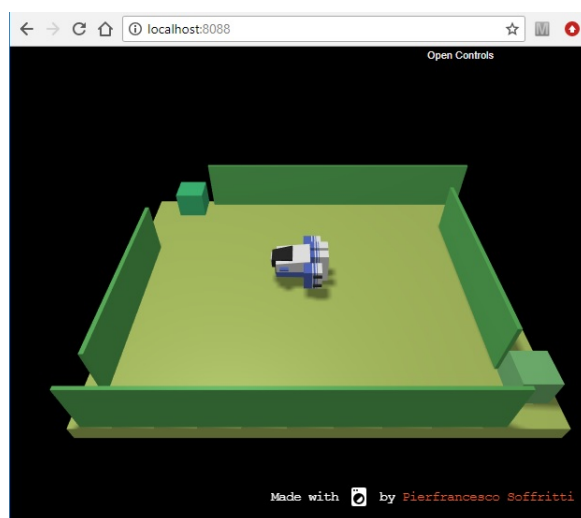


Fig. 1. Esempio di pavimento con il robot in ambiente simulato

Il robot lavora secondo le seguenti condizioni:

1. **R-Start**: un utente autorizzato (**authorized user**) ha inviato un comando **START** usando un'interfaccia GUI umana (**console**) in esecuzione su un normale PC oppure su uno smart device (**Android**).
2. **R-TempOk**: il valore di temperatura della città non è superiore ad un valore prefissato (per esempio 25 ° Celsius).
3. **R-TimeOk**: l'orario corrente è all'interno di un intervallo dato (per esempio fra le 7 e le 10 di mattina).

Mentre il robot è in movimento:

- un Led posto su di esso deve lampeggiare, se il robot è un **real** robot (**R-BlinkLed**);
- una Led Hue Lamp disponibile nella casa deve lampeggiare, se il robot è un **virtual** robot (**R-BlinkHue**);
- deve evitare gli ostacoli fissi (per esempio i mobili) presenti nella stanza (**R-AvoidFix**) e/o gli ostacoli mobili come palloni, gatti, ecc. (**R-AvoidMobile**).

Inoltre il robot deve interrompere la sua attività quando si verifica una delle seguenti condizioni:

1. **R-Stop**: un utente autorizzato (**authorized user**) ha inviato il comando di **STOP** utilizzando la **console**.
2. **R-TempKo**: il valore di temperatura della città diventa più alto del valore prefissato.
3. **R-TimeKo**: l'orario corrente non è più all'interno dell'intervallo dato.
4. **R-Obstacle**: il robot ha trovato un ostacolo che non è in grado di evitare.
5. **R-End**: il robot ha finito il suo lavoro.

Durante il suo funzionamento il robot può opzionalmente:

- **R-Map**: costruire una mappa del pavimento della stanza con la posizione degli ostacoli fissi. Una volta ottenuta, la mappa può essere utilizzata per definire un piano per un percorso (ottimo) dallo **start-point** all'**end-point**.

4 Analisi dei requisiti

Il sistema da modellare sarà, come esplicitato dai requisiti, eterogeneo e distribuito, in particolare composto da almeno due nodi: il nodo "Robot" e il nodo "PC/Android".

Per la modellazione si utilizza il linguaggio *QActor* in quanto adatto alla modellazione di sistemi distribuiti.

Il primo dei due nodi che si è modellati è il nodo "PC/Android" che si occupa di mostrare la GUI e di interagire direttamente con un utente umano, richiedendone l'autenticazione. Come da requisito **R-Start** l'interfaccia utente deve poter essere utilizzabile sia su PC che su un dispositivo **Android**. Tuttavia, essendo le funzioni che essa deve svolgere identiche in entrambi i casi, si sono

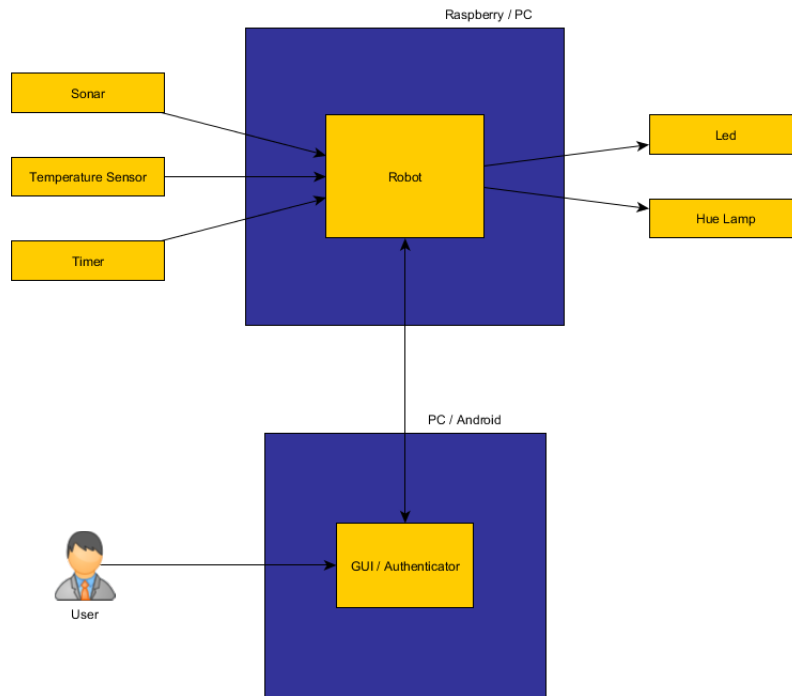


Fig. 2. Diagramma informale dell'analisi dei requisiti

rappresentati entrambi i nodi come un unico nodo. Su questo nodo esegue l'attore "GUI/Authenticator", che consente all'utente di autenticarsi e inviare i comandi di **START** e **STOP** al robot (**R-Start** e **R-Stop**).

Il secondo nodo che si è modellato è il nodo "Raspberry/PC", responsabile del controllo del robot. Esso può essere in esecuzione su un PC, nel caso del **virtual** robot, oppure su un Raspberry Pi nel caso del **real** robot.

L'attore "Robot" si pone in attesa dei comandi inviati da "GUI/Authenticator" ed è in grado di ricevere informazioni relative alle condizioni di temperatura ed al tempo (**R-TempOk**, **R-TimeOk**, **R-TempKo**, **R-TimeKo**). Durante l'esecuzione, se il robot è in movimento, l'attore "Robot" invia a "Led" e a "Hue Lamp" i comandi per l'accensione e lo spegnimento necessari a farli lampeggiare (**R-BlinkLed**, **R-BlinkHue**).

