

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	12
6 Progettazione	21
7 Implementazione	45
8 Autori	48

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`.

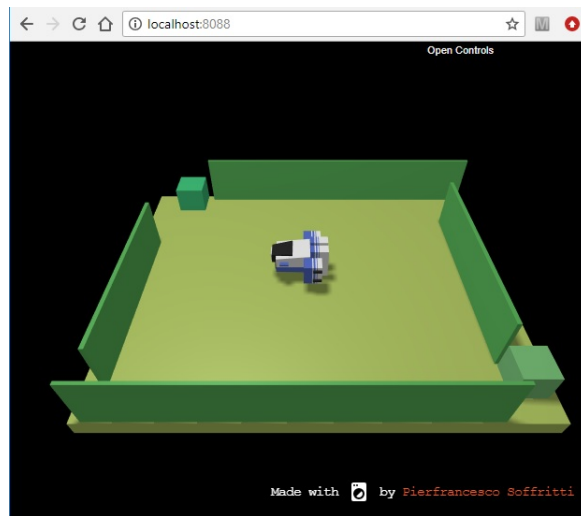


Fig. 1. Esempio di pavimento con il robot in ambiente simulato

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 processo di produzione del software è stato impostato secondo il framework di sviluppo agile **scrum**. La collaborazione interna tra i membri del team ed esterna tra il team e il committente è stata al centro del processo. Il lavoro è stato suddiviso in cicli orientati alla realizzazione di prototipi funzionanti, incrementalmente estendibili lungo due direzioni:

- una direzione riguardante l'aggiunta incrementale dei requisiti;
- una direzione relativa al passaggio dalla logica definita in fase di analisi alle tecnologie scelte in progettazione e implementazione.

Il lato delle celle dovrà essere di lunghezza non superiore al lato di dimensione maggiore del robot. Occorre quindi introdurre il concetto di **basic step**, ovvero un movimento che copra la distanza pari al lato della cella. Un basic step ha successo se il robot riesce ad avanzare nella cella successiva, mentre fallisce se la cella successiva è occupata da un ostacolo fisso. Al termine di un basic step la cella in cui il robot si trova è da considerarsi pulita. Qualsiasi percorso del robot dovrà essere espresso come una sequenza di basic step e di rotazioni di 90° . Fatte queste premesse, pulire tutta la stanza equivale a pulire ogni cella della stanza non occupata da un ostacolo fisso. Come da requisito **R-Map**, se è già stata costruita una mappa della stanza, il robot segue un percorso predefinito dallo **start-point** all'**end-point**, altrimenti procede nella pulizia della stanza costruendone la mappa.

Il sistema da modellare è **eterogeneo e distribuito**, in particolare composto da **almeno due nodi**: il nodo "Robot" e il nodo "PC/Android". Trattandosi di un sistema eterogeneo distribuito, i componenti all'interno del sistema possono interagire tramite:

- **messaggi**: informazioni che il mittente invia ad uno **specifico destinatario**;
- **eventi**: informazioni **senza specifico destinatario** che possono essere ricevute da tutti i componenti del sistema interessati agli eventi stessi.

In particolare, le entità coinvolte nelle interazioni possono essere distinte in:

- **message/event-based**: l'entità stabilisce se e come gestire i messaggi e gli eventi. Nel caso di entità message-based ci può essere la possibilità che l'entità non riceva immediatamente il messaggio, che viene messo in una coda prima di essere recapitato. Nel caso invece di entità event-based è possibile che l'entità possa perdere gli eventi.
- **message/event-driven**: l'entità è guidata nel suo comportamento indipendentemente dalla sua volontà.

Per la modellazione è possibile utilizzare il linguaggio *QActor*, che risulta adatto alla modellazione di sistemi distribuiti eterogenei.

Il primo dei due nodi ad essere modellato 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 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

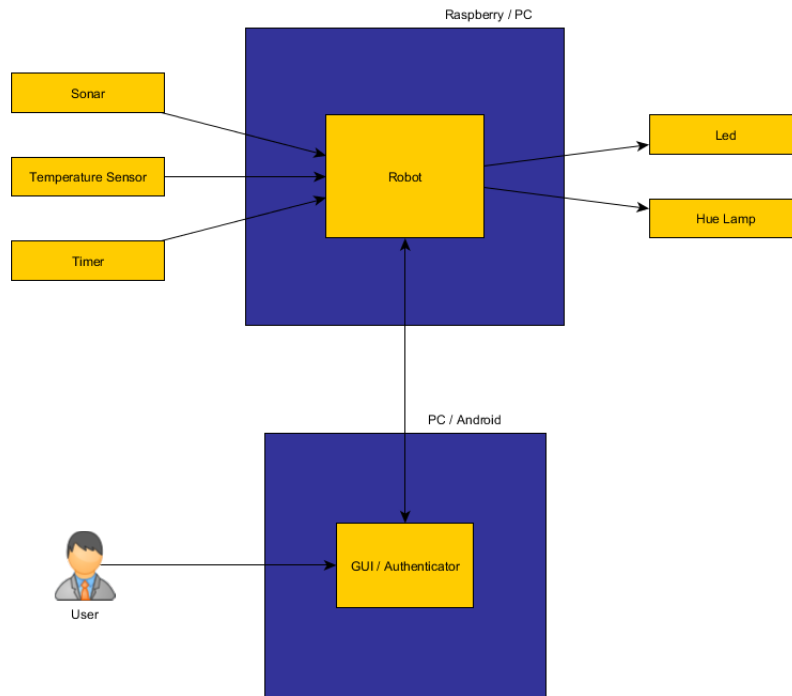


Fig. 3. Diagramma informale dell'analisi dei requisiti

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.

Analizzando i requisiti emerge la necessità di interazione tra:

- il robot e i sonar;
- il robot e le sorgenti di tempo e temperatura;
- il robot e i dispositivi attuatori, che da requisiti vengono a coincidere con il led e la 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.

Per quanto riguarda l'interazione user-GUI non si evidenziano particolari differenze nel modellarla tramite messaggi piuttosto che tramite eventi. In merito all'interazione gui-robot risulta invece più opportuno che venga modellata mediante eventi piuttosto che messaggi, in quanto risultano più vantaggiosi per disaccoppiare l'interfaccia grafica dallo specifico robot comandato.

