UNIVERSITA' DEGLI STUDI DI BARI

FACOLTA' DI SCIENZE MATEMATICHE, FISICHE E NATURALI
CORSO DI LAUREA IN INFORMATICA

TESI DI LAUREA
IN
ARCHITETTURA DEGLI ELABORATORI

AGENTI AUTONOMI UBIQUI

Relatore:
Prof.ssa Berardina DE CAROLIS

Laureando:
Ignazio PALMISANO

Anno Accademico 2001-2002

Dedicato ai miei genitori,

perché lo hanno reso possibile,

ai miei amici,

perché senza di loro non sarebbe stata la stessa cosa,

e a chi ci ha creduto.

Capitolo 1: Ubiquitous computing

"- I need a PC here

- You just turn around, and you'll get it in front of you

- Yes, I know, but I need it here!"

1.1 Introduzione

Lo scopo di questa tesi è ampliare un progetto già esistente (il sistema D-Me [11], [17], [18]), per introdurre il trattamento esplicito del concetto di autonomia. Il sistema D-Me è un sistema multiagente (MAS - MultiAgent System) implementato interamente in Java, che sfrutta la piattaforma JADE (Java Agent DEvelopment Framework) per la definizione, creazione e gestione dei suoi agenti. In tale ambiente, ogni utente che interagisce con il sistema è rappresentato da un agente, chiamato MainAgent, che riceve dall'utente la delega ad agire come tramite tra l'utente stesso e gli altri agenti presenti nel sistema, allo scopo di realizzare i compiti che gli vengono affidati dall'utente e gli obiettivi propri dell'agente. Vedremo più avanti una descrizione precisa dei meccanismi di ragionamento che il MainAgent utilizza per perseguire i propri obiettivi.

1.2 La novità dell'Ubiquitous computing

Ubiquitous computing è il nome che è stato dato al fenomeno in cui si assiste in questi anni: l'invasione (pacifica) dei computer.

Fino a non molto tempo fa, il modo classico di immaginare i computer era: "Sulla mia scrivania, c'è una grossa macchina da scrivere capace di fare anche i conti. Quando ne ho bisogno, l'accendo e la uso, poi mi alzo e torno al lavoro normale".

Negli ultimi tempi, il numero dei computer è cresciuto in maniera impressionante, e di pari passo sono aumentate le funzionalità che la "macchina da scrivere" mette a disposizione, fino al punto da poter dire che i computer sono ovunque e fanno di tutto.

Come dice Negroponte[1], "Computing is not about computers anymore. It is about living". Negli ultimi tempi, troviamo computer, e più in generale processori, anche nei frigoriferi [2], nei televisori, nelle automobili. A questo punto diventa imperativo capire qual è il modo migliore per far comunicare i device intelligenti, sia tra loro che con l'utente finale. La comunicazione tra device è ormai talmente necessaria che i colossi dei vari rami (elettrodomestici e in generale appliances per la casa) stringono veri e propri patti di collaborazione, come la già citata WhirlPool con la Nokia [3] per sviluppare in joint venture nuove soluzioni. L'obiettivo finale, secondo Weiser [4], è quello di fornire all'utente applicazioni flessibili, sensibili al contesto, all'esperienza accumulata dall'utente, addirittura al suo stato emotivo.

1.3 L’impatto nella vita quotidiana

Considerando l’impatto di questa rivoluzione tecnologica sugli utenti, emergono alcuni problemi fondamentali: non tutti gli utenti hanno le stesse competenze, e solo pochi sono in grado di gestire la complessità dell’interazione tra tutti i sistemi elettronici che è possibile trovare in un ambiente intelligente. La presenza di servizi “ovunque” e “in qualunque situazione” impone un diverso modo di pensare agli utenti, che sono quindi costretti a un cambio di paradigma nel porsi di fronte alle novità tecniche. Inoltre, la complessità derivante dalla quantità di tecnologia presente in ogni casa finisce per spaventare molti utenti, i quali, imparate le funzioni di base di un elettrodomestico, smettono di curiosare, e quindi non sfruttano appieno le potenzialità di ciò che hanno acquistato.

E’ essenziale, perché le innovazioni tecnologiche siano utili alla società e non restino banali giocattoli, permettere all’utente di sfruttare tutti i servizi disponibili in maniera semplice e naturale. Per far ciò, è necessario che l’utente abbia la possibilità di delegare a qualcun altro la parte “difficile” dell’utilizzo delle risorse di calcolo a sua disposizione, o meglio la parte “noiosa”: dal riconoscimento automatico delle periferiche disponibili, concetto che nell’ambito dell’Ubiquitous Computing si evolve diventando riconoscimento automatico dei dispositivi e delle funzionalità offerte, aggiornato momento per momento, alla configurazione di queste ultime. Il “qualcun altro” in questione, che Negroponte identificava, sempre in [1], con “la cognata appassionata di cinema”, è rappresentato da un programma il cui scopo principale è eseguire, facendo le veci dell’utente, i compiti che l’utente gli delega.

Si configurano quindi nuovi spunti di ricerca:

- In che misura l'utente "delega" al programma i propri "poteri" per l'esecuzione di questi compiti?
- Qual è il grado di autonomia con cui il programma può affrontare l'esecuzione di questi compiti?

Lo scopo di questa tesi è fornire un'analisi di alcuni dei metodi possibili per la gestione esplicita di questo modello, e un prototipo funzionante che ne mostri l'effettiva implementazione.

Capitolo 2: Autonomia e ragionamento

2.1 Agenti ubiqui

Date le caratteristiche principali che il nostro “programma” deve avere, tra i vari paradigmi di progettazione per la realizzazione di un sistema reale bisogna trovarne uno che (possibilmente) risponda ai requisiti salienti del sistema:

- La capacità di rispondere in maniera flessibile agli stimoli, sia quelli derivanti dal contesto in cui il programma si trova ad operare che quelli provenienti dall’utente
- Le sue caratteristiche di autonomia.

Quest’ultima in particolare ha orientato la scelta verso il paradigma di progettazione e realizzazione “Agent-Oriented”.

“Un agente è un sistema (incapsulato) situato in un ambiente, capace di azioni autonome e flessibili, in quell’ambiente, per raggiungere i suoi obiettivi di progetto” [6]

La prima parte di questa definizione è calzante anche per il paradigma “Object-Oriented”, che infatti ha molti punti in comune con il paradigma ad agenti: dal punto di vista implementativo, infatti, esistono svariate implementazioni di agenti come oggetti particolari, che basano le loro caratteristiche di autonomia sull’incapsulamento di dati e metodi, tipico degli oggetti.

Tuttavia, il paradigma Agent-Oriented si distingue per la seconda parte della definizione: un agente non è un oggetto passivo, ma è in grado di ragionare sui propri obiettivi e di compiere le azioni necessarie a raggiungerli; esso è quindi intelligente e proattivo.

In questo contesto, intelligenza e proattività sono strettamente legate: se la proattività è la capacità di prendere decisioni dirette dalla volontà di raggiungere un set di obiettivi, o goal, allora l'intelligenza è il meccanismo (il ragionamento automatico) che permette la realizzazione della decisione (analisi della situazione e scelta tra le varie alternative).

Queste caratteristiche rendono questo paradigma idoneo alla progettazione di sistemi di ubiquitous computing. Esse consentono all'utente di poter specificare i suoi desideri usando un linguaggio ad alto livello (idealmente il linguaggio naturale), e quindi in modo semplice. Sarà poi competenza dell'agente decidere quale strategia adottare e quali servizi dell'ambiente utilizzare per soddisfare i bisogni dell'utente.

In particolare, con riferimento a quanto riassunto da Flores-Mendez in [7], il tipo di agente che appare più adatto ad interagire con l'utente per soddisfarne i bisogni è un agente che realizza la "weak notion of agency", vale a dire che soddisfa la definizione di Jennings e Woolridge [8]: si tratta quindi di un agente:

- Autonomo: opera senza il diretto intervento dell'utente, e ha controllo sulle proprie azioni e stati interni
- Sociale: interagisce con altri agenti (potenzialmente umani) attraverso un linguaggio per la comunicazione tra agenti
- Reattivo: percepisce l'environment e reagisce ai cambiamenti

- Proattivo: in grado di prendere l’iniziativa per raggiungere i propri goal

Lasciamo quindi da parte le caratteristiche che Woolridge e Jennings considerano specifiche della “strong notion of agency”, come stati emozionali e carattere dell’agente (le caratteristiche che di solito associamo ad agenti umani).

2.2 Agenti BDI

In precedenza, abbiamo descritto il sistema D-Me come un MAS, cioè un MultiAgent System, che è un sistema che rispetta le seguenti caratteristiche [9]:

- ogni agente ha delle capacità, ma esse sono insufficienti al completamento del task
- non c’è un meccanismo globale di controllo
- i dati non sono centralizzati
- la computazione è asincrona

Per far ciò, il MainAgent già esistente nel sistema (il rappresentante dell’utente) è strutturato secondo l’approccio BDI (Beliefs-Desires-Intentions), come descritto in [10].

In questo approccio, l’agente basa il proprio comportamento su tre elementi:

- Beliefs: le credenze dell'agente
- Desires: i desideri dell’agente
- Intentions: le intenzioni dell’agente per soddisfare i suoi desideri

E’ possibile notare un parallelismo tra il modo in cui è strutturato un agente BDI e la mente umana; infatti, l’uomo ragiona nello stesso modo, eseguendo delle Intentions, appropriate ai Beliefs, per raggiungere i suoi gola, cioè i suoi Desires.

2.3 Autonomia e delega

Il problema di dare una definizione teorica valida per il concetto di autonomia e di delega è stato affrontato da Castelfranchi e Falcone in [5], che hanno basato il loro modello sull’idea che è necessario esplicitare i concetti di autonomia e delega, estrapolandoli quindi dal codice ed astraendo dalla particolare implementazione, in modo che sia possibile per l’utente mantenere il controllo dell’interazione, pur non facendosi carico dei compiti più onerosi o che richiedono maggior attenzione o prontezza di riflessi (i compiti che tipicamente un computer sa eseguire meglio di un essere umano).

In particolare, essi distinguono l’autonomia in due tipologie:

- “**Metaautonomia sulla negoziazione del servizio**”: è l’autonomia che viene conferita al programma al momento della negoziazione del servizio, cioè la sua libertà di contrattazione dell’autonomia di realizzazione.
- “**Autonomia di realizzazione**”: essa si situa a un livello più basso, cioè al livello dell’esecuzione del compito delegato, e si riferisce alla libertà di iniziativa dell’agente nell’esecuzione delle varie porzioni in cui può essere

diviso il compito assegnato, o task; essa riguarda quindi l'interazione con altri agenti, con dispositivi fisici, la deduzione di dati non esplicitamente introdotti dall'utente, l'avvio di sottotask non indispensabili per l'esecuzione del task principale ma ritenuti utili all'utente, e l'eventuale mancata esecuzione di alcune parti del task esplicitamente richieste.

Su tale base si configurano delle variazioni nel tipo di aiuto fornito dal programma all'utente, che si possono classificare considerando tre categorie:

- "**OverHelper**": il programma può eseguire un task, rappresentato come un piano, che include il task iniziale e altri sottotask ritenuti utili
- "**SubHelper**": il programma può limitarsi a eseguire solo una porzione del piano originale
- "**CriticalHelper**": il programma può anche decidere di attuare un piano diverso da quello definito dall'utente, spinto a ciò da limiti di tipo ambientale o da constraints imposte da un'autorità superiore a quella dell'utente, come potrebbe essere, ad esempio, la presenza di filtri per contenuti non adatti ai minori nell'accesso a risorse su web.

2.4 Definizione di agente autonomo

L'autonomia è la caratteristica che distingue un programma da un agente: essa è la capacità di scegliere tra più alternative (flusso di controllo o valori da assegnare a una variabile), necessaria perché l'agente non debba dipendere da un'altra entità esterna ad esso. Come in [14], un agente "autonomo" è un agente che ha controllo sulle proprie azioni. In base alla definizione di agente, secondo cui "**un agente è un'entità in grado di percepire l'ambiente attraverso dei sensori, e agire su di esso con degli attuatori**" [14], si possono escludere dispositivi del mondo fisico che rientrano nella prima definizione, ma che è eccessivo considerare agenti (per esempio, un termostato).

Castelfranchi e Falcone considerano, per la trattazione teorica della delega e dell'autonomia nelle interazioni tra agenti i seguenti insiemi:

- un insieme di atti o azioni: gli atti che un agente può mettere in esecuzione
- un insieme di agenti: gli agenti che popolano l'ambiente, e che possono comunicare tra loro
- una libreria di piani: i piani per l'esecuzione degli obiettivi

Con questi insiemi di primitive, si può considerare il piano (o il bisogno) che un agente (o l'utente) A sta cercando di soddisfare come un piano MA

(MultiAgente), in cui parte del piano è affidata ad altri agenti (S) che il primo agente può contattare. In tal senso, questi ultimi agenti hanno un goal, nella loro azione, limitato al goal dell'agente che ha richiesto aiuto, comportandosi quindi da serventi (questi agenti non prendono iniziative, all'interno della loro porzione di piano; si limitano ad eseguire quanto richiesto, rispettando lo spirito cooperativo dell'ambiente).

Nel modo in cui l'agente servente viene incluso nel piano, è possibile distinguere tre casi diversi:

- "**Weak delegation**": l'agente A sfrutta i servizi offerti da un agente S, senza influenzare la sua autonomia (S è completamente autonomo);
- "**Mild delegation**": l'agente A non esegue una richiesta, ma crea le condizioni per cui S si senta spinto ad eseguire l'azione che A desidera
- "**Strong delegation**": in tal caso, è esplicita sia la richiesta che l'accettazione del task, cioè vi è una contrattazione esplicita tra A e S, con evevntuale negoziazione del livello di autonomia di S per l'esecuzione del task.

Si introducono quindi tre operatori, uno per ognuno dei tre casi:

- W-Delegates (A S P): A weak-delegates P (il piano o porzione di piano) to S
- M-Delegates (A S P): A mild-delegates P to S
- S-Delegates (A S P): A strong-delegates P to S

D'altra parte, è possibile considerare due modi in cui S può intraprendere l'azione:

- “**Weak-adoption**”: si verifica quando S crede che A abbia bisogno di una particolare azione da parte sua, e decide di eseguirla. Non è detto che ci sia comunicazione esplicita tra S e A, e neanche che A utilizzi i risultati prodotti da S. In questo caso, A potrebbe aver attuato una “weak-delegation” o una “mild-delegation”, o anche nessuna “delegation”.
- “**Strong-adoption**”: si verifica quando S esegue l’azione dopo aver ricevuto una richiesta ed aver negoziato il task con A. In tal caso, è necessario che da parte di A ci sia stata una “strong-delegation”

E’ possibile anche introdurre una distinzione sul livello di dettaglio a cui è specificato il piano di A che S deve eseguire:

- “**Close-delegation**”: il piano è completamente specificato.
- “**Open-delegation**”: il piano non è completamente specificato, per cui a S è lasciata più autonomia su come realizzare quelle parti del paino che non sono dettagliate.
- “**Control-based delegation**”: viene specificato il controllo che A esercita su S.

A livello di negoziazione (delega/accettazione) è possibile che vi siano delle modifiche al piano o al goal sia da parte di S che di A:

Da parte di S:

- “**SubHelp**”: S realizza solo una parte del piano originario
- “**OverHelp**”: S non cambia il piano, ma adotta come proprio goal un goal di livello più alto

- "**CriticalHelp**": S raggiunge lo stesso goal, ma cambia il piano, considerando una soluzione alternativa

Dal punto di vista di A:

- "**SubHelp**": analogamente al caso precedente, A riduce il piano che S deve eseguire
- "**OverHelp**": A aumenta la dimensione del piano che S deve eseguire, considerando un piano che va oltre gli obiettivi precedenti
- "**CriticalHelp**": A modifica il piano, delegando a S una soluzione alternativa al piano precedentemente negoziato, ma con lo stesso goal
- "**OpeningDelegation**": A delega a S un piano più astratto del precedente
- "**ClosingDelegation**": A delega ad S un piano meno astratto, specificato a un livello più basso

In base a queste possibili situazioni di modifica della delega, l'autonomia dell'agente S viene variata: essa può essere variata a livello della negoziazione del piano, oppure a livello dell'autonomia di realizzazione, del livello di controllo o del tipo di delega/accettazione. In tutti questi casi, si può dire:

- Se a variare la delega è S, la sua autonomia di negoziazione deve variare sempre: infatti, è S a prendere l'iniziativa di variare la delega; invece, non è detto che vari la sua autonomia di realizzazione
- Se a variare la delega è A, l'autonomia di negoziazione di S non cambia mai; infatti, le ragioni che spingono A a modificare la delega non influenzano S. E' possibile invece che venga modificata l'autonomia di realizzazione di S: ciò

avviene nel caso in cui A attua la ClosingDelegation, specificando un piano più a basso livello, e di fatto limitando l'autonomia di S nell'esecuzione

2.5 Il nostro approccio

L'idea alla base del sistema multiagente, che sarà presentato in dettaglio nel capitolo 3, è quella di rappresentare l'utente attraverso un agente del sistema, che chiameremo MainAgent.