L'attore "Robot" si occupa inoltre di gestire la logica applicativa, che consiste, in seguito alla ricezione del comando **START** da parte dell'utente, nel prendere decisioni circa il movimento del robot all'interno della stanza – per il robot reale – e all'interno dell'ambiente simulato – per il robot virtuale – tentando di evitare gli ostacoli fissi e mobili (**R-AvoidFix**, **R-AvoidMobile**), costruendo una mappa del pavimento (**R-Map**) e interrompendone l'attività una volta completato il proprio lavoro (**R-End**).

Inoltre, se l'attore "Robot" trova un ostacolo che non riesce ad evitare si deve fermare (**R-Obstacle**). Questa situazione si verifica quando il robot trova uno o più ostacoli che gli impediscono di raggiungere il secondo sonar.

Vengono di seguito riportati i modelli formali risultati dall'analisi dei requisiti:

```
1 System systemRobot
2
3 Event robotCmd : robotCmd(X)
4 Event sensorEvent : sensorEvent(X)
5 Event outCmd : outCmd(X)
6
7 Context ctxRobot ip[host="localhost" port=5400]
8
9 QActor robot context ctxRobot {
10   Plan initial normal [
11     println("Robot started");
12     delay 2000
13   ]
14
15   switchTo waitForEvent
16
17   Plan waitForEvent [
18   ]
19   transition stopAfter 600000
20     whenEvent robotCmd -> handleEvent,
21     whenEvent sensorEvent -> handleEvent
22   finally repeatPlan
23
24   Plan handleEvent resumeLastPlan [
25     onEvent robotCmd : robotCmd(X) -> {
26       println("Robot receives event from user");
27       emit outCmd : outCmd(X)
28     };
29     onEvent sensorEvent : sensorEvent(sonar1) -> println("
Robot receives event from sonar1");
30     onEvent sensorEvent : sensorEvent(sonar2) -> println("
Robot receives event from sonar2");
31     onEvent sensorEvent : sensorEvent(temp) -> println("Robot
receives event from temperature sensor");
32     onEvent sensorEvent : sensorEvent(timer) -> println("
Robot receives event from timer sensor");
33     printCurrentEvent
34   ]
35 }
36
37 QActor sonarsensor1 context ctxRobot {
38   Plan initial normal[
39     println("Sonar1 started");
40     delay 3000
```

```

41     ]
42     switchTo emitEvents
43
44     Plan emitEvents[
45         emit sensorEvent : sensorEvent(sonar1)
46     ]
47 }
48
49 QActor sonarsensor2 context ctxRobot {
50     Plan initial normal[
51         println("Sonar2 started");
52         delay 3500
53     ]
54     switchTo emitEvents
55
56     Plan emitEvents[
57         emit sensorEvent : sensorEvent(sonar2)
58     ]
59 }
60
61 QActor temperaturesensor context ctxRobot {
62     Plan initial normal[
63         println("Temperature sensor started");
64         delay 4000
65     ]
66     switchTo emitEvents
67
68     Plan emitEvents[
69         emit sensorEvent : sensorEvent(temp)
70     ]
71 }
72
73 QActor timersensor context ctxRobot {
74     Plan initial normal[
75         println("Timer sensor started");
76         delay 4500
77     ]
78     switchTo emitEvents
79
80     Plan emitEvents[
81         emit sensorEvent : sensorEvent(timer)
82     ]
83 }
84
85 QActor led context ctxRobot {
86     Plan initial normal[
87         println("Led started")
88     ]
89     switchTo waitForEvent
90

```



```

91   Plan waitForEvent []
92   transition stopAfter 600000
93   whenEvent outCmd -> handleEvent
94   finally repeatPlan
95
96   Plan handleEvent resumeLastPlan [
97     println("Led receives event");
98     printCurrentEvent
99   ]
100 }
101
102 QActor huelamp context ctxRobot {
103   Plan initial normal [
104     println("Hue Lamp started")
105   ]
106   switchTo waitForEvent
107
108   Plan waitForEvent []
109   transition stopAfter 600000
110   whenEvent outCmd -> handleEvent
111   finally repeatPlan
112
113   Plan handleEvent resumeLastPlan [
114     println("Hue Lamp receives event");
115     printCurrentEvent
116   ]
117 }

```

Listato 1.1. ../src/sprint5/req_analysis/reqAnalysisRobot.qa

```

1  System systemRobot
2
3  Dispatch userCmd : userCmd(X)
4  Event robotCmd : robotCmd(X)
5
6  Context ctxRobot ip[host="localhost" port=5400] -standalone
7  Context ctxUser ip[host="localhost" port=5500]
8
9  QActor gui context ctxUser {
10   Plan initial normal [
11     println("Gui started")
12   ]
13   switchTo waitForMsg
14
15   Plan waitForMsg [
16   ]
17   transition stopAfter 600000
18   whenMsg userCmd -> handleMsg
19   finally repeatPlan
20

```

```

21 Plan handleMsg resumeLastPlan [
22     println("Gui receives user message - User pressed button"
23     );
24     onMsg userCmd : userCmd(X) -> emit robotCmd : robotCmd(X)
25 ]
26 }
27 QActor user context ctxUser {
28     Plan initial normal [
29         println("User started")
30     ]
31     switchTo sendMsg
32
33     Plan sendMsg[
34         println("User send messages");
35         forward gui -m userCmd : userCmd(X);
36         forward gui -m userCmd : userCmd(Y)
37     ]
38 }

```

Listato 1.2. ../src/sprint5/req_analysis/reqAnalysisUser.qa

Prima di descrivere più dettagliatamente i modelli prodotti durante l'analisi dei requisiti è necessaria una premessa relativa alle modalità di interazione dei componenti del sistema. Trattandosi di un sistema eterogeneo distribuito, ed utilizzando il linguaggio di modellazione QActor, le due modalità con cui i componenti all'interno del sistema possono interagire sono quelle ad eventi e messaggi.

In questo ambito un messaggio non è altro che un'informazione che il mittente invia ad uno specifico destinatario; al contrario, un evento cattura il concetto di informazione senza specifico destinatario: tutti i componenti del sistema interessati all'evento possono riceverlo.

Il sistema **systemRobot** è stato modellato come un sistema costituito di due contesti, il contesto stand-alone **ctxRobot** ed il contesto **ctxUser**.

