Capitolo 6
Il modello Swarm

I ricercatori avevano l'esigenza di sviluppare le loro applicazioni senza perdere tempo sui problemi tipici della programmazione, quali ad esempio la generazione di grafici, lo sviluppo di interfacce grafiche ecc. inoltre vi era la necessità di rendere le ricerche riproducibili da altri ricercatori, ciò richiedeva chiaramente l'uso di un linguaggio comune.

Swarm nasce nel 1995 nell'Santa Fe Institute in New Mexico, USA. L'obiettivo degli ideatori di Swarm era di creare un insieme di programmi e librerie da utilizzarsi per simulare ed analizzare sistemi complessi di comportamento, nell'ambito delle scienze sociali e naturali[1].

La nascita delle librerie di funzioni di Swarm soddisfa anche le necessità dei ricercatori, dotando loro di una piattaforma comune e liberandoli dai tipici problemi di programmazione in quanto Swarm prevede già interfaccie grafiche e grafici all'interno delle sue librerie.

Le librerie di Swarm sono scritte in ObjectiveC, però gli ideatori hanno previsto un'interfaccia che permette di scrivere i programmi in Java.

[1] Tratto da *Una piccola introduzione a Swarm: ObjectiveC e Java*, preparata da Marie-Edith Bissey Dipartimento di Politiche Pubbliche e Scelte Collettive POLIS Università del Piemonte Orientale, 7 marzo 2001

Nella simulazione del 118 userò proprio l'interfaccia che permette di utilizzare il linguaggio di programmazione Java e quindi è necessaria una digressione su di esso.

6.1 Programmazione ad oggetti

L'Object-Oriented Programming (OOP) è il paradigma di programmazione che ha sostituito le tecniche di programmazione sviluppate nei primi anni Settanta e basate su procedure. L'idea portante della OOP è che un programma è costituito da oggetti che hanno determinate proprietà e da operazioni che gli oggetti possono svolgere. Quel che interessa ai programmatori è che gli oggetti utilizzati all'interno del programma si comportino come da loro desiderato e non interessa loro come questi oggetti vengono costruiti.

La programmazione ad oggetti rivoluziona il modo di programmare precedente perché permette di invertire l'ordine precedentemente utilizzato per realizzare un programma. Con la struttura tradizionale si impostavano in primo luogo una serie di funzioni (chiamate algoritmi) allo scopo di risolvere un determinato problema, solo successivamente i programmatori cercano un modo appropriato di memorizzare i dati. Per questo motivo, l'autore del Pascal originale, Niklaus Wirth, ha chiamato il suo libro sulla programmazione "*Algorithms + Data Structures = Programs"* (*Algoritmi+Strutture di dati = Programmi*, Prentice Hall, 1975)[2]. Il fatto che Wirth abbia messo la parola "algoritmi" prima di "strutture dati" è indicatore di come i programmatori dell'epoca lavorassero, infatti si decideva prima come manipolare i dati e poi quale struttura impostare ai dati per facilitarne le manipolazioni.

Ciò cha accade con la OOP è esattamente il contrario, con i linguaggi di programmazione ad oggetti si dà la precedenza alla struttura dei dati lasciando in secondo piano gli algoritmi che agiscono sugli stessi.

[2] Cay S. Horstmann, Gary Cornell *Java2 i fondamenti*, McGrawHill

Importantissimo con questo tipo di programmazione è che ad ogni oggetto deve essere assegnata la responsabilità dell'esecuzione di una serie di attività correlate. Se però un oggetto si basa su un'operazione che non rientra nelle sue responsabilità deve avere accesso ad un altro oggetto che tra le sue responsabilità comprende quest'operazione. Chiaramente un oggetto non dovrebbe mai avere la possibilità di manipolare direttamente i dati interni di un altro oggetto, né dovrebbe mai esporre i suoi in modo che altri oggetti possano accedervi. La comunicazione tra oggetti avviene tramite la chiamata di metodi.

Quindi la programmazione ad oggetti sia un modo diverso di programmare. La OOP ha alcune proprietà essenziali:

1. ***Astrazione:*** il programma che descrive un oggetto è scritto in una classe, poi nel programma principale, gli oggetti sono creati come istances della classe.
2. ***Incapsulamento:*** gli oggetti nascondono i loro metodi e dati, separano l'interfaccia dell'implementazione. Si sa cosa fa un oggetto, ma come lo fa si trova da un'altra parte, e può essere modificato senza dover cambiare i programmi che usano l'oggetto. I valori delle variabili nell'oggetto sono propri all'oggetto. Devono essere scritti per passare quest'informazione all'esterno dell'oggetto.
3. ***Eredità:*** ogni classe di oggetti può essere derivata da altre classi già scritte. In questi casi la classe *"figlia"* eredita di tutte le variabili e i metodi della classe *"madre"* , li può modificare e può aggiungerne altri.
4. ***Polimorfismo:*** Si possono avere metodi con lo stesso nome, ma che hanno parametri diversi (si chiama *overloading*).