Vengono di seguito riportati i modelli formali risultanti dall'analisi dei requisiti che evidenziano una prima **architettura logica**:

```

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
32         receives event from temperature sensor");
33     onEvent sensorEvent : sensorEvent(timer) -> println("
34         Robot receives event from timer sensor");
35     printCurrentEvent
36 ]
37 }
38
39 QActor sonarsensor1 context ctxRobot {
40     Plan initial normal[
41         println("Sonar1 started");
42         delay 3000
43     ]
44     switchTo emitEvents
45
46     Plan emitEvents[
47         emit sensorEvent : sensorEvent(sonar1)
48     ]
49 }
50
51 QActor sonarsensor2 context ctxRobot {
52     Plan initial normal[
53         println("Sonar2 started");
54         delay 3500
55     ]
56     switchTo emitEvents
57
58     Plan emitEvents[
59         emit sensorEvent : sensorEvent(sonar2)
60     ]
61 }
62
63 QActor temperaturesensor context ctxRobot {
64     Plan initial normal[
65         println("Temperature sensor started");
66         delay 4000
67     ]
68     switchTo emitEvents
69
70     Plan emitEvents[
71         emit sensorEvent : sensorEvent(temp)
72     ]
73 }
74
75 QActor timersensor context ctxRobot {
76     Plan initial normal[
77         println("Timer sensor started");
78         delay 4500
79     ]
80     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 }

```

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
28 QActor user context ctxUser {
29     Plan initial normal [
30         println("User started")
31     ]
32     switchTo sendMsg
33
34     Plan sendMsg[
35         println("User send messages");
36         forward gui -m userCmd : userCmd(X);
37         forward gui -m userCmd : userCmd(Y)
38     ]
39 }

```

reqAnalysisUser.qa

5 Analisi del problema

Tenendo conto delle assunzioni e delle considerazioni già affrontate in analisi dei requisiti, il primo problema è posto dal requisito [R-FloorClean](#).

Tale problema consiste nello stabilire una sequenza di movimenti che portino il robot a pulire il massimo numero di celle in cui è suddivisa la stanza. Nella stanza possono essere presenti degli ostacoli di varie forme e disallineati rispetto alla griglia. In quest'ultimo caso alcune celle potrebbero essere coperte solo parzialmente da degli ostacoli. Per semplicità, si considerano tali celle come totalmente coperte dagli ostacoli. Un esempio è riportato in figura 4.

Gli ostacoli presenti nella stanza possono essere classificati in fissi e mobili:

- gli ostacoli fissi occupano le stesse celle per una durata indefinita;

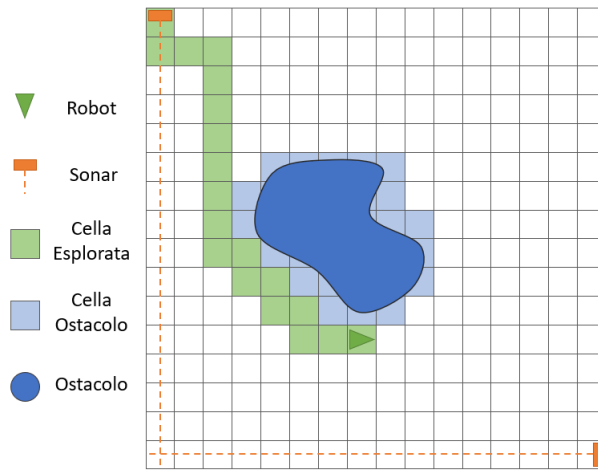


Fig. 4. Esempio di ostacolo fisso nella stanza

- gli ostacoli mobili si spostano dalla cella precedentemente occupata in un'altra cella allo scadere di un dato intervallo di tempo.

Il robot deve essere in grado di pulire le celle momentaneamente occupate da ostacoli mobili.

Nel caso in cui vi siano celle non occupate ma irraggiungibili, tali celle vengono considerate come ostacoli fissi.

Per pulire la stanza in autonomia, il robot ha accesso a tre tipi di informazione:

- la prima è fornita dal sonar montato sulla parte anteriore del robot, in grado di rilevare un ostacolo immediatamente davanti al robot;
- la seconda riguarda invece le distanze dal **sonar1** e dal **sonar2**, posizionati come in figura 1;
- la terza riguarda i dati memorizzati durante l'esplorazione della stanza (ad esempio la mappa della stanza).

Per semplicità, si assume che non ci siano ostacoli nel raggio di azione dei due sonar presenti nella stanza.

L'unico modo che ha il robot per rilevare la presenza di un ostacolo è tentare di effettuare un basic step nella direzione desiderata e mettersi in ascolto dell'evento emesso dal sonar anteriore.

Quando il robot è in movimento i requisiti **R-BlinkLed** e **R-BlinkHue** pongono il problema di far lampeggiare un dispositivo attuatore, che coincide con un **Led** nel caso di **real** robot e con una **Led Hue Lamp** nel caso di **virtual** robot.

L'interazione robot-dispositivo attuatore può essere modellata tramite messaggi o tramite eventi: l'utilizzo degli eventi per la gestione del blinking consente un maggiore disaccoppiamento tra il robot e i possibili dispositivi attuatori che

vengono utilizzati. Occorre inoltre distinguere due possibilità di configurazione del comportamento:

- la prima prevede che gli eventi di gestione del blinking relativi al **real** e al **virtual** robot vengano gestiti dallo stesso attore. In tal caso possono emergere problemi nel caso si vogliano comportamenti differenti dei dispositivi attuatori a seconda del tipo di robot che è in movimento;
- la seconda prevede che gli eventi di gestione del blinking vengano gestiti da due attori differenti a seconda del tipo di robot che è in movimento. Questa seconda opzione garantisce maggiore disaccoppiamento tra il tipo di robot e i dispositivi attuatori che devono lampeggiare.

Per poter entrare in esecuzione, il robot deve ricevere un comando **START** da un utente autorizzato. Sorge dunque il problema di stabilire quale tra i due nodi, riportati in figura 3, 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 relativo all'interfaccia **GUI** che l'utente utilizza per interagire con il robot. Questa interfaccia 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. Usando l'approccio **event-based**, il robot potrebbe non essere sensibile a tutti gli eventi, potendone perdere alcuni. Al contrario, utilizzando l'approccio **event-driven**, il robot sarebbe sempre sensibile agli eventi, perdendo tuttavia reattività.

I requisiti **R-TempOk** e **R-TimeOk** pongono invece il problema della comunicazione tra le sorgenti dei dati di temperatura e tempo e il robot. Le sorgenti di dati possono essere modellate come entità che emettono eventi a ripetizione aggiornando i valori di tempo e temperatura correnti. In tal caso si rende necessaria la gestione degli eventi di aggiornamento di tempo e temperatura: se la sensibilità ai cambiamenti di questi valori rappresenta un requisito fondamentale risulta più opportuno applicare un approccio **event-driven**; se al contrario la reattività del robot è considerata un requisito essenziale, risulta migliore l'approccio **event-based**, che comporterebbe la possibilità di perdere alcuni eventi.

Vengono di seguito riportati i modelli formali prodotti dall'analisi del problema, in cui viene delineata l'**architettura logica** del sistema risultante dalla fase di analisi:

```

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

```

```

47     onEvent robotCmd : robotCmd(cmdstart) -> {
48         [ !? startRequirementsOk ] {
49             println("Robot start");
50             emit outCmd : outCmd(startblinking);
51             [ !? map ]
52                 println("Robot cleans room (following optimal path)
53 ");
54             forward robotmind -m robotMindCmd : robotMindCmd(
55 clean)
56             else
57                 println("Robot builds room map");
58                 forward robotmind -m robotMindCmd : robotMindCmd(
59 explore)
60             }
61         };
62     onEvent robotCmd : robotCmd(cmdstop) -> {
63         println("Robot stop from user");
64         emit outCmd : outCmd(stopblinking)
65     };
66     onEvent sensorEvent : sensorEvent(temp(VALUE)) ->
67     ReplaceRule currentTempValue(X) with currentTempValue(
68 VALUE);
69     onEvent sensorEvent : sensorEvent(temp(X)) -> {
70         [ not !? evalTemp ] {
71             println("Robot stop from temperature sensor");
72             emit outCmd : outCmd(stopblinking)
73         }
74     };
75     onEvent sensorEvent : sensorEvent(timer(VALUE)) ->
76     ReplaceRule currentTimeValue(X) with currentTimeValue(
77 VALUE);
78     onEvent sensorEvent : sensorEvent(timer(X)) -> {
79         [ not !? evalTime ] {
80             println("Robot stop from time sensor");
81             emit outCmd : outCmd(stopblinking)
82         }
83     };
84     printCurrentEvent
85 ]
86 }
87
88 QActor robotmind context ctxProbRobot {
89     Rules{
90         // size: index of last X cell and Y cell
91         dimX(0).
92         dimY(0).
93         incrementX:- dimX(X), retract(dimX(_)), X1 is X + 1,
94         assert(dimX(X1)).
95         incrementY:- dimY(Y), retract(dimY(_)), Y1 is Y + 1,
96         assert(dimY(Y1)).

```



```

88     // robot direction
89     dir(Y).
90 }
91 Plan initial normal [
92     println("Robotmind started")
93 ]
94 switchTo waitForMessage
95
96 Plan waitForMessage [
97 ]
98 transition stopAfter 600000
99     whenMsg robotMindCmd -> handleMovement,
100     whenEvent sensorEvent -> handleMovement
101 finally repeatPlan
102
103 // ASSUMPTION: there are no obstacles in the room
104 Plan handleMovement [
105     onMsg robotMindCmd : robotMindCmd(explore) -> {
106         // doBasicStep() static method of robot class that must
107         // listen to events sensorEvent : sensorEvent(onboardsonar)
108         javaRun robot.doBasicStep();
109         // robot starts from start-point towards the bottom-
110         // side, it changes its direction when it receives
111         // the event sensorEvent : sensorEvent(sonar2) and it
112         // goes forward until it hits the second sonar
113         [?? basicStepResult(true)] {
114             [!? dir(X)]
115             demo incrementX
116         else
117             demo incrementY;
118         selfMsg robotMindCmd : robotMindCmd(explore)
119     }
120     else {
121         demo assert(map);
122         [!? dimX(X)]
123         javaRun planner.setSizeX(X);
124         [!? dimY(Y)]
125         javaRun planner.setSizeY(Y);
126         selfMsg robotMindCmd : robotMindCmd(clean)
127     }
128 ];
129
130 onEvent sensorEvent : sensorEvent(sonar2) -> {
131     demo retract(dir(Y));
132     demo assert(dir(X))
133 };
134
135 onMsg robotMindCmd : robotMindCmd(clean) -> {
136     // nextMove = basicStep or 90 degree rotation +
137     basicStep
138     javaRun planner.nextMove();
139     // nextMove(X,Y):

```

```

134         // X = rotation (n = none, l = left, r = right);
135         // Y = move (n = none, w = forward)
136         [!? nextMove(n,w)]
137             javaRun robot.doBasicStep();
138         [!? nextMove(l,w)] {
139             javaRun robot.turnLeft();
140             javaRun robot.doBasicStep()
141         };
142         [!? nextMove(r,w)] {
143             javaRun robot.turnRight();
144             javaRun robot.doBasicStep()
145         };
146         [?? nextMove(_,w)]
147             selfMsg robotMindCmd : robotMindCmd(clean)
148     }
149
150 ]
151 }
152
153 QActor sonarsensor1 context ctxProbRobot {
154     Plan initial normal[
155         println("Sonar1 started");
156         delay 3000
157     ]
158     switchTo emitEvents
159
160     Plan emitEvents[
161         emit sensorEvent : sensorEvent(sonar1)
162     ]
163 }
164
165 QActor sonarsensor2 context ctxProbRobot {
166     Plan initial normal[
167         println("Sonar2 started");
168         delay 3500
169     ]
170     switchTo emitEvents
171
172     Plan emitEvents[
173         emit sensorEvent : sensorEvent(sonar2)
174     ]
175 }
176
177 QActor sonarrobot context ctxProbRobot {
178     Plan initial normal[
179         println("Sonar on board started");
180         delay 5000
181     ]
182     switchTo emitEvents
183 }

```

```

184     Plan emitEvents[
185         emit sensorEvent : sensorEvent(onboardsonar)
186     ]
187 }
188
189 QActor temperaturesensor context ctxProbRobot {
190     Plan initial normal[
191         println("Temperature sensor started");
192         delay 4000
193     ]
194     switchTo emitEvents
195
196     Plan emitEvents[
197         emit sensorEvent : sensorEvent(temp(20));
198         delay 2000;
199         emit sensorEvent : sensorEvent(temp(30))
200     ]
201 }
202
203 QActor timersensor context ctxProbRobot {
204     Plan initial normal[
205         println("Timer sensor started");
206         delay 4500
207     ]
208     switchTo emitEvents
209
210     Plan emitEvents[
211         emit sensorEvent : sensorEvent(timer(9));
212         delay 2000;
213         emit sensorEvent : sensorEvent(timer(12))
214     ]
215 }
216
217 QActor led context ctxProbRobot {
218     Plan initial normal[
219         println("Led started")
220     ]
221     switchTo waitForEvent
222
223     Plan waitForEvent[]
224     transition stopAfter 600000
225     whenEvent outCmd -> handleEvent
226     finally repeatPlan
227
228     Plan handleEvent resumeLastPlan [
229         onEvent outCmd : outCmd(startblinking) -> println("Led
230 start blinking");
231         onEvent outCmd : outCmd(stopblinking) -> println("Led
232 stop blinking")
233     ]

```

```

232 }
233
234 QActor hueLamp context ctxProbRobot {
235   Plan initial normal [
236     println("Hue Lamp started")
237   ]
238   switchTo waitForEvent
239
240   Plan waitForEvent []
241   transition stopAfter 600000
242     whenEvent outCmd -> handleEvent
243   finally repeatPlan
244
245   Plan handleEvent resumeLastPlan [
246     onEvent outCmd : outCmd(startblinking) -> println("Hue
247     Lamp start blinking");
247     onEvent outCmd : outCmd(stopblinking) -> println("Hue
248     Lamp stop blinking")
248   ]
249 }

```

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 }

```

probAnalysisUser.qa

6 Progettazione

Avendo a disposizione l'architettura logica individuata in fase di analisi, risulta possibile impostare la progettazione del sistema che risolve il problema definito dai requisiti.

