UNIVERSITA' DEGLI STUDI DI TORINO

FACOLTA' DI ECONOMIA

Corso di laurea in Economia e Commercio

Tesi di laurea in Economia Matematica

La simulazione ad agenti e lo studio dell'economia: ricostruzione virtuale del mercato borsistico.

Relatore:
Pietro Terna

Correlatore:
Sergio Margarita

Candidato:
Marco Agagliate

ANNO ACCADEMICO 2002-2003

INDICE

INTRODUZIONE

La simulazione ad agenti si propone come innovativo strumento di indagine a disposizione degli economisti e permette di affrontare lo studio dei fenomeni economici con possibilità che superano quelle offerte dai modelli classici.

Ormerod (2003) definisce economie e società moderne come sistemi complessi sospesi sull'orlo del caos. I convenzionali schemi di pensiero dell'economia offrono una visione del mondo parziale. Il mondo è considerato una "macchina molto complicata", ma del tutto comprensibile, mentre invece dovrebbe essere visto come un "essere vivente". Siccome gli individui non agiscono isolati gli uni dagli altri, ma si influenzano a vicenda attraverso modalità complesse, il comportamento del sistema economico e sociale non può essere compreso con lo studio separato degli individui che lo compongono. Secondo Ormerod, è necessario:

> "*costruire un scienza economica nuova, organica, una scienza dal tocco delicato e con un livello di raffinatezza tale da permettere l'analisi dell'interazione*".

Con lo sviluppo delle tecnologie informatiche, la comprensione dei fenomeni economici ha a disposizione innovativi strumenti con cui poter studiare i comportamenti individuali, riproducendoli attraverso un programma per computer. Per lo studio di un sistema complesso si può suddividere il sistema in parti, riprodurle all'interno di un computer ed effettuare esperimenti virtuali per analizzare cosa emerge dalla loro interazione.

L'utilizzo delle simulazioni può essere di grande aiuto per lo studio delle dinamiche che si osservano sui mercati borsistici: ad esempio, si può simulare la formazione di bolle e *crash*, e registrare le condizioni "artificiali" che ne sono la causa.

Esistono principalmente due tipologie di modelli di simulazione di borsa. La prima tipologia è rappresentata dai modelli basati sulle equazioni, dove il meccanismo di formazione dei prezzi si basa sulla condizione di equilibrio di mercato, in cui è come se operasse un "banditore" che ricava il prezzo di mercato dall'uguaglianza tra domanda e offerta. In questi modelli si automatizzano semplicemente dei modelli semplificati con cui si interpreta la realtà assumendo delle ipotesi. Le funzioni di domanda e di offerta sono le stesse utilizzate nei modelli classici dell'economia.

La seconda tipologia è rappresentata dai modelli in cui si adotta pienamente la filosofia della "simulazione ad agenti" e in cui si riproducono esattamente le diverse componenti del mercato. I modelli possono riprodurre fedelmente tutte le caratteristiche delle "parti" del mercato, come ad esempio agenti, *book* e organismi di vigilanza, senza necessariamente assumere ipotesi semplificatrici che allontanano il modello dalla realtà che cerca di riprodurre artificialmente.

Artificial Stock Market (*ASM*) è il più importante esempio di modelli di simulazione del primo tipo, mentre *Surprise (Un)realistic Market* (*SUM*) è un esempio della seconda tipologia: sono ricostruiti attraverso il codice informatico gli agenti e il *book*, e il prezzo si forma esattamente come avviene nel mercato reale, ossia coincide sempre con l'ultimo contratto concluso tra gli agenti. Nel capitolo 5 si approfondiscono gli aspetti fondamentali di questi due modelli.

Il successo dei modelli di simulazione è determinato, oltre che dalle possibilità di studio che offrono, anche dalla diffusione e dalla compatibilità dell'ambiente in cui sono sviluppati. Nei modelli di simulazione ad agenti che sono destinati a raggiungere dimensioni notevoli è determinante, affinchè si sviluppino nel tempo, l'ambiente di sviluppo che si decide di adottare. *Java* si presenta come uno dei linguaggi di programmazione più diffusi al mondo e ciò ha determinato il successo degli ambienti di simulazione basati su questo linguaggio. In futuro è auspicabile che *Swarm* possa essere sostituito da ambienti come *Jas*, che eliminano qualsiasi legame con il linguaggio *Objective C* e che presentano la massima compatibilità con qualsiasi sistema operativo.

Il modello di simulazione di borsa *SUM* rinasce nella versione *Java*, *JavaSum*, in cui si eliminano alcune limitazioni del modello di partenza, pur mantenendolo come riferimento fisso. Gli obiettivi della costruzione del nuovo modello sono legati in parte all'aspetto tecnico del codice e in parte all'aumento del realismo del modello.

La tesi si articola in sei capitoli che racchiudono il cammino di studio necessario per la costuzione di un modello di simulazione come *JavaSum*. Nel capitolo 1 si accenna alle tematiche legate allo studio dei comportamenti individuali e dei fenomeni complessi. Alcuni paragrafi sono dedicati allo studio del connessionismo e dell'intelligenza artificiale da cui derivano alcune idee per la costruzione di agenti cognitivi da includere nelle simulazioni.

Nel capitolo 2 si definiscono e si approfondiscono le simulazioni e come queste possano avere un ruolo nello studio dei fenomeni economici.

Con particolare riguardo a *Swarm*, in cui sono sviluppati i modelli di simulazione *SUM* e *JavaSum*, nel capitolo 3 si effettua una rapida rassegna dei più conosciuti ambienti di simulazione.

Il capitolo 4 è dedicato all'analisi del mercato borsistico reale. In particolare si affrontano gli aspetti principali del mercato italiano, con un'analisi delle leggi che regolano intermediari e strumenti finanziari. Inoltre si analizza il regolamento del Mercato Telematico Azionario (MTA), che è alla base delle procedure di contrattazione che sono riprodotte nella simulazione.

L'analisi del modello *Sum* e il confronto con alcuni dei modelli di simulazione di borsa costruiti in *Swarm* e in altri ambienti di simulazione, sono presentati nel capitolo 5, con particolare attenzione alle differenze interne tra i modelli analizzati.

Infine nel capitolo 6 sono descritte le diverse fasi per la costruzione di *JavaSum*, con l'esposizione degli accorgimenti tecnici e delle differenze con il modello *SUM*.

In allegato sono riportati alcuni schemi in cui è analizzato il codice del modello *SUM* (Allegato A), sono riportate le diverse parti del codice che costituiscono il nuovo modello *JavaSum* (Allegato B) ed è riportata la documentazione tecnica del codice *Java* (Allegato C).

CAPITOLO 1

LO STUDIO DEI COMPORTAMENTI INDIVIDUALI E DEI FENOMENI COMPLESSI

Lo studio dell'economia è direttamente legato allo studio dei comportamenti individuali e dei fenomeni complessi. Mises (1949) fornisce una visione filosofica dello studio dell'economia e la considera come una componente fondamentale per la comprensione delle azioni umane, come la più giovane delle scienze e come parte di una scienza molto più generale, indicata con il termine *praxeology*. L'aspetto economico delle azioni umane deve essere inquadrato in un contesto più ampio, ossia nello studio delle azioni umane in generale. Una scienza di questo tipo manca di un proprio metodo di studio e si colloca tra i metodi delle scienze naturali e i metodi delle scienze storiche, facendo apparire le teorie economiche poco utili e di scarso valore.

Il primo fine di una qualsiasi impresa scientifica è l'esaustiva descrizione e definizione di tutte le condizioni e assunzioni sotto cui le diverse dichiarazioni hanno validità. Secondo Mises è sbagliato porre l'economia sullo stesso piano di scienze come la fisica e la matematica, e quindi impostare lo studio dei fenomeni basandosi sulle stesse metodologie.

La caratteristica che distingue lo studio dei problemi economici dallo studio di problemi fisici o chimici è che in economia non esiste la possibilità di vere e proprie misurazioni. Per misurazioni si intende lo stabilire una relazione numerica tra un oggetto e un altro oggetto, che è l'unità di misura. In economia non vi sono entità "misurabili" in senso fisico e determinati "eventi storici", ossia ciò che accade in un determinato istante e sotto determinate condizioni non è ripetibile. Le stesse circostanze possono portare a misurazioni diverse, rendendo difficile ogni inferenza sul

legame causale sottostante. L'evoluzione nel calcolo economico è di fondamentale importanza per stabilire una coerente e sistematica scienza delle azioni umane, e per creare dei metodi che permettano di prevedere le azioni umane.

Parlando dei problemi dell'esistenza dell'uomo in rapporto all'economia, Mises collega in questo modo la cognizione economica e le azioni umane:

> *"Man's freedom to choose and to act is restricted in a threefold way. There are first the physical laws to whose unfeeling absoluteness man must adjust his conduct if he wants to live. There are second the individual's innate constitutional characteristics and dispositions and the operation of environmental factors; we know that they influence both the choice of the ends and that of the means, although our cognizance of the mode of their operation is rather vague. There is finally the regularity of phenomena with regard to the interconnectedness of means and ends, viz., the praxeological law as distinct from the physical and from the physiological law.*
>
> *The elucidation and the categorial and formal examination of this third class of laws of the universe is the subject matter of praxeology and its hitherto best-developed branch, economics. The body of economic knowledge is an essential element in the structure of human civilization; it is the foundation upon which modern industrialism and all the moral, intellectual, technological, and therapeutical achievements of the last centuries have been built. It rests with men whether they will make the proper use of the rich treasure with which this knowledge provides them or whether they will leave it unused. But if they fail to take the best advantage of it and disregard its teachings and warnings, they will not annul economics; they will stamp out society and the human race."*

La riflessione sugli aspetti metodologici dell'economia e sul suo ruolo nella comprensione dei comportamenti individuali prosegue, nei paragrafi che seguono, con l'analisi dei principali studi riguardanti la mente e la possibilità di riprodurre artificialmente le capacità dell'uomo, e di come questi studi possano fornire all'economia innovativi strumenti di studio.

LA COMPLESSITÀ

Come sottolina Rosser (1999), negli ultimi anni sono stati eseguiti numerosi studi sulla "complessità" dalle discipline più diverse, tra cui anche l'economia. Esistono diverse versioni della teoria della complessità, data l'interdisciplinarietà del campo di studio. In molti casi la parola "complessità" è utilizzata in modo erroneo per descrivere situazioni od oggetti particolarmente complicati.

Se ad esempio si considera un'automobile, suddivisa in tutte le sue componenti meccaniche, sicuramente è possibile considerarla come "complicata". Ma le funzioni a cui un'auto assolve risultano dalla interazione lineare delle diverse parti e l'intero sistema è un insieme costituito dalla semplice somma delle componenti meccaniche. Se una delle parti si guasta il sistema totale risulta influenzato ed è agevole risalire alle cause. Lo studio separato delle proprietà delle singole parti permette di capire il funzionamento dell'intera auto considerata come "somma".

Un formicaio può invece essere considerato un sistema complesso. Procedendo allo studio separato delle singole formiche, non è possibile trarre inferenze sul sistema totale. Le proprietà del sistema restano immutate anche se si eliminano da esso una o più formiche. Ad esempio, non si spiega come il formicaio possa mantenere una temperatura pressochè costante tra estate ed inverno. Dall'interazione di parti semplici emerge la complessità e il sistema "formicaio" non può essere considerato come semplice insieme di formiche, bensì come entità autonoma.

Lo stesso si può dire dei mercati. Terna (2002) sostiene che né studiando i consumatori come singole entità considerate a sé, né studiando la domanda quale fenomeno aggregato, ma solo tenendo conto di comportamenti e interazioni sia dei consumatori che delle imprese è possibile spiegare la complessità del mercato.

Kilpatrick (2001) sostiene che il lavoro di Hayek (1950 e 1967), sulla conoscenza e l'ordine spontaneo, sia una delle prime versioni della teoria della complessità applicata a tematiche economiche. Nella descrizione del sistema economico, secondo Hayek, i mercati non sono sempre in equilibrio e il prezzo varia a seguito dei contratti conclusi dagli agenti che operano. Egli esclude che un pianificatore centrale possa migliorare le condizioni a cui il mercato giunge autonomamente. I teorici della complessità sono molto vicini a queste posizioni, con la differenza che un pianificatore centrale non è escluso che possa avere effetti positivi sull'economia, ma non è possibile valutare a priori gli effetti del suo operato.

Inoltre, il lavoro di Hayek si presenta come un'espansione della filosofia della "mano invisibile" del mercato di Adam Smith, per cui

l'economia deriva dalle azioni umane piuttosto che dalla realizzazione di un progetto. La società risulta troppo complessa per operare seguendo la pianificazione centrale e si organizza autonomamente in una specie di ordine spontaneo che non può essere previsto. Queste tesi si avvicinano molto alla nozione di sistemi adattivi complessi delle moderne teorie sulla complessità.

Waldrop (2001) descrive l'atmosfera di fermento ed eccitazione intellettuale del *Santa Fe Institute*, fondato nel 1984, e dedicato agli studi sulla complessità. La particolarità di questo istituto è la completa mancanza di vincoli ai suoi partecipanti, nella convinzione che proprio dal confronto interdisciplinare possa evolvere una nuova visione unificatrice della scienza, per scoprire quelle leggi elementari sottese ad eventi eterogenei apparentemente inspiegabili quali la decadenza di civiltà progredite, la formazione di organi sofisticati come l'occhio e il cervello, famosi crolli di borsa come ad esempio il "lunedì nero" nell'ottobre del 1987. In Terna (2002) sono elencate le condizioni per lo studio dell'economia, costituenti la cosiddetta "visione economica di Santa Fe", che ripropone l'impostazione di Hayek:

- interazione dispersa;
- nessuna capacità di gestione globale;
- organizzazioni gerarchiche che si intersecano;
- adattamento continuo;
- innovazione continua;
- dinamica senza equilibrio.

Simon (1988) definisce un sistema complesso come un sistema composto da un gran numero di parti che interagiscono in modo non semplice. In tali sistemi l'insieme è qualcosa di più che la somma delle parti, non in senso metaforico né metafisico, ma nell'importante senso pragmatico per il quale, date le proprietà delle parti e le leggi della loro interazione, non è semplice inferirne le proprietà del tutto. Simon utilizza un semplice esempio per spiegare da cosa emerge la complessità. Si consideri una formica sulla spiaggia e il tragitto a "zig-zag" che compie per arrivare ad una meta: la formica ha un'idea di massima del luogo in cui si trova la meta, ma non ha idea degli ostacoli che dovrà affrontare per raggiungerla. La complessità si trova sulla superficie della spiaggia non nella formica.

Una formica, considerata come sistema capace di avere un comportamento, è semplicissima. L'apparente complessità del suo comportamento nel tempo è in gran parte un riflesso della complessità dell'ambiente in cui essa si trova.

Inoltre gli aspetti microscopici dell'ambiente interno del sistema formica non sono quasi per nulla rilevanti per il comportamento della formica in rapporto all'ambiente esterno. Per questo motivo un automa potrebbe benissimo simulare il comportamento della formica.

Lo stesso ragionamento si può fare per l'uomo: un uomo, considerato come sistema capace di avere un comportamento, è semplicissimo. L'apparente complessità del suo comportamento nel tempo è in gran parte un riflesso della complessità dell'ambiente in cui egli si trova.

A questo punto è possibile considerare semplicemente "l'uomo pensante", trascurando il fatto che questo sia un "uomo completo" formato da tessuti, ossa, organi. Simon considera la mente come un magazzino di informazioni, una componente dell'ambiente a cui l'organismo si adatta. L'essere umano pensante è un sistema adattivo: i suoi scopi si pongono come interfaccia tra il suo ambiente interno e il suo ambiente esterno, includendo nel secondo la sua riserva di memoria. Nella misura in cui egli è efficacemente adattivo, il suo comportamento rifletterà soprattutto caratteristiche dell'ambiente esterno (alla luce dei suoi scopi) e metterà in evidenza solo alcune proprietà vincolanti del suo ambiente interno, e cioè del meccanismo fisiologico che lo mette in condizione di pensare. Solo poche caratteristiche "intrinseche" dell'ambiente interno dell'uomo pensante limitano le capacità del suo pensiero di adattarsi alla forma dell'ambiente problema. Tutto il resto, nel suo pensiero e nel suo comportamento diretto a risolvere problemi, è artificiale: è appreso e può essere perfezionato attraverso l'assimilazione mentale di nuovi e migliori schemi.

Per ambiente va inteso anche l'insieme delle informazioni memorizzate nei libri e nella memoria a lungo termine. Tali informazioni e procedure sono ripescate in presenza di adeguati stimoli e permettono a semplici processi di informazione di base di elaborare un vasto repertorio di informazioni e strategie. Il sistema interno di elaborazione dati è costituito da entità semplici e la complessità emerge dalla ricchezza dell'ambiente esterno (mondo conosciuto attraverso i cinque sensi e informazioni presenti nella memoria a lungo termine).

Simon (2000) afferma che, sebbene negli ultimi anni ci sia stato un forte sviluppo in campo matematico, il comportamento di molti sistemi complessi rimane intrinsecamente incomprensibile. Tali sistemi sono detti "caotici" e la loro fondamentale caratteristica è che un minimo spostamento delle condizioni iniziali talvolta li induce a spostarsi lungo sentieri completamente divergenti. Ad esempio, se l'atmosfera fosse un sistema caotico, "un battito di ali di una farfalla a Singapore potrebbe provocare un tornado a Chicago".

Sistema semplice e sistema complesso

Riprendendo la distinzione effettuata da Parisi (2001), i sistemi complessi sono sistemi formati da molte componenti semplici che interagiscono localmente, generando proprietà globali non predicibili pur conoscendo alla perfezione le proprietà delle singole componenti.

In un sistema semplice una singola causa produce un singolo effetto, per cui è agevole, conoscendo le cause, prevederne gli effetti. L'effetto può anche essere conseguenza di più cause, ma il più delle volte è possibile conoscere il ruolo della singola causa. Il sistema è il risultato della somma di più effetti. Le altre principali caratteristiche di un sistema semplice sono:

- il futuro del sistema è prevedibile, ossia ogni stato del sistema si può conoscere se si conoscono gli stati precedenti;
- il modo in cui il sistema si trasforma nel tempo è noto;
- qualora il sistema sia "perturbato" da un evento esterno, l'effetto della perturbazione è commisurato alla sua entità;
- due sistemi che partono da condizioni iniziali differenti si sviluppano nel tempo in maniera diversa, proporzionalmente alla differenza tra gli stati iniziali;
- due sistemi tendono a non influenzarsi reciprocamente, quindi, ad esempio, se l'ambiente può influenzare il sistema, il sistema non può influenzare l'ambiente;
- un sistema semplice tende a non far parte di una gerarchia di sistemi;
- le varie componenti del sistema hanno un ruolo preciso nel determinare il comportamento complessivo;
- possono essere prodotte copie identiche del sistema.

Anche se i sistemi semplici sono quelli maggiormente comprensibili, spesso la realtà è fatta di sistemi complessi, nei quali molte cause producono un dato effetto e le relazioni tra le cause non sono lineari. L'effetto di ogni singola causa non è indipendente da quello delle altre cause e quindi non può essere isolato. Gli elementi di un sistema complesso sono diversi tra loro e si influenzano localmente. Dalla loro interazione emergono proprietà globali non deducibili o prevedibili, pur conoscendo alla perfezione i singoli elementi.

Per i sistemi complessi si possono individuare le seguenti proprietà:

- gli stadi futuri del sistema complesso generalmente non possono essere previsti in base agli stadi precedenti;
- le trasformazioni nel tempo del sistema non sono prevedibili;

- perturbazioni esterne producono effetti che tendono ad essere poco commisurati all'entità della perturbazione;
- vi è una grande sensibilità alle condizioni iniziali, quindi due sistemi possono svilupparsi in modo totalmente differente anche se le condizioni iniziali sono molto simili;
- gli elementi si influenzano reciprocamente, così come avviene per il sistema stesso e l'ambiente circostante;
- esistono gerarchie di sistemi con numerose interazioni tra i diversi livelli;
- il ruolo del singolo elemento non risulta ben identificabile;
- non è possibile riprodurre copie identiche di un sistema complesso.

Parisi (1999) prospetta per il secolo XXI i seguenti cambiamenti nella ricerca sul comportamento e sulla vita mentale:

- il metodo della simulazione affiancherà i metodi tradizionali della scienza consentendo lo studio di fenomeni complessi;
- i sistemi complessi costituiranno il quadro teorico che farà da sfondo alla ricerca sulla mente e sul comportamento, mostrando che è possibile collegare strettamente tra loro lo studio del sistema nervoso e lo studio della mente;
- le reti neurali saranno viste come capitolo della Vita Artificiale e ricopriranno un ruolo fondamentale nello studio della mente;
- le divisioni e i confini tra le discipline scientifiche subiranno modificazioni e perderanno di importanza, e la suddivisione tra scienze come la psicologia, la sociologia, l'economia, la storia e l'antropologia apparirà sempre meno plausibile e meno utile dal punto di vista dello sviluppo della conoscenza.

LE SCIENZE COGNITIVE

Lo studio della cognizione, dell'intelligenza e della mente vede interessate diverse discipline, che vanno dall'intelligenza artificiale alla filosofia, dalla linguistica all'antropologia, tanto che negli ultimi anni è stato coniato il termine "scienza cognitiva". Le scienze cognitive sono tutte quelle scienze, o branche delle stesse, che si occupano dello studio della mente e delle sue capacità di manipolazione simbolica.

Riprendendo le riflessioni effettuate da Legrenzi (2002), le scienze cognitive hanno come oggetto di studio la cognizione, ossia la capacità di un

sistema, naturale o artificiale, di apprendere e comunicare agli altri delle conoscenze. Possono essere catalogate come scienze cognitive la filosofia, l'informatica, la psicologia, la linguistica, l'antropologia e la biologia, in quanto si interessano di alcuni particolari problemi comuni. Spesso si utilizza il termine singolare "scienza cognitiva" per indicare l'unitarietà degli intenti, ma rischiando così di confondere scienze che rimangono ben separate tra loro, con la loro storia e i loro metodi di studio. Con il termine singolare si può indicare un obiettivo particolare che è quello di cercare di capire come funziona un qualsiasi sistema che sia in grado di filtrare le informazioni che riceve dall'ambiente che lo circonda, di rielaborarle creandone delle nuove, di archiviarle e cancellarle, di comunicarle ad altri sistemi e di prendere decisioni, adattandosi ai problemi ambientali circostanti o adattando il mondo circostante attraverso la costruzione di artefatti. Il sistema può essere naturale o artificiale e deve percepire e selezionare le informazioni, pensare, ricordare, comunicare, decidere e agire. L'obiettivo è quindi molto simile a quello della psicologia cognitiva, con la differenza che questa si occupa esclusivamente di uomini e animali e non di sistemi artificiali. Con il termine plurale invece si deve considerare un orizzonte assai più ampio, andando a studiare tutto ciò che ha a che fare con le capacità creative dell'uomo e con gli artefatti da lui creati. Si cerca di riunire, definito un oggetto di studio, diversi filoni di ricerca che sono supportati da diversi strumenti e metodi, per giungere a risultati migliori di quelli raggiungibili dalle singole scienze separatamente.

Per scienze come la psicologia, la sociologia, l'economia e l'antropologia, alla luce di linee di ricerca parallele, si sono sistematizzati i saperi comuni, per analizzare e rivisitare le teorie alla luce dei vincoli biologici della mente umana derivati dalla storia dell'evoluzione, anche grazie all'aiuto di sistemi artificiali come i computer.

I due strumenti di studio più utilizzati nelle scienze cognitive sono gli esperimenti e le simulazioni. Il programma di lavoro futuro in questo campo consiste nel costruire simulazioni e controllare, tramite gli esperimenti, se i modelli raffigurano oppure no la realtà. Il computer e le possibilità che offre sono alla base del rapporto tra mente umana e mente artificiale, il cuore delle scienze cognitive.

Lo strumento scientifico fondamentale è la simulazione al computer, con la costruzione di modelli e teorie computabili. L'interrogativo di fondo è quanto un uomo sia simile ad una macchina e in che misura la mente umana assomigli ad un computer. Johnson-Larid (1990) esamina i processi mentali dal punto di vista della computazione, sostenendo la tesi secondo cui, con l'utilizzo di un computer, è possibile riprodurre alcuni aspetti dell'intelligenza. Tuttavia, per poter affermare che la mente può stare al cervello come un programma di computer sta alla macchina elettronica su

cui è eseguito, occorre vi sia una verifica empirica e non solamente delle analisi filosofiche della questione.

Un programma per computer non può capire nulla, né possedere qulsiasi altro stato cognitivo, e per quanto intelligenti possano essere rese le macchine, ci sono attività del pensiero che devono essere affrontate solo dagli esseri umani; inoltre, se si può ammettere che un programma per computer possa essere costruito in modo tale che incorpori una teoria del significato, non si può affermare che il computer su cui il programma è eseguito effettivamente capisca il linguaggio. Un linguaggio non ha un significato finchè non gli è assegnata un'interpretazione. I computer non capiranno il linguaggio finchè non saranno in grado di metterlo in relazione con il suo dominio appropriato di interpretazione.

Una lezione importante che deriva dallo studio della mente è quella della necessità di livelli separati di spiegazione. E' necessario formulare una teoria generale su ciò che la mente computa e una teoria di come le computazioni sono eseguite dal sistema nervoso. Le teorie delle diverse scienze, come ad esempio l'economia, non dovrebbero violare ciò che si conosce sul funzionamento della mente.

INTELLIGENZA ARTIFICIALE (IA)

L'intelligenza artificiale è la branca dell'informatica che utilizza i calcolatori elettronici per studiare e riprodurre attività definibili come prodotto dell'intelligenza. Secondo la classificazione di Searle (1990), l'intelligenza artificiale può essere interpretata in senso forte come creazione di macchine pensanti o in senso debole come costruzione di strumenti per risolvere compiti che solo gli esseri umani sanno eseguire. In questo ultimo filone rientra lo studio delle attività mentali attraverso modelli computazionali. In questo senso, l'intelligenza artificiale si occupa di risolvere problemi intelligenti attraverso la costruzione di programmi per computer che seguono strategie utilizzate dagli esseri umani ed entra a far parte della scienza cognitiva, in quanto permette la validazione e la sperimentazione delle teorie sulla mente.

Haugeland (1988) afferma che i filosofi, per millenni, si sono interrogati su cosa siano la mente e il pensiero, su cosa contraddistingua gli esseri umani rispetto al resto dell'universo conosciuto, ma purtroppo le risposte a questi interrogativi sono ancora oggi poco soddisfacenti. Negli ultimi decenni non solo la psicologia, ma numerose altre scienze si sono occupate di filosofia della mente e al centro di tutte vi è l'intelligenza artificiale. L'obiettivo primario di questa scienza non è semplicemente quello di imitare l'intelligenza umana o produrne una copia, bensì la costruzione di macchine dotate di mente, di forme di intelligenza originali e

non frutto dell'imitazione delle capacità umane. Sarebbe forse più corretto parlare di intelligenza "sintetica", in analogia con la terminologia che si usa solitamente, ad esempio, con i diamanti: un diamante artificiale è un diamante che imita le caratteristiche del diamante vero, mentre il diamante sintetico è un vero e proprio diamante, creato sinteticamente in laboratorio.

Le posizioni al riguardo sono varie, ma si nota la suddivisione in due gruppi. Un primo gruppo composto da denigratori, che sorridono al solo pensiero che possa essere creata una qualche forma di intelligenza concorrente a quella umana, e un secondo gruppo di "tifosi", che sostengono che sia solo questione di tempo. Nell'immaginario collettivo quest'idea ha suscitato molte reazioni e sono nate fantasie, che hanno avuto manifestazione pratica soprattutto nella letteratura e nella cinematografia. Si pensi alle "creature", come il dottor Frankenstein, oppure ai robot meccanici immaginati da Asimov.

L'IA fonda le sue radici nella meccanica e nell'elettronica programmabile, ma non vi sono stati studi e realizzazioni di congegni meccanici o elettromeccanici intelligenti, se non a livello romanzesco. Per contro, si è notevolmente sviluppata la ricerca sui calcolatori intelligenti. Ciò deriva dalla tradizione della filosofia occidentale, secondo cui il pensiero, inteso come attività intellettiva, è essenzialmente una manipolazione razionale di simboli mentali, di idee. Quindi appare ovvio il motivo per cui i congegni meccanici siano tagliati fuori dalle ricerche sull'intelligenza, mentre i calcolatori, che possono manipolare simboli arbitrari, giocano un ruolo da protagonisti. Le speranze sono tutte rivolte ai calcolatori, perchè si ritiene che possano compiere operazioni molto simili a quelle mentali.

L'IA segue, come assunto di base che la caratterizza come scienza cognitiva, che la mente operi su principi computazionali e che l'intelligenza dipenda solo dall'organizzazione di un sistema e dal suo operato come manipolatore di simboli, i quali devono soddisfare definizioni astratte, senza che sia rilevante conoscerne forma e sostanza. Quindi il tipo di sistema fisico di partenza per riprodurre l'intelligenza è completamente irrilevante.

La definizione di intelligenza non risulta necessaria e generalmente si accetta il criterio proposto da Turing, il quale propose di ignorare i problemi linguistici e adottare un semplice test, da lui progettato, per giudicare se una macchina sia o no intelligente[1]. L'obiettivo del test è quello di capire se

[1] Il test di Turing consiste nell'organizzare un gioco, detto "gioco dell'imitazione": i giocatori sono tre, fra loro estranei, il primo svolge il ruolo di interrogante, il secondo è un uomo e il terzo una donna. Le domande rivolte sono finalizzate a capire il sesso dei due interrogati. Uno dei due interrogati mente, fingendo di essere dell'altro sesso, mentre l'altro cerca di aiutare l'interrogante. Le domande e le risposte sono trasmesse per telescrivente, per evitare che si possano trarre delle conclusioni dal timbro vocale. Il calcolatore deve essere sostituito

un calcolatore può conversare come una persona, ossia comprendere di cosa si parli e restituire risposte adeguate e sensate. Una volta accettato il test di Turing, agli scienziati occorre ideare la struttura interna e le operazioni necessarie affinché un sistema riesca a dire la cosa giusta nel momento adatto, ossia devono badare agli aspetti cognitivi del problema.

Numerose critiche all'IA riguardano la possibilità di riprodurre al calcolatore qualità umane come la creatività, la libertà artistica, la responsabilità, dal momento che un calcolatore può fare solamente ciò per cui stato programmato. Ma in realtà si può pensare che l'essere umano si trovi in una situazione analoga e le sue capacità siano fondate su definizioni interiori accurate di tutti i processi immagazzinati nel cervello. Così i programmatori dell'intelligenza artificiale inseriscono nei computer solo fasci di informazioni e principi generali, simili a quelli che gli insegnanti umani instillano nei loro allievi. Quel che accade in seguito non è prevedibile e spesso i risultati sono sorprendenti. Un esempio molto famoso è quello delle macchine che giocano a scacchi e che riescono a battere i loro programmatori con mosse brillanti e inaspettate. Esempi simili non bastano per poter affermare che i sistemi informatici possono essere creativi, ma non si può nemmeno essere certi del contrario. Il vero nocciolo della questione rimane la comprensione del fatto che l'uomo possa o meno essere considerato un calcolatore.

I sistemi esperti sono uno degli sviluppi di maggior successo dell'IA. Affinché in un determinato campo possano essere utilizzati sistemi esperti occorre che:

1. le decisioni pertinenti dipendano da un insieme definito di fattori;
2. gli effetti dei diversi fattori siano quantificabili;
3. sia possibile derivare una decisione dal valore assunto dalle diverse variabili;
4. le interrelazioni tra i fattori siano abbastanza complesse da giustificare l'utilizzo di un sistema esperto.

Attualmente sono utilizzati in campi come la diagnosi medica, analisi geologica di campioni per rilevare l'esistenza di petrolio e ottimizzazione di configurazioni microscopiche per la costruzione di *chip*. Gli interessi economici garantiscono l'arrivo di fondi per la ricerca in questo campo.

al giocatore che mente e, per essere giudicato intelligente, deve riuscire nell'opera d'inganno allo stesso modo dei giocatori umani (deve ottenere gli stessi risultati del giocatore medio).

Gli antenati dell'IA

Le linee guida degli studi sull'intelligenza artificiale si sviluppano a seguito di un interesse sempre più sentito nei confronti del concetto di "mente", che compie i suoi primi passi nel corso del XVII secolo, accompagnato dalla scienza moderna e dalla matematica moderna. In questo secolo molti celebri filosofi e scienziati cominciano ad avanzare ipotesi, riflessioni e a proporre modelli riguardanti la mente e l'intelligenza.

Nei paragrafi seguenti si effettua una veloce carrellata dei contributi che costituiscono le basi ideali da cui derivano gli attuali studi sull'intelligenza.

Copernico (1473-1543). Con Copernico si stravolge l'idea medioevale del mondo. Fino ad allora tutto era visto come un'imitazione di Dio, comprese le idee, che risultavano vere nella misura in cui fossero vicine a quelle divine. Il primo elemento della rivoluzione copernicana fu senza ombra di dubbio la teoria eliocentrica, ma anche la distinzione tra apparenza e realtà da cui consegue la distinzione tra mente e mondo.

Galilei (1564-1642). Il primo a sostenere che l'unico modo di capire la natura fisica fosse sotto forma di relazioni matematiche fra variabili quantitative fu Galileo Galilei. Egli diede molta importanza alla geometria, che utilizzò in modo astratto per formulare i propri teoremi, senza, come Copernico, filosofeggiare molto sulla mente e sull'anima. Furono Hobbes e Cartesio ad accorgersi che la geometria avrebbe avuto conseguenze importanti per la teoria della rappresentazione mentale.

Hobbes (1588-1679). Hobbes è considerato il "nonno" dell'intelligenza artificiale. Fu il primo ad affermare esplicitamente che "raziocinio" coincide con "calcolo" e ammirava molto l'opera di Galileo, da cui apprese il metodo di indagine scientifica. Egli sosteneva che il pensiero è "discorso mentale" ed è costituito da operazioni simboliche, come il parlare a voce alta o il calcolare con carta e matita. I pensieri sono immaginati come particolari elementi del cervello, "particelle di pensiero", che risultano operare in modo più chiaro e razionale se si seguono regole metodiche. La mente risulta essere un aggeggio meccanico, come un "abaco". La realtà risulta essere composta da minuscole particelle in movimento e le qualità sensibili non appartengono agli oggetti ma sono interne a chi le percepisce.

Cartesio (Renè Descartes, 1596-1650). Le scoperte di Cartesio furono rivoluzionarie in matematica, ma anche nella filosofia della mente. L'innovazione principale è la nuova concezione della relazione tra i simboli e ciò che simboleggiano, ossia della teoria del significato. Egli considerava i pensieri come rappresentazioni simboliche analoghe a quelle usate in matematica. Inoltre riteneva che ciò che è ragionevole è determinato dalle regole di ragionamento, dalle regole per manipolare i "simboli-pensieri", nel

sistema di notazione della mente. Questa innovazione ebbe notevoli conseguenze sulla filosofia moderna. Le macchine, secondo Cartesio, non possono pensare perchè non sono in grado di manipolare razionalmente simboli. Questa opinione rientra nel "territorio" dell'intelligenza artificiale.

Hume (1711-1776). Locke fu uno degli ispiratori della filosofia di Hume, il quale vedeva la mente come una "tabula rasa" su cui l'esperienza "scrive" la conoscenza e tutte le impressioni sensoriali. Dopodiché le capacità di associazione naturali della mente combinano queste "idee". Egli appoggiava i metodi empirici di Newton e non quelli cartesiani basati sull'intuizione come per la matematica:

> *"Lo stesso è stato fatto riguardo ad altre parti della natura. E non c'è ragione per non sperare un pari successo per le nostre ricerche relative ai poteri mentali e all'economia, se saranno proseguite con altrettanta capacità e cautela."* - **HUME (1749), SEZIONE1- AN ENQUIRY CONCERNING HUMAN NATURE, ED. OR. 1749, IN SELBY-BIGGE (1902)**

Hume condivideva il meccanicismo di Hobbes e sosteneva che la mente dovesse essere studiata come un fenomeno fisico (in quanto trattasi di particelle fisiche in movimento), e quindi non doveva essere affrontata come lo studio e la manipolazione di simboli fisici. I suoi ragionamenti rimangono intrappolati dal paradosso della ragione meccanica: se i significati hanno importanza per le manipolazioni, allora i processi non sono realmente meccanici, mentre se i significati non hanno importanza, i processi non sono realmente razionali.

IL CONNESSIONISMO E LE RETI NEURALI ARTIFICIALI

L'utilizzo delle simulazioni come strumento di indagine in economia, spinge i ricercatori in campi che tradizionalmente non competono direttamente gli economisti. Ci si deve infatti occupare di come le persone decidono e con quali strumenti. Il connessionismo è il tentativo di simulare l'intelligenza biologica, riproducendo su un calcolatore le capacità mentali umane. Rappresenta una vera e propria rivoluzione nello studio della mente e del cervello che ha avuto un notevole sviluppo negli ultimi decenni.

Parisi (1989) sostiene che le conoscenze su mente e cervello potrebbero essere più avanzate e individua le seguenti cause:

1. lo studio del comportamento e delle facoltà umane è considerato indipendentemente dallo studio del cervello;

2. lo studio del cervello è stato affrontato dalle neuroscienze, ma non si è giunti a cogliere il suo funzionamento in modo da comprendere comportamenti e capacità mentali;
3. la lontananza tra lo studio del comportamento e lo studio del cervello porta a una concorrenza tra neuroscienze e psicologia, che nonostante i tentativi di integrazione, come la psicofisiologia o la neuropsicologia, procedono separatamente con modelli teorici e metodi differenti.

Uno degli obiettivi primari del connessionismo è quello di favorire lo studio del cervello e, riproducendone la struttura, cercare di comprendere come si sviluppano i comportamenti e le facoltà umane.

Per simulare il funzionamento del cervello si utilizzano dei modelli conosciuti come sistemi dinamici non lineari o complessi, e sistemi di elaborazione parallela distribuita. Non si osservano i fenomeni reali per interpretarli con una teoria, come ha fatto la scienza fino ad oggi, ma si riproducono all'interno di un calcolatore (le simulazioni sono trattate nel capitolo 2).

In campi come l'intelligenza artificiale e l'informatica, verso la metà del XX secolo, in particolare con lo sviluppo della neurocibernetica,si tenta di riprodurre l'intelligenza, badando solo alle proprietà puramente funzionali, che possono essere riprodotte da qualsiasi macchina fisica che riesca ad eseguire una serie di istruzioni ben specificate.

La concezione connessionista è opposta: le caratteristiche fisiche della macchina su cui si riproduce l'intelligenza sono fondamentali e l'obiettivo del connessionismo è la riproduzione delle capacità mentali umane su sistemi artificiali concepiti secondo i principi di funzionamento del cervello.

Un sistema connessionistico è composto da un insieme molto grande di unità molto semplici, che comunicano tra loro con legami unidirezionali, attraverso i quali le unità comunicano attivazioni o inibizioni alle unità a cui sono collegate. Questi apparati, vista la grande somiglianza con la struttura neurale del cervello umano, prendono il nome di reti neurali artificiali (RNA). Il cervello umano è composto da reti di neuroni, che attraverso le sinapsi, le quali collegano gli assoni ai dendridi, trasferiscono un impulso elettrico di una certa intensità. Questa struttura si presenta molto lontana dalla logica dei sistemi computazionali classici dell'informatica e dell'intelligenza artificiale, definita inizialmente da John von Neumann, ossia dalla struttura di un calcolatore tradizionale.

Il cervello non può essere pensato come un insieme di istruzioni, con depositi passivi di dati, in cui ogni singola attività è programmata ed eseguita mentre tutto il resto è in attesa. Inoltre è difficile accettare che

l'intelligenza sia un qualcosa creato dall'esterno, invece che considerarla un processo di crescita attraverso l'esperienza e l'accumulazione di conoscenza.

Ogni attivazione o inibizione, trasmessa dai diversi nodi della rete neurale, dipende da un peso quantitativo che caratterizza la connessione attraverso cui è trasferita. Ogni unità trasforma la somma degli impulsi che riceve dalle unità precedenti, e manda a sua volta impulsi alle unità che seguono. Vi sono infine delle unità di uscita che restituiscono dei valori trasformati dai diversi nodi della rete.

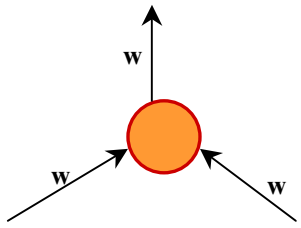
La rete è in grado di apprendere e la "conoscenza" è accumulata nei pesi relativi alle connessioni della rete. Inizialmente sulle connessioni vi sono dei pesi casuali che poi sono modificati dalla rete in base ad un metodo di *training*, il quale permette di ricavare, attraverso delle correzioni, pesi che riescano a collegare dei valori in *input* con valori in *output*.

Attualmente le reti neurali sono utilizzate con i tradizionali calcolatori. Potrebbe essere utile la costruzione di macchine apposite, anche se i normali calcolatori, visto il grande sviluppo tecnologico e le velocità per operazione elementare che si sono raggiunte, non rappresentano un intralcio nell'affrontare calcoli paralleli come quelli dei sistemi connessionistici.

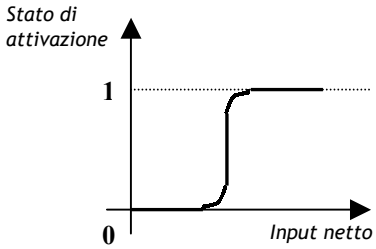
Il dibattito sull'intelligenza artificiale coinvolge numerose scienze. L'interrogativo principale è se sia possibile riprodurre l'intelligenza su un sistema fisicamente diverso dal cervello. Searle (1990), ad esempio, sostiene che i programmi al calcolatore non possono mai dare origine ad una mente, mentre altri, come ad esempio i Churchland (1990), sostengono che circuiti modellati sul cervello potrebbero effettivamente raggiungere l'intelligenza.

La struttura delle reti neurali artificiali

La rete neurale è formata da un certo numero di unità collegate tramite connessioni che trasmettono attivazioni o inibizioni da un'unità all'altra, e sono in genere unidirezionali. Le attivazioni o inibizioni che giungono dai vari nodi o neuroni, determinano lo stato di attivazione dell'unità. Su ogni connessione vi è un peso quantitativo che può essere considerato come una misura della conduttività della connessione. Quindi il valore che giunge ad un neurone dipende dallo stato di attivazione dell'unità precedente e dal peso della connessione collegante.



Nodo di una rete con pesi e connessioni

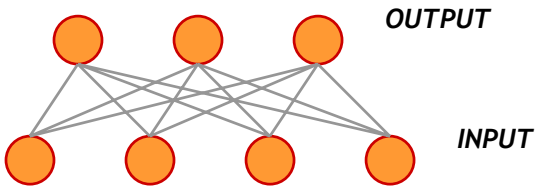


Funzione a soglia (approssimata dalla funzione logistica)

Di solito lo stato di attivazione varia tra due valori (ad esempio 0 e 1). Il neurone quindi calcola la somma degli impulsi che giungono dalle varie connessioni, e tale valore è trasformato da una funzione a soglia in modo che sia compreso nell'intervallo di valori desiderato (tra 0 e 1). Una funzione molto utilizzata è la funzione logistica o sigmoide, che rappresenta una buona approssimazione della funzione a soglia (rappresentata in figura). Il valore risultante dalla trasformazione è il valore di soglia relativo allo stato di attivazione del neurone, che determinerà l'impulso che sarà inviato al livello di connessioni successivo.

La rete neurale è organizzata in strati. Ogni strato è costituito da un insieme di neuroni che si trovano ad un certo livello nella struttura della rete. Le unità con sole connessioni in uscita sono neuroni di *input*, mentre le unità con sole connessioni in entrata sono dette neuroni di *output*. I valori degli stati di attivazione delle unità di *output* sono letti dall'esterno e restituiscono la reazione della rete a determinati valori di *input*. Tra lo strato dei neuroni di *input* e lo strato dei neuroni di *output* vi possono essere diversi strati, composti da neuroni con connessioni in entrata e in uscita.

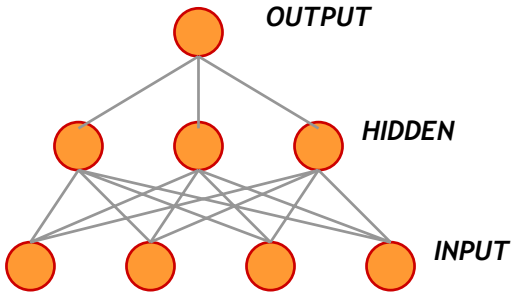
La struttura che nelle ricerche degli anni ’40 e ’50 era detta *peceptrone* è una particolare rete neurale composta da due soli strati , uno di *input* e uno di *output*.



Percettrone

Le reti neurali più sofisticate hanno uno o più strati di unità nascoste, collocate tra le unità di *input* e le unità di *output*. Ogni unità di uno strato è connessa con le unità di un altro strato, ma non vi sono connessioni tra le

unità dello stesso strato o tra le unità di uno strato precedente (dette di *feedback*). Reti così configurate sono dette reti *forward*.



Rete neurale a tre strati

Il funzionamento delle RNA

Una volta strutturata una rete neurale, occorre trovare un insieme di pesi che permetta alla rete di comportarsi in un determinato modo a seconda della situazione in cui si trova. Con questo "apprendimento" la rete neurale riproduce artificialmente il comportamento di una rete neurale naturale che associa delle risposte a degli stimoli esterni. Le reti neurali artificiali (RNA) diventano in questo modo veri e propri modelli di simulazione delle reti neurali naturali.

Dato un certo *pattern* di valori di attivazione alle unità di *input*, il passo successivo è quello di calcolare per ciascuna unità nascosta prima l'*input* netto e poi il suo stato di attivazione (somma algebrica e trasformazione con la funzione sigmoide). Determinato lo stato di attivazione delle unità nascoste si passa allo stato delle unità di *output* e anche qui si calcola l'*input* netto e lo stato di attivazione. Tale stato di attivazione rappresenta l'*output* della rete, derivato dagli stimoli in *input* iniziali.

Il modo in cui la rete risponde ad un *pattern* di attivazione esterna dipende interamente dai pesi che stanno sulle connessioni tra le unità. Volendo ottenere un determinato stato di attivazione dalle unita di *output*, dati i *pattern* in *input*, occorre impostare i pesi in modo adeguato. Per impostare i pesi occorre avvalersi di determinati metodi di correzione che permettano di arrivare alla serie di pesi desiderata partendo da una serie di pesi messi a caso. Attraverso questi metodi si modificano automaticamente i pesi fino ad assegnare loro quei valori che consentano di rispondere nel modo adeguato ad un certo stimolo esterno.

Uno di questi metodi è quello della *backpropagation*, o di propagazione all'indietro dell'errore. Durante l'apprendimento è fornito alla

rete il cosiddetto *input* di insegnamento, che è un insieme di dati contenente i *pattern* di *input* e i corrispondenti *pattern* che si attendono in *output*. Dati i *pattern* di *input* la rete calcola, in base ai pesi casuali presenti sulle connessioni, i *pattern* di *output*, che inizialmente si discostano dagli *output* desiderati. Gli errori relativi alle singole unità di *output* sono calcolati come differenza, positiva o negativa, tra lo stato di attivazione raggiunto dalla rete e lo stato di attivazione desiderato presente nell'*input* di insegnamento. I pesi della rete sono modificati in base all'errore: se l'errore è positivo il peso è diminuito, e viceversa.

La minimizzazione dell'errore è ottenuta (vedere Terna 1994) misurando l'errore come somma dei quadrati degli scostamenti degli *output* della RNA dai valori originali (*target*). La correzione del singolo peso avviene sulla base di una quota, detta *learning rate*, dell'opposto della derivata dell'errore di ogni *pattern* rispetto al peso stesso. Alla correzione è inoltre aggiunta una quota della correzione precedente, detta *momentum*, che ha lo scopo di mantenere la direzione del "movimento della correzione". Valori di *learning rate* e *momentum* troppo alti possono allontanare la rete dalla soluzione invece di avvicinarla.

Esposta ad esperienze ripetute, la rete modifica progressivamente i suoi pesi in modo da riuscire a rispondere adeguatamente agli stimoli. Per questo si dice che le reti apprendono, e non devono essere programmate fin dall'inizio per dare una certa prestazione, ma imparano da sole ad auto-organizzarsi. L'apprendimento è guidato dall'esterno e, ad ogni esperienza, alla rete è suggerito il segno della correzione necessaria affinchè l'*output* si avvicini a quello desiderato, ed è concluso quando l'errore è sufficientemente basso oppure non accenna a ridursi aumentando il numero delle epoche. In seguito ad un adeguato numero di esperienze, o cicli di apprendimento, la rete è in grado di produrre da sola gli *output* esatti.

Di solito l'errore scende con il passare delle epoche in cui la rete apprende ma può capitare che l'errore salga e scenda e non diminuisca più, oppure rimanga piuttosto alto diminuendo impercettibilmente. In questi casi la rete trova un punto di minimo locale.

Un aspetto interessante è il fatto che la rete sia in grado di progredire da sola. Una volta effettuato l'apprendimento, la rete può rispondere in modo adeguato agli stimoli ed è sempre in grado di fornire una risposta qualora lo stimolo iniziale sia confuso o risulti essere simile ad uno dei casi avuti in input per l'apprendimento. La rete arriva sempre ad una risposta, anche se parzialmente errata.

Come sottolinea Parisi (1989), il pregio della rete è la minor rigidità rispetto ai sistemi simbolici. Un sistema simbolico che è programmato a rispondere in un certo modo ad un determinato stimolo, non riesce a reagire se lo stimolo non corrisponde a quello per cui è stato programmato. La rete

invece cerca di dare una risposta che si avvicini allo stimolo ricevuto, e risponde in modo sensato anche se il *pattern* ricevuto non è compreso in quelli su cui è avvenuto l'addestramento.

Nel caso si trovi come stimolo un *pattern prototipo*, la risposta della rete risulta ancora migliore a quella che avrebbe dato per uno dei *pattern* della casistica utilizzata per l'apprendimento. Un prototipo è un *pattern* da cui è possibile, cambiando alcune sue componenti, derivare tutti o una parte dei *pattern* presentati alla rete. Se i *pattern* sono del tutto casuali, e quindi non derivano da un prototipo, la rete ha difficoltà di classificazione.

Le reti raramente danno risultati completamente corretti o completamente sbagliati e, ripetendo lo stesso esperimento con una rete, non è detto che si trovino gli stessi risultati, sia come aderenza dell'output a quello desiderato che come matrice finale dei pesi. Il risultato sarà solo simile, ma non identico, a seconda degli inneschi casuali iniziali, della frequenza e dell'ordine dei *pattern* che compongono la casistica iniziale.

I precettroni hanno possibilità limitate, analizzate da Minsky e Papert (1969, 1988), in quanto effettuano semplicemente una serie di associazioni dirette tra input e output, senza che avvenga alcuna "rappresentazione interna" (*pattern* di attivazione sulle unità nascoste) che permetta di mettere in luce somiglianze e differenze tra i *pattern* su cui la rete apprende. Per Parisi (1989) questa è una delle principali cause che hanno favorito l'abbandono dello studio delle reti neurali negli anni '70. La ripresa delle ricerche si ha con le reti a nodi nascosti e il metodo della *backpropagation*. Le rappresentazioni costruite dalle reti non sono simboliche, ma sono rappresentzioni quantitative distribuite internamente su tutti i nodi nascosti.

La struttura delle reti può essere complicata in modo da poter simulare alcune componenti del comportamento intelligente. L'unico limite è quello di mantenere la rete di tipo *forward*, per poter utilizzare la *backpropagation*. Ad esempio, è possibile simulare la memoria, con la suddivisione tra memoria a breve e a lungo termine, tenendo conto della diversa influenza delle azioni più o meno distanti nel tempo: si crea un quarto gruppo di unità, dette di stato o contestuali o di memoria, che conserveranno una traccia dello stato di attivazione di determinate unità della rete. Questi stati di attivazione conservati sono comunicati alle unità di attivazione in un periodo successivo, e subito dopo sono sostituiti dai nuovi valori di stato di attivazione (sulle connessioni tra unità di *output* e unità di memoria vi è un peso fisso a 1). Per le unità nascoste è come se ci fossero altre unità di *input* (ciascuna unità di memoria è connessa con ogni unità nascosta). Il risultato è che l'*output* sarà influenzato da tutti gli *output* precedenti della rete, ma la maggiore influenza è data dagli *output* passati più recenti.

Connessionismo e cognitivismo

Parisi (1989) sostiene che la principale differenza tra cognitivismo (psicologia cognitiva, intelligenza artificiale e linguistica) e connessionismo sia il diverso concetto di "rappresentazione interna dei modelli". Nel cognitivismo è il ricercatore che decide quali siano le rappresentazioni interne del suo modello, quali concetti usare in queste rappresentazioni e quali siano le regole per combinarli. Le reti invece apprendono autonomamente le rappresentazioni interne. Per i cognitivisti il problema principale è quello di trovare delle rappresentazioni simboliche corrette da immettere nei loro modelli. I connessionisti invece hanno il problema di comprendere le rappresentazioni quantitative interne create autonomamente dalla rete. La rete può risultare poco trasparente riguardo ai meccanismi incorporati per arrivare alle soluzioni del problema a cui sono applicate.

Per i connessionisti assume fondamentale importanza lo studio della struttura interna delle reti, per comprendere cosa sia successo durante l'apprendimento. Uno degli strumenti utilizzati è la *cluster analysis*, con cui si classificano le rappresentazioni interne della rete e, a seconda delle condizioni in cui è affrontato l'apprendimento, è possibile notare le relazioni di somiglianza rilevate dalla rete. Un'altra tecnica consiste nel lesionamento della rete, per analizzarne gli effetti sui risultati che si ottengono.

Il connessionismo e l'intelligenza artificiale (IA) presentano una serie di divergenze nell'utilizzo degli strumenti informatici. Dal lato dei connessionisti si ha l'auto-organizzazione delle reti e il problema è quello di costruire reti in grado di apprendere e che, dopo un adeguato apprendimento, siano in grado di svolgere il compito assegnato, mentre dal lato dei cognitivisti vi è la programmazione esplicita con la costruzione di sistemi computazionali in grado di fornire una certa prestazione.

Nel primo caso occorre definire i risultati che dobbiamo ottenere a fronte di determinati dati in entrata, mettendo insieme la casistica su cui la rete può apprendere, e progettare la sua struttura. Nel secondo caso, invece, si analizza un problema, lo si suddivide in una serie di passi, di strutture dati, si considerano le diverse possibilità e si esprime tutto in un linguaggio di programmazione. Il programma non è in grado di fornire risposte a problemi non previsti dal programmatore, mentre la rete può dare qualche risultato imprevisto.

L'interesse di fondo che guida l'intelligenza artificiale è la costruzione di macchine utili ad uno scopo, non necessariamente legate strutturalmente a qualcosa presente in natura. Nell'IA vale l'ipotesi che la macchina è un elaboratore di simboli, vi è quindi una concezione simbolica dell'intelligenza, diversa dalla concezione sub-simbolica del connessionismo. Simon (1988)

afferma che il computer, insieme alla mente e al cervello, fa parte della famiglia dei "sistemi simbolici fisici". La quintessenza dei sistemi simbolici è l'artificialità, dato che l'adattamento ad un ambiente è la loro unica ragione di esistenza. Si tratta di sistemi che elaborano informazioni e che sono volti ad uno scopo, di solito inseriti in contesti più ampi nell'ambito dei quali svolgono un determinato compito. I computer hanno trasportato i sistemi simbolici dal mondo platonico delle idee al mondo empirico dei processi svolti da macchine o cervelli, o da entrambi in collaborazione.

Per "intelligenza" Simon intende il lavoro di sistemi simbolici, ossia accetta l'ipotesi che un sistema simbolico fisico abbia i mezzi necessari (mostrare come il cervello umano operi come un sistema simbolico) e sufficienti (costruzione di programmi per computer che siano in grado di compiere azioni intelligenti) per azioni intelligenti. Nell'IA si ha l'obiettivo di costruire programmi molto articolati che permettano al calcolatore di comportarsi in modo intelligente, ma non è necessario che la struttura dei programmi riproduca la struttura naturale del cervello.

Parisi (1989) sostiene che per l'IA il calcolatore è un modello, una "metafora" di ciò che è studiato (mente e intelligenza), mentre per il connessionismo è uno strumento per fare simulazioni. Si contrappone la sequenzialità al parallelismo, con sistemi composti da unità elementari, anche molto lente, ma di numero molto grande. L'IA "emula" e non "simula" l'intelligenza umana, cerca di riprodurne i risultati ignorando la struttura e le modalità. Il programmatore predispone una sequenza di istruzioni in un qualche simbolismo che il calcolatore esegue su dei dati passivi.

La simulazione si propone come metodologia di indagine per lo studio del cervello. I modelli connessionistici appartengono all'insieme di modelli che sono formulati per studiare sistemi complessi, a dinamica non lineare o sistemi di tipo caotico. Si possono usare le reti anche per studiare fenomeni sociali o evoluzionistici, per lo studio di sistemi difficilmente affrontabili con i metodi di studio classici e gli strumenti matematici tradizionali. Per questo possiamo dire che le reti neurali sono una vera e propria rivoluzione scientifica oltre che il tentativo di congiungere lo studio del cervello allo studio della mente e a quello della realtà fisica in generale.

La simulazione è un metodo che differisce molto dai metodi tradizionali della scienza, non mira a descrivere, prevedere o spiegare i fenomeni reali, ma a ricrearli. La scienza classica procede formulando, verbalmente o con qualche simbolismo, teorie e ipotesi, derivando da esse predizioni e verificando, in condizioni controllate, se queste ultime siano corrette oppure no. Con la simulazione, al contrario, le teorie e le ipotesi sono espresse sotto forma di progetti, si costruiscono i sistemi e ci si aspetta che si comportino come la realtà che si vuole studiare. Secondo Parisi, sono

destinate a svolgere un ruolo sempre più importante nella scienza per i seguenti motivi:

a) se un'ipotesi o una teoria servono a costruire una simulazione occorre che siano complete, dettagliate ed esplicite;

b) offre maggiori gradi di libertà (o altrettanti) del metodo sperimentale nel manipolare le variabili del sistema che si è costituito;

c) si possono studiare fenomeni per cui è impossibile impostare esperimenti;

d) appare particolarmente appropriato per lo studio di fenomeni complessi, nei quali il comportamento globale è il risultato dell'interazione di molte componenti e ha caratteristiche emergenti rispetto alle singole componenti prese separatamente.

IL METODO DEI CROSS-TARGET (CT)

Per sviluppare esperimenti con l'utilizzo di agenti, Terna (2000) introduce le seguenti ipotesi: un agente che opera in un ambiente economico deve adattare le proprie capacità di valutazione in modo coerente al fine di conseguire risultati specifici e tenendo conto delle conseguenze che le proprie decisioni potrebbero avere. Oltre a questa coerenza interna l'agente deve essere in grado di compiere azioni o valutare degli effetti in base a regole fornite dal sistema oppure dagli altri agenti, per imitazione del loro comportamento.

Terna (2000c) sostiene che l'idea guida sia che un agente economico sviluppi con l'apprendimento la capacità di valutare in modo coerente. La tecnica dei *Cross Target* è sviluppata per costruire Agenti Artificiali Adattivi (AAA) senza necessariamente fare ricorso a regole economiche a priori. La rete neurale artificiale che rappresenta un agente presenta lo strato di *output* suddiviso in due parti caratterizzate da:

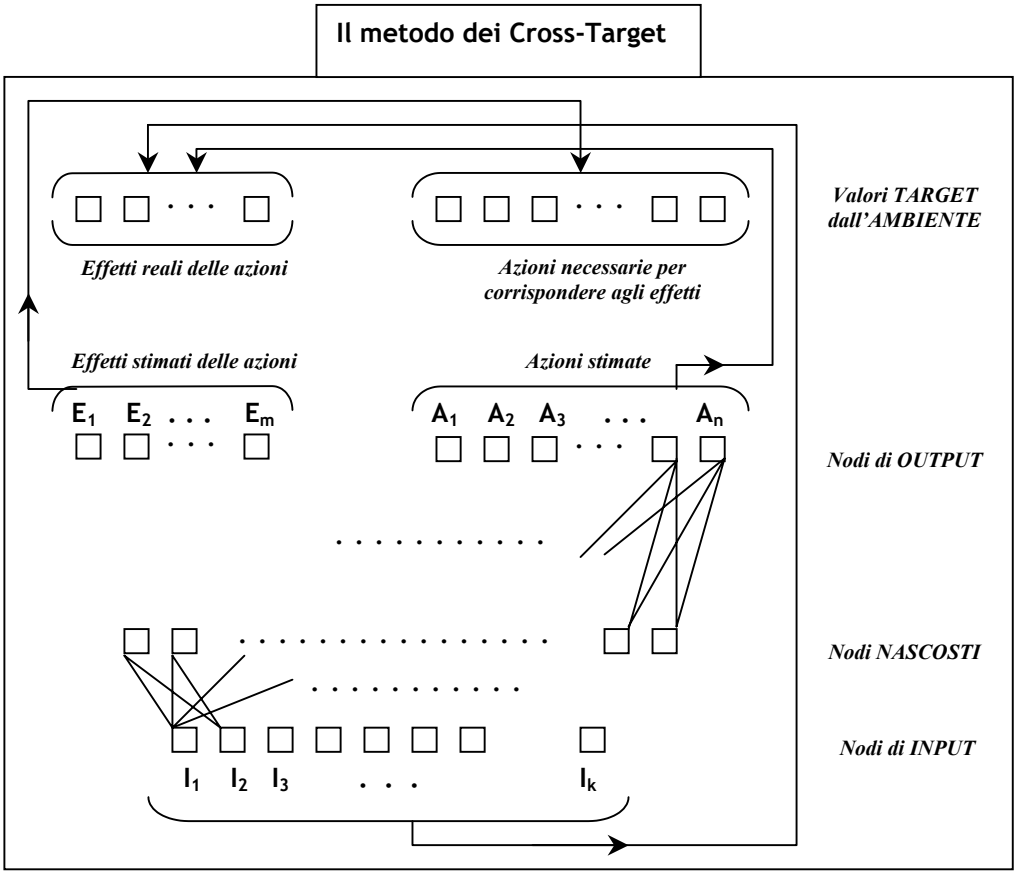
1. nodi di uscita relativi alle azioni da compiere;
2. nodi di uscita relativi agli effetti di tali azioni.

In entrambi i casi sono effettuate congetture di merito. Con i *CT*, sia i *target* per l'apprendimento della rete necessari dal lato delle azioni, costruiti in coerenza con gli *output* della rete concernenti gli effetti, sia quelli necessari dal lato degli effetti, costruiti in coerenza con gli *output* della rete relativi alle congetture di azione, sono determinati in modo incrociato. L'agente può quindi sviluppare la capacità di decidere azioni che

producano i risultati attesi e, allo stesso tempo, in coerenza con le stime degli effetti che le azioni comportano.

Utilizzando questo meccanismo di apprendimento, è possibile condurre esperimenti senza dover assumere a priori troppe ipotesi economiche. Sulla base di un apparato elementare, gli agenti sono in grado di produrre risultati complessi che, ad un osservatore esterno, appaiono come il frutto della pianificazione e del perseguimento di obiettivi specifici di agenti intelligenti. Le capacità computazionali e le disponibilità informative dell'agente sono in linea con le ipotesi di razionalità limitata, e non vi è la definizione di alcun obiettivo o l'utilizzo di tecniche di massimizzazione.

La tecnica *CT* è sviluppata con l'utilizzo di una rete neurale artificiale a tre strati. L'*output* della rete è suddiviso in due parti. Da un lato vi sono le azioni che il soggetto congettura di compiere, mentre dall'altro vi sono le congetture in merito agli effetti delle azioni da compiere. La scelta dell'utilizzo della rete neurale è dovuta alle caratteristiche intrinseche di adattabilità di una funzione di questo tipo, ma gli stessi risultati potrebbero essere raggiunti con l'utilizzo di altre metodologie.



In figura è rappresentato uno schema che descrive le azioni di un agente artificiale adattivo che segue il metodo *CT*. La giornata dell'agente è composta di quattro fasi che consistono in un ciclo completo della

produzione degli *output*, valutazione degli errori, retropropagazione degli stessi e correzione dei parametri della rete neurale artificiale.

Inizialmente la rete utilizza dei pesi casuali per produrre gli *output*. I pesi sono in seguito corretti con la *backpropagation*. Le quattro fasi della giornata dell'agente sono quindi le seguenti:

1. *output* della rete sulla base dei valori di *input* e dei pesi, che costituiscono le congetture su azioni ed effetti;
2. *target* per il lato sinistro della figura, la rete deve imparare a riprodurre le valutazioni degli effetti coerenti con le valutazioni sulle azioni;
3. *target* per il lato destro della figura, occorre che le azioni siano modificate in modo da renderle coerenti con le congetture sugli effetti;
4. *backpropagation*, si effettua infine l'apprendimento con la correzione dei pesi della rete in modo da ottenere stime di effetti più prossime alle conseguenze delle azioni congetturate, e congetture di azioni più coerenti con le stime sugli effetti.

Analizzando le quattro fasi è evidente da cosa derivi la denominazione *Cross-Target*: il processo di avvicinamento è duplice, in quanto avviene dagli effetti verso le azioni e viceversa, costruendo i *target* in modo incrociato.

L'apprendimento presenta la distinzione tra apprendimento a breve e a lungo termine, in analogia con la distinzione tra memoria a breve e a lungo termine. Gli agenti modificano continuamente i propri pesi, adattandosi in modo locale ai cambiamenti dell'ambiente. Inoltre è possibile introdurre un nuovo apprendimento sulla base dei dati storici ottenuti, in modo da avere una rete in grado di reagire correttamente a cambiamenti di rilievo che intervengano nell'ambiente, senza più modificare i pesi. Questo tipo di apprendimento può avvenire anche periodicamente, su un breve intervallo di dati oppure su serie storiche.

Tipologie di agenti cosi strutturati generano azioni complesse. Nella conduzione di esperimenti economici in Terna (2000c) si introducono degli obiettivi esterni che sostituiscono quelli costruiti in modo incrociato, i quali continuano ad essere utilizzati per le valutazioni necessarie e per le correzioni dal lato delle azioni. Per influenzare le congetture relative alle azioni si introducono proposte esterne che rappresentano uno dei *target* disponibili per l'apprendimento dal lato delle azioni, tra cui è scelto il maggiore in modulo.

LA VITA ARTIFICIALE

I calcolatori, secondo Langton (1992), possono essere utilizzati per riprodurre artificialmente dei fenomeni, ma senza un programma che specifichi loro cosa debbano fare non sono altro che "materia informe". Attraverso la programmazione è possibile fare emulare al calcolatore qualsiasi comportamento, a condizione che questo possa essere scomposto in una serie di semplici comandi.

I calcolatori possono essere definiti come macchine di secondo ordine: data la specificazione formale di una macchina (del primo ordine), essi "diventano" quella macchina. Da un calcolatore ci si può attendere due tipologie di limite:

1. la computabilità di principio;
2. la computabilità di fatto.

Con la prima espressione si intende l'impossibilità di riprodurre comportamenti perché incomputabili, mentre con la seconda si intende l'impossibilità di costruire una sequenza di passi che permetta al calcolatore di esibire un certo comportamento. In questi casi è possibile in linea di principio dare una specificazione formale di una macchina che manifesti un certo comportamento, ma non c'è alcuna procedura formale per produrre di fatto quella specificazione, se non una ricerca per tentativi nello spazio delle descrizioni possibili.

L'intelligenza artificiale e la vita artificiale sono entrambe interessate a utilizzare i calcolatori per lo studio di fenomeni naturali complessi, anche se utilizzano la tecnologia della computazione in modo radicalmente diverso.

L'intelligenza artificiale, per generare comportamenti intelligenti, usa la tecnologia della computazione come modello. La vita artificiale, invece, cerca di sviluppare un nuovo paradigma computazionale basato sui processi naturali che caratterizzano gli esseri viventi. Partendo dalle conoscenze biologiche, si esplora la dinamica dell'interazione tra le strutture dell'informazione. A differenza dell'intelligenza artificiale, non utilizza il paradigma computazionale come metodologia per la generazione di comportamenti, e non cerca di spiegare la vita in termini di programma informatico.

I calcolatori offrono un ambiente alternativo al laboratorio biologico per ricreare la vita. Lo studio della vita può essere affrontato con la costruzione di strutture di informazione, con il vantaggio di non essere legati a strumenti di laboratorio sofisticati, e soprattutto di non dover studiare le parti di un sistema in maniera separata per poi ricomporlo e osservarne il comportamento nel suo complesso.

L'utilità dell'utilizzo delle simulazioni e dei calcolatori è cruciale per lo studio di sistemi non lineari. Infatti, anche pervenendo ad una comprensione completa delle singole parti costruttive in isolamento, non si potrebbe affermare di avere compreso il sistema totale. Le proprietà di un sistema complesso sono proprietà delle interazioni tra le parti e non delle parti in se stesse. Le proprietà del sistema totale scompaiono quando si studiano separatamente le sue parti, quindi i fenomeni complessi non possono essere studiati come insieme di fenomeni semplici.

Per lo studio dei fenomeni non lineari è possibile utilizzare come metodologia di studio la sintesi oltre che l'analisi: invece di partire dal comportamento del sistema e cercare di analizzarlo nelle sue parti costitutive, è possibile riprodurre le diverse parti per poi comporle allo scopo di sintetizzare il comportamento oggetto di studio.

La vita è una proprietà della forma, non della materia: i singoli componenti di un corpo umano, come ad esempio aminoacidi e nucleotidi, non hanno vita propria, ma combinati in un certo modo interagiscono generando un comportamento dinamico che si identifica come "vita". La vita è un tipo di comportamento e non un tipo di materia e come tale è formata da comportamenti semplici che sono parti fondamentali di un sistema non lineare e che dipendono da interazioni non lineari tra parti fisiche. Sono quindi parti "virtuali" che dipendono dall'interazione di parti "fisiche" e lo studio separato delle seconde provoca l'interruzione del funzionamento delle prime. La vita artificiale è alla ricerca di queste "parti virtuali" e la sintesi è il suo principale strumento metodologico di ricerca.

La complessità dei comportamenti globali è solitamente dovuta ad interazioni non lineari che hanno luogo a livello locale. Qundi è più facile generare comportamenti complessi applicando semplici regole locali piuttosto che applicando complesse regole globali. Un sistema complesso presenta un elevato numero di stati globali, i quali devono essere classificati tenendo conto che piccolissime differenze nelle condizioni inziali possono portare a differenze enormi nel comportamento globale.

La componente "artificiale" della "vita artificiale" riguarda la ricostruzione del sistema che si studia, ma non i processi emergenti. Se le diverse parti del sistema sono sviluppate correttamente, i fenomeni che derivano dalla loro interazione saranno simili a quelli osservati nella realtà. La somiglianza o meno con i fenomeni reali non compromette la loro genuinità. L'organismo artificiale che ne deriva sarà non meno "vivo" dell'organismo naturale che cerca di riprodurre. L'unica differenza è la materia di cui i due organismi, quello naturale e quello artificiale, sono costituiti.

Gli studi nel campo della vita artificiale hanno reso possibile lo sviluppo di algoritmi di calcolo che attribuiscono capacità di apprendimento

ad un sistema in base alla sua evoluzione genetica. L'idea di base è l'evoluzione darwiniana: l'ambiente in cui il sistema opera non fornisce indicazioni su come modificare il proprio comportamento per adattarsi ad una situazione, ma permette la sopravvivenza solamente ai sistemi che presentano determinate caratteristiche e che si adattano meglio all'ambiente. Due applicazioni pratiche di questi metodi sono gli algoritmi genetici e i *classifier system*.

Gli algoritmi genetici

Gli algoritmi genetici, formalizzati per la prima volta da Holland (1975), sono metodi per la soluzione di problemi che si ispirano al paradigma evolutivo. In un algoritmo genetico ogni individuo è rappresentato da una sequenza di valori 0 e 1 che codifica le caratteristiche e le strategie adottate per la risoluzione di determinati tipi di problemi. Ad ogni individiuo è assegnato un valore che quantifica il successo ottenuto nella risoluzione dei problemi, detto *fitness*.

Nel tempo è data la possibilità di riprodursi solamente agli individui con i migliori valori di *fitness*, in modo che prolifichino solamente le categorie di individui che sono portatori di strategie vincenti.

Le categorie di individui con i valori di *fitness* peggiori sono invece selezionati per l'estinzione. In questo modo si eliminano dal sistema gli individui che applicano regole che portano a scarso successo.

Nella riproduzione, in perfetta analogia con quanto avviene in natura, la prole è generata attraverso la combinazione del patrimonio genetico dei genitori, costituito da parti delle stringhe numeriche che li rappresentano. L'incrocio (*crossover*) si basa sull'individuazione, in modo casuale, di una porzione qualunque, interna alla stringa, e sul successivo scambio delle parti a sinistra fra i due individui coinvolti nell'operazione. La conoscenza esistente, costituita dai genitori, è utilizzata per produrre nuova conoscenza.

Inoltre, nel processo è introdotta una mutazione che consiste nell'inversione di valore (da 0 a 1 o viceversa) di una o più posizioni della stringa prese casualmente (con un certo livello di probabilità). La popolazione iniziale del sistema può essere generata a caso a condizione che almeno il valore di *fitness* di un individuo differisca dagli altri.

Dopo un certo numero di riproduzioni ed estinzioni, si nota che larghe quote di popolazione convergono ad un unico tipo che presenta un alto livello di *fitness*, ossia una strategia che porta a buoni risultati nell'ambiente in cui è applicata.

I classifier system

I *classifier system* utilizzano la rappresentazione di regole in stringhe di 0 e 1. Ad ogni regola corrisponde uno schema antecedente, o condizione, e uno schema conseguente, detto azione. In analogia con la *fitness* degli algoritmi genetici, ogni regola è in possesso di un patrimonio di crediti che gli sono attribuiti in base ai risultati ottenuti effettuando le azioni da loro consigliate.

Tramite i cosiddetti *detector* sono codificate le informazioni provenienti dall'ambiente, mentre tramite gli *effector* si trasferiscono all'ambiente le decisioni prese. Le regole che più si adattano alle condizioni provenienti dal *detector* sono attivate in base alla somiglianza tra la sequenza di cui è costituita e la sequenza prodotta dal *detector* in base alle condizioni ambientali. Ogni regola attivata concorre per ottenere il diritto ad immettere un messaggio, corrispondente alla descrizione dell'azione da compiere, che entra a fare parte di una lista. L'*effector* provvede alla selezione del messaggio da inviare all'ambiente, scegliendolo tra quelli presenti nella lista. La partecipazione a questa "asta" comporta un costo in termini di crediti per le singole regole. Quando il messaggio di una regola arriva all'ambiente, la regola è remunerata con un numero di crediti proporzionale al successo ottenuto dall'azione consigliata.

Come per gli algoritmi genetici, si effettua di tanto in tanto una generazione di nuove regole, che accoppia tra loro due regole esistenti tramite *crossover* e mutazioni. La riproduzione o l'estinzione dipende dal patrimonio crediti delle regole.

A differenza degli algoritmi genetici si ha l'evoluzione esplicita di strategie, che non sono più mascherate in agenti, con l'evoluzione dell'intera specie più che del singolo individuo. In questo modo è possibile evolvere specie diverse che seguono gruppi di regole differenti e sono quindi in grado di affrontare diverse classi di problemi.

LA COSTRUZIONE DI UN ROBOT OECONOMICUS

Tutti gli studi esaminati in questo capitolo si presentano come strumenti innovativi a disposizione dell'economista. Margarita (1992) descrive le tappe fondamentali attraverso cui si è passati con lo sviluppo delle diverse discipline legate all'intelligenza artificiale. Inizialmente si affermano i metodi simbolici, i quali sono stati prevalentemente rivolti all'economia aziendale (ad esempio, i sistemi di supporto alle decisioni) e presto soppiantati dalle metodologie connessioniste, che permettono di introdurre nei modelli agenti che agiscono in base a forme di apprendimento autonomo

e per cui non è più necessario specificare dall'esterno le condizioni dell'ambiente che avrebbero dovuto affrontare.

Dalla separazione netta delle due metodologie, nel tempo è avvenuta una sempre maggiore integrazione, a cui si è giunti passando attraverso una fase di "confusione". Pur persistendo alcune incompatibilità di fondo, in particolare tra connessionismo e cognitivismo, reti neurali artificiali, algoritmi genetici, *classifier system*, tecniche di ottimizzazione e simulazione sono sempre più integrate in un sistema di strumenti a disposizione degli economisti.

Nel metodo classico seguito dall'econometria si parte dalla scelta del modello, si definisce la forma, lineare o non lineare, delle equazioni che definiscono i legami tra le grandezze coinvolte, si stimano i parametri sulla base dei dati a disposizione e, in un secondo momento, si valuta la validità interpretativa o le possibilità di previsione del modello stesso. Adottando questi strumenti innovativi, si ha un rovesciamento del metodo classico: la struttura del modello non è definita a priori, ma in modo automatizzato a posteriori in base ai dati a disposizione.

L'applicazione di queste metodologie si rivela particolarmente conveniente per la stima di parametri di modelli non lineari. Solitamente infatti, per determinare analiticamente il comportamento di sistemi complessi, in cui si hanno interazioni non lineari tra le variabili dipendenti, si utilizzano insiemi di equazioni differenziali o alle differenze.

Inoltre la soluzione trovata è sub-ottimale e in economia ciò è coerente con le teorie della razionalità limitata ed i modelli di comportamento degli agenti economici, che in questo modo risultano dotati di un sistema decisionale approssimativo, ma molto più realistico e rapido. I meccanismi evolutivi sono molto "meno matematici" dei metodi classici ma allo stesso tempo sono affini all'economia quantitativa.

Le reti neurali, gli algoritmi genetici e i *classifier system* costituiscono le componenti per la costruzione di un *robot oeconomicus*. Rimangono comunque da superare le seguenti questioni:

a) lo sfasamento temporale tra apprendimento e azione, che andrebbe annullato con la costruzione di sistemi che apprendono nell'istante stesso in cui compiono un'azione, e non necessitano di cicli di apprendimento, come le reti neurali, o di determinati periodi di evoluzione, come gli algoritmi genetici e i *classifier system*;

b) occorre dirigere le ricerche per lo sviluppo di sistemi di apprendimento autonomo, che quindi non necessiti di un supporto di dati fornito a priori, come avviene per le reti neurali;

c) mancano ancora gli strumenti per una comprensione più approfondita dei modelli stessi, per evitare le fondate critiche di scarsa trasparenza delle reti neurali, e i tentativi di interpretazione, come quello di Parisi (1989), non si rivelano ancora sufficienti;

d) occorre un maggiore interesse degli economisti agli sviluppi dell'intelligenza artificiale, che potrebbe innalzare il potenziale di ricerca dell'economia quantitativa grazie allo sviluppo di strumenti di più ampio respiro rispetto a quelli tradizionali e che non presentano particolari difficoltà di utilizzo.

La simulazione e gli agenti "cognitivi"

Ferraris (2000) osserva che nelle simulazioni ad agenti possono essere impiegati agenti che sono in grado di sviluppare strategie proprie e comportamenti individuali. In questo modo si possono analizzare i diversi comportamenti che si sviluppano in determinate condizioni ambientali. Il realismo dei modelli è notevolmente migliorato, anche se occorre tenere conto delle limitazioni dovute all'ampiezza dell'insieme di regole di cui gli agenti dispongono.

Gli agenti risultano composti di un genotipo, formato dalle opinioni e dai loro processi mentali, e da un fenotipo, costituito dall'aspetto fisico del singolo agente. L'ambiente è in continua evoluzione e ogni agente è chiamato ad effettuare sempre nuove inferenze in base alle condizioni in cui si trova. Il meccanismo di *crossover* e la mutazione favoriscono la nascita di individui con genotipi sempre nuovi e possono portare al miglioramento delle strategie già presenti nell'ambiente simulato.

L'utilizzo degli algoritmi descritti nei paragrafi precedenti per la costruzione di agenti offre alla simulazione economica sconfinate possibilità di studio. In Terna (2000a) si analizza un mercato simulato in cui operano agenti che sono in grado di sviluppare autonomamente delle regole di comportamento con il metodo dei *Cross-Target*. In Terna (2001) sono descritti gli agenti del modello *Surprising (Un)realistic Market* (*SUM*) che operano secondo il metodo *CT* (si veda la descrizione del capitolo 5), dimostrando come in agenti artificiali possa emergere autonomamente una forma di comportamento intelligente che provoca fenomeni osservabili anche sul mercato reale come le bolle e i *crash*.

CAPITOLO 2

SIMULAZIONE E STUDIO DELL'ECONOMIA

La simulazione si pone come nuovo strumento a disposizione degli studiosi di numerose discipline e come metodo per lo studio di fenomeni complessi. In questo capitolo si analizzano i diversi metodi di studio adottati dai due diversi gruppi di discipline, conosciuti come "scienze della natura" e "scienze dell'uomo"; si descrivono le diverse concezioni di "razionalità" che sono a supporto dei diversi modelli economici; si espongono le principali critiche di cui è oggetto l'economia neoclassica; si descrive, infine, come la simulazione possa essere utilizzata per lo studio dei fenomeni economici.

SCIENZE DELL'UOMO E SCIENZE DELLA NATURA

L'osservazione empirica è alla base della scienza e i ricercatori osservano la realtà in modo sistematico, accurato e professionale, per scoprire se le idee da cui sono partiti sono fondate o no, e per arrivare a conclusioni il più possibile oggettive. Per completare il quadro, oltre all'osservazione e alla descrizione accurata della realtà, occorre considerare le teorie che spiegano i fatti. In questo modo si arriva alla vera comprensione della realtà e non ci si ferma alla sola conoscenza. Una teoria scientifica è un insieme di concetti e idee con il quale lo scienziato cerca di individuare quali siano le entità, i meccanismi e i processi che stanno dietro ai fenomeni. Da non confondere con le "teorie ingenue" del senso comune, ossia le teorie filosofiche che cercano di interpretare tutta la realtà, oppure le teorie derivanti da una determinata religione, in cui vi è una visione ristretta e molto soggettiva dei fatti, senza che necessariamente esista un supporto empirico, caratteristico della scienza. Se una teoria scientifica risulta confermata dalle osservazioni empiriche, sarà possibile formulare previsioni attendibili.

Nelle scienze dell'uomo, i ricercatori rischiano di lasciarsi guidare dal desiderio che i fenomeni che studiano siano "in un certo modo", perdendo l'oggettività che sta alla base della scienza. Secondo Parisi (2001), per evitare questo rischio, la scienza ha a disposizione tre strumenti che sono il carattere collettivo della ricerca, il linguaggio quantitativo dei numeri e gli esperimenti di laboratorio. I vari studi condotti dai singoli ricercatori sono sotto il costante controllo dell'intera comunità scientifica, che ha il diritto e il dovere di mettere alla prova le teorie proposte. Le teorie solitamente puntano sull'aspetto quantitativo, limitando descrizioni della realtà puramente qualitative. Gli esperimenti di laboratorio sono lo strumento essenziale affinché possano avvenire delle verifiche empiriche delle teorie formulate. Si osservano i fenomeni in condizioni controllate, manipolando le condizioni e i fattori per osservare le conseguenze. Ogni esperimento deve essere ripetibile anche da parte di ricercatori differenti. A volte però non è possibile riprodurre in laboratorio determinati fenomeni e quindi si è costretti a condurre indagini direttamente nella realtà, senza poter isolare determinate variabili o effettuare semplificazioni al sistema osservato.

L'idea moderna di scienza, così come è stata appena descritta, nasce nel Seicento, con l'opera di personaggi come Galileo e Newton, e conduce le scienze della natura, come la fisica, la biologia e la chimica, ad una grande serie di successi. Lo stesso non si può dire delle scienze dell'uomo. Visti i risultati nelle scienze della natura, si è cercato di estendere il metodo scientifico per studiare i fenomeni umani, la mente, le società, le culture, le economie e i sistemi politici, ma non si sono raggiunti i livelli di soddisfazione sperati. Le scienze dell'uomo si sviluppano a partire dall'Ottocento, ma raramente si sviluppa un dialogo tra le teorie e i fatti osservati; molte volte le teorie si fermano ad aspetti qualitativi senza la possibilità di utilizzare strumenti come il laboratorio sperimentale.

Parisi (2001) sostiene che l'arretratezza delle scienze dell'uomo rispetto alle scienze della natura possa essere eliminata, o almeno ridotta, con l'aiuto delle simulazioni.

COSA SONO LE SIMULAZIONI

La simulazione rappresenta un nuovo strumento a disposizione della scienza per conoscere e comprendere la realtà, in via di diffusione in tutte le discipline.

Le teorie scientifiche sono concetti e idee attraverso i quali si cerca di individuare e spiegare meccanismi, processi e fattori su cui si fandano i fenomeni osservati nella realtà. Concetti e idee sono successivamente espressi in modo concreto e percettibile attraverso l'utilizzazione di rappresentazioni simboliche. Alcune teorie possono essere espresse

attraverso i simboli del linguaggio, mentre altre possono essere espresse tramite i simboli quantitativi della matematica o tramite simboli e schemi grafici. Le simulazioni non sono altro che un nuovo e rivoluzionario metodo per esprimere delle teorie. La rivoluzione sta nel fatto che non si utilizzano simboli per esprimere una teoria, ma si utilizza un programma per computer e il computer riproduce i fenomeni che la teoria intende spiegare. Anche in questo caso si utilizzano dei simboli, per la costruzione del programma, ma la differenza fondamentale è che questi sono diretti ad una macchina (il computer) e non ad altri esseri umani, come avviene nelle teorie tradizionali. Affinché la teoria-simulazione svolga predizioni e spiegazioni non è necessario che alcun essere umano comprenda l'insieme di simboli che costituiscono il programma sottostante. Il ricercatore deve solo osservare i fenomeni simulati, che costituiscono le predizioni della teoria, e confrontarli con quelli osservati nella realtà.

Le simulazioni possono essere viste come laboratori sperimentali virtuali e, proprio come avviene in quelli reali, i fenomeni virtuali che lo scienziato osserva durante la simulazione si verificano in condizioni che sono da lui controllate. Attraverso la manipolazione delle condizioni iniziali si analizza come queste determinano o influenzano il verificarsi dei fenomeni, per poi effettuare la registrazione dei dati relativi al sistema osservato.

Con le simulazioni è possibile derivare predizioni empiriche dalle teorie. Il processo di derivazione risulta automatizzato attraverso l'utilizzo del computer. Gli stessi fenomeni virtuali risultanti dalla simulazione sono le predizioni empiriche che varieranno in funzione delle manipolazioni effettuate dallo scienziato sulle variabili rilevanti.

Un'altra importante funzione delle simulazioni è la possibilità di effettuare esperimenti mentali. Un importante contributo delle simulazioni si ha in fase di costruzione di una teoria, perché con esse si automatizzano gli esperimenti mentali, che stanno alla base del metodo di ricerca della scienza. Il lavoro mentale che lo scienziato compie in fase di elaborazione di una teoria si modifica, perché la teoria è elaborata insieme alla simulazione, la quale confermerà empiricamente le diverse intuizioni e previsioni dello scienziato. Un esperimento mentale è un esperimento non condotto realmente, osservando e manipolando la realtà, bensì immaginato. Con il computer e le simulazioni si ampliano le possibilità di sperimentazione "mentale", che sono indispensabili per la sperimentazione "reale" successiva. L'esperimento mentale così concepito diventa uno strumento centrale nella ricerca scientifica e, con l'ausilio informatico, si libera delle restrizioni dettate dalle capacità mentali del singolo scienziato.

Con le simulazioni, quindi, si ha a disposizione un potente strumento per sviluppare e verificare le teorie scientifiche, ma si ha anche la

possibilità di creare nuove "realtà artificiali". Parisi (2001) definisce la realtà come:

a) ciò a cui abbiamo accesso con i nostri sensi;

b) ciò su cui possiamo agire e che risponde alle nostre azioni;

c) ciò che costituisce un vincolo, un limite alle nostre azioni, e nello stesso tempo un mezzo per svolgerle.

La realtà artificiale non è altro che un prodotto delle azioni degli esseri umani, come ad esempio la tecnologia, le modificazioni dell'ambiente, gli artefatti artistici e i segnali comunicativi. La simulazione risponde appieno alle tre caratteristiche che definiscono la "realtà": è percepibile attraverso lo schermo del computer, sono possibili interazioni tramite i comandi e rende possibili determinate azioni. In questa prospettiva le simulazioni sono da un lato "teorie" e dall'altro "realtà".

La somiglianza con i "modelli fisici" è molto alta: in entrambi i casi si ricrea una realtà manipolabile semplificata, da cui è possibile derivare in modo "meccanico" una gran quantità di predizioni. La qualità che distingue la simulazione è la mancanza di vincoli "fisici" e la possibilità di sviluppare modelli dinamici che fondono insieme modelli e formule matematiche.

VANTAGGI E PROBLEMI DELLE SIMULAZIONI

Per descrivere a fondo le simulazioni come strumento scientifico, è utile analizzare i vantaggi e gli svantaggi del loro utilizzo.

Il primo vantaggio delle simulazioni è quello di limitare gli errori nel derivare le previsioni empiriche, perché i processi sono meccanizzati all'interno di un computer. I risultati raggiunti sono dovuti alla teoria espressa nel programma di simulazione.

Il secondo vantaggio è che con la simulazione si ha la garanzia che la teoria non contenga concetti vaghi, parti poco chiare o mancanti. Il programma di computer deve incorporare tutte le parti della teoria, altrimenti sarà modificato fino al raggiungimento dei risultati sperati.

Le simulazioni possono inoltre essere considerate "laboratori sperimentali virtuali" in cui lo scienziato ha molte possibilità in più rispetto a quelle offerte in un laboratorio reale. Innanzitutto si riproduce virtualmente un ambiente, evitando quindi le difficoltà pratiche che questo potrebbe comportare. Si pensi ad esempio agli studi sui fenomeni atmosferici o tettonici, come uragani o terremoti, difficili o impossibili da riprodurre, oppure a simulazioni che comporterebbero periodi di osservazione troppo lunghi, come ad esempio le variazioni climatiche o di crescita delle popolazioni. La creazione di un ambiente virtuale inoltre dà la possibilità di

riutilizzare numerose volte l'ambiente che si crea per la simulazione. Poi si ha la possibilità di effettuare misurazioni in modo automatizzato, potendo così raccogliere una consistente mole di dati.

Nel laboratorio reale fenomeni complicati devono essere studiati in piccole parti separate, mentre nel laboratorio virtuale non è necessario e ciò può portare a risultati complessivi migliori della somma di tanti piccoli esperimenti separati.

Con le simulazioni, le teorie sono sottoposte a una verifica "interna" e una verifica "esterna". La verifica interna di una teoria mira a stabilire se la teoria spieghi effettivamente determinati fatti empirici, e i fatti simulati discendono direttamente dalla teoria sottostante, mentre la verifica esterna consiste nel confronto tra le previsioni effettuate con il modello simulato e la realtà.

Occorre inoltre considerare i vantaggi derivanti dalla creazione di "mondi possibili". Cercare di ricreare al computer parti del mondo reale con le dovute semplificazioni, significa effettivamente creare un mondo "parallelo" a quello reale. Risulta dunque possibile creare mondi artificiali, con lo studio dei quali si può giungere alla comprensione del mondo reale, oppure mettere alla prova delle teorie in situazioni artificiali.

Il metodo di ricerca basato sulla simulazione avvicina tra loro diverse discipline scientifiche. Con le simulazioni si eliminano molti dei problemi che hanno causato nel tempo la suddivisione tra le discipline. La grande varietà dei fenomeni osservabili nel mondo reale porta i ricercatori alla specializzazione per tipo di problemi, soprattutto per questioni di memoria e per la diversità nel metodo di studio. Con i moderni strumenti informatici, la memoria limitata non costituisce più un intralcio perché la scienza diventi interdisciplinare, e la simulazione si propone come metodo di studio adatto alla soluzione di qualsiasi tipo di problema. La simulazione può quindi essere vista come un "linguaggio comune" a qualsiasi disciplina.

Se tanti sono i vantaggi riscontrabili nell'utilizzo delle simulazioni nella ricerca scientifica, numerose possono essere le critiche al suo utilizzo. I problemi delle simulazioni sono suddivisibili in due classi. La prima comprende i problemi "apparenti", che sono una serie di problemi derivanti da una mancata comprensione della loro natura, mentre la seconda comprende i reali problemi delle simulazioni.

Le principali critiche che solitamente sono rivolte alle simulazioni, e che derivano dalla loro scarsa diffusione e comprensione, sono le seguenti:

- risultano troppo semplificate rispetto alla realtà;
- sembrano non dire nulla di nuovo rispetto a ciò che si conosce del sistema che è simulato;

- è improbabile che si riesca a simulare una realtà, se prima non la si conosce a fondo;
- anche se riproducono un fenomeno con successo, non necessariamente ne facilitano la comprensione;
- gli utilizzatori solitamente danno maggior peso alla verifica interna delle teorie, piuttosto che alla verifica esterna;
- possono essere inclusi in una simulazione elementi superflui rispetto alla questione che si studia;
- confusione tra scopi di conoscenza e scopi pratici.

La prima delle critiche elencate è basata sull'errata comprensione di cosa sia una simulazione. Qualsiasi teoria che cerchi di spiegare la realtà presenta delle semplificazioni e le simulazioni non sono altro che teorie sotto forma di programma per computer. L'importante è che le semplificazioni siano quelle giuste, necessarie per ridurre all'essenza il fenomeno da studiare e facilitarne la comprensione. Una critica del genere calza con qualsiasi teoria, la differenza è che, quando si parla di simulazioni, è facile pensarle come "giochi per computer", invece di modelli quali sono. Se si considera la simulazione come un modo per ricreare la realtà, maggiori sono i dettagli e maggiore è la qualità del modello. In ogni caso lo sviluppo dipende solamente dalle capacità del sistema informatico su cui si opera e le semplificazioni possono essere eliminate in diverse fasi successive.

Effettuando le semplificazioni per riprodurre artificialmente dei fenomeni, il rischio maggiore, che rimane in parte anche se si utilizzano strumenti di indagine scientifica tradizionali, è quello di effettuare le semplificazioni sbagliate, escludendo dalla simulazione alcuni degli aspetti rilevanti del problema. Il problema può però presentarsi in forma speculare, cioè la considerazione di aspetti non essenziali complica il modello senza migliorare la somiglianza dei risultati reali con quelli artificiali.

La seconda critica si confuta facilmente se si pensa alla grande varietà di esperimenti che si possono compiere in un ambiente simulato, che consentono di approfondire le conoscenze e di comprendere come le singole variabili influenzino lo stesso sistema. Inoltre, attraverso il confronto dei dati empirici reali e dei risultati empirici simulati, è possibile testare il livello di validità della teoria in base alla quale si è costruita la simulazione. Il caso limite è quello in cui i risultati simulati coincidono con i risultati riscontrati nella realtà.

Con la terza critica, si tocca uno dei fini primari della simulazione. Se di un sistema si conoscessero i minimi particolari e se lo si fosse compreso a fondo, non vi sarebbe motivo di simularlo. L'aiuto fondamentale fornito dalle simulazioni è proprio quello di aumentare la conoscenza e la comprensione

di un determinato fenomeno, da cui possono derivare stimoli alla ricerca per verificare se situazioni riscontrate nel mondo simulato possano o meno riscontrarsi nel mondo reale.

Le simulazioni sono criticate per le difficoltà con cui ci si scontra se si cerca di comprendere il codice informatico di cui sono costituite, che rende oscuro il meccanismo attraverso cui la simulazione produce dei risultati. Tuttavia sono difficoltà superabili con un minimo di esperienza o con l'affidamento a documentazioni che spieghino in termini non informatici ciò che succede all'interno della simulazione.

Gli utilizzatori delle simulazioni tendono a preoccuparsi maggiormente di capire se una teoria spieghi effettivamente certi fatti empirici, invece di stabilire se le previsioni derivate da una teoria corrispondano ai fatti empirici osservati nella realtà. Affinché la simulazione possa diventare uno strumento credibile per la scienza occorre che vi siano maggiori verifiche esterne, ossia occorre che vi sia un esplicito, dettagliato e ampio confronto tra i risultati delle simulazioni e l'evidenza empirica.

Vi è il rischio che si tralasci l'aspetto scientifico per l'aspetto pratico e si attribuisca successo ad una simulazione solamente se la si collega ad un determinato successo tecnologico. In questo caso si incorre in una confusione tra gli scopi per cui si utilizza la simulazione. I successi della scienza non vanno comunque valutati solamente in funzione delle applicazioni tecnologiche che ne derivano.

Utilizzare la simulazione come strumento di indagine scientifica significa considerare la realtà non come insieme di sistemi semplici, come di solito avviene con i metodi di studio tradizionali, bensì come insieme di sistemi complessi. Senza l'utilizzazione dei computer e delle simulazioni, lo studio dei fenomeni complessi non sarebbe possibile. Infatti le teorie tradizionali e il laboratorio sperimentale, a differenza di quello virtuale, risultano non adatti per lo studio di sistemi complessi. Nei laboratori tradizionali, lo scienziato può solamente analizzare e manipolare i fenomeni studiati, che rimangono comunque delle "scatole nere", mentre nella simulazione i fenomeni sono ricostruiti nel programma informatico. La differenza essenziale rimane quella tra analisi dei fenomeni, condotta dalle metodologie tradizionali, e sintesi dei fenomeni; la simulazione infatti parte dalle componenti del fenomeno per studiare come queste, interagendo fra loro, provocano le proprietà del sistema nel suo complesso. La simulazione, in questo modo, avvicina le diverse scienze che si trovano a dover trattare fenomeni complessi e fornisce loro un valido strumento di studio.

Simon (1988) afferma che il calcolatore ha notevolmente esteso la gamma di sistemi di cui è possibile imitare il comportamento attraverso la simulazione. La simulazione è comunque più antica dei calcolatori numerici.

Le principali critiche che a suo parere si possono rivolgere alla simulazione sono:

a) una simulazione non può valere al di là delle ipotesi su cui si fonda;

b) un calcolatore può fare solo quello per cui è stato programmato.

In molti casi si può prevedere il comportamento di un sistema alla luce dei suoi scopi e dell'ambiente esterno, avendo solamente una conoscenza minima del suo ambiente interno.

L'ambiente esterno determina le condizioni per il raggiungimento degli scopi. Se il sistema interno è ben progettato si adatterà all'ambiente esterno, cosicché il suo comportamento sarà in gran parte determinato da quest'ultimo, proprio come nel caso dell'uomo economico. Per predire il modo in cui si agirà basta chiedersi: "Come si comporterà in queste circostanze un sistema progettato razionalmente?". Il comportamento prende forma sulla base dell'ambiente-compito.

Gli obiettivi di un progetto non sempre sono realizzati pienamente. In tal caso le proprietà del sistema interno saranno soltanto intravedibili. Il comportamento del sistema risponderà soltanto in parte all'ambiente-compito dato che in parte sarà condizionato dai vincoli costituiti dalle proprietà del sistema interno.

La simulazione è utile quando in origine si sa poco delle leggi naturali che governano il comportamento del sistema interno. Innanzitutto quasi sempre si studiano e si è interessati solo ad alcune proprietà della complessa realtà. Non è quindi necessario conoscere tutti i particolari della struttura interna del sistema, ma solo la parte di esso che presenta importanza critica per l'astrazione.

I PROBLEMI DELLE SCIENZE DELL'UOMO

Parisi (2001) analizza la differenza tra i successi delle scienze della natura rispetto a quelli delle scienze che studiano l'uomo. Gli esseri umani , il loro comportamento, la loro vita mentale, le loro società e culture e il modo in cui queste cambiano nel tempo, sono fenomeni notevolmente più complicati dei fenomeni naturali. Le teorie, in campi come l'economia, la sociologia e le scienze storiche, sono notevolmente influenzati dalle opinioni e non garantiscono di riuscire effettivamente a comprendere i fenomeni. Inoltre manca l'interazione tra teorie e fatti empirici alla base di qualsiasi scienza.

Il problema si potrebbe trovare nel modo in cui le scienze hanno proceduto fino ad oggi e nei limiti degli strumenti concettuali e metodologici che hanno avuto a disposizione.

Le scienze dell'uomo si possono classificare in tre tipologie:

1. scienze con osservazioni empiriche, ma senza teorie;
2. scienze in cui manca il dialogo tra teorie e fatti empirici;
3. scienze con una limitata superficie di contatto tra teorie e fatti empirici.

Un esempio del primo tipo di scienza è rappresentato dalla storia, in cui fondamentalmente ci si limita al racconto di eventi passati in base alle tracce ritrovate, come documenti o reperti archeologici. In ogni caso mancano vere e proprie teorie, vista l'irripetibilità degli eventi passati.

Una scienza del secondo tipo è la sociologia, intesa come scienza che studia le società moderne, la quale è ricca di osservazioni empiriche ed elaborazioni teoriche, da cui raramente si giunge a una predizione specifica per poi poterla verificare empiricamente. Vi sono teorie che sono elaborazioni molto generali, con definizioni di termini e schemi interpretativi fortemente strutturati. Da queste teorie è molto difficile ricavare predizioni empiriche specifiche da confrontare con i fatti. In sostituzione della teoria, in questo campo sono frequenti le interpretazioni di specifici fatti osservati, illuminanti dal punto di vista intuitivo, ma lontane dalle teorie scientifiche tradizionali.

All'ultima tipologia appartengono scienze come l'economia, in cui le teorie sono formulate in modo esplicito e quantitativo, con una interazione che risulta limitata tra teorie e dati osservati. Le teorie non sempre riescono a spiegare la realtà e poggiano su ipotesi molto profonde che ne limitano la plausibilità e l'applicabilità, come ad esempio le ipotesi di razionalità degli agenti economici.

Altri problemi sono riscontrabili a livello di dati empirici, i quali non sempre sono facilmente reperibili e limitano l'applicabilità del metodo sperimentale alle scienze dell'uomo. All'interno della scienza economica, ad esempio, raramente è possibile effettuare esperimenti di laboratorio, rendendo possibile l'osservazione di persone che operano in condizioni controllate. Si possono individuare i seguenti motivi:

- non sempre è possibile isolare i fenomeni studiati dal contesto in cui sono inseriti perché sono il risultato di un grande numero di cause che interagiscono tra di loro in maniera non lineare;
- i fenomeni solitamente interessano spazi e durate nel tempo non riproducibili in laboratorio;

- i fenomeni tendono ad essere complessi e a non ripetersi in maniera identica o a non essere ripetibili;
- non è facile individuare le cause dei fenomeni e, di conseguenza, lo sperimentatore non può controllarle né manipolarle in laboratorio.

Tutti questi elementi portano le scienze dell'uomo ad essere considerate più qualitative che quantitative, e porta gli stessi ricercatori a preferire linguaggi e teoremi poco formali.

Un ultimo aspetto è il problema della suddivisione in discipline. A differenza delle scienze della natura sono molti i punti di contatto tra le varie discipline, che si trovano interessate a tipi di fenomeni molto simili al punto da rendere difficile, in alcuni casi, la ripartizione delle competenze. Ciò può portare a fenomeni di aggregazione delle discipline ai fini dello studio di una categoria di fenomeni, come è avvenuto per le scienze cognitive.

E' evidente come la simulazione, viste le caratteristiche analizzate nei paragrafi precedenti, possa venire incontro ai problemi delle scienze dell'uomo, sostituendosi al laboratorio sperimentale delle scienze della natura e rendendo possibile un avvicinamento tra le teorie e i fatti empirici. Inoltre risulta essere un metodo adatto allo studio dei fenomeni complessi che avvicina e porta ad interagire diverse discipline.

L'ECONOMIA NEOCLASSICA

Simon(1988) include anche l'economia tra i sistemi artificiali e la definisce come una teoria della razionalità umana che coinvolge sia la razionalità procedurale, intesa come insieme di regole necessarie per prendere decisioni, sia la razionalità sostanziale, intesa come capacità di trovare il modo corretto di procedere. Per analizzare un sistema economico occorre studiare il suo ambiente interno, in particolare i limiti e le capacità delle aziende e dei consumatori di procurarsi informazioni. Risulta di fondamentale importanza lo studio dei processi cognitivi. Occorre concentrare l'attenzione sui modi in cui si formano le attese e sugli effetti stabilizzanti o destabilizzanti derivanti dai tentativi degli attori di indovinare le intenzioni altrui.

Nell'economia neoclassica si idealizza la razionalità umana e si rivolge l'attenzione principalmente all'ambiente esterno del pensiero umano, ossia alle decisioni che sono ottimali per realizzare un sistema di scopi adattivo in cui si massimizza il profitto e l'utilità, cercando di definire decisioni razionali nelle circostanze determinate dall'ambiente esterno.

Ad esempio, in rapporto al meccanismo dei prezzi si assumono ipotesi molto più forti della semplice richiesta di "equilibrio dei mercati". Servendosi di ipotesi come la competitività perfetta o la massimizzazione dell'utilità da parte degli attori economici, è possibile mostrare che l'equilibrio prodotto dal mercato sarà ottimale, nel senso che non potrebbe essere spostato al fine di migliorare il benessere della collettività. I teoremi considerano sempre situazioni di "ottimizzazione" e non di "soddisfazione". Simon (1988) ritiene che sarebbe più produttivo concentrarsi sulle proprietà dei sistemi economici che permettono un semplice equilibrio dei mercati piuttosto che sugli assunti che occorre fare per ottenere l'ottimizzazione delle posizioni di equilibrio.

In economia è particolarmente evidente l'interazione tra ambiente interno e ambiente esterno. L'adattamento di un sistema intelligente al suo ambiente esterno è condizionato dalla sua abilità nello scoprire un appropriato comportamento adattabile. In altri termini, la razionalità sostanziale di un individuo, cioè la sua capacità di trovare il modo corretto di procedere, dipende dalla sua razionalità procedurale, intendendo con questo termine il possesso delle procedure per calcolare la natura di tale procedere.

Sempre Simon (2000) sottolinea che nell'economia classica e in quella neoclassica, obiettivi e motivazioni si considerano come date a priori, sotto forma di una funzione di utilità che consente all'individuo di compiere scelte coerenti tra tutte le combinazioni di beni e servizi. Inoltre si ipotizza che gli agenti economici preferiscano tra le scelte a loro disposizione quella che comporta la massima utilità. Si può ipotizzare che siano note le conseguenze delle scelte, oppure che si conosca la distribuzione di probabilità congiunte dei risultati e gli agenti siano in grado di calcolare quale alternativa generi la massima utilità.

L'obiezione principale all'economia neoclassica è se esiste la funzione di utilità e, se non esiste, se vale l'*as if*. La funzione di utilità non è fissa. Nuove esperienze creano nuovi gusti; sarebbe più opportuno considerare la funzione di utilità come una struttura in evoluzione. L'economia è la scienza che celebra la razionalità umana, ragion per cui emerge in ogni frangente del comportamento umano ed è la base operativa delle società create dall'uomo. E' stata erroneamente etichettata come una scienza "oscura", soprattutto perché nella sua forma ricardiana non offriva molte speranze al progresso umano. L'economia classica presenta l'umanità, individualmente e collettivamente, come in grado di risolvere dei problemi immensamente complessi di ottimizzazione dello stanziamento delle risorse. L'abilità dell'uomo economico gli permette di realizzare il migliore adattamento possibile ai suoi bisogni nell'ambiente in cui egli si trova. Una rappresentazione veritiera dell'uomo economico e delle istituzioni

economiche deve incorporare un elenco delle sue strategie di adattamento in relazione ai limiti di elaborazione di informazione imposti dall'ambiente interno.

Una tale rappresentazione deve anche conciliare sia la razionalità conscia di chi prende delle decisioni economiche, sia i processi evolutivi non pianificati ma adattabili che hanno forgiato le istituzioni economiche. La ricerca operativa e l'intelligenza artificiale hanno introdotto nuove tecniche che aumentano la razionalità procedurale degli attori economici e li aiutano a prendere decisioni migliori.

L'economia comportamentista

All'economia classica si contrappone l'economia comportamentista, che si occupa della validità empirica delle assunzioni neoclassiche sull'agire umano. Se queste si rivelano insostenibili, si preoccupa di trovare leggi empiriche che descrivano i comportamenti nel modo più corretto e accurato possibile. Un altro obiettivo dell'economia comportamentista è quello di analizzare le implicazioni delle divergenze delle assunzioni neoclassiche dal comportamento reale, per ciò che concerne il sistema economico. Un terzo scopo è quello di fornire una dimostrazione empirica della forma e del contenuto della funzione di utilità (o di una costruzione mentale equivalente), che serva a rafforzare le previsioni sul comportamento economico degli uomini.

L'economia comportamentista si propone non solo di proporre delle teorie, ma di verificare empiricamente le assunzioni neoclassiche sul comportamento umano, e di modificare la teoria economica sulla base di ciò che si scopre nel processo di verifica.

La ricerca comportamentista si occupa delle ipotesi di massimizzazione dell'utilità e dei profitti, e della possibilità di poterle sostituire con assunzioni alternative che descrivano meglio le motivazioni umane sul mercato. Poi vi è l'attenzione alla formazione delle decisioni in condizioni di incertezza, la capacità di determinare se gli agenti siano in grado di massimizzare l'utilità e se effettivamente la massimizzino, come sostiene l'economia neoclassica. Ci si occupa quindi sia delle motivazioni che delle capacità cognitive umane in materia economica per prendere decisioni ottimali.

Con "razionalità limitata", idea di base su cui poggia la corrente comportamentista, si intende l'insieme dei limiti della conoscenza umana e delle capacità umane di calcolo, che precludono la reale possibilità agli agenti economici di avvicinarsi alle previsioni della teoria classica e neoclassica. Alcuni dei limiti sono l'assenza di una funzione di utilità completa e coerente per ordinare tutte le possibili scelte, l'incapacità di

generare una serie completa di scelte adeguate e di prevederne a priori le rispettive conseguenze, l'incapacità di associare ad eventi futuri non certi un livello di probabilità.

L'espressione "razionalità limitata" si usa per designare una scelta razionale che prende in considerazione i limiti cognitivi del soggetto decisionale. La teoria neoclassica postula che vi sia una serie data e fissa di possibili scelte, con una distribuzione delle probabilità degli esiti di ciascuna scelta conosciuta, e che le scelte siano compiute in modo tale da massimizzare l'utilità. Le teorie della razionalità limitata possono essere generate abbandonando una o più assunzioni classiche.

I limiti cognitivi sono considerati sia come carenza di informazioni che come adeguatezza delle teorie scientifiche di cui avvalersi per prevedere fenomeni relativi. L'accuratezza delle previsioni per mezzo dei modelli a computer è seriamente limitata dalla mancanza di conoscenza dei meccanismi economici fondamentali rappresentati nelle equazioni dei modelli. In alcuni casi è possibile evitare di fondare i modelli sulle equazioni, come ad esempio accade nel modello di simulazione di borsa *SUM* (si veda il capitolo 5), ma riprodurre un sistema partendo dalla ricostruzione dei singoli individui per poi studiare l'effetto della loro interazione sul mercato.

Considerare la funzione di utilità significa accettare l'ipotesi che nella scelta umana vi sia un elevato grado di coerenza, cosa che nella realtà non sempre è riscontrabile. Senza considerare poi i pesanti oneri di calcolo che sono addossati all'agente economico.

Le ipotesi di razionalità limitata sono dunque ipotesi sul processo decisionale, che tengono conto delle capacità reali della mente umana su cui si indaga anche con la ricerca psicologica.

Contrapposti alla teoria neoclassica vi sono inoltre i fenomeni che Simon (2000) definisce "convenzionali", ossia che non poggiano sull'ipotesi di razionalità. Ad esempio, molti fenomeni sociali importanti come i salari, l'accesso ai capitali per la crescita o i tassi di risparmio non dipendono da calcoli razionali, ma sono determinati convenzionalmente, in base alla conoscenza di regole socialmente accettate.

Da tutti questi elementi prende spunto la "nuova economia istituzionale", che è il complesso dell'analisi economica moderna che si avvale di concetti quali informazioni limitate, costi di transazione, opportunismo, per spiegare fenomeni economici osservati. L'azione reale delle derivazioni non discende da assunzioni di razionalità bensì da ipotesi sui limiti delle informazioni o altri limiti della razionalità, e non si considerano più le ipotesi di razionalità perfetta e massimizzazione dell'utilità. Considerando un insieme discreto di alternative, ci si aspetta che

un agente che soddisfa il criterio del "soddisfacimento" (*satisficing*) scelga la migliore tra le alternative disponibili. Un soggetto che scelga la migliore alternativa a disposizione secondo un qualche criterio si dice che "ottimizza", mentre se sceglie un'alternativa che risulti superiore a determinati criteri specificati, si dice che "soddisfa". In molte situazioni reali non è possibile calcolare entro limiti di sforzo sostenibile l'ottimo, perché la complessità del mondo reale non si limita a migliaia di variabili e di vincoli, e nemmeno conserva sempre la linearità e la convessità che facilitano i calcoli. Di solito risulta più semplice soddisfare che ottimizzare e non necessita che sia postulata la coerenza del decisore nel tempo.

Simon (2000) critica Milton Friedman e i suoi sostenitori i quali affermano che una teoria economica va giudicata in base alla bontà delle previsioni che permette di realizzare e non sulla base della correttezza dei meccanismi che postula per produrre quei risultati. Quindi non è possibile, secondo Friedman (1953), giudicare una teoria in base al realismo delle ipotesi, ma soltanto in base all'efficacia delle previsioni. Ma la teoria neoclassica non porta spesso a previsioni corrette ed è debole anche considerando le potenzialità di calcolo moderne. Friedman cita Galileo, ad esempio, che trascurò la resistenza dell'aria per formulare la legge di caduta dei gravi, ma la sottopose a verifica tenendo conto delle condizioni limitanti, e così fecero anche i suoi successori, dimostrando che si può rinunciare al realismo solo se si ottiene in cambio un buon livello di approssimazione.

Una delle principali argomentazioni dei difensori dell'economia neoclassica è che gli agenti che si comportano in modo diverso dall'agente rappresentativo sono destinati ad essere eliminati dalla concorrenza. Queste considerazioni derivano da una lettura distorta della teoria darwiniana. La teoria evoluzionistica infatti non parla di soggetti massimizzanti ma di massimizzazione locale. La concorrenza non può eliminare imprese anche molto inefficienti se non esistono sul mercato imprese migliori. Nello studio degli aggregati, si postula l'individuo "rappresentativo" e lo si utilizza ai fini dell'aggregazione. I comportamentisti invece studiano su base empirica le unità da aggregare anziché postulare a priori le loro proprietà.

Secondo Simon (2000), coltivare la ricerca empirica è l'unico modo per ampliare i confini dell'economia come scienza e per accrescere la sua forza di previsione e di spiegazione. Nei modelli neoclassici difficilmente si possono trarre conclusioni senza servirsi di un elevato numero di ipotesi sulla funzione di utilità, e spesso sono proprio i limiti imposti alla razionalità a spiegare le differenze nelle previsioni derivate da diverse teorie economiche. Le simulazioni al computer possono fornire un potente strumento per collegare i diversi tipi di dati alla teoria.

Fondamenti metodologici dell'economia

Prima di parlare di aspetti metodologici occorre caratterizzare la disciplina economica. Seguendo l'impostazione di Simon (2000), gli aspetti più importanti dei fenomeni economici sono i seguenti:

- l'economia ha un aspetto sia descrittivo sia normativo, descrive come agiscono le persone e ne prescrive il comportamento razionale;
- è una "scienza dell'artificiale", si occupa di sistemi che cercano di adattarsi al proprio ambiente al fine di conseguire degli obiettivi;
- ponendo limiti alla capacità di adattamento di un sistema, è possibile prevedere appieno il suo comportamento esaminando gli obiettivi che persegue e la forma dell'ambiente al quale si adatta;
- le proprietà del sistema influenzano il comportamento solo in quanto limitano le capacità di adattamento all'ambiente e i limiti possono essere nelle capacità di calcolo oppure nelle capacità di realizzare i comportamenti calcolati;
- conviene tracciare un confine del sistema, in modo che tutti i limiti della capacità di compiere delle azioni si trovino all'esterno, mentre i limiti della capacità di calcolare si trovino all'interno del sistema, individuabili come cause delle deviazioni dalla razionalità.

I limiti nelle capacità di calcolo sono stati notevolmente trascurati dalle teorie economiche con la conseguenza dell'impossibilità di descrivere correttamente le faccende umane nella sfera economica e di prescrivere correttamente i comportamenti da adottare, tenendo conto del rapporto dei limiti delle capacità di calcolo umane con la complessità del mondo reale.

La teoria neoclassica non spiega l'origine delle scelte, né il contenuto della funzione di utilità, né le decisioni da prendere, né quali siano i mezzi e le capacità di calcolo dell'agente economico o il modo in cui mette in relazione le scelte con le loro conseguenze, misurate in termini di utilità. Essa deriva da una serie di ipotesi empiriche sul comportamento molto potenti. Le teorie della razionalità limitata poggiano invece su ipotesi più articolate che derivano da osservazioni della realtà più estese. Oggi si ha a disposizione un notevole corpus teorico, prodotto dalla ricerca delle scienze cognitive, che afferma che le nuove soluzioni si scoprono attraverso la ricerca euristica all'interno degli spazi del problema.

Nella maggior parte dei problemi del mondo reale il numero delle componenti e il modo in cui possono essere combinate fra loro è talmente elevato che non si può impostare il processo di ricerca di una soluzione sulla logica "prova ed errore" di tutte le possibilità. Occorre quindi che la ricerca sia guidata dalla conoscenza, in modo da restringere i campi di scelta e dirigere l'attenzione verso una serie ridotta di alternative.

Gli studi sul processo umano di soluzione di problemi dimostrano che la ricerca euristica è molto usata e soprattutto molto efficace. Gli esempi più famosi sono rappresentati da programmi per il gioco degli scacchi o programmi di supporto alla diagnosi medica. La logica di fondo è quella di restringere il campo di scelta ad una serie di alternative, escludendo dall'analisi le direzioni che non condurrebbero ad una soluzione, e da questo deriva la denominazione "sistemi esperti". Una componente importante della ricerca euristica è la "regola d'arresto", che consiste nello stabilire quando cessare la ricerca e prendere una decisione. Il risultato della ricerca non è necessariamente ottimale, ma si presenta come la scelta migliore effettuabile entro determinati livelli di costo. La ricerca quindi soddisfa, ma non necessariamente ottimizza. Sulla base dell'esperienza passata, chi deve risolvere un problema si forma un giudizio sulla qualità della soluzione che intende raggiungere con un investimento ragionevole in termini di sforzo. Le aspirazioni tendono ad alzarsi quando si rivela facile trovare nuove soluzioni migliori, e ad abbassarsi qualora la ricerca diventi improduttiva.

Nella soluzione di problemi attraverso la ricerca euristica, per stabilire se sia stata trovata l'alternativa accettabile, può essere utile un criterio di *satisficing*, che può avere proprietà e richiedere assunzioni molto più deboli di quelle richieste dall'economia neoclassica. Il livello di soddisfazione dipende dalle aspettative, che si abbassano finché non si trova un'alternativa. Simon (1988) definisce il criterio di *satisficing* come il "motore principale per la soluzione dei problemi dell'uomo".

L'individuo in economia e psicologia

Con le ipotesi di piena razionalità, non si rende necessario distinguere tra il mondo reale e la percezione che ne ha il soggetto decisore, perché egli percepisce il mondo esattamente come è. Inoltre potranno essere perfettamente previste le decisioni del soggetto razionale partendo dalla nostra conoscenza del mondo reale e della sua funzione di utilità, senza conoscere le percezioni o le modalità di calcolo del soggetto.

Se invece si considera un individuo con conoscenza e capacità di calcolo limitate, non è possibile ignorare la distinzione tra il mondo reale e le percezioni e le capacità di rielaborazione del soggetto. I nostri modelli devono prevedere e comprendere il modo di ragionare e i processi che generano una rappresentazione soggettiva del problema decisionale dell'agente. Nelle teorie dell'economia neoclassica, l'individuo raggiunge sempre la migliore decisione oggettivamente e sostanzialmente, data la funzione di utilità. Diversa risulta essere l'impostazione delle teorie della psicologia cognitiva, che considera l'individuo capace di prendere le sue decisioni in modo ragionevole alla luce delle conoscenze e dei mezzi di calcolo a disposizione.

Simon (1988) afferma che spesso gli economisti neoclassici supportano le teorie semplicemente modificando le ipotesi su cui si basano, e si rifiutano di osservare il mondo se prima non hanno una teoria su di esso, invece di partire dall'osservazione dei fenomeni reali. Il ragionamento dello psicologo è molto diverso e pone al centro l'osservazione dei fenomeni, rinunciando alla visione quantitativa della realtà.

Le ipotesi sulla razionalità

L'uomo risulta fortemente limitato dalle capacità della propria mente. Considerando i limiti della ragione si possono spiegare razionalmente molti comportamenti sia individuali che collettivi, altrimenti giudicabili incoerenti o inadeguati sotto diversi aspetti. Secondo Simon (1988), nelle scienze del comportamento umano, la razionalità assolve un ruolo analogo alla selezione naturale nella biologia.

Nell'ultimo mezzo secolo, si è formato un corpus teorico notevole attorno al concetto di ragione. Sono state formulate da studiosi di statistica ed economia numerose teorie, con l'idea di convogliare tutti i valori in un'unica funzione, la funzione di utilità, al fine di comparare valori tra loro disomogenei, anche se la comparazione è di fatto già effettuata quando si presuppone un certo grado di utilità assegnandolo ad un particolare stato di cose.

Simon (1984) classifica in quattro tipologie le diverse concezioni della razionalità e ne analizza i relativi modelli formali, che sono stati formulati nel corso del secolo XX. Il primo è il modello cosiddetto "olimpico", conosciuto con il nome di "teoria dell'utilità soggettiva prevista", nel quale l'uomo assume capacità quasi divine ed è in grado di effettuare scelte complete all'interno di un universo integrato. La teoria presume che il soggetto decisionale conosca la propria funzione di utilità, può quindi ordinare per livello di gradimento qualsiasi scenario di eventi futuri, e imposti le proprie decisioni in modo da massimizzare la stessa. Il soggetto è inoltre posto di fronte ad un gruppo determinato di possibili scelte. Le decisioni possono anche comportare sequenze di scelta o strategie, e ogni sottoscelta deriva dall'analisi di tutta la base informativa disponibile. Il soggetto fissa una distribuzione di probabilità congiunta per tutti gli eventi futuri. Gli elementi basilari della teoria sono quindi quattro:

1) funzione di utilità cardinale;
2) gruppo completo di strategie possibili;
3) distribuzione di probabilità di scenari futuri collegati a ciascuna strategia;
4) politica di massimizzazione dell'utilità attesa.

Questa teoria appena esposta è la fonte principale di critiche rivolte all'economia neoclassica. Il soggetto considerato nella teoria è in grado di contemplare in modo onnicomprensivo tutto ciò che ha di fronte in ogni dimensione di spazio e di tempo, può valutare l'intera gamma di alternative presenti e future, tenendo conto sia delle conseguenze che delle singole strategie, almeno fino al punto di poter assegnare una distribuzione di probabilità congiunta ai futuri stati del mondo, e le componenti che entrano nella funzione di utilità sono conosciuti a priori. Questo scenario non aderisce per nulla alla realtà e annulla l'applicabilità del modello al mondo reale. Si ha scarso contatto tra teoria e procedure effettivamente seguite per prendere decisioni. Le semplificazioni e gli assunti risultano essere troppo approssimativi e determinano previsioni superabili in qualità da qualsiasi altra teoria. La causa principale dell'insuccesso consiste nel fatto che gli esseri umani non hanno le capacità razionali necessarie per applicare i principi della teoria.

Il secondo modello è il modello "comportamentale", nel quale si parte dal presupposto che la mente umana sia molto limitata e molto condizionata dalle situazioni e dai poteri valutativi dell'uomo. Con la teoria comportamentista si ha una valida descrizione del processo decisionale dell'uomo. L'uomo, possedendo capacità limitate di valutazione, effettua scelte di adattamento.

Solitamente le scelte riguardano piccole aree della vita umana e le scelte sono inerenti ad argomenti ristretti ritenuti indipendenti e prioritari rispetto ad altri. Raramente sono presi in considerazione tutti gli scenari futuri, al massimo si parte dalla visione d'insieme del proprio stile di vita e delle prospettive future, focalizzando l'attenzione su alcuni aspetti e valori della vita a scapito di altri. Le scelte effettuate secondo uno schema di questo tipo sono esempi di "razionalità limitata". Si procede sezionando la realtà in tanti problemi separati.

Per utilizzare la propria razionalità limitata, un individuo deve mantenere un certo grado di attenzione, la quale risulta influenzata dalle emozioni. La teoria comportamentale non dissocia l'emozione dal pensiero umano e non sottovaluta i potenti effetti dell'emozione nel definire l'ordine di precedenza nella ricerca della soluzione dei vari problemi umani. Un esempio di come l'emozione possa influenzare il comportamento umano si ha nell'apprendimento: in qualsiasi campo si ha coinvolgimento maggiore se l'informazione è fornita in un contesto emozionale. Ad esempio, risulta più agevole memorizzare una vicenda o un concetto se questi sono studiati leggendo un romanzo.

Le possibilità di scelta sono elaborate dopo aver acquisito dati dall'ambiente in cui si è inseriti, e tratto inferenze che permettano di valutarne le conseguenze. In questo modo si ha un'immagine semplificata del

mondo e si impostano le decisioni in base a quest'immagine. Questo modello descrive esattamente il modo in cui effettivamente sono prese decisioni e inoltre tiene conto delle limitate capacità di valutazione. La decisione finale non rappresenta l'ottimo e non è detto sia ragionevole. Inoltre può risultare influenzata dall'ordine in cui le possibilità di scelta si sono presentate. Il modello comportamentale rinuncia a molte delle caratteristiche formali del modello olimpico, ma fornisce un avvicinamento alla razionalità che chiarisce come le creature dotate delle nostre capacità mentali, considerando anche l'ausilio dei più moderni strumenti informatici, possano avere successo in un mondo che è fin troppo complicato da capire anche utilizzando l'olimpica visione della teoria dell'utilità soggettiva attesa.

Legrenzi (2002) propone un esempio per illustrare le diverse implicazioni che comportano il modello olimpico e la razionalità limitata. Si immagini un uomo che debba raggiungere la cima di una montagna ricca di sentieri e frastagliata in più vette, e il suo problema sia quello di scoprire quali sentieri percorrere per raggiungere la vetta più alta. Se l'individuo fosse dotato di razionalità olimpica, conoscerebbe la mappa della montagna prima di iniziare la scalata e sarebbe in grado di individuare la strada ottima combinando tra loro vari sentieri, ossia la via più breve e meno difficile. Se l'individuo invece fosse dotato di razionalità limitata, non conoscerebbe a priori il modello globale della montagna che si trova a scalare, e si costruirebbe solamente modelli parziali sviluppati durante la salita e la continua esplorazione di nuovi sentieri. In questo modo lo scalatore è in grado di raggiungere una vetta, la quale sarà la più alta tra quelle che le sue capacità gli permettono di analizzare, ma non è detto che sia la più alta in assoluto. Anche esplorando i dintorni non riuscirebbe a trovare una posizione migliore senza che questo comporti un peggioramento, almeno iniziale, della sua situazione. Infatti, l'unica soluzione che potrebbe consentirgli di trovare una vetta migliore sarebbe quella di ridiscendere e tentare la risalita su altri sentieri. Ma non potendo né sapere né vedere se nei dintorni vi sia un altro picco, si convincerebbe di essere giunto alla vera vetta e si accontenterebbe di un "massimo locale". Occorre quindi tenere conto, nella risoluzione dei problemi attraverso l'utilizzo di teorie, di una sorta di compromesso tra una rappresentazione completa, ma cognitivamente onerosa e di fatto spesso impossibile, e una scomposizione del problema che permette di semplificare i processi di ragionamento ma che porta a soluzioni di ottimo locale.

Modelli che tengono conto dei limiti imposti dalla biologia del cervello e che, nel derivare previsioni, considerano strategie maggiormente simili a quelle realmente adottate dal nostro sistema cognitivo, potrebbero avere maggior successo di quelli che, in economia e nelle scienze dell'uomo, sono stati costruiti ponendosi "alla cima della montagna" e che raramente trovano riscontro empiricamente.

Il terzo modello individuato da Simon (1984) è il modello "intuitivo", con il quale si sposta l'attenzione sul tema dell'intuizione. Questa teoria è una componente della teoria comportamentale. L'intuizione è intesa come processo di riconoscimento che è alla base di numerosi tipi di attività. Risultano di fondamentale importanza le esperienze ed il loro accumulo. Inoltre è riconosciuto il ruolo delle emozioni, le quali possono influenzare la concentrazione umana su particolari problemi in particolari momenti. Il modello postula che gran parte del pensiero umano, e quindi gran parte del successo degli esseri umani nel pervenire a decisioni valide e convenienti, sia dovuto al fatto che essi siano dotati di buone facoltà di intuizione e giudizio.

Uno dei problemi importanti consiste nel capire se esistano due forme del pensiero umano radicalmente distinte, il pensiero analitico ed il pensiero intuitivo, evitando di avventurarsi in inverosimili descrizioni delle attività cerebrali, come ad esempio la suddivisione delle funzioni tra emisfero destro, a cui molti attribuiscono facoltà quali l'immaginazione, la fantasia, la creatività e la risoluzione dei problemi in modo creativo, ed emisfero sinistro, a cui appartengono le doti di analisi. Per intuizione si intende il processo di riconoscimento "improvviso" della soluzione. In base all'esperienza accumulata nel tempo si è in grado di reagire ad un sempre maggiore numero di stimoli differenti e l'attività di "riconoscimento" avviene più facilmente.

Non vi sono conflitti tra il modello intuitivo e il modello comportamentale. Qualsiasi rilevante attività di pensiero coinvolge entrambi gli schemi e, senza il riconoscimento basato su esperienze precedenti, la ricerca di "alternative" in spazi complessi risulterebbe molto lunga e difficile.

Il quarto ed ultimo modello analizzato è la visione della razionalità come adattamento evolutivo. L'implicazione di fondo è che solo quegli organismi che si adattano, che si comportano cioè come se fossero razionali, possono sopravvivere. Friedman (1953) elabora la teoria del "come-se": gli uomini d'affari e le imprese si comportano sempre come se avessero fatto calcoli corretti e razionali, necessari per conseguire il massimo dell'utilità o dei profitti, e solo coloro che hanno successo nella massimizzazione dei loro profitti sopravvivono nel mondo economico. In questo caso tutto si basa sul "risultato-successo" nell'adattamento all'ambiente economico e non ha alcuna importanza tramite quale procedimento raziocinante, o quale processo puramente casuale, si sia giunti all'adattamento.

Vi sono alcuni interessanti paralleli tra la teoria comportamentale e i meccanismi darwiniani. Secondo la teoria comportamentale, la scelta razionale può rendere necessarie un gran numero di ricerche selettive, al fine di individuare soluzioni che realizzino l'adattamento. Ogni processo di

ricerca, anche il più semplice, richiede che le possibili risposte siano prima messe a punto e quindi sottoposte a verifica per comprendere se siano appropriate. Questo meccanismo di messa a punto e verifica è il diretto corrispondente del meccanismo di variazione-selezione della teoria darwiniana. Come nella teoria biologica dell'evoluzione il meccanismo della selezione naturale elimina le varianti che hanno dimostrato scarsa capacità di adattamento, così il processo di verifica nel pensiero umano rifiuta le idee diverse da quelle che contribuiscono alla risoluzione dei problemi di cui ci si sta occupando. Nelson e Winter (1982) si sono interessati alla definizione dei meccanismi che possono essere rilevanti ai fini dell'adattamento e dell'evoluzione delle imprese commerciali. I "geni" di Nelson e Winter sono le abitudini e le procedure operative che le imprese adottano nella gestione dei propri affari. Le nuove strategie che sono elaborate devono essere messe alla prova insieme alle vecchie strategie. In questo modo, la posizione finale risponderà ai criteri dell'adattamento, ma non sarà necessariamente una posizione ottimale. In entrambi i casi si scoprono nuove possibilità raggiungendo ottimizzazioni di carattere locale, si confrontano con quelle esistenti e si stabilisce se vi siano miglioramenti.

SIMULAZIONE ED ECONOMIA

Come sostenuto da Terna (2002c), in economia possono essere individuati due principali usi della simulazione. Il primo di questi è senz'altro la conduzione di esperimenti mentali, quindi l'utilizzo dei calcolatori come protesi mentali per affrontare problemi di elevata dimensione e profondità. Il secondo uso è l'impiego del modello di simulazione come strumento per la rappresentazione formale della realtà. In questo caso possono essere rappresentate situazioni anche molto complicate, raggiungendo il dettaglio che si ritiene necessario.

La simulazione può essere vista come la "terza via " per la realizzazione dei modelli. I modelli, infatti, si possono classificare in:

- modelli letterario-descrittivi;
- modelli statistico-matemetici;
- modelli realizzati con il codice informatico.

Nei tre casi si hanno di fronte diversi livelli di flessibilità e coerenza con i fenomeni che cercano di spiegare. Nei modelli letterario-descrittivi si ha il livello massimo di flessibilità, ma la loro rappresentazione della realtà non è verificabile da processi di calcolo che rendano possibile il confronto tra i risultati del modello e la realtà empirica, per testarne la coerenza. I modelli statistico-matematici presentano caratteristiche speculari, in quanto sono computabili, ma le numerose semplificazioni possono mantenerli anche

molto distanti dalla realtà. Nel caso particolare dei modelli economici, non è insolito ritrovare studiosi come Friedman, secondo cui non è importante la plausibilità del modello, ma piuttosto la possibilità di effettuare buone previsioni, sopportando l'idea di aver effettuato assunzioni poco realistiche. Gran parte dell'economia neoclassica si fonda su simili concezioni.

Le diverse esigenze di flessibilità, adattabilità descrittiva e computazione trovano un valido compromesso nella simulazione. Il codice informatico con cui si costruiscono le simulazioni risulta dotato di tutte le caratteristiche formali di specificabilità, versatilità ed efficienza desiderabili in un modello.

Con la simulazione sono specificabili tutte le caratteristiche del mondo, il quale può essere inizialmente considerato nelle sue caratteristiche fondamentali e successivamente complicato a piacere. Per ogni singola parte del modello è necessario formulare delle ipotesi. La versatilità deriva dalle possibilità offerte da un modello ben costruito, con cui si possono testare ipotesi, esplorare nuove idee, generare basi di dati ed eventualmente costruire vere e proprie estensioni del mondo reale. Tutte queste caratteristiche portano ad ottenere un elevato grado di efficienza, intesa come possibilità di raggiungere risultati utili con sforzi minori di quelli richiesti dalle metodologie sperimentali tradizionali, che in scienze come l'economia non sempre sono applicabili.

Oltre alle tre caratteristiche ora descritte occorre considerare il valore maieutico della simulazione. Durante la costruzione del modello possono sorgere nuovi stimoli ad affrontare problemi nuovi o latenti, estraendo nuove conoscenze riguardo al mondo che si studia.

Viste le caratteristiche appena esaminate, in economia l'utilizzo della simulazione dovrebbe trovare maggiore spazio e dovrebbe essere riposta maggiore fiducia nell'interdisciplinarietà metodologica che si realizzerebbe con questo metodo.

L'economia dovrebbe occuparsi di studiare più a fondo come agiscono gli individui, senza pretendere di scoprire schemi generali da cui deriva ogni conoscenza. Occorre considerare l'economia non come un progetto degli uomini, ma come complessa interazione tra individui e azioni.

Occorre quindi porsi tra due estremi. Da una parte vi è lo studio dell'economia attraverso i modelli astratti dell'economia matematica e la ricerca dell'equilibrio economico generale, mentre dall'altra vi è la considerazione dell'economia come sistema complesso e quindi la necessità di riprodurne artificialmente il funzionamento, perchè, come appreso da Simon (1984), nei sistemi complessi si può mostrare che i progetti formulati dai singoli attori non confluiscono armonicamente negli obiettivi più generali

del sistema, che risulta guidato e vincolato da principi di evoluzione e conservazione.

La simulazione ad agenti in contesti economici

Le simulazioni offrono uno strumento che permette di studiare insieme i singoli individui e i fenomeni collettivi che derivano dalla loro interazione. Quindi si propongono come valido ed interessante strumento per lo studio dell'economia. Nella definizione data da Parisi (2001), la simulazione ad agenti è un particolare tipo di simulazione che cerca di riprodurre fenomeni collettivi facendo interagire tra di loro un determinato numero di "agenti", i quali hanno determinate "regole", ossia reagiscono in modo determinato agli stimoli che ricevono. Questo tipo di simulazione risulta di particolare utilità nello studio dei fenomeni sociali. Il comportamento del singolo agente è in grado di influenzare i comportamenti di altri agenti e dall'insieme degli agenti emerge la complessità. Riproducendo nella simulazione determinati tipi di agenti, a cui si danno sembianze il più possibile fedeli a quelle degli agenti che si intendono studiare, è possibile affrontare lo studio di numerosi sistemi reali, composti da numerosi individui interagenti. Con il vantaggio, sottolineato in Terna (1996), di poter verificare in qualsiasi momento le caratteristiche degli agenti e i loro comportamenti.

Ancora Terna (2002c) spiega che la costruzione di modelli di simulazione fondati su agenti consiste nello studio dell'azione dei soggetti economici in termini cognitivi e nella comprensione delle conseguenze delle caratteristiche degli agenti che si scoprono nel tempo, tenendo conto di azioni, reazioni e interazioni tra i singoli agenti.

La costruzione dei modelli è facilitata dall'esistenza di protocolli di programmazione ad oggetti, come ad esempio Swarm, Jas ed altri, e dall'utilizzo di schemi di riferimento, come ad esempio lo schema ERA (si veda capitolo 5), che consentono di sviluppare progetti molto ampi con ordine e velocità.

In un qualsiasi modello di simulazione ad agenti è possibile effettuare la distinzione tra tipologie di agenti e tipologie di ambienti in cui questi operano. Vi possono essere agenti "con mente", che sono agenti che possiedono diversi gradi di capacità di adattamento all'ambiente, oppure "senza mente", quindi privi di capacità di adattamento. L'ambiente in cui operano gli agenti può essere classificato come strutturato in base a delle regole oppure del tutto neutrale.

Si può osservare un esempio in Terna (2001), dove si descrive un modello di simulazione di un mercato molto semplice in cui si sperimentano i seguenti scenari:

- agenti "senza mente" operanti in ambiente non strutturato;

- agenti "senza mente" operanti in ambiente strutturato;
- agenti "con mente" operanti in ambiente non strutturato;
- agenti "con mente" operanti in ambiente strutturato.

Nel primo caso si nota che gli agenti "senza mente" operanti in contesti non strutturati producono risultati complessi ma non realistici. Consumatori e venditori operano in ordine casuale e ogni consumatore cerca un venditore e confronta il proprio prezzo-soglia con quello del venditore. L'acquisto è effettuato solo se quello di quest'ultimo è inferiore. Questa struttura provoca prezzi ciclici, con transazioni caotiche da una fase all'altra.

Inserendo agenti "senza mente" in un ambiente strutturato, dove per struttura si intende introduzione di un meccanismo di contrattazione telematica come quello di una borsa senza "grida", si producono risultati molto realistici, come, ad esempio, bolle o *crash*. In questo caso il *book*, come avviene nella realtà, abbina gli ordini in sequenza e non sono utilizzate equazioni per determinare il prezzo che eguaglia domanda e offerta ad ogni intervallo di tempo. L'aspetto interessante è rappresentato dal fatto che il realismo dei risultati derivi dalla struttura del mercato e non dalla presenza di agenti con caratteristiche sofisticate.

Nel caso degli agenti "con mente", in Terna (2002c) le capacità di adattamento derivano dall'adozione della tecnica dei *Cross Target*, fondata sulla metodologia delle reti neurali (questa tecnica è descritta nel capitolo 1). L'idea di fondo è quella che l'agente non operi facendo ricorso a regole economiche specifiche definite a priori e abbia delle possibilità di apprendimento che diano coerenza alle sue azioni. Con questa semplice struttura, l'agente appare in grado di compiere azioni che sembrano pianificate e dirette al perseguimento di specifici obiettivi. Agenti di questo tipo, che operano in un ambiente non strutturato, possono determinare risultati aggregati complessi e plausibili.

Se gli agenti "con mente" sono inseriti in un ambiente strutturato è possibile che si verifichino risultati di rilievo a livello micro-individuale. Oltre a riscontrare risultati aggregati complessi e plausibili, è possibile che l'agente sviluppi una modalità di azione che gli permetta di ottenere risultati economicamente positivi anche in presenza di turbolenze di prezzo.

CAPITOLO 3

CONFRONTO TRA AMBIENTI DI SIMULAZIONE

In questo capitolo sono descritti diversi ambienti di simulazione, le loro principali caratteristiche e le possibilità operative. In particolare, si approfondisce la struttura dei modelli di simulazione in *Swarm* e si mettono in luce alcuni aspetti fondamentali del suo diretto discendente *Jas*.

SWARM

Swarm nasce nel 1994 su iniziativa di Chris Langton al *Santa Fe Institute* nel New Messico. Il principale obiettivo è quello di creare uno "strumento" per la costruzione di simulazioni ad agenti, sfruttando le caratteristiche della programmazione ad oggetti. In un primo momento si era pensato di sviluppare *Swarm* in modo da permettere la costruzione di simulazioni che potessero essere gestite da sistemi paralleli, ossia computer dalla struttura diversa dai computer tradizionali, capaci di gestire in parallelo parti di codice differente. Le tracce di questa grande ambizione, che non ha avuto esiti, si ritrovano tuttora nel codice delle applicazioni sviluppate con *Swarm*.

I campi di studio in cui è possibile utilizzare *Swarm* sono molteplici e vanno dalla biologia all'economia, dalla finanza alla psicologia. Il significato della parola "*swarm*", in italiano "sciame", si ricollega alla possibilità di studio dei fenomeni complessi, in cui si hanno sistemi in cui numerosi agenti semplici interagiscono tra di loro e generano dinamiche non lineari che non possono essere studiate con metodi tradizionali.

Innanzitutto, *Swarm* può essere definito come una biblioteca di codice informatico, che nel tempo è diventata molto ampia, e che permette di sviluppare simulazioni in modo molto rapido. Ma non solo, infatti *Swarm* è

anche un "protocollo" per lo sviluppo di modelli di simulazione, e permette di creare delle simulazioni in maniera standardizzata che possono facilmente essere capite, utilizzate ed eventualmente ulteriormente sviluppate da qualsiasi appartenente alla comunità degli utilizzatori.

Il linguaggio di programmazione utilizzato inizialmente per la costruzione delle librerie è *Objective C*, ma dal 1999 si introduce il linguaggio *Java*, permettendo così l'utilizzo di *Swarm* ad un più ampio insieme di utenti. Il sistema operativo per cui è stato concepito è *Unix*, ma attualmente, grazie all'esistenza di *Cygwin*, che è un'applicazione che emula un sistema operativo *Unix*, è possibile sviluppare simulazioni con *Swarm* anche su sistemi operativi *Windows*.

Nel paragrafo che segue si analizza brevemente lo schema generale, descrotto da Terna (2002a), che deve essere seguito per lo sviluppo di una simulazione con *Swarm*, il quale permette una grande elasticità sia dal punto di vista del riutilizzo del codice, che dal punto di vista della gestione del fenomeno che si intende riprodurre.

Lo schema generale delle simulazioni

La prima caratteristica di un modello di simulazione sviluppato con *Swarm* è la presenza di un insieme di *file* separati, tipico delle applicazioni sviluppate con programmazione orientata agli oggetti. La particolarità fondamentale è la presenza di uniformità nella denominazione dei diversi *file*. Ciò permette a chiunque conosca il protocollo *Swarm* di orientarsi e arrivare alla comprensione della simulazione in maniera molto veloce, potendola adattare alle proprie esigenze. *Swarm* diventa una specie di "lingua franca", che avvicina tra loro tutti gli utilizzatori delle simulazioni, anche se provenienti da campi molto diversi.

In una generica simulazione si ritrovano le seguenti parti, separate in diversi file:

- il *model*, nel quale sono costruiti tutti gli oggetti necessari alla simulazione, e si definiscono e gestiscono i diversi eventi e la loro successione temporale;
- l'*observer*, nel quale vi è la creazione del *model* e di tutta una serie di oggetti necessari all'osservazione della simulazione e alla rilevazione dei dati;
- il *main*, che è il programma principale che avvia tutta la simulazione ed è contenuto all'interno di un file il cui nome inizia con la parola "*Start*";
- le diverse "classi" da cui derivano tutti gli oggetti utilizzati all'interno della simulazione, ossia il codice informatico

costituente gli agenti e tutti gli altri oggetti che interagiscono nella simulazione;

- il *Makefile*, contenente le istruzioni necessarie per la compilazione dell'insieme dei *file* della simulazione.

L'aspetto fondamentale ed innovativo dei modelli così sviluppati è lo sdoppiamento tra modello di simulazione, costituito dal *model*, e sistema di osservazione, costituito dall'*observer*. Nelle simulazioni *Swarm* si riproduce esattamente lo schema di un esperimento di laboratorio reale, in cui si ha un sistema, composto da diversi oggetti che interagiscono tra loro e in cui gli eventi si succedono in un determinato ordine, e un osservatore, il quale provvede, con gli appropriati strumenti di misura, a raccogliere le informazioni e i dati che in seguito potranno essere elaborati.

Una applicazione di *Swarm* si presenta a due strati: uno strato interno in cui avviene la simulazione, organizzata in una serie di eventi ordinati nel tempo, e uno strato esterno, in cui è strutturata l'osservazione dei fenomeni simulati, la quale è gestita in una serie di rilevazioni ordinate nel tempo. La particolarità di questa struttura è che il modello di simulazione e l'osservatore del modello possiedono due separate gestioni del tempo, che scandiscono in modo non necessariamente uguale gli eventi. Le scansioni temporali delle rilevazioni da parte dell'osservatore sono del tutto indipendenti dal succedersi degli eventi nel modello. La programmazione degli eventi, i quali sono suddivisi in gruppi, avviene attraverso la costruzione di due elenchi separati, uno interno al *model* e uno interno all'*observer*, detti *schedule*. Ad ogni istante della simulazione è eseguito un gruppo di eventi, nell'ordine dato dallo *schedule*.

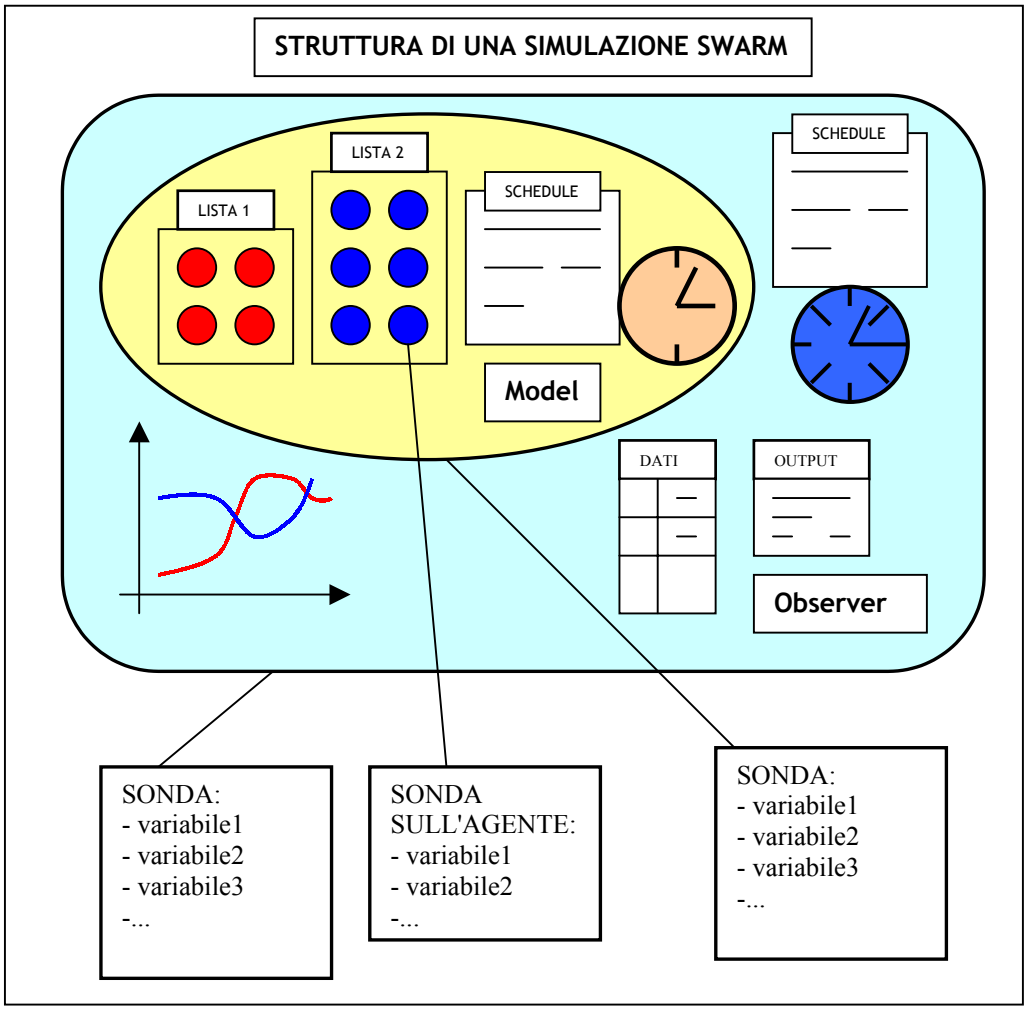
Inoltre è possibile, attraverso l'utilizzo delle sonde, che sono particolari oggetti per l'osservazione e il controllo della simulazione, avere continuamente visualizzati i valori delle principali variabili del modello e introdurre dei cambiamenti sulle condizioni iniziali per sperimentarne gli effetti.

Le sonde possono essere create a tre livelli. Il primo livello è il livello dell'*observer*, in cui attraverso la sonda è possibile effettuare cambiamenti nell'osservazione dei fenomeni. In questo caso la sonda si presenta come una specie di "cassetta degli attrezzi" contenente i diversi strumenti di rilevazione dei dati. Lo sperimentatore, attraverso questa sonda può decidere se rilevare o meno determinati dati, visualizzare grafici o messaggi sullo schermo durante la simulazione. Il secondo tipo di sonda è quella sul modello di simulazione, ossia sul *model*. In questo caso è possibile visualizzare il valore delle variabili rilevanti del fenomeno che si studia, ed eventualmente agire sulla struttura e sulla scaletta degli eventi del modello.

Un'ultima interessantissima possibilità è quella di aprire una sonda direttamente su uno degli agenti operanti nel sistema simulato. Ciò dà la possibilità di visualizzare le variabili interne dell'agente ed eventualmente registrarne la situazione attraverso la raccolta di dati o la rappresentazione grafica.

Gli agenti nel *model* possono essere gestiti in gruppo, attraverso l'utilizzo delle liste. Nella programmazione degli eventi del *model* è possibile comunicare direttamente con le liste di agenti invece che con il singolo agente. Questo rende molto più agevole la comunicazione con gli agenti e risparmia all'utilizzatore della simulazione la costruzione di complicate strutture informatiche.

Nello schema seguente è rappresentata la struttura di una simulazione *Swarm* e le particolarità del protocollo appena descritto.



LE LIBRERIE DI SWARM

La grande quantità di librerie disponibili in *Swarm* permette agli utilizzatori di sviluppare velocemente le proprie applicazioni utilizzando parti di codice già sviluppate, in piena sintonia con i principi base della programmazione ad oggetti.

Il *Reference-book* (2000) presenta le principali librerie disponibili in *Swarm*:

- *objectbase*, la quale contiene le classi fondamentali per il funzionamento degli agenti e la gestione delle sonde;
- *defobj*, che contiene le funzionalità per la definizione dei metodi di base per la creazione e l'eliminazione degli oggetti;
- *activity*, la quale contiene tutti gli strumenti per la gestione delle attività della simulazione (creazione dei gruppi di azioni e dello *schedule*);
- *random*, per la generazione di numeri casuali;
- *collections*, che contiene le classi necessarie alla creazione e gestione di liste, vettori e mappe.

Le librerie appena citate rappresentano solamente alcuni esempi tratti dall'ampia gamma di strumenti, che vanno ad aggiungersi a quelli offerti dalle librerie di base dei linguaggi di programmazione, come *Objective C* e *Java*. Inoltre esistono molti "pacchetti" aggiuntivi, che risolvono esigenze specifiche e che sono il frutto dell'attività della comunità degli utilizzatori di *Swarm*.

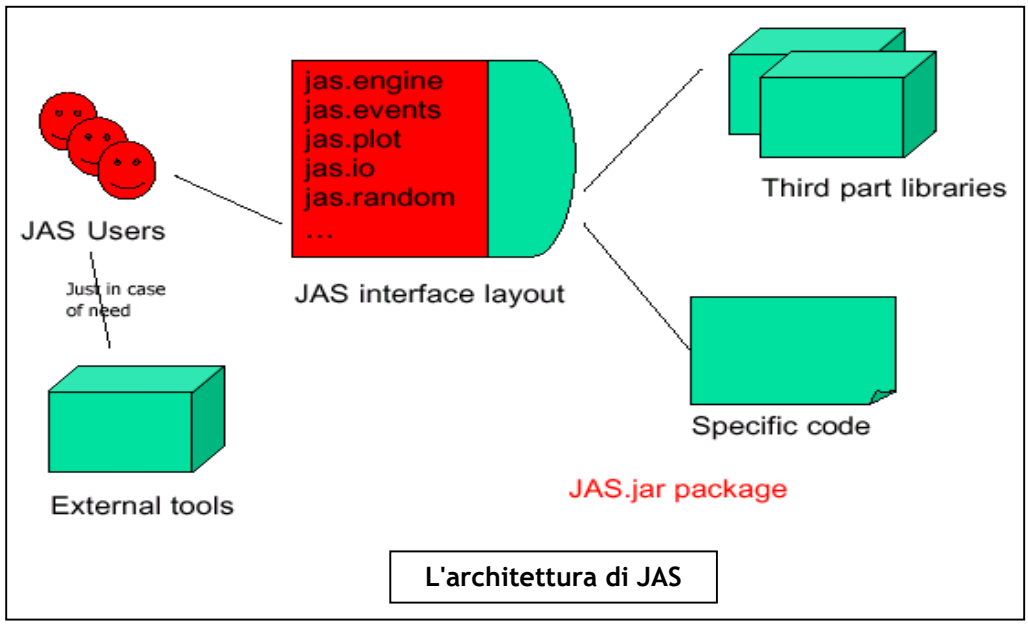
JAS: JAVA AGENT-BASED SIMULATION LIBRARY

Sulle orme di *Swarm*, si sta sviluppando un progetto Sonnessa (2003), per ricreare tutte le possibilità offerte da *Swarm* in linguaggio *Java*, con notevoli vantaggi dal punto di vista della compatibilità e della facilità di utilizzo.

Java Agent-based Simulation library (*JAS*) è un insieme di strumenti, che possono essere utilizzati per la costruzione di modelli di simulazione strutturati nella logica ad agenti, costituito da una libreria in codice *Java*. Il progetto è del tutto libero, ossia costruito seguendo logiche *open source*, ed è ospitato in *SourceForge*, che è un portale su cui è possibile trovare numerosi progetti per la costruzione di modelli di simulazione ad agenti. *JAS* è stato costruito adottando i protocolli di *Swarm*, e la struttura di base di

una simulazione riproduce esattamente la struttura analizzata nei paragrafi precedenti.

L'attuale insieme di librerie *standard* potrà essere in futuro arricchito con l'aggiunta di nuove librerie esterne o con lo sviluppo del codice che è alla base degli strumenti ora a disposizione. La separazione tra interfaccia ed implementazione, tipica dei linguaggi di programmazione ad oggetti come *Java*, permette agli utilizzatori finali degli strumenti, in questo caso i costruttori di simulazioni, di ignorare la struttura interna degli strumenti che utilizzano, limitandosi alla conoscenza dell'interfaccia. Sonnessa (2003) rappresenta questo concetto servendosi del seguente schema:



L'architettura di JAS

Le principali differenze tra Swarm e JAS

La principale differenza tecnica tra *Swarm* e *JAS* è il linguaggio in cui sono sviluppate le librerie: *Java* per *JAS*, *Objective C* per *Swarm*. Con *JAS* scompaiono tutti i problemi legati all'installazione e alla configurazione delle librerie, con la garanzia di massima compatibilità tipica di *Java*. In ambiente *Windows*, infatti, non si è legati alla necessità di emulare un sistema *Unix* con l'utilizzo di *Cygwin*.

Con *JAS* è possibile far compiere in parallelo delle azioni agli agenti, per la cui costruzione è possibile utilizzare strumenti molto avanzati dell'intelligenza artificiale grazie alla presenza di una libreria apposita.

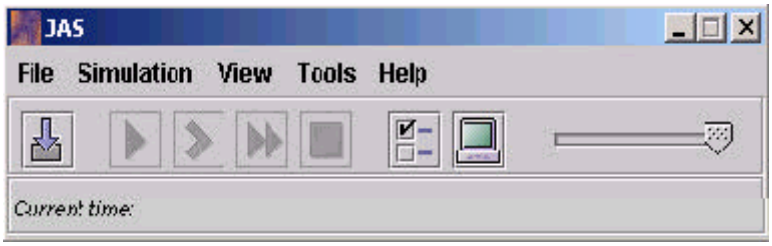
Attraverso il protocollo *Sim2Web* di Margarita e Sonnessa (2003), è possibile pubblicare le simulazioni sul *web* e permettere l'interazione degli agenti umani nella simulazione.

JAS può essere definito non solo come una libreria, ma come un'applicazione. Infatti il modello di simulazione diventa un'applicazione *Java* composta dalle classi contenute nell'archivio "*JAS.jar*", compilate da un qualsiasi compilatore *Java* (come per esempio il compilatore gratuito della *Sun*, *JDK*).

Ogni simulazione può essere riavviata senza necessariamente uscire dall'ambiente *JAS* e le azioni sono controllabili attraverso un pannello di controllo che offre le seguenti possibilità:

- creare il progetto del modello specificandone posizione in memoria, librerie e parametri necessari, attraverso il "*project editor*";
- caricare in memoria e compilare un modello di simulazione;
- salvare la posizione delle finestre della simulazione;
- aprire una finestra su cui sono visualizzati gli *output* della simulazione;
- controllare lo stato della simulazione che si sta eseguendo.

Il pannello di controllo si presenta come segue:



Struttura di una simulazione JAS

Una simulazione costruita in *JAS* è un'applicazione *Java* con una struttura di base rigida. Il primo elemento della struttura è il *main*, all'interno del quale sono creati i diversi oggetti della simulazione.

L'interazione tra l'utilizzatore e il modello avviene attraverso il *SimModel*, con cui si attiva il *timer* della simulazione e una serie di *eventList*, particolari oggetti contenenti le azioni che il modello deve compiere. Nel *model*, inoltre, è definita una serie di metodi che sono richiamati in corrispondenza del verificarsi di determinati eventi. Ad esempio, quando è azionato il pulsante per la creazione degli oggetti della simulazione, è automaticamente richiamato il metodo *builtModel()* definito all'interno del *model*, il quale crea i diversi agenti e gli oggetti della simulazione e definisce la scaletta delle azioni da compiere durante la simulazione.

A differenza delle simulazioni costruite in *Swarm*, in *JAS* non si definisce una classe *Observer*. Le ragioni di questa scelta sono dovute al fatto che il *Model* e l'*Observer* presentano una struttura molto simile. In *JAS*, *observer* e *model* sono esemplari della stessa classe e sono eseguiti indipendentemente. Con questa struttura è possibile attivare in parallelo più di un *model*. L'*observer* è messo in condizione di poter ispezionare le variabili interne di un *model*, ma non presenta caratteristiche costruttive diverse rispetto al *model*.

Il vantaggio di questa struttura è la possibilità di attivare l'*observer* solo quando serve, specificando l'attivazione nel *project definition file*. Questo *file* è costruito in linguaggio *XML* e permette di specificare al suo interno tutti i *model* che si desiderano creare nella simulazione. Ogni *model* è governato a sua volta dal *JAS engine*, il quale contiene il *timer* della simulazione.

Le principali librerie

Java permette ai programmatori di organizzare il loro codice in librerie, che sono richiamate all'interno delle definizioni delle classi che compongono le simulazioni. *JAS* è composto da dieci pacchetti, le cui funzioni sono brevemente descritte nei paragrafi che seguono.

jas.ai

La libreria dell'intelligenza artificiale contiene numerosi algoritmi che possono essere utilizzati per conferire caratteristiche particolari agli agenti. In particolare sono disponibili algoritmi basati sulle reti neurali descritti in Terna (2000a), conosciuti come "*bp-ct*", e algoritmi basati sui *classifier system* e sugli algoritmi genetici, descritti in Ferraris (2000).

jas.engine

La classe *engine* è responsabile dell'impostazione e della guida della simulazione. La libreria contiene tutti gli strumenti per la gestione della sequenza degli eventi che compongono la simulazione.

jas.events

Gli eventi sono il cuore dello *schedule* della simulazione e permettono l'invio di messaggi dal *model* agli altri oggetti della simulazione. I messaggi possono in particolare essere inviati a singoli oggetti, a intere collezioni di oggetti (come ad esempio le liste) o a una parte della collezione specificata. In questa libreria è definito un "*garbage collector*" che è impiegato per ripulire gli spazi di memoria.

jas.io

Le operazioni di *input* e *output* sono fondamentali per il modello di simulazione. E' possibile ottenere come *output* dei dati in formato "*comma separated value*" (*CVS*), che possono essere trattati in seguito con un qualsiasi foglio elettronico (ad esempio *Microsoft Excel*), e per le operazioni *I/O* su disco è supportato il formato "*eXtensible Markup Language*" (*XML*).

jas.net

Questo pacchetto contiene tutte le interfacce necessarie per *Sim2Web*, con cui è possibile rendere accessibile sulla rete *Internet* una simulazione, con l'appoggio di un *server Zope*. Inoltre è possibile fare interagire con i modelli operatori umani che si collegano in modo remoto.

jas.plot

Sono disponibili numerose tipologie di oggetti grafici, con cui è possibile rappresentare lo spazio e i risultati della simulazione.

jas.probe

La costruzione delle sonde permette di gestire una simulazione in modo interattivo, con la possibilità di modificare parametri. L'*observer* utilizza le sonde per visualizzare le variabili interne al *model*, o per osservare le dinamiche interne di particolari elementi del sistema.

jas.random

Con l'utilizzo degli strumenti di questa libreria è possibile utilizzare dei generatori di numeri casuali. Questi strumenti si basano sulla libreria *COLT*, sviluppata dal *CERN* di Ginevra.

jas.space

In questa libreria sono contenuti gli strumenti grafici a due dimensioni, che servono per creare delle mappe dello spazio che si utilizzano per la simulazione, particolarmente utili per modelli basati su automi cellulari. Questa libreria si integra con quella *jas.plot* descritta in precedenza.

jas.stats

Con questo pacchetto è possibile utilizzare particolari sonde sugli agenti ed effettuare statistiche riassuntive sulla loro situazione, grazie a una ricca serie di indicatori statistici, anch'essi basati sulla libreria *COLT* precedentemente menzionata.

Un terzo esempio di ambiente in cui è possibile sviluppare delle simulazioni è rappresentato da *NetLogo*. La sua caratteristica fondamentale è l'utilizzo di un linguaggio di programmazione proprio, diverso dai linguaggi più conosciuti, come lo sono *Objective C* o *Java*.

NETLOGO

NetLogo (2003) può essere definito come un insieme di strumenti per la costruzione di modelli di simulazione che riproducono fenomeni naturali e sociali, in cui, dall'interazione di un insieme di individui, emergono fenomeni complessi.

In *NetLogo* è possibile amministrare collettivamente centinaia di individui che operano in parallelo, permettendo di esplorare le connessioni tra i comportamenti a livello micro-individuale e osservare il risultato delle interazioni.

Le possibilità di sperimentazione sono molto ampie, in quanto è possibile variare le condizioni iniziali del sistema e analizzare i risultati ottenuti. Sviluppare simulazioni, una volta apprese alcune regole fondamentali, è facile e veloce, in particolare se si confrontano le strutture dei modelli sviluppati con *NetLogo* e con quelli sviluppati con *Swarm*.

La documentazione delle librerie è molto accurata e sono disponibili sulla rete Internet numerosi archivi che raccolgono modelli sviluppati da ricercatori operanti nei campi più diversi, come la medicina, la biologia, l'economia, la chimica e la psicologia.

La semplicità di utilizzo si nota particolarmente se si vuole costruire una simulazione che preveda di poter fare operare anche agenti umani. In *NetLogo*, attraverso il cosiddetto *HubNet*, dalle diverse postazioni di una rete di computer è possibile controllare le azioni di un agente. In ambienti come *Swarm*, l'interazione di agenti umani è possibile con la costruzione di particolari agenti che sono in contatto con l'agente umano tramite una pagina *web* (si veda il capitolo 5).

NetLogo è la nuova generazione di una serie di linguaggi per lo sviluppo di modelli di simulazione ad agenti, conosciuti sotto il nome di *StarLogo*, di cui ne ereditano e ne ampliano le caratteristiche. Il fatto che sia scritto in *Java* ne assicura la possibilità di utilizzo sulla maggior parte dei sistemi operativi esistenti, e permette, con la costruzione delle cosiddette "*applet*", di rendere eseguibili le simulazioni su una pagina *web*.

Uno degli aspetti fondamentali per cui *NetLogo* potrebbe essere preferito agli altri ambienti di simulazione presentati in questo capitolo, è costituito dall'indipendenza tra il linguaggio della simulazione e il linguaggio *Java*, con cui effettivamente la simulazione finale risulta costruita. Infatti, non è necessario conoscere il linguaggio *Java* per costruire una simulazione, ma è sufficiente imparare le regole di programmazione in *NetLogo*.

ASCAPE

Ascape è un progetto *software* sviluppato dalla *The Brooking Institution* per lo sviluppo e l'analisi di modelli basati su logiche ad agenti. Gli agenti sono racchiusi in gruppi (o collezioni) che a loro volta sono considerati agenti del modello. Nei modelli quindi operano "collezioni di collezioni" di agenti. Per ogni insieme di agenti esistono delle regole di comportamento e degli spazi di interazione. Ci sono numerose possibilità per visualizzare graficamente la simulazione, controllarne i parametri, immagazzinarne i dati ed effettuare statistiche.

Anche *Ascape* è scritto interamente in *Java* e risulta molto più simile a *Swarm* che a *NetLogo*, in quanto richiede che l'utilizzatore sia in grado di programmare in *Java*.

REPAST

RePast è un progetto dell'Università di Chicago, in particolare del *Social Research Computing*, per la creazione di modelli di simulazione ad agenti con l'utilizzo del linguaggio *Java*. Assomiglia molto a *Swarm* e ne riproduce le fondamentali caratteristiche: è possibile creare e fare interagire gli agenti, osservare la simulazione e raccogliere e visualizzare i dati servendosi di numerosi oggetti grafici, in alcuni casi molto sofisticati. Ad esempio, è possibile fotografare o filmare ciò che avviene durante una simulazione per poi rivederlo in un secondo momento. Alcune delle sue funzioni derivano direttamente da *Ascape*.

La visione della simulazione permette di controllare le singole parti che compongono il modello che si osserva. Le diverse componenti possono essere suddivise in due categorie: infrastruttura e rappresentazione. Per infrastruttura si intendono i vari meccanismi che permettono l'avvio e lo svolgimento della simulazione. Per rappresentazione si intendono tutti gli oggetti che sono creati durante la simulazione. Lo stato della rappresentazione del modello è quindi il valore corrente di tutte le sue variabili. Attraverso lo *Schedule*, esattamente come avviene in *Swarm*, si ha la programmazione delle variazioni di stato del modello sia a livello di infrastruttura che a livello di rappresentazione. L'intera simulazione è vista come una macchia che passa attraverso una serie di stati, pianificati nello *Schedule*.

Il nome *RePast* è l'acronimo di *Recursive Porous Agent Simulation Toolkit*. L'obiettivo principale che guida lo sviluppo di questo ambiente di simulazione è quello di andare oltre la semplice rappresentazione separata dei singoli agenti, per giungere ad una visione degli agenti come attori sociali che interagiscono in modelli di cui è possibile manipolare le variabili fondamentali e raccogliere i risultati delle sperimentazioni.

CAPITOLO 4

IL MERCATO AZIONARIO

Per costruire modelli di simulazione che riproducano fedelmente la struttura e il funzionamento del mercato reale, è necessario analizzare in modo approfondito le caratteristiche generali del mercato e il regolamento che guida la gestione delle contrattazioni. Lo scopo di questo capitolo è quello di introdurre le regole della realtà italiana ricostruita in *JavaSum*.

EVOLUZIONE NORMATIVA

Le tappe fondamentali dell'evoluzione normativa dell'intermediazione finanziaria, riassunte da Damilano e altri (2002), sono le seguenti:

- ✓ Legge n. 272/1913, costruzione delle borse valori;
- ✓ Legge n. 216/1974, istituzione della Consob e introduzione di nuove tipologie di titoli azionari (azioni di risparmio e obbligazioni convertibili);
- ✓ Legge n. 77/1983, costituzione dei fondi comuni di investimento (fci) mobiliare;
- ✓ Legge n. 1/1991, istituzione delle SIM e riorganizzazione dei mercati mobiliari (nascita dei mercati dei derivati);
- ✓ Legge n. 157/1991, disciplina dell'*insider trading*;
- ✓ Legge n. 149/1992, disciplina delle offerte pubbliche di acquisto, vendita, sottoscrizione e scambio di titoli;
- ✓ Legge n. 124/1993, disciplina dei fondi pensione;

- ✓ Legge n. 344/1993, costituzione dei fondi mobiliari chiusi;
- ✓ Legge n. 86/1994, costituzione dei fondi immobiliari;
- ✓ D.lgs. n. 415/1996, decreto di recepimento della Direttiva comunitaria n.93/CEE sui servizi di investimento;
- ✓ T.U. 20.2.1998, Testo Unico della Finanza.

I mutamenti di maggiore interesse avvengono con la legge del 2 gennaio 1991, n.1, conosciuta anche come "legge sulle SIM" e intitolata "Disciplina dell'attività di intermediazione mobiliare e disposizioni sull'organizzazione dei mercati mobiliari", con la quale si cerca di sopperire alla lacunosa e incompleta normativa vigente fino a quel momento. I punti principali della normativa sono l'istituzione delle SIM, caratterizzate da polifunzionalità operativa, ossia la possibilità di svolgere, previa autorizzazione della Consob e iscrizione in un apposito albo, tutte le attività di intermediazione mobiliare definite dalla legge, ed esclusività operativa, salvo permettere lo svolgimento di specifiche attività di intermediazione mobiliare ad alcune categorie di intermediari quali:

1. le banche, che possono svolgere attività di intermediazione mobiliare sui titoli emessi e garantiti dallo Stato, ma non sugli altri valori mobiliari quotati sui mercati ufficiali;
2. gli agenti di cambio in attività, che possono continuare a svolgere attività di negoziazione fino alla cessazione dell'attività;
3. le società fiduciarie, che possono svolgere attività di gestione dinamica di patrimoni mobiliari in nome proprio e per conto terzi, se iscritti in una sezione speciale dell'albo delle SIM, oppure attività di gestione statica, ossia semplice amministrazione titoli per conto della clientela;
4. fondi comuni di investimento mobiliare, che possono esercitare la gestione collettiva dei patrimoni mobiliari.

Inoltre nasce la figura del "promotore finanziario", di cui le SIM hanno l'obbligo di avvalersi per l'offerta fuori sede degli strumenti finanziari. I promotori devono possedere requisiti di professionalità e onorabilità, valutati tramite un concorso dalla Consob, e possono operare solamente per l'interesse di un unico soggetto.

La legge prevede l'obbligo di concentrazione degli scambi nei mercati regolamentati, valido per tutti i titoli emessi nei mercati ufficiali, con esclusione di quelli emessi o garantiti dallo Stato, e con la possibilità di derogare solo se l'operazione ha per oggetto un blocco di titoli o il cliente autorizza preventivamente e in forma scritta la singola operazione fuori borsa.

Al sistema di negoziazione "alle grida" (asta a chiamata) si sostituisce il nuovo sistema di negoziazione telematica, in cui gli intermediari inseriscono gli ordini di acquisto e di vendita attraverso il proprio terminale e i contratti sono automaticamente conclusi dal sistema non appena sia possibile l'abbinamento di proposte di segno opposto e compatibilità di prezzo. Ciò, oltre a favorire la formazione di prezzi più significativi, ha permesso di superare le barriere geografiche e di creare un unico mercato nazionale di dimensioni più consistenti e di elevata liquidità.

La legge sulle SIM permette alla Consob di costituire nuovi mercati per la negoziazione degli strumenti derivati, con la conseguente nascita del mercato italiano dei *future* (Mif) nel 1992, il mercato telematico delle opzioni (Mto) e *l'Italian derivatives market* (*Idem*).

La seconda tappa di notevole importanza è rappresentata dal D.Lgs del 23 luglio 1996, n.415, conosciuto come il "decreto Eurosim", di attuazione della direttiva 93/22/CEE, sui servizi di investimento nel settore dei valori mobiliari, e della direttiva 93/6/CEE, sull'adeguatezza patrimoniale delle imprese di investimento e degli enti creditizi. Le due direttive mirano all'eliminazione degli ostacoli alla circolazione dei servizi, sulla base di principi di armonizzazione minima e mutuo riconoscimento, per una piena integrazione dei mercati finanziari europei.

Gli aspetti fondamentali del decreto sono:

- principio del passaporto unico europeo, in base ai principi di base della direttiva le Sim italiane possono operare nei paesi europei, previa autorizzazione della Banca d'Italia, e le imprese di investimento comunitarie possono operare in Italia, sentito il parere di Consob e Banca d'Italia;
- equiparazione tra istituti di credito e società di intermediazione mobiliare, scompare ogni discriminazione nei confronti delle banche, che sono autorizzate, al pari delle Sim, ad esercitare attività di intermediazione mobiliare ed acquisto diretto di titoli;
- semplificazione delle attività rientranti nei servizi d'investimento, la consulenza in materia finanziaria e la sollecitazione al pubblico risparmio diventano servizi accessori;
- non diretta previsione dell'obbligo di concentrazione degli scambi nei mercati regolamentati, la direttiva comunitaria lascia agli stati membri la facoltà di inserire l'obbligo;
- privatizzazione della borsa, la gestione e regolamentazione dei mercati mobiliari italiani è affidata ad una società per azioni

mentre alle autorità pubbliche competono le attività di controllo.

L'ultima fondamentale tappa si ha con il D.Lgs. n.58/1998, intitolato "Testo unico delle disposizioni in materia di intermediazione finanziaria", conosciuto con la sigla TUIF. Le parti essenziali riguardano la disciplina dei mercati regolamentati, degli intermediari, degli emittenti e la disciplina di tutela delle minoranze.

Gli aspetti salienti del Testo Unico della Finanza sono:

- disciplina dei mercati regolamentati, sotto la pressione del costante confronto internazionale si avverte l'esigenza di potenziare l'imprenditorialità nell'offerta di servizi finanziari, con particolare attenzione alle funzioni di controllo delle autorità pubbliche;
- disciplina degli intermediari, nasce la figura del gestore unico, con l'affidamento in via esclusiva alle SGR di offrire congiuntamente servizi di gestione in monte ed individuale, mentre banche, Sim e società fiduciarie possono esercitare direttamente la gestione individuale di portafogli mobiliari;
- disciplina degli emittenti, l'obiettivo di fondo è quello di tutelare gli azionisti di minoranza, affinché si incentivi l'investimento azionario, con la conseguente diminuzione dei costi di raccolta di risorse da parte delle aziende e uno sviluppo del mercato;
- repressione dei reati di *insider trading*, che consiste nell'abuso di informazioni privilegiate per compiere delle transazioni finanziarie, e di aggiotaggio, che consiste nella diffusione di notizie false, esagerate o tendenziose, o nella realizzazione di operazioni simulate idonee a provocare una sensibile alterazione di prezzo degli strumenti finanziari o a dare l'apparenza di un mercato attivo.

GLI INTERMEDIARI FINANZIARI

Oltre alle banche e alle società di intermediazione mobiliare (SIM), sono intermediari finanziari vigilati anche le società di *leasing*, le società di *factoring*, le società di credito al consumo, le *holding* finanziarie, le *merchant bank*, le società di *venture capital* e le società per la cartolarizzazione dei crediti, ciascuna con diverse possibilità e funzioni operative.

Nel paragrafo seguente si analizzano le caratteristiche fondamentali dei due intermediari maggiormente conosciuti sul mercato.

Le banche e le SIM

L'operatività delle banche ha subito notevoli mutamenti nel corso del tempo, conseguenti all'evoluzione normativa di riferimento. Con la legge sulle SIM del 1991 si privano le banche della facoltà di negoziare titoli diversi da titoli di Stato, ma nel 1995 avviene una parziale liberalizzazione operativa, con la possibilità di negoziare direttamente sul sistema telematico gli strumenti derivati e i valori mobiliari relativi al proprio portafoglio. Infine nel 1996, in seguito al recepimento della Direttiva Eurosim e all'emanazione del TUIF nel 1998, si è giunti all'attuale situazione, con la completa equiparazione tra banche e SIM.

Fino al 1998 le banche si trovavano a dover scegliere tra la costituzione di una propria SIM e l'acquisizione di servizi di intermediazione da SIM esterne e indipendenti dalla banca stessa. Questo ha contribuito ad un fortissimo sviluppo delle società di intermediazione mobiliare, ora presenti in numero molto inferiore proprio a causa dell'attuale assetto normativo, che ha portato le banche a rivedere le proprie decisioni e a produrre internamente i servizi di intermediazione mobiliare.

Le società di intermediazione mobiliare (SIM), in base al TUIF, possono esercitare due tipologie di servizio, che sono i servizi di investimento e i servizi accessori.

Tra i servizi di investimento vi sono:

- la negoziazione in conto proprio, ossia l'attività di acquisto e di vendita (*dealing*);
- la negoziazione per conto terzi, attività di acquisto e di vendita in nome proprio ma per conto terzi (*brokering*);
- il collocamento di strumenti finanziari, che può avvenire con o senza garanzia di acquistare la parte di titoli che non dovesse essere collocata;
- la gestione su base individuale di portafogli di investimento per conto terzi, contrapposta alla gestione "in monte";
- la ricezione e la trasmissione degli ordini nonché la mediazione.

I servizi accessori sono invece i seguenti:

- la custodia e l'amministrazione degli strumenti finanziari;
- la locazione di cassette di sicurezza;

- la concessione di finanziamenti agli investitori, con prestiti in denaro o in titoli;
- la consulenza in materia di finanza d'impresa (*corporate finance*);
- i servizi connessi all'emissione e al collocamento degli strumenti finanziari, come ad esempio la costituzione di consorzi di garanzia e collocamento;
- la consulenza in materia di investimenti in strumenti finanziari;
- l'intermediazione in cambi.

Le SIM e le banche sostituiscono la figura dell'agente di cambio[2], al quale competeva la prestazione di servizi di negoziazione per conto terzi, collocamento senza assunzione di garanzia, gestione individuale di portafogli e ricezione e trasmissione di ordini.

GLI ORGANI DI VIGILANZA

Come accennato in precedenza, il Testo Unico della Finanza ha rafforzato notevolmente l'indipendenza degli organi preposti alla vigilanza, che sono definite Autorità Amministrative Indipendenti. Le principali istituzioni a cui è affidato il controllo sul mercato sono la Commissione Nazionale per le Società e la Borsa (CONSOB), la Banca d'Italia e il Ministero dell'Economia e delle Finanze, a cui si aggiungono alcune istituzioni che si occupano di particolari settori, come ad esempio l'Istituto per la vigilanza sulle assicurazioni private e di interesse collettivo (ISVAP) e la Commissione di vigilanza dei fondi pensione (COVIP).

L'articolo 5 del TUIF individua gli scopi e le competenze di vigilanza nel seguente modo:

> «La vigilanza sulle attività disciplinate dalla presente parte ha per scopo la trasparenza e la correttezza dei comportamenti e la sana e prudente gestione dei soggetti abilitati, avendo riguardo alla tutela degli investitori e alla stabilità, alla competitività e al buon funzionamento del sistema finanziario.»
>
> «La Banca d'Italia è competente per quanto riguarda il contenimento del rischio e la stabilità patrimoniale (...) la Consob è competente per quanto riguarda la trasparenza e la correttezza dei comportamenti.»

[2] Nel TUIF è previsto che gli agenti di cambio possano continuare a svolgere la propria attività fino a cessazione dal ruolo, che non può avvenire oltre il settantesimo anno di età.

Nei prossimi paragrafi si riassumono le competenze delle istituzioni appena menzionate.

La Consob

La Commissione Nazionale per le Società e la Borsa[3], è stata istituita con la legge 7 giugno 1974, n. 216, ed è un'autorità amministrativa indipendente con il compito di controllare il mercato mobiliare italiano. La sua attività è rivolta alla tutela degli investitori, all'efficienza e alla trasparenza del mercato. Per questi scopi assolve ai seguenti compiti:

- ***regolamenta*** la prestazione dei servizi di investimento da parte degli intermediari, gli obblighi informativi delle società quotate e le offerte al pubblico di strumenti finanziari;
- ***autorizza*** la pubblicazione dei prospetti informativi in caso di offerte pubbliche di vendita e dei documenti d'offerta in caso di offerte pubbliche di acquisto; l'esercizio dei mercati regolamentati; l'esercizio dell'attività di gestione accentrata degli strumenti finanziari; le iscrizioni agli Albi;
- ***vigila*** sulle società di gestione dei mercati e sulla trasparenza e l'ordinato svolgimento delle negoziazioni, nonché sulla trasparenza e la correttezza dei comportamenti degli intermediari e dei promotori finanziari;
- ***sanziona*** i soggetti vigilati, direttamente o formulando una proposta al Ministero del Tesoro, del Bilancio e della Programmazione economica;
- ***controlla*** le informazioni fornite al mercato dalle società quotate e da chi promuove offerte al pubblico di strumenti finanziari, nonché le informazioni contenute nei documenti contabili delle società quotate;
- ***accerta*** eventuali andamenti anomali delle contrattazioni su titoli quotati e compie ogni altro atto di verifica di violazioni delle norme in materia di abuso di informazioni privilegiate (*insider trading*) e di aggiotaggio su strumenti finanziari.

La Banca d'Italia

La Banca d'Italia è dotata in generale di poteri normativi e di vigilanza per la salvaguardia della stabilità del sistema finanziario nel suo complesso. Con riferimento agli intermediari, è di sua competenza la disciplina dell'adeguatezza patrimoniale, del contenimento del rischio nelle sue

diverse configurazioni, delle partecipazioni detenibili e dell'organizzazione amministrativa e contabile.

Nello svolgere le sue attività la Banca d'Italia può richiedere dati e notizie ai soggetti su cui indaga e può avviare indagini ed ispezioni. Effettua una continua supervisione del mercato all'ingrosso dei titoli di Stato (MTS) e sul mercato interbancario dei depositi (MID), ai fini del controllo in materia di politica monetaria.

Il Ministero dell'Economia e delle Finanze

Il Ministero dell'Economia e delle Finanze è in continuo contatto con la Consob, con cui provvede allo scambio di informazioni sugli eventi di maggior rilievo, per informarne in seguito il Parlamento.

Nei confronti degli operatori del mercato mobiliare può esercitare i seguenti poteri:

- poteri normativi, con cui si definiscono i requisiti di onorabilità e professionalità degli esponenti aziendali degli intermediari;
- poteri di autorizzazione dei sistemi di indennizzo a favore degli investitori;
- poteri sanzionatori, i quali sono esercitati in caso di insolvenza degli intermediari, di cui può stabilire lo scioglimento degli organi amministrativi, la revoca delle autorizzazioni o la liquidazione coatta.

LE AZIONI

Le azioni sono degli strumenti di capitale che comportano la partecipazione diretta di chi li detiene alla gestione e al rischio d'impresa. Ogni azione rappresenta una quota del capitale sociale della società emittente e l'insieme delle azioni detenute esprime la partecipazione del socio al rischio d'impresa. La disciplina giuridica di questi strumenti è contenuta nel Codice Civile (2003), Capo V, Sezione V, contenente gli articoli che vanno dal 2346 al 2362. Il valore del capitale sociale è determinabile moltiplicando il numero totale di azioni in circolazione per il valore nominale della singola azione, in quanto le azioni non possono essere emesse per somma inferiore al loro valore nominale. Valore dell'azione e Capitale sociale sono quindi legate dalla seguente relazione:

$$\text{Valore nominale azione} = \frac{\text{Capitale Sociale}}{\text{Numero azioni emesse}}$$

[3] Informazioni reperibili al sito www.consob.it

Il reddito delle azioni, a differenza di quello dei titoli di debito, come le obbligazioni o i Titoli di Stato, è variabile, non garantendo ai possessori un ritorno di misura e tempistiche determinabili. L'azionista percepisce dei dividendi che dipendono direttamente dall'effettivo conseguimento di utili da parte della società e dalla decisione di quest'ultima a favore della distribuzione, poichè gli utili potrebbero essere destinati ad accantonamenti a riserve.

Non esiste una data di scadenza fissata e il rimborso del titolo può avvenire solamente in casi particolari. Se il socio detentore delle azioni desidera disinvestire, è obbligato a rivolgersi al mercato e vendere i titoli al prezzo corrente.

Ogni socio possiede una serie di diritti e di doveri stabiliti dal Codice Civile e dallo statuto della società di cui possiede le partecipazioni. I diritti sono classificabili in:

a) diritti amministrativi;

b) diritti patrimoniali;

c) diritti di natura mista.

Tra i diritti amministrativi vi sono il diritto di voto, che consiste nella facoltà dell'azionista di esercitare un voto per ogni azione posseduta; il diritto di partecipazione all'assemblea, ordinaria o straordinaria, e la possibilità di impugnare le delibere assembleari; il diritto di informazione, che consiste nella possibilità di visionare il libro dei soci, il libro delle adunanze e le deliberazioni assembleari; il diritto, esercitabile singolarmente o unitamente ad altri soci al fine di raggiungere una certa percentuale di capitale sociale, di convocare l'assemblea e denunciare gravi irregolarità commesse da amministratori e sindaci. Alcuni di questi diritti, come ad esempio il diritto di voto, sono limitati per alcune particolari categorie di azioni.

I diritti patrimoniali consistono nel diritto agli utili e nel diritto alla quota di patrimonio netto risultante dalla liquidazione. La parte di utili e la quota di liquidazione sono proporzionali al numero di azioni possedute.

Tra i diritti di natura mista vi sono il diritto di opzione e il diritto di recesso. Il primo indica il diritto che spetta ai vecchi azionisti di sottoscrivere, in proporzione alle azioni possedute, azioni di nuova emissione, in occasione di aumento di capitale a pagamento, prima di terzi estranei alla società. L'obiettivo è quello di dare la possibilità all'azionista di mantenere inalterata la propria quota relativa di partecipazione (percentuale del capitale sociale posseduta), mantenendo così inalterate le possibilità di influenzare il voto in assemblea e le quote percentuali di utile percepibili. Nel caso di aumenti di capitale gratuiti, il diritto di opzione

muta in diritto di "assegnazione", e i vecchi azionisti godono di una prelazione nella distribuzione gratuita delle nuove azioni, in proporzione ai titoli già posseduti. Il diritto di recesso invece consiste nella possibilità di ottenere la liquidazione delle quote nel caso in cui si modifichi l'oggetto sociale o si trasferisca all'estero la sede sociale.

Oltre ai diritti appena esposti esistono gli obblighi dell'azionista, che sono il conferimento e l'astensione dal voto in assemblea in caso di conflitto d'interessi.

Le azioni emesse dalle società per azioni quotate in borsa o da società non quotate ma che presentano un azionariato molto diffuso, a seguito del processo di "dematerializzazione" introdotto dal D.Lgs. 24 giugno 1998, n.213, non sono rappresentate da titoli di credito, ma sono definibili come quote "ideali" di capitale sociale e sono depositate presso il Monte Titoli.

Diverse categorie di azioni

Le principali categorie di azioni individuate dal Codice Civile e dal Testo Unico della Finanza sono le azioni ordinarie, le azioni privilegiate e le azioni di risparmio.

Azioni ordinarie. Le azioni di questa categoria incorporano tutti i diritti patrimoniali e amministrativi previsti dalla legge. Conferiscono il potere di voto nelle assemblee ordinarie e straordinarie, grazie al quale i soci possono contribuire alla determinazione degli indirizzi dell'impresa, scegliendo gli amministratori ai quali affidare la gestione e partecipando direttamente ad alcune decisioni di importanza strategica, quali ad esempio gli aumenti di capitale o lo scioglimento della società. Le azioni ordinarie sono nominative e la persona giuridica o fisica a cui sono intestate è registrata nel libro dei soci. Il diritto al rimborso, in caso di liquidazione o scioglimento della società, e al dividendo è ridotto rispetto alle altre categorie di azioni, in quanto l'azionista ordinario è soddisfatto dopo aver remunerato o rimborsato i soggetti detentori di azioni privilegiate o di risparmio.

Azioni privilegiate. Per questa categoria i "privilegi" sono stabiliti non nella legge, ma nello statuto della società emittente. Rispetto alle azioni ordinarie presentano maggiori diritti patrimoniali. L'azionista concorre in via prioritaria alla distribuzione degli utili, rispetto agli azionisti ordinari, fino al raggiungimento di un dividendo minimo, la cui entità corrisponde ad una percentuale fissa del valore nominale delle azioni. Ciò non comporta necessariamente la superiorità di remunerazione rispetto alle azioni ordinarie, ma garantisce un livello minimo se avvenissero delle distribuzioni di utili. Inoltre è possibile, per un numero limitato di esercizi stabiliti dallo statuto, cumulare i dividendi che non sono distribuiti. Questo vantaggio patrimoniale determina una minor possibilità di partecipazione alla gestione

della società. In particolare è permessa la partecipazione alle sole assemblee straordinarie.

Azioni di risparmio. Queste azioni possono essere emesse soltanto da società quotate in mercati regolamentati italiani o di altri paesi dell'Unione Europea e sono totalmente prive del diritto di voto. L'obiettivo principale è quello di rendere appetibili le azioni come investimento per il piccolo risparmiatore, che non sempre risulta essere interessato alle possibilità di influenzare la gestione offerte dalle normali azioni, ma piuttosto a maggiori privilegi patrimoniali, i quali sono stabiliti dallo statuto. Inoltre gli azionisti di risparmio, in caso di aumento di capitale sociale, hanno diritto a ricevere in opzione nuove azioni della stessa categoria, ovvero, in mancanza o per la differenza, nell'ordine, azioni privilegiate e ordinarie.

L'emissione di azioni privilegiate e azioni di risparmio permette al gruppo di comando di raccogliere capitale di rischio, senza perdere il controllo della società. Per evitare che questo comporti l'affidamento ad un ristretto numero di azionisti del governo della società, la quantità emessa per queste due tipologie di azioni non può superare il 50% del capitale sociale.

LA SEGMENTAZIONE DEGLI STRUMENTI FINANZIARI

Una prima classificazione dei titoli azionari può essere effettuata utilizzando come parametro la capitalizzazione di borsa. I titoli sono quindi classificabili in:

- *large cap stock*, sono titoli ad ampia capitalizzazione, ossia con capitalizzazione superiore a 800 milioni di Euro;
- *mid cap* o *medium cap stock*, sono titoli a media capitalizzazione, ossia con capitalizzazione compresa tra 300 e 800 milioni di Euro;
- *small* e *micro cap stock*, sono titoli a bassa capitalizzazione, ossia con capitalizzazione inferiore a 300 milioni di Euro, detti anche titoli "sottili", a causa della loro ridotta liquidità che genera bassi volumi delle contrattazioni.

Una seconda classificazione possibile è la suddivisione tra *value stock*, che sono titoli emessi da società operanti in settori tradizionali, con un passato caratterizzato da redditività e solidità patrimoniale, e *growth stock*, che sono azioni di società operanti in settori tecnologici e innovativi con elevata possibilità di rapido sviluppo, ma che implicano anche un alto livello di rischio.

In base alle classificazioni appena esposte le società sono ammesse alla quotazione in diversi segmenti del mercato di borsa. Borsa Italiana offre le seguenti possibilità di quotazione:

- Borsa o Mercato Telematico Azionario (MTA), dedicato alle società operanti in settori tradizionali, suddivisibile in tre segmenti, che sono *blue chip*, in cui sono quotate le *large cap stock*, STAR (Segmento Titoli con Alti Requisiti), in cui sono quotate le aziende a media-bassa capitalizzazione che rispondono ad elevati requisiti di liquidità e trasparenza informativa, e ordinario, in cui si quotano tutte le aziende non rientranti nei segmenti precedenti;
- Nuovo Mercato, dedicato alle aziende ad alto potenziale di crescita e ad elevata innovazione;
- Mercato Ristretto, dedicato alle aziende operanti in settori tradizionali che non intendono chiedere l'ammissione in Borsa o sul Nuovo Mercato.

L'ammissione alla quotazione ufficiale può avvenire di diritto o su richiesta dell'emittente. Nel primo caso, riguardante esclusivamente i titoli di Stato, la quotazione non è soggetta ad un giudizio di merito da parte degli organi competenti.

Nella pagina seguente, uno schema illustra sinteticamente i principali requisiti di ammissione ai vari segmenti del MTA (2003).

	BLUE CHIP	Segmento Ordinario	STAR
Requisiti degli emittenti	• Capacità di generare ricavi in condizioni di autonomia gestionale • Pubblicazione e deposito degli ultimi 3 bilanci annuali[1] • Ultimo bilancio sottoposto a revisione contabile • L'attivo di bilancio o i ricavi dell'emittente non devono essere rappresentati in misura preponderante dall'investimento o dai risultati dell'investimento in una società quotata		
Requisiti delle azioni	• Capitalizzazione di mercato superiore a € 800 milioni • Flottante[2] minimo del 25%	• Capitalizzazione di mercato pari almeno a € 20 milioni e inferiore a € 800 milioni • Flottante[2] minimo del 25%	• Capitalizzazione di mercato pari almeno a € 20 milioni e inferiore a € 800 milioni • Flottante[2] minimo: - Nuove quotazioni: 35% - Passaggi da altri segmenti e permanenza: 20%
Obblighi di informativa e trasparenza	• Obblighi di informativa secondo la normativa vigente: - Informativa periodica (trimestrale, semestrale e annuale) - Informativa *price sensitive* - Informativa sul pagamento dei dividendi - Informativa relativa alle operazioni straordinarie (aumenti di capitale, Opa, raggruppamento o frazionamento di azioni, fusioni, scissioni) - Comunicazione relativa alle variazioni del capitale sociale - Informativa sulle modifiche statutarie rilevanti		• Obblighi aggiuntivi: - Nomina di un *Investor Relation Manager* - Pubblicazione sul proprio sito in italiano e in inglese di: ▪ Dati di bilancio: annuali, semestrali e trimestrali ▪ Comunicati stampa - Pubblicazione delle relazioni trimestrali entro 45 giorni
Requisiti di *Corporate Governance*	• Adozione del Codice di Comportamento diretto a disciplinare le operazioni di *Internal Dealing* • Adesione facoltativa al codice di Autodisciplina, con obbligo di dichiarazione annuale sulla sua implementazione		• Obblighi aggiuntivi: - Nomina di amministratori non esecutivi e indipendenti nel Consiglio di Amministrazione - Nomina di un Comitato per il Controllo Interno - Remunerazione incentivante per il *top management*
Operatori del mercato	• Nomina di uno Sponsor che accompagni la società in quotazione e la supporti per almeno 1 anno producendo analisi finanziarie e organizzando incontri con investitori		• Obblighi aggiuntivi: - Nomina di uno Specialista che garantisca la liquidità del titolo, produca analisi finanziarie e organizzi incontri con investitori

(1) In via eccezionale può essere accettato un numero inferiore di bilanci.
(2) Questo requisito si applica alle sole azioni ordinarie.

ORGANIZZAZIONE DEI MERCATI

Lo scopo principale del mercato è quello di favorire l'incontro tra gli investitori e lo scambio di titoli in modo rapido, economico e il più possibile trasparente dal punto di vista informativo. Come sottolineato in Damilano e

altri (2002), nel tempo si sono sviluppate diverse logiche di organizzazione degli scambi. La forma primordiale di mercato è senz'altro quella in cui vi è una ricerca autonoma della controparte, e chiunque voglia negoziare un determinato strumento finanziario si attiva su base individuale per cercare un altro investitore interessato alla compravendita.

I costi e i tempi per la stipulazione dei contratti hanno portato nel tempo a forme più organizzate di mercato. I *broker* rappresentano il primo tentativo di superamento dei limiti della ricerca diretta delle controparti. Il loro compito è semplicemente quello di aiutare le potenziali controparti ad entrare in contatto, lucrando una commissione per il servizio reso e senza assumere alcuna posizione negoziale in proprio. Nei moderni mercati finanziari non è più identificabile alcun settore basato solo sull'azione dei *broker*. Attualmente sono identificabili principalmente due forme organizzative che sono: il mercato di *market-makers*, detto anche *quote-driven*, e il mercato ad asta, detto *order-driven*.

Il mercato quote driven

Nel mercato di *market-maker* (*quote driven*) operano fondamentalmente intermediari che, possedendo una certa quantità di una attività finanziaria, si pongono costantemente come controparti degli altri operatori presenti sul mercato. Il termine "*market-maker*" deriva dal fatto che questi operatori si espongono al mercato quotando due prezzi per ogni titolo con cui sono disposti a effettuare compravendite, uno al quale sono disposti ad acquistare (detto *bid price* o prezzo denaro) un determinato titolo, e uno al quale sono disposti a vendere (detto *ask price* o prezzo lettera). Quindi, essendo continuamente disposti ad effettuare scambi a chiunque ne faccia richiesta, sono indicati come coloro che "fanno il mercato". Il compenso della loro attività è dato dalla differenza (*spread*) tra i due prezzi, denaro e lettera, ma la parte preponderante degli utili o delle perdite di questi oggetti deriva dai guadagni o perdite in conto capitale legati all'attività di compravendita dei titoli.

Gli operatori che usufruiscono delle quotazioni dei *market-maker* sono indicati invece come *market-taker*. I *market-maker* appartengono alla categoria più generale dei *dealer*, ossia soggetti che operano per conto proprio, figure complementari ai *broker*, i quali operano solamente per conto terzi. La funzione principale che assolvono consiste nel fluidificare il funzionamento del mercato, dando la sicurezza di poter trovare una controparte per le contrattazioni, e spesso interi mercati ruotano attorno alle contrattazioni tra i diversi *market-maker*. Nei mercati regolamentati questo tipo di intermediario è istituzionalizzato e svolge un'importante funzione di sostegno degli scambi.

Il mercato order driven

Al mercato organizzato attraverso l'attività svolta dai *market-maker*, si contrappone il funzionamento detto "ad asta". In questo caso si fanno convergere tutti gli operatori interessati ad un determinato strumento finanziario in un certo luogo. Il "luogo" oggi è inteso non solamente nel senso fisico del termine, bensì si parla di luogo intendendo tutte le possibilità di contatto tra gli operatori, sia fisiche che telematiche. In questo modo i diversi operatori emettono le proprie proposte e si arriva alla conclusione dei contratti. Alla concentrazione di tutti gli operatori nello stesso luogo consegue un'elevata concorrenza, che permette che si formino dei prezzi che sono effettivamente espressione della domanda e dell'offerta di una determinata attività. Le due varianti principali di questa tipologia organizzativa sono i mercati ad asta a chiamata e i mercati ad asta continua.

Nei mercati ad asta a chiamata, l'incontro degli operatori avviene in uno o più momenti prestabiliti della giornata di contrattazione. Nel corso di questi momenti si fissa, attraverso un processo dialettico tra domanda e offerta, un valore di prezzo dell'attività considerata tale per cui possa essere soddisfatto il maggior numero di contratti in base alla serie di offerte effettuate dagli operatori. Nelle offerte sono fissati il prezzo limite e la quantità che si è interessati ad acquistare o a vendere. Al termine dell'asta sono soddisfatte tutte le offerte di acquisto il cui prezzo limite è superiore o uguale al prezzo fissato e tutti gli ordini di vendita il cui prezzo limite è inferiore o uguale al prezzo fissato. Questa logica di funzionamento è tipica delle contrattazioni "alle grida" dei mercati fisici. In questo caso, a determinati orari, i diversi strumenti negoziati sono "chiamati" e gli operatori interessati si riuniscono attorno al *corbeille*, che è la postazione del banditore, il quale, partendo dal prezzo fissato nell'asta precedente, lo alza o lo abbassa progressivamente fino a trovare una quotazione che renda il numero di contratti di acquisto pari al numero di contratti di vendita. Questo tipo di contrattazione è oggi praticamente estinto, ma la logica sottostante è riprodotta nei mercati telematici, in cui il "banditore" è sostituito da un algoritmo informatico che, dopo un periodo di raccolta degli ordini, fissa il prezzo che meglio rappresenta le curve di domanda e offerta.

Il principale difetto dell'asta a chiamata è l'impossibilità di concludere contratti al di fuori dei momenti in cui avvengono le contrattazioni. Nei mercati ad asta continua, utilizzati prevalentemente nei circuiti telematici, gli ordini di acquisto e di vendita confluiscono tutti in un *book* di negoziazione a cui hanno accesso tutti gli operatori autorizzati e i contratti sono conclusi continuamente, nel corso di tutta la giornata, non appena due volontà negoziali risultano compatibili in termini di prezzo e due contratti di segno opposto possono essere abbinati. Il prezzo delle attività oggetto dei contratti varia continuamente durante la giornata in base ai prezzi a cui

sono conclusi i contratti. La continuità delle contrattazioni permette di operare in ogni momento della giornata favorendo concorrenza e trasparenza informativa.

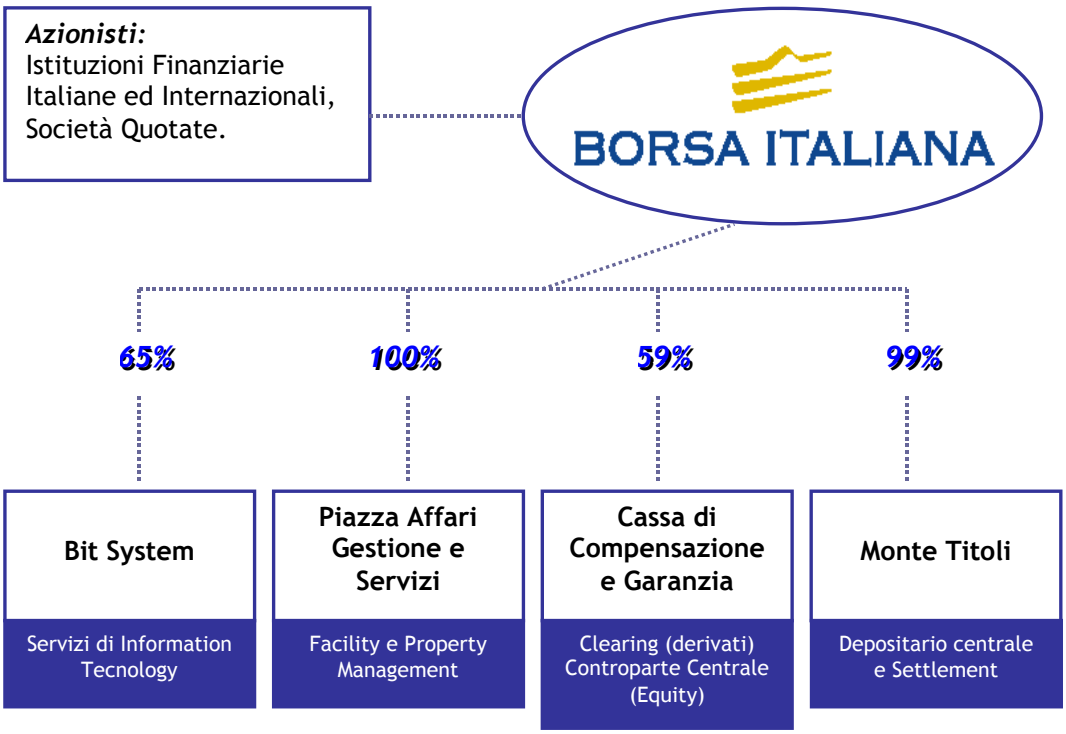
BORSA ITALIANA S.P.A.

Il 2 gennaio 1998 Borsa Italiana riceve dalla Consob l'autorizzazione all'esercizio dell'attività di organizzazione e gestione dei mercati, ponendo fine al processo di privatizzazione avviato nel 1996 con il decreto Eurosim.

Nei paragrafi seguenti, dopo un'analisi delle funzioni e della struttura di Borsa Italiana, si trattano i principali aspetti regolamentari della gestione delle contrattazioni sul Mercato Telematico Azionario e sul Mercato Ristretto. Su questa base si struttura il modello di simulazione *JavaSum*, il cui realismo è strettamente legato alla riproduzione virtuale di procedure di negoziazione (in particolare delle aste) il più possibile simili a quelle effettivamente seguite nel mercato reale.

La gestione delle contrattazioni

Come illustrato in Borsa Italiana Spa (2003), il Gruppo Borsa Italiana organizza e gestisce i mercati mobiliari e le attività di *post-trading* in Italia, cercando di promuovere l'efficienza e lo sviluppo dei mercati gestiti, massimizzandone la liquidità, la trasparenza e la competitività. Tutti i mercati organizzati e gestiti da Borsa Italiana sono telematici e *order driven*. La struttura del gruppo è sintetizzata nello schema seguente:



Le principali attività del gruppo si possono riassumere come segue:

- *MARKET INTELLIGENCE, STRATEGIE E PROMOZIONE:*
  - studio e analisi delle prospettive di evoluzione dei sistemi finanziari e dei mercati;
  - *marketing* strategico, promozione e sviluppo dei mercati.
- *REGOLAMENTAZIONE:*
  - definizione del quadro regolamentare dei mercati organizzati e gestiti da Borsa Italiana;
  - istruttoria e ammissione alla quotazione degli strumenti finanziari;
  - istruttoria e ammissione alla negoziazione degli operatori finanziari.
- *IT E OPERATION:*
  - piattaforma di negoziazione;
  - servizi di *post trading* (*clearing*, garanzia e *settlement*);
  - distribuzione dei dati relativi ai mercati di Borsa Italiana.
- *INFORMATIVA E VIGILANZA:*
  - diffusione delle informazioni su fatti rilevanti e comunicazioni societarie attraverso il *Network Information System* (*NIS*);
  - sorveglianza su svolgimento delle negoziazioni e trasparenza di mercato;
  - diffusione dei dati giornalieri delle sedute di Borsa Italiana (Listino Ufficiale) e di altre informazioni di mercato.
- *SERVIZI ALLA CLIENTELA:*
  - attività di informazione e promozione degli strumenti quotati;
  - creazione e aggiornamento del sito internet;
  - attività di formazione per il mercato finanziario;
  - infrastrutture e servizi per la realizzazione di eventi;
  - diffusione di dati e prodotti informativi.

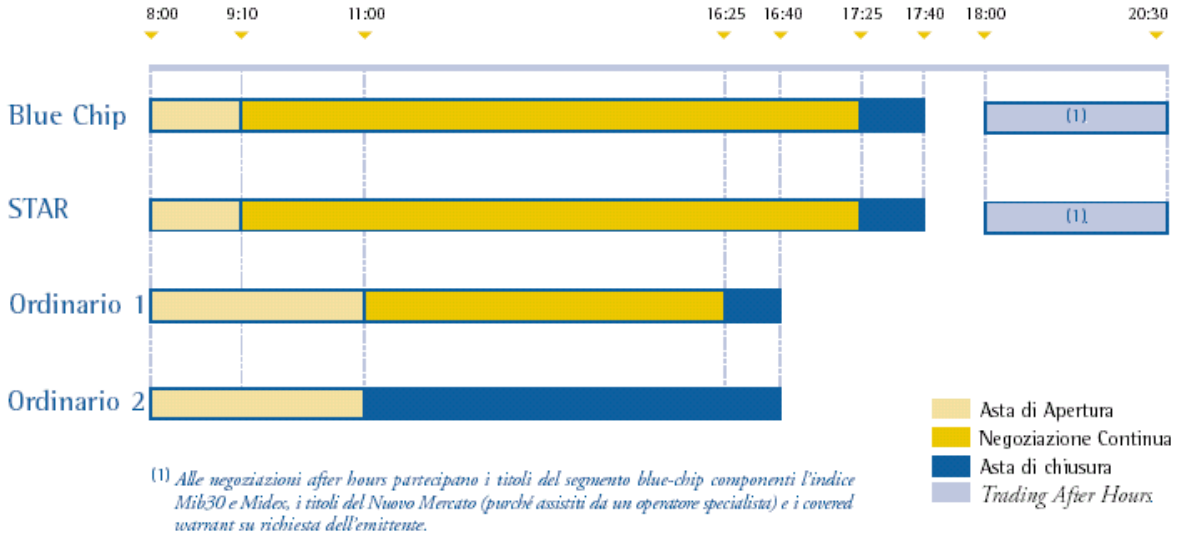
In tutti i mercati gestiti da Borsa Italiana Spa si utilizza una rete telematica per lo svolgimento delle contrattazioni. La società che fornisce e cura questo servizio è la Società Interbancaria per l'Automazione (SIA). Il sistema di contrattazione è di tipo *order-driven* e la giornata si articola in diverse fasi, le quali possono essere estese o ridotte temporalmente in base alla prevista liquidità dei singoli titoli. Il principio generale è quello di ridurre la fase di negoziazione continua per concentrare l'affluenza degli

ordini, facilitando il regolare funzionamento delle negoziazioni nel caso in cui il quantitativo di ordini sia limitato.

Nella tabella sottostante sono riportati gli orari di contrattazione dettati dalle Istruzioni (2003).

	MTA segmenti blue-chip e STAR	**MTA segmento ordinario (classe 1)**	**Nuovo mercato**	**MTA segmento ordinario (classe 2) e Mercato Ristretto**
Asta di apertura	Dalle 8.00 alle 9.30	Dalle 8.00 alle 11.00	Dalle 8.00 alle 9.30	Dalle 8.00 alle 11.00
Negoziazione continua	Dalle 9.30 alle 17.25	Dalle 11.00 alle 16.25	Dalle 9.30 alle 17.25	–
Asta di chiusura	Dalle 17.25 alle 17.40	Dalle 16.25 alle 16.40	Dalle 17.25 alle 17.40	Dalle 11.00 alle 16.40

Nello schema che segue, tratto da MTA (2003), sono schematizzati gli orari appena descritti e si può notare anche il *Trading After Hours*, dalle 18.00 alle 20.30.



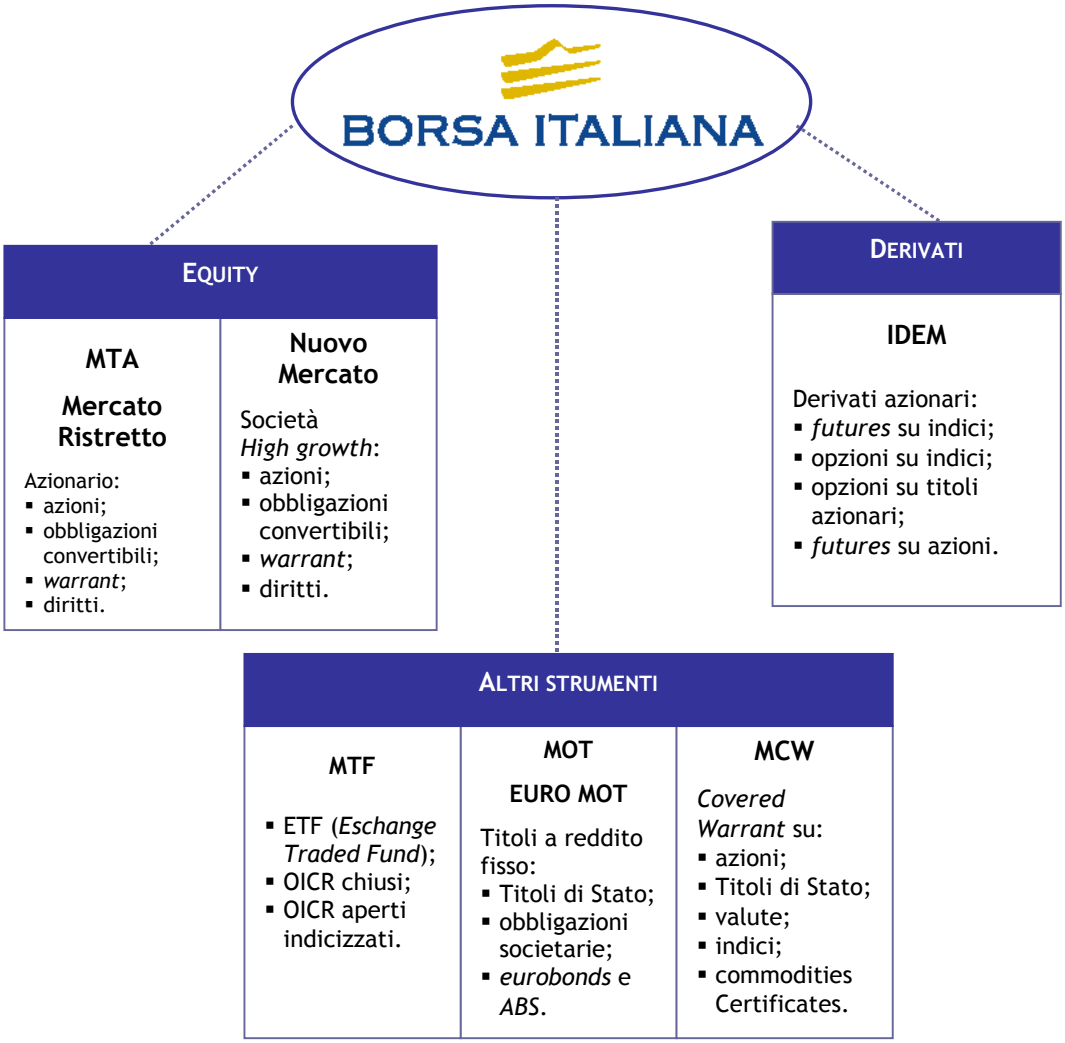
Per aumentare liquidità ed efficienza dei mercati, solitamente è prevista la presenza di *market-maker* a supporto del funzionamento del meccanismo di contrattazione ad asta. La presenza di questi operatori specialisti garantisce, in particolare per strumenti con scarsa liquidità, un

quantitativo minimo di proposte negoziali in acquisto e in vendita su un dato strumento, assicurando agli investitori la costante possibilità di trovare una controparte per le operazioni di compravendita desiderate.

Il Mercato di Borsa è articolato nei seguenti comparti:

- Mercato telematico azionario (MTA);
- Mercato telematico dei *covered warrant* (*MCW*);
- Mercato "*after hours*" (*TAH*);
- Mercato telematico delle obbligazioni e dei titoli di Stato (MOT);
- Mercato telematico delle euro-obbligazioni, obbligazioni di emittenti esteri e *asset backed securities* (EuroMOT).

Competono a Borsa Italiana inoltre il Mercato Ristretto e il Mercato dei Derivati (IDEM). Nello schema sottostante sono riportati la struttura e i collegamenti tra i vari mercati.



Il Regolamento (2003) disciplina l'organizzazione e la gestione dei seguenti mercati regolamentati e ne stabilisce le condizioni e le modalità di organizzazione e funzionamento. In particolare sono trattati nel Regolamento i seguenti punti:

a) le condizioni e le modalità di ammissione, esclusione e sospensione degli strumenti finanziari dalle negoziazioni;

b) le condizioni e le modalità di ammissione, esclusione e sospensione degli operatori dalle negoziazioni;

c) le condizioni e le modalità di svolgimento delle negoziazioni e di funzionamento dei servizi ad esse strumentali;

d) gli obblighi degli operatori e degli emittenti;

e) le modalità di accertamento, pubblicazione e diffusione dei prezzi e delle informazioni.

Sempre secondo il Regolamento (2003), Borsa Italiana esercita le proprie funzioni, in particolare quelle di ammissione, sospensione e revoca degli strumenti finanziari e degli operatori dalle negoziazioni, e quella di vigilanza sui mercati, secondo modalità non discriminanti e sulla base di procedure definite in via generale. Inoltre si dota e mantiene un assetto organizzativo idoneo a prevenire potenziali conflitti di interesse e a mantenere un sistema di controllo interno che vigili sul rispetto delle leggi, dei regolamenti e delle procedure interne.

Nel paragrafo seguente sono esaminati gli aspetti salienti del regolamento del MTA e del Mercato Ristretto gestiti da Borsa Italia S.p.A., in vigore dal 17 novembre 2003.

Modalità di negoziazione nel MTA e nel Mercato Ristretto

Nel mercato telematico azionario sono negoziabili, per qualunque quantitativo, azioni, obbligazioni convertibili, diritti di opzione, *warrant* e quote o azioni di OICR. Borsa Italiana si riserva di stabilire per singolo strumento finanziario un quantitativo minimo negoziabile qualora lo richiedessero esigenze di funzionalità del mercato, di agevole accesso da parte degli investitori e di economicità nell'esecuzione degli ordini.

La liquidazione dei contratti di compravendita, aventi come oggetto gli strumenti finanziari, avviene il terzo giorno di borsa aperta successivo alla stipulazione, qualora siano relativi ad azioni, obbligazioni convertibili, *warrant* e quote o azioni di OICR, o nel termine stabilito da Borsa Italiana qualora siano relativi a diritti d'opzione.

Le negoziazioni si possono svolgere secondo le modalità di asta e di negoziazione continua, articolate nelle seguenti fasi:

1. asta di apertura, suddivisa a sua volta nelle fasi di determinazione del prezzo teorico d'asta di apertura (detta di "pre-asta"), validazione del prezzo teorico d'asta di apertura (detto "validazione") e conclusione dei contratti (detta "apertura");
2. negoziazione continua;
3. asta di chiusura, articolata a sua volta nelle fasi di determinazione del prezzo teorico d'asta di chiusura (detta "pre-asta"), validazione del prezzo teorico d'asta di chiusura (detta "validazione") e conclusione dei contratti (detta "chiusura").

Le proposte di negoziazione contengono le informazioni relative allo strumento finanziario da negoziare, alla quantità, al tipo di operazione, al tipo di conto nonché alle condizioni di prezzo. L'immissione, la modifica e la cancellazione delle proposte di negoziazione possono essere effettuate dagli operatori sia nelle fasi di pre-asta, sia nella negoziazione continua. Le proposte sono automaticamente ordinate nel mercato per ciascuno strumento finanziario in base al prezzo. Nel caso di contratti di acquisto, l'ordine è decrescente di prezzo, mentre nel caso di contratti di vendita è crescente. A parità di prezzo, gli ordini immessi sono soddisfatti in base alla priorità temporale determinata dall'orario di immissione. Le proposte modificate perdono la priorità temporale acquisita se la modifica implica un aumento del quantitativo o una variazione del prezzo. I quantitativi oggetto delle proposte di negoziazione devono essere almeno pari al lotto minimo di negoziazione. Borsa Italiana può comunque imporre, con apposito provvedimento, limiti all'immissione o alla modifica di proposte di negoziazione in termini di frequenza giornaliera, numero complessivo giornaliero, rapporto tra proposte e contratti conclusi.

Ai fini del controllo automatico della regolarità delle contrattazioni, nelle Istruzioni (2003) sono stabilite le condizioni di negoziazione. Il limite massimo di variazione del prezzo delle proposte rispetto al prezzo di controllo di azioni, *warrant*, diritti, obbligazioni convertibili e quote di fondi chiusi è pari al 90%, mentre per quote o azioni di OICR indicizzati è del 10%. Nel caso dei contratti, il limite massimo di variazione dei prezzi rispetto al prezzo di controllo è pari al 10% per azioni e quote di fondi chiusi, al 30% per *warrant* e diritti, e al 5% per obbligazioni convertibili e quote o azioni di OICR indicizzati. La variazione dei prezzi tra due contratti consecutivi non deve essere superiore al 5% nel caso di azioni, *warrant* e quote di fondi chiusi, 15% nel caso di diritti, e al 2.5% nel caso di obbligazioni convertibili e quote o azioni di OICR indicizzati.

Fase di pre-asta

Le proposte di negoziazione nelle fasi di pre-asta possono essere immesse con o senza limite di prezzo (dette proposte "al prezzo d'asta") e possono essere specificate con le seguenti modalità di esecuzione:

- "valida fino alla cancellazione", in tal caso l'eventuale quantità ineseguita della proposta permane nel mercato fino al termine della seduta, quando è automaticamente cancellata;
- "valida fino alla data specificata", quindi la proposta permane nel mercato mantenendo la priorità temporale originaria per la quantità ineseguita fino alla data di scadenza specificata; le proposte sono cancellate automaticamente solo in occasione di stacco dividendi, aumento di capitale, frazionamento, raggruppamento, fusione, scissione, qualora il loro prezzo ecceda i limiti percentuali stabiliti da Borsa Italiana o qualora, a seguito di variazioni del lotto minimo di negoziazione, le proposte non siano più compatibili con i nuovi parametri di quantità o di prezzo;
- "esegui e cancella", la proposta è eseguita, anche parzialmente, per la quantità disponibile in asta e l'eventuale saldo residuo è cancellato automaticamente al termine della fase di asta;
- "immesse al prezzo di asta", in questo caso le proposte assumono dinamicamente il prezzo al quale avrebbero le maggiori possibilità di essere eseguite.

Fase di negoziazione continua e pre-asta di chiusura

Nella fase di negoziazione continua, oltre alle modalità di immissione delle proposte già illustrate per la fase di pre-asta, possono essere immessi ordini con le seguenti modalità:

- "esegui quantità minima specificata": la proposta è eseguita anche parzialmente almeno per il quantitativo minimo e alle condizioni di prezzo indicate o migliori; se detto quantitativo non è disponibile nel mercato la proposta è cancellata automaticamente;
- "tutto o niente": la proposta è eseguita unicamente per l'intero quantitativo indicato al momento dell'inserimento e alle condizioni di prezzo indicate; se ciò non è possibile la proposta è cancellata automaticamente;
- se le proposte sono immesse senza limite di prezzo, possono essere specificate con la modalità di esecuzione "esegui comunque", che fa sì che la conclusione dei contratti avvenga automaticamente ai prezzi

delle proposte di segno contrario più convenienti. Se invece sono immesse con prezzo limite, è possibile la modalità "esponi al raggiungimento del prezzo specificato": la proposta è accettata, ma è esposta sul mercato solo al raggiungimento del prezzo indicato.

Durante la negoziazione continua e la pre-asta di chiusura possono essere immesse con o senza limite di prezzo proposte "valide solo in asta di chiusura". Durante la fase di negoziazione continua le proposte non sono esposte sul mercato e sono conservate nel sistema per la loro partecipazione alle negoziazioni nella fase di pre-asta. L'ordine dipende dalla priorità temporale determinata dall'orario di immissione e le proposte non eseguite, o eseguite solo parzialmente, sono automaticamente cancellate al termine della seduta.

Non è consentita l'immissione di proposte con limite di prezzo aventi prezzi superiori o inferiori ai limiti percentuali di variazione massima dei prezzi stabiliti da Borsa Italiana. I prezzi delle proposte di negoziazione possono essere multipli di valori ("*tick*") stabiliti per ogni strumento finanziario e per ogni seduta di Borsa in relazione ai prezzi di riferimento dei singoli strumenti.

Le proposte di negoziazione aventi l'indicazione "valida fino alla data specificata" possono essere immesse con validità (data specificata) compresa in un intervallo di tempo non eccedente trenta giorni di calendario dalla data di immissione.

Fase di validazione

La fase di validazione ha inizio una volta chiusa la fase di pre-asta. Al termine della fase di validazione, qualora sia stato determinato un prezzo teorico d'asta, questo è considerato valido ed è assunto come prezzo d'asta per la conclusione dei contratti se il suo scostamento dal prezzo di controllo non supera la percentuale di variazione massima stabilita da Borsa Italiana nelle Istruzioni (2003).

Limitatamente alla fase di asta di apertura, nel caso in cui lo scostamento del prezzo teorico d'asta dal prezzo di controllo superi la percentuale di variazione massima, è riattivata la fase di pre-asta per un intervallo di tempo stabilito da Borsa Italiana nelle Istruzioni (2003). Qualora non sia stato determinato un prezzo teorico d'asta, al termine della validazione le proposte di negoziazione sono trasferite automaticamente alla negoziazione continua con la priorità temporale delle proposte originarie e con il prezzo:

a) della proposta originaria, se si tratta di proposte con limite di prezzo;

b) della migliore proposta con limite di prezzo presente sul mercato alla fine della fase di pre-asta, se si tratta di proposte al prezzo di apertura;

c) pari a quello di controllo, se si tratta di proposte al prezzo di apertura e se al termine della fase di pre-asta non sono presenti proposte con limite di prezzo.

Nella fase di asta di chiusura, qualora non sia stato determinato un prezzo teorico d'asta, ovvero non sia stato validato, le proposte di negoziazione, ove sia stata specificata la modalità di esecuzione "valida sino a data specificata", sono trasferite automaticamente alla fase di pre-asta di apertura del giorno successivo con il mantenimento della priorità temporale delle proposte originarie.

Conclusione dei contratti

Nelle fasi d'asta, qualora ricorra la condizione di validazione, sono conclusi i contratti al prezzo d'asta che risultano dall'abbinamento automatico delle proposte in acquisto aventi prezzi uguali o superiori al prezzo d'asta con quelle in vendita aventi prezzi uguali o inferiori allo stesso prezzo, secondo le priorità di prezzo e di tempo delle singole proposte e fino ad esaurimento delle quantità disponibili.

Al termine dell'asta di apertura le proposte ineseguite, in tutto o in parte, sono automaticamente trasferite alla negoziazione continua come proposte con limite di prezzo con il prezzo e la priorità temporale della proposta originaria, se si tratta di proposte con limite di prezzo, oppure con il prezzo d'asta e la priorità temporale della proposta originaria, se si tratta di proposte senza limite di prezzo.

Al termine dell'asta di chiusura le proposte ineseguite, in tutto o in parte, qualora sia stata specificata la modalità di esecuzione "valida sino a data specificata", sono automaticamente trasferite alla fase di pre-asta di apertura del giorno successivo con il prezzo e la priorità temporale della proposta originaria, se si tratta di proposte con limite di prezzo, e con il prezzo d'asta e la priorità temporale della proposta originaria, se si tratta di proposte senza limite di prezzo.

Durante la negoziazione continua la conclusione dei contratti avviene, per le quantità disponibili, mediante abbinamento automatico di proposte di segno contrario presenti nel mercato e ordinate secondo i criteri di priorità. L'immissione di una proposta con limite di prezzo in acquisto determina l'abbinamento con una o più proposte di vendita aventi prezzo inferiore o uguale a quello della proposta immessa. Analogamente, l'immissione di una proposta con limite di prezzo in vendita determina l'abbinamento con una o più proposte di acquisto aventi prezzo superiore o uguale a quello della

proposta immessa. L'immissione di una proposta senza limite di prezzo, invece, in acquisto determina l'abbinamento con una o più proposte di vendita aventi prezzo uguale al miglior prezzo di vendita esistente al momento della sua immissione, mentre in vendita determina l'abbinamento con una o più proposte di acquisto aventi prezzo uguale al miglior prezzo di acquisto esistente al momento della sua immissione. L'immissione di proposte senza limite di prezzo può essere effettuata solo in presenza di almeno una proposta di negoziazione di segno contrario con limite di prezzo. In ogni contratto concluso mediante abbinamento automatico il prezzo sarà pari a quello della proposta avente priorità temporale superiore.

L'esecuzione parziale di una proposta al prezzo di mercato dà luogo, per la quantità ineseguita, alla creazione di una proposta che rimane esposta con il prezzo dell'ultimo contratto concluso e la priorità temporale della proposta originaria. L'esecuzione parziale di una proposta con limite di prezzo dà luogo, per la quantità desiderata, alla creazione di una proposta che rimane esposta con il prezzo e la priorità temporale della proposta originaria.

Le proposte di negoziazione ancora in essere, anche parzialmente, al termine della negoziazione continua, sono automaticamente trasferite alla fase di pre-asta dell'asta di chiusura con il prezzo e la priorità temporale presenti al termine della negoziazione continua. Gli operatori possono eseguire contratti, mediante apposita funzione detta "*cross order*", abbinando due proposte di segno contrario e di pari quantità a condizione che le proposte riflettano ordini di terzi e il prezzo di esecuzione sia compreso tra il prezzo della migliore proposta in acquisto e quello della migliore proposta in vendita presenti nel mercato al momento dell'immissione, estremi esclusi.

Il prezzo di riferimento è pari al prezzo di asta di chiusura. Qualora non sia possibile determinare il prezzo dell'asta di chiusura, il prezzo di riferimento è posto pari alla media ponderata dell'ultimo 10% delle quantità negoziate, al netto delle quantità scambiate mediante l'utilizzo della funzione *cross-order*. Se non sono stati conclusi contratti nel corso della seduta, il prezzo di riferimento è pari al prezzo di riferimento del giorno precedente. Borsa Italiana può stabilire nelle Istruzioni che, in relazione a specifici segmenti di negoziazione, il prezzo di riferimento sia determinato anche se esiste un prezzo di asta di chiusura. Inoltre, al fine di garantire la regolarità delle negoziazioni e la significatività dei prezzi, può stabilire, in relazione ad un singolo strumento finanziario, che il prezzo di riferimento sia determinato anche se esiste un prezzo di asta di chiusura, dandone comunicazione al pubblico con Avviso di Borsa.

Il prezzo ufficiale giornaliero di ciascuno strumento finanziario è dato dal prezzo medio ponderato dell'intera quantità dello strumento medesimo

negoziata nel mercato durante la seduta, al netto della quantità scambiata mediante l'utilizzo della funzione *cross-order*.

Il prezzo di controllo della giornata di ciascuno strumento finanziario è dato da:

- il prezzo di riferimento, in asta di apertura;
- il prezzo di asta di apertura, durante la negoziazione continua e, se non è determinato un prezzo di asta di apertura, il prezzo di controllo è pari a quello di riferimento;
- il prezzo di asta di apertura, in asta di chiusura e, se non è determinato un prezzo di asta di apertura, il prezzo di controllo è pari a quello di riferimento.

Se durante la negoziazione continua di uno strumento finanziario il prezzo del contratto in corso di conclusione supera uno dei limiti di variazione dei prezzi, la negoziazione continua dello strumento finanziario è automaticamente sospesa per un intervallo di tempo. Durante la sospensione temporanea della negoziazione non sono consentite l'immissione, la modifica o la cancellazione di proposte.

La durata dell'intervallo di sospensione automatica delle negoziazioni al superamento dei limiti di variazione dei prezzi, stabilita nelle Istruzioni (2003), è fissata in 5 minuti e ha comunque termine al momento di inizio della fase di asta di chiusura. La durata del rinvio alla fase di pre-asta è fissata in 25 minuti e può tuttavia avere una durata diversa, non inferiore a 5 minuti, solo se ciò si rendesse necessario al fine di garantire il regolare svolgimento delle negoziazioni sui singoli strumenti finanziari, e ha comunque termine al momento di inizio della fase di asta di chiusura. La durata dell'intervallo di riattivazione automatica della fase d'asta è pari a 25 minuti e ha comunque termine al momento di inizio della fase di asta di chiusura.

Allo scadere della sospensione temporanea le negoziazioni riprendono con le modalità della negoziazione continua, salvo diverse disposizioni di Borsa Italiana. Qualora sia riattivata una fase d'asta, questa si svolge secondo le modalità previste per l'asta di apertura.