All'interno del primo contesto operano i Qactor:

- **robot**: si mette in attesa di due eventi:
 - **robotCmd : robotCmd(X)**: evento emesso dall'utente autenticato per comandare il robot (start/stop), alla ricezione del quale viene emesso l'evento **outCmd : outCmd(X)**, necessario per comandare il funzionamento del led e della lampada Hue;
 - **sensorEvent : sensorEvent(X)**: evento che può essere emesso dai sonar presenti nella stanza o dalle sorgenti dei dati di temperatura e tempo.
- **sonarsensor1, sonarsensor2, temperaturesensor, timersensor**: emettono eventi **sensorEvent : sensorEvent(X)**, catturati dall'attore **robot**.
- **led, huelamp**: si mettono in attesa degli eventi **outCmd : outCmd(X)**, emessi dall'attore **robot**.

Dall'analisi dei requisiti sono emerse le necessità di interazione tra i sonar ed il robot, tra le sorgenti dei dati di temperatura e tempo ed il robot, nonché tra il robot stesso e i dispositivi attuatori, in questo caso led e lampada hue.

Sarebbe possibile modellare queste interazioni come messaggi: in questo caso sia i sonar che le sorgenti di temperatura e tempo dovrebbero conoscere lo specifico destinatario dei loro messaggi; allo stesso modo il robot dovrebbe conoscere i specifici dispositivi attuatori a cui inviare le informazioni.

Alla luce di ciò risulta più conveniente modellare queste interazioni tramite eventi, che permettono un maggiore disaccoppiamento delle entità in gioco, consentendo eventualmente a più robot distinti di raccogliere i dati in input e a differenti attuatori di ricevere comandi dal robot.

All'interno del contesto `ctxUser` operano invece gli attori:

- **user**: invia messaggi **userCmd : userCmd(X)** all'attore **gui**;
- **gui**: riceve i messaggi **userCmd : userCmd(X)** emettendo eventi **robotCmd : robotCmd(X)** destinati a comandare il robot.

In questo caso non si evidenziano particolari differenze nel modellare l'interazione user-gui tramite messaggi piuttosto che tramite eventi. Per quanto riguarda l'interazione gui-robot è più opportuno che essa sia modellata tramite eventi piuttosto che tramite messaggi, in quanto risultano più vantaggiosi per disaccoppiare l'interfaccia grafica dallo specifico robot comandato.

5 Analisi del problema

Il primo problema che sorge è quello di stabilire quale nodo si occuperà di autenticare l'utente. Una possibilità è quella di relegare l'autenticazione al nodo del robot: in questo caso il robot potrebbe non disporre delle adeguate risorse computazionali per gestire il processo di autenticazione, tuttavia questo garantirebbe maggiore sicurezza. Un'altra possibilità è che l'autenticazione venga gestita da un nodo diverso rispetto a quello del robot: ciò consentirebbe di non utilizzare le risorse computazionali del robot richiedendo però maggiori accortezze sulla sicurezza. In quest'ultimo caso l'autenticazione potrebbe essere gestita dal nodo dell'utente oppure da un nodo distinto, il quale comporterebbe costi maggiori.

Un altro problema è quello dell'interfaccia GUI, che deve poter eseguire su dispositivi eterogenei. A tal proposito, una possibilità sarebbe creare client nativi per ogni piattaforma con costi elevati oppure più semplicemente utilizzare una pagina web.

La comunicazione tra utente e robot tramite GUI può avvenire via messaggi o via eventi. La comunicazione ad eventi permette di disaccoppiare GUI e robot, consentendo di utilizzare un'unica GUI per comunicare con diversi robot. Utilizzando gli eventi può essere adottato un approccio **event-based** o un approccio **event-driven**. Nell'approccio **event-based** il robot non sarebbe sempre sensibile agli eventi, potendone perdere alcuni. Al contrario, nell'approccio **event-driven** il robot sarebbe sempre sensibile agli eventi perdendo tuttavia reattività.

Vengono di seguito riportati i modelli formali risultati dall'analisi del problema:

```
1 System systemRobot
2
3 Event robotCmd : robotCmd(X)
4 Event sensorEvent : sensorEvent(X)
5 Event outCmd : outCmd(X)
6
7 Context ctxProbRobot ip[host="localhost" port=5400]
8
9 QActor robot context ctxProbRobot {
10   Rules {
11     limitTemperatureValue(25).
12     minTime(7).
13     maxTime(10).
14     currentTempValue(0).
15     currentTimeValue(0).
16     evalTemp:-
17       limitTemperatureValue(MAX),
18       currentTempValue(VALUE),
19       eval(ge, MAX, VALUE).
20     evalTime:-
21       minTime(MIN),
22       maxTime(MAX),
23       currentTimeValue(VALUE),
24       eval(ge, VALUE, MIN),
25       eval(ge, MAX, VALUE).
26     startRequirementsOk :- evalTemp, evalTime.
27     map.
28   }
29
30   Plan initial normal [
31     println("Robot started");
32     delay 2000
33   ]
34
35   switchTo waitForEvent
36
37   Plan waitForEvent [
38   ]
39   transition stopAfter 600000
40     whenEvent robotCmd -> handleEvent,
41     whenEvent sensorEvent -> handleEvent
42   finally repeatPlan
43
44   Plan handleEvent resumeLastPlan [
45     onEvent robotCmd : robotCmd(cmdstart) -> {
46       [ !? startRequirementsOk ] {
47         println("Robot start");
```

```

48         emit outCmd : outCmd(startblinking);
49         [ !? map ]
50             println("Il robot segue il percorso ottimo")
51         else
52             println("Mentre il robot in azione deve costruire
la mappa")
53     }
54 };
55 onEvent robotCmd : robotCmd(cmdstop) -> {
56     println("Robot stop from user");
57     emit outCmd : outCmd(stopblinking)
58 };
59 onEvent sensorEvent : sensorEvent(sonar1) -> println("
Robot receives event from sonar1");
60 onEvent sensorEvent : sensorEvent(sonar2) -> {
61     emit robotCmd : robotCmd(cmdstop); // il robot ha
finito
62     [ !? map ]
63         addRule map // se non c'era una mappa precedente
viene creata
64 };
65 onEvent sensorEvent : sensorEvent(onboardsonar) -> {
66     //the robot stops for a while and then retries
67     println("Robot stop");
68     // se la prima volta che incontra un ostacolo:
69     delay 500;
70     println("Il robot prova di nuovo ad andare avanti");
71     // se invece la seconda volta che incontra lo stesso
ostacolo:
72     println("Il robot prova ad andare a destra o a sinistra
");
73     // se non si pu andare n a destra n a sinistra allora
il robot si ferma:
74     emit robotCmd : robotCmd(cmdstop)
75 };
76 onEvent sensorEvent : sensorEvent(temp(VALUE)) ->
ReplaceRule currentTempValue(X) with currentTempValue(
VALUE);
77 onEvent sensorEvent : sensorEvent(temp(X)) -> {
78     [ not !? evalTemp ] {
79         println("Robot stop from temperature sensor");
80         emit outCmd : outCmd(stopblinking)
81     }
82 };
83 onEvent sensorEvent : sensorEvent(timer(VALUE)) ->
ReplaceRule currentTimeValue(X) with currentTimeValue(
VALUE);
84 onEvent sensorEvent : sensorEvent(timer(X)) -> {
85     [ not !? evalTime ] {
86         println("Robot stop from time sensor");