In quanto rappresentante dell'utente, il MainAgent gioca il ruolo di A nell'intraprendere un piano ad alto livello; esso, tuttavia, può anche agire da servente per un altro agente MainAgent o per un ServiceAgent, in determinate condizioni; ad esempio, può accadere che uno degli utenti desideri conoscere un particolare concetto; può allora richiedere a un altro utente, e quindi al suo MainAgent, di fornire le informazioni richieste.

Possiamo considerare l'autonomia del MainAgent come una funzione in uno spazio multidimensionale. Tale funzione non è conosciuta con precisione, poiché è problematico stabilire quante siano le dimensioni e se sia possibile applicare a queste dimensioni metriche discrete o continue; per ovviare a questo problema, abbiamo effettuato delle scelte, in parte arbitrarie, su tipo e numero di dimensioni. Considereremo quindi una proiezione di tale funzione, scegliendo un numero di dimensioni e fissando una metrica discreta per tali dimensioni. Otterremo (in base a come si vuole che il sistema risponda all'utente) un insieme di valori che

rappresentano una mappatura parziale della funzione. In particolare, cercheremo di mettere in evidenza quei punti in cui i valori della funzione determinano un comportamento “interessante” (per il nostro campo d’indagine) dell’agente:

- punti in cui il livello di autonomia permette al sistema di generare un nuovo task (prevedendo le necessità dell’utente e cercando di anticiparle, comportandosi quindi in modo proattivo)
- punti in cui il livello di autonomia è insufficiente per l’esecuzione del piano originario, e ciò costringe il sistema a cambiarlo e a introdurre una richiesta all’utente per ottenere l’autorizzazione all’esecuzione di un particolare compito.

In tali condizioni limite, infatti, verrà evidenziato il modo in cui il sistema modifica esplicitamente la sua autonomia, in relazione all’ambiente e all’utente che utilizza il sistema.

2.5.1 Categorizzazione dell’autonomia

Considerando l’autonomia di un agente, abbiamo già visto che emerge la necessità di considerare diverse dimensioni [5] e diversi livelli di autonomia, con riferimento a una metrica ben definita. E’ possibile considerare un numero finito di livelli, dal punto di vista quantitativo, e un numero finito di dimensioni, dal punto di vista qualitativo.

Dato che non è possibile definire un numero massimo di dimensioni, a causa della molteplicità di ambienti e di tipologie di agenti, né un numero massimo di livelli,

poiché la granularità appropriata potrebbe essere differente al variare sia dell'ambiente che dell'agente, è necessario scegliere un sottoinsieme di dimensioni e livelli di autonomia per il particolare sistema. Tale sottoinsieme va scelto in modo da mettere in evidenza le caratteristiche peculiari che si vogliono studiare, che saranno quindi il più possibile tra loro "ortogonali", cioè indipendenti.

In questa tesi, prenderemo in considerazione le seguenti dimensioni (variazione qualitativa):

- "**Autonomia di esecuzione**": quanta autonomia ha un agente, quando deve eseguire una particolare azione (ad esempio, intraprendere un determinato compito senza richiedere conferma all'utente sia per l'esecuzione che per i parametri necessari – se mancanti, dedotti dalla base di conoscenza -)?

- "**Autonomia di comunicazione**": quanta autonomia ha un agente, quando vuole comunicare con l'utente? L'agente può prendere l'iniziativa di comunicare con l'utente in ogni momento o esistono delle constraints dettate dall'ambiente o dall'utente? Quanto deve essere invadente la comunicazione? Per questa particolare dimensione, sorge anche il problema di determinare "quanto" un certo messaggio sia invadente. Tale concetto è fondamentale per una trattazione organica, ma, mancando del background di analisi psicologica, per semplicità considereremo questa dimensione alla stregua delle altre.

- "**Autonomia di diffusione di dati personali**": l'agente può svolgere di propria iniziativa compiti che richiedono la diffusione di dati personali? Se sì, quali, e in che misura?

- "**Autonomia di sfruttamento risorse**": l'agente può utilizzare, per eseguire i propri task, risorse dell'utente (come il numero della carta di credito oppure il suo tempo – fissare appuntamenti, per esempio -)?

Per ognuna di queste dimensioni, scegliamo una suddivisione dei possibili valori in quattro categorie:

- "**null**": nessuna autonomia; il sistema deve attenersi strettamente a quanto ordinato dall'utente, e qualunque variazione al piano o dato mancante deve essere esplicitamente richiesto
- "**low**": livello di autonomia bassa; il sistema può prendere iniziative che riducano il piano originario. Per esempio, in caso di fallimenti di sottopiani, ma deve evitare di prendere iniziative; le deduzioni devono essere tutte sottoposte all'utente per approvazione
- "**middle**": livello di autonomia media; le iniziative che il sistema può prendere sono più numerose che nel caso precedente, e non deve essere immediatamente notificato all'utente che per i suoi dati sono state effettuate delle deduzioni dalla base di conoscenza
- "**high**": livello di autonomia massima; il sistema ha la più ampia libertà di azione possibile, fermi restando gli eventuali vincoli imposte dall'utente (ad esempio, limiti massimi di spesa)

2.5.2 Come l’utente interagisce con gli agenti

L’interazione dell’utente con l’agente che lo rappresenta (e che quindi maschera la complessità del sistema, presentandosi come unica entità all’utente) è basata sul concetto di task, vale a dire di azione che l’utente desidera che l’agente svolga. Uno dei Desire dell’agente è appunto “Eseguire i task dell’utente nell’ambiente opportuno”.

L’utente vede un task come “voglio che il sistema esegua questo compito”, dove il compito può essere “prenota un esame”, “prenota una vacanza”, “avvisami se ho dimenticato di comprare qualcosa al supermercato”, e via dicendo, ovviamente fornendo al sistema le informazioni essenziali per l’esecuzione di questi task. Da parte sua, il sistema opera una valutazione dei task per stabilire quali dati sono effettivamente necessari e quali possono essere dedotti (dalla storia delle interazioni passate oppure da specifici dati relativi a ciò che il sistema sa dell’utente), basando quindi le sue richieste di dati sulla conoscenza dell’utente. In tal modo, si semplificano notevolmente le procedure richieste all’utente per l’esecuzione di un task, fornendo l’equivalente di un wizard che propone come valori di default quelli che più sono vicini alle caratteristiche dell’utente. Inoltre, dato che la modellazione dell’utente è un’attività continua, perché per sua natura evolve col passare del tempo, il modello dell’utente varia tenendo conto di ciò che l’utente fa: sia il wizard per la generazione dei nuovi task che l’interfaccia utente forniscono quindi un feedback al sistema, permettendo di rendere il modello

dell’utente, e di conseguenza il ragionamento, più vicino a quello di cui l’utente ha bisogno.

2.5.3 Modalità di comportamento del MainAgent

Il livello predefinito di aiuto dell’agente MainAgent nei confronti dell’utente è, secondo la categorizzazione vista prima, quello di “**OverHelper**”: l’agente quindi esegue tutto ciò che l’utente ordina esplicitamente, e in più prende l’iniziativa di eseguire nuovi task, generati automaticamente, se “crede” che l’utente abbia necessità di tali servizi, cioè se le sue regole di ragionamento deducono dall’input implicito (cioè dalle caratteristiche dello User Model[20] e del Context Model[19], che vedremo più avanti) che uno o più nuovi task devono essere intrapresi. Tale modalità di aiuto può essere modificata durante l’utilizzo del sistema, sia automaticamente (l’agente adatta l’esecuzione dei propri task in rispetto ad alcune condizioni ambientali o dell’utente), sia con ordini espliciti dell’utente (ad esempio, l’utente può disabilitare tutti i messaggi, per un certo periodo di tempo, se è impegnato in qualche attività per cui non può essere disturbato).

2.6 Meccanismi di ragionamento autonomo

2.6.1 Definizione di ragionamento automatico

Il ragionamento automatico che considereremo in questo contesto, come già evidente dalla scelta di un agente BDI, è basato sul modello di un sistema a produzioni: vi saranno quindi regole formalizzabili come

((Condizione 1) AND (Condizione 2) AND … (Condizione n))

→

Azione.

La strategia per il raggiungimento del goal che utilizzeremo è volutamente semplice. Questa scelta è motivata dalla complessità computazionale dell'intero sistema D-Me e del framework JADE (che vedremo in dettaglio nel Capitolo 3), unita alla penalizzazione di prestazioni dovuta all'utilizzo del linguaggio Java e quindi alla necessità dell'utilizzo di una Virtual Machine.

Tuttavia, il progetto del sistema non preclude l'utilizzo di una strategia più complessa; è sufficiente modificare la classe che implementa la strategia di ragionamento (ThinkBehaviour) e le classi che rappresentano i Desires e le Intentions (Desire e Intention rispettivamente).

Abbiamo optato per una strategia a un solo passo. Le regole di ragionamento (le Intentions dell'agente) prevedono la possibilità di realizzare il Desire (o goal) direttamente, utilizzando come unico operatore la sequenza, e senza consentire la nidificazione delle Intentions. Inoltre, non è previsto un meccanismo di backtracking a livello di Intentions.

A livello di descrizione dei task, invece, è possibile rappresentare un task complesso come una FSM (Finite State Machine, macchina a stati finiti), per la quale ogni stato è rappresentato dall'esecuzione di un sottotask, che a sua volta può essere un task complesso o un task elementare.

A seconda dell'esito, positivo o negativo, di un sottotask, il task principale può prevedere una strada alternativa per raggiungere il proprio scopo. Attualmente, il sistema non prevede la possibilità che la struttura dei task possa essere modificata dinamicamente (creando quindi una vera struttura di ragionamento, in sostituzione all'uso di piani preordinati e statici), ma è possibile sviluppare il sistema in tal senso, adattando la classe CompositeTaskActuator.

Vediamo la struttura gerarchica dei vari tipi di task: (Figura 2.1)

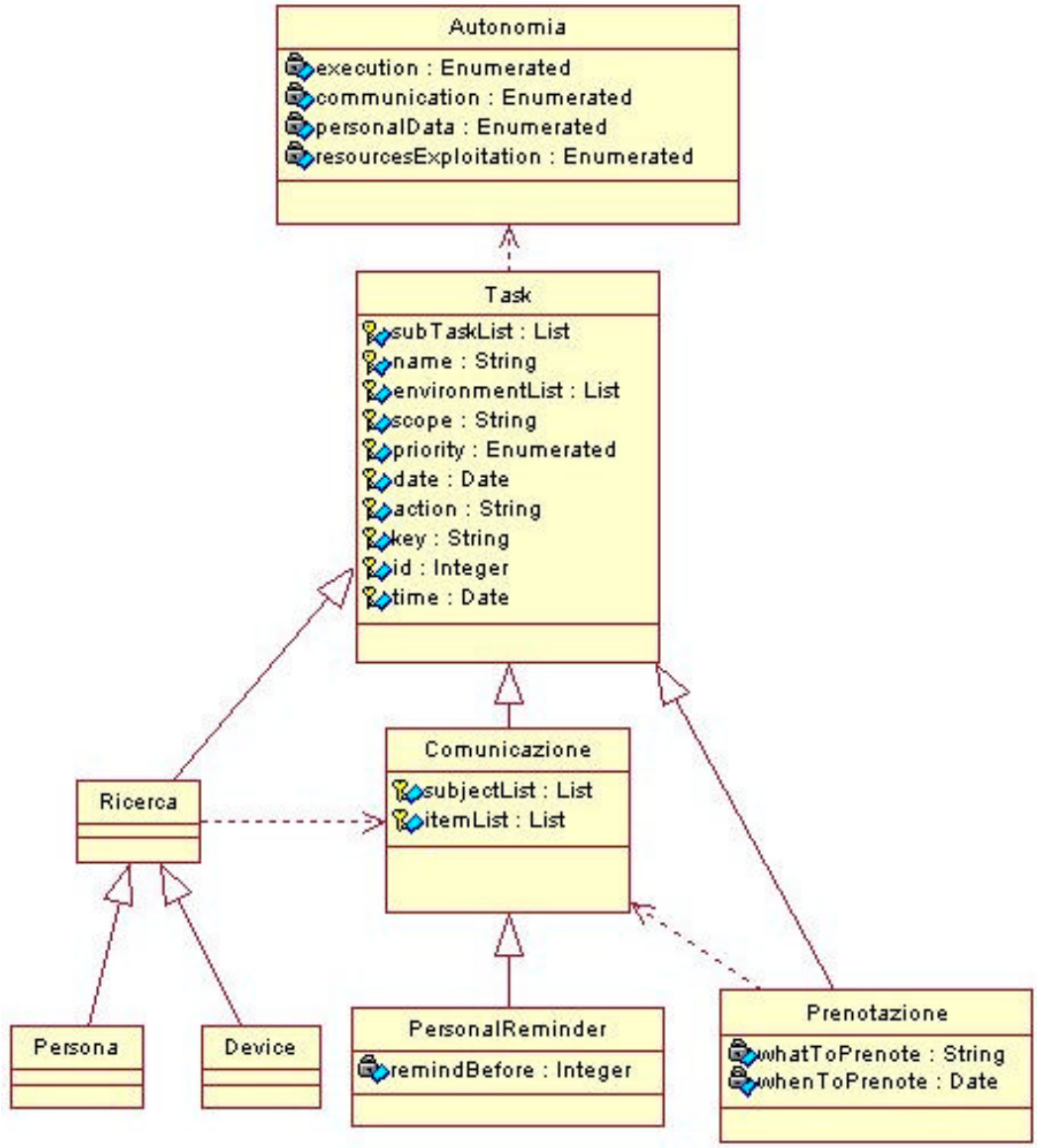


Figura 2.1

In particolare, la lista SubTaskList è una lista di Task, come è più evidente dalla rappresentazione XML di un task (Figura 2.2):

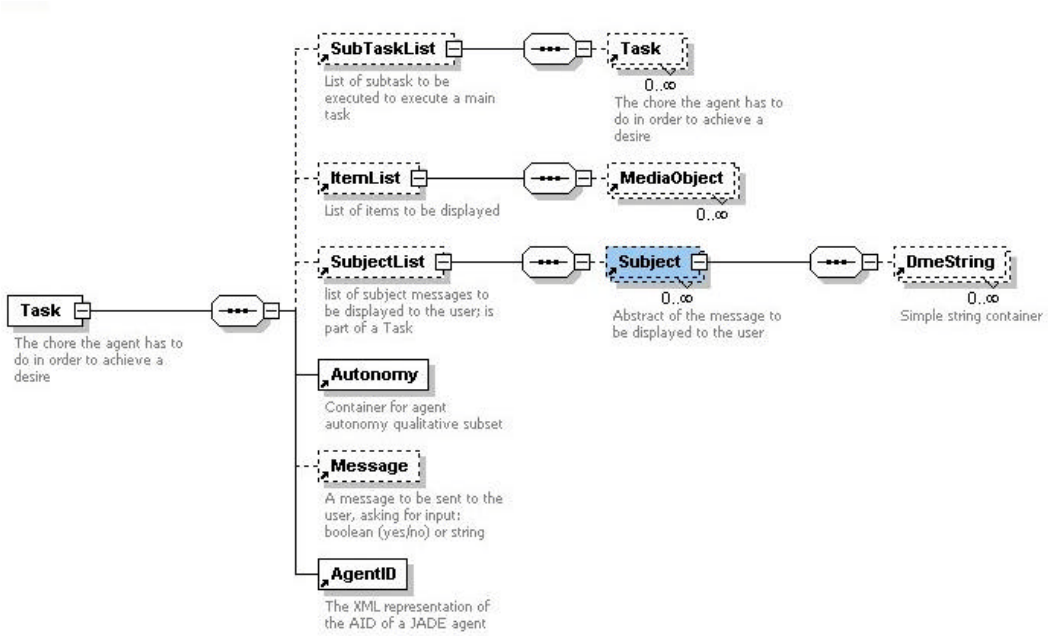


Figura 2.2

La rappresentazione XML di un task è un flattening (appiattimento) della gerarchia, vale a dire che il record XML possiede tutti i campi della classe padre Task, più gli attributi di tutte le classi figlie; tali attributi hanno valore solo se il tipo di task, identificato dall'attributo "**action**", è quello corrispondente alla sottoclasse in cui quel valore esiste. Ciò è stato fatto per semplificare sia la progettazione della struttura del file XML che la realizzazione delle classi Java e delle relative classi ontologiche.

Sfruttando tale rappresentazione, è possibile esprimere il piano associato a un task come grafo, trasformato in una lista e memorizzato con questo formalismo XML.

