Corso di Sistemi Multi-Agente LS (A.A. 2008-2009)

DAGLI OGGETTI AGLI AGENTI: VERSO UN SALTO DI PARADIGMA

Approfondimento a cura di Salvatore Presta

Introduzione

Scopo del presente approfondimento è illustrare ed argomentare il salto di paradigma attualmente in atto nel mondo dell'informatica.
E' sempre più evidente, infatti, la necessità e la consapevolezza, almeno nella comunità scientifica, di un cambio di rotta: da una visione del "mondo" orientata agli oggetti ad una orientata agli agenti. Ciò non implica necessariamente, come verrà anche evidenziato in questo lavoro, l'abolizione o la cancellazione di tutte le tecniche e le metodologie orientate agli oggetti, ma è evidente che queste sono nettamente superate e sempre più spesso messe in crisi dalla complessità e dalle caratteristiche del software moderno. Si va in sostanza verso un mondo dove non tutto è un "oggetto", dove non tutto è "un agente", ma dove agenti ed oggetti possono, anzi dovrebbero, coesistere per sfruttare al meglio le loro potenzialità.

Per quanto detto sopra appare fondamentale definire, in prima istanza, che cosa si intende con salto di paradigma (Capitolo 1) ed evidenziare i fattori e le motivazioni che mettono in luce la necessità di tale salto (Capitolo 2). Si proseguirà poi, più in dettaglio, fornendo una descrizione generale di un paradigma di programmazione (Capitolo 3) per poter, quindi, meglio confrontare il paradigma di programmazione orientato agli agenti con il paradigma di programmazione orientato agli oggetti (Capitolo 4). Verrà fornita di seguito, per completezza, una breve sinossi dell'evoluzione dei linguaggi di programmazione (Capitolo 5), per poi mettere in chiaro i rapporti che legano i linguaggi di programmazione con l'ingegneria del software (Capitolo 6). Sarà così possibile, infine, fornire una breve descrizione dell'ingegneria del software orientata agli agenti, AOSE (Capitolo 7), con particolare riferimento alle caratteristiche che distinguono gli agenti dagli oggetti, che evidenziano, ancora una volta, la necessità di un salto di paradigma.

1. Cos'è un salto di paradigma.

Secondo la definizione fornita dal Dizionario Zanichelli (Zingarelli2010) un paradigma è:

> *nella filosofia della scienza, un insieme coerente e articolato di teorie, metodi e procedimenti che contraddistinguono una determinata fase dell'evoluzione di un sapere scientifico.*

Tuttavia il primo a teorizzare un salto di paradigma fu, nel 1962, Thomas Kuhn [1]. Egli infatti ideò, definì e poi diffuse il concetto di "salto di paradigma". Kuhn sostiene che il progresso scientifico non è un processo evolutivo, ma piuttosto una "serie di pacifici interludi interrotti da violente rivoluzioni intellettuali", e in queste rivoluzioni "una visione concettuale del mondo è sostituita da un'altra".
Generalizzando si può pensare al salto di paradigma come ad un cambiamento da un modo di pensare ad un'altro. È una rivoluzione, una trasformazione, una sorta di metamorfosi, che non accade, ma è piuttosto guidata da agenti di cambiamento.
Gli agenti di cambiamento, per esempio, aiutarono a creare un salto di paradigma muovendo la teoria scientifica dal sistema tolemaico al sistema copernicano, e muovendo dalla fisica newtoniana alla fisica relativistica e quantistica. Entrambi i movimenti cambiarono alla fine la visuale del mondo. Queste trasformazioni furono graduali dal momento che le vecchie credenze furono rimpiazzate da un nuovo paradigma conforme a quelle teorie.
Allo stesso modo la stampa, la realizzazione di libri e l'uso del volgare inevitabilmente cambiarono la cultura della gente ed ebbero un impatto diretto sulla rivoluzione scientifica. L'invenzione di Gutenberg, nel 1440, fu un agente di cambiamento. I libri diventarono facilmente accessibili, più piccoli, facili da maneggiare e più economici. Una grande quantità di gente ebbe direttamente accesso alle scritture. Gli atteggiamenti cominciarono a cambiare dal momento che la gente si emancipò dalla dominazione della chiesa.
In modo analogo, gli agenti di cambiamento ci hanno guidato in tempi recenti ad un nuovo salto di paradigma. I segni sono sotto gli occhi di tutti. Per esempio, l'introduzione del personal computer e di internet hanno avuto un impatto sia sulle persone sia sull'ambiente degli affari, essendo state queste due innovazioni dei veri e propri catalizzatori per un salto di paradigma. Ci stiamo spostando, o più probabilmente ci siamo già spostati, da una società industriale meccanica e manifatturiera a una società centrata sull'informazione e sui servizi e l'aumento di tecnologia continuerà ad avere un impatto globale. Il cambiamento è inevitabile.
Allo stesso modo, la società dell'informazione sta vivendo anch'essa un salto di paradigma, ancora quasi tutto in divenire, ma i cui segnali sono evidenti. In questo caso è il paradigma dei linguaggi di programmazione a cambiare, da una programmazione orientata agli oggetti ad una orientata agli agenti. Tale processo è sicuramente guidato da diversi fattori, tra cui probabilmente il più importante è la complessità che caratterizza sempre più i software moderni.

Generalizzando, per milioni di anni ci siamo evoluti e continuiamo a farlo. Il cambiamento è difficoltoso. La natura umana resiste al cambiamento; il processo è stato avviato molto tempo fa e continueremo a co-creare la nostra esperienza personale. Kuhn afferma, infatti, che "la consapevolezza è il prerequisito a tutte le modifiche accettabili della teoria". Tutto comincia nella mente della persona. Quello che percepiamo, normale o meta-normale, conscio o inconscio, è soggetto alle limitazioni e alle distorsioni prodotte dalla nostra natura che è innata e condizionata dalla società. Comunque non siamo limitati da questo, dato che possiamo cambiare. In sostanza, ogni volta che ci troviamo di fronte a un salto di paradigma, ci muoviamo rapidamente e il nostro stato di consapevolezza trascende e si trasforma, molti aprono gli occhi dato che la nostra consapevolezza si espande.

2. Verso un salto di paradigma.

Come si afferma in [2] la complessità introdotta dai moderni sistemi software attraverso diversi scenari di computazione emergenti oltrepassa le capacità fornite dall'informatica tradizionale e le attuali astrazioni dell'ingegneria del software, come le metodologie orientate agli oggetti e basate su componenti.
Lo scenario che delinea l'imminente crisi del software sta emergendo rapidamente: i sistemi informatici saranno dappertutto, sempre connessi, sempre attivi e inglobati in qualsiasi oggetto. Le tecnologie senza fili, per di più, renderanno la connettività di rete un aspetto pervasivo poiché ogni dispositivo computazionale sarà connesso ad una qualche rete.
Questo scenario non riguarda soltanto la progettazione e lo sviluppo di sistemi software dal punto di vista quantitativo (in termini di numero di componenti e capacità richieste), ma anche da quello qualitativo per quanto concerne i cambiamenti nelle caratteristiche dei sistemi software, così come le metodologie adottate per il loro sviluppo. Più in particolare, quattro caratteristiche, in aggiunta all'incremento del numero di sistemi informatici interconnessi, distinguono i futuri sistemi software da quelli tradizionali:

- localizzazione ("situatedness"): i componenti software operano in un ambiente, che possono influenzare e dal quale possono essere influenzati;
- apertura ("openness"): i sistemi software sono soggetti ad una gestione decentralizzata e possono dinamicamente cambiare la loro struttura;
- località nel controllo ("locality in control"): i componenti di un sistema software rappresentano luoghi di controllo autonomi e proattivi;
- località nelle interazioni ("locality in interaction"): nonostante operino in un mondo totalmente connesso, i componenti software interagiscono tra di loro secondo modelli di interazione locale (geografici o logici).