```

```

87         emit outCmd : outCmd(stopblinking)
88     }
89 };
90     printCurrentEvent
91 ]
92 }
93
94 QActor sonarsensor1 context ctxProbRobot {
95     Plan initial normal [
96         println("Sonar1 started");
97         delay 3000
98     ]
99     switchTo emitEvents
100
101     Plan emitEvents [
102         emit sensorEvent : sensorEvent(sonar1)
103     ]
104 }
105
106 QActor sonarsensor2 context ctxProbRobot {
107     Plan initial normal [
108         println("Sonar2 started");
109         delay 3500
110     ]
111     switchTo emitEvents
112
113     Plan emitEvents [
114         emit sensorEvent : sensorEvent(sonar2)
115     ]
116 }
117
118 QActor sonarrobot context ctxProbRobot {
119     Plan initial normal [
120         println("Sonar on board started");
121         delay 5000
122     ]
123     switchTo emitEvents
124
125     Plan emitEvents [
126         emit sensorEvent : sensorEvent(onboardsonar)
127     ]
128 }
129
130 QActor temperaturesensor context ctxProbRobot {
131     Plan initial normal [
132         println("Temperature sensor started");
133         delay 4000
134     ]
135     switchTo emitEvents
136

```

```

137     Plan emitEvents[
138         emit sensorEvent : sensorEvent(temp(20));
139         delay 2000;
140         emit sensorEvent : sensorEvent(temp(30))
141     ]
142 }
143
144 QActor timersensor context ctxProbRobot {
145     Plan initial normal[
146         println("Timer sensor started");
147         delay 4500
148     ]
149     switchTo emitEvents
150
151     Plan emitEvents[
152         emit sensorEvent : sensorEvent(timer(9));
153         delay 2000;
154         emit sensorEvent : sensorEvent(timer(12))
155     ]
156 }
157
158 QActor led context ctxProbRobot {
159     Plan initial normal[
160         println("Led started")
161     ]
162     switchTo waitForEvent
163
164     Plan waitForEvent[]
165     transition stopAfter 600000
166     whenEvent outCmd -> handleEvent
167     finally repeatPlan
168
169     Plan handleEvent resumeLastPlan [
170         onEvent outCmd : outCmd(startblinking) -> println("Led
171         start blinking");
172         onEvent outCmd : outCmd(stopblinking) -> println("Led
173         stop blinking")
174     ]
175 }
176
177 QActor huelamp context ctxProbRobot {
178     Plan initial normal[
179         println("Hue Lamp started")
180     ]
181     switchTo waitForEvent
182
183     Plan waitForEvent[]
184     transition stopAfter 600000
185     whenEvent outCmd -> handleEvent
186     finally repeatPlan

```

```

185
186 Plan handleEvent resumeLastPlan [
187     onEvent outCmd : outCmd(startblinking) -> println("Hue
188     Lamp start blinking");
189     onEvent outCmd : outCmd(stopblinking) -> println("Hue
190     Lamp stop blinking")
189 ]
190 }

```

Listato 1.3. ../src/sprint5/prob_analysis/probAnalysisRobot.qa

```

1 System systemRobot
2
3 // payload: cmdstart o cmdstop
4 Dispatch userCmd : userCmd(X)
5 Event robotCmd : robotCmd(X)
6
7 Context ctxProbRobot ip[host="localhost" port=5400] -
    standalone
8 Context ctxProbUser ip[host="localhost" port=5500]
9
10 QActor gui context ctxProbUser {
11     Plan initial normal [
12         println("Gui started")
13     ]
14     switchTo waitForMsg
15
16     Plan waitForMsg [
17     ]
18     transition stopAfter 600000
19     whenMsg userCmd -> handleMsg
20     finally repeatPlan
21
22     Plan handleMsg resumeLastPlan [
23         println("Gui receives user message");
24         onMsg userCmd : userCmd(X) -> emit robotCmd : robotCmd(X)
25     ]
26 }
27
28 QActor user context ctxProbUser {
29     Rules {
30         isUserAuthenticated.
31     }
32
33     Plan initial normal [
34         println("User started")
35     ]
36     switchTo sendMsg
37
38     Plan sendMsg[

```



```

39     println("User send messages");
40     [ !? isUserAuthenticated ]
41         forward gui -m userCmd : userCmd(cmdstart);
42     delay 2000;
43     [ !? isUserAuthenticated ]
44         forward gui -m userCmd : userCmd(cmdstop)
45     ]
46 }

```

Listato 1.4. ../src/sprint5/prob_analysis/probAnalysisUser.qa

In conseguenza a quanto emerso dall'analisi dei requisiti, il sistema **systemRobot** è stato modellato come composizione di due contesti, rinominati per comodità **ctxProbRobot** e **ctxProbUser**.

Per ciò che riguarda il contesto standalone **ctxProbRobot**, la logica necessaria a soddisfare i requisiti **R-TempOk** e **R-TimeOk** è stata modellata attraverso una base di conoscenza espressa tramite regole Prolog inserite all'interno del QActor **robot**.

Come emerso in analisi dei requisiti il robot deve essere sensibile agli eventi **robotCmd : robotCmd(X)** e **sensorEvent : sensorEvent(X)**.