Il sistema può essere visto come il risultato dell'interazione e del comportamento di entità modellabili come risorse, quali i robot, i sensori di tempo e temperatura, i dispositivi attuatori (led e lampada Hue) e i sonar.

Per questo motivo, si sceglie di adottare un'architettura di progetto esagonale con il modello delle risorse al centro, in cui ogni risorsa è modellata con un nome e uno stato. Oltre al modello delle risorse si rende necessario modellare la logica applicativa, che agisce come observer del modello delle risorse: cambiamenti dello stato di una risorsa si ripercuotono sulla logica applicativa, che va a modificare conseguentemente lo stato della risorsa nel modello delle risorse. Per l'input/output viene utilizzato un frontend server, che consente sia ad un utente umano sia a un'altra macchina di visualizzare o modificare il modello delle risorse in modo uniforme, permettendo quindi una duplice interazione human-to-machine e machine-to-machine. Questo garantisce disaccoppiamento tra il modello delle risorse e la logica applicativa, che può essere modificata facilmente senza impatti sull'esterno. Tra le risorse fisiche e il modello delle risorse si sceglie di interporre degli adapter specifici per le particolari risorse fisiche, al fine di garantire disaccoppiamento tra le risorse fisiche e la logica applicativa. Per far fronte al problema posto dal requisito [R-FloorClean](#) ci è stato messo a disposizione un

pianificatore in grado di stabilire la mossa più conveniente che il robot deve eseguire, utilizzando le informazioni pervenute al robot dall'interazione con i sonar, riguardanti le celle già pulite o occupate da ostacoli e la posizione corrente.

Resource model:

```

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
    pfrs robot, the physical robot...)
12 // VALUE is always true
13 Event obstacleDetected : obstacleDetected(VALUE)
14
15 // Emitted when A sonar senses something (unique sonars
    present are in pfrs environment)
16 // VALUE is always true
17 // DVALUE is the value of the distance of the object sensed
18 Event sonarDetected : sonarDetected(name(NAME),
    somethingDetected(VALUE), distance(DVALUE))
19
20 // Emitted when someone/thing modify the movement state of
    the robot
21 // VALUE can be:
22 // - stopped
23 // - movingForward
24 // - movingBackward
25 // - turningLeft
26 // - turningRight
27 Event robotMovement : robotMovement(VALUE)
28
29 // Like the Event. Used to turn the model-controller into an
    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, sonarDetected -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(
79             currentTime(CUR_H, CUR_M, CUR_S)))) :-
80             minTime(MIN_H, MIN_M, MIN_S),
81             maxTime(MAX_H, MAX_M, MAX_S),
82             sec_tot(ST_CUR, CUR_H, CUR_M, CUR_S),
83             sec_tot(ST_MIN, MIN_H, MIN_M, MIN_S),
84             sec_tot(ST_MAX, MAX_H, MAX_M, MAX_S),
85             eval(ge, ST_CUR, ST_MIN),
86             eval(ge, ST_MAX, ST_CUR),
87             !,

```

```

87         changeModelItem(timeIsOk, true).
88
89         changedModelAction(resource(name(timer), state(
90             currentTime(CUR_H, CUR_M, CUR_S)))) :-
91             changeModelItem(timeIsOk, false).
92     }
93
94     Plan init normal [
95         println("resource_model STARTED");
96         demo consult("resourceModel.pl")
97     ]
98     switchTo waitForInputs
99
100     Plan waitForInputs []
101     transition
102         stopAfter 1000000
103         whenEvent updateTime -> handleUpdateTime
104         finally repeatPlan
105
106     Plan handleUpdateTime resumeLastPlan [
107         onEvent updateTime : updateTime(timer, STATE) ->
108             demo changeModelItem(timer, updateTime(STATE))
109     ]
110 }
111
112 QActor resource_model_temperature context ctxResourceModel {
113
114     Rules {
115         limitTemperatureValue(25).
116
117         changedModelAction(resource(name(temp), state(temperature
118             (VALUE)))) :-
119             limitTemperatureValue(MAX),
120             eval(ge, MAX, VALUE), !,
121             changeModelItem(temperatureIsOk, true).
122
123         changedModelAction(resource(name(temp), state(temperature
124             (VALUE)))) :-
125             changeModelItem(temperatureIsOk, false).
126     }
127
128     Plan init normal [
129         println("resource_model STARTED");
130         demo consult("resourceModel.pl")
131     ]
132     switchTo waitForInputs
133
134     Plan waitForInputs []
135     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
172
173 Plan handleObstacle resumeLastPlan [
174     onMsg msg_obstacleDetected : obstacleDetected(VALUE) ->
175     demo changeModelItem(robot, obstacleDetected(VALUE))
176 ]
177
178 Plan handleRobot resumeLastPlan [
179     onMsg msg_robotMovement : robotMovement(VALUE) ->
180     demo changeModelItem(robot, movement(VALUE))
181 ]

```

```

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 }

```

resourceModel.qa

Teoria prolog:

```

1  /*
2  =====
3  resourceModel.pl
4  =====
5  */
6
7  /*
8   * Resources are modelled like:
9   * resource(name(NAME), STATE)
10  */
11
12  resource(name(robot), state(movement(stopped),
13    obstacleDetected(false))).
14
15  /*
16   * for the robot movement can be:
17   * - stopped
18   * - movingForward
19   * - movingBackward
20   * - turningLeft
21   * - turningRight
22  */

```

```

21
22 resource(name(temp), state(temperature(0))).
23 resource(name(timer), state(currentTime(0, 0, 0))).
24 /*
25  * currentTime is in seconds from midnight
26  */
27
28 resource(name(led), state(off)).
29
30 resource(name(sonar1), state(somethingDetected(false),
31   distance(0))).
32 resource(name(sonar2), state(somethingDetected(false),
33   distance(0))).
34
35 resource(name(temperatureIsOk), state(false)).
36 resource(name(timeIsOk), state(false)).
37
38
39 getResource(NAME, STATE) :-
40   resource(name(NAME), STATE).
41
42
43
44
45 changeModelItem(robot, movement(VALUE)) :-
46   resource(name(robot), state(movement(_), obstacleDetected
47     (X))),
48   commonChangeModelItem(name(robot), state(movement(VALUE),
49     obstacleDetected(X))).
50
51
52
53
54 changeModelItem(robot, obstacleDetected(VALUE)) :-
55   resource(name(robot), state(movement(X), obstacleDetected
56     (_))),
57   commonChangeModelItem(name(robot), state(movement(X),
58     obstacleDetected(VALUE))).
59
60
61
62
63 changeModelItem(NAME, turnLed(VALUE)) :-
64   commonChangeModelItem(name(NAME), state(VALUE)).
65
66
67
68
69 changeModelItem(temp, updateTemperature(VALUE)) :-
70   commonChangeModelItem(name(temp), state(temperature(VALUE)
71     )).
72
73
74
75
76 changeModelItem(timer, updateTime(currentTime(H, M, S))) :-
77   commonChangeModelItem(name(timer), state(currentTime(H, M
78     , S))).
79
80
81
82
83 changeModelItem(temperatureIsOk, STATE) :-
84   commonChangeModelItem(name(temperatureIsOk), state(STATE)
85     ).
86
87
88
89
90

