UNIVERSITÀ DEGLI STUDI DI TRENTO

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica

Condivisione di conoscenza in ambienti distribuiti: uso della Cultura Implicita nello scenario Hunter Prey

Laureando

Matteo Marchi

Relatore

Paolo Giorgini

Anno Accademico 2006/2007

Indice

Introduzione

Oggigiorno uno degli ambiti che sta acquisendo più importanza e quindi che sta ricevendo sempre maggiore attenzione da parte degli istituti di ricerca informatica e tecnologica di tutto il mondo è sicuramente quello riguardante i sistemi distribuiti. Essi infatti vengono usati in un numero crescente di sistemi e tecnologie utilizzate quotidianamente da un sempre maggior numero di persone in tutto il mondo (basta pensare alla continua espansione del numero di utenti in Internet, l'esempio più famoso di sistema distribuito). Collegato con il continuo aumento dell'uso di questi sistemi c'è il continuo proliferare di problemi connessi ad essi. Uno dei più importanti e più considerati è sicuramente quello della mancanza di conoscenza da parte di agenti in ambienti distribuiti nuovi per loro. Quando infatti un agente agisce in un ambiente senza le necessarie conoscenze ed abilità, l'efficacia del suo comportamento non può che essere inferiore a quella ottimale. Il problema diventa ancor più complicato se l'agente deve interagire con altri agenti che vivono abitualmente nello stesso ambiente ed hanno conoscenze riguardo ad esso. Si pensi ad esempio a quando un utente di Internet si collega per cercare informazioni utili su un argomento che non conosce oppure ad un commerciante che si affaccia su un nuovo mercato che non conosce. In entrambi i casi ci sarà un problema serio per gli attori in quanto a tutti e due mancherà la conoscenza dell'ambiente e degli attori che vi operano.

Per risolvere questo problema l'agente in questione dovrebbe agire secondo quanto la “cultura” del nuovo ambiente suggerisce, cioè secondo gli usi degli agenti che già vi operano all'interno. Un metodo risolutivo concreto può essere quello di indurre gli agenti a comportarsi in maniera accettabile tramite il controllo dell'ambiente e la modifica della visione che l'agente ha di esso. Questo controllo può permette al sistema di dare dei suggerimenti agli agenti sul come comportarsi portando gli agenti ad una completa comunanza di comportamento che rappresenta la cosiddetta cultura implicita .

Un fenomeno di Cultura Implicita avviene quando un gruppo di agenti si comporta in un

modo simile ad un altro gruppo di agenti senza che i primi abbiano una precedente conoscenza approfondita dei secondi. Questa condivisione di conoscenza chiamata Cultura Implicita viene realizzato da un Sistema di Supporto alla Cultura Implicita (SICS) tramite il completamento di due passi: il primo consiste nell'elaborazione di una cultura di gruppo tramite l'osservazione dei comportamenti degli utenti all'interno dell'ambiente mentre il secondo consiste nel suggerire possibili azioni agli agenti che devono inserirsi nell'ambiente per far sì che essi si comportino in maniera adeguata.

Il Supporto alla Cultura Implicita migliora le performance di agenti in ambienti dove stanno agendo altri agenti con più conoscenza. Ciò è molto importante in quei sistemi, detti di raccomandazione, nei quali un alto livello di performance è uno dei requisiti fondamentali.

L'obiettivo della tesi è dimostrare che è possibile applicare l'approccio della Cultura Implicita ad un software basato sull'interazione tra agenti, come Jadex. Per fare questo andremo ad applicare il sistema SICS direttamente ad un esempio implementato dai costruttori di Jadex chiamato Hunter Prey. Da questa integrazione potremo inoltre ricavare dati utili su benefici apportati ed eventuali problemi riscontrati.

Nel primo capitolo andremo più a fondo nel problema della mancanza di conoscenza per poi arrivare a spiegare in che cosa consistono i reccomendation system; infine presenteremo un esempio pratico del loro uso.

Nel secondo capitolo si parlerà in maniera più approfondita del concetto di Cultura Implicita e andremo a presentare la struttura di SICS.

Nel terzo capitolo poi introdurremo la struttura e il funzionamento del software ad agenti Jadex e andremo a spiegare come applicare in pratica l'Cultura Implicita tramite SICS all'esempio di Jadex chiamato Hunter Prey.

Nel quarto e ultimo capitolo andremo a presentare i risultati ottenuti e gli eventuali problemi riscontrati nell'implementazione.

Capitolo 1
I Sistemi di Raccomandazione

Come già detto nell'introduzione quando un agente agisce in un ambiente senza le necessarie conoscenze ed abilità, l'efficacia del suo comportamento non può che essere inferiore a quella ottimale. Il problema diventa ancor più complicato se l'agente deve interagire con altri agenti che vivono abitualmente nello stesso ambiente ed hanno conoscenze riguardo ad esso. Si pensi ad esempio a quando un utente di Internet si collega per cercare informazioni utili su un argomento che non conosce oppure ad un commerciante che si affaccia su un nuovo mercato che non conosce. In entrambi i casi ci sarà un problema serio per gli attori in quanto a tutti e due mancherà la conoscenza dell'ambiente e degli attori che vi operano.

Per risolvere questo problema l'agente in questione dovrebbe agire secondo quanto la “cultura” del nuovo ambiente suggerisce, cioè secondo gli usi degli agenti che già vi operano all'interno. Un metodo di risoluzione pratica può essere quello di indurre gli agenti a comportarsi in maniera accettabile tramite il controllo dell'ambiente e la modifica della visione che l'agente ha di esso. Questo controllo può permette al sistema di dare dei suggerimenti agli agenti sul come comportarsi portando gli agenti ad una completa comunanza di comportamento che rappresenta la cosiddetta Cultura Implicita .Questa Cultura Implicita, che rappresenta una possibile soluzione al precedentemente esposto problema della mancanza di conoscenza in ambienti distribuiti, si inquadra in un più ampio panorama che comprende altre tecnologie che sono conosciute col nome di Sistemi di Raccomandazione. In questo capitolo andremo a presentare alcune caratteristiche comuni di alcuni di questi sistemi e i loro problemi attualmente riscontrati. Andremo poi a presentare un esempio pratico di uso di questi sistemi di Raccomandazione.

1.1 Sistemi di Raccomandazione: strategie di suggerimenti ad utenti poco esperti

Per definizione data da Burke [3] un sistema di Raccomandazione è

“qualunque sistema che produce raccomandazioni personalizzate come output o ha l'effetto di guidare l'utente in un modo personalizzato verso interessanti o utili oggetti in un largo spazio di azioni possibili”.

Essi sono diventati recentemente un potente mezzo di aiuto per gli utenti nella condivisione di risorse. Infatti hanno molteplici applicazioni pratiche, da quelle informatiche tradizionali ai più recenti sistemi di commercio elettronico, dove essi tentano di predire le preferenze degli utenti passando al setaccio grandi quantità di informazioni per scoprire elementi comuni interessanti. I principali campi in cui i sistemi sono stati applicati sono quelli dei film, della musica, dei libri, dei link web, degli hotel, etc...
I sistemi di Raccomandazione possono essere implementati con differenti tecniche di suggerimento che andranno ad influenzare i risultati e i comportamenti degli utenti che li usano. Le principali tecniche sono sostanzialmente due: quelle basate sull'approccio contenuto-dipendente e quelle basate sull'approccio collaborativo.

1.1.1 L'approccio contenuto-dipendente

In un sistema basato sui contenuti gli oggetti di interesse sono definiti dalle loro caratteristiche associate. Un suggeritore contenuto-dipendente impara un profilo degli interessi dell'utente, basato sulle caratteristiche presenti in oggetti che l'utente ha valutato. Da ciò deriva che il sistema suggerisce all'utente le raccomandazioni in base a ciò che piaceva nel passato allo stesso utente e che quindi si pensa che gli piaccia ancora.

1.1.2 L'approccio collaborativo

L'approccio collaborativo alle raccomandazioni è totalmente diverso [9]: invece che suggerimenti di oggetti dovuti al fatto che essi sono simili agli oggetti che piacevano all'utente nel passato, i sistemi di Raccomandazione di questo tipo suggeriscono all'utente oggetti che sono simili a quelli che altri utenti simili preferiscono. Infatti i suggerimenti per un utente sono fatti solamente sulla base di similarità con altri utenti. Esempi di sistemi di questo tipo includono GroupLens [4], suggeritore di video Bellcore [5] e Ringo[6].

1.1.3 Raccomandazioni, predizioni e valutazioni

Nell'area dei sistemi di Raccomandazione alcuni concetti-chiave sono usati per denotare differenti aspetti degli stessi. In questa sezione andremo a presentarne alcuni dei più importanti.

1.1.3.1 Suggeritore

“Un'entità, persona o modulo di software che produce suggerimenti come output o che ha l'effetto di guidare gli utenti in un modo personalizzato verso oggetti interessanti”

Un suggeritore è la parte attiva del sistema che che genera e fornisce le raccomandazioni agli utenti; un suggeritore può essere un pezzo di software così come una persona.
Si possono distinguere tre tipi di suggeritori: quelli che deliberano le informazioni agli utenti in maniera “tirata”, dove gli utenti fanno una specifica richiesta per raccomandazioni al sistema, quelli che le deliberano in maniera “spinta”, dove le raccomandazioni vengono spedite all'utente senza specifica richiesta e quelli che, come asseriscono Schafer, Kostan e Riedl [7], le deliberano in maniera “passiva o organica”, dove le informazioni sono deliberate nel naturale contesto del sistema.

1.1.3.2 Raccomandazioni

“Un oggetto o una lista di oggetti che è interessante per un utente secondo un suggeritore; la lista contiene solo quegli oggetti che vengono ritenuti abbastanza rilevanti da un suggeritore per un utente”

Di tutti gli oggetti che sono scoperti dal suggeritore soltanto quelli che esso ritiene

abbastanza importanti per l'utente. Un suggeritore prende la sua decisione attraverso la predizione di quanto interessante è ciascun oggetto per l'utente. Un suggerimento quindi consiste negli oggetti più predetti.

Nederland 1	Nederland 2	BBC 1
17:00-17:10 NOS-Journaal	**17:35-17:59** Voor alle fans: Tineke Schouten	**19:00-19:30** Nieuws en weerbericht
18:30-19:00 That's the question	**17:59-18:55** Twee Vandaag	**19:30-20:00** Regionaal nieuws en weerbericht
19:00-19:30 Het gevoel van de Vierdaagse	**18:00-18:25** NOS-Journaal	**20:00-21:00** Sahara with Michael Palin
19:30-20:00 De polikliniek	**18:25-18:55** Actualiteiten	**22:00-23:00** Undercover nurse: A panorama special
20:00-20:30 NOS-Journaal	**19:10-19:40** Lingo	
20:30-21:00 Netwerk	**21:15-21:30** NOS-Journaal	
22:50-23:25 Wat Zou JIJ Doen?	**21:30-22:30** Heartbeat vips	

Figura 1.1 Suggerimenti fatti da un sistema di Raccomandazione per la TV

1.1.3.3 Predizioni

“L'interesse anticipato di un utente riguardo un oggetto”

Suggeritori potrebbero o no ritornare predizioni con raccomandazioni; alcuni suggeritori semplicemente ritornano una lista di oggetti che loro raccomandano senza dare nessuna indicazione dell'interesse anticipato; altre raccomandazioni forniscono dettagliate informazioni circa le predizioni.

1.1.3.4 Valutazioni

“Un valore concreto rappresentante un interesse dell'utente, ad esempio un valore concreto che da' un indicazione di quanto un utente apprezza un oggetto”

Questo valore concreto è misurato secondo una certa scala, per esempio da 1 a 5 oppure da -10 a 10, che può essere presentato all'utente. Un oggetto può ricevere una valutazione per l'oggetto intero o valutazioni per vari aspetti di esso. La scala effettiva da usare è determinata dal designer del sistema di Raccomandazione

1.1.4 Problemi nei sistemi di Raccomandazione

In questa sezione discuteremo brevemente le varie problematiche che sono presenti all'interno dei sistemi di Raccomandazione.

1.1.4.1 Il problema della “partenza a freddo”

Uno dei principali problemi in molti sistemi di Raccomandazione è la cosiddetta partenza a freddo o problema di iniziazione (Maltz & Ehrlich, [8]); ciò si riferisce alla situazione in cui un suggeritore non ha sufficienti informazioni su un utente od un oggetto per fare buone raccomandazioni per esso. Ci sono tre tipi di problemi di partenza a freddo: nuovo utente, nuovo oggetto e nuovo sistema.
- il problema della partenza a freddo per un nuovo utente occorre quando un nuovo utente comincia ad usare un sistema di Raccomandazione; all'inizio c'è poca conoscenza di questo utente rendendo difficile per molte tecniche di predizione generare predizioni accurate e raccomandazioni.
- Il problema della partenza a freddo per un nuovo oggetto avviene quando un nuovo oggetto è aggiunto ad un sistema di Raccomandazione; alcune tecniche di predizione, come quelle collaborative, non sono in grado di generare predizioni per questi oggetti dato che non ci sono valutazioni date loro da utenti.
-Il problema della partenza a freddo per nuovi sistemi è una combinazione ingrandita di entrambi i problemi precedentemente accennati, dato che in nuovi sistemi di Raccomandazione non c'è niente di conosciuto sui (nuovi) utenti e non ci sono valutazioni per nessun oggetto.

Sistemi di Raccomandazione ibridi sono una soluzione al problema della partenza a freddo dato che questi sistemi incorporano diverse tecniche di predizione, facendo diminuire la probabilità che nessuna delle tecniche sia capace di fornire una predizione accurata. Comunque istanze di partenza a freddo possono ancora verificarsi.
Un'altra possibilità di risoluzione di questo problema è l'usare profili di utenti stereotipici (Ardissono et al., [10]) che sono dedotti dalle informazioni precedenti sull'utente; ad esempio categorizzazione di utenti e loro interessi basati su indagini.

1.1.4.2 Questioni di privacy, sicurezza e fiducia

I sistemi di Raccomandazione utilizzano informazioni di utenti, così come caratteristiche degli utenti, i loro interessi e opinioni nella forma di valutazioni per oggetti per fornire servizi che incontrano in modo migliore i bisogni di ogni utente individuale.
L'altra faccia della medaglia è che i sistemi informativi forniscono strumenti perfetti per venditori e altri per invadere la privacy degli utenti.
Quindi la privacy è un aspetto importante da prendere in considerazione non solo da un punto di vista morale ma anche da un punto di vista legale; gli utenti devono sapere o essere in grado di credere che la loro privacy sia garantita da un sistema di Raccomandazione.
La sicurezza è un importante fattore che assicura che i dati personali rimangano personali e che siano usati solamente da quelli aventi diritto d'accesso e di uso di quei dati.
Perfino all'interno di un sistema di Raccomandazione che ha i migliori metodi di di sicurezza è comunque ancora l'utente che deve decidere se fidarsi del sistema e delle persone che sono dietro di esso.
Un altro elemento di sicurezza è la possibilità di intrusi che provano a modificare le raccomandazioni fatte dal sistema nell'ordine di manovrare il sistema portandolo a fornire raccomandazione favorevoli per loro.; per esempio provano a rendere i loro prodotti raccomandati più spesso rispetto a quelli dei loro concorrenti.

1.1.4.3 Altre caratteristiche dei sistemi a Raccomandazione

Oltre a quelle fin qua discusse esistono altre caratteristiche che un sistema di raccomandazione deve avere: una è quella dei sitemi dipendenti dal contesto. Infatti le soluzioni attuali sono solitamente fatte su misura per le varie applicazioni e solo in rari casi sono liberi dal contesto stesso dove vengono utilizzati. Ciò limita la loro inclusione in applicazioni diverse da quella d'origine e quindi il loro riuso.

Una fondamentale qualità dei sistemi informativi deve essere anche l'interoperabilità, con l'obiettivo di permettere uno scambio su larga scala di dati.
Inoltre dietro alle questioni tecniche, l'interoperabilità include anche aspetti culturali e organizzativi. Infatti essere interoperabili significa essere capaci di gestire la cultura di un'organizzazione e conseguentemente di massimizzare le opportunità della condivisione delle informazioni.

1.2 Esempio di sistemi di Raccomandazione

La quantità di informazioni pubblicate in formato elettronico e il numero di utenti che vi accedono per soddisfare la loro necessità di informazioni giornaliere sta crescendo ad una velocità impressionante negli ultimi anni. Stranamente però nonostante più informazioni siano facilmente raggiungibili e in minor tempo rispetto ad una decina d'anni fa, sta diventando sempre più difficile per gli individui controllare ed effettivamente trovare informazioni tra il potenzialmente infinito numero di risorse di informazioni disponibili in internet. Trovare informazioni in un discreto periodo si tempo sta diventando sempre più complicato man mano che il numero di persone online cresce, a meno che uno sappia come, dove e cosa cercare esattamente.

1.2.1 Le Librerie Digitali (Dls)

Perciò nuovi servizi sono urgentemente necessari in internet per prevenire che gli utenti vengano sommersi dal flusso di informazioni disponibili.
Tra le varie tecnologie, le Librerie Digitali (Dls) giocheranno un importante ruolo non semplicemente in termini di fornitura di informazioni, ma in termini di servizi che essi forniscono alla società d'informazione. Informalmente, le Dls possono essere definite come collezioni di informazioni che hanno servizi associati, consegnate alle comunità degli utenti tramite una varietà di tecnologie. Fondamentalmente infatti dobbiamo compiere uno shift concettuale nel nostro modo di intendere le Dls: dove la classica visione delle Dls è la manipolazione di dati da parte di individui isolati, la nostra visione delle Dls è quella di manipolazione e scambio di dati ed informazioni così come la cooperazione da parte di individui consci del loro ambiente così come degli altri utenti.
Riguardo la mansione di ricerca di informazioni, la raccomandazione di elementi basati sugli schemi di preferenze degli altri utenti è probabilmente la più importante. L'uso di opinioni e conoscenza di altri utenti per prevedere che il valore rilevante degli elementi sia raccomandato ad ogni utente in una comunità è conosciuto come Collaborative o Social

Filtering. Questi metodi sono basati sull'assunzione che un buon modo di trovare contenuti interessanti è trovare altri utenti che hanno interessi comuni e così raccomandare elementi a cui quegli utenti sono interessati.

1.2.2 Un'applicazione delle DIs: CYCLADES

Un'istanza del modello di un ambiente che usa collaborative DIs è implementato nel sistema chiamato CYCLADES [1].
L'obiettivo di CICLADES è quello di fornire un ambiente integrato per utenti e gruppi di utenti (comunità) che vogliono usare, in un fortemente personalizzato e flessibile modo, “archivi aperti”. CYCLADES fornisce un ambiente di archivi virtuali, che (tra gli altri) supporta gli utenti e le comunità con funzionalità per

- ricerca avanzata in archivi digitali larghi, eterogenei e multidisciplinari;
- collaborazione;
- filtraggio, e raccomandazioni.

Da un punto di vista logico possiamo rappresentare la funzionalità del sistema CYCLADES come in figura 1.1.

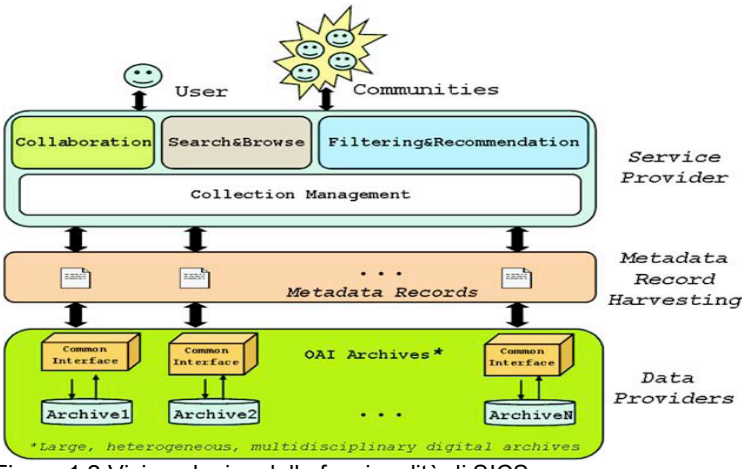


Figura 1.2 Visione logica della funzionalità di SICS

La figura 1.2 invece presenta la principale interfaccia dell'utente. In essa si può riconoscere la cartella home (livello principale) di un utente. Esso contiene svariate cartelle. Tra esse ci sono alcune cartelle (condivise) appartenenti alle comunità (create da qualcuno) alle quali l'utente può unirsi, mentre altre sono cartelle private e sono state create direttamente dall'utente. Queste cartelle contengono alcuni record di comunità o di utenti. Alcuni record sono stati valutati e altri hanno allegato delle note. C'è anche un forum di discussione. Queste funzioni sono solo alcune di quelle contenute nel package di supporto.
Il sistema CYCLADES inoltre fornisce già alcuni record, comunità, collezioni e raccomandazioni dell'utente ritenute dal sistema rilevanti per queste cartelle.

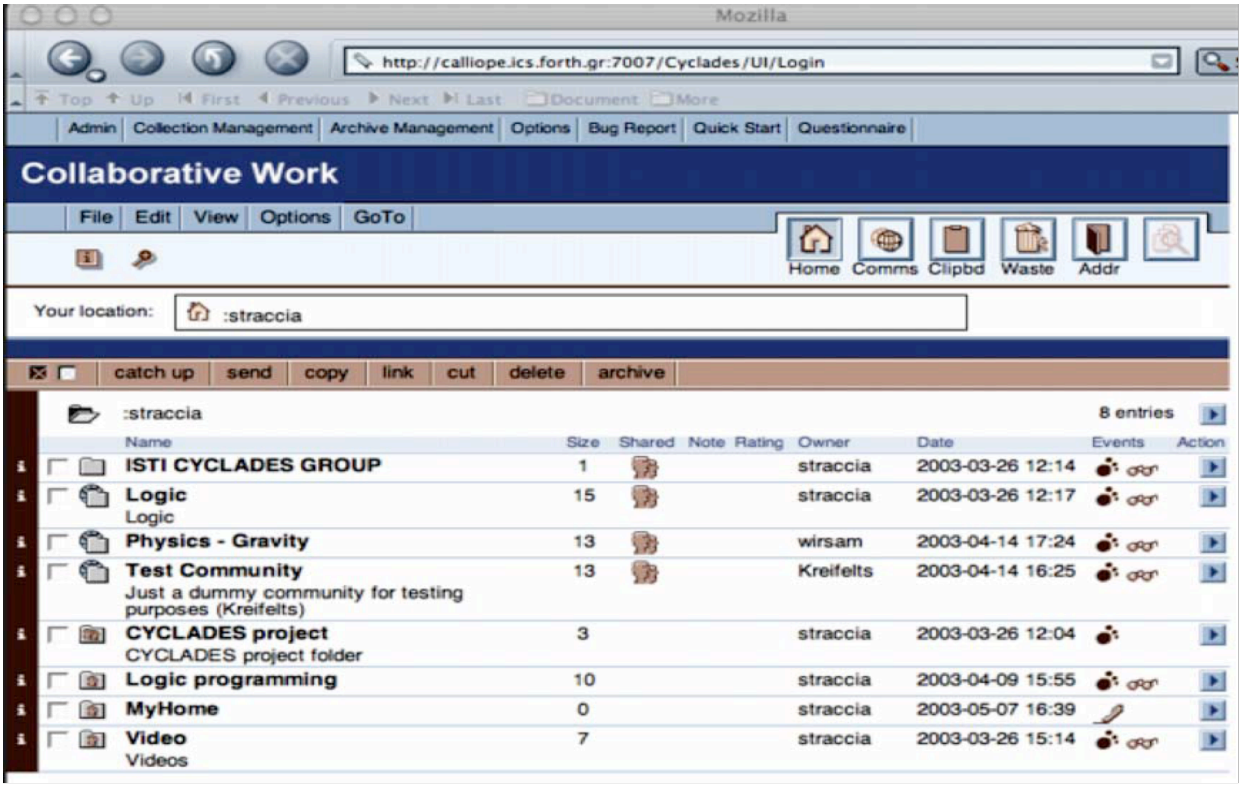


Figura 1.3 Interfaccia utente: cartella home

1.3 La Cultura Implicita nei sistemi di raccomandazione

Il Supporto alla Cultura Implicita migliora le performance degli agenti in ambienti in cui stanno già agendo altri agenti più esperti. Questo concetto può aiutare nell'analisi di sistemi già esistenti e può suggerire idee per nuovi. Per questo il supporto alla Cultura Implicita può essere utile in quelle applicazioni, in particolare, basate sulla gestione della conoscenza e in quei sistemi per lo studio di profili o basati sull'interazione di agenti diversi.

1.4 Conclusioni

La difficoltà di orientarsi nella ricerca di informazioni in internet oppure i problemi che si presentano quando un nuovo utente deve affacciarsi in un nuovo ambiente sono tutti esempi che rappresentano in maniera chiara il problema della mancanza di conoscenza all'interno di un ambiente nuovo. Per risolvere il problema si è lavorato negli ultimi anni ad una serie di sistemi detti sistemi di Raccomandazione che tramite l'aiuto di diversi strumenti sostengono l'utente nel districarsi all'interno degli ambienti distribuiti più complessi e lo aiutano a soddisfare in maniera semplice e ottimale le proprie necessità tra le diverse offerte. In questo capitolo abbiamo dato uno sguardo allo stato dell'arte dei vari sistemi di Raccomandazione analizzando le caratteristiche comuni e i problemi riscontrati. Inoltre abbiamo dato un esempio di sistema già esistente.

Capitolo 2
L’approccio della Cultura Implicita

2.1 Cultura Implicita

“ *La Cultura Implicita (CI) è una relazione tra un set e un gruppo di attori, tale che gli elementi del set si comportano secondo la cultura del gruppo “*

Il concetto di Cultura Implicita è stato introdotto per la prima volta nel 2001 da Blanzieri e Giorgini [12, 13, 14]. Quando una persona o un ente qualsiasi deve agire e comportarsi all'interno di un nuovo ambiente sociale o di altro tipo il suo comportamento molto probabilmente è lontano dall'essere ottimale.
Possiamo pensare a molte situazioni diverse dove, a causa della mancanza di conoscenza, diventa molto difficile per la persona prendere la decisione esatta. Questo potrebbe non essere il caso per persone che hanno precedentemente vissuto altre situazioni molto simili.

Infatti, essi potrebbero aver ottenuto le conoscenze necessarie per poter comportarsi correttamente all'interno dell'ambiente. Questa conoscenza è generalmente detta implicita e rappresenta una sorta di “conoscenza di comunità” in quanto è di carattere generale ed è condivisa da tutti.

La Cultura Implicita si basa sul presupposto che è possibile estrapolare la cultura di comunità osservando semplicemente l'interazione tra persone all'interno del e con l'ambiente e incoraggiare i neo-arrivati a comportarsi similmente alle persone già esperte.

La Cultura Implicita è basata inoltre su diversi elementi che ne compongono la struttura e ne permettono la messa in atto pratica: la CI assume che *agenti* compiano delle *azioni* su *oggetti* all'interno di un *ambiente.* Le azioni sono inoltre considerate nel contesto delle diverse *situazioni .* La “cultura” contiene informazioni circa le azioni e le loro relazioni con le situazioni, cioè quali azioni vengono compiute da quali attori e in quali circostanze. Queste informazioni sono poi utilizzate per istruire i neo-arrivati sul comportamento degli altri in situazioni simili.

Più approfonditamente presentiamo ora un esempio concreto di come può essere applicata l'Cultura Implicita.

2.2 L'esempio del compratore di libri

In questo esempio assumiamo, come è di prassi per l'uso dell' Cultura Implicita, che ogni agente agisce in un ambiente composto da oggetti e da altri agenti. Azioni hanno come argomenti oggetti, come ad esempio *offer(book1, price1)* or *demand(book2, price2),* agenti, come in *look_for(buyer)* oppure *ask_about(seller)* oppure entrambi oggetti e agenti, come in *send(message,seller).*

Prima di eseguire un'azione un agente si interfaccia con una scena formata da una porzione di ambiente, cioè oggetti ed altri agenti, e azioni che sono possibili in esso. Per esempio un agente *buyer* si interfaccia con *seller1, seller2, book1, gadget1, price1, price2* e può eseguire *buy_from(seller1, book1,price1), buy_from(seller2, gadget1, price2)* e *buy_nothing().* Quindi un agente esegue un'azione in una data situazione, cioè l'agente si interfaccia con una scena in un dato momento, così che l'agente esegua un'azione relata alla situazione. L'agente buyer ha eseguito l'azione *buy_from(seller1, book1, price1)* mentre stava interfacciando la scena composta da *seller2, price2* e le azioni possibili.

L'azione eseguita che uno specifico agente dipende dal suo stato interno ed inaccessibile e in generale non può essere deterministicamente preventivabile. Invece, noi assumiamo che può essere caratterizzato in termini di probabilità e aspettative.

Per esempio dato un *buyer* interfacciante una scena nella quale può essere rappresentato *buy_from(seller1, book1, unreasonable_price), buy_from(seller2, book1, low_price*) or *buy_nothing()* l'azione attesa può essere *buy_from(seller2, book1, low_price*).

Dato un gruppo di agenti supponiamo che esista una teoria circa le loro azioni probabili. Se la teoria è coerente con le azioni eseguite del gruppo, può essere considerata come un

valido vincolo culturale per il gruppo. La teoria cattura la conoscenza e le capacità dei membri all'interno dell'ambiente. Quindi con la teoria possiamo predire la più probabile delle azioni che l'agente andrà ad eseguire, nel nostro caso l'azione eseguita del *buyer* sarà l'attesa azione *buy_from(seller2, book1, low_price*).

Se un gruppo di nuovi agenti compie azioni che soddisfano il valido vincolo culturale del gruppo il problema del loro comportamento sub-ottimo in relazione al gruppo è risolto.

Avere un gruppo di agenti tale che le loro azioni soddisfino un valido vincolo culturale di un altro gruppo senza bisogno di conoscerlo approfonditamente realizza ciò che noi chiamiamo Cultura Implicita.

2.3 Un'implementazione del Sistema di Supporto per la Cultura Implicita (SICS)

Nella prossima sezione andremo a presentare la struttura e le caratteristiche del Sistema di Supporto per la Cultura Implicita (SICS).

2.3.1 Caratteristiche principali

Questo servizio fornisce un accesso semplice e configurabile al Sistema per il Supporto alla Cultura Implicita (SICS), che è implementato adottando il Service Oriented Architecture (SOA). Quindi esso può essere semplicemente aggiunto ai software esistenti in qualunque campo, descritto in termini di agenti, oggetti, azioni ed attributi. Ciò permette lo sviluppo di complesse applicazioni che includono strumenti di suggerimenti che usano l'idea della *composizionalità* del software invece della loro costruzione partendo da zero. L'uso di SOA fornisce un'*interoperabilità* tecnica e garantisce una fluida comunicazione tra vari agenti in giro per il mondo.

Un'istanza di Ic-Service può essere configurata per essere accessibile da differenti applicazioni. Molti Ic-Services inoltre possono collaborare per facilitare la condivisione della conoscenza (ad esempio tra comunità diverse).

Un meccanismo di configurazione flessibile permette un alto livello di *riusabilità* che fornisce una rapido e poco costoso modo di inserire il servizio di suggerimenti in ogni sistema.

La SOA è stata scelta tra le possibili architetture perché essa supporta i principi di accesso universale e indipendenza da piattaforma e permette ai servizi di suggerimento di essere trasparentemente posti all'interno o all'esterno dell'impresa.

L'Ic-Service è una soluzione generale che può essere facilmente usata sia applicazioni sotto sviluppo che in sistemi già esistenti, in particolare in complesse

applicazioni software dove la personalizzazione dell'utente e il suggerimento dell'item/azione sono stati esclusi dai requisiti per colpa dei loro costi addizionali.

L'Ic-Service può essere anche applicato al problema dei suggerimenti nel cross-selling (la pratica di mettere accessori o prodotti complementari o in relazione vicini tra di loro in un negozio), dove i dati circa il cliente provengono da differenti rivenditori e devono essere integrati.

2.3.2 La struttura

Fondamentalmente SICS compie un'azione chiave per la Cultura Implicita, chiamata “trasferimento di conoscenze”, tra due diversi gruppi di attori (uno di attori neo-arrivati, quindi senza conoscenze dell'ambiente e uno di attori già precedentemente presenti nell'ambiente, quindi con un'esperienza maggiore) che rappresenta la base della nostra Cultura Implicita.

L'architettura generale di SICS consiste dei seguenti tre componenti:

- L'*Osservatore*, la parte di SICS che salva in un database le informazioni circa le azioni eseguite dall'utente;
- IL *Modulo Induttivo*, che analizza le osservazioni salvate e implementa delle tecniche di data mining per scoprire modelli di comportamento dell'utente.
- Il *Composer,* che sfrutta le informazioni raccolte dall'Osservatore e la teoria nell'ordine di suggerire le azioni in una data situazione.

L'Osservatore supporta due possibili facilitazioni di salvataggio:

file XML e salvataggio in un database. I file XML offrono una soluzione semplice, facilmente estendibile e portabile per applicazioni dove la storia delle osservazioni non è grande e a cui non si deve accedere spesso.

La variante del database dovrebbe essere scelta quando si ha a che fare con applicazioni più complesse con una grande quantità di dati.

Il Composer opera con l'astrazione architetturale della situazione -

cioè la scena, che contiene un set di oggetti e un set di azioni che possono essere compiute da questi oggetti.

Nel Composer sono inclusi i seguenti due sotto-moduli:

- Il *Cultural Action Finder (CAF) ,* che filtra quelle azioni che soddisfano la teoria;
- Il *Scene Producer,* che ricerca le scene dove le azioni (trovate dal CAF) sono più probabilmente eseguite.

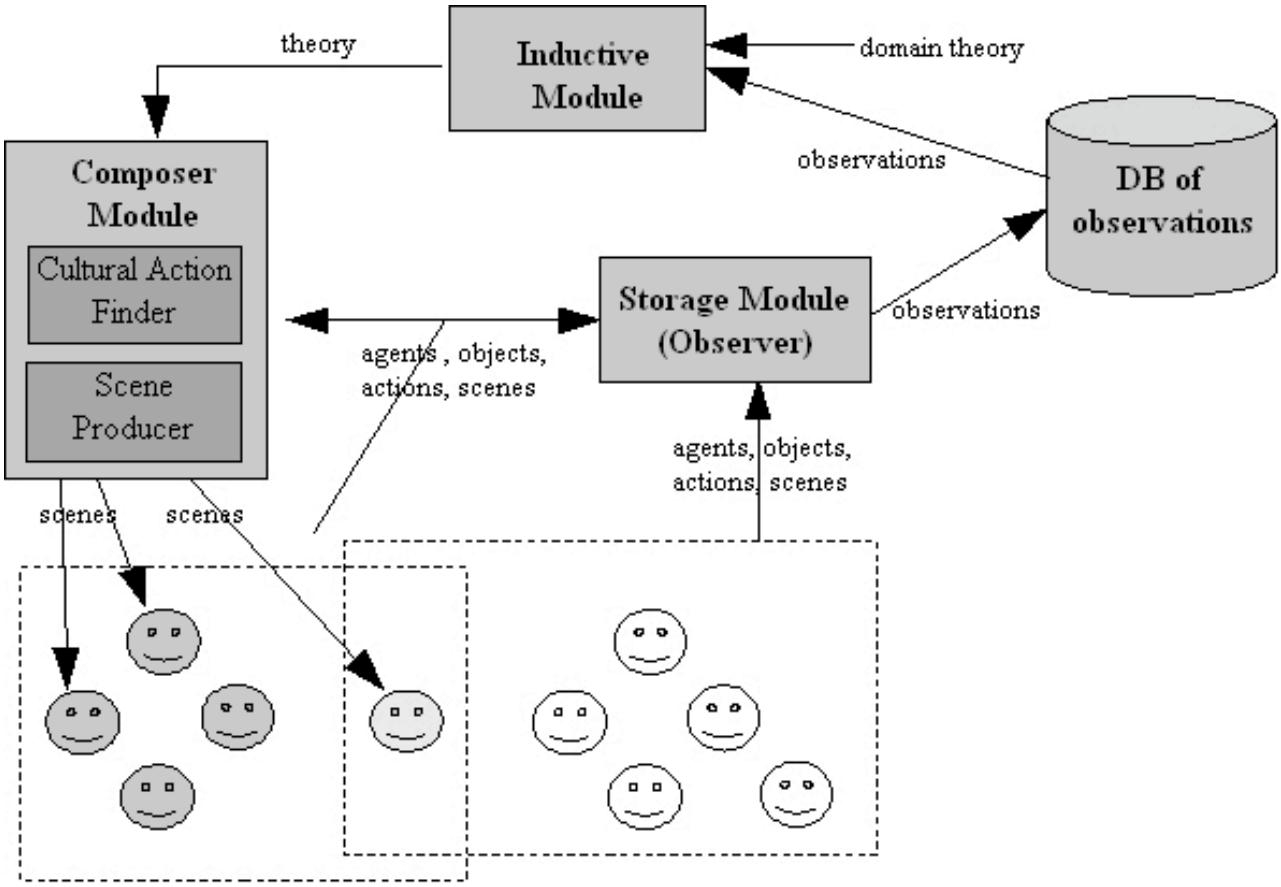


Fig. 2.1 L'architettura generale del System for Implicit Culture Support (SICS)

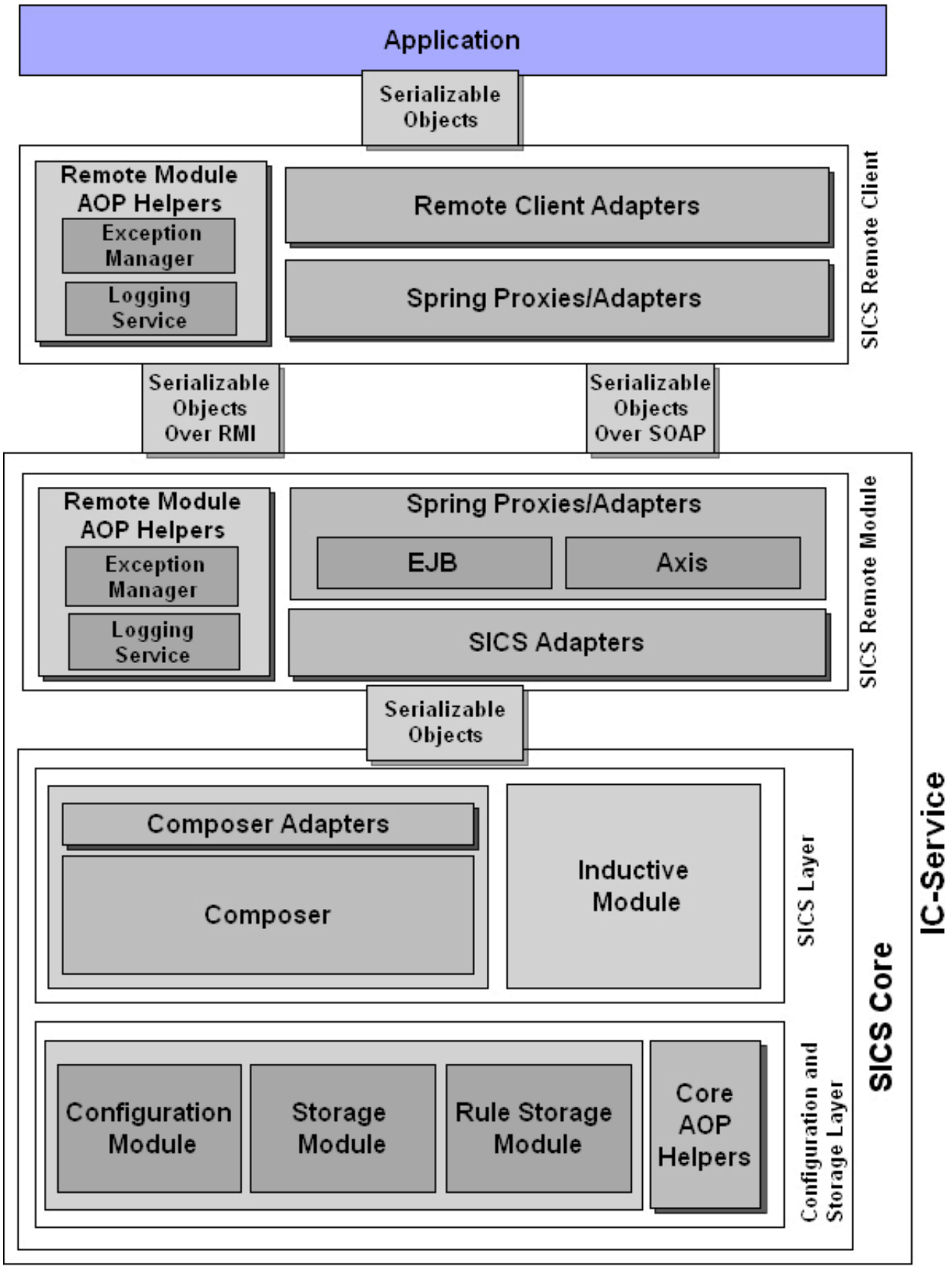


Fig. 2.2. L'architettura dettagliata del SICS.

Dalla figura 2.1 possiamo vedere come la *teoria* (rappresentante la “cultura comunitaria”) è estratta dalle osservazioni sul set di agenti ed è poi usata per produrre suggerimenti per un altro set di agenti.

Questi due set possono essere disgiunti, possono sovrapporsi o coincidere.

La *teoria* è basata sulle regole e contiene due parti: una parte usata dal Composer può essere specificata a priori (*teoria di campo*) mentre l'altra è imparata dal Modulo Induttivo e può evolvere nel tempo. Per esempio, per il generale problema di fornire alle persone suggerimenti, la teoria di campo può dire che il sistema deve suggerire gli item che sono più probabilmente accettati dall'utente, mentre la teoria imparata dal Modulo Induttivo può contenere informazioni circa quali item sono accettati.

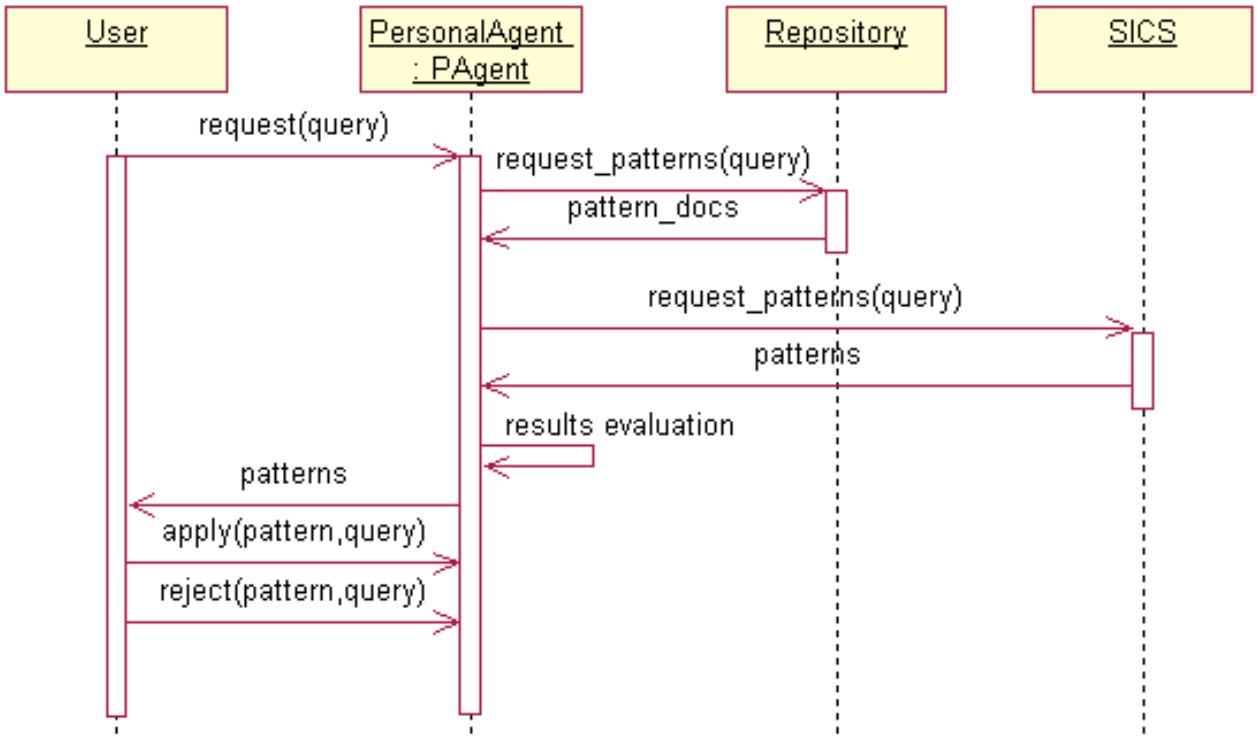


Fig. 2.3. Diagramma di sequenza del processo di ricerca.

Azione	Oggetti	Attributi
Richiesta	descrizione_problema	nome_progetto
Applicata	schema, descrizione_problema	nome_progetto
Rifiutata	schema, descrizione_problema	nome_progetto

Tabella 2.1: Le azioni che possono essere osservate dal sistema

2.3.3 Scenari d'uso

Il SICS può essere usato all'interno di un'applicazione in tre modi differenti, come si può vedere nella figura 2.4:

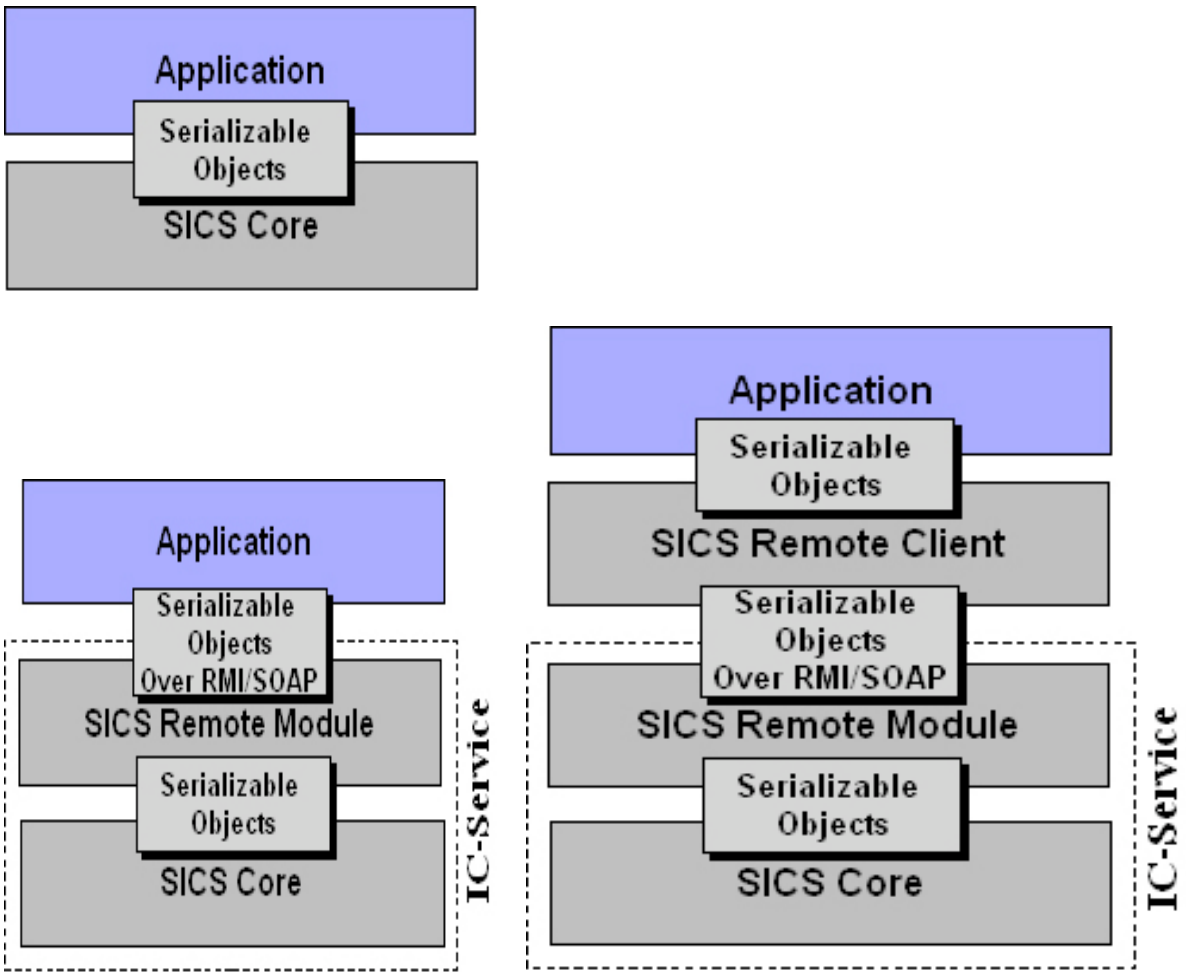
i. SICS può essere incluso nell'applicazione come una libreria(Figura 2.4(a)).

   In questo caso il SICS *Core* ha a che fare direttamente con gli oggetti, le azioni, ecc... dell'applicazione. Questa modalità dovrebbe essere scelta quando l'applicazione non è necessariamente distribuita e può essere strettamente accoppiata con la libreria.

ii. Per permettere l'accesso remoto (Figura 2.4(b)), il core di SICS può essere invocato attraverso il SICS *Remote Module* come un web service SOAP. Questo scenario dovrebbe essere seguito quando il servizio è una parte di un sistema distribuito ma per alcune ragioni (per esempio risorse limitate del client, come nei dispositivi portatili) non c'è bisogno o possibilità per usare il SICS *Remote Client.* Comunque in questo caso l'applicazione deve fare attenzione alla comunicazione col servizio.

iii. Il modo più semplice per aggiungere servizi di suggerimenti in un'applicazione è di accedere all'Ic-Service attraverso il SICS *Remote Client* (Figure 2.4(c)) che nasconde i dettagli tecnici del meccanismo di comunicazione dal designer dell'applicazione. Questa modalità dovrebbe essere adottata quando abbiamo a che fare con applicazioni complesse e l'Ic-Service deve essere introdotto in un modo completamente disaccoppiato.

(a) SICS in una libreria



(b) SICS come un servizio

(c) SICS come un servizio attraverso SICS Remote Client

Figure 2.4: Scenari di invocazione di SICS

Gli scenari descritti illustrano la possibilità di includere l'Ic-Service in varie applicazioni comprese tra quelle di piccole dimensioni e quelle di grandi dimensioni come sistemi distribuiti complessi.

L'Ic-Service può essere inoltre aggiunto in un'applicazione in una maniera completamente disaccoppiata e vi si può accedere da qualunque parte in qualunque momento. Ciò garantisce *ubiquità,* permettendo al sistema di produrre suggerimenti usando dati raccolti da differenti risorse. Per esempio come già detto l'ubiquità è particolarmente utile nel fornire suggerimenti per il cross-selling.

Molti Ic-Service comunicanti possono essere visti come blocchi di un edificio nello sviluppo di un efficiente e robusto sistema di suggerimenti decentralizzato. Allo stesso tempo Ic-

Service è un'applicazione general-purpose e domain-indipendente che fornisce strumenti per il salvataggio, l'analisi e le deduzioni circa il comportamento osservato.

Nella prossima sotto-sezione analizziamo la struttura e la funzione di SICS che è strettamente correlata con l'Ic-Service.

2.4 Conclusioni

In questo capitolo abbiamo visto come il concetto di Cultura Implicita può essere una soluzione ottimale al problema della mancanza di conoscenza in un sistema distribuito. Inoltre abbiamo visto anche che il Sistema per il Supporto alla Cultura Implicita (SICS) è un potente mezzo per l'applicazione della Cultura Implicita nelle situazioni pratiche concernenti l'informatica. Esso può infatti essere usato in maniera differente, come illustrato nella sezione 2.3.3, in base allo scenario e alle caratteristiche delle applicazioni in cui verrà usato.

Capitolo 3
Un’implementazione in Jadex nell’esempio Hunter Prey

In questo terzo capitolo andremo ad analizzare più da vicino le funzionalità di SICS attraverso la sua applicazione all’esempio di Jadex chiamato Hunter Prey. Prima daremo uno sguardo alla struttura e alle caratteristiche generali del software Jadex, derivante dal più famoso programma ad agenti Jade. Jadex è stato pensato e creato da un gruppo di sviluppatori dell’università tedesca di Amburgo.

3.1 Jadex: software per la creazione di programmi ad agenti

In questa sezione andremo a presentare la struttura ad agenti di Jadex. Esso supporta semplici capacità di ragionamento attraverso lo sfruttamento del modello BDI, combinandolo con le tecniche di ingegneria del software già allo stato dell'arte come Java e XML.

3.1.1 Il modello BDI

La parola BDI è una sigla derivante dall'inglese “Belief-Desire-Intention” che significano “Convinzione-desiderio-intenzione”, cioè le tre basi su cui si fonda Jadex. Questo modello è stato scelto dagli sviluppatori di Jadex per la sua semplicità e per il suo background psicologico popolare, cioè un insieme di nozioni psicologiche che più si avvicinano al modo in cui gli umani parlano di aspetti comportamentali . Ma cosa sono esattamente le tre basi del modello BDI?

- *Beliefs:* sono attitudini informative di un agente, cioè i belief rappresentano le informazioni che un agente ha del mondo in cui vive e del loro stato interno. I belief forniscono inoltre un'astrazione dominio-dipendente delle entità attraverso la sottolineatura di importanti proprietà omettendo allo stesso tempo dettagli poco importanti. Ciò introduce una personale visione del mondo all'interno dell'agente: il modo in cui esso percepisce e considera il mondo.
- *Desires:* le attitudini comportamentali degli agenti sono catturati in desideri. Essi rappresentano i desideri degli agenti e guidano il corso delle loro azioni. I desideri non hanno bisogno necessariamente di essere in accordo e quindi possono non essere raggiunti simultaneamente. Un processo di “delibera di goal” ha il compito di selezionare un sottogruppo di desideri in accordo (spesso chiamati goal).
- *Plans:* sono i mezzi attraverso i quali gli agenti raggiungono i loro goal e reagiscono ad eventi che succedono. Quando un agente decide di perseguire un determinato goal con un certo plan, si assegna momentaneamente a questo tipo di creazione del goal e così ha stabilito una cosiddetta *intenzione* verso una sequenza di azioni presenti nel piano.

3.1.2 Architettura di Jadex

Visto da fuori un agente è come una scatola nera che riceve e spedisce messaggi. Messaggi entranti, così come eventi interni e nuovi goal funzionano da input per le azioni interne dell'agente e per il meccanismo di delibera. Basandosi sui risultati del processo di delibera questi eventi sono inviati ai plan già in funzione oppure a nuovi plan creati dalla apposita libreria. I plan già funzionanti possono poi accedere e modificare i belief, mandare messaggi ad altri agenti e causare eventi interni.
Il comportamento comunque di uno specifico agente è determinato esclusivamente dai suoi concreti belief, goal e plan. Qui di seguito andremo a descrivere in dettaglio ognuno di questi concetti centrali dell'architettura BDI di Jadex:

- *Beliefs:* Un obiettivo del progetto Jadex è la facilità d'uso. Perciò Jadex non obbliga una rappresentazione dei belief basata sulla logica. Al contrario, gli oggetti Java ordinari di qualunque tipo possono essere contenuti nel beliefbase. Gli oggetti sono salvati come fatti nominativi (chiamati belief) o set nominativi di fatti (chiamati beliefset).

Usando i nomi dei belief, il beliefbase può essere direttamente manipolato settando, aggiungendo o rimuovendo fatti.

```
01 <belief name="alarm_time" class="long">
02 <fact>System.currentTimeMillis()+360000</fact>
03 </belief>
04
05 <belief name="system_time" class="long" updaterate="1000">
06 <fact>System.currentTimeMillis()</fact>
07 </belief>
08
09 <beliefset name="alarm_times" class="long">
10 <fact>$beliefbase.system_time+360000*2</fact>
11 <fact>$beliefbase.system_time+360000*3</fact>
12 </beliefset>
13
14 <beliefset name="alarm_times_from_db" class="long">
15 <facts>Database.queryAlarmTimes()</facts>
16 </beliefset>
```

FIGURA 3.1. Esempio di un belief e di un beliefbase

- *Goals*: Jadex segue l'idea generale che i goal sono desideri concreti e momentanei di un agente. Per qualunque goal che egli ha, un agente verrà più o meno direttamente impiegato in azioni adatte, affinché esso non considera il goal come raggiunto, irraggiungibile o non più desiderato. Quando un goal è adottato, esso diventa un opzione che è aggiunta alla struttura dei desideri dell'agente. Un meccanismo di delibera è responsabile della gestione delle transizioni degli stati di tutti i goal adottati.

Jadex supporta quattro tipi di goal che mostrano diversi comportamenti in relazione ai loro processi:

- Un *perform* goal è direttamente relato con l'esecuzione delle azioni. Quindi un goal è considerato raggiunto quando alcune azioni sono eseguite, indipendentemente dal risultato delle azioni.

- Un *achieve* goal è un gol nel senso più comune e cioè definisce un un desiderato risultato senza specificare come è stato raggiunto.

- Un *query* goal è simile ad un achieve goal. Il suo risultato non è definito come uno stato del mondo, ma come alcune informazioni che l'agente ha bisogno di sapere.

- Un *maintain* goal, dove un agente tiene traccia dello stato desiderato ed eseguirà sempre dei plan appropriati per ristabilire lo stato da mantenere dove necessario.

```
01 <performgoal name=”play_song”>
02 <parameter name=”song” class=”MediaLocator”/>
03 </performgoal>
04
05 <achievegoal name=”notify_user” retrydelay=”600000” exclude=”never”>
06 <creationcondition>
07 $beliefbase.system_time==$beliefbase.alarm_time
08 </creationcondition>
09 <targetcondition>$beliefbase.user_notified</targetcondition>
10 </achievegoal>
11
12 <querygoal name=”retrieve_song”>
13 <parameter name=”song_name” class=”String”/>
14 <parameter name=”song” class=”MediaLocator” direction=”out”/>
15 </querygoal>
16
17 <maintaingoal name=”keep_clock_adjusted”>
18 <maintaincondition>
19 Math.abs($beliefbase.system_time-$beliefbase.reference_time)<500
20 </maintaincondition>
21 </maintaingoal>
```

FIGURA 3.2. Esempio di goal

- *Plans:* il motore razionale tratta tutti gli eventi, come una ricezione di messaggi o l'attivazione di un goal, attraverso la selezione e l'esecuzione di plan appropriati. Invece che eseguire piani ad-hoc per ogni evento, Jadex usa un approccio tramite una libreria di piani per rappresentare i piani di un agente. Per ogni piano un'introduzione definisce le circostanze sotto le quali il piano potrebbe essere selezionato e un corpo specifica le azioni che devono essere eseguite. Le parti più importanti dell'introduzione sono i goal e/o gli eventi che il piano può trattare e una referenza al corpo del piano.
- *Agent Definition:* per creare ed inizializzare un agente il sistema ha bisogno di conoscere le proprietà dell'agente che deve essere creato. Questo stato iniziale di un agente è determinato tra le altre cose dai belief, dai goal e dalla libreria dei piani conosciuti. Jadex usa un approccio dichiarativo e procedurale per definire i componenti di un agente. I corpi dei piani devono essere implementati come ordinarie classi Java che estendono una certa classe strutturale, fornendo così un generico accesso alle specifiche facilitazioni di BDI.

Per un completo motore razionale alcuni componenti aggiuntivi sono necessari. Il cuore dell'architettura BDI è ovviamente il meccanismo di selezione dei plan. I plan non devono essere selezionati solo per i goal, ma anche anche per eventi interni e messaggi entranti. Per raccogliere messaggi entranti e spedirli al meccanismo di selezione dei plan un componente specializzato è necessario. Un altro meccanismo è necessario per eseguire i plan selezionati e per tenere traccia dei passi del plan per scoprire eventuali fallimenti. In Jadex tutte le funzionalità richieste sono implementate in componenti chiaramente separati. Le informazioni importanti circa i belief, i goal e i plan sono salvate in strutture dati accessibili da tutti questi componenti.

3.1.3 Un esempio d'uso: Hunter Prey

All'interno di Jadex sono presenti alcuni esempi già precedentemente implementati che ci mostrano l'uso e la funzione del software. Per il nostro lavoro abbiamo scelto quello che ci sembrava più adatto al nostro caso e lo andremo adesso a presentare.

Hunter Prey

Lo scenario di Hunter Prey consiste in due tipi di creature che vivono in un mondo formato da una griglia. Il compito base degli hunter è quello di cacciare mentre i prey si muovono attorno andando alla ricerca di cibo. Entrambi i tipi di creature devono agire autonomamente nell'ambiente sulle basi della loro visione locale corrente, delle esperienze fatte in passato e della comunicazione con gli altri. Oltre a hunter e prey l'ambiente sistema altri oggetti passivi all'interno del mondo. Da una parte ci sono alberi su molti quadrati che proibiscono le creature di passare su tali campi e dall'altra piccole piante crescono su quadratini a random sulla mappa. Queste piante possono essere mangiate dai prey se sono sui campi medesimi. Lo scenario è a base rotonda con uno slot temporale fissato per ogni turno. Ciò significa che tutte le creature nel mondo devono compiere la loro azione successiva (muoversi in qualche quadratino adiacente o mangiare qualcosa sul quadratino attuale) seguendo quel turno temporale. Se nessuna azione è annunciata nessuna azione sarà eseguita. L'ambiente deciderà ad ogni turno se un'azione riesce o fallisce.

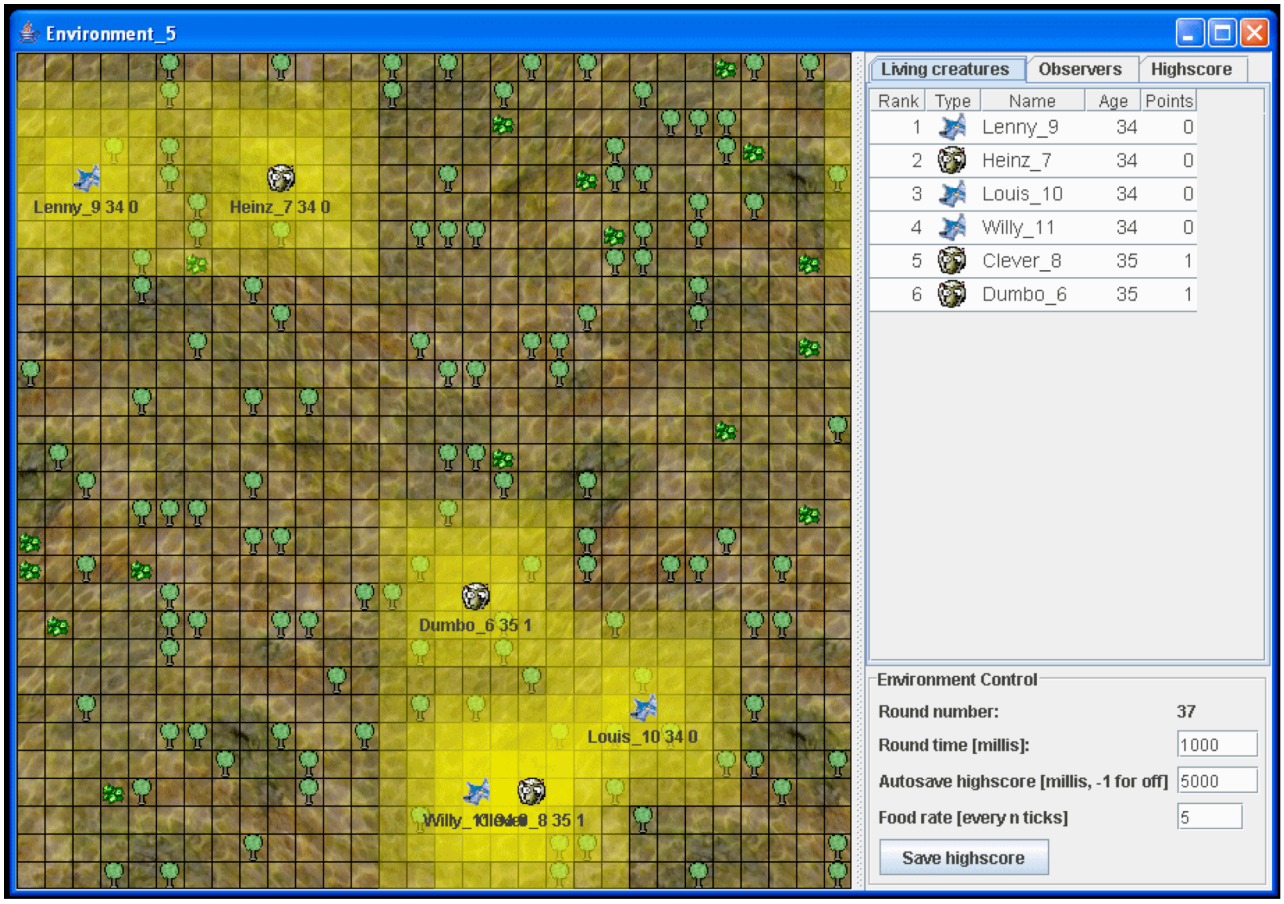


Figura 3.3.La griglia di Hunter Prey con i vari elementi

3.2 Realizzazione della Cultura Implicita nel caso dell'Hunter Prey

Il nostro obiettivo sulla strada che porta all'applicazione della Cultura Implicita, inizialmente, consiste nel modificare i prey esistenti in modo che essi salvino le loro osservazioni nel database.

Per prima cosa però spiegherò come sono strutturati i diversi tipi di prey, dato che essi non hanno tutti una struttura identica.

3.2.1 Struttura dei prey

In Hunter Prey esistono due diversi tipi di prey: quelli con struttura propria, cioè composta da file di loro uso esclusivo, e quelli con struttura “in comune”, cioè che usano alcuni file condivisi da altri prey. Un esempio del primo tipo è dumbprey, cioè il prey “ignorante”, che non ha nessuna intelligenza propria, mentre un esempio del secondo è lazyprey, il prey “pigro”, cioè quello che se non inseguito da nessun hunter rimane fermo senza compiere nessuna azione.

Analizziamo ora la struttura del primo tipo di prey: essa è composta da un file xml,cioè il file agent, e da un file Java.

Nel file xml, simile anche per l'altro tipo di prey, vengono indicati tutti quegli elementi che, come abbiamo visto precedentemente, sono la base del funzionamento di Jadex. Qui infatti vengono indicati i belief (Fig. 3.4),

```
<beliefs>
<!-- The creature →
<belief name="my_self" class="Creature">
<assignto ref="actsensecap.my_self"/>
<fact>new Prey($agent.getName(), (AgentIdentifier)$agent.getAgentIdentifier(), new Location(10,10))</fact>
</belief>
<!-- The current vision. -->
<beliefref name="vision">
<concrete ref="actsensecap.vision"/>
</beliefref>
</beliefs>
```

Figura 3.4 I belief di dumbprey

i goal (Fig. 3.5),

```
<goals>
<achievegoalref name="move">
<concrete ref="actsensecap.move"/>
</achievegoalref>
<achievegoalref name="eat">
<concrete ref="actsensecap.eat"/>
</achievegoalref>
</goals>
```

Figura 3.5 I goal di dumbprey

e i plan (Fig. 3.6),

```
<plans>
<!-- Dumb move plan. -->
<plan name="dumbmove">
<body>new DumbPreyPlan()</body>
</plan>
</plans>
```

Figura 3.6 Il plan di dumbprey

assieme ad altri componenti come l'head, lo stato iniziale, etc... necessari al regolare funzionamento del prey.

Come si vede dalla figura 3.6 nei plan c'è il riferimento al secondo file facente parte della struttura del prey: esso è un'istanza del file Java, in questo caso chiamato DumbPreyPlan. In esso è contenuto il codice che permetterà al prey di raggiungere tutti i suoi goal, cioè, come già detto nelle sezioni precedenti, fuggire dall'hunter, eventualmente mangiare, etc...

La struttura del secondo tipo di prey è, come abbiamo già detto, leggermente diversa. Essa infatti è composta da due file xml e numerosi file Java. Il primo file xml, cioè il file agent è identico a quello dell'altro prey senonché per il riferimento al file Java che qua non esiste più ma è sostituito da un riferimento ad un altro file xml, chiamato

BasicBehaviour.capability. Questo file è condiviso da tutti i prey dello stesso tipo, ha la stessa struttura dei file agent e fa riferimento ai vari file Java, posizionati nella stessa cartella, in base a quali goal e quali azioni il prey di turno deve portare a termine.

3.2.2 Primo passo: modifica dei prey esistenti

Come abbiamo detto, il primo step per raggiungere il nostro scopo è il salvataggio delle informazioni provenienti dai prey già esperti.

Per fare ciò dobbiamo andare a modificare il file Java EscapePreyPlan, visto che è quello che gestisce i movimenti dei prey quando devono scappare da eventuali hunter.

Per questa modifica noi siamo partiti da un condizione fondamentale: abbiamo diviso la visione del prey della griglia (che corrisponde ad un quadrato di 7x7 quadratini) in otto parti, dove quattro corrispondono ai quattro angoli mentre le rimanenti quattro ai quattro settori posti a sinistra, a destra, sopra e sotto la posizione del prey di riferimento, e per comodità abbiamo considerato la posizione dell'hunter possibile solo nel settore in alto oppure nell'angolo in alto a destra.

Inoltre, dato questo presupposto, consideriamo possibili i movimenti del prey per sfuggire all'hunter solo verso il basso e verso sinistra, se l'hunter proviene dall'angolo in alto a destra, e verso il basso, verso destra e verso sinistra per quanto riguarda il caso in cui l'hunter provenga dall'alto.

La reale posizione dell'hunter e i reali movimenti del prey saranno poi calcolate ruotando i nostri quadrati di riferimento secondo le reali posizioni dell'hunter.

Detto questo diamo un'occhiata al codice: la parte nuova più importante è sicuramente quella della figura 3.7 dove, se sono presenti hunter, in base all'azione eseguita ci sono dei riferimenti a diversi metodi.

```
if(hunters>0) {
FaceSceneWithHunter(obs,me,hu,(Prey)me);
if (SUtil.compareTo(sortpoints[0].toString(), "left=0")== 0) ICPatternsLeft(obs, me,vision,hu,(Prey) me);
else if(SUtil.compareTo(sortpoints[0].toString(),"right=0")== 0) ICPatternsRight(obs, me,vision,hu,(Prey)me);
else if (SUtil.compareTo(sortpoints[0].toString(), "up=0")== 0) ICPatternsUp(obs, me, vision,hu,(Prey)me);
else if (SUtil.compareTo(sortpoints[0].toString(), "down=0")== 0) ICPatternsDown(obs, me,vision,hu ,(Prey)me);
}
```

Figura 3.7 Chiamata di metodo per il salvataggio di osservazioni

In questi metodi, differenti tra di loro solo per l'azione eseguita, come si vede nelle figura 3.8 prima si imposta l'azione eseguita (new ExecutedActionDTO), si impostano gli attori in gioco e poi si setta la scena attuale con le azioni possibili e gli oggetti presenti. Quest'ultima può essere di due tipi: nel caso l'hunter sia in uno dei quattro angoli, le azioni possibili, come abbiamo già detto, sono due (Fig. 3.9), flee_down (muovi verso il basso) e flee_left (muovi a sinistra), mentre se l'hunter è posto in uno degli altri quattro settori le azioni possibili sono tre (Fig. 3.10), le stesse due del precedente,con in più flee_right (muovi a destra).

```
ObservationDTO observation = new ObservationDTO(identifier);
ExecutedActionDTO action = new ExecutedActionDTO(checkPosHunt(hu, pre, str ),new Timestamp(new
Date().getTime()));
ActorDTO actor = new ActorDTO("lazyprey");
action.addActor(actor);
observation.setExecutedAction(action);
s = posHunt(pre.getLocation().getX(), pre.getLocation().getY(), hu);
if(s == "s1" || s=="s2" || s=="s3" || s=="s4"){
observation.setScene(scene(ob, me));}
else { observation.setScene(scene2(ob,me));}
SicsComposer sicsComposer = SicsComposerGetter.getSicsComposer();
sicsComposer.sendObservation(observation);
```

Figura 3.8 Salvataggio delle osservazioni

```
ActionDTO d = new ActionDTO("flee_left");
sc.addAction(d);
ActionDTO e = new ActionDTO("flee_down");
sc.addAction(e);
```

Figura 3.9 Scene(ob,me)

```
ActionDTO d = new ActionDTO("flee_left");
sc.addAction(d);
ActionDTO f = new ActionDTO("flee_right");
sc.addAction(f);
ActionDTO e = new ActionDTO("flee_down");
sc.addAction(e);
```

Figura 3.10 Scene2(ob,me);

Infine, nella parte finale della figura 3.8 abbiamo il vero e proprio salvataggio delle osservazioni in sicsComposer.sendObservation(observation).

Ora abbiamo salvato le informazioni che ci servono ma dove possiamo andare a vederle? Esattamente nel file *sics_store.xml* dove vengono salvate una dopo l'altra e potranno poi essere riprese da altri prey (vedasi un esempio nella figura 3.11).

```
<observation identifier="testConfig">
<action name="flee_down" timestamp="1181231348708">
<actors>
<actor name="lazyprey" />
</actors>
</action>
<scene>
<actions>
<action name="flee_down" />
<action name="flee_left" />
<action name="flee_right" />
</actions>
<objects>
<object name="Hunters">
<attributes>
<attribute name="number" type="Integer">1</attribute>
```

```
</attributes>
</object>
<object name="Obstacles">
<attributes>
<attribute name="number" type="Integer">2</attribute>
</attributes>
</object>
<object name="Preys">
<attributes>
<attribute name="number" type="Integer">1</attribute>
</attributes>
</object>
</objects>
</scene>
</observation>
<observation identifier="testConfig">
```

Figura 3.11 Osservazioni salvate in sics_store.xml

3.2.3 Secondo passo: impostazione della teoria e configurazione della similitudine

Il secondo passo verso la totale e completa applicazione della Cultura Implicita è la definizione della teoria usata dall'Ic-Service e la configurazione delle similitudini per far si che SICS ridia indietro le esatte osservazioni che noi gli chiediamo.
Ma partiamo dalla teoria. Come detto nel precedente capitolo la *teoria* rappresenta per noi la cosiddetta “cultura comunitaria” ed è estratta dalle osservazioni salvate dal set di agenti più esperti e verrà poi usata per produrre suggerimenti per il set di agenti neo-arrivati, cioè nel nostro caso sicsprey. La teoria quindi deve essere impostata da noi affinché si adatti alle nostre esigenze. Ciò si può fare nel file *sics_*rules.xml dove per l'appunto andranno stabilite le regole.
Queste regole singolarmente avranno la struttura seguente: dovranno seguire la legge “*se face_scene_with_hunter* allora *azione*” dove *face_scene_with_hunter* sta per la scena

delineatasi attualmente sulla griglia, comprensiva di oggetti e azioni possibili, cioè quella compresa negli *antecedents*, mentre *azione* rappresenta l'azione da compiere nello step successivo e può essere scelta tra le seguenti possibilità: *flee_down* o *flee_left* se hunter in uno degli angoli, *flee_down*, *flee_lef*t o *flee_right* se hunter in uno degli altri settori; essa sarà compresa tra i *consequents (*Esempio figura 3.12).


```
<rule identifier="testConfig">
<antecedents>
<action-predicate>
<action-rule name="face_scene_with_hunter" name_type="constant" timestamp="*"timestamp_type="variable">
<actors>
<actor-rule name="_X" name_type="variable"/>
</actors>
</action-rule>
<scene-rule>
<actions>
<action-rule name="face_scene_with_hunter" name_type="constant" />
<action-rule name="flee_down" name_type="constant" />
<action-rule name="flee_left" name_type="constant" />
<action-rule name="flee_right" name_type="costant" />
</actions>
<objects>
<object-rule name="Hunters" name_type="constant">
<attributes>
<attribute-rule name="number" type="Integer" variable_type="variable">_nH</attribute-rule>
</attributes>
</object-rule>
</objects>
</scene-rule>
</action-predicate>
</antecedents>
<consequents>
<action-predicate>
<action-rule name="flee_down" name_type="constant" timestamp="*" timestamp_type="variable">
<actors>
<actor-rule name="_X" name_type="variable"/>
</actors>
</action-rule>
<scene-rule>
<actions>
<action-rule name="flee_down" name_type="constant" />
<action-rule name="flee_left" name_type="constant" />
<action-rule name="flee_right" name_type="costant" />
</actions>
<objects>
<object-rule name="Hunters" name_type="constant">
```

```
<attributes>
<attribute-rule name="number" type="Integer" variable_type="variable">_nH</attribute-rule>
</attributes>
</object-rule>
</objects>
</scene-rule>
</action-predicate>
</consequents>
</rule>
```

Figura 3.12 Definizione di una regola della teoria

Dopo la teoria come già detto c'è la necessità di configurare le similarità.
Questa configurazione è necessaria in quanto permette a SICS di capire quando un scena collegata ad una certa azione a cui sicsprey si trova davanti è simile o uguale ad una che è stata salvata da un altro prey.
E ciò lo comprende quando, tramite i nostri settaggi (figura 3.13), la somma di tutti i valori dei componenti facenti parte dell'azione e della scena supera un dato valore, nel nostro caso 0.9, settabile nel file sicsInstanceConfiguration.xml.

```
<similarity-configuration identifier="testConfig">
<defaults>
<default-actions> valueForName="0.90" valueForAttributes="0.0" valueForObjects="0.0" valueForActors="0.10"
valueForTimestamp="0.00"
caseInsensitiveMatching="true" useOnlyRegularExpression="false" processRegularExpression="false"
processRegularExpressionAsOr="true">
<regExp-ForName>^*</regExp-ForName>
</default-actions>
<default-objects valueForName="0.50" valueForAttributes="0.50"
caseInsensitiveMatching="true" useOnlyRegularExpression="false" processRegularExpression="false"
processRegularExpressionAsOr="true"><regExp-ForName>^*</regExp-ForName>
</default-objects>
<default-actors valueForName="1.0" valueForAttributes="0.0"
caseInsensitiveMatching="true" useOnlyRegularExpression="false" processRegularExpression="false"
processRegularExpressionAsOr="true">
<regExp-ForName>^*</regExp-ForName>
</default-actors>
<default-scenes valueForActions="0.60" valueForObjects="0.40" />
<default-observations valueForExecutedAction="0.50" valueForScene="0.50" />
</defaults>
</similarity-configuration>
```

Figura 3.13 File di configurazione delle similarità similarity_config.xm

3.2.4 Terzo passo: creazione di sicsprey e applicazione della “cultura di gruppo”

Dopo aver settato le impostazioni indispensabili per il buon funzionamento dell'Ic-Service e dopo aver modificato i prey in modo che ci siano utili per il salvataggio delle informazioni, ora non ci resta altro che creare il prey, che, sfruttando queste osservazioni, si comporti in maniera appropriata all'interno dell'ambiente, cioè nel nostro caso scappando dagli hunter.
Per prima cosa creiamo quindi il prey vero e proprio. Ricordandoci dei due tipi di strutture possibili e dopo aver fatto un tentativo infruttuoso con l'altra, decidiamo di crearlo simile al dumbprey, cioè composto dal file agent vero e proprio che chiameremo SicsPrey.agent.xml e sarà simile nei contenuti a quello del DumbPrey, e da un file Java collegato contenente il plan, che infatti verrà chiamato SicsPreyPlan.
Detto questo passiamo ad analizzare quest'ultimo file che è il principale responsabile dei movimenti “suggeriti” del prey.
Come primo passo bisogna settare l'azione predefinita, cioè face_scene_with_hunter, e la scena che si presenta in base alla posizione dell'hunter, cioè come già detto se esso è in uno dei quattro angoli oppure in uno degli altri quattro settori. Questi due parametri ci serviranno per richiedere le osservazioni tramite la relazione già incontrata “*se face_scene_with_hunter* in una particolare scena allora *azione*”.
Poi si passa alla richiesta vera e propria tramite il codice che si può osservare nella figura 3.14.

```
ObservationDTO ob = new ObservationDTO(identifier);
SicsComposer composer = SicsComposerGetter.getSicsComposer();
ResultDTO res = new ResultDTO();
res = composer.performQuery(ob);
```

Figura 3.14: Codice per la richiesta di suggerimenti

Ora nella variabile di classe ResultDTO abbiamo un serie di scene suggerite dall'Ic-Service, quindi adesso dobbiamo sceglierne una che ci permetta di fuggire dall'hunter. Questo sarà fatto a random.
Dopo questa scelta finalmente abbiamo la scena richiesta. Ora bisognerà optare per una delle azioni possibili ed anche questo può essere fatto a random.

Ecco ora siamo arrivati alla fine: dopo aver scelto l'azione da compiere la si esegue affinché il prey si salvi dall'hunter (Figura 3.15). Nel caso L'Ic-Service non trovi alcuna soluzione compatibile col caso richiesto il prey eseguirà un'azione a random.
Dopo che il prey si sarà mosso l'intera operazione verrà ripetuta nuovamente e così via finché o il sicsprey riesce a scappare dall'hunter oppure lo stesso hunter riesce a raggiungere il prey e a mangiarlo: allora un nuovo sicsprey verrà messo sulla griglia,

```
IGoal move = createGoal("move");
move.getParameter("direction").setValue(str);
dispatchSubgoalAndWait(move);
```

Figura 3.15 Codice che permette il movimento al prey

A questo punto siamo arrivati alla fine del nostro lavoro: infatti il sicsprey adesso sarà in grado di fuggire con i propri mezzi dall'hunter così come fanno gli altri prey già implementati dai progettisti di Jadex. E ci riuscirà senza una conoscenza precedente dell'ambiente e delle azioni da compiere.

3.3 Conclusioni

In questo capitolo abbiamo inizialmente fatto una breve introduzione della struttura di Jadex, programma per la creazione di programmi ad agenti. In particolare ci siamo focalizzati sull'uso del modello BDI all'interno di esso. Poi siamo passati ad analizzare la sua struttura vera e propria basata sul precedente modello dei "Belief, Desiree, Intention". Nel prosieguo del capitolo, dopo aver fatto una breve descrizione dell'esempio di Jadex Hunter Prey, siamo passati alla modifica dell'esempio stesso. Questa è la parte centrale della nostra tesi dove, attraverso l'uso di SICS, abbiamo reso un nuovo prey, sicsprey, abile a comportarsi come gli altri suoi simili nell'ambiente e cioè a scappare dall'hunter, senza che esso conoscesse precedentemente l'ambiente e senza che esso venisse a diretto contatto con gli altri prey.

Capitolo 4
Risultati sperimentali e problemi riscontrati

In questo capitolo andremo a presentare i risultati dei test effettuati sul prey creato da noi e che segue la cultura di gruppo.

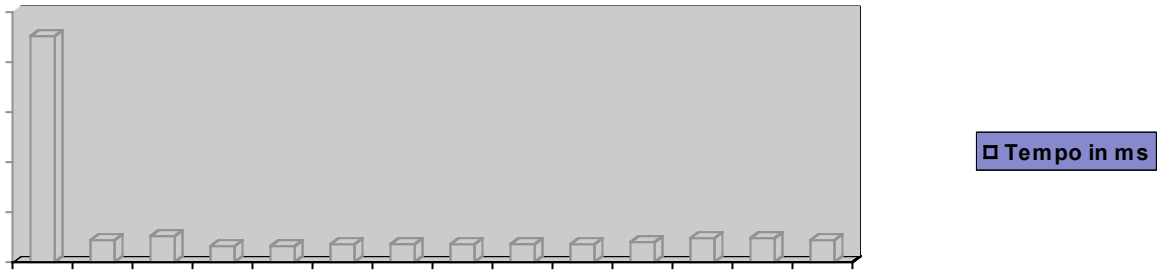
4.1 Risultati ottenuti

Eseguendo dei test sul comportamento del sicsprey da noi creato abbiamo visto che esso si comporta esattamente come i prey già presenti nell'esempio, ad esempio dumbprey e lazyprey, tranne che in alcune occasioni: quando si verifica il cosiddetto problema del tempo di risposta del sistema.

4.1.1 Il problema del tempo di risposta

Il tempo di risposta del sistema è il tempo che impiega SICS a soddisfare una richiesta. Esso infatti deve ricevere una richiesta, analizzarla, valutare la scena in cui si svolge, controllare le informazioni salvate, trovare quelle che soddisfano la richiesta e suggerire le scene possibili al prey. Questo può impiegare un periodo di tempo anche relativamente lungo, nel caso in cui le osservazioni salvate siano un gran numero e nel caso in cui ci siano altri fattori che rallentano le operazioni.
Nel nostro caso abbiamo che il prey e l'hunter si muovano di un quadratino al secondo, per cui se il sistema impiega più di un secondo a suggerire le indicazioni per un'azione il prey non riuscirà a muoversi in tempo e verrà così raggiunto e mangiato dall'hunter.
Questo problema nel nostro ambiente si verifica in particolar modo la prima volta che

appare un hunter nel campo visivo del prey (come si vede nel grafico 4.1) che rischia così di essere mangiato immediatamente. Se questo non accade, nelle scene successive il sistema risponde in un tempo nettamente inferiore al secondo permettendo al prey di fuggire dall'hunter.



Tab. 4.1 Grafico del tempo di risposta del sistema

Questo problema inoltre non si verifica solamente la prima volta ma, anche se molto più raramente, può presentarsi pure durante l'inseguimento rendendo così la fuga impossibile.

4.2 Conclusioni

In questo quarto capitolo siamo passati ad analizzare i test eseguiti sul nostro nuovo prey ed in particolare abbiamo notato come il prey abbia a volte un problema dovuto al sistema SICS. Esso è il cosiddetto problema del tempo di risposta del sistema che in qualche caso, per colpa della lentezza del sistema, fa sì che il sicsprey non riesca a fuggire dall'hunter bensì venga raggiunto e mangiato dal cacciatore.

Conclusioni

L'obiettivo di questo lavoro è stato quello di verificare la possibilità di applicare la Cultura Implicita tramite l'Ic-Service ed il Sistema per il Supporto alla Cultura Implicita (SICS) ad un esempio qualsiasi all'interno di Jadex. Noi abbiamo scelto quello di Hunter Prey perché era quello che più si adattava allo scopo. Inoltre attraverso un testing abbiamo verificato il comportamento di SICS e gli eventuali problemi che esso ci da'.
In un prima fase abbiamo fatto una panoramica sullo stato dell'arte dei sistemi di Raccomandazione attraverso le loro caratteristiche principali e la struttura generale. Lo scopo di questa parte è quello di acquisire delle informazioni sullo stato dell'arte delle soluzioni al problema della mancanza di conoscenza nei sistemi distribuiti.
Successivamente si è passati all'analisi approfondita del concetto di Cultura Implicita attraverso un esempio e allo studio della struttura e delle funzioni di SICS. Inoltre abbiamo visto i diversi tipi di scenari in cui esso può essere utilizzato.
Nel terzo capitolo dopo una prima breve presentazione della struttura e delle caratteristiche del software Jadex siamo passati alla parte principale del nostro lavoro, cioè la modifica dell'esempio Hunter Prey di Jadex in modo che sia creato un nuovo prey che si comporti esattamente come gli altri già presenti nell'ambiente. Per far ciò abbiamo applicato la Cultura Implicita tramite Ic-Service e SICS al prey stesso.
Nel quarto e ultimo capitolo attraverso delle prove di test abbiamo scoperto un problema nell'applicazione del SICS: il problema del tempo di risposta del sistema. Il tempo utilizzato dal sistema per suggerire un scena, e quindi un'azione possibile, può essere troppo elevato perché il prey scappi, facendo sì che l'hunter lo possa raggiungere e così mangiare.
Sviluppi futuri potranno riguardare ulteriori e più approfonditi test sull'applicazione di SICS in Jadex e la valutazione di ulteriori applicazioni di esso in altri software basati sugli agenti.

Bibliografia

[1] M. Elena Renda, Umberto Straccia “A personalized collaborative Digital Library environment: a model and an application“ 2004 Elsevier Ltd disponibile su gaia.isti.cnr.it/~**straccia**/download/papers/IP**M**05/IP**M**05.pdf

[2] M. von Setten “Supporting People In Finding Information”, Telematica Instituut Fundamental Research Series, No. 016 (TI/FRS/016), Enschede, The Netherlands, 2005

[3] Burke, R. (2002). Hybrid Recommender Systems: Survey and Experiments. *User Modeling and User-Adapted Interaction, 12,* 331-370.

[4] Resnick, P., Iacovou, N., Suchak, M., Bergstrom, P., and Riedl, J. GroupLens: An open architecture for collaborative filtering of netnews. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (Chapel Hill, NC) 1994.

[5] Hill, W., Stead, L., Rosenstein, M., and Furnas, G. Recommending and evaluating choices in a virtual community of use. In *Conference on Human Factors in Computing Systems—CHI ’95.* (Denver, May 1995).

[6] Shardanand, U., and Maes, P. Social information filtering: Algorithms for automating “word of mouth.” In *Conference on Human Factors in Computing Systems—CHI ’95.* (Denver, May 1995).

[7]Schafer, J.B., Konstan, J.A., & Riedl, J. (2001). E-Commerce Recommendation Applications. *Journal of Data Mining & Knowledge Discovery,* 115-153.

[8] Maltz, D. & Ehrlich, E. (1995).Pointing the way: Active collaborative filtering. In *Proceedings of the CHI’95 Human Factors in Computing Systems* (pp. 202-209). NewYork: ACM.

[9] Balabanovic, M & Shoam, Y “Fab: Content-based, Collaborative Reccomendation”, COMMUNICATIONS OF THE ACM March 1997/Vol. 40, No. 3

[10] Ardissono, et al. (2004). User Modeling and Recommendation Techniques for Personalized Electronic Program Guides. In L. Ardissono, A. Kobsa & M.T. Maybury (Eds.), *Personalized Digital Television: Targeting Programs to Individual Viewers*. Dordrecht, the Netherlands: Kluwer Academic Publishers.

[11] Recla, S. “Progetto e sviluppo di un Sistema di Supporto all'interazione multiagente basato su Cultura Implicita” tesi di laurea a.a. 1999 – 2000

[12] Blanzieri E. Giorgini P. Massa P. and Recla S. Collaborative ltering via implicit culture support. In Proceedings of the 8th International COnference on User Modeling (UM2001), Sonthofen, Germany, 2001.

[13] Blanzieri E. Giorgini P. Massa P. and Recla S. Implicit culture for multi-agent system interaction support. In Proceedings of the International Conference on Cooperative Information System, (CoopIS2001), Trento,Italy, 2001.

[14] Blanzieri E. Giorgini P. Massa P. and Recla S. Information access in implicit culture framework. In Proceedings of the Tenth ACM International Conference on Information and Knowledge Management, (CIKM2001), Atlanta, Georgia, 2001.

[15] Birukou, A Blanzieri, E Giorgini, P et a. “IC-Service: A Service-Oriented Approach to the Development of Recommendation Systems”, SAC’07 March 11-15, 2007, Seoul Korea

[16] Blanzieri, E Giorgini, P “Implicit Culture for Information Agent”, ITC-Irst Trento

Ringraziamenti

Ed eccoci arrivati alla fine di questa laurea triennale dura ma anche carica di soddisfazioni e lati positivi. Qui sono dovuti dei ringraziamenti alle persone che hanno fatto sì che abbia potuto arrivare alla fine di questa prima avventura. Come prima cosa vorrei ringraziare il Professor Giorgini Paolo, mio relatore, che con i suoi suggerimenti mi ha aiutato a metter il mio lavoro nero su bianco. Poi vorrei ringraziare particolarmente Aliaksandr Birukou che mi ha seguito dall'inizio alla fine nel lavoro svolto, aiutandomi nei vari momenti di difficoltà e fugando sempre ogni mio dubbio. Poi ovviamente vorrei ringraziare con grande affetto tutta la mia famiglia, mamma Daniela, papà Sergio e Camilla, che mi è stata vicina in questi quattro anni e mi ha permesso di arrivare fin qua. Poi un ringraziamento speciale va a Uta che, anche se da lontano, mi ha sempre supportato nel mio lavoro. Vorrei poi ringraziare tutti gli amici di università e di tempo libero che hanno reso molto meno noiosi questi quattro anni passati assieme. Infine vorrei ringraziare le mie due nonne, Elda e Silvia, sempre pronte a rifocillarmi coi loro deliziosi manicaretti e ad aiutarmi coi loro preziosi consigli.