2.6.2 Regole per la descrizione del grafo di una FSM

Il grafo di un task è soggetto ai seguenti vincoli:

- L'insieme degli stati componenti la FSM è contenuto in una lista memorizzata all'interno del task (la subTaskList);
- Lo stato iniziale della FSM deve essere il primo elemento della lista, per identificare il punto di inizio in modo semplice;
- Ogni stato nella lista ha due attributi, "nextOk" e "nextError"; entrambi contengono un valore "long", che corrisponde all'identificativo di un altro stato nella stessa lista. Essi rappresentano la transizione dallo stato precedente (l'elemento Task di cui fanno parte) allo stato successivo. "nextOk" contiene l'identificativo del sottotask successivo da eseguie quando il task corrente viene eseguito con successo, "nextError" l'identificativo del sottotask alternativo, quando il task corrente fallisce l'esecuzione. Di conseguenza, ci possono essere solo due transizioni in uscita da uno stato, una etichettata con Ok e memorizzata in nextOk, e l'altra etichettata con Error e memorizzata in nextError;
- uno stato i cui due successori sono entrambi 0 (cioè non puntano a nessuno stato valido, poiché gli identificatori dei task sono tutti valori strettamente positivi) è uno stato finale, da cui si va solo nello stato di conclusione del behaviour;
- Lo stato iniziale e quello finale del behaviour FSM, come definito nel framework, non sono rappresentati nel task, in quanto identici per tutti i possibili grafi;

Questo consente di realizzare piani complessi, senza modifiche al codice, semplicemente scrivendo un file XML adatto. In particolare, è possibile tradurre un grafo di tipo AND-OR in questo formalismo (vedi figura 2.3).

In questo esempio, è rappresentato il grafo, costruito rispettando il formalismo appena descritto, corrispondente al grafo AND-OR che si può esprimere come

((SottoTask A) and (SottoTask C) and (SottoTask D)) or (SottoTask B)

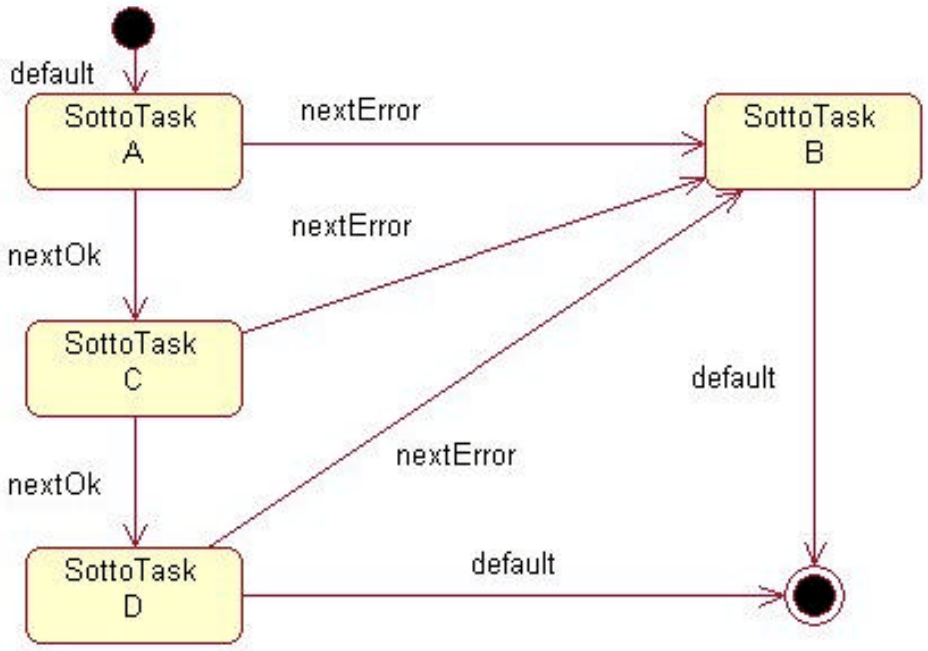


Figura 2.3

2.6.3 Influenza dell'autonomia sul ragionamento

L'introduzione dell'autonomia nel ragionamento automatico inserisce un nuovo vincolo nel database globale: le Intentions di un agente, il cui grado di autonomia è variabile in più dimensioni indipendenti, devono tener conto esplicitamente di tale grado. Esse perciò saranno del tipo:

IF

((Autonomia(Categoria) = Livello) AND

(Condizione 1) AND (Condizione 2) AND … (Condizione N))

THEN

Azione

Per semplificare il processo di matching, è possibile suddividere le Intentions in sottoinsiemi aventi il medesimo livello di autonomia. In tal modo, al momento di determinare il "conflict set" (l'insieme delle Intentions applicabili a una data istanza dell'insieme dei Beliefs), si prendono in considerazione solo quelle Intentions appartenenti agli insiemi la cui autonomia richiesta è minore o uguale a quella del task in quel momento in esecuzione.

Per esempio, avendo due Intentions:

```
<Intention name="Get Contextualized User Model" slotName="intention">
        <Precondition slotName="precondition" value="dme.onto.um.UserIdentify"/>
        <Precondition slotName="precondition" value="dme.onto.architecture.AgentID"/>
        <Obtain slotName="obtain" value="dme.onto.um.User"/>
```

```
        </Intention>

        <Intention name="Find a list of suitable tasks for the available environment"
slotName="intention">
            <Precondition slotName="precondition" value="dme.onto.um.UserIdentify"/>
            <Precondition slotName="precondition" value="dme.onto.architecture.AgentID"/>
            <Obtain slotName="obtain" value="dme.onto.architecture.ListServiceTask"/>
        </Intention>
```

si nota che entrambe hanno le stesse precondizioni; se esse richiedono differenti livelli di autonomia per l'esecuzione, il conflict set viene ridotto alla Intention il cui livello di autonomia è compatibile col livello di autonomia dell'agente.
In questo modo, se l'agente possiede un livello di autonomia elevato, potrà scegliere di eseguire una qualunque Intention, mentre se il suo grado di autonomia è basso il conflict set sarà creato a partire da un set di Intentions più ristretto.

Considerando il Desire caratteristico dell'agente D-Me, che è quello di "Eseguire i task dell'utente nell'ambiente adatto", risulta conveniente considerare una diversa autonomia per ogni task, data la natura eterogenea degli stessi. E' infatti inconcepibile per l'utente che per un task come "stasera devi andare al cinema" l'agente abbia la stessa autonomia di azione che per il task "prenota l'esame di Programmazione". E' evidente che il secondo task è di gran lunga più importante per l'utente, perciò il sistema potrà prendere in considerazione molti più modi di portare a termine il task. Avrà quindi maggiore autonomia, in particolare maggiore autonomia di comunicazione e maggiore autonomia di azione.

Di conseguenza, nell'implementazione del sistema ogni task (e, per ricorsività, ogni sottotask) possiede una classe Autonomia, che funge da contenitore per i valori dell'autonomia di quel particolare task; possiamo considerarle come le coordinate di quel task nello spazio dell'autonomia. Tale classe serve per consentire all'agente di ragionare sul task, nel momento in cui il piano richiede che il task venga eseguito.

Supponendo, ad esempio, che l'utente abbia inserito un task che richiede al MainAgent di intraprendere una certa azione, che però richiede un livello di autonomia superiore a quella che l'utente ha concesso per il task principale, allora l'azione non può essere eseguita; a questo punto, le alternative possibili per il MainAgent sono le seguenti:

- "**Richiesta all'utente di modificare il livello di autonomia dell'agente per il task corrente**": in questo caso, l'agente sta sfruttando la sua metaautonomia [5], negoziando con l'utente una modifica della propria autonomia;
- "**Modifica del livello di aiuto fornito all'utente**": in tal caso, il MainAgent interrompe l'esecuzione di quella parte del piano e continua seguendo la strada alternativa, se esiste; altrimenti termina il compito. Si tratta quindi di un cambiamento da OverHelper (che è la strategia standard del sistema, come accennato in precedenza) a SubHelper, poiché l'agente si limita ad eseguire una porzione del piano originale.
- "**Richiesta all'utente di modificare il livello di autonomia del tipo di sottotask che sta eseguendo**": si tratta, in questo caso, di modificare l'interpretazione del mondo dell'agente. Questo può essere utile quando

l’utente si accorge che l’azione di un particolare sottotask è stata valutata male; nel caso specifico, è stata definita un’autonomia di esecuzione troppo alta, restringendo inutilmente la libertà d’azione dell’agente. La modifica dell’autonomia richiesta dal sottotask permette all’agente di intraprenderlo senza chiedere conferma all’utente, rendendolo quindi meno intrusivo.

2.6.4 Feedback: Incidenza del comportamento dell’utente sull’autonomia

Bisogna prendere in considerazione altri aspetti del ragionamento:

- **Accettazione** da parte dell’utente delle iniziative dell’agente
- Capacità dell’agente di **eseguire** ciò che l’utente desidera
- **Conflitti** tra Desires dell’agente

Questi tre punti vengono presi in esame e formalizzati sempre in [5]. A tale proposito vengono definiti vari livelli di aiuto dell’agente in base alla personalità dell’utente (e quindi al suo tipo di delega).

Si distinguono gli utenti secondo le categorie:

- “**sempre delegante**”: l’utente è “pigro”, delega i suoi task all’agente anche quando potrebbe svolgerli da solo
- “**delegante per necessità**”: l’utente delega un task all’agente solo quando non è in grado di eseguirlo personalmente
- “**non delegante**”: l’utente non delega mai task all’agente

e gli agenti come:

- "**ipercooperativo**": l'agente esegue sempre i task che l'utente gli affida, e interviene quando crede che l'utente possa aver necessità di un certo task, o quando pensa che un certo task sia gradito all'utente (anche quando l'utente è in grado di svolgere da solo tale task)
- "**benevolente**": l'agente esegue un task sia quando l'utente glielo ordina, sia quando pensa che l'utente non sia in grado di eseguirlo da solo
- "**aiutante**": l'agente esegue sempre i task dell'utente, ma non prende iniziative
- "**egoista**": l'agente non esegue mai i task dell'utente

Si distinguono diverse coppie possibili utente/agente; scartiamo le possibilità chiaramente prive di interesse (come utente "non delegante" o agente "egoista"), che nel nostro sistema non hanno riscontro e sarebbero di poca rilevanza, in quanto un utente non delegante lascerebbe il sistema inattivo, e viceversa un agente "egoista" sarebbe un pessimo rappresentante dell'utente.

Esamineremo quindi le coppie:

Utente	Agente
"sempre delegante"	"ipercooperativo"
"delegante per necessità"	"benevolente" oppure "aiutante"

In tutti i casi, il feedback dell'utente (vale a dire il modo in cui l'utente "approva" il comportamento del sistema) può far cambiare categoria all'agente: per esempio,

un agente “ipercooperativo” può diventare un agente “aiutante” se il comportamento dell’utente evidenzia che l’utente è un “delegante per necessità”; l’autonomia di azione dell’agente diminuisce in conseguenza di questo, impedendo all’agente di prendere iniziative.

2.7 Meccanismi di feedback

Prenderemo in esame due tipi possibili di feedback:

- “**Feedback positivo**”: tale tipo di feedback si realizza nel momento in cui l’utente, avendo davanti a sé l’interfaccia che mostra come l’agente abbia preso un’iniziativa, la visualizza e l’approva.
- “**Feedback negativo**”: tale tipo di feedback avviene quando, sempre visualizzando le attività intraprese autonomamente dall’agente, l’utente ne trova una o più che non vuole vengano eseguite. In tal caso, è possibile sospendere l’attività o annullarla.

Nel primo caso, l’agente ritiene che la generazione del task sia stata una scelta giusta, per cui le regole di decisione restano invariate. Nel secondo caso, invece, è necessario modificare i criteri di scelta delle regole.

2.7.1 Modifica delle regole di inferenza

Considerando dal punto di vista dell'agente il feedback dell'utente, tale feedback si traduce in una modifica ai valori dell'autonomia. Vi sono due possibili modi di variare l'autonomia di un agente, relativamente all'inferenza di nuovi task:

- Si può modificare l'autonomia di esecuzione delle regole di inferenza per l'agente; tale modifica ha l'effetto di rendere l'agente più o meno autonomo, riguardo alla generazione di nuovi task, ma non distingue tra una regola e l'altra;
- Si può considerare l'autonomia necessaria per l'esecuzione di ogni singola regola di inferenza; per far ciò, bisogna definire l'autonomia di ogni regola come "livello minimo di autonomia dell'agente perché la regola possa essere applicata". In tal modo, l'agente può eseguire, in un particolare momento, solo quelle regole la cui autonomia minima è compatibile con l'autonomia di cui dispone l'agente in quell'istante di tempo. A un feedback positivo o negativo corrisponde rispettivamente un abbassamento o un innalzamento dell'autonomia minima della regola la cui attuazione ha causato il feedback. Quando il livello di autonomia necessario per l'esecuzione di una regola è massimo, un ulteriore feedback negativo da parte dell'utente ha come conseguenza la rimozione della regola dalle regole che l'agente usa normalmente per l'inferenza. Perché tale regola possa essere reintrodotta nel sistema, è necessario un intervento esplicito dell'utente (che deve modificare manualmente l'autonomia necessaria per l'esecuzione di una regola).

Nell’implementazione del sistema, abbiamo scelto la seconda alternativa, in quanto essa permette un controllo più fine sul comportamento dell’agente, permettendo di agire separatamente su ogni regola.

2.7.2 Modifica del profilo utente

L’agente deve sempre permettere all’utente di verificare i dati che vengono inferiti sulla personalità di delega, quindi sui livelli di autonomia; questo perché tale trasparenza è necessaria perché l’utente mantenga il controllo dell’interazione e quindi possa fidarsi dell’agente e delegare un qualsiasi task.

Navigando questi dati, l’utente può confermare o negare le inferenze dell’agente, o ancora modificare alcuni dei dati precedentemente asseriti; in tal modo, egli fornisce sia un feedback positivo che uno negativo: positivo sulle inferenze che sono giuste, negativo su quelle sbagliate.

In tal modo, viene rivista l’autonomia delle regole di decisione che hanno portato alle inferenze in questione. La tematica della modellazione dell’utente è stata già descritta in [20].

Capitolo 3: Caratteristiche strutturali

3.1 Piattaforma di implementazione

3.1.1 JADE: descrizione generale

JADE (Java Agent DEvelopment Framework) [12] è un framework che rende possibile la creazione e la comunicazione tra agenti software, fornendo sia una shell per l'avvio e la gestione degli agenti, che protocolli per la comunicazione (conformi allo standard FIPA [13], Foundation for Intelligent Physical Agents). Inoltre, sono disponibili tool per l'analisi degli agenti presenti e il debug. E' sviluppato dal TILAB (Telecom Italia Lab, ex CSELT) [16] ed è distribuito sotto licenza LGPL (Lesser GNU Public License).

L'intero framework è realizzato in Java. Esso fornisce i componenti standard per la gestione della piattaforma:

- ACC (Agent Communication Channel): canale per la comunicazione interpiattaforma e intrapiattaforma
- AMS (Agent Management System): agente il cui compito è gestire gli altri agenti (servizio di naming)
- DF (Directory Facilitator): agente che si occupa di registrare i servizi offerti dagli altri agenti (servizio di yellow pages)

E’ anche possibile per un agente muoversi all’interno della piattaforma (che è composta di diversi Container, non necessariamente residenti sullo stesso host). Lo scambio di informazioni tra agenti avviene attraverso l’uso di messaggi, codificati secondo gli standard FIPA, definiti ACL messages.

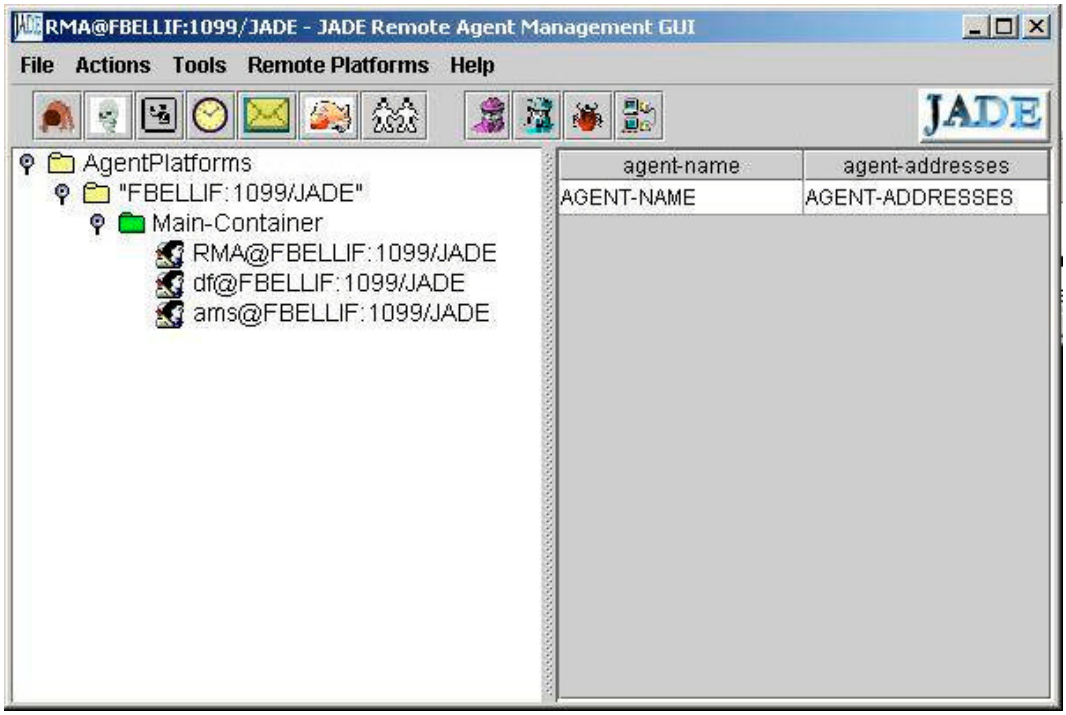


Figura 3.1

In figura 3.1 è possibile vedere un esempio dell’interfaccia grafica del framework JADE. Da questa interfaccia è possibile attivare o eliminare un agente in un particolare container, visualizzare l’elenco delle piattaforme, dei container e degli agenti presenti, oppure lanciare gli strumenti di debug.