```

```

62 changeModelItem(timeIsOk, STATE) :-
63     commonChangeModelItem(name(timeIsOk), state(STATE)).
64
65
66 changeModelItem(NAME, state(somethingDetected(VALUE),
67     distance(DVALUE))) :-
68     commonChangeModelItem(name(NAME), state(somethingDetected
69     (VALUE), distance(DVALUE))).
70
71 commonChangeModelItem(NAME, STATE) :-
72     (
73         retract(resource(NAME, _));
74         true
75     ),
76     assert(resource(NAME, STATE)),
77     !,
78     emit(event(modelChanged, modelChanged(resource(NAME, STATE)
79     )),
80     (
81         changedModelAction(resource(NAME, STATE))
82         ; true
83     ).
84
85 eval(ge, X, X) :- !.
86 eval(gt, X, V) :- eval(gt, X, V).
87
88 emit(event(EVID, EVCONTENT) :-
89     actorobj(Actor),
90     Actor <- emit(EVID, EVCONTENT).
91
92
93 mul(RES, A, B) :- RES is A * B.
94
95 sec_tot(SEC_TOT, HOURS, MINS, SECS) :-
96     S1 is HOURS * 3600,
97     S2 is MINS * 60,
98     S3 is S1 + S2,
99     SEC_TOT is S3 + SECS.
100
101 %%% initialize
102 initResourceTheory :- output("initializing the
103     initResourceTheory ...").
104 :- initialization(initResourceTheory).

```

Application logic:

```

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 Event userstart        : userstart(X)
11 Event userstop         : userstop(X)
12
13 Event robotMovement    : robotMovement(VALUE)
14
15 Dispatch doBasicStep   : doBasicStep
16 Dispatch doRotation    : doRotation(VALUE)
17 Dispatch foundObstacle : foundObstacle
18
19 Event basicStepResult  : basicStepResult(VALUE)
20 Event rotationResult   : rotationResult(VALUE)
21
22 Event startRobot       : startRobot
23 Event stopRobot        : stopRobot
24
25 Dispatch noMoreMoves   : noMoreMoves
26
27 // se si usa con il robot fisico al posto di localhost
28 // bisogna mettere l'indirizzo del pc
29 Context ctxResourceModel ip[host="localhost" port=8099] -
30 standalone
31 Context ctxApplicationLogic ip[host="localhost" port=8097]
32
33 EventHandler log_start for userstop -print;
34
35 QActor robot_movement_finder context ctxApplicationLogic {
36   Plan init normal [
37     println("robot_movement_finder STARTED")
38   ]
39   //switchTo waitForModelChanged
40   switchTo waitForUserStartOrStop
41
42   Plan waitForModelChanged []
43   transition
44     stopAfter 1000000
45     whenEvent modelChanged -> applLogic
46   finally
47     repeatPlan
48
49   Plan applLogic resumeLastPlan [
50     onEvent modelChanged : modelChanged(resource(name(robot),
51       state(movement(stopped), X))) ->

```

```

49     emit local_BlinkOff : blinkOff;
50
51     onEvent modelChanged : modelChanged(resource(name(robot),
52     state(movement(movingForward), X))) ->
53         emit local_BlinkOn : blinkOn;
54
55     onEvent modelChanged : modelChanged(resource(name(robot),
56     state(movement(movingBackward), X))) ->
57         emit local_BlinkOn : blinkOn;
58
59     onEvent modelChanged : modelChanged(resource(name(robot),
60     state(movement(movingForward), obstacleDetected(true))))
61     ->
62         emit local_BlinkOff : blinkOff
63 ]
64
65 Plan waitForUserStartOrStop []
66 transition
67     stopAfter 1000000
68     whenEvent userstart -> startBlinking,
69     whenEvent userstop -> stopBlinking
70 finally
71     repeatPlan
72
73 Plan startBlinking resumeLastPlan [
74     emit local_BlinkOn : blinkOn
75 ]
76
77 Plan stopBlinking resumeLastPlan [
78     emit local_BlinkOff : blinkOff
79 ]
80 }
81
82 QActor blink_controller context ctxApplicationLogic {
83     // the behavior is the same for the real led and for the
84     hue lamp
85
86     Rules {
87         // rules needed by the application logic
88         ledName(led).
89     }
90
91     Plan init normal [
92         println("blink_controller STARTED")
93     ]
94     switchTo ledOff
95
96     Plan ledOff [
97         println("Stato: ledOff");

```

```

94     [ !? ledName(NAME) ]
95         emit turnLed : turnLed(NAME, off)
96     ]
97     transition
98         stopAfter 6000000
99         whenEvent local_BlinkOn -> ledBlinkingOn
100
101 Plan ledBlinkingOn [
102     println("Stato: ledBlinking on");
103     [ !? ledName(NAME) ]
104         emit turnLed : turnLed(NAME, on)
105 ]
106 transition
107     whenTime 200 -> ledBlinkingOff,
108     whenEvent local_BlinkOff -> ledOff
109
110 Plan ledBlinkingOff [
111     println("Stato: ledBlinking off");
112     [ !? ledName(NAME) ]
113         emit turnLed : turnLed(NAME, off)
114 ]
115 transition
116     whenTime 200 -> ledBlinkingOn,
117     whenEvent local_BlinkOff -> ledOff
118
119 }
120
121 QActor initial_conditions_checker context ctxApplicationLogic
122 {
123     Rules {
124         timeIsOk(true).
125         temperatureIsOk(true).
126
127         startRequirementsOk :- timeIsOk(true), temperatureIsOk(
128             true).
129     }
130     Plan init normal [
131         println("initial_conditions_checker STARTED")
132     ]
133     switchTo waitForEvents
134
135 Plan waitForEvents []
136 transition
137     stopAfter 1000000
138     whenEvent modelChanged -> checkEvent,
139     whenEvent userstart -> checkConditions
140 finally
141     repeatPlan
142
143 Plan checkEvent resumeLastPlan [