Si deve notare che Swarm non permette il polimorfismo. I metodi creati in Swarm devono dunque essere unici[3].

6.2 Il linguaggio Java

Il vantaggio del linguaggio Java è che è completamente indipendente dalla piattaforma in cui viene utilizzato. Con Java è possibile utilizzare lo stesso codice in Windows, Linux, Macintosh ecc. inoltre questo linguaggio di programmazione è interamente orientato ad oggetti. Tutto in Java è un oggetto tranne che alcuni tipi di numeri basilari. Il punto essenziale è però il fatto che con Java è più facile produrre codice senza bachi, questo perché gli autori hanno molto meditato su quello che provoca così tanti errori in altri linguaggi di programmazione (come per esempio il C+ +). Quest'opera di meditazione ha permesso di inserire in Java alcune funzioni che vanno ad eliminare le possibilità di creare codici con i tipi più frequenti di bachi:

- L'allocazione e la deallocazione manuali della memoria sono state eliminate
- La memoria in Java viene automaticamente sottoposta a garbage collection (ripulitura di spazio). Gli sprechi di memoria non devono mai costituire una preoccupazione.
- Sono stati introdotti veri array ed è stata eliminata l'aritmetica dei puntatori. Non bisognerà mai preoccuparsi di sovrascrivere un'area di memoria essenziale a causa di un errore quando si lavora con un puntatore.
- È stata eliminata la possibilità di confondere un assegnazione con una verifica dell'uguaglianza di un'istruzione condizionale.*[...]*

[3] Quanti scritto nelle note è tratto da *Una piccolo introduzione a Swarm: ObjectiveC e Java*, preparata da Marie-Edith Bissey Dipartimento di Politiche Pubbliche e Scelte Collettive POLIS Università del Piemonte Orientale, 7 marzo 2001

- È stata eliminata l'ereditarietà multipla, sostituita dal nuovo concetto di interfaccia, tratto da Objective C[4]. *[...]*

6.2.1 La nascita di Java

Java risale al 1991, quando un gruppo di ingegneri della Sun, guidati da Patrick Naughton e da James Goslin, ha voluto elaborare un linguaggio per computer utilizzabile per dispositivi di consumo quali le scatole di commutazione per la TV via cavo. La problematica principale è data dal fatto che questi dispositivi non sono dotati di molta potenza e neanche di molta memoria quindi era nata la necessità di avere un linguaggio piccolo e solido. Inoltre, poiché i produttori potevano scegliere CPU a loro più congegnali tra quelle in commercio vi era anche l'esigenza di non legarsi ad un'unica architettura. Il neonato progetto prende il nome di Green.

Date le esigenze del progetto l'équipe decise di riprendere il modello provato con alcune implementazioni di Pascal. Quello che il padre del Pascal aveva inventato e che l'UCSD Pascal aveva commercializzato era un linguaggio portabile che generava codice intermedio per macchine ipotetiche, le virtual machine (VM), da cui JVM. Tale codice quindi poteva essere utilizzato da qualunque computer possedesse una VM.

Tuttavia gli ingegneri assegnati al progetto dalla Sun provenivano da un ambiente UNIX e quindi il linguaggio di programmazione si cui si basavano era più C + + che Pascal, inoltre il linguaggio utilizzato era orientato ad oggetti anziché essere un linguaggio procedurale. Ma come ha affermato Goslin *"Il linguaggio è sempre stato un mezzo e non un fine"*.

Alla nascita del nuovo linguaggio di programmazione fu lo stesso Goslin a proporre come nome quello di *OAK* nome che però venne cambiato ben presto in *Java* perché OAK era già presente sul mercato dei linguaggi di programmazione.