In particolare:

- alla ricezione dell'evento **robotCmd : robotCmd(cmdstart)** il QActor **robot** controlla che siano soddisfatti i requisiti di tempo e temperatura ed emette l'evento **outCmd : outCmd(startblinking)** necessario ad avviare il blinking del led e della lampada hue. Quindi controlla se sia già stata costruita una mappa della stanza: in caso affermativo segue il percorso ottimo fino alla destinazione, mentre in caso contrario procede alla costruzione della mappa della stanza;
- alla ricezione dell'evento **robotCmd : robotCmd(cmdstop)** viene emesso l'evento **outCmd : outCmd(stopblinking)** necessario ad interrompere il blinking del led e della lampada hue;
- alla ricezione dell'evento **sensorEvent : sensorEvent(sonar2)**, emesso in corrispondenza del passaggio del robot davanti al secondo sonar, se non presente in precedenza viene aggiunta la mappa della stanza alla base di conoscenza Prolog del robot;
- alla ricezione dell'evento **sensorEvent : sensorEvent(onboardsonar)**, emesso nel caso in cui il robot si imbatta in un ostacolo, il robot si ferma per poi tentare di superare l'ostacolo: nel caso in cui si tratti di un ostacolo fisso cerca di aggirarlo;
- alla ricezione degli eventi **sensorEvent : sensorEvent(temp(X))** e **sensorEvent : sensorEvent(timer(X))** viene controllato che siano ancora soddisfatti rispettivamente il requisito di temperatura ed il requisito di tempo, ed in caso contrario viene emesso l'evento **outCmd : outCmd(stopblinking)** necessario ad interrompere il blinking di led e lampada hue.

I QActor **sonarsensor1**, **sonarsensor2**, **sonarrobot**, **temperaturesensor** e **timersensor** emettono eventi **sensorEvent : sensorEvent(X)** con un payload

specifico in base al tipo di sensore che rappresentano: il QActor robot deve essere sensibile a questi eventi.

I QActor **huelamp** e **led** modellano invece i dispositivi attuatori, ed entrambi si mettono in attesa degli eventi **outCmd : outCmd(startblinking)** e **outCmd : outCmd(stopblinking)**, alla ricezione dei quali rispettivamente avviano o interrompono il blinking.

Per quanto riguarda il contesto **ctxProbUser** il QActor **user** modella l'utente che, una volta autenticato, può inviare messaggi **userCmd : userCmd(cmdstart)** e **userCmd : userCmd(cmdstop)** all'attore **gui**, che a sua volta, alla ricezione dei messaggi, si occupa di emettere gli eventi corrispondenti, eventi a cui deve essere sensibile il robot.

6 Progettazione

```
1  System robot
2
3  Event updateTemperature : updateTemperature(NAME, NEW_TEMP)
4  Event updateTime       : updateTime(NAME, CURRENT_TIME)
5  Event turnLed          : turnLed(NAME, NEW_STATE)
6  Event temperatureIsOk  : temperatureIsOk(STATE)
7  Event timeIsOk         : timeIsOk(STATE)
8
9  Event modelChanged     : modelChanged(resource(NAME, STATE))
10
11 // Emitted when A robot senses an obstacle (it can be the
12 //   pfrs robot, the physical robot...)
13 // VALUE is always true
14 Event obstacleDetected : obstacleDetected(VALUE)
15
16 // Emitted when A sonar senses something (unique sonars
17 //   present are in pfrs environment)
18 // VALUE is always true
19 // DVALUE is the value of the distance of the object sensed
20 Event sonarDetected : sonarDetected(name(NAME),
21 //   somethingDetected(VALUE), distance(DVALUE))
22
23 // Emitted when someone/thing modify the movement state of
24 //   the robot
25 // VALUE can be:
26 // - stopped
27 // - movingForward
28 // - movingBackward
29 // - turningLeft
30 // - turningRight
31 Event robotMovement : robotMovement(VALUE)
32
33 // Like the Event. Used to turn the model-controller into an
34 //   event-driven component
```

```

30 Dispatch msg_obstacleDetected : obstacleDetected(VALUE)
31
32 // Like the Event. Used to turn the model-controller into an
   event-driven component
33 Dispatch msg_robotMovement : robotMovement(VALUE)
34
35
36
37 // se si usa con il robot fisico al posto di localhost
   bisogna mettere l'indirizzo del pc
38 Context ctxResourceModel ip[host="localhost" port=8099]
39
40
41 // Mappers from Event to Dispatch. Used to turn the model
   into an event-driven component
42 EventHandler evt_obstacle for obstacleDetected {//-pubsub {
43     forwardEvent resource_model_robot -m msg_obstacleDetected
44 };
45 EventHandler evt_robot for robotMovement {//-pubsub {
46     forwardEvent resource_model_robot -m msg_robotMovement
47 };
48
49 EventHandler logger for modelChanged -print;
50
51 QActor resource_model_led context ctxResourceModel {
52
53     Plan init normal [
54         println("resource_model STARTED");
55         demo consult("resourceModel.pl")
56     ]
57     switchTo waitForInputs
58
59     Plan waitForInputs []
60     transition
61         stopAfter 1000000
62         whenEvent turnLed -> handleTurnLed
63     finally repeatPlan
64
65     Plan handleTurnLed resumeLastPlan [
66         onEvent turnLed : turnLed(NAME, STATE) ->
67             demo changeModelItem(NAME, turnLed(STATE))
68     ]
69
70 }
71
72 QActor resource_model_time context ctxResourceModel {
73
74     Rules {
75         minTime(7, 0, 0).
76         maxTime(9, 41, 0).

```

```

77
78     changedModelAction(resource(name(timer), state(
currentTime(CUR_H, CUR_M, CUR_S)))) :-
79         minTime(MIN_H, MIN_M, MIN_S),
80         maxTime(MAX_H, MAX_M, MAX_S),
81         sec_tot(ST_CUR, CUR_H, CUR_M, CUR_S),
82         sec_tot(ST_MIN, MIN_H, MIN_M, MIN_S),
83         sec_tot(ST_MAX, MAX_H, MAX_M, MAX_S),
84         eval(ge, ST_CUR, ST_MIN),
85         eval(ge, ST_MAX, ST_CUR),
86         !,
87         changeModelItem(timeIsOk, true).
88
89     changedModelAction(resource(name(timer), state(
currentTime(CUR_H, CUR_M, CUR_S)))) :-
90         changeModelItem(timeIsOk, false).
91 }
92
93 Plan init normal [
94     println("resource_model STARTED");
95     demo consult("resourceModel.pl")
96 ]
97 switchTo waitForInputs
98
99 Plan waitForInputs []
100 transition
101     stopAfter 1000000
102     whenEvent updateTime -> handleUpdateTime
103 finally repeatPlan
104
105 Plan handleUpdateTime resumeLastPlan [
106     onEvent updateTime : updateTime(timer, STATE) ->
107         demo changeModelItem(timer, updateTime(STATE))
108 ]
109
110 }
111
112 QActor resource_model_temperature context ctxResourceModel {
113
114     Rules {
115         limitTemperatureValue(25).
116
117         changedModelAction(resource(name(temp), state(temperature
(VALUE)))) :-
118             limitTemperatureValue(MAX),
119             eval(ge, MAX, VALUE), !,
120             changeModelItem(temperatureIsOk, true).
121
122         changedModelAction(resource(name(temp), state(temperature
(VALUE)))) :-