```

```

142     onEvent modelChanged : modelChanged(resource(name(
    temperatureIsOk), state(X))) -> ReplaceRule
    temperatureIsOk(_) with temperatureIsOk(X);
143     onEvent modelChanged : modelChanged(resource(name(
    timeIsOk), state(X))) -> ReplaceRule timeIsOk(_) with
    timeIsOk(X)
144 ]
145
146 Plan checkConditions resumeLastPlan [
147     onEvent userstart : userstart(user) -> {
148         [ !? startRequirementsOk ]
149         emit startRobot : startRobot
150     }
151 ]
152
153 }
154
155 QActor stop_conditions_checker context ctxApplicationLogic {
156     Plan init normal [
157         println("stop_conditions_checker STARTED")
158     ]
159     switchTo waitForEvents
160
161     Plan waitForEvents []
162     transition
163         stopAfter 1000000
164         whenEvent modelChanged -> stopRobot,
165         whenEvent userstop -> stopRobot
166     finally
167         repeatPlan
168
169     Plan stopRobot resumeLastPlan [
170         onEvent modelChanged : modelChanged(resource(name(
    temperatureIsOk), state(false))) -> emit stopRobot :
    stopRobot;
171         onEvent modelChanged : modelChanged(resource(name(
    timeIsOk), state(false))) -> emit stopRobot : stopRobot;
172         onEvent userstop : userstop(user) -> emit stopRobot :
    stopRobot
173     ]
174 }
175
176 QActor sonar_checker context ctxApplicationLogic {
177     Plan init normal [
178         println("sonar_checker STARTED")
179     ]
180     //switchTo waitForEvents
181
182     Plan waitForEvents []
183     transition

```



```

184     stopAfter 1000000
185     whenEvent modelChanged -> handleSonar
186 finally
187     repeatPlan
188
189 Plan handleSonar resumeLastPlan [
190     onEvent modelChanged : modelChanged(resource(name(sonar1)
191     , state(somethingDetected(true), _))) -> delay 1000; //
192     TODO interagisce con classe Java che si occupa di muovere
193     il robot
194     onEvent modelChanged : modelChanged(resource(name(sonar2)
195     , state(somethingDetected(true), _))) -> delay 1000 //
196     TODO interagisce con classe Java che si occupa di muovere
197     il robot
198 ]
199 }
200
201 QActor robot_basic_movements context ctxApplicationLogic {
202     Plan init normal [
203         println("robot_basic_movements STARTED")
204     ]
205     switchTo waitForMsgs
206
207     Plan waitForMsgs []
208     transition
209         stopAfter 1000000
210         whenMsg doBasicStep -> handleBasicStep,
211         whenMsg doRotation -> handleRotation
212     finally
213         repeatPlan
214
215     Plan handleBasicStep resumeLastPlan [
216         emit robotMovement : robotMovement(movingForward);
217         javaRun it.unibo.myBasicStepUtils.myObstacleHandler.start
218         ()
219     ]
220     transition
221         whenTime 300 -> goodResultBasicStep,
222         whenMsg foundObstacle -> badResultBasicStep
223
224     Plan goodResultBasicStep resumeLastPlan [
225         emit robotMovement : robotMovement(stopped);
226         [ !? notFirstTry ]
227         removeRule notFirstTry;
228         //emit
229         something_that_says_the_basic_Step_had_a_good_result
230         emit basicStepResult : basicStepResult(good)
231     ]
232
233     Plan badResultBasicStep resumeLastPlan [

```

```

226     javaRun it.unibo.myBasicStepUtils.myObstacleHandler.
        stopObstacleAndReset();
227
228 //     javaRun it.unibo.myBasicStepUtils.myObstacleHandler.
        stopObstacle();
229 //     emit robotMovement : robotMovement(stopped);
230 //     [ ?? resetBasicStep(DELAY) ] {
231 //         emit robotMovement : robotMovement(movingBackward);
232 //         delay DELAY;
233 //         emit robotMovement : robotMovement(stopped)
234 //     };
235
236     [ ?? notFirstTry ]
237         emit basicStepResult : basicStepResult(bad)
238         //emit
        something_that_says_the_basic_Step_had_a_bad_result
239     else {
240         addRule notFirstTry;
241         delay 500;
242         selfMsg doBasicStep : doBasicStep
243     }
244 ]
245
246 Plan handleRotation resumeLastPlan [
247     onMsg doRotation : doRotation(d) ->
248         emit robotMovement : robotMovement(turningRight);
249     onMsg doRotation : doRotation(a) ->
250         emit robotMovement : robotMovement(turningLeft)
251 ]
252 transition
253     whenTime 800 -> goodResultRotation
254
255 Plan goodResultRotation resumeLastPlan [
256     emit rotationResult : rotationResult(good)
257 ]
258
259 }
260
261 QActor collision_detector context ctxApplicationLogic {
262     Plan init normal [
263         println("collision_detector STARTED")
264     ]
265     switchTo waitForEvents
266
267     Plan waitForEvents []
268     transition
269         stopAfter 1000000
270         whenEvent modelChanged -> handleSonar
271     finally
272         repeatPlan

```

```

273
274 Plan handleSonar resumeLastPlan [
275     onEvent modelChanged : modelChanged(resource(name(robot),
276         state(movement(stopped), obstacleDetected(true)))) ->
277         forward robot_basic_movements -m foundObstacle :
278         foundObstacle
279 ]
280 }
281
282 QActor handle_planner context ctxApplicationLogic {
283     Plan init normal [
284         println("handle_planner STARTED");
285         javaRun it.unibo.myPlannerIntegrator.myPlanner.init()
286     ]
287     switchTo waitForStart
288
289     Plan waitForStart [
290         println("Waiting for start...")
291     ]
292     transition
293         stopAfter 1000000
294         whenEvent startRobot -> moveRobot
295     finally
296         repeatPlan
297
298     Plan moveRobot [
299         delay 1000;
300         javaRun it.unibo.myPlannerIntegrator.myPlanner.getMove();
301         // move(X) --> X: n = none, a = left, d = right, w =
302         forward)
303         [?? move(n)]{
304             selfMsg noMoreMoves : noMoreMoves
305         };
306         [?? move(w)]{
307             forward robot_basic_movements -m doBasicStep :
308             doBasicStep
309         };
310         [?? move(a)] {
311             forward robot_basic_movements -m doRotation :
312             doRotation(a)
313         };
314         [?? move(d)] {
315             forward robot_basic_movements -m doRotation :
316             doRotation(d)
317         }
318     ]
319     transition
320         stopAfter 1000000
321         whenEvent basicStepResult -> handleResult,
322         whenEvent rotationResult -> handleResult,

```

```

317     whenEvent stopRobot -> waitForStart,
318     whenMsg noMoreMoves -> waitForStart
319 finally
320     repeatPlan
321
322 Plan handleResult resumeLastPlan [
323     onEvent basicStepResult : basicStepResult(bad) ->
324         javaRun it.unibo.myPlannerIntegrator.myPlanner.
325             setMoveResult("bad");
326     onEvent basicStepResult : basicStepResult(good) ->
327         javaRun it.unibo.myPlannerIntegrator.myPlanner.
328             setMoveResult("good");
329     onEvent rotationResult : rotationResult(good) ->
330         javaRun it.unibo.myPlannerIntegrator.myPlanner.
331             setMoveResult("good")
332 ]
333
334 }
335
336 QActor mock_sender context ctxApplicationLogic { //-g gray {
337     Plan init normal [
338         println("mock_sender STARTED")
339     ]
340     //switchTo move
341
342     Plan move [
343         delay 2500;
344         emit userstart : userstart(user)
345     ]
346 }

```

applicationLogic.qa

Input:

```

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
10 // the robot
11 // VALUE can be:
12 // - stopped
13 // - movingForward
14 // - movingBackward
15 // - turningLeft