[4] Quanto nell'elenco puntato è tratto da Cay S. Horstmann, Gary Cornell *Java2 i fondamenti*, McGrawHill

Già nel 1992 il progetto Green dava vita al suo primo prodotto. *"*7"*. Si trattava di un comando a distanza estremamente intelligente, e programmato con il neonato Java. Sfortunatamente nessuno in Sun era disposto a produrre questo tipo di tecnologia, così il progetto Green dovette cercare altre strade per commercializzare i suoi prodotti, tuttavia, le aziende di elettrodomestici non erano interessate e così il gruppo cercò di indirizzarsi verso un progetto consistente in una scatola di commutazione per TV via cavo in grado di gestire i nuovi servizi quali, ad esempio, il *"video on demand"* . il gruppo Green non riuscì ad ottenere il contratto. L'accordo lo raggiunse un'altra società appartenente a Jim Clark (ideatore di Netscape), il quale successivamente contribuì in modo rilevante allo sviluppo di Java.

L'équipe Green Project (che nel frattempo aveva cambiato il nome in First Person Inc.) trascorse tutto il 1993 e i primi sei mesi del 1994 cercando di vendere la sua tecnologia senza riuscirci. Nel 1994 il gruppo venne sciolto.

Mentre avveniva tutto questo la componente World Wide Web di Internet cresceva velocemente ed ininterrottamente, la chiave di volta è la nascita del browser. Nel 1994 molti utenti utilizzavano Mosaic, browser non commerciale sviluppato all'università dell'Illinois nel 1993[5].

Verso la metà del 1994, secondo quanto affermato da Goslin nell'intervista a Sun World, gli sviluppatori del linguaggio si resero conto che una delle applicazioni per questo linguaggio era quello di impiegarlo nel mondo dei browser. Questo era dovuto al fatto che nel mondo delle workstation erano importantissimi l'indipendenza dell'architettura, il tempo reale, l'affidabilità e la sicurezza, tutti pilastri su cui si basava il linguaggio Java.

Il vero e proprio browser, che si è poi evoluto nel HotJava che conosciamo, è stato costruito da Patrick Naughton e da Jonathan Payne. HotJava venne costruito con il preciso obiettivo di mettere in risalto la

[5] Mosaic fu parzialmente scritto da Marc Andressen che ottenne fama e fortuna in qualità di cofondatore e capo della tecnologia di Netscape.

potenza del linguaggio e di quelle che gli autori chiamano applet, nacque così un browser in grado di eseguire il codice all'interno della pagina Web. La tecnologia venne presentata nel corso del SunWorld '95[6] .

Il grande miglioramento che ha contribuito alla diffusione di Java si è verificato nell'autunno di quello stesso anno, quando Netscape decide di abilitare all'uso di Java il browser Navigator che uscì nel gennaio del 1996. In seguito altre società (IBM, Symantec, Imprise, Microsoft e molte altre) ottennero la licenza .

Sun rilasciò la prima versione di Java all'inizio del 1996 alla quale fa seguito un paio di mesi più tardi la versione 1.02 che però risultò poco idonea per applicazioni serie.

Solo nella conferenza JavaOne tenutasi a San Francisco nel maggio del 1996 fu chiarita la direzione che Java stava seguendo. In questa occasione Sun annunciò una serie infinita di miglioramenti che dovevano essere apportati al neonato linguaggio.

La grande novità si ha in seguito alla conferenza JavaOne del 1998, quando venne annunciato l'imminente rilascio di Java 2. Con questa nuova versione vengono sostituite le prime versioni dei toolkit, delle GUI e della grafica con toolkit sofisticati. Finalmente il linguaggio diviene utilizzabile per applicazioni professionali e si avvicina alla promessa *"Write once, run everywhere"*. Il nome viene cambiato in Java 2 tre giorni dopo il rilascio nel dicembre del 1998.

Da allora sono stati fatti moltissimi passi avanti e l'anima della piattaforma si è stabilizzata. L'attuale versione di Java (J2SE 1.4.2 SDK[7]) è molto migliorata rispetto alla versione iniziale, sono state introdotte nuove funzioni, sono migliorate le prestazioni e naturalmente sono stati eliminati alcuni bachi.

[6] Il 23 maggio 1995
[7] Ultima visita al sito della sun il 25 gennaio 2004.

6.2.2 Il "White Paper" di Java

Gli autori di Java hanno scritto un White Paper[8] che spiega i loro intenti e conseguimenti. Le parole chiave del linguaggio sono:

- Semplice
- Ad oggetti
- Distribuito
- Robusto
- Sicuro
- Indipendente dall'architettura
- Portabile
- Interpretato
- Ad alte prestazioni
- Multithreaded
- Dinamico

Vediamo ora quale significato danno gli autori di Java alle parole chiave[9].