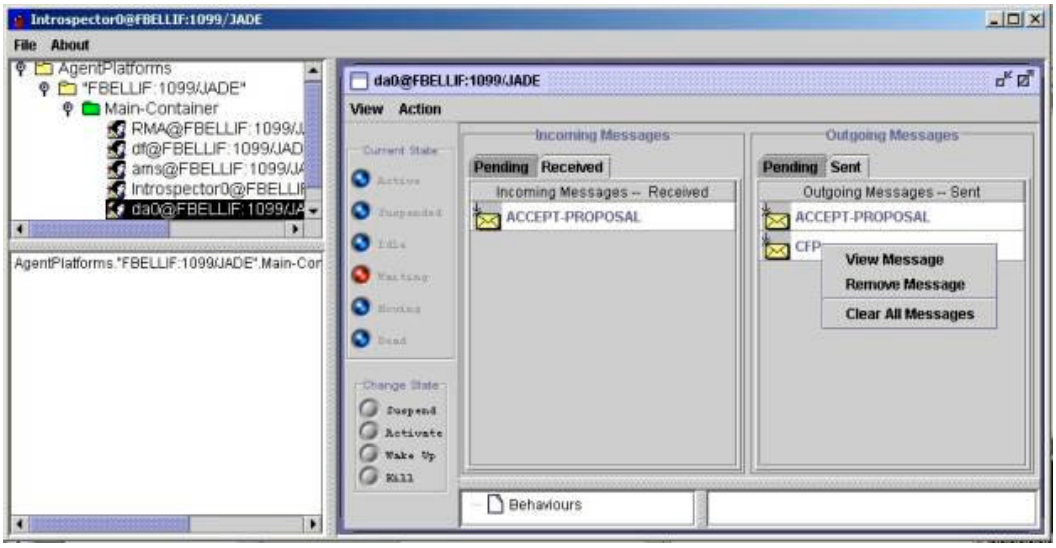


Figura 3.2

In figura 3.2 si può osservare la schermata dell'Introspector. Esso è un tool che consente di visualizzare tutti i behaviour di un agente e tutti i messaggi da esso ricevuti o inviati, compresi quelli ancora in attesa di essere letti dall'agente. E' utile per verificare se i behaviour vengono caricati ed eseguiti nel modo previsto.

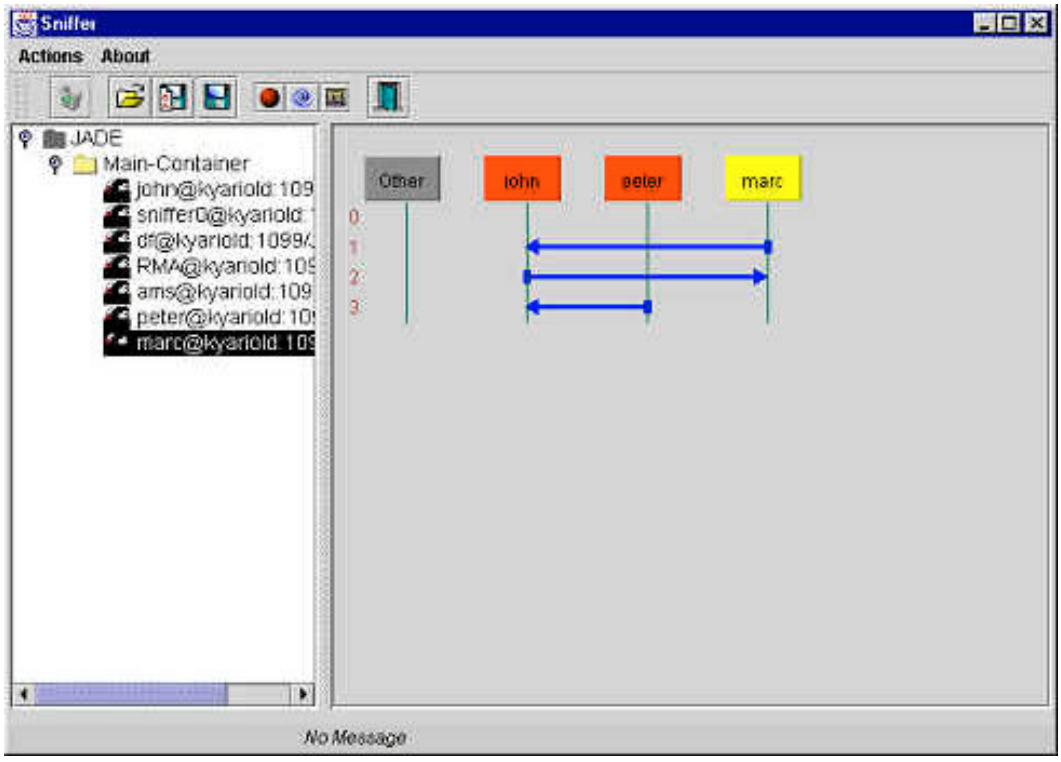


Figura 3.3

In figura 3.3 si vede la schermata dello Sniffer. Tale tool è in grado di “spiare” tutti i messaggi che vengono scambiati dagli agenti sulla piattaforma. Scegliendo un sottoinsieme di agenti, è possibile verificare la corretta esecuzione dei protocolli di comunicazione tra agenti, nonché visualizzare il contenuto dei messaggi, per verificarne la correttezza.

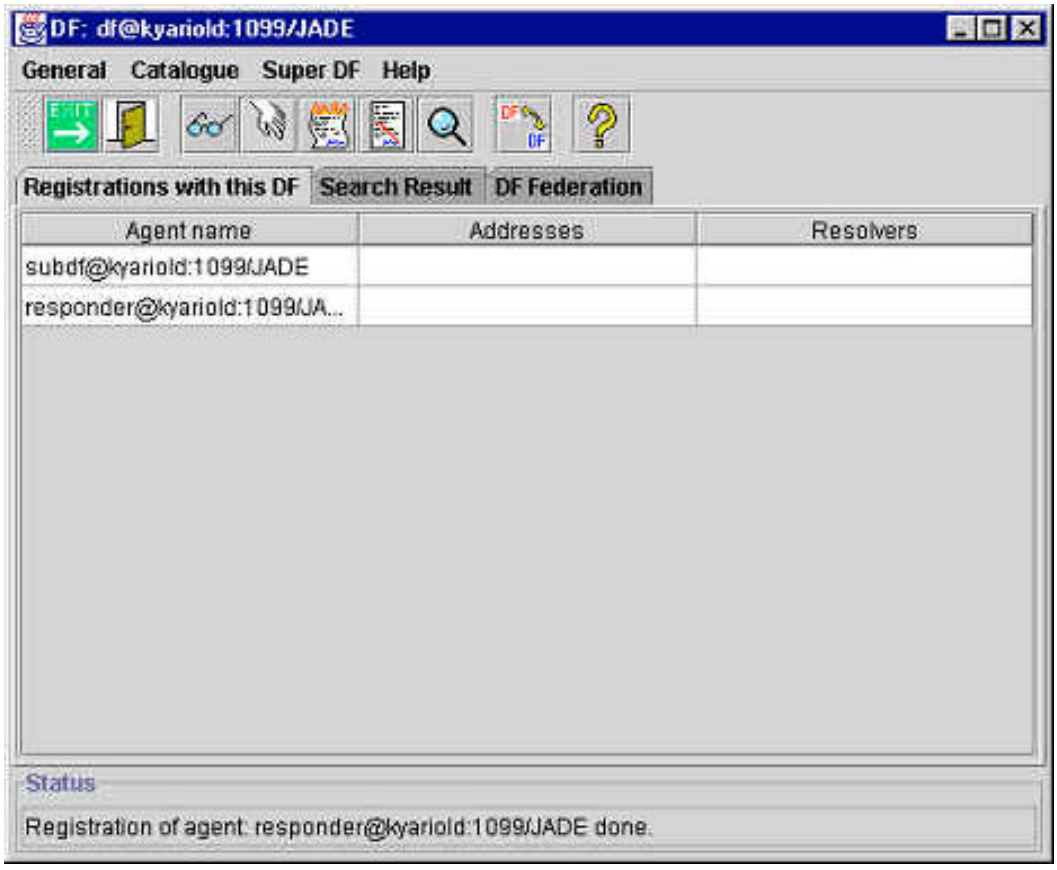


Figura 3.4

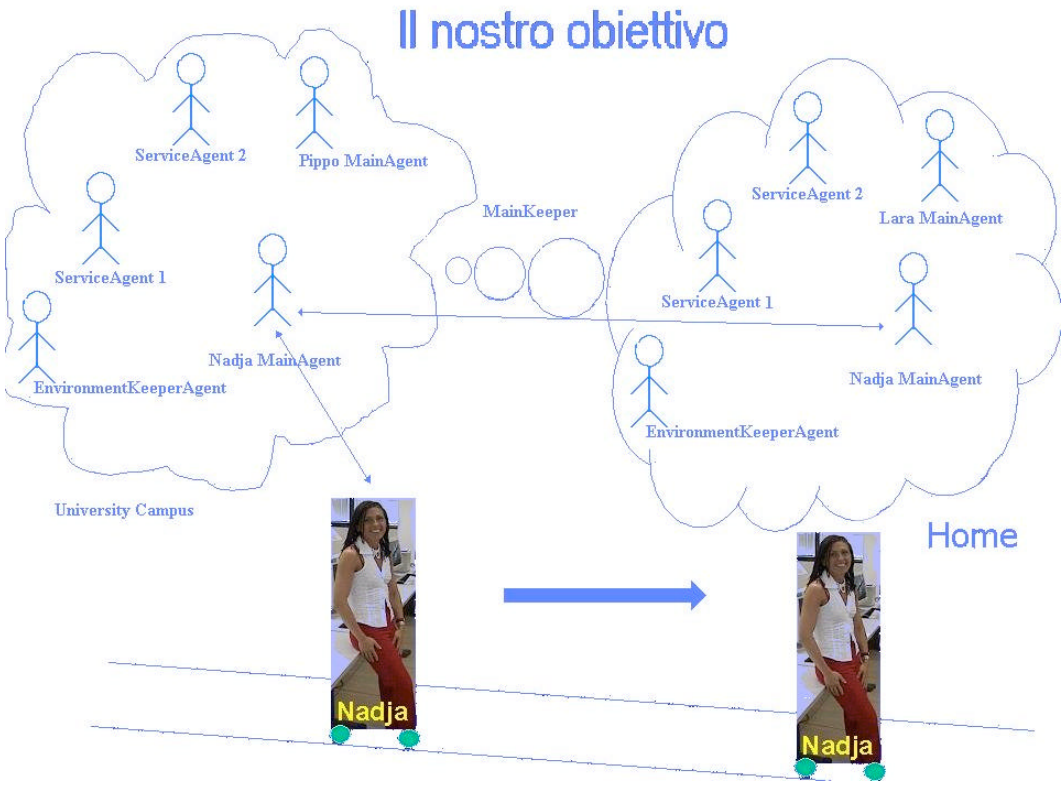
In figura 3.4 si può osservare l'interfaccia grafica del DF, che è uno degli agenti che obbligatoriamente devono essere presenti su una piattaforma. Tale agente fornisce il servizio di Yellow Pages, cioè fornisce un servizio di registrazione per tutti gli agenti che offrono servizi presenti sulla piattaforma.

3.1.2 LEAP: descrizione generale

LEAP (Lightweight Extensible Agent Platform)[http://leap.crm-paris.com] è una piattaforma a footprint ridotto, quindi con richieste minime di risorse hardware, indipendente dal sistema operativo, e conforme a FIPA. Tale piattaforma consiste

in un adattamento della piattaforma JADE (da cui mutua parte del codice sorgente) alle specifiche delle macchine virtuali Pjava (PersonalJava) e MIDP (Mobile Information Device Profile), rilasciate da Sun Microsystems [www.sun.com]; l'utilizzo di tale framework alternativo consente di far funzionare gli agenti Java che lavorano con JADE anche su dispositivo palmare. In particolare, l'utilizzo della versione di LEAP compilata per Personal Java (o Pjava) è pensata per essere utilizzata su dispositivi palmari del tipo Compaq Ipaq[www.compaq.it] o equivalenti, con virtual machine come Jeode[www.insignia.com] oppure con la virtual machine di riferimento della Sun. Inoltre, è possibile lavorare con i rispettivi emulatori e ambienti di simulazione (in particolare J2ME Wireless Toolkit per la versione MIDP e Personal Java Environment Emulation per la versione Pjava).

3.2 D-Me: descrizione generale



Il sistema D-Me consiste di un insieme di agenti, organizzati per Environment. Un agente, il MainAgent, rappresenta l'utente del sistema: le varie istanze del MainAgent possono essere residenti in vari container, anche su piattaforme remote, e registrano la loro presenza in un ben definito Environment.

L'approccio che abbiamo seguito prevede che l'utente che desidera utilizzare i servizi di un Environment si sposti (fisicamente o logicamente) in questo ambiente, spostando quindi il suo MainAgent. L'Environment corrisponde, nella metafora del sistema, a un luogo fisico oppure logico, come ad esempio la biblioteca, il supermercato, il pub, fisicamente realizzato come un ambiente

“smart”, cioè dotato di tecnologia Ubiquitous computing, con opportuni devices, come sensori, microfoni, schermi a parete; dato che simili ambienti esistono solo a livello sperimentale, gli Environment sono realizzati in questo sistema come entità software, composte quindi solo da agenti software.

In particolare, un Environment è caratterizzato da:

- “**Key**”: il nome dell’Environment
- “**EnvironmentKeeperAgent**”: un agente specifico per l’Environment
- Insieme dei “**Service**” afferenti all’Environment.
- Insieme dei “**MainAgent**” registrati presso un EnvironmentKeeperAgent; tale insieme varia nel tempo, quando i MainAgent migrano da un ambiente all’altro.

I Service sono essi stessi agenti, che forniscono particolari servizi ad altri agenti che li richiedano. Essi si registrano, al momento della loro creazione, col Keeper di ambiente al quale afferiscono. In dipendenza dalla natura dei servizi offerti, i ServiceAgent possono essere accessibili solo agli agenti presenti nell’Environment, oppure essere disponibili per qualunque agente che sappia come contattarli. A tal fine, i vari Environment sono registrati presso un **MainKeeper**, il cui scopo è tener traccia dei Keeper di ambiente presenti e fornirne l’indirizzo ai richiedenti. Il MainKeeper è l’unico agente il cui indirizzo deve essere noto a priori per il generico agente MainAgent, nel momento in cui questo viene attivato. Successivamente, esso ricerca l’Environment in cui è situato e si collega al relativo Keeper, in modo da poter svolgere le sue funzioni.

E’ possibile richiedere informazioni e servizi non solo ai Service di un ambiente, ma anche agli altri MainAgent presenti nel sistema: essi sono infatti registrati presso il Keeper in modo simile a quanto avviene per i Service, e possono essere interrogati alla stessa maniera. Data la natura potenzialmente eterogenea degli agenti MainAgent (essi sono specializzazioni dell’agente standard di JADE, ma non ci sono limiti alla loro differenziazione), e al particolare utente di cui sono rappresentanti, i MainAgent possono decidere se rispondere a una particolare richiesta degli altri agenti o no, a differenza dei Service; questo perché, per fare un esempio, un utente può non aver delegato al suo agente la diffusione dei suoi dati personali. Pertanto, alla richiesta di un altro agente, del tipo “chi sei?”, il MainAgent non è autorizzato a rispondere.

Il MainAgent ha bisogno, per eseguire le sue funzioni e prendere le decisioni necessarie, di due tipi di informazioni: informazioni sull’utente che esso rappresenta (User Modeling Components, vedi [20]) e informazioni sul contesto fisico in cui è immerso (Context Modeling Components, vedi [19]).