I contratti conclusi nel MTA sono registrati in un apposito archivio elettronico con le seguenti indicazioni: numero progressivo del contratto, orario di inserimento delle proposte, orario di esecuzione, strumento finanziario negoziato, quantità e prezzo unitario, codice identificativo e posizione contrattuale degli operatori acquirenti e venditori, nonché se l'operazione è stata effettuata per conto proprio o per conto terzi. Nello stesso archivio sono registrate le medesime informazioni, per quanto compatibili, relative alle proposte di negoziazione immesse nel mercato.

INDICI AZIONARI

Gli indici di borsa fanno parte, insieme al listino di borsa, di una serie di dati grezzi che ogni giorno sono resi pubblici attraverso i quotidiani e costituiscono l'informativa di base riguardante il mercato a cui tutti possono accedere.

Come sottolineato in Damilano e altri (2002), l'indice è un dato che sintetizza l'andamento del mercato azionario a cui si riferisce. Esso rappresenta il valore di un determinato paniere di titoli, nel quale sono inclusi, con diversi pesi, i titoli più rappresentativi del mercato di riferimento. Le variazioni del valore dell'indice sono riferite ad una determinata data, nella quale è stato fissato il valore di base. Rappresenta un valido strumento di analisi del *trend* di mercato ed è alla base di numerosi strumenti derivati o utilizzato come *benchmark*, per valutare l'andamento della gestione di un fondo o il rendimento di uno strumento finanziario.

Esistono diverse tipologie di indice, che differiscono per le regole utilizzate per la costruzione. Si parla quindi di indici globali, se includono tutte le azioni quotate nel mercato considerato, oppure di indici parziali, qualora se ne considerasse solo una parte. Questi ultimi possono essere ulteriormente suddivisi in indici chiusi o aperti, a seconda che mantengano fissa la loro composizione oppure siano possibili variazioni dovute alla scelta di seguire precisi criteri per includere o escludere i titoli. Sarà quindi possibile che nel tempo titoli inizialmente inclusi nel paniere ne siano poi esclusi, mentre ne siano inclusi altri che in precedenza non possedevano le caratteristiche adeguate.

I titoli inclusi nel paniere hanno un peso all'interno del paniere stesso. Tale peso può essere determinato con diversi criteri, quali il prezzo o la capitalizzazione. Un altro aspetto che può generare diverse tipologie di indice sono le modalità di trattamento dei dividendi. La maggior parte degli indici considera i titoli al corso *tel-quel*, ossia comprensivo dei dividendi in corso di maturazione, ma esistono anche indici che considerano il corso secco presunto dei titoli. Nel primo caso l'indice è influenzato anche dalle distribuzioni di dividendi, che provocano il crollo del valore dell'indice, mentre nel secondo caso, non potendo calcolare il corso secco a priori, l'indice può risentire di valutazioni arbitrarie.

Un'ultima differenza tra gli indici deriva dalla frequenza nella rilevazione dei dati necessari. Alcuni indici sono calcolati istantaneamente, ossia il loro valore segue le fluttuazioni di prezzo dei titoli durante tutta la giornata, mentre altri sono calcolati solo a fine giornata sui prezzi di chiusura delle sedute di borsa.

Nella tabella seguente sono elencate le caratteristiche dei principali indici calcolati e diffusi da Borsa Italiana relativi al Mercato Telematico Azionario (MTA, 2003).

Nome	Tipologia	Base	Composizione
MIBTEL	Continuo	3/1/1994= 10.000	Tutte le azioni italiane sui mercati di Borsa Italiana e tutte le azioni estere incluse negli indici Mib30 e Midex.
MIB 30*	Continuo	31/12/1994= 10.000	30 azioni più capitalizzate e liquide
MIDEX*	Continuo	31/12/1994= 10.000	25 azioni nazionali quotate sull'MTA selezionate sulla base di criteri di liquidità e di capitalizzazione, classificatesi successivamente a quelle che rientrano nel MIB30
MIB	Calcolato con i prezzi ufficiali al termine della seduta di Borsa	2/1/1975= 10.000	Tutte le azioni quotate sui mercati di Borsa Italiana e da tutte le azioni estere incluse negli Indici MIB30 E MIDEX
MIB STAR	Calcolato con i prezzi ufficiali al termine della seduta di Borsa	31/12/2000= 1.000	Azioni nazionali quotate nel segmento STAR
MIB SETTORIALI	Calcolati con i prezzi ufficiali dei rispettivi componenti al termine della seduta di Borsa	31/12/1994= 1.000	24 indici informativi calcolati su panieri che riflettono la suddivisione settoriale del Listino Ufficiale
S&P/MIB			In fase di realizzazione. Le azioni dell'S&P/MIB saranno selezionate sulla base del *Global Industry Classification Standard* (GISC) e ponderate in base al criterio del flottante.

** Utilizzato anche come sottostante di prodotti derivati.*

Borsa Italiana fornisce il valore degli indici nei vari segmenti, calcolando la media aritmetica ponderata degli indici semplici di prezzo dei titoli e utilizzando come pesi di ponderazione i valori di capitalizzazione di ciascun titolo alla data base.

Il valore di capitalizzazione del singolo titolo è dato dalla seguente formula, in cui prezzi e quantità sono indicati rispettivamente con *p* e *q* e tra parentesi vi è il periodo a cui sono riferiti (il periodo *0* è il periodo in cui è stata fissata la base):

$$\omega_i = \frac{p_i(0) \cdot q_i(0)}{\sum_{i=1}^{n} (p_i(0) \cdot q_i(0))}$$

Per "indici semplici di prezzo" si intende il rapporto tra il prezzo del titolo al momento del calcolo dell'indice e il prezzo del titolo alla data base, come indicato nella formula che segue:

$$I_i = \frac{p_i(t)}{p_i(0)}$$

L'indice risulta quindi calcolato tramite la seguente formula:

$$I(t) = \sum_{i=1}^{n} I_i(t) \cdot \omega_i \cdot vb$$

dove *vb* è il valore preso come base.

CARATTERISTICHE DEI PRODOTTI DERIVATI

I prodotti derivati nascono per facilitare la gestione del rischio conseguente alla volatilità dei mercati finanziari. Il prezzo di tali attività, come si intuisce dal nome, "deriva" dal prezzo dell'attività su cui si basano.

E' possibile classificare gli strumenti secondo tre criteri. Il primo criterio di classificazione è la tipologia dell'attività sottostante. In base a questo criterio si possono distinguere le *commodity derivatives*, che sono contratti che hanno per oggetto attività reali come petrolio, oro o altro, e le *financial derivatives*, che sono contratti basati sulle attività finanziarie, a loro volta classificabili in tassi di interesse, valute, azioni o indici azionari.

Il secondo criterio fa riferimento alle caratteristiche tecniche dei diversi strumenti, che risultano sempre essere una combinazione di più strumenti semplici. Nella totalità delle tipologie di strumenti sono individuabili tre grandi categorie:

1. i contratti *swap*, nei quali le controparti assumono l'impegno di scambiarsi reciprocamente delle somme di denaro di ammontare determinato secondo precise regole, a scadenze prefissate;
2. i contratti *future*, con i quali si assume un impegno, per l'acquirente e per il venditore del contratto, di effettuare un'operazione di acquisto o di vendita di una determinata attività ad una data prefissata;
3. i contratti *option*, che, dietro il pagamento di un premio, danno il diritto di acquisto o di vendita di attività ad una determinata scadenza e ad un prezzo prefissato.

Il terzo criterio di classificazione è relativo al mercato in cui sono negoziati. La distinzione principale è tra strumenti negoziati sul mercato regolamentato e strumenti negoziati sui mercati *over-the-counter (OTC)*, ossia fuori borsa. Nel secondo caso non vi è una identificazione fisica precisa del mercato, non vi sono quotazioni ufficiali, è quasi del tutto assente la standardizzazione dei contratti e non vi sono istituzioni che gestiscono e controllano l'operato degli investitori.

Aspetti generali del future sull'indice di borsa

Il *future* sull'indice di borsa è un contratto con il quale due soggetti si impegnano a vendere l'uno e ad acquistare l'altro, un paniere di titoli azionari che presenta conformazione identica al paniere sottostante un indice azionario di riferimento. Il prezzo del paniere è fissato e l'operazione di compravendita è regolata ad una data prestabilita fra le parti. La particolarità che differenzia il *future* sull'indice da altre tipologie di prodotti derivati, consiste nel fatto che le attività su cui si basa il contratto non sono scambiate fisicamente fra le parti, ma è fissato un valore monetario per il singolo punto dell'indice (ad esempio, per il *FIB 30* un punto dell'indice vale 5 Euro) e il regolamento dell'operazione avviene in denaro (*cash delivery*). La scelta di un regolamento puramente monetario fra le parti è giustificata dalla difficoltà che comporterebbe mettere insieme esattamente i titoli in quantità date dal peso di questi nell'indice azionario di riferimento.

I detentori di un *future* sull'indice possono perseguire diverse finalità. Gli investitori in *stock index future* sono classificabili in tre categorie:

a) speculatori, i quali assumono deliberatamente delle posizioni in base alle loro previsioni future sull'andamento dei mercati;

b) *hedger*, investono nell'indice per coprirsi dai rischi derivanti dalla detenzione di un portafoglio di titoli;

c) arbitraggisti, si inseriscono nelle anomalie dei prezzi di mercato delle attività negoziate, in totale assenza di rischio operativo.

Diversi sono i fattori che hanno favorito il successo di questo strumento tra gli investitori. I principali sono la spiccata volatilità delle attività sottostanti, la notevole flessibilità dello strumento, che risulta adattabile a numerose esigenze, i minori costi in termini di liquidità, collegati alla scelta di investire in prodotti derivati piuttosto che direttamente nelle attività sottostanti, e il ruolo che questi ricoprono in una gestione efficiente di portafoglio.

CAPITOLO 5

ARTICOLAZIONE DI UN MODELLO DI SIMULAZIONE DI BORSA

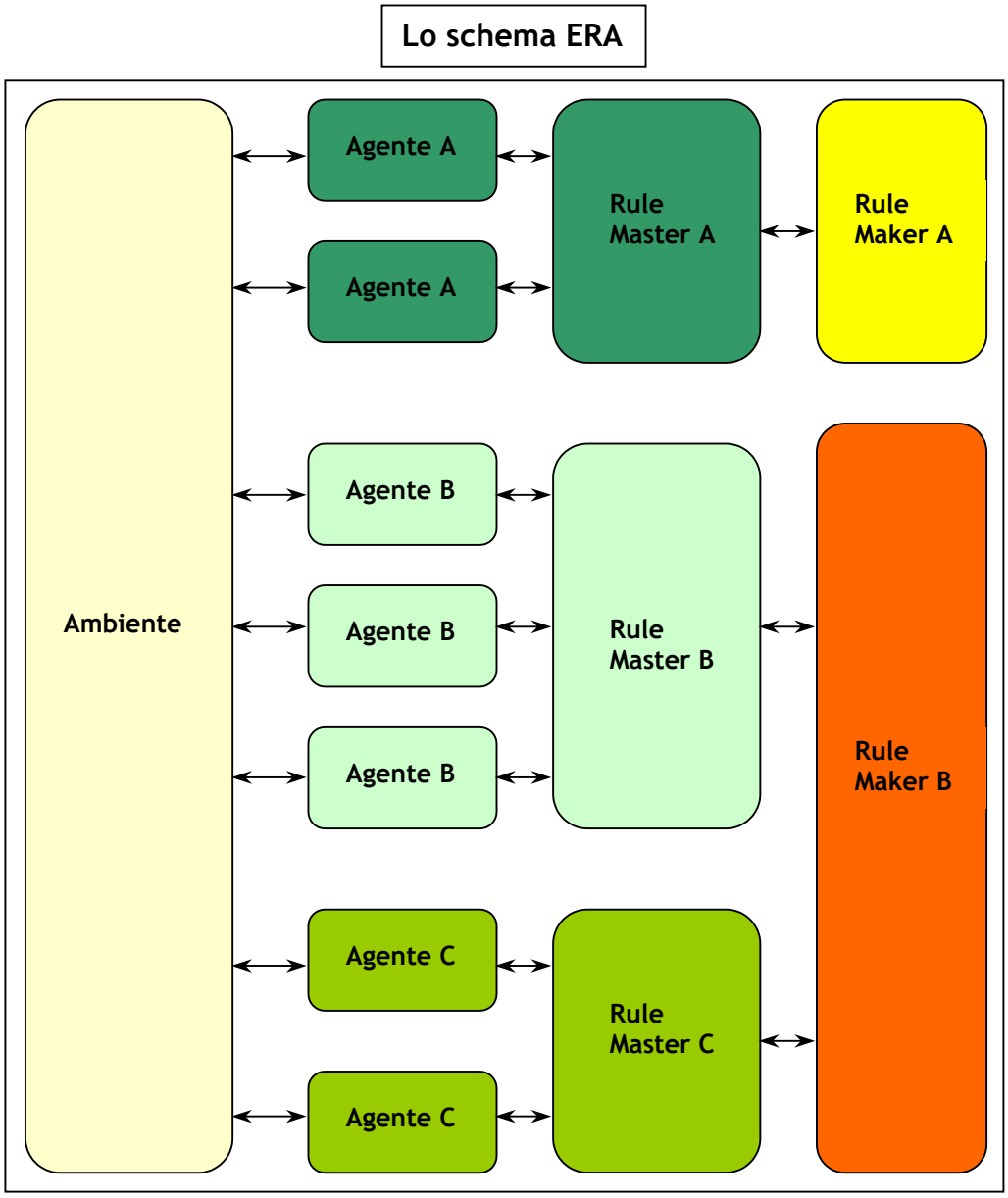
Si introduce ora lo schema generale seguito nella costruzione del modello *SUM*, che è ripreso per la ricostruzione del modello *JavaSum*, e si effettua una rapida descrizione delle diverse componenti dell'ultima versione del modello di simulazione in *Objective C*.

Infine si presenta una semplice simulazione effettuata con il modello *SUM-0.66* e lo si confronta con alcuni modelli di simulazione di borsa che presentano un'impostazione molto diversa.

LO SCHEMA ERA

Per la costruzione di modelli di simulazione ad agenti , è utile seguire uno schema standardizzato in modo da riuscire a gestire progetti di qualsiasi dimensione e livello di complessità. Lo schema *ERA* (*Environments Rules Agents*), introdotto in Gilbert e Terna (2000), permette di suddividere la simulazione in quattro livelli. Il primo livello è l'ambiente in cui si svolge la simulazione. Vi sono poi gli agenti che operano nell'ambiente, i quali possono appartenere a diverse tipologie. Gli ultimi due livelli sono le regole che gli agenti seguono per determinare le proprie azioni, con un gruppo di regole differente per ogni tipologia di agente presente nel modello, e il costruttore di regole, attraverso cui è possibile intervenire per effettuare delle modifiche alle regole seguite dagli agenti.

Questo schema si adatta perfettamente alla logica di programmazione ad oggetti e può essere rappresentato come segue:



In un progetto di questo tipo il comportamento dei diversi agenti è determinato da un oggetto esterno, detto *Rule Master*, che contiene le capacità cognitive dell'agente e le regole di rielaborazione delle informazioni che utilizza per prendere delle decisioni. I vari *Rule Master* sono collegati a loro volta ai *Rule Maker*, attraverso cui sono generate nuove regole guida del comportamento degli agenti, in base a determinate procedure di evoluzione.

Anche se questa struttura a prima vista sembra complicare la costruzione dei modelli di simulazione, rende però molto semplice modificare le parti del modello esistenti, oppure svilupparne altre nuove.

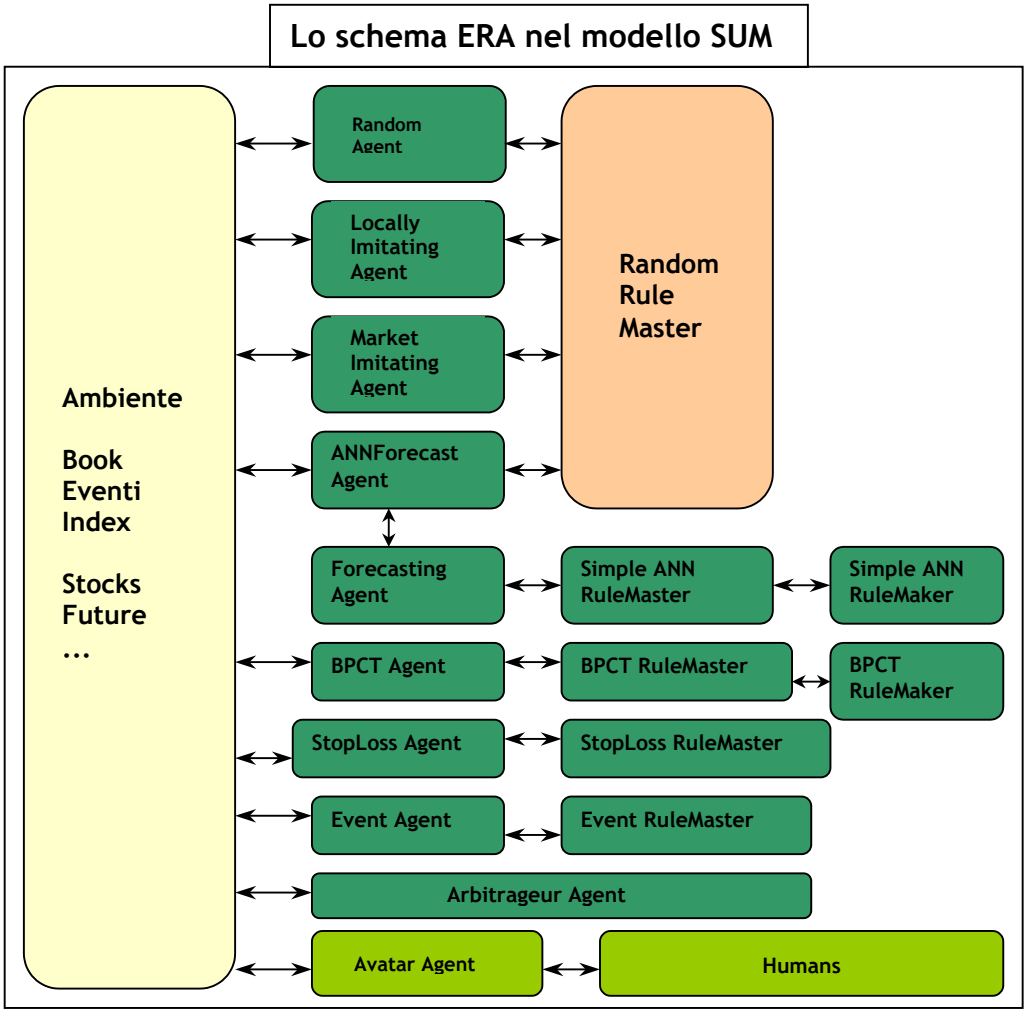
Inoltre la modularità del codice così strutturato rende agevole la costruzione di modelli molto ampi con la collaborazione e lo sviluppo parallelo da parte di più ricercatori.

In Terna (2000) si aggiunge che questa struttura può essere arricchita da oggetti che gestiscono l'osservazione della simulazione, come avviene nel protocollo *Swarm* con l'oggetto *Observer*, o che permettono ai singoli agenti di immagazzinare dei dati (*DataWareHouse*).

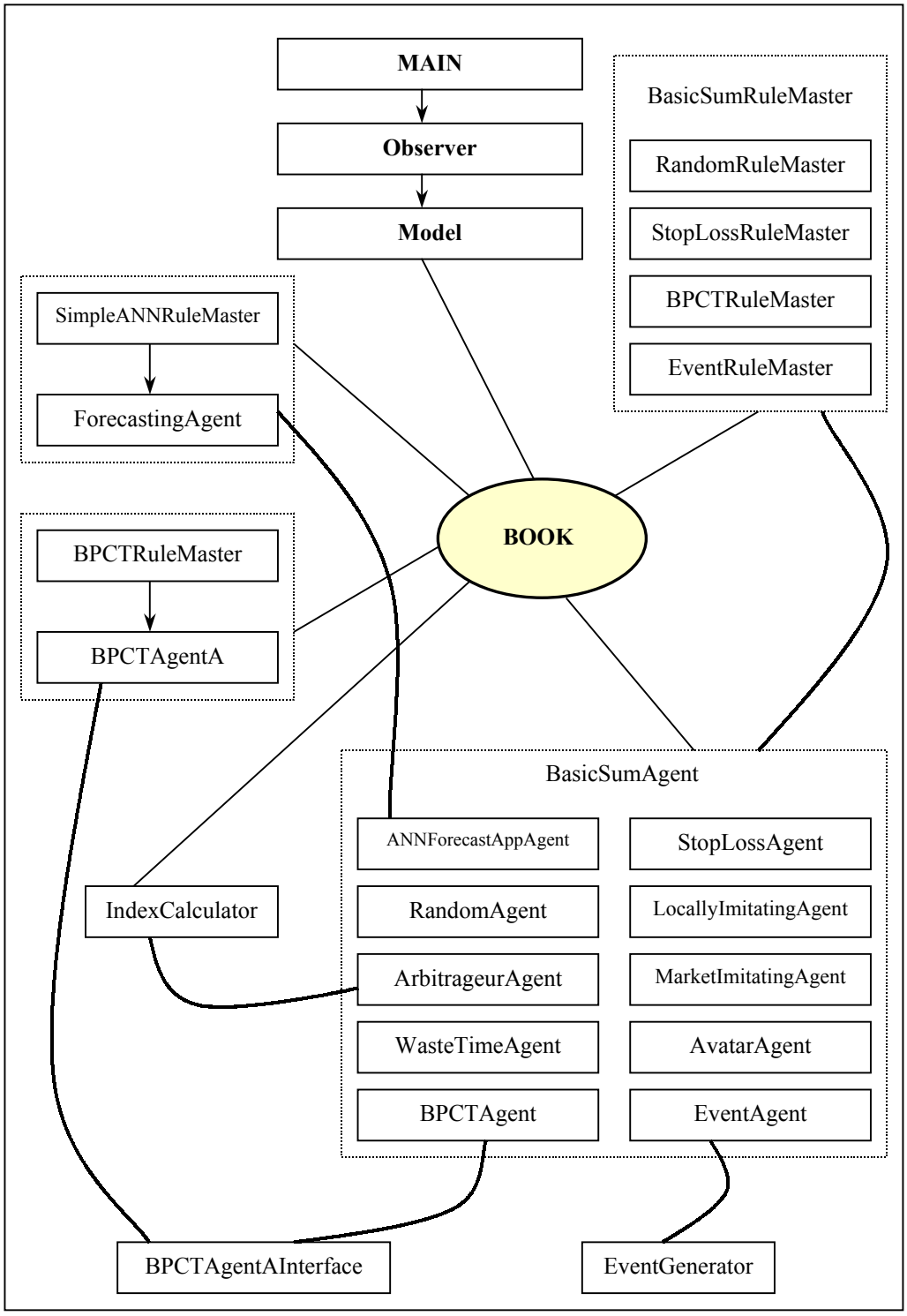
IL MODELLO SUM

Lo schema *ERA* appena descritto è utilizzato nel modello di simulazione di borsa *SUM* ed è mantenuto nella costruzione di *JavaSum*. In questo caso l'ambiente è rappresentato dal *book*, in cui gli agenti inseriscono gli ordini e a cui richiedono le informazioni necessarie per formularli.

La struttura completa, con la suddivisione in quattro livelli successivi, quindi con il ricorso al *Rule Maker*, si ha per due sole tipologie di agenti, che sono il *ForecastingAgent* e il *BPCTAgent*. La struttura generale, come descritto in Cappellini (2003), è la seguente:



Lo schema fondamentale del funzionamento del modello pone al centro il *book*, al quale si collegano il *Model* e l'*Observer*, i *RuleMaster* e le diverse famiglie di agenti. Parallelamente sono presenti degli oggetti, come *l'IndexCalculator* e l'*EventGenerator*, che forniscono alcuni dati necessari per le decisioni degli agenti. Lo schema grafico seguente riassume i diversi collegamenti tra gli oggetti della simulazione.



Nei paragrafi che seguono si effettua una panoramica del modello *SUM*, con la descrizione del *book*, delle diverse categorie di agenti e degli oggetti della simulazione.

Il book e il mercato simulato

Il *book* del modello *SUM* riproduce fedelmente le procedure di raccolta e di combinazione degli ordini del mercato reale (si veda in proposito il capitolo 4, in cui è descritto dettagliatamente il regolamento dell'MTA). Le modifiche più consistenti sono state apportate da Cappellini (2003) e Mezzera (2003).

Sul mercato sono negoziati più titoli e sono presenti le fasi di apertura e chiusura. Con l'introduzione di queste novità, che contribuiscono ad aumentare il realismo del modello, si nota la riduzione di fenomeni come bolle e *crash*, che nel mercato ad un solo titolo erano molto frequenti.

La presenza di più titoli ha permesso di costruire un indice, con il relativo contratto *future*, che è liberamente negoziato come un titolo qualsiasi e il cui prezzo rimane allineato con il mercato grazie all'intervento di un agente arbitraggista, che interviene sul mercato ogni volta che vi è divergenza tra il prezzo del *future* e il valore dell'indice. L'indice è calcolato come media dei prezzi dei diversi titoli negoziati sul mercato. Questa scelta è resa necessaria dalla mancanza del valore di capitalizzazione dei titoli.

Sul mercato simulato possono intervenire anche agenti umani. Attraverso la costruzione di un'interfaccia *web*, un agente umano collegato in rete può inserire ordini sul mercato simulato, che in seguito sono combinati con gli ordini compatibili di altri agenti artificiali o umani. Tecnicamente questo avviene attraverso il cosiddetto *AvatarAgent*, il quale opera sul mercato guidato dalle istruzioni che un agente umano trasmette tramite *web*.

Sul mercato simulato sono diffuse periodicamente delle notizie, positive o negative, che simulano il succedersi di eventi che influenzano il mercato. Le notizie sono percepite da una specifica categoria di agenti, detti *EventAgent*, che prendono le decisioni di acquisto o di vendita a seconda che sia diffusa una notiza buona o una notizia cattiva. Inoltre le notizie possono influenzare eventuali decisioni degli operatori che agiscono tramite gli *AvatarAgent*. A seconda della composizione del gruppo di agenti che opera nel mercato, si osservano diversi gradi di influenza della presenza di agenti sensibili alle notizie sugli andamenti dei prezzi.

Le diverse classi di agenti

Le principali categorie di agenti, oltre agli *AvatarAgent*, che permettono agli umani di interagire sul mercato, e agli *EventAgent*, che decidono di acquistare o vendere titoli sul mercato in funzione delle notizie che sono diffuse, dei quali si è già accennato nel precedente paragrafo, sono brevemente descritte come segue:

- i *RandomAgent* sono gli agenti più semplici, i quali decidono casualmente, in base ad una serie di parametri, quantità, prezzi e tipo di ordine da inserire sul mercato (si veda il capitolo 6 per una descrizione più approfondita);
- i *ForecastingAgent* sono agenti che utilizzano le reti neurali artificiali per la previsione dell'andamento futuro dei prezzi, e svolgono la funzione di valutazione dei titoli, in analogia con le società di *rating* reali;
- gli *ANNForecastingAppAgent* sono agenti che inseriscono ordini di acquisto o di vendita di un titolo, scegliendo un *book* a caso, seguendo le istruzioni fornite dal *ForecastingAgent*;
- gli agenti imitatori sono i *MarketImitatingAgent*, i quali inseriscono ordini di acquisto se il prezzo medio di mercato è salito e ordini di vendita se il prezzo medio è sceso, e i *LocallimitatingAgent*, i quali inseriscono gli ordini tenendo conto del tipo di quelli che sono stati inseriti per ultimi nel *book*;
- gli *StopLossAgent*, i quali operano tenendo conto degli ultimi valori di prezzo passati, contribuendo così a mantenere la direzione delle variazioni nei prezzi;
- gli agenti cognitivi, denominati *BPCTAgent*, presenti in due tipologie, tipo *A* e tipo *B*, i quali operano seguendo la metodologia dei *Cross Target* per determinare le loro azioni, mantenendo una certa coerenza nelle decisioni di inserimento degli ordini.

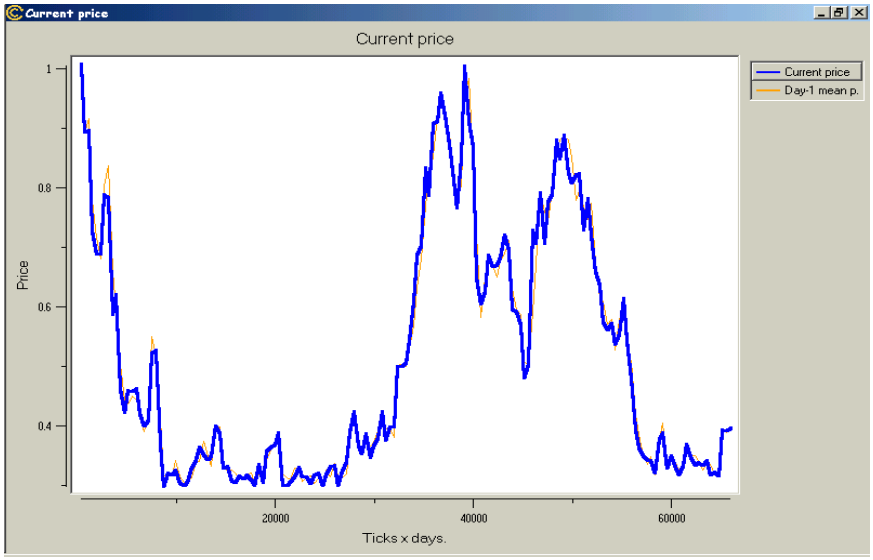
Un esempio pratico di simulazione con SUM

Per questo esperimento si utilizza la versione del modello *SUM-0.66*, nella quale non sono ancora presenti le aste di apertura e chiusura ed è negoziato un solo titolo. Nella simulazione sono fatti operare 400 agenti casuali (*RandomAgent*), in modo da sperimentare una situazione che è riproducibile nella prima versione del modello *JavaSum*.

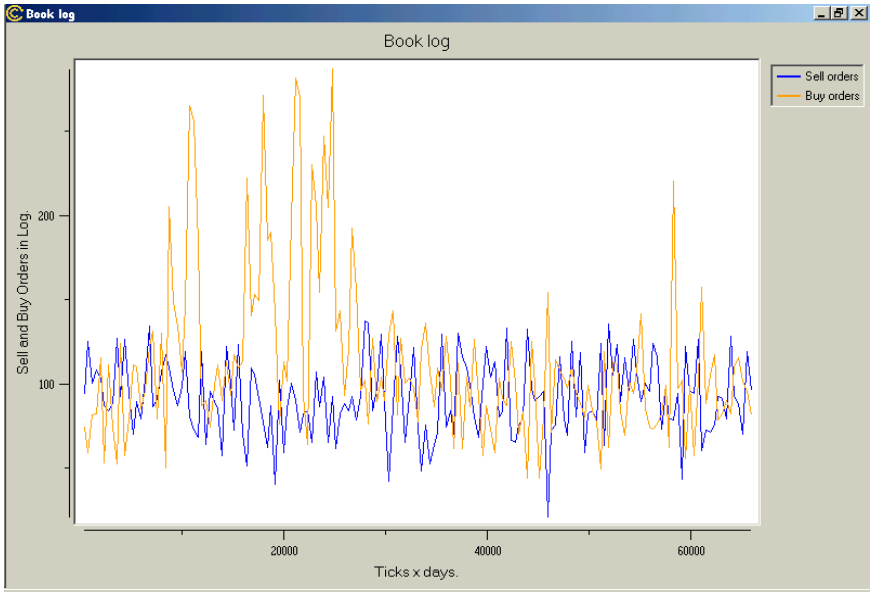
Date le caratteristiche dei *RandomAgent*, gli agenti operano nel mercato conoscendo solamente l'ultimo prezzo e decidono casualmente se inserire o no ordini nel mercato. Prezzi e quantità dell'ordine di acquisto o di

vendita eventualmente emessi sono casuali: il prezzo è dato da quello dell'ultimo contratto concluso sul mercato, moltiplicato per un coefficiente casuale, mentre la quantità è un numero intero casuale compreso tra zero e un valore massimo (*maxOrderQuantity*).

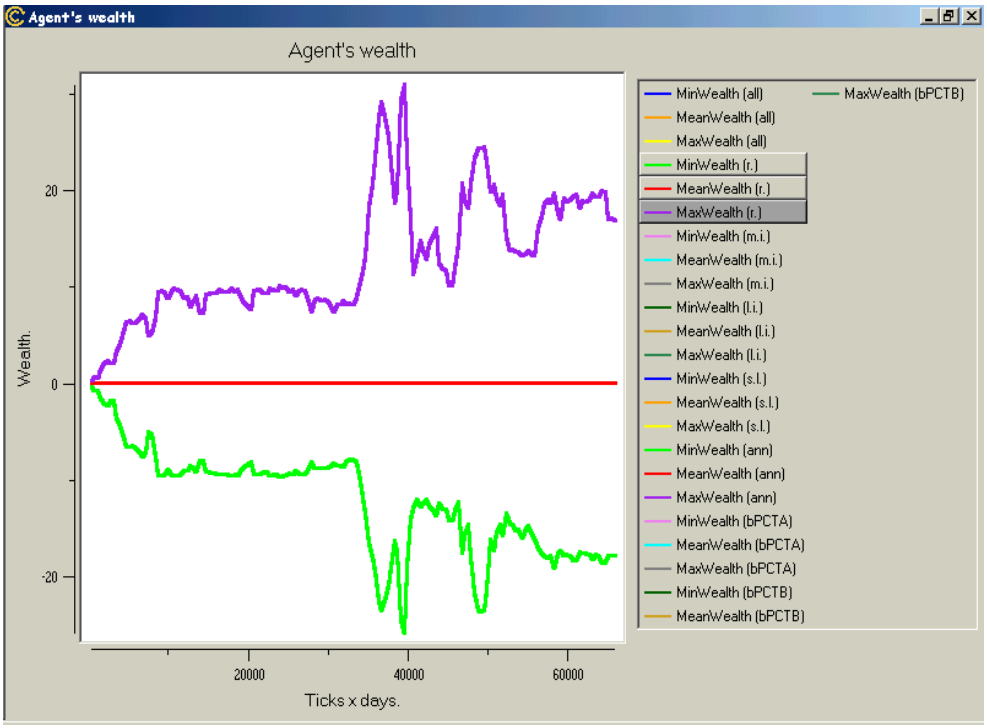
Dai grafici che seguono, costruiti dal modello durante la simulazione, si nota che nel tempo i prezzi presentano un andamento molto volatile con il verificarsi frequente di bolle e *crash*. Questo caso riproduce l'esperimento descritto in Terna (2000d), nel quale si fanno operare agenti "senza mente" in un mercato strutturato. La principale conclusione che si può trarre dall'esperimento è che già dall'interazione di agenti semplici emergono i fenomeni che normalmente si riscontrano sul mercato reale.



Nel grafico seguente sono rappresentati gli andamenti delle quantità degli ordini di acquisto e di vendita, e si può notare come alle frequenti bolle e *crash* corrispondano dei picchi nelle code degli ordini di acquisto o di vendita.



Il modello inoltre fornisce un grafico che visualizza le informazioni relative alla ricchezza degli agenti. Nel grafico qui riportato sono evidenziate le serie di valori relative agli agenti casuali, ossia la serie di valori minimi, massimi e medi della ricchezza dell'insieme di agenti operanti.



Questo semplice esempio dell'utilizzo di *SUM* dimostra che è possibile riprodurre artificialmente i fenomeni che si osservano sul mercato reale. Facendo operare tutte le tipologie di agenti si possono effettuare analisi molto più interessanti. Cappellini(2003), ad esempio, utilizzando la versione più evoluta del modello in cui operano anche agenti umani, riesce a catalogare i diversi comportamenti e le diverse strategie utilizzate dagli umani. In questo modo si utilizza la simulazione come vero laboratorio sperimentale ed è possibile trarne spunto per ricreare agenti artificiali che riproducano i comportamenti umani osservati durante la simulazione. In Mezzera (2003) sono introdotte le aste di apertura e chiusura. Questo permette di aumentare il realismo della struttura del *book* di negoziazione e di ridurre le variazioni dei prezzi nelle fasi iniziali e finali della giornata, le quali potrebbero essere causate da operazioni degli agenti che hanno l'obiettivo di falsare il corretto andamento dei titoli. Nel modello sono inseriti, oltre alle aste, controlli sulle immissioni delle proposte da parte degli agenti e sulle variazioni di prezzo. Nonostante queste misure, sia nel mercato reale che in quello simulato, permane la frequente formazione di bolle e *crash*.

ARTIFICIAL STOCK MARKET (ASM)

Molti sistemi sociali sono caratterizzati da interazioni complesse tra individui. LeBaron (2002) sostiene che le tradizionali metodologie di studio dell'economia hanno avuto scarso successo, in particolare nella finanza. In questo campo sono stati effettuati numerosi studi empirici per tentare di dare supporto alle teorie più disparate. Una delle direzioni dei ricercatori è quella dell'utilizzo della simulazione ad agenti, che ha come obiettivo principale la comprensione delle dinamiche dei fenomeni che le teorie tradizionali dell'economia tentano di spiegare.

Ancora LeBaron (2002) afferma che, per una comprensione più profonda dei mercati, occorre costruire modelli attorno ad una determinata condizione di equilibrio, anche se questa non è una condizione necessaria quando si utilizzano modelli di simulazione ad agenti. Il modello di simulazione *Artificial Stock Market* (*ASM*) del *Santa Fe Institute* segue questa filosofia di fondo.

Esistono diverse simulazioni precedenti ad *ASM*, che tentano di studiare l'impatto di comportamenti casuali di agenti su diverse strutture di mercato, come ad esempio il modello proposto da Cohen e altri (1983), e che influenzano lo sviluppo dello stesso *ASM*. Numerosi sono anche i modelli che si sviluppano contemporaneamente ad *ASM*. Alcuni esempi sono Lux (1997), Kirman (1991), Chiarella (1992), Rieck (1994), Levi, Levi&Salomon (1994) e Beltratti e Margarita (1992).

La nascita di ASM

L'idea di *ASM* si sviluppa tra la fine degli anni '80 e l'inizio degli anni '90 da Brian Arthur e John Holland. Il mercato simulato nasce come insieme di strategie concorrenti tra loro in continua evoluzione, in modo da lasciare che il sistema si evolva escludendo le strategie peggiori. Il progetto si basa su complessi algoritmi di apprendimento che sono gli algoritmi genetici e i *classifier system*, ideati da John Holland.

In seguito il gruppo di ricercatori si allarga, comprendendo anche il fisico Richard Palmer e l'informatico Paul Tayler. Da questo primo gruppo di studio nel 1994 è pubblicata la prima versione del modello , che consiste in un semplice modello di mercato in cui sono registrate le decisioni di acquisto e di vendita. Le operazioni di compravendita avvengono nel seguente modo: un valore di prezzo è annunciato da un *market maker* a tutti gli operatori; gli agenti, in base a proprie regole, emettono ordini di acquisto o di vendita per quantità unitarie di titoli e il prezzo si modifica in base all'eccesso di domanda o offerta di titoli, calcolate in base al numero di ordini di acquisto e di vendita presenti sul mercato.

Le modificazioni di prezzo avvengono al fine di mantenere l'equilibrio di mercato. Per esempio, se ci sono 100 ordini di acquisto e 50 di vendita si ha una situazione di eccesso di domanda di titoli e il prezzo è aumentato per ristabilire l'equilibrio. L'aggiustamento di prezzo dipende direttamente dalla differenza tra ordini di acquisto e ordini di vendita. La relazione tra il prezzo in un periodo (p_{t+1}) e il prezzo nel periodo precedente (p_t) è la seguente:

$$p_{t+1} = p_t + \lambda(B_t - S_t)$$

Come si può notare, la variazione di prezzo dipende da $\lambda(B_t - S_t)$ dove B_t è la quantità di ordini di acquisto, S_t è la quantità di ordini di vendita e λ è un coefficiente che stabilisce la velocità con cui il prezzo si modifica nel tempo per ristabilire l'equilibrio.

I problemi derivanti dall'assunzione di questo meccanismo, evidenziati da LeBaron (2001), sono principalmente due. Il primo consiste nella grande sensibilità del prezzo a λ. Per valori di λ molto bassi, infatti, si protraggono nel tempo situazioni di eccesso di domanda o eccesso di offerta di titoli, mentre per valori di λ molto elevati il mercato tende a passare velocemente da situazioni di eccesso di domanda a situazioni di eccesso di offerta.

Il secondo problema è che le regole degli agenti non seguono strategie precise di acquisto o di vendita; ciò porta gli agenti ad abbandonare il mercato prima della completa soddisfazione degli ordini immessi.

Nel 1993 LeBaron effettua alcune modifiche al modello per lo studio delle dinamiche di prezzo e dei volumi delle contrattazioni. Nei prossimi paragrafi sono analizzate le caratteristiche più rilevanti del modello.

Swarm e la costruzione di ASM

Secondo Johnson (2001), *ASM* del *Santa Fe Institute* è il più famoso esempio di modello di simulazione di un mercato finanziario ad agenti. Con i modelli di simulazione ad agenti è possibile elaborare teorie che poggiano su ipotesi meno restrittive dal punto di vista della razionalità degli agenti. Per la costruzione dei modelli solitamente si utilizzano le logiche della programmazione ad oggetti per favorirne la leggibilità e lo sviluppo.

Il primo modello *ASM*, costruito con l'utilizzo delle librerie di *Swarm*, è scritto in linguaggio *Objective C*, per l'utilizzo sul sistema operativo *NEXT*. Da questa prima versione ne deriva direttamente una seconda, denominata *SFSM*, con alcune differenze grafiche dovute all'eliminazione delle librerie legate al sistema *NEXT*.

Un primo sguardo al codice di *SFSM* rivela che il modello risulta in gran parte scritto in linguaggio *C standard*, con l'unica differenza che il modello è articolato nelle diverse classi da cui si creano, durante la simulazione, gli

esemplari necessari. Vi sono classi che descrivono il comportamento degli investitori, che regolano la distribuzione dei dividendi, che descrivono l'operato del banditore (detto *specialist*) addetto all'amministrazione del mercato; vi è la classe dell'ambiente, i cui metodi sono necessari per la raccolta dei dati della simulazione e che restituiscono gli stessi su richiesta degli agenti. Inoltre esistono numerose altre classi che regolano le interazioni tra le classi appena elencate.

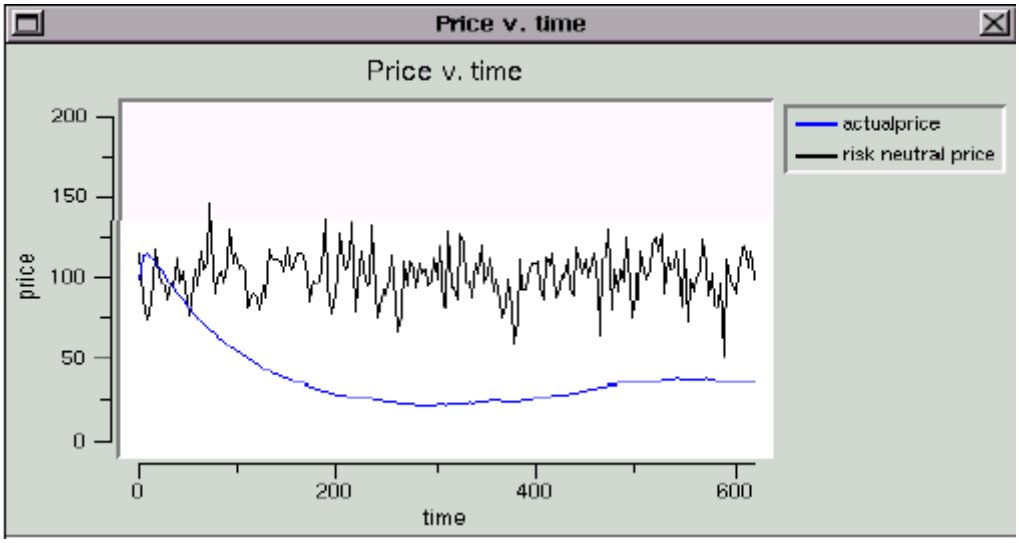
Partendo dal modello *SFSM*, numerosi membri del gruppo di costruttori, insieme ad alcuni studiosi del *Santa Fe Institute*, decidono di adattare il modello alle librerie di *Swarm*, spinti dai vantaggi insiti nell'utilizzo del protocollo. Il primo vantaggio è che *Swarm* possiede una serie di *routine* collaudate per la generazione di numeri casuali e per l'amministrazione degli eventi della simulazione (come ad esempio lo *schedule*). Il secondo vantaggio consiste nella possibilità di utilizzare strumenti di visualizzazione grafica dei dati che erano andati persi con l'abbandono del sistema operativo *NEXT*.

Tra il 1998 e il 1999 nasce la prima versione *Swarm* di *ASM*, che va ad aggiungersi alla serie di modelli di simulazione sviluppati in questo ambiente, con la differenza di essere uno dei primi modelli di simulazione che ha per oggetto lo studio di un fenomeno di competenza delle scienze sociali, notevolmente diverso dalle numerose simulazioni sui "*bug*".

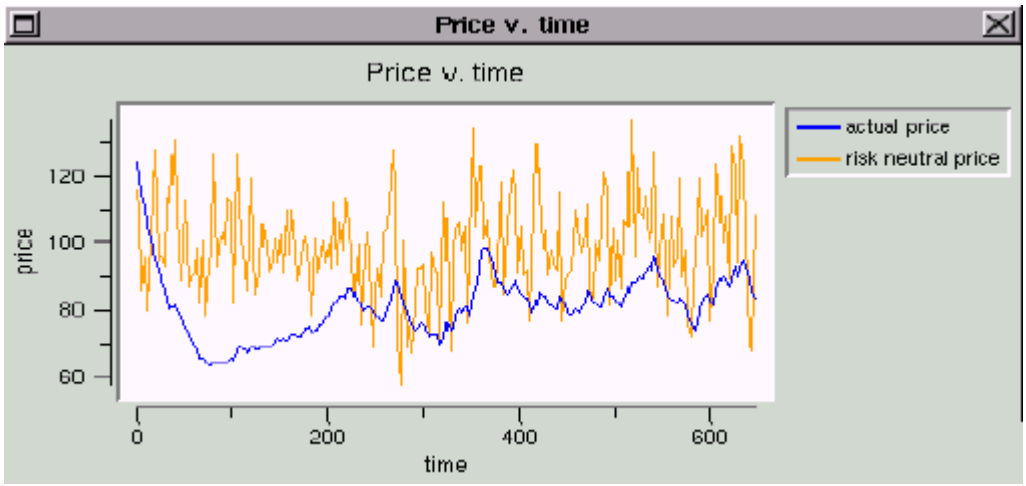
Nell'aprile 2000 il *Santa Fe Institute* annuncia l'uscita della versione *ASM*-2.0, la quale include tre elementi:

- la copia della versione *Objective C* di *SFSM*;
- la copia della versione *Swarm* di *ASM*-2.0;
- un documento scritto dai principali revisori del codice.

Questa prima versione contiene alcuni errori che influiscono notevolmente sulla dinamica del mercato simulato. In particolare, il prezzo di mercato scende allontanandosi dal prezzo delle attività prive di rischio quando invece sarebbe ragionevole un andamento che rispecchiasse strettamente quello delle attività prive di rischio (si veda la figura seguente).



Per porre rimedio agli errori del modello, Johnson rivisita il codice e, dopo quasi un anno, si arriva alla versione *ASM*-2.2, che non presenta più il problema, come si può notare dalla figura sottostante.



La nuova versione comprende diverse nuove classi con numerose semplificazioni nello *schedule* e la sostituzione del codice relativo al comportamento degli agenti revisori (*forecasting agent*). Scompaiono i problemi nell'andamento del prezzo di mercato che risultano ora più simili agli andamenti dei prezzi delle attività prive di rischio.

Il passaggio alla nuova versione è importante soprattutto perchè costituisce un esempio di scrittura di un modello di simulazione ad agenti con l'utilizzo di *Swarm*.

Alcune differenze tecniche tra la versione ASM-2.0 e ASM-2.2

Nel codice di *ASM*-2.0 le compravendite di titoli sono effettuate dagli esemplari della classe *Bfagent*, dove le iniziali *BF* stanno per *bitstring forecasting*.

Nella versione *ASM*-2.2 questa struttura è sostituita da diverse classi: la classe *BFCast*, i cui esemplari formulano previsioni sui prezzi dei titoli, la classe *BitVector*, creata per la registrazione delle informazioni e la classe *BFParams*, contenente i parametri, sostituiscono le classi *BF_fcast* e *Bfparams* della versione 2.0. Inoltre sono introdotte numerose semplificazioni al codice, con la definizione di metodi che permettono di richiamare procedure di calcolo più volte all'interno di diverse parti del codice, e si limita l'utilizzo delle variabili dichiarate "*static*" (si veda il capitolo 6 per la descrizione tecnica), limitando così i possibili errori derivanti da questo tipo di dichiarazione.

In questa versione aumenta molto la flessibilità di utilizzo della simulazione. Con l'utilizzo delle liste e degli indici delle liste di agenti di *Swarm* si ha la possibilità di far operare gli agenti in numero diverso di volta in volta. I parametri del modello sono gestiti in *file* separati (con estensione "*scm*") e sono generati dei *file* di dati, che in seguito possono essere rielaborati, o convertiti in altro formato, con l'utilizzo del programma statistico *R* (http://www.R-project.org).

LA STRUTTURA DI ASM

La struttura interna del modello è descritta da LeBaron (2002). Nel mercato sono negoziate due tipologie di titoli:

- un titolo senza rischio (*risk-free*), che presenta un tasso di rendimento fisso;
- un titolo rischioso, il quale restituisce un rendimento sotto forma di dividendi.

I dividendi del titolo rischioso sono determinati casualmente nel modello e derivano dal seguente processo autoregressivo:

$$d_t = \overline{d} + \rho(d_{t-1} - \overline{d}) + \mu_t$$

Inoltre si assume che:

a) $\overline{d} = 10$;

b) $\rho = 0.95$;

c) $\mu_t \sim N(0,\sigma_{\mu}^{2})$.

Gli investitori allocano l'intera ricchezza tra le due tipologie di titoli (attività *risk-free* e attività rischiose) e, in ogni periodo, possono negoziare fino a dieci titoli. Se sono effettuate vendite allo scoperto il limite si abbassa a cinque titoli. Il modello risulta di derivazione neoclassica, infatti si basa sulla teoria delle aspettative razionali e si assume la normalità della distribuzione dei rendimenti.

Il prezzo dei titoli è determinato in modo endogeno nel modello, senza che sia utilizzato alcun dato storico, con la conseguenza che non risulta alcun collegamento tra mercato simulato e mercato attuale.

Le principali critiche al modello sono la presenza di un solo titolo rischioso negoziato sul mercato e l'assenza di restrizioni all'indebitamento degli agenti. Aggiungere ulteriori titoli alle contrattazioni virtuali da un lato risulterebbe molto utile per lo studio del volume delle contrattazioni, dall'altro però aumenterebbe molto la complessità del modello. Secondo LeBaron (2002) rimangono da sviluppare le modalità di distribuzione dei dividendi e la loro frequenza nel tempo. I dividendi giocano un ruolo fondamentale nei processi di informazione del mercato.

Nelle ultime versioni del modello si utilizza una avversione al rischio costante. Come specificato in Arthur e altri (1997), gli individui ottimizzano l'allocazione tra titoli rischiosi e privi di rischio, mentre la loro domanda, data l'ipotesi di normalità della distribuzione dei rendimenti, con media $E_{i,t}[p_{t+1} + d_{t+1}]$ e varianza $\sigma_{i,p+d}^{2}$, è definita come segue:

$$x_t^i = \frac{\hat{E}_t^i(p_{t+1} + d_{t+1}) - (1 + r_f)p_t}{\gamma\hat{\sigma}_{p+d,i}^{2}} \qquad (5.1)$$

Nella formula con r_f è indicato il tasso di interesse del titolo senza rischio e con γ l'avversione al rischio dell'investitore, che è costante per assunzione.

La somma delle quantità domandate dai diversi agenti è pari al totale dei titoli emessi sul mercato:

$$\sum_{i=1}^{N} x_{i,t} = N$$

Da queste relazioni deriva il valore corrente del prezzo di mercato del titolo negoziato.

Se, ad esempio, si considera il generico periodo simulato *t*, all'inizio del periodo gli agenti conoscono il valore del dividendo d_t e hanno a disposizione le informazioni generali riguardanti lo stato del mercato, le quali includono le serie storiche dei dividendi e dei prezzi, con cui possono

formare le proprie aspettative sugli andamenti, di prezzi e dividendi, nel periodo seguente. I parametri della funzione di domanda sono passati allo *specialist*, che ne deriva il prezzo di mercato. Nel periodo seguente questa procedura è ripetuta partendo da un valore del dividendo pari a d_{t+1}.

Gli agenti usano i *classifier system* per fare previsioni sul rendimento dei titoli, e hanno a disposizione una serie di modelli lineari che utilizzano per fare previsioni a seconda delle diverse condizioni di mercato. Le previsioni sono assunte come lineari in funzione del prezzo corrente e del dividendo:

$$\hat{E}_t^i(p_{t+1} + d_{t+1}) = a_j(p_t + d_t) + b_j$$

Il suffisso *j* si riferisce alla regola scelta dall'agente *i*.

Le regole permettono agli agenti di formulare previsioni e di stimarne la varianza, e si modificano attraverso l'apprendimento nel tempo. Ciò è possibile grazie ad un algoritmo genetico; la velocità di apprendimento è impostata attraverso un parametro.

La funzione di domanda è quindi lineare in p_t. Impostando il numero totale di titoli ad un valore fisso, è possibile risolvere l'equazione 5.1 e giungere al valore di prezzo che permette l'equilibrio temporaneo dei mercati. Una volta fissato il prezzo, gli agenti aggiornano il loro portafoglio e il volume dei titoli negoziati è registrato.

Il sistema di combinazione degli ordini è molto semplice e si discosta notevolmente da ciò che accade sul mercato reale. Le principali critiche al modello, riassunte da LeBaron (2002), sono le seguenti:

- il prezzo previsto è considerato lineare in p_t, e non è chiaro quali siano le conseguenze sulle dinamiche del mercato simulato;
- la funzione di domanda di titoli non include la ricchezza degli investitori, in questo modo non si considera la maggiore influenza sul prezzo di un titolo che possono avere gli investitori più facoltosi. Un possibile rimedio a questo problema potrebbe essere l'utilizzo di un diverso livello di avversione al rischio per investitori con diverse ricchezze;
- il mercato reale, a differenza di quello simulato, non è mai realmente in equilibrio.

Quest'ultima critica è la più importante in quanto i mercati finanziari sono istituzioni designate per la combinazione in tempo reale degli ordini di acquisto e di vendita, ma che possono essere molto lontani dall'ideale walrasiano della compensazione. Il modello è completamente privo della microstruttura del mercato e in esso si ipotizza che domanda e offerta siano

sempre e continuamente bilanciate. Ciò può essere plausibile in una prospettiva di lungo periodo, anche se la considerazione di modelli più realistici delle istituzioni di mercato costituirebbe un'importante estensione del modello di simulazione.

Come sottolineato in Terna (2003), LeBaron (2002) si riferisce a Daniels e altri (2002), in cui gli autori mostrano l'importanza della microstruttura del mercato in un modello di simulazione. Nel modello analizzato operano agenti privi di intelligenza e capaci solamente di decisioni casuali. Possono inserire ordini al "prezzo di mercato", ossia al miglior prezzo presente sul mercato al momento dell'immissione dell'ordine (la cosiddetta soddisfazione dell'ordine "al meglio"), e inserire ordini con un prezzo limite nel cosiddetto "*limit order book*" del mercato. In questo caso gli ordini di acquisto sono denominati come "*bid*", mentre gli ordini di vendita sono denominati come "*ask*". La presenza di questo *book* è sufficiente a generare fenomeni osservabili sul mercato reale, nonostante non sia presente sul mercato alcuna forma di intelligenza. Ciò si ricollega ai risultati dello studio di Terna (2001), in cui agenti casuali, ossia "senza mente", operanti in un mercato strutturato, nel quale il *book* assolve alla funzione di raccoglitore e combinatore di ordini, portano al verificarsi di fenomeni come bolle e *crash*.

Alcuni risultati empirici, ricavati dal modello *ASM*, sono presentati in LeBaron e altri (1999), in cui si mostra come il modello generi, in sintonia con numerosi altri modelli di simulazione di mercato ad agenti, serie di dati caratterizzati da bassissima autocorrelazione lineare e persistente volatilità. Il volume delle contrattazioni risulta strettamente correlato alla volatilità dei prezzi, come avviene nel mercato reale. In particolare, il modello si avvicina al mercato reale quando si utilizzano agenti con elevato livello di apprendimento e quando si aggiorna frequentemente l'algoritmo genetico.

ASM E SUM: LA DIVERSA IMPOSTAZIONE DEL MODELLO

Come sottolineato in Terna (2003), i modelli di simulazione ad agenti offrono possibilità di studio superiori ai modelli classici, poiché eliminano la necessità di assumere ipotesi troppo restrittive e di dover considerare un'unica tipologia di agente con piena razionalità e comportamento ottimizzante, molto distante dagli agenti che effettivamente operano sul mercato reale. Infatti, nei modelli di simulazione ad agenti, sono costruibili numerose famiglie di agenti, ciascuna con delle proprie regole di azione.

Come nel caso del modello *ASM*, esposto nei paragrafi precedenti, numerosi modelli di simulazione si basano su equazioni che sintetizzano l'intera domanda e offerta del mercato. Dall'uguaglianza di domanda e

offerta si ricava un prezzo di equilibrio che costituisce il prezzo di mercato. In modelli di questo tipo, quindi, si ipotizza l'esistenza di un prezzo di equilibrio, mentre invece nella realtà si forma un prezzo di mercato alla conclusione di ogni contratto. Proprio su questa ipotesi implicita si basa la principale differenza tra *ASM*, modello costruito sulle equazioni, e SUM (e la nuova versione *JavaSum*), nel quale si riproducono esattamente i processi base del mercato: domanda e offerta si incontrano attraverso il *book* con l'inserimento degli ordini da parte degli agenti, e il prezzo di mercato coincide sempre con il prezzo dell'ultimo contratto concluso.

In questo modo si utilizzano i modelli di simulazione ad agenti per andare oltre la semplice soluzione di equazioni, permettendo agli agenti di interagire tra di loro senza la necessità di effettuare semplificazioni legate all'impostazione delle equazioni aggregate.

Come visto per il modello *SUM*, è possibile impostare una vasta gamma di tipologie di esperimenti, inventando sempre nuove famiglie di agenti e facendo interagire con questi anche degli agenti umani, attraverso gli "*avatar*".

Con *ASM* e *SUM* si hanno due esempi di utilizzo della simulazione ad agenti per lo studio dei mercati che, pur utilizzando lo stesso tipo di strumenti tecnici, rappresentati dalle librerie e dai protocolli di *Swarm*, hanno impostazioni molto diverse.

DINAMICHE DI MERCATO REALISTICHE

In questo paragrafo si fa riferimento ad uno studio effettuato da Neuberg e Bertels (2003), nel quale si studia un modello di simulazione di borsa in cui opera un insieme di agenti eterogenei, al fine di scoprire quali caratteristiche del modello portino ad ottenere fenomeni simili a quelli osservati nella realtà. La sua somiglianza con *ASM*, il modello di Arthur e altri (1997), permette di approfondirne lo studio e facilitarne la comprensione dei meccanismi interni.

In *ASM* gli agenti formano le loro aspettative sul mercato in base a quelle che credono essere le aspettative future degli altri agenti, che influenzeranno l'andamento dei prezzi. In questo modo il prezzo di mercato converge al livello che è atteso in base alle aspettative razionali degli agenti, che hanno a disposizione un certo numero di modelli di decisione.

Un altro modello molto simile ad *ASM* come impostazione è descritto da Wan e Hunter (1997). In questo caso ogni agente è rappresentato da una funzione matematica e si serve di una serie di regole per la formazione di aspettative riguardo l'andamento futuro dei prezzi. L'elemento di maggiore differenza con *ASM* e con il modello di Neuberg e Bertels (2003) è che

l'apprendimento avviene con la modificazione dei parametri della funzione matematica che definisce i singoli agenti. Tuttavia i modelli si presentano molto simili in quanto il successo dei singoli agenti dipende in larga parte dalla ricchezza iniziale e dal metodo utilizzato per le previsioni.

Descrizione del modello

Gli agenti che operano nel modello sono caratterizzati da diverse capacità di valutazione e sono in grado di giudicare un titolo in base a delle aspettative sui valori futuri di prezzo. Partendo da queste aspettative sono decisi prezzi e quantità delle transazioni che si intendono effettuare sul mercato. Ciò genera un'offerta che è valutata dagli altri agenti operanti sul mercato.

Ogni agente valuta i risultati ottenuti in termini di profitto ed esclude le regole che hanno portato ai risultati peggiori. In questo modo l'agente mette in moto un processo di apprendimento che lo porta a considerare solamente le regole più profittevoli tra quelle che ha a disposizione.

Ogni agente quindi processa le informazioni di cui dispone seguendo determinate regole, per giungere alla formulazione di un ordine di acquisto o di vendita per un particolare titolo. Nel modello in esame vi è un *classifier system* dove i diversi modelli di decisione sono rappresentati da regole nella forma "*if-then*". In un determinato momento, al verificarsi di determinate condizioni corrispondono determinate azioni dell'agente. Le condizioni del mercato sono identificate attraverso sequenze di "0", "1" o "#" (che identifica la presenza di "0" o "1" indifferentemente). Le regole di cui dispone l'agente sono definite anch'esse da una stringa di numeri e simboli. Quindi l'agente deve utilizzare la regola, tra quelle in suo possesso, che si avvicina maggiormente alla situazione di mercato: la stringa che identifica la regola deve essere il più possibile vicina a quella che identifica la situazione di mercato.

Come in *ASM*, sono negoziati titoli *risk-free* e titoli rischiosi. Per ogni titolo rischioso vi sono due valori, *a* e *b*, che permettono di valutare il prezzo futuro atteso per il periodo immediatamente successivo all'istante di valutazione, in funzione dei valori di prezzo e di dividendo conosciuti:

$$E_t^i(p_{t+1} + d_{t+1}) = a_i(p_t + d_t) + b_i$$

Per ogni agente, un insieme di regole, che evolvono con l'utilizzo di un algoritmo genetico, permette di calcolare questo valore atteso. Inizialmente si hanno a disposizione 900 regole, che successivamente possono essere ridotte dal processo di apprendimento.

L'avversione al rischio è espressa in termini di una funzione di utilità di tipo *CARA*, della stessa tipologia di quella utilizzata in *ASM*:

$$U(\omega) = (-e^{-\lambda\omega}),$$

dove ω indica la ricchezza iniziale dell'agente e λ è il grado di avversione al rischio[4] definito come segue:

$$\lambda = \frac{U''(\omega)}{U'}.$$

Il processo di apprendimento aggiorna l'insieme delle regole a frequenze diverse da agente ad agente, a seconda del loro grado di abilità ad apprendere. A differenza degli altri modelli, in cui il comportamento razionale degli agenti si manifesta dopo circa 200.000 iterazioni, grazie ad una fase iniziale di apprendimento effettuata su serie storiche reali, è possibile velocizzare il processo di apprendimento in modo che gli agenti utilizzino, sin dall'inizio della simulazione, regole realistiche. L'efficienza di una regola, da cui dipende la sua sopravvivenza, è definita in funzione dell'errore generato dalla regola stessa, calcolato nel seguente modo:

$$Error = (E[p_{t+1} + d_{t+1}] - p_{t+1})^2$$

definibile anche come:

$$Error - (a(p_t + d_t) + b - p_{t+1})^2$$

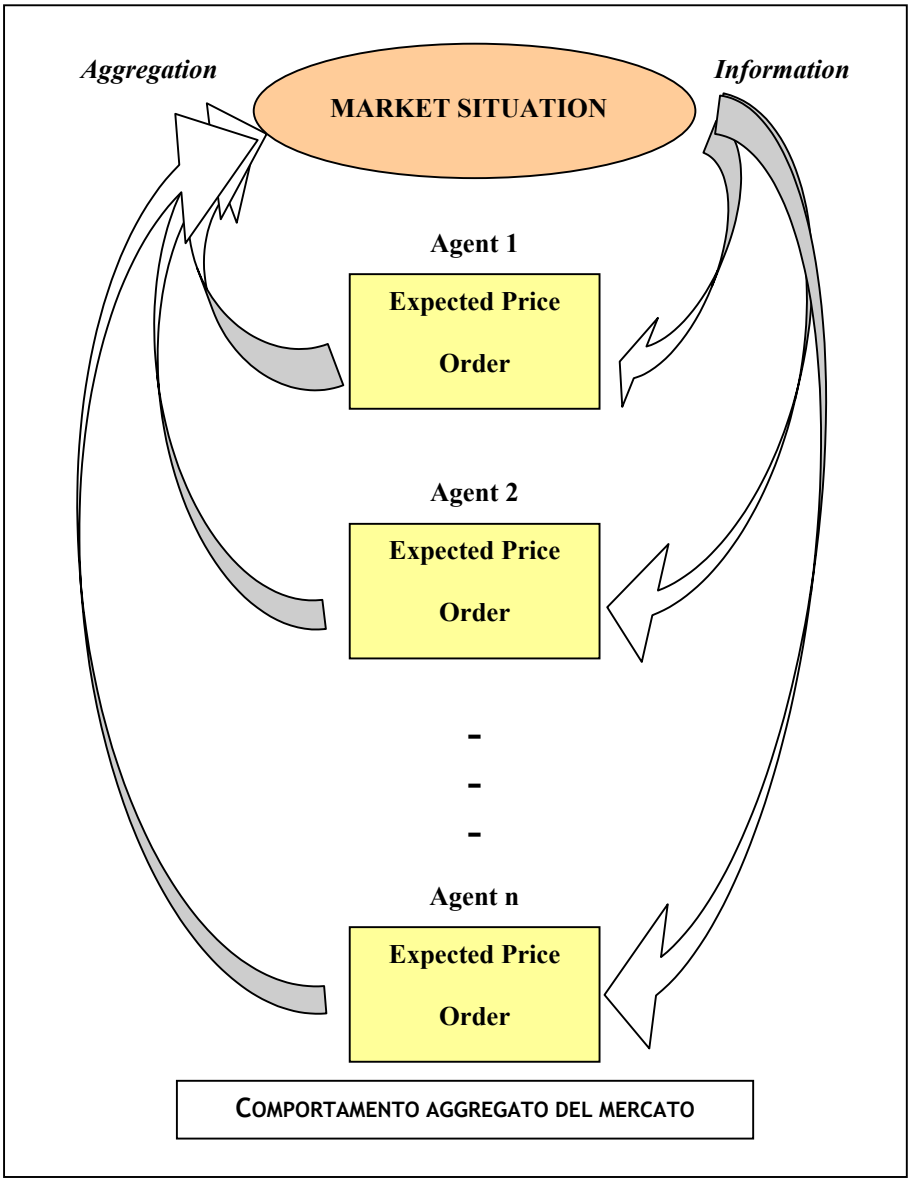
La cosiddetta "forza" di una regola è data da:

$$EVAL(rule) = C - (a(p_t + d_t) + b - p_{t+1})^2$$

Nel caso ipotetico in cui una regola permetta di prevedere esattamente il valore futuro di prezzo, presenta un errore nullo e una forza pari a *C*, che rappresenta il valore massimo di forza che una regola può ottenere.

[4] In Bodie e altri (2002), le scelte di investimento dipendono dalle preferenze e dal modo di porsi nei confronti del rischio dei diversi investitori. Se si accetta come ipotesi la razionalità degli investitori, intesa come consuetudine ad intraprendere azioni che tendano a massimizzare la propria funzione di utilità, si possono individuare tre tipologie di comportamento. Vi possono infatti essere investitori ***avversi al rischio***, che quindi non investono in una attività se questa non remunera a sufficienza il livello di rischio; investitori ***neutrali al rischio***, che quindi valutano gli investimenti badando solamente al livello di rendimento offerto da una attività; e infine investitori ***amanti del rischio***, che a parità di rendimento trarranno maggiore utilità da investimenti più rischiosi. Maggiore è il valore di λ, maggiore sarà l'avversione al rischio dell'agente.

Il mercato simulato può essere rappresentato con il seguente schema:



Come avviene nel mercato reale,le aspettative sono influenzate dalle informazioni provenienti dal mercato. Nel modello simulato le informazioni arrivano ad intervalli regolari e assumono un valore compreso tra -3 e 3. I casi estremi sono notizie:

1. molto negative (rappresentate dal valore -3);
2. neutrali (rappresentate dal valore 0);
3. molto positive (rappresentate dal valore 3).

Come ipotesi è assunta la normalità della distribuzione dei differenti tipi di notizie.

Durante la fase di pre-apprendimento, necessaria a ridurre le regole generate casualmente, in modo che gli agenti operino in modo più coerente

sin dall'inizio della simulazione, si utilizza una serie di dati costruita artificialmente secondo la seguente relazione:

$$p_t = (1 + \alpha i_{t-1}) p_{t-1}$$

dove i_{t-1} è il numero che identifica il tipo di informazione, p è il valore di prezzo e α è la sensibilità del prezzo alle informazioni ricevute dal mercato. Utilizzando per l'apprendimento una serie costruita in questo modo, gli agenti elimineranno le regole che non collegano in modo esatto il segno della variazione di prezzo con il tipo di informazione ricevuta.

In analogia con quanto avviene nel modello *ASM*, ad ogni periodo gli agenti tentano di ottimizzare l'allocazione tra attività rischiose ed attività prive di rischio, le previsioni dell'agente *i* al periodo *t* sono normalmente distribuite con valore atteso $E_{i,t}[p_{t+1} + d_{t+1}]$ e varianza $\sigma^2_{i,p+d}$, e la domanda dell'agente *i* al periodo *t* è pari a

$$x_t^i = \frac{E_t^i(p_{t+1} + d_{t+1}) - (1 + r_f) p_t}{\gamma \sigma^2_{i,t,p+d}},$$

dove p_t è il prezzo del titolo al periodo *t*, λ è il grado di avversione al rischio e r_f è il rendimento dei titoli privi di rischio. Siccome il sistema è chiuso, la domanda totale è uguale al numero di titoli presenti sul mercato:

$$\sum_{i=1}^{N} x_{i,t} = N .$$

Risultati delle simulazioni

Sul mercato sono fatti operare quattro tipi di agenti:

- agenti che operano totalmente a caso (*Crazy agent*);
- agenti che operano tenendo conto delle regole e prendono decisioni seguendo il modello descritto nei paragrafi precedenti (*RAT agent*);
- agenti che operano in senso opposto agli agenti precedenti (*Inverse agent*);
- agenti che tengono meno in considerazione le informazioni che ottengono dal mercato (*Filter agent*).

Durante la prima simulazione operano solamente agenti di tipo *RAT*. Nella tabella che segue sono riassunti i numeri corrispondenti ai diversi tipi di agenti che sono fatti operare durante le simulazioni effettuate con il modello.

Type	7 RAT	5 RAT	4 RAT	3 RAT	2 RAT
Inverse	1	2	2	3	3
Filter	1	2	3	3	3
Crazy	1	1	1	1	2

Come si può notare, l'obiettivo delle simulazioni è quello di testare l'effetto sul mercato dell'operato di un gruppo di agenti sempre più eterogeneo. In tutto si effettuano sei prove modificando la composizione del gruppo di investitori. Queste sei prove sono ripetute più volte modificando i diversi parametri della simulazione. In particolare si analizzano i seguenti casi:

1. diffusione sul mercato di informazioni che sono in gran parte neutrali;
2. diffusione sul mercato di informazioni che sono in gran parte cattive;
3. diffusione sul mercato di informazioni che sono in gran parte buone;
4. si analizzano gli effetti sulla volatilità dei prezzi variando, in fasi successive, i parametri λ e α.

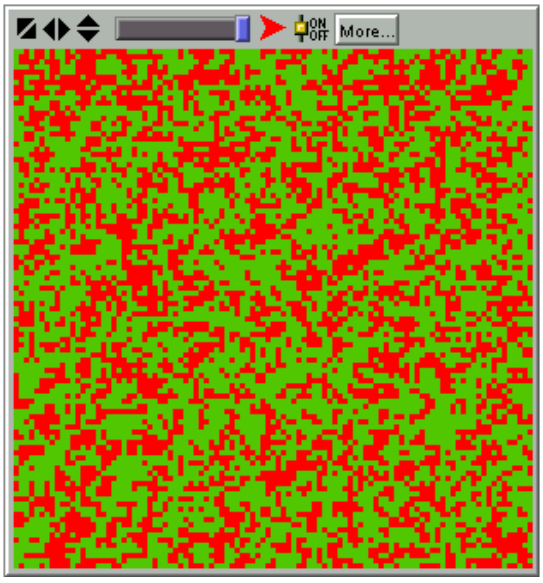
I risultati della simulazione portano a concludere che l'informazione gioca un ruolo fondamentale nel determinare i comportamenti del mercato. Il modello riproduce andamenti caratteristici osservabili sul mercato reale. In particolare si ha la formazione di *crash* quando sono diffuse sul mercato le "cattive" notizie, mentre la formazione di bolle è favorita dalla diffusione delle notizie "buone". Oltre a ciò si osserva che il modello è notevolmente influenzato dall'eterogeneità. Infatti si osserva un comportamento simile a quello reale se operano agenti appartenenti alle diverse tipologie.

ARTIFICIAL FINANCIAL MARKET MODEL

Nel capitolo 3 si sono descritti alcuni ambienti di simulazione. *NetLogo* risulta particolarmente interessante data la sua facilità di utilizzo e, anche se con *Swarm* è possibile costruire modelli di simulazione molto più complessi, Gonçalves (2003) dimostra come in modo semplice e veloce è possibile costruire un modello di simulazione di un mercato finanziario. Nel mercato artificiale emergono fenomeni come bolle e *crash* molto simili a quelli del mercato reale.

Nel mercato artificiale sono fatti operare agenti razionali che sono sensibili alle informazioni diffuse sul mercato e che si lasciano influenzare nelle proprie decisioni dagli agenti con cui sono a contatto. Quindi i risultati del modello possono essere interessanti, oltre che dal punto di vista finanziario, anche dal punto di vista dell'influenza reciproca degli agenti nella formazione di decisioni individuali. Nel modello la decisione riguarda l'acquisto o la vendita di un titolo, ma potrebbe facilmente essere sostituita, ad esempio, da una decisione di voto, per studiare la diffusione delle opinioni che influenzano le scelte.

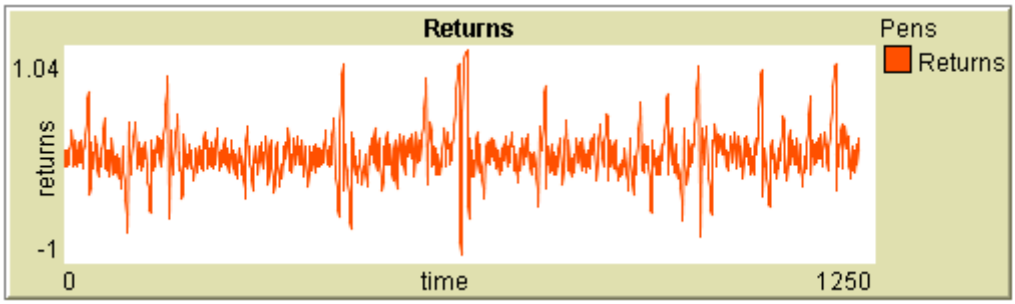
L'*output* del modello è costituito dal cosiddetto "*NetLogo's world*", nel quale sono raffigurati gli agenti in uno spazio, e da tre grafici. Nella figura sottostante è raffigurato il mondo su cui sono distribuiti gli agenti. La colorazione degli agenti varia a seconda delle aspettative che nutrono verso il mercato. In particolare l'agente assume colorazione verde se ha aspettative positive, e rosso se ha aspettative negative. Dalle aspettative derivano le decisioni di acquisto o di vendita.



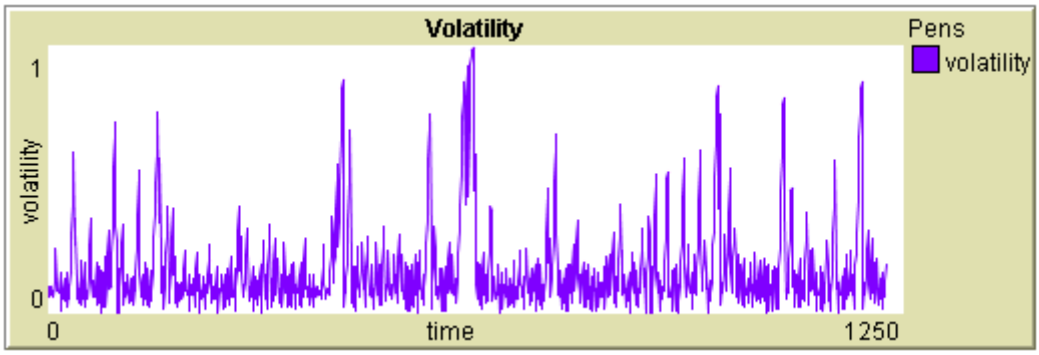
I grafici visualizzati sono i seguenti:

1. l'andamento del logaritmo del prezzo nel tempo, da cui si possono notare andamenti molto simili a quelli del mercato reale, con la formazione di bolle e *crash*;

2. l'andamento dei rendimenti;



3. l'andamento di un indicatore di volatilità, in particolare del rendimento assoluto.