Semplice

Gli autori desideravano costruire un sistema che potesse essere programmato facilmente senza richiedere una pesante formazione e che facesse leva sulla pratica standard. Java è così stato progettato attenendosi quanto più possibile al C + + omettendo però molte funzioni poco utilizzate, mal comprese e fuorvianti di questo linguaggio. Un altro aspetto della semplicità che gli autori volevano dare al linguaggio è la dimensione

[8] E' reperibile sul sito della sun http://java.sun.com
[9] Il significato delle parole chiave qui riportato è tratto da Cay S. Horstmann, Gary Cornell *Java2 i fondamenti*, McGrawHill

limitata, infatti uno degli obiettivi di Java consiste nell'agevolare la costruzione di software in piccoli computer[10].

A oggetti

La progettazione ad oggetti è una tecnica di programmazione che si concentra sui dati, che sono gli oggetti, e sulle relative interfacce. I servizi ad oggetti di Java sono quelli di C + +, le differenze risiedono nell'ereditarietà multipla e nel modello di metaclassi di Java.

Distribuito

Java possiede un'estesa libreria di routine per gestire i protocolli TCP/IP quali http e FTP. Le applicazioni java possono aprire oggetti e accedere ad essi attraverso la Rete, tramite URL, con la stessa agevolezza con cui accede ad un file system locale.

Robusto

Java insiste molto sul controllo iniziale degli eventuali problemi, sul successivo controllo run-time e sull'eliminazione di situazioni che possono comportare errori.

Sicuro

Gli autori di Java hanno molto insistito sulla sicurezza, questo perché Java è destinato all'impiego in ambienti di rete/distribuiti. Secondo gli autori Java è tanto sicuro da permettere la costruzione di sistemi privi di virus e al sicuro da interferenze indesiderate.

Tuttavia un gruppo di esperti di sicurezza presso la Princeton University[11] in ha trovato i primi buchi inerenti alla sicurezza di Java 1.0

[10] Si pensi alla diffusione di questo linguaggio nei software dei cellulari di nuova generazione. Questo è dovuto al fatto che l'interprete di base e il supporto delle classi pesano circa 40KB, ed aggiungendo le librerie standard di base ed il supporto dei thread la dimensione aumenta di altri 175 KB. È quindi chiaro che questo linguaggio risulta ideale per essere applicato a computer che hanno limitate capacità di memoria.
Bisogna però notare che non tutti i componenti di Java sono leggeri, infatti le librerie GUI sono molto più voluminose.

poco dopo il rilascio di JDK 1.0. altri bachi hanno continuato ad essere rilevati anche in altre versioni successive[12] .

Tuttavia l'équipe di Java ha dichiarato di non tollerare le lacune in fatto di sicurezza e pertanto è all'opera per correggere gli eventuali bachi rilevati nei meccanismi di sicurezza. Per aiutarsi a trovare i difetti nei sistemi di sicurezza Sun[13] ha adottato la politica di rendere pubbliche le specifiche interne delle modalità di funzionamento dell'interprete Java coinvolgendo in questo modo la comunità esterna nella ricerca dei buchi di sicurezza.

Indipendente dall'architettura

Il codice compilato è eseguibile su molti processori, data la presenza di un sistema run.time Java. Java infatti genera istruzioni bytecode che non hanno nulla a che vedere con una specifica architettura ma sono progettate per risultare facilmente interpretabili da qualsiasi computer.

Portabile

Le librerie appartenenti al sistema definiscono interfacce portabili, ma scrivere un programma che si presenti bene in Windows, Machintosh e UNIX è uno sforzo immenso. Java ha però compiuto l'impresa fornando un toolkit semplice che mappava gli elementi comuni dell'interfaccia utente in numerose piattaforme. All'inizio però questa libreria risultava oberata di lavoro e dava risultati appena accettabili. Ormai il toolkit dell'interfaccia utente è stato completamente riscritto rispetto a quello presente in Java 1.0 ed ormai risulta tale da non basarsi più sull'interfaccia utente dell'host.

Interpretato

L'interprete Java può eseguire bytecode Java in qualsiasi computer in cui sia stato trasportato l'interprete. Poiché il collegamento è un processo

[11] Cay S. Horstmann, Gary Cornell *Java2 i fondamenti*, McGrawHill
[12] Per conoscere le opinioni di consulenti esperti riguardo all'attuale stato sulla sicurezza di di Java si può consultare il sito http://www.cs.princeton.edu/sip
[13] Per questioni legate alla sicurezza l'URL di Sun è attualmente http://java.sun.com/sfaq

più incrementale e leggero, il processo di sviluppo può essere molto più rapido ed esplorativo.

