

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	19
7 Implementazione	19
8 Autori	20

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`.

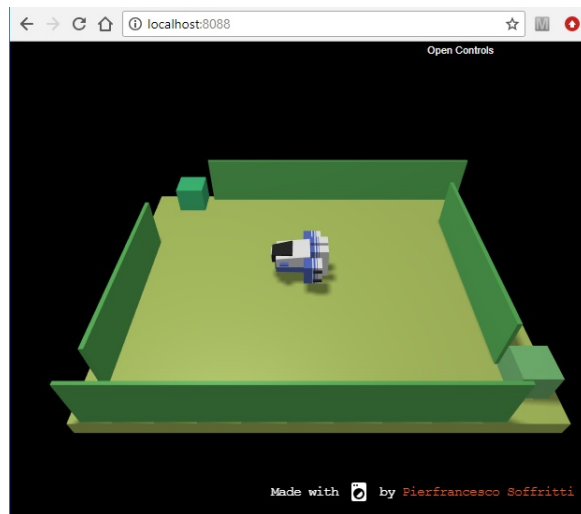


Fig. 1. Esempio di pavimento con il robot in ambiente simulato

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

I requisiti sono stati analizzati e formalizzati in modo iterativo in ordine di importanza, come riportato nel Product Backlog di Table 1.

Per formalizzare il requisito **R-FloorClean** è prima necessario stabilire cosa si intenda con pulire tutto il pavimento. Introducendo l'assunzione che la stanza sia rettangolare è possibile suddividerne la superficie in celle quadrate di dimensione fissa. 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

Requisito	Priorità
R-FloorClean	1
R-Map	2
R-AvoidFix	3
R-AvoidMobile	4
R-Obstacle	5
R-BlinkHue	6
R-BlinkLed	6
R-Start	7
R-TempOk	7
R-TimeOk	7
R-Stop	7
R-TempKo	7
R-TimeKo	7
R-End	7

Table 1. Product Backlog

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

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

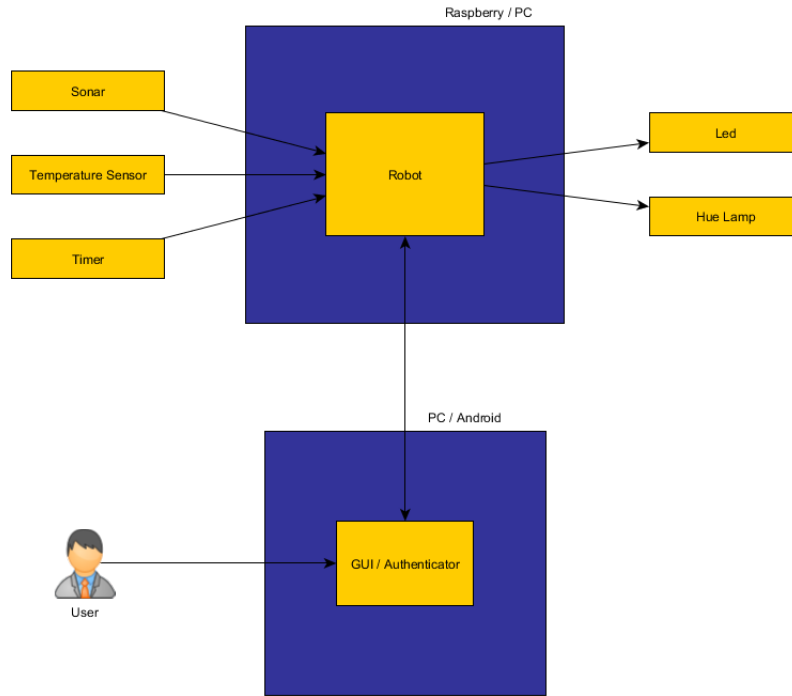


Fig. 2. Diagramma informale dell'analisi dei requisiti

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 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
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 }

```

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 [

```

```

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 }

```

reqAnalysisUser.qa

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.

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.

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

5 Analisi del problema

Tenendo conto delle assunzioni e delle considerazioni già affrontate in analisi dei requisiti, assumendo inizialmente per semplicità che la stanza sia rettangolare e priva di ostacoli, il requisito **R-FloorClean** è stato modellato attraverso l'introduzione del QActor **robotmind**.