Osservando la simulazione, è interessante notare come gli andamenti dei prezzi siano sensibili ai cambiamenti di "colore" visibili nel "*NetLogo's world*", che rappresentano la diffusione di notizie sul mercato.

Le principali conclusioni della simulazione

Dagli esperimenti effettuati da Gonçalves (2003) si possono trarre principalmente queste tre conclusioni:

a) guardando le colorazioni degli agenti si nota che nel tempo si susseguono fasi in cui gli agenti sono in maggioranza ottimisti o pessimisti;

b) eliminando le assunzioni di razionalità ed omogeneità degli agenti, si nota che l'andamento della volatilità nel tempo dimostra che nel mercato non è rispettata l'ipotesi di efficienza di mercato, con periodi caratterizzati da elevata volatilità seguiti da periodi con volatilità più bassa;

c) la persistenza delle aspettative positive degli agenti per più di un periodo consecutivo genera la formazione di bolle speculative, da cui si esce con il susseguirsi di nuove notizie che modificano in senso contrario le aspettative. Allo stesso modo la persistenza di aspettative negative genera dei *crash*.

CAPITOLO 6

JAVASUM: STRUTTURA E SVILUPPO DEL MODELLO

In questo capitolo si analizzano le varie fasi della costruzione del modello *JavaSum*. La prima parte del capitolo è dedicata all'analisi della programmazione orientata agli oggetti e in seguito sono riassunte alcune nozioni fondamentali dei linguaggi utilizzati in *Swarm*, con alcuni riferimenti a *Jas*. Nella parte finale del capitolo si espongono le caratteristiche del nuovo modello, con la descrizione delle particolarità tecniche della costruzione, e lo si confronta con modello *SUM*.

PECULIARITÀ DELLA OBJECT ORIENTED PROGRAMMING (OOP)

Eckel (2002) sostiene che la grande diffusione dei computer e il ruolo sempre più importante che occupano nella vita di tutti i giorni li portano ad essere considerati come veri e propri veicoli espressivi. I calcolatori infatti diventano strumenti per amplificare la mente umana, tendendo a somigliare sempre più ad estensioni delle nostre menti e sempre meno a macchine. La programmazione ad oggetti (Object Oriented Programming: OOP) è una delle componenti essenziali di questa innovativa concezione di computer.

Tutti i linguaggi di programmazione forniscono astrazioni, a partire dal linguaggio assemblatore, il quale fornisce l'astrazione della macchina sottostante, fino ad arrivare ai linguaggi "asseverativi", come il *C* o il *Basic*, che forniscono un'astrazione del linguaggio assemblatore. In ogni caso tutti questi linguaggi impongono a chi li usa di pensare in termini di struttura

della macchina sottostante. Una seconda alternativa consiste nel modellare i linguaggi sulla struttura del problema che si vuole risolvere.

Con la programmazione ad oggetti il programmatore ha a disposizione una serie di strumenti per rappresentare elementi nello "spazio dei problemi", senza che vi siano particolari vincoli sul tipo di problemi che ci si appresta a risolvere. Gli elementi che si trovano nello spazio dei problemi e la loro rappresentazione nello "spazio delle soluzioni" sono detti "oggetti". L'OOP quindi permette di descrivere il problema nei suoi termini e non nei termini del computer che dovrà risolverlo. Ciascun oggetto assomiglia ad un piccolo computer e presenta forti analogie con gli oggetti fisici del mondo reale: possiede caratteristiche e comportamenti, stati ed operazioni che gli si può chiedere di eseguire.

Il primo linguaggio ad oggetti è *Smalltalk*. Da esso derivano molti dei linguaggi oggi utilizzati, come ad esempio *Java* e *Objective C*. E' possibile individuare cinque caratteristiche fondamentali che distinguono la programmazione orientata agli oggetti:

1. qualsiasi componente concettuale del problema è rappresentabile come un oggetto;
2. un programma si presenta come un insieme di oggetti che interagiscono fra loro con l'invio di messaggi;
3. ogni oggetto ha un tipo, dove per tipo si intende la "classe" da cui deriva un esemplare di un determinato oggetto;
4. ogni tipo di oggetto è creato componendo diversi oggetti già esistenti e la complessità di un programma è celata dalla semplicità degli oggetti;
5. tutti gli oggetti di un determinato tipo possono ricevere gli stessi messaggi.

Nei paragrafi che seguono sono esposte alcune delle principali caratteristiche della programmazione ad oggetti, che sono i diversi livelli di protezione del codice, l'ereditarietà, il polimorfismo e le varie forme di dinamismo della gestione della memoria.

Interfaccia e implementazione

Come si è accennato, nella OOP si raggruppano operazioni in unità modulari chiamate *oggetti*. Un programma risulta essere una struttura a rete e nasce dalla combinazione di vari oggetti che interagiscono fra loro. Le diverse parti di codice, che assolvono ognuna ad un determinato compito, risultano indipendenti e utilizzabili più volte. Il nome del tipo di oggetto che si crea deve essere unico. In questo modo si possono formare biblioteche di

strumenti, i quali sono utilizzabili senza che sorgano conflitti, in quanto ogni tipo di oggetto è indipendente dagli altri.

Gli oggetti dello stesso tipo appartengono alla stessa *classe*. Una classe non è altro che un tipo di dati astratto, con il quale si opera esattamente come con i tipi intrinseci. La principale differenza concettuale tra OOP e programmazione tradizionale è che, per costruire delle applicazioni, occorre ragionare in termini di possibilità operative di un oggetto. Un oggetto è un'unità di alto livello, composta da una serie di caselle di memoria, chiamate *variabili instance*, che è in grado di compiere operazioni sulle variabili attraverso delle funzioni, chiamate *metodi*. I metodi agiscono sulle variabili *instance* e le variabili *instance* sono invisibili al di fuori di un certo oggetto. Gli oggetti della stessa classe possono accedere agli stessi metodi. Ogni volta che si crea un esemplare di una classe, ossia una *instance*, sono create in memoria le variabili *instance*, ma non vi è alcuna allocazione di memoria per i metodi. In memoria rimane un unica copia dei metodi accessibile a tutti gli oggetti della classe. Si opera sulle variabili *instance* inviando messaggi con i quali si richiede l'attivazione di un metodo che l'oggetto riconosce, perché definito all'interno della propria classe.

Ad ogni oggetto è possibile effettuare una serie di richieste, definite dalla sua *interfaccia*. Il tipo di oggetto determina l'*interfaccia* e, in relazione alle funzionalità offerte dall'oggetto, esisterà da qualche parte il relativo codice. Oltre all'*interfaccia* esiste quindi l'*implementazione*. La separazione effettuata nella programmazione ad oggetti tra *interfaccia* e *implementazione* è detta *encapsulation*. L'analogia con gli oggetti reali è molto chiara. Che si pensi ad una radio oppure ad un telefonino cellulare, il generico utente ha sempre a disposizione una serie di tasti o manopole (*interfaccia*) che gli permettono di comunicare con l'oggetto per attivare alcune sue funzionalità, senza che necessariamente si debba preoccupare di conoscere a fondo il tipo di componenti elettronici e i collegamenti tra essi su cui poggiano le funzionalità dell'oggetto (*implementazione*).

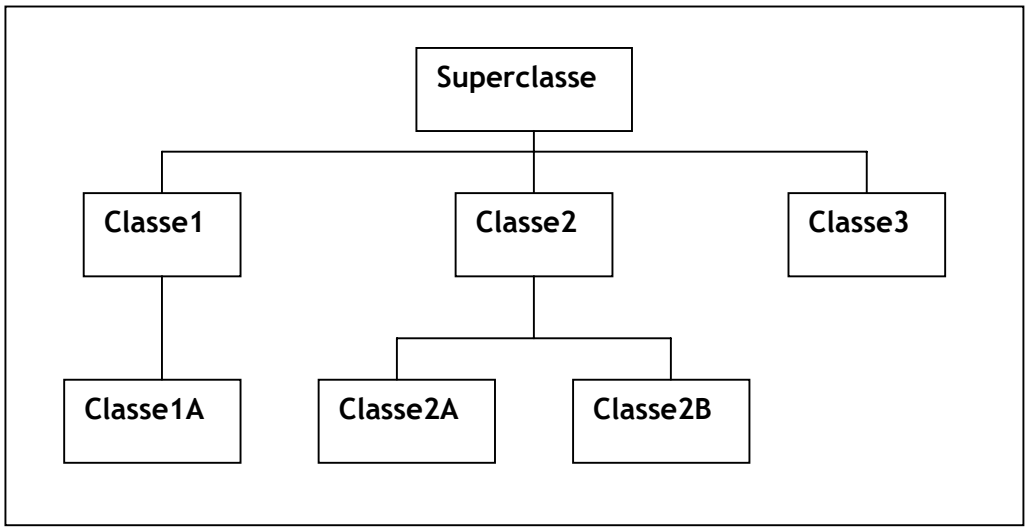
Si possono quindi classificare anche gli utilizzatori della programmazione ad oggetti, separando i "creatori delle classi", ossia chi crea effettivamente i tipi di dati, e i "programmatori *client*", ossia chi utilizza i tipi di dati nelle proprie applicazioni. Continuando l'esempio precedente, si può pensare che chi progetta e realizza i componenti elettronici di un telefonino possa non essere la stessa persona che progetta e assembla il prodotto finale. L'obiettivo primario del primo tipo di programmatori è quello di esporre al programmatore *client* solamente le parti del programma che effettivamente gli interessano e mantenere nascosto tutto il resto. Per il programmatore *client* sono invece importanti gli strumenti che gli permettono di sviluppare velocemente le applicazioni. Tali strumenti, a livello di struttura del codice nascosto, potranno essere nel tempo

modificati dai creatori di classi, per migliorare gli strumenti offerti, senza che i programmatori *client* se ne preoccupino.

Dai diversi linguaggi sono offerte diverse possibilità di limitare l'accesso alle classi, in modo da stabilire i confini tra competenze dei *client* e competenze dei creatori di classi, e proteggere il codice da accessi o azioni indesiderate degli utilizzatori. Solitamente è possibile gestire l'accesso al codice delle classi attraverso tre tipi di dichiarazione. Dichiarando una classe di tipo ***public*** non si hanno particolari limitazioni di accesso alle *variabili instance* e con questo tipo di dichiarazione si rinuncia totalmente alla proprietà di *encapsulation* offerta dalla OOP. Con una dichiarazione di tipo ***protected*** è permesso l'accesso alle variabili solamente all'interno della classe in cui sono dichiarate e all'interno delle classi che ereditano le caratteristiche della classe dichiarante (si veda il paragrafo relativo all'ereditarietà). Infine, con la dichiarazione ***private*** si ha il livello di protezione più elevato e l'accesso alle variabili è permesso solamente all'interno della classe in cui sono dichiarate, accettando in questo modo la perdita di alcuni gradi di flessibilità di utilizzo del codice.

Ereditarietà

Nel caso si presenti la necessità di avere tipi di oggetti molto simili, ma con funzionalità che differiscono per numero o per tipologia, l'OOP offre la possibilità di creare nuove classi partendo da classi preesistenti, a cui si aggiungono le funzionalità necessarie. Questa facoltà è detta ereditarietà. Partendo da un tipo base si generano, a seconda delle esigenze, altri tipi derivati. Nel tipo base saranno risolti i problemi relativi alle funzionalità che accomunano tutti i tipi derivati. Qualsiasi classe può essere a sua volta ereditata e ciò può portare a strutture ereditarie articolate, rappresentabili graficamente come una struttura ad albero.



Questa caratteristica permette al programmatore di sviluppare molto velocemente le proprie applicazioni, sfruttando biblioteche di codice creato da altri programmatori. Il polimorfismo è il principale elemento di differenza concettuale tra la programmazione tradizionale e la programmazione ad oggetti. Nella programmazione tradizionale, il codice deve comprendere le istruzioni necessarie a risolvere ogni parte del problema che si affronta. Nella programmazione orientata agli oggetti invece molte parti del problema sono già state affrontate in precedenza da altri programmatori ed è quindi sufficiente utilizzare degli strumenti già disponibili. Qualora questi strumenti non rispondessero esattamente alle proprie esigenze, è possibile adattarli creando nuove classi che hanno come base di partenza le funzionalità offerte dalle classi già esistenti.

Oltre alla possibilità di aggiungere delle funzionalità ad una classe, inserendo nuovi metodi, è possibile modificare le funzionalità già esistenti, semplicemente aggiungendo nuove caratteristiche al relativo metodo nella classe derivata.[5] Da queste possibilità derivano direttamente le proprietà di polimorfismo dei metodi di diverse classi.

Polimorfismo

La capacità, di oggetti di diverso tipo, di rispondere agli stessi messaggi in modo differente è detta polimorfismo. All'interno dei singoli oggetti di diverso tipo possono infatti esistere metodi con lo stesso nome ma la cui chiamata genera reazioni anche molto diverse.

Viste le opportunità offerte dall'ereditarietà delle classi, spesso si crea la necessità di trattare gli oggetti non come tipi specifici bensì come tipi aventi determinate caratteristiche di base. Ad esempio, se si crea la classe dei poligoni e in seguito si creano le classi derivate relative a tipi particolari di poligoni, le quali ereditano le caratteristiche di base, è utile che funzioni generali come la funzione "disegna" o "calcola il perimetro", possano essere riferite a qualsiasi tipo di oggetto derivato. Se tutti i tipi di oggetti hanno accesso ad un metodo con lo stesso nome, la chiamata di un metodo genera esecuzione di parti di codice diverse a seconda dell'oggetto che si considera. In fase di scrittura del programma, non deve necessariamente essere specificato l'indirizzo esatto del metodo da eseguire. Se, ad esempio, si vuole costruire un programma generico che calcoli il perimetro e permetta di stampare qualsiasi tipo di poligono, nel codice deve essere specificato solamente il nome del metodo, ma l'oggetto ricevente il messaggio deve poter essere determinato nel momento in cui si conosce la natura specifica

[5] Dal punto di vista tecnico, per aggiungere delle funzionalità ad un metodo ereditato, occorre definire un nuovo metodo che richiami al suo interno il metodo meno evoluto dalla *superclasse* e contenga le ulteriori istruzioni necessarie affinché il metodo risponda alle nuove esigenze.

dell'oggetto con cui si opera. L'indirizzo del codice non deve essere determinato in fase di compilazione, ma al momento in cui avviene l'esecuzione del programma. Tale proprietà è detta *late* o *dinamic binding* (collegamento ritardato o dinamico) e si contrappone a ciò che normalmente accade nei compilatori non *OOP*, che funzionano con *early binding* (collocamento anticipato o statico), ossia il compilatore genera una chiamata ad una funzione specifica e l'indirizzo del codice è determinato in termini assoluti (nel paragrafo relativo al dinamismo si approfondiscono questi aspetti).

Il polimorfismo è molto vicino al fenomeno detto *overloading*, che consiste nel "sovraccarico" di determinati operatori o funzioni. Una funzione con lo stesso nome, ad esempio una generica funzione "somma", può trovarsi a dover operare con diversi tipi di dati che gli sono passati. Dal punto di vista tecnico, questo comporta l'esecuzione di funzioni diverse a seconda del tipo di dati passati alla funzione (si pensi ad esempio ad una funzione "somma" che sia in grado di sommare sia stringhe che numeri). La stessa cosa può avvenire con i metodi a cui tendenzialmente si assegna un nome che è semanticamente collegato con la funzione che svolgono. Per oggetti di diverso tipo, è possibile che siano necessari metodi che svolgano una stessa operazione, ma che tale operazione comporti l'esecuzione di codice diverso a seconda dell'oggetto.

Dinamismo

Come spiegato in Apple Computer (2003), l'allocazione della memoria è uno dei maggiori problemi tecnici con cui ci si scontra in fase di costruzione di un programma. Un'adeguata gestione della memoria consente di raggiungere notevoli vantaggi in termini di velocità di esecuzione e di maggiore flessibilità e chiarezza del codice, in modo da favorirne lo sviluppo e la manutenzione. I linguaggi orientati agli oggetti offrono la possibilità di utilizzare dinamicamente la memoria, senza necessariamente specificare nel codice del programma, e quindi staticamente, gli indirizzi delle caselle di memoria, spostando le varie allocazioni, dalla fase di compilazione alla fase di esecuzione del programma (*run-time*).

Le tipologie principali di dinamismo offerte dalla programmazione ad oggetti sono il *dinamic typing*, il *dinamic binding* e il *dinamic loading*.

Dinamic typing. Con questo termine si intende la possibilità di determinare il tipo di oggetto al momento dell'esecuzione e non durante la compilazione. Nel codice, l'oggetto è dichiarando in modo generico, senza specificazione di classe. Questa tipologia di dinamismo diventa cruciale se si pensa alle possibilità offerte dal polimorfismo. Essendo possibile determinare il tipo dell'oggetto al *run-time*, nel codice sorgente diventa importante solamente

il riferimento al nome del metodo da richiamare. Durante l'esecuzione il messaggio arriva all'oggetto e in base al suo tipo è attivato il metodo esatto. Si possono in questo modo richiamare metodi in maniera generica e vi è la possibilità di rendere la struttura del programma molto flessibile. Il *dinamic typing* è necessario per il *dinamic binding*.

Dinamic binding. La possibilità di determinare l'indirizzo di memoria del metodo richiamato, al momento dell'esecuzione del programma, è detta *dinamic binding*. Considerate le proprietà del *dinamic typing* e del polimorfismo dei metodi, è possibile spostare al momento dell'esecuzione (e non al momento della compilazione) la determinazione del metodo da eseguire. In questo modo si evita di costruire complicate procedure per fare eseguire funzioni diverse a seconda della situazione che si deve affrontare: la funzione giusta è richiamata in base al tipo di oggetto che ci si trova di fronte. I vantaggio principale derivante dal *dinamic binding* e dal *dinamic typing* consiste nella possibilità di inviare messaggi a oggetti generici, anche appartenenti a classi che potranno essere inventate in futuro. Naturalmente occorre che vi sia coerenza tra il nome dei metodi che sono presenti nelle varie classi che si creano.

Dinamic loading (o dinamic linking). Suddividendo adeguatamente in moduli un programma, in particolare se si tratta di un progetto di una certa grandezza, è possibile mandare in esecuzione solamente le parti utili a compiere le azioni di interesse dell'utilizzatore, con notevoli vantaggi per quanto riguarda la velocità di esecuzione e per l'eventuale aggiunta di nuove funzionalità alle applicazioni costruite. Inoltre è possibile compilare separatamente le diverse parti del codice.

OBJECTIVEC

Il linguaggio *ObjectiveC* è un linguaggio orientato agli oggetti che deriva direttamente dal linguaggio *C*, aggiungendo ad esso la possibilità di programmare ad oggetti. Esso può anche essere utilizzato come estensione del linguaggio *C++*. Le aggiunte derivano sostanzialmente da *Smalltalk e* il vantaggio principale consiste nella grande flessibilità data dalla possibilità di scelta tra lo sviluppo di programmi in modo procedurale oppure con orientamento agli oggetti.

La sintassi è praticamente identica al C *standard*, ma sono numerose le difficoltà concettuali che occorre affrontare se non si ha famigliarità con i linguaggi orientati agli oggetti. Il principale pregio di questo linguaggio, che ne ha determinato un discreto successo tra gli sviluppatori di simulazioni, è rappresentato dalla semplicità di utilizzo e dal dinamismo della gestione della memoria che ha notevoli ritorni in termini di velocità di esecuzione.

Infatti, gran parte delle azioni di gestione della memoria sono compiute in fase di esecuzione e non in fase di compilazione.

Nel paragrafo seguente sono esposti sinteticamente alcuni aspetti tecnici rilevanti ai fini della comprensione dei programmi in *ObjectiveC*.

Nozioni tecniche generali

In *ObjectiveC* le classi sono solitamente composte da due *file* separati (anche se non è obbligatorio). Entrambi i *file* sono denominati esattamente come la classe, ma hanno diversa estensione. Il primo *file*, con estensione "h", è detto *file di interfaccia* e contiene tutte le dichiarazioni delle variabili globali[6] e le dichiarazioni dei metodi.

```
#import <nome.h>
...
@interface NomeClasse: Superclasse
{
...
dichiarazione variabili
...
}
...
dichiarazione metodi
...
@end
```

Come si può notare dallo schema precedente, i *file di interfaccia* sono composti principalmente da due blocchi di istruzioni. In un primo blocco sono elencati una serie di nomi di *file interfaccia*, preceduti dall'istruzione "`#import`". In questo modo si richiamano, all'interno della classe che si crea, tutti i metodi contenuti nelle classi qui importate.

Un secondo blocco di istruzioni è racchiuso tra l'istruzione "`@interface NomeClasse: Superclasse`" e l'istruzione "`@end`" ed è composto a sua volta da due parti. In una prima parte, racchiusa tra parentesi graffe, vi sono le dichiarazioni delle variabili globali, mentre nella seconda parte sono dichiarati i metodi della classe. Con queste dichiarazioni si esplicita il nome del metodo, la presenza di eventuali parametri e il tipo di dato che il metodo restituisce una volta richiamato.[7] Il nome del metodo

[6] Per variabile globale si intende una variabile vista da tutti i metodi della classe. All'interno dei singoli metodi è possibile che vi siano altre dichiarazioni di variabili "locali", le quali sono utilizzate solamente all'interno del metodo. Una variabile globale può avere un nome che coincide con una variabile locale, ed è possibile che in metodi diversi siano dichiarate variabili locali con lo stesso nome.

[7] I parametri e il tipo del valore di ritorno non sempre sono necessari.

è preceduto da un segno "+" qualora il metodo sia un "metodo di classe", in questo caso risulta possibile richiamarlo solamente dalla classe stessa, oppure un segno "-", in modo che il metodo risulti accessibile alle *instance* generate dalla classe. La sintassi è la seguente:

```
-(tipo ritorno) nomeMetodo:(tipo parametro) parametro;
```

Nel caso la classe sia un'estensione di una classe già esistente, nell'interfaccia è specificata la classe da cui eredita le caratteristiche ed è importato il *file* di interfaccia relativo a tale classe.[8]

Il secondo *file* necessario per definire una classe è detto *file di implementazione* e al suo interno è sviluppato il codice relativo ai metodi della classe. Questa seconda tipologia di *file* ha estensione "m" e permette al compilatore di distinguere un *file* contenente un programma scritto in *ObjectiveC* da uno contenente un programma in altri linguaggi.

```
#import "NomeClasse.h"
@implementation NomeClasse
+ metodo1
{
...
return self
}
...
- metodo2
{
...
return self
}
@end
```

All'inizio del *file* è presente l'istruzione necessaria ad importare il *file* di interfaccia relativo alla classe. Tra le istruzioni "`@implementation NomeClasse`" e "`@end`" vi sono i diversi blocchi di codice relativi ai singoli metodi della classe. Se non è specificato un valore di ritorno (il valore contenuto nella variabile specificata dopo l'istruzione "`return`"), il metodo restituisce l'indirizzo di memoria della classe in cui è contenuto (contenuto in "`self`").

Oltre alla dichiarazione delle variabili, se all'interno della classe sono utilizzati degli oggetti, derivati da altre classi, questi devono essere

[8] Nel primo blocco, con un'istruzione `#import` si importa il *file* di interfaccia della *superclasse*. Il nome della *superclasse* è inoltre specificato nell'intestazione del secondo blocco di istruzioni: l'istruzione `@interface` è seguita dal nome della nuova classe e dal nome della *superclasse*.

dichiarati. Sono possibili due tipi di dichiarazioni. Si può infatti dichiarare genericamente un oggetto seguendo la seguente sintassi:

```
id <classe> nomeOggetto
```

Tra le virgolette si specifica la classe che contiene i metodi che possono essere utilizzati dall'oggetto, ma tale specificazione può essere omessa. In questo modo al compilatore si fornisce solamente l'informazione che i nomi così dichiarati non sono riferiti a variabili semplici, ma sono oggetti.

Oltre alla dichiarazione generale, è possibile dichiarare un oggetto attraverso la cosiddetta "tipizzazione statica". L'oggetto è dichiarato come puntatore alla classe da cui deriva. In questo modo l'oggetto risulta dichiarato con maggiore precisione in quanto, a differenza della dichiarazione generale, il compilatore può sapere in precedenza la struttura della classe da cui discende l'oggetto. La sintassi in questo caso è la seguente:

```
NomeClasse * nomeOggetto
```

Il passo successivo la dichiarazione è la creazione vera e propria dell'oggetto, che si effettua in due fasi. La prima fase consiste nell'allocazione dello spazio in memoria necessario all'oggetto, mentre nella seconda, l'oggetto è "inizializzato".[9] Una volta creati gli esemplari di una classe, è possibile mandare loro dei messaggi, i quali attiveranno determinati metodi. L'istruzione necessaria per richiamare il metodo relativo ad un oggetto è composta dal nome dell'oggetto seguito dai messaggi diretti ad un metodo specifico, il tutto contenuto tra parentesi quadre:

```
[nomeOggetto messaggio1:valore messaggio2:valore]
```

Esiste inoltre la possibilità che un valore passato come messaggio ad un metodo, sia a sua volta risultante dal richiamo di un altro metodo. Ciò comporta il formarsi di istruzioni "annidate" l'una all'interno dell'altra. Di seguito è riportato un esempio:

```
[nomeOggetto1 messaggio:[nomeOggetto2 messaggio:valore]]
```

Nel caso in cui vi sia la necessità di richiamare un metodo della classe all'interno di un metodo della classe stessa, occorre specificare nel messaggio l'indirizzo di memoria della classe di appartenenza (con l'aggiunta di "`self`" all'inizio dell'istruzione di richiamo).

```
[self nomeOggetto messaggio1:valore]
```

Se il metodo che si richiama è situato nella *superclasse*, al posto di "`self`", occorre vi sia "`super`".

[9] Per inizializzazione si intende la conferma dell'indirizzo di memoria delle variabili relative all'oggetto e la pulizia delle aree di memoria che saranno utilizzate da esso. La trattazione

```
[super nomeOggetto messaggio1:valore]
```

Normalmente nei metodi, per fare riferimento alle variabili dichiarate, si utilizza il nome loro assegnato. Nel caso in cui all'interno di un metodo vi sia il riferimento a variabili che non appartengono all'oggetto ricevente i messaggi, l'oggetto deve essere in precedenza tipizzato staticamente, dopodiché è possibile agire sulle variabili attraverso il simbolo "->", che è detto "puntatore alla struttura".

```
oggettoTipizzato->variabileInstance=variabile;
```

Come si può notare in tutti gli esempi fatti finora, è convenzione che il nome della classe sia sempre scritto con la lettera iniziale maiuscola, mentre gli esemplari abbiano sempre lettera iniziale minuscola. Del tutto arbitrari sono i nomi delle classi e dei metodi, ma solitamente sono scelti nomi collegati al fine per cui il codice è stato scritto.

JAVA

Come per *Objective C*, la sintassi di *Java* è essenzialmente la stessa del *C standard*, in quanto deriva direttamente dal linguaggio *C++*. Uno degli aspetti più interessanti di *Java*, il quale ha favorito la sua grande diffusione, è la possibilità di sviluppare applicazioni senza preoccuparsi di problemi di compatibilità che queste devono possedere con sistemi operativi di diverso tipo. Dopo la compilazione, un programma scritto in *Java* può essere eseguito su quasi tutti i sistemi operativi, nei quali esiste una cosiddetta *Java Virtual Machine (JVM)*, che consiste in un interprete di programmi compilati in *Java*. In seguito alla compilazione si creano dei *file* con estensione "*class*", che rappresentano una fase intermedia della compilazione vera e propria (il codice che li costituisce è detto "*mesocode*"), la quale è conclusa solo sul computer in cui si esegue il programma, grazie all'interpretazione dei *file* "*class*" da parte della *JVM*.

Grazie alle sue caratteristiche di flessibilità, compatibilità, facilità di utilizzo e la possibilità di fare eseguire le applicazioni all'interno di pagine *web* (con la costruzione delle cosiddette "*applet*"), *Java* ha costruito intorno a sé una grande comunità di utilizzatori, i quali possono usufruire di librerie di codice molto grandi e complete, che permettono di costruire applicazioni in maniera semplice e veloce.

specifica dell'argomento è affrontata in maniera più approfondita nei paragrafi seguenti, in cui sono elencate le differenze tecniche di creazione degli oggetti nei vari linguaggi.

Nozioni tecniche generali

Le applicazioni sono suddivise in diversi *file* e il *file* principale, il quale possiede il nome del programma, contiene il metodo *main*. La struttura di base è la seguente:

```
class nomeprogramma
{
public static void main (String arg[])
{
...istruzioni...
}
}
```

Il metodo *main* ha la caratteristica di essere statico, ossia al momento della compilazione sono allocate le aree di memoria necessarie all'esecuzione del programma. All'interno di questo metodo si creano gli oggetti e si richiamano i metodi necessari per l'esecuzione dell'applicazione.

Oltre alla classe principale, possono essere definite altre classi, le quali possono essere statiche o dinamiche. Nel caso si crei una classe statica, questa è direttamente utilizzabile, ma allo stesso tempo è possibile creare delle *instance* della classe. Nel caso una classe sia dichiarata non statica, essa non può essere utilizzata se non è stato prima creato un esemplare della classe. Le classi sono quindi un insieme di "regole di fabbricazione" degli oggetti. Le classi statiche sono particolarmente utili se nell'applicazione che si costruisce si utilizza un solo esemplare della classe.

Una generica classe presenta la seguente struttura:

```
#import...
accesso class nome_classe extend superclasse
{
...dichiarazioni variabili di classe...
//costruttore
nome_classe(parametri)
{
...dichiarazioni per la costruzione della classe in base
ai parametri...
}
...definizione metodi...
}
```

Nel *file* della classe sono importate, innanzitutto, le librerie di funzioni necessarie per lo sviluppo della classe stessa. L'intestazione della

classe è composta dal tipo di accesso, che può essere *public*, *protected* o *private*, dal nome della classe e dal nome della *superclasse* da cui eventualmente eredita le caratteristiche. Il primo metodo di qualsiasi classe è detto "costruttore" ed è denominato esattamente come la classe. In una classe possono essere definiti più costruttori, a seconda del tipo e del numero di parametri che possono recepire. Al momento della creazione di un esemplare della classe, in base ai parametri passati al costruttore, la classe attiva internamente il costruttore adeguato.

```
nome_esemplare = new nome_classe(parametri);
```

Oltre ai metodi costruttori possono essere definiti altri metodi nel seguente modo:

```
accesso tipo_di_ritorno nome_metodo(parametri)
{
...dichiarazione variabili interne al metodo...
...istruzioni del metodo...
}
```

Nell'intestazione di ogni metodo si definisce il tipo di accesso al metodo, il suo tipo di valore di ritorno ed infine il nome ed eventualmente i parametri necessari. All'interno di ogni metodo possono essere dichiarate delle variabili, le quali risultano invisibili all'esterno del metodo.

La logica di fondo del linguaggio consiste nel considerare qualsiasi cosa come un oggetto. Quindi, a meno che le classi di derivazione degli oggetti non siano statiche e quindi direttamente utilizzabili, per utilizzare un metodo di una classe occorre creare una *instance* della stessa.

Una delle caratteristiche dei programmi scritti in *Java* è la particolare attenzione rivolta alla gestione delle eventualità di un errore, il quale genererebbe un'eccezione. Eckel (2002) afferma che le eccezioni sono generate da errori rilevabili solamente in fase di esecuzione del programma e non possono essere rilevati in fase di compilazione. In questi casi è necessario predisporre nel codice le istruzioni di emergenza, da eseguire nel momento in cui l'errore si verifica. Ciò è reso possibile dalla seguente coppia di istruzioni:

```
try
{
...istruzioni...
{
catch (Exeption e)
{
...istruzioni di emergenza...
```

```
}
```

All'interno del blocco di istruzioni "try" è definita l'istruzione che potrebbe generare un errore in fase di esecuzione, e nel blocco di istruzioni "*catch*", nel caso sia generata una eccezione, questa è "catturata" e sono eseguite le istruzioni di emergenza, che solitamente consistono in un messaggio di errore e la successiva interruzione del programma.

Una volta costruita un'applicazione, è possibile generare una completa documentazione delle classi create attraverso l'utilizzo dei cosiddetti *javadoc*. Infatti, inserendo i commenti al codice seguendo una determinata sintassi, è possibile, tramite un'istruzione inserita nella riga di comando, una volta posizionatisi all'interno della cartella contenente i *file* con estensione "*java*", creare delle pagine *web* contenenti la struttura delle classi arricchita dei commenti inseriti fra il codice.

Per inserire un commento che sia riprodotto nei *javadoc*, occorre inserirlo nella riga precedente il nome della variabile o del metodo che si desidera commentare, nel seguente modo:

```
/** commento */
dichiarazione variabile;

/** commento */
intestazione metodo o classe
```

Se si vuole che il commento sia visualizzato su più righe, è possibile inserirlo nel modo seguente:

```
/**
  *commento1
  *commento2
  */
```

Se si vogliono inserire informazioni specifiche, esistono alcune parole-chiave che permettono di inserire commenti di un determinato tipo, come ad esempio il nome dell'autore del programma:

```
/** @author nome autore */
```

Oltre al nome dell'autore è possibile inserire collegamenti ipertestuali ad altre pagine contenenti la documentazione di altre classi o metodi (con il comando *@see*), oppure inserire informazioni sulla versione del programma (con *@version*).

Per la costruzione delle pagine *web* risulta utile creare un *file* di tipo "*bacth MS-DOS*", contenente le istruzioni necessarie alla creazione di una cartella, denominata ad esempio "*doc*", nella quale possono essere salvate le pagine della documentazione create. Il *file* dovrà contenere almeno le seguenti istruzioni:

```
deltree doc

md doc
```

C:\j2sdk1.4.2\bin\javadoc *.java -author -d doc\[10]

In questo modo, lanciando il *file* con estensione "*bat*", si elimina la cartella "*doc*" precedentemente creata, per poi ricrearla e richiamare il comando per la creazione dei *javadoc*.

SWARM: PRINCIPALI ASPETTI TECNICI

Nel capitolo 3 sono illustrate le principali caratteristiche delle simulazioni in *Swarm*. Nel presente paragrafo si illustrano i principali aspetti tecnici di un modello di simulazione costruito in linguaggio *Java*, con le varianti necessarie per l'utilizzo delle librerie di *Swarm* (si veda anche Johnson e altri 1999).

Il cuore della simulazione è rappresentato dal *model*, il quale è definito in un *file*, il cui nome contiene, per convenzione, la parola "*Model*" e coincide con il nome della classe definita al suo interno. La struttura essenziale del *model* è la seguente:

```
...importazione delle librerie di codice
utilizzate nella classe e inizializzazione
delle variabili per la probe...
public class ModelSwarm extends SwarmImpl
{
...dichiarazione variabili
e oggetti utilizzati nel model...

    public ModelSwarm(Zone azone)
    {
        super(azone);
        ...costruzione della sonda (probe)...
    }
    public Object buildObjects()
    {
        ...dichiarazione variabili...
        super.buildObjects();
        ...creazione di oggetti,
        agenti e liste necessari...
        return this;
```

[10] L'indicazione della posizione di memoria della cartella "*javadoc*", dipende dal tipo di installazione e dalla versione del *Java Development Kit* presente sul computer che si utilizza.

```
    }
    public Object buildActions()
    {
        ...dichiarazione variabili...
        super.buildActions();

        ...creazione dei gruppi di azioni
        e dello schedule...
        return this;
    }
    public Activity activateIn(Swarm swarmContext)
    {
     super.activateIn(swarmContext);
     modelSchedule.activateIn(this);
     return getActivity();
    }
    ...definizione altri metodi...
}
```

Come si può notare, il *model* risulta composto da quattro metodi principali, il primo dei quali (denominato come la classe) è il costruttore. Al suo interno si costruisce la sonda, la quale è visualizzata sullo schermo con i valori di base assegnati alle variabili in fase di inizializzazione. Nel metodo *builtObject()* sono costruiti e inizializzati tutti gli oggetti necessari alla simulazione, compresi gli agenti e le liste di agenti. Nel metodo *builtAction()* sono invece costruiti i gruppi di azioni e lo *schedule*, il quale programma nel tempo l'esecuzione dei diversi gruppi. Il metodo *activateIn()* è presente in tutte le simulazioni (fa parte del protocollo) e permette all'intera simulazione di funzionare. Si noti che all'interno di questi metodi sono inviati messaggi alla *superclasse SwarmImpl*. Oltre a questi metodi fondamentali, possono in seguito essere definiti altri metodi a seconda delle esigenze.

La struttura dell'*observer* si presenta simile a quella del *model*, e al suo interno si crea il *model* e se ne richiamano i metodi appena descritti. Come esposto nel capitolo 3, l'*observer* "avvolge" il *model* e si presenta come il "piano" dell'osservazione, separato dal "piano" della simulazione.

L'*observer* presenta la seguente struttura di base:

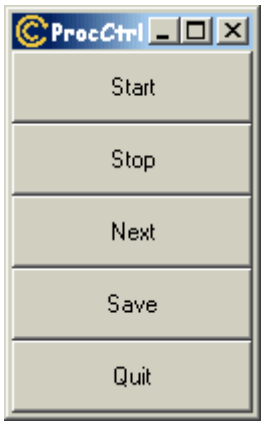
```
...importazione delle librerie di codice
utilizzate nella classe e inizializzazione
delle variabili per la probe...
```

```
public class ObserverSwarm extends GUISwarmImpl
{
...dichiarazione variabili
e oggetti utilizzati nell'observer...
     public ObserverSwarm (Zone azone)
     {
         super(azone);
         ...costruzione della sonda...
     }
     public Object buildObjects()
     {
         ...dichiarazione variabili...
         super.buildObjects();
         ...creazione del model e delle sonde
         sia del model che dell'observer...
         getControlPanel().setStateStopped();
         ...richiamo del metodo buildObject() del model
         e creazione dei grafici e dei file di dati...
         return this;
    }
    public Object buildActions()
    {
         ...dichiarazione variabili...
         super.buildActions();
         ...richiamo del metodo builtAction() del model...
         ...creazione dei gruppi di azioni
         e dello schedule...
         return this;
    }
    public Activity activateIn(Swarm swarmContext)
    {
         super.activateIn(swarmContext);
         modelSwarm.activateIn(this);
         displaySchedule.activateIn(this);
         return getActivity();
    }
    ...definizione altri metodi...
```

```
}
```

In particolare, tra gli "altri metodi", solitamente si definisce un metodo per il controllo della fine della simulazione.

Il metodo "*main*" è quello attraverso il quale si fa funzionare l'intera simulazione ed è contenuto all'interno del *file*, il cui nome inizia con la parola-chiave "*Start*" (sempre in base al protocollo di *Swarm*). La particolarità di questa classe è la staticità. Essendo la classe "di partenza", deve essere necessariamente statica, in quanto non è creata all'interno di nessun'altra classe e, per permettere al sistema operativo di fare funzionare la simulazione, occorre che le zone di memoria di questa classe esistano. Al suo interno sono richiamati tutti i metodi della classe "*Observer*", più alcuni metodi delle classi di *Swarm* che avviano la simulazione e il pannello di controllo rappresentato in figura.



Solitamente, la struttura del codice è la seguente:

```
import swarm.Globals;
import swarm.defobj.Zone;
public class StartJavaSum
{
    public static void main (String[] args)
    {
     ObserverSwarm displaySwarm;
     // Swarm initialization.
     Globals.env.initSwarm ("nomeApplicazione", "versione",
                          "indirizzo_e-mailAutore", args);
    // Create a top-level Swarm object, now DisplaySwarm,
    // and build its internal objects and activities.
    displaySwarm = new
    ObserverSwarm(Globals.env.globalZone);
    displaySwarm.buildObjects();
```

```
        displaySwarm.buildActions();
        displaySwarm.activateIn(null);
        // Now start the displaySwarm and the control panel it
        // provides.
        displaySwarm.go();

         displaySwarm.drop();
         }
    }
```

Come si può notare, dopo la dichiarazione dell'oggetto *observer* vi è l'istruzione, tipica ed obbligatoria per tutte le applicazioni sviluppate in *Swarm*, "*Globals.env.initSwarm (...)*", che contiene tra parentesi il nome dell'applicazione, la versione, l'indirizzo *e-mail* dello sviluppatore e la possibilità di recepire dei parametri dalla riga di comando (*args*). Tutte queste informazioni possono essere richieste digitando alcuni parametri in fase di lancio dell'applicazione. Alla fine del *file*, dopo che si sono richiamati i metodi dell'*observer*, si richiama il metodo "*go()*", che crea il pannello di controllo precedentemente illustrato, e il metodo "drop()" che serve a ripulire le zone di memoria quando la simulazione è terminata.

Per la compilazione dell'intero pacchetto di *file* costituenti la simulazione, si utilizza il "*Makefile*", il quale presenta la seguente struttura:

```
JAVA_SRC = StartNomeSimulazione.java ObserverSwarm.java
ModelSwarm.java ...altri file da compilare...
all: $(JAVA_SRC)
   $(SWARMHOME)/bin/javacswarm $(JAVA_SRC)
clean:
   rm -f *.class
cleanall:
   rm -f *.class
   rm -f *~
   rm *.stackdump
```

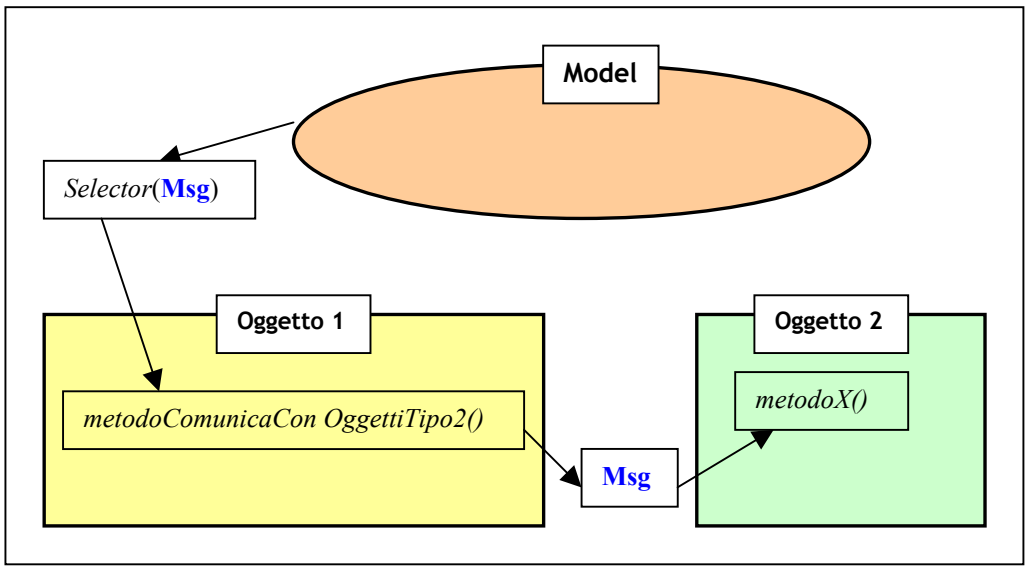
Il principale fine di questo *file* è la possibilità di compilare in blocco i *file* indicati nella prima istruzione semplicemente lanciando il comando "*make*" dalla riga di comando. In questo modo si evita di dover compilare singolarmente i vari *file*. Lanciando il comando "*make*" seguito dal comando "*clean*" si ottiene la cancellazione dei *file* con estensione "*class*" (frutto di compilazioni precedenti). Con "*clean all*" si cancellano anche i *file* di backup creati da *Emacs* (se lo si è utilizzato per la costruzione dei *file* in *Java*) e dei *file* creati in seguito ad errori di esecuzione della simulazione.

Oltre ai *file* finora illustrati vi sono quelli relativi alle classi di agenti e alle classi di oggetti necessari alla simulazione, la cui struttura è completamente a discrezione dello sviluppatore.

Selector e SwarmUtils

Molte volte, costruendo simulazioni, si crea la necessità di dover comunicare ad un oggetto un metodo, affinché questo lo trasferisca ad un altro oggetto, il quale appartiene alla classe in cui il metodo è definito. Questo non è possibile senza l'utilizzo dei *Selector* di *Swarm*.

Supponiamo, per esempio, che il *model* debba comunicare un metodo ad un oggetto di una determinata classe, ma non sia noto l'indirizzo di quest'ultimo e che, per conoscere l'indirizzo, sia necessario richiederlo ad un altro oggetto. Creando un *Selector* è possibile "nascondere" il metodo ed inviarlo ad un secondo oggetto, che a sua volta provvederà a recapitarlo all'oggetto destinatario. Senza l'utilizzo dei *Selector* questa operazione risulterebbe impossibile. Lo schema sottostante raffigura l'operazione:



L'utilizzo più frequente nei modelli di simulazione avviene quando nello *schedule* del *model* si indirizzano messaggi a liste di agenti. Questo comporta che ogni agente, nell'ordine dato dalla lista, attivi un proprio metodo. Con il *Selector* il metodo destinato agli agenti può essere inviato ad una lista, la quale contiene dei metodi per inviare il messaggio agli agenti. In particolare, è possibile inviare il messaggio ad un singolo agente oppure a tutti gli agenti della lista.

Il *Selector* è utilizzato all'interno di un blocco "*try-catch*", necessario per prevenire l'eventuale mancato ritrovamento del metodo all'interno della classe di appartenenza dell'oggetto a cui il messaggio è destinato. Fino all'esecuzione, infatti, non è possibile verificare se il metodo inviato attraverso il *Selector*, in quanto "nascosto", esista o meno. Per evitare di dover costruire un intero blocco di istruzioni per la gestione delle eccezioni ogni volta che si utilizza un *Selector*, gli autori di *Swarm* hanno creato la

classe *SwarmUtils*, con la quale la sintassi per l'utilizzo del *Selector* è notevolmente semplificata.

Nel paragrafo che segue si analizzano la costruzione di uno *schedule* e la sintassi dei comandi relativi al *Selector*.

La gestione del tempo

La gestione delle diverse fasi della simulazione è gestita attraverso lo *schedule*. La "giornata" simulata sarà composta da un determinato numero di istanti, detti anche "tic", e per ognuno di questi sono stabiliti i diversi passi della simulazione. Ad ogni istante sarà possibile compiere un gruppo di azioni, a differenza di quanto avviene nei modelli sviluppati con la programmazione tradizionale.

La sintassi per la costruzione di un gruppo di azioni è la seguente:

```
modelActions = new ActionGroupImpl(getZone());
try
{
sel = new Selector(Class.forName("Classe"),
      "metodo",false);
modelActions.createActionForEach$message(oggetto, sel);
}
catch (Exception e)
{
System.err.println("Exception: " + e.getMessage());
System.exit(1);
}
```

Se supponiamo che "oggetto" sia una lista di oggetti che hanno accesso al metodo "Metodo", nel gruppo di azioni denominato "*modelAction*" si aggiunge il richiamo del metodo per ogni oggetto della lista.[11] Seguendo questa sintassi, allo stesso gruppo di azioni possono essere aggiunte tutte le azioni che è necessario siano compiute nello stesso istante della simulazione (lo stesso "tic").

Utilizzando la classe *SwarmUtils* per la costruzione del *Selector*, la sintassi risulta notevolmente semplificata e per aggiungere una azione al gruppo bastano le seguenti due istruzioni (dopo che è stato creato un esemplare della classe *ActionGroupImpl*):

```
sel = SwarmUtils.getSelector("Classe", "Metodo");
```

[11] Se si vuole inviare "metodo" ad un solo agente della lista occorre utilizzare il metodo `"createActionTo$message()"` al posto di `"createActionForEach$message()`.

```
modelActions.createActionForEach$message(oggetto, sel);
```

Nel caso in cui, al posto della classe di appartenenza dell'oggetto destinatario del metodo, si conosca direttamente l'indirizzo di memoria dell'oggetto, occorre inserire al posto di "Classe" il nome dell'oggetto.

Una volta costruiti tutti i gruppi di azione necessari è possibile creare lo *schedule* nel seguente modo:

```
modelSchedule = new ScheduleImpl(getZone(), number);
modelSchedule.at$createAction(0, modelActions);
modelSchedule.at$createAction(1, modelActions1);
...
modelSchedule.at$createAction(number-1,
                              modelActionsnumber-1);
```

Come si nota, creato l'oggetto *schedule*, il quale conterrà un numero di passi da eseguire ("tic" della simulazione) pari a "*number*", si inseriscono al suo interno i gruppi di azioni in numero pari a "*number*". Ad ogni istante della simulazione è eseguito almeno un gruppo di azioni (è possibile anche fare eseguire più di un gruppo per ogni "tic"). Lo stesso gruppo di azioni può essere eseguito in istanti diversi della simulazione[12], rendendo possibile la suddivisione della giornata simulata in parti, nelle quali ad ogni "tic" si ripete un determinato gruppo di azioni.

Come illustrato nel capitolo 3, sia nel *model* che nell'*observer* si crea uno *schedule*, permettendo così la separazione tra il tempo del modello di simulazione ed il tempo dell'osservatore.

Strumenti per l'osservazione della simulazione

All'interno dell'*observer* sono creati tutti gli oggetti necessari alla registrazione su *file* e alla rappresentazione grafica dei dati della simulazione. Sia i *file* che i grafici appartengono alla classe *EZGraph*. La creazione avviene nel metodo *builtObject()*, ma la serie di dati che si forma durante la simulazione è aggiornata da una specifica istruzione nello *schedule*, il quale si trova nel metodo *builtAction()*.

Per i grafici, la sintassi delle istruzioni necessarie per la creazione sono le seguenti:

[12] Ad esempio risulta possibile, attraverso un ciclo "for", inserire lo stesso gruppo di azioni in ogni istante della simulazione:

```
modelSchedule = new ScheduleImpl(getZone(), number);
for (i=0;i<number-1;i++)
        modelSchedule.at$createAction(i, modelActions);
```

```
nameGraph = new EZGraphImpl(getZone (), "Titolo", "x",
                                          "y", "Titolo2");
Globals.env.setWindowGeometryRecordName(nameGraph,
                                              "nameGraph");
nameGraph.createSequence$withFeedFrom$andSelector
("nomeSerie", oggetto, SwarmUtils.getSelector(oggetto,
                                               "metodo"));
```

Durante la creazione dei gruppi di azioni dello *schedule* dell'*observer*, occorre aggiungere ad un gruppo la seguente azione:

```
sel = SwarmUtils.getSelector(nameGraph, "step");
nameActionGroup.createActionTo$message(nameGraph, sel);
```

Con questa azione si aggiunge un dato alla serie denominata "nomeSerie", che registra il valore restituito dal metodo mandato all'oggetto specificato nel *Selector* al momento della creazione del grafico.

La stessa procedura è necessaria per la creazione di un *file* di testo contenente i valori della serie; la differenza è che, al posto della rappresentazione grafica, i dati sono scritti in un *file* e, al posto di "nomeSerie", occorre vi sia scritto "txt".

Nell'esempio sopra riportato, per aggiungere una serie al grafico, si è utilizzato il metodo della classe *EZGraph* seguente:

```
createSequence$withFeedFrom$andSelector(...)
```

In questo modo si aggiunge (al grafico o al *file*) l'intera serie di dati. Nell'elenco che segue sono riportati tre metodi necessari per rappresentare o salvare su *file* solamente i valori minimi, medi o massimi di una serie di dati:

- `createMinSequence$withFeedFrom$andSelector(...)`, ogni volta che lo *schedule* aggiorna la serie, il dato aggiunto è quello con valore minimo tra i valori seguenti la rilevazione precedente (l'ultimo dato aggiunto alla serie);
- `createAverageSequence$withFeedFrom$andSelector(...)`, ogni volta che lo *schedule* aggiorna la serie, il dato aggiunto è il valore medio dei valori seguenti la rilevazione precedente (l'ultimo dato aggiunto alla serie);
- `createMaxSequence$withFeedFrom$andSelector(...)`, ogni volta che lo *schedule* aggiorna la serie, il dato aggiunto è quello con valore massimo tra i valori seguenti la rilevazione precedente (l'ultimo dato aggiunto alla serie).

OBJECTIVEC, JAVA, SWARM E JAS: UN CONFRONTO

Una particolarità di *Objective C* consiste nell'impossibilità di utilizzare una classe se prima non è stata creata una *instance*. Concettualmente la classe è considerata come un contenitore di regole di fabbricazione degli oggetti, i quali, per essere utilizzati, devono essere prima creati. In *Java* questo non rappresenta un obbligo, infatti dichiarando una classe di tipo "*static*", questa può essere direttamente utilizzata, oppure, partendo da essa, possono essere create delle *instance*. Con la dichiarazione non statica, invece, ci si trova esattamente in una situazione analoga ad *Objective C*.

La logica di *Objective C* mette al sicuro da possibili confusioni derivanti dalla possibilità di utilizzo diretto della classe. Nel caso di utilizzo diretto, infatti, sono create le posizioni di memoria necessarie alla classe, e i metodi che sono richiamati operano sempre sulle stesse caselle. L'utilizzo delle classi di tipo "*static*" in *Java* è quindi consigliabile solamente nel caso in cui non vi sia la necessità di utilizzare più volte il codice per la creazione di diversi esemplari della classe, andando contro ad uno dei principi base della programmazione ad oggetti. Creare delle *instance* delle classi "*static*" è possibile, ma rischia di creare confusione, in particolare se si lavora a progetti di una certa dimensione.

Nella costruzione del modello *JavaSum*, per la programmazione delle classi di agenti e del *book*, è necessario utilizzare dichiarazioni dinamiche, perché deve essere possibile creare molti esemplari di agenti e deve essere possibile creare un esemplare di *book* per ogni titolo presente sul mercato simulato. La scelta di dichiarare classi non statiche, oltre a sembrare comunque la soluzione migliore in alcuni casi, è resa obbligatoria dall'utilizzo di *Swarm*. Ad esempio, il *model* e l'*observer*, che per loro natura sono unici all'interno della simulazione e quindi potrebbero essere dichiarati come classi statiche, devono obbligatoriamente, per esplicita decisione degli autori di *Swarm*, essere classi non statiche.

Le grandi differenze tra *Objective C* e *Java* si notano nelle modalità di creazione degli esemplari di una classe. Come accennato nel precedente paragrafo, in *Objective C* la creazione di una *instance* si compie attraverso tre passi:

1. la dichiarazione dell'oggetto, la quale può essere effettuata in modo generale oppure attraverso la tipizzazione statica (si veda in proposito il paragrafo relativo a *Objective C*);
2. l'assegnazione delle aree di memoria necessarie;
3. conferma ed inizializzazione delle aree di memoria.

I metodi necessari per compiere gli ultimi due passi sono il metodo *alloc* ed il metodo *init*, i quali possono essere richiamati insieme nel seguente modo:

```
nomeOggetto=[[NomeClasse alloc] init]
```

In *Swarm* (*Objective C*) questi metodi sono sostituiti dai seguenti:

- *createBegin*, destina le aree di memoria;
- *createEnd*, con questo metodo si conferma l'indirizzo dell'area di memoria precedentemente destinata alla creazione dell'oggetto e si inizializzano le caselle di memoria;
- *create*, il quale può essere utilizzato in sostituzione dei due metodi precedenti, in quanto ne riunisce le funzioni.[13]

In *Java* la creazione di oggetti avviene in modo radicalmente diverso, non esistono due metodi separati per l'assegnazione delle aree di memoria, ma le istruzioni necessarie alla costruzione della *instance* sono definite all'interno di un metodo "costruttore", richiamato attraverso l'istruzione "*new*".

In *Objective C* è permesso che le variabili possano essere tipizzate al momento dell'esecuzione del programma (*dinamic typing*), mentre in *Java* è permessa solo la tipizzazione statica. Questo aspetto limita la flessibilità di programmazione, ma permette di trovare già in fase di compilazione alcuni errori di assegnazione delle aree di memoria.

In *Java* esiste il cosiddetto *Garbage Collection*, che è un dispositivo che elimina dalla memoria gli oggetti non più utilizzati. In *Objective C* questa gestione compete direttamente al programmatore, il quale si deve preoccupare di predisporre determinate procedure per eliminare gli oggetti non utilizzati. Dalla diversa gestione della memoria effettuata in *Objective C* e in *Java* risulta una notevole differenza di velocità in fase di esecuzione. *Java* è più protettivo dal punto di vista dei possibili errori di programmazione, mentre *Objective C*, se utilizzato da un programmatore esperto, può portare a prestazioni molto superiori in termini di velocità.

Uno dei problemi tecnici affrontato in maniera diversa nei diversi ambienti di simulazione è la comunicazione di un metodo ad un oggetto, il quale a sua volta deve trasferirlo ad un secondo oggetto, che ha accesso alla classe in cui il metodo è definito. In *Swarm* ciò avviene attraverso la macro istruzione "*M()*" nella versione in *Objective C*, o con l'utilizzo dei "*Selector*" nella versione *Java*, mentre in *Jas* si utilizza la cosiddetta "*reflection*", in cui il metodo dell'agente che si vuole richiamare è ricercato attraverso una

[13] Risulta scarsamente utilizzato perché preclude la possibilità di inizializzare variabili specifiche dell'oggetto che si crea. Inoltre l'utilizzo è sconsigliato da parte degli autori di *Swarm*.

stringa alfanumerica. L'agente interpellato, se riconosce la stringa, restituisce l'indirizzo del metodo.

LE VARIE FASI DI PROGETTAZIONE E COSTRUZIONE DEL MODELLO

Il nuovo modello *JavaSUM* è progettato partendo dallo studio del modello *SUM* ora esistente. Le ragioni della costruzione del nuovo modello sono legate essenzialmente alle caratteristiche tecniche del linguaggio di programmazione *Java*, rispetto al linguaggio *ObjectiveC* (si veda paragrafo precedente). L'idea di fondo è quella di costruire un modello molto semplice, articolato secondo lo schema *ERA*, che possa in seguito essere sviluppato e arricchito di nuove parti, ma che possa essere funzionante da subito. In questo modo, è possibile verificare che le varie parti di codice che sono aggiunte siano funzionanti, ed evitare problemi in seguito.

In una prima fase, le varie parti del modello, che sono gli agenti, il *book*, il *model* e l'*observer*, possiedono solamente le funzioni essenziali. Qui di seguito sono esposte le caratteristiche strutturali che le singole parti devono possedere e come queste evolvono nelle varie fasi di studio per la costruzione del modello.

Gli agenti

Gli agenti operanti nel modello sono esemplari che derivano da diverse *classi*. Ogni *classe* possiede determinate caratteristiche, che cercano di replicare nel mondo simulato comportamenti osservabili nel mondo reale. In ogni caso, tutti gli agenti hanno in comune una serie di caratteristiche di base e una serie di regole di base che stabiliscono fondamentalmente le azioni che qualunque agente può compiere nel mercato simulato.

La versione "base" degli agenti è rappresentata dalla classe *BasicSumAgent* e presenta caratteristiche generali, che qualsiasi agente possiede. Il *BasicRuleMaster* contiene la serie di regole comuni a tutti gli agenti. Partendo da queste due *superclassi*, sono creati in seguito tutti i tipi di agenti che operano effettivamente nel mercato, i quali ereditano le caratteristiche elementari e ne hanno in più altre specifiche.

L'agente di base deve essere in grado di effettuare tre tipi di azioni alternative:

- emettere un certo numero di ordini di acquisto di quantità unitaria di un determinato titolo;

- emettere un certo numero di ordini di vendita di quantità unitaria di un determinato titolo;
- astenersi dall'operare.

L'agente, una volta richiesto al *book* l'ultimo prezzo a cui è stato concluso un contratto, deve scegliere fra i tre tipi di azioni. Ad ogni azione quindi corrisponde un livello di probabilità, determinato secondo le regole contenute nel *rule master* relativo all'agente. Nel caso l'agente decida di operare, nel *rule master* devono esservi le regole necessarie alla formulazione di un valore di prezzo limite relativo all'ordine emesso[14]. Essendo gli ordini di quantità unitaria, insieme al prezzo deve essere fissato anche il numero di ordini da emettere.

L'agente deve inoltre essere in grado di tenere la contabilità relativa alle proprie azioni e alla propria ricchezza.

La prima classe di agenti creata è denominata *RandomAgent*, a cui corrisponde il relativo *RandomRuleMaster*. Gli agenti di questo tipo presentano una struttura decisionale molto semplice e compiono azioni casuali. Gli agenti conoscono solamente il prezzo dell'ultimo contratto concluso sul mercato, scelgono a caso se emettere ordini di acquisto o di vendita, ed è casuale anche la determinazione del numero di contratti da emettere (l'agente è a conoscenza solo del numero massimo di contratti che può emettere). Il limite di prezzo è ricavato moltiplicando un coefficiente casuale, al prezzo conosciuto (prezzo dell'ultimo contratto concluso nel *book*).

Nei paragrafi seguenti sono trattate le funzioni del *book*, del *model* e dell'*observer*, che permettono agli agenti di operare nel mercato simulato.

Il book

L'esigenza primaria del mercato simulato è quella di poter inserire diversi titoli su cui gli agenti possano operare. La struttura informatica del *book* quindi deve essere tale da permettere l'esistenza di tanti *book* quanti sono i titoli negoziabili sul mercato.

Il *book* è il semplice registro degli ordini di acquisto o di vendita relativi ad un titolo. Attraverso adeguati controlli, il *book* combina tra loro i vari ordini compatibili, e concude di contratti. Esso si presenta come un oggetto "in attesa", i cui metodi sono attivati dai vari agenti. Ad esempio, un agente attiva un metodo del *book* per l'inserimento di un ordine di acquisto

[14] Per prezzo limite si intende il livello di prezzo massimo a cui l'agente è disposto a concludere un contratto.

o di vendita ad un determinato prezzo limite, oppure per la richiesta dell'ultimo prezzo a cui sono stati conclusi dei contratti.

La procedura seguita nel *book* è simile ad una normale asta "a chiamata": quando nel *book* sono presenti ordini di acquisto e di vendita, ogni ordine è soddisfatto nel modo più favorevole per l'agente che lo ha emesso. Quindi gli ordini di vendita sono inseriti nel *book* in ordine crescente di prezzo, mentre gli ordini di acquisto sono ordinati in ordine decrescente di prezzo. Nel caso in cui un ordine non sia interamente soddisfatto, rimane inserito nel *book* un ordine dello stesso tipo per la quantità rimanente, finché non è trovata una contropartita[15].

Il prezzo del titolo negoziato[16] varia ad ogni istante della simulazione ed è determinato in base alle condizioni a cui sono stati conclusi i contratti.

Per il *book* non è previsto alcun tipo di contabilità. Nella fase iniziale esso svolge semplicemente il compito di registro e combinatore di ordini. Lo sviluppo immediatamente successivo della struttura del *book* è l'aggiunta delle aste di apertura e chiusura, in modo da riprodurre le procedure realmente seguite nel mercato. La *classe book* non eredita caratteristiche da classi più generali, come nel caso degli agenti, ma è utile che vi sia la possibilità di creare degli "esemplari" di *book*, in modo da poter negoziare sul mercato simulato più titoli (ad ogni titolo corrisponderà il proprio *book*). Ciò è realizzabile grazie all'utilizzo di una classe *book* dinamica.

Il model

Il *model* inizialmente possiede una struttura molto semplice. La caratteristica fondamentale è quella di poter avviare le simulazioni scegliendo tra due modalità. La prima deve riprodurre esattamente il funzionamento del vecchio modello di borsa. Il *model*, nella fase di programmazione delle fasi della simulazione, deve ordinare la lista di agenti ed interpellarli nell'ordine dato dalla lista, per permettere loro di operare. La lista è ordinata alla conclusione di ogni ciclo, ossia quando hanno operato (o deciso di non operare) tutti gli agenti della lista, e ad ogni "tic" della simulazione opera un solo agente. La seconda opzione di simulazione deve permettere che, ad ogni periodo, possa operare più di un agente. E' quindi necessario che il *model* riordini la lista di agenti ed intervisti tutti gli agenti nell'ordine dato dalla lista, ad ogni "tic". Con questa modalità di

[15] Gli agenti inseriscono sempre ordini unitari, quindi la "quantità di titoli" di un ordine è da intendersi come quantità di ordini emessi per lo stesso titolo. La scelta di considerare il numero di ordini al posto di quantità di titoli, rende più semplice la gestione dei casi in cui nel *book* non vi siano ordini contrapposti della stessa quantità.

[16] Inizialmente il modello contiene un solo titolo negoziabile, quindi un solo "esemplare" di *book*.

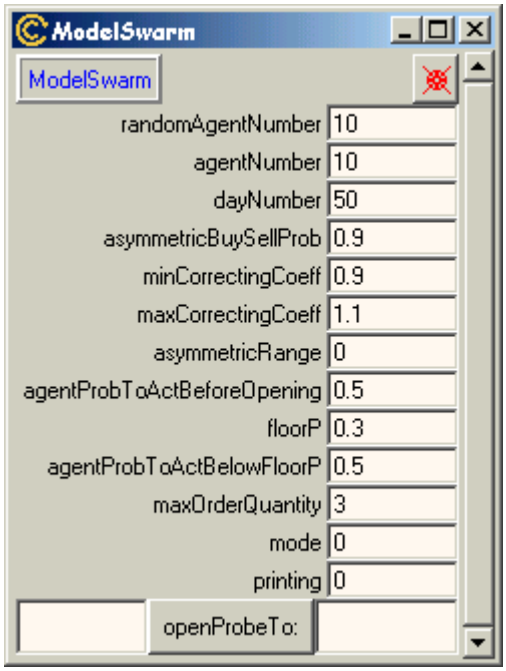
funzionamento, aumenta la plausibilità del modello pur escludendo la possibilità che vi siano inserimenti contemporanei di ordini[17].

Con questa possibilità di opzione, sarà possibile verificare le differenze con il vecchio modello e vedere se ci sono dei miglioramenti dovuti al maggiore realismo.

INNOVAZIONI TECNICHE DEL MODELLO

In *JavaSum*, per quanto riguarda il *model* e l'*observer*, sono riprodotte tutte le funzionalità offerte dal modello SUM in *Objective C*. La differenza principale è la presenza di una sola famiglia di agenti, i *RandomAgent*. In futuro potranno essere create diverse tipologie di agenti e potrà essere complicata la struttura del *book*.

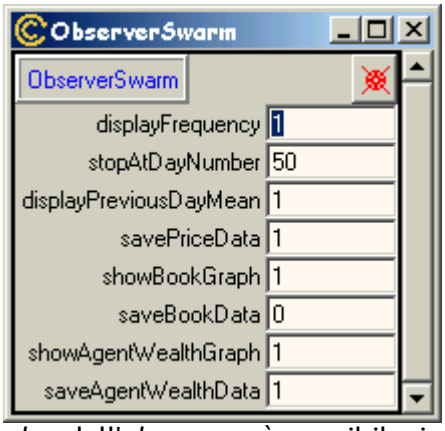
La *probe* del *model* si presenta nel seguente modo:



Attraverso la *probe* del *model* sono impostati tutti i parametri della simulazione. Inoltre è possibile scegliere la "modalità" in cui la simulazione deve essere effettuata (si veda paragrafo relativo). Una volta avviata la simulazione, è possibile aprire la sonda su un singolo agente, specificandone il numero identificativo in corrispondenza del tasto "*openProbeTo*:"

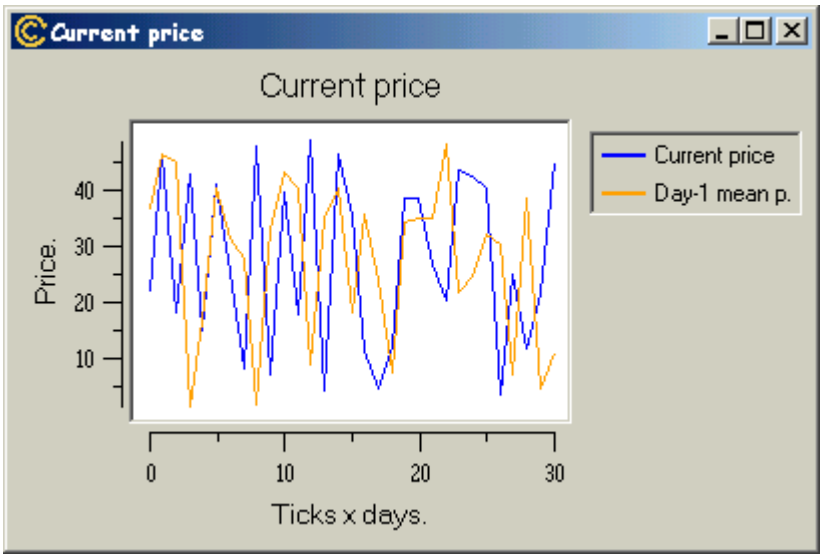
[17] Tale ipotesi, nella realtà, risulta poco probabile. E' quasi impossibile, infatti, che due agenti reali riescano ad inserire un ordine esattamente nello stesso istante.

La *probe* dell'*observer* è invece la seguente:

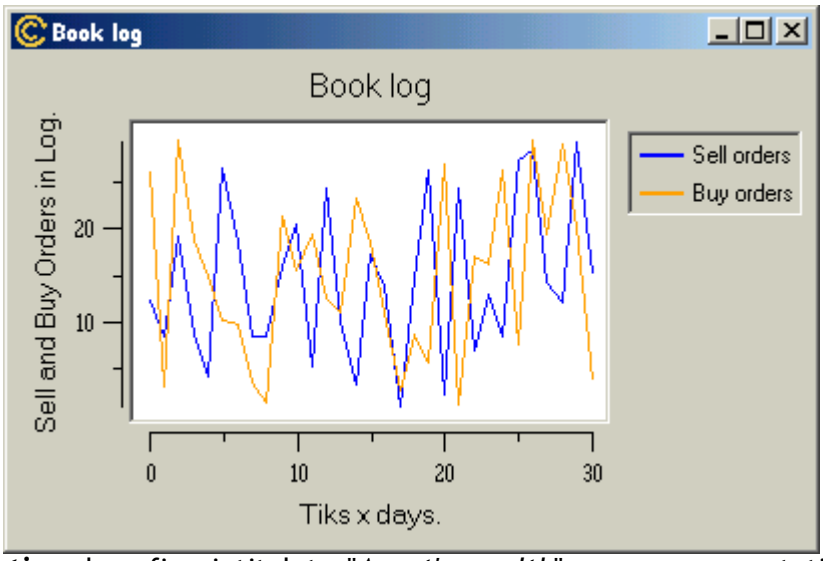


Attraverso la *probe* dell'*observer*, è possibile inserire delle scelte di visualizzazione e salvataggio dei dati. Le diverse possibilità sono illustrate di seguito (i dati rappresentati in figura sono serie casuali).

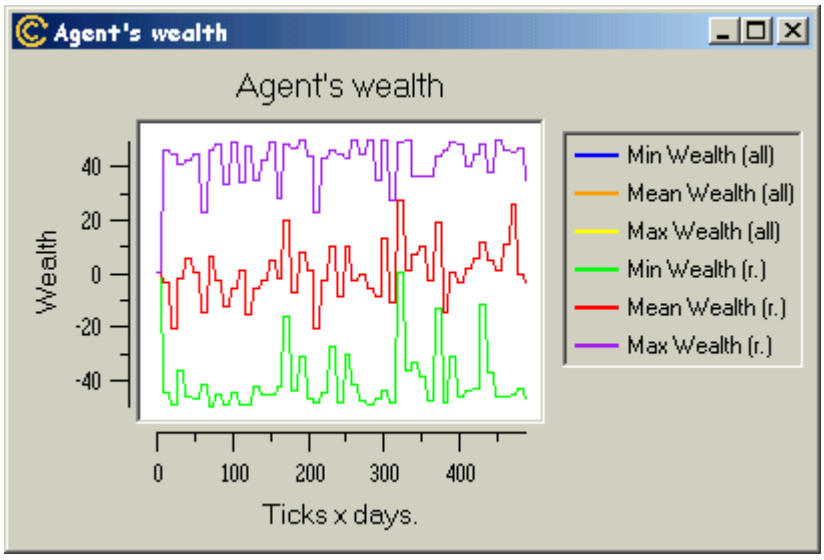
- ***Prezzi***: come si può notare nel grafico riportato qui di seguito, sono rappresentate due serie di dati, che sono rispettivamente "*Current Price*", la quale è formata dai valori di prezzo corrente del titolo negoziato, e "*Day-1 mean p.*", la quale è formata dai valori medi di prezzo calcolati sui valori della giornata precedente. Quest'ultima serie è rappresentata e salvata solo se la variabile "*displayPreviousDayMean*" dell'*observer* ha valore 1.



- ***Book***: possono essere salvate e rappresentate le serie contenenti il numero di ordini di acquisto e di vendita. Siccome gli agenti inseriscono ordini di quantità unitaria, il numero degli ordini è pari alla quantità richiesta in acquisto o in vendita del titolo negoziato.



- ***Agenti***: nel grafico intitolato "*Agent's wealth*" sono rappresentati i valori di ricchezza minimi, massimi e medi relativi ad ogni singola famiglia di agenti e all'intero insieme degli agenti (in legenda le serie sono contraddistinte da "all."), che per ora, vista la presenza dei soli *RandomAgent*, coincidono. Oltre alla visualizzazione, è possibile salvare i dati su un *file*.



Una piccola differenza tecnica rispetto al precedente modello consiste nella mancanza di un'*array* di agenti, la quale è stata sostituita da una lista (*indexAgentList*). Su questa lista, che non è mai riordinata durante la simulazione, è stato costruito un indice (*indexAgentList*) attraverso il quale

è possibile ritrovare l'indirizzo di memoria di un agente in base al proprio numero identificativo, in maniera molto veloce, esattamente come avveniva nel vecchio modello con l'*array* e il relativo indice. L'utilità della lista contenente gli agenti nell'ordine di creazione rende possibile aprire una sonda relativa al singolo agente, in seguito all'inserimento nella *probe* dell'*observer* del numero identificativo dell'agente.

CurrentAgent

La classe *CurrentAgent* riproduce esattamente le funzionalità della classe in *Objective C*. Il codice relativo alla classe è il seguente:

```
import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;
public class CurrentAgent extends SwarmObjectImpl
{
    ListImpl agentList;
   // Constructor
    public CurrentAgent(Zone aZone)
    {
     super(aZone);
    }
    public void setAgentList(ListImpl l)
    {
    agentList = l;
    }
    public void act1()
    {
    BasicSumAgent actingAgent;
    actingAgent=
          (BasicSumAgent)agentList.removeFirst();
    agentList.addLast( actingAgent );
    actingAgent.act1();
    }
}
```

La funzione di un oggetto di tipo *CurrentAgent* è di rimuovere il primo agente di una lista, attivarne il metodo denominato "act1" e aggiungere lo stesso agente al termine della lista. Attivando un *CurrentAgent* su una lista più volte di seguito si ottiene come risultato lo scorrimento verso l'alto degli

elementi della lista. Grazie ai *CurrentAgent* è stato possibile apportare una modifica di grande rilievo al modello di simulazione, ossia permettere a più agenti di operare nello stesso istante ("tic" della simulazione). Nel paragrafo seguente sono illustrati gli aspetti tecnici relativi a questo aspetto.

Due modalità di funzionamento

Come accennato nel paragrafo precedente, la differenza più rilevante rispetto al vecchio modello, riguarda il *model*. Infatti in *JavaSum* è possibile gestire due tipologie operative di simulazione. L'utilizzatore del modello può decidere se mantenere l'impostazione "tradizionale", ossia l'impostazione di base del vecchio modello di simulazione, nel quale, nella fase centrale della giornata borsistica simulata, può operare un solo agente per "tic", oppure optare per un maggiore realismo dato dalla possibilità di fare operare più agenti nello stesso istante della simulazione.

L'inserimento della scelta avviene attraverso la probe del *model* in cui, in corrispondenza della casella relativa alla variabile denominata "*mode*", può essere inserito un valore numerico. Quando il valore della variabile "*mode*" è uguale a 1, il modello opera con la "nuova" modalità.

A livello di codice informatico, nel metodo *builtObject()* del *model* è costruita una lista di *CurrentAgent*, tutti con caratteristiche uguali al *CurrentAgent* (*theCurrentAgent*) utilizzato nella modalità tradizionale. Nel metodo *builtAction()*, in base al valore assunto dalla variabile *mode*, il gruppo di azioni, relativo alle operazioni da effettuare nei "tic" della fase centrale della giornata borsistica simulata (*modelAction2*), si costruisce in modo diverso. Nel caso in cui *mode* sia uguale a 1, *modelAction2* è composto dalle due seguenti operazioni:

1. riordino della lista generale degli agenti (*agentList*);
2. attivazione del metodo "act1" per ogni elemento della lista di *CurrentAgent*.

Nel caso in cui si mantenga la vecchia modalità, il *modelAction2* è composto da un sola operazione, ossia il richiamo del metodo "act1" di un esemplare di *CurrentAgent* (*theCurrentAgent*), il quale ad ogni "tic" risulta collegato ad un diverso agente della lista.

Lo *schedule* rimane lo stesso per le due modalità ma, se si sceglie la nuova modalità di funzionamento, ad ogni istante della fase centrale della simulazione la lista generale degli agenti è riordinata e ogni *CurrentAgent* della lista risulta collegato ad un diverso agente. Le azioni compiute dal singolo *CurrentAgent* rimangono quelle descritte in precedenza: con una lista di *n CurrentAgent*, ognuno dei quali, se interpellato, effettua lo spostamento di un agente della lista (contenente n agenti di vario tipo) dalla

prima all'ultima posizione, attivandone il metodo "act1", comporta il richiamo del metodo "act1" di tutti i componenti della lista generale degli agenti. In questo modo, ogni agente ha la possibilità di inserire ordini in ogni istante, senza che però si verifichino inserimenti contemporanei: gli ordini sono inseriti dagli agenti nell'ordine dato dalla lista.