Le informazioni sull’utente sono gestite da un ulteriore agente specializzato, l’**UMKeeper**. Tale agente ha come compito quello di conservare in modo permanente i dati statici dell’utente (nome, indirizzo) e quelli riguardanti la personalità, le capacità, le conoscenze dell’utente. Tali dati vengono immessi direttamente dall’utente, tramite un’interfaccia grafica, o dedotti dal MainAgent, analizzando l’interazione con l’utente e traendo le conclusioni sulla base di un set di regole specializzate. E’ responsabilità dell’UMKeeper aggiornare i dati memorizzati e fornirli su richiesta del MainAgent. Esso può risiedere sul dispositivo portatile dell’utente, oppure essere localizzato fisicamente su un server

remoto. In tal caso, è possibile che esso sia responsabile della memorizzazione dei dati di più MainAgent, agendo quindi da database centralizzato.

Le informazioni sul contesto sono fornite da altri agenti, che rappresentano i sensori del sistema: essi incapsulano i dispositivi fisici, e forniscono informazioni sulla posizione dell'utente, l'attività svolta, orario, data, tempo meteorologico, lo stato emotivo.

Poiché in questa versione del prototipo D-Me mancano dispositivi reali in grado di fornire queste informazioni, se non in modo parziale, tali agenti ricevono gli input da trasmettere all'utente da un ulteriore agente, il **DynamicAgent**, il quale fornisce all'utente un'interfaccia grafica che permette di simulare condizioni emotive e ambientali diverse, in modo da testare il comportamento del sistema al variare dell'ambiente.

Tali agenti non sono parte dell'Environment, ma sono specifici per ogni MainAgent; esiste quindi un'istanza di essi per ogni istanza di MainAgent.

Esiste poi l'agente di interfaccia del MainAgent (**InterfaceAgent**), che si occupa di fornire l'interfaccia utente vera e propria; esso esegue tutti in compiti di input/output da e verso l'utente, mostrando i risultati dei vari task e inviando al MainAgent gli ordini dell'utente e i dati richiesti allo stesso. Esso fa da tramite anche per tutti gli eventi di interazione con l'utente, vale a dire per il feedback (che fornisce al MainAgent le informazioni necessarie per inferire dati sul profilo utente, come le preferenze, gli interessi e via dicendo); inoltre, funge da monitor per l'attività corrente del MainAgent, informando l'utente sui task attualmente in esecuzione. L'InterfaceAgent si occupa anche di nascondere le differenze tra le varie interfacce reali che l'utente può usare: in particolare, esso può gestire

l'interfaccia su PC e su palmare (PersonalJava o MIDP); quest'ultima soffre tuttavia di notevoli limitazioni, dovute alle caratteristiche proprie dei dispositivi e delle virtual machine JAVA disponibili, pertanto non è possibile eseguire la stessa varietà di compiti sulle varie interfacce.
I dati forniti all'utente dall'InterfaceAgent e la comunicazione tra tale agente e il MainAgent sono correlati strettamente al concetto di Task e di ToDoList.

3.2.1 La ToDoList

La ToDoList è l'insieme dei task che il MainAgent deve eseguire; essa è costituita da una lista di task, ordinati per orario di esecuzione e priorità, i quali vengono eseguiti nell'ambiente idoneo. La creazione e l'esecuzione di tali task è compito del MainAgent, mentre l'InterfaceAgent si occupa di mostrare all'utente:

- la lista di task che il sistema deve ancora eseguire
- il sottoinsieme di task che sono attualmente in esecuzione
- i messaggi di output diretti all'utente (che sono particolari task la cui esecuzione è decisa dal MainAgent e delegata all'InterfaceAgent)

Ogni volta che un nuovo task viene inserito nella ToDoList, oppure vengono apportate modifiche ai task già presenti, il MainAgent aggiunge i task alla lista e inizia un altro ciclo di thinking: viene attivato un Desire di esecuzione dei task, e l'agente esegue le Intentions il cui scopo è trovare ed eseguire i task compatibili con l'environment in cui l'agente si trova. Vedremo questi meccanismi più in dettaglio nel prossimo capitolo.

Capitolo 4: Architettura D-Me

4.1 Thinking: ontologie e pensiero

4.1.1 Le ontologie in JADE

In JADE, le ontologie rivestono il ruolo di mediatori tra agenti. Esse sono delle definizioni di predicati, di concetti (elementari e composti) e di azioni degli agenti. Gli agenti basano la comunicazione su di esse: in un ACLMessage (messaggio tra agenti che usa il linguaggio ACL definito da FIPA), infatti, viene sempre inserita una ben definita classe ontologica di riferimento, e il contenuto del messaggio deve essere l'istanza di un oggetto a cui descrizione astratta è contenuta nella classe ontologica.

Una classe ontologica è una particolare classe Java, estensione della classe jade.content.onto.Ontology, che contiene la descrizione astratta di altre classi. Tale descrizione prende il nome di ObjectSchema. Essa può essere specializzata per descrivere:

- "**PredicateSchema**": un predicato, del tipo (Padre(padre, figlio))

- “**AgentActionSchema**”: un’azione di un agente, come (Search(agente, argomento))
- “**ConceptSchema**”: un concetto elementare o composto, come ad esempio una lista di nomi, un record che contiene i dati relativi ad una persona, o un tipo più complesso

La classe ontologica contiene inoltre un riferimento alla classe Java che implementa l’ObjectSchema; questo consente di istanziare oggetti reali che abbiano le caratteristiche degli oggetti descritti a livello astratto.

Perché due agenti possano comunicare, è necessario che essi condividano le stesse classi ontologiche (limitatamente a quelle utilizzate nella comunicazione) e abbiano a disposizione un’implementazione degli oggetti descritti a livello astratto. Vi è un grado di libertà nell’implementazione, in modo che non sia necessario che le stesse identiche implementazioni siano in comune tra due agenti; inoltre, è possibile modificare l’implementazione, aggiungendo, per esempio, metodi o attributi, senza che venga persa la possibilità di comunicare con un altro agente, ovviamente in modo limitato alla descrizione astratta. Essa corrisponde, quindi, alla definizione di “che cosa” un agente intende quando utilizza un oggetto di un certo tipo per comunicare, definendo una semantica per la comunicazione tra agenti.

4.1.2 Le ontologie in D-Me

Le ontologie utilizzate nel sistema D-Me si dividono in categorie:

- "**architecture**": si tratta della classe ontologica di base dell'architettura; essa contiene le descrizioni necessarie per scambiare tra agenti stringhe, descrizioni di ambiente a livello di scope, descrizioni di task, descrizioni di Desires e Intentions, fondamentali per implementare il MainAgent secondo l'approccio BDI già visto. Non sono descritti esplicitamente i Beliefs, in quanto si tratta delle conoscenze dell'agente, composte da dati di vario tipo, ognuno rappresentato nell'opportuna classe ontologica.
- "**cm**": questa classe ontologica descrive in dettaglio tutti i tipi di dato che l'agente usa per descrivere il contesto in cui si trova: tipo di ambiente, scope dell'ambiente, caratteristiche fisiche, posizione dell'utente rispetto all'ambiente
- "**um**": questa classe ontologica descrive tutti i tipi di dato che possono essere scambiati con l'agente remoto per lo user modeling, l'UMKeeper, e con gli agenti di personalizzazione interni agli Environment, quando disponibili (EnviroUMAgent): si tratta dei dati base sull'utente (nome, cognome, età, sesso…) e di informazioni sulla sua personalità, sui suoi interessi, sulle sue abilità ed eventuali inabilità fisiche, che possano essere di interesse per il sistema.
- "**resultpresentation**": questa classe ontologica contiene la descrizione dei risultati dei Service esterni, vale a dire la descrizione di ciò che i ServiceAgent possono restituire all'agente.

Per ognuna di tali classi, sono definite delle classi Java, che implementano la descrizione astratta contenuta nelle classi ontologiche. In particolare, tali classi rispettano la seguente gerarchia (Figura 4.1):

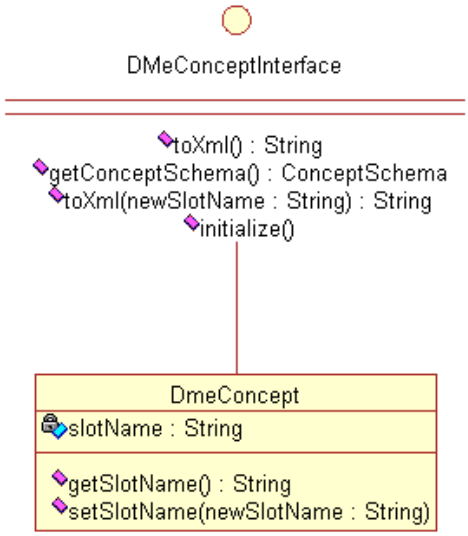


Figura 4.1

Tutte le classi Java che vengono utilizzate nel sistema D-Me sono specializzazione di DmeConcept; rispetto alle tipiche ontologie JADE, esse hanno i metodi mostrati in figura 4.1, le cui funzioni sono le seguenti:

- "**toXml()**" e "**toXml(newSlotname)**": entrambi i metodi restituiscono la serializzazione XML dell'oggetto, la prima con l'attributo "slotName" originale e la seconda con un attributo "slotName" definito dal chiamante; tali metodi permettono di scrivere facilmente un oggetto in un file XML (tipicamente, il file che contiene i dati particolari di ogni agente o il file che contiene la ToDoList, come descritto in [11] e nel capitolo 5)

- "**initialize()**" consente di inizializzare tutti i campi obbligatori per un certo oggetto; esso è distinto dal costruttore poiché il costruttore senza parametri viene utilizzato dal framework al momento della trasmissione di un oggetto a mezzo di un messaggio ACL, e l'inizializzazione all'interno del costruttore vuoto produrrebbe un overhead inutile.
- "**getConceptSchema()**": tale metodo restituisce al chiamante lo schema astratto (specializzazione di ObjectSchema) che rappresenta l'oggetto; tale metodo viene richiamato nell'inizializzazione della classe ontologica. Questa caratteristica permette di semplificare la creazione e la modifica di un nuovo concetto o azione: è infatti sufficiente implementare una classe che specializza DmeConcept e fornisce un'implementazione di DmeConceptInterface. Tutti i dati che dovrebbero essere descritti nella classe ontologica risiedono nella classe appena creata, come pure la loro descrizione astratta, a differenza delle tradizionali ontologie di JADE, in cui le due entità, descrizione astratta e descrizione di implementazione, sono separate in due file, classe ontologica e classe dell'oggetto. Questa caratteristica permette anche di scambiare facilmente descrizioni ontologiche fra agenti: l'impatto dell'aggiunta di un nuovo ObjectSchema è ridotto a una riga di codice nella classe ontologica relativa, mentre le modifiche all'implementazione dell'ObjectSchema non la modificano del tutto.
- Infine, tutte le classi implementate nel sistema sono state modificate in modo da basare la propria descrizione astratta sul supporto più recente fornito dal framework JADE; tale supporto differisce da quello utilizzato nelle precedenti versioni (la versione di riferimento di JADE è la 2.61); il vantaggio del nuovo

supporto, oltre a garantire la compatibilità con le future versioni del framework di sviluppo, è la migliore astrazione che questo supporto fornisce, e la maggior aderenza allo standard FIPA.

E' quindi possibile, data la conoscenza di un concetto, istanziare un oggetto che rappresenta una "materializzazione" di quel concetto. A tale realizzazione è legato sia il modo in cui il sistema memorizza i dati (che vedremo più avanti), sia il meccanismo di ragionamento.

4.2 Meccanismi di ragionamento basati su task

Come abbiamo già visto, il meccanismo di ragionamento del sistema è abbastanza semplice: dato un insieme di fatti o Beliefs, il sistema, in base ai propri Desires, attiva le Intentions opportune. Esiste inoltre un meccanismo aggiuntivo, che consente di ragionare sul contesto in cui l'agente si trova e che permette all'agente di creare automaticamente nuovi task, da eseguire esattamente come quelli che l'utente inserisce esplicitamente. Questo tipo di input al sistema, definito "implicito", è basato su un ragionamento di tipo analogo al precedente, ma applicato su un altro livello: mentre Desires e Intentions sono piuttosto statiche, fanno cioè parte del "modo di essere" dell'agente, questo meccanismo fa riferimento a "che cosa" l'agente ha effettivamente da fare (quali task sono contenuti nella ToDoList).

Descriviamo in dettaglio questo meccanismo:

a partire dai dati disponibili sul contesto e sull'utente, il sistema verifica l'applicabilità di un set di regole, le quali sono del tipo:

IF

(Condizione 1) AND (Condizione 2) AND… (Condizione n)

THEN

CreateNewTask (Task data)

cioè sono regole il cui effetto non è modificare lo stato del mondo dell'agente o raggiungere un particolare Desire, ma aggiungere un ulteriore task alla ToDoList. In tal modo, l'agente può prendere l'iniziativa, ad esempio, di chiamare un taxi se l'utente non ha l'automobile, deve spostarsi in un punto lontano della città, e magari sta anche piovendo. Vediamo un esempio (Figura 4.2):

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Questo template controlla alcune condizioni dell'environment e, se esse corrispondono alle sue precondizioni, crea un nuovo Task (Vedi Knowledge.xsd per lo schema)-->
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:fo="http://www.w3.org/1999/XSL/Format">
    <xsl:template match="//Knowledge">
        <!-- tempo meteorologico-->
        <xsl:variable name="weather" select="./classTime/@weather"/>
        <!-- attività dell'utente-->
        <xsl:variable name="activity" select="./classActivity/@actionMake"/>
        <!-- verifica precondizioni-->
```

```
<xsl:if test="$weather='rainy'">
    <xsl:if test="$activity='walk'">
        <xsl:element name="Task">
            <xsl:attribute name="name">CallTaxi</xsl:attribute>
            <xsl:attribute name="id">1</xsl:attribute>
            <xsl:attribute name="action">"GenericSearch"</xsl:attribute>
            <xsl:attribute name="key">"Find TaxiNumber"</xsl:attribute>
            <xsl:attribute name="date">0</xsl:attribute>
            <xsl:attribute name="belongingScope">None</xsl:attribute>
            <xsl:attribute name="environment">None</xsl:attribute>
            <xsl:attribute name="priority">High</xsl:attribute>
            <xsl:attribute name="slotName">taskDefinition</xsl:attribute>
            <xsl:attribute name="whatToPrenote">Nothing</xsl:attribute>
            <xsl:attribute name="whenToPrenote">0</xsl:attribute>
            <xsl:attribute name="remindBefore">0</xsl:attribute>
            <xsl:attribute name="nextOk">0</xsl:attribute>
            <xsl:attribute name="nextError">0</xsl:attribute>
            <xsl:element name="SubTaskList"/>
            <xsl:element name="ItemList"/>
            <xsl:element name="SubjectList">
                <xsl:element name="Subject">
                    <xsl:attribute name="slotName">subject</xsl:attribute>
                    <xsl:attribute name="content">The agent is searching for a taxi, because the
weather is <xsl:value-of select="./classTime/@weather"/> and you are <xsl:value-of
select="./classActivity/@actionMake"/></xsl:attribute>
                    <xsl:element name="DmeString">
                        <xsl:attribute name="slotName">detail</xsl:attribute>
                        <xsl:attribute name="content">This task, called <xsl:value-of
select="./@name"/>, has <xsl:value-of select="./@priority"/> priority.</xsl:attribute>
                    </xsl:element></xsl:element></xsl:element>
            <xsl:element name="Autonomy">
                <xsl:attribute name="communication">Middle</xsl:attribute>
                <xsl:attribute name="execution">Middle</xsl:attribute>
                <xsl:attribute name="personalData">Low</xsl:attribute>
```