Quest'ultimo possiede una propria base di conoscenza necessaria a tenere traccia delle dimensioni della stanza e della direzione corrente del robot. Tale QActor è sensibile ai messaggi **robotMindCmd** ed agli eventi **sensorEvent**, in particolare:

- alla ricezione del messaggio **robotMindCmd : robotMindCmd(explore)** viene invocato il metodo statico **doBasicStep()** della classe Java **robot**, a cui viene delegata l'esecuzione del basic step del robot. Questo metodo deve tenere conto degli eventi **sensorEvent : sensorEvent(onboardsonar)** per determinare quando il robot si è imbattuto in un ostacolo, che secondo le precedenti assunzioni viene a coincidere con il secondo sonar. In seguito a ciò viene discriminato il buon esito del basic step, determinato dalla presenza o dall'assenza di un ostacolo lungo il percorso del robot. A seconda della direzione del robot vengono incrementate le dimensioni della stanza. Il caso di fallimento del basic step in questa fase di cosiddetta "esplorazione" si verifica quando il robot colpisce il secondo sonar: in tal caso vengono invocati i metodi statici della classe **planner** necessari ad indicare al planner le dimensioni della stanza per poi passare alla fase di pulizia della stanza vera e propria con l'invio del messaggio **robotMindCmd : robotMindCmd(clean)**.
- alla ricezione dell'evento **sensorEvent : sensorEvent(sonar2)** viene modificata la direzione del robot memorizzata nella base di conoscenza;
- alla ricezione del messaggio **robotMindCmd : robotMindCmd(clean)** viene dapprima invocato il metodo statico **nextMove()** della classe Java **planner**, dove per planner si intende un'entità a conoscenza delle dimensioni della stanza in grado di determinare la prossima mossa del robot (basic step o basic step + rotazione di 90 gradi), inserita nella base di conoscenza di volta in volta, col fine di coprire l'intera area della stanza. A seconda della mossa indicata dal planner vengono invocati i metodi statici della classe **robot** necessari ad eseguire la mossa.
La pulizia della stanza si intende conclusa quando viene aggiunto dal planner alla base di conoscenza il fatto **nextMove(n,n)**, che indica al robot di non eseguire nessun movimento ulteriore.

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 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).

```

```

86     incrementX:- dimX(X), retract(dimX(_)), X1 is X + 1,
      assert(dimX(X1)).
87     incrementY:- dimY(Y), retract(dimY(_)), Y1 is Y + 1,
      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
          listen to events sensorEvent : sensorEvent(onboardsonar)
107         javaRun robot.doBasicStep();
108         // robot starts from start-point towards the bottom-
          side, it changes its direction when it receives
109         // the event sensorEvent : sensorEvent(sonar2) and it
          goes forward until it hits the second sonar
110         [?? basicStepResult(true)] {
111             [!? dir(X)]
112             demo incrementX
113         else
114             demo incrementY;
115         selfMsg robotMindCmd : robotMindCmd(explore)
116     }
117     else {
118         demo assert(map);
119         [!? dimX(X)]
120         javaRun planner.setSizeX(X);
121         [!? dimY(Y)]
122         javaRun planner.setSizeY(Y);
123         selfMsg robotMindCmd : robotMindCmd(clean)
124     }
125 };
126 onEvent sensorEvent : sensorEvent(sonar2) -> {
127     demo retract(dir(Y));
128     demo assert(dir(X))
129 };
130 onMsg robotMindCmd : robotMindCmd(clean) -> {

```

```

131     // nextMove = basicStep or 90 degree rotation +
    basicStep
132     javaRun planner.nextMove();
133     // 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 ]
234 }
235
236 QActor huelamp context ctxProbRobot {
237     Plan initial normal [
238         println("Hue Lamp started")
239     ]
240     switchTo waitForEvent
241
242     Plan waitForEvent []
243     transition stopAfter 600000
244     whenEvent outCmd -> handleEvent
245     finally repeatPlan
246
247     Plan handleEvent resumeLastPlan [
248         onEvent outCmd : outCmd(startblinking) -> println("Hue
249         Lamp start blinking");
250         onEvent outCmd : outCmd(stopblinking) -> println("Hue
251         Lamp stop blinking")
252     ]
253 }

```

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] -
8     standalone
9 Context ctxProbUser ip[host="localhost" port=5500]
10
11 QActor gui context ctxProbUser {
12     Plan initial normal [
13         println("Gui started")
14     ]
15     switchTo waitForMsg
16
17     Plan waitForMsg [
18     ]
19     transition stopAfter 600000
20     whenMsg userCmd -> handleMsg
21     finally repeatPlan

```

```

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

7 Implementazione

8 Autori

Foto degli autori		
		
Luca Bonfiglioli	Nicola Fava	Antonio Grasso