```

```

123         changeModelItem(temperatureIsOk, false).
124     }
125
126     Plan init normal [
127         println("resource_model STARTED");
128         demo consult("resourceModel.pl")
129     ]
130     switchTo waitForInputs
131
132     Plan waitForInputs []
133     transition
134         stopAfter 1000000
135         whenEvent updateTemperature -> handleUpdateTemperature
136     finally repeatPlan
137
138     Plan handleUpdateTemperature resumeLastPlan [
139         onEvent updateTemperature : updateTemperature(temp, STATE
140         ) ->
141             demo changeModelItem(temp, updateTemperature(STATE))
142     ]
143 }
144
145
146 QActor resource_model_robot context ctxResourceModel {//-
147     pubsub {
148
149     Rules {
150         // It is needed to stop the chain of changedModelAction
151         changedModelAction(resource(name(robot), state(movement(
152         stopped), obstacleDetected(true)))) :-
153         !.
154         // When an obstacle is sensed, stop the robot
155         changedModelAction(resource(name(robot), state(X,
156         obstacleDetected(true)))) :-
157             changeModelItem(robot, movement(stopped)).
158     }
159
160     Plan init normal [
161         println("resource_model_robot STARTS");
162         demo consult("resourceModel.pl")
163     ]
164     switchTo waitMsgs
165
166     Plan waitMsgs []
167     transition
168         stopAfter 1000000
169         whenMsg msg_obstacleDetected -> handleObstacle,
170         whenMsg msg_robotMovement -> handleRobot
171     finally repeatPlan

```

```

169
170 Plan handleObstacle resumeLastPlan [
171     onMsg msg_obstacleDetected : obstacleDetected(VALUE) ->
172         demo changeModelItem(robot, obstacleDetected(VALUE))
173 ]
174
175 Plan handleRobot resumeLastPlan [
176     onMsg msg_robotMovement : robotMovement(VALUE) ->
177         demo changeModelItem(robot, movement(VALUE))
178 ]
179 }
180
181
182 QActor resource_model_sonar context ctxResourceModel {
183     Plan init normal [
184         println("resource_model_sonar STARTS");
185         demo consult("resourceModel.pl")
186     ]
187     switchTo waitForInputs
188
189     Plan waitForInputs []
190     transition
191         stopAfter 6000000
192         whenEvent sonarDetected -> handleSonar
193     finally repeatPlan
194
195     Plan handleSonar resumeLastPlan [
196         onEvent sonarDetected : sonarDetected(name(NAME),
197             somethingDetected(VALUE), distance(DVALUE)) ->
198             demo changeModelItem(NAME, state(somethingDetected(
199                 VALUE), distance(DVALUE)))
200     ]
201 }

```

Listato

1.5.

../src/sprint4/design/it.unibo.finaltask2018.design/src/resourceModel.qa

```

1 System robot
2
3 Event turnLed          : turnLed(NAME, NEW_STATE)
4
5 Event modelChanged     : modelChanged(resource(NAME, STATE))
6
7 Event local_BlinkOn    : blinkOn
8 Event local_BlinkOff   : blinkOff
9
10 // se si usa con il robot fisico al posto di localhost
    bisogna mettere l'indirizzo del pc

```

```

11 Context ctxResourceModel ip[host="localhost" port=8099] -
    standalone
12 Context ctxApplicationLogic ip[host="localhost" port=8097]
13
14
15 QActor robot_movement_finder context ctxApplicationLogic {
16     Plan init normal [
17         println("robot_movement_finder STARTED")
18     ]
19     switchTo waitForModelChanged
20
21     Plan waitForModelChanged []
22     transition
23         stopAfter 1000000
24         whenEvent modelChanged -> applLogic
25     finally
26         repeatPlan
27
28     Plan applLogic resumeLastPlan [
29         onEvent modelChanged : modelChanged(resource(name(robot),
30             state(movement(stopped), X))) ->
31             emit local_BlinkOff : blinkOff;
32
33         onEvent modelChanged : modelChanged(resource(name(robot),
34             state(movement(movingForward), X))) ->
35             emit local_BlinkOn : blinkOn;
36
37         onEvent modelChanged : modelChanged(resource(name(robot),
38             state(movement(movingBackward), X))) ->
39             emit local_BlinkOn : blinkOn;
40
41         onEvent modelChanged : modelChanged(resource(name(robot),
42             state(movement(movingForward), obstacleDetected(true))))
43             ->
44             emit local_BlinkOff : blinkOff
45     ]
46 }
47
48 QActor blink_controller context ctxApplicationLogic {
49     // the behaviour is the same for the real led and for the
50     hue lamp
51
52     Rules {
53         // rules needed by the application logic
54         ledName(led).
55     }
56
57     Plan init normal [
58         println("blink_controller STARTED")
59     ]
60 }

```

```

54 ]
55 switchTo ledOff
56
57 Plan ledOff [
58     println("Stato: ledOff");
59     [ !? ledName(NAME) ]
60     emit turnLed : turnLed(NAME, off)
61 ]
62 transition
63     stopAfter 6000000
64     whenEvent local_BlinkOn -> ledBlinkingOn
65
66 Plan ledBlinkingOn [
67     println("Stato: ledBlinking on");
68     [ !? ledName(NAME) ]
69     emit turnLed : turnLed(NAME, on)
70 ]
71 transition
72     whenTime 500 -> ledBlinkingOff,
73     whenEvent local_BlinkOff -> ledOff
74
75 Plan ledBlinkingOff [
76     println("Stato: ledBlinking off");
77     [ !? ledName(NAME) ]
78     emit turnLed : turnLed(NAME, off)
79 ]
80 transition
81     whenTime 500 -> ledBlinkingOn,
82     whenEvent local_BlinkOff -> ledOff
83
84 }

```

Listato

1.6.