Ad alte prestazioni

Se si utilizza un interprete la dicitura *"ad alte prestazioni"* non sembra adeguata però esiste, in molte piattaforme, un'altra modalità di compliazione, che è il compilatore *just-in-time* (JIT). Questo compilatore funziona compilando una volta il bytecode nel codice nativo, inserendo in una cache i risultati e richiamandoli nuovamente all'occorrenza. Questo accelera il codice perché l'interpretazione del codice utilizzato deve essere eseguita una sola volta. Sebbene il compilatore in questione sia sempre più lento di un vero e proprio compilatore di codice nativo risulta comunque più veloce di un normale interprete[14].

Multithreaded

I vantaggi del multithreading consistono nella garanzia di una risposta interattiva. Inoltre in Java i thread hanno la possibilità di avvalersi di sistemi multiprocessore, sempre che il sistema operativo di base sia in grado di gestirli. Il lato negativo di tutto ciò è che le implementazioni dei thread differiscono sensibilmente sulle varie piattaforme e Java non fa nulla per rendersi indipendente da queste ultime per quanto riguarda questo aspetto. In sostanza Java demanda la realizzazione del multithreading al sistema operativo sottostante o ad una libreria di thread.

Dinamico

Il linguaggio Java è stato studiato per adattarsi ad un ambiente in evoluzione. Infatti possono essere aggiunte liberamente nuove istanze e nuovi metodi senza che vi siano ripercussioni sui rispettivi client. Inoltre in Java è possibile prelevare informazioni di tipo run-time, ciò facilita

[14] Quanto sopra si trova in Cay S. Horstmann, Gary Cornell, *Java2 I fondamenti*, McGraw-Hill, 2001

l'aggiunta di codice in un programma in esecuzione[15]. Questa facilità nel reperire informazioni di tipo rin-time è molto utile per sistemi che richiedono di analizzare gli oggetti in fase di costruzione[16].

6.2.3 Le Classi

Una classe è solitamente descritta come modello o schema dettagliato dal quale l'oggetto sarà effettivamente creato. Quando un oggetto è creato da una classe si dice che è stata creata un'*istanza* della classe. Tutto in Java è contenuto in una classe, le librerie standard forniscono migliaia di classi per gli scopi più diversi però ogni programmatore dovrà comunque creare delle nuove classi specifiche per gli scopi che si è prefissato.

Le classi possono essere relazionate tra loro e le relazioni più comuni sono:

1. *Dipendenza*
2. *Aggregazione*
3. *Ereditarietà*

La relazione di *dipendenza* nasce se una classe manipola gli oggetti di appartenenti ad un'altra classe. Naturalmente è necessario ridurre al minimo la dipendenza tra classi e questo nasce dal fatto che se una classe ***A*** non è consapevole[17] dell'esistenza di un'altra classe ***B*** non risente delle modifiche apportate in quest'ultima e quindi si riducono al minimo i tempi di ricerca dell'errore nel caso si verifichino problemi con il programma.

La relazione di *aggregazione* è data dal fatto che gli oggetti di una classe contengono gli oggetti di un'altra classe. Quindi per tornare alle due

[15] Un buon esempio di quanto detto è il codice prelevato dainternet per essere fatto girare su un browser
[16] Si pensi ad esempio ai costruttori di GUI, ai debugger intelligenti ecc.
[17] E quindi non dipende in nessun modo da un'altra classe

classi di prima si può dire che gli oggetti della classe ***A*** contengono gli oggetti della classe ***B***.

La relazione di *ereditarietà* denota una relazione tra una classe specializzata ed una più generale. La classe specializzate contiene metodi specializzati per gestire alcune problematiche ma eredita i metodi di una classe più generale per gestire situazioni che necessitano di un trattamento più generale. Detto in altri termini si può affermare che se la classe ***A*** estende la classe ***B*** la prima eredita i metodi della seconda ma possiede funzionalità supplementari.

6.3.4 Gli Oggetti

In Java tutto è trattato come un oggetto, ciò permette l'esistenza di un'unica sintassi coerente da utilizzarsi dappertutto. Sebbene in questo linguaggio tutto è trattato come un oggetto ciò che viene manipolato durante la programmazione non è l'oggetto direttamente ma il riferimento all'oggetto che si vuole modificare. Naturalmente il solo fatto di avere creato un riferimento non vuol dire che questo riferimento sia automaticamente collegato ad un un oggetto. In *thinking in Java* vi è un bell'esempio che chiarisce quanto detto sopra. L'esempio individua una situazione in cui ci sono un televisore (oggetto) ed un telecomando (riferimento) fino a che si tiene in pugno il riferimento si ha un collegamento con l'oggetto, se vogliamo cambiare canale manipoliamo il riferimento per modificare l'oggetto. Inoltre se vogliamo spostarci in un'altra stanza ma desideriamo continuare a controllare il televisore ci basta portare con noi il telecomando e non spostare il televisore. Infine il nostro telecomando può stare senza televisore, ciò implica, come gia detto sopra, che il solo fatto di avere un riferimento non significa necessariamente che esista un oggetto collegato a questo.