```

```

15 // - turningRight
16 Event robotMovement : robotMovement(VALUE)
17
18 // se si usa con il robot fisico al posto di localhost
19 // 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         javaRun it.unibo.temperature_sensor_adapter.
            webTemperatureSensorAdapter.init();
27         delay 500
28     ]
29     switchTo sendEvents
30
31     Plan sendEvents [
32         //delay 10000;
33         //javaRun it.unibo.temperature_sensor_adapter.
            webTemperatureSensorAdapter.updateTemperature()
34         delay 1000;
35         emit updateTemperature : updateTemperature(temp, 12)
36     ]
37     finally repeatPlan
38
39 }
40
41 QActor timer_adapter context ctxInput {
42
43     Plan init normal [
44         println("resource_model STARTED")
45     ]
46     switchTo sendEvents
47
48     Plan sendEvents [
49         delay 1000;
50         //javaRun it.unibo.timer_adapter.systemTimerAdapter.
            updateTime()
51         emit updateTime : updateTime(timer, currentTime(8,0,0))
52     ]
53     finally repeatPlan
54
55 }
56
57
58 // It simulates the robot movement
59 QActor input_element context ctxInput {

```

```

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

```

input.qa

Output:

```

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
    component
7 Dispatch msg_modelChanged : modelChanged(resource(NAME, STATE
    ))

```

```

8
9 // il contesto del raspberry. bisogna mettere l'ip giusto.
10 // a logica non dovrebbe essere qui, dovrebbe essere il
    raspberry che "si collega"
11 // 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="192.168.43.225" port
    =9010] -standalone
15 // questa invece per il led sul robot reale
16 //Context ctxRealLedAdapter ip[host="192.168.43.225" port
    =9011] -standalone
17
18 //Context ctxRealRobotAdapter ip[host="192.168.1.15" port
    =9010] -standalone
19
20
21 // se si usa con il robot fisico al posto di localhost
    bisogna mettere l'indirizzo del pc
22 Context ctxResourceModel ip[host="localhost" port=8099] -
    standalone
23 Context ctxOutput ip[host="localhost" port=8098]
24
25 // It turns the pfrs robot into an event-driven component
26 EventHandler pfrs_event_driven for modelChanged {
27     forwardEvent adapter_to_pfrs_mbot -m msg_modelChanged
28 };
29
30
31 QActor mock_output_led context ctxOutput {
32
33     Rules {
34         // rules needed by the application logic
35         ledName(led).
36     }
37
38
39     Plan init normal [
40         println("resource_representation_element STARTED");
41         javaRun it.unibo.custom.gui.customBlsGui.
            createCustomLedGui();
42         javaRun it.unibo.custom.gui.customBlsGui.setLed("off")
43     ]
44     switchTo waitForModelChanged
45
46     Plan waitForModelChanged []
47     transition
48         stopAfter 1000000
49         whenEvent modelChanged -> outputtingData

```

```

50     finally
51         repeatPlan
52
53     Plan outputtingData resumeLastPlan [
54         [ !? ledName(NAME) ]
55         onEvent modelChanged : modelChanged(resource(name(NAME)
56         , state(on))) ->
57             javaRun it.unibo.custom.gui.customBlsGui.setLed("on")
58         ;
59
60         [ !? ledName(NAME) ]
61         onEvent modelChanged : modelChanged(resource(name(NAME)
62         , state(off))) ->
63             javaRun it.unibo.custom.gui.customBlsGui.setLed("off"
64         )
65     ]
66 }
67
68 QActor hue_lamp_adapter context ctxOutput {
69
70     Rules {
71         // rules needed by the application logic
72         ledName(led).
73     }
74
75     Plan init normal [
76         println("hue_lamp_adapter STARTED");
77         javaRun it.unibo.myUtils.hueLampHandler.init("bridge ip",
78         "username", "lamp id")
79     ]
80     switchTo waitForModelChanged
81
82     Plan waitForModelChanged []
83     transition
84         stopAfter 1000000
85         whenEvent modelChanged -> outputtingData
86     finally
87         repeatPlan
88
89     Plan outputtingData resumeLastPlan [
90         [ !? ledName(NAME) ]
91         onEvent modelChanged : modelChanged(resource(name(NAME)
92         , state(on))) ->
93             javaRun it.unibo.myUtils.hueLampHandler.turnOn();
94
95         [ !? ledName(NAME) ]
96         onEvent modelChanged : modelChanged(resource(name(NAME)
97         , state(off))) ->

```



```

93         javaRun it.unibo.myUtils.hueLampHandler.turnOff()
94     ]
95 }
96 }
97
98 QActor mock_output_temperature context ctxOutput -g green {
99
100     Plan init normal [
101         println("Temperature Observer STARTED")
102     ]
103     switchTo waitForEvents
104
105     Plan waitForEvents []
106     transition
107         stopAfter 1000000
108         whenEvent modelChanged -> handleModelChanged
109     finally repeatPlan
110
111     Plan handleModelChanged resumeLastPlan [
112         onEvent modelChanged : modelChanged(resource(name(temp),
113             state(temperature(VALUE)))) ->
114             println(temp(VALUE))
115     ]
116 }
117
118 QActor mock_output_time context ctxOutput -g yellow {
119
120     Plan init normal [
121         println("Timer Observer STARTED");
122         demo consult("resourceModel.pl")
123     ]
124     switchTo waitForEvents
125
126     Plan waitForEvents []
127     transition
128         stopAfter 1000000
129         whenEvent modelChanged -> handleModelChanged
130     finally repeatPlan
131
132     Plan handleModelChanged resumeLastPlan [
133         onEvent modelChanged : modelChanged(resource(name(timer),
134             state(currentTime(H, M, S)))) ->
135             println(now(H, M, S))
136     ]
137 }
138
139
140 // It makes pfrs robot a QActor entity.

```

```

141 QActor adapter_to_pfrs_mbot context ctxOutput { //-pubsub {
142
143     Plan init normal [
144         println("adapter_to_pfrs_mbot STARTS");
145         javaRun it.unibo.pfrs.mbotConnTcp.initClientConn()
146     ]
147     switchTo waitMsgs
148
149     Plan waitMsgs []
150     transition
151         stopAfter 1000000
152         whenMsg msg_modelChanged -> moveRobot
153     finally repeatPlan
154
155
156     Plan moveRobot resumeLastPlan [
157         onMsg msg_modelChanged : modelChanged(resource(name(robot
158             ), state(movement(stopped), X))) ->
159             javaRun it.unibo.pfrs.mbotConnTcp.mbotStop();
160         onMsg msg_modelChanged : modelChanged(resource(name(robot
161             ), state(movement(movingForward), obstacleDetected(false)
162             ))) ->
163             javaRun it.unibo.pfrs.mbotConnTcp.mbotForward();
164         onMsg msg_modelChanged : modelChanged(resource(name(robot
165             ), state(movement(movingBackward), X))) ->
166             javaRun it.unibo.pfrs.mbotConnTcp.mbotBackward();
167         onMsg msg_modelChanged : modelChanged(resource(name(robot
168             ), state(movement(turningLeft), X))) ->
169             javaRun it.unibo.pfrs.mbotConnTcp.mbotLeft();
170         onMsg msg_modelChanged : modelChanged(resource(name(robot
171             ), state(movement(turningRight), X))) ->
172             javaRun it.unibo.pfrs.mbotConnTcp.mbotRight()
173     ]
174 }
175 //