../src/sprint5/design/it.unibo.finaltask2018.design/src/applicationLogic.qa

```

1 System robot
2
3 Event modelChanged : modelChanged(resource(NAME, STATE))
4 Event timeCheck : timeCheck(SEC_TOT, HOURS, MINS, SECS)
5
6 // Dispatch used to turn the pfrs robot into an event-driven
7 // component
8 Dispatch msg_modelChanged : modelChanged(resource(NAME, STATE))
9
10 // il contesto del raspberry. bisogna mettere l'ip giusto.
11 // a logica non dovrebbe essere qui, dovrebbe essere il
12 // raspberry che "si collega"
13 // al resource model come abbiamo qui sotto.

```



```

12 // facendo cos invece non bisogna modifica ogni volta l'ip
    nel jar nel raspberry
13 // commentando la riga si esclude il robot reale
14 //Context ctxRealRobotAdapter ip[host="192.168.1.4" port
    =9010] -standalone
15 // questa invece per il led sul robot reale
16 //Context ctxRealLedAdapter ip[host="192.168.1.4" port=9011]
    -standalone
17
18 // se si usa con il robot fisico al posto di localhost
    bisogna mettere l'indirizzo del pc
19 Context ctxResourceModel ip[host="localhost" port=8099] -
    standalone
20 Context ctxOutput ip[host="localhost" port=8098]
21
22 // It turns the pfrs robot into an event-driven component
23 EventHandler pfrs_event_driven for modelChanged {
24     forwardEvent adapter_to_pfrs_mbot -m msg_modelChanged
25 };
26
27
28 QActor mock_output_led context ctxOutput {
29
30     Rules {
31         // rules needed by the application logic
32         ledName(led).
33     }
34
35
36     Plan init normal [
37         println("resource_representation_element STARTED");
38         javaRun it.unibo.custom.gui.customBlsGui.
            createCustomLedGui();
39         javaRun it.unibo.custom.gui.customBlsGui.setLed("off")
40     ]
41     switchTo waitForModelChanged
42
43     Plan waitForModelChanged []
44     transition
45         stopAfter 1000000
46         whenEvent modelChanged -> outputtingData
47     finally
48         repeatPlan
49
50     Plan outputtingData resumeLastPlan [
51         [ !? ledName(NAME) ]
52         onEvent modelChanged : modelChanged(resource(name(NAME)
            , state(on))) ->
53             javaRun it.unibo.custom.gui.customBlsGui.setLed("on")
54     ];

```

```

54     [ !? ledName(NAME) ]
55     onEvent modelChanged : modelChanged(resource(name(NAME)
56     , state(off))) ->
57         javaRun it.unibo.custom.gui.customBlsGui.setLed("off"
58     )
59 ]
60 }
61
62 QActor mock_output_temperature context ctxOutput -g green {
63
64     Plan init normal [
65         println("Temperature Observer STARTED")
66     ]
67     switchTo waitForEvents
68
69     Plan waitForEvents []
70     transition
71         stopAfter 1000000
72         whenEvent modelChanged -> handleModelChanged
73     finally repeatPlan
74
75     Plan handleModelChanged resumeLastPlan [
76         onEvent modelChanged : modelChanged(resource(name(temp),
77         state(temperature(VALUE)))) ->
78         println(temp(VALUE))
79     ]
80 }
81
82 QActor mock_output_time context ctxOutput -g yellow {
83
84     Plan init normal [
85         println("Timer Observer STARTED");
86         demo consult("resourceModel.pl")
87     ]
88     switchTo waitForEvents
89
90     Plan waitForEvents []
91     transition
92         stopAfter 1000000
93         whenEvent modelChanged -> handleModelChanged
94     finally repeatPlan
95
96     Plan handleModelChanged resumeLastPlan [
97         onEvent modelChanged : modelChanged(resource(name(timer),
98         state(currentTime(H, M, S)))) ->
99         println(now(H, M, S))

```

```

100 }
101 }
102
103
104 // It makes pfrs robot a QActor entity.
105 QActor adapter_to_pfrs_mbot context ctxOutput {//-pubsub {
106
107     Plan init normal [
108         println("adapter_to_pfrs_mbot STARTS");
109         javaRun it.unibo.pfrs.mbotConnTcp.initClientConn()
110     ]
111     switchTo waitMsgs
112
113     Plan waitMsgs []
114     transition
115         stopAfter 1000000
116         whenMsg msg_modelChanged -> moveRobot
117     finally repeatPlan
118
119
120     Plan moveRobot resumeLastPlan [
121         onMsg msg_modelChanged : modelChanged(resource(name(robot)
122             ), state(movement(stopped), X))) ->
123             javaRun it.unibo.pfrs.mbotConnTcp.mbotStop();
124         onMsg msg_modelChanged : modelChanged(resource(name(robot)
125             ), state(movement(movingForward), obstacleDetected(false)
126             ))) ->
127             javaRun it.unibo.pfrs.mbotConnTcp.mbotForward();
128         onMsg msg_modelChanged : modelChanged(resource(name(robot)
129             ), state(movement(movingBackward), X))) ->
130             javaRun it.unibo.pfrs.mbotConnTcp.mbotBackward();
131         onMsg msg_modelChanged : modelChanged(resource(name(robot)
132             ), state(movement(turningLeft), X))) ->
133             javaRun it.unibo.pfrs.mbotConnTcp.mbotLeft();
134         onMsg msg_modelChanged : modelChanged(resource(name(robot)
135             ), state(movement(turningRight), X))) ->
136             javaRun it.unibo.pfrs.mbotConnTcp.mbotRight()
137     ]
138 }
139 //

```

Listato 1.7. ../src/sprint5/design/it.unibo.finaltask2018.design/src/output.qa

```

1 System robot
2
3 Event updateTemperature : updateTemperature(NAME, NEW_TEMP)
4 Event updateTime      : updateTime(NAME, CURRENT_TIME)
5
6 // It's only used for the input_element