Per lavorare con gli oggetti si deve:

- Costruire gli oggetti e specificare il loro stato iniziale
- Applicare un metodo agli oggetti

Nel linguaggio di programmazione Java si utilizzano dei costruttori (che sono speciali metodi il cui scopo è costruire ed inizializzare oggetti) per costruire nuovi esemplari di oggetti.

Per costruire un oggetto si combinano il costruttore con l'operatore **new**. Per esempio:

```
new Date
```

l'oggetto è poi inizializzato.

Se si desidera, ora che l'oggetto è stato creato, si può passare lo stesso ad un metodo

```
System.out.println(new Date());
```

o applicare all'oggetto un metodo.

```
String s = new Date(). toString;
```

In genere però si preferisce rimanere legati agli oggetti che si sono creati per poterli utilizzare più volte e per fare ciò si procede semplicemente memorizzando l'oggetto in una varaibile:

```
Date compleanno = new Date ()
```

Quindi ora la variabile oggetto (o riferimento) *compleanno* si riferisce all'oggetto appena costruito *Date*.

È importante sottolineare che, a differenza dell'oggetto, la variabile oggetto costruita senza riferimento all'oggetto, esempio:

```
Date scadenza;
```

non potrà vedere applicati su di essa i metodi.

Quindi

```
s= scadenza. toString();
```

produrrà un errore quando il codice verrà compilato. È necessario inizializzare la variabile oggetto:

```
scadenza = new Date();
```

o impostare la variabile in modo che si riferisca ad un oggetto esistente:

```
scadenza = compleanno
```

Le caratteristiche fondamentali degli oggetti in un linguaggio di programmazione ad oggetti sono sostanzialmente tre:

1. *Comportamento dell'oggetto*: cosa si può fare con l'oggetto e quali metodi possono essere applicati? Il comportamento di un oggetto è quindi definito dai metodi che si possono chiamare e gli oggetti della stessa classe supportano un comportamento analogo e quindi possono richiamare metodi del tutto simili.
2. *Lo stato dell'oggetto*: come reagisce l'oggetto quando si applicano i metodi? Lo stato di un oggetto non cambia nel tempo (infatti l'oggetto memorizza informazioni riguardo al proprio aspetto) se non conseguentemente alle chiamate dei metodi. Se il cambiamento dell'oggetto non è avvenuto in questo modo vuol dire che si è interrotto l'incapsulamento.
3. *L'identità dell'oggetto*: come si distingue l'oggetto dagli altri se questi possono avere lo stesso comportamento e stato? Lo stato

di un oggetto non descrive completamente l'oggetto, infatti ogni oggetto ha un'identità distinta, gli oggetti di una stessa classe differiscono sempre per l'identità e solitamente per lo stato.

6.3 Gli agenti e gli oggetti del modello

Rappresentare un modello con la simulazione implica la comprensione del funzionamento della realtà.

Dopo aver individuato il livello di dettaglio ideale per la rappresentazione della realtà che si vuole studiare all'interno del modello di simulazione e dopo aver isolato le semplificazioni ammissibili è possibile realizzare oggetti software utili per rappresentare i componenti che hanno un ruolo significativo nella realtà.

6.3.1 Gli Agenti del modello

Moltissimi oggetti operanti all'interno della realtà possono essere considerati degli agenti. Le caratteristiche fondamentali di un agente sono:

1. *Autonomia:* stato interno secondo cui si prendono le decisioni, tenendo conto di altre considerazioni
2. *Redditività:* interazione con l'ambiente e gli altri agenti
3. *Pro*-attività: il comportamento degli agenti deve essere guidato da obiettivi.
4. *Capacità sociale:* è la capacità di interazione degli agenti attraverso un "linguaggio"

Gli agenti sono poi chiamati con diversi nomi a seconda del tipo di agente che si vuole descrivere. Abbiamo quindi:

1. *Agenti mobili*: agenti descritti da pezzi di software che si spostano sulla rete.
2. *Protocolli di comunicazione*: agenti stabili, e quindi pezzi di software che rimangono fermi e che si limitano a inviare richieste in rete
3. *Modelli ad agenti*: è questo il caso che più ci interessa. In questa circostanza diventa importantissimo come è fatto l'agente al suo interno. In questa categoria si possono riportare ad esempio due tipi di agenti, ossia l'agente caratterizzato da un obiettivo da raggiungere e l'agente che è un oggetto software inserito all'interno di un ambiente ed è capace di azioni autonome e flessibili che gli permettono di raggiungere l'obiettivo per cui è stato progettato.