Il metodo buitAction() del modello JavaSum è il seguente:

```
public Object buildActions()
{
Selector sel;
int i;
ListShufflerImpl listShuffler
        = new ListShufflerImpl(getZone());
super.buildActions();
//primo gruppo di azioni (compiute nella prima fase della
//giornata)
modelActions1 = new ActionGroupImpl(getZone());
sel = SwarmUtils.getSelector(this,
         "increaseCurrentDayNumber");
modelActions1.createActionTo$message(this, sel);
sel = SwarmUtils.getSelector("Book", "setClean");
modelActions1.createActionTo$message(theBook, sel);
sel = SwarmUtils.getSelector(listShuffler,
         "shuffleWholeList");
modelActions1.createActionTo$message(listShuffler,
         sel, agentList);
sel = SwarmUtils.getSelector("BasicSumAgent", "act0");
modelActions1.createActionForEach$message(agentList, sel);
//secondo gruppo di azioni, creato in funzione di "mode"
modelActions2 = new ActionGroupImpl(getZone());
if (mode==1)
  {
  sel = SwarmUtils.getSelector(listShuffler,
              "shuffleWholeList");
  modelActions2.createActionTo$message(listShuffler,
              sel, agentList);
  sel = SwarmUtils.getSelector("CurrentAgent", "act1");
  ModelActions2.
```

```
            createActionForEach$message(currentAgentList,
                                                    sel);
    }
  else
    {
    sel = SwarmUtils.getSelector("CurrentAgent", "act1");
    modelActions2.createActionTo$message(theCurrentAgent,
                sel);
    }
  //terzo gruppo di azioni (compiute a fine giornata)
  modelActions3 = new ActionGroupImpl(getZone());
  sel = SwarmUtils.getSelector("Book", "setMeanPrice");
  modelActions3.createActionTo$message(theBook, sel);
  sel = SwarmUtils.getSelector("BasicSumAgent", "act2");
  modelActions3.createActionForEach$message(agentList, sel);
  modelSchedule = new ScheduleImpl(getZone(), agentNumber);
  //lo schedule
  modelSchedule.at$createAction(0, modelActions1);
  for (i=0;i<agentNumber;i++)
           modelSchedule.at$createAction(i, modelActions2);
  modelSchedule.at$createAction(agentNumber-1,
           modelActions3);
  return this;
  }
```

VERSIONE FINALE DEL MODELLO

Al fine di ottenere un maggiore livello di realismo del modello, si sono apportate ulteriori modifiche alla struttura di base della simulazione. Con le modifiche descritte nei paragrafi precedenti, rispetto al modello *SUM*, si è aggiunta la possibilità di fare operare tutti gli agenti ad ogni istante della simulazione. Questa scelta, se da un lato amplia notevolmente le possibilità di sperimentazione offerte dal modello, dall'altro potrebbe apparire poco elastica. Si è quindi deciso di modificare ulteriormente il *model*, affinché si possa fare operare un certo numero di agenti ad ogni istante della simulazione e non ci si ponga di fronte la scelta estrema "uno o tutti".

Il nuovo modello rende possibili le seguenti quattro possibilità di sperimentazione:

- fare operare un solo agente ad ogni istante della simulazione;
- fare operare tutti gli agenti ad ogni istante della simulazione;
- fare operare una parte fissa del totale degli agenti ad ogni istante della simulazione;
- fare operare un numero variabile di agenti ad ogni istante della simulazione.

Come si può notare, le prime due possibilità elencate sono già offerte dalla prima versione di *JavaSum*, mentre le ultime due costituiscono l'innovazione apportata al modello, che permette di effettuare simulazioni con una certa percentuale di agenti operanti ad ogni istante simulato. Questa quota di agenti può essere fissa o rappresentare un massimo, quando in ogni istante opera un numero di agenti diverso, ma che non supera mai una certa percentuale.

Nei prossimi paragrafi si analizzano le particolarità tecniche che hanno permesso la costruzione della versione finale del modello. Infine si riassumono le principali differenze di *JavaSum* rispetto al modello *SUM* in *Objective C*.

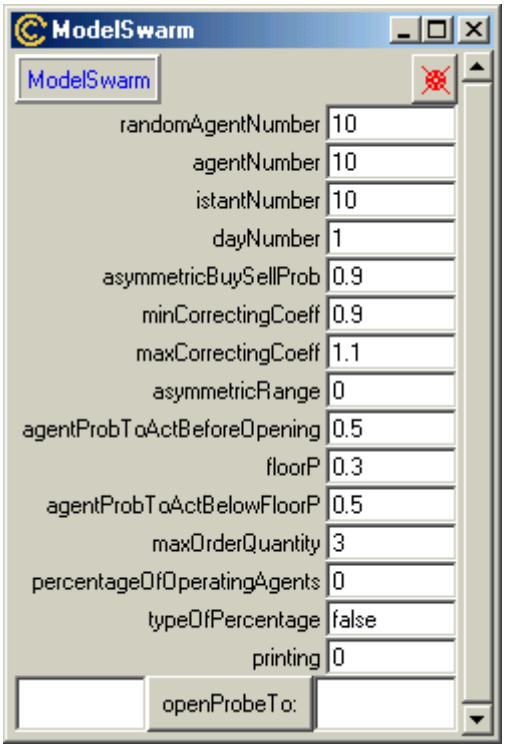
Il nuovo model

Nel *model* sono state introdotte due nuove variabili ed è stata eliminata la variabile "*mode*".

Le nuove variabili sono:

- *percentageOfOperatingAgent*, l'utente specifica la percentuale del totale degli agenti che desidera operino durante la simulazione;
- *typeOfPercentage*, può assumere valore "true" o "false" a seconda che si intenda considerare *percentageOfOperatingAgent* come una percentuale massima o fissa di agenti.

La probe del *model* si presenta come segue:



In ogni caso il numero di agenti che effettivamente inserisce degli ordini nel *book* dipende anche dalla decisione di operare o no dei singoli agenti. Inoltre, per ragioni tecniche, il numero di agenti calcolato in base alla percentuale subisce sempre una approssimazione per difetto[18].

Nel metodo *builtObject()* del *model* è calcolato il numero massimo di agenti che operano in ogni istante con le seguenti istruzioni:

```
maxNumberOfOperatingAgents =
        percentageOfOperatingAgents*agentNumber/100;
if (percentageOfOperatingAgents == 0)
        maxNumberOfOperatingAgents = 1;
```

Durante la simulazione, se *typeOfPercentage* ha valore *false* il numero degli agenti operanti per ogni istante è pari al numero massimo di agenti, mentre se ha valore *true*, il numero di agenti che opera è casuale e compreso tra zero e il numero massimo di agenti.

La classe CurrentIstant

[18] Il numero di agenti è calcolato moltiplicando il numero totale di agenti per il valore percentuale inserito dall'utente. In seguito il prodotto è diviso per 100. La divisione tra due valori interi, in linguaggio *Java*, restituisce un risultato intero, troncando la parte decimale del quoziente. La semplificazione è stata effettuata per evitare rallentamenti della simulazione dovuti all'utilizzo di una funzione che approssimasse il numero di agenti calcolato arrotondandolo all'intero più vicino. Tuttavia l'errore di approssimazione che ne deriva è accettabile, perché solitamente il modello è utilizzato con un numero di agenti molto elevato.

Per rendere variabile il numero di agenti operanti ad ogni istante della simulazione, occorre che il richiamo di un *CurrentAgent* avvenga un numero variabile di volte all'interno di un gruppo di azioni dello *schedule*. Essendo lo *schedule* un "oggetto" che è creato all'inizio della simulazione e che non è più modificabile in seguito, per introdurre la "variabilità" all'interno di esso occorre nascondere all'interno di un oggetto la parte "variabile" che ci interessa sia richiamata durante la simulazione. In questo modo è possibile mandare un messaggio un numero fisso di volte dallo *schedule* a questo oggetto.

L'oggetto *CurrentInstant* segue la logica appena illustrata. Al suo interno si calcola il numero di agenti che devono operare in un determinato istante in base alla percentuale specificata dall'utilizzatore della simulazione. In seguito un ciclo è eseguito un numero di volte pari al numero di agenti che devono operare, all'interno del quale si richiama il *CurrentAgent*. Questo a sua volta risulta collegato ogni volta ad un diverso agente della lista ordinata a caso.

Nello *schedule* è incluso un gruppo di azioni che manda un messaggio all'esemplare di *CurrentIstant*, attivandone il metodo che fa scattare il ciclo di richiamo degli agenti. In questo modo la struttura dello *schedule* rimane fissa, ma è possibile ottenere la programmazione di giornate simulate diverse tra di loro.

Il *CurrentAgent* presenta un funzionamento molto simile. Con il richiamo di un esemplare di *CurrentAgent* è possibile, nonostante la struttura fissa dello *schedule*, richiamare agenti diversi in ogni istante della giornata e soprattutto rendere tra loro differenti le giornate. Infatti, se si volessero richiamare agenti diversi per ogni istante della giornata, nulla impedirebbe di specificare direttamente nello *schedule* quali agenti richiamare e in quali istanti. Il risultato sarebbe poco utile in quanto le giornate sarebbero tutte programmate in base allo *schedule*, e quindi nel tempo la sequenza degli agenti richiamati sarebbe la stessa ogni giorno. Nascondendo il richiamo di un agente all'interno di un oggetto è invece possibile richiamare l'agente al momento in cui effettivamente si svolge la simulazione. Lo stesso avviene per il numero degli agenti da richiamare: solo durante la simulazione è calcolato in numero di cicli che devono essere eseguiti.

La classe *CurrentIstant* (si veda in allegato) contiene una serie di metodi necessari all'assegnazione di valori alle variabili interne che sono:

a) la percentuale degli agenti che devono operare (*percentageOfOperatingAgents*);

b) il tipo di percentuale (*typeOfPercentage*);

c) numero massimo di agenti che devono operare (*maxNumberOfOperatingAgents*), che serve come limite massimo per il numero di agenti se *typeOfPercentage* ha valore "*true*";

d) l'indirizzo di memoria del *CurrentAgent* che è richiamato (*theCurrentAgent*).

Il metodo richiamato nello *schedule* è denominato "*callAgents*" e si presenta come segue:

```
public void callAgents()
{
int j;
numberOfOperatingAgents = maxNumberOfOperatingAgents;
if (typeOfPercentage == true)
   {
   numberOfOperatingAgents=
       Globals.env.uniformIntRand.getIntegerWithMin$withMax
         (0 , maxNumberOfOperatingAgents);
   if (percentageOfOperatingAgents == 0)
       numberOfOperatingAgents = 1;
   }
for (j=0;j<numberOfOperatingAgents;j++)
   theCurrentAgent.act1();
}
```

Come si può notare dal codice, il numero di agenti è nuovamente calcolato solo se l'utente inserisce "*true*" come tipo di percentuale, ed il ciclo di richiamo degli agenti è eseguito esattamente un numero di volte pari al numero degli agenti.

Il nuovo schedule

La struttura della simulazione risulta notevolmente diversa da quella progettata in partenza. Infatti, dal metodo *builtObject()* del *model* scompare la lista di *CurrentAgent* che si utilizzava per fare operare l'intera lista di agenti, e il numero massimo di agenti che devono operare è calcolato in base ai dati inseriti dall'utente. Questo valore è ricalcolato durante lo svolgimento della simulazione se si fa operare un numero di agenti variabile ad ogni istante simulato.

Per la costruzione dello *schedule* si creano quattro gruppi di azioni:

- *modelAction1*, all'interno del quale si richiama il metodo *setClean* del *book*, che ripulisce il book dagli ordini in sospeso, si riordina la lista e si richiama il metodo *act0* degli agenti che possono, in base alle proprie regole, inserire ordini nel *book*;
- *modelActionLS*, in cui si rimescola la lista degli agenti;
- *modelAction2*, in cui si richiama il metodo *callAgents* di *CurrentIstant* per permettere agli agenti di operare (con il richiamo del metodo *act1* degli agenti tramite il *CurrentAgent*);
- *modelAction3*, in cui si richiede al *book* di calcolare il prezzo medio di conclusione dei contratti e agli agenti di attivare il metodo *act2* per la contabilità di fine giornata, e si incrementa il numero del giorno della simulazione.

Il nuovo metodo *builtAction()* del *model* è il seguente:

```
public Object buildActions()
  {
  Selector sel;
  int i, j;
  ListShufflerImpl listShuffler
       = new ListShufflerImpl(getZone());
  super.buildActions();
  // Creazione dei gruppi di azioni della simulazione.
  modelActions1 = new ActionGroupImpl(getZone());
  sel = SwarmUtils.getSelector("Book", "setClean");
  modelActions1.createActionTo$message(theBook, sel);
  sel = SwarmUtils.getSelector(listShuffler,
       "shuffleWholeList");
  modelActions1.createActionTo$message(listShuffler, sel,
       agentList);
  sel = SwarmUtils.getSelector("BasicSumAgent", "act0");
  modelActions1.createActionForEach$message(agentList,
       sel);
  // Si rimescola lista di agenti.
  modelActionsLS = new ActionGroupImpl(getZone());
  sel = SwarmUtils.getSelector(listShuffler,
       "shuffleWholeList");
  modelActionsLS.createActionTo$message(listShuffler, sel,
```

```
        agentList);
    // Le azioni degli agenti nel mercato.
    modelActions2 = new ActionGroupImpl(getZone());
    sel = SwarmUtils.getSelector("CurrentIstant",
        "callAgents");
    modelActions2.createActionTo$message(theCurrentIstant,
        sel);
    // Contabilità degli agenti a fine giornata.
    modelActions3 = new ActionGroupImpl(getZone());
    sel = SwarmUtils.getSelector("Book", "setMeanPrice");
    modelActions3.createActionTo$message(theBook, sel);
    sel = SwarmUtils.getSelector("BasicSumAgent", "act2");
    modelActions3.createActionForEach$message(agentList,
        sel);
    sel = SwarmUtils.getSelector(this,
        "increaseCurrentDayNumber");
    modelActions3.createActionTo$message(this, sel);
    // Creazione dello schedule.
    modelSchedule = new ScheduleImpl(getZone(),
        istantNumber);
    modelSchedule.at$createAction(0, modelActions1);
    for (i=0;i<istantNumber;i++)
        {
        modelSchedule.at$createAction(i, modelActionsLS);
        modelSchedule.at$createAction(i, modelActions2);
        }
    modelSchedule.at$createAction(istantNumber-1,
        modelActions3);
    return this;
    }
```

Secondo lo *schedule*, la giornata simulata ha una durata pari a *istantNumber* e ad ogni istante è rimescolata la lista di agenti ed è eseguito il *modelAction2*. Nel primo e nell'ultimo istante della giornata si richiamano rispettivamente il *modelAction1* e il *modelAction3*.

Riepilogo delle innovazioni di JavaSum

Nel vecchio modello *SUM*, la simulazione era legata al numero di agenti che si intendeva fare interagire. La giornata borsistica era conclusa quando a tutti gli agenti era stato chiesto di operare. Nel nuovo modello questa restrizione è scomparsa e il numero di istanti di cui è composta una giornata non è più legato al numero di agenti creati nel modello. L'utilizzatore della simulazione può inserire liberamente il numero di istanti nella *probe* del *model*, in corrispondenza della variabile *IstantNumber*.

A differenza di *SUM*, non vi sono restrizioni per l'operatività degli agenti, i quali possono inserire ordini in qualsiasi istante anche se li hanno già inseriti nel primo istante della giornata, durante il quale non vi è la conclusione di alcun contratto, ma si permette l'inserimento di ordini per poter iniziare la giornata con un *book* non completamente vuoto. Inoltre la lista di agenti è ordinata ad ogni istante della simulazione. Queste novità aprono la possibilità per ogni agente di inserire più ordini durante la stessa giornata e spetterà al *book* permettere la combinazione di ordini che provengono dal medesimo agente.

Lo spostamento dell'incremento del numero della giornata dal gruppo di azioni *modelAction1* al gruppo *modelAction3* dello *schedule* ha risolto un piccolo problema di *SUM* che rendeva poco intuitivo il funzionamento della variabile *stopAtDayNumber*, con cui si stabilisce l'interruzione della simulazione. In *JavaSum*, stabilita la durata dell'esperimento, la simulazione è interrotta all'inizio della giornata specificata. Con il tasto "*Start*" è comunque possibile mandare avanti la simulazione per un numero di istanti pari a *displayFrequency*. Questo permette di far proseguire a passi la simulazione andando avanti istante per istante, se *displayFrequency* è uguale a 1, oppure giornata per giornata (o multiplo di giornata) se si inserisce un valore di *displayFrequency* uguale al numero di istanti per giornata (o un multiplo di esso). In *SUM*, siccome l'incremento del numero della giornata è inserito nel primo gruppo di azioni (*modelAction1*), la simulazione si interrompe alla fine della giornata specificata nella variabile *stopAtDayNumber*, se la variabile *displayFrequency* ha un valore pari al numero di agenti operanti, ma se *displayFrequency* è pari a 1 la simulazione si interrompe alla fine del primo istante della giornata seguente quella specificata dall'utente della simulazione[19].

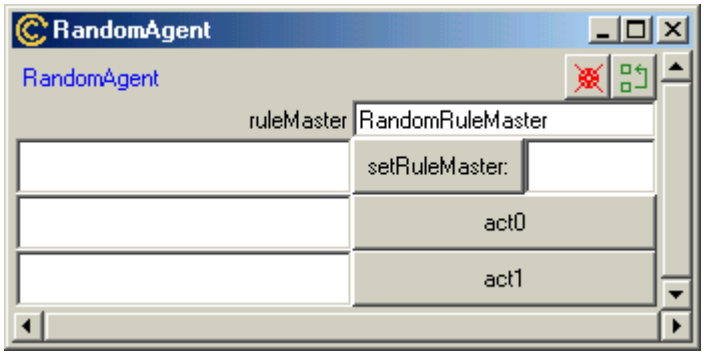
Nell'ultima versione di *JavaSum* vi è una piccola differenza per l'apertura delle sonde sui singoli agenti. In *BasicSumAgent* scompare il

[19] Ciò succede perché i controlli per l'interruzione della simulazione sono effettuati nell'*observer*, il cui *schedule* è governato da *displayFrequency*. Quando *displayFrequency* è uguale a 1, alla fine di ogni istante è effettuato il controllo e siccome l'incremento del numero di giornata è effettuato nel primo istante, la simulazione si interrompe quando le azioni del primo istante della giornata sono già compiute.

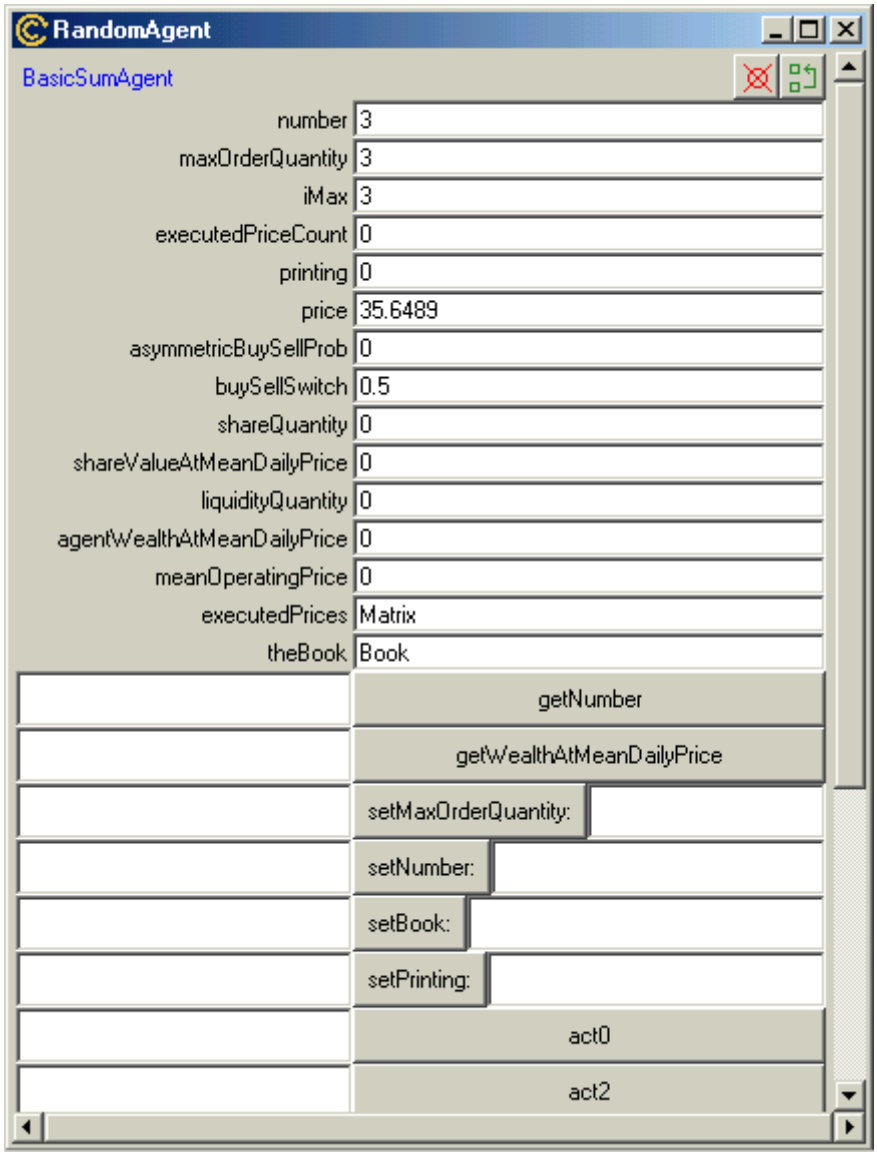
metodo *getProbe()* e, all'interno del metodo *openProbeTo()* del *model*, una volta ritrovato l'indirizzo dell'agente a cui si è interessati, si crea la sonda. Tutte le variabili che sono dichiarate "*public*" diventano visibili dalla sonda che si apre specificando il numero identificativo dell'agente nella *probe* del *model*, con la possibilità di aumentare il dettaglio della sonda nel seguente modo:

I. all'apertura, la sonda si presenta come in figura;

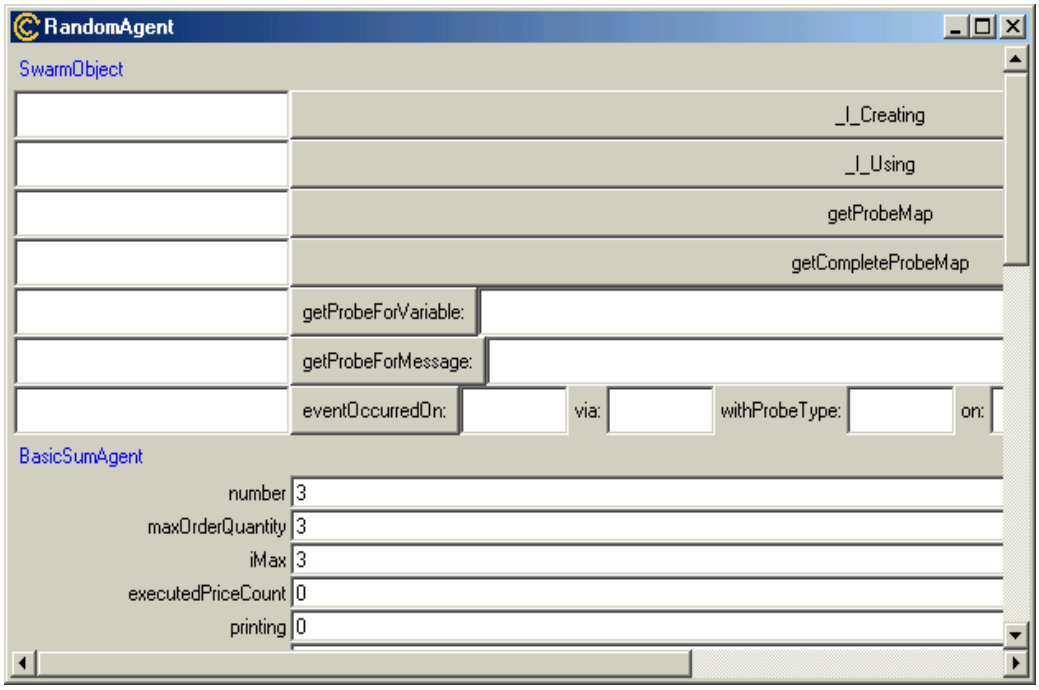
II. cliccando con il tasto destro del mouse sulla scritta "*RandomAgent*" (o altra scritta indicante la classe dell'agente su cui si è aperta la sonda) si visualizzano le *instance variable* della tipologia a cui l'agente appartiene (nel caso in figura *RandomAgent*, unica famiglia di agenti per ora presente nel modello);



III. con il tasto  si accede alle variabili della classe da cui discende l'agente (*BasicSumAgent*);


RandomAgent
BasicSumAgent
number 3
maxOrderQuantity 3
iMax 3
executedPriceCount 0
printing 0
price 35.6489
asymmetricBuySellProb 0
buySellSwitch 0.5
shareQuantity 0
shareValueAtMeanDailyPrice 0
liquidityQuantity 0
agentWealthAtMeanDailyPrice 0
meanOperatingPrice 0
executedPrices Matrix
theBook Book
getNumber
getWealthAtMeanDailyPrice
setMaxOrderQuantity:
setNumber:
setBook:
setPrinting:
act0
act2

IV. ripetendo l'operazione descritta al punto precedente, si giunge alla visualizzazione delle variabili della classe da cui discende la classe *BasicSumAgent*, che è *SwarmObject*.



L'ultima novità introdotta in *JavaSum* è l'utilizzo dei *file* "*scm*" per la lettura di parametri da un *file* esterno al modello. In questo modo è possibile modificare i parametri di partenza del *model* e dell'*observer* senza necessariamente ricompilare l'intero modello, ma semplicemente andando a modificare i valori contenuti nel *file JavaSum.scm*. I parametri sono letti al momento della creazione degli esemplari della classe *Observer* e *Model*, tramite il richiamo del costruttore delle classi in cui si specifica l'esistenza del *file* "*scm*". Il codice del *file*, riportato sotto, ha una sintassi molto simile al linguaggio di programmazione *Lisp*.

```
(list
 (
  cons 'observerSwarm
       (
  make-instance 'ObserverSwarm
                        #:displayFrequency 1
                        #:stopAtDayNumber 4
                        #:displayPreviousDayMean 1
                        #:savePriceData 1
                        #:showBookGraph 1
                        #:saveBookData 0
```

```
                        #:showAgentWealthGraph 1
                        #:saveAgentWealthData 1
                        )
            )
        (
         cons 'modelSwarm
              (
         make-instance 'ModelSwarm
                        #:randomAgentNumber 10
                        #:istantNumber 10
                        #:asymmetricBuySellProb 0.9
                        #:minCorrectingCoeff 0.9
                        #:maxCorrectingCoeff 1.1
                        #:asymmetricRange 0.0
                        #:agentProbToActBeforeOpening 0.5
                        #:floorP 0.3
                        #:agentProbToActBelowFloorP 0.5
                        #:maxOrderQuantity 3
                        #:percentageOfOperatingAgents 0
                        #:typeOfPercentage #f
                        #:printing 0
                        )
            )
        )
```

ALLEGATO A

STUDIO DEL CODICE *OBJECTIVE C* DEL MODELLO *SUM*

MAIN

```
// Pietro Terna, Universita' di Torino, Dipartimento di Scienze economiche
// e finanziarie, corso Unione Sovietica 218bis, 10134 Torino, Italy
// terna@econ.unito.it

// this code requires Swarm 2.1.1 or 2.2

// sum project starting date:      December  1999
// this version:                   December  2001


#import <simtools.h>
#import "ObserverSwarm.h"

int main (int argc, const char **argv)
{
  ObserverSwarm *observerSwarm;

  initSwarmApp (argc, argv,"0.66","pietro.terna@unito.it");

// Below we have the standard main sequence of any Swarm model:
// 1. creation of the observer; 2. objects creation (one is the Swarm model
// we are dealing with, see ObserverSwarm.m); 3. schedule creation;
// 4. activation of 2. and 3.; 5. go
```

Le parti in grigio non sono ricostruite in JavaSum.

Dichiarazione dell'oggetto observer.

```
  observerSwarm =  [ObserverSwarm create: globalZone];
  SET_WINDOW_GEOMETRY_RECORD_NAME (observerSwarm);
  [observerSwarm buildObjects];
  [observerSwarm buildActions];
  [observerSwarm activateIn: nil];

  [observerSwarm go];

  return 0;

}
```

La simulazione si avvia con la creazione dell'observer e il richiamo dei suoi metodi.

MODELSWARM.H

```
#import <objectbase/Swarm.h>
#import <simtools.h>      // necessary to invoke ObjectLoader
#import <objectbase.h>    // needed by <ProbeMap> in ModelSwarm.m
#import <activity.h>
#import <collections.h>

  // agentForge step 2
#import "BasicSumAgent.h"
#import "RandomAgent.h"
#import "MarketImitatingAgent.h"
#import "LocallyImitatingAgent.h"
#import "StopLossAgent.h"
#import "ANNForecastAppAgent.h"
#import "ForecastingAgent.h"
#import "CurrentAgent.h"
#import "BPCTAgentA.h"
#import "BPCTAgentAInterface.h" // related to the agents used in our simulation
#import "BPCTAgentB.h"
#import "BPCTAgentBInterface.h" // related to the agents used in our simulation
#import "BPCTPriceRuleMaster.h"

#import "Book.h"
#import "RandomRuleMaster.h"
#import "StopLossRuleMaster.h"
#import "SimpleANNRuleMaster.h"
#import "SimpleANNRuleMaker.h"
#import "MatrixMult.h"
#import "VectorTransFunc.h"
#import "TransFunc.h"
#import "Quota.h"
#import <random.h> // to generate ad hoc private distributions to be used
                   // in SimpleANN and BPCT, to avoid interferences with random
                   // sequences used in determining agent behavior
#import "BPCTRuleMaster.h"
#import "BPCTRuleMaker.h"
#import "BPCTDataWarehouse.h"

@interface ModelSwarm: Swarm
{
```

Step2:importazione delle interfacce necessarie.

```
int dayNumber, maxOrderQuantity, meanPriceHistoryLength,
    localHistoryLength, stopLossInterval, checkingIfShortOrLong, printing,
    dataWindowLength, nAheadForecasting, forecastingTrainingSetLength,
    epochNumberInEachForecastingTrainingCycle, learningProcessEveryNDays,
    cleanForecastingANNEveryMgtemNDays;
float asymmetricBuySellProb, agentProbToActBeforeOpening,
      minCorrectingCoeff, maxCorrectingCoeff,
      asymmetricRange, floorP, agentProbToActBelowFloorP,
      maxLossRate,
      aNNInactivityRange, aNNForecastAppAgentActDailyProb;

// agentForge step 3
int agentNumber, randomAgentNumber, marketImitatingAgentNumber,
  locallyImitatingAgentNumber, stopLossAgentNumber,
  aNNForecastAppAgentNumber, bPCTAgentANumber,
  bPCTAgentAEO_EP_0_Number, bPCTAgentAEO_EP_1_Number,
  bPCTAgentAEO_EP_2_Number, bPCTAgentAEO_EP_3_Number,
                    bPCTAgentBNumber,
  bPCTAgentBEO_EP_0_Number, bPCTAgentBEO_EP_1_Number,
  bPCTAgentBEO_EP_2_Number, bPCTAgentBEO_EP_3_Number;
id <List> agentList, randomAgentList, marketImitatingAgentList,
        locallyImitatingAgentList, stopLossAgentList, aNNForecastAppAgentList,
        bPCTAgentAList, bPCTAgentBList;

id <Array> bPCTAgentAArray;
id <Index> bPCTAgentAArrayIndex;

id <Array> bPCTAgentBArray;
id <Index> bPCTAgentBArrayIndex;

id <Array> agentArray;
id <Index> agentArrayIndex;
id <ListShuffler> listShuffler;

id <ActionGroup> modelActions1, modelActions2, modelActions3;
id <Schedule> modelSchedule;

// agentForge step 3r

// see above about private distributions
id <SimpleRandomGenerator> myGenerator;
id <UniformDoubleDist> myUniformDblRand;
id <UniformIntegerDist> myUniformIntRand;
id <SimpleRandomGenerator> myGenerator2;
id <UniformDoubleDist> myUniformDblRand2;
id <UniformIntegerDist> myUniformIntRand2;
id <SimpleRandomGenerator> myGenerator3;
id <UniformDoubleDist> myUniformDblRand3;
id <UniformIntegerDist> myUniformIntRand3;

Book * theBook;
RandomRuleMaster * randomRuleMaster;
StopLossRuleMaster * stopLossRuleMaster;

CurrentAgent * theCurrentAgent;
```

Step3: contiene le variabili contenenti il numero di agenti delle varie famiglie (presenti anche nella probe), la lista contenente le instance dei nuovi agenti.

Oggetto per riordinare le liste di agenti.

Gruppi di azioni.

Classi di generatori di numeri casuali.

Dichiarazioni per i diversi inneschi casuali utilizzati dagli agenti.

```
  ForecastingAgent * forecastingAgent;

  SimpleANNRuleMaster * simpleANNRuleMaster;
  SimpleANNRuleMaker * simpleANNRuleMaker;
  MatrixMult * matrixMult;
  VectorTransFunc * vectorTransFunc;
  TransFunc * transFunc;
  Quota * quota;


  // BPCT

  // agentForge step 3b
  int epochNumberInEachBPCTTrainingCycle;
  float agentAEO_EPDelta, agentBEO_EPDelta;

  // we are creating independent RuleMaster/Maker for each type of
  // BPCT agent, to have separated random distributions also in weight
  // generation
  BPCTRuleMaster * bPCTRuleMasterA, * bPCTRuleMasterB;
  BPCTRuleMaker * bPCTRuleMakerA, * bPCTRuleMakerB;
  BPCTPriceRuleMaster * bPCTPriceRuleMasterA, * bPCTPriceRuleMasterB;

  // agentForge step 3c
  int bPCTAgentAInputNodeNumber, bPCTAgentAHiddenNodeNumber,
      bPCTAgentAOutputNodeNumber,
      bPCTAgentBInputNodeNumber, bPCTAgentBHiddenNodeNumber,
      bPCTAgentBOutputNodeNumber,

      bPCTPatternNumberInVerificationSet,
      bPCTPatternNumberInTrainingSet, bPCTAgentsAreDisplayingData,
      usingRandomOrderInBPCTLearning,
      longTermLearningInBPCT_OnlyWithCompleteTrainingSet,
      useOutputsAsTargetsInBPCT_RelearningScheme;
  float bPCTWeightRange, bPCTEps, bPCTAlpha;

}

+ createBegin: aZone;
- createEnd;
- buildObjects;
- buildActions;
- activateIn: swarmContext;

- increaseCurrentDayNumber;
- (int) getCurrentDay;
- getAgentArrayIndex;
- getBPCTAgentAArrayIndex;
- getBPCTAgentBArrayIndex;
- getBook;


  // agentForge step 4
- (int) getBPCTAgentANumber;
```

Step 3b e **3c**: relativi a BPCT.

Metodo di classe (preceduto da "+"), serve a creare i model e assegnare i parametri iniziali della simulazione.

Creazione degli oggetti della simulazione.

Contiene lo schedule.

Step4: vi è un metodo "get" per ogni lista dichiarata.

```
- getBPCTAgentAArrayIndex;
- (int) getBPCTAgentBNumber;
- getBPCTAgentBArrayIndex;
- getAgentList;
- getRandomAgentList;
- getMarketImitatingAgentList;
- getLocallyImitatingAgentList;
- getStopLossAgentList;
- getANNForecastAppAgentList;
- getBPCTAgentAList;
- getBPCTAgentBList;

- getForecastingAgent;
- openProbeTo: (int) ag;

@end
```

Metodi corrispondenti alle liste dichiarate nello step3.

Metodo che apre la sonda su un determinato agente.

MODELSWARM.M

```
#import "ModelSwarm.h"

@implementation ModelSwarm

+ createBegin: aZone
{
  ModelSwarm *obj;
  id <ProbeMap> probeMap;

  // in createBegin, we set up the simulation parameters

  // first, call our superclass createBegin - the return value is the
  // allocated Swarm object.

  obj = [super createBegin: aZone];

  // now fill in simulation parameters with default values.

  // agentForge step 5
  obj-> randomAgentNumber                = 300;
  obj-> marketImitatingAgentNumber       = 0;
  obj-> locallyImitatingAgentNumber      = 0;
  obj-> stopLossAgentNumber              = 0;
  obj-> aNNForecastAppAgentNumber        = 0;
  obj-> bPCTAgentAEO_EP_0_Number         = 0;
  obj-> bPCTAgentAEO_EP_1_Number         = 0;
  obj-> bPCTAgentAEO_EP_2_Number         = 0;
  obj-> bPCTAgentAEO_EP_3_Number         = 0;
  obj-> bPCTAgentBEO_EP_0_Number         = 0;
  obj-> bPCTAgentBEO_EP_1_Number         = 0;
  obj-> bPCTAgentBEO_EP_2_Number         = 0;
  obj-> bPCTAgentBEO_EP_3_Number         = 0;
```

Tipizzazione statica di obj.

Allocazione zone di memoria e inizializzazione dell'oggetto obj (della classe ModelSwarm).

Step5: assegnazioni parametri iniziali della simulazione.

Assegnazione del numero dei membri delle varie famiglie di agenti.

```
  obj->bPCTAgentANumber = obj->bPCTAgentAEO_EP_0_Number + obj-
>bPCTAgentAEO_EP_1_Number
                  + obj->bPCTAgentAEO_EP_2_Number + obj-
>bPCTAgentAEO_EP_3_Number;
  obj->bPCTAgentBNumber = obj->bPCTAgentBEO_EP_0_Number + obj-
>bPCTAgentBEO_EP_1_Number
                  + obj->bPCTAgentBEO_EP_2_Number + obj-
>bPCTAgentBEO_EP_3_Number;

  obj-> agentNumber = obj-> randomAgentNumber + obj->
marketImitatingAgentNumber +
                obj-> locallyImitatingAgentNumber + obj->stopLossAgentNumber +
                obj-> aNNForecastAppAgentNumber + obj->bPCTAgentANumber+
                obj -> bPCTAgentBNumber;
```

Calcolo del numero totale di agenti*.

*: In questo caso sono richiamate le variabili di oggetti diversi da "obj", che non sono stati in precedenza "tipizzati staticamente". Nello Step5b sono ricalcolate le somme, ma non è più necessaria la sintassi "->", perchè sono state create le instance degli oggetti (con la tipizzazione statica).

```
 // repeat this sum below, in BuildObjects method

  obj-> dayNumber                    = 0;
  obj-> asymmetricBuySellProb          = 0.9;
  obj-> minCorrectingCoeff           = 0.9;
  obj-> maxCorrectingCoeff            = 1.1;
  obj-> asymmetricRange               = 0.0;
  obj-> agentProbToActBeforeOpening      = 0.05;
  obj-> floorP                     = 0.3;
  obj-> agentProbToActBelowFloorP       = 0.5;
  obj-> maxOrderQuantity             = 3; // max order number per agent
```

Assegnazione valori a parametri utilizzati dagli agenti.

```
  // agentForge step 6
  // used by imitating agents
  obj-> meanPriceHistoryLength          = 200
  obj-> localHistoryLength            = 20;
  // used by stop loss agent
  obj-> maxLossRate                  = 0.10;
  obj-> stopLossInterval             = 2;
  obj-> checkingIfShortOrLong          = 1;
  // used by forecasting agent
  obj-> dataWindowLength             = 30;
  obj-> nAheadForecasting            = 10;
  // used by forecasting agent
  obj-> forecastingTrainingSetLength     = 100;
  obj->
epochNumberInEachForecastingTrainingCycle = 100;
  obj-> learningProcessEveryNDays       = 10;
  obj-> cleanForecastingANNEveryMgtemNDays = 50;
  // used by agents applying ANN forecast
  obj-> aNNInactivityRange           = 0.02;
  obj-> aNNForecastAppAgentActDailyProb  = 0.1;
  // used by agent of BPCT type
  obj->epochNumberInEachBPCTTrainingCycle = 100;
  obj->agentAEO_EPDelta              = 0.1;
  obj->agentBEO_EPDelta              = 10;
```

Step6: sono assegnati i valori base alle variabili che si vuole siano modificabili attraverso il "ModelSwarm" (la finestra che appare durante la simulazione).

Tali variabili sono inserite nella "probe".

Impostazione valori.

```
obj-> printing                  =  0; // if 1 many objects print
                                // data on the terminal
                                // window; if 2 only
                                // forecastingAgent prints;
                                // forecastinAgent uses also 3;
                                // 4 is used in BasicSumAgent;
                                // 5 in ANNForecastAppAgent;

// build a customized probe map. Without a probe map, the default
// is to show all variables and messages. Here we choose to
// customize the format of the probe to give a nicer interface.

probeMap = [EmptyProbeMap createBegin: aZone];
[probeMap setProbedClass: [self class]];
probeMap = [probeMap createEnd];

[probeMap addProbe: [probeLibrary
 getProbeForVariable: "randomAgentNumber"          inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "marketImitatingAgentNumber"  inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "locallyImitatingAgentNumber" inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "stopLossAgentNumber"         inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "aNNForecastAppAgentNumber"   inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "bPCTAgentAEO_EP_0_Number"    inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "bPCTAgentAEO_EP_1_Number"    inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "bPCTAgentAEO_EP_2_Number"    inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "bPCTAgentAEO_EP_3_Number"    inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "bPCTAgentBEO_EP_0_Number"    inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "bPCTAgentBEO_EP_1_Number"    inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "bPCTAgentBEO_EP_2_Number"    inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "bPCTAgentBEO_EP_3_Number"    inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "agentNumber"                inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "asymmetricBuySellProb"       inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "minCorrectingCoeff"         inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "maxCorrectingCoeff"         inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "asymmetricRange"            inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "agentProbToActBeforeOpening" inClass: [self class]]];
[probeMap addProbe: [probeLibrary
```

Impostazione della probe.

```
 getProbeForVariable: "floorP"                    inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "agentProbToActBelowFloorP"   inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "maxOrderQuantity"            inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "meanPriceHistoryLength"      inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "localHistoryLength"          inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "maxLossRate"                 inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "stopLossInterval"            inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "checkingIfShortOrLong"       inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "dataWindowLength"            inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "nAheadForecasting"           inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable:"forecastingTrainingSetLength" inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable:"epochNumberInEachForecastingTrainingCycle"
                                    inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "learningProcessEveryNDays"   inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "cleanForecastingANNEveryMgtemNDays"
                                    inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "aNNInactivityRange"          inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "aNNForecastAppAgentActDailyProb"
                                    inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "epochNumberInEachBPCTTrainingCycle"
                                    inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "agentAEO_EPDelta"            inClass: [self class]]];
[probeMap addProbe: [probeLibrary
 getProbeForVariable: "agentBEO_EPDelta"            inClass: [self class]]];

[probeMap addProbe: [probeLibrary
 getProbeForVariable: "printing"            inClass: [self class]]];

[probeMap addProbe: [probeLibrary
 getProbeForMessage:  "openProbeTo:"     inClass: [self class]]];

// Now install our custom probeMap into the probeLibrary.

[probeLibrary setProbeMap: probeMap For: [self class]];

return obj;
}
```

Creazione casella necessaria per l'inserimento del numero di agente su cui si vuole aprire la sonda. Una volta inserito il numero, si attiva l metodo "openProbeTo" con parametro pari al numero inserito.

```
- createEnd
{
  return [super createEnd];
}
```

Richiamo di createEnd (inizializzazione).

```
- buildObjects
{
  int i;
  float otherAgents;

  // agentForge step 7
  RandomAgent * anAgent1;
  MarketImitatingAgent * anAgent2;
  LocallyImitatingAgent * anAgent3;
  StopLossAgent * anAgent4;
  ANNForecastAppAgent * anAgent5;
  BPCTAgentA * anAgent6;
  BPCTAgentB * anAgent7;

  BPCTAgentAInterface * anInterface6;
  BPCTAgentBInterface * anInterface7;

  BPCTDataWarehouse * aDataWarehouse;

  // BPCT specific
  char * bPCTMinmaxFileNameA      ="minmaxA.data",  // mandatory, one for each
                                    // BPCT agent type
      * bPCTMinmaxFileNameB      ="minmaxB.data",

      * bPCTInitValuesFileNameA  ="initA.val",     // this file is used only
                                    // with internal data
                                    // generation (CT scheme)
                                    // but anyway the existence
                                    // of the file is not
                                    // mandatory
      * bPCTInitValuesFileNameB  ="initB.val",
      * unusedFile               ="";
  // BPCT end

  // creating tools to generate ad hoc distributions to be used
  // in SimpleANN and BPCT, to avoid interferences with random
  // sequences used in determining agent behavior

  // agentForge step 7r

  unsigned int mySeed, mySeed2, mySee
  // used by forecastingAgent
  mySeed = 223776;
  myGenerator = [MT19937gen create: self setStateFromSeed: mySeed];
            // MT19937gen is the generator internally used for default call
            // to Dbl and Int uniform distributions
  myUniformDblRand = [UniformDoubleDist create: self
                                    setGenerator: myGenerator];
  myUniformIntRand = [UniformIntegerDist create: self
```

Step7: crea le instance dei vari tipi di agenti.

Step7r: impostazione dei generatori di numeri casuali (diverso innesco), per evitare che si verifichino correlazioni nei comportamenti degli agenti.

```
                                    setGenerator: myGenerator];
  // used by BPCTAgentA
  mySeed2 = 112233;
  myGenerator2 = [MT19937gen create: self setStateFromSeed: mySeed2];
            // MT19937gen is the generator internally used for default call
            // to Dbl and Int uniform distributions
  myUniformDblRand2 = [UniformDoubleDist create: self
                                    setGenerator: myGenerator2];
  myUniformIntRand2 = [UniformIntegerDist create: self
                                    setGenerator: myGenerator2];

  // used by BPCTAgentB
  mySeed3 = 333112;
  myGenerator3 = [MT19937gen create: self setStateFromSeed: mySeed3];
            // MT19937gen is the generator internally used for default call
            // to Dbl and Int uniform distributions
  myUniformDblRand3 = [UniformDoubleDist create: self
                                    setGenerator: myGenerator3];
  myUniformIntRand3 = [UniformIntegerDist create: self
                                    setGenerator: myGene

  // agentForge step 5b
  // this is an operating second definition of the
  // followin sums (see above in obj-> agentNumber a

  bPCTAgentANumber = bPCTAgentAEO_EP_0_Number +
bPCTAgentAEO_EP_1_Number
              + bPCTAgentAEO_EP_2_Number + bPCTAgentAEO_EP_3_Number;

  bPCTAgentBNumber = bPCTAgentBEO_EP_0_Number +
bPCTAgentBEO_EP_1_Number
              + bPCTAgentBEO_EP_2_Number + bPCTAgentBEO_EP_3_Number;

  agentNumber = randomAgentNumber + marketImitatingAgentNumber +
            locallyImitatingAgentNumber + stopLossAgentNumber +

aNNForecastAppAgentNumber+bPCTAgentANumber+bPCTAgentBNumber;

  // if dataWindowLength+nAheadForecasting > meanPriceHistoryLength
  // a matrix error arises
  if(dataWindowLength+2*nAheadForecasting>meanPriceHistoryLength)
  {
    printf("The sum of dataWindowLength plus 2*nAheadForecasting is\n"
          "greater than meanPriceHistoryLength; this is a nonsense.\n"
          "See the comment 'NB about lagged values and return indexes' in\n"
          "ForecastingAgent.m\n");
    exit(0);
  }
```

Step5b: definizione delle somme per il calcolo del numero di agenti (nello step5 si sono inserite le somme in obj e nella probe).

```
// randomRuleMaster
randomRuleMaster=[RandomRuleMaster createBegin: self];
[randomRuleMaster setAgentProbToActBeforeOpening:
                        agentProbToActBeforeOpening];
[randomRuleMaster setMinCorrectingCoeff: minCorrectingCoeff];
[randomRuleMaster setMaxCorrectingCoeff: maxCorrectingCoeff];
[randomRuleMaster setAsymmetricRange: asymmetricRange];
[randomRuleMaster setFloorP: floorP
          andAgentProbToActBelowFloorP: agentProbToActBelowFloorP];
randomRuleMaster=[randomRuleMaster createEnd];
```

Creazione del rule master del RandomAgent.

```
// stopLossRuleMaster
stopLossRuleMaster=[StopLossRuleMaster createBegin: self];
[stopLossRuleMaster setAgentProbToActBeforeOpening:
                        agentProbToActBeforeOpening];
[stopLossRuleMaster setMinCorrectingCoeff: minCorrectingCoeff];
[stopLossRuleMaster setMaxCorrectingCoeff: maxCorrectingCoeff];
[stopLossRuleMaster setAsymmetricRange: asymmetricRange];
[stopLossRuleMaster setFloorP: floorP
          andAgentProbToActBelowFloorP: agentProbToActBelowFloorP];
stopLossRuleMaster=[stopLossRuleMaster createEnd];

listShuffler=[ListShuffler createBegin: self];
listShuffler=[listShuffler createEnd];

// agentForge step 5bb
agentList=[List create: self];
randomAgentList=[List create: self];
marketImitatingAgentList=[List create: self];
locallyImitatingAgentList=[List create: self];
stopLossAgentList=[List create: self];
aNNForecastAppAgentList=[List create: self];
bPCTAgentAList=[List create: self];
bPCTAgentBList=[List create: self];
```

Creazione delle liste di agenti. "agentList" conterrà tutti gli agenti creati, di qualsiasi classe. "randomAgentList" conterrà tutti gli agenti della classe "RandomAgent". Le liste possono essere riordinate (perdendo così l'ordine iniziale dei vari agenti creati).

```
agentArray = [Array create: self];
if (agentNumber==0) {printf("Nonsense: agentNumber cannot be 0");exit(0);}
[agentArray setCount: agentNumber];
agentArrayIndex = [agentArray begin: self];

// agentForge step 5bbb

if (bPCTAgentANumber!=0)
{
 bPCTAgentAArray = [Array create: self];
 [bPCTAgentAArray setCount: bPCTAgentANumber
 bPCTAgentAArrayIndex = [bPCTAgentAArray begin
}

if (bPCTAgentBNumber!=0)
{
 bPCTAgentBArray = [Array create: self];
 [bPCTAgentBArray setCount: bPCTAgentBNumber];
 bPCTAgentBArrayIndex = [bPCTAgentBArray begin: self];
```

Creazione dell'array degli agenti: serve per ottenere un "indice" degli agenti (l'array contiene gli indirizzi di memoria relativi ai vari agenti nell'ordine di creazione e quindi gli agenti sono ordinati secondo il proprio numero identificativo), in modo da rendere agevole l'eventuale ricerca di un determinato agente.

```
}
theBook = [Book createBegin: self];
[theBook setAgentArrayIndex: agentArrayIndex];
[theBook setNAheadForecasting: nAheadForecasting];
[theBook setAgentNumber: agentNumber];          // to build the matrixes
[theBook setMaxOrderQuantity: maxOrderQuantity];  // "   "
  if (meanPriceHistoryLength<2){printf("The length of the history of mean "
                                "prices cannot be <2 (internally "
                                "set to 2).\n");
                          meanPriceHistoryLength=2;
                          }
[theBook setMeanPriceHistoryLength: meanPriceHistoryLength];
  if (localHistoryLength<1){printf    ("The length of the local history"
                                "cannot be <1       (internally "
                                "set to 1).\n");
                          localHistoryLength=1;
                          }
[theBook setLocalHistoryLength: localHistoryLength];
[theBook setPrinting: printing];
theBook = [theBook createEnd];
```

Creazione del book.

```
// a few checks
  if (asymmetricBuySellProb<0.5){printf("The asymmetricBuySellProb "
                                "cannot be < 0.5 (internally "
                                "set to 0.5).\n");
                          asymmetricBuySellProb=0.5;
                          }
```

Controllo di asimmetricBuySellProb.

```
  if (stopLossInterval>meanPriceHistoryLength)
                          {printf("stopLossInterval "
                                "cannot be > meanPriceHistoryLength"
                                "\n(internally "
                                "set to meanPriceHistoryLength).\n");
                          stopLossInterval=meanPriceHistoryLength;
                          }
```

Step8: creazione delle famiglie di agenti (agenti e liste di agenti).

```
// randomAgent
for (i=1;i<=randomAgentNumber;i++)
{
anAgent1=[RandomAgent createBegin: self];
[anAgent1 setNumber: i];
[anAgent1 setMaxOrderQuantity: maxOrderQuantity];
[anAgent1 setBook: theBook];
[anAgent1 setRuleMaster: randomRuleMaster];
[anAgent1 setPrinting: printing];
anAgent1=[anAgent1 createEnd];

[agentList addLast: anAgent1];
[randomAgentList addLast: anAgent1];

[agentArrayIndex next];
[agentArrayIndex put: anAgent1];
}
```

Creazione dei RandomAgent.

Creazione dell'indice dell'array.

```
// marketImitatingAgent
```

```
 for (i=randomAgentNumber+1;i<=randomAgentNumber +
marketImitatingAgentNumber;i++)
 {
 anAgent2=[MarketImitatingAgent createBegin: self];
 [anAgent2 setNumber: i];
 [anAgent2 setAsymmetricBuySellProb: asymmetricBuySellProb];
 [anAgent2 setMaxOrderQuantity: maxOrderQuantity];
 [anAgent2 setBook: theBook];
 [anAgent2 setRuleMaster: randomRuleMaster];
 [anAgent2 setPrinting: printing];
 anAgent2=[anAgent2 createEnd];

 [agentList addLast: anAgent2];
 [marketImitatingAgentList addLast: anAgent2];

 [agentArrayIndex next];
 [agentArrayIndex put: anAgent2];
 }

 // locallyImitatingAgent
 for (i=randomAgentNumber+marketImitatingAgentNumber+1;

i<=randomAgentNumber+marketImitatingAgentNumber+locallyImitatingAgentNumber;
i++)
 {
 anAgent3=[LocallyImitatingAgent createBegin: self];
 [anAgent3 setNumber: i];
 [anAgent3 setAsymmetricBuySellProb: asymmetricBuySellProb];
 [anAgent3 setMaxOrderQuantity: maxOrderQuantity];
 [anAgent3 setBook: theBook];
 [anAgent3 setRuleMaster: randomRuleMaster];
 [anAgent3 setPrinting: printing];
 anAgent3=[anAgent3 createEnd];

 [agentList addLast: anAgent3];
 [locallyImitatingAgentList addLast: anAgent3];

 [agentArrayIndex next];
 [agentArrayIndex put: anAgent3];
 }

 // stopLossAgent
 for
(i=randomAgentNumber+marketImitatingAgentNumber+locallyImitatingAgentNumber+
1;

i<=randomAgentNumber+marketImitatingAgentNumber+locallyImitatingAgentNumber
+
        stopLossAgentNumber;i++)
 {
 anAgent4=[StopLossAgent createBegin: self];
 [anAgent4 setNumber: i];
 [anAgent4 setAsymmetricBuySellProb: asymmetricBuySellProb];
 [anAgent4 setMaxOrderQuantity: maxOrderQuantity];
 [anAgent4 setStopLossInterval: stopLossInterval];
```

```
 [anAgent4 setMaxLossRate: maxLossRate
         andCheckingIfShortOrLong: checkingIfShortOrLong];
 [anAgent4 setBook: theBook];
 [anAgent4 setRuleMaster: stopLossRuleMaster];
 [anAgent4 setPrinting: printing];
 anAgent4=[anAgent4 createEnd];

 [agentList addLast: anAgent4];
 [stopLossAgentList addLast: anAgent4];

 [agentArrayIndex next];
 [agentArrayIndex put: anAgent4];
 }

 // we must create here a lot of objects before anAgent5, because we have to
 // pass it the address of ForecastingAgent, needing the other objects created
 // immediately here

 // we create an instance of MatrixMult, VectorTransFunc
 // and of RuleMaster-RuleMaker

 matrixMult = [MatrixMult createBegin: self];
 matrixMult = [matrixMult createEnd];

 transFunc = [TransFunc createBegin: self];
 transFunc = [transFunc createEnd];

 vectorTransFunc = [VectorTransFunc createBegin: self];
 vectorTransFunc = [vectorTransFunc setTransFunc: transFunc];
 vectorTransFunc = [vectorTransFunc createEnd];

 // creating the simpleANNRuleMaker to evolve the ANN owned by the
 // forecastingAgent
 simpleANNRuleMaker=[SimpleANNRuleMaker createBegin: self];
 [simpleANNRuleMaker setMyUniformIntRand: myUniformIntRand];
 [simpleANNRuleMaker setMatrixMult: matrixMult];
 [simpleANNRuleMaker setVectorTransFunc: vectorTransFunc];
 simpleANNRuleMaker=[simpleANNRuleMaker createEnd];

 // creating the simpleANNRuleMaster to apply the ANN owned by the
 // forecastingAgent
 simpleANNRuleMaster=[SimpleANNRuleMaster createBegin: self];
 [simpleANNRuleMaster setSimpleANNRuleMaker: simpleANNRuleMaker];
 [simpleANNRuleMaster setMatrixMult: matrixMult];
 [simpleANNRuleMaster setVectorTransFunc: vectorTransFunc];
 simpleANNRuleMaster=[simpleANNRuleMaster createEnd];

 // creating the forecastingAgent
   if (cleanForecastingANNEveryMgtemNDays<learningProcessEveryNDays ||
      cleanForecastingANNEveryMgtemNDays%learningProcessEveryNDays !=0)
            {printf("cleanForecastingANNEveryMgtemNDays must be\n"
                  "must be greater than or equal and multiple of\n"
                  "learningProcessEveryNDays.\n");
           exit(0);
                        }
```

```
 forecastingAgent=[ForecastingAgent createBegin: self];
 [forecastingAgent setBook: theBook];
 [forecastingAgent setDataWindowLength: dataWindowLength
            andNAheadForecasting: nAheadForecasting
            andForecastingTrainingSetLength: forecastingTrainingSetLength
            andEpochNumberInEachForecastingTrainingCycle:
              epochNumberInEachForecastingTrainingCycle
            andLearningProcessEveryNDays: learningProcessEveryNDays
            andCleanForecastingANNEveryMgtemNDays:
                          cleanForecastingANNEveryMgtemNDays
            andForecastHistoryLength: meanPriceHistoryLength];
 [forecastingAgent setModelSwarmAddress: self];
 [forecastingAgent setSimpleANNRuleMasterAddress: simpleANNRuleMaster];
 [forecastingAgent setMyUniformDblRand: myUniformDblRand]; // ad hoc distr.
 [forecastingAgent setPrinting: printing];
 forecastingAgent=[forecastingAgent createEnd];

 quota = [Quota createBegin: self];
 [quota setForecastingAgent: forecastingAgent];
 [quota setBook: theBook];
 [quota setCleanForecastingANNEveryMgtemNDays:
                      cleanForecastingANNEveryMgtemNDays];
 quota = [quota createEnd];


 // aNNForecastAppAgent
 for
(i=randomAgentNumber+marketImitatingAgentNumber+locallyImitatingAgentNumber+
       stopLossAgentNumber + 1;

i<=randomAgentNumber+marketImitatingAgentNumber+locallyImitatingAgentNumber
+
        stopLossAgentNumber+aNNForecastAppAgentNumber;i++)
 {
 anAgent5=[ANNForecastAppAgent createBegin: self];
 [anAgent5 setNumber: i];
 [anAgent5 setAsymmetricBuySellProb: asymmetricBuySellProb];
 [anAgent5 setMaxOrderQuantity: maxOrderQuantity];
 [anAgent5 setBook: theBook];
 [anAgent5 setANNInactivityRange: aNNInactivityRange
         andANNForecastAppAgentActDailyProb:
aNNForecastAppAgentActDailyProb];
 [anAgent5 setMyUniformDblRand: myUniformDblRand]; // ad hoc distr.
 [anAgent5 setRuleMaster: randomRuleMaster];
 [anAgent5 setForecastingAgent: forecastingAgent];
 [anAgent5 setPrinting: printing];
 anAgent5=[anAgent5 createEnd];

 [agentList addLast: anAgent5];
 [aNNForecastAppAgentList addLast: anAgent5];

 [agentArrayIndex next];
 [agentArrayIndex put: anAgent5];
 }
```

```
// BPCT

// agentForge step 7b
    bPCTAgentAInputNodeNumber  = 7;
    bPCTAgentAHiddenNodeNumber = 5;
    bPCTAgentAOutputNodeNumber = 3;

    bPCTAgentBInputNodeNumber  = 8;
    bPCTAgentBHiddenNodeNumber = 6;
    bPCTAgentBOutputNodeNumber = 5;

    bPCTPatternNumberInVerificationSet = -1;
    bPCTPatternNumberInTrainingSet     = -10;
    bPCTAgentsAreDisplayingData        =  0;
    usingRandomOrderInBPCTLearning     =  1;
    longTermLearningInBPCT_OnlyWithCompleteTrainingSet = 1;
    useOutputsAsTargetsInBPCT_RelearningScheme = 0;
    bPCTWeightRange = 0.3;
    bPCTEps         = 0.6;
    bPCTAlpha       = 0.9;

[ObjectLoader load: self fromFileNamed: "bp.setup"];

otherAgents=randomAgentNumber+marketImitatingAgentNumber+
            locallyImitatingAgentNumber+
            stopLossAgentNumber+aNNForecastAppAgentNumber;

// bPCTAgent A
// the various rule Master/Maker are private, to keep independent the random
// distributions
bPCTRuleMakerA = [BPCTRuleMaker createBegin: self];
[bPCTRuleMakerA setMyUniformIntRand: myUniformIntRand2]; // ad hoc distr.
[bPCTRuleMakerA setMatrixMult: matrixMult];
[bPCTRuleMakerA setVectorTransFunc: vectorTransFunc];
bPCTRuleMakerA = [bPCTRuleMakerA createEnd];

bPCTRuleMasterA = [BPCTRuleMaster createBegin: self];
[bPCTRuleMasterA setRuleMaker: bPCTRuleMakerA];
[bPCTRuleMasterA setMatrixMult: matrixMult];
[bPCTRuleMasterA setVectorTransFunc: vectorTransFunc];
bPCTRuleMasterA = [bPCTRuleMasterA createEnd];

bPCTPriceRuleMasterA = [BPCTPriceRuleMaster createBegin: self];
[bPCTPriceRuleMasterA setMyUniformDblRand: myUniformDblRand2]; // ad hoc
distr.
[bPCTPriceRuleMasterA setAgentProbToActBeforeOpening:
                        agentProbToActBeforeOpening];
[bPCTPriceRuleMasterA setMinCorrectingCoeff: minCorrectingCoeff];
[bPCTPriceRuleMasterA setMaxCorrectingCoeff: maxCorrectingCoeff];
[bPCTPriceRuleMasterA setAsymmetricRange: asymmetricRange];
bPCTPriceRuleMasterA = [bPCTPriceRuleMasterA createEnd];

for (i=otherAgents + 1;
     i<=otherAgents + bPCTAgentANumber;i++)
{
```

```
// first we create the datawarehouse where BPCTagent technical data are stored
aDataWarehouse= [BPCTDataWarehouse createBegin: self];
[aDataWarehouse setMinmaxRowToBeModifiedFromInt: 9
                    usingGenericIntVariableAddress: &maxOrderQuantity];
[aDataWarehouse setVerificationFileName: unusedFile // never used here
            andTrainingFileName:     unusedFile // never used here
            andMinmaxName: bPCTMinmaxFileNameA
            andInitValuesFileName: bPCTInitValuesFileNameA];
[aDataWarehouse setInputNodeNumber:  bPCTAgentAInputNodeNumber
            andHiddenNodeNumber: bPCTAgentAHiddenNodeNumber
            andOutputNodeNumber: bPCTAgentAOutputNodeNumber
        andPatternNumberInVerificationSet: bPCTPatternNumberInVerificationSet
        andPatternNumberInTrainingSet:     bPCTPatternNumberInTrainingSet
        andEpochNumberInEachTrainingCycle:
                                epochNumberInEachBPCTTrainingCycle];
[aDataWarehouse setBackPropagationParametersWeightRange: bPCTWeightRange
            eps: bPCTEps alpha: bPCTAlpha
            andWithOrderInLearning: usingRandomOrderInBPCTLearning
            andLongTermLearningInCT:
            longTermLearningInBPCT_OnlyWithCompleteTrainingSet
            andUseOutputsAsTargetsInCT:
            useOutputsAsTargetsInBPCT_RelearningScheme];
[aDataWarehouse setMyUniformDblRand: myUniformDblRand2]; // ad hoc dist.

aDataWarehouse=[aDataWarehouse createEnd];

// then we create an interface for our agent, to simplifly its links
// with the observer, if any, but mainly as a help in CT building

anInterface6 = [BPCTAgentAInterface createBegin: self];
if (i<=otherAgents+bPCTAgentAEO_EP_0_Number+bPCTAgentAEO_EP_1_Number
              +bPCTAgentAEO_EP_2_Number+bPCTAgentAEO_EP_3_Number)
  [anInterface6 setUseEO_EP: 3]; // EO_EP 3
if (i<=otherAgents+bPCTAgentAEO_EP_0_Number+bPCTAgentAEO_EP_1_Number
              +bPCTAgentAEO_EP_2_Number)
  [anInterface6 setUseEO_EP: 2]; // EO_EP 2
if
(i<=otherAgents+bPCTAgentAEO_EP_0_Number+bPCTAgentAEO_EP_1_Number)
  [anInterface6 setUseEO_EP: 1]; // EO_EP 1
if (i<=otherAgents+bPCTAgentAEO_EP_0_Number)
  [anInterface6 setUseEO_EP: 0]; // no EO_EP use
[anInterface6 setEO_EPDelta: agentAEO_EPDelta];
[anInterface6 setMyUniformIntRand: myUniformIntRand2]; // ad hoc distr.
[anInterface6 setAgentNumber: i];
[anInterface6 setDataWarehouse: aDataWarehouse];
[anInterface6 setBook: theBook];
anInterface6 = [anInterface6 createEnd];
[anInterface6 initialize]; // NB. after createEnd

anAgent6 = [BPCTAgentA createBegin: self];
[anAgent6 setNumber: i andSetReadWeightsFromFile: 0]; // it never reads weights
                                        // from a file
[anAgent6 setMaxOrderQuantity: maxOrderQuantity];
[anAgent6 setDataWarehouse: aDataWarehouse];
[anAgent6 setInterface: anInterface6];        // double declaration for the parent
```

```
  [anAgent6 setSpecificInterface: anInterface6]; // and for the inheriting class
  [anAgent6 setRuleMaster: bPCTRuleMasterA];
  [anAgent6 setPriceRuleMaster: bPCTPriceRuleMasterA];
  [anAgent6 setDisplayDataWhileRunning: bPCTAgentsAreDisplayingData];
  [anAgent6 setBook: theBook]; // used by accounting method act2
  [anAgent6 setPrinting: printing];
  anAgent6 = [anAgent6 createEnd];

  [agentList addLast: anAgent6];
  [bPCTAgentAList addLast: anAgent6];
  [agentArrayIndex next];
  [agentArrayIndex put: anAgent6];
  [bPCTAgentAArrayIndex next];
  [bPCTAgentAArrayIndex put: anAgent6];
  }

  otherAgents+=bPCTAgentANumber;
  // bPCTAgent B
  // the various rule Master/Maker are private, to keep independent the random
  // distributions
  bPCTRuleMakerB = [BPCTRuleMaker createBegin: self];
  [bPCTRuleMakerB setMyUniformIntRand: myUniformIntRand3]; // ad hoc distr.
  [bPCTRuleMakerB setMatrixMult: matrixMult];
  [bPCTRuleMakerB setVectorTransFunc: vectorTransFunc];
  bPCTRuleMakerB = [bPCTRuleMakerB createEnd];
  bPCTRuleMasterB = [BPCTRuleMaster createBegin: self];
  [bPCTRuleMasterB setRuleMaker: bPCTRuleMakerB];
  [bPCTRuleMasterB setMatrixMult: matrixMult];
  [bPCTRuleMasterB setVectorTransFunc: vectorTransFunc];
  bPCTRuleMasterB = [bPCTRuleMasterB createEnd];
  bPCTPriceRuleMasterB = [BPCTPriceRuleMaster createBegin: self];
  [bPCTPriceRuleMasterB setMyUniformDblRand: myUniformDblRand3]; // ad hoc
distr.
  [bPCTPriceRuleMasterB setAgentProbToActBeforeOpening:
                              agentProbToActBeforeOpening];
  [bPCTPriceRuleMasterB setMinCorrectingCoeff: minCorrectingCoeff];
  [bPCTPriceRuleMasterB setMaxCorrectingCoeff: maxCorrectingCoeff];
  [bPCTPriceRuleMasterB setAsymmetricRange: asymmetricRange];
  bPCTPriceRuleMasterB = [bPCTPriceRuleMasterB createEnd];

  for (i=otherAgents + 1;
       i<=otherAgents + bPCTAgentBNumber;i++)
  {
  // first we create the datawarehouse where BPCTagent technical data are stored
  aDataWarehouse= [BPCTDataWarehouse createBegin: self];
  [aDataWarehouse setMinmaxRowToBeModifiedFromInt: 12
                      usingGenericIntVariableAddress: &maxOrderQuantity];
  [aDataWarehouse setVerificationFileName: unusedFile // never used here
                andTrainingFileName:     unusedFile // never used here
                andMinmaxName: bPCTMinmaxFileNameB
                andInitValuesFileName: bPCTInitValuesFileNameB];
  [aDataWarehouse setInputNodeNumber:  bPCTAgentBInputNodeNumber
                andHiddenNodeNumber: bPCTAgentBHiddenNodeNumber
                andOutputNodeNumber: bPCTAgentBOutputNodeNumber
           andPatternNumberInVerificationSet: bPCTPatternNumberInVerificationSet
```

```
        andPatternNumberInTrainingSet:     bPCTPatternNumberInTrainingSet
        andEpochNumberInEachTrainingCycle:
                                epochNumberInEachBPCTTrainingCycle];
 [aDataWarehouse setBackPropagationParametersWeightRange: bPCTWeightRange
              eps: bPCTEps alpha: bPCTAlpha
              andWithOrderInLearning: usingRandomOrderInBPCTLearning
              andLongTermLearningInCT:
              longTermLearningInBPCT_OnlyWithCompleteTrainingSet
              andUseOutputsAsTargetsInCT:
              useOutputsAsTargetsInBPCT_RelearningScheme];
 [aDataWarehouse setMyUniformDblRand: myUniformDblRand3]; // ad hoc dist.

 aDataWarehouse=[aDataWarehouse createEnd];
 // then we create an interface for our agent, to simplifly its links
 // with the observer, if any, but mainly as a help in CT building
 anInterface7 = [BPCTAgentBInterface createBegin: self];
 if (i<=otherAgents+bPCTAgentBEO_EP_0_Number+bPCTAgentBEO_EP_1_Number
              +bPCTAgentBEO_EP_2_Number+bPCTAgentBEO_EP_3_Number)
   [anInterface7 setUseEO_EP: 3]; // EO_EP 3
 if (i<=otherAgents+bPCTAgentBEO_EP_0_Number+bPCTAgentBEO_EP_1_Number
              +bPCTAgentBEO_EP_2_Number)
   [anInterface7 setUseEO_EP: 2]; // EO_EP 2
 if
(i<=otherAgents+bPCTAgentBEO_EP_0_Number+bPCTAgentBEO_EP_1_Number)
   [anInterface7 setUseEO_EP: 1]; // EO_EP 1
 if (i<=otherAgents+bPCTAgentBEO_EP_0_Number)
   [anInterface7 setUseEO_EP: 0]; // no EO_EP use
 [anInterface7 setEO_EPDelta: agentBEO_EPDelta];
 [anInterface7 setMyUniformIntRand: myUniformIntRand3]; // ad hoc distr.
 [anInterface7 setAgentNumber: i];
 [anInterface7 setDataWarehouse: aDataWarehouse];
 [anInterface7 setBook: theBook];
 [anInterface7 setForecastingAgent: forecastingAgent];
 anInterface7 = [anInterface7 createEnd];
 [anInterface7 initialize]; // NB. after createEnd
 anAgent7 = [BPCTAgentB createBegin: self];
 [anAgent7 setNumber: i andSetReadWeightsFromFile: 0]; // it never reads weights
                                        // from a file
 [anAgent7 setMaxOrderQuantity: maxOrderQuantity];
 [anAgent7 setDataWarehouse: aDataWarehouse];
 [anAgent7 setInterface: anInterface7];        // double declaration for the parent
 [anAgent7 setSpecificInterface: anInterface7]; // and for the inheriting class
 [anAgent7 setRuleMaster: bPCTRuleMasterB];
 [anAgent7 setPriceRuleMaster: bPCTPriceRuleMasterB];
 [anAgent7 setDisplayDataWhileRunning: bPCTAgentsAreDisplayingData];
 [anAgent7 setBook: theBook]; // used by accounting method act2
 [anAgent7 setPrinting: printing];
 anAgent7 = [anAgent7 createEnd];
 [agentList addLast: anAgent7];
 [bPCTAgentBList addLast: anAgent7];

 [agentArrayIndex next];
 [agentArrayIndex put: anAgent7];

 [bPCTAgentBArrayIndex next];
```

```
 [bPCTAgentBArrayIndex put: anAgent7];
 }

 // agentForge step 8 (this is the more difficult step: adding the block
 // of lines creating the new family of agents
 // current agent
 // see the comment in CurrentAgent.m to understand this trick
 theCurrentAgent=[CurrentAgent createBegin: self];
 [theCurrentAgent setAgentList: agentList];
 theCurrentAgent=[theCurrentAgent createEnd];

 return self;
}

- buildActions
{
 int i;

 // we create the list of simulation actions

 modelActions1 = [ActionGroup create: self];
 [modelActions1 createActionTo: self message: M(increaseCurrentDayNumber)];
 [modelActions1 createActionTo: theBook message: M(setClean)];
 [modelActions1 createActionTo: forecastingAgent message: M(step)];
 [modelActions1 createActionTo: quota message: M(step)];
 // this shuffling action is not relevant if agent are equal instances of
 // the same class, but it is useful to avoid biased situations when agents
 // are not homegeneous
 [modelActions1 createActionTo: listShuffler
                 message: M(shuffleWholeList:) :
 // acting before opening
 [modelActions1 createActionForEach: agentList message: M(act0)];


 // acting in the market
 modelActions2 = [ActionGroup create: self];
 [modelActions2 createActionTo: theCurrentAgent message: M(act1)];

 // accounting ...
 modelActions3 = [ActionGroup create: self];
 [modelActions3 createActionTo: theBook message: M(setMeanPrice)];
 [modelActions3 createActionForEach: agentList message: M(act2)];

 // then we create a schedule that executes the modelActions.
 modelSchedule = [Schedule createBegin: self];
 // we use here agentNumber steps in each cycle, while the observer uses
 // a low display frequency (e.g. 1) to show the price of each step-tick
 // we can also use a high d.f. (e.g.1000 with 100 agents) to r
 // simulation
 [modelSchedule setRepeatInterval: agentNumber];
 modelSchedule = [modelSchedule createEnd];
```

Creazione dell'oggetto "theCurrentAgent" della classe "CurrentAgent". Si richiama il metodo "setAgentList" che imposta come lista di agenti "agentList". Si vedano i commenti alla classe CurrentAgent.

Incremanta il giorno della simulazione.

Ripulisce il book.

I gruppo

Rimescola la lista di agenti (contiene tutti gli agenti creati), richiamando il metodo "shuffleWholeList" della classe "listShuffler").

Interpella in ordine gli agenti della lista. Si richiama il metodo "act0" per ogni agente della lista.

II gruppo

Creazione dei gruppi di azioni che sarranno compiute durante la simulazione.

Interpella un solo agente (richiama il metodo act1 dell'agente, si veda la classe "CurrentAgent").

III gruppo

Interpella in ordine gli agenti della lista. E' richiamato il metodo "act2" per ogni agente della lista.

Richiede al book il prezzo medio.

Creazione dello schedule.

Il periodo della simulazione ("tic").

```
  [modelSchedule at: 0 createAction: modelActions1];
  for (i=0;i<agentNumber;i++)
  [modelSchedule at: i createAction: modelActions2];
  [modelSchedule at: agentNumber-1 createAction: mode

  return self;
}

- activateIn: swarmContext
{
  // here, we activate the swarm in the context passed in
  // then we activate our schedule in ourselves

  [super activateIn: swarmContext];

  [modelSchedule activateIn: self];


  return [self getSwarmActivity];
}
```

Questo gruppo di istruzioni forma "l'orologio" della simulazione. Durante il primo "tic" si compie il primo gruppo di azioni (si interpella l'itera lista di agenti). Nella fase centrale (controllata dal ciclo "for", in cui è compiuto il secondo gruppo di azioni) agisce un solo agente per ogni "tic". Nell'ultimo periodo (o "tic") si compie il terzo gruppo di azioni (si interpella l'intera lista).

E' incrementata la variabile "dayNumber" e, se "printing=1", è data in output.

```
- increaseCurrentDayNumber
{
  dayNumber++;
  if(printing==1)printf("Day number #%5d\n",dayNumber);
  return self;
}

- (int) getCurrentDay
{
  return dayNumber;
}
```

Restituisce il valore del giorno corrente.

Step9: definizione dei metodi "get...", i quali restituiscono dei valori di variabili.

```
  // agentForge step 9
- (int) getBPCTAgentANumber{return bPCTAgentANumber;}
- (int) getBPCTAgentBNumber{return bPCTAgentBNumber;}
- getAgentList{return agentList;}
- getRandomAgentList{return randomAgentList;}
- getMarketImitatingAgentList{return marketImitatingAgentList;}
- getLocallyImitatingAgentList{return locallyImitatingAgentList;}
- getStopLossAgentList{return stopLossAgentList;}
- getANNForecastAppAgentList{return aNNForecastAppAgentList;}
- getBPCTAgentAList{return bPCTAgentAList;}
- getBPCTAgentBList{return bPCTAgentBList;}

- getAgentArrayIndex{return agentArrayIndex;}
- getBPCTAgentAArrayIndex{return bPCTAgentAArrayIndex;}
- getBPCTAgentBArrayIndex{return bPCTAgentBArrayIndex;}


- getForecastingAgent
{
  return forecastingAgent;
```

```
}

- getBook
{
  return theBook;
}

- openProbeTo: (int) n
{
  BasicSumAgent * anAgent;

  if (n<1 || n>agentNumber) return self;

  [agentArrayIndex setOffset: n-1];
  anAgent=[agentArrayIndex get];
  [anAgent getProbe];

  return anAgent;
}
@end
```

Metodo che apre una sonda sull'agente specificato attraverso il numero passato come parametro (numero identificativo dell'agente).

Controllo sul numero dell'agente inserito dall'utente (deve essere compreso tra 1 e "agentNumber").

Si ricerca l'agente nell'indice (l'agente n sarà alla posizione n-1, perchè l'indice comincia da 0). "get" restituisce l'indirizzo di memoria dell'agente cercato, di cui (nell'ultima istruzione) si richiama il metodo "getProbe", che apre la sonda su quell'agente.

Il metodo restituisce l'indirizzo di memoria dell'agente di cui si richiede l'apertura della sonda.

OBSERVERSWARM.H

```
#import "ModelSwarm.h"
#import "Book.h"
#import "ForecastingAgent.h"

  //agentForge step 10b
#import "BPCTAgentA.h"
#import "BPCTAgentAInterface.h"
#import "BPCTAgentB.h"
#import "BPCTAgentBInterface.h"

#import <simtoolsgui/GUISwarm.h>
#import <analysis.h> // to use EZgraph


@interface ObserverSwarm: GUISwarm
{
  int displayFrequency;       // freq. of update in the
                              // Observer widgets
  int showBookGraph, showAgentWealthGraph, showForecastingAgentGraph,
      displayPreviousDayMean, savePriceData, saveForecastingData,
     saveBookData,saveAgentWealthData,

  //agentForge step 10c
      numberOfTheBPCTAgentA_ToBeObservedDirectly,
      saveBPCTAgentAData,
      numberOfTheBPCTAgentB_ToBeObservedDirectly,
      saveBPCTAgentBData,

      bPCTPatternNumberInVerificationSet;

  id displayActions;          // schedule data stru
  id displaySchedule;

  // agentForge step 10
  id <EZGraph> priceGraph, bookGraph, agentWealth, agentWealthGraph,
               forecastingAgentGraph, bPCTAgentA_Graph, bPCTAgentB_Graph;

  int stopAtDayNumber;

  ModelSwarm *modelSwarm;// the Swarm containing our model
  Book * theBook;
  ForecastingAgent * forecastingAgent;


 //agentForge step 10d
  BPCTAgentA * agentA;
  BPCTAgentAInterface * agentAInterface;
  BPCTAgentB * agentB;
  BPCTAgentBInterface * agentBInterface;

}
+ createBegin: aZone;
- createEnd;
```

Step10: dichiarazioni dei grafici relativi alle varie famiglie di agenti.

Dichiarazioni degli oggetti grafici per le rappresentazioni relative agli oggetti della simulazione.

Dichiarazione dell'oggetto model e del book.

```
- buildObjects;
- buildActions;
- activateIn: swarmContext;
- checkToStop;

@end
```

OBSERVERSWARM.M

```
#import "ObserverSwarm.h"
#import <activity.h>
#import <simtoolsgui.h>

@implementation ObserverSwarm

+ createBegin: aZone
{
  ObserverSwarm *obj;
  id <ProbeMap> probeMap;

  // Superclass createBegin to allocate ourselves.

  obj = [super createBegin: aZone];

  // Fill in the relevant parameters.

  obj->displayFrequency         = 300;

  obj->stopAtDayNumber          = 2000; // to stop the program (if != 0)
  obj->displayPreviousDayMean   = 1;
  obj->savePriceData            = 1;
  obj->showBookGraph            = 0;
  obj->saveBookData             = 0;
  obj->showAgentWealthGraph     = 1;
  obj->saveAgentWealthData      = 1;
  obj->showForecastingAgentGraph = 1;
  obj->saveForecastingData      = 1;

  //agentForge step 11b
  obj->numberOfTheBPCTAgentA_ToBeObservedDirectly = 1;
  obj->saveBPCTAgentAData = 1;
  obj->numberOfTheBPCTAgentB_ToBeObservedDirectly = 1;
  obj->saveBPCTAgentBData = 1;

  // to build a customized probe map
  // without a probe map, the default is to show all variables and messages
  // here we choose to customize the appearance of the probe, to give a nicer
  // interface

  probeMap = [EmptyProbeMap createBegin: aZone];
  [probeMap setProbedClass: [self class]];
  probeMap = [probeMap createEnd];

  // add variables to be probed
```

Dichiazione dell'oggetto observer.

Step11b: impostazione valori iniziali.

Impostazione dei parametri iniziali dell'observer.

Step11c: creazione della probe.

Creazione della finestra che sarà visualizzata durante la simulazione.

```
  [probeMap addProbe: [probeLibrary getProbeForVariable:
          "displayFrequency"          inClass: [self class]]];
  [probeMap addProbe: [probeLibrary getProbeForVariable:
          "stopAtDayNumber"           inClass: [self class]]];
  [probeMap addProbe: [probeLibrary getProbeForVariable:
          "displayPreviousDayMean"    inClass: [self class]]];
  [probeMap addProbe: [probeLibrary getProbeForVariable:
          "savePriceData"             inClass: [self class]]];
  [probeMap addProbe: [probeLibrary getProbeForVariable:
          "showBookGraph"             inClass: [self class]]];
  [probeMap addProbe: [probeLibrary getProbeForVariable:
          "saveBookData"              inClass: [self class]]];
  [probeMap addProbe: [probeLibrary getProbeForVariable:
          "showAgentWealthGraph"      inClass: [self class]]];
  [probeMap addProbe: [probeLibrary getProbeForVariable:
          "saveAgentWealthData"       inClass: [self class]]];
  [probeMap addProbe: [probeLibrary getProbeForVariable:
          "showForecastingAgentGraph"  inClass: [self class]]];
  [probeMap addProbe: [probeLibrary getProbeForVariable:
          "saveForecastingData"       inClass: [self class]]];

  //agentForge step 11c
  [probeMap addProbe: [probeLibrary getProbeForVariable:
          "numberOfTheBPCTAgentA_ToBeObservedDirectly"
                                inClass: [self class]]];
  [probeMap addProbe: [probeLibrary getProbeForVariable:
          "saveBPCTAgentAData"        inClass: [self class]]];

  [probeMap addProbe: [probeLibrary getProbeForVariable:
          "numberOfTheBPCTAgentB_ToBeObservedDirectly"
                                inClass: [self class]]];
  [probeMap addProbe: [probeLibrary getProbeForVariable:
          "saveBPCTAgentBData"        inClass: [self class]]];

  // set our custom probeMap into the probeLibrary as the default probe for
  // the ObserverSwarm class

  [probeLibrary setProbeMap: probeMap For: [self class]];

  return obj;
}

- createEnd
{
  return [super createEnd];
}


- buildObjects
{
  [super buildObjects];

  modelSwarm = [ModelSwarm create: self];
```

Inizializzazione.

Creazione del model.

```
// to create probe objects on the model and the observer (self, here)
// ARCHIVED to allow the "Save" button to operate


CREATE_ARCHIVED_PROBE_DISPLAY (modelSwarm);
CREATE_ARCHIVED_PROBE_DISPLAY (self);

// we pause here to allow the parameters to be changed.

[controlPanel setStateStopped];


// The system will wait until the user hits "Start" or "Next"
// on the control panel

[modelSwarm buildObjects];

// checking the consistence of the displayFrequency with the agentNumber:
// for display reasons it is necessary to update the graphic widgets
// after all agent actions (mainly the BPCT ones, that calculate their
// targets at the end of a day and their inputs at the beginning of the day)
// so the displays are updated at displayFrequency-1
// if displayFrequency=1, in the agentNumber steps of a day we have the
// new BPCT input and output updated at the first step and the BPCT target
// updated at the last
// if we adopt a displayFrequency multiple of angentNumber we have the
// apparent coincidence of these values (##)

if(displayFrequency !=1 && displayFrequency %
   [[modelSwarm getAgentList] getCount] != 0)
 {
   printf("displayFrequency must be 1 or multiple of agentNumber.\n");
   exit(0);
 }

theBook=[modelSwarm getBook];
forecastingAgent=[modelSwarm getForecastingAgent];

// Observer display objects.

// The current price graph
priceGraph = [EZGraph createBegin: self];
SET_WINDOW_GEOMETRY_RECORD_NAME (priceGraph); // to allow "Save"

if(savePriceData==1)[priceGraph setFileOutput: (BOOL) 1];
                // to send the data also to a file

[priceGraph setTitle: "Current price"];
[priceGraph setAxisLabelsX: "Ticks x days." Y: "Price"];
priceGraph = [priceGraph createEnd];
// that above is the old way of creating an EZgraph; now a unique
// methos exists

// pay attention: the lines below cannot be placed before createEnd
[priceGraph createSequence: "Current price" withFeedFrom: theBook
```

Creazione delle finestre della simulazione (dell'observer e del model).

"theBook" conterrà l'indirizzo di memora del book.

Step11:creazione dei grafici.

Creazione del grafico dei prezzi.

```
                                    andSelector: M(getPrice)];
  if(displayPreviousDayMean==1)
  [priceGraph createSequence: "Day-1 mean p." withFeedFrom: theBook
                                    andSelector: M(getMeanPrice)];

  // The book graph

  if (showBookGraph==1)
  {
  bookGraph = [EZGraph createBegin: self];
  SET_WINDOW_GEOMETRY_RECORD_NAME (bookGraph); // to allow "Save"

  if(saveBookData==1)[bookGraph setFileOutput: (BOOL) 1];
                // to send the data also to a file

  [bookGraph setTitle: "Book log"];
  [bookGraph setAxisLabelsX: "Ticks x days." Y: "Sell and Buy Orders in Log."];
  bookGraph = [bookGraph createEnd];
  // that above is the old way of creating an EZgraph; now a unique
  // methos exists

  // pay attention: the lines below cannot be placed before createEnd

  [bookGraph createSequence: "Sell orders" withFeedFrom: theBook
                               andSelector: M(getSellOrderNumber)];
  [bookGraph createSequence: "Buy orders"  withFeedFrom: theBook
                               andSelector: M(getBuyOrderNumber)];
  }

  // The agent's wealth graph

  if (showAgentWealthGraph==1)
  {
  agentWealthGraph = [EZGraph createBegin: self];
  SET_WINDOW_GEOMETRY_RECORD_NAME (agentWealthGraph); // to allow
"Save"

  if(saveAgentWealthData==1)[agentWealthGraph setFileOutput: (BOOL) 1];
                // to send the data also to a file

  [agentWealthGraph setTitle: "Agent's wealth"];
  [agentWealthGraph setAxisLabelsX: "Ticks x days."
                               Y: "Wealth."];
  agentWealthGraph = [agentWealthGraph createEnd];
  // that above is the old way of creating an EZgraph; now a unique
  // methos exists

  // pay attention: the lines below cannot be placed before createEnd

  [agentWealthGraph createMinSequence: "MinWealth (all)"
                withFeedFrom: [modelSwarm getAgentList]
                andSelector: M(getWealthAtMeanDailyPrice)];
  [agentWealthGraph createAverageSequence: "MeanWealth (all)"
                withFeedFrom: [modelSwarm getAgentList]
                andSelector: M(getWealthAtMeanDailyPrice)];
```

Creazione del grafico del book.

Creazione del grafico del patrimonio dell'agente.

```
[agentWealthGraph createMaxSequence: "MaxWealth (all)"
        withFeedFrom: [modelSwarm getAgentList]
        andSelector: M(getWealthAtMeanDailyPrice)];
[agentWealthGraph createMinSequence: "MinWealth (r.)"
        withFeedFrom: [modelSwarm getRandomAgentList]
        andSelector: M(getWealthAtMeanDailyPrice)];
[agentWealthGraph createAverageSequence: "MeanWealth (r.)"
        withFeedFrom: [modelSwarm getRandomAgentList]
        andSelector: M(getWealthAtMeanDailyPrice)];
[agentWealthGraph createMaxSequence: "MaxWealth (r.)"
        withFeedFrom: [modelSwarm getRandomAgentList]
        andSelector: M(getWealthAtMeanDailyPrice)];
[agentWealthGraph createMinSequence: "MinWealth (m.i.)"
        withFeedFrom: [modelSwarm getMarketImitatingAgentList]
        andSelector: M(getWealthAtMeanDailyPrice)];
[agentWealthGraph createAverageSequence: "MeanWealth (m.i.)"
        withFeedFrom: [modelSwarm getMarketImitatingAgentList]
        andSelector: M(getWealthAtMeanDailyPrice)];
[agentWealthGraph createMaxSequence: "MaxWealth (m.i.)"
        withFeedFrom: [modelSwarm getMarketImitatingAgentList]
        andSelector: M(getWealthAtMeanDailyPrice)];
[agentWealthGraph createMinSequence: "MinWealth (l.i.)"
        withFeedFrom: [modelSwarm getLocallyImitatingAgentList]
        andSelector: M(getWealthAtMeanDailyPrice)];
[agentWealthGraph createAverageSequence: "MeanWealth (l.i.)"
        withFeedFrom: [modelSwarm getLocallyImitatingAgentList]
        andSelector: M(getWealthAtMeanDailyPrice)];
[agentWealthGraph createMaxSequence: "MaxWealth (l.i.)"
        withFeedFrom: [modelSwarm getLocallyImitatingAgentList]
        andSelector: M(getWealthAtMeanDailyPrice)];
[agentWealthGraph createMinSequence: "MinWealth (s.l.)"
        withFeedFrom: [modelSwarm getStopLossAgentList]
        andSelector: M(getWealthAtMeanDailyPrice)];
[agentWealthGraph createAverageSequence: "MeanWealth (s.l.)"
        withFeedFrom: [modelSwarm getStopLossAgentList]
        andSelector: M(getWealthAtMeanDailyPrice)];
[agentWealthGraph createMaxSequence: "MaxWealth (s.l.)"
        withFeedFrom: [modelSwarm getStopLossAgentList]
        andSelector: M(getWealthAtMeanDailyPrice)];
[agentWealthGraph createMinSequence: "MinWealth (ann)"
        withFeedFrom: [modelSwarm getANNForecastAppAgentList]
        andSelector: M(getWealthAtMeanDailyPrice)];
[agentWealthGraph createAverageSequence: "MeanWealth (ann)"
        withFeedFrom: [modelSwarm getANNForecastAppAgentList]
        andSelector: M(getWealthAtMeanDailyPrice)];
[agentWealthGraph createMaxSequence: "MaxWealth (ann)"
        withFeedFrom: [modelSwarm getANNForecastAppAgentList]
        andSelector: M(getWealthAtMeanDailyPrice)];

[agentWealthGraph createMinSequence: "MinWealth (bPCTA)"
        withFeedFrom: [modelSwarm getBPCTAgentAList]
        andSelector: M(getWealthAtMeanDailyPrice)];
[agentWealthGraph createAverageSequence: "MeanWealth (bPCTA)"
        withFeedFrom: [modelSwarm getBPCTAgentAList]
        andSelector: M(getWealthAtMeanDailyPrice)];
```

```
 [agentWealthGraph createMaxSequence: "MaxWealth (bPCTA)"
                withFeedFrom: [modelSwarm getBPCTAgentAList]
                andSelector: M(getWealthAtMeanDailyPrice)];

 [agentWealthGraph createMinSequence: "MinWealth (bPCTB)"
                withFeedFrom: [modelSwarm getBPCTAgentBList]
                andSelector: M(getWealthAtMeanDailyPrice)];
 [agentWealthGraph createAverageSequence: "MeanWealth (bPCTB)"
                withFeedFrom: [modelSwarm getBPCTAgentBList]
                andSelector: M(getWealthAtMeanDailyPrice)];
 [agentWealthGraph createMaxSequence: "MaxWealth (bPCTB)"
                withFeedFrom: [modelSwarm getBPCTAgentBList]
                andSelector: M(getWealthAtMeanDailyPrice)];

  //agentForge step 11d (before this point)
 }
 // The forecastingAgent graph

 if (showForecastingAgentGraph==1)
 {
 forecastingAgentGraph = [EZGraph createBegin: self];
 SET_WINDOW_GEOMETRY_RECORD_NAME (forecastingAgentGraph); // to allow
"Save"

 if(saveForecastingData==1)[forecastingAgentGraph setFileOutput: (BOOL) 1];
                                  // to send the data also to a file

 [forecastingAgentGraph setTitle: "Forecasting agent log"];
 [forecastingAgentGraph setAxisLabelsX:
                   "Ticks x days." Y: "Estimates and errors."];
 forecastingAgentGraph = [forecastingAgentGraph createEnd];
 // that above is the old way of creating an EZgraph; now a unique
 // methos exists

 // pay attention: the lines below cannot be placed before createEnd

 [forecastingAgentGraph createSequence: "Day-1 mean p. (index)"
                      withFeedFrom: theBook
                     andSelector: M(getMeanPriceIndex)];

 [forecastingAgentGraph createSequence: "Day-1 estimate (index)"
                      withFeedFrom: forecastingAgent
                     andSelector: M(getPastEstimateIndex)];

 [forecastingAgentGraph createSequence: "Back propagation error"
         withFeedFrom: forecastingAgent
         andSelector: M(getBackPropagationErrorInTrainingSet)];
 [forecastingAgentGraph createSequence: "Proportional error"
         withFeedFrom: forecastingAgent
         andSelector: M(getProportionalErrorInTrainingSet)];
 }

 // --- BPCT ---

 // agent A
```

```
 if([modelSwarm getBPCTAgentANumber] == 0)
                    numberOfTheBPCTAgentA_ToBeObservedDirectly=0;

 // this may be 0 if we do not have any agent (above) or if we choose to
 // have no display
 if (numberOfTheBPCTAgentA_ToBeObservedDirectly > 0)
                          // the value comes from probe
 {
 // to identify the address of the chosen agent

 [[modelSwarm getBPCTAgentAArrayIndex] setOffset:
                          numberOfTheBPCTAgentA_ToBeObservedDirectly-1];
 agentA = [[modelSwarm getBPCTAgentAArrayIndex] get];
 agentAInterface = [agentA getInterface];

  // agent A graph
  bPCTAgentA_Graph = [EZGraph createBegin: [self getZone]];
  SET_WINDOW_GEOMETRY_RECORD_NAME (bPCTAgentA_Graph); // to allow
"Save"
  [bPCTAgentA_Graph setTitle: "BPCTAgent A data"];
  [bPCTAgentA_Graph setAxisLabelsX:
   "Ticks x days" Y: "Value"];
  bPCTAgentA_Graph = [bPCTAgentA_Graph createEnd];

  if(saveBPCTAgentAData==1)[bPCTAgentA_Graph setFileOutput: (BOOL) 1];
                          // to send the data also to a file

  [bPCTAgentA_Graph createSequence: "meanPrice1_A"
                          withFeedFrom: agentAInterface
                          andSelector: M(getMeanPrice1)];

  [bPCTAgentA_Graph createSequence: "liquidityQuantity_out_A"
                          withFeedFrom: agentAInterface
                          andSelector: M(getLiquidityQuantity_out)];
  [bPCTAgentA_Graph createSequence: "liquidityQuantity_target_A"
                          withFeedFrom: agentAInterface
                          andSelector: M(getLiquidityQuantity_target)];
  [bPCTAgentA_Graph createSequence: "shareQuantity_out_A"
                          withFeedFrom: agentAInterface
                          andSelector: M(getShareQuantity_out)];
  [bPCTAgentA_Graph createSequence: "shareQuantity_target_A"
                          withFeedFrom: agentAInterface
                          andSelector: M(getShareQuantity_target)];
  [bPCTAgentA_Graph createSequence: "buySell_out_A"
                          withFeedFrom: agentAInterface
                          andSelector: M(getBuySell_out)];
  [bPCTAgentA_Graph createSequence: "buySell_target_A"
                          withFeedFrom: agentAInterface
                          andSelector: M(getBuySell_target)];

 }

 // agent B
```

```
 if([modelSwarm getBPCTAgentBNumber] == 0)
                    numberOfTheBPCTAgentB_ToBeObservedDirectly=0;

 // this may be 0 if we do not have any agent (above) or if we choose to
 // have no display
 if (numberOfTheBPCTAgentB_ToBeObservedDirectly > 0)
                          // the value comes from probe
 {
 // to identify the address of the chosen agent

 [[modelSwarm getBPCTAgentBArrayIndex] setOffset:
                         numberOfTheBPCTAgentB_ToBeObservedDirectly-1];
 agentB = [[modelSwarm getBPCTAgentBArrayIndex] get];
 agentBInterface = [agentB getInterface];

  // agent B graph
  bPCTAgentB_Graph = [EZGraph createBegin: [self getZone]];
  SET_WINDOW_GEOMETRY_RECORD_NAME (bPCTAgentB_Graph); // to allow
"Save"
  [bPCTAgentB_Graph setTitle: "BPCTAgent B data"];
  [bPCTAgentB_Graph setAxisLabelsX:
   "Ticks x days" Y: "Value"];
  bPCTAgentB_Graph = [bPCTAgentB_Graph createEnd];

  if(saveBPCTAgentBData==1)[bPCTAgentB_Graph setFileOutput: (BOOL) 1];
                          // to send the data also to a file

  [bPCTAgentB_Graph createSequence: "meanPrice1_B"
                          withFeedFrom: agentBInterface
                          andSelector: M(getMeanPrice1)];

  [bPCTAgentB_Graph createSequence: "liquidityQuantity_out_B"
                          withFeedFrom: agentBInterface
                          andSelector: M(getLiquidityQuantity_out)];
  [bPCTAgentB_Graph createSequence: "liquidityQuantity_target_B"
                          withFeedFrom: agentBInterface
                          andSelector: M(getLiquidityQuantity_target)];
  [bPCTAgentB_Graph createSequence: "shareQuantity_out_B"
                          withFeedFrom: agentBInterface
                          andSelector: M(getShareQuantity_out)];
  [bPCTAgentB_Graph createSequence: "shareQuantity_target_B"
                          withFeedFrom: agentBInterface
                          andSelector: M(getShareQuantity_target)];
  [bPCTAgentB_Graph createSequence: "closingPriceWealth_out_B"
                          withFeedFrom: agentBInterface
                          andSelector: M(getClosingPriceWealth_out)];
  [bPCTAgentB_Graph createSequence: "closingPriceWealth_target_B"
                          withFeedFrom: agentBInterface
                          andSelector: M(getClosingPriceWealth_target)];
  [bPCTAgentB_Graph createSequence: "forecastedPriceWealth_out_B"
                          withFeedFrom: agentBInterface
                          andSelector: M(getForecastedPriceWealth_out)];
  [bPCTAgentB_Graph createSequence: "forecastedPriceWealth_target_B"
                          withFeedFrom: agentBInterface
                          andSelector: M(getForecastedPriceWealth_target)];
```

```
  [bPCTAgentB_Graph createSequence: "buySell_out_B"
                          withFeedFrom: agentBInterface
                          andSelector: M(getBuySell_out)];
   [bPCTAgentB_Graph createSequence: "buySell_target_B"
                          withFeedFrom: agentBInterface
                          andSelector: M(getBuySell_target)];
  }
  // agentForge step 11 (before this point, add ...)
  // to check (pressing only once the next or the start button) the initial
  //settings, uncomment the following line
  //  [controlPanel setStateStopped];
  return self;
}

- buildActions
{
  [super buildActions];

  // model schedule.
  [modelSwarm buildActions];
  //agentForge step 11e
  bPCTPatternNumberInVerificationSet=0;
  if(agentA != nil)bPCTPatternNumberInVerificationSet=
                    [agentA getBPCTPatternNumberInVerificationSet];
  if(agentB != nil)bPCTPatternNumberInVerificationSet=
                    [agentB getBPCTPatternNumberInVerificationSet];
  // this variable is used below for a consistence check, as we need
  // here to have a verification set on length 1 (the same valued is used also
  // by the other BPCT agent; we have to repeat the assign operation in case
  // of missing categories of agents)
  // ActionGroup for Observer display

  displayActions = [ActionGroup create: self];

  // to update probes
  [displayActions createActionTo: probeDisplayManager message: M(update)];
  [displayActions createActionTo: actionCache message: M(doTkEvents)];

  [displayActions createActionTo: priceGraph message: M(step)];
  if (showBookGraph==1)
  [displayActions createActionTo: bookGraph  message: M(step)];
  if (showAgentWealthGraph==1)
  [displayActions createActionTo: agentWealthGraph  message: M(step)];
  if (showForecastingAgentGraph==1)
  [displayActions createActionTo: forecastingAgentGraph  message: M(step)];

  [displayActions createActionTo: self message: M(checkToStop)];

  // Display schedule. Note that the repeat interval is set by our
  // own Swarm data structure. The display is frequently the slowest part of a
  // simulation, when it is redraw less frequently things are faster

  displaySchedule = [Schedule createBegin: self];
  // we can use here a display frequency lower (e.g.1) than the number of steps
  // in the model (step number = agentNumber) to display the prices of each
```

```
  // step-tick
  [displaySchedule setRepeatInterval: displayFrequency];
  displaySchedule = [displaySchedule createEnd];

  [displaySchedule at: displayFrequency-1 createAction: displayActions];
                                  // see ## comment above
  //agentForge step 11f
  // BPCT
  if (numberOfTheBPCTAgentA_ToBeObservedDirectly > 0)
  // the following if condition is related to this CT use of the program, to
  // avoid nonsens
   if (bPCTPatternNumberInVerificationSet == -1)
     [displaySchedule at: displayFrequency-1  // see ## comment above
                    createActionTo: bPCTAgentA_Graph message: M(step)];
  if (numberOfTheBPCTAgentB_ToBeObservedDirectly > 0)
  // the following if condition is related to this CT use of the program, to
  // avoid nonsens
   if (bPCTPatternNumberInVerificationSet == -1)
     [displaySchedule at: displayFrequency-1  // see ## comment above
                    createActionTo: bPCTAgentB_Graph message: M(step)];
 return self;
}

- activateIn: swarmContext
{
// activateIn: - to activate the schedules to make them ready to run.
  [super activateIn: swarmContext];
  [modelSwarm activateIn: self];
  [displaySchedule activateIn: self];
  return [self getSwarmActivity];
}

  // to check for the stopping conditions
- checkToStop
{
  // if stopAtDayNumber is left to 0, the program will never stop
  // this stop occurs after the first tick of the time in modelSwarm,
  // but this fact does not affect the results in the observerSwarm
  // being the stop performed before the graph update
  if (stopAtDayNumber !=0 &&
      stopAtDayNumber <= [modelSwarm getCurrentDay])
  {
      printf("Stopping at day number %4d\n",
                          [modelSwarm getCurrentDay]);
       // we can restart by pressing "Start" or "Next", but the simulation
      // will run for displayFrequency ticks and then it will stop again
      [controlPanel setStateStopped];
  }
  return self;
}
@end
```

Lo schedule dell'observer

Controllo fine simulazione.

BASICSUMAGENT.H

```
// This is the root Agent in Sum; all operating agents inherit from it,
// but ForecastingAgent; also CTAgent must inherit from it, to have
// accounting capality

#import <objectbase/SwarmObject.h>
#import <random.h>
#import <math.h> // to use fabs etc.
#import "Book.h"
#import <simtoolsgui/GUISwarm.h> // to use
CREATE_ARCHIVED_PROBE_DISPLAY
#import <simtoolsgui.h>          // ""
#import "Matrix2.h"

@interface BasicSumAgent: SwarmObject
{

  int number, maxOrderQuantity, iMax, executedPriceCount,
  printing, actingBeforeOpening;

  float price,
        asymmetricBuySellProb, buySellSwitch,
        shareQuantity, shareValueAtMeanDailyPrice,
        liquidityQuantity, agentWealthAtMeanDailyPrice,
        meanOperatingPrice;
  Matrix2 * executedPrices;
  Book * theBook;

}

- createEnd;

- setNumber: (int) n;

- setAsymmetricBuySellProb: (float) p;

- setMaxOrderQuantity: (int) m;

- setBook: b;

- setPrinting: (int) p;

- setConfirmationOfExecutedPrice: (float) p;
```

I file *.h contengono la dichiarazione delle variabili e dei metodi utilizzati nella classe.

Step1:costruzione dei file *.m e *.h relativi ai diversi tipi di agenti.

Dichiarazione delle variabili utilizzate nella classe BasicSumAgent

Inizializzazione di un oggetto, per cui è già stato creato uno spazio in memoria

Impostazione del numero assegnato all'agente.

Impostazione scostamento simmetrico delle probabilità di acquisto e vendita degli agenti casuali

Imposta il numero di ordini inseribili da ciascun agente.

Imposta l'indirizzo di memoria del book

Scelta di dare in output le azioni degli agenti.

Incrementa la variabile "executedPriceCount" se p è diverso da 0 (conteggio numero ordini).

Elenco dei metodi. In questo caso sono tutti metodi richiamabili dalle instance (nome preceduto dal segno -).

```
- act0; // defined for compatibilty reasons with currentAgent
- act1;  // "" ""
- act2; // accounting at the end of a day
- getProbe;
- (float) getWealthAtMeanDailyPrice;

@end
```

Azione durante la fase di apertura

Azione durante la contrattazione continua

Azione durante la fase di chiusura

Apertura della sonda sul singolo agente.

Restituisce il valore della variabile agentWealthAtMeanDailyPrice

BASICSUMAGENT.M

I file *.m contengono il codice relativo ai metodi della classe.

```
#import "BasicSumAgent.h"

@implementation BasicSumAgent

  // we set the agent number
- setNumber: (int) n
{
  number = n;
  return self;
}

- setAsymmetricBuySellProb: (float) p
{
  asymmetricBuySellProb=p;
  return self;
}

  // the address of the book of the market
- setBook: b
{
  theBook=b;
  return self;
}

  // how many orders are we placing at each time?
- setMaxOrderQuantity: (int) m
{
  maxOrderQuantity=m;
  return self;
}

  // this is a toggle: print / don't print
- setPrinting: (int) p
{
  printing=p;
  return self;
}
```

Si importa il file di interfaccia.

Con "self" si restituisce l'indirizzo della classe di appartenenza.

In questi metodi si assegnano semplicemente dei valori a delle variabili.

"p" contiene un opzione di stampa (serve come controllo per i dialoghi agente-book).

```
- createEnd
{
  [super createEnd];

  // initialize
  executedPriceCount=0;

  // keeping data for accounting reasons
  executedPrices=[Matrix2 createBegin: [self getZone]];
  [executedPrices setDimensionRows: maxOrderQuantity Cols: 1
                                Code: 1000+number];
  executedPrices=[executedPrices createEnd];

  shareQuantity=0;
  shareValueAtMeanDailyPrice=0;
  liquidityQuantity=0;
  agentWealthAtMeanDailyPrice=0;
  return self;
}

- act0  // defined for compatibility reasons with current agent
{
 return self;
}

- act1  // defined for compatibility reasons with current
{
 return self;
}


- setConfirmationOfExecutedPrice: (float) p
{
 [executedPrices R: executedPriceCount C: 0 setFrom: p];
 if(p!=0)++executedPriceCount;
 return self;
}

 // accounting
- act2
{
 float mP, q;
 int i;

 // accounting operation and display of
 // a message about the operation eventually done: if price is not zero
 // we have been buyer (executedPrice>0) or seller (executedPrice<0)
  if(printing==4)
  for(i=0;i<executedPriceCount;i++)
  printf("I'm agent #%3d and the book told me: %7.4f\n",
                        number,[executedPrices R:i C:0]);

  mP=[theBook getMeanPrice];
```

Crea un oggetto (attribuendole una specifica zona di memoria) della classe Matrix2. Tale oggetto avrà nome "executedPrices".

Invia tre messaggi all'oggetto "executedPrices" (è un vettore che conterrà i prezzi dei contratti conclusi).

Conclude l'inizializzazione di "executedPrices".

Inizializzazione variabili

Questi due metodi sono solo dichiarati, saranno implementati nelle classi di agenti più evolute (è utile che nella classe "base" siano già dichiarati tutti i metodi che saranno utilizzati dalle classi che erediteranno).

I prezzi dei contratti conclusi (se di acquisto hanno segno+, se di vendita hanno segno -) dagli agenti sono memorizzati in una matrice (l'oggetto "executedPrices"). "executedPriceCount" contiene il numero dei prezzi diversi da 0 ed è incrementato solo se p è diverso da 0 (restituisce il numero di contratti conclusi).

Variabili interne al metodo.

Output dei prezzi a cui sono stati conclusi i contratti.

Assegnazione ad mP del valore di prezzo medio contenuto nel book

```
  // quantity of shares
  q=0;
  for (i=0;i<executedPriceCou
  {
  if([executedPrices R:i C:0]>
  if([executedPrices R:i C:0]<
  }

  shareQuantity += q;

  shareValueAtMeanDailyPrice=shareQuantity*mP;
                          // meanPrice at the end of each "day

  meanOperatingPrice=0; // used in CT
  for (i=0;i<executedPriceCount;i++)
   {
    liquidityQuantity  -= [executedPrices R:i C:0];
    meanOperatingPrice += fabs([executedPrices R:i C:0]);
   }

  if(executedPriceCount != 0) meanOperatingPrice/=executedPriceCount;
else meanOperatingPrice=mP;

  if(printing==4)
  printf("I'm agent #%3d and the meanOperatingPrice is: %7.4f\n\n",
                          number,meanOperatingPrice);


  agentWealthAtMeanDailyPrice=shareValueAtMeanDailyPrice+liquidityQuantity;

  return self;
}

- getProbe
{
 CREATE_ARCHIVED_PROBE_DISPLAY (self);
 return self;
}

- (float) getWealthAtMeanDailyPrice
{
 return agentWealthAtMeanDailyPrice;
}

@end
```

Si incrementa la variabile q se il contratto concluso è di acquisto e viceversa. La quantità per ogni contratto concluso è pari a un titolo.

Somma q alla quantità già detenuta di titoli

Valore del portafoglio titoli al prezzo medio giornaliero.

Variazioni della liquidità dell'agente

Calcolo del prezzo medio di conclusione dei contratti, fabs è una funzione che restituisce il valore assoluto degli executedPrices.

meanOperatingPrice/executedPriceCount

Se "printing=4" è dato in output il valore di meanOperatingPrice.

Valore del patrimonio dell'agente (composto da valore dei titoli al prezzo medio giornaliero e liquidità posseduta).

Apertura della sonda relativa alla sonda dell'agente.

Commenti

Questa classe eredita solamente da SwarmObject (è una superclasse).
Sono dichiarate le seguenti variabili:

- int *number* (variabile contenente il numero identificativo dell'agente);

- int *maxOrderQuantity* (variabile contenente il numero di titoli massimo che si vuole inserire nell'ordine);
- int *iMax* (numero di ordini inseriti dall'agente);
- int *executedPriceCount* (numero di valori di prezzo diversi da 0, contenuti nel vettore "*executedPrices*");
- int *printing* (opzione di stampa);
- int *actingBeforeOpening* (vale 1 o 0 a seconda che l'agente operi o meno in fase di apertura);
- float *price* (prezzo limite proposto dall'agente);
- float *asymmetricBuySellProb* (impostazione della asimmetria della probabilità di acquisto e vendita);
- float *buySellSwitch* (probabilità di acquisto dell'agente, è utilizzata nella classe RandomAgent);
- float *shareQuantity* (variabile contenente il numero di titoli totale posseduto dall'utente);
- float *shareValueAtMeanDailyPrice* (valore monetario del portafoglio, al prezzo medio giornaliero);
- float *liquidityQuantity* (liquidità totale dell'agente);
- float *agentWealthAtMeanDailyPrice* (patrimonio dell'agente = liquidità + valore portafoglio);
- float *meanOperatingPrice* (prezzo medio a cui sono stati conclusi i contratti);
- *Matrix2 * executedPrices* (puntatore all'oggetto *Matrix2*, è un vettore contenente valori di prezzo dei contratti che l'agente ha concluso, con segno + se si tratta di acquisti e segno - se si tratta di vendite);
- *Book * theBook* (puntatore all'oggetto Book).

Sono dichiarati i seguenti metodi:
- *createEnd*(non restituisce alcun valore), non possiede alcun parametro.
Inizializzazione di un oggetto, per cui è già stato creato uno spazio in memoria.

Algoritmo:
richiamo del metodo di inizializzazione di un oggetto (crateEnd) da Swarm
executedPriceCount=0;
creazione dell'oggetto "*executedPrices*" (attribuendole una specifica zona di memoria) della classe *Matrix2*
richiamo metodo *setDimensionRows* dell'oggetto *executedPrices*, con parametri *maxOrderQuantity*, 1 e 1000+*number*
inizializzazione di *executedPrices*
shareQuantity=0;
shareValueAtMeanDailyPrice=0;
liquidityQuantity=0;
agentWealthAtMeanDailyPrice=0;

- *setNumber* (non restituisce alcun valore), possiede il seguente parametro:
1) (int) *n* (numero identificativo dell'agente).
Impostazione del numero assegnato all'agente.

Algoritmo:
number = n

- *setAsymmetricBuySellProb* (non restituisce alcun valore), possiede il seguente parametro:
1) (float) p;
Impostazione scostamento simmetrico delle probabilità di acquisto e di vendita degli agenti casuali

Algoritmo:
asymmetricBuySellProb=p

- *setMaxOrderQuantity* (non restituisce alcun valore), possiede il seguente parametro:
1) (int) *m* (numero di ordini inseribili da ciascun agente).
Imposta il numero di ordini inseribili da ciascun agente.

Algoritmo:
maxOrderQuantity=m

- *setBook* (non restituisce alcun valore), possiede il seguente parametro:
1) b (indirizzo di memoria del book).
Imposta l'indirizzo di memoria del book.

Algoritmo:
theBook=b

- *setPrinting* (non restituisce alcun valore), possiede il seguente parametro:
1)(int) *p* (contiene un opzione di stampa, per dare in output i dialoghi agente-book).
Scelta di dare in output le azioni degli agenti.

Algoritmo:
printing=p

- *setConfirmationOfExecutedPrice* (non restituisce alcun valore), possiede il seguente parametro:
1) (float) *p*;
Incrementa la variabile "*executedPriceCount*" se *p* è diverso da 0.

Algoritmo:
passo messaggi all'oggetto *executedPrices* (R: *executedPriceCount* C: 0 setFrom: p), equivale a inserire in p il valore registrato nella posizione *executedPriceCount* del vettore *executedPrices*
Se (*p*!=0) ++*executedPriceCount*

- *act0* (semplice dichiarazione di metodo, si sviluppa nelle classi che ereditano).
Azione durante la fase di apertura.

- *act1* (semplice dichiarazione di metodo, si sviluppa nelle classi che ereditano).
Azione durante la contrattazione continua.

- *act2* (non ha parametri e non ha valori di ritorno).

Azione durante la fase di chiusura

Algoritmo:
Dichiarazione delle seguenti variabili:
float *mP* (contiene il prezzo medio giornaliero che è richiesto al book)
float *q* (variazione del numero di titoli posseduti, risultante dal conteggio dei segni +e- dei valori in *executedPrices*)
int i (i variabile contatore)
Se (*printing*==4) Output delle differenze di prezzo(prezzo dato dal book - prezzo limite) conosciuto dall'agente
for(i=0;i<*executedPriceCount*;i++)
printf("I'm agent #%3d and the book told me: %7.4f\n",
number,[*executedPrices* R:i C:0])
mP=[*theBook getMeanPrice*]
q=0
for (i=0;i<*executedPriceCount*;i++)
Se (*executedPrices*(i) > 0) q++
Se (*executedPrices*(i) < 0) q--
shareQuantity += q
*shareValueAtMeanDailyPrice=shareQuantity*mP*
meanOperatingPrice=0
for (i=0;i<*executedPriceCount*;i++)
liquidityQuantity -= [*executedPrices* R:i C:0]
meanOperatingPrice += valore assoluto(*executedPrices*(i))
Se(*executedPriceCount* != 0)
meanOperatingPrice/=executedPriceCount;
altrimenti
meanOperatingPrice=mP
Se(*printing*==4)
printf("I'm agent #%3d and the meanOperatingPrice is: %7.4f\n\n", *number*, *meanOperatingPrice*)
agentWealthAtMeanDailyPrice=
shareValueAtMeanDailyPrice+liquidityQuantity;

- *getProbe* (non ha parametri e non ha valori di ritorno).
Creazione della sonda sull'agente.

Algoritmo:
CREATE_ARCHIVED_PROBE_DISPLAY (self)

- (float) *getWealthAtMeanDailyPrice*
Restituisce il valore float della variabile agentWealthAtMeanDailyPrice.

Algoritmo:
return *agentWealthAtMeanDailyPrice*

BASICSUMRULEMASTER.H

```
// This is the root RuleMaster in Sum; all operating rule masters inherit from
// it, but ForecastingAgent and CTAgent

#import <objectbase/SwarmObject.h>
#import <random.h>

@interface BasicSumRuleMaster: SwarmObject
{
  float agentProbToActBeforeOpening,
        minCorrectingCoeff, maxCorrectingCoeff,
        asymmetricRange, floorP, agentProbToActBelowFloorP;
}

- setAgentProbToActBeforeOpening: (float) p;
- setMinCorrectingCoeff: (float) min;
- setMaxCorrectingCoeff: (float) max;
- setAsymmetricRange: (float) a;
- setFloorP: (float) f andAgentProbToActBelowFloorP: (float) p;
- createEnd;

@end
```

Imposta la probabilità che un agente operi in fase di apertura

Imposta il valore minimo del coefficiente casuale di correzione che gli agenti casuali moltiplicano per il prezzo che conoscono (lastPrice) per inserire gli

Imposta il valore massimo del coefficiente casuale di correzione che gli agenti casuali moltiplicano per il prezzo che conoscono (lastPrice) per inserire gli

Imposta il valore di scostamento dell'intervallo di valori in cui sarà compreso il coefficiente casuale che è moltiplicato al prezzo conosciuto dagli agenti (lastPrice) in fase di inserimento ordini (RuleMaster).

Imposta la probabilità di agire se il prezzo scende sotto un valore minimo

Inizializzazione oggetto

BASICSUMRULEMASTER.M

```
#import "BasicSumRuleMaster.h"

@implementation BasicSumRuleMaster
- setAgentProbToActBeforeOpening: (float) p
{
  agentProbToActBeforeOpening=p;
  return self;
}
- setMinCorrectingCoeff: (float) min
{
```

Semplici assegnazioni (notare che i metodi hanno tutti un parametro).

```
  minCorrectingCoeff=min;
  return self;
}
- setMaxCorrectingCoeff: (float) max
{
  maxCorrectingCoeff=max;
  return self;
}
- setAsymmetricRange: (float) a
{
  asymmetricRange=a;
  return self;
}
- setFloorP: (float) f andAgentProbToActBelowFloorP: (float) p
{
  floorP=f;
  agentProbToActBelowFloorP=p;
  return self;
}
```

Semplici assegnazioni (notare che i metodi hanno tutti un parametro).

```
- createEnd
{
   [super createEnd];

   return self;
}
```

In questo caso si richiama solamente il metodo "crateEnd" dalle librerie di Swarm per l'inizializzazione dell'oggetto, ma non si inizializzano altre variabili.

```
@end
```

Commenti

Questa classe eredita da SwarmObject
Sono dichiarate le seguenti variabili:

- float *agentProbToActBeforeOpening* (valore di probabilità che l'agente operi in fase di apertura);
- float *minCorrectingCoeff* (valore minimo del coefficiente casuale di correzione che gli agenti casuali moltiplicano per il prezzo che conoscono (*lastPrice*) per inserire gli ordini);
- float *maxCorrectingCoeff* (valore massimo del coefficiente casuale di correzione che gli agenti casuali moltiplicano per il prezzo che conoscono (*lastPrice*) per inserire gli ordini);
- float asymmetricRange (valore che va ad aggiungersi alle due variabili precedenti in fase di determinazione del valore casuale del coefficiente da moltiplicare al *lastPrice*);
- float *floorP* (livello di prezzo minimo);
- float *agentProbToActBelowFloorP* (probabilità che l'agente operi sotto un determinato livello di prezzo minimo).

Metodi della classe:
- *setAgentProbToActBeforeOpening* (non restituisce alcun valore), possiede i seguenti parametri:
1) (float) *p*
Imposta la probabilità che un agente operi in fase di apertura.

Algoritmo:
agentProbToActBeforeOpening=p;

- *setMinCorrectingCoeff* (non restituisce alcun valore), possiede i seguenti parametri:
1) (float) *min*
Imposta il valore minimo del coefficiente casuale di correzione che gli agenti casuali moltiplicano per il prezzo che conoscono (*lastPrice*) per inserire gli ordini.

Algoritmo:
minCorrectingCoeff=min;

- *setMaxCorrectingCoeff* (non restituisce alcun valore), possiede i seguenti parametri:
1) (float) *max*
Imposta il valore massimo del coefficiente casuale di correzione che gli agenti casuali moltiplicano per il prezzo che conoscono (*lastPrice*) per inserire gli ordini.

Algoritmo:
maxCorrectingCoeff=max;

- *setAsymmetricRange* (non restituisce alcun valore), possiede i seguenti parametri:
1) (float) *a*
Imposta il valore di scostamento dell'intervallo di valori in cui sarà compreso il coefficiente casuale che viene moltiplicato al prezzo conosciuto dagli agenti (*lastPrice*) in fase di inserimento ordini (RuleMaster).

Algoritmo:
asymmetricRange=a;

- *setFloorP* (non restituisce alcun valore), possiede i seguenti parametri:
1) (float) *f*
2) (float) *p*
Imposta la probabilità di agire se il prezzo scende sotto un valore minimo.

Algoritmo:
floorP=f;
agentProbToActBelowFloorP=p;

- *createEnd* (non restituisce alcun valore), non possiede alcun parametro.
Richiama "crateEnd" dalle librerie di Swarm.
Inizializzazione oggetto.

RANDOMAGENT.H

```
// RandomAgent inherits from BasicSumAgent and operates randomly;
// anyway, in case of price sliding below low price, a buying behavior
// prevails
```

```
#import "BasicSumAgent.h"
#import "RandomRuleMaster.h"

@interface RandomAgent: BasicSumAgent
{
  RandomRuleMaster * ruleMaster;

}
- setRuleMaster: r

- act0; // acting - placing buy or sell orders - before opening
- act1;  // placing buy or sell orders

@end
```

Eredita le caratteristiche di BasicSumAgent

Dichiarazione di un puntatore ("ruleMaster") ad un oggetto della classe "RandomRuleMaster". Questo modo di dichiarare oggetti è detto "tipizzazione statica".

Imposta l'indirizzo di memoria del rule master.

Sono implementati i metodi relativi alle azioni da compiere in fase di apertura e in fase di contrattazione continua, già presenti nell'agente base, ma non sviluppate.

RANDOMAGENT.M

```
#import "RandomAgent.h"
@implementation RandomAgent

  // our ruleMaster
- setRuleMaster: r
{
  ruleMaster=r;
  return self;
}

  // Acting before opening
- act0
{
   float lastPrice;
   int i;

     // clearing executedPrices
        executedPriceCount=0;
        for (i=0;i<maxOrderQuantity;i++) [executedPrices R:i C:0 $

   actingBeforeOpening=0;

   buySellSwitch=0.5; //the standard random choice situation
```

Variabili interne al metodo.

E' inizializzato il vettore "executedPrices".

Questa variabile intera (qui azzerata) è dichiarata in "BasicSumAgent": può valere 0 (l'agente non opera in fase di apertura) oppure 1 (opera).

E' passato come parametro (p) ai metodi del rule master.

```
  lastPrice=[theBook getPrice];
  // Asking the ruleMaster for a position (if the price is > 0 we are
  // buyer; if it is <0 we are seller; if it is =0 we do nothing
  // the ruleMaster uses in this case the probability of operating
  // before opening
  price=[ruleMaster getPriceBeforeOpening: lastPrice
                withBuySellP: buySellSwitch];
  if (price!=0) actingBeforeOpening=1;
  // how many orders?
  if (maxOrderQuantity==1) iMax=1;
   else iMax=[uniformIntRand getIntegerWithMin: 1 withMax: maxOrderQuantity];
  // if price==0 theBook is not recording it
  [theBook setOrderBeforeOpeningFromAgent: number atPrice: price];

  return self;
}

 // acting in the market
- act1
{
  float lastPrice;
  int i;

  buySellSwitch=0.5; // the standard random choice situation

  // the agent has operated before opening
  if (actingBeforeOpening==1) return self;

  lastPrice=[theBook getPrice];

  price=[ruleMaster getPrice: lastPrice
                withBuySellP: buySellSwitch];

  // how many orders?
  if (maxOrderQuantity==1) iMax=1;
   else iMax=[uniformIntRand getIntegerWithMin: 1 withMax: maxOrderQuantity];
  for (i=1;i<=iMax;i++)
  [theBook setOrderFromAgent: number atPrice: price];

  return self;
}
@end
```

Si richiama il metodo del book che restituisce l'ultimo prezzo.

Si passano i valori di "lastPrice" e "buySellSwitch" al metodo "getPriceBeforeOpening" del rule master. Il risultato è un float che è assegnato a "price".

Se il prezzo è diverso da 0, l'agente opera in fase di apertura.

Se maxOrderQuantity=1, iMax=1, altrimenti è un numero casuale tra 1 e maxOrderQuantity.

L'agente comunica al book il proprio numero di riconoscimento e il prezzo limite. La quantità è data dal numero di ordini, infatti ad ogni ordine corrisponde la richiesta di acquisto o vendita di un solo titolo.

Il metodo act1 è analogo ad act0.

Se l'agente ha operato in fase di apertura non opera più.

Commenti

Il RandomAgent eredita dal BasicSumAgent, utilizza le variabili del BasicSumAgent e del BasicRuleMaster.
Nella classe è dichiarato l'oggetto *ruleMaster* come puntatore (RandomRuleMaster * *ruleMaster*) alla classe RandomRuleMaster.
Sono sviluppati i seguenti metodi:

- *setRuleMaster* (non restituisce alcun valore), possiede il seguente parametro:
1) r, contiene l'indirizzo di memoria del rule master.
Imposta l'indirizzo di memoria del rule master.

Algoritmo:
ruleMaster=r

- *act0* (non restituisce alcun valore), non ha parametri.
Contiene le azioni da compiere in fase di apertura.

Algoritmo:
Dichiarazioni variabili interne:
float *lastPrice* (contiene l'ultimo prezzo, è richiesto al book);
int i (variabile contatore);
executedPriceCount=0;
for (i=0;i<*maxOrderQuantity*;i++)
elemento i-esimo del vettore *executedPrices* = 0
actingBeforeOpening=0;
buySellSwitch=0.5;
lastPrice=richiamo il metodo *getPrice* del Book
price = richiamo il metodo *getPriceBeforeOpening* dell'oggetto *ruleMaster*, dando come parametri *lastPrice* e *buySellSwitch*
Se (*price*!=0)
actingBeforeOpening=1;
Se (*maxOrderQuantity*==1)
iMax=1;
altrimenti
iMax=valore casuale intero compreso tra 1 e *maxOrderQuantity*
for (i=1;i<=iMax;i++)
  richiamo il metodo del Book *setOrderBeforeOpeningFromAgent* con parametri *number* e *price*

- *act1*(non restituisce alcun valore), non ha parametri.
Contiene le azioni da compiere contrattazione continua.

Algoritmo:
Dichiarazioni variabili interne:
float *lastPrice* (contiene l'ultimo prezzo; è richiesto al book);
int i (variabile contatore);
buySellSwitch=0.5;
lastPrice=richiamando il metodo *getPrice* del Book
price = richiamo il metodo getPrice dell'oggetto *ruleMaster*, dando come parametri *lastPrice* e *buySellSwitch*

Se (*price*!=0)
actingBeforeOpening=1;
Se (*maxOrderQuantity*==1)
iMax=1;
altrimenti
iMax=valore casuale intero compreso tra 1 e *maxOrderQuantity*
for (i=1;i<=*iMax*;i++)
  richiamo il metodo del Book *setOrderFromAgent* con parametri *number* e *price*

RANDOMRULEMASTER.H

```
// This is the RandomRuleMaster, guiding the behavior of RandomAgent,
// MarketImitatingAgent, LocallyImitatingAgent

#import "BasicSumRuleMaster.h"

@interface RandomRuleMaster: BasicSumRuleMaster
{

}

- (float) getPriceBeforeOpening: (float) lp
                    withBuySellP: (float) p;
- (float) getPrice: (float) lp withBuySellP: (float) p;

@end
```

Attiva il metodo "getPrice" con probabilità pari a "agentProbToActBeforeOpening".

Dato il valore dell'ultimo prezzo e la probabilità di comprare relativa all'agente, il metodo restituisce la variazione di liquidità dell'agente (entrata o uscita monetaria, che può esserevista anche come valore del singolo contratto).

RANDOMRULEMASTER.M

```
#import "RandomRuleMaster.h"

@implementation RandomRuleMaster


- (float) getPriceBeforeOpening: (float) lastPrice
        withBuySellP: (float) p
{
  if (agentProbToActBeforeOpening <
     (float) [uniformDblRand getDoubleWithMin: 0.0 withMax: 1.0])
   return 0.0;
  else return [self getPrice: lastPrice withBuySellP: p];
}



- (float) getPrice: (float) lastPrice withBuySellP: (float) p
{
  float price, choice;
```

Se il valore casuale risulta maggiore di "agentProbToActBeforeOpening" il metodo restituisce 0.

Richiamo del metodo "getPrice" della classe corrente, passandogli come parametri le variabili "lastPrice" e "p" (queste due sono a loro volta parametri di questo metodo).

Variabili interne al metodo.

```
  price = lastPrice * (float)
     [uniformDblRand getDoubleWithMin: minCorrectingCoeff + asymmetricRange
                    withMax: maxCorrectingCoeff + asymme
  choice=1; // buy with probability p
if ((float) [uniformDblRand getDoubleWithMin: 0.0 withMax: 1.0] > p)
  choice=-1;// sell
  // buying when the price glide below the floor price, with probability
  // agentProbToActBelowFloorP
  if (lastPrice<floorP){
  if (agentProbToActBelowFloorP > (float)
     [uniformDblRand getDoubleWithMin: 0.0 withMax: 1.0])
     {
     price=lastPrice;
     choice=1;
     }            }

  return choice*price;
}

@end
```

Il "price" è dato dalla moltiplicazione di "lastPrice" con un coefficiente

"p" è un parametro del metodo che conterrà il valore di probabilità di acquistare.

Scelta di vendere o comprare.

Se "lastPrice" è minore di "floorP" l'agente compra solo se il numero casuale compreso tra 0 e 1 è minore di "agentProbToActBelowFloorP".

Il valore di ritorno del metodo è il valore del prezzo con segno +o- a seconda della decisione dell'agente di comprare o vendere (registrata da "choice").

Commenti

Questo è il RandomRuleMaster, guida il comportamento di RandomAgent, MarketImitatingAgent e LocallyImitatingAgent.

RandomRuleMaster eredita da BasicSumRuleMaster

Non sono dichiarate nuove variabili.
Sono sviluppati i seguenti metodi (entrambi restituiscono un prezzo con segno +o- a seconda che si tratti di una scelta di acquisto o di vendita):
- *getPriceBeforeOpening* (restituisce un valore float) e possiede i seguenti parametri:
1) (float) *lp*, l'ultimo prezzo (*lastPrice*);
2) (float) *p*, probabilità di acquisto relativa all'agente.
Attiva il metodo "*getPrice*" con probabilità pari a "*agentProbToActBeforeOpening*".

Algoritmo:
Se (*agentProbToActBeforeOpening*) < (valore casuale float tra 0.0 e 1.0)
  restituisce 0.0;
altrimenti
  restituisce [valore del metodo *getPrice* con parametri *lastPrice* e *p*]

- *getPrice* (restituisce un valore float) e possiede i seguenti parametri:
1) (float) *lp*, l'ultimo prezzo (*lastPrice*);
2) (float) *p*, probabilità di acquisto relativa all'agente.
Dato il valore dell'ultimo prezzo e la probabilità di comprare relativa all'agente, il metodo restituisce la variazione di liquidità dell'agente (entrata o uscita monetaria).

Algoritmo:
Sono dichiarate internamente le seguenti variabili:
float *price* (è dato da *lastPrice**un coef. casuale)
float *choice* (può valere +1 con probabilità pari a p o -1
con probabilità (1-p), contiene la scelta di acquistare o vendere)
price = *lastPrice* * (valore casuale float compreso tra "*minCorrectingCoeff* + *asymmetricRange*" e "*maxCorrectingCoeff* + *asymmetricRange*")
choice=1
Se (valore casuale float tra 0.0 e 1.0 > p)
  choice=-1
Se (*lastPrice*<*floorP*)
Se (*agentProbToActBelowFloorP* > (valore casuale float compreso tra 0.0 e 1.0)
price=*lastPrice*
choice=1
restituisce *choice***price*

CURRENTAGENT.H

```
#import <objectbase/SwarmObject.h>
#import <collections.h>
#import "BasicSumAgent.h"
@interface CurrentAgent: SwarmObject
{
  id <List> agentList;
}
- setAgentList: l;
- createEnd;
- act1;
@end
```

Assegna l'indirizzo della lista di agenti da considerare (è passato come parametro "l").

Conclusine dell'inizializzazione.

Richiamo del metodo "act1 di un agente della lista.

CURRENTAGENT.M

```
#import "CurrentAgent.h"
// this is a ghost agent which simply transfers the 'act' message to
// the first agent in the list agentList and rotate it
// this trick is necessary to send the act message agentNumber times; so
// the observer can dispaly the price of each acting step
@implementation CurrentAgent
// we set the agent list
- setAgentList: l
{
   agentList = l;
```

Assegnazione dell'indirizzo della lista passata come parametro del metodo.

```
  return self;
}
- createEnd
{
  [super createEnd];
  return self;
}
- act1
{
  BasicSumAgent * actingAgent;
  actingAgent=[agentList removeFirst];
  [agentList addLast: actingAgent];
  [actingAgent act1];
  return self;
}
@end
```

Tipizzazione dell'oggetto "actingAgent": serve per far sapere al compilatorie a quali metodi ha accesso l'oggetto della lista che è richiamato (l'oggetto non dovrà necessariamente essere della classe "BasicSum Agent", ma potrà appartenere a classi che ereditano da questa classe).

E' rimosso il primo oggetto della lista "agentList" e assegnato ad "actingAgent". Nella seconda istruzione "actingAgent" è aggiunto al fondo della lista e, nell'ultima istruzione, si richiama il metodo "act1" a cui ha accesso "actingAgent".
Con il richiamo del metodo "act1" di un oggetto di tipo "CurrentAgent" si ha il richiamo del metodo "act1" del primo oggetto della lista considerata. Richiamandolo più volte di seguito si percorre l'intera lista di oggetti (perchè il primo oggetto è sempre spostato in coda agli altri).

ALLEGATO B

JAVASUM: CODICE DEL MODELLO

MAKEFILE

```
JAVA_SRC = StartJavaSum.java ObserverSwarm.java ModelSwarm.java
RandomAgent.java RandomRuleMaster.java CurrentAgent.java CurrentIstant.java
Book.java SwarmUtils.java Matrix.java

all: $(JAVA_SRC)
  $(SWARMHOME)/bin/javacswarm $(JAVA_SRC)

clean:
  rm -f *.class

cleanall:
  rm -f *.class
  rm -f *~
  rm *.stackdump
```

STARTJAVASUM.JAVA

```
import swarm.Globals;
import swarm.defobj.Zone;

/**
 * The StartJavaSum class contains main().
 * We follow here the typical Swarm structure with main() (in Start...
 * as a convention) generating the Observer and the Observer
 * generating the Model.
 *
```

```
 * @author Marco Agagliate
 */
public class StartJavaSum
      {
      /**
  * The main() function is the top-level place where everything starts.
  */
      public static void main (String[ ] args)
            {
            /** The observer in our application. */
            ObserverSwarm observerSwarm;

            // Swarm initialization: all Swarm apps must call this first.
            Globals.env.initSwarm ("JavaSum", "0.0",
                        "pietro.terna@unito.it", args);

            // Create a top-level Swarm object, observerSwarm, and
            // build its internal objects and activities.

            observerSwarm =
              (ObserverSwarm)Globals.env.lispAppArchiver.getWithZone$key(
                        Globals.env.globalZone, "observerSwarm");

           // to save control panel position
           Globals.env.setWindowGeometryRecordName
                     (observerSwarm, "observerSwarm");

           // build objets into the obsever
           observerSwarm.buildObjects();

           // build actions into the obsever
           observerSwarm.buildActions();

           // activate
           observerSwarm.activateIn(null);

            // Now start the displaySwarm and the control panel it
            // provides.
            observerSwarm.go();

            // The user has pressed Quit.  Drop everything and return.
            observerSwarm.drop();
            }
      }
```

OBSERVERSWARM.JAVA

```
import swarm.Globals;
import swarm.Selector;

import swarm.defobj.Zone;
import swarm.defobj.ZoneImpl;
import swarm.defobj.Symbol;
```

```
import swarm.simtoolsgui.GUISwarm;
import swarm.simtoolsgui.GUISwarmImpl;

import swarm.activity.ActionGroup;
import swarm.activity.ActionGroupImpl;
import swarm.activity.Schedule;
import swarm.activity.ScheduleImpl;
import swarm.activity.Activity;

import swarm.objectbase.Swarm;
import swarm.objectbase.SwarmImpl;
import swarm.objectbase.EmptyProbeMap;
import swarm.objectbase.EmptyProbeMapImpl;

import swarm.collections.ListImpl;

import swarm.analysis.EZGraph;
import swarm.analysis.EZGraphImpl;

/**
 * ObserverSwarm.java The observer swarm is collection of objects that
 * are used to run and observe the ModelSwarm that actually comprises
 * the simulation.
 *
 *
 * @author  Marco Agagliate
 */
public class ObserverSwarm extends GUISwarmImpl
      {
      // Declare the display parameters and their default values.
      /** Update frequency. */
      public int displayFrequency                = 10;
      /** To stop simulation at the end of a day. */
      public int stopAtDayNumber                 = 4;
      /** To create file and represent previous day mean price. */
      public int displayPreviousDayMean          = 1;
      /** To save on a file the price data. */
      public int savePriceData                   = 1;
      /** To show book graph. */
      public int showBookGraph                   = 1;
      /** To save book data. */
      public int saveBookData                    = 0;
      /** To show agent's wealth graph. */
      public int showAgentWealthGraph            = 1;
      /** To save agent's wealth data.*/
      public int saveAgentWealthData             = 1;

      // Declare other variables local to ObserverSwarm.
      /** the ModelSwarm we are observing */
      public ModelSwarm modelSwarm;
      /** the single Schedule instance */
      public ScheduleImpl displaySchedule;
      /** The book of the simulation. */
      public Book theBook;
      /** two ActionGroup for sequence of GUI events */
```

```
public ActionGroupImpl displayActions;
/** our graphics or EZGraph output to files*/
public EZGraphImpl priceGraph, priceFile, bookGraph,
            agentWealth, agentWealthGraph,
            sellOrderFile, buyOrderFile,
            minWealthAllFile, meanWealthAllFile,
            maxWealthAllFile, minWealthRandomFile,
            meanWealthRandomFile, maxWealthRandomFile,
            meanPriceFile;

/** Constructor for a new ObserverSwarm. */
public ObserverSwarm(Zone azone)
    {
    // Use the parent class to create an observer swarm.
    super(azone);

    // Build a custom probe map.  Without a probe map, the default
    // is to show all variables and messages.  Here we choose to
    // customize the appearance of the probe, giving a nicer
    // interface.

    // Create the probe map and give it the ObserverSwarm class.
    EmptyProbeMapImpl probeMap = new EmptyProbeMapImpl(azone,
                                                    getClass());

    probeMap.addProbe(Globals.env.probeLibrary
        .getProbeForVariable$inClass("displayFrequency",
            ObserverSwarm.this.getClass ()));
    probeMap.addProbe(Globals.env.probeLibrary
        .getProbeForVariable$inClass("stopAtDayNumber",
            ObserverSwarm.this.getClass ()));
    probeMap.addProbe(Globals.env.probeLibrary
        .getProbeForVariable$inClass("displayPreviousDayMean",
            ObserverSwarm.this.getClass ()));
    probeMap.addProbe(Globals.env.probeLibrary
        .getProbeForVariable$inClass("savePriceData",
            ObserverSwarm.this.getClass ()));
    probeMap.addProbe(Globals.env.probeLibrary
        .getProbeForVariable$inClass("showBookGraph",
            ObserverSwarm.this.getClass ()));
    probeMap.addProbe(Globals.env.probeLibrary
        .getProbeForVariable$inClass("saveBookData",
            ObserverSwarm.this.getClass ()));
    probeMap.addProbe(Globals.env.probeLibrary
        .getProbeForVariable$inClass("showAgentWealthGraph",
            ObserverSwarm.this.getClass ()));
    probeMap.addProbe(Globals.env.probeLibrary
        .getProbeForVariable$inClass("saveAgentWealthData",
            ObserverSwarm.this.getClass ()));

    // And finaly install our probe map into the probeLibrary.
    // Note that this library object was automatically created by
    // initSwarm.
    Globals.env.probeLibrary.setProbeMap$For(probeMap, getClass());
    }
```

```
/** Create the objects used to display the model. */
public Object buildObjects()
    {
    // Zone modelZone;
    Selector sel;

    // Use the parent class to initialize the process.
    super.buildObjects();

    // We create the model that we'are actually observing, by
    // creating an instance of the ModelSwarm class, modelSwarm.
    modelSwarm =
        (ModelSwarm)Globals.env.lispAppArchiver.getWithZone$key(
                getZone(), "modelSwarm");

    // Now create probe objects on the model and on ourselves.
    Globals.env.createArchivedProbeDisplay(modelSwarm, "modelSwarm");
    Globals.env.createArchivedProbeDisplay(this, "observerSwarm");

    // Instruct the control panel to wait for a button event: we
    // halt here until someone hits a control panel button.
    // Eventually this will allow the user a chance to fill in
    // parameters before the simulation runs.
    getControlPanel().setStateStopped();

    // Allow the model swarm to build its objects.
    modelSwarm.buildObjects();

    // checking the consistence of the displayFrequency with the agentNumber:
    if(displayFrequency !=1 && displayFrequency %
        modelSwarm.getAgentList().getCount() != 0)
        {
         System.out.println("displayFrequency must be 1 or multiple of
                                                    istantNumber.\n");
        System.exit(1);
        }

    theBook = modelSwarm.getBook();

    bookGraph = new EZGraphImpl(getZone (), "Book log", "Tiks x days.",
                "Sell and Buy Orders in Log.", "bookGraph");
    Globals.env.setWindowGeometryRecordName (bookGraph, "bookGraph");

    // the data for the bookGraph
    bookGraph.createSequence$withFeedFrom$andSelector
        ("Sell orders", theBook,
        SwarmUtils.getSelector(theBook,"getSellOrderNumber"));
    bookGraph.createSequence$withFeedFrom$andSelector
        ("Buy orders", theBook,
        SwarmUtils.getSelector(theBook,"getBuyOrderNumber"));

    // file
    sellOrderFile =  new EZGraphImpl(getZone(), "sellOrder");
        sellOrderFile.createSequence$withFeedFrom$andSelector
```

```
                ("txt", theBook, SwarmUtils.getSelector(theBook,
                                                "getSellOrderNumber"));
            buyOrderFile =  new EZGraphImpl(getZone(), "buyOrder");
                buyOrderFile.createSequence$withFeedFrom$andSelector
                ("txt", theBook, SwarmUtils.getSelector(theBook,
                                                    getBuyOrderNumber"));


            agentWealthGraph = new EZGraphImpl(getZone (), "Agent's wealth",
                            "Ticks x days.", "Wealth", "agentWealthGraph");
            Globals.env.setWindowGeometryRecordName (agentWealthGraph,
                                                    "agentWealthGraph");

            // the data for the agentGraph
            agentWealthGraph.createMinSequence$withFeedFrom$andSelector
                ("Min Wealth (all)", modelSwarm.getAgentList(),

SwarmUtils.getSelector("BasicSumAgent","getWealthAtMeanDailyPrice"));
            agentWealthGraph.createAverageSequence$withFeedFrom$andSelector
                ("Mean Wealth (all)", modelSwarm.getAgentList(),

SwarmUtils.getSelector("BasicSumAgent","getWealthAtMeanDailyPrice"));
            agentWealthGraph.createMaxSequence$withFeedFrom$andSelector
                ("Max Wealth (all)", modelSwarm.getAgentList(),

SwarmUtils.getSelector("BasicSumAgent","getWealthAtMeanDailyPrice"));
            agentWealthGraph.createMinSequence$withFeedFrom$andSelector
                ("Min Wealth (r.)", modelSwarm.getRandomAgentList(),

SwarmUtils.getSelector("BasicSumAgent","getWealthAtMeanDailyPrice"));
            agentWealthGraph.createAverageSequence$withFeedFrom$andSelector
                ("Mean Wealth (r.)", modelSwarm.getRandomAgentList(),

SwarmUtils.getSelector("BasicSumAgent","getWealthAtMeanDailyPrice"));
            agentWealthGraph.createMaxSequence$withFeedFrom$andSelector
                ("Max Wealth (r.)", modelSwarm.getRandomAgentList(),

SwarmUtils.getSelector("BasicSumAgent","getWealthAtMeanDailyPrice"));

            // file
            minWealthAllFile =  new EZGraphImpl(getZone(), "minWealthAll");
                minWealthAllFile.createMinSequence$withFeedFrom$andSelector
                ("txt", modelSwarm.getAgentList(),

SwarmUtils.getSelector("BasicSumAgent","getWealthAtMeanDailyPrice"));
            meanWealthAllFile =  new EZGraphImpl(getZone(), "meanWealthAll");

meanWealthAllFile.createAverageSequence$withFeedFrom$andSelector
                ("txt", modelSwarm.getAgentList(),

SwarmUtils.getSelector("BasicSumAgent","getWealthAtMeanDailyPrice"));
            maxWealthAllFile =  new EZGraphImpl(getZone(), "maxWealthAll");
                maxWealthAllFile.createMaxSequence$withFeedFrom$andSelector
                ("txt", modelSwarm.getAgentList(),
```

```
SwarmUtils.getSelector("BasicSumAgent","getWealthAtMeanDailyPrice"));
            minWealthRandomFile =  new EZGraphImpl(getZone(),
"minWealthRandom");

minWealthRandomFile.createMinSequence$withFeedFrom$andSelector
                  ("txt", modelSwarm.getRandomAgentList(),

SwarmUtils.getSelector("BasicSumAgent","getWealthAtMeanDailyPrice"));
            meanWealthRandomFile =  new EZGraphImpl(getZone(),
"meanWealthRandom");

meanWealthRandomFile.createAverageSequence$withFeedFrom$andSelector
                  ("txt", modelSwarm.getRandomAgentList(),

SwarmUtils.getSelector("BasicSumAgent","getWealthAtMeanDailyPrice"));
            maxWealthRandomFile =  new EZGraphImpl(getZone(),
"maxWealthRandom");

maxWealthRandomFile.createMaxSequence$withFeedFrom$andSelector
                  ("txt", modelSwarm.getRandomAgentList(),

SwarmUtils.getSelector("BasicSumAgent","getWealthAtMeanDailyPrice"));

            priceGraph = new EZGraphImpl(getZone (), "Current price",
                        "Ticks x days.", "Price.", "priceGraph");
            Globals.env.setWindowGeometryRecordName (priceGraph, "priceGraph");

            // the data for the priceGraph
            priceGraph.createSequence$withFeedFrom$andSelector
                  ("Current price", theBook,
                  SwarmUtils.getSelector(theBook,"getPrice"));

            // file
            priceFile =  new EZGraphImpl(getZone(), "price");
                  priceFile.createSequence$withFeedFrom$andSelector
                  ("txt", theBook, SwarmUtils.getSelector(theBook, "getPrice"));

            if (displayPreviousDayMean==1)
                  {
                  priceGraph.createSequence$withFeedFrom$andSelector
                        ("Day-1 mean p.", theBook,
                        SwarmUtils.getSelector(theBook,"getMeanPrice"));
                  // file
                  meanPriceFile =  new EZGraphImpl(getZone(), "meanPrice");
                        meanPriceFile.createSequence$withFeedFrom$andSelector
                        ("txt", theBook, SwarmUtils.getSelector(theBook,
                                                      "getMeanPrice"));
                  }

            return this;
            }

      /**
       * Create the actions necessary for the simulation. This is where
```

```
        * the schedule is built (but not run!)  Here we create a display
        * schedule - this is used to display the state of the world and
        * check for user input.
        */
       public Object buildActions()
            {
            Selector sel;

            // Use the parent class to begin the process.
            super.buildActions();

            // Call on the model swarm to build and schedule its actions.
            modelSwarm.buildActions();

            // Create an ActionGroup for display.  This is a list of
            // display actions that we want to occur at each step in
            // simulation time.
            // "doTkEvents", is required at the end to make everything
            // happen. We then check to see if modelSwarm has
            // told us to stop the simulation.
            displayActions = new ActionGroupImpl(getZone());

            // to update probes
            sel = SwarmUtils.getSelector(Globals.env.probeDisplayManager, "update");

displayActions.createActionTo$message(Globals.env.probeDisplayManager, sel);
            sel = SwarmUtils.getSelector(getActionCache(), "doTkEvents");
            displayActions.createActionTo$message(getActionCache(), sel);

            sel = SwarmUtils.getSelector(priceGraph, "step");
            displayActions.createActionTo$message(priceGraph, sel);

            if (savePriceData==1)
                  {
                  sel = SwarmUtils.getSelector(priceFile, "step");
                  displayActions.createActionTo$message(priceFile, sel);
                  if (displayPreviousDayMean==1)
                        {
                        sel = SwarmUtils.getSelector(meanPriceFile, "step");
                        displayActions.createActionTo$message(meanPriceFile, sel);
                        }
                  }

            if (showBookGraph==1)
                  {
                  sel = SwarmUtils.getSelector(bookGraph, "step");
                  displayActions.createActionTo$message(bookGraph, sel);
                  }

            if (saveBookData==1)
                  {
                  sel = SwarmUtils.getSelector(sellOrderFile, "step");
                  displayActions.createActionTo$message(sellOrderFile, sel);
                  sel = SwarmUtils.getSelector(buyOrderFile, "step");
                  displayActions.createActionTo$message(buyOrderFile, sel);
```

```
            }

        if (showAgentWealthGraph==1)
            {
            sel = SwarmUtils.getSelector(agentWealthGraph, "step");
            displayActions.createActionTo$message(agentWealthGraph, sel);
            }

        if (saveAgentWealthData==1)
            {
            sel = SwarmUtils.getSelector(minWealthAllFile, "step");
            displayActions.createActionTo$message(minWealthAllFile, sel);
            sel = SwarmUtils.getSelector(meanWealthAllFile, "step");
            displayActions.createActionTo$message(meanWealthAllFile, sel);
            sel = SwarmUtils.getSelector(maxWealthAllFile, "step");
            displayActions.createActionTo$message(maxWealthAllFile, sel);
            sel = SwarmUtils.getSelector(minWealthRandomFile, "step");
            displayActions.createActionTo$message(minWealthRandomFile, sel);
            sel = SwarmUtils.getSelector(meanWealthRandomFile, "step");
            displayActions.createActionTo$message(meanWealthRandomFile,
                                                                    sel);
            sel = SwarmUtils.getSelector(maxWealthRandomFile, "step");
            displayActions.createActionTo$message(maxWealthRandomFile, sel);
            }

        sel = SwarmUtils.getSelector(this, "checkToStop");
        displayActions.createActionTo$message(this, sel);

        // Finally, put the ActionGroup into a display schedule.
        // Note that the repeat interval is set by our
        // own Swarm data structure. The display is frequently the slowest part of a
        // simulation, when it is redraw less frequently things are faster
        // We can use here a display frequency lower (e.g.1) than the number of
        // steps in the model (step number = agentNumber) to display the prices of
        // each step-tick
        displaySchedule = new ScheduleImpl(getZone(), displayFrequency);
        displaySchedule.at$createAction(displayFrequency-1, displayActions);

        return this;
        }

    /**
     *  Activate the schedules so that they are ready to run.  The
     *  swarmContext argument is the zone in which the ObserverSwarm is
     *  activated.  Typically the ObserverSwarm is the top-level swarm,
     *  so it is activated in "null".  The other (sub)swarms and
     *  schedules will be activated inside of the ObserverSwarm
     *  context.
     */
     public Activity activateIn(Swarm swarmContext)
        {
        // Use the parent class to activate ourselves in the context
        // passed to us.
        super.activateIn(swarmContext);
```

```
        // Now activate the model swarm in the ObserverSwarm context.
        modelSwarm.activateIn(this);

        // Then activate the ObserverSwarm schedule in the
        // ObserverSwarm context.
        displaySchedule.activateIn(this);

        // Finally, return the activity we have built - the thing that
        // is ready to run.
        return getActivity();
        }

    /** To check for the stopping conditions. */
    public void checkToStop()
        {
        // if stopAtDayNumber is left to 0, the program will never stop
        // this stop occurs after the first tick of the time in modelSwarm,
        // but this fact does not affect the results in the observerSwarm
        // being the stop performed before the graph update
        if (stopAtDayNumber !=0 && stopAtDayNumber <=
                                        modelSwarm.getCurrentDay())
            {
            System.out.println("Stopping at day number" +
                                modelSwarm.getCurrentDay());
            // we can restart by pressing "Start" or "Next", but the simulation
            // will run for displayFrequency ticks and then it will stop again
            getControlPanel().setStateStopped();
            }
        }
    }
```

MODELSWARM.JAVA

```
import swarm.Globals;
import swarm.Selector;
import swarm.defobj.Zone;

import swarm.objectbase.Swarm;
import swarm.objectbase.SwarmImpl;
import swarm.objectbase.EmptyProbeMap;
import swarm.objectbase.EmptyProbeMapImpl;

import swarm.activity.ActionGroupImpl;
import swarm.activity.ScheduleImpl;
import swarm.activity.Activity;

import swarm.collections.ListImpl;
import swarm.collections.ListIndex;
import swarm.collections.ListShufflerImpl;
import swarm.collections.ArrayImpl;

/**
 * The Model of JavaSum. It comprises
 * the simulation.
```

```
 *
 * @author Marco Agagliate
 *
 */
public class ModelSwarm extends SwarmImpl
	{
	// Declare the model parameters and their default values.
	/** The number of the RandomAgents. */
	public int randomAgentNumber		= 10;
	/** The total number of agents. */
	public int  agentNumber			= randomAgentNumber;
	/** The total number of istants per day. */
	public int  istantNumber			= 10;
	/** The number of current day. */
	public int dayNumber			= 1;
	/** The asimmetry of buy and sell probability. */
	public double asymmetricBuySellProb	= 0.9;
	/** The coefficient that RandomAgents multiplays
	 *  by the last price for the order.
	 */
	public double  minCorrectingCoeff		= 0.9;
	/** The coefficient that RandomAgents multiplays
	 *  by the last price for the order.
	 */
	public double  maxCorrectingCoeff		= 1.1;
	/**The asimmetry of max/minCorrectingCoeff.  */
	public double  asymmetricRange		= 0.0;
	/** The probability of placing an order in the opening phase.
	 *  So a day starts without an empty book, with a realistic effect.
	 */
	public double  agentProbToActBeforeOpening	= 0.5;
	/** The minimum price at which the agents acts. */
	public double  floorP			= 0.3;
	/** The probability to act below floor price. */
	public double  agentProbToActBelowFloorP	= 0.5;
	/** The maximum number of order per agent. */
	public int maxOrderQuantity			= 3;
	/**  The percentage quota of agents which act in the market.*/
	public int percentageOfOperatingAgents	= 0;
	/**  The maximum number of operating agents. */
	public int maxNumberOfOperatingAgents	= 0;
	/**  The type of percetage of agents. If it is true percentage
	 *   is a maximum value; if it is false percentage is a
	 *   fixed value.
	 */
	public boolean typeOfPercentage		= false;

	/** If 1 many objects print data on the terminal window,
	  * 4 is used in BasicSumAgent. */
	public int printing			= 0;

	// Declare some other needed variables.
	/** The list of all the agents. */
	public ListImpl agentList;
	/** The list of the RandomAgents. */
```

```
public ListImpl randomAgentList;
/** The list of the agents in create order. */
public ListImpl indexAgentList;
/** Its iterator. */
public ListIndex indexAgentListIndex;

/** ActionGroup for holding an ordered sequence of action. */
public ActionGroupImpl  modelActions1,
                        modelActionsLS,
                        modelActions2,
                        modelActions3;
/** The Schedule operating in the Model. */
public ScheduleImpl modelSchedule;

/** The book of the simulation. */
public Book theBook;
/** The agents of the simulation. */
public RandomAgent anAgent1;

/** The randomRuleMaster manages RandomAgents. */
public RandomRuleMaster randomRuleMaster;
/** This is a "ghost agent".
 *  See the comments of CurrentAgent class.
 */
public CurrentAgent theCurrentAgent;
/**  The object for the calls of the agents. */
public CurrentIstant theCurrentIstant;

/** Constructor for a new ModelSwarm. */
public ModelSwarm(Zone azone)
      {
      // Use the parent class to create a top-level swarm.
      super(azone);

      // Build a customized probe map.  Without a custom probe map
      // the default is to show all variables and messages.  Here we
      // choose to customize the appearance of the probe, giving a
      // nicer interface.

      // Create the probe map and give it the ModelSwarm class.
      EmptyProbeMapImpl probeMap = new EmptyProbeMapImpl(azone,
                                                         getClass());

      // Now add probes for the variables we wish to probe.
      probeMap.addProbe(Globals.env.probeLibrary
            .getProbeForVariable$inClass("randomAgentNumber",
                  ModelSwarm.this.getClass ()));
      probeMap.addProbe(Globals.env.probeLibrary
            .getProbeForVariable$inClass("agentNumber",
                  ModelSwarm.this.getClass ()));
      probeMap.addProbe(Globals.env.probeLibrary
            .getProbeForVariable$inClass("istantNumber",
                  ModelSwarm.this.getClass ()));
      probeMap.addProbe(Globals.env.probeLibrary
            .getProbeForVariable$inClass("dayNumber",
```

```
                ModelSwarm.this.getClass ()));
        probeMap.addProbe(Globals.env.probeLibrary
            .getProbeForVariable$inClass("asymmetricBuySellProb",
                ModelSwarm.this.getClass ()));
        probeMap.addProbe(Globals.env.probeLibrary
            .getProbeForVariable$inClass("minCorrectingCoeff",
                ModelSwarm.this.getClass ()));
        probeMap.addProbe(Globals.env.probeLibrary
            .getProbeForVariable$inClass("maxCorrectingCoeff",
                ModelSwarm.this.getClass ()));
        probeMap.addProbe(Globals.env.probeLibrary
            .getProbeForVariable$inClass("asymmetricRange",
                ModelSwarm.this.getClass ()));
        probeMap.addProbe(Globals.env.probeLibrary
            .getProbeForVariable$inClass("agentProbToActBeforeOpening",
                ModelSwarm.this.getClass ()));
        probeMap.addProbe(Globals.env.probeLibrary
            .getProbeForVariable$inClass("floorP",
                ModelSwarm.this.getClass ()));
        probeMap.addProbe(Globals.env.probeLibrary
            .getProbeForVariable$inClass("agentProbToActBelowFloorP",
                ModelSwarm.this.getClass ()));
        probeMap.addProbe(Globals.env.probeLibrary
            .getProbeForVariable$inClass("maxOrderQuantity",
                ModelSwarm.this.getClass ()));
        probeMap.addProbe(Globals.env.probeLibrary
            .getProbeForVariable$inClass("percentageOfOperatingAgents",
                ModelSwarm.this.getClass ()));
        probeMap.addProbe(Globals.env.probeLibrary
            .getProbeForVariable$inClass("typeOfPercentage",
                ModelSwarm.this.getClass ()));
        probeMap.addProbe(Globals.env.probeLibrary
            .getProbeForVariable$inClass("printing",
                ModelSwarm.this.getClass ()));
        probeMap.addProbe(Globals.env.probeLibrary
            .getProbeForMessage$inClass("openProbeTo:",
                ModelSwarm.this.getClass ()));

        // And finaly install our probe map into the probeLibrary.
        // Note that this library was created by initSwarm().
        Globals.env.probeLibrary.setProbeMap$For(probeMap, getClass());
        }

    /** Build the model objects. */
    public Object buildObjects()
        {
        int i;

        // use the parent class buildObject() method to initialize the
        // process
        super.buildObjects();

        // Now create the model's objects.
        // randomRuleMaster
        randomRuleMaster = new RandomRuleMaster(getZone());
```

```
randomRuleMaster.setAgentProbToActBeforeOpening
                    ( agentProbToActBeforeOpening );
randomRuleMaster.setMinCorrectingCoeff( minCorrectingCoeff );
randomRuleMaster.setMaxCorrectingCoeff( maxCorrectingCoeff );
randomRuleMaster.setAsymmetricRange( asymmetricRange );
randomRuleMaster.setFloorP$andAgentProbToActBelowFloorP
                    ( floorP, agentProbToActBelowFloorP );

// Now create a List object to manage all the agent we are
// about to create.
agentList = new ListImpl(getZone());
randomAgentList = new ListImpl(getZone());
indexAgentList = new ListImpl(getZone());

if (agentNumber==0)
      {
      System.out.println("Nonsense: agentNumber cannot be 0");
      System.exit(1);
      }

theBook = new Book(getZone());
theBook.setAgentListIndex( indexAgentListIndex );
theBook.setAgentNumber( agentNumber );
theBook.setMaxOrderQuantity( maxOrderQuantity );

// a few checks
if (asymmetricBuySellProb<0.5)
      {
      System.out.println
      ("The asymmetricBuySellProb cannot be < 0.5 (internally set
                                                      to 0.5).\n");
      asymmetricBuySellProb=0.5;
      }

// randomAgent
for (i=1;i<=randomAgentNumber;i++)
      {
      anAgent1 = new RandomAgent(getZone(), maxOrderQuantity);
      anAgent1.setNumber( i );
      anAgent1.setMaxOrderQuantity( maxOrderQuantity );
      anAgent1.setBook( theBook );
      anAgent1.setRuleMaster( randomRuleMaster );
      anAgent1.setPrinting( printing );
      agentList.addLast(anAgent1);
      randomAgentList.addLast(anAgent1);
      indexAgentList.addLast(anAgent1);

      }
indexAgentListIndex = indexAgentList.listBegin(getZone());

// current agent
theCurrentAgent = new CurrentAgent(getZone());
theCurrentAgent.setAgentList( agentList );

// number of operating agents
```

```
        maxNumberOfOperatingAgents
              = percentageOfOperatingAgents*agentNumber/100;
        if (percentageOfOperatingAgents == 0)
              maxNumberOfOperatingAgents = 1;

        // current istant
        theCurrentIstant = new CurrentIstant(getZone());
        theCurrentIstant.setCurrentAgent( theCurrentAgent );
        theCurrentIstant.setMaxNumberOfOperatingAgents
                        ( maxNumberOfOperatingAgents );
        theCurrentIstant.setPercentageOfOperatingAgents
                        ( percentageOfOperatingAgents );
        theCurrentIstant.setTypeOfPercentage( typeOfPercentage );

        return this;
        }


/**
 * Here is where the model schedule is built, the data structures
 * that define the simulation of time in the model. The core is an
 * actionGroup that has a list of actions. Then that's put in a Schedule.
 */
public Object buildActions()
        {
        Selector sel;

        int i, j;

        ListShufflerImpl listShuffler = new ListShufflerImpl(getZone());

        // First, use the parent class to initialize the process.
        super.buildActions();

        // We create the list of simulation actions
        modelActions1 = new ActionGroupImpl(getZone());
        sel = SwarmUtils.getSelector("Book", "setClean");
        modelActions1.createActionTo$message(theBook, sel);
        sel = SwarmUtils.getSelector(listShuffler, "shuffleWholeList");
        modelActions1.createActionTo$message(listShuffler, sel, agentList);
        sel = SwarmUtils.getSelector("BasicSumAgent", "act0");
        modelActions1.createActionForEach$message(agentList, sel);

        //rimescolamento lista
        modelActionsLS = new ActionGroupImpl(getZone());
        sel = SwarmUtils.getSelector(listShuffler, "shuffleWholeList");
        modelActionsLS.createActionTo$message(listShuffler, sel, agentList);

        // acting in the market
        modelActions2 = new ActionGroupImpl(getZone());
        sel = SwarmUtils.getSelector("CurrentIstant", "callAgents");
        modelActions2.createActionTo$message(theCurrentIstant, sel);

        // accounting
        modelActions3 = new ActionGroupImpl(getZone());
```

```
        sel = SwarmUtils.getSelector("Book", "setMeanPrice");
        modelActions3.createActionTo$message(theBook, sel);
        sel = SwarmUtils.getSelector("BasicSumAgent", "act2");
        modelActions3.createActionForEach$message(agentList, sel);
        sel = SwarmUtils.getSelector(this, "increaseCurrentDayNumber");
        modelActions3.createActionTo$message(this, sel);

        // Then we create a schedule that executes the modelActions.
        // We use here agentNumber steps in each cycle, while
        // the observer usesa low display frequency (e.g. 1)
        // to show the price of each step-tickwe can also use a high d.f.
        // (e.g.1000 with 100 agents) to run a faster
        // a high d.f. (e.g.1000 with 100 agents) to run a faster simulation.

        modelSchedule = new ScheduleImpl(getZone(), istantNumber);
        modelSchedule.at$createAction(0, modelActions1);

        for (i=0;i<istantNumber;i++)
                {
                modelSchedule.at$createAction(i, modelActionsLS);
                modelSchedule.at$createAction(i, modelActions2);
                }
        modelSchedule.at$createAction(istantNumber-1, modelActions3);

        return this;
        }

/**
 * Now set up the model's activation. swarmContext indicates where
 * we're being started in - typically, this model is run as a
 * subswarm of an observer swarm. */
public Activity activateIn(Swarm swarmContext)
        {
        // Use the parent class to activate ourselves in the context
        // passed to us.
        super.activateIn(swarmContext);

        // Then activate the schedule in ourselves.
        modelSchedule.activateIn(this);

        // Finally, return the activity we have built.
        return getActivity();
        }

/** The method increases the number of the day. */
public void increaseCurrentDayNumber()
        {
        dayNumber++;
        if(printing==1)
                System.out.println("Day number " + dayNumber);
        }

/** The method returns the number of the day. */
public int getCurrentDay()
        {
```

```
        return dayNumber;
        }

    /** The method returns the list of the agents. */
    public ListImpl getAgentList()
        {
        return agentList;
        }

    /** The method returns the list of the RandomAgents. */
    public ListImpl getRandomAgentList()
        {
        return randomAgentList;
        }

    /** The method returns the index of indexAgentList. */
    public ListIndex getAgentListIndex()
        {
        return indexAgentListIndex;
        }

    /** The method returns the book. */
    public Book getBook()
        {
        return theBook;
        }

    /** The method opens the probe on an agent. */
    public void openProbeTo(int n)
        {
        BasicSumAgent anAgent;

        if (n>=1 && n<=agentNumber)
            {
            indexAgentListIndex.setOffset(n-1);
            anAgent = (BasicSumAgent)indexAgentListIndex.get();
            Globals.env.createArchivedProbeDisplay(anAgent, "anAgent");
            }
        }
    }
```

BASICSUMRULEMASTER.JAVA

```
import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

/**
 * This is the basic class of rule master, which is inherited from others.
 * A rule master contains the rule for the agent's acts.
 *
 *
 * @author Marco Agagliate
 *
```

```
 */
public class BasicSumRuleMaster extends SwarmObjectImpl
	{
	/**
	 * The probability of placing an order in the opening phase.
	 * So a day starts without an empty book, with a realistic effect.
	 */
	public double agentProbToActBeforeOpening;
	/**
	 * The coefficient that RandomAgents multiplays
	 * by the last price for the order.
	 */
	public double minCorrectingCoeff;
	/**
	 * The coefficient that RandomAgents multiplays
	 * by the last price for the order.
	 */
	public double maxCorrectingCoeff;
	/** The asimmetry of max/minCorrectingCoeff. */
	public double asymmetricRange;
	/** This is the floor price. */
	public double floorP;
	/**
	 * The probability that an agent would buy
	 * if the price is below floorP.
	 */
	public double agentProbToActBelowFloorP;

	/** Constructor for a new BasicSumRuleMaster. */
	public BasicSumRuleMaster(Zone aZone)
		{
		super(aZone);
		}

	/** This method sets the agentProbToActBeforeOpening. */
	public void setAgentProbToActBeforeOpening(double p)
		{
		agentProbToActBeforeOpening=p;
		}

	/** This method sets the minCorrectingCoeff. */
	public void setMinCorrectingCoeff(double min)
		{
		minCorrectingCoeff=min;
		}

	/** This method sets the setMaxCorrectingCoeff. */
	public void setMaxCorrectingCoeff(double max)
		{
		maxCorrectingCoeff=max;
		}

	/** This method sets the asymmetricRange. */
	public void setAsymmetricRange(double a)
		{
```

```
        asymmetricRange=a;
        }

    /** This method sets the floorP and AgentProbToActBelowFloorP. */
    public void setFloorP$andAgentProbToActBelowFloorP(double f, double p)
        {
        floorP=f;
        agentProbToActBelowFloorP=p;
        }
    }
```

BASICSUMAGENT.JAVA

```
import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

import swarm.objectbase.SwarmImpl;

/**
 * This is the basic class of agents, that is inherited from all the others.
 * It contains the common caracteristics of the agents.
 *
 *
 * @author Marco Agagliate
 *
 */
public class BasicSumAgent extends SwarmObjectImpl
    {
    /** This is the number of the agent. */
    public int number;
    /** The maximum number of order per agent. */
    public int maxOrderQuantity;
    /** The quantity of orders sending out by the agent.
      * There is one share per order:
      * the quantity of the order is the number of orders.
      */
    public int iMax;
    /** This is the number of satisfied orders. */
    public int executedPriceCount;
    /** If printing=1 many objects print data on the terminal
      * window; If printing=4 BasicSumAgents print data on
      * the terminal window.
      */
    public int printing;
    /** The price of the order. */
    public double price;
    /** The asimmetry of buy and sell probability. */
    public double asymmetricBuySellProb;
    /** The distribution of buy and sell order (fixed to 0.5). */
    public double buySellSwitch;
    /** The total agent's quantity of shares*/
    public double shareQuantity;
    /** The value of the shares at mean daily price. */
```

```
public double shareValueAtMeanDailyPrice;
/** The liquidity of the agent. */
public double liquidityQuantity;
/** The agent's wealth at mean daily price. */
public double agentWealthAtMeanDailyPrice;
/** The average price of the satified order. */
public double meanOperatingPrice;
/** The matrix of orders' prices. */
public Matrix executedPrices;
/** The book of the model. */
public Book theBook;

/** Constructor for a new BasicSumAgent. */
public BasicSumAgent(Zone aZone, int maxOrderQuantity)
      {
      super(aZone);
      executedPriceCount=0;
      executedPrices = new Matrix(getZone(), maxOrderQuantity);
      theBook = new Book(getZone());
      shareQuantity=0;
      shareValueAtMeanDailyPrice=0;
      liquidityQuantity=0;
      agentWealthAtMeanDailyPrice=0;
      }

/** This method sets the number of the agent. */
public void setNumber(int n)
      {
      number = n;
      }

/** This method sets the asymmetricBuySellProb. */
public void setAsymmetricBuySellProb(double p)
      {
      asymmetricBuySellProb=p;
      }

/** This method sets the maximum number of order per agent. */
public void setMaxOrderQuantity(int m)
      {
      maxOrderQuantity=m;
      }
/** This method sets the book. */
public void setBook(Book b)
      {
      theBook=b;
      }

/** Option about the agent print capability. */
public void setPrinting(int p)
      {
      printing=p;
      }

/** The number of executed prices is increased */
```

```
public void setConfirmationOfExecutedPrice(double p)
	{
	executedPrices.P$setFrom(executedPriceCount,p);
	if (p!=0) ++executedPriceCount;
	}

/**
 * This method exists only for the CurrentAgent.
 * Its implementation is defined into specify agent class.
 */
public void act0()
	{
	}

/**
 * This method exists only for the CurrentAgent.
 * Its implementation is defined into specify agent class.
 */
public void act1()
	{
	}

/**
 * This is the method that the agent calls at the end of the day.
 * The agent makes daily accounting.
 */
public void act2()
	{
	double mP, q;
	int i;

	if (printing==4)
		for(i=0;i<executedPriceCount;i++)
			System.out.println("I'm agent" + number
			+ "and the book told me: "
			+ executedPrices.P(i));
	mP=theBook.getMeanPrice();
	q=0;
	for (i=0;i<executedPriceCount;i++)
		{
		if (executedPrices.P(i) > 0) q++;
		if (executedPrices.P(i) < 0) q--;
		}
	shareQuantity += q;
	shareValueAtMeanDailyPrice=shareQuantity*mP;
	meanOperatingPrice=0;
	for (i=0;i<executedPriceCount;i++)
		{
		liquidityQuantity  -= executedPrices.P(i);
		meanOperatingPrice +=Math.abs(executedPrices.P(i));
		}
	if(executedPriceCount != 0)
		meanOperatingPrice/=executedPriceCount;
	else
		meanOperatingPrice=mP;
```

```
        if (printing==4)
            System.out.println("I'm agent " +  number
            + " and the meanOperatingPrice is: "
            +  meanOperatingPrice);
        agentWealthAtMeanDailyPrice
            =shareValueAtMeanDailyPrice+liquidityQuantity;
        }

    /** Return agent's wealth at mean daily price. */
    public double getWealthAtMeanDailyPrice()
        {
        return agentWealthAtMeanDailyPrice;
        }
    }
```

RANDOMRULEMASTER.JAVA

```
import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

/**
 * This is the RandomRuleMaster. A rule master contains the rule
 * for the agent's acts.
 *
 *
 * @author Marco Agagliate
 *
 */
public class RandomRuleMaster extends BasicSumRuleMaster
    {
    /** Constructor for a new RandomRuleMaster. */
    public RandomRuleMaster(Zone aZone)
        {
        super(aZone);
        }

    /**  This method is called before the market is open. */
    public double getPriceBeforeOpening(double lp, double p)
        {
        if (agentProbToActBeforeOpening < Globals.env.uniformDblRand
                                    .getDoubleWithMin$withMax(0.0, 1.0))
            return 0.0;
        else
            return this.getPrice(lp, p);
        }

    /**  This method is called  when the agent acts in the market. */
    public double getPrice(double lp, double p)
        {
        double price,  choice;

        price = lp * Globals.env.uniformDblRand.getDoubleWithMin$withMax
           (minCorrectingCoeff + asymmetricRange, maxCorrectingCoeff
```

```
                                                                    + asymmetricRange);
        choice=1;
        if (Globals.env.uniformDblRand.getDoubleWithMin$withMax(0.0, 1.0) > p)
            choice=-1;
        if (lp<floorP)
            {
            if (agentProbToActBelowFloorP > Globals.env.uniformDblRand
                                    .getDoubleWithMin$withMax(0.0, 1.0))
                {
                price=lp;
                choice=1;
                }
            }
        return choice*price;
        }
    }
```

RANDOMAGENT.JAVA

```
import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

/**
 * This is the RandomAgent. RandomAgent acts at random.
 *
 *
 *
 * @author Marco Agagliate
 *
 */
public class RandomAgent extends BasicSumAgent
    {
    /** This is the agent's rule master. */
    public RandomRuleMaster ruleMaster;

    /** Constructor for a new RandomAgent. */
    public RandomAgent(Zone aZone, int maxOrderQuantity)
        {
        super(aZone, maxOrderQuantity);
        }

    /** This method sets the rule master. */
    public void setRuleMaster(RandomRuleMaster r)
        {
        ruleMaster=r;
        }

    /** This method is called in the first phase of the day. */
    public void act0()
        {
        double lastPrice;
        int i;
```

```
        executedPriceCount=0;
        for (i=0;i<maxOrderQuantity;i++)
            executedPrices.P$setFrom(i, 0.0);
        buySellSwitch=0.5;
        lastPrice=theBook.getPrice();
        price = ruleMaster.getPriceBeforeOpening (lastPrice, buySellSwitch);
        if (maxOrderQuantity==1)
            iMax=1;
        else
            iMax=Globals.env.uniformIntRand.getIntegerWithMin$withMax
                                        (1, maxOrderQuantity);
        for (i=1;i<=iMax;i++)
            theBook.setOrderBeforeOpeningFromAgent (number, price);
        }

    /** This method is called in the first phase of the day.
     *  The agent acts in the market.
     */
    public void act1()
        {
        double lastPrice;
        int i;

        buySellSwitch=0.5;
        lastPrice=theBook.getPrice();
        price = ruleMaster.getPrice (lastPrice, buySellSwitch);
        if (maxOrderQuantity==1)
            iMax=1;
        else
            iMax=Globals.env.uniformIntRand.getIntegerWithMin$withMax
                                        (1, maxOrderQuantity);
        for (i=1;i<=iMax;i++)
            theBook.setOrderFromAgent (number, price);
        }
    }
```

CURRENTAGENT.JAVA

```
import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;
import swarm.collections.ListImpl;

/**
 * This is a ghost agent which simply transfers the 'act' message to
 * the first agent in the list agentList and rotate it
 * this trick is necessary to send the act message agentNumber times; so
 * the observer can dispaly the price of each acting step.
 *
 * @author Marco Agagliate
 *
 */
public class CurrentAgent extends SwarmObjectImpl
    {
```

```
	/** This is the agent list. */
	ListImpl agentList;

	/** Constructor for a new CurrentAgent. */
	public CurrentAgent(Zone aZone)
		{
		super(aZone);
		}

	/** The method sets agentList. */
	public void setAgentList(ListImpl l)
		{
		agentList = l;
		}

	/** The method calls the act1 method from an agent, which is in a list. */
	public void act1()
		{
		BasicSumAgent actingAgent;
		actingAgent = (BasicSumAgent)agentList.removeFirst();
		agentList.addLast( actingAgent );
		actingAgent.act1();
		}
	}
```

CURRENTISTANT.JAVA

```
import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

import swarm.collections.ListImpl;

/**
 * This is a class for calling a different number of agents
 * for every istant of simulating day.
 * The CurrentAgent is called a numberOfOperatinAgents times.
 *
 *
 * @author Marco Agagliate
 *
 */
public class CurrentIstant extends SwarmObjectImpl
	{
	/** The Current Agent of the simulation. */
	CurrentAgent theCurrentAgent;
	/** The number of operating agents per istant. */
	public int numberOfOperatingAgents;
	/**  The maximum number of operating agents. */
	public int maxNumberOfOperatingAgents;
	/**  The percentage quota of agents which act in the market.*/
	public int percentageOfOperatingAgents;
	/**  The type of percetage of agents. If it is true percentage
	 *   is a maximum value; if it is false percentage is a
```

```
 *   fixed value.
 */
public boolean typeOfPercentage;

/** Constructor for a new CurrentIstant. */
public CurrentIstant(Zone aZone)
      {
      super(aZone);
      }
/** The method sets theCurrentAgent. */
public void setCurrentAgent(CurrentAgent c)
      {
      theCurrentAgent = c;
      }

/** This method sets the maximum value of operating agents. */
public void setMaxNumberOfOperatingAgents(int n)
      {
      maxNumberOfOperatingAgents = n;
      }

/** This method sets the percentage of operating agents. */
public void setPercentageOfOperatingAgents(int p)
      {
      percentageOfOperatingAgents = p;
      }

/** This method sets the type of percentage. */
public void setTypeOfPercentage(boolean t)
      {
      typeOfPercentage = t;
      }


/** The method calls the CurrentAgent numberOfOperatingAgents times.
 *  The CurrentAgent calls act1 method from an agent, which is in a list.
 */
public void callAgents()
      {
      int j;

      numberOfOperatingAgents = maxNumberOfOperatingAgents;
      if (typeOfPercentage == true)
            {
            numberOfOperatingAgents =
                  Globals.env.uniformIntRand.getIntegerWithMin$withMax
                        (0 , maxNumberOfOperatingAgents);
            if (percentageOfOperatingAgents == 0) numberOfOperatingAgents = 1;
            }

      for (j=0;j<numberOfOperatingAgents;j++)
            theCurrentAgent.act1();
      }
}
```

SWARMUTILS.JAVA

```
import swarm.Globals;
import swarm.Selector;

   /** These two static methods create a selector.  A selector is an
    * object of the Selector class used by Swarm to encapsulate a
    * "message" destined for an object, where the message is the name
    * of a method defined for the class to which the object belongs.
    * Because the method must indeed be defined for the class of the
    * object and because this can be determined only at run time,
    * there is a possibility that the creation of the selector will
    * throw an exception if the class and the method do not match.
    * Java requires that events that might throw exceptions be
    * enclosed in try/catch blocks.  If there is an error creating
    * the new selector in the try block, the catch block can handle
    * the resulting exception.  Here we have taken a pretty crude
    * approach to handling the exception: we simply call
    * System.exit(1).  (Note that the "return null" which ends the
    * catch block is there only to tell the compiler that "return
    * sel" will never be reached if an exception occurs and sel is
    * undefined.  We'll exit on an exception before ever returning
    * sel to the calling method.) <br>
    *
    * Note that the getSelector method overloaded.  It can be
    * called with either a string containing the class name as its
    * first argument, or with an object of the desired class.  In the
    * first case, the string is converted to a class identifier using
    * the forName() method, while in the second case getClass() is
    * used to find the class identifier for the object.  The second
    * argument to getSelector is always a string containing the
    * method name.  (The boolean "false" at the end of the Selector
    * constructor is theobjCFlag. It allows one to use
    * ObjectiveC-type key/value method syntax.  Since we always use
    * Java-style method names, for us the flag is always false.)
    *
    * @author Charles P. Staelin
    */
public class SwarmUtils
{
    public static Selector getSelector(String name, String method)
    {
  Selector sel;

  try
     {
     sel = new Selector(Class.forName(name), method, false);
     } catch (Exception e)
            {
            System.err.println
                  ("There was an error in creating a Selector for method "
                        + method + "\nin Class " + name + ".");
            System.err.println (name + "." + method + " returns "
                        + e.getMessage());
            System.err.println("The process will be terminated.");
```

```
                System.exit(1);
                return null;
                }

    return sel;
     }

     public static Selector getSelector(Object obj, String method)
     {
    Selector sel;

    try
       {
       sel = new Selector(obj.getClass(), method, false);
       } catch (Exception e)
                {
                System.err.println
                        ("There was an error in creating a Selector for method "
                                + method + "\nin Class " +
                                        (obj.getClass()).getName() + ".");
                System.err.println ((obj.getClass()).getName() + "." + method
                                        + " returns " + e.getMessage());
                System.err.println("The process will be terminated.");
                System.exit(1);
                return null;
                }

    return sel;
     }

}
```

MATRIX.JAVA

```
import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

/**
 * This is the class for matrixes and vectors.
 *
 *
 *
 * @author Marco Agagliate
 *
 */
public class Matrix extends SwarmObjectImpl
        {
        /** This is a vector of double */
        public double[] vect;
        /** This is a matrix of double */
        public double[][] matr;

        /** Constructors for a new Matrix. */
```

```
	public Matrix(Zone aZone, int rows)
		{
		super(aZone);
		vect = new double[rows];
		}
	public Matrix(Zone aZone, int rows, int cols)
		{
		super(aZone);
		matr = new double[rows][cols];
		}

	/** This method sets a double in a position of the vector. */
	public void P$setFrom(int rows, double x)
		{
		vect[rows] = x;
		}

	/** This method returns a double from a position of the vector. */
	public double P(int rows)
		{
		return vect[rows];
		}

	/** This method sets a double in a position of the matrix. */
	public void R$C$setFrom(int rows, int cols, double x)
		{
		matr[rows][cols] = x;
		}

	/** This method returns a double from a position of the matrix. */
	public double R$C(int rows, int cols)
		{
		return matr[rows][cols];
		}
	}
```

BOOK.JAVA (PROVVISORIO)

Classe provvisoria utilizzata per le prove di funzionamento del codice.

```
import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

import swarm.collections.ListIndex;

public class Book extends SwarmObjectImpl
	{
	public Matrix order0, order1;
	public int i0, i1;
	public int maxOrderQuantity, agentNumber;
	public ListIndex indexAgentListIndex;

	// Constructor
	public Book(Zone aZone)
		{
```

```
        super(aZone);
        order0 = new Matrix (getZone(), 10000);
        order1 = new Matrix (getZone(), 10000);
        i0=0;
        i1=0;
        }

    public double getBuyOrderNumber()
        {
        return Globals.env.uniformDblRand.getDoubleWithMin$withMax(0, 30);
        }

    public double getSellOrderNumber()
        {
        return Globals.env.uniformDblRand.getDoubleWithMin$withMax(0, 30);
        }

    public double getPrice()
        {
        return Globals.env.uniformDblRand.getDoubleWithMin$withMax(0, 50);
        }

    public double  getMeanPrice()
        {
        return Globals.env.uniformDblRand.getDoubleWithMin$withMax(0, 50);
        }

    public void incrI(int i)
        {
        i++;
        }

    public void setMaxOrderQuantity(int m)
        {
        maxOrderQuantity=m;
        }

    public void setOrderFromAgent (int n, double p)
        {
        incrI(i0);
        order0.P$setFrom(i0, p);
        System.out.println("I'm agent " + n + ": order at price" + p);
        }

    public void setOrderBeforeOpeningFromAgent (int n, double p)
        {
        incrI(i1);
        order1.P$setFrom(i1, p);
        System.out.println("I'm agent " + n + ": order, before opening, at price" + p);
        }

    public void setClean ()
        {
        System.out.println("Book: clean");
        }
```

```
        public void setMeanPrice ()
                {
                System.out.println("Book: setMeanPrice");
                }

        public void setAgentListIndex(ListIndex i)
                {
                indexAgentListIndex=i;
                }

        public void setAgentNumber(int n)
                {
                agentNumber=n;
                }
        }
```

JAVASUM.SCM

```
(list
 (
  cons 'observerSwarm
       (
   make-instance 'ObserverSwarm
                 #:displayFrequency 1
                 #:stopAtDayNumber 4
                 #:displayPreviousDayMean 1
                 #:savePriceData 1
                 #:showBookGraph 1
                 #:saveBookData 0
                 #:showAgentWealthGraph 1
                 #:saveAgentWealthData 1
                 )
       )
 (
  cons 'modelSwarm
       (
   make-instance 'ModelSwarm
                 #:randomAgentNumber 10
                 #:istantNumber 10
                 #:asymmetricBuySellProb 0.9
                 #:minCorrectingCoeff 0.9
                 #:maxCorrectingCoeff 1.1
                 #:asymmetricRange 0.0
                 #:agentProbToActBeforeOpening 0.5
                 #:floorP 0.3
                 #:agentProbToActBelowFloorP 0.5
                 #:maxOrderQuantity 3
                 #:percentageOfOperatingAgents 0
                 #:typeOfPercentage #f
                 #:printing 0
                 )
       )
 )
```

DOC.BAT

```
@echo off
echo The file used j2sdk1.4.0
deltree doc
md doc
C:\j2sdk1.4.0\bin\javadoc *.java -author -d doc\
```

ALLEGATO C

JAVADOC: DOCUMENTAZIONE DEL CODICE

Class StartJavaSum

java.lang.Object

|

+--**StartJavaSum**

---

public class **StartJavaSum**
extends java.lang.Object

The StartJavaSum class contains main(). We follow here the typical Swarm structure with main() (in Start... as a convention) generating the Observer and the Observer generating the Model.

Author:
Marco Agagliate

Constructor Summary
StartJavaSum()

Method Summary	
static void	main(java.lang.String[] args) The main() function is the top-level

	place where everything starts.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

StartJavaSum

public **StartJavaSum**()

Method Detail

main

public static void **main**(java.lang.String[] args)
The main() function is the top-level place where everything starts.

Class ObserverSwarm

java.lang.Object
|
+--GUISwarmImpl
|
+--**ObserverSwarm**

---

public class **ObserverSwarm**
extends GUISwarmImpl

ObserverSwarm.java The observer swarm is collection of objects that are used to run and observe the ModelSwarm that actually comprises the simulation.

Author:
Marco Agagliate
See Also:
Serialized Form

Field Summary

EZGraphImpl	**agentWealth** our graphics or EZGraph output to files

EZGraphImpl	**agentWealthGraph** our graphics or EZGraph output to files
EZGraphImpl	**bookGraph** our graphics or EZGraph output to files
EZGraphImpl	**buyOrderFile** our graphics or EZGraph output to files
ActionGroupImpl	**displayActions** two ActionGroup for sequence of GUI events
int	**displayFrequency** Update frequency.
int	**displayPreviousDayMean** To create file and represent previous day mean price.
ScheduleImpl	**displaySchedule** the single Schedule instance
EZGraphImpl	**maxWealthAllFile** our graphics or EZGraph output to files
EZGraphImpl	**maxWealthRandomFile** our graphics or EZGraph output to files
EZGraphImpl	**meanPriceFile** our graphics or EZGraph output to files
EZGraphImpl	**meanWealthAllFile** our graphics or EZGraph output to files
EZGraphImpl	**meanWealthRandomFile** our graphics or EZGraph output to files
EZGraphImpl	**minWealthAllFile** our graphics or EZGraph output to files
EZGraphImpl	**minWealthRandomFile** our graphics or EZGraph output to files
ModelSwarm	**modelSwarm** the ModelSwarm we are observing
EZGraphImpl	**priceFile** our graphics or EZGraph output to files
EZGraphImpl	**priceGraph** our graphics or EZGraph

	output to files
int	**saveAgentWealthData** To save agent's wealth data.
int	**saveBookData** To save book data.
int	**savePriceData** To save on a file the price data.
EZGraphImpl	**sellOrderFile** our graphics or EZGraph output to files
int	**showAgentWealthGraph** To show agent's wealth graph.
int	**showBookGraph** To show book graph.
int	**stopAtDayNumber** To stop simulation at the end of a day.
Book	**theBook** The book of the simulation.

Constructor Summary

ObserverSwarm(Zone azone)
Constructor for a new ObserverSwarm.

Method Summary

Activity	**activateIn**(Swarm swarmContext) Activate the schedules so that they are ready to run.
java.lang.Object	**buildActions**() Create the actions necessary for the simulation.
java.lang.Object	**buildObjects**() Create the objects used to display the model.
void	**checkToStop**() To check for the stopping conditions.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

displayFrequency

public int **displayFrequency**
Update frequency.

stopAtDayNumber

public int **stopAtDayNumber**
To stop simulation at the end of a day.

displayPreviousDayMean

public int **displayPreviousDayMean**
To create file and represent previous day mean price.

savePriceData

public int **savePriceData**
To save on a file the price data.

showBookGraph

public int **showBookGraph**
To show book graph.

saveBookData

public int **saveBookData**
To save book data.

showAgentWealthGraph

public int **showAgentWealthGraph**
To show agent's wealth graph.

saveAgentWealthData

public int **saveAgentWealthData**
To save agent's wealth data.

modelSwarm

public ModelSwarm **modelSwarm**
the ModelSwarm we are observing

displaySchedule

public ScheduleImpl **displaySchedule**
the single Schedule instance

---

theBook

public Book **theBook**
The book of the simulation.

---

displayActions

public ActionGroupImpl **displayActions**
two ActionGroup for sequence of GUI events

---

priceGraph

public EZGraphImpl **priceGraph**
our graphics or EZGraph output to files

---

priceFile

public EZGraphImpl **priceFile**
our graphics or EZGraph output to files

---

bookGraph

public EZGraphImpl **bookGraph**
our graphics or EZGraph output to files

---

agentWealth

public EZGraphImpl **agentWealth**
our graphics or EZGraph output to files

---

agentWealthGraph

public EZGraphImpl **agentWealthGraph**
our graphics or EZGraph output to files

---

sellOrderFile

public EZGraphImpl **sellOrderFile**
our graphics or EZGraph output to files

---

buyOrderFile

public EZGraphImpl **buyOrderFile**
our graphics or EZGraph output to files

---

minWealthAllFile

public EZGraphImpl **minWealthAllFile**
our graphics or EZGraph output to files

---

meanWealthAllFile

public EZGraphImpl **meanWealthAllFile**
our graphics or EZGraph output to files

---

maxWealthAllFile

public EZGraphImpl **maxWealthAllFile**
our graphics or EZGraph output to files

---

minWealthRandomFile

public EZGraphImpl **minWealthRandomFile**
our graphics or EZGraph output to files

---

meanWealthRandomFile

public EZGraphImpl **meanWealthRandomFile**
our graphics or EZGraph output to files

---

maxWealthRandomFile

public EZGraphImpl **maxWealthRandomFile**
our graphics or EZGraph output to files

---

meanPriceFile

public EZGraphImpl **meanPriceFile**
our graphics or EZGraph output to files

Constructor Detail

ObserverSwarm

public **ObserverSwarm**(Zone azone)
Constructor for a new ObserverSwarm.

Method Detail

buildObjects

public java.lang.Object **buildObjects**()
Create the objects used to display the model.

---

buildActions

public java.lang.Object **buildActions**()

Create the actions necessary for the simulation. This is where the schedule is built (but not run!) Here we create a display schedule - this is used to display the state of the world and check for user input.

---

activateIn

public Activity **activateIn**(Swarm swarmContext)

Activate the schedules so that they are ready to run. The swarmContext argument is the zone in which the ObserverSwarm is activated. Typically the ObserverSwarm is the top-level swarm, so it is activated in "null". The other (sub)swarms and schedules will be activated inside of the ObserverSwarm context.

---

checkToStop

public void **checkToStop**()

To check for the stopping conditions.

Class ModelSwarm

java.lang.Object
 |
 +--SwarmImpl
     |
     +--**ModelSwarm**

---

public class **ModelSwarm**
extends SwarmImpl

The Model of JavaSum. It comprises the simulations.

Author:
Marco Agagliate
See Also:
Serialized Form

Field Summary	
ListImpl	**agentList** The list of all the agents.
int	**agentNumber** The total number of agents.
double	**agentProbToActBeforeOpening** The probability of placing an order in the opening phase.
double	**agentProbToActBelowFloorP** The probability to act below floor price.
RandomAgent	**anAgent1** The agents of the simulation.
double	**asymmetricBuySellProb** The asimmetry of buy and

	sell probability.
double	**asymmetricRange** The asimmetry of max/minCorrectingCoeff.
int	**dayNumber** The number of current day.
double	**floorP** The minimum price at which the agents acts.
ListImpl	**indexAgentList** The list of the agents in create order.
ListIndex	**indexAgentListIndex** Its iterator.
int	**istantNumber** The total number of istants per day.
double	**maxCorrectingCoeff** The coefficient that RandomAgents multiplays by the last price for the order.
int	**maxNumberOfOperatingAgents** The maximum number of operating agents.
int	**maxOrderQuantity** The maximum number of order per agent.
double	**minCorrectingCoeff** The coefficient that RandomAgents multiplays by the last price for the order.
ActionGroupImpl	**modelActions1** ActionGroup for holding an ordered sequence of action.
ActionGroupImpl	**modelActions2** ActionGroup for holding an ordered sequence of action.
ActionGroupImpl	**modelActions3** ActionGroup for holding an ordered sequence of action.
ActionGroupImpl	**modelActionsLS** ActionGroup for holding an ordered sequence of action.
ScheduleImpl	**modelSchedule** The Schedule operating in the Model.
int	**percentageOfOperatingAgents** The percentage quota of agents which act in the market.

int	**printing** If 1 many objects print data on the terminal window, 4 is used in BasicSumAgent.
ListImpl	**randomAgentList** The list of the RandomAgents.
int	**randomAgentNumber** The number of the RandomAgents.
RandomRuleMaster	**randomRuleMaster** The randomRuleMaster manages RandomAgents.
Book	**theBook** The book of the simulation.
CurrentAgent	**theCurrentAgent** This is a "ghost agent".
CurrentIstant	**theCurrentIstant** The object for the calls of the agents.
boolean	**typeOfPercentage** The type of percetage of agents.

Constructor Summary

ModelSwarm(Zone azone) Constructor for a new ModelSwarm.

Method Summary

Activity	**activateIn**(Swarm swarmContext) Now set up the model's activation. swarmContext indicates where we're being started in - typically, this model is run as a subswarm of an observer swarm.
java.lang.Object	**buildActions**() Here is where the model schedule is built, the data structures that define the simulation of time in the model.
java.lang.Object	**buildObjects**() Build the model objects.
ListImpl	**getAgentList**() The method returns the list of

	the agents.
ListIndex	**getAgentListIndex**() The method returns the index of indexAgentList.
Book	**getBook**() The method returns the book.
int	**getCurrentDay**() The method returns the number of the day.
ListImpl	**getRandomAgentList**() The method returns the list of the RandomAgents.
void	**increaseCurrentDayNumber**() The method increases the number of the day.
void	**openProbeTo**(int n) The method opens the probe on an agent.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

randomAgentNumber

public int **randomAgentNumber**

The number of the RandomAgents.

agentNumber

public int **agentNumber**

The total number of agents.

istantNumber

public int **istantNumber**

The total number of istants per day.

dayNumber

public int **dayNumber**

The number of current day.

asymmetricBuySellProb

public double **asymmetricBuySellProb**

The asimmetry of buy and sell probability.

minCorrectingCoeff

public double **minCorrectingCoeff**

The coefficient that RandomAgents multiplays by the last price for the order.

maxCorrectingCoeff

public double **maxCorrectingCoeff**

The coefficient that RandomAgents multiplays by the last price for the order.

asymmetricRange

public double **asymmetricRange**

The asimmetry of max/minCorrectingCoeff.

agentProbToActBeforeOpening

public double **agentProbToActBeforeOpening**

The probability of placing an order in the opening phase. So a day starts without an empty book, with a realistic effect.

floorP

public double **floorP**

The minimum price at which the agents acts.

agentProbToActBelowFloorP

public double **agentProbToActBelowFloorP**

The probability to act below floor price.

maxOrderQuantity

public int **maxOrderQuantity**

The maximum number of order per agent.

percentageOfOperatingAgents

public int **percentageOfOperatingAgents**

The percentage quota of agents which act in the market.

maxNumberOfOperatingAgents

public int **maxNumberOfOperatingAgents**

The maximum number of operating agents.

typeOfPercentage

public boolean **typeOfPercentage**

The type of percetage of agents. If it is true percentage is a maximum value; if it is false percentage is a fixed value.

---

printing

public int **printing**

If 1 many objects print data on the terminal window, 4 is used in BasicSumAgent.

---

agentList

public ListImpl **agentList**

The list of all the agents.

---

randomAgentList

public ListImpl **randomAgentList**

The list of the RandomAgents.

---

indexAgentList

public ListImpl **indexAgentList**

The list of the agents in create order.

---

indexAgentListIndex

public ListIndex **indexAgentListIndex**

Its iterator.

---

modelActions1

public ActionGroupImpl **modelActions1**

ActionGroup for holding an ordered sequence of action.

---

modelActionsLS

public ActionGroupImpl **modelActionsLS**

ActionGroup for holding an ordered sequence of action.

---

modelActions2

public ActionGroupImpl **modelActions2**

ActionGroup for holding an ordered sequence of action.

---

modelActions3

public ActionGroupImpl **modelActions3**

ActionGroup for holding an ordered sequence of action.

---

modelSchedule

public ScheduleImpl **modelSchedule**
The Schedule operating in the Model.

---

theBook

public Book **theBook**
The book of the simulation.

---

anAgent1

public RandomAgent **anAgent1**
The agents of the simulation.

---

randomRuleMaster

public RandomRuleMaster **randomRuleMaster**
The randomRuleMaster manages RandomAgents.

---

theCurrentAgent

public CurrentAgent **theCurrentAgent**
This is a "ghost agent". See the comments of CurrentAgent class.

---

theCurrentIstant

public CurrentIstant **theCurrentIstant**
The object for the calls of the agents.

Constructor Detail

ModelSwarm

public **ModelSwarm**(Zone azone)
Constructor for a new ModelSwarm.

Method Detail

buildObjects

public java.lang.Object **buildObjects**()
Build the model objects.

---

buildActions

public java.lang.Object **buildActions**()
Here is where the model schedule is built, the data structures that define the simulation of time in the model. The core is an actionGroup that has a list of actions. Then that's put in a Schedule.

---

activateIn

public Activity **activateIn**(Swarm swarmContext)

Now set up the model's activation. swarmContext indicates where we're being started in - typically, this model is run as a subswarm of an observer swarm.

---

increaseCurrentDayNumber

public void **increaseCurrentDayNumber**()

The method increases the number of the day.

---

getCurrentDay

public int **getCurrentDay**()

The method returns the number of the day.

---

getAgentList

public ListImpl **getAgentList**()

The method returns the list of the agents.

---

getRandomAgentList

public ListImpl **getRandomAgentList**()

The method returns the list of the RandomAgents.

---

getAgentListIndex

public ListIndex **getAgentListIndex**()

The method returns the index of indexAgentList.

---

getBook

public Book **getBook**()

The method returns the book.

---

openProbeTo

public void **openProbeTo**(int n)

The method opens the probe on an agent.

Class BasicSumAgent

java.lang.Object

|

+--SwarmObjectImpl

|

+--**BasicSumAgent**

Direct Known Subclasses:
RandomAgent

public class **BasicSumAgent**
extends SwarmObjectImpl

This is the basic class of agents, that is inherited from all the others.

It contains the common caracteristics of the agents.

Author:
Marco Agagliate
See Also:
Serialized Form

Field Summary	
double	**agentWealthAtMeanDailyPrice** The agent's wealth at mean daily price.
double	**asymmetricBuySellProb** The asimmetry of buy and sell probability.
double	**buySellSwitch** The distribution of buy and sell order (fixed to 0.5).
int	**executedPriceCount** This is the number of satisfied orders.
Matrix	**executedPrices** The matrix of orders' prices.
int	**iMax** The quantity of orders sending out by the agent.
double	**liquidityQuantity** The liquidity of the agent.
int	**maxOrderQuantity** The maximum number of order per agent.
double	**meanOperatingPrice** The average price of the satified order.
int	**number** This is the number of the agent.
double	**price** The price of the order.
int	**printing** If printing=1 many objects print data on the terminal window; If printing=4 BasicSumAgents print data on the terminal window.
double	**shareQuantity** The total agent's quantity of shares
double	**shareValueAtMeanDailyPrice** The value of the shares at mean daily price.

Book	**theBook** The book of the model.

Constructor Summary

BasicSumAgent(Zone aZone, int maxOrderQuantity) Constructor for a new BasicSumAgent.

Method Summary

void	**act0**() This method exists only for the CurrentAgent.
void	**act1**() This method exists only for the CurrentAgent.
void	**act2**() This is the method that the agent calls at the end of the day.
double	**getWealthAtMeanDailyPrice**() Return agent's wealth at mean daily price.
void	**setAsymmetricBuySellProb**(double p) This method sets the asymmetricBuySellProb.
void	**setBook**(Book b) This method sets the book.
void	**setConfirmationOfExecutedPrice**(double p) The number of executed prices is increased
void	**setMaxOrderQuantity**(int m) This method sets the maximum number of order per agent.
void	**setNumber**(int n) This method sets the number of the agent.
void	**setPrinting**(int p) Option about the agent print capability.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

number

public int **number**

This is the number of the agent.

---

maxOrderQuantity

public int **maxOrderQuantity**

The maximum number of order per agent.

---

iMax

public int **iMax**

The quantity of orders sending out by the agent. There is one share per order: the quantity of the order is the number of orders.

---

executedPriceCount

public int **executedPriceCount**

This is the number of satisfied orders.

---

printing

public int **printing**

If printing=1 many objects print data on the terminal window; If printing=4 BasicSumAgents print data on the terminal window.

---

price

public double **price**

The price of the order.

---

asymmetricBuySellProb

public double **asymmetricBuySellProb**

The asimmetry of buy and sell probability.

---

buySellSwitch

public double **buySellSwitch**

The distribution of buy and sell order (fixed to 0.5).

---

shareQuantity

public double **shareQuantity**
The total agent's quantity of shares

---

shareValueAtMeanDailyPrice

public double **shareValueAtMeanDailyPrice**
The value of the shares at mean daily price.

---

liquidityQuantity

public double **liquidityQuantity**
The liquidity of the agent.

---

agentWealthAtMeanDailyPrice

public double **agentWealthAtMeanDailyPrice**
The agent's wealth at mean daily price.

---

meanOperatingPrice

public double **meanOperatingPrice**
The average price of the satified order.

---

executedPrices

public Matrix **executedPrices**
The matrix of orders' prices.

---

theBook

public Book **theBook**
The book of the model.

Constructor Detail

BasicSumAgent

public **BasicSumAgent**(Zone aZone,
int maxOrderQuantity)
Constructor for a new BasicSumAgent.

Method Detail

setNumber

public void **setNumber**(int n)
This method sets the number of the agent.

---

setAsymmetricBuySellProb

public void **setAsymmetricBuySellProb**(double p)

This method sets the asymmetricBuySellProb.

---

setMaxOrderQuantity

public void **setMaxOrderQuantity**(int m)
This method sets the maximum number of order per agent.

---

setBook

public void **setBook**(Book b)
This method sets the book.

---

setPrinting

public void **setPrinting**(int p)
Option about the agent print capability.

---

setConfirmationOfExecutedPrice

public void **setConfirmationOfExecutedPrice**(double p)
The number of executed prices is increased

---

act0

public void **act0**()
This method exists only for the CurrentAgent. Its implementation is defined into specify agent class.

---

act1

public void **act1**()
This method exists only for the CurrentAgent. Its implementation is defined into specify agent class.

---

act2

public void **act2**()
This is the method that the agent calls at the end of the day. The agent makes daily accounting.

---

getWealthAtMeanDailyPrice

public double **getWealthAtMeanDailyPrice**()
Return agent's wealth at mean daily price.

Class BasicSumRuleMaster

java.lang.Object
|
+--SwarmObjectImpl
|

+--BasicSumRuleMaster

Direct Known Subclasses:
RandomRuleMaster

---

public class **BasicSumRuleMaster**
extends SwarmObjectImpl

This is the basic class of rule master, which is inherited from others.

A rule master contains the rule for the agent's acts.

Author:
Marco Agagliate
See Also:
Serialized Form

Field Summary	
double	**agentProbToActBeforeOpening** The probability of placing an order in the opening phase.
double	**agentProbToActBelowFloorP** The probability that an agent would buy if the price is below floorP.
double	**asymmetricRange** The asimmetry of max/minCorrectingCoeff.
double	**floorP** This is the floor price.
double	**maxCorrectingCoeff** The coefficient that RandomAgents multiplays by the last price for the order.
double	**minCorrectingCoeff** The coefficient that RandomAgents multiplays by the last price for the order.

Constructor Summary
BasicSumRuleMaster(Zone aZone) Constructor for a new BasicSumRuleMaster.

Method Summary	
void	**setAgentProbToActBeforeOpening**(double p) This method sets the agentProbToActBeforeOpening.
void	**setAsymmetricRange**(double a) This method sets the asymmetricRange.
void	**setFloorP$andAgentProbToActBelowFloorP**(

	double f, double p) This method sets the floorP and AgentProbToActBelowFloorP.
void	setMaxCorrectingCoeff(double max) This method sets the setMaxCorrectingCoeff.
void	setMinCorrectingCoeff(double min) This method sets the minCorrectingCoeff.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

agentProbToActBeforeOpening

public double **agentProbToActBeforeOpening**

The probability of placing an order in the opening phase. So a day starts without an empty book, with a realistic effect.

minCorrectingCoeff

public double **minCorrectingCoeff**

The coefficient that RandomAgents multiplays by the last price for the order.

maxCorrectingCoeff

public double **maxCorrectingCoeff**

The coefficient that RandomAgents multiplays by the last price for the order.

asymmetricRange

public double **asymmetricRange**

The asimmetry of max/minCorrectingCoeff.

floorP

public double **floorP**

This is the floor price.

agentProbToActBelowFloorP

public double **agentProbToActBelowFloorP**

The probability that an agent would buy if the price is below floorP.

Constructor Detail

BasicSumRuleMaster

public **BasicSumRuleMaster**(Zone aZone)
Constructor for a new BasicSumRuleMaster.

Method Detail

setAgentProbToActBeforeOpening

public void **setAgentProbToActBeforeOpening**(double p)
This method sets the agentProbToActBeforeOpening.

---

setMinCorrectingCoeff

public void **setMinCorrectingCoeff**(double min)
This method sets the minCorrectingCoeff.

---

setMaxCorrectingCoeff

public void **setMaxCorrectingCoeff**(double max)
This method sets the setMaxCorrectingCoeff.

---

setAsymmetricRange

public void **setAsymmetricRange**(double a)
This method sets the asymmetricRange.

---

setFloorP$andAgentProbToActBelowFloorP

public void **setFloorP$andAgentProbToActBelowFloorP**(double f,
double p)
This method sets the floorP and AgentProbToActBelowFloorP.

Class RandomAgent

java.lang.Object
|
+--SwarmObjectImpl
|
+--BasicSumAgent
|
+--**RandomAgent**

---

public class **RandomAgent**
extends BasicSumAgent

This is the RandomAgent. RandomAgent acts at random.

Author:
Marco Agagliate
See Also:
Serialized Form

Field Summary	
RandomRuleMaster	**ruleMaster** This is the agent's rule master.

Fields inherited from class BasicSumAgent
agentWealthAtMeanDailyPrice, asymmetricBuySellProb, buySellSwitch, executedPriceCount, executedPrices, iMax, liquidityQuantity, maxOrderQuantity, meanOperatingPrice, number, price, printing, shareQuantity, shareValueAtMeanDailyPrice, theBook

Constructor Summary
RandomAgent(Zone aZone, int maxOrderQuantity) Constructor for a new RandomAgent.

Method Summary	
void	**act0**() This method is called in the first phase of the day.
void	**act1**() This method is called in the first phase of the day.
void	**setRuleMaster**(RandomRuleMaster r) This method sets the rule master.

Methods inherited from class BasicSumAgent
act2, getWealthAtMeanDailyPrice, setAsymmetricBuySellProb, setBook, setConfirmationOfExecutedPrice, setMaxOrderQuantity, setNumber, setPrinting

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

ruleMaster

public RandomRuleMaster **ruleMaster**
This is the agent's rule master.

Constructor Detail

RandomAgent

public **RandomAgent**(Zone aZone,
int maxOrderQuantity)
Constructor for a new RandomAgent.

Method Detail

setRuleMaster

public void **setRuleMaster**(RandomRuleMaster r)
This method sets the rule master.

---

act0

public void **act0**()
This method is called in the first phase of the day.
Overrides:
act0 in class BasicSumAgent

---

act1

public void **act1**()
This method is called in the first phase of the day. The agent acts in the market.
Overrides:
act1 in class BasicSumAgent

Class RandomRuleMaster

java.lang.Object
|
+--SwarmObjectImpl
|
+--BasicSumRuleMaster
|
+--**RandomRuleMaster**

---

public class **RandomRuleMaster**
extends BasicSumRuleMaster

This is the RandomRuleMaster. A rule master contains the rule for the agent's acts.

Author:
Marco Agagliate

See Also:
Serialized Form

Field Summary

Fields inherited from class BasicSumRuleMaster

agentProbToActBeforeOpening, agentProbToActBelowFloorP, asymmetricRange, floorP, maxCorrectingCoeff, minCorrectingCoeff

Constructor Summary

RandomRuleMaster(Zone aZone)
Constructor for a new RandomRuleMaster.

Method Summary

double	**getPrice**(double lp, double p) This method is called when the agent acts in the market.
double	**getPriceBeforeOpening**(double lp, double p) This method is called before the market is open.

Methods inherited from class BasicSumRuleMaster

setAgentProbToActBeforeOpening, setAsymmetricRange, setFloorP$andAgentProbToActBelowFloorP, setMaxCorrectingCoeff, setMinCorrectingCoeff

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

RandomRuleMaster

public **RandomRuleMaster**(Zone aZone)
Constructor for a new RandomRuleMaster.

Method Detail

getPriceBeforeOpening

public double **getPriceBeforeOpening**(double lp,
double p)

This method is called before the market is open.

---

getPrice

public double **getPrice**(double lp,
double p)

This method is called when the agent acts in the market.

Class CurrentAgent

java.lang.Object
|
+--SwarmObjectImpl
|
+--**CurrentAgent**

---

public class **CurrentAgent**
extends SwarmObjectImpl

This is a ghost agent which simply transfers the 'act' message to the first agent in the list agentList and rotate it this trick is necessary to send the act message agentNumber times; so the observer can dispaly the price of each acting step.

Author:
Marco Agagliate
See Also:
Serialized Form

Constructor Summary
CurrentAgent(Zone aZone) Constructor for a new CurrentAgent.

Method Summary	
void	**act1**() The method calls the act1 method from an agent, which is in a list.
void	**setAgentList**(ListImpl l) The method sets agentList.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

CurrentAgent

public **CurrentAgent**(Zone aZone)
Constructor for a new CurrentAgent.

Method Detail

setAgentList

public void **setAgentList**(ListImpl l)
The method sets agentList.

---

act1

public void **act1**()
The method calls the act1 method from an agent, which is in a list.

Class CurrentIstant

java.lang.Object
|
+--SwarmObjectImpl
|
+--**CurrentIstant**

---

public class **CurrentIstant**
extends SwarmObjectImpl

This is a class for calling a different number of agents for every istant of simulating day. The CurrentAgent is called a numberOfOperatinAgents times.

Author:
Marco Agagliate
See Also:
Serialized Form

Field Summary

int	**maxNumberOfOperatingAgents** The maximum number of operating agents.

int	**numberOfOperatingAgents** The number of operating agents per istant.
int	**percentageOfOperatingAgents** The percentage quota of agents which act in the market.
boolean	**typeOfPercentage** The type of percetage of agents.

Constructor Summary

CurrentIstant(Zone aZone)
Constructor for a new CurrentIstant.

Method Summary

void	**callAgents**() The method calls the CurrentAgent numberOfOperatingAgents times.
void	**setCurrentAgent**(CurrentAgent c) The method sets theCurrentAgent.
void	**setMaxNumberOfOperatingAgents**(int n) This method sets the maximum value of operating agents.
void	**setPercentageOfOperatingAgents**(int p) This method sets the percentage of operating agents.
void	**setTypeOfPercentage**(boolean t) This method sets the type of percentage.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

numberOfOperatingAgents

public int **numberOfOperatingAgents**
The number of operating agents per istant.

maxNumberOfOperatingAgents

public int **maxNumberOfOperatingAgents**

The maximum number of operating agents.

---

percentageOfOperatingAgents

public int **percentageOfOperatingAgents**

The percentage quota of agents which act in the market.

---

typeOfPercentage

public boolean **typeOfPercentage**

The type of percetage of agents. If it is true percentage is a maximum value; if it is false percentage is a fixed value.

Constructor Detail

CurrentIstant

public **CurrentIstant**(Zone aZone)

Constructor for a new CurrentIstant.

Method Detail

setCurrentAgent

public void **setCurrentAgent**(CurrentAgent c)

The method sets theCurrentAgent.

---

setMaxNumberOfOperatingAgents

public void **setMaxNumberOfOperatingAgents**(int n)

This method sets the maximum value of operating agents.

---

setPercentageOfOperatingAgents

public void **setPercentageOfOperatingAgents**(int p)

This method sets the percentage of operating agents.

---

setTypeOfPercentage

public void **setTypeOfPercentage**(boolean t)

This method sets the type of percentage.

---

callAgents

public void **callAgents**()

The method calls the CurrentAgent numberOfOperatingAgents times. The CurrentAgent calls act1 method from an agent, which is in a list.

Class SwarmUtils

java.lang.Object
|
+--**SwarmUtils**

---

public class **SwarmUtils**
extends java.lang.Object

These two static methods create a selector. A selector is an object of the Selector class used by Swarm to encapsulate a "message" destined for an object, where the message is the name of a method defined for the class to which the object belongs. Because the method must indeed be defined for the class of the object and because this can be determined only at run time, there is a possibility that the creation of the selector will throw an exception if the class and the method do not match. Java requires that events that might throw exceptions be enclosed in try/catch blocks. If there is an error creating the new selector in the try block, the catch block can handle the resulting exception. Here we have taken a pretty crude approach to handling the exception: we simply call System.exit(1). (Note that the "return null" which ends the catch block is there only to tell the compiler that "return sel" will never be reached if an exception occurs and sel is undefined. We'll exit on an exception before ever returning sel to the calling method.)

Note that the getSelector method overloaded. It can be called with either a string containing the class name as its first argument, or with an object of the desired class. In the first case, the string is converted to a class identifier using the forName() method, while in the second case getClass() is used to find the class identifier for the object. The second argument to getSelector is always a string containing the method name. (The boolean "false" at the end of the Selector constructor is theobjCFlag. It allows one to use ObjectiveC-type key/value method syntax. Since we always use Java-style method names, for us the flag is always false.)

Author:
Charles P. Staelin

Constructor Summary

SwarmUtils()

Method Summary

static Selector	getSelector(java.lang.Object obj, java.lang.String method)
static Selector	getSelector(java.lang.String name, java.lang.String method)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

SwarmUtils

public **SwarmUtils**()

Method Detail

getSelector

public static Selector **getSelector**(java.lang.String name,
java.lang.String method)

getSelector

public static Selector **getSelector**(java.lang.Object obj,
java.lang.String method)

Class Matrix

java.lang.Object
|
+--SwarmObjectImpl
|
+--**Matrix**

---

public class **Matrix**
extends SwarmObjectImpl

This is the class for matrixes and vectors.

Author:
Marco Agagliate
See Also:
Serialized Form

Field Summary	
double[][]	**matr** This is a matrix of double
double[]	**vect** This is a vector of double

Constructor Summary
Matrix(Zone aZone, int rows) Constructors for a new Matrix.
Matrix(Zone aZone, int rows, int cols)

Method Summary	
double	**P**(int rows) This method returns a double from a position of the vector.
void	**P$setFrom**(int rows, double x) This method sets a double in a position of the vector.
double	**R$C**(int rows, int cols) This method returns a double from a position of the matrix.
void	**RCsetFrom**(int rows, int cols, double x) This method sets a double in a position

	of the matrix.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

vect

public double[] **vect**

This is a vector of double

---

matr

public double[][] **matr**

This is a matrix of double

Constructor Detail

Matrix

public **Matrix**(Zone aZone,
int rows)

Constructors for a new Matrix.

Matrix

public **Matrix**(Zone aZone,
int rows,
int cols)

Method Detail

P$setFrom

public void **P$setFrom**(int rows,
double x)

This method sets a double in a position of the vector.

P

public double **P**(int rows)

This method returns a double from a position of the vector.

RCsetFrom

public void **RCsetFrom**(int rows,

int cols,

double x)

This method sets a double in a position of the matrix.

R$C

public double **R$C**(int rows,

int cols)

This method returns a double from a position of the matrix.

BIBLIOGRAFIA

APPLE COMPUTER (2003), *The Objective-C Programming Language*. Apple Computer, Inc., Cupertino. Reperibile al sito http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC

ARTHUR, W. B., HOLLAND, J., LEBARON, B., PALMER, R. & TAYLER, P. (1997), *Asset pricing under endogenous expectations in an artificial stock market*, in W. B. Arthur, S. Durlauf & D. Lane, eds, '*The Economy as an Evolving Complex System II*', Addison-Wesley, Reading, MA, pp. 15–44.

BELTRATTI, A. & MARGARITA, S. (1992), *Evolution of trading strategies among heterogeneous artificial economic agents*, in J. A. Meyer, H. L. Roitblat & S. W. Wilson, eds, '*From Animals to Animats 2*', MIT Press, Cambridge, MA.

CAPPELLINI, A. N. (2003), *Esperimenti su mercati finanziari con agenti naturali e artificiali, tesi di laurea*, Facoltà di Economia, Torino.

CHURCHLAND, P. M., CHURCHLAND, P. S. (1990), *Può una macchina pensare? E' improponibile nell'ambito dell'IA classica, ma potrebbero farlo sistemi che imitano il cervello*, Le Scienze n.259, marzo 1990.

CHIARELLA (1992), CHIARELLA, C. (1992), *The dynamics of speculative behaviour*, Annals of Operations Research 37, 101–123.

COHEN, K. J., MAIER, S. F., SCHWARTZ, R. A. &WHITCOMB, D. K. (1983), *A simulation model of stock exchange trading*, Simulation 41, 181–191.

DAMILANO M., DE VINCENTIS P., ISAIA E., PIA P. (2002), *Il mercato azionario*. G. Giapichelli Editore, Torino.

DANIELS, M.G., FARMER, J.D., IORI, G., SMITH, E. (2002), *How storing supply and demand affects price diffusion*, Santa Fe working paper at www.santafe.edu/sfi/publications/Working-Papers/02-01-001.ps

ECKEL, B. (2002), *Thinking in Java.The definitive introduction to object-oriented programming in the language of the world wide web.- 3rd edition*. Prentice-Hall.

FRIEDMAN, M. (1953), The methodology of Positive Economics, in *Essay in positive economics*. University of Chicago Press, Chicago, pp.3-43.

FERRARIS, G. (2000), *Algoritmi genetici e classifier system nei modelli ad agenti*, http://www.swarm.org/comunity-contrib.html

GONÇALVES, C. P. (2003), *Artificial Financial Market Model*. http://ccl.northwestern.edu/netlogo/models/community/Artificial Financial Market Model.

GILBERT, N., TERNA, P. (2000), *How to built and use agent-based models in social sience*. Mind & Society, no. 1, pp.57-72.

HAYEK, F. (1950), *The sensory Order*. University of Chicago Press, Chicago.

HAYEK, F. (1967), *Studies in philosophy, politics and economics*. University of Chicago Press, Chicago.

HAUGELAND, J. (1988), *Intelligenza artificiale. Il significato di un'idea*. Superunniversale Bollati Bolighieri, Torino.

HOLLAND, J. J. (1975), *Adaptation in Natural and Artificial Systems*, MIT Press, Cambridge MA.

ISTRUZIONI (2003), *Istruzioni al regolamento dei mercati organizzati gestiti da Borsa Italiana S.p.A.*, 17 novembre 2003, Borsa Italiana S.p.A., Milano.

JHONSON-LARID, P. N. (1990), *La mente e il computer. Introduzione alla scienza cognitiva*. Il Mulino Biblioteca, Bologna.

JOHNSON, P. E. (2001), *What I learned from the Artificial Stock Market*. lark.cc.ukans.edu/~pauljohn/ResearchPapers/ASM01/ASM_Essay.pdf.

JOHNSON P., LANCASTER A., STEFANSSON B. (1999*). Swarm user guide. Relazione tecnica*, Santa Fe Institute, <http://www.santafe. edu/projects/swarm/swarmdocs/userbook/userbook.html>.

KILPATRICK, H. E. (2001), *Complexity, Spontaneous Order, and Friedrich Haiek: Are Spontaneous Order and Complexity Essentially the Same Thing?* Complexity, John Wiley & Sons, Vol.6, No.4.

KIRMAN, A. P. (1991), *Epidemics of opinion and speculative bubbles in financial markets*, in M. Taylor, ed., 'Money and Financial Markets', Macmillan, London.

LANGTON, G. C. (1992), *Vita artificiale*. Sistemi intelligenti / a. IV, n.2, agosto 2002.

LEBARON, B., ARTHUR, W. B. & PALMER, R. (1999), *Time series properties of an artificial stock market*, Journal of Economic Dynamics and Control 23, 1487–1516.

LEBARON, B. (2001), *A builder's guide to agent-based financial markets*, Quantitative Finance 1(2), 254–261.

LEBARON, B. (2002), *Building the Santa Fe Artificial Stock Market*.

LEGRENZI (2002) *Prima lezione di scienze cognitive*. Editori Laterza, Bari.

LEVY, M., LEVY, H. & SOLOMON, S. (1994), *A microscopic model of the stock market: cycles, booms, and crashes*, Economics Letters 45, 103–111.

LUX, T. (1997), *Time variation of second moments from a noise trader/infection model*, Journal of Economic Dynamics and Control 22, 1–38.

MARGARITA, S. (1992), *Verso un "robot oeconomicus": algoritmi genetici ed economia*. Sistemi Intelligenti /a.IV, n.3, dicembre 1992.

MARGARITA, S., SONNESSA, M. (2003), *Sim2Web: an Open Source system for web-enabling economic and financial simulations*. Journal of Artificial Societies and Social Simulation vol. 6, no.4
http://jasss.soc.surrey.ac.uk/6/4/12.html

MEZZERA, P. (2003), *Aste a chiamata, controllo dei prezzi ed esperimenti con agentiumani e artificiali in un modello di simulazione di borsa*, tesi di laurea, Facoltà di Economia, Torino.

MINSKY, M. E PAPERT, S. (1969), *Perceptrons – An Introduction to Computational Geometry*, Mit Press, Cambridge.

MINSKY, M. E PAPERT, S. (1988), *Perceptrons – An Introduction to Computational Geometry (Expanded edition)*, Mit Press, Cambridge.

MTA (2003), *MTA. Il Mercato Telematico Azionario di Borsa Italiana*, Borsa Italiana S.p.A., Milano.

NELSON, R., WINTER S. (1982), *An evolutionary theory of economic change*. Belknap, Cambridge, Mass. London.

NETLOGO(2003), *Net Logo 2.0.0 User Manual*, http://ccl.northwestern.edu/netlogo/

NEUBERG, L. E BERTELS, K. (2003), *Heterogeneous Trading Agents*, Complexity, John Wiley & Sons, Vol.8, No.5.

ORMEROD, P. (2003), *L'economia della farfalla. Società, mercato e comportamento*. Instar Libri, Torino.

PARISI, D. (1989), *Intervista sulle reti neurali. Cervello e macchine intelligenti*. Universale Paparbacks, Il Mulino, Bologna.

PARISI, D. (1999), *Mente. Nuovi modelli della vita artificiale*. Universale Paparbacks, Il Mulino, Bologna.

PARISI, D. (2001), *Simulazioni: la realtà rifatta nel computer*. Universale Paparbacks, Il Mulino, Bologna.

REFERENCE-BOOK (2000) *Swarm 2.1.1 Reference Guide. Documentazione tecnica di Swarm (versione Java)*, www.swarm.org.

REGOLAMENTO (2003), *Regolamento dei mercati organizzati e gestiti da Borsa Italiana S.p.A.*, Deliberato dalla Assemblea di Borsa Italiana S.p.A. del 29 aprile 2003 e approvato dalla Consob con delibera n. 14169 del 16 luglio 2003, Borsa Italiana S.p.A, Milano.

RIECK, C. (1994), *Evolutionary simulation of asset trading strategies*, in E. Hillebrand & J. Stender, eds, 'Many-Agent Simulation and Artificial Life', IOS Press.

ROSSER, J. B. (1999), *On the Complexity of Complex Economic Dynamics*. Journal of Economic Perspectives, Vol.13, No.4.

SEARLE,J. R. (1990), *La mente è un programma? NO. Un programma di calcolatore manipola simboli; il cervello annette un significato*, Le Scienze n.259, marzo 1990.

SIMON, H. A. (1984), *La ragione nelle vicende umane*. Il Mulino, Bologna.

SIMON, H. A. (1988), *Le scienze dell'artificiale*. Universale Paparbacks, Il Mulino, Bologna.

SIMON, H. A. (2000), *Scienza economica e comportamento umano*. Edizioni di Comunità, Torino.

SONNESSA, M. (2003), *JAS: Java Agent-based Simulation library. An open framework for algorithm-intensive simulations*.

TERNA, P. (1994), *Reti neurali artificiali e modelli con agenti adattivi*. XXV riunione scientifica annuale della società italiana degli economisti, Milano.

TERNA, P. (1996), *Economia e simulazione: una rivoluzione nel metodo*. Sistemi Intelligenti, pp. 496-502.

TERNA, P. (2000), *CT Scheme and ERA Scheme*. http://web.econ.unito.it/terna/ct-era/ct_era.html

TERNA, P. (2000a), *Economic experiments with swarm: a neural network approach to the self-development of consistency in agents' behavior*, F. Luna and B. Stefansson (eds.), *Economic Simulation in Swarm: Agent-Based Modelling and Object Oriented Programming*. Dordrecht and London: Kluwer Academic.

TERNA, P. (2000c), Hayek e il connessionismo: modelli con agenti che apprendono, in G. Clerico e S. Rizzello (eds.), *Il pensiero di Friedrich von Hayek*. Torino, Utet.

TERNA, P. (2000d), *SUM: a Surpricing (Un)realistic Market: Building a Simple Stock Market Structure with Swarm*. Presentato al CEF 2000, Barcellona, 5-8 giugno.

TERNA, P. (2001), Cognitive Agents Behaving in a Simple Stock Market Structure, in F. Luna and A. Perrone (eds), *Agent-Based Methods in Economics and Finance: Simulations in Swarm*. Kluwer Academic, Dordrecht and London, pp.188-227.

TERNA, P. (2002), *Consumatori e mercato*. Atlante del XXI secolo, in pubblicazione, UTET, Torino.

TERNA, P. (2002a), *Economic Simulation in Swarm: Agent-Based Modelling and Object Oriented Programming - by Benedikt Stefansson and Francesco Luna: A Rewiew and some Comments about "Agent Based Modeling"*, The Electronic Journal of Evolutionary Modeling and Economic Dynamics, n° 1013, http://www.e-iemed.org/1013/index.php

TERNA, P. (2002c), *La simulazione come strumento di indagine per l'economia*. Workshop su "Scienze Cognitive ed Economia" organizzato dalla Associazione Italiana di Scienze Cognitive, 21 settembre 2002, Rovereto.

TERNA, P. (2003), *Another ASM (Artificial Stock Marker), so AASM: Why?*, in pubblicazione.

WALDROP, M. M. (2001), *Complessità. Uomini e idee al confine tra ordine e caos*. Instar Libri, Torino.

WAN, H. A. E HUNTER, A. (1997), *On artificial adaptive agents models of stock markets*. Simulation 68:5, 279-289.