```

```

7  Event modelChanged      : modelChanged(resource(NAME, STATE))
8
9  // Emitted when someone/thing modify the movement state of
   the robot
10 // VALUE can be:
11 // - stopped
12 // - movingForward
13 // - movingBackward
14 // - turningLeft
15 // - turningRight
16 Event robotMovement : robotMovement(VALUE)
17
18 // se si usa con il robot fisico al posto di localhost
   bisogna mettere l'indirizzo del pc
19 Context ctxResourceModel ip[host="localhost" port=8099] -
   standalone
20 Context ctxInput ip[host="localhost" port=8096]
21
22 QActor temperature_sensor_adapter context ctxInput {
23
24     Plan init normal [
25         println("resource_model STARTED");
26         delay 500
27     ]
28     switchTo sendEvents
29
30     Plan sendEvents [
31         delay 1000;
32         javaRun it.unibo.temperature_sensor_adapter.
           webTemperatureSensorAdapter.updateTemperature()
33         //emit updateTemperature : updateTemperature(temp, 12)
34     ]
35     finally repeatPlan
36
37 }
38
39 QActor timer_adapter context ctxInput {
40
41     Plan init normal [
42         println("resource_model STARTED")
43     ]
44     switchTo sendEvents
45
46     Plan sendEvents [
47         delay 1000;
48         javaRun it.unibo.timer_adapter.systemTimerAdapter.
           updateTime()
49         //emit updateTime : updateTime(timer, currentTime(35000))
50     ]
51     finally repeatPlan

```

```

52 }
53
54
55
56 // It simulates the robot movement
57 QActor input_element context ctxInput {
58
59     Plan init normal [
60         println("input_element STARTED")
61     ]
62     switchTo working
63
64     Plan working [
65         // interact with the implementation of the specific input
66         // element and emit the data to modify the resourceModel
67         // delay 450;
68         // println("Now the robot is moving");
69         // emit modelChanged : modelChanged(resource(name(robot),
70         // state(movement(movingForward), obstacleDetected(false))))
71         // ;
72         // delay 2350;
73         // println("Now the robot is stopped");
74         // emit modelChanged : modelChanged(resource(name(robot),
75         // state(movement(stopped), obstacleDetected(false))));
76
77         delay 2450;
78         emit robotMovement : robotMovement(movingForward);
79         println("Now the robot is moving");
80         // delay 1000;
81         // emit robotMovement : robotMovement(turningLeft);
82         // delay 1000;
83         // emit robotMovement : robotMovement(movingForward);
84         // delay 2350;
85         // println("Now the robot is stopped");
86         // emit modelChanged : modelChanged(resource(name(robot),
87         // state(movement(stopped), obstacleDetected(false))));
88
89         delay 4000
90     ]
91     //finally repeatPlan
92 }

```

Listato 1.8. ../src/sprint5/design/it.unibo.finaltask2018.design/src/input.qa

```

1 System robot
2
3 // Emitted PROLOG-side when the model is changed
4 // It makes the model observable
5 Event modelChanged : modelChanged(resource(NAME, STATE))

```

```

6
7 // Dispatch used to turn the real robot into an event-driven
  component
8 Dispatch msg_modelChanged : modelChanged(resource(NAME, STATE
  ))
9
10
11 Context ctxRealRobotAdapter ip[host="localhost" port=9010]
12
13 // It turns the real robot into an event-driven component
14 EventHandler evt_modelchanged for modelChanged {
15     forwardEvent adapter_to_physical_mbot -m msg_modelChanged
16 };
17
18
19
20 QActor adapter_to_physical_mbot context ctxRealRobotAdapter {
    //-pubsub {
21
22     Rules {
23
24     }
25
26     Plan init normal [
27 //         javaRun it.unibo.myArduinoUtils.connArduino.initPc("
          COM6", "115200");
28         javaRun it.unibo.myArduinoUtils.connArduino.initRasp("
          115200");
29         println("adapter_to_physical_mbot STARTS")
30     ]
31     switchTo waitMsgs
32
33     Plan waitMsgs []
34     transition
35         stopAfter 1000000
36         whenMsg msg_modelChanged -> moveRobot
37     finally repeatPlan
38
39
40     Plan moveRobot resumeLastPlan [
41         onMsg msg_modelChanged : modelChanged(resource(name(robot
          ), state(movement(stopped), X))) ->
42             javaRun it.unibo.myArduinoUtils.connArduino.mbotStop();
43         onMsg msg_modelChanged : modelChanged(resource(name(robot
          ), state(movement(movingForward), obstacleDetected(false)
          ))) ->
44             javaRun it.unibo.myArduinoUtils.connArduino.mbotForward
          ();
45         onMsg msg_modelChanged : modelChanged(resource(name(robot
          ), state(movement(movingBackward), X))) ->

```

```

46         javaRun it.unibo.myArduinoUtils.connArduino.
mBotBackward();
47         onMsg msg_modelChanged : modelChanged(resource(name(robot
), state(movement(turningLeft), X))) -> {
48             javaRun it.unibo.myArduinoUtils.connArduino.mBotLeft
();
49             delay 600; //test needed
50             javaRun it.unibo.myArduinoUtils.connArduino.mBotStop
();
51         };
52         onMsg msg_modelChanged : modelChanged(resource(name(robot
), state(movement(turningRight), X))) -> {
53             javaRun it.unibo.myArduinoUtils.connArduino.mBotRight
();
54             delay 600; //test needed
55             javaRun it.unibo.myArduinoUtils.connArduino.mBotStop
();
56         }
57     ]
58 }
59 }

```

Listato

1.9.

../src/sprint5/design/it.unibo.finaltask2018.design/src/real_robot_adapter.qa

```

1 System robot
2
3 Event modelChanged : modelChanged(resource(NAME, STATE))
4
5 Context ctxRealLedAdapter ip[host="localhost" port=9011]
6
7 QActor real_led context ctxRealLedAdapter {
8
9     Rules {
10         // rules needed by the application logic
11         ledName(led).
12     }
13
14
15     Plan init normal [
16         println("resource_representation_element STARTED")
17     ]
18     switchTo waitForModelChanged
19
20     Plan waitForModelChanged []
21     transition
22         stopAfter 1000000
23         whenEvent modelChanged -> outputingData
24     finally
25         repeatPlan

```

```

26
27 Plan outputtingData resumeLastPlan [
28   [ !? ledName(NAME) ]
29   onEvent modelChanged : modelChanged(resource(name(NAME)
, state(on))) ->
30     javaRun it.unibo.myUtils.executor.execBash("./
led28GpioTurnOn.sh");
31
32   [ !? ledName(NAME) ]
33   onEvent modelChanged : modelChanged(resource(name(NAME)
, state(off))) ->
34     javaRun it.unibo.myUtils.executor.execBash("./
led28GpioTurnOff.sh")
35 ]
36
37 }

```




Listato

1.10.

../src/sprint5/design/it.unibo.finaltask2018.design/src/real_led_adapter.qa

7 Implementazione

8 Autori

Foto degli autori		
		
Luca Bonfiglioli	Nicola Fava	Antonio Grasso