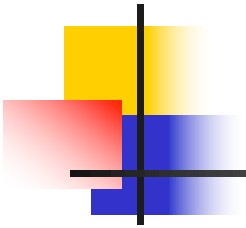
Corso di Laurea Specialistica in Ingegneria Informatica

Dagli oggetti agli agenti: verso un salto di paradigma

Corso di Sistemi Multi-Agente LS
Prof. Andrea Omicini
A. A. 2008-2009

Approfondimento di:
Salvatore Presta



Agenda

- Salto di paradigma
- Motivazioni/Evidenze
- Paradigma di programmazione
- OOP vs AOP
- Evoluzione dei linguaggi di programmazione
- Linguaggi di programmazione e IS
- AOSE

Paradigma

Definizione:

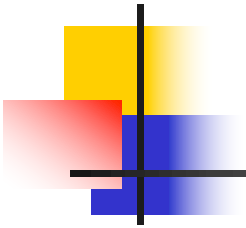
(tratta dal dizionario Zanichelli on-line)

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.

Salto di paradigma

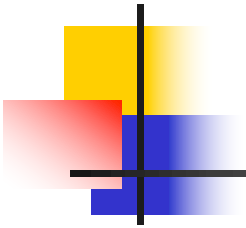
- Il primo a teorizzare un salto di paradigma fu, nel 1962, Thomas Kuhn [1].
- Secondo Kuhn il progresso scientifico non è un processo evolutivo, ma piuttosto una "serie di pacifici interludi interrotti da violente rivoluzioni intellettuali", e in queste rivoluzioni "una visione concettuale del mondo 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, che non accade, ma è piuttosto guidata da agenti di cambiamento.

Salto di paradigma nell'informatica e nell'IS



- La complessità introdotta dai moderni sistemi software può essere sicuramente vista come il principale agente di cambiamento.
- Lo scenario che delinea l'imminente crisi del software [2] sta emergendo rapidamente: i sistemi informatici saranno dappertutto, sempre connessi, sempre attivi e inglobati in qualsiasi oggetto.
- Tale scenario influenza i sistemi software non solo dal punto di vista quantitativo (numero di componenti e capacità richieste) ma anche da quello qualitativo.

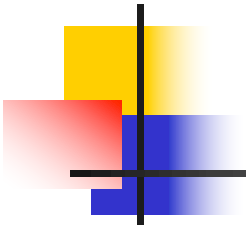
Crisi del software: impatto qualitativo



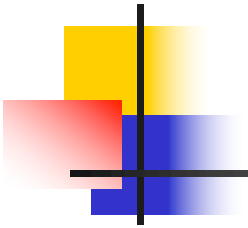
Quattro caratteristiche distinguono i futuri sistemi software da quelli tradizionali [2]:

- Localizzazione: i componenti software operano in un ambiente, che possono influenzare e dal quale possono essere influenzati.
- Apertura: i sistemi software sono soggetti ad una gestione decentralizzata e possono dinamicamente cambiare la loro struttura.
- Località nel controllo: i componenti di un sistema software rappresentano luoghi di controllo autonomi e proattivi.
- Località nelle interazioni: i componenti software interagiscono tra di loro secondo modelli di interazione locale.

Localizzazione [2]



- I sistemi informatici odierni hanno una nozione esplicita dell'ambiente.
- In realtà i sistemi software non hanno mai lavorato in modo isolato ma nella maggior parte dei casi l'ambiente veniva "mascherato".
- Il "mascheramento" però non è sempre la scelta giusta:
  - la complessità dei sistemi software moderni non consente un "mascheramento" banale;
  - se un sistema software deve monitorare o controllare un ambiente, "mascherare" l'ambiente non è una scelta naturale, anzi è necessaria una conoscenza esplicita;
  - poiché l'ambiente ha dinamiche proprie, indipendenti dal sistema software, "mascherare" l'ambiente introduce imprevedibili forme di non-determinismo all'interno delle applicazioni.
- E' necessario modellare l'ambiente come astrazione primaria.



Apertura [2]

- Un sistema non può più essere concepito come un mondo isolato, ma deve invece essere considerato come un sub-sistema permeabile.
- Poiché differenti e indipendenti sistemi software, possono "vivere" nello stesso ambiente e interagire è necessario garantire l'interoperabilità.
- Tuttavia la sola interoperabilità non è sufficiente quando i sistemi software possono nascere e morire, in un ambiente, o muoversi esplicitamente attraverso differenti ambienti:
  - difficile individuare i confini dei sistemi software;
  - bisogna garantire ad ogni componente la conoscenza dell'ambiente in cui esso nasce o si sposta;
  - molto difficile capire e controllare il comportamento globale dei sistemi software, quando i componenti possono liberamente entrare o lasciare un ambiente e interagire tra di loro

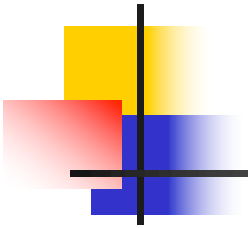
Località nel controllo [2]

- 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: scomparire il concetto di "controllo di esecuzione globale" .
- La tradizionale programmazione concorrente e parallela sfrutta flussi multipli di esecuzione, evitando però di renderli flussi multipli di controllo.
- Tuttavia, l'odierna autonomia dei componenti di un'applicazione ha motivazioni differenti e deve essere gestita in modo diverso:
  - l'autonomia di esecuzione consente ad un componente di muoversi attraverso sistemi e ambienti, indipendentemente dall'applicazione;
  - se un ambiente altamente dinamico deve essere monitorato e controllato, un componente autonomo può occcuprasi dell'ambiente (o di una parte) indipendentemente dal flusso globale di controllo;
  - 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: l'autonomia può diventare una ulteriore dimensione della modularità.

Località nelle interazioni [2]

- Il concetto di "interazioni locali in un mondo globale" deriva direttamente dalle tre questioni esposte precedentemente, che portano ad un rafforzamento della località delle interazioni:
  - dal fatto che componenti autonomi possono interagire con l'ambiente in cui sono immersi, percependolo ed influenzandolo discende la limitazione dell'effetto di un singolo componente sull'ambiente (fisico o logico);
  - poiché i componenti possono interagire con altri sistemi è utile modellare l'interazione tra componenti a livello locale;
  - 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.

Dimensioni di un salto di tecnologia



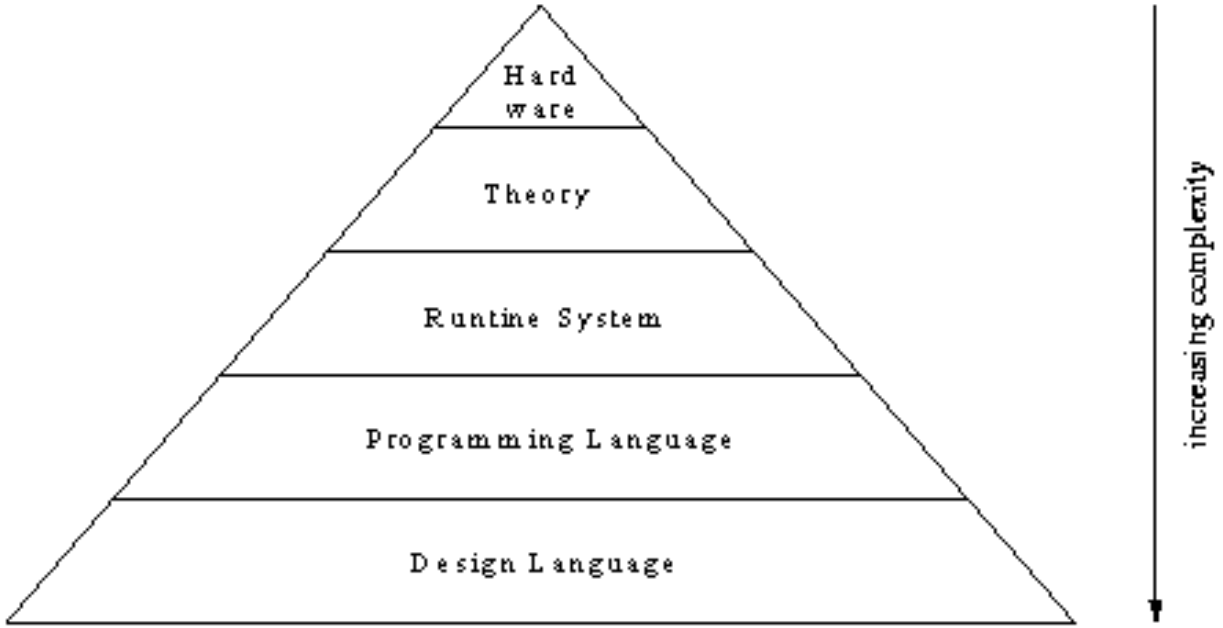
Lo scenario di una tecnologia ha almeno tre dimensioni [3]:

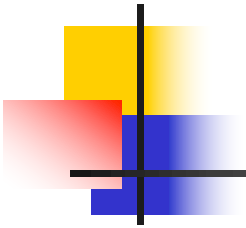
- paradigma di programmazione: le nuove tecnologie cambiano il modo in cui i sistemi vengono concepiti;
- processo di sviluppo: le nuove tecnologie cambiano il modo in cui i sistemi vengono sviluppati;
- contesto economico: le nuove tecnologie cambiano l'equilibrio di mercato ed il loro successo è influenzato dalla situazione del mercato.

In questo lavoro si è scelto di focalizzare l'attenzione sul il primo aspetto.

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 localizzati a differenti livelli di astrazione [4].

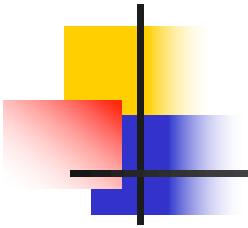




Livelli di astrazione (I)

- HARDWARE: incapsula l'architettura che è implementata attraverso l'hardware di un computer.
  - Sebbene sequenziale, parallelo o distribuito, dal punto di vista di un paradigma di programmazione, tutto l'hardware è visto allo stesso modo.
  - Tutti i paradigmi di programmazione condividono la stessa base.
- TEORIE: concettualizzazioni di un particolare modello computazionale che astrae dalle caratteristiche dell'hardware.
  - Strumenti per aiutare il programmatore ad esprimere l'idea di cosa un programma può fare in modo naturale.

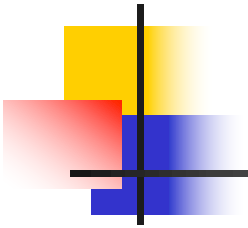
Livelli di astrazione (II)



- SISTEMA DI RUNTIME: fornisce l'ambiente per l'interpretazione di un programma. Possono essere molto diversi:
  - Semplici: gestione dell'heap, garbage collection.
  - Più complessi: motori di ragionamento (es. Prolog).
- LINGUAGGIO DI PROGRAMMAZIONE: struttura sintattica per la manipolazione delle entità del livello di runtime.
  - Dovrebbe consentire al programmatore di usare i sottostanti concetti semantici in modo efficiente e di esprimere elegantemente le funzionalità desiderate di un programma.

Livelli di astrazione (III)

- LINGUAGGIO DI PROGETTAZIONE: astrazioni, a partire da un particolare linguaggio, il cui scopo è la modellazione concettuale di un sistema ad un livello più alto.
  - 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 (UnifiedModelingLanguage), che cerca di integrare diverse notazioni progettuali prima separate.



OOP vs AOP: Teorie (I)

TEORIE (oggetti): entità gestite sono gli oggetti.

- Un OGGETTO può essere 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 staticamente ad una particolare classe che ne determina le proprietà fondamentali.
- Gli OGGETTI comunicano inviando messaggi gli uni agli altri, che possono essere utilizzati per richiedere servizi all'oggetto ricevente.
- L'insieme dei concetti orientati agli OGGETTI è chiaro e gestibile in termini di dimensioni e non varia molto tra i diversi approcci orientati agli oggetti.

OOP vs AOP: Teorie (II)

TEORIE (agenti): entità gestite sono gli agenti.

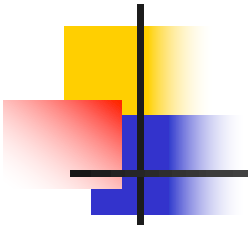
- Non esiste una definizione univoca delle entità che vengono trattate ma le teorie esistenti sono più o meno tutte basate su uno dei due concetti di agenzia ampiamente accettati.
- AGENZIA FORTE: un AGENTE è modellato in termini di nozioni mentali come credenze, desideri e intenzioni.
  - Richiede che i concetti mentali abbiano una rappresentazione esplicita nella realizzazione di un agente.
  - Impone per l'agente una visione di tipo white-box.
- AGENZIA DEBOLE: definisce un AGENTE solo in termini di proprietà osservabili.
  - Secondo questa nozione un agente è qualsiasi cosa che esibisce autonomia, reattività, proattività, attività sociale.
  - Impone per l'agente una visione di tipo black-box.

OOP vs AOP: Teorie (III)

- Esistono anche delle analogie, secondo Lind, tra i due approcci se si sostituisce il concetto di *classe* con quello di *ruolo*, le *variabili di stato* con le *credenze/conoscenze* e i *metodi* con i *messaggi* (v. tabella).
- 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.

	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

OOP vs AOP: Sistema di runtime (I)



OGGETTI

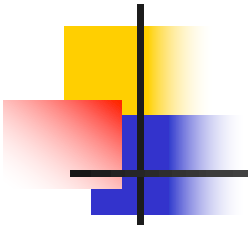
- Gli OGGETTI sono rappresentati staticamente dall'architettura dell'oggetto, che contiene lo stato corrente dell'oggetto e la relazione con la classe dell'oggetto.
- Un OGGETTO è di solito rappresentato come una collezione arbitraria di dati e funzioni.
- Il sistema di gestione degli OGGETTI è responsabile della rappresentazione delle relazioni e della manipolazione degli oggetti.
- Il sistema di gestione degli OGGETTI è anche responsabile di aspetti dinamici: selezione di oggetti polimorfi, gestione delle eccezioni, garbage collection.

OOP vs AOP: Sistema di runtime (II)

AGENTI

- Gli AGENTI sono implementati attraverso la corrispondente architettura (spesso basata sulla teoria BDI).
- L'architettura stabilisce il collegamento tra il concetto astratto di AGENTE e il "ruolo", nel senso che fornisce al sistema di runtime le descrizioni dei ruoli che costituiscono gli agenti.
- Il minimo comun denominatore per le architetture sembra essere il ciclo base *percepire - ragionare – agire* [5].
- 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.

OOP vs AOP: Linguaggio di programmazione

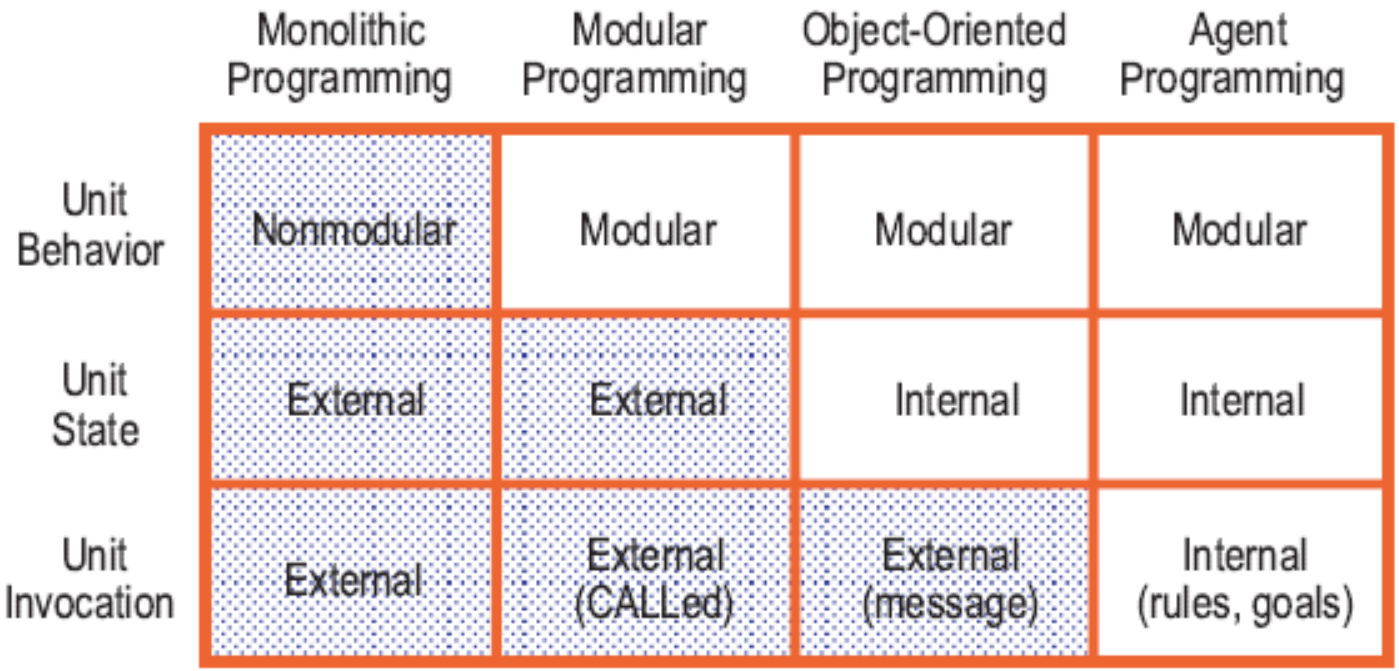


- Secondo Lind [4] l'orientamento agli OGGETTI così come l'orientamento agli AGENTI sono concetti così generali che possono essere collegati a qualsiasi altro linguaggio di programmazione.
- Nel caso degli OGGETTI, questo approccio funzionò per numerosi linguaggi (es. C e C++).
- Nel contesto dell' ingegneria del software orientata agli AGENTI, queste tendenze non sono ancora chiare.
- 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 (es. JAM Agents o Jason che combinano concetti orientati agli agenti con Java).

OOP vs AOP: Linguaggio di progettazione

- Nella comunità orientata agli OGGETTI UML è un linguaggio di progettazione ben definito sostenuto da strumenti "*case*" (es. Rational Rose) 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.
- Sebbene non esista alcun linguaggio di progettazione uniformemente accettato, principalmente a causa della mancanza di un ben definito paradigma di programmazione orientato agli AGENTI [6], c'è un gran numero di strumenti di progettazione per specifiche architetture e piattaforme orientate agli agenti (es. SIF, SWARM, ZEUS).
- E' in fase di sviluppo, inoltre, AgentUML, che propone un'estensione di UML verso i concetti orientati agli AGENTI.

Evoluzione dei linguaggi di programmazione



	Monolithic Programming	Modular Programming	Object-Oriented Programming	Agent Programming
Unit Behavior	Nonmodular	Modular	Modular	Modular
Unit State	External	External	Internal	Internal
Unit Invocation	External	External (CALLed)	External (message)	Internal (rules, goals)

Modello evolutivo proposto da James Odell [7]

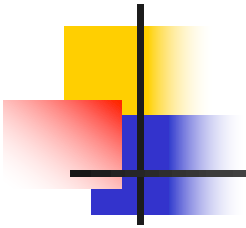
Evoluzione degli approcci alla programmazione

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

- Con l'evolvere dei linguaggi di programmazione cambiano anche gli approcci alla programmazione [6] 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.

Integrare gli agenti in un'applicazione software (I)



- A causa delle nuove caratteristiche portate dal concetto di agenzia, sta sorgendo un nuovo approccio allo sviluppo del software.
- Gli AGENTI devono essere concepiti in modo diverso dagli OGGETTI.
- Le soluzioni proposte per integrare gli agenti in un'applicazione software che contenga entità assimilabili ad agenti e non, possono essere gestite, secondo Juneidi [6], attraverso due metodi.

Integrare gli agenti in un'applicazione software (II)

- *Primo metodo (MAS e ABS)*: utilizzare un ambiente open source dedicato di una piattaforma commerciale ad agenti insieme ad un linguaggio orientato agli oggetti.
  - Una piattaforma ad agenti è un ambiente software che fornisce strumenti e funzionalità atte a garantire il funzionamento degli agenti software.
  - Una piattaforma ad agenti può anche fornire uno speciale linguaggio per la manipolazione di agenti interfacciati con linguaggi orientati agli oggetti.
- *Secondo metodo (AOSE)*: implementare gli agenti da zero utilizzando un preesistente linguaggio di programmazione orientato agli oggetti con librerie di supporto costruite per gli agenti.
  - E' necessario poter gestire le differenze tra agenti e oggetti allo stesso livello di astrazione di un paradigma di programmazione.
  - 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.

AOSE, ABS, MAS

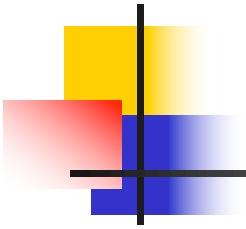
- Diverse aree, ma strettamente legate tra loro sono [6]:
  - Agent Oriented Software Engineering (AOSE).
  - Modellazione di Agent-Based Systems (ABS).
  - modellazione di Multi-Agent Systems (MAS).

	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

Verso la coesistenza di AGENTI e OGGETTI

- Secondo l'approccio OO ogni entità è rappresentata da un oggetto: qualsiasi entità può dunque essere "oggettivata".
- Non è possibile proporre un simile approccio ("agentificazione") per l' AOSE [6]: la maggior parte delle applicazioni software comuni e reali contengono entità di tipo "agente" e "non-agente".

- AOSE non è solo un'evoluzione dell' OOSE, ma 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 possono coesistere.



Bibliografia (I)

- [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] A. Omicini, *The Evolution of Computational Systems: Foundations of Agent-Oriented Computing*, Course of Multiagent Systems LS, 2009.
- [4] J. Lind, *Issues in Agent–Oriented Software Engineering - The First International Workshop on Agent Oriented Software Engineering (AOSE 2000)*, 2001.

Bibliografia (II)

- [5] S. Russell, P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, 1995.
- [6] S. J. Juneidi, *Toward programming paradigms for agent oriented software engineering*, IASTED Conf. on Software Engineering, pp. 428-432, 2004.
- [7] J. Odell, *Object and Agents Compared ,* Journal of Object Technology Vol. 1, No. 1 , 2002.