```
                    <xsl:attribute name="resourcesExploitation">Low</xsl:attribute>
                    <xsl:attribute name="slotName">autonomy</xsl:attribute>
                </xsl:element>
                <xsl:element name="Message">
                    <xsl:attribute name="slotName">message</xsl:attribute>
                    <xsl:attribute name="response"></xsl:attribute>
                    <xsl:attribute name="id">1</xsl:attribute>
                    <xsl:attribute name="content"></xsl:attribute>
                    <xsl:attribute name="type"></xsl:attribute>
                </xsl:element>
                <xsl:element name="AgentID">
                    <xsl:attribute name="slotName"></xsl:attribute>
                    <xsl:attribute name="name"></xsl:attribute>
                    <xsl:attribute name="address"></xsl:attribute>
                </xsl:element></xsl:element>
            </xsl:if>
        </xsl:if>
    </xsl:template>
</xsl:stylesheet>
```

Figura 4.2

In questo tipo di task, l'agente non sta semplicemente eseguendo un piano che gli viene commissionato dall'utente, ma si comporta da proponente, anticipando (secondo le sue convinzioni) i desideri dell'utente.

Non a caso, abbiamo specificato "secondo le sue convinzioni"; è infatti possibile che l'agente sbagli, nel prendere una data iniziativa, per un'errata interpretazione dei fatti (errata in generale o relativamente a un particolare utente). Il meccanismo di feedback descritto precedentemente permette all'utente di correggere tali errori nelle regole dell'agente.

4.2.1 Descrizione dei protocolli utilizzati

I protocolli implementati nel sistema D-Me sono le modalità di comunicazione tra agenti: perché uno scambio di messaggi possa definirsi protocollo, esso deve avvenire per uno scopo ben preciso e seguire modalità di svolgimento precise.
I nuovi protocolli implementati, che si aggiungono a quelli già esistenti, riguardano tre categorie distinte, differenziate in base allo scopo:

4.2.2 Protocolli di registrazione

Sono stati introdotti due protocolli che permettono la registrazione di agenti:
Registrazione di un EnvironmentKeeperAgent con il MainKeeper:
questo protocollo permette a un EnvironmentKeeperAgent di diventare visibile per tutti gli agenti che conoscono il MainKeeper con cui l'agente si registra. La registrazione avviene inviando al MainKeeper un messaggio con Performative "**REQUEST**", con ConversationId (definito nella classe StringContainer) "**EnvironmentKeeperRegistrationRequest**"; il messaggio ha ontologia "**D-MeArchitectureOntology**" e contiene un oggetto di tipo Keeper, contenente i dati dell'agente: AID (identificativo univoco per l'agente sulla piattaforma e indirizzi a cui esso è contattabile) e key, che è una stringa caratteristica che descrive la locazione dell'agente. Ad esempio, l'EnvironmentKeeperAgent usato per le prove del sistema ha locazione = "**library**". La stringa è definita nella classe StringContainer.

La risposta del MainKeeper a tale messaggio consiste di un messaggio con ConversationId "**EnvironmentKeeperRegistrationResponse**", Performative "**INFORM**", e come contenuto sempre un oggetto Keeper, identico all'oggetto ricevuto in input se la registrazione avviene con successo, diverso in caso di insuccesso (il che avviene quando un keeper con stesso AID e stessa key è già registrato). Un esempio in figura 4.3.

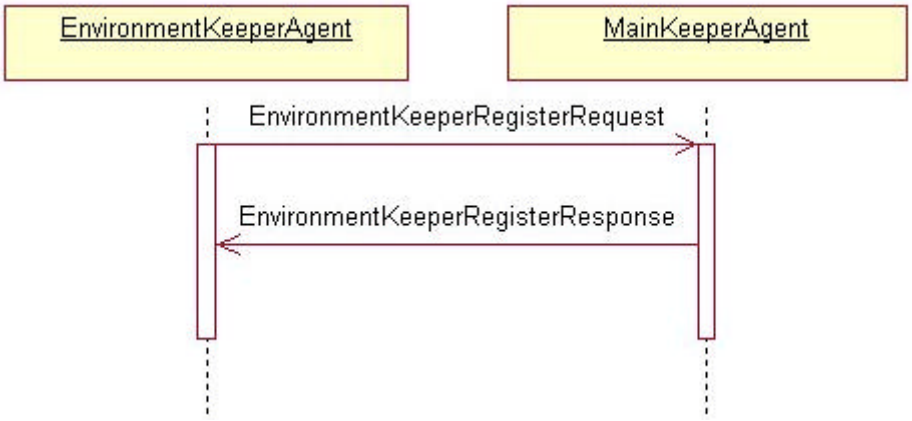


Figura 4.3

Registrazione di un agente (MainAgent o ServiceAgent) con un EnvironmentKeeperAgent:

Analogamente al protocollo precedente, il messaggio ha "**RegisterD-Me**" o "**RegisterService**" come ConversationId, e un oggetto di tipo AgentID come contenuto; la Performative è REQUEST. Il messaggio di risposta ha "**RegisterD-MeResponse**" o "**RegisterServiceResponse**" come ConversationId e Performative "**INFORM**"; il contenuto è lo stesso oggetto di partenza, se la registrazione avviene senza errori; in caso di doppia registrazione, il contenuto è diverso. Esempi in figura 4.4 e 4.5

Figura 4.4

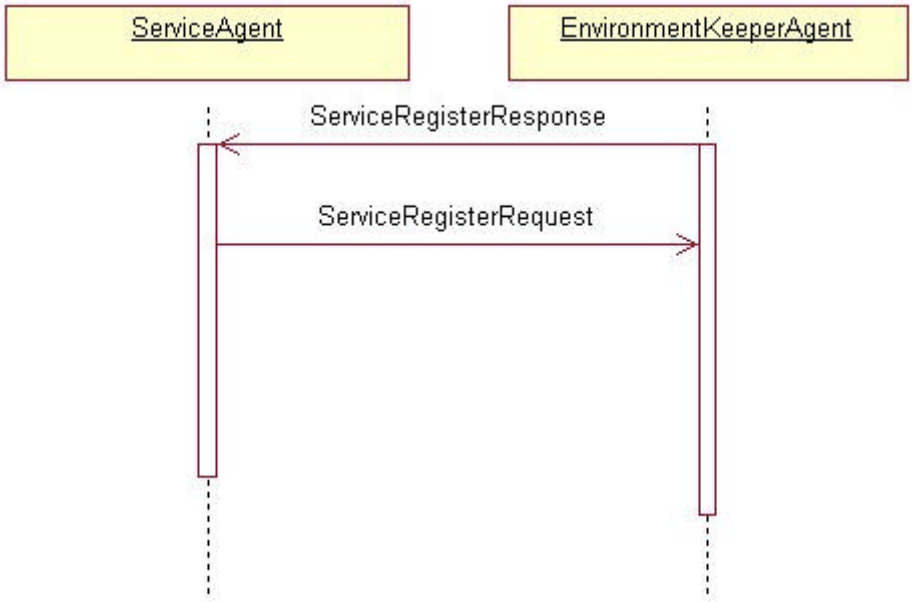


Figura 4.5

4.2.3 Protocolli di ricerca di un agente:

Protocollo di richiesta dell'AID del Keeper:

l'agente interessato a conoscere l'indirizzo del keeper manda un messaggio al MainKeeper con Performative "**REQUEST**", ConversationId "**KeeperRequestKey**" o "**KeeperRequestName**", e contenuto un oggetto Keeper con il campo nome o il campo key avvalorati. A seconda del caso, il MainKeeper compie una ricerca tra gli agenti registrati per nome o per key. Se la ricerca si conclude con esito positivo, viene restituito un oggetto Keeper con i dati completi, altrimenti un oggetto incompleto. La risposta ha ConversationId "**KeeperRequestResponse**" e Performative "**INFORM**".

In questa fase, il protocollo lato agente prevede che l'agente cerchi di contattare l'EnvironmentKeeperAgent ripetutamente, attendendo la risposta per un certo tempo; scaduto questo timeout, l'agente genera un'eccezione di Environment non trovato (può essere dovuta alla disconnessione dell'EnvironmentKeeperAgent o del MainAgent, nel caso uno o entrambi siano connessi alla rete con connessioni wireless, o in altri casi particolari); in tal caso, il protocollo si conclude e l'agente non viene caricato.

Esempio in figura 4.6

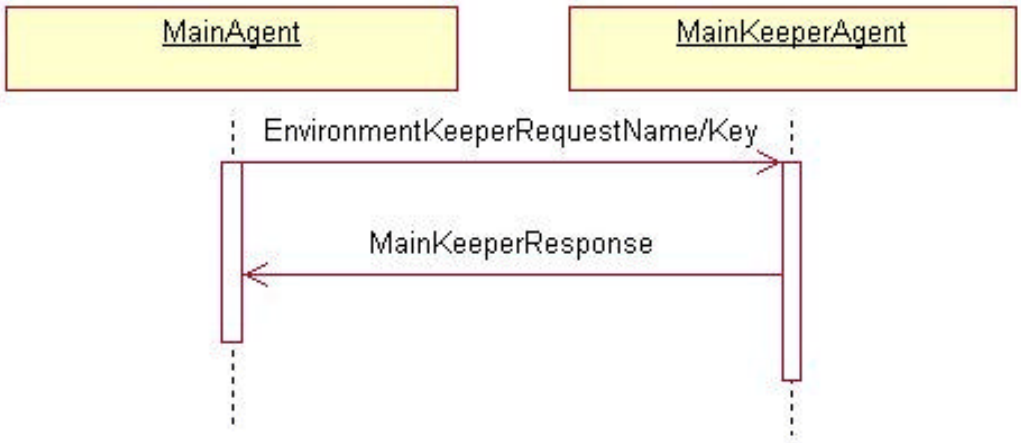


Figura 4.6

4.2.4 Protocollo di richiesta di uno UserConcept:

Quando si hanno più agenti eterogenei, ognuno facente capo a un diverso utente, vi sono sicuramente delle differenze nelle abilità dei vari utenti e nelle conoscenze dei vari agenti. Per permettere lo scambio di questo tipo di informazioni, è stato implementato un protocollo apposito, che sfrutta la struttura dell'architettura per consentire a un MainAgent che ha necessità di una particolare informazione di ricercarla tra tutti gli agenti presenti nel sistema che siano disponibili a fornirla.

Tale protocollo è stato implementato come prototipo per una classe di protocolli che permetta lo scambio di frammenti di conoscenza tra MainAgent: in particolare, esso è relativo alla ricerca di UserConcept che rappresentano capacità dell'utente, cioè "**KnowHow**" (conoscenza del "come fare" una determinata azione).

UserConcept è stato modificato rispetto all'implementazione iniziale in [17], in modo da poter rappresentare non solo il nome e l'ambito di un concetto utente, ma

anche la specifica delle modalità di esecuzione di un'azione, in termini di azioni elementari per il sistema, come si vede in figura 4.7:

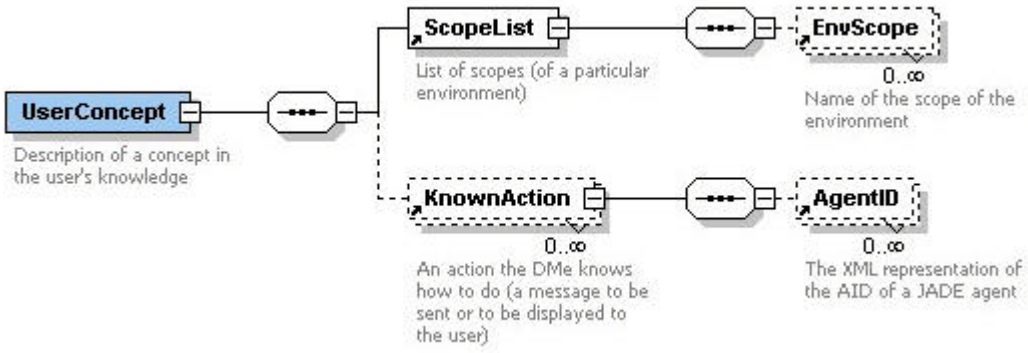


Figura 4.7

UserConcept rappresenta un'azione espressa come "**action**" + "**target**", e contiene una sequenza di azioni elementari, "**KnownActionList**". Un'azione elementare, per il sistema, è:

- mandare un messaggio a un agente
- visualizzare un messaggio all'utente.

Per poterla specificare, occorre la stringa da visualizzare all'utente oppure (nel caso di un messaggio) occorre l'indirizzo dell'agente a cui spedire il messaggio, l'ontologia con cui spedire tale messaggio e la classe dell'oggetto che dovrà effettivamente comparire nel messaggio.

Lo scambio di questi frammenti di conoscenza è un service fornito agli altri MainAgent, ed equivale al chiedere a un'altra persona "**come si arriva**" in un determinato luogo, "**come si fa**" una determinata attività. Esso permette quindi di scambiare frammenti di conoscenza, permettendo al MainAgent di sfruttare una

base di conoscenza condivisa tra tutti i MainAgent attualmente collegati al sistema.

Il protocollo prevede interazione tra più agenti (descrizione in figura 4.8):

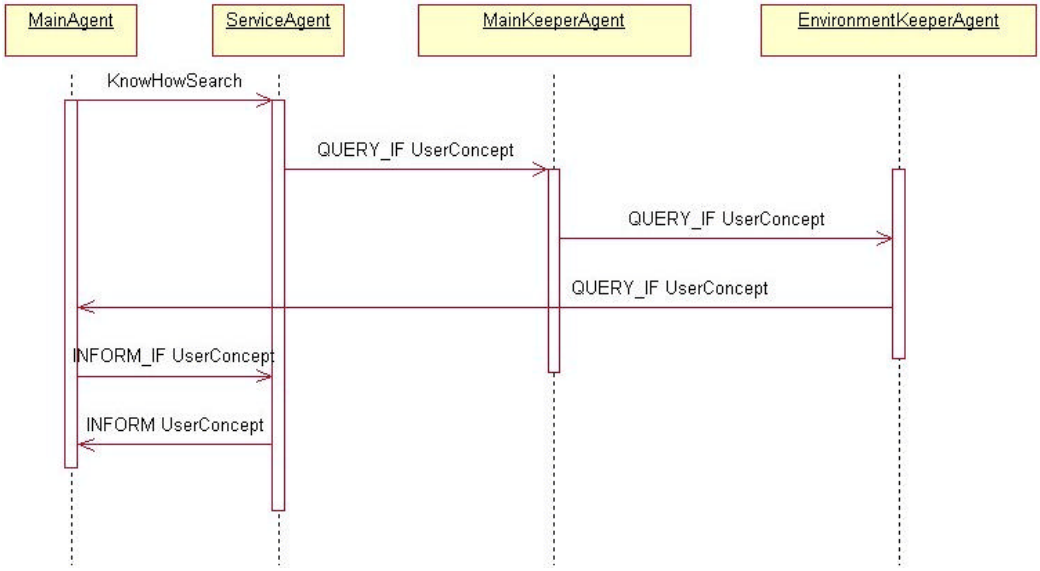


Figura 4.8

Esso viene avviato da un ServiceAgent che possiede il service "**KnowHowSearch**"; tale service invia, in conseguenza della richiesta di un MainAgent, al MainKeeper una richiesta con Performative "**QUERY_IF**" e ontologia "**UMOntology**", contenente uno UserConcept con action e target inizializzati ai valori cercati. Il MainKeeper inoltra tale richiesta a tutti gli EnvironmentKeeperAgent, dopo aver aggiunto alla lista ReplyTo del messaggio l'indirizzo dell'agente. In tal modo, la risposta tornerà direttamente al ServiceAgent, senza ripercorrere all'indietro il percorso originale.

A loro volta, gli EnvironmentKeeperAgent inoltrano la richiesta ai MainAgent registrati presso di loro; ciò è sensato, in quanto non è necessario essere presenti

in un determinato ambiente per possedere un'informazione; inoltre, ciò aumenta le probabilità che un altro agente abbia le conoscenze adatte per rispondere alla richiesta.

Quando la richiesta giunge a un MainAgent, esso cerca nella propria base di conoscenza. Se trova un UserConcept corrispondente ai parametri di ricerca, esso prepara un messaggio di risposta, con Performative "**INFORM_IF**" e ontologia "**UMOntology**", contenente lo UserConcept con i dati, e lo invia direttamente al ServiceAgent che l'ha richiesto, il quale completa l'esecuzione del service restituendo al richiedente i risultati.

Se il MainAgent non trova risultati, non manda nessuna risposta indietro al ServiceAgent; questo perché il numero di agenti è potenzialmente molto grande, e il numero di messaggi privi di informazione potrebbe causare un decadimento delle prestazioni, non motivato da un'effettiva esigenza informativa. Per garantire la terminazione del protocollo, il ServiceAgent attende per 20 secondi le risposte, quindi procede assumendo che non ci sia stata risposta. Dati i tempi medi riscontrati in esecuzione, tale limite risulta abbondantemente sufficiente ad ottenere risposte, quando esse vi siano.

4.2.5 Protocollo di esecuzione di un service di ricerca generico:

Tale protocollo prevede l'invio di una richiesta di ricerca che non specifica quale tipo di dati cercare: il messaggio da inviare al ServiceAgent è una "**REQUEST**", con ontologia "**D-MeArchitectureOntology**", "**UM-Ontology**" o

"**ResultPresentation-Ontology**", contenente un Task la cui action corrisponde a "**GenericSearch**" e la cui key è la chiave da usare per la ricerca. Il service utilizza un proprio database XML, in cui sono memorizzati tutti gli "**item**" che il service conosce, e che contiene anche, per ogni concetto, una parola chiave e un riferimento (Entry). In tal modo, utilizzando più riferimenti, è possibile recuperare lo stesso concetto partendo da due parole chiave diverse. Ciò aumenta le possibilità di trovare i dati richiesti: supponendo, per fare un esempio, che siano presenti nel database i dati relativi a un film e le biografie degli autori principali, cercare il nome del protagonista consentirà di ottenere la trama del film e la storia personale del protagonista, mentre il nome del film troverà i dati relativi al film e a tutti gli attori che vi hanno partecipato. Tale database è limitato, poiché va popolato manualmente, ma, avendo a disposizione una base di dati più consistente, si può ipotizzare collegata a un motore di ricerca per Internet, ciò permette al sistema di effettuare ricerche anche molto estese.

Il messaggio di risposta deve essere costruito utilizzando alcune informazioni ulteriori: dato che non è noto a priori il tipo di dato che verrà estratto dal database, è necessario utilizzare una classe ontologica che contenga le descrizioni per tutti i tipi di dati che il database può contenere. Poiché ciò non è utile, nei dati della Entry è presente un campo che indica qual è la classe che rappresenta l'oggetto puntato. Conoscendo tale classe, è possibile ottenere un oggetto Java che contiene i dati. Il messaggio di risposta sarà poi spedito con l'ontologia richiesta e Performative INFORM. E non ci sono risultati, la Performative è FAILURE.

Esempio in figura 4.9

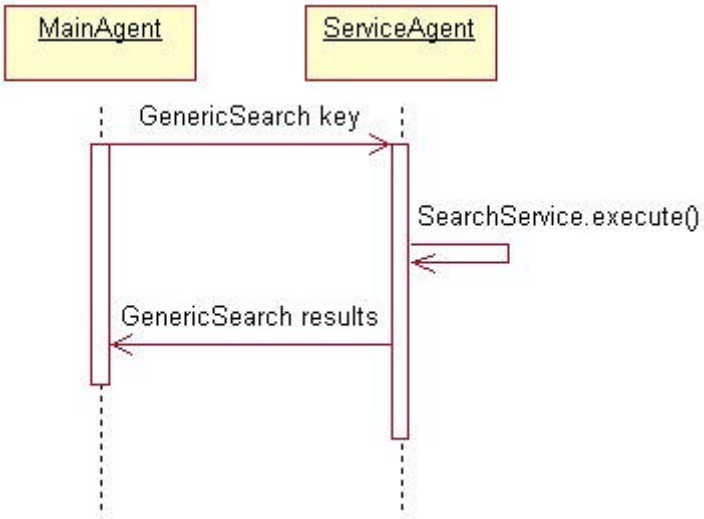


Figura 4.9

Attualmente il sistema comprende solo service di ricerca, ma è semplice creare nuovi service: è sufficiente implementare nuove classi che implementino il metodo generico "**execute()**", e registrarle con un opportuno ServiceAgent.

Capitolo 5: Dettagli sul funzionamento

5.1 Struttura della mente dell'agente D-Me

5.1.1 Accesso ai dati permanenti

Ogni agente contenuto nel sistema D-Me usa file di conoscenza XML per salvare dati persistenti [11]; l'accesso a tali dati avviene attraverso la classe **Knowledge**. Essa contiene metodi per leggere e scrivere oggetti (specializzazioni di **DMeConcept**) su file XML (quindi metodi per la serializzazione di oggetti), e metodi per cancellare e aggiungere nodi; tale classe maschera completamente l'uso di file XML per la memorizzazione persistente.

Tutte le sottoclassi specializzazione di **DMeConcept** implementano un costruttore che accetta come parametro un nodo XML (che implementi l'interfaccia **org.w3c.dom.Element**); in tal modo, la lettura e la scrittura di un oggetto da un file XML risultano operazioni elementari, e permettono agli agenti di essere indipendenti dal particolare parser utilizzato, a patto che l'interfaccia del parser sia consistente con quella utilizzata. Di default, il sistema usa il parser Xerces, sviluppato dal team di Apache [15].

Sfortunatamente, tale parser non può funzionare su piattaforma J2ME (Java 2 Micro Edition, di SUN), in quanto, sia per le dimensioni ragguardevoli, sia su disco che in memoria, che per le caratteristiche della J2ME, esso risulta troppo oneroso per i dispositivi attuali. Più in dettaglio, esso utilizza package non implementati nella piattaforma J2ME, ad esempio il package **java.lang.reflect**. Tale limite della J2ME costringe all'utilizzo di un parser di dimensioni più ridotte, come KXML [www.kxml.org], le cui richieste di memoria sono minori e che è utilizzabile su piattaforma J2ME. Tuttavia, poiché l'interfaccia di KXML non è del tutto conforme con l'interfaccia di Xerces, il suo utilizzo richiede la riscrittura di alcune porzioni di codice, in particolare la classe Knowledge e i costruttori parametrici di tutte le classi descritte nelle classi ontologiche. Dato il grosso impatto che tale modifica avrebbe avuto sul codice, abbiamo trascurato, peril momento, la piattaforma J2ME, a favore delle più potenti PJava e J2SE.

La possibilità di ottenere una descrizione XML di ogni oggetto che rappresenta i dati in possesso dell'agente è alla base del meccanismo di inferenza di nuovi task, come vedremo più avanti.

5.1.2 Scambio di messaggi: il MainReceiver

Il MainReceiver è un behaviour ciclico (esso viene rischedulato nella coda di behaviour dell'agente non appena il suo metodo **action()** termina), il cui compito è leggere e rimuovere dalla coda di messaggi in arrivo all'agente gli ACLMessage inviati da altri agenti, e smistarli ai behaviour competenti per l'estrazione dei dati.

Questo behaviour possiede due liste, la lista **Forever** e la lista **OneShot**, che rispettivamente ospitano i behaviour che restano in ascolto, pronti a ricevere messaggi, per tutta la vita dell'agente (esempi tipici sono i behaviour di ricezione della locazione e di ricezione del feedback), ed i behaviour che invece devono ricevere un solo messaggio prima di essere rimossi dalla lista. Ognuno di questi behaviour implementa l'interfaccia **dme.behaviours.behaviourinterface. DynamicLoadableBehaviour**, che comprende il metodo **execute()** e il metodo **getMessageTemplate()**.

Il metodo "**getMessageTemplate()**" fornisce al chiamante il MessageTemplate che contiene le caratteristiche del messaggio che quel behaviour è programmato a ricevere, e viene usato dal MainReceiver per determinare quale behaviour debba essere eseguito per rispondere correttamente a un messaggio. Per risolvere possibili conflitti, il MainReceiver adotta la politica del "first found": il primo behaviour il cui MessageTemplate corrisponde al messaggio viene attivato, ed il messaggio viene consumato. E' necessario prestare attenzione ai conflitti che possono verificarsi se due behaviour utilizzano lo stesso MessageTemplate, in quanto potrebbe derivarne un comportamento non previsto del sistema.

Tale meccanismo è ancora perfezionabile, implementando meccanismi di soluzione dei conflitti più efficienti, o una politica di precedenza tra behaviour. Esso è stato introdotto per ovviare a un problema più complesso, cioè la molteplicità di behaviour ciclici che erano presenti nel sistema in precedenza. La presenza di più behaviour riceventi, infatti, aveva non solo il difetto appena enunciato, vale a dire la possibilità che due behaviour avessero lo stesso MessageTemplate, e quindi che fossero concorrenti nella lettura dei messaggi, ma

aveva anche l'aggravante di essere un problema molto più difficile da scoprire. Infatti JADE offre due tool per il debug, lo Sniffer e l'Introspector, che mostrano rispettivamente lo scambio di messaggi tra un agente e l'altro, e lo stato corrente di un agente (messaggi ricevuti e inviati, behaviour caricati), ma non consentono di verificare quali behaviour consumano effettivamente i messaggi. Con l'utilizzo del MainReceiver, tutti i messaggi vengono consumati dallo stesso behaviour, e, col semplice espediente di stampare il behaviour di destinazione di un particolare messaggio, diviene banale trovare i punti in cui un particolare protocollo fallisce per un conflitto di ricezione messaggi; inoltre, viene indicato il responsabile del conflitto. Infine, la presenza di un unico behaviour ciclico diminuisce in maniera considerevole il carico computazionale del sistema (60/70% di utilizzo del processore in meno).

Il metodo "**execute()**" consente di far eseguire immediatamente il compito del behaviour; la necessità di questo metodo, in sostituzione dell'**action()** del behaviour, che viene eseguita automaticamente quando il behaviour viene caricato dall'agente, dipende dal tipo di scheduling dei behaviours: essendo uno scheduling di tipo **cooperative**, quando un behaviour viene aggiunto alla lista dei behaviour in esecuzione, l'action() non viene eseguita subito, ma dopo che il MainReceiver ha terminato un ciclo di esecuzione. Questo provoca un cambiamento nei valori dei parametri, in particolare si modifica il messaggio ricevuto, per cui il behaviour si trova ad operare su dati errati. Col metodo execute(), tale problema scompare. L'action() dei behaviours rimane disponibile per eseguire altri eventuali compiti, da progettare però tenendo presente i problemi sui parametri. Diventa necessario eseguire una copia locale di tutti i dati

necessari per l'esecuzione del behaviour, e tale copia può essere molto pesante dal punto di vista computazionale, sia per il tempo che per la memoria richiesti.

5.1.3 Esecuzione dei Desires: agente BDI

I Desires sono memorizzati nel file di conoscenza dell'agente MainAgent. Essi vengono caricati all'avvio dell'agente, e per ognuno di essi l'agente legge un insieme di Intentions, che, se eseguite, assicurano la soddisfazione del Desire. Tale impostazione, attualmente, permette solo di costruire piani per il soddisfacimento di un Desire utilizzando la struttura di sequenza; è tuttavia possibile aumentare la capacità espressiva dei piani, introducendo una strutturazione più complessa, in analogia alla strutturazione per la lista di sottotask di un task. Questo è un possibile sviluppo del sistema; aggiungendo anche un meccanismo per la costruzione dinamica di piani, sarebbe possibile ottenere un agente realmente flessibile, in grado di comportarsi come un sistema a produzioni di tipo classico.

Per ogni Intention, l'agente cerca, tra i suoi behaviours, un behaviour che possa essere eseguito, date le precondizioni, e che abbia come risultato quelli che sono gli Obtains della Intention, vale a dire gli effetti. Tali behaviours sono classi Java che implementano l'interfaccia ThinkableBehaviour, e rendono possibile aggiungere all'agente un comportamento preciso. Essi non vengono aggiunti all'agente in modo predefinito, ma la loro invocazione viene decisa a runtime, n base ai Desire e alla loro esecuzione.

Il Desire principale dell'agente è:

"**Realizzare i task dell'utente nell'ambiente opportuno**";

per far ciò, l'agente deve:

- **Riconoscere la propria locazione e le caratteristiche contestuali**: tale riconoscimento avviene grazie ai messaggi inviati all'agente dagli agenti di contesto che effettuano il monitoraggio dei parametri ambientali: posizione, situazione meteorologica, data e ora, situazione emotiva, attività dell'utente. Il riconoscimento prioritario è quello della locazione, infatti è necessario per l'agente, per iniziare la propria attività, potersi collegare a un Keeper di ambiente, per la cui ricerca è necessaria appunto la conoscenza della locazione. L'agente rimane in stato quiescente finché non viene rilevata tale posizione, quindi inoltra al MainKeeper la richiesta relativa all'indirizzo del Keeper di ambiente che gestisce la sua locazione (il protocollo relativo verrà descritto più avanti).

- **Ottenere un modello dell'utente aggiornato**: tale funzione è svolta dall'Intention "**Get contextualized user model**". Il behaviour che implementa l'Intention è **RequestUserModel**: esso invia all'UMKeeper una richiesta, a cui l'agente interpellato risponde fornendo lo user model contestuale, cioè contenente le informazioni rilevanti per il particolare ambiente. Esso provvede poi a porre un opportuno sottobehaviour in ascolto per ricevere i dati, aggiungendolo alla lista dei behaviour in ascolto.

- **Eseguire i task presenti nella ToDoList**: a tal fine, viene attivato il behaviour **IntroduceBehaviour**, il cui scopo è ottenere una lista dei task eseguibili nel particolare ambiente, unita alla lista dei Service necessari per l'esecuzione degli stessi, e il behaviour **ExecuteTaskRequest**, che si occupa dell'esecuzione vera e propria dei task.

Vediamoli più in dettaglio:

- **IntroduceBehaviour** è un **FSMBehaviour**, quindi un behaviour strutturato come una macchina a stati finiti. Il suo compito è riconoscere lo scope dell'Environment in cui si trova immerso, in modo da poter determinare quali task sono adatti all'Environment, e trovare poi, per ognuno di questi task, un Service accessibile che permetta di eseguirli. Nel caso in cui il task da eseguire non necessiti di un Service esterno (per esempio, un task di comunicazione con l'utente), viene assegnato il behaviour che realizza effettivamente il servizio. Per far ciò, esso utilizza un semplice protocollo di richiesta al Keeper, richiedendo l'elenco dei service offerti dai vari ServiceAgent (vedi figura 5.1).
- **ExecuteTaskBehaviour** si occupa invece dell'esecuzione vera e propria dei task: esso è un **SequentialBehaviour**, si compone quindi di una serie di sottobehaviour da eseguire in sequenza. La sua funzione principale è raccogliere i task pronti per l'esecuzione e eseguire i behaviour collegati, siano essi behaviour dell'agente o richieste a Service. Una componente importante di questo behaviour è la possibilità di eseguire task la cui struttura

non è nota a priori (ovviamente, che non siano task semplici, ma strutturati, con sottotask) (vedi figura 5.2). La classe CompositeTaskActuator, infatti, è in grado di leggere la descrizione XML di un task e creare una FSM a partire dalla descrizione di un grafo. Attualmente, per semplicità, la struttura è stata provata con un solo livello di nidificazione, ma il progetto prevede la possibilità di innestare come sottotask di un task composto un altro task composto, permettendo una strutturazione ad albero.

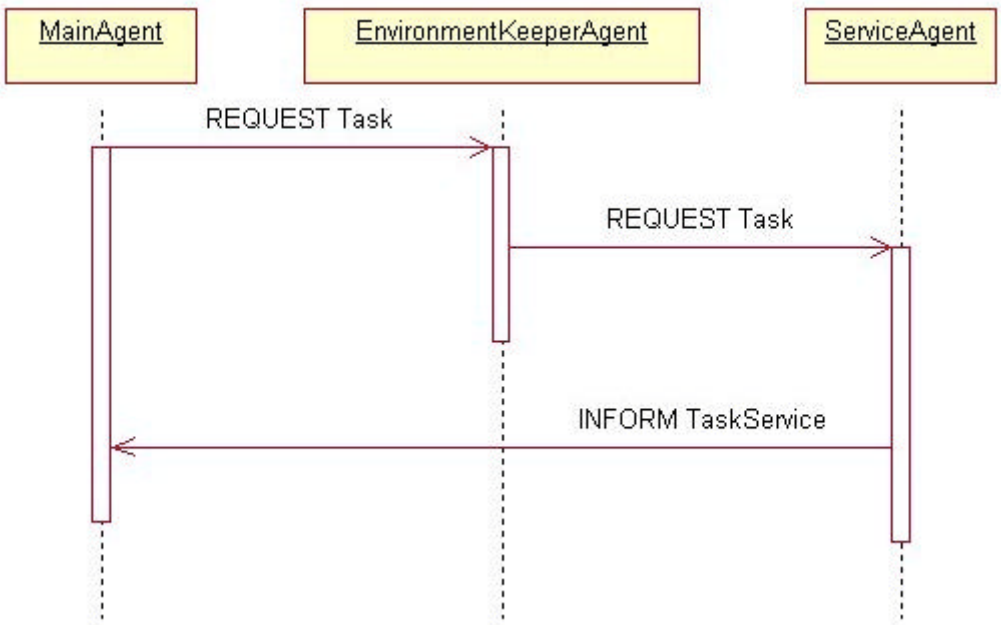


Figura 5.1

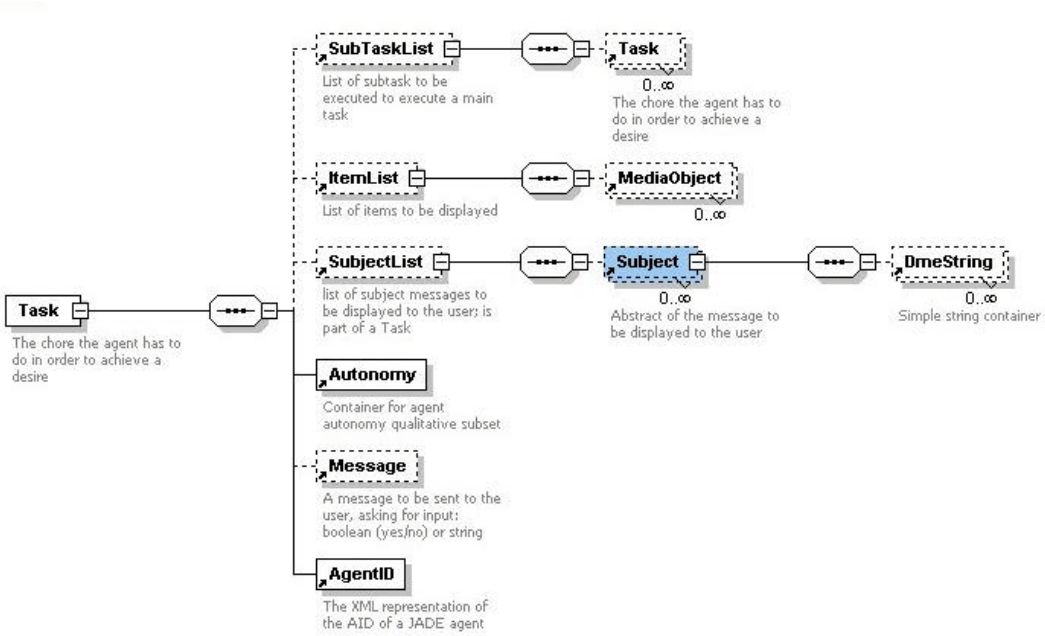

Task
The chore the agent has to do in order to achieve a desire
SubTaskList
List of subtask to be executed to execute a main task
Task
0..∞
The chore the agent has to do in order to achieve a desire
ItemList
List of items to be displayed
MediaObject
0..∞
SubjectList
list of subject messages to be displayed to the user; is part of a Task
Subject
0..∞
Abstract of the message to be displayed to the user
DmeString
0..∞
Simple string container
Autonomy
Container for agent autonomy qualitative subset
Message
A message to be sent to the user, asking for input: boolean (yes/no) or string
AgentID
The XML representation of the AID of a JADE agent

Figura 5.2

5.2 Esecuzione di un task e generazione di nuovi task

5.2.1 Inferenza di nuovi task: trasformazione XSL

Il meccanismo che permette di creare nuovi task su iniziativa dell'agente è strutturato in questo modo:

- L'agente è in possesso di tutte le informazioni necessarie ad avviare il ragionamento, rappresentate come oggetti descritti nelle classi ontologiche del sistema; tale insieme di dati è lo stato del mondo, o meglio i Beliefs dell'agente.
- A partire da questi oggetti, l'agente può ottenere una rappresentazione XML della propria conoscenza in un particolare istante di tempo, semplicemente richiamando per ogni oggetto il metodo toXml() e concatenando i risultati ottenuti. Si ottiene così un file XML che può essere considerato il database globale del sistema.
- A tale file si può applicare una trasformazione XSL, analoga concettualmente all'esecuzione di un pattern matching, che restituisce come risultato un altro file XML contenente i nuovi task che il sistema deve eseguire. Tali task possono essere letti e diventare nuovi oggetti, che possono essere aggiunti alla lista di task del sistema.

- Le regole di decisione sono pertanto scritte in forma di template XSL (vedi figura 5.3).

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This template can be used to format a communication task with high priority; it generates the Subject of this task, in XML-->
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:fo="http://www.w3.org/1999/XSL/Format">
    <xsl:template match="//Task">
        <xsl:variable name="action" select="./@action"/>
        <xsl:if test="$action='Communication'">
            <xsl:element name="Subject">
                <xsl:attribute name="slotName">subject</xsl:attribute>
                <xsl:attribute name="content">Attention! You should <xsl:value-of select="./@key"/> in the <xsl:value-of select="./
@environment"/> at <xsl:value-of select="./@time"/> of <xsl:value-of select="./@date"/>.</xsl:attribute>
                <xsl:element name="DmeString">
                    <xsl:attribute name="slotName">detail</xsl:attribute>
                    <xsl:attribute name="content">This task, called <xsl:value-of select="./@name"/>, has <xsl:value-of select="./
@priority"/> priority, and consequently its autonomy of communication is <xsl:value-of select="./Autonomy/@communication"/>.
</xsl:attribute>
                </xsl:element>
            </xsl:element>
        </xsl:if>
    </xsl:template>
</xsl:stylesheet>
```