L'integrazione dei concetti e delle astrazioni appena citati nella modellazione e progettazione del software non rappresenta semplicemente un'evoluzione dei modelli e delle tecnologie attuali bensì una vera rivoluzione o, secondo la già citata definizione data da Kuhn, un cambiamento scientifico di paradigma che influenzerà la maggior parte delle comunità di ricerca in maniera orizzontale e cambierà il modo in cui si concepiscono, modellano e costruiscono componenti e sistemi software.

2.1. Localizzazione.

I sistemi informatici odierni sono localizzati. Hanno, cioè, una nozione esplicita dell'ambiente in cui i componenti vengono allocati ed eseguiti, sono influenzati nella loro esecuzione dalle caratteristiche ambientali, e spesso i loro componenti interagiscono esplicitamente con l'ambiente.
In realtà i sistemi software non hanno mai lavorato in modo isolato, ma sono sempre stati e saranno sempre immersi in una qualche sorta di ambiente. Tuttavia, gli

approcci tradizionali di modellazione e ingegnerizzazione hanno sempre cercato di mascherare la presenza dell'ambiente. Nella maggior parte dei casi, specifici oggetti e componenti "mascherano" l'ambiente per modellarlo in termini di un "normale" componente, in modo che l'ambiente di per sé non esista come astrazione primaria. Purtroppo, l'ambiente in cui i componenti vivono e con cui essi interagiscono in realtà esiste e può influenzare l'esecuzione e la modellazione di un'applicazione:

• Diverse entità con le quali i componenti software avrebbero bisogno di interagire sono troppo complesse dal punto di vista strutturale e comportamentale per consentire un mascheramento banale.

• Per un sistema software, il cui obiettivo esplicito è quello di monitorare e controllare un ambiente logico o fisico, mascherare tale ambiente non è un scelta naturale. Al contrario, fornire una consapevolezza esplicita può diventare un obiettivo primario.

• L'ambiente può avere le proprie dinamiche, indipendentemente da quelle intrinseche ad un sistema software. Mascherare l'ambiente con un'entità di un'applicazione introduce inoltre forme di non-determinismo imprevedibili all'interno delle applicazioni.

Per le ragioni sopra esposte, la tendenza attuale sia nell'informatica che nell' ingegneria del software è quella di definire esplicitamente l'ambiente in cui un sistema software viene eseguito come una astrazione primaria, definendo in modo esplicito, sia i "confini" che separano i sistemi software e il loro ambiente, che le influenze reciproche tra i due sistemi [3]. Questo approccio evita strane pratiche di mascheramento necessarie per modellare ogni componente del mondo esterno come un' entità interna ad un'applicazione e consente anche ad un sistema software di far fronte in modo più naturale all'ambiente del mondo reale che il sistema software stesso potrebbe essere destinato a monitorare e controllare. Inoltre, la modellazione esplicita dell'ambiente e delle sue attività permette di identificare e limitare chiaramente le fonti di dinamicità e imprevedibilità, consentendo di concentrarsi sui componenti software come entità deterministiche che hanno a che fare con un ambiente dinamico ed eventualmente imprevedibile.

2.2. Apertura.

Vivere in un ambiente, percepirlo, ed essere influenzati da esso, intrinsecamente implica una sorta di apertura di un sistema software. Infatti, un sistema non può più essere concepito come un mondo isolato, ma deve invece essere considerato come un sub-sistema permeabile, i cui confini consentono reciproci effetti collaterali. In diversi casi, per raggiungere i loro obiettivi, i sistemi software devono interagire con componenti software esterni, per fornire loro servizi o dati, o per acquisirli. Più in generale, differenti sistemi software, progettati e modellati in modo indipendente, possono "vivere" nello stesso ambiente e interagire in modo esplicito gli uni con gli altri. Queste forme di interazione aperte richiedono ontologie comuni, protocolli di comunicazione e infrastrutture intermedie adeguate, per consentire l'interoperabilità.

Tuttavia, questa è solo una piccola parte del problema, e la sola interoperabilità non è sufficiente quando i sistemi software possono nascere e morire, in un ambiente, indipendentemente l'uno dall'altro, in modo molto dinamico, o quando un sistema software (o i suoi componenti) può muoversi esplicitamente attraverso differenti ambienti nel corso della sua vita. Queste caratteristiche introducono ulteriori problemi:

• Quando un componente interagisce con altri componenti in altri sistemi software, e quando i componenti si muovono attraverso ambienti diversi, può diventare difficile se non impossibile identificare chiaramente il sistema al quale un componente appartiene. In altre parole, diventa difficile individuare i confini dei sistemi software in modo chiaro.

• Quando un componente nasce in un ambiente (o viene spostato in uno specifico ambiente), si deve in qualche modo metterlo a conoscenza del suo contesto ambientale, di quali altri componenti sono adatti all'interazione, e in che modo può interagire con il nuovo sistema in cui è entrato senza mettere in pericolo sia il sistema che se stesso.

• Rendere capaci i componenti di entrare e lasciare un ambiente in modo completamente libero e di interagire l'uno con l'altro può rendere molto difficile capire e controllare il comportamento globale dei sistemi software, anche di uno solo.

A causa dei problemi di cui sopra, lo sviluppo del software potrebbe richiedere non solo di far fronte al problema di modellazione dell'ambiente in cui i sistemi sono in esecuzione, ma anche di comprendere e modellare i cambiamenti dinamici nella struttura del sistema. Inoltre, la necessità di preservare la coerenza del sistema nonostante l'apertura potrebbe richiedere l'identificazione e la messa in opera di specifiche politiche, atte a sostenere la riorganizzazione del sistema software in risposta ai cambiamenti nella sua struttura, o di adottare leggi contestuali sull'attività dei componenti che entrano nel sistema. L'obiettivo generale è aumentare la capacità di controllare l'esecuzione del sistema, nonostante i suoi dinamici cambiamenti strutturali.

2.3. Controllo locale.

Il concetto di "flusso di controllo" è sempre stato uno degli aspetti chiave dell' informatica e dell' ingegneria del software a tutti i livelli, da quello hardware fino alla progettazione di alto livello delle applicazioni. Tuttavia, quando i sistemi di software e i componenti vivono e interagiscono in un mondo aperto, il concetto di flusso di controllo diventa privo di significato.

Sistemi software indipendenti hanno propri ed autonomi flussi di controllo, e la loro reciproca interazione non implica alcuna unione di questi flussi. Pertanto, la modellazione e la progettazione di software aperti non solo rende il concetto di "sistema software" piuttosto vago, come discusso precedentemente, ma fa anche scomparire il concetto di "controllo di esecuzione globale". Questa tendenza è aggravata dal fatto che ogni sistema indipendente non solo ha il proprio flusso di

controllo, ma può anche avere componenti autonomi. Infatti, la maggior parte dei componenti dei sistemi software odierni sono entità attive (come oggetti attivi con flussi di controllo interni, processi, demoni) piuttosto che passive (come "normali" oggetti, funzioni, ecc.).
Anche se la programmazione concorrente e parallela sono dei paradigmi ben definiti, e le applicazioni con più threads e processi pratiche consolidate, tuttavia la tradizionale programmazione concorrente e parallela sfrutta, al fine di migliorare l'efficienza e le prestazioni, flussi multipli di esecuzione, evitando però di renderli threads multipli di controllo. In realtà, la maggior parte degli approcci tradizionali limita quanto più possibile l'autonomia di tali flussi multipli di esecuzione, tramite la sincronizzazione e rigorosi meccanismi di coordinamento, con l'obiettivo di preservare il determinismo e il controllo sull'esecuzione di un'applicazione.
Tuttavia, l'odierna autonomia dei componenti di un'applicazione ha motivazioni differenti e deve essere gestita in modo diverso:
• In un mondo aperto, l'autonomia di esecuzione consente ad un componente di muoversi attraverso sistemi e ambienti senza dover fare rapporto (o attendere la conferma) alla sua originale applicazione.
• Quando i componenti e i sistemi sono immersi in un ambiente altamente dinamico la cui evoluzione deve essere monitorata e controllata, un componente autonomo può di fatto occuparsi dell'ambiente (o di una parte) indipendentemente dal flusso globale di controllo.
• Diversi sistemi software non sono costituiti solamente da componenti software, ma integrano anche sistemi basati su computers, che sono per loro stessa natura sistemi autonomi, e possono quindi essere modellati di conseguenza.
• Con l'aumento della dimensione di un sistema software, la necessità di delegare il controllo a componenti non è più solo una questione di prestazioni, ma diventa una questione di semplicità concettuale. Infatti, coordinare un flusso globale di controllo tra un gran numero di componenti può diventare impossibile, e l'autonomia può diventare una ulteriore dimensione della modularità [4].

2.4. Interazione locale.

Direttamente derivante dalle tre questioni sopra esposte, il concetto di "interazioni locali in un mondo globale" è sempre più pervasivo nei sistemi software odierni.
Finora, si è delineato uno scenario in cui i componenti di un sistema software sono immersi in uno specifico ambiente, vengono eseguiti nel contesto di uno specifico (sub)sistema, e sono delegati ad eseguire un qualche compito in modo autonomo. Presi tutti insieme, questi aspetti portano naturalmente ad un rigoroso rafforzamento della località delle interazioni. Infatti:
• Componenti autonomi possono interagire con l'ambiente in cui sono immersi, percependolo ed influenzandolo. Se l'ambiente è fisico, la porzione del mondo fisico che un singolo componente è in grado di percepire e di influenzare è limitata localmente da leggi fisiche. Se l'ambiente è logico, la minimizzazione della

complessità concettuale e di gestione favorisce ancora la sua modellazione in termini locali e la limitazione dell'effetto di un singolo componente sull'ambiente.

• I componenti possono interagire normalmente l'uno con l'altro nel contesto del sistema software cui essi appartengono, che è, a livello locale il loro sistema. In un mondo aperto, tuttavia, un componente di un sistema può anche interagire con (componenti di) altri sistemi. In questi casi, la minimizzazione della complessità concettuale suggerisce di modellare un componente, come se si fosse temporaneamente "spostato" in un altro contesto, e in cui esso interagisce a livello locale [5].

• In un mondo aperto, i componenti hanno bisogno di una qualche forma di consapevolezza del contesto per un'esecuzione e un'interazione efficace. Tuttavia, per rendere un componente in grado di venire a conoscenza del suo contesto in modo efficace (ed efficiente), tale contesto deve essere necessariamente limitato a livello locale.

La località nelle interazioni è un requisito impellente quando il numero dei componenti di un sistema aumenta, o quando aumenta il grado di distribuzione. La presenza di un flusso autonomo di controllo può rendere il monitoraggio ed il controllo di interazioni concorrenti ed autonome ben più difficile che nelle applicazioni basate su oggetti e componenti, anche se tali interazioni sono strettamente locali.

3. Paradigma di programmazione.

Il termine “paradigma di programmazione” è estremamente sfumato perché è spesso usato con riferimento ad un insieme di diversi aspetti relativi al software in particolari circostanze. Tali aspetti sono spesso localizzati a differenti livelli di astrazione e la correlazione tra di essi è raramente formulata esplicitamente. Per descrivere i livelli di astrazione che compongono un paradigma di programmazione si può fare riferimento alla figura che segue tratta da [6].

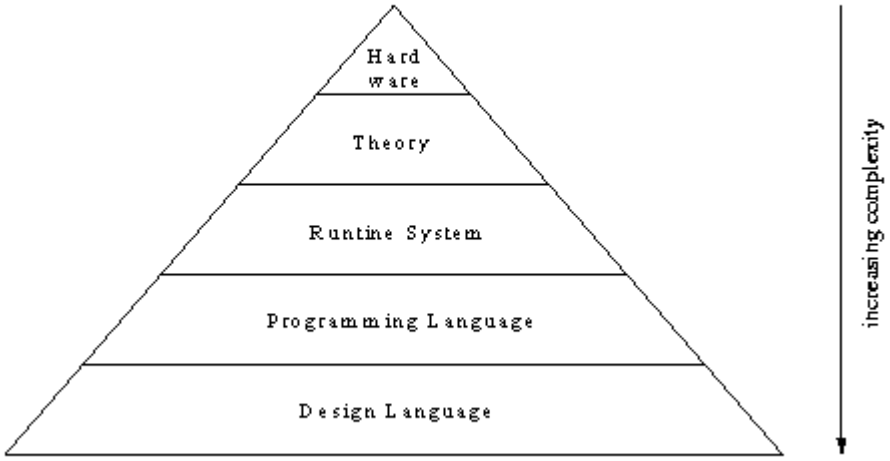

Figura 1

Nella Figura 1 Lind utilizza una forma triangolare per esprimere il fatto che il numero di concetti (e dunque la complessità) di un particolare livello di astrazione cresce nei livelli più alti. Inoltre l’approccio a livelli è abbastanza comune nella teoria informatica per separare chiaramente i concetti tra i diversi livelli di astrazione. Il vantaggio principale dell’approccio a livelli è che non è necessario conoscere i livelli inferiori per capire e lavorare con i concetti di un livello superiore perché, in teoria, ogni livello di astrazione rappresenta una struttura concettualmente chiusa. In realtà, però, le teorie relative ai livelli più alti sono, non solo molto più complesse di quelle a livelli inferiori, ma spesso anche incomplete. Perciò è spesso necessario combinare diverse teorie dei livelli più alti per ottenere una copertura completa della parte che si vuole modellare.
Si noti, inoltre, che le distinzioni tra i differenti livelli non sono eccessivamente marcate. Poiché infatti (secondo la tesi di Church) si assume che la maggior parte dei modelli di programmazione siano equivalenti dal punto di vista della potenza computazionale, qualsiasi modello di programmazione può essere implementato rispetto a qualsiasi altro modello. E’ così possibile scrivere un software orientato agli oggetti in un linguaggio di programmazione puramente imperativo o implementare un database deduttivo attraverso la programmazione orientata agli oggetti.

3.1. Livello hardware.

Il primo livello di astrazione incapsula l'architettura che è implementata attraverso l'hardware di un computer. Al giorno d'oggi molti computers sono basati ancora sull'architettura di Von Neumann introdotta negli ultimi anni '40. Tale architettura è composta da un processore, suddiviso in unità per l'elaborazione ed il controllo, ed una memoria che contiene le istruzioni e i dati di un programma.
Quest'architettura è ancora comune nei computer moderni anche se è stata decisamente ottimizzata utilizzando tecniche come pipelining, caching o parallelismo per velocizzare l'elaborazione. Una tendenza recente è quella di abbandonare i sistemi integrati su larga scala a favore di reti di comuni personal computers che lavorano insieme per scopi che richiedono un'alta capacità computazionale. Questi super-computers virtuali combinano il vantaggio del basso costo, grazie all'utilizzo di hardware convenzionale, con quello di un'estrema scalabilità che consente di aggiungere più risorse computazionali ogni qual volta ciò è necessario. In futuro l'intera rete Internet potrebbe diventare un super-computer virtuale che renderebbe il potere computazionale individuale obsoleto.
Tuttavia, sebbene sequenziale, parallelo o distribuito, dal punto di vista di un paradigma di programmazione, tutto l'hardware è visto allo stesso modo. Ci sono stati tentativi di costruire architetture hardware che inglobassero direttamente nell'hardware particolari paradigmi di programmazione, ma nessuno di questi ha avuto successo. Si può perciò sicuramente affermare che tutti i paradigmi di programmazione condividono la stessa base.

3.2. Livello delle teorie.

Al livello di astrazione superiore, tuttavia, le cose cambiano. Le teorie sono concettualizzazioni di un particolare modello computazionale che astrae dalle caratteristiche dell'hardware. Le prime teorie miravano a catturare la teorica capacità di un dispositivo computazionale per cercare di capire quali problemi di carattere generale fosse stato possibile risolvere automaticamente e quali no. La teoria di Turing, ad esempio, è una radicale concettualizzazione matematica dell'architettura di Von Neumann che consente di analizzare formalmente tutti i programmi che possono essere eseguiti su tale architettura. Altre teorie computazionali sono viste come strumenti per aiutare il programmatore ad esprimere l'idea di cosa un programma possa fare in modo naturale. Una delle prime teorie computazionali intesa come il fondamento di un modo "naturale" di programmare è il linguaggio dichiarativo ma è stato dimostrato da indagini nella psicologia cognitiva che tale rivendicazione non è necessariamente vera [7].

3.3. Livello del sistema di runtime.

Il sistema di runtime di un particolare paradigma di programmazione fornisce l'ambiente per l'interpretazione di un programma; tali ambienti possono essere radicalmente diversi. Nelle forme più semplici, questi si limitano a compiti amministrativi, come la gestione dell'heap, ma possono anche fornire servizi ben più elaborati come ad esempio quello di garbage collection. Tuttavia esistono anche ambienti di runtime molto complessi che forniscono motori di ragionamento completi per la programmazione logica, usati per esempio nei linguaggi di programmazione dichiarativa come il Prolog.

3.4. Livello del linguaggio di programmazione.

A questo livello di astrazione viene definita la struttura sintattica per la manipolazione delle entità del livello di runtime. I programmi scritti con un particolare linguaggio di programmazione sono poi direttamente interpretati dal sistema di runtime, o compilati in un formato intermedio congeniale al sistema di runtime o direttamente in codice assembler.
I costrutti sintattici forniti da un linguaggio di programmazione dovrebbero consentire al programmatore di usare i sottostanti concetti semantici in modo efficiente e di esprimere elegantemente le funzionalità desiderate di un programma. E' possibile, tuttavia, implementare un particolare modello concettuale con qualsiasi linguaggio di uso generale, come per esempio è possibile scrivere programmi orientati agli oggetti in C; ma per un programmatore è molto più semplice e comodo poter usare direttamente i termini di una struttura concettuale. Persino l'integrazione di differenti modelli concettuali in un singolo linguaggio di programmazione di alto livello può essere problematica poiché spesso è difficile trovare la giusta combinazione di concetti che non sia eccessiva per l'utente medio e al tempo stesso trovare una concisa rappresentazione sintattica per tali concetti.

3.5. Livello del linguaggio di progettazione.

I linguaggi di progettazione sono ulteriori astrazioni, a partire da un particolare linguaggio, il cui scopo è la modellazione concettuale di un sistema ad un livello più alto.
Tali linguaggi spesso utilizzano notazioni grafiche che semplificano l'accesso del progettista all'intera struttura di un sistema. Attualmente il linguaggio di progettazione più conosciuto è probabilmente UML (Unified Modeling Language), che cerca di integrare diverse notazioni progettuali prima separate. Proprio a causa della sua natura generale UML non è sempre adatto a qualsiasi campo di lavoro e, perciò, sono state proposte diverse estensioni che coprono casi particolari.

4. Paradigma di programmazione orientato agli agenti (AOP) VS paradigma di programmazione orientato agli oggetti (OOP).

Dopo aver esposto (Capitolo 3) gli aspetti caratterizzanti un paradigma di programmazione, si prosegue, sulla scia di quanto detto nei capitoli precedenti, andando ad indagare più in dettaglio, ai diversi livelli di astrazione, le differenze tra il paradigma di programmazione orientato agli agenti e quello orientato agli oggetti. Si metterà così, ancora di più, in luce l'evidenza di un salto di paradigma, tanto necessario quanto in atto.
Si ricordi, inoltre, che, come già detto, il livello hardware non incide in maniera significativa sui differenti paradigmi di programmazione.

4.1. Livello delle teorie.

Si comincia quindi il confronto tra il paradigma di programmazione orientato agli oggetti e quello orientato agli agenti con le entità che vengono gestite a questo livello di astrazione. Nel mondo orientato agli oggetti, queste entità sono gli oggetti. Un oggetto può essere qualsiasi cosa che va da un'entità concreta del mondo reale a un'entità concettuale che esiste solo nella testa del progettista. Ogni oggetto all'interno del sistema è associato ad una particolare classe che determina le proprietà fondamentali di ciascun oggetto. Le classi possono essere collegate tra di loro in diversi modi. Probabilmente il più noto rapporto tra classi è l'ereditarietà che modella un'estensione concettuale di una comune specifica di base. Durante la loro vita, gli oggetti comunicano inviando messaggi gli uni agli altri. Questi messaggi possono essere utilizzati per richiedere servizi all'oggetto ricevente, come ad esempio fornire informazioni interne o cambiare lo stato attuale. Sebbene ci siano molti concetti aggiuntivi nel paradigma di programmazione orientato agli oggetti in questo caso quanto detto è sufficiente a caratterizzare le più grandi differenze con il paradigma di programmazione orientato agli agenti. In sintesi, la raccolta dei concetti orientati agli oggetti è chiara e gestibile in termini di dimensioni e non varia molto tra i diversi approcci orientati agli oggetti.
Nell'universo orientato agli agenti, d'altra parte, ci troviamo di fronte ad un primo grave problema in quanto non esiste una definizione univoca delle entità che vengono trattate. Le teorie esistenti sugli agenti sono più o meno tutte basate su uno dei due concetti di agenzia ampiamente accettati. Nella *nozione forte* di agenzia, un agente è modellato in termini di nozioni mentali come credenze, desideri e intenzioni. Inoltre, la nozione forte richiede che questi concetti mentali abbiano una rappresentazione esplicita nella realizzazione di un agente. Questa nozione, quindi, impone per l'agente una visione di tipo white-box. La *nozione debole* di agenzia, d'altra parte, richiede per l'agente soltanto una visione di tipo black-box, in quanto definisce un agente solo in termini di proprietà osservabili. Secondo questa definizione, un agente è qualsiasi cosa che esibisce autonomia, reattività, proattività, attività sociale [8].

Secondo Lind [6] queste due nozioni di agenzia sono entrambe troppo rigide. Egli sostiene infatti la necessità di una definizione di agenzia più pragmatica che consenta al progettista di decidere quello che dovrebbe essere un agente a prescindere da una particolare implementazione o un minimo grado di proprietà esterne. Questo concetto viene definito "*nozione molto debole di agenzia*". Per supportare la sua tesi Lind si rifà ad un famoso articolo degli albori dell'intelligenza artificiale [9] in cui McCarthy afferma che è utile attribuire qualità mentali come credenze, scopi, desideri, ecc., alle macchine (o ai programmi) ogni qual volta ciò aiuta a comprendere la struttura di una macchina o di un programma. Inoltre non si impone alcun vincolo, come ad esempio la complessità minima richiesta dalle entità cui si vogliono attribuire categorie mentali o sulle categorie mentali che si desidera utilizzare. A suo giudizio, attribuire qualità mentali è un mezzo di comprensione e di comunicazione tra gli esseri umani, cioè è uno strumento puramente concettuale che ha lo scopo di esprimere le conoscenze esistenti su un determinato programma o il suo stato attuale.
Una conclusione più generale derivante dal lavoro di McCarthy è l'idea che qualsiasi cosa può essere un agente. Si è dibattuto a lungo su tale visione [8] da diversi punti di vista. Sulla scia di McCarthy, Lind [6] sostiene che l'integrità concettuale raggiunta vedendo qualsiasi entità - per semplice che sia - come un agente conduce ad un sistema di progettazione più chiaro e previene il problema di decidere se una particolare entità è un agente o no.
Dopo aver identificato gli elementi strutturali di base, e quindi le differenze, delle teorie orientate agli oggetti o agli agenti si possono quindi evidenziare anche alcune analogie tra i due approcci.
Come detto in precedenza, l'elemento descrittivo di base della programmazione orientata agli oggetti è la classe. La definizione di una classe specifica le variabili di classe di un oggetto ed i metodi che l'oggetto accetta. Le classi possono essere collegate tra di loro in maniera diversa: una classe eredita da un'altra in modo tale che la nuova classe è un'estensione della classe esistente, le istanze di due classi possono collaborare tra di loro scambiando messaggi e, infine, possono avere una connessione strutturale nel senso che un'istanza di una classe può contenere un'istanza di un'altra classe.
Questi concetti si ritrovano nel mondo orientato agli agenti sostituendo la classe con il *ruolo*, le variabili di stato con le credenze/conoscenze e i metodi con i messaggi. Quindi la definizione di un ruolo descrive le capacità dell'agente, i dati che sono necessari per produrre i risultati desiderati e le richieste che attivano un particolare servizio. Oltre a questa fondamentale relazione, vi sono molte altre similitudini concettuali tra l'orientamento agli oggetti ed quello agli agenti che possono essere mappate le une sulle altre, come riassunto nella Tabella 1, tratta da [6].

	OOP	AOP
Structural Elements		
	abstract class	generic role
	class	domain specific role
	class variables	knowledge, belief
	methods	capabilities
Relations		
	collaboration (uses)	negotiation
	composition (has)	holonic agents
	inheritance (is)	role multiplicity
	instantiation	domain-specific role + individual knowledge
	polymorphism	service matchmaking

Tabella 1

4.2. Livello del sistema di runtime.

Gli oggetti e gli agenti e le varie relazioni che esistono tra di loro nei rispettivi modelli di programmazione sono astrazioni concettuali che richiedono una implementazione per far si che possano essere utilizzati dai più alti livelli di astrazione. Nei paragrafi seguenti, si distingue tra l'implementazione dei concetti teorici nelle entità stesse e l'implementazione di un meta-livello che manipola le entità di base.

In un sistema di runtime orientato agli oggetti, gli oggetti sono rappresentati staticamente dall'architettura dell'oggetto. Tale architettura è di solito abbastanza semplice in quanto contiene soltanto lo stato corrente dell'oggetto e la relazione con la classe degli oggetti (che determina le operazioni che possono essere effettuate su un oggetto). Un oggetto è di solito rappresentato come una collezione arbitraria di dati con le corrispondenti funzioni; la granularità degli oggetti non è in teoria limitata. Tuttavia, le questioni di efficienza impongono che non tutte le entità vengano modellate come oggetti e quindi, in realtà, il beneficio concettuale è leggermente indebolito. Il sistema di gestione degli oggetti è responsabile della rappresentazione delle relazioni, come ad esempio l'ereditarietà tra le classi, e della manipolazione di oggetti, come nel caso della creazione o della distruzione degli oggetti stessi. Inoltre, il sistema di gestione degli oggetti è anche responsabile di aspetti dinamici, quali il metodo di selezione di oggetti polimorfi, la gestione delle eccezioni o il garbage collection.

In un sistema di runtime orientato agli agenti, le cose sono decisamente più complicate, anche se simili nella loro struttura generale. Le entità di base sono gli agenti che sono implementati attraverso la corrispondente architettura. Le architetture ad agenti sono spesso basate su una particolare teoria, come ad esempio BDI [10], e stabiliscono il collegamento tra il concetto astratto di agente e il "ruolo", nel senso

che forniscono al sistema di runtime le descrizioni dei ruoli che costituiscono gli agenti.
Tuttavia, le architetture ad agenti sono di gran lunga più complesse di quelle ad oggetti, soprattutto a causa degli aspetti dinamici che devono essere affrontati. La ricchezza del mondo orientato agli agenti ha prodotto un gran numero di differenti architetture ad agenti. A causa del gran numero di approcci, è impossibile individuare la migliore o la più generale architettura. Tuttavia, il minimo comun denominatore sembra essere il ciclo base *percepire - ragionare - agire* [11]: ad ogni iterazione, l'agente percepisce lo stato del proprio ambiente, integra la percezione nella sua base di conoscenza che viene poi utilizzata per derivare l'azione successiva da eseguire. Questo ciclo generico è un'utile astrazione in quanto fornisce un vista di tipo black-box sull'architettura ad agenti e racchiude in sé aspetti specifici.
Il compito del sistema di gestione degli agenti, come meta-livello di un ambiente di runtime basato sugli agenti, è quello di fornire agli agenti uno "spazio di vita", vale a dire un insieme di meccanismi che consenta agli agenti di entrare in contatto l'uno con l'altro. Per Consentire agli agenti di diversi progettisti di interagire, è necessario uniformare i servizi di base forniti dal sistema di gestione degli agenti. Uno di questi standard è, ad esempio, FIPA.

4.3. Livello del linguaggio di programmazione.

Secondo Lind [6] l'orientamento agli oggetti così come l'orientamento agli agenti sono concetti così generali che possono essere collegati a qualsiasi altro linguaggio di programmazione. Nel caso degli oggetti, questo approccio funzionò per linguaggi come il C, portando a C++, Cobol (ObjectCobol), Perl e numerosi altri linguaggi. Ma non solo i linguaggi imperativi sono stati migliorati con gli oggetti. Il sistema di programmazione Mozart, per esempio, offre una combinazione molto elegante di programmazione con vincoli logici e di concetti orientati agli oggetti.
Nel contesto dell'ingegneria del software orientata agli agenti, queste tendenze non sono così chiare fino ad ora. Attualmente, non vi è - almeno secondo Lind - un linguaggio di programmazione orientato agli agenti ampiamente accettato che va al di là dello stato sperimentale. Tuttavia, alcuni approcci sono stati concepiti come un'estensione di linguaggi consolidati, come ad esempio JAM Agents o Jason che combinano concetti orientati agli agenti con Java.

4.4. Livello del linguaggio di progettazione.

Nella comunità orientata agli oggetti UML è un linguaggio di progettazione ben definito sostenuto da strumenti "*case*" come Rational Rose, che può trasformare l'architettura di un oggetto nel codice di una classe in Java o C + +, insieme ad altri utili programmi di supporto alla progettazione.

Nel mondo orientato agli agenti - anche se è un settore relativamente nuovo - esiste già un gran numero di diversi frameworks. Ciò può essere dovuto al fatto che la crescente complessità del software può essere affrontata solo mediante il sostegno di strumenti adeguati. Sebbene non esista alcun linguaggio di progettazione uniformemente accettato, principalmente a causa della mancanza di un ben definito paradigma di programmazione orientato agli agenti [12], tuttavia, c'è un gran numero di strumenti di progettazione per specifiche architetture e piattaforme orientate agli agenti. Gli esempi esistenti di linguaggi di progettazione orientati agli agenti vanno da frameworks di basso livello come SIF e Swarm fino a strumenti più complessi e potenti, come ZEUS della British Telecom, che fornisce meccanismi di drag-and-drop per assemblare sistemi multiagente.
E' in fase di sviluppo, inoltre, AgentUML, incentivato da OMG/FIPA, che propone un'estensione di UML verso i concetti orientati agli agenti. Una delle principali estensioni riguarda un nuovo particolare tipo di diagrammi, i diagrammi di protocollo. Tali diagrammi combinano elementi dei diagrammi UML di interazione e di stato per modellare i ruoli che possono essere assunti da un agente nel corso dell'interazione con altri agenti. Questo nuovo tipo di diagramma consente di specificare più flussi all'interno di un protocollo di interazione e supporta l'annidamento dei protocolli e modelli di protocolli basti su una generica descrizione.

5. Sinossi dell'evoluzione dei linguaggi di programmazione.

Dopo aver descritto le principali differenze e analogie tra i paradigmi di programmazione orientati agli oggetti rispetto agli agenti è utile presentare una breve descrizione dell'evoluzione dei linguaggi di programmazione. La Figura 2, tratta da [13], fornisce appunto una visione d'insieme dell'evoluzione dei linguaggi di programmazione.

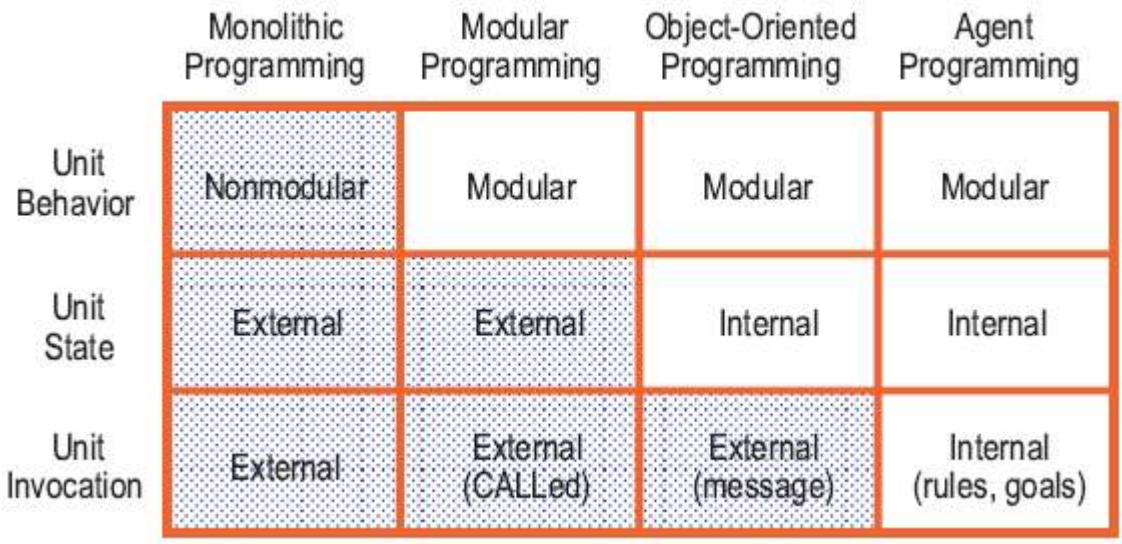

Figura 2

Secondo il modello evolutivo proposto da Odell, in origine, l'unità di base del software era costituita dall'intero programma sul quale il programmatore aveva pieno controllo. Lo stato del programma era di responsabilità del programmatore e la sua invocazione era determinata da un operatore del sistema. Il termine modulare non aveva senso in quanto il programma non poteva essere visto come un'unità riutilizzabile in varie circostanze.
Con l'aumentare della complessità dei programmi e dello spazio utilizzabile dalla memoria, tra i programmatori sorse la necessità di introdurre un certo grado di organizzazione nel loro codice. L' approccio modulare alla programmazione impiegava unità di codice più piccole che avrebbero potuto essere riutilizzate in determinate e diversificate circostanze. In questo caso, i cicli strutturati e le subroutines venivano progettate per avere un alto grado di integrità locale. Ogni subroutine di codice era incapsulata, mentre il suo stato era determinato da argomenti forniti dall'esterno, e il controllo veniva acquisito soltanto dopo una invocazione esterna con una CALL. Questa era l'epoca delle procedure come unità principale della decomposizione.
Al contrario, l'orientamento agli oggetti si è aggiunto all'approccio modulare, mantenendo all'interno di un oggetto i suoi segmenti di codice (o metodi), nonché fornendo all'oggetto stesso il controllo locale sulle variabili manipolate attraverso i

propri metodi. Tuttavia, nella tradizionale programmazione orientata agli oggetti, gli oggetti sono considerati passivi perché i loro metodi vengono invocati solo quando qualche entità esterna invia loro un messaggio.
Gli agenti software hanno un loro flusso di controllo, localizzando così non solo il codice e lo stato ma anche la loro invocazione. Tali agenti possono anche avere delle regole e degli scopi individuali, il che li fa apparire come "oggetti attivi dotati di iniziativa". In altre parole, quando e come un agente agisce è stabilito dall'agente stesso.
Gli agenti sono comunemente considerati entità *autonome*, perché possono prendere in considerazione un loro insieme di responsabilità interne. Inoltre, gli agenti sono entità *interattive* in grado di utilizzare ricche forme di messaggi. Tali messaggi possono supportare l'invocazione di metodi, così come informare gli agenti di particolari eventi, chiedere qualcosa agli agenti o ricevere una risposta a una precedente richiesta. Infine, poiché gli agenti sono autonomi possono avviare un'interazione e rispondere ad un messaggio nel modo che preferiscono. In altre parole, si può pensare agli agenti come oggetti che possono dire "No", nonché "Vai". Grazie alla natura interattiva e autonoma degli agenti, l'interazione richiesta per lanciare fisicamente un'applicazione è poca o nulla. Van Parunak riassume bene il concetto: "In ultima analisi, lo sviluppatore di un'applicazione identifica semplicemente gli agenti che desidera nell'applicazione finale, e gli agenti si organizzano da soli per eseguire le funzionalità richieste" [14]. Non è necessario alcun flusso di controllo centralizzato o alcuna organizzazione top-down in quanto i sistemi ad agenti possono organizzarsi da soli.

6. Il rapporto tra linguaggi di programmazione e ingegneria del software.

Dopo aver analizzato differenze e somiglianze tra agenti ed oggetti ad ogni livello di astrazione (relativamente al paradigma di programmazione) e dopo aver presentato una breve descrizione dell'evoluzione dei linguaggi di programmazione è chiara l'importanza di stabilire un paradigma per la programmazione ad agenti e la progettazione di sistemi orientati agli agenti che debbono prendere in considerazione la diversità delle caratteristiche degli agenti rispetto agli oggetti ai livelli del sistema di runtime, dei linguaggi di programmazione e dei linguaggi di progettazione.
Come già detto, un linguaggio di programmazione è lo strumento essenziale per costruire software. Storicamente, come descritto nel paragrafo precedente, con l'evolvere dei linguaggi di programmazione cambiano anche gli approcci alla programmazione e, di conseguenza prendono piede nuovi metodi e nuove tecniche. Il risultato di questo processo è la definizione di nuovi approcci per l'ingegneria del software.

Programming languages, and Software Modelling Techniques

Programming Paradigms	Programming language	Analysis	Design
Top down (Monolithic)	Assembly High level language	Textual, Algorithms	Flowcharts Algorithms
Structural (Modular)	High level languages with built-in support routines	Dataflow diagram HIPO Chart	Data Structure Diagram and Structure Chart
Object Oriented	Object Oriented high-level languages, Object support libraries.	UML: Use Case & Collaboration Diagrams	UML: Class diagrams and its relations, State Machine…
Agent Oriented	Agent Platform, an Agent Oriented Language does not exist yet.	Under * construction	Under * construction

Tabella 2

La Tabella 2, tratta da [12] definisce il rapporto riflessivo tra l'evoluzione dell'ingegneria del software e gli sviluppi degli strumenti di implementazione (linguaggi di programmazione). A partire dalla programmazione top-down, passando per quella modulare (o strutturale), fino a quella orientata agli oggetti, l'ingegneria del software si è sviluppata in un senso o in un altro per venire incontro a linguaggi di programmazione che potessero essere compresi e applicati dai programmatori. In altre parole, l'ingegneria del software rispecchia i diversi paradigmi di

* Esistono, in realtà, diverse metodologie e tecniche di modellazione per l'AOSE (per esempio AUML, Gaia, MaSE, ecc.), ma sono, secondo Juneidi, troppo immature per essere utilizzate in applicazioni reali con agenti software.

programmazione, fondamentalmente perché fornisce modelli e tecniche per progettare applicazioni software sviluppate grazie a linguaggi di programmazione.
Attualmente, ancora una volta, a causa delle nuove caratteristiche portate dal concetto di agenzia, sta sorgendo un nuovo approccio allo sviluppo del software. Come sottolineato in precedenza, gli agenti devono essere concepiti in modo diverso dagli oggetti. Considerando, inoltre, le caratteristiche già citate relative agli agenti, ovvero, che hanno un proprio flusso di controllo, regole individuali e scopi precisi, gli agenti possono dunque essere visti come "oggetti attivi dotati di iniziativa" [13]. Tuttavia, gli oggetti vengono considerati relativamente passivi rispetto agli agenti, perché i loro metodi vengono invocati solo quando una qualche entità esterna invia un messaggio per agire. A causa di tutte queste differenze tra gli agenti e gli oggetti, è necessario un ulteriore passo avanti nell'ingegneria del software e nei linguaggi di programmazione per far si che gli agenti software possano essere introdotti e rappresentati nei sistemi software.
Come discusso nella sezione precedente ci sono sostanziali differenze tra gli oggetti e gli agenti in termini di astrazioni a tutti i livelli che influenzano un paradigma di programmazione. Le soluzioni proposte per integrare gli agenti in un'applicazione software che contenga entità assimilabili ad agenti e non, possono essere gestite, secondo Juneidi, attraverso due metodi [12]. Il *primo* metodo è quello di utilizzare un ambiente open source dedicato di una piattaforma commerciale ad agenti con un linguaggio orientato agli oggetti. Il ciclo generico di un agente che percepisce e agisce nell'ambiente è un utile astrazione in quanto fornisce una vista di tipo black-box sull'architettura di un agente e incapsula gli aspetti dinamici relativi alla fase di esecuzione degli agenti. A causa della ricchezza di ricerca e sviluppo di sistemi orientati agli agenti, esiste una gran numero di diverse architetture e piattaforme orientate agli agenti ognuna delle quali dotata di un diverso sistema di runtime. Come si afferma in [15], una piattaforma ad agenti è un ambiente software che fornisce strumenti e funzionalità atte a garantire il funzionamento degli agenti software. Alcune di queste sono [16]:
• Sofisticate infrastrutture di comunicazione (come ad esempio in JADE, FIPA-OS).
• Supporto per la mobilità degli agenti (come ad esempio in ADK, Voyager).
• Motore di ragionamento (come ad esempio in Jack, ABLE).
Una piattaforma ad agenti può anche fornire uno speciale linguaggio per la manipolazione di agenti interfacciati con linguaggi orientati agli oggetti come Java o C++. Il *secondo* metodo è implementare gli agenti da zero utilizzando un preesistente linguaggio di programmazione orientato agli oggetti con librerie di supporto costruite per gli agenti. In realtà, questo metodo non è così semplice come appare: è necessario infatti poter gestire le differenze tra agenti e oggetti allo stesso livello di astrazione (in particolare runtime e linguaggi di programmazione) di un paradigma di programmazione. Come evidenziato in precedenza, ciò che differenzia un agente da un oggetto è che l'agente possiede un proprio stato interno, un proprio ambiente, e propri protocolli di comunicazione con le altre entità software (utenti, oggetti, altri agenti), e soprattutto una propria rappresentazione della base di conoscenza e proprie regole per manipolarla; gli oggetti posseggono invece attributi e metodi rappresentati

da dati assoluti. I linguaggi standard esistenti orientati agli oggetti non possono quindi essere utilizzati per supportare l'implementazione di componenti ad agenti [13]. Le librerie proposte per gli agenti possono essere viste come contenitori di conoscenza che possono essere utilizzati dagli agenti per rappresentare le loro facoltà mentali e il loro comportamento; per ogni agente, inoltre, può essere creato un flusso di interazione, utilizzando i threads tipici dei linguaggi orientati agli oggetti, non appena un agente inizia ad operare (nascita di un agente); poi l'agente sarà costantemente in esecuzione come entità residente in memoria (vita di un agente); infine l'esecuzione (ruolo) non potrà essere disturbata o interrotta fino a quando non si verificherà un'interruzione ad un livello superiore (morte dell'agente).
Il primo metodo consente di sviluppare un software basato su un'architettura di "soli agenti" o un'applicazione che dia maggior enfasi alla teoria degli agenti. Questa è un'architettura appropriata per applicazioni basate sui MAS e sulla modellazione ABS (vedi in seguito). Ma usare il primo metodo in un'applicazione software generica non è una buona idea fondamentalmente perché gli sviluppatori ne avranno più restrizioni che aiuti [15]. La limitazione più grave è che lo sviluppatore dovrebbe "agentificare" tutti i componenti software per poterli manipolare con le stesse modalità. Il secondo metodo è più appropriato per applicazioni software di tipo generale in cui si hanno oggetti e agenti che cooperano per eseguire i compiti principali di un'applicazione software. Utilizzando questo metodo l'AOSE sarebbe il processo di ingegnerizzazione adatto per questo tipo di sviluppo del software, che comprende diversi tipi di entità: oggetti (non-agenti) e agenti.
Tuttavia per poter parlare dell'AOSE è doveroso premettere alcune considerazioni di carattere generale. Per prima cosa bisogna notare che in ogni processo di ingegnerizzazione il peso e la priorità maggiore vengono dati allo strumento di costruzione. Per esempio, se un architetto mostra progetti sorprendenti per una casa sull'acqua, tutto è inutile fino a quando non verranno forniti gli strumenti e le attrezzature per costruire una casa del genere. Lo stesso vale per l'ingegneria del software e gli strumenti di programmazione. Pertanto, qualsiasi metodologia di ingegneria del software per essere utile deve essere supportata da adeguati strumenti di programmazione. La maggior parte dei sistemi software generici può contenere sia oggetti che agenti, e gli oggetti non devono essere più di quello che realmente sono. D'altra parte, gli agenti perderebbero la maggior parte delle loro caratteristiche, se venissero trattati come un tipo particolare di oggetti. Nella maggior parte dei sistemi software comuni e generici si avrà quindi a che fare con entrambi: oggetti e agenti [12]; si va in sostanza verso la coesistenza di oggetti e agenti. L'AOSE dovrebbe avere la capacità di rappresentare, in un sistema, sia gli agenti che gli oggetti. Bisogna dunque costruire il più adatto strumento di implementazione (linguaggio di programmazione) che ha questo tipo di capacità: un linguaggio orientato agli oggetti (ad esempio C++ o Java) con librerie di supporto orientate agli agenti.

7. Verso l'AOSE.

Lo scopo dell' AOSE è quello di integrare gli agenti nei sistemi software. Questo campo di ricerca è relativamente nuovo ed è stato motivato da almeno due principali settori di ricerca quali l' ingegneria del software e l' intelligenza artificiale. Pertanto, è naturale avere argomenti di ricerca strettamente legati tra loro:
• Agent Oriented Software Engineering (AOSE),
• modellazione di Agent-Based Systems (ABS),
• modellazione di Multi-Agent Systems (MAS).
Per chiarire la portata di questi argomenti, bisogna fare un passo indietro, quando nacque l'ingegneria del software orientata agli oggetti (OOSE). L'intenzione era quella di modellare le applicazioni software usano il paradigma orientato agli oggetti (OO). Secondo l'approccio OO ogni entità è rappresentata da un oggetto. Come risultato, il software OO contiene solo oggetti che hanno propri metodi e attributi manipolati dal programmatore. In generale, l'approccio OO può essere utilizzato per la costruzione di qualsiasi applicazione software e così facendo qualsiasi entità può dunque essere "oggettivata". Per esempio, un'informazione o un documento possono essere considerati come degli oggetti. D'altra parte però, non è possibile individuare o proporre un simile approccio ("agentificazione") per l' Agent Oriented Software Engineering (AOSE), perché la maggior parte delle applicazioni software comuni e reali contengono entità di tipo "agente" e "non-agente". Inoltre, come evidenziato in [17] la tecnologia per i sistemi multi-agente (MAS) deriva dal campo di ricerca dell'intelligenza artificiale distribuita (DAI). La modellazione di MAS e ABS fornisce tecniche di modellazione per applicazioni software dal punto di vista dell'agente. Come detto in precedenza, le entità in tali sistemi sono "solo agenti" che interagiscono/si coordinano per raggiungere specifici obiettivi. Pertanto, le piattaforme ad agenti costruite all'interno di questo contesto sono le più adatte ad essere utilizzate come strumenti di implementazione per le tecniche di modellazione per MAS e ABS.

	AOSE	ABS & MAS modelling
Entities	"Agent", "non-agent"	Agent only
Programming Language	OO Standard language with Agent-support libraries	Agent platform languages
Software Application	Common, generic and traditional (*with agents added*)	Artificial intelligence, simulation and robotics

Tabella 3

La Tabella 3, tratta da [12], mostra le differenze tra l' AOSE e la modellazione di MAS e ABS, in termini di natura delle entità all'interno del sistema modellato, strumento di implementazione (linguaggio di programmazione) e natura delle applicazioni software.
Aree interessanti per le applicazioni MAS e ABS sono [18]:
• Risoluzione di problemi: riguarda tutte le situazioni in cui un agente software assolve compiti che sono di solito svolti da esseri umani.
• Simulazione multi-agente: L'uso frequente della simulazione viene utilizzato per cercare di spiegare o prevedere fenomeni naturali.
• Costruzione di mondi artificiali: Costruzione di mondi artificiali per capire l'influenza del comportamento su di una società regolamentata (mondo virtuale).
• Robotica collettiva: Creazione di non solo di un singolo robot, ma di un insieme di robots, che cooperano per portare a compimento una missione, evolvendosi attraverso l'interazione e l'adattamento.
• Progettazione di programmi: Progettare un software che vada al di là delle tecniche di elaborazione dei dati, ma dotato di agenti autonomi in grado di agire in un universo fisicamente distribuito.
In aggiunta, l'AOSE mira a fondere le caratteristiche di agenzia nell' ingegneria del software. Questo nuovo approccio deve rispettare i criteri dell' ingegneria del software ed i criteri di modellazione delle caratteristiche di un agente, come è indicato in [19]. Uno strumento di programmazione che sia conforme con l' approccio dell' AOSE può essere un linguaggio orientato agli oggetti che fornisce librerie di supporto per la costruzione di agenti, per manipolare agenti e oggetti, in modo tale da poter preservare le loro caratteristiche distintive ai livelli di astrazione del paradigma di programmazione (Figura 1). Riassumendo, quindi, la nuova generazione dell'ingegneria del software (AOSE) non è solo un'evoluzione dell' OOSE, ma rappresenta un punto di partenza per l'effettiva attuazione di un salto di paradigma, dagli oggetti agli agenti, dove però le due diverse tipologie di entità non sono mutuamente esclusive, ma possono e dovrebbero coesistere per poter sfruttare al meglio le potenzialità di ognuna.

Bibliografia

[1] T. S. Kuhn, *The Structure of Scientific Revolutions*, University of Chicago Press, 1962.

[2] F. Zambonelli, H. V. D. Parunak, *Towards a Paradigm Change in Computer Science and Software Engineering: A Synthesis*, The Knowledge Engineering Review, 2004.

[3] J. Odell, H. V. D. Parunak, M. Fleischer, S. Brueckner, *Modeling Agents and their Environment - Proceedings of the 3rd International Workshop on Agent-Oriented Software Engineering*, Lecture Notes in Computer Science, Springer Verlag (Berlin, D), Vol. 2585, pp. 16-31, 2003.

[4] H. V. D. Parunak, Go *to the Ant: Engineering Principles from Natural Agent Systems - Annals of Operations Research*, 75 pp. 69-101, 1997.

[5] G. Cabri, L. Leonardi, F. Zambonelli, *Engineering Mobile Agent Applications via Context-Dependent Coordination - IEEE Transactions on Software Engineering,* 28(9) pp. 1041-1057, 2002.

[6] J. Lind, *Issues in Agent–Oriented Software Engineering - The First International Workshop on Agent Oriented Software Engineering (AOSE 2000)*, 2001.

[7] T. Ormerod. *Human cognition and programming - In Psychology of Programming*, Academic Press Ltd., London, 1990.

[8] M. Wooldridge, N. R. Jennings, *Intelligent agents: Theory and practice*, The Knowledge Engineering Review, 10(2): pp. 115-152, 1995.

[9] J. McCarthy, *Ascribing mental qualities to machines* in M. Ringle, *Philosophical Aspects in Artificial Intelligence*, Harvester Press, 1979.

[10] A. S. Rao, M. Georgeff, *BDI Agents: from theory to practice* in *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pp. 312-319, San Francisco, CA, 1995.

[11] S. Russell, P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, 1995.

[12] S. J. Juneidi, *Toward programming paradigms for agent oriented software engineering*, IASTED Conf. on Software Engineering, pp. 428-432, 2004.

[13] J. Odell, *Object and Agents Compared ,* Journal of Object Technology Vol. 1, No. 1 , 2002.

[14] H. V. D. Parunak, *'Go to the Ant': Engineering Principles from Natural Agent Systems*, Annals of Operations Research, 75, pp. 69-101, 1997.

[15] A. J. N. van Breemen. *Integrating Agent in SoftwareApplication,* Agent Technology Workshops 2002, LNAI 2592, pp.343-354, Springer-Verlag, Berlin-Heidelberg, 2003.

[16] N. T. Giang, D. T. Tung, *Agent Platforms Evaluation and Comparison,* Pellucid 5FP IST-2001-34519, Ins of Inf, 2002.

[17] S. Green, L. Hurst, B. Nangle, P. Cunningham, F. Somers, R. Evans, *SoftwareAgents: A Review*, Technical report TCD-CS-1997-06, Trinity College, Dublin, 1997.

[18] J. Ferber, *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*, pp. 31-33, Addison Wesley Longman, 1999.

[19] F. Bergenti, O. Shehory , F. Zambonelli, *Agent-Oriented Software Engineering*, EASSS, Italy, 2002.