```

output.qa

Real robot adapter:

```

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
8 // component
9 Dispatch msg_modelChanged : modelChanged(resource(NAME, STATE
10 ))

```

```

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 {
21     //-pubsub {
22
23     Rules {
24
25     }
26
27     Plan init normal [
28         //     javaRun it.unibo.myArduinoUtils.connArduino.initPc("
29             COM6", "115200");
30         javaRun it.unibo.myArduinoUtils.connArduino.initRasp("
31             115200");
32         println("adapter_to_physical_mbot STARTS")
33     ]
34     switchTo waitMsgs
35
36     Plan waitMsgs []
37     transition
38         stopAfter 1000000
39         whenMsg msg_modelChanged -> moveRobot
40     finally repeatPlan
41
42     Plan moveRobot resumeLastPlan [
43         onMsg msg_modelChanged : modelChanged(resource(name(robot
44             ), state(movement(stopped), X))) ->
45             javaRun it.unibo.myArduinoUtils.connArduino.mbotStop();
46         onMsg msg_modelChanged : modelChanged(resource(name(robot
47             ), state(movement(movingForward), obstacleDetected(false)
48             ))) ->
49             javaRun it.unibo.myArduinoUtils.connArduino.mbotForward
50             ();
51         onMsg msg_modelChanged : modelChanged(resource(name(robot
52             ), state(movement(movingBackward), X))) ->
53             javaRun it.unibo.myArduinoUtils.connArduino.
54             mbotBackward();
55         onMsg msg_modelChanged : modelChanged(resource(name(robot
56             ), 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 }

```

real_robot_adapter.qa

Real led adapter:

```

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 outputingData 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 }

```

real_led_adapter.qa

7 Implementazione

Per utilizzare il pianificatore fornito, si è reso necessario realizzare un componente Java che espone all'utilizzatore QActor metodi per recuperare la mossa successiva e fornire al pianificatore l'esito del tentativo di eseguire la mossa.

```

1 package it.unibo.myPlannerIntegrator;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 import aim.core.agent.Action;
7 import it.unibo.exploremap.program.aiutil;
8 import it.unibo.qactors.akka.QActor;
9
10 public class myPlanner {
11
12     public static List<Action> lastMove = new ArrayList<Action>
13         <>();
14
15     public static void init(QActor actor) {
16         try {
17             aiutil.initAI();
18         } catch (Exception e) {
19             // TODO Auto-generated catch block
20             e.printStackTrace();
21         }
22     }
23
24     public static void getMove(QActor actor) {
25         try {
26             if (lastMove.isEmpty())
27                 lastMove = aiutil.doPlan();
28         }
29     }
30 }

```

```

27         if(lastMove == null) {
28             actor.addRule("move(n)");
29         } else {
30             // move(X) --> X: n = none, a = left, d = right, w =
forward)
31             actor.addRule("move(" + lastMove.get(0).toString() +
");");
32             System.out.println("\n-----\nThe
next move is: " + lastMove);
33             aiutil.showMap();
34             System.out.println("\n-----");
35         }
36
37     } catch (Exception e) {
38         // TODO Auto-generated catch block
39         e.printStackTrace();
40     }
41 }
42
43 public static void setMoveResult(QActor actor, String
result) {
44     try {
45         if(result.equalsIgnoreCase("good")) {
46             aiutil.doMove(lastMove.get(0).toString());
47             lastMove.remove(0);
48         }
49         else if(result.equalsIgnoreCase("bad")) {
50             switch (aiutil.initialState.getDirection()) {
51                 case RIGHT:
52                     aiutil.doMove("obstacleOnRight");
53                     break;
54                 case LEFT:
55                     aiutil.doMove("obstacleOnLeft");
56                     break;
57                 case UP:
58                     aiutil.doMove("obstacleOnUp");
59                     break;
60                 case DOWN:
61                     aiutil.doMove("obstacleOnDown");
62                     break;
63             }
64             lastMove.clear();
65         }
66     } catch (Exception e) {
67         // TODO Auto-generated catch block
68         e.printStackTrace();
69     }
70 }
71
72 }

```

7.1 Frontend Server

L'implementazione del frontend server, necessario per fornire all'utente la possibilità di autenticarsi e quindi di dare i comandi di inizio e stop al robot, è avvenuta in **NodeJs**, sfruttando il framework Express. In figura 5 è presente un modello informale che rappresenta le tecnologie utilizzate nell'integrazione fra il frontend server e l'infrastruttura *QActor* già presente.

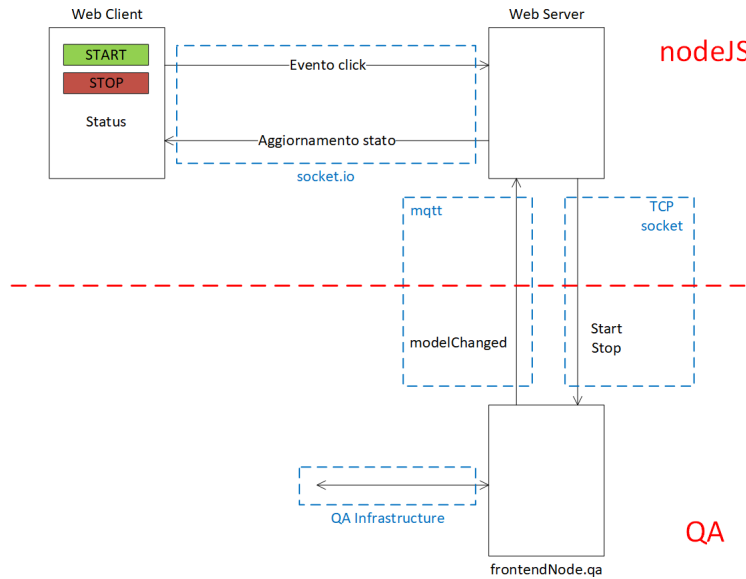





Fig. 5. Architettura informale che rappresenta l'integrazione del frontend server con l'infrastruttura *QActor*

Come si può vedere la comunicazione tra la pagina web fornita all'utente e il server web avviene tramite `socket.io` e rimane pienamente in ambiente **NodeJs**. La comunicazione con l'infrastruttura *QActor* utilizza invece una **TCP socket** nativa del contesto; la comunicazione nella direzione opposta avviene invece tramite un server `mqtt`.

8 Autori

Foto degli autori		
		
Luca Bonfiglioli	Nicola Fava	Antonio Grasso