

UNIVERSITÀ DEGLI STUDI DI TORINO

Facoltà di Economia
Corso di Laurea in Economia e Commercio

Tesi di Laurea

L'impresa rifatta nel computer

Applicazione del modello jES al caso BasicNet

Relatore:

prof. Pietro Terna

Correlatore:

prof. Sergio Margarita

Candidato:

Francesco MERLO

ANNO ACCADEMICO 2001 - 2002

Alla mia famiglia

Ringraziamenti

Desidero ringraziare tutte le persone che mi hanno seguito in questo lavoro: la dott.sa Paola Bruschi di BasicNet, il relatore prof. Pietro Terna ed il correlatore prof. Sergio Margarita. Un particolare ringraziamento a Marco Lamieri, mio compagno di lavoro ed amico. Il ringraziamento più sentito va a tutta la mia famiglia ed ai miei amici.

Indice

Ringraziamenti	II
I La teoria	1
Introduzione	2
1 La rappresentazione della realtà	6
1.1 Fenomeni complessi ed emergenza	6
1.1.1 Sistemi lineari e non lineari	7
1.1.2 Complicato o complesso?	10
1.1.3 Fenomeni emergenti	11
1.1.4 Tra il margine del caos e i sistemi adattivi	13
1.1.5 La complessità e l'ordine spontaneo in Hayek	16
1.1.6 Perché studiare la complessità	18
1.2 La simulazione al computer	23
1.2.1 I problemi delle scienze sociali	24
1.2.2 I vantaggi della simulazione	26
1.2.3 I problemi delle simulazioni	29
1.3 Simulazioni ad agenti	30
1.3.1 Sugarscape	30
1.3.2 Individualismo metodologico e simulazione ad agenti	32
1.3.3 Lo schema ERA	35
Bibliografia	37
2 Lo sviluppo di una simulazione	39
2.1 La programmazione orientata agli oggetti	39
2.1.1 Breve storia della programmazione	39
2.1.2 La struttura di un programma	41
2.1.3 Il paradigma della programmazione ad oggetti	45
2.1.4 Il linguaggio di programmazione Java	50
2.2 La libreria Swarm	56
2.2.1 Il progetto e le sue finalità	56
2.2.2 Uno sciame di...	59

2.2.3	La gestione del tempo	60
2.2.4	Come costruire un modello	62
2.2.5	Le librerie a disposizione	66
2.3	La notazione UML	67
2.3.1	Perchè un linguaggio comune?	67
2.3.2	Il meta-modello	69
2.3.3	I diagrammi	73
	Bibliografia	82
3	Gli studi sull'impresa	83
3.1	La visione neoclassica	84
3.2	Schumpeter e la distruzione creatrice	87
3.3	L'impresa di Coase	88
3.4	L'imprenditore di Kirzner	90
	Bibliografia	93
II	jES e BasicJes	94
4	jES - Java Enterprise Simulator	95
4.1	Una duplice descrizione dell'impresa	95
4.1.1	Il modello	95
4.1.2	Che cosa fare?	100
4.1.3	Chi fa che cosa?	104
4.1.4	Un semplice esempio	106
4.2	Le funzionalità del simulatore	107
4.2.1	La gestione dei magazzini	107
4.2.2	Le capacità computazionali	109
4.2.3	La gestione della conoscenza	110
4.2.4	I layer	111
4.2.5	La contabilità	112
	Bibliografia	114
5	L'azienda da simulare: BasicNet	115
5.1	Breve storia dell'azienda	115
5.2	La BasicNet vista dall'interno	117
5.2.1	Introduzione al modello di business	117
5.2.2	Posizionamento dei marchi	118
5.2.3	Approvvigionamento dei prodotti	119
5.2.4	Sviluppo e gestione del Network	119
5.2.5	Fasi del processo organizzativo	120
5.2.6	Ruolo delle divisioni dotcom	122
	Bibliografia	125

6	Da BasicNet a BasicJes	126
6.1	Il processo di formalizzazione	127
6.1.1	Scopo della simulazione	127
6.2	Le fasi del processo di formalizzazione	129
6.2.1	Prima formalizzazione	129
6.2.2	Seconda formalizzazione	134
6.2.3	Terza formalizzazione	137
6.2.4	Quarta formalizzazione	140
6.2.5	Quinta formalizzazione	142
6.3	Proposte di modifica al codice	145
6.3.1	Prime proposte	145
6.3.2	Le modifiche apportate	147
6.4	BasicJes-0.9.7.30.b: BasicJes la simulazione di BasicNet	149
6.4.1	Dibattito sull'uso dei layers	149
6.4.2	Il modello: le unità	151
6.4.3	Il modello: le ricette	155
6.4.4	Il modello: le matrici di memoria	164
6.4.5	Il modello: la sequenza degli ordini	170
6.4.6	Il modello: i parametri della simulazione	172
6.5	Le modifiche al codice	174
6.5.1	Modifiche a "OrderDistiller.java"	174
6.5.2	Modifiche a "Recipe.java"	180
6.5.3	La classe "ComputationalAssembler.java"	181
6.6	Esperimenti e analisi dei risultati	197
6.6.1	Primo esperimento	199
6.6.2	Secondo esperimento	201
	Bibliografia	213
III	Conclusioni e appendici	214
	Conclusioni	215
	Appendici	221
	ComputationalAssembler.java	221
	OrderDistiller.java	233
	Recipe.java	238
	Modifiche a ExcelReader.java	244

Elenco delle tabelle

4.1	Formalismo di una generica ricetta	100
4.2	Formalismo di una generica ricetta per un prodotto non finito	101
4.3	Formalismo di un passo <i>sequential batch</i>	101
4.4	Formalismo di un passo <i>stand alone batch</i>	102
4.5	Formalismo di un passo <i>procurement</i>	102
4.6	Formalismo di un or	103
4.7	Formalismo delle unità	105
4.8	Formalismo delle unità complesse	105
4.9	Formalismo delle endUnit	106
4.10	Descrizione unità dell'esempio	106
4.11	Ricette dell'esempio	107
4.12	Sequenza ordini dell'esempio	107
4.13	Produzione dell'esempio	108
4.14	Formalismo degli oggetti computazionali	109
6.1	Esempio dei file "map" per coordinare gli articoli e le collezioni descritte in <i>orderSequences.xls</i>	144
6.2	Esempio di <i>orderSequences.xls</i>	144
6.3	Calendario stimato per le principali attività organizzative	156
6.4	Analisi del prezzo degli articoli	156
6.5	Analisi degli ordini dei singoli licenziatari in un semestre	157
6.6	Analisi degli ordini dei singoli licenziatari su una collezione grande in un semestre	158
6.7	Analisi degli ordini dei singoli licenziatari su una collezione piccola in un semestre	159
6.8	Numero articoli e quantità ordinate per le collezioni Kappa in un semestre	161
6.9	Numero articoli e quantità ordinate per le collezioni Robe di Kappa in un semestre	161
6.10	Stime dei tempi di produzione per ordine e singolo articolo	162
6.11	Stime dei tempi di produzione di ogni Trading Company per ordine e singolo articolo	163
6.12	Le ricette utilizzate nella simulazione	165
6.13	Matrice di memoria <i>designMatrix</i>	167
6.14	Matrice di memoria <i>agentMatrix</i>	168
6.15	Matrice di memoria <i>tcMatrix</i>	169

6.16	Matrice di memoria <i>basicNetMatrix</i>	169
6.17	Matrice di memoria <i>orMatrix</i>	170
6.18	Esempio di sequenza degli ordini per una collezione di 300 articoli	171
6.19	Esempio di sequenza degli ordini per una collezione di 30 articoli	172
6.20	Secondo esperimento: collezioni introdotte	203
6.21	Secondo esperimento: lancio delle collezioni	204
6.22	Secondo esperimento: le unità produttive	205
6.23	Secondo esperimento: ricette produttive	206
6.24	Secondo esperimento: sequenza del lancio degli ordini di una collezione principale	207
6.25	Secondo esperimento: sequenza del lancio degli ordini di una collezione SMU	207
6.26	Secondo esperimento: articoli programmati, disegnati e prodotti	209
6.27	Secondo esperimento: ripartizione dei costi tra le divisioni	210
6.28	Secondo esperimento: ripartizione dei costi in fissi e variabili per ogni divisione	210
6.29	Secondo esperimento: ripartizione costi per tick	211
6.30	Secondo esperimento: Ripartizione dei costi fissi per unità produttiva	211

Elenco delle figure

1.1	Rappresentazione di una rete neurale artificiale	19
1.2	Tabella dei guadagni del dilemma del prigioniero	21
1.3	Localizzazione delle divisioni di una famosa impresa multinazionale	21
1.4	Orbite caotiche	22
1.5	Mappa di Venezia	23
1.6	Sugarscape	31
1.7	Schema ERA	35
2.1	Il logo del progetto Swarm	57
2.2	Gli agenti di una simulazione: swarm e sub-swarm	60
2.3	Rappresentazione di un modello di simulazione in Swarm	62
2.4	Un esempio di <i>probe</i>	65
2.5	L'albero delle librerie di Swarm	66
2.6	Esempio di diagramma caso d'uso	75
2.7	Esempio di diagramma delle classi	76
2.8	Esempio di diagramma di sequenza	76
2.9	Esempio di diagramma di collaborazione	77
2.10	Esempio di diagramma di stato	78
2.11	Esempio di diagramma di attività	79
2.12	Esempio di diagramma dei componenti	80
2.13	Esempio di diagramma di distribuzione	80
4.1	Schema generale del modello jES	99
4.2	Le ricette	99
4.3	Passo di procurement	103
4.4	I magazzini	108
4.5	La gestione delle news	110
4.6	La contabilità nel modello jES	112
5.1	Loghi di Proprietà Basic Net	118
5.2	Organigramma del modello di business BasicNet	124
6.1	La simulazione BasicJVE in corso	198
6.2	Rapporto tra tempi della ricetta e tempi di produzione	200
6.3	Tempi di attesa minimi, massimi e medi degli ordini	201
6.4	Secondo esperimento: tempi di attesa minimi, massimi e medi degli ordini .	208
6.5	Secondo esperimento: rapporto tra tempi della ricetta e tempi di produzione	208
6.6	Secondo esperimento: costi, ricavi e utile totali	212

Listings

6.1	OrderDistiller.java: modifica 1	175
6.2	OrderDistiller.java: modifica 2	175
6.3	OrderDistiller.java: modifica 3	175
6.4	OrderDistiller.java: modifica 4	176
6.5	OrderDistiller.java: modifica 5	176
6.6	OrderDistiller.java: modifica 5	176
6.7	OrderDistiller.java: modifica 6	177
6.8	OrderDistiller.java: modifica 7	177
6.9	OrderDistiller.java: modifica 8	178
6.10	OrderDistiller.java: modifica 9	178
6.11	OrderDistiller.java: modifica 9	179
6.12	OrderDistiller.java: modifica 10	179
6.13	Recipe.java: modifica 1	180
6.14	Recipe.java: modifica 2	180
6.15	ComputationalAssembler.java: modifica 1	181
6.16	ComputationalAssembler.java: checkMatrixes	182
6.17	ComputationalAssembler.java: getCounter	183
6.18	ComputationalAssembler.java: logOpen	184
6.19	ComputationalAssembler.java: c1902	185
6.20	ComputationalAssembler.java: c1903	187
6.21	ComputationalAssembler.java: c1904	188
6.22	ComputationalAssembler.java: c1905	189
6.23	ComputationalAssembler.java: c1906	190
6.24	ComputationalAssembler.java: c1907	192
6.25	ComputationalAssembler.java: c1908	193
6.26	ComputationalAssembler.java: c1910	195
6.27	ComputationalAssembler.java: c1911	195
6.28	ComputationalAssembler.java: c1912	196
6.29	ComputationalAssembler.java	221
6.30	ComputationalAssembler.java	233
6.31	Recipe.java	238
6.32	ExcelReader.java: modifica 1	244
6.33	ExcelReader.java: modifica 2	244
6.34	ExcelReader.java: modifica 3	244

Parte I

La teoria

Introduzione

Il tema di questa tesi è l'applicazione degli strumenti di simulazione in contesti d'impresa; il lavoro, in particolare, è volto alla ri-costruzione dentro il computer di un'azienda reale. Utilizzando l'applicazione jES, sviluppata dal professore Pietro Terna, e grazie alla disponibilità della dottoressa Paola Bruschi, è stato possibile costruire un modello simulativo in grado di rappresentare le principali caratteristiche del *business system* dell'azienda torinese BasicNet.

La tesi è suddivisa in sei capitoli che, seguendo un'ideale percorso di approfondimento, affrontano il tema proposto sotto il punto di vista economico ed informatico. Si incomincia con una riflessione sulla natura delle scienze sociali e degli strumenti di simulazione che permettono di studiare i complessi fenomeni economici, fino a giungere alla metodologia utilizzata in questo lavoro, ai risultati ottenuti ed alle riflessioni maturate.

Con il primo capitolo si presenta il quadro scientifico entro il quale un'attività di questo tipo può essere collocata. Punto di partenza è la natura "complessa" che caratterizza tutti i fenomeni legati alle scienze dell'uomo, tra i quali l'attività d'impresa ne è un esempio.

Un sistema è "complesso" nel senso che non può essere spiegato isolando e studiando le singole parti che lo costituiscono; solo indagando sulle relazioni che legano i suoi componenti e le interazioni che da esse sorgono è possibile fornire delle spiegazioni alla maggior parte dei fenomeni che caratterizzano le scienze sociali. L'organizzazione di un'azienda, nel nostro caso particolare, è il frutto (emerge) delle interazioni che avvengono tra i soggetti (umani e non) che la compongono.

Per affrontare lo studio dei sistemi complessi nelle scienze sociali si propone, in seguito, la simulazione fatta attraverso il computer, intesa come una "terza via" che ogni ricercatore

ha a disposizione per esprimere le sue teorie. La simulazione, infatti, si affianca ai metodi già conosciuti, come quello verbale (utilizzato ad esempio per spiegare la storia) ed a quello matematico (tipico nelle scienze della natura). Oggi sono finalmente disponibili a costi ridotti, grazie ai continui progressi in campo informatico, degli straordinari laboratori virtuali che stanno dentro un comune computer da ufficio. Al loro interno è possibile riprodurre, osservare e spiegare fenomeni complessi come le imprese o i mercati di borsa, per i quali, fino ad ora, il metodo sperimentale era impraticabile.

Il capitolo si conclude con la descrizione del metodo di simulazione adottato per poter raggiungere l'obiettivo prestabilito. E' stata utilizzata una procedura detta "dal basso verso l'alto" adottando la metodologia fondata su agenti, particolarmente indicata per l'analisi dei fenomeni complessi e distribuiti.

Il secondo capitolo è un'esposizione delle tecniche e degli strumenti informatici utilizzati. Il modello jES è stato scritto in linguaggio Java, utilizzando a supporto le librerie Swarm. Java è un moderno linguaggio che possiede le caratteristiche della programmazione ad oggetti; diversamente da quanto avviene nella programmazione classica, seguendo il paradigma ad oggetti, il programmatore per costruire un'applicazione deve scomporre il problema in una serie di concetti (gli oggetti) che solo successivamente saranno uniti e fatti interagire. Questa caratteristica informatica richiama sia logicamente sia praticamente la struttura di un modello ad agenti. Swarm, invece, è una collezione di librerie informatiche per realizzare simulazioni di sistemi complessi ad agenti. Queste librerie semplificano notevolmente la scrittura del codice informatico e, soprattutto, mettono a disposizione un protocollo per la creazione dei modelli. Nell'ultima parte si presenta UML, un recente standard universale per la rappresentazione dei modelli informatici.

Nel terzo capitolo, prima di procedere nell'analisi del modello simulativo adottato, si affronta una riflessione su alcuni modelli, legati al contesto d'impresa, già presenti nella letteratura economica. Punto di partenza è la teoria dell'impresa neoclassica derivata dalla più ampia teoria dell'equilibrio economico generale descritta da Walras. Rimanendo nel contesto, si affrontano in seguito le principali critiche rivolte agli autori neoclassici avanzate da Schumpeter (l'introduzione delle innovazioni), Coase (la natura dell'impresa) e Kirzner

(la scoperta imprenditoriale).

Nel quarto capitolo è descritto il modello d'impresa simulata jES, in seguito utilizzato per ri-costruire la realtà osservata in BasicNet. Il modello permette di far "funzionare" l'azienda dentro il computer, poiché essa non è rappresentata semplicemente in modo animato sulla base di eventi previsti in precedenza dal programmatore; in jES gli eventi accadono "veramente", spesso in modo indipendente e generando, come nei sistemi complessi, interazioni imprevedute o imprevedibili.

L'azienda simulata nasce dall'unione di due descrizioni separate e fatte con formalismi differenti. Da un lato, utilizzando il formalismo delle ricette produttive, si descrive il "che cosa fare", dall'altro, con il formalismo delle unità produttive, il "chi fa che cosa". Dopo aver spiegato i due formalismi, nel capitolo si presentano le funzionalità del simulatore, prestando particolarmente attenzione agli sviluppi che sono stati necessari per raggiungere l'obiettivo di questa tesi.

Col quinto capitolo si fornisce una descrizione dell'azienda oggetto di studio. La BasicNet è proprietaria di alcuni marchi di abbigliamento come Kappa e Robe di Kappa; l'attività aziendale consiste nel diffondere e valorizzare questi marchi attraverso il suo *business system*. Il gruppo che fa capo a BasicNet ha impostato il proprio sviluppo su un modello di impresa "virtuale" a rete, puntando a non partecipare direttamente nell'attività produttiva, ma offrendo un servizio di intermediazione tra i principali soggetti che compongono il network: le *trading company* ed i licenziatari. A fronte di questa attività, l'azienda percepisce delle commissioni dai licenziatari commisurate alla merce da essi acquistata direttamente dalle trading company.

Nel sesto e ultimo capitolo si riportano, come in un diario di bordo, tutte le fasi di analisi e sviluppo che abbiamo affrontato, io ed il mio collega Marco Lamieri, per ottenere l'applicazione finale, da noi denominata BasicJes.

Quando abbiamo iniziato il lavoro il modello jES (inizialmente chiamato jVE) non era ancora sufficientemente maturo per affrontare la ri-costruzione di un'azienda "virtuale" come BasicNet, dove la maggior parte delle operazioni svolte sono immateriali (esempi sono le previsioni e gli ordini); il programma doveva però essere migliorato nella prospettiva di

riutilizzare le nuove funzionalità introdotte anche in altri contesti. Per questo motivo sono state avanzate, prima di impostare gli esperimenti, cinque diverse proposte di realizzazione che hanno comportato un lungo, ma molto interessante, periodo di analisi.

Il capitolo si conclude con due esperimenti finali, ognuno dei quali ha permesso di trarre delle riflessioni sia teoriche (sul modello utilizzato) sia pratiche (sui possibili miglioramenti da apportare alla nostra applicazione).

Capitolo 1

La rappresentazione della realtà

1.1 Fenomeni complessi ed emergenza

Gli esseri umani hanno sempre cercato di descrivere la realtà che li circonda: la letteratura, le sculture, i dipinti, i diorami, le carte geografiche, la fotografia, i filmati sono stati, negli anni, strumenti utili all'uomo per esprimere la sua conoscenza del mondo.

L'attività dell'uomo non è però solo volta a rappresentare ciò che lo circonda; il suo desiderio più grande è di comprendere e prevedere il comportamento di tutti i fenomeni che osserva.

Con questo fine crea i modelli. Un modello matematico, ad esempio, è un insieme di relazioni quantitative (anche una sola equazione) che descrivono in modo semplificato un certo gruppo di fenomeni. Tali relazioni sono usate sia nella formulazione della teoria, sia nella loro verifica empirica.

Quando si costruisce un modello bisogna decidere quali sono i fenomeni da spiegare (variabili endogene) e quali fenomeni interagiscono con i primi; questi ultimi saranno presenti nel modello ma non riceveranno una spiegazione esplicita (variabili esogene).

Se le relazioni che legano le variabili sono affette da errori, secondo le supposizioni di chi costruisce il modello, parliamo di modelli stocastici; al contrario quando le relazioni si ritengono prive di errori il modello è deterministico.

I modelli vengono poi distinti in statici e dinamici se l'effetto di una variabile su un'altra

è immediato o distribuito nel tempo.

In rapporto agli strumenti matematici utilizzati, infine, un modello può essere lineare o non lineare.

1.1.1 Sistemi lineari e non lineari

Chi costruisce un modello, col fine di trasmettere la sua conoscenza sui fenomeni naturali o sociali, tende solitamente a rappresentare la realtà come un sistema semplice; in questi modelli una singola causa produce un singolo effetto.

I sistemi semplici sono definiti lineari poiché il ruolo che ciascuna causa ha nel produrre l'effetto è separabile da quello delle altre cause. Le cause si sommano e si può isolare e prevedere l'effetto di una di esse senza preoccuparsi delle altre.

Tutti i sistemi semplici sono caratterizzati da alcune proprietà che possiamo così elencare (Parisi, 2001):

- è possibile prevedere gli stati futuri del sistema se sono noti gli stati precedenti;
- quando il sistema si trasforma nel tempo, è possibile prevedere la sua evoluzione futura;
- quando il sistema è perturbato da un evento esterno, l'effetto che tale perturbazione ha sul sistema è commisurato all'entità della perturbazione (lievi perturbazioni influenzano poco il sistema);
- se due sistemi analoghi partono da condizioni iniziali diverse, il loro sviluppo nel tempo sarà diverso quanto diverse sono le condizioni iniziali;
- il sistema non cambia il suo funzionamento in contesti diversi potendo così essere isolato dal contesto;
- nel sistema non ci sono rapporti di causazione reciproca; un elemento del sistema può influenzarne un altro, ma non viceversa;
- il sistema non fa parte di una gerarchia di sistemi che si influenzano reciprocamente;

- il sistema è composto da parti il cui ruolo nel determinare il comportamento complessivo del sistema è ben individuabile;
- il sistema può essere riprodotto in copie identiche.

I sistemi semplici sono di facile comprensione, ma nelle scienze economiche esistono fenomeni per i quali è difficile fornire spiegazioni e previsioni utilizzando i modelli lineari. Volendo rappresentare un'impresa è facile constatare che il modello non soddisferebbe nessuno dei punti sopra riportati: date le condizioni iniziali, è possibile prevedere la produzione futura a seguito di qualche cambiamento produttivo? La stessa impresa si comporterebbe allo stesso modo in contesti geografici diversi?

Per quanto riguarda l'economia, ma il discorso vale per tutte le discipline, si pensi ancora ai mercati di borsa o ad un distretto industriale; non esistono modelli per spiegare, in termini di causa ed effetto, perché si formano delle bolle speculative su certi titoli, o come può variare la natura di un distretto industriale a seguito di forti variazioni della domanda.

Il comportamento di questi sistemi è difficile da rappresentare e da comprendere poiché non può essere ricavato dalla semplice somma dei comportamenti delle singole parti. Fenomeni come le oscillazioni delle quotazioni di un titolo *emergono* sotto i nostri occhi perché dipendono dal modo specifico con cui si svolgono le interazioni tra le parti che compongono il sistema che stiamo osservando (le decisioni di investimento dei soggetti economici); anche la produzione di un'impresa *emerge* dalle interazioni tra gli individui che la compongono (ognuno con le sue capacità ed obiettivi). Non siamo in grado di spiegare le oscillazioni di un titolo di borsa studiando le caratteristiche di un unico investitore, e neppure lo studio meticoloso di una macchina per la produzione può spiegarci l'organizzazione di un'impresa.

Ultimamente suscita grande interesse lo studio di modelli non-lineari per la comprensione di quei sistemi per i quali non è possibile utilizzare semplificazioni concettuali in grado di ricondurre il fenomeno in esame ad un modello semplice. Le scienze della complessità sono un nuovo campo della scienza che si occupa di studiare e modellare sistemi di questo tipo.

Ruolo delle scienze della complessità è la comprensione di quei sistemi il cui comportamento non può essere compreso in maniera semplice (lineare) a partire dal comportamento

dei loro elementi.

Un sistema complesso presenta le seguenti caratteristiche (Parisi, 2001):

- non è possibile prevedere gli stati futuri del sistema pur conoscendo gli stati precedenti;
- il sistema si trasforma nel tempo ma la sua evoluzione è imprevedibile;
- quando il sistema è perturbato da un evento esterno, l'effetto che tale perturbazione ha sul sistema non può essere commisurato all'entità della perturbazione (lievi perturbazioni possono influenzare molto il sistema);
- il sistema è molto sensibile alle condizioni iniziali; due sistemi analoghi che partono da condizioni molto simili possono divergere notevolmente nel tempo;
- il sistema è sensibile al contesto in cui opera e quindi non può essere isolato dal suo ambiente;
- nel sistema ci sono rapporti di causazione reciproca: un elemento del sistema ne influenza un altro ed a sua volta è influenzato da questo;
- il sistema è inserito in una gerarchia di sistemi che si influenzano a vicenda;
- non è ben identificabile il ruolo che ciascun elemento di un sistema ha nel determinare il comportamento globale del sistema;
- il sistema non può essere riprodotto in copie identiche.

Sono stati gli studiosi di cibernetica e di teoria dell'informazione i primi ad occuparsi di complessità; ad essi si sono aggiunti, nel corso degli anni, pensatori provenienti da tutte le discipline che hanno reso, così, interdisciplinare la natura di questa scienza.

I sistemi complessi sono un settore di ricerca straordinariamente affascinante che trova applicazione praticamente in tutti i campi scientifici come la fisica, la chimica, la biologia, la medicina, l'economia, la sociologia e la psicologia.

Le scienze della complessità affrontano anche domini che hanno a lungo frustrato i tentativi di descrizione quantitativa rigorosa: in economia, ad esempio, concentrandosi sull'interazione fra agenti economici è finalmente possibile rinunciare alle irrealistiche ipotesi dell'economia classica, come il comportamento perfettamente razionale.

1.1.2 Complicato o complesso?

Prima di procedere in un'analisi delle teorie della complessità occorre chiarire la differenza, non solo linguistica, che corre fra due termini apparentemente simili: complicato e complesso.

Se, nel costruire un modello, siamo obbligati a prendere in considerazione un numero elevato di variabili, possiamo considerare "complesso" il fenomeno osservato? Oppure, se osservando lo sviluppo di un fenomeno riusciamo a mettere in evidenza alcune caratteristiche ricorrenti, possiamo affermare che esiste una struttura nascosta all'interno del sistema la cui spiegazione è semplice anche se nascosta sotto un'apparente difficoltà di lettura?

In questo campo di studi non è il numero di variabili che stabilisce la natura complessa di un fenomeno; la presenza di equazioni matematiche può rendere sì complicato il modello, ma non è indizio di complessità. Potremmo azzardarci ad affermare che un fenomeno modellabile anche attraverso equazioni "difficili" è così poco complesso che per descriverlo si può usare la matematica di cui già disponiamo.

Un confronto per chiarire la differenza che intercorre tra il termine complesso e complicato può essere fatto tra due sistemi indubbiamente "difficili" da comprendere come il motore di un'automobile ed il formicaio.

Il motore è composto da molti meccanismi, spesso anche sofisticati, ma il suo funzionamento, per quanto difficile da spiegare, è comunque il frutto della somma tra le parti che lo compongono, e può essere studiato attraverso una sua scomposizione in elementi sempre più piccoli.

Pensiamo poi ad un formicaio: una caratteristica affascinante è che la sua temperatura rimane costante, o con minime variazioni, sia in estate sia in inverno. Ma non possiamo

scomporre il formicaio e studiare una singola formica per spiegare il fenomeno; la temperatura costante del formicaio non rientra, infatti, tra gli obiettivi della formica. Il formicaio è un sistema complesso poiché da esso *emergono* fenomeni che non sono spiegabili con la semplice spiegazione delle caratteristiche delle sue parti.

Il contesto entro cui la scienza contemporanea parla di una scoperta della complessità si individua così nella scoperta del carattere *imprevedibile* di alcuni fenomeni, e nella compressione del fatto che (Bettelli, senza anno):

- nella scienza non esistono oggetti semplici, cioè la ricostruzione di un evento osservato *sembra* rispondere a leggi deterministiche ma va ben oltre queste leggi;
- la previsione dello stato futuro di un sistema può sembrare possibile, ma a costo di ridurre qualitativamente la portata del un fenomeno studiato;
- le qualità riscontrate in un oggetto studiato non sono proprie di quell'oggetto, ma sono la risposta della sua interazione con l'osservatore, sono il suo "modo di vederle".

Se, dunque, per studiare i sistemi lineari (il motore di un'automobile), si procede alla loro scomposizione ed allo studio analitico di ciascuna delle sue parti, questo non può avvenire per lo studio dei sistemi non-lineari (il formicaio).

L'economia, come tutte le scienze dell'uomo, sono ricche di sistemi il cui comportamento dipende dall'interazione delle parti più che dal comportamento delle parti stesse; l'impresa ne è solo un esempio ed è per questo motivo che, ai fini di questo lavoro, si propone un'approfondimento sui temi della complessità.

1.1.3 Fenomeni emergenti

Abbiamo visto che la realtà è ricca di sistemi che possono essere definiti complessi; ma la complessità che ci circonda manifesta sia aspetti inquietanti sia caratteristiche affascinanti.

La complessità è inquietante perché, al momento, ci impedisce di comprendere come e perché stiano accadendo certi fenomeni, ma soprattutto non ci permette di prevedere quali saranno gli sviluppi futuri (quanto varranno le mie azioni domani? A quando la prossima alluvione?).

Ma la complessità è anche affascinante perché ad essa è legata la comparsa di sorprendenti strutture organizzate a partire da semplici elementi e semplici regole. La nascita della vita stessa rappresenta probabilmente il più spettacolare esempio di comparsa di un elevato livello di organizzazione in un sistema complesso.

La non-linearità di certi sistemi si presta a spiegare il fenomeno dell'emergenza, cioè il processo grazie al quale, dall'interazione di una moltitudine di soggetti (formiche, investitori, molecole, ...), prende forma un'autorganizzazione spontanea.

Come quando si osserva la struttura di un fiocco di neve, ciò che maggiormente crea curiosità nei sistemi complessi (sia nei bambini, sia negli scienziati) è proprio la comparsa di queste "proprietà emergenti" che appaiono ad un certo punto della storia di un sistema complesso.

La nozione di proprietà emergenti di un sistema è uno dei concetti chiave di un lavoro collettivo di ricerca di natura spiccatamente interdisciplinare che vede coinvolti studiosi provenienti da molteplici ambiti della scienza.

Una delle grandi imprese scientifiche di questi anni è costituita proprio dal tentativo di costruire una teoria che, basandosi sui risultati ottenuti in diverse discipline, fornisca idee e metodi, purtroppo ancora lontani, per affrontare lo studio di queste proprietà.

Tinti (senza anno) esaminando il lavoro di Holland e le varie interpretazioni fatte da altri studiosi, definisce emergente ogni fenomeno che:

- è associato al funzionamento di un sistema complesso che evolve nel tempo;
- presenta contemporaneamente le seguenti proprietà:
 1. **è una novità**, un fenomeno che è descrivibile soltanto mediante un linguaggio *qualitativamente* diverso da quello utilizzato per descrivere il sistema e le sue componenti;
 2. **ha origine "dal basso all'alto"**, la sua formazione è dovuta esclusivamente alle interazioni *locali* tra le componenti del sistema;
 3. **è imprevedibile**, poiché le regole che descrivono il sistema negli stati locali presentano caratteristiche di *non-linearità*;

4. **non è scomponibile**, è indipendente dall'esistenza e dalle proprietà delle *singole* componenti del sistema.

E' facile intuire che i fenomeni emergenti sono gli stessi fenomeni che fanno sì che la nostra esistenza sia possibile e che sia proprio così come la conosciamo; è da questa intuizione che bisogna far risalire l'importanza della teoria della complessità e la sua natura di studio interdisciplinare dei fenomeni emergenti.

1.1.4 Tra il margine del caos e i sistemi adattivi

Data la sua natura interdisciplinare, non esiste un consenso generale, nella comunità scientifica, sulla definizione di sistema complesso e di complessità.

Spesso esso viene definito in negativo, come sinonimo di "non semplice, difficile da capire", ma definire complesso tutto ciò che in qualche modo ci sfugge non è particolarmente utile a connotare una classe interessante di sistemi.

E' comunque possibile tracciare alcune caratteristiche ricorrenti che si riscontrano in tutti i lavori delle scienze della natura e dell'uomo:

- **In ogni sistema complesso esistono tante componenti a loro volta più o meno complesse:** in generale, più numerosi e complessi sono i sotto-sistemi che lo compongono, più complesso è il sistema nel suo insieme.
- **Le componenti del sistema interagiscono tra loro:** le componenti si passano informazioni sotto diverse forme; la quantità di connessioni e la presenza di sotto-strutture ricorsive aumentano la complessità del sistema.
- **Il sistema non è interamente "governato":** se vi è un'unica componente che, da sola, governa il comportamento dell'intero sistema, questo non può essere complesso;

A queste proprietà si aggiunge un'ulteriore caratteristica che rende adattivi i sistemi complessi:

- **Il sistema si adatta all'ambiente:** il sistema è tanto più complesso, quanto più numerosi sono i fattori che influiscono sul suo adattamento all'ambiente (dei quali

deve tener conto il modello).

Come afferma Stein (1989):

Complexity is almost a theological concept; many people talk about it, but nobody knows what "it" really is. For example, organizations and systems are often called complex, not because they seen as dynamic and adaptive, but because they defy easy notions as to how they are organized or function.

Si possono individuare attualmente (Kilpatrick, 2001) due correnti di pensiero all'interno delle scienze della complessità. Oggetto di studio della prima corrente è quello che viene definito "margine del caos" (edge of chaos), mentre la seconda è associata al concetto dei "sistemi adattivi complessi" (complex adaptive system, CAS).

La corrente del "margine del caos" deriva principalmente dalle scienze matematiche e fisiche. La moderna teoria del caos (o del caos deterministico) può essere ricondotta ai lavori di Lorenz in campo meteorologico. Lorenz sviluppò un'importante teoria comunemente ribattezzata dell'"effetto farfalla"; semplificando la teoria, si afferma che un battito di ali di una farfalla in Cina potrebbe creare le condizioni necessarie perché si formi un uragano nel Nord America. Questo è un esempio di amplificazione di piccoli disturbi all'interno dei sistemi dinamici non lineari.

I modelli di Lorenz si rifanno alla teoria del caos, teoria nata quando la scienza classica non aveva più mezzi per spiegare gli aspetti irregolari e incostanti della natura; è una teoria scientifica che ha fatto la sua prima apparizione nello studio di fenomeni meteorologici, per poi allargarsi alle sperimentazioni fisiche, biologiche, matematiche, socio-economiche, fino ad arrivare ad essere sintetizzata nelle arti espressive (frattali).

Con il termine caos si indica generalmente la natura di situazioni complesse che sono proprie di qualsiasi settore scientifico, dalla turbolenza dei fluidi alle fluttuazioni delle popolazioni. In termini matematici la teoria del caos ha principalmente origine dalla dinamica lineare, dallo studio dei fenomeni retti da equazioni differenziali non lineari dove sistemi anche molto semplici possono manifestare una notevole complessità di comportamento.

I sistemi caotici sono infatti considerati complessi, sebbene essi possano anche avere

pochi gradi di libertà: la nozione di complessità suggerita in questo caso comprende anche sistemi di questo tipo, che non potrebbero essere inclusi in una definizione di complessità che richiedesse la presenza di un gran numero di elementi interagenti.

Ma anche un sistema caotico può, ad un certo punto della sua storia, presentare delle regolarità, può apparentemente non risultare più caotico come se si adattasse all'ambiente che lo ospita. Leggiamo in Waldrop (1992):

... at a kind of abstract phase transition called "the edge of Chaos," you also find complexity: a class of behaviors in which the components of the system never quite lock into place, yet never quite dissolve into turbulence, either. These are systems that are both stable enough to store information, and yet evanescent enough to transmit it. These are the systems that can be organized to perform complex computations, to react to the world, to be spontaneous, adaptive, and alive.

Differente è invece l'orientamento in questo campo delle scienze legate alla biologia: il loro interesse è la descrizione dei sistemi adattivi complessi (CAS), sistemi che hanno la proprietà di adattarsi al suo ambiente per perseguire i propri scopi. Sistemi che non necessariamente presentano comportamenti caotici.

Un'impresa, una multinazionale, un mercato borsistico, un ospedale, internet, un distretto industriale, possono essere visti come dei sistemi adattivi complessi.

Ogni sistema adattivo complesso può essere costituito da altri sistemi analoghi al suo interno e la sua evoluzione è legata ad essi. Le parti che compongono un sistema adattivo complesso possono agire in modo imprevedibile e gli effetti delle loro azioni sono influenzate dalle azioni delle altre parti.

Un mercato borsistico, ad esempio, è composto dai venditori, i compratori, le società quotate e le organizzazioni regolatrici del mercato. Ogni attore del sistema opera secondo le sue proprie strategie o modelli mentali, questi possono essere condivisi con altri attori o essere individuali, ma il formarsi di bolle speculative emerge dall'interazione tra gli attori.

Un sistema adattivo complesso presenta caratteristiche per le quali è praticamente

imprevedibile la sua evoluzione, tuttavia, in alcuni casi, è possibile identificare comportamenti ciclici e situazioni ricorrenti che permettono una certa prevedibilità a corto termine e a lungo termine.

Per quanto concerne la ricerca scientifica in campo economico, Arthur, Durlauf e Lane (1997) suggeriscono sei caratteristiche che i modelli economici dovrebbero presentare, sebbene esistano molte difficoltà ad essere trattate con i tradizionali strumenti matematici:

1. interazione dispersa tra gli agenti che compongono il sistema;
2. nessuna capacità di controllo del modello;
3. organizzazioni gerarchiche multi-livello che si intersecano;
4. adattamento continuo degli agenti che sono in grado di imparare ed evolvere;
5. innovazioni continue (nuovi mercati, tecnologie, ...);
6. dinamica (intesa come sequenza di modificazioni del sistema nel tempo) priva di un equilibrio globale, ma con molti punti di equilibrio instabili.

Sistemi che presentano queste caratteristiche sono definiti dagli autori come "adaptive nonlinear networks".

1.1.5 La complessità e l'ordine spontaneo in Hayek

Quando si propongono in un contesto economico i temi della complessità e dei fenomeni emergenti ad essa collegati, risulta naturale tracciare un collegamento con le opere dell'economista, psicologo e politologo Friedric von Hayek. Le maggiori analogie si presentano con la teoria dell'ordine spontaneo.

La teoria dell'ordine spontaneo di Hayek fu sviluppata per cercare una soluzione alle difficoltà logiche ed alla scarsa rilevanza empirica che la teoria neoclassica dell'equilibrio stazionario presentava nell'elaborazione di una qualsiasi teoria del ciclo, del capitale e della moneta.

In questi modelli, ancorati al concetto di equilibrio, risulta difficile affrontare i problemi dell'investimento, del risparmio e dell'accumulazione del capitale e neppure quelli della

moneta, del credito e delle attività finanziarie; praticamente non è possibile alcun tipo di analisi sulle scelte intertemporali degli agenti.

Per trovare una via d'uscita a questo concetto, Hayek introdusse il concetto di ordine che, a differenza dell'equilibrio, può coesistere con un certo grado di disequilibrio; l'ordine secondo Hayek è distinto in:

Ordine spontaneo: è quell'ordine che si forma per evoluzione ed è in grado di perpetuarsi e di autoriprodursi grazie al meccanismo endogeno che ne regola il funzionamento. Le strutture relazionali presenti in un ordine spontaneo non sono direttamente e facilmente comprensibili e il comportamento del sistema non è riconducibile alla volontà degli individui.

Organizzazione: è un ordine costruito artificialmente da una o più persone. Ne consegue che un'organizzazione è un tipo di ordine relativamente semplice con semplici strutture relazionali. In un'organizzazione ciascun individuo ha un suo ruolo determinato e ha dei compiti da svolgere mentre un ordine spontaneo consente agli individui che ne fanno parte di perseguire i propri fini particolari. Se un'organizzazione può essere facilmente usata per scopi semplici e limitati, l'ordine spontaneo arriva là dove l'organizzazione non può arrivare.

Secondo Hayek, pianificare centralmente l'economia, creare quindi un'*organizzazione* economica, presuppone l'individuazione di fini semplici e limitati in grado di sussistere con l'organizzazione stessa; ma gli individui hanno fini molteplici e spesso mutualmente incompatibili. L'unico modo per superare questo ostacolo è quello di privare i singoli della libertà di perseguire i propri fini e di promuovere una gerarchia di fini compatibili con l'organizzazione. Ma la creazione di un'organizzazione economica comporta anche una perdita di benessere, dato che in un'organizzazione le conoscenze che possono essere accumulate si riducono necessariamente a quelle concentrabili nell'organo di controllo preposto alla formulazione del piano. Le conoscenze di un'economia di mercato invece variano da individuo a individuo, sono disperse fra milioni di persone diverse, mutano nel tempo e nello spazio e pertanto è impossibile poterle concentrare o codificare.

Hayek sottolinea anche che spesso le conoscenze sono tacite, gli individui non sanno di possederle; ciò che spinge i singoli a scoprirle è il meccanismo della concorrenza, è grazie alle variazioni dei prezzi che gli imprenditori scoprono i differenti modi di ridurre i costi di produzione, modi che non erano loro noti prima che la concorrenza li spingesse a ricercarli.

Il mercato concorrenziale è quindi una "procedura di scoperta", è un "processo dinamico" che favorisce la diffusione di informazioni. In un ordine concorrenziale di mercato un certo grado di disequilibrio è indispensabile per il suo funzionamento, è il disequilibrio che provoca il cambiamento dei prezzi che segnala agli agenti come modificare i loro piani di azione e che attiva l'incentivo alla scoperta e alla diffusione delle informazioni.

Bisogna ammettere che il collegamento tra l'immensa opera di Hayek e gli studi sulla complessità è molto forte. Kilpatrick (2001) sottolinea quali sono, al momento, le differenze che intercorrono tra i più autorevoli studiosi delle scienze della complessità (Arthur, Durlauff, Lane, 1997) ed il pensiero di Hayek:

Heyek's theory of spontaneous order meshes meritoriously with complexity theorist, and would do well to study his works. However, his writings and those of complexity theorist appear to be qualitatively different. Belivers of spontaneous order find a benevolent force creating a more efficient system than humans could devise by planning. Belivers of complex adaptive behaviour, or complexity theory, find that forces may at time be malevolent and that the collective actions of humans may be necessary to return to optimum efficiency.

Sebbene quindi l'intuizione iniziale sia comune, esistono forti differenze sulle conclusioni e gli sugli studi da intraprendere.

1.1.6 Perché studiare la complessità

Il comportamento umano è sicuramente complesso e la sua complessità può essere fatta risalire all'ambiente nel quale l'uomo si trova.

Si pensi a come la rivoluzione dei mezzi di calcolo e di comunicazione sta portando ad un aumento senza precedenti dell'ampiezza e della intensità delle interazioni tra persone, organizzazioni e istituzioni; l'interazione tra soggetti di un sistema è la principale fonte di

complessità. Il fenomeno è particolarmente evidenziabile nei contesti d'impresa dove, al giorno d'oggi le interazioni (a livello di persone, unità produttive o tra imprese stesse) sono notevolmente aumentate.

Le scienze della complessità intendono contribuire allo sviluppo di una nuova cultura in grado di descrivere un mondo (il nostro *vero* mondo) fatto di sistemi che cambiano, sistemi che nella loro storia presentano crisi e cambiamenti repentini, sistemi che evolvono, si adattano e apprendono, o, ancora, un mondo fatto da sistemi che insieme co-esistono e, soprattutto, che co-evolvono.

Per saggiare la natura multidisciplinare delle teorie della complessità, ma soprattutto per constatare quanto queste teorie abbiano un campo di applicazione vastissimo si presentano di seguito alcuni esempi.¹

Le reti neurali

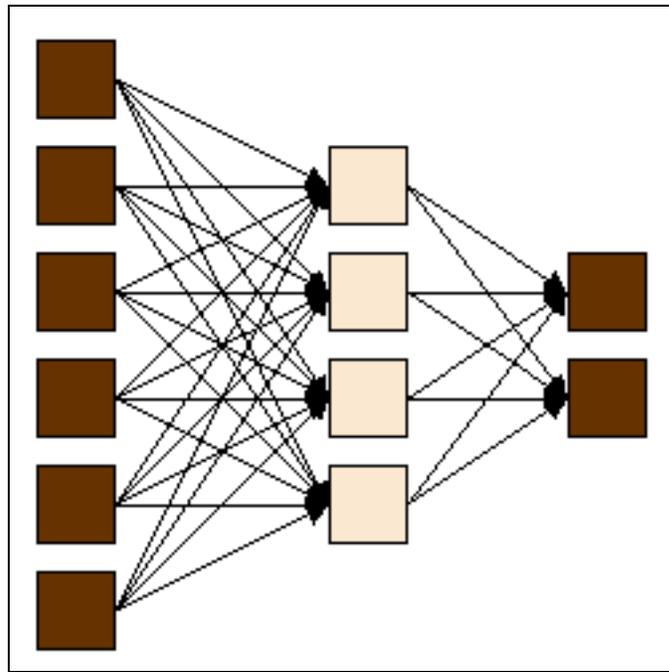


Figura 1.1. Rappresentazione di una rete neurale artificiale

¹Esempi tratti da <http://www.dst.unive.it/egesco/manifesto.html>

In figura 1.1 è rappresentata una rete neurale artificiale, uno strumento di analisi modellato in modo molto simile al sistema più complesso che, probabilmente, conosciamo: il cervello umano.

Una rete neurale simula una struttura computazionale altamente interconnessa, formata da molti elementi di processo individuali, relativamente semplici, i neuroni, che effettuano calcoli in parallelo.

Questi oggetti elementari sono in genere organizzati in gruppi o strati. Gli strati possono ricevere degli inputs (strati input), emettere degli outputs (strati output), o essere inaccessibili ad entrambi gli inputs e outputs, avendo solo delle connessioni con altri strati (strati interni).

Date le loro caratteristiche, le reti neurali sono impiegate in compiti altamente complessi come la classificazione di forme incerte, nel riconoscimento delle immagini, nella descrizione delle interazioni nelle strutture organizzative aziendali, nella previsione delle quotazioni di borsa.

Una rete neurale è un sistema complesso poiché il suo comportamento (l'output restituito) non può essere compreso analizzando le (semplici) unità computazionali che rappresentano i neuroni; la complessità è data dall'insieme di relazioni che tra questi si formano, relazioni matematiche che rappresentano gli assoni del cervello.

Il dilemma del prigioniero

Nella teoria dei giochi il dilemma del prigioniero è stato intensamente utilizzato per discutere la complessità dell'interazione umana quando essa presenta una divaricazione tra razionalità individuale e razionalità collettiva.

Nel gioco del dilemma del prigioniero due giocatori, che non possono comunicare tra loro, hanno a disposizione due strategie, L e R . Dalla figura 1.2 si vede chiaramente come per ciascun giocatore, qualunque sia la scelta dell'avversario, sia razionale giocare la strategia L . Ma il guadagno atteso della strategia L risulta nettamente minore di quello che si otterrebbe se i giocatori scegliessero entrambi R .

		Giocatore 2	
		L_2	R_2
Giocatore 1	L_1	1, 1	5, 0
	R_1	0, 5	4, 4

Figura 1.2. Tabella dei guadagni del dilemma del prigioniero

La localizzazione delle attività di una grande multinazionale

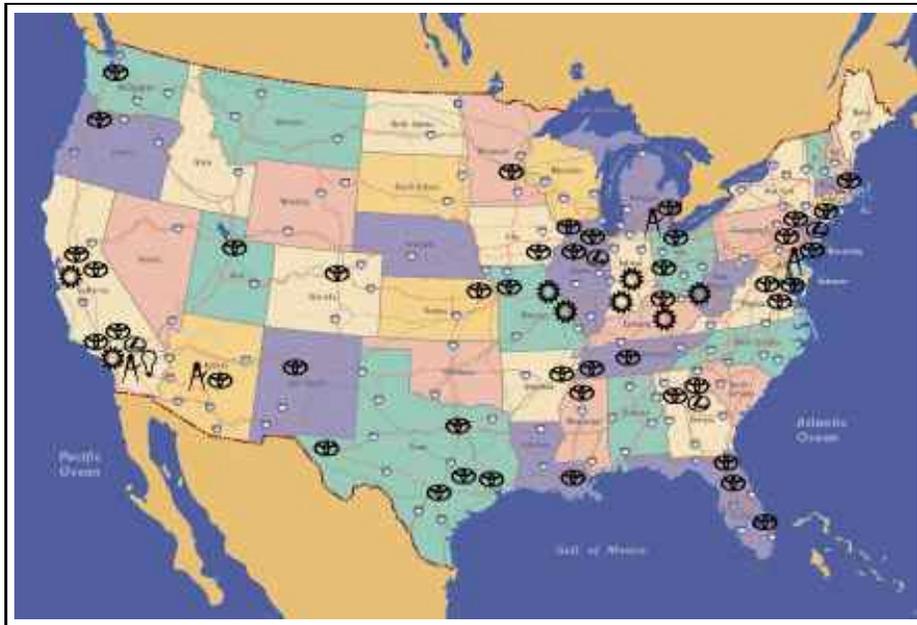


Figura 1.3. Localizzazione delle divisioni di una famosa impresa multinazionale

Nella carta geografica degli Stati Uniti in figura 1.3 sono rappresentate le divisioni delle attività di ricerca, di progettazione e design, di manufacturing e di vendita di una grande

impresa multinazionale.

Data la dispersione dei simboli è facile intuire quanto complessa possa essere l'attività di comunicazione delle aziende dislocate in regioni diverse e come da essa possa emergere l'organizzazione della multinazionale. Pur conoscendo perfettamente la struttura di una divisione non è possibile individuare il suo specifico peso sull'organizzazione globale del sistema.

L'orbita caotica

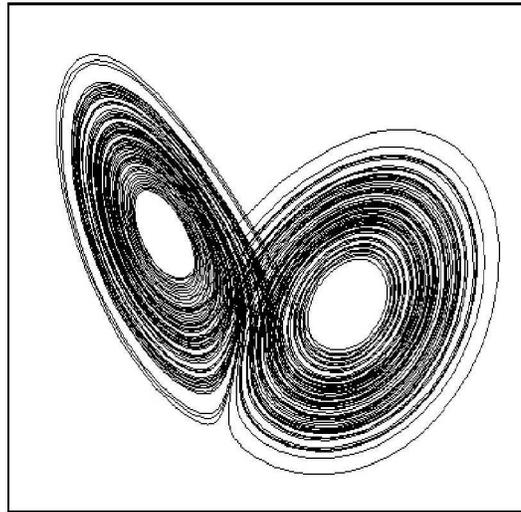


Figura 1.4. Orbite caotiche

Come abbiamo visto, i fenomeni studiati dalle scienze naturali e da quelle sociali sono tipicamente non-lineari. In particolare, essi possono manifestare quella "dipendenza sensibile dalle condizioni iniziali" che è alla radice delle dinamiche caotiche: piccole variazioni nello stato del sistema generano effetti sproporzionatamente grandi.

Un'orbita caotica è molto sensibile a piccolissimi cambiamenti delle condizioni iniziali e basta una piccolissima variazione per farla divergere. Per questo motivo, questo genere di orbita non è prevedibile su scale di tempo relativamente lunghe. Uno degli esempi più classici è l'orbita di un asteroide o di una cometa che, essendo influenzata da molti meccanismi fisici, può facilmente essere perturbata.

La mappa di Venezia



Figura 1.5. Mappa di Venezia

Infine, una metafora della complessità delle interazioni tra le attività compiute dai soggetti di un sistema economico e l'ambiente nel quale devono operare si può immaginare attraverso la mappa di Venezia riportata in figura 1.5.

Venezia, con la ramificazione dei suoi canali e l'intrico delle sue calli suggerisce la complessità delle relazioni sociali nel tessuto urbano, ma anche l'influenza che l'ambiente può avere sul comportamento del sistema economico. Il sistema non può quindi essere isolato dal suo contesto come potrebbe essere fatto per i sistemi semplici.

1.2 La simulazione al computer

Leggiamo in Hayek(1948):

[Le scienze sociali] non si occupano dei rapporti tra cose [come le scienze naturali], ma si occupano invece dei rapporti fra uomini e cose o fra uomo e uomo; si interessano delle azioni degli uomini e il loro scopo è quello di spiegare i risultati non voluti o non prestabiliti dalle azioni di molti uomini.

Le scienze che studiano i fenomeni e i cambiamenti legati alle attività dell'uomo sono ricche di problemi da risolvere e comportamenti da comprendere. Questi fenomeni sono più complicati di quelli studiati dalle scienze della natura, più sfuggenti, con più interazioni e più imprevedibilità. Non va dimenticato che quando si studiano le attività che coinvolgono in prima persona l'uomo lo scienziato studia sé stesso, con i suoi interessi e i suoi valori, che spesso fanno da velo all'oggettività richiesta dalla scienza.

1.2.1 I problemi delle scienze sociali

Le teorie della scienza mirano a identificare i meccanismi, i processi, i fattori che stanno dietro ai fenomeni e che spiegano i fenomeni, ce li fanno capire.

Uno dei punti di forza delle scienze della natura è il dialogo costante tra teorie e dati empirici, con le teorie che generano predizioni empiriche e i dati empirici che confermando o smentendo queste predizioni portano a modificare le teorie.

Le teorie sviluppate dalle scienze sociali, tra le quali l'economia si colloca, corrono il rischio di essere formulate in modo vago perché non possono ricorrere agli strumenti offerti dalla matematica tradizionale; i termini rischiano di avere un significato diverso per scienziati diversi, e le attuali teorie, formulate generalmente per via letteraria, non hanno la capacità di generare predizioni specifiche e dettagliate da verificare poi con l'osservazione della realtà empirica.

Nelle scienze sociali teorie e dati empirici sono in genere separati. Le teorie restano semplici prodotti del pensiero, formulazioni suggestive ma puramente teoriche perché prive di una verifica nella realtà empirica.

Un altro problema è l'impossibilità di usare nelle scienze sociali quello che costituisce uno dei punti di forza delle scienze della natura: il laboratorio sperimentale. E' l'uso sistematico degli esperimenti che spiega, insieme al fatto che le teorie della natura hanno cominciato a essere formulate in termini quantitativi e matematici, i grandi progressi delle scienze della natura negli ultimi tre-quattro secoli dalla rivoluzione scientifica dal '600 in poi.

Ogni economista vorrebbe avere a disposizione un laboratorio nel quale confrontare i

risultati delle sue teorie, ma i fenomeni che studia sono troppo grandi fisicamente (una società, un mercato), durano troppo nel tempo (la vita di un individuo o di un'impresa), sono il frutto di troppe variabili, richiedono un'autonomia da parte degli individui studiati che contrasta con il rigido controllo che caratterizza il metodo sperimentale.

Il risultato è che le scienze economiche sono prive di quei riscontri empirici ricchi, puntuali e direttamente ispirati dalle predizioni teoriche che solo il metodo sperimentale può fornire.

Ma anche quando si poteva ricorrere a notazioni matematiche per esprimere i modelli, fino a pochi anni fa, l'unico strumento di analisi dinamica quantitativa era costituito dalle equazioni differenziali e dalle tecniche di soluzione con "carta e penna". Per quanto l'ingegno umano avesse sviluppato tecniche estremamente sofisticate, il dominio dei problemi risolubili era molto piccolo, rispetto all'universo di quelli immaginabili.

Oggi lo scienziato può ricorrere ad una "terza via" per esprimere le sue teorie che si affianca a quelle classiche che fanno ricorso al metodo verbale (si pensi ad un libro di storia) o al metodo matematico (una teoria fisica): le simulazioni al computer.

Una simulazione consiste nel *costruire* una teoria, quantitativa o anche qualitativa, per mezzo di un programma che *gira* in un computer. Sono particolarmente utili per modellare dei processi lineari, ma soprattutto per ricostruire relazioni non lineari tra le variabili del modello.

La logica che sta dietro alla costruzione di un modello per mezzo di una simulazione al computer non è molto differente da quella utilizzata nei più famigliari modelli statistici; in entrambi i casi, infatti, il ricercatore è alla ricerca di una spiegazione migliore per il fenomeno che sta osservando. Utilizzando equazioni, regressioni o programmi per computer, il modello è costruito sulla base di una teoria che il ricercatore ha già formulato in precedenza in modo astratto per poi confrontare i risultati ottenuti dal modello con i dati riscontrabili nella realtà. Se il risultato del modello ed i dati della realtà sono simili il modello è in grado di spiegare il fenomeno, in caso contrario la teoria andrà riformulata (Gilbert, Terna 2000).

Come abbiamo visto nel paragrafo precedente le scienze economiche devono fornire

spiegazioni a fenomeni legati a dei sistemi complessi: l'obiettivo appare oggi più vicino grazie alla disponibilità di elevata potenza di calcolo dei moderni computer. Naturalmente, si tratta di una condizione necessaria ma non sufficiente: la pura forza computazionale non è sufficiente a risolvere alcun problema complesso.

Il metodo della simulazione è un nuovo e potente strumento di indagine scientifica che si affianca a quelli già disponibili, cioè teorie ed esperimenti. Si può così disporre di uno straordinario laboratorio virtuale nel quale gli scienziati osservano e studiano i fenomeni in condizioni controllate, manipolando le variabili e osservando immediatamente tutti gli effetti del proprio intervento (Parisi, 2000).

1.2.2 I vantaggi della simulazione

Nelle scienze più mature e consolidate, cioè nelle scienze della natura (fisica, chimica e biologia), teorie e dati sperimentali interagiscono costantemente e in modo ravvicinato. Lo scienziato formula la sua teoria, deriva dalla teoria determinate predizioni empiriche e verifica in laboratorio se queste predizioni empiriche sono corrette oppure no. Se non lo sono, modifica la teoria.

Le simulazioni sono un terzo strumento nelle mani di uno scienziato che si affianca ai due strumenti di ricerca tradizionali: le teorie e gli esperimenti di laboratorio. Ma quali sono i vantaggi introdotti da questo nuovo strumento?

Un nuovo modo per esprimere le teorie

Come abbiamo visto, le simulazioni sono teorie interpretative dei fenomeni della realtà formulate come un programma che gira in un computer. Adottare il metodo della simulazione significa tradurre una teoria in un programma di computer, far girare il programma nel computer e verificare se il programma, cioè la simulazione, riproduce i fenomeni che la teoria intende spiegare.

Il primo vantaggio è che esprimere una teoria come un programma di computer costringe a formulare la teoria in modi necessariamente chiari, espliciti, univoci, senza "buchi"

e contraddizioni nascoste. La ragione è che se la teoria non è formulata in questo modo il programma, scritto con un linguaggio informatico, resterà incompleto o incoerente; presentando questi problemi il programma non potrà neppure essere avviato nel computer.

Se una teoria interpretativa si presenta come una simulazione, i suoi concetti non possono non essere chiari, espliciti e univoci. Il significato dei concetti è tradotto interamente nei termini operativi della simulazione.

Una macchina per derivare previsioni empiriche dalle teorie

I concetti teorici delle scienze dell'uomo non sono in genere concetti quantitativi, matematici, i quali sono per definizione chiari, espliciti e univoci, e inoltre non si connettono in modo diretto con la realtà osservabile con i sensi.

Un vantaggio delle simulazioni è che esse permettono un tipo di verifica completamente nuovo delle teorie; le simulazioni, infatti, si inseriscono a mezza strada tra teorie e dati empirici della realtà.

Una simulazione permette di verificare se le previsioni che lo scienziato trae dalla sua teoria discendono effettivamente dalla teoria oppure no. Una teoria, per lo scienziato che la formula, pretende di spiegare certi fatti. Le simulazioni permettono di verificare prima di tutto se effettivamente la teoria spiega tali fatti.

Le simulazioni tendono ad avere una verifica a due livelli:

- Il primo livello è quello in cui si verifica se la teoria incorporata nella simulazione effettivamente produce i risultati previsti. In questa prima verifica si assumono certi dati empirici, considerati però in modo un globale e già noti, e si verifica se la simulazione li riproduce. In realtà, tuttavia, quello che si verifica in questo primo livello è la coerenza interna della teoria, la chiarezza e univocità dei suoi concetti, la sua completezza e capacità di riprodurre effettivamente determinate previsioni.
- Il secondo livello di verifica di una simulazione è quello della verifica empirica reale. In questa fase bisogna accertare se i risultati della simulazione corrispondono con i fatti della realtà, cioè se la simulazione riesce a riprodurre nel dettaglio i dati empirici e se riesce a fare predizioni su fatti empirici ancora non noti.

Ma le simulazioni risultano utili allo scienziato anche prima di formulare la teoria; la simulazione sta infatti valorizzando un metodo che viene utilizzato solo marginalmente e implicitamente in tutti i campi di ricerca: gli esperimenti mentali.

Un esperimento mentale è un esperimento non condotto realmente, osservando e manipolando la realtà, ma è un esperimento immaginato. Sia in un esperimento reale, che in un esperimento mentale, lo scienziato mette alla prova le sue idee e le sue teorie sulla realtà, e più precisamente le idee e le predizioni tratte da queste idee e teorie. Ma mentre nell'esperimento reale lo scienziato si basa sulla osservazione della realtà, sulla manipolazione fisica di cose fisiche, e sulle conseguenze osservate di queste manipolazioni, in un esperimento mentale tradizionale lo scienziato si limita a immaginare delle cose, delle manipolazioni di tali cose, e i risultati, anch'essi immaginati, di queste manipolazioni. Con l'ausilio dei computer i ragionamenti dello scienziato possono essere automatizzati, facilitando così la formulazione della teoria.

Laboratori sperimentali virtuali

In un laboratorio, ma anche con metodi di osservazione sul campo, non possono essere studiati fenomeni e entità troppo grandi come una società o un'impresa, fenomeni che durano troppo nel tempo o che sono successi in passato come la rivoluzione industriale, oggetti o esseri viventi che non si possono manipolare a piacere per ragioni etiche o perché non esistono più come i dinosauri. Tutti questi fenomeni invece possono essere simulati.

In effetti, l'impossibilità per le scienze dell'uomo di usare il laboratorio sperimentale così come fanno le scienze della natura è una causa della loro debolezza rispetto alle scienze della natura.

Poiché le teorie sono incorporate nel programma del computer e i dati empirici emergono quando il programma "gira" nel computer, una simulazione funziona come un laboratorio sperimentale virtuale, nel quale lo scienziato, come nel laboratorio reale, osserva i fenomeni (simulati) in condizioni controllate, manipola le variabili e osserva gli effetti delle sue manipolazioni. Ovviamente, la verifica ultima di una simulazione è che i dati empirici così come emergono nella simulazione corrispondano ai dati empirici così come si osservano

nella realtà.

Un calcolatore può essere usato come un ambiente di sperimentazione tramite il quale si può studiare un fenomeno complesso, come l'evoluzione di alcune forme di vita o di un mercato e si può verificare il suo comportamento in base ai valori assunti dai parametri che lo caratterizzano.

In quanto laboratorio virtuale per i modelli degli scienziati cognitivi e dei teorici della complessità, il computer ha creato un nuovo modo di fare scienza, a metà strada tra teoria matematica ed esperimento di laboratorio.

1.2.3 I problemi delle simulazioni

Nonostante gli indubbi vantaggi che l'adozione del metodo di ricerca della simulazione è in grado di apportare nella comprensione del reale, come ogni metodo di ricerca anch'essa presenta limiti e svantaggi che sono stati posti in luce dalla comunità scientifica.

Le principali critiche volte a sfavore della simulazione possono così essere riassunte:

- la simulazione risulta alcune volte un'eccessiva semplificazione della realtà;
- simulando ci si limita a riprodurre i fenomeni, quindi non si spiega nulla di nuovo;
- non è possibile simulare (ricostruire) una realtà che non si conosce ancora bene;
- le simulazioni si presentano come una scatola nera che riproduce un fenomeno senza spiegarlo in maniera esplicita.

Questi sono dei rischi ai quali il ricercatore-simulatore può andare incontro, ma che possono essere evitati utilizzando un rigoroso metodo scientifico nella preparazione degli esperimenti condotti con le simulazioni.

Il rischio maggiore delle simulazioni è lo scarso peso attribuito al ruolo della verifica esterna delle teorie, spesso subordinata alla ricerca di forme di coerenza interna tra formulazioni teoriche e predizioni empiriche che da essa dovrebbero derivare. Superato il momento della verifica interna, si rischia di trascurare la verifica tra l'effettiva corrispondenza dei risultati della simulazione e la realtà empirica, chiudendo il circolo del processo cognitivo.

Portando alle estreme conseguenze il ragionamento, il rischio diventa quello prospettato da Pryor (2000) in un lavoro auto-ironico, in cui un autore sconosciuto nel 2028, dopo la caduta di un asteroide sulla terra, ritrova un libro sulla complessità degli anni 90. Dopo una attenta lettura, l'autore catalogherà il ritrovato come un tipico libro sull'argomento di anni in cui quasi tutti i lavori non contengono applicazioni empiriche reali, a parte qualche interessante aneddoto.

1.3 Simulazioni ad agenti

E' però importante distinguere, nell'ambito della costruzione di modelli simulativi, due modi molto diversi di procedere: "dall'alto al basso" e "dal basso all'alto".

Procedere "dall'alto al basso" significa costruire il modello di un sistema in base alla conoscenza globale che abbiamo del sistema stesso: come si comporta, quali fenomeni si possono associare al suo funzionamento, a quali leggi ubbidisce. Significa, in altre parole, studiare un sistema partendo dalla riproduzione della sua completezza e trascurando, almeno inizialmente, i dettagli relativi alle prestazioni delle singole componenti.

Procedere "dal basso all'alto" significa, al contrario, costruire il modello di un sistema in base alla conoscenza locale che se ne ha: come sono fatte e quante sono le sue componenti, come interagiscono tra loro, a quali leggi ubbidiscono.

I modelli di simulazione ad agenti (ABM) seguono questa procedura e si prestano all'analisi di problemi complessi e distribuiti, dove cioè il comportamento complessivo del sistema dipende dai comportamenti individuali dei suoi componenti.

1.3.1 Sugarscape

Sugarscape, sviluppato dall'informatico Robert Axtell e dal sociologo Joshua Epstein della Brookings Institution di Washington è un modello simulativo utilizzato per lo studio di civiltà artificiali. Può essere opportunamente citato come uno dei più celebri modelli costruiti *dal basso verso l'alto*.

I due ricercatori hanno pubblicato un libro intitolato *Growing artificial societies* in

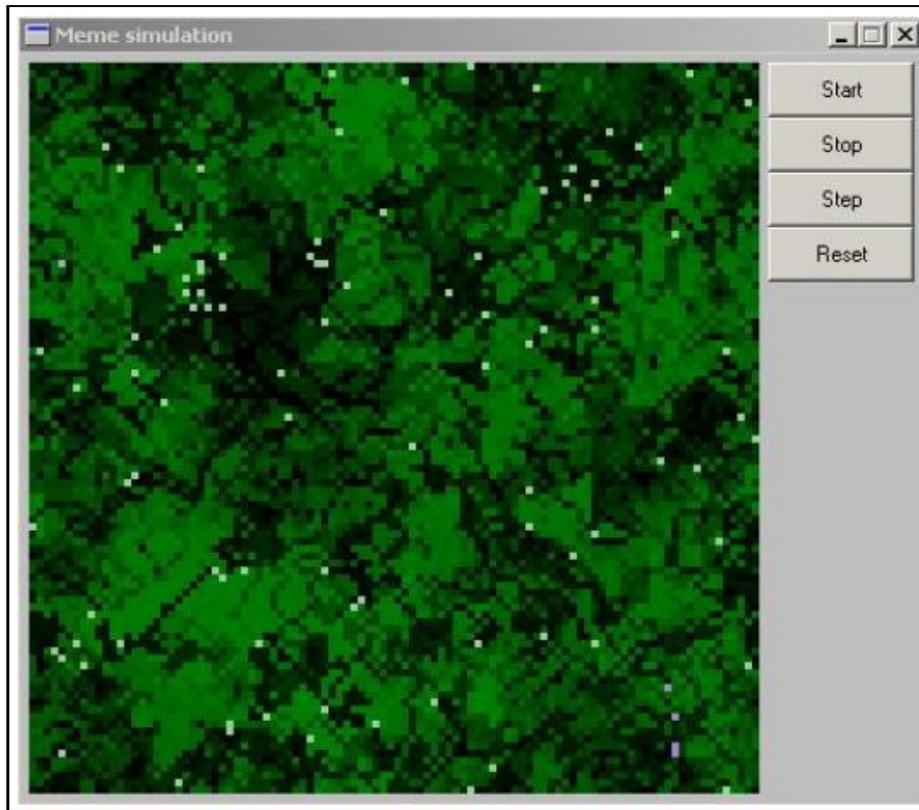


Figura 1.6. Sugarscape

cui descrivono un'ipotetica popolazione di individui, eterogenei nelle capacità (vista) ed esigenze (metabolismo), che vive in un ambiente in cui la risorsa di base è lo zucchero. Questi individui devono guardarsi intorno, trovare lo zucchero e mangiarlo.

Lo zucchero si trova sul territorio (come rappresentato in figura 1.6) con una distribuzione spaziale che può essere manipolata dal ricercatore; facendo, ad esempio, crescere lo zucchero in tempi diversi da nord a sud è possibile ricreare l'alternarsi delle stagioni e osservare la migrazione delle creature.

Gli autori, grazie a questo modello costruito su semplici regole, riescono a riprodurre una varietà di fenomeni sociali ed economici che avvengono in Sugarscape.

Leggiamo in Epstein e Axtel (1996):

Il nostro punto di partenza nella costruzione di modelli fondati su agenti è l'individuo: attribuiamo delle regole di comportamento agli agenti e quindi

facciamo procedere il sistema nel tempo, per scoprire quale struttura emerga a livello macroscopico. Questa scelta contrasta notevolmente con la rappresentazione fortemente aggregata della macroeconomia, della sociologia, di certi campi della scienza politica, in cui aggregati sociali, come classi e stati, sono definiti *ab initio*. In questo senso il nostro lavoro può essere a ragione etichettato come "individualismo metodologico".

1.3.2 Individualismo metodologico e simulazione ad agenti

L'individualismo metodologico è la teoria che si oppone al collettivismo metodologico, cioè alla concezione che sostiene che ai concetti collettivi (stato, nazione, elettorato, mercato, impresa, ...) corrispondono specifiche realtà autonome, distinte e indipendenti dagli individui che le compongono. Per gli individualisti, invece, ai concetti collettivi non corrisponde niente di specifico, essendo questi il frutto di azioni individuali.

Secondo gli autori della Scuola Austriaca, per i quali l'individualismo metodologico è la tesi fondamentale, il collettivismo metodologico è una concezione erronea e pericolosa. Erronea perché presume acriticamente che, se certi concetti sono di uso corrente, devono anche esistere in concreto quelle date cose che essi designano. Pericolosa perché l'aver postulato l'esistenza di collettività indipendenti e autonome dagli individui è all'origine di atrocità indicibili (il riferimento è alle dittature).

Solo gli individui pensano, ragionano e agiscono (non il mercato o lo stato): è questo il nucleo teorico dell'individualismo metodologico di Menger, Mises e Hayek. Per gli autori, le azioni intenzionali, o meno, degli individui comportano conseguenze inattese, o indesiderate; il parallelismo con i fenomeni emergenti nei sistemi complessi è quindi forte.

Leggiamo in Hayek (1948):

Lo scienziato sociale deve considerare come niente di più che teorie provvisorie, astrazioni popolari, le idee prodotte dalla mentalità popolare a proposito di entità collettive quali la "società" o il "sistema economico", il "capitalismo" o l'"imperialismo", e altre consimili; e sono proprio queste idee che lo studioso

di scienze sociali non deve prendere erroneamente per fatti. Astenersi dal trattare queste pseudo-entità alla stregua di "fatti", e prendere sistematicamente le mosse dalle concezioni che guidano gli individui nelle proprie azioni, e non dai risultati delle loro teorizzazioni sulle proprie azioni: questo è il tratto caratteristico di quell'individualismo metodologico che è strettamente connesso con il soggettivismo delle scienze sociali.

Seguendo la tesi dell'individualismo metodologico, per studiare la complessità dei fenomeni sociali dobbiamo ricostruirla (simularla) partendo dagli elementi che ci sono direttamente noti, come le concezioni e le opinioni degli individui. Solo con una successiva composizione dalle azioni dei singoli individui possiamo *scoprire* dei principi di coerenza strutturale nei fenomeni complessi; strutture emergenti che, probabilmente, non eravamo stati in grado di identificare attraverso un'osservazione diretta.

Nelle simulazioni che adottano una metodologia ad agenti (ABM) ogni componente del sistema in analisi è modellata come entità a sé e il comportamento complessivo viene ricostruito dall'interazione dei singoli costituenti; ma individualismo metodologico e metodologie fondate su agenti non sono necessariamente coincidenti.

Attraverso le simulazioni ABM si può studiare l'emergenza di fenomeni imprevisti o imprevedibili all'interno di sistemi costruiti su più livelli; ogni livello del sistema è un agente ma a sua volta un agente può essere costituito da parti. Un mercato, ad esempio, è formato da imprese e istituzioni, ma imprese e istituzioni sono a loro volta formate da individui. Nelle simulazioni ABM l'emergenza nasce dalle interazioni che si costruiscono all'interno di ogni livello e da quelle tra livelli.

La metodologia è quindi differente e può essere spiegata attraverso le parole di Kirmann (1992):

La somma dei comportamenti di semplici individui economicamente plausibili può generare complesse dinamiche, mentre per costruire un singolo individuo il cui comportamento mostri la stessa dinamica può essere necessario fondarlo su caratteristiche ben poco naturali. Inoltre se si rifiuta [sulla base del

modello] una particolare ipotesi di comportamento, non è chiaro se si sta realmente rifiutando l'ipotesi in questione o l'ipotesi che esista un solo individuo [rappresentativo].

Nelle simulazioni ABM gli agenti che costituiscono il sistema possono, infatti, essere degli individui, ma anche aggregati più ampi come imprese, uffici, stati; il vantaggio delle simulazioni ABM è che sono modulari e scalabili. Il livello di dettaglio ottimale dipende dall'obiettivo della simulazione e dalle risorse disponibili (dati, sistemi calcolatori, ...).

La modularità ne permette l'adattamento e la modifica con estrema semplicità. Per esempio, risulta facile utilizzare la simulazione come laboratorio virtuale introducendo altri agenti nel sistema con un comportamento differente, oppure spostando gli agenti in un ambiente alternativo.

Allo sperimentatore è sufficiente occuparsi di modellare il comportamento (semplice) di ciascun agente, e le regole (semplici) che governano le interazioni dei diversi agenti. Non deve occuparsi di descrivere la complessità dell'interazione di tutti gli agenti insieme.

Data la sua relativa giovinezza, è difficile ricavare dagli autori e dagli autori una definizione univoca di questa metodologia. Seguendo Axtell (2000) si possono però individuare tre tipologie di simulazioni ad agenti.

It is argued that there exist three distinct uses of agent modeling techniques. One such use the simplest is conceptually quite close to traditional simulation in operations research. This use arises when equations can be formulated that completely describe a social process, and these equations are explicitly soluble, either analytically or numerically. In the former case, the agent model is merely a tool for presenting results, while in the latter it is a novel kind of Monte Carlo analysis. A second, more commonplace usage of computational agent models arises when mathematical models can be written down but not completely solved. In this case the agent-based model can shed significant light on the solution structure, illustrate dynamical properties of the model, serve to test the dependence of results on parameters and assumptions, and be a source of counter-examples. Finally, there are important classes of problems for which

writing down equations is not a useful activity. In such circumstances, resort to agent-based computational models may be the only way available to explore such processes systematically, and constitute a third distinct usage of such models.

1.3.3 Lo schema ERA

Gilbert e Terna (2000) suggeriscono uno schema generale col quale costruire le simulazioni in campo sociale denominato ERA (*Environment-Rules-Agents*); seguendo questa impostazione la costruzione del modello ad agenti deve avvenire su quattro livelli ben distinti:

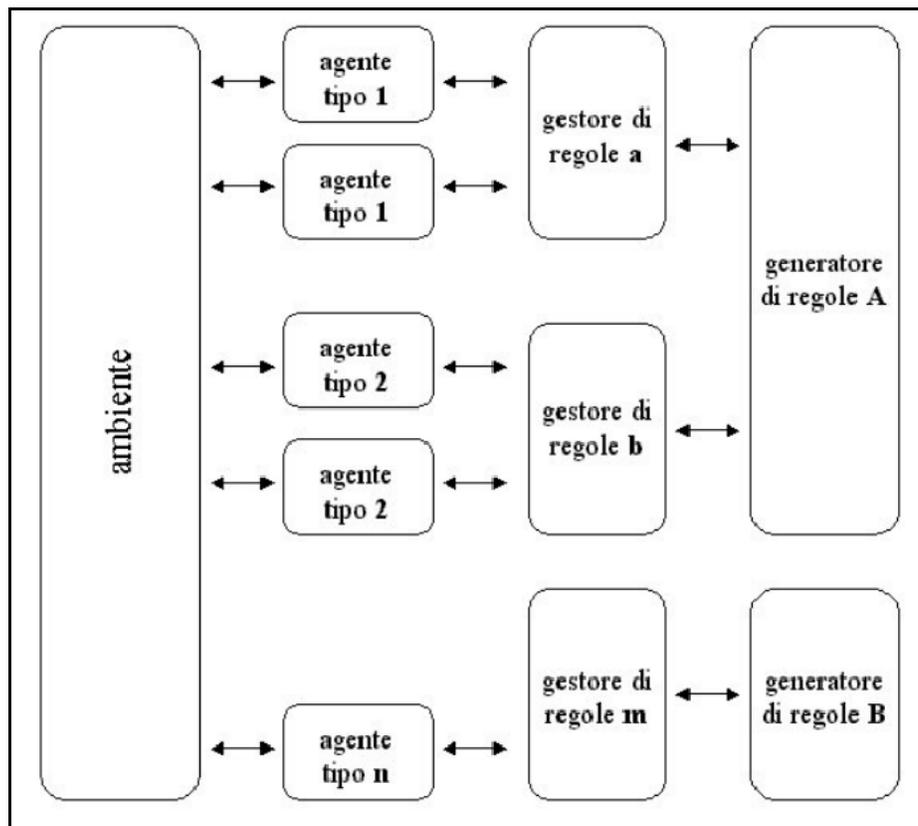


Figura 1.7. Schema ERA

Ambiente: il primo strato fornisce una rappresentazione dell'ambiente all'interno del quale gli agenti sono immersi e devono interagire; per semplificare il codice della simulazione, si assume che gli agenti non comunichino tra di loro in maniera diretta, ma che i flussi informativi passino sempre attraverso l'ambiente.

Agenti: il secondo livello è quello specifico degli agenti; a livello di codice gli agenti derivano, come esemplari, da una classe tipo.

Gestore di regole: il terzo strato è quello in cui avviene la gestione delle regole con le quali gli agenti intraprendono le loro azioni; può essere considerato come la rappresentazione delle capacità cognitive del singolo agente;

Generatore di regole: il quarto strato concerne la costruzione delle regole: allo stesso modo con cui gli agenti interrogano i loro gestori di regole, questi ultimi possono interrogare un oggetto a loro superiore per modificare nel tempo le proprie regole d'azione; il generatore di regole rappresenta le diverse strutture cognitive degli agenti.

Da un punto di vista teorico, il principio fondamentale del modello proposto è il mantenimento su due piani concettuali distinti l'ambiente, che rappresenta il contesto per mezzo di regole e dati generali, e gli agenti, con le loro strutture di dati interne.

Un simile schema presenta l'indubbio pregio di rendere rigorosa la scrittura del codice informatico della simulazione, rendendone modulare il processo di costruzione se effettuato attraverso linguaggi di programmazione orientati agli oggetti.

Bibliografia

- [1.1] Arthur W.B., Durlauf S.N., Lane D.A., *The Economy as an Evolving Complex System II*, Addison-Wesley, 1997.
- [1.2] Axtell R., *Why agents? On the varie motivations for agent computing in the social sciences*, Center on Social and Economic Dynamics, Working Paper No. 17, 2000.
- [1.3] Bettelli O., *Modelli per sistemi complessi*, <http://newk.alma.unibo.it/oscar/cmplx/>, ultima visita gennaio 2003.
- [1.4] Gilbert N. and Terna P., *How to build and use agent-based models in social science*, Mind Society, no. 1, 2000.
- [1.5] Hayek, F., *Economics and knowledge. Economica, new series*, IV (13), 1937.
- [1.6] Hayek F., *Individualism and Economic Order*, University of Chicago Press, 1948.
- [1.7] Hayek F., *Studies in philosophy, politics and economics*, University of Chicago Press, 1967.
- [1.8] Kilpatrick H. E. Jr., *Complexity, spontaneous order, and Friedrich Hayek: are spontaneous order and complexity essentially the same thing?*, Complexity Vol. 6 N. 3, John Willey Sons, 2001.
- [1.9] Kirman, K., *Whom or what does the representative agent represent?*, Journal of Economy Perspectives No. 6, 1992.
- [1.10] Parisi D., *Simulazioni. La realtà ifatta nel computer*, Il mulino, 2001.
- [1.11] Stein D. L., *Lectures in the sciences of complexity*, Addison-Wesley, 1989.
- [1.12] Terna P., *Hayek e il connessionismo: modelli con agenti che apprendon*, in G. Clerico e S. Rizzello (a cura di), *Il pensiero di Friedrich von Hayek*, Utet, 2000.
- [1.13] Tucker W., *Complex questions: the new science of spontaneous order*, <http://reason.com/9601/TUCKERfeat.html>, ultima visita gennaio 2003.
- [1.14] Pryor F. L., *Looking Backwards: Complexity Theory in 2028*, in D. Colander (ed.) *The complexity vision and the teaching of economics*, Edward Elgar, 2000.
- [1.15] Terna P., *Creare mondi artificiali: una nota su Sugarscape e due commenti*, Sistemi Intelligenti, no. 3, 1998.
- [1.16] Tinti T., *Il concetto di emergenza tra dualismo e materialismo*,

http://utenti.tripod.it/tullio71/concetto_di_emergenza.htm, ultima visita dicembre 2002.

- [1.17] Waldrop, M. Mitchell, *Complexity: the emergence of science at the edge of order and chaos*, Simon and Schuster, 1992.

Capitolo 2

Lo sviluppo di una simulazione

2.1 La programmazione orientata agli oggetti

2.1.1 Breve storia della programmazione

Il desiderio di creare strumenti in grado di automatizzare i calcoli potrebbe essere fatto risalire a più di 2000 anni fa, con la comparsa dell'abaco; questo è il primo strumento, che la storia ricordi, in grado di eseguire ogni tipo di operazione matematica mediante una serie di istruzioni impartite dall'utente.

La nascita della prima macchina per calcolare è generalmente fatta risalire al 1642, data dei primi studi in questo campo di Blaise Pascal. Anche Gottfried von Leibniz realizzò nel 1694 una prima macchina in grado di effettuare somme e moltiplicazioni. Queste macchine però non vennero mai utilizzate, né in campo scientifico né in campo commerciale.

Solo dopo più di un secolo questi strumenti trovarono un discreto utilizzo in campo commerciale. Le macchine di fine '800, infatti, erano già in grado di effettuare somme, sottrazioni, moltiplicazioni e divisioni. Sviluppi futuri permisero poi di aggiungere funzionalità come l'accumulazione di risultati parziali, l'utilizzo di memorie per riutilizzare dati di precedenti operazioni e la stampa dei risultati.

Le prime macchine in grado di calcolare videro però la luce solo agli inizi degli anni '40 grazie alla tecnologia elettronica, come ad esempio il famoso *ENIAC*. In quegli anni l'unico

metodo per programmare era il *Linguaggio Macchina*; le istruzioni erano codificate in codice binario, dove sequenze di 0 e 1 rappresentavano le operazioni impartite dal programmatore all'elaboratore.

Gli sviluppi tecnologici hanno poi portato, negli anni '50, alla nascita dei primi linguaggi di programmazione. Una prima evoluzione ha portato ad una codifica simbolica, anziché binaria, dei programmi.

Queste versioni simboliche sono note come *linguaggi assembleri*, dal termine usato per indicare i programmi traduttori che, ricevendo come dato la versione simbolica di un programma, producono la corrispondente forma binaria direttamente eseguibile dalla macchina. L'utilizzo di un linguaggio più naturale, e più simile a quello matematico, facilitò il compito dei programmatori, anche a costo di caricare la macchina di ulteriori compiti.

Il passo successivo nell'evoluzione dei linguaggi di programmazione tese a rendere la codifica delle istruzioni da eseguire più simile al problema da risolvere anziché all'architettura della macchina destinata all'esecuzione dei calcoli. I primi, e più celebri, linguaggi di programmazione di questo tipo che possiamo ricordare sono il Fortran (FORmula TRANslator) e il Cobol (COmmon Business Oriented Language).

Da quel momento il numero e la tipologia di linguaggi di programmazione continuò ad aumentare, ognuno con finalità e capacità diverse in base agli ambiti di utilizzo. Il Cobol degli anni '50, ad esempio, venne ideato per applicazioni nei campi dell'amministrazione e del commercio, per l'organizzazione dei dati e la manipolazione dei file; il Basic nacque nel '64 come linguaggio per i principianti, mentre il Pascal era indirizzato ai neo-programmatori, ed è stato il primo esempio di prodotto di origine accademica che abbia conosciuto vasto successo ed applicazione anche nel mondo dell'industria.

Il linguaggio tuttora più affermato comparve solo pochi anni più tardi; il C¹ si distinse per la sua versatilità nella rappresentazione dei dati e per la sua predisposizione, pur offrendo un limitato numero di istruzioni, ad essere utilizzato per ogni tipo di operazione.

Ma la vera rivoluzione avvenne nel 1983 quando Bjarne Stroustrup sviluppò il C++ (o come era stato chiamato inizialmente "C con classi") che introduceva, sfruttando come

¹Il nome segue quello dei suoi predecessori *A* e *B*

base il C, la programmazione Orientata agli Oggetti (OO - Object Oriented) usando una nuova struttura, la classe.

L'attuale stato dell'arte presenta un settore ancora in continua evoluzione. Se i vecchi linguaggi continuano ad essere ampiamente utilizzati negli ambiti per i quali si sono affermati, i linguaggi ad oggetti si stanno imponendo come strumenti efficaci per risolvere una grande quantità di problemi e stanno vivendo un periodo di grande diffusione. Il linguaggio attualmente più diffuso è Java, noto per la sua potenza, versatilità e portabilità.

Il linguaggio Java, il concetto di classe e la programmazione orientata agli oggetti verranno analizzati in seguito più in dettaglio in quanto strumenti fondamentali per le simulazioni oggetto di questa tesi.

2.1.2 La struttura di un programma

Un elaboratore può essere impiegato al fine di risolvere problemi di varia natura, come la gestione di una base di dati storici, il calcolo di dati scientifici o la realizzazione di simulazioni economiche come nel nostro caso.

Per ottenere questi ed altri risultati il problema deve essere formulato in modo opportuno affinché l'elaboratore sia in grado di risolverlo. Due sono le attività fondamentali da svolgere per raggiungere questo obiettivo:

- L'analisi e la definizione di un **algoritmo**
- La definizione e creazione di un **programma**

L'*arte* della programmazione consiste nella progettazione o selezione di algoritmi e nella loro descrizione in un linguaggio di programmazione. Un algoritmo² è una successione di **istruzioni** o **passi** che definiscono le operazioni da eseguire sui dati per ottenere i risultati. Esempi di algoritmi possono essere ottenuti dall'esperienza quotidiana; le regole per un gioco di carte, le istruzioni d'uso di un elettrodomestico, le ricette di cucina.

Supponiamo di voler creare un programma che descriva come un sarto possa produrre in modo artigianale una maglia per un suo cliente. Per poter realizzare questo programma

²Il termine deriva dal nome del matematico arabo Al Khuwarizmi vissuto nel nono secolo d.c.; esempi di algoritmi sono stati ritrovati in Mesopotamia su antiche tavolette babilonesi risalenti al 1600-1800 a.c.

dobbiamo individuare quali sono i passi che il nostro ipotetico sarto deve compiere per ottenere il risultato voluto. I passi individuati saranno inseriti in un algoritmo che potrebbe essere così definito:

1. Prendi le misure del cliente
2. Calcola i quanti m^2 di stoffa che sono necessari
3. Acquista la stoffa
4. Realizza la maglia
5. Consegnala maglia al cliente

Questo elenco di istruzioni affinché possa essere considerato un algoritmo deve soddisfare i seguenti requisiti:

Finitezza: ogni algoritmo deve essere finito, questo implica che ogni istruzione deve poter essere eseguita in un tempo finito (anche se elevato) di volte.

Generalità: ogni algoritmo deve fornire la soluzione ad una classe di problemi. Deve pertanto essere applicabile a qualsiasi insieme di dati appartenente all'**insieme di definizione** o **dominio dell'algoritmo** e deve produrre risultati che appartengono all'**insieme di arrivo** o **codominio dell'algoritmo**, che è l'insieme di tutti i possibili risultati ottenibili dall'esecuzione dell'algoritmo.

Non ambiguità: devono essere definiti in modo univoco i passi successivi da eseguire per ottenere i risultati voluti. Devono essere evitati paradossi, contraddizioni ed ambiguità. Il significato di ogni istruzione deve essere chiaro e univoco.

Un programma è la ricetta che traduce l'algoritmo ed è direttamente comprensibile ed eseguibile da parte dell'elaboratore. La formalizzazione di un algoritmo in un programma avviene tramite un linguaggio di programmazione. E' possibile immaginare i linguaggi di programmazione come elementi intermedi di una varietà di linguaggi ai cui estremi si trovano il linguaggio macchina da un lato ed i linguaggi naturali (italiano, inglese ...)

dall'altro. Essi consentono al programmatore di trattare i problemi da risolvere senza doversi preoccupare dei dettagli della particolare macchina sulla quale il programma viene eseguito.

I linguaggi di programmazione, in quanto specificatamente progettati per la manipolazione di informazioni, differiscono dai linguaggi naturali: sono infatti molto più limitati ma non ambigui. Per poter tradurre le istruzioni del programma (dette di alto livello) in istruzioni direttamente eseguibili dalla macchina (di basso livello) è necessario un **compilatore** o un **interprete**; in presenza del primo parliamo di linguaggi di programmazione *compilati*, altrimenti di linguaggi di programmazione *interpretati*.

I linguaggi che non dipendono dall'architettura della macchina offrono due vantaggi fondamentali:

- i programmatori non devono preoccuparsi dei dettagli architetturali di ogni calcolatore;
- i programmi risultano più facili da leggere e modificare.

Quando un programmatore ha imparato un linguaggio di alto livello, non è richiesto che si occupi delle modalità con cui il compilatore traduce i programmi nel linguaggio di basso livello. Ciò implica che i programmi scritti per un calcolatore possano essere utilizzati anche su un altro, previa una semplice ricompilazione. Questa caratteristica è nota come **portabilità**.

Un altro vantaggio dei linguaggi di programmazione è la loro **leggibilità**: la relativa similitudine con i linguaggi naturali (e matematici) rende i programmi non solo più semplici da scrivere ma anche da leggere. Ad esempio, nel linguaggio di programmazione *JAVA*, l'espressione:

$$x = (a + b) * (c + d)$$

calcola l'espressione x ottenuta eseguendo una serie di operazioni sulle variabili a, b, c, d . In linguaggio di basso livello (linguaggio macchina) la stessa operazione risulterebbe scritta nel seguente modo:

- LOAD a,%r0

- LOAD b,%r1
- ADD %r0,%r1
- LOAD c,%r2
- LOAD d,%r3
- ADD %r0,%r1
- MULT %r1,%r3
- STORE %r3,x

che è chiaramente di comprensione meno immediata rispetto alla tradizionale notazione matematica.

L'essenza della programmazione di alto livello, ovvero dell'uso di linguaggi di elevata potenza espressiva, risiede nella capacità di **astrazione**, cioè nella possibilità di prescindere dai dettagli considerati inessenziali ai fini della soluzione del problema, favorendo con ciò la concentrazione sugli elementi fondamentali.

La possibilità di codificare algoritmi in maniera astratta si traduce in una migliore comprensibilità del codice e quindi in una facile *analisi di correttezza*. Strettamente correlata alla leggibilità è quindi la **mantenibilità**: essendo più leggibili, i programmi sono anche più semplici da correggere e modificare.

L'utilizzo di linguaggi ad alto livello comporta comunque anche alcuni svantaggi, come ad esempio la *riduzione di efficienza* dei programmi. Quando il compilatore traduce un programma in linguaggio macchina la conversione può non essere effettuata nel modo migliore: sono diversi i modi per esprimere la stessa sequenza di istruzioni e il programmatore spesso non conosce il modo in cui questa sarà tradotta a livello basso; come risultato si può così ottenere un programma poco efficiente, soprattutto utilizzando compilatori poco sofisticati.

Durante la fase di sviluppo del programma possiamo individuare tre operazioni fondamentali da compiere quando si utilizza un linguaggio di programmazione di tipo compilato.

- **La compilazione del file sorgente**

Al termine della fase di progettazione deve essere individuato un insieme di funzione (dette *routine*), ognuna delle quali risolve una piccola parte del problema di programmazione. Passaggio successivo è la stesura del codice per ogni funzione, mediante la creazione e la scrittura di file di testo in linguaggio di alto livello, ottenendo così il **file sorgente**. Il compilatore ha il compito di produrre il **codice oggetto** partendo dal codice sorgente.

- **L'effettuazione del link dei file oggetto**

I file oggetto ottenuti non sono ancora pronti per essere eseguiti dal calcolatore; per poterlo fare abbiamo bisogno di un programma di **link**. Grazie a questo programma tutti gli oggetti che compongono il programma (e le librerie necessarie) vengono unite per ottenere un **programma eseguibile**.

- **Il caricamento dei file eseguibili** La fase di caricamento in memoria del programma eseguibile (*loading*) è spesso ignorata poichè viene gestita automaticamente dal sistema operativo della macchina. Per avviare il programma infatti è sufficiente digitare il nome del file eseguibile.

2.1.3 Il paradigma della programmazione ad oggetti

In gergo informatico con l'abbreviazione **OOP** (*Object Oriented Programming*) si denota l'insieme delle caratteristiche che formano il paradigma della programmazione ad oggetti. Storicamente, le prime idee di queste caratteristiche nascono alla fine degli anni '60 con il linguaggio di simulazione di scuola scandinava Simula '67 e successivamente con il linguaggio SmallTalk.

La differenza principale rispetto alla metodologia classica orientata a procedure e funzioni (tipica dei linguaggi Fortran e Pascal) consiste nel diverso punto di vista che il programmatore deve adottare in fase di sviluppo delle applicazioni. Secondo il paradigma procedurale un problema complesso deve essere suddiviso in tanti problemi più semplici in modo che sia risolvibile tramite una successione di semplici funzioni.

Il nuovo paradigma della programmazione ad oggetti sposta invece l'attenzione sui soggetti che compongono il problema; il programmatore deve scomporre il problema in una serie di concetti (gli oggetti) che si dovranno unire e far interagire per sviluppare un'applicazione; il primo vantaggio che si può riscontrare è un codice più semplice da leggere e da mantenere.

Un linguaggio di programmazione per poter essere definito "ad oggetti" deve soddisfare quattro caratteristiche fondamentali: il concetto di oggetto e di classe, l'incapsulamento, l'ereditarietà ed il polimorfismo. Un linguaggio che non offre strumenti sufficienti per fruire di queste caratteristiche non può essere definito *ad oggetti*.

Oggetti e classi

Un oggetto è un'entità (informatica) dotata di:

- **identità**, per poter distinguere un oggetto da un altro. Volendo creare, ad esempio, un oggetto "studente" per un'applicazione questo deve essere distinto dagli altri suoi simili attraverso il numero di matricola;
- **stato**, una sorta di memoria interna all'oggetto che può essere modificata dallo stesso oggetto. Gli elementi che caratterizzano lo stato interno di un oggetto vengono anche definiti *attributi*. Per l'oggetto "studente" un attributo potrebbe essere il numero di esami superati;
- **comportamento**, la capacità di osservare o modificare lo stato degli oggetti tramite l'invocazione di metodi. Continuando l'esempio, un metodo potrebbe essere utile per calcolare la media dei voti di uno studente;

Seguendo una metodologia ampiamente condivisa i metodi sono classificati in *comandi* e *osservatori*; con i primi si identificano quelle operazioni che alterano lo stato locale dell'oggetto, con i secondi l'oggetto restituisce semplicemente un valore presente nel suo stato locale. Questa separazione è dettata dalla necessità di migliorare la leggibilità del codice.

Per creare un oggetto la tecnica utilizzata dalla maggior parte dei linguaggi di programmazione è di definire prima uno schema di creazione degli oggetti di un certo tipo contenente la descrizione del suo stato interno e dei suoi comportamenti; questa definizione è identificabile con una porzione del codice del programma detta **classe**.

Rifacendoci al nostro esempio la classe che deve descrivere un generico studente deve prevedere delle zone di memoria dove poter immagazzinare i risultati degli esami e la sua media (stato locale) e dove descrivere le operazioni necessarie per calcolare la media dei suoi voti (comportamento).

Per creare i singoli oggetti (un gruppo di studenti) la classe deve essere **istanziata**; questo avviene tramite un'operazione detta *costruttore o creatore* in grado di determinare l'identità univoca dell'oggetto (il numero di matricola) e il suo stato iniziale (il numero di esami da sostenere). Una generica notazione per creare le istanze di una classe (gli oggetti) potrebbe essere:

```
nomeOggetto nuovo costruttore ( parametri)
```

Creato l'oggetto si può accedere ad un suo valore interno, o invocare un suo metodo con una notazione particolare del tipo:

```
nomeOggetto.metodo(eventuali parametri)
```

questa è la notazione più utilizzata attualmente nei linguaggi ad oggetti ed è nota come **notazione punto**.

Incapsulamento

Un altro concetto importante della OOP è l'*occultamento di informazione* (information hiding) o incapsulamento; consiste nella progettazione delle classi in modo tale che l'utilizzatore possa fruire delle capacità di un oggetto senza essere tenuto a comprenderne il meccanismo interno.

Tecnicamente si afferma che l'utente dovrebbe interagire con l'oggetto solo tramite la sua **interfaccia**. Il principale vantaggio offerto da questo metodo di sviluppo è la possibilità di migliorare o cambiare il codice di una classe senza che questo possa alterare il

funzionamento del resto del programma; oltre che uno strumento per migliorare la leggibilità del codice, l'occultamento dell'informazione è utile per il riutilizzo del codice già scritto.

L'esempio più esplicativo è quello dell'orologio; come nella realtà non ci poniamo il problema di sapere come un orologio si in grado di tracciare lo scorrere del tempo quando desideriamo sapere l'ora (se questo avviene grazie ad un sistema digitale o meccanico) anche all'interno delle applicazioni gli "utenti" di un ipotetico oggetto "orologio" non dovranno conoscerne il codice interno. Sempre in analogia alla realtà, quando cambiamo orologio è indifferente per noi sapere se il meccanismo interno di quello nuovo è simile o meno a quello vecchio, quello che ci interessa è la sua efficienza; in una applicazione potrebbe essere necessario riscrivere il codice interno dell'oggetto "orologio" per renderlo più efficiente, questo però non comporta una riscrittura del codice interno degli "utenti".

Ereditarietà

Nella sua forma più semplice l'ereditarietà (*inheritance*) è un meccanismo che consente ad una classe di considerarsi erede di un'altra, detta *classe principale* (parent class); una generica notazione per creare una relazione di questo tipo può essere espressa nel seguente modo:

```
classeFiglia ereditaDa (classePadre)
```

Così facendo la *classeFiglia* (classe derivata), eredita tutte le proprietà della *classePadre* specificata, tutti gli attributi e i metodi (anche quelli nascosti).

Da un punto di vista pratico si può ereditare da qualsiasi classe, a patto di non introdurre cicli nel *grafo di ereditarietà*. Tuttavia, da un punto di vista metodologico è opportuno e consigliato ereditare una classe X da una Y solo quando si è certi che tra Y e X esiste una relazione **Y IS-A X** ("Y è un X"), o in altri termini, quando la classe figlia è una specializzazione di quella padre.

Oltre a ereditare, la classe figlia può:

- aggiungere propri attributi o metodi
- ridefinire (overriding) metodi ereditati

Supponiamo di voler scrivere un'applicazione in cui sono presenti degli oggetti "televisione"; in una classe (padre) possiamo descrivere quelle che sono le caratteristiche comuni di ogni oggetto (visualizza un canale, regola il volume...), mentre per creare delle tipologie diverse di "televisione" si scrivono delle nuove classi (figlie) che modificano le proprietà della super-classe (visualizza un canale in bianco e nero), o che ne presentano di nuove (visualizza il televideo).

I linguaggi ad oggetti più evoluti consentono l'**ereditarietà multipla**, la possibilità per una classe di ereditare da più di una classe (cioè di avere più padri). Sorgono però problemi non banali di ereditarietà ripetuta: esiste cioè la possibilità di ereditare più di un metodo con lo stesso nome e occorre risolvere in qualche modo le ambiguità che si vengono a creare.

Polimorfismo

Il concetto di polimorfismo è strettamente legato a quello di ereditarietà; letteralmente il termine indica una pluralità di forme ed è la quarta caratteristica fondamentale della programmazione ad oggetti.

Per polimorfismo si intende la possibilità di invocare la stessa operazione (metodo) ad una pluralità di oggetti differenti. Spesso il polimorfismo viene introdotto con le **classi astratte**, vale a dire classi in cui compaiono uno o più metodi la cui definizione viene rinviata alle sottoclassi (metodi astratti). Un esempio potrebbe essere una classe "animale" che ha un metodo astratto *faiVerso* che trova una concreta definizione solo nelle classi discendenti "cane" o "gatto".

Il polimorfismo trova applicazione soprattutto in schemi simili a quello esemplificato dal seguente frammento:

```
perOgni a : animale in S esegui
a.faiVerso
finePerOgni
```

Si immagina che la variabile *a* assuma di volta in volta i vari oggetti (di tipo "animale") che si trovano in un insieme *S*. Alla variabile *a* verrà quindi di volta in volta associato

un animale potenzialmente diverso e a priori sconosciuto. La chiamata *faiVerso* assumerà forme diverse e per questo è detta polimorfa.

2.1.4 Il linguaggio di programmazione Java

Java³ è un linguaggio di programmazione ad oggetti nato in tempi recenti, ma che ha già avuto un notevole successo e diffusione in tutto il mondo. La fortuna di questo linguaggio è legata in maniera inequivocabile all'espansione della rete Internet che ne ha valorizzato le caratteristiche e gli ha offerto un vasto campo di sviluppo.

Breve storia

La nascita di Java è avvenuta presso i laboratori di Sun Microsystem nei primi anni 90 per opera del programmatore James Gosling. Lo sviluppo di un nuovo linguaggio fu intrapreso per soddisfare determinate esigenze per le quali il C++, che in quel momento era il linguaggio fondamentale dello sviluppo del software, era inadatto. L'indipendenza dalla piattaforma, una gestione della grafica più semplice e una maggiore sicurezza del codice erano i tre principali obiettivi che si volevano raggiungere con questo nuovo linguaggio.

Inizialmente il linguaggio doveva essere destinato ad applicazioni nel campo della televisione interattiva e degli elettrodomestici "intelligenti", ma il fattore decisivo per il suo sviluppo avvenne nel 1992 con la nascita del *Web*. Data la sua indipendenza dalla piattaforma, Java si rivelò subito il miglior linguaggio per sviluppare software in questo nuovo ambito.

A tale scopo furono introdotte le *applet*; queste sono piccole applicazioni di facile implementazione destinate al *Web* nate per dotare di interattività le "statiche" pagine scritte in HTML. Le *applet* sono state senza dubbio l'elemento che ha decretato il successo di Java, portando ad avvicinarsi a tale linguaggio un numero sempre più alto di persone.

Ad oggi vi sono state quattro fasi nell'evoluzione di questo linguaggio:

- Java 1.0 (1995): una versione prevalentemente destinata al World Wide Web;

³L'origine del nome è alquanto misteriosa: si sostiene che potrebbe essere l'acronimo di *Just Another Vouge Acronym*, oppure che derivi dal marca di caffè preferita dallo sviluppatore o semplicemente che sia stato scelto per il suo "suono".

- Java 1.1 (1997): un aggiornamento che includeva vari miglioramenti nel modo in cui le interfacce grafiche venivano create e gestite;
- Java 2 (1998): la versione che ha offerto al linguaggio gli strumenti per competere con gli altri linguaggi presenti sul mercato per un impiego generale;
- Java 2 versioni 1.3, 1.4 (2000-2002): gli aggiornamenti per rendere l'esecuzione dei programmi più veloce, funzioni multimediali avanzate ed il supporto al sistema operativo Linux.

Le caratteristiche

Java mantiene alcune affinità con il C++ che sono state volutamente lasciate per renderlo più familiare e semplice da imparare al vasto numero di programmatori di questo linguaggio.

La similitudine sintattica con il più diffuso linguaggio di programmazione permette un più rapido apprendimento di *Java* da parte dei programmatori di C e di C++. Per esempio, le istruzioni `for`, `if`, `else` sono del tutto analoghe a quelle del C.

Ma pur mantenendo alcune affinità Java nasce proprio dall'esigenza di colmare alcune lacune lasciate proprio dal C++ e cioè un linguaggio che permettesse di utilizzare costrutti più semplici, sicuri ed affidabili.

Sintetizzando, le caratteristiche più "pericolose" di C e C++ che mancano in Java sono:

- **Assenza del preprocessore.** In Java non esistono file di *include*, né dichiarazioni di *define* che possano generare confusione. Attraverso le dichiarazioni di *define*, infatti, si possono rendere i programmi incomprensibili a chi li legge per la prima volta. Non è prevista neppure l'istruzione `typedef`.
- **Assenza di strutture e unioni.** Esse possono essere implementate come classi contenenti le variabili volute insieme ai metodi con cui accedere a queste variabili.
- **Assenza di funzioni.** Esse sono rimpiazzate dai metodi delle classi. Riducendo tutto a classi, si guadagna in semplicità poiché si evita di introdurre la vecchia distinzione tra procedura e funzione. Ciò permette anche di non corrompere la purezza di un linguaggio completamente orientato all'oggetto.

- **Assenza di ereditarietà multipla.** Per ovviare alla carenza di metodi che una classe può ereditare, in Java sono state create le interfacce (*interface*), opportune collezioni di metodi che possono essere implementati negli oggetti.
- **Assenza completa del "goto".** L'istruzione `goto` è usata nella pratica solo per uscire da cicli `for` annidati. In Java le istruzioni di *break* multilivello e *continue* eliminano completamente la necessità del `goto`.
- **Assenza di forzatura automatica dei tipi.** Nei casi in cui il cambiamento di tipo di un certo dato può causare perdita di informazione (per esempio quando si assegna un numero `float` ad una variabile di tipo `int`), il compilatore richiede una opportuna istruzione di *cast*.
- **Assenza di puntatori.** Questa caratteristica è stata introdotta per limitare il gran numero di errori dovuti all'uso dei puntatori nella programmazione in C e C++. Del resto, in Java non c'è bisogno di puntatori, dal momento che sia le stringhe che i vettori (`array`) sono oggetti e le strutture non sono previste. Agli elementi degli `array` si accede tramite il loro indice; inoltre durante l'esecuzione di un programma è sempre controllato se l'indice dell'`array` eccede la dimensione dell'`array` stesso.

Altra caratteristica di Java sono le dimensioni ridotte del software, in modo che esso possa funzionare anche su sistemi molto piccoli (come i telefoni cellulari).

Java è un linguaggio *object oriented* "puro" poiché rispetta tutti i quattro principi di questo paradigma, cosa non vera per il C++ che perde alcune delle caratteristiche del paradigma OO per mantenere la compatibilità con il C.

Esistono infatti le classi, le istanze delle quali costituiscono gli oggetti. Le variabili istanziate mantengono l'informazione di stato dei vari oggetti. Si può rendere riservata l'informazione attraverso la dichiarazione `private`, oppure la si può rendere astratta tramite la dichiarazione `abstract` (incapsulamento). Esiste comunicazione tra gli oggetti. I metodi posseduti da un oggetto stabiliscono il comportamento dell'oggetto stesso e manipolano le variabili istanziate della classe. In classi distinte possono esistere metodi diversi con lo stesso nome. Ogni oggetto risponde ai messaggi ricevuti secondo le caratteristiche che esso

possiede (comportamento polimorfico). Da una classe possono essere create sottoclassi, le quali possiedono i metodi della classe da cui sono derivate (ereditarietà). Le classi derivate possono aggiungere nuovi metodi alla classe da cui derivano oppure possono sostituire ad essa alcuni metodi.

Come già accennato, una delle caratteristiche fondamentali della programmazione ad oggetti è quella legata alla riusabilità del codice, cioè al fatto che una classe creata per un certo programma possa essere riutilizzata tranquillamente per un altro. Sebbene il C++ permettesse il riutilizzo di porzioni di codice, Java sfrutta meglio questa peculiarità poiché la sua sintassi costringe il programmatore a ragionare in termini di "oggetti".

Diametralmente opposto al C++, Java è un tipo di linguaggio che per volontà dei propri creatori non privilegia le prestazioni, ma favorisce in generale la semplicità nella gestione di gran parte delle risorse ricorrendo spesso all'automazione di queste. Tale caratteristica fa sì che Java risulti più potente, rispetto agli altri linguaggi, nella gestione di parti complesse della programmazione come la gestione della grafica o dei collegamenti in rete.

In ogni programma Java è sempre attivo l'*automatic garbage collection*, uno strumento che provvede ad analizzare la memoria e a liberare quella che non è utilizzata. In Java quindi esiste soltanto la possibilità di allocare memoria attraverso la creazione di oggetti mediante l'istruzione `new`, similmente al C++. Una volta che un oggetto è stato creato, il *garbage collection* in fase di esecuzione osserva la vita dell'oggetto e provvede a liberare la memoria occupata quando essa non viene più utilizzata dall'oggetto in questione. Per esempio provvede a liberare la memoria occupata da un oggetto creato in un metodo una volta che il medesimo metodo termina la sua esecuzione. Siccome si può allocare memoria solo con oggetti, il controllo degli oggetti è sufficiente a gestire con buona efficienza la memoria stessa. Ciò solleva il programmatore dal provvedere alla gestione della memoria, compito questo che in C e C++ conduce spesso all'introdurre errori che si verificano in fase di esecuzione dei programmi.

La neutralità dell'architettura rappresenta un'altra caratteristica importante in un mondo informatico sempre maggiormente popolato dai più svariati sistemi hardware e sistemi operativi. Con la fortissima crescita delle reti telematiche questa esigenza è divenuta

ancora più pressante. Inoltre, per sviluppare prodotti software per il *Web*, la neutralità dall'architettura costituisce un parametro assolutamente indispensabile. Al fine di raggiungere la neutralità dall'architettura, Java affida l'esecuzione dei suoi programmi ad una macchina virtuale. I programmi Java quindi non hanno bisogno di alcun riadattamento quando vengono portati su sistemi differenti da quelli su cui sono stati sviluppati.

Il linguaggio Java di base fornisce le librerie di classi adatte a sviluppare numerose applicazioni di qualsiasi genere. I principali gruppi delle classi fondamentali sono:

- **java.lang:** tipi di base sempre importati in ogni programma. Per esempio qui si trovano la dichiarazione di `Object` (superclasse di tutte le classi) e la dichiarazione del tipo `Class`, che serve per creare una classe di qualunque genere.
- **java.io:** librerie di classi adatte per gestire stream di dati e accesso casuale ai file.
- **java.net:** librerie di classi che provvedono a gestire *socket*, interfacce *telnet*, *Url*.
- **java.util:** librerie di utilità varie, come data, orologio, tecniche di codifica e decodifica.
- **java.awt:** (Abstract Windowing Toolkit) libreria di classi adatta a sviluppare applicazioni per ambienti a finestra. Contiene interfacce grafiche standard come gestione degli eventi, colori, barre di scorrimento e pulsanti.

Interpretato o compilato?

Un programma Java deve essere compilato per l'esecuzione, ma ciò che si ottiene non è un codice eseguibile dal processore, ma pseudo-codice solitamente chiamato *bytecode*. Il codice ottenuto è eseguibile solo con l'uso di un apposito interprete, il comando *java*. Poiché molti linguaggi interpretati vengono eseguiti direttamente a partire dal codice sorgente, ci si potrebbe chiedere perché in Java sia necessario questo ulteriore passaggio. Sebbene esistano degli interpreti Java "puri", la presenza del *bytecode* permette un avvio più rapido dei programmi creati poiché non è necessaria una fase di analisi generale del codice.

Da notare anche come Java carichi dinamicamente le classi quando sono richieste tramite il comando `java -verbose`; ogni programma, anche il più semplice, richiama molte classi collegate tra loro. Se ogni classe dovesse essere controllata prima dell'esecuzione il rallentamento all'avvio dell'applicazione sarebbe inevitabile. Lanciando questo comando otterremo un output del tipo:

```
[Opened /usr/local/j2sdk1.4.1_01/jre/lib/rt.jar]
[Opened /usr/local/j2sdk1.4.1_01/jre/lib/sunrsasign.jar]
[Opened /usr/local/j2sdk1.4.1_01/jre/lib/jsse.jar]
[Opened /usr/local/j2sdk1.4.1_01/jre/lib/jce.jar]
[Opened /usr/local/j2sdk1.4.1_01/jre/lib/charsets.jar]
[Loaded java.lang.Object from /usr/local/j2sdk1.4.1_01/jre/lib/rt.jar]
[Loaded java.io.Serializable from /usr/local/j2sdk1.4.1_01/jre/lib/rt.jar]
[Loaded java.lang.Comparable from /usr/local/j2sdk1.4.1_01/jre/lib/rt.jar]
[Loaded java.lang.CharSequence from /usr/local/j2sdk1.4.1_01/jre/lib/rt.jar]
[Loaded java.lang.String from /usr/local/j2sdk1.4.1_01/jre/lib/rt.jar]
[Loaded java.lang.Class from /usr/local/j2sdk1.4.1_01/jre/lib/rt.jar]
[Loaded java.lang.Cloneable from /usr/local/j2sdk1.4.1_01/jre/lib/rt.jar]
```

In sostanza il compito del compilatore Java è di togliere l'aspetto di analisi della sintassi da parte dell'interprete, rendendolo così più efficiente. Per vedere cosa effettivamente esegue l'interprete è possibile lanciare il comando `javap -c nomeProgramma` ottenendo un output del tipo:

```
public class HelloWorld extends java.lang.Object {
    public HelloWorld();
    public static void main(java.lang.String[]);
}
```

```
Method HelloWorld()
  0 aload_0
```

```
1 invokespecial #1 <Method java.lang.Object()>
4 return
```

```
Method void main(java.lang.String[])
```

```
0 getstatic #2 <Field java.io.PrintStream out>
3 ldc #3 <String "Hello, World!">
5 invokevirtual #4 <Method void println(java.lang.String)>
8 return
```

Il comando *javap* è un analizzatore di classi (per vedere quali sono i metodi disponibili) ma può anche essere utilizzato per mostrare il codice delle istruzioni. L'interprete Java, detto Java Virtual Machine (JVM), è simile a un microprocessore software, ma è progettato per essere potenzialmente realizzabile in hardware.

2.2 La libreria Swarm

2.2.1 Il progetto e le sue finalità

Il progetto Swarm nacque nel 1995 presso il Santa Fe Institute del New Mexico da un'idea di Chris Langton.

Swarm is a kernel and library for the multi-agent simulation of complex systems. The basic architecture of Swarm is a collection of concurrently interacting agents: within this architecture, a large variety of agent based models can be implemented⁴.

L'obiettivo principale era la creazione di uno strumento grazie al quale gli sviluppatori di modelli di simulazione potessero risparmiare tempo nello studio e nella scrittura del codice per concentrare maggiormente la loro attenzione sugli aspetti di loro competenza. Swarm semplifica la creazione di modelli di simulazione Agent Based poiché fornisce un insieme di librerie software utili per la creazione di grafici, interfacce utente, gestione del tempo

⁴<http://savannah.nongnu.org/projects/swarm/>

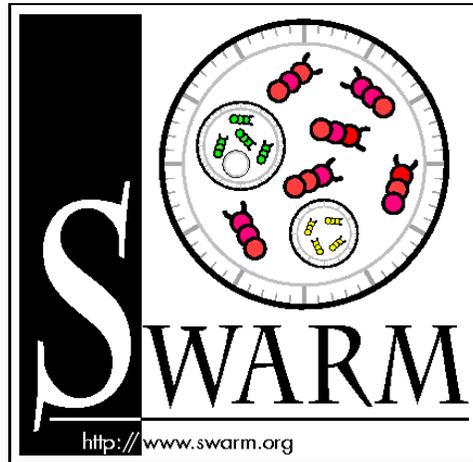


Figura 2.1. Il logo del progetto Swarm

ed altre funzionalità che risultano spesso, ai programmatori neofiti, troppo complicate e dispendiose da realizzare.

In particolare le librerie Swarm sono orientate alla costruzione di simulazioni nelle quali uno sciame (da qui il nome) di agenti eterogenei interagiscono in un ambiente sulla base di eventi discreti. Potenzialmente si possono creare simulazioni in tutti i campi delle scienze, dalla biologia all'antropologia, dalle scienze economiche all'ecologia.

La principale motivazione che ha portato allo sviluppo di Swarm è che le simulazioni al computer presentano, ancora oggi, il grande svantaggio di non utilizzare una serie di strumenti standard e conosciuti dall'intera comunità scientifica: chi sviluppa delle simulazioni è, purtroppo, in parte obbligato a crearsi i propri "ferri del mestiere". Un tale situazione comporta alcune conseguenze durante la fase di divulgazione dei risultati scientifici: il ricercatore deve specificare le caratteristiche degli strumenti utilizzati per dimostrarne l'affidabilità dei risultati che ha ottenuto, ma questo comporta un aumento della documentazione necessaria e quindi la difficoltà di esprimere in modo sintetico gli argomenti oggetto della ricerca.

Il progetto Swarm intende superare questi ed altri ostacoli nel campo delle simulazioni fornendo strumenti di sviluppo affidabili e testati in diverse situazioni da una larga comunità scientifica; tutti i componenti di Swarm e le sue librerie sono, infatti, distribuite sotto licenza GNU Public Lincese (GPL) il che permette ad ogni utilizzatore di studiare,

modificare e correggere liberamente il codice sorgente di tutte le librerie.

La GPL (General Public License) è la più diffusa licenza di distribuzione del software libero (detto open source). Fu Richard M. Stallman, nei primi anni Ottanta, a formalizzare per la prima volta il concetto di software libero; secondo l'autore perchè un prodotto software possa essere defito tale deve soddisfare quattro principi di libertà:

- **libertà 0:** libertà di eseguire il programma per qualunque scopo, senza vincoli sul suo utilizzo;
- **libertà 1:** libertà di studiare il funzionamento del programma, e di adattarlo alle proprie esigenze;
- **libertà 2:** libertà di redistribuire copie del programma;
- **libertà 3:** libertà di migliorare il programma, e di distribuirne i miglioramenti.

Nel 1984 Richard M. Stallman diede vita al progetto GNU, con lo scopo di tradurre in pratica il concetto di software libero, e creò la Free Software Foundation per dare supporto logistico, legale ed economico al progetto GNU.

La licenza d'uso è un documento legale generalmente distribuito assieme a ogni programma. Essa, appoggiandosi alle norme sul diritto d'autore, specifica diritti e doveri di chi riceve tale programma. La licenza del progetto GNU concede all'utente del programma tutte e quattro le libertà suddette. Inoltre si occupa anche di proteggerle: chi modifichi un programma protetto da GPL e lo distribuisca con tali modifiche, deve distribuirlo sotto licenza GPL. È grazie a questo tipo di protezione che la GPL è attualmente la licenza più usata per il software libero⁵.

⁵Una panoramica completa, per la gran parte tradotta in lingua italiana, sulle diverse tipologie di licenze open source è disponibile sul sito GNU, agli indirizzi <http://www.gnu.org/licenses/license-list.it.html>, e <http://www.gnu.org/philosophy/categories.it.html>

2.2.2 Uno sciame di...

Le applicazioni in Swarm sono basate sul concetto di swarm (sciame). Uno swarm è un insieme di uno o più oggetti e di una tabella dei tempi in cui vengono regolate, cronologicamente, le attività di tali oggetti.

Il modello di simulazione è quindi rappresentato da uno o più sciami che interagiscono all'interno di un ambiente; ogni sciame è costituito da agenti e tempi ed ogni agente è rappresentato da un insieme di regole e dalla capacità di rispondere a degli stimoli.

Gli agenti che compongono la simulazione possono rappresentare le entità che abbiamo intenzione di osservare nel nostro esperimento (agenti primari), oppure possono essere oggetti non riconducibili a nessuna entità realmente esistente, ma utili ai fini della simulazione (agenti ausiliari).

Si potrebbe immaginare una simulazione in cui l'ambiente è l'Università che detta i tempi per frequentare i corsi e poi sostenere gli esami; gli agenti, in questo caso, sarebbero sciami di studenti e docenti che si comportano in base al tempo (il periodo dei corsi, le sessioni d'esame, ...) ed alle interazioni con gli altri agenti.

Grazie agli strumenti messi a disposizione da Swarm risulta di facile implementazione di simulazioni in cui alcuni agenti sono a loro volta uno sciame; una metodologia di questo tipo permette di sviluppare simulazioni osservabili a più livelli (swarm e sub-swarm).

Nel nostro esempio potremmo creare uno sciame di Università per studiare, ad esempio, se esistono fenomeni di competizione o di emulazione; ogni agente-Università sarà composto da sciami di docenti e studenti con le loro tabelle dei tempi.

La possibilità di creare sciami all'interno di altri sciami ci permette anche di simulare la rappresentazione del mondo che ogni agente si crea, rappresentazione che influenzerà le sue decisioni nel mondo "reale".

In questo caso d'analisi, ogni agente-studente prevederebbe al suo interno uno sciame di Università col fine di simulare la "personale" rappresentazione del "vero" ambiente Università nel quale si trova immerso. Le rappresentazioni potrebbero essere differenti da studente a studente, e quindi differenti saranno le loro azioni intraprese.

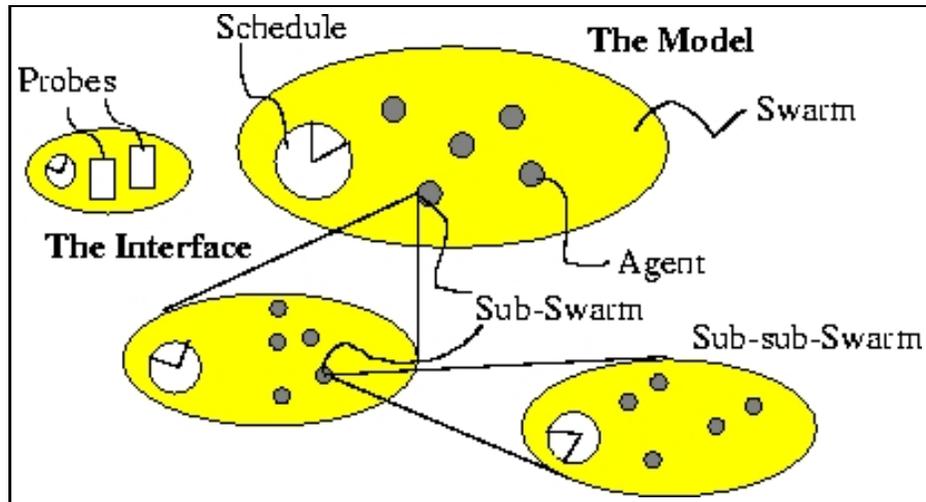


Figura 2.2. Gli agenti di una simulazione: swarm e sub-swarm

2.2.3 La gestione del tempo

La gestione del tempo è una delle componenti fondamentali di ogni simulazione. Gli eventi reali che si intende simulare sono normalmente formati da una serie di azioni che avvengono contemporaneamente; nel nostro esempio, due studenti nello stesso istante potrebbero sostenere un esame oppure seguire un corso.

Nella programmazione tradizionale (sia ad oggetti, sia procedurale) non è normalmente possibile scrivere programmi in grado di eseguire realmente dei calcoli in modo parallelo; è per questo motivo che nelle simulazioni deve essere simulato anche il trascorrere del tempo.

Swarm semplifica questo aspetto fornendo alcuni strumenti di programmazione, ma soprattutto un insieme di regole per costruire i modelli simulativi in modo coerente. Abbiamo già visto che ogni sciame oltre a contenere un insieme di agenti, per essere definito tale, deve contenere una tabella dei tempi. La sincronizzazione tra sciami avviene grazie ad un orologio principale al quale fanno riferimento tutti gli sciami: per ottenere la scincronia degli eventi, durante l'esecuzione della simulazione si scorre la lista degli orologi di ogni sciame per informare gli agenti, tramite un messaggio, di svolgere le operazioni previste (Lin, Tan, Shaw, 1996).

In un'applicazione Swarm possono specificarsi messaggi da inviare ai diversi oggetti e ai

diversi swarm componenti il modello: questi messaggi rappresentano le azioni che i nostri agenti (o sciame) dovranno compiere in determinati momenti della simulazione.

Le azioni sono strettamente correlate ai metodi delle diverse classi presenti nel programma che il programmatore ha scritto utilizzando uno dei due linguaggi messi a disposizione (Java e ObjectiveC). Tutte le azioni che devo essere attivate attraverso un messaggio sono incapsulate in un oggetto istanziato dalla classe *Selector* messa a disposizione dalle librerie di Swarm.

Terminata questa fase, vengono creati altri oggetti istanze di un'altra classe di Swarm detta *ActionGroup*; ciascun oggetto di tipo *actionGroup* contiene un certo numero di *selector* a ciascuno dei quali viene associato un destinatario od un gruppo di destinatari (i nostri studenti o docenti). Procedendo in questo modo è ora possibile indirizzare le azioni contenute dei vari *selector* ad un determinato oggetto (o gruppo di oggetti).

Un *actionGroup* è quindi un insieme di azioni che devono essere svolte simultaneamente all'interno del tempo simulato. Un *actionGroup*, per l'esempio che stiamo utilizzando, potrebbe racchiudere tutte le azioni richieste a studenti e docenti per portare a termine una sessione d'esame, che ovviamente dovranno essere svolte dai due tipi di agenti in modo parallelo (se il docente non si presenta alla sessione d'esame lo studente non può sostenere la prova orale).

Giunti a questo punto è necessario inserire tutti gli *actionGroup* creati in una tabella dei tempi ottenibile istanziando un oggetto della classe Swarm *Schedule*. Per ogni *schedule* devono essere specificati gli *actionGroup*, il tempo di inizio e l'eventuale periodo durante il quale devono essere lanciati i messaggi. Utilizzando le *schedule* saremo in grado di sviluppare simulazioni all'interno delle quali un insieme di messaggi, raggruppati in un *actionGroup*, vengono inviati ai rispettivi destinatari in un determinato ciclo di simulazione e, eventualmente, ripetuto a determinati intervalli. La *schedule* del nostro esempio potrebbe essere programmata in modo tale che ad intervalli di 3 settimane (simulate) gli agenti (studenti e docenti) effettuino un insieme di azioni necessarie per sostenere prova d'esame.

2.2.4 Come costruire un modello

Ogni modello Swarm prevede tra fasi di sviluppo:

- la scrittura degli agenti;
- la scrittura e formalizzazione del modello ;
- la scrittura di un oggetto in grado di osservare il modello.

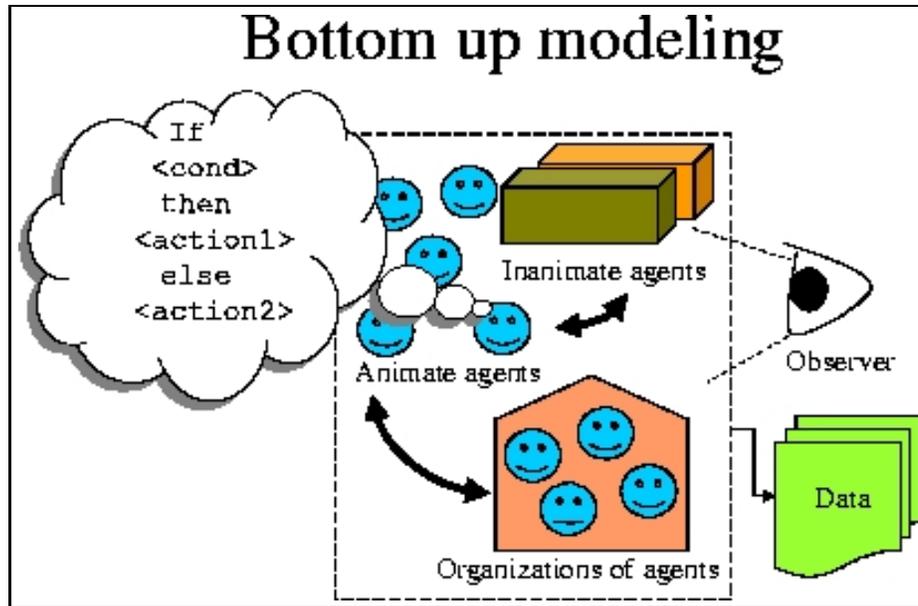


Figura 2.3. Rappresentazione di un modello di simulazione in Swarm

Gli agenti

La prima operazione consiste nello scrivere le classi che descrivono i soggetti che compongono la simulazione che vogliamo rappresentare; questi saranno o agenti dei quali intendiamo osservarne il comportamento o altri oggetti comunque necessari al funzionamento della simulazione. Ogni oggetto creato nell'ambiente simulativo deve essere un'istanza di una classe che estende la classe *SwarmObjectImpl*. In pratica, continuando ad utilizzare il solito esempio, per la nostra simulazione dell'Università dovremo creare almeno due classi (tutte estensioni della classe *SwarmObjectImpl*): *Studente.java* e *Docente.java*. Queste due classi

descrivono gli agenti "primari" poiché siamo interessati ai loro comportamenti; supponendo invece che l'Università sia semplicemente lo spazio all'interno del quale si muovono studenti e docenti, sarà da considerare come un agente ausiliario.

ModelSwarm

La seconda fase prevede la scrittura di una classe denominata *ModelSwarm*; al suo interno si descrive come devono essere aggregati gli oggetti creati nella precedente fase, ma soprattutto si specificano i tempi con i relativi messaggi che devono essere a loro inviati.

Una volta creato il *ModelSwarm*, è necessario richiamare due suoi metodi fondamentali: *buildObjects()* e *buildActions()*. Quando viene richiamato il metodo *buildObjects()*, il *ModelSwarm* dà luogo ad una duplice operazione: crea tutti gli oggetti in esso contenuti, e richiama il metodo *buildObjects()* degli eventuali swarm contenuti in esso. Analogamente, quando viene richiamato il metodo *buildActions()*, il *ModelSwarm* dà luogo ad una duplice operazione: crea tutti gli *actionGroups* e le *schedules* necessari, e richiama il metodo *buildActions()* degli eventuali swarm contenuti in esso.

Ultimo compito del *ModelSwarm* è l'unione di tutti i pezzi appena creati all'interno di un unico oggetto che successivamente potremo eseguire; questa fase viene realizzata attraverso il metodo *ActivateIn*.

ObserverSwarm

Con la scrittura della classe *ObserverSwarm*, alla fine, saremo in grado di osservare in tempo reale lo stato della simulazione nel suo aggregato (osservando il *ModelSwarm*) oppure lo stato di un singolo agente. Grazie a questa classe, in oltre, attiviamo un'interfaccia grafica utile sia per avviare o fermare la simulazione, sia per intervenire (con delle sonde) sulle variabili che caratterizzano il nostro modello.

Il *ModelSwarm* viene costruito (istanziato) all'interno di questa classe; questo quindi permette, potenzialmente, anche di utilizzare un unico "osservatore" per più modelli (uno scame di modelli) che vogliamo analizzare e confrontare in parallelo.

La nostra simulazione d'esempio potrebbe contenere un grafico con il numero di esami

sostenuti, quelli superati, una media dei voti di tutti gli studenti; in questo caso osserveremo il modello nel suo aggregato. Allo stesso tempo potrebbe essere utile analizzare l'andamento di un singolo studente per confrontare il suo andamento rispetto a ciò che sta avvenendo in aggregato.

Il concetto di zona (Zone)

La classe *ZoneImpl* definisce oggetti utilizzati come zone di memoria in cui salvare gli oggetti utilizzati nella simulazione. Ogni volta che un nuovo oggetto viene creato, deve essere identificata una zona che contenga le sue variabili di istanza o altri tipi di dati interni all'oggetto stesso.

Un programma può stabilire più zone nelle quali allocare oggetti che abbiano una durata media paragonabile, o simili esigenze di memoria. L'intenzione degli sviluppatori si Swarm è di ottimizzare l'allocazione dinamica della memoria, e il riutilizzo di porzioni di essa lasciate libere da oggetti non più in uso nel programma.

Lo swarm principale viene creato ed allocato in una zona denominata *globalZone*. Ogni swarm successivo, generato all'interno della simulazione, ha la possibilità di essere utilizzato come spazio di memoria, costituisce quindi di per sé una zona in cui possono essere allocati gli oggetti che lo compongono.

Al contrario, gli oggetti (istanze di classi derivate dalla classe *SwarmObject*) non hanno questa possibilità, per cui devono essere salvati in zone di memoria specifiche di determinati swarm, oppure nella *globalZone*.

Le sonde (probe)

L'utente può interagire con una simulazione, e di conseguenza sul modello, inserendo o modificando variabili anche in tempo reale grazie all'impiego delle probes. Le probes sono come delle sonde sul modello che l'osservatore può utilizzare attraverso una comoda interfaccia grafica, senza quindi dover intervenire direttamente sul codice. La libreria che consente l'inserimento di sonde è *ObjectBase*.

Figura 2.4. Un esempio di *probe*

Le liste

Sono previsti in Swarm dei raccoglitori dinamici di oggetti, come agenti o sciami, detti liste.

Per il programmatore che utilizza tecniche Object Oriented queste caratteristiche sono molto utili poiché consentono di istanziare da una classe (pensiamo a *Studente.java*) un unico oggetto (seguito la consuetudine si crea un oggetto chiamato *unoStudente*) per poi inserirlo più volte all'interno di una lista.

Completata la lista il programmatore non dovrà più operare sui singoli oggetti, ma sulla lista stessa. La libreria Swarm per la gestione delle liste è *Collections* e contiene le classi necessarie alla strutturazione delle liste, come i metodi di inserimento o rimozione degli oggetti dalle stesse.

2.2.5 Le librerie a disposizione

Swarm si presenta all'utente finale come un insieme di librerie che possono essere utilizzate programmando in linguaggio Objective-C o Java. Le principali librerie sono:

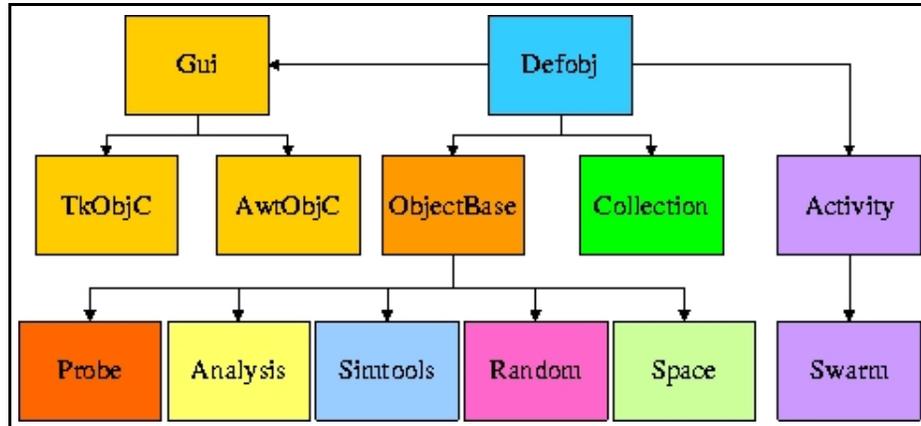


Figura 2.5. L'albero delle librerie di Swarm

defobj E' la classe principale (root) e fornisce i metodi per la creazione e distruzione degli oggetti. Contiene anche delle classi per generare, ma soprattutto archiviare, le istanze degli oggetti (gli sciami).

objectbase Contiene le classi fondamentali per il funzionamento degli agenti all'interno della simulazione. Al suo interno troviamo le classi *Swarm* e *SwarmObject*; la prima viene utilizzata per la definizione del *ModelSwarm* e dell'*ObserverSwarm*, la seconda è la classe dalla quale ereditano tutti gli oggetti che andremo ad utilizzare.

activity Permette la creazione del calendario degli eventi (*Schedule*) e l'aggiornamento dei grafici presenti nell'interfaccia utente.

collections Fornisce le classi necessarie per creare liste, vettori e mappe. Sono presenti anche i metodi per inviare messaggi agli oggetti della simulazione, l'ordinamento e l'eliminazione di gruppi di agenti.

space Viene utilizzata per inserire gli agenti della simulazione all'interno di uno spazio bidimensionale, una griglia definita *discrete2DLattice*.

gui Comprende le classi per la gestione di tutta la componente grafica; sono presenti diverse tipologie di grafici facilmente integrabili nella simulazione.

analysis Utile per la creazione di grafici su serie di tempo e per la gestione degli output grafici in tempo reale.

random L'utilizzodi numeri casuali risulta essere sempre la parte più delicata di ogni simulazione; gli strumenti per generarli possono influenzare l'affidabilità dei risultati che osserveremo nel nostro modello. Swarm fornisce una solida e vasta libreria di generatori di numeri casuali secondo gli algoritmi matematici tratti dalla letteratura in materia. L'utente può specificare la distribuzione che intende utilizzare (gamma, beta, normale, ...) con il seme per la generazione dei numeri, in questo modo potrà riutilizzare la stessa sequenza di numeri in esperimenti successivi (il modello risulta confrontabile).

simtools Fornisce le classi necessarie per il controllo della simulazione; in particolare contiene gli strumenti per l'interazione con file esterni (solitamente utilizzati per leggere o archiviare dati).

tkobjc Basata su Tcl/Tk⁶, è la libreria che permette la creazione e gestione di tutti gli oggetti grafici.

2.3 La notazione UML

2.3.1 Perché un linguaggio comune?

La continua nascita di nuovi metodi per l'analisi e la progettazione ad oggetti, il cui numero delle più diffuse sembra oramai sfiorare il centinaio, ha determinato la necessità di uno standard notazionale. Nasce così **Unified Modeling Language** (UML) dall'evoluzione

⁶Il linguaggio Tcl, insieme con il toolkit grafico Tk, è stato sviluppato da John Ousterhout e costituisce uno strumento facile e potente per la creazione di applicazioni personalizzate con Linux. In particolare Tcl è un linguaggio di scripting (come Perl, gli shell script UNIX, e il Visual Basic) il cui scopo è principalmente quello di essere esteso. Tk è un'estensione molto potente di Tcl che rende più efficiente e semplice la creazioni di interfacce utente.

di strumenti già utilizzati in passato, come i diagrammi *Flow-Chart*, e l'unificazione (da qui il nome) di tre notazioni precedentemente esistenti: Booch (dell'omonimo autore), OMT (di Rumbaugh) e OOSE (di Jacobson). Il "progetto UML" parte nel 1994, con l'ingresso di Jim Rumbaugh nella società di Grady Booch, la Rational, e si perfeziona con l'acquisizione da parte della Rational, a fine 1995, dell'azienda svedese Objectory, in cui opera Ivar Jacobson.

L'*Object Management Group* (OMG)⁷, il consorzio che approva le specifiche del linguaggio, all'interno dell'*OMG Unified Modeling Language Specification* definisce:

The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of the best engineering practices that have proven successful in the modeling of large and complex systems.

Sostituendosi alla grande varietà di metodi di analisi Object Oriented praticati in passato, UML sta diventando una notazione ed una semantica comune a livello internazionale. L'obiettivo degli autori è ambizioso: secondo le intenzioni dell'OMG dovrebbe diventare un linguaggio universale per la **progettazione** dei sistemi informatici. UML è, infatti, un linguaggio ed una notazione universale per analizzare, rappresentare, specificare e documentare lo sviluppo di qualsiasi tipo di sistema ad oggetti (software, hardware, organizzativo, ...), ma oltre ad essere una notazione, UML è un meta-modello che descrive la notazione stessa.

UML non deve essere confuso con altri linguaggi di programmazione come C++ o Java, poiché è "solamente" un linguaggio di rappresentazione dei modelli; lo sviluppo del progetto, sulla base del modello UML, viene effettuato comunque utilizzando i classici strumenti di programmazione.

Il sempre maggior successo riscosso da UML è giustificato dal fatto che finalmente si offre una risposta ai problemi legati allo sviluppo di sistemi complessi all'interno di

⁷Il sito dell'OMG è consultabile all'indirizzo <http://www.omg.org>

ambienti visuali, permettendo una maggiore attenzione alla modellizzazione degli aspetti architettonici e favorendo, grazie al suo meta-modello, la comunicazione tra gli strumenti che vengono utilizzati dai diversi progettisti coinvolti nello sviluppo del sistema.

UML non è, però, privo di difetti. In primo luogo bisogna ammettere che si presenta, agli occhi di uno sviluppatore principiante, molto complesso. La sua complessità è dovuta principalmente dai seguenti fattori:

- intende rappresentare qualunque tipo di sistema software, a livelli di astrazione differenziati;
- il numero degli strumenti è elevato, ad esempio è possibile scegliere tra forme di rappresentazione diverse;
- non suggerisce, né prescrive una sequenza di realizzazione dei diagrammi;
- offrendo un'ampia gamma di possibili modalità di utilizzo, tra le quali i progettisti sono liberi di scegliere, è possibile comprendere ogni tipologia di modello solo grazie ad una padronanza completa dell'intero linguaggio.

2.3.2 Il meta-modello

Il principale difetto della maggior parte dei modelli di rappresentazione Object Oriented è di essere poco rigorosi; un modo per aumentare il rigore di presentazione di una notazione consiste nel descrivere prima di tutto il suo meta-modello. UML è basato su un meta-modello integrato, composto da numerosi elementi collegati tra loro secondo regole precise; utilizzando gli elementi che il meta-modello ci mette a disposizione è possibile costruire i modelli per i sistemi che vogliamo realizzare.

Il linguaggio UML contiene molti elementi grafici che vengono messi insieme durante la creazione dei diagrammi; l'obiettivo dei diagrammi è di rappresentare visualizzazioni differenti del sistema in base al contesto che i progettisti intendono analizzare. La maggior parte degli elementi di un diagramma sono rappresentati con un'icona e possono apparire in diagrammi differenti a seconda del tipo di osservazione che stiamo facendo. UML, oltre ad essere una notazione, è un linguaggio: esistono nel meta-modello delle regole per unire

i vari elementi dei diagrammi. La presenza di regole rigide per la realizzazione dei grafici ci permette di verificare (in tempo reale) la correttezza e coerenza del modello in fase di sviluppo.

Le classi

Dal momento che si tratta di una notazione Object Oriented, le entità del sistema sono rappresentate mediante classi; di ogni classe si specifica la struttura e l'elenco delle operazioni che mettono a disposizione all'interno del modello. In un tipico diagramma UML una classe viene generalmente rappresentata da un rettangolo con al suo interno il nome della classe che, per convenzione, è una parola con l'iniziale maiuscola ed appare vicino nella sommità. Se il nome della classe definita consiste di una parola composta a sua volta da più parole allora viene utilizzata la notazione in cui tutte le iniziali di ogni parola sono scritte in maiuscolo.

Ogni classe può avere al suo interno diversi attributi che rappresentano delle sue proprietà; ogni attributo descrive un insieme di valori che la proprietà può avere quando vengono istanziati oggetti di quella determinata classe. Un attributo il cui nome è costituito da una sola parola viene scritto sempre in caratteri minuscoli, se, invece, il nome dell'attributo è formato da più parole il nome dell'attributo è scritto unendo tutte le parole che ne costituiscono il nome stesso, con la particolarità che la prima parola è scritta in minuscolo mentre le successive avranno la loro prima lettera in maiuscolo. Il nome della classe, presenta sulla sommità del rettangolo, viene separato dalla lista degli attributi con una linea orizzontale.

In alcuni modelli possono essere presenti delle classi che non permettono di istanziare nessun tipo di oggetto: classi di questo tipo sono definite astratte. Graficamente, in UML, una classe astratta si indica scrivendo il suo nome in corsivo.

Le informazioni addizionali che possono essere unite agli attributi di una classe possono essere di tipo *constraints* o note. Le *constraints* sono delle caselle di testo racchiuse tra parentesi e sono utilizzate per specificare una o più regole che la classe è tenuta a seguire obbligatoriamente; le note sono elementi di testo e grafici, solitamente associati agli

attributi ed ai metodi, utilizzate per fornire informazioni aggiuntive ad una classe.

In modo analogo agli attributi, i metodi di una classe sono scritti con caratteri minuscoli se composti da una sola parola, con le iniziali maiuscole (ad esclusione della prima parola) se composti da più parole. La lista dei metodi è rappresentata graficamente sotto la lista degli attributi e separata da questa tramite una linea orizzontale. I metodi possono anche presentare delle informazioni aggiuntive: nelle parentesi che seguono il nome di un metodo è possibile mostrare gli eventuali parametri necessari specificandone il tipo (int, String, ...); quando il metodo rappresenta una funzione è necessario specificare anche il tipo restituito (return).

Per ogni attributo e metodo di una classe bisogna specificare la possibilità che hanno le altre classi di utilizzarli. La possibilità di utilizzo o meno viene impostata attraverso la visibilità per la quale sono consentiti tre livelli di visibilità, rappresentati da altrettanti elementi grafici:

livello pubblico: l'utilizzo viene esteso a tutte le classi (segno grafico +);

livello protetto: l'utilizzo è consentito soltanto alle classi che derivano dalla classe originale (segno grafico #);

livello privato: soltanto la classe originale può utilizzare questi attributi e metodi (segno grafico -).

Un insieme di operazioni che una classe offre ad altre classi è rappresentato attraverso un'interfaccia; per modellare un'interfaccia si utilizza lo stesso modo utilizzato per modellare una classe, con un rettangolo. La differenza consiste nel fatto che un'interfaccia non ha attributi ma soltanto metodi. La simbologia utilizzata in UML per rappresentare il legame tra interfacce e classi è un piccolo cerchio che si unisce tramite una linea alle classi che implementano l'interfaccia. Per distinguere le interfacce dalle classi, nel formalismo UML, si utilizza la scrittura *interface* all'inizio del nome di ogni interfaccia.

Le relazioni

Ogni modello è composto da diversi elementi (classi ed oggetti) in grado di svolgere ruoli specifici. Se però non si indicano le relazioni che uniscono i vari elementi, un modello di classi sarebbe soltanto un insieme di rettangoli in grado di rappresentare solo il "vocabolario" del sistema. Specificando le relazioni che uniscono le classi presenti nel sistema, si mostra come i termini del vocabolario si connettono tra di loro.

Da un punto di vista concettuale, quando due classi sono connesse l'una con l'altra si determina un'*associazione* che, graficamente, è rappresentata con una linea che le connette; la linea riporta al suo fianco il nome dell'associazione creata. Quando una classe si associa con un'altra, ognuna di esse gioca un ruolo all'interno dell'associazione: è possibile mostrare questi ruoli sul diagramma, scrivendoli vicino alla linea orizzontale dalla parte della classe che svolge un determinato ruolo. Le associazioni non avvengono unicamente tra due classi: nella maggior parte dei modelli è possibile trovare più classi che si connettono ad una singola classe. Qualche volta un'associazione tra due classi deve seguire una regola, indicata inserendo una "constraint" vicino la linea che rappresenta l'associazione.

Esattamente come una classe, un'associazione può avere attributi ed operazioni. In questo caso si parla di una *classe associazione*; le classi associazione vengono visualizzate allo stesso modo con cui si mostra una classe normale utilizzando però una linea tratteggiata per connettere una classe associazione alla linea di associazione.

Esiste un tipo speciale di associazione detta *molteplicità*; grazie ad essa si mostra un gruppo di oggetti appartenenti ad una classe che interagisce con un altro gruppo di oggetti appartenente alla classe associata.

Una classe, in generale, può essere correlata ad una altra nei seguenti modi:

- Uno ad uno
- Uno a molti
- Uno ad uno o più
- Uno a zero o uno

- Uno ad un intervallo limitato (da 1 a 2 e 20)
- Uno ad un numero esatto n
- Uno ad un insieme di scelte (da 1 a 5 o 8)

Altre volte, una classe può trovarsi in associazione con se stessa, determinando un'associazione detta *riflessiva*.

In UML l'*ereditarietà* tra classi viene rappresentata con una freccia che connette la super-classe alla classe discendente; l'*ereditarietà* può essere quindi vista come un tipo di associazione.

In certi modelli una classe può rappresentare il risultato di un insieme di altre classi che la compongono: questo è un tipo speciale di relazione denominata *aggregazione*. Le classi che costituiscono i componenti e la classe finale sono in una relazione particolare del tipo: *parte - intero*. Un'aggregazione è rappresentata come una gerarchia in cui l'*intero* si trova in cima e i componenti (la *parte*) al di sotto. Una linea unisce l'*intero* ad un componente con un rombo raffigurato sulla linea stessa vicino all'*intero*. Un insieme di possibili componenti di un'aggregazione si identifica in una relazione di tipo OR; per modellare tale situazione è necessario utilizzare una constraint con la parola OR all'interno di parentesi graffe su una linea tratteggiata.

Una *composizione* è un tipo più forte di aggregazione. Ogni componente in una composizione può appartenere soltanto ad un *intero*. Il simbolo utilizzato per una composizione è lo stesso utilizzato per un'aggregazione eccetto il fatto che il rombo è colorato di nero.

La relazione tra una classe ed un'interfaccia viene definita *realizzazione*. La relazione è visualizzata nel modello da una linea tratteggiata con un triangolo largo aperto costruito sul lato dell'interfaccia.

2.3.3 I diagrammi

UML è una notazione che prevede più di tipologie di diagramma divise in categorie; concentrandoci sui diagrammi principali, potremmo effettuare la seguente suddivisione:

- **Diagrammi logici**

- diagramma dei casi d'uso (*use case*)
- diagramma delle classi (*class*)
- diagramma di sequenza (*sequence*)
- diagramma di collaborazione (*collaboration*)
- diagramma di stato (*statechart*)
- diagramma delle attività (*activity*)

- **Diagrammi fisici**

- diagramma dei componenti (*component*)
- diagramma di distribuzione (*deployment*)

Diagramma dei casi d'uso

I diagrammi dei casi d'uso vengono utilizzati per rappresentare le modalità di utilizzo del sistema da parte di uno o più utilizzatori detti attori; risultano particolarmente utili per specificare requisiti del sistema che si intende sviluppare senza rilevarne l'organizzazione interna.

Un caso d'uso è una classe che definisce un insieme di sequenze di azioni eseguite da un sistema che rivestono valore per un particolare attore. Un attore è, invece, un insieme di ruoli che gli utenti di un caso d'uso possono impersonare interagendo con il sistema; un singolo attore può eseguire più casi d'uso, e un caso d'uso può prevedere più attori. Un attore può essere un programma, o addirittura un intero sistema esterno interagente col sistema in analisi.

Si immagini di voler sviluppare un sistema per la gestione dei prestiti in una biblioteca. La figura 2.6 riporta un ipotetico diagramma dei casi d'uso in cui sono previsti due attori di tipo *studente* e *responsabile* che interagiscono col sistema secondo due modalità: lo studente può interagire con il solo caso d'uso *richiesta prestito*, mentre il responsabile può interagire anche con il caso d'uso *statistiche prestiti*.

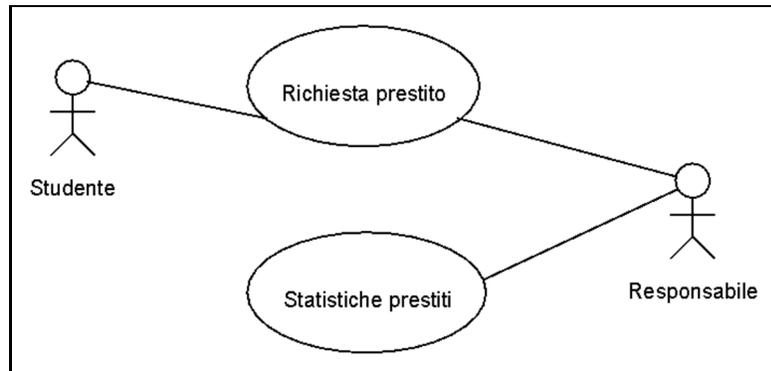


Figura 2.6. Esempio di diagramma caso d'uso

Diagramma delle classi

In un diagramma delle classi si rappresentano le entità significative del sistema e le relazioni che esistono fra le entità. Si rappresentano così le classi e gli oggetti (con i loro attributi e metodi) che compongono il sistema, specificando, mediante le associazioni, i vincoli che legano le classi tra loro. Una classe è rappresentata da un rettangolo scomposto in tre parti: il nome della classe, gli attributi della classe, le operazioni della classe. È importante capire che ci sono tre modi ben distinti di concepire e comprendere un diagramma delle classi:

prospettiva concettuale: il diagramma rappresenta concetti sviluppati in fase d'analisi; tali concetti potranno non trovare corrispondenza nel sistema finale;

prospettiva analitica: il diagramma rappresenta le interfacce di moduli da implementare in seguito; si dichiarano tipi piuttosto che classi, in quanto un tipo rappresenta un'interfaccia che può sopportare parecchie diverse implementazioni;

prospettiva implementativa: le classi rappresentano direttamente moduli software.

Potremmo considerare solamente tre entità per sviluppare il sistema di gestione prestiti; in figura 2.7 sono rappresentate le tre classi necessarie per implementare un modello basato sulle entità *studente*, *libro* e *prestito*. Per ogni classe sono visibili i metodi (ad esempio, *getMatricola()* restituisce il numero di matricola di uno studente) e gli attributi (*isDisponibile()* indica lo stato di disponibilità o meno di un libro).

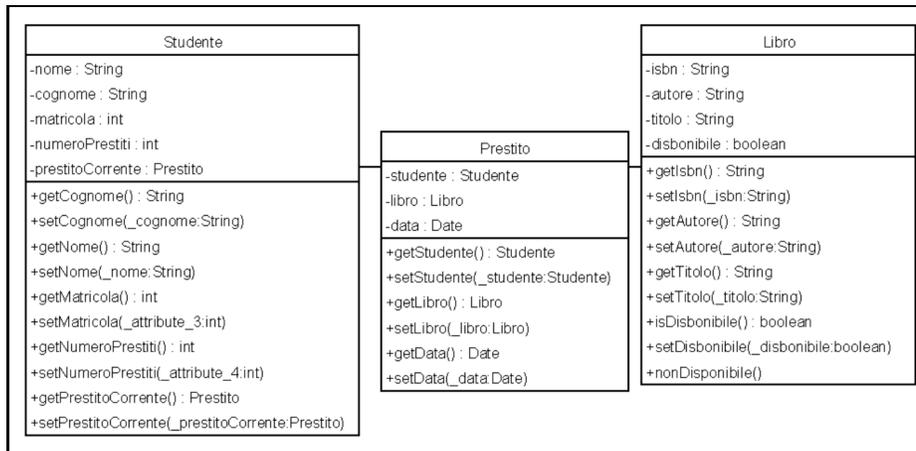


Figura 2.7. Esempio di diagramma delle classi

Diagramma di sequenza

Il diagramma di sequenza è un grafico bidimensionale in cui su di un asse si dispongono gli oggetti che interagiscono nel sistema mentre sul secondo asse è rappresentato il tempo. Si utilizza per evidenziare il modo in cui uno scenario (uno specifico caso d'uso) viene risolto dalla collaborazione tra un insieme di oggetti, specificando la sequenza dei messaggi che gli oggetti si scambiano tra loro.

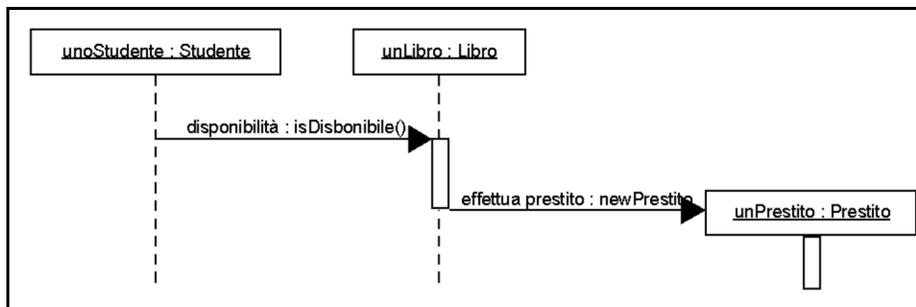


Figura 2.8. Esempio di diagramma di sequenza

Continuando l'esempio della biblioteca, con un grafico come quello riportato in figura 2.8 si può analizzare la sequenza degli eventi per portare a termine un prestito. Nel caso in esame è previsto prima il controllo sulla disponibilità di un libro da parte di uno studente attraverso il metodo *isDisponibile()*, se l'operazione è portata a termine con successo si utilizza il metodo *newPrestito()* per la creazione di un nuovo oggetto di tipo prestito.

Diagramma di collaborazione

Il diagramma di collaborazione specifica, come nel diagramma di sequenza, gli oggetti che collaborano tra di loro in un dato scenario, ma in questo caso si mettono più in evidenza i legami tra gli oggetti rispetto ai messaggi. Un diagramma di collaborazione evidenzia il contesto di un'interazione tra oggetti, il diagramma di sequenza illustra invece esplicitamente la successione temporale degli eventi.

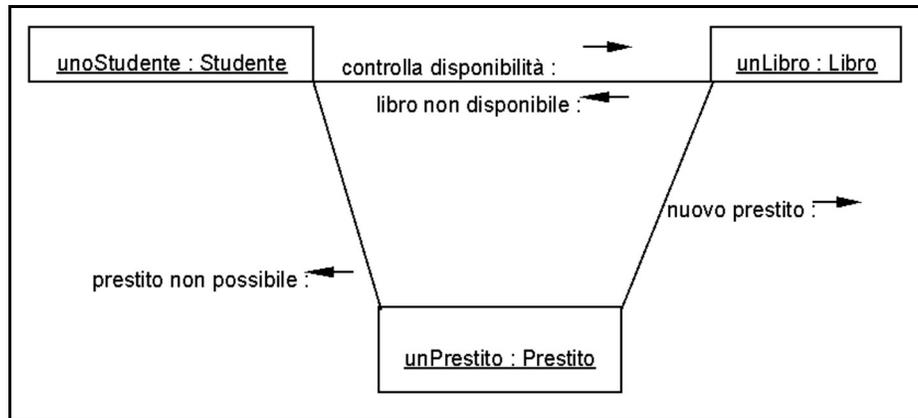


Figura 2.9. Esempio di diagramma di collaborazione

In figura 2.9 possiamo vedere i messaggi che vengono scambiati tra le diverse entità con la rispettiva direzione. Vediamo, ad esempio, che se il prestito non può essere effettuato (supponiamo che lo studente non sia abilitato al servizio prestiti) l'oggetto di tipo prestito invia all'oggetto di tipo studente il messaggio *prestito non possibile*.

Diagramma di stato

Un diagramma di stato è un grafo che specifica il ciclo di vita degli oggetti di una classe, definendo le regole che lo governano. Vengono rappresentati con dei rettangoli smussati gli stati, mentre con gli archi le transizioni di stato associate ad un evento; il grafico mostra così come cambia un oggetto durante il suo ciclo di vita all'interno del sistema.

Volendo analizzare come cambia lo stato di un oggetto di tipo libro all'interno del sistema, si potrebbe costruire un grafico come quello riportato in figura 2.10. Il cerchio nero rappresenta la creazione di un nuovo oggetto libro che potrebbe avvenire all'atto di

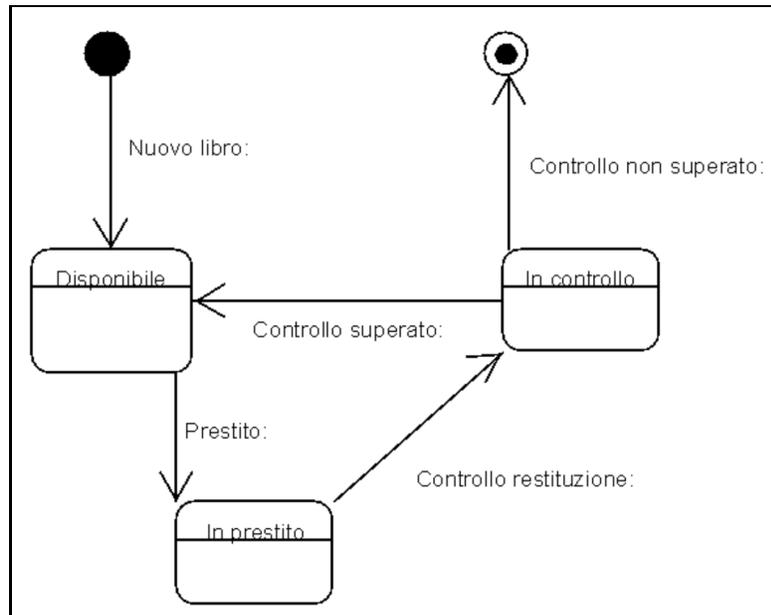


Figura 2.10. Esempio di diagramma di stato

acquisto da parte della biblioteca: a questo punto il suo stato rimane impostato a *disponibile* fino a quando un evento di tipo *prestito* non lo cambia in *in prestito*; al momento della restituzione il libro deve superare un controllo quindi lo stato cambia nuovamente per diventare *in controllo*; se il controllo viene superato il libro torna ad essere *disponibile*, altrimenti viene eliminato dal sistema. La distruzione di un oggetto viene rappresentata con un cerchio bianco e nero.

Diagramma di attività

I diagrammi di attività potrebbero essere considerati un'evoluzione dei diagrammi di flusso (flow chart) e sono tipicamente utilizzati per rappresentare dei sistemi di workflow⁸. Utilizzando questi grafi si possono rappresentare sistemi paralleli col fine di controllare la loro

⁸Un workflow è una descrizione formale e sintetica di un processo costituito da una serie di attività elementari cicliche o ripetitive da eseguire per ottenere un risultato.

Secondo gli standard del Workflow Management Coalition (WfMC) e del Workflow And Reengineering International Association (WARIA) viene definito (<http://www.e-workflow.org/>):

The automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant (human or machine) to another for action, according to a set of procedural rules.

sincronizzazione.

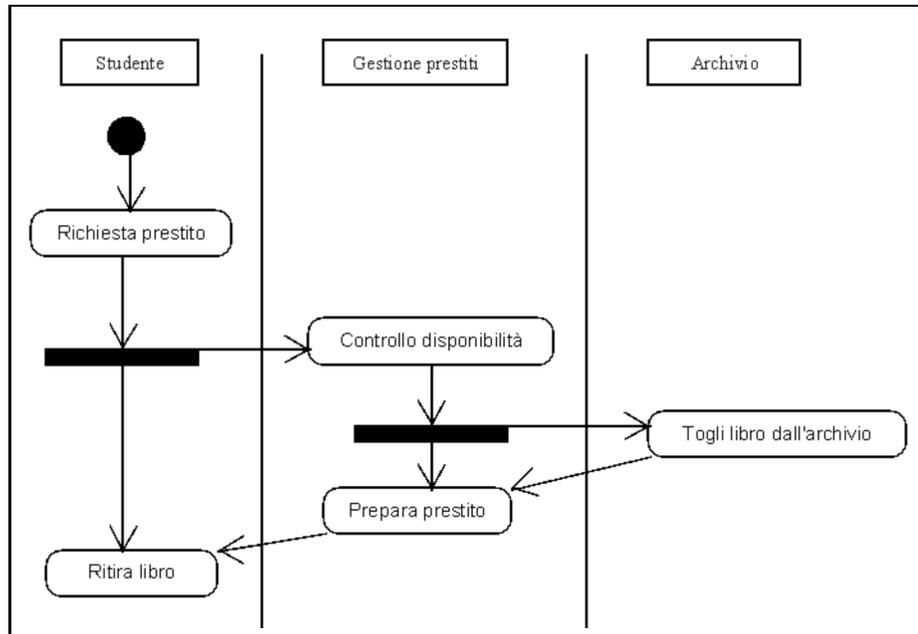


Figura 2.11. Esempio di diagramma di attività

In figura 2.11 possiamo osservare l'insieme operazioni che devono essere intraprese per portare a termine un prestito; le barre nere rappresentano il legame tra due azioni che devono essere svolte in parallelo. Notiamo così che la preparazione del prestito (pensiamo alla compilazione dei moduli) può essere svolta in parallelo alla rimozione del libro dall'archivio; solo nel momento in cui entrambe le attività saranno terminate lo studente potrà ritirare il libro.

Diagramma dei componenti

Il diagramma dei componenti mostra la struttura del codice in termini di componenti; i componenti sono moduli software dotati di identità e con un'interfaccia ben specificata. Solitamente i componenti sono raggruppati in entità di tipo *package*. In particolare è un grafo che ci permette di analizzare la struttura delle dipendenze tra codice sorgente, codice binario o eseguibile.

Supponendo di aver sviluppato il sistema di gestione dei prestiti con due software

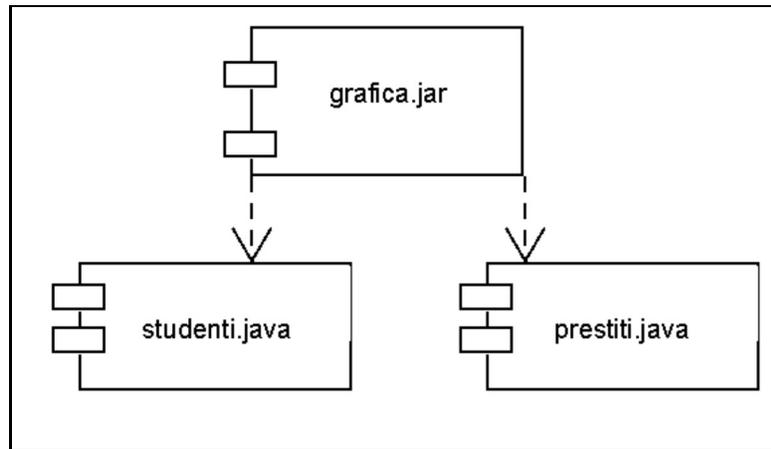


Figura 2.12. Esempio di diagramma dei componenti

scritti in linguaggio Java (*studenti.java* e *prestiti.java*) che utilizzano entrambi una libreria grafica per l'interfaccia utente (*grafica.jar*), potremmo rappresentare le dipendenze tra questi componenti con un grafo come quello in figura 2.12.

Diagramma di distribuzione

Con il diagramma di distribuzione si mostra la prevista allocazione degli elementi del sistema (processi, oggetti, ...) fra le "unità di elaborazione" (calcolatori) sulle quali il sistema dovrà funzionare.

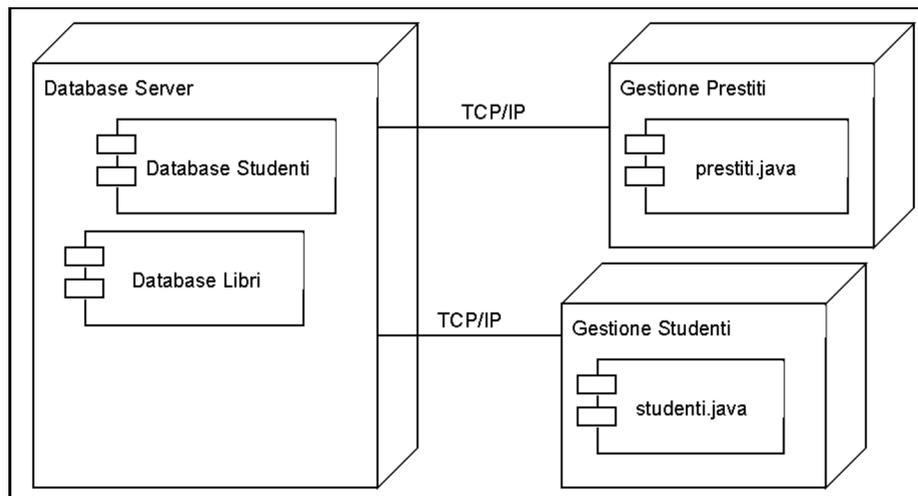


Figura 2.13. Esempio di diagramma di distribuzione

E' un grafo che mostra la struttura esecutiva (*run-time*) dei componenti, permettendo di rappresentare, a diversi livelli di dettaglio, l'architettura fisica del sistema evidenziando, con gli archi, i flussi di comunicazione tra componenti.

Il sistema di gestione dei prestiti potrebbe essere distribuito su tre calcolatori: un database server, sul quale sono presenti l'archivio degli studenti e l'archivio dei libri, e due macchine client, per l'esecuzione dei programmi *studenti.java* e *prestiti.java*. La comunicazione tra queste tre unità avviene attraverso il protocollo TCP/IP.

Bibliografia

- [2.1] Bissey, M., *Una piccola introduzione a swarm: Objectivec e java*, 2001.
- [2.2] Eckel, B., *Thinking in java* (Second ed.), Prentice-Hall, 2000.
- [2.3] Johnson P., Lancaster A., *Documentation set for swarm 2.1.1*, 2000.
- [2.4] Johnson P., Lancaster A., *Swarm user guide*, 2000.
- [2.5] Lin F. R., Tan G. W., Shaw M. J., *Multi-agent enterprise modeling*, Working Paper 96-0314, University of Illinois at Urbana-Campaign, 1996.
- [2.6] Meyers J., *A Short History of the Computer*, <http://www.softlord.com/comp/>, ultima visita Aprile 2002.
- [2.7] Minar N., Burkhart R., Langton C., Askenazi M., *The Swarm Symulation System: a Toolkit for Building Multi-Agent Simulation*, 1996.
- [2.8] Object Management Group, *Omg unified modeling language specification*, <http://www.omg.org>, ultima visita Maggio 2002.
- [2.9] Tarquini M., *Java mattone dopo mattone*, <http://www.java-net.tv>, ultima visita Aprile 2002.
- [2.10] Terna P., *Simulation tools for social scientists: Building agent based models with swarm*, Journal of Artificial Societies and Social Simulation, vol. 1 no. 2, 1998.

Capitolo 3

Gli studi sull'impresa

In campo microeconomico, la visione neoclassica ha occupato un ruolo preponderante nell'analisi dei fenomeni economici; le ragioni del suo successo sono riconducibili al metodo "perfettista" che questa visione prescrive a chi ne fa uso.

Seguendo Ricossa (1996) il metodo "perfettista"

presuppone tre idee: quella di perfezione, quella di diagnosi del male che separa dalla perfezione, e quella di rimedio definitivo dal male. [...] Ciò che cambia e resta identico a se stesso non è creduto perfetto, perché ha un'esistenza precaria, anzi, non ha una vera esistenza: comporta la nascita del nuovo, la discrepanza rispetto a un assoluto, che invece non può migliorare ancora e non deve peggiorare. Dunque il perfetto è immutabile, immobile, fuori del tempo, atemporale.

Una visione "perfettista" fornisce così, agli attori di politica economica, una giustificazione agli interventi delle autorità su persone o imprese (rimedio al male). I fallimenti di mercato (diagnosi del male), ad esempio, allontanano il sistema economico da una situazione di concorrenza perfetta (l'ideale).

Concentrandoci sulla teoria dell'impresa, per i neoclassici essa è come una "scatola nera", le cui azioni dipendono dall'interazione tra il mercato dei fattori, la funzione di

produzione e il mercato dei prodotti. Manca quindi in questo quadro la figura dell'imprenditore. Sempre in Ricossa (1996):

L'imprenditore del perfettismo non è l'innovatore avventuroso, da allettare con alti profitti (ancorché incerti); è piuttosto l'esperto, il dirigente burocrate, cui tocca nulla più di un profitto-salario prestabilito (e certo).

3.1 La visione neoclassica

Economia neoclassica è la denominazione corrente della teoria economica marginalista e dei suoi sviluppi contemporanei, specialmente a partire dagli anni trenta. Il metodo di analisi marginalista consiste nell'individuare le scelte ottime dei soggetti economici mediante il confronto tra il beneficio e il costo marginale ottenuti modificando una data scelta: solo se queste due grandezze sono uguali la scelta è da ritenersi ottima.

A partire dagli anni cinquanta la teoria neoclassica tende ad indentificarsi con la teoria dell'equilibrio economico generale. L'equilibrio economico è una situazione caratterizzata dal fatto che le varie forze agenti in dato sistema sono tali che nessun operatore economico è sollecitato a modificare le sue scelte; sebbene questa nozione economica sia già presente negli autori "classici" come Smith e Ricardo, essa trova una prima elegante formalizzazione nell'opera dell'economista francese Walras.

Walras ha espresso, in linguaggio matematico, le relazioni fondamentali che esistono, in solo periodo, tra le principali grandezze rappresentative di un determinato sistema economico isolato. Gli operatori economici considerati sono i *consumatori* e le *imprese*, le variabili economiche sono invece le quantità di *beni* utilizzate da ogni consumatore e quelle impiegate, o prodotte, da ogni impresa durante il periodo di tempo esaminato.

Ogni consumatore è descritto da una *funzione di utilità* e da una certa quantità di risorse possedute; dalla vendita di queste risorse il consumatore ottiene il suo *reddito* da destinare, seguendo la funzione di utilità, all'acquisto di beni di consumo. Questa operazione porta al formarsi delle *funzioni di domanda* che dipendono dai *prezzi relativi* dei beni presenti nel sistema.

Ogni impresa può svolgere un certo numero di processi produttivi descritti da una *funzione di produzione*. Un processo produttivo è rappresentato dai *fattori produttivi* impiegati e dai *prodotti ottenuti*. Quando l'impresa opera in una situazione di concorrenza perfetta la decisione di realizzare uno tra i vari processi produttivi è presa in modo da rendere massimo il suo *profitto totale*. Questa operazione porta al formarsi delle *funzioni di offerta* che dipendono anch'esse dai prezzi relativi.

Si può così descrivere lo schema fondamentale della circolazione delle merci e dei servizi: all'inizio del periodo di tempo considerato ogni consumatore possiede certe quantità di fattori produttivi che vende alle imprese, le imprese trasformano i fattori produttivi in prodotti, i prodotti sono successivamente venduti ai consumatori. Date le funzioni di domanda ed offerta, abbiamo una *configurazione di equilibrio* quando i prezzi sono tali che la quantità domandata di ciascun bene è uguale alla corrispondente quantità totale offerta.

L'impostazione neoclassica si presenta, sotto il profilo concettuale, in termini molto semplici, anche se "complicati" da un punto di vista formale (che in questa sede abbiamo trascurato); volendo imitare i metodi delle scienze naturali, gli autori neoclassici propongono una concezione meccanicistica dell'economia, che culmina nella nozione statica dell'equilibrio concorrenziale.

La teoria dell'impresa neoclassica può essere definita come un *derivato*, inquanto è parte integrante della teoria dell'equilibrio generale. L'ipotesi di fondo è che l'impresa sia assimilabile a una sorta di *robot*, che trasforma fattori produttivi in beni seguendo una regola a tutti conosciuta: la funzione di produzione.

Un funzione di produzione è una rappresentazione, tipicamente matematica, di un processo produttivo che sottolinea il rapporto diretto tra input (fattori produttivi) e output (beni prodotti).

Il processo produttivo associato ad ogni bene è visto come una macchina a sé stante, in grado di modificare le proprie modalità operative a fronte di variazioni del prezzo del bene o dei fattori, ma che comunque lavora secondo meccanismi del tutto noti e prevedibili (infatti è descritto per mezzo di un formalismo matematico). L'impresa neoclassica è tipicamente di piccole dimensioni, produce pochi beni omogenei e non in misura da influenzare il mercato.

La funzione imprenditoriale, quindi, si esaurisce quasi esclusivamente nella scelta fra i diversi processi produttivi e sulla scelta delle quantità di bene da produrre. In Walras (1874), infatti, leggiamo:

[Gli imprenditori] allo stato d'equilibrio della produzione, [...] non fanno né guadagno né perdita. Essi quindi sussistono non come imprenditori, ma come proprietari fondiari, lavoratori o capitalisti nelle proprie imprese o in altre.

Sebbene inquadrata in un contesto "perfetto", come quello descritto da Walras, la teoria dell'impresa neoclassica non ignora comunque un aspetto, difficilmente trascurabile, che caratterizza tutti i fenomeni legati all'uomo: l'incertezza. Per gli autori neoclassici, però, l'incertezza non riguarda il funzionamento o l'efficienza della macchina che trasforma gli input in output, né l'attività dell'imprenditore poiché non è visto come un innovatore o un operatore che si ingegna a scoprire nuovi bisogni da soddisfare.

L'imprenditore neoclassico convive con una duplice fonte di rischio: la prima è legata ad una carenza di informazioni sulle variabili esterne all'attività d'impresa, la seconda, interna, è legata alla selezione dei fattori produttivi.

In questo contesto l'incertezza presente nell'attività d'impresa potrebbe essere neutralizzata; l'imprenditore neoclassico, infatti, ha la possibilità di conoscere le distribuzioni di probabilità legate agli eventi incerti che lo interessano. In prospettiva marginalista, è così ragionevole supporre che l'acquisizione di informazione da parte dell'imprenditore abbia termine quando il costo di acquisizione diventa superiore al beneficio che una sua decisione meglio documentata consente.

In regime di concorrenza, rimanendo in un contesto di equilibrio generale, le imprese non registrano profitti, ma nel breve periodo, date le incertezze descritte sopra, queste possono presentare profitti positivi o negativi. Nel lungo periodo, però, le imprese meno efficienti (con profitti negativi) potranno riprodurre (copiando) le modalità di produzione dei concorrenti migliori (con profitti positivi). L'equilibrio concorrenziale neoclassico non prevede, quindi, dei profitti, in quanto essi non rappresenterebbero la remunerazione di alcun fattore produttivo (sarebbero il frutto dell'alea).

Si conclude che l'imprenditore è visto come una sorta di dispositivo meccanico, la

cui remunerazione è assimilabile a quello di un dispositivo capace di fare *calcoli*: a lui spetta una remunerazione fattoriale, non un profitto. Non essendo concepita una capacità imprenditoriale, in un modo neoclassico, tutti potrebbero essere imprenditori, dal momento che chiunque può disporre degli elementi necessari per operare secondo quanto prescritto dalle funzioni di produzione.

3.2 Schumpeter e la distruzione creatrice

L'impostazione neoclassica non è in grado di rispondere ad una domanda alla quale Schumpeter ha cercato di dare risposta: come si introduce il nuovo nell'economia?

Nella teoria dell'equilibrio economico generale di Walras i fattori esterni (come le istituzioni, ma anche i fattori demografici, ...) e quelli interni (come la tecnologia, le preferenze dei consumatori, ...) sono considerati come variabili esogene, alle quali il sistema economico si adatta per riprodursi in modo immutato.

A questa visione "statica", Schumpeter contrappone una visione "dinamica", basata sull'idea che lo sviluppo economico dipenda soprattutto dalle innovazioni tecnologiche introdotte dagli imprenditori; in particolare, l'introduzione delle innovazioni è resa possibile dalla creazione di credito da parte delle banche ed è stimolata dalle aspettative di un profitto monopolistico.

E' quindi grazie a Schumpeter che si mette in evidenza la figura dell'imprenditore, caratterizzato dalla sua capacità di iniziativa e di innovazione dei processi produttivi; la funzione dell'imprenditore sta nel rompere la staticità, la routine del meccanismo economico attraverso l'introduzione delle "nuove combinazioni". Schumpeter, nella sua interpretazione iniziale, indica la figura dell'imprenditore come ruolo chiave per il progresso tecnologico.

E' proprio grazie alle innovazioni tecniche che i vari imprenditori cercano di competere e di affermarsi sul mercato; per battere la concorrenza, infatti, l'imprenditore compie una continua ricerca sulle tecniche di produzione le quali, unite a quelle dell'organizzazione del lavoro e dei servizi di vendita, permettono di incrementare i profitti.

Le dinamiche delle innovazioni possono essere, secondo Schumpeter, *distruittivo-creatrici*:

distruttive perché tendono alla sostituzione di nuove produzioni alle vecchie, creatrici perché tale processo permette l'espansione della produzione. Le nuove combinazioni danno luogo alla nascita di nuove imprese che entrano in competizione con quelle già esistenti e attraverso questo scontro (causato dalla distruzione creatrice) le vecchie imprese, che adottano ancora le vecchie soluzioni e sono avverse al cambiamento, sono destinate a soccombere.

Da innovazioni nascono nuove innovazioni, ecco perché Schumpeter parla di sviluppo a onde successive, ondate che si propagano da una o da pochi imprenditori ad altri, provocando così una concatenazione di innovazioni successive.

L'antagonismo costituisce così l'essenza del *movimento ciclico*. Questo, secondo Schumpeter, spiega perché lo sviluppo economico proceda per vigorose espansioni, seguite da periodi di recessione: le recessioni sono viste come un rientro dell'economia nell'equilibrio del flusso circolare (quello descritto da Walras) dal quale il sistema è stato strappato a seguito delle innovazioni imprenditoriali.

3.3 L'impresa di Coase

Coase (1937) pone un'altra domanda alla quale le teorie neoclassiche non potevano dare una risposta: perché in un sistema economico nascono le imprese?

Secondo lo schema dell'equilibrio economico generale, ciò che regola il sistema è, in ultima analisi, il meccanismo dei prezzi. I prezzi relativi hanno influenza sia sulla funzione di domanda sia su quella d'offerta. Dal lato della domanda troviamo i consumatori, singoli individui; dal lato dell'offerta troviamo le imprese, delle organizzazioni. Su queste considerazioni Coase si chiede:

Alla luce del fatto che gli economisti, mentre considerano il meccanismo dei prezzi come uno strumento di coordinamento, ammettono anche la funzione coordinatrice dell'imprenditore, è certamente importante chiedersi perché il coordinamento sia svolto dal meccanismo dei prezzi in un caso e dall'imprenditore nell'altro. Il proposito di questo saggio è collegare ciò che sembra un salto nella

teoria economica tra l'ipotesi (introdotta per certi motivi) che le risorse siano allocate per mezzo del meccanismo dei prezzi e l'ipotesi (introdotta per altri motivi) che questa allocazione dipenda dall'imprenditore coordinatore. Si deve spiegare insomma il fondamento su cui, in pratica, viene effettuata una scelta tra due alternative.

Secondo Coase nascono le imprese quando si registra un *costo d'uso* del sistema dei prezzi; ad esempio quando si intende effettuare un acquisto, uno dei principali costi è la rilevazione dei prezzi dei beni o dei servizi presenti sul mercato. Anche stipulare singoli contratti per ogni scambio ha i suoi costi, legati ad esempio all'incertezza su certi fenomeni. Per questo motivo nascono le imprese, poiché

creando un'organizzazione e permettendo a una certa autorità (un "imprenditore") di allocare le risorse, vengono risparmiati i costi del mercato. L'imprenditore deve svolgere la sua funzione a un costo più basso di quello che nasce dal ricorso al mercato, perché qualora egli non possa ottenere i fattori di produzione a un prezzo minore rispetto alle transazioni di mercato, è sempre possibile tornare a farvi ricorso.

L'impresa nasce, quindi, come una modalità di governo delle transazioni, che utilizza la gerarchia interna per il coordinamento degli scambi, contrapponendosi al mercato che utilizza invece il meccanismo dei prezzi.

Il costo d'uso del meccanismo dei prezzi è legato ai costi di transazione. La transazione è il trasferimento di un bene o servizio, che comporta uno scambio di valori tra le parti; le transazioni sono fonti di costo in quanto generano frizioni decisionali. L'esistenza di questi costi di contrattazione spinge gli agenti economici ai comportamenti opportunistici tipici delle economie di mercato.

Nella vita di un'azienda, i costi di transazione sono quei costi che essa deve affrontare se decide di delegare alcune funzioni a ditte esterne, in aggiunta ai veri e propri costi di produzione. Costi di transazione tipici sono il tempo (e il costo) che comporta il fatto di mettersi alla ricerca dei fornitori e dei prodotti giusti, il tempo (e il costo) per imparare a

conoscere le caratteristiche dei prodotti che vengono offerti, o anche il tempo (e il costo) di raggiungere accordi il più possibile favorevoli all'azienda. L'azienda preferirà svolgere alcune attività al suo interno, e non rivolgersi al mercato, quando i costi di coordinamento di tali attività all'interno sono inferiori ai costi di transazione, cioè di negoziazione, redazione dei contratti e attuazione degli stessi, e quando la carenza di canali di informazione necessari per la gestione degli scambi fanno preferire l'organizzazione interna.

Seguendo Coase, l'impresa nasce quando, per gestire certe transazioni, il sistema dei prezzi fallisce. Ma questa impostazione fornisce anche una giustificazione alla dimensione delle imprese (che il modello neoclassico non poteva dare):

Un'impresa diventa più grande quando ulteriori transazioni (che potrebbero essere coordinate dal meccanismo dei prezzi) sono organizzate dall'imprenditore; ed essa diventa più piccola quando l'imprenditore cessa di organizzare queste transazioni.

I confini dell'impresa non sono più netti, ma sono definiti dalle transazioni gestite internamente; l'assetto dell'impresa, e di conseguenza la struttura industriale, dipendono dalle modalità di gestione delle transazioni che vengono scelte.

3.4 L'imprenditore di Kirzner

Secondo la prospettiva neoclassica, la caratteristica principale del mercato è la sua capacità di portare un sistema economico, in modo automatico, in una condizione di ottimo dove tutte le imprese registrano profitti nulli. Solo in questo modo, sostengono gli autori, si giunge al migliore utilizzo delle risorse disponibili e ad un consumo efficiente dei beni prodotti.

Per gli autori della scuola austriaca, invece, lo studio dell'economia non consiste nell'esame delle condizioni di ottimo, ma nell'analisi delle modalità attraverso le quali gli individui possono rendere meno gravosi i vincoli di scarsità delle risorse e migliorare la propria condizione in un futuro più o meno prossimo.

Kirzner identifica nell'imprenditore il vero protagonista di un'economia di mercato di

questo tipo, dove i vincoli di scarsità promuovono gli stimoli necessari al progresso del sistema. Per l'autore, quindi, l'imprenditore ha un ruolo ben preciso: non più l'esecutore di uno schema preordinato, come la funzione di produzione dei neoclassici, ma è lo *scopritore* per eccellenza.

Se Coase ci offre una spiegazione sul perché nasce un'impresa, Kirzner vuole spiegare la natura dell'atto imprenditoriale; l'atto imprenditoriale non è tanto inteso come l'invenzione di ciò che prima non esisteva o che fino a quel momento non era noto, consiste nel "vedere" realtà già presenti nel sistema, ma non ancora percepite da altri imprenditori come opportunità di crescita e di benessere. L'imprenditore di Kirzner, è' colui che coglie le opportunità che altri avevano trascurato e rende possibile la definizione di nuove produzioni.

Grazie a questa prospettiva Kirzner riesce anche a dare una giustificazione a ciò che i neoclassici vedevano come un "male": il profitto.

Il profitto, infatti, non può essere inteso come remunerazione delle capacità esecutive e organizzative dell'imprenditore, perché questa remunerazione spetterebbe al manager dell'impresa. Non può neppure remunerare il rischio di una attività produttiva, perché altrimenti andrebbe corrisposto ai detentori dei fattori produttivi (al capitalista). Tanto meno il lavoro, i cui detentori ricevono salari più o meno elevati in proporzione al loro livello di specializzazione, infatti, in condizioni di incertezza, potrebbero trovarsi disoccupati per periodi più o meno prolungati.

Secondo Kirzner, il profitto remunera l'abilità dell'imprenditore nel percepire gli errori altrui e dunque le opportunità di miglioramento, ancor prima che il manager organizzi le risorse per trasformare l'occasione imprenditoriale in realtà d'azienda. Il profitto si materializza solo quando la scoperta viene concretizzata, ma questo remunera un atto che è avvenuto prima: la "scoperta imprenditoriale". E' lo stimolo di una qualche "scoperta" all'interno di un mercato imperfetto, caratterizzato da squilibri e di informazioni scarse e costose che motiva (e quindi spiega) l'attività imprenditoriale.

Secondo gli autori della scuola austriaca, un'impresa che registra profitti positivi non deve più essere vista in senso negativo come dai neoclassici: il profitto è indice di vitalità

imprenditoriale e quindi di crescita e miglioramento per la collettività.

Un'ultima considerazione: la concezione dell'imprenditore di Kirzner si discosta parzialmente da quella della scuola austriaca in quanto vede l'individuo come fulcro non intenzionale dell'attività economica nel suo insieme.

La casualità delle scoperte attraverso le quali l'imprenditore individua imperfezioni e opportunità di cui non immaginava l'esistenza è il nodo cruciale della questione. Kirzner avanza la teoria dell'ignoranza inconsapevole, mentre l'ortodossia, anche Austriaca, sostiene la tesi dell'ignoranza razionale.

Se si nega alla funzione imprenditoriale la casualità del processo di scoperta si rischia di considerare l'imprenditore come un fattore produttivo anomalo: la ricerca sistematica richiede tempo e quindi l'impiego di una risorsa. Al contrario, accettare l'attributo della casualità significa ammettere che l'imprenditore puro è destinato a non essere remunerato e che la scoperta imprenditoriale non solo è casuale, ma sporadica.

Bibliografia

- [3.1] Blanchard O, *Macroeconomia*, a cura di Giavazzi F., Il mulino, 1997.
- [3.2] Coase R, *The nature of the firm*, *Economica*, vol. 4, 1937.
- [3.3] Colombatto E., *Dall'impresa dei neoclassici all'imprenditore di Kirzner*, *Economia Politica*, a. XVIII n. 2, 2001.
- [3.4] Kirzner I., *Entrepreneurial discovery and the competitive market: an austrian approach*, *Journal of Economic Literature*, vol. XXXV n. 1, 1997.
- [3.5] Ricossa S., *La fine dell'economia*, Sugarco Edizioni, 2001.
- [3.6] Schumpeter J., *L'imprenditore e la storia dell'impresa: scritti 1927-1949*, a cura di Salano A., Universale Bollati Boringhieri, 1993.
- [3.7] Varian H. R., *Microeconomia*, Cafoscarina, 1993.
- [3.8] Walras L. , *Elementi di economia politica pura*, UTET, 1874.

Parte II

jES e BasicJes

Capitolo 4

jES - Java Enterprise Simulator

4.1 Una duplice descrizione dell'impresa

Il progetto jES¹ (Java Enterprise Simulator) è nato nel corso dell'anno 2000 all'interno del Dipartimento di Scienze economiche e finanziarie G.Prato dell'Università degli Studi di Torino; leggiamo in Terna (2002):

Con il modello introdotto qui di seguito si intende far funzionare l'azienda simulata, non rappresentarla in modo animato sulla base di sequenze predeterminate di eventi; nel nostro modello gli eventi accadono in modo indipendente, generando interazione anche imprevedibili tra atti produttivi e unità produttive, proprie della complessità. Certo su un ideale asse che vada dall'astrazione dei cosiddetti vetri di spin come strumento per studiare la complessità allo sviluppo di un videogioco sofisticato con accadimenti non definiti a priori, soprattutto nelle loro reciproche relazioni, ci collochiamo in prossimità della seconda prospettiva.

4.1.1 Il modello

jES è un modello d'impresa realizzato attraverso un programma che gira nel computer.

Il programma è scritto in linguaggio *javaSwarm*, ma lo stesso modello può essere replicato anche con altri linguaggi di programmazione. E' per questo motivo che, in questo

¹Il nome originale del progetto era jVE (Java Virtual Enterprise)

capitolo, sarà presentato soltanto il modello che è alla base del progetto non occupandoci del codice informatico che permette il funzionamento di jES².

Il modello jES può essere utilizzato per due scopi:

- per ri-costruire un'impresa esistente, simulando le azioni che deve compiere al fine di ottenere i risultati desiderati;
- per costruire imprese virtuali o ipotetiche;

Il primo è un aspetto più pratico del modello: permette, in una prima fase, di mettere alla prova la conoscenza e la comprensione che l'utente ha dell'impresa oggetto di simulazione. Se, infatti, durante la fase di *ri-costruzione* all'interno del computer non sono stati inseriti degli aspetti caratteristici dell'impresa i dati della simulazione risulteranno in qualche modo differenti rispetto alla realtà, spingendo l'utente ad approfondire gli aspetti che caratterizzano l'impresa studiata.

Ri-costruita l'impresa sarà, successivamente, possibile ipotizzare scenari differenti o situazioni non ancora affrontate dall'impresa reale, utilizzando in questo modo il modello come un laboratorio sperimentale.

Il modello di jES è stato utilizzato e adattato per simulare la VIR s.p.a., un'azienda tradizionale che produce valvole idrauliche e, come vedremo nei capitoli successivi, per simulare la BasicNet s.p.a..

Il secondo aspetto del modello è più teorico: la costruzione di imprese ipotetiche, anche semplificate, ci permette di affrontare i temi come la creazione della conoscenza all'interno dell'impresa o l'innovazione imprenditoriale. In questo caso la simulazione permette di formulare teorie sulla natura dell'impresa in modo chiaro, esplicito e senza "buchi" poiché il significato dei suoi concetti è tradotto interamente nei termini operativi del programma che gira nel computer.

Il modello jES è fondato su una duplice descrizione della realtà che intendiamo ricostruire attraverso due formalismi diversi e ben distinti.

²Per maggiori informazioni sul funzionamento del programma scritto in *javaSWarm* è possibile consultare la documentazione scaricabile, insieme ai sorgenti di jES, all'indirizzo <http://web.econ.unito.it/terna/jes/>.

Con la prima descrizione si analizza il *Cosa fare* (WD, What to DO): il formalismo adottato permette la rappresentazione di tutti gli obiettivi che l'azienda intende perseguire o che si trova ad affrontare (prodotti, servizi, organizzazione interna, ...).

Con la seconda descrizione si analizza il *Chi fa che cosa* (DW, which is Doing What): il formalismo adottato permette in questo caso la rappresentazione delle risorse che l'impresa ha a disposizione (unità produttive, personale, magazzini, ...).

Quando la simulazione gira nel computer vengono unite le due descrizioni facendo, attraverso l'interazione degli oggetti descritti nel lato WD e nel lato DW, *emergere* la realtà dell'impresa.

All'interno del modello, e dei suoi formalismi, troviamo tre concetti che conviene definire subito:

Unità produttiva: è una struttura produttiva, o organizzativa, presente all'interno o all'esterno dell'impresa; è in grado di compiere uno o più *passi* (step) necessari per completare un ordine.

Ordine: è un oggetto che rappresenta un bene che deve essere prodotto (finito o semi-lavorato) o un'attività che deve essere intrapresa (un servizio, un atto organizzativo); la sua natura viene descritta attraverso una ricetta.

Ricetta: è una sequenza di *passi* che devono essere eseguiti in una determinata sequenza per ottenere un generico ordine.

In figura 4.1 sono rappresentati gli ordini che devono essere presi in carico da un'ipotetica impresa. L'informazione relativa ai passi da compiere è gestita in modo decentrato, considerando ogni ordine come un modulo che riporta la storia dