ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea Specialistica in Ingegneria Informatica

SIMPA E IBM AGLETS: PIATTAFORME AD AGENTI A CONFRONTO

Elaborata nel corso di: Sistemi Multiagente LS

Tesina di:
SIMONE VIGNODELLI

Professore:
Prof. ANDREA OMICINI

ANNO ACCADEMICO 2010–2011
SESSIONE I

PAROLE CHIAVE

Agente

Artefatto

Ambiente

Autonomia

Indice

Introduzione

Il paradigma ad agenti è un modello sviluppatosi intorno alla metà degli anni novanta volto alla risoluzione di problemi complessi. La sua adozione in ambito industriale è stata ostacolata da molti fattori tra cui il diffondersi del paradigma object-oriented, la mancanza di un'unico filone di pensiero che accomuni i principi enunciati in tale modello e soprattutto dalla completa assenza di efficaci piattaforme operative in grado di produrre sistemi funzionanti agent-based.

Il principale scopo di questa tesina è quello di effettuare un confronto fra due delle soluzioni proposte agli sviluppatori per produrre sistemi ad agenti: la prima, *Simpa*, è in continua evoluzione e supportata dal team di ricerca universitario aliCE, mentre la seconda, *IBM Aglets*, nasce verso la fine dello scorso millennio negli IBM Labs e al momento sta attraversando un periodo di stasi.

Per una migliore chiarezza e comprensione dell'argomento, si è scelto di articolare la tesina che segue in quattro capitoli:

- nel primo capitolo viene esposto in maniera sommaria quali sono state le fasi evolutive dei paradigmi di programmazione e le motivazioni che stanno alla loro base;
- il secondo capitolo si focalizza sul metamodello *Agent & Artifact* sul quale si basa simpA;
- al terzo capitolo invece si va nel dettaglio di Simpa, introducendo le astrazioni e le modalità di sviluppo di questa piattaforma;
- nel quarto capitolo l'oggetto di studio è IBM Aglets che sarà osservato da due punti di vista: il meta-modello da cui prende ispirazione e lo stile di programmazione che introduce;

- l'ultimo capitolo è riservato alle conclusioni e alle deduzioni che è possibile evincere da questa tesina.

Capitolo 1

Evoluzione dei paradigmi di programmazione

In questo capitolo verranno passati in rassegna in ordine temporale i vari tipi di paradigmi che hanno ispirato i più famosi linguaggi di programmazione fino ad arrivare all'introduzione del concetto di "agente" e alle motivazioni che hanno portato alla sua creazione.

1.1 Programmazione monolitica

Tipo di programmazione in cui l'unità elementare di lavoro è il programma stesso. Lo sviluppatore ha il pieno controllo sul codice scritto gestendo in maniera autonoma lo stato del programma: una volta terminata l'applicazione, questa di fatto si comporta come una black-box per l'utilizzatore, che la invoca in maniera opportuna dall'esterno. La riusabilità in questo ambito è nulla: il programma esegue tutto o niente.

1.2 Programmazione modulare

Attraverso l'evoluzione tecnologica, l'elettronica mette a disposizione computer con processori veloci e memoria più ampia permettendo così la creazione di programmi grandi e complessi. Nasce quindi un nuovo livello di difficoltà che la mente umana deve gestire: vengono ideate

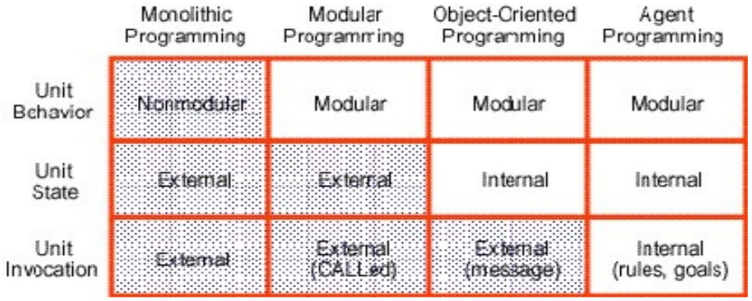

Figura 1.1: Visione evolutiva dei paradigmi di Odell

le cosidette *subroutine*. Esse consistono in piccole porzioni di codice che svolgono una determinata funzione e che vengono richiamate ogni volta che se ne ha l'esigenza all'interno un programma. Quindi è possibile affermare che il *comportamento* svolto da una procedura è incapsulato al suo interno mentre lo stato del programma risiede ancora all'esterno, demandato a parametri di input che l'utente passa al programma prima di invocarlo.

1.3 Programmazione object-oriented

Nasce il concetto di *oggetto* o *classe*. Esso racchiude al suo interno lo *stato*, composto da una o più variabili: queste vengono modificate attraverso l'invocazione di metodi interni all'oggetto che, svolgendo una determinata funzione, aggiornano lo stato che rimane persistente nel tempo. In termini di riusabilità si ha un notevole miglioramento rispetto alla programmazione modulare ma ha ancora un difetto: non viene integrata l'invocazione dell'oggetto che rimane esterna ad esso. Il controllo del programma non è ancora incapsulato all'interno dell'astrazione.

1.4 Programmazione agent-oriented

Come naturale evoluzione dei paradigmi precedenti, l'*agente* è la nuova entità base con la quale costruire applicazioni software. Sono riusabili,

incapsulano uno stato controllandolo in maniera autonoma, sono entità attive che non necessitano di un'invocazione esterna e che hanno pieno potere decisionale sul loro comportamento.

1.4.1 Da dove nasce il concetto di agente

La necessità di trovare un nuovo modello di programmazione è dettata dal fatto che il paradigma object-oriented non è più in grado di modellare tutte le caratteristiche richieste nelle moderne applicazioni. In particolare i sistemi artificiali complessi sono caratterizzati da quattro proprietà:

- **Situatedness** - questa caratteristica mette in luce la stretta relazione che esiste fra la *computazione del sistema* e l'*ambiente* in cui esso si trova: questi due elementi si influenzano l'un l'altro. Durante la sua esecuzione, la computazione deve essere in grado di percepire stimoli provenienti dall'ambiente e monitorare eventuali cambiamenti esterni che possono modificare il suo comportamento. La proattività è una caratteristica fondamentale di un agente che deve essere garantita in tutti i possibili stati che la computazione può assumere. Anche il concetto di ambiente deve essere modellato e ben definito in modo da creare una sorta di confine tra l'impredicibilità del mondo esterno e le proprietà che si vuole siano garantite all'interno dell'ambiente di lavoro dell'agente.

- **Openness** - i sistemi moderni richiedono di essere attivi 24 ore al giorno e sono soggetti a cambiamenti, ad una evoluzione della propria struttura e ad una variazione del numero di componenti interni. Nella proprietà di situatedness è stato ribadito come sia importante definire i confine dell'ambiente di lavoro del sistema; nella proprietà di openness si aggiunge un'ulteriore caratteristica: i confini stabiliti devono essere permeabili, in ingresso e in uscita, al transito di agenti che per i più svariati motivi hanno necessità di muoversi da un ambiente all'altro. Sarà premura di chi ha modellato l'ambiente garantire la sua validità in ogni singolo instante regolando l'ingresso dei componenti che possono avere accesso o meno all'ambiente.

- **Local Control** - ogni componente di un sistema tende ad essere attivo con un proprio flusso di controllo. Da un punto di vista esterno però è importante osservare anche l'*autonomia* di un componente perchè questo permette una migliore gestione di particolari situazioni, quali il disaster management e il disaster recovery, dove è comodo delegare responsabilità ai singoli elementi quando il sistema si trova in affanno. Dall'altro lato, l'autonomia porta ad una peggiore osservabilità del sistema nel suo complesso in quanto l'ispezionabilità di ogni singolo elemento risulta difficoltosa.

- **Local Interactions** - è inutile progettare un componente che sia in grado di comunicare con il mondo intero. Esistono vincoli fisici (sensori che ispezionano solamente un intorno dell'ambiente) e logici (sensori fra loro vicini sono organizzati in cluster perchè più conveniente) che sanciscono una rilevanza maggiore delle interazioni più su base locale che su scala globale (*context awareness*). Per cui è necessario definire il contesto in cui opera un componente tenendo sempre presente che questo non deve essere chiuso su se stesso ma anche aperto a possibili interazioni esterne.

1.4.2 Caratteristiche di un agente

La definizione di agente deriva da un'evoluzione di pensieri e di contributi dalle più disparate aree di ricerca quali la robotica, l'intelligenza artificiale, la software engineering... Questo da l'idea di come sia difficile dare una definizione di tale concetto in modo completo ed esaustivo in poche parole. Di seguito si cercherà di delineare che cos'è un agente, specificando le caratteristiche fondamentali che sanciscono la sua natura:

- **riusabilità** - un agente deve poter essere utilizzato in più contesti e in ambienti diversi;

- **autonomia** - termine che delinea tre caratteristiche fondamentali: la capacità di controllare il proprio stato, la capacità di variare o meno il proprio comportamento in relazione a stimoli

esterni, la possibilità di rispondere o meno ad impulsi esterni a propria discrezione;

- **disaccoppiamento dati e controllo** - nel paradigma object-oriented dati e controllo sono una cosa sola mentre nel paradigma agent-oriented il controllo deve essere confinato all'interno dell'agente. Questo permette una progettazione dei sistemi più snella plasmata intorno al solo flusso informativo;
- **comunicazione a conversazione** - mirata allo scambio di informazione, non essendoci scambio di controllo, quella che si instaura è una vera propria conversazione composta da semplici atti comunicativi volti ad assolvere una semantica globale;
- **decentralizzazione e dinamicità di ruolo** - mentre un oggetto detiene un controllo centralizzato e un ruolo definito dalla classe di appartenenza, l'agento è in grado di interpretare più ruoli e adempiere a più goal in base a propri criteri decisionali interni. Due istanze di agente quindi possono essere completamente diverse a differenza di due istanze di oggetto, che possono differire unicamente da un punto di vista di stato interno;
- **small in impact** - la caratteristica di autonomia rende un sistema composto da agenti meno vulnerabile al fallimento di una singola istanza perchè il controllo di fatto è decentralizzato e il task dell'agente deceduto può essere affidato ad un altro agente in grado di cambiare il proprio ruolo. In un sistema ad oggetti, il fallimento di una singola istanza, nella maggior parte dei casi, comporta la caduta dell'intera applicazione in quanto il controllo è centralizzato;
- **small in time** - una gestione della memoria più snella perchè strumenti come il garbage collector non sarebbero più di alcuna utilità in quanto intrinsecamente al concetto di agente è già mappato il fatto che l'ambiente, alla morte di un agente, dimentichi ogni riferimento all'instanza scomparsa;
- **small in scope** - le interazioni degli agenti sono per la maggior parte locali basate su una base di conoscenza costruita a partire da osservazioni eseguite sull'ambiente circostante;

- **impredicibilità** - il comportamento di un agente segue una propria logica interna che da un punto di vista esterno non può essere predetta.

Capitolo 2

Metamodello Agent and Artifact

Di seguito verrà illustrato il pensiero teorico che sta alla base di questo modello, specificando i suoi elementi costitutivi e le loro caratteristiche chiave.

2.1 Struttura

Il metamodello Agent and Artifact (A&A) è formato dai seguenti elementi:

1. **Agente**
2. **Artefatto**
3. **Workspace**

2.2 Agente

Il concetto di *agente* è stato ampiamente trattato nel corso del primo capitolo. In questo contesto quindi si è scelto di rimarcare brevemente i due aspetti principali già visitati che lo contraddistinguono:

- **autonomia**: un agente ha il completo controllo del suo comportamento che è teso al raggiungimento di un determinato obiettivo;
- **interazione con l'ambiente**: un agente non viene invocato perchè non possiede un'interfaccia d'uso. Per interagire con l'ambiente utilizza un insieme di sensori ed attuatori che permettono di osservare ed eventualmente modificare l'ambiente circostante.

2.3 Artefatto

2.3.1 Caratteristiche e proprietà

L'artefatto è definito come un'*entità computazionale costruita per essere utilizzata da un agente*. Da questa concezione è possibile dedurre un insieme di caratteristiche derivate che non contribuiscono alla definizione originale perchè già ivi incapsulate:

- **function oriented**: incapsulano una funzionalità eseguita integrando i meccanismi di basso di livello volti alla gestione della concorrenza. Gli agenti devono poter usare l'artefatto in piena libertà;
- **trasparenza**: dall'esterno un agente deve poter capire quale funzionalità un artefatto dichiara di svolgere;
- **dotato di interfaccia**: le operazioni che è possibile richiedere all'artefatto sono esposte in maniera esplicita all'agente;
- **predicibilità**: il comportamento di un artefatto deve essere stimabile, cioè deve essere possibile capire quali saranno gli effetti di una interazione;
- **non è autonomo**: l'artefatto non è in grado di governarsi da solo;
- **reattività**: la sua funzionalità è svolta come sola conseguenza di un utilizzo da parte di un agente;

- **situato**: l'utilizzo di un artefatto è l'obiettivo primario delle azioni di un agente; queste sono fortemente correlate con l'ambiente in cui l'agente opera e di conseguenza anche gli artefatti dovranno essere situati nell'ambiente in cui si pongono.

Quando l'agente richiede lo svolgimento di un'operazione da parte di un artefatto, quest'ultimo sarà soggetto ad un aggiornamento del proprio stato e genererà uno o più eventi nell'ambiente circostante visibili a tutti gli agenti in ascolto. Le proprietà indispensabili che deve avere un artefatto sono:

- **ispezionabilità**: quando un nuovo agente entra in maniera dinamica nell'ambiente, è necessario che venga a conoscenza degli artefatti che ha a disposizione: stato, operazioni, interfaccia e funzione;
- **controllabilità**: capacità di monitorare il funzionamento dell'artefatto, con la possibilità di fermarlo e farlo ripartire, controllando eventualmente step by step l'esecuzione;
- **malleabilità**: deve essere possibile la modifica a run-time, senza introduzione di inconsistenze, del comportamento dell'artefatto, al fine di adattarsi ad un cambiamento dell'ambiente;
- **predicibilità**: la funzionalità esposta dall'artefatto deve essere sempre la stessa al fine di riuscire a capire quali saranno gli effetti generati dal suo comportamento;
- **formalizzabilità**: importante è la possibilità di verificare automaticamente le proprietà ed il comportamento dei servizi resi disponibili dagli artefatti;
- **linkabilità**: utile per fini di riusabilità, deve essere possibile collegare fra loro più artefatti per assolvere ad una funzionalità complessa richiesta. Anche la scomposizione di un artefatto in tanti sotto-artefatti è un aspetto da non trascurare;
- **distribuibilità**: correlata alla linkabilità, questa proprietà torna utile nell'ambito distribuito quando più artefatti risiedono su più nodi della rete. Deve essere possibile la ricerca e la composizione al fine di creare operazioni complesse.

2.3.2 Modalità di utilizzo e di interazione

Dalle conoscenze che derivano dal campo di ricerca dell'Activity Theory, è possibile definire tre livelli di utilizzo degli artefatti:

1. **Co-ordination**: gli agenti usano gli artefatti per raggiungere il proprio goal, semplicemente prendendoli così come sono;
2. **Co-operation**: un agente intelligente, in base alla sua visione, può decidere di migliorare l'ambiente in cui vive non modificando gli altri agenti ma agendo sugli artefatti, modificandoli o creandone di nuovi.
3. **Co-construction**: si inserisce all'interno di un artefatto un'intelligenza in grado di monitorare ed agire sull'ambiente in modo da cambiare il comportamento sociale degli agenti.

Dal punto di vista degli agenti, un artefatto può essere visto e sfruttato in tre modalità:

1. **uso**: uso consapevole o inconsapevole a seconda dell'intelligenza degli agenti;
2. **selezione**: si identifica un artefatto che verrà usato in futuro;
3. **costruzione e manipolazione**: si modifica l'artefatto oppure ne si creano di nuovi.

2.4 Workspace

Rappresentano dei contenitori logici volti a strutturare l'intero ambiente all'interno del quale gli agenti operano; idealmente a workspace diversi corrispondono tipologie di operazioni diverse. Dentro un nodo è possibile avere più workspaces ma, a differenza dell'ambiente, non è possibile avere un workspace distribuito sulla rete. Un agente può dinamicamente inserirsi all'interno di un workspace e simultaneamente operare su più workspace diversi.

Capitolo 3

simpA

simpA è un framework per lo sviluppo di sistemi software ispirato al metamodello A&A; è basato su Java, sfruttando le Java Annotation, ma è concettualmente indipendente dal paradigma OO. L'elemento di novità che rende interessante questa nuova piattaforma risiede nel livello di astrazione introdotto pensato per progettare e sviluppare applicazioni fortemente concorrenti senza preoccuparsi troppo dei meccanismi di sincronismo a basso livello di cui, in questo ambiente, occorre tener conto e saper gestire. Di fatto, questo nuovo modo di organizzare le idee e i concetti imposto dal framework stesso mira a colmare il gap esistente fra la fase di progettazione e quella di implementazione. Nelle sezioni che seguono si cercherà di dare un'idea generale di come le astrazioni base dell'A&A siano state realizzate all'interno di simpA cercando di chiarire anche con qualche esempio esemplificativo.

3.1 Agente

Questa astrazione, a livello implementativo, può essere vista come un interprete di un programma contenente la descrizione delle attività che l'agente deve svolgere per raggiungere il suo obiettivo. La definizione di un agente avviene attraverso l'estensione della classe *alice.simpa.Agent*. Le possibili attività ivi dichiarate possono essere di due tipi:

- **atomiche**: sono dei singoli blocchi di istruzioni (azioni) che generalmente sono volte all'interazione con l'ambiente circostante. Sono indicate con l'annotazione *@ACTIVITY*;

- **strutturate**: sono la composizione di sotto-attività. Sono indicate con l'annotazione *@ACTIVITY_WITH_AGENDA*.

Di base un agente deve avere un'attività chiamata *main* che può essere sia atomica che composta. Ecco alcuni esempi:

```
public class HelloAgent extends Agent {
   @ACTIVITY void main() {
      ArtifactId cons = lookupArtifact("console");
      use(cons, new Op("println", "Hello, world!"));
   }
}
```

In questo primo caso si può osservare un agente con un'attività main atomica. Da notare sono i metodi *lookupArtifact*, che fornisce il riferimento ad un artefatto richiesto, e *use* che permette di sfruttare un artefatto di cui si ha il riferimento.

```
public class MyAgent extends Agent {
   @ACTIVITY_WITH_AGENDA({
      @TODO(activity="a"),
      @TODO(activity="b", pre="completed(a)"),
      @TODO(activity="c", pre="completed(a)"),
      @TODO(activity="d", pre="completed(b),completed(c)")
   }) void main() {}

   @ACTIVITY void a() {
      log("completed a");
   }
   @ACTIVITY void b() {
      waitFor(2500);
      log("completed b");
   }
   @ACTIVITY void c() {
```

```
      waitFor(1500);
      log("completed c");
   }
   @ACTIVITY void d() {
      log("completed d");
   }
}
```

Il secondo esempio rappresenta un'attività main strutturata dove, attraverso l'annotazione *@TODO*, sono specificate le azioni che la compongono. All'avvio, tutte le azioni sono eseguite in maniera concorrente, a meno che non siano indicate delle precondizioni, che abilitano l'esecuzione dell'attività a cui sono riferite, solo al momento del loro soddisfacimento.
E' importante fare mente locale anche sulla memoria con la quale gli agenti stessi interagiscono. Ce ne sono due:

- una locale alla singola attività rappresentata da tutte quelle variabili locali interne usate per dare forma alle azioni da svolgere;

- una globale all'agente, chiamata *memo space*, che memorizza le informazioni sotto forma di *memo* che l'agente stesso può creare, recuperare, modificare e rimuovere in maniera atomica attraverso le funzionalità fornite dalla piattaforma. Di fatto lo stato di un agente è mappato sotto forma di una collezione di memo. Questa memoria è realizzata attraverso l'utilizzo di uno spazio di tuple e ogni memo è una tupla contraddistinta da un nome e una lista di argomenti.

Il memo space può essere usato anche per coordinare varie sottoattività: è importante sottolineare il fatto che tutte le operazioni siano eseguite in maniera thread safe, ma che comunque ci sia la possibilità di incorrere in corse critiche nel momento in cui più attività operano sulla stessa memo: in generale sono errori di progettazione che sono influenzati dal mancato riconoscimento di una correlazione fra più attività e/o da precondizioni errate.

Un'ultima caratteristica da sottolineare riguarda l'esecuzione di task ciclici che devono eseguire una serie di operazioni ripetitive: per

definire un'attività ciclica occorre settare la proprietà *persistent* e ogni volta che questa è terminata, se sono nuovamente soddisfatte eventuali precondizioni, riparte.

3.2 Artefatto

La definizione di un artefatto avviene attraverso l'estensione della classe *alice.cartago.Artifact*. La sua implementazione può essere studiata su due dimensioni: struttura e comportamento.
La struttura di un artefatto è formata da:

- un numero fisso di variabili che rappresentano lo stato interno non osservabile di un artefatto, implementate come semplici campi di una classe;
- un numero dinamico di proprietà osservabili che possono essere aggiunte/rimosse/modificate attraverso le opportune primitive fornite.

Il comportamento invece è definito da un set di operazioni eseguite nel momento in cui un agente interagisce con l'interfaccia d'uso dell'artefatto. Una operazione può essere:

- **atomica**: è un unico blocco di istruzioni (*operation step*) eseguito in maniera atomica al suo richiamo e dichiarato attraverso un metodo con valore di ritorno *void* e annotazione *@OPERATION*. Il nome e il numero di parametri accettati in ingresso da tale metodo rappresentano le informazioni che devono essere usate e fornite da un agente per utilizzare tale funzionalità. In un dato momento, solo un operation step alla volta può essere eseguito all'interno di un artefatto;
- **strutturata**: permette l'esecuzione di più operation step. Mentre nel caso della operazione atomica, l'operation step era l'operazione stessa, in questo caso è possibile definire in maniera custom più operation step attraverso la definizione di metodi annotati come *@OPSTEP*; questo permette l'esecuzione di operazioni concorrenti composte verificando di volta in volta le eventuali proprietà, che la computazione deve mantenere per il buon funzionamento dell'applicazione.

Esiste un metodo *init* eseguito di default all'istanziazione dell'artefatto e che svolge l'analoga funzione del costruttore nel paradigma OO. Di seguito si riporta qualche esempio.

```
public class Counter extends Artifact {
   @OPERATION void init(){
      defineObsProperty("count",0);
   }
   @OPERATION void inc(){
      int count = getObsProperty("count").intValue();
      updateObsProperty("count",count+1);
   }
}
```

Questo è il caso di un artefatto che non ha nessuna variabile di stato interna ma che definisce comunque una proprietà osservabile. L'atto dell'incremento non è segnalato a priori a qualcuno: l'evento associato viene generato in automatico.

```
public class Counter extends Artifact {
   int count;
   @OPERATION void init(){
      count = 0;
   }
   @OPERATION void inc(){
      count++;
      signal("count_incremented",count);
   }
}
```

In questa situazione invece si ha un altro artefatto che svolge la stessa funzione del primo ma che possiede uno stato privato interno con la notifica di un avvenuto incremento eseguita attraverso una signal. In generale un evento viene automaticamente notificato ai sensori degli agenti che di fatto sono dei buffer privati usati per gestire le percezioni.

E' possibile associare delle guardie alle operazioni, attraverso la loro proprietà *guard*, definendo all'interno dell'artefatto dei metodi con valore di ritorno *boolean* e annotazione *@GUARD*. I parametri di ingresso che una guardia riceve sono gli stessi che vengono inviati all'operazione che adotta quella guardia.

3.3 Interazione artefatto - agente

Un agente sfrutta un artefatto attraverso l'azione primitiva *use* che, utilizzata su un'operazione dichiarata nell'interfaccia dell'artefatto con il numero giusto di parametri, ha successo nel momento in cui l'artefatto stesso ha validato eventuali guardie e fatto partire la sua computazione interna. In linea generale l'esecuzione di un'operazione, da parte di un artefatto, genera una serie di eventi osservabili che sono percepiti da chi esegue l'operazione ma anche da tutti gli agenti interessati ai movimenti dello specifico artefatto usato.
L'azione primitiva *sense* richiede:

- il riferimento all'artefatto;
- l'operazione richiesta a tale artefatto;
- opzionalmente il riferimento al sensore dell'agente che dovrà captare gli eventi generati dall'operazione richiesta;
- opzionalmente un valore di timeout, utile nel caso in cui l'operazione richiesta sia temporaneamente sospesa dall'artefatto stesso; l'agente quindi attenderà il tempo specificato e se la richiesta continuerà allo scadere a non aver successo, restituirà una condizione di fallimento.

A prima vista sembrerebbe una semplice comunicazione sincrona che ha successo nel momento in cui l'artefatto esegue l'operazione richiesta; questo è in pieno contrasto col metamodello utilizzato che predica la comunicazione totalmente asicrona. Però se analizziamo una use, questa può essere sospesa sia se l'artefatto sta eseguendo un'altra operazione e sia se la guardia della operazione richiesta non è valida. Questo vuol dire che la comunicazione è asincrona perchè la use non produce nessun trasferimento di controllo.

E' importante poi notare come gli agenti abbiamo la capacità di creare e trovare gli artefatti attraverso i metodi *makeArtifact* e *lookupArtifact*: queste in realtà sono azioni ausiliarie che interagiscono con due artefatti che appartengono al Workspace in cui gli agenti sono situati, il *factory* e il *registry artifact*. Come è possibile intuire, questi artefatti si occupano della creazione e del censimento degli artefatti appartenenti ad uno specifico workspace.

3.3.1 Sensing degli eventi

Poco fa è stato visto come gli agenti, con l'azione *use*, possono specificare un sensore per osservare eventuali eventi generati dall'artefatto richiamato e che, per ricevere questi eventi, possiedono una coda dalla quale poi andranno a leggere. Il recupero degli eventi dalla queue la esegue l'azione *sense* che, in base ai parametri d'ingresso, possono selezionare e prelevare anche più di un evento alla volta. Questo rende la gestione degli eventi molto flessibile con la capacità dell'agente di autocollegarsi sensori di tipo diverso a seconda della strategia che vuole seguire per la percezione.

Invece dal punto di vista di un agente esterno, per osservare la proprietà di un artefatto, si usano due tipi di azioni:

- **observeProperty**: è una richiesta diretta all'artefatto che fornisce in maniera sincrona il valore della proprietà richiesta;
- **focus/stopFocus**: l'agente vuole essere informato da quel momento in avanti di tutti gli eventi che l'artefatto genera. Con stopFocus, questo vincolo si dissolve.

E' doveroso puntualizzare che in entrambe le azioni l'agente non utilizza mai l'artefatto e quindi lo stato dell'artefatto non viene mai intaccato; gli unici effetti si hanno sul comportamento dell'agente.

Un accenno all'ultimo tipo di interazione: agent-to-agent. In questo caso è necessario l'instaurazione di una comunicazione a scambio di messaggi di tipo request-response e da entrambe le parti è necessario tutto un insieme di operazioni per mantenere viva la conversazione.

Capitolo 4

IBM Aglets

Rappresenta la seconda soluzione ad agenti trattata in questa tesina, sviluppata dalla sezione giapponese di IBM. In maniera analoga a Simpa, è costruita sopra la piattaforma Java ma presenta differenze rilevanti. Si cercherà di condurre un analisi di questo prodotto sotto due punti di vista: il metamodello utilizzato e le modalità implementative esposte.

4.1 Metamodello

Con il termine *aglet* si indica un agente software autonomo Java-based. Per *agente* si intende un'entità computazionale autonoma, in grado di regolare la propria esecuzione in maniera indipendente e avere pieno controllo sul proprio stato. Il ciclo di vita di un agente può essere scandito da un insieme definito di eventi:

- **creazione**: l'agente nasce, inizializza il proprio stato e incomincia le sue attività;
- **clonazione**: un agente esistente viene duplicato; il nuovo aglet generato sarà così identico al primo sia nella forma che nello stato ma, ovviamente, indipendente da esso;
- **invio**: l'agente decide di spostarsi su un altro nodo della rete, portandosi dietro nella sua interezza anche lo stato. Al termine

del suo spostamento, l'aglet è in grado di rispristinare lo stato e riprendere l'esecuzione da dove si era interrotta;

- **ritorno**: l'agente esprime la volontà di ritornare al nodo di rete dal quale in origine si era allontanato;
- **disattivazione**: l'aglet sospende la propria esecuzione;
- **riattivazione**: l'esecuzione dell'aglet riprende da dove precedentemente si era sospesa;
- **morte**: l'agente termina il suo ciclo di vita, perdendo di fatto per sempre il suo stato.

Quindi l'aglet autonomo, è in grado di controllare e decidere in piena indipendenza come il suo ciclo di vita evolverà nel tempo. Da cosa dipendono le sue decisioni? Oltre che dalla sua mera computazione, anche dalle interazioni che avvengono nell'ambiente in cui è situato composto solamente da altri agenti che, contemporaneamente all'aglet target, stanno cercando di assolvere alle funzionalità a loro assegnate. A ogni impulso esterno, l'aglet mette in moto un ragionamento che porta all'evoluzione del suo stato e all'accettazione o meno della richiesta ricevuta. A questa interazione più di tipo event-listener c'è poi da aggiungere una semplice comunicazione basata su scambio di messaggi per il passaggio di informazioni fra aglet.

L'ambiente in cui vive l'agente è chiamato *aglet host* e deve essere presente su ogni nodo nel quale un agente vuole spostarsi e interagire. Gli aglet host svolgono funzionalità accessorie agli agenti, quali l'implementazione di politiche di sicurezza e l'assistenza al loro spostamento da un nodo all'altro della rete. Attraverso questa astrazione, un agente può creare e interagire con nuovi aglet che diventano così parte integrante dell'ambiente.

4.1.1 Metamodello vs realizzazione: considerazioni

Un agente, spostandosi, porta con sè due elementi: lo stato e il controllo, inteso come il ricordo del punto in cui la sua esecuzione si era

sospesa. Come avviene questo processo di migrazione a livello implementativo? Innanzitutto occore specificare il fatto che sono gli aglet host in prima persona ad occuparsi della trasmissione tramite rete dell'agente e al suo ripristino nel nodo destinazione. In secondo luogo, il mezzo utilizzato per il salvataggio dei dati è la *serializzazione* fornita dal pacchetto Java. Questa tecnologia è in grado di convertire tutti i dati presenti nella memoria Heap di un agente in una sequenza di byte che possono poi essere facilmente inviati sulla rete. Questa scelta implementativa porta con sè rilevanti conseguenze in ambito progettuale:

- si ricordi che lo stato di un oggetto in Java è formato da un insieme di dati residenti in tre locazioni: l'heap, lo stack di esecuzione e i registri. La serializzazione agisce solo sulla prima! Questa limitazione, dovuta alle politiche di sicurezza dall JVM, porta con sè una forzatura nel modo di pensare che obbliga a salvare lo stato dell'agente dello stack e dei registri nella heap in modo che questa possa essere prontamente serializzata in caso di migrazione;

- questa operazione di salvataggio deve essere programmata in modo da essere già disponibile all'atto di un'eventuale migrazione, quindi periodicamente l'aglet deve eseguire questo trasferimento di informazioni fra memorie. Bisogna contemplare il fatto che comunque alla richiesta di uno spostamento, l'agente può avere la necessità di terminare prima un'attività in corso oppure di esprimere il rifiuto alla proposta fatta.

4.2 Implementazione

In questa ultima sezione verrà illustrata la struttura che compone l'implementazione di un IBM Aglet e di come un'istanza di agente interagisca con l'ambiente circostante e con gli altri agenti.

4.2.1 Agente

Per definire un agente è necessario estendere la classe *Aglet* che include un insieme di metodi da riscrivere per personalizzarne il comportamen-

to. Le operazioni ivi presenti sono suddivisibili i tre categorie:

- metodi *make-event*;
- metodi *pre-event*;
- metodi *post-event*;
- metodi *behavior-define*.

Prima di avventurarsi nella spiegazione delle varie operazioni, occorre specificare che il modello che realizza la generazione e la notifica degli eventi è basato su callback. Questa scelta porta con sè problematiche rilevanti riguardanti la concorrenza e la gestione della sincronizzazione di più processi in quanto, come si avrà modo di vedere, per il corretto recepimento di un evento, deve essere invocato (invocazione OO) uno specifico handler che si occuperà di gestire la chiamata. E' attraverso questo modo di notificare che l'agente assume consapevolezza di una possibile e imminente serializzazione. Non deve stupire quindi l'organizzazione dei metodi editabili dall'utente perchè sono diretta conseguenza di questo modello adottato.

Al fine di recepire le esigenze provenienti da altri aglets, un agente mette a disposizione i metodi *make-event* che hanno il compito di notificare una richiesta all'agente che li possiede. Sono i seguenti:

- *clone()*: clonazione di agente;
- *dispatch()*: migrazione di un agente presso un altro aglet host;
- *retract()*: successivo ad una migrazione, indica la richiesta all'aglet di ritornare al suo aglet host di origine;
- *deactivate()*: sospensione dell'agente;
- *dispose()*: eliminazione dell'agente.

Ovviamente un aglet può invocare su sè stesso queste operazioni al fine di regolare il suo ciclo di vita. I metodi *pre-event* invece sono quelli che vengono richiamati sull'agente target alla ricezione di una richiesta fatta attraverso i metodi make-event: questi hanno il compito di incapsulare tutta la parte di reasoning per decidere se avvallare o meno la richiesta. Sono i seguenti:

- *onCloning()*: in risposta a clone();
- *onDispatch()*: in risposta a dispatch();
- *onReverting()*: in risposta a retract();
- *onDeactivating()*: in risposta a deactivate();
- *onDisposing()*: in risposta a dispose().

I metodi *post-event* sono quelli che invece vengono richiamati dopo aver accettato ed assolto alla richiesta incapsulata nell'evento:

- *onCreation()*: caso particolare, richiamato solo alla istanziazione dell'agente che sta per prendere vita. In linea generale serve per inizializzare correttamente l'aglet;
- *onClone()*: richiamato dopo aver eseguito una corretta clonazione dell'agente;
- *onArrival()*: richiamato subito dopo una corretta migrazione o ritorno all'aglet host di orgine di un agente;
- *onActivation()*: richiamato dopo un corretto ripristino dell'agente.

Rimangono fuori i metodi *behaviour-define* che sono rappresentati unicamente dall'operazione *run()*. Al suo interno viene definito il comportamento che l'agente deve svolgere per assolvere al suo obiettivo.

Per capire meglio come questi metodi si intrecciano l'uno con l'altro, di seguito verrà riportato un esempio: si supponga di avere un nodo con al suo interno un 'aglet host 1' che contiene un agente chiamato *SlaveAglet*. Su un altro nodo è presente una altro 'aglet host 2' con un agente denominato *MasterAglet*. Entrambi sono già stati instanziati per cui dapprima è stato eseguito il loro metodo *onCreation()* dopodichè hanno iniziato a svolgere le loro mansioni descritte nel *run()*. Ora, si supponga che MasterAglet voglia trasferire, su 'aglet host 2', SlaveAglet: MasterAglet in qualche modo invocherà *dispatch()* su SlaveAglet, il quale, eseguendo *onDispatch()*, si suppone accetti la richiesta. Questo comporta la serializzazione del suo stato corrente e

il suo invio, comprensivo di class loader, all' 'aglet host 2' di MasterAglet. Arrivato e situato, SlaveAglet eseguirà il metodo *onArrival()* e inizierà ad interagire con l'ambiente per assolvere al suo task.

4.2.2 Interazione

Agente - ambiente

Un aglet interagisce con il suo aglet host attraverso un oggetto di tipo *AgletContext*. Questà entità viene richiesta all'ambiente dall'agente invocando il metodo *getAgletContext()* presente nella classe base *Aglet*. Il contesto ricevuto dall'aglet può essere poi utilizzato per:

1. **creare un nuovo aglet**: attraverso l'operazione *createAglet()*;
2. **recuperare un aglet spostato**: attraverso l'operazione *retractAglet()*.

Agente - agente

In precedenza è già stato introdotto il metodo di interazione che sussiste fra agenti basato su callback ed è già stato evidenziato come questa interazione porti inevitabili problemi in ambito di concorrenza e sincronismo. Per risolvere questa situazione spinosa si è creato un oggetto comunicativo ad hoc: l'*AgletProxy*. Si supponga che un MasterAglet richieda lo spostamento di uno SlaveAglet: in primis MasterAglet farà richiesta al suo aglet host dell'AgletProxy legato a SlaveAglet che ne rappresenterà sostanzialmente l'interfaccia d'uso. Dopodiché ne richiederà lo spostamento attraverso di esso. La cosa importante da sottolineare è il fatto che l'AgletProxy permette di astrarre dalla specifica posizione dell'agente richiesto che può essere anche distribuito nella rete: sarà l'AgletProxy locale gestito dall'aglet host del nodo ad occuparsi dell'interazione con lui. Quindi l'accesso a SlaveAglet non è diretto ma è mediato da questo secondo attore, che garantisce il disaccoppiamento dei flussi di controllo e l'autonomia dei vari agenti. L'AgletProxy permette anche la seconda forma di interazione sopra-citata: lo scambio di messaggi. Questo può avvenire in maniera sincrona (*sendMessage*) e asincrona (*sendAsynchMessage*) e si passa

in generale un oggetto di tipo *Message* formato da due campi: il primo, di tipo *String*, rappresenta il tipo del dato trasportato, mentre il secondo è il contenuto informativo vero e proprio che può essere qualsiasi dato primitivo Java. In ultimo si citano i modi a disposizione per ottenere un AgletProxy di un agente:

1. si crea un aglet: attraverso *createAglet()* si ottiene come valore di ritorno il riferimento all'agente istanziato;

2. attraverso una ricerca negli AgletProxy locali con il metodo *getAgletProxies()*;

3. fornendo al metodo *getAgletProxies()* un id (ad ogni Aglet creato viene assegnato un id globale unico) ed eventualmente un URL se l'agente è remoto, dell'aglet cercato.

Capitolo 5

Conclusioni

Le due soluzioni analizzate dichiarano entrambe di essere piattaforme ad agenti ma in realtà sono caratterizzate da forti differenze su due livelli concettuali:

- il metamodello utilizzato: IBM Aglets modella il suo mondo attraverso due sole astrazioni, l'agente e l'ambiente. SimpA invece basa il suo sviluppo su un metamodello più elaborato come l'A&A, che introduce anche il concetto di artefatto;

- il gap che divide progettazione da implementazione fra le due piattaforme è diverso: studiando le metodologie di sviluppo nelle due piattaforme è facile notare che IBM Aglets è ancora legata a pattern e modus operandi tipici dell'object oriented, quali le callback e pattern architetturali come il Proxy che non vengono nascosti dalla piattaforma ma sono ben visibili allo sviluppatore che in qualche modo deve comprenderne il significato e tenerne conto. Su simpA il livello di astrazione è molto più definito: quando si va ad operare direttamente sul codice ci si accorge di pensare molto ad alto livello in termini di astrazioni del metamodello A&A e di dimenticarsi dei piccoli trucchetti implementativi OO che sicuramente sono presenti e rivestono un ruolo fondamentale nel funzionamento della piattaforma, ma che comunque sono nascosti al suo interno e non visibili allo sviluppatore.

Da un punto di vista più business oriented, la piattaforma simpA ha un qualcosa in più: molte delle soluzioni proposte in ambito multiagen-

te non riescono ad affermarsi nel mondo industriale per la mancanza di una politica efficace di integrazione del prodotto sviluppato con le applicazioni già esistenti. Con la definizione del concetto di artefatto, simpA mira a risolvere proprio questo problema, integrando all'interno dell'applicazione multiagente tutto il codice legacy sotto forma di artefatti e aggiungendo all'esterno tutte le funzionalità richieste dal nuovo metamodello A&A. La stessa cosa in IBM Aglets non accade in maniera così chiara e palese. Come ultimo punto di forza comune ad entrambe le piattaforme è da evidenziare l'utilizzo della piattaforma Java, ormai punto di riferimento della comunità sviluppatrice. Anche qui però il modo di utilizzo è diverso: mentre su IBM Aglets, Java è utilizzato come mezzo per la costruzione della struttura delle applicazioni, su simpA la struttura di base è già fornita a priori dalla piattaforma e lo sviluppatore integra in Java le sole istruzioni che modellano il behaviour dell'agente.

Capitolo 6

Bibliografia

- **Introduzione, A&A**

  1. James Odell (May 2002) - *Objects and Agents Compared, Journal of Object Technology, Vol 1, Number 1*;
  2. Omicini, Ricci, Viroli (2008) - *Artifacts in the A&A meta-model for multi-agent systems.*

- **simpA**

  1. Ricci, Viroli, Piancastelli (July 2010) - *simpA: An agent-oriented approach for programming concurrent applications on top of Java, aliCE team, DEIS, Università di Bologna, Italy*;
  2. Ricci - *simpA Annotated Manual, aliCE team, DEIS, Università di Bologna, Italy*;
  3. Ricci - *CARTAGO Annotated Manual, aliCE team, DEIS, Università di Bologna, Italy.*

- **IBM Aglets**

  1. Official IBM Documentation - *http://www.trl.ibm.com/aglets/documentation_e.htm documentazione ibm*;
  2. Bill Venners (April 1997) - *http://www.artima.com/underthehood/aglets.html*;

3. Mokabyte (February 1998) - *http://www.mokabyte.it/1998/02/agenti.htm*.