Al nostro fine, ossia la simulazione, sono importanti soprattutto i modelli ad agenti ed in particolare gli agenti del secondo tipo i quali permettono di trovare soluzioni plausibili a problemi di tipo sociale.

Per capire meglio quanto affermato sopra cito quanto detto in Terna[18] (1994) a proposito dell'uso di agenti in modelli che descrivono l'economia e sulla capacità degli stessi di far emergere non linearità.

(...) Un prerequisito per l'impostazione che segue è l'uso di modelli semplici, non solo nella direzione consueta della semplificazione propria delle microfondazioni dei modelli economici, ma soprattutto nella eliminazione delle ipotesi relative alla completa razionalità ed alla illimitata capacità computazionale degli agenti. Per contro, introducendo l'interazione tra agenti, di fatto assente nell'impostazione tradizionale, comportamenti molto complessi, anche eventualmente non previsti o non prevedibili, possono emergere dall'aggregazione di singole componenti elementari, in modo non lineare.

[18] Reti neurali artificiali e modelli con agenti adattivi

Ciò non avviene nei modelli neoclassici, dove gli agenti sono supposti interagire tramite il sistema dei prezzi e operando con meccanismi impersonali qual è astrattamente il mercato.

All'interno del modello Swarm alcuni oggetti hanno un corrispettivo con la realtà , in questo caso si parla di *agenti* veri e propri, altri rappresentano una componente fisica o immateriale del 118 virtuale ed altri ancora hanno funzioni di servizio. Soprattutto l'ultima categoria di oggetti rientra nel modello per l'adozione di un rigoroso paradigma di progettazione che impone di mantenere separati gli agenti e le relative regole di funzionamento. Questo paradigma prende il nome di *Environment-Agents-Rules* (ERA) e sarà descritto più avanti.

6.3.2 Gli Oggetti del modello

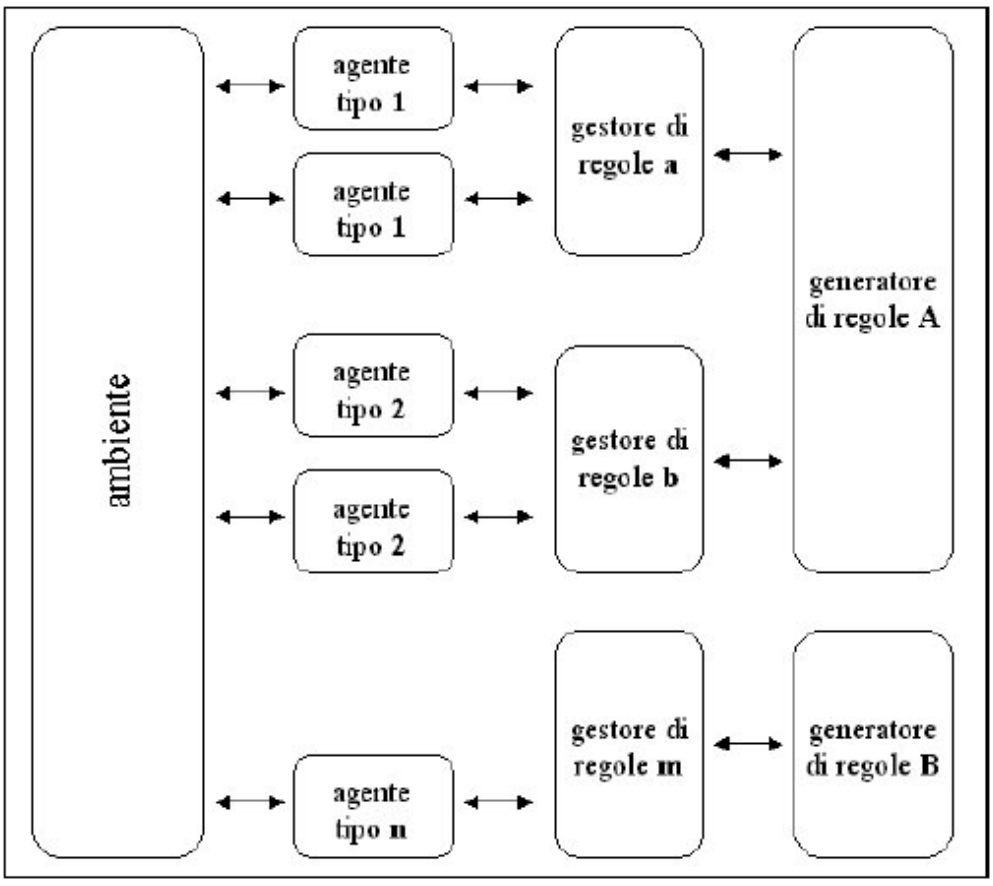
I modelli realizzati in Swarm prevedono diversi livelli nei quali gli oggetti del modello si situano ed operano. L'oggetto principale all'interno del quale si situano è *l'ObserverSwarm.* Questo oggetto, presente in tutti i modelli costruiti con swarm, è una sorta di contenitore all'interno del quale si il mondo artificiale (*ModelSwarm)* osservare e gli strumenti di misura dello sperimentatore sotto forma di oggetti utili per la rappresentazione grafica. Per ciò che riguarda la rappresentazione grafica, è molto importante l'oggettto di tipo *Schedale* il quale permette l'aggiornamento dei grafici in maniera asincrona rispetto al tempodella simulazione, questo per permettere una riduzione dei tempi macchina. Infatti la gestione dei grafici è quello che richiede maggiori tempi di esecuzione e maggiori risorse al computer.