Figura 5.3

Tale soluzione presenta i seguenti svantaggi:

- Necessità di un XSLT transformer, come ad esempio Xalan[15]: le difficoltà riguardano le dimensioni del componente, che richiede un grande spazio di memoria, sia per la memorizzazione che per l'esecuzione;
- Prestazioni scadenti in esecuzione: la trasformazione XSL, con relativo pattern matching, può richiedere molto tempo macchina;

questi svantaggi impediscono di eseguire la trasformazione XSL direttamente su dispositivi portatili; è quindi necessario che il MainAgent risieda su un sistema che fornisca la sufficiente potenza di calcolo. Tuttavia, tale limitazione era già riscontrabile sia in JADE che nell'architettura D-Me realizzata precedentemente [11], sia per i problemi già visti di dimensioni e di complessità che per la

mancanza di alcuni package indispensabili al funzionamento; pertanto tali svantaggi non impattano in maniera evidente sul progetto del sistema.

I vantaggi dell'utilizzo di tale metodo sono:

- Non è necessario **implementare** o **incorporare** un **motore inferenziale** nel sistema: ciò permette di risparmiare un considerevole sforzo implementativo;
- Le regole sono scritte in uno **standard indipendente** dal sistema e riconosciuto a livello mondiale; è quindi possibile utilizzare le stesse regole su qualunque sistema. L'unico vincolo all'implementazione è la struttura del file XML di partenza, che tuttavia contiene solo elementi definiti attraverso un XML Schema, quindi anch'esso in linguaggio standard e aperto;
- E' possibile **modificare con facilità le regole**, senza necessità né di modifica del codice sorgente né di ricompilazione; è quindi possibile pensare a un tool esterno che consenta di scrivere regole per il sistema, a partire dalla descrizione XSD delle entità in gioco; ciò permetterebbe di fornire all'utente uno strumento per personalizzare il proprio agente in modo consono alle esigenze personali, o, più verosimilmente, per permettere di sperimentare configurazioni diverse senza dover conoscere il sistema in dettaglio.

Visti i vantaggi offerti dall'indipendenza rispetto al sistema implementato, il meccanismo di creazione dei task ha sfruttato l'XSLT transformer di Xalan.

Capitolo 6: Caso di studio

6.1 Gestione della ToDoList

Come esempio di esecuzione, rendiamo la rappresentazione XML di una ToDoList contenente un task di prenotazione di un esame (vedi figura 6.1):

```
<Task slotName="taskDefinition" id="2" name="ExamToDo" action="Reminding" key="Programming" date="1213311" belongingScope="studio" environment="library" priority="high" whatToPrenote="exam" whenToPrenote="445645644" remindBefore="34343444" nextOk="0" nextError="0">
    <SubTaskList slotName="subTaskList"/>
    <SubjectList slotName="message">
        <Subject slotName="subject" content="You should go to sign for the Exam of Programming" sound="false" voice="false">
            <DmeString slotName="detail" content="You can do it going on third floor…" sound="false" voice="false"/>
        </Subject>
    </SubjectList>
    <Autonomy slotName="autonomy" execution="high" communication="high" personalData="high" resourcesExploitation="high"/>
    <AgentID slotName="feedbackAgent" name="" address=""/>
</Task>
```

Figura 6.1

inoltre, tra gli interessi dell'utente c'è "Cinema", cioè all'utente piace andare al cinema (questa informazione può essere introdotta all'atto della registrazione o durante l'interazione; in figura 6.2 è rappresentato un frammento del file di conoscenza contenente uno UserConcept rappresentante un Interest)

```
<Interest slotName="Interest">
    <UserConcept slotName="userConcept" confidence="high" topic="cinema" publicly="true" action=""
target="">
        <ScopeList slotName="Scope">
            <EnvScope slotName="scope" name="Cinema"/>
            <EnvScope slotName="scope" name="FreeTime"/>
        </ScopeList>
    </UserConcept>
</Interest>
```

Figura 6.2

Riguardo alle condizioni ambientali, l'utente si trova nella hall del dipartimento di informatica, e che stia piovendo (vedi figura 6.3).

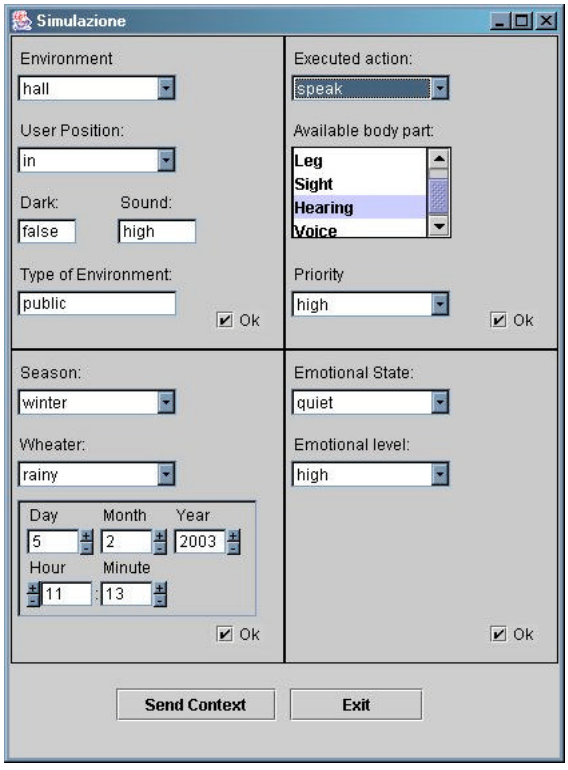


Figura 6.3

In tale situazione, il comportamento previsto del sistema è:

- **esecuzione del task di prenotazione**: ricerca delle modalità di esecuzione, eventuale richiesta all'utente di dati non in possesso del sistema, inoltro dei messaggi (ove previsto), aggiunta di un task di **Reminding** da eseguire nel momento opportuno (in dipendenza dalla data della prenotazione).
- **creazione di nuovi task**: il sistema deve elaborare, in base alle caratteristiche dell'utente, due nuovi task: **ricerca di novità cinematografiche** (perché all'utente piace il cinema) e **ricerca dei numeri di telefono per la società dei taxi** (poiché sta piovendo, per recarsi al cinema l'utente potrebbe averne bisogno). Per ulteriori esempi e per una descrizione dettagliata dell'interfaccia utente, esse sono descritte in dettaglio in [21].

6.2 Aggiunta di task impliciti

6.2.1 Task ad alto livello di autonomia

Nell'esempio precedente, il task di ricerca di novità cinematografiche richiede un elevato livello di autonomia. Infatti, prevedendo anche la possibilità della prenotazione di un posto a sedere nel cinema, è necessario che l'utente deleghi l'esecuzione completa del piano e dia un alto livello di autonomia per lo sfruttamento delle risorse (tempo e carta di credito per la prenotazione), altrimenti l'agente eseguirà solo la parte del task che richiede meno autonomia, cioè la raccolta di informazioni.

6.2.2 Task a basso livello di autonomia

Sempre nell'esempio precedente, in tale tipo di task rientra la ricerca dei numeri di telefono dei taxi: infatti, la semplice richiesta di informazioni non richiede né risorse né diffusione di dati personali, e la sua autonomia di esecuzione è bassa; pertanto, l'unico effetto non gradito all'utente può essere un messaggio di avviso non richiesto, cioè un eccesso di intrusività da parte dell'agente. Tale iniziativa può essere annullata dall'utente, attivando quindi i meccanismi di feedback precedentemente descritti, per cui la regola applicata per generare il task non verrà più applicata finché non si cambierà livello di autonomia.

6.3 Gestione del feedback

Per verificare la gestione del feedback, si può procedere annullando alcuni dei task intrapresi dall'agente e osservando la modifica alla base di regole che avviene in conseguenza di ciò. Si può anche verificare che il profilo utente viene aggiornato, in conseguenza delle eventuali nuove deduzioni che è possibile fare.

I protocolli che consentono di ricevere il feedback sono semplici protocolli che prevedono l'invio tra l'InterfaceAgent e il MainAgent di un oggetto di classe **TaskRef;** tale oggetto contiene l'attributo **id**, il cui valore corrisponde all'identificativo univoco di un task, e l'attributo **slotName**, il cui valore indica

l’operazione da effettuare sul task (eliminare, inserire, modificare, mettere in pausa). Il feedback viene dedotto dal MainAgent alla ricezione di un TaskRef, e viene modificato il livello di autonomia della regola interessata o dell’agente, a seconda dei casi.

Conclusioni

I risultati ottenuti durante lo svolgimento di questa tesi hanno permesso di dotare il sistema D-Me di un meccanismo di gestione esplicita dell'autonomia che consente di trattare tale aspetto dell'interazione con l'utente a prescindere dall'implementazione; è possibile variare il comportamento del sistema agendo direttamente sui dati su cui il sistema lavora, i quali sono interamente memorizzati in formato XML, e descritti con formalismo XSD, quindi facilmente modificabili da un'entità esterna.

Analogo discorso vale per le regole di ragionamento che permettono di generare i task descritti nel capitolo precedente: lo standard XSL permette di leggere, comprendere e modificare facilmente le regole di inferenza del sistema, oltre a garantire la portabilità di tutto il sistema su qualunque piattaforma, grazie all'utilizzo di Java e XML.

Inoltre, dal punto di vista progettuale e implementativo, la struttura del sistema è stata migliorata; in particolare, il sistema è più facilmente espandibile, prevedendo meccanismi per l'aggiunta di nuovi behaviour e nuove caratteristiche non note al momento della compilazione; tali meccanismi comprendono l'introduzione di nuove classi di oggetti ontologici, di nuovi behaviour, di nuove regole XSL.

Restano punti aperti del sistema i meccanismi di ragionamento dinamici, cioè la possibilità di modificare dinamicamente i piani descritti sia nei Desire che nei task. Inoltre, è necessario dotare il sistema di un meccanismo per la gestione della comunicazione tra più piattaforme: infatti, al momento l'agente MainAgent lavora considerando, ove non indicato espressamente, gli altri agenti come operanti sulla sua stessa piattaforma. E' possibile specificare dei parametri, per ogni agente, in modo da fornire le informazioni relative agli agenti con cui deve comunicare, ma non è ancora possibile, per il MainAgent, ottenere i dati del suo agente d'interfaccia senza conoscere almeno il nome di tale agente. Questa è una limitazione, in quanto dovrebbe essere auspicabile una maggior libertà di movimento per gli agenti del sistema.

Bibliografia

[1] Negroponte, N. Being Digital. Vintage Books, 1995

[2] http://www.whirlpool.fi/fi/uutinen_06052002_swe.asp

[3] http://press.nokia.com/PR/200007/785051_5.html

[4] Mark Weiser, "Hot Topics: Ubiquitous Computing" IEEE Computer, October 1993

[5] Tuning the Collaboration Level with Autonomous agents: a Principled Theory, www.csce.uark.edu/~hexmoor/AA01/ camerarady/Falcone.pdf

[6] M.Woolridge (1992), “The logical Modelling of computational multi-agent systems”, Ph.D. Thesis, Dep. Of Computation, UMINST, Manchester, UK

[7] http://www.acm.org/crossroads/xrds5-4/multiagent.html

[8] Jennings, N.R. and Wooldridge, M. Intelligent Agents: Theory and Practice. In: The Knowledge Engineering Review, 1995, Volume 10, Number 2, pages 115-152

[9] Jennings, N.R. and Wooldridge, M. Intelligent Agents: Theory and Practice

[10] A.S. Rao, M.P. Georgeff, “BDI agents: From theory to practice”, April 1995

[11] Luigi Iannone, “Un’architettura multiagente per Ubiquitous Computing”, tesi di laurea dicembre 2001 Università degli studi di Bari

[12] http://jade.cselt.it

[13] www.fipa.org

[14] Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents, Stan Franklin and Art Graesser

[15] ww w.apache.org

[16] http://www.telecomitalialab.com

[17] Nicola Tino, “Modellizzazione dell’utente in Ubiquitous Computing”, tesi di laurea dicembre 2001 Università degli studi di Bari

[18] Vincenza Belviso, “Adattività in Ubiquitous Computing”, tesi di laurea dicembre 2001 Università degli studi di Bari

[19] Addolorata Cavalluzzi, “Modellizzazione ed uso del contesto nell’Ubiquitous Computing”, tesi di laurea luglio 2002, Università degli Studi di Bari

[20]Pasqua Anna Maria Lionetti, “User Modeling in un sistema ad agenti mobili”, tesi di laurea luglio 2002, Università degli Studi di Bari

[21] Antonio Giuliani, “Un’interfaccia adattiva per un sistema Ubiquitous”, tesi di laurea dicembre 2002, Università degli Studi di Bari

Ringraziamenti:

Desidero ringraziare alcune persone in particolare:

mamma e papà, perché senza i loro sacrifici quello che avete davanti non esisterebbe;

le mie sorelle, i miei amici e tutte le altre persone che non hanno mai dubitato che ce l'avrei fatta, perché mi hanno dato la fiducia di cui avevo bisogno;

tanta altra gente, troppi perché io possa ricordarmene ora, ma che non ho dimenticato.

Consapevole del fatto che un ringraziamento non basta, a tutte queste persone non dico grazie, ma prometto che ricambierò.

Infine un ringraziamento speciale a tutti quelli che nessuno ringrazia: tutti quelli che mi hanno creato problemi, che hanno pensato che non sarei mai arrivato in fondo all'università, che hanno sottovalutato me o quello che stavo facendo; a loro il mio grazie più sentito, perché ora posso dire:

NONOSTANTE voi, ce l'ho fatta! ☺