L'oggetto *ModelSwarm*, come già detto, è in posizione gerarchica inferiore rispetto all'*ObserverSwarm*, ma è molto importante perché è il contenitore dell'esperimento che si vuole fare con la simulazione in Swarm.

6.4 Lo schema ERA

Lo schema ERA (*Envoirment-Rules-Agents)* permette di costruire gli oggetti che rappresentano gli agenti all'interno delle nostre simulazioni seguendo alcune regole molto importanti.

Lo schema ERA gestisce quattro strati differenti nella costruzione del modello e degli agenti come è ben visibile nella figura qui sotto riportata.



Ora descriviamo i differenti strati gestiti dallo schema come descritti da Terna (2002).

1. Un primo strato rappresenta l'ambiente in cui gli elementi sono chiamati ad interagire tra loro. All'interno del

protocollo Swarm normalmente la classe è definita *ModelSwarm,* vale a dire il contesto all'interno del quale si definiscono gli agenti, se ne strutturano le liste, si individuano gli eventi nel tempo, si chiariscono le regole di interazione tra gli agenti grazie ai metodi (interpretabili come messaggi che gli agenti sono in grado di gestire, anche reagendo con azioni ed informazioni) definiti all'interno degli oggetti creati dal Modello.

2. Un secondo strato è appunto quello degli agenti, che possono essere costruiti come esemplari di una o di più classi, a loro volta generate ereditando proprietà, caratteri, dati e metodi da classi più generali.
3. Il terzo strato gestisce le modalità attraverso cui gli agenti decidono il proprio comportamento. Ad ogni scelta, l'agente interroga un oggetto sovraordinato, definito gestore di regole (classi dette *RuleMaster*), comunicandogli i dati necessari ed ottenendo le indicazioni di azione.
4. Il quarto strato tratta la costruzione delle regole. Esattamente come gli agenti interrogano i gestori di regole, i gestori di regole interrogano i generatori di regole (classi dette *RuleMaker*) per modificare la propria linea di azione.

Un'importante precisazione metodologica riguarda il terzo e il quarto strato, indicati di cui si è parlato poco sopra: grazie ai produttori di regole, ed ai gestori di regole, è possibile sviluppare ogni tipo di agente che possa essere scritto con un codice inforamatico; ciò vale in particolare per gli agenti cosiddetti cognitivi o BDI (*Beliefs, Desires, Intentions*), cioè agenti costruiti in modo che esprimano comportamenti correlati a un protocollo di comportamento, cui si attribuiscono significati.

Lo schema ERA è un tentativo che va nella direzione di rendere ordinato e modulare il codice informatico, infatti se il codice viene scritto seguendo questo schema risulta piuttosto agevole sostituire di volta in volta

i gestori di regole semplicemente sostituendo gli oggetti introdotti nel modello, come per esempio un sistema a regole fisse, una rete neurale, un algoritmo genetico, un sistema classificatore.

A loro volta reti neurali, algoritmi genetici, sistemi classificatori ecc. avranno la necessità di ricorrere a generatori di regole, differenziati a seconda della loro

rispettiva tipologia: ciò sarà facilitato proprio dalla modularità del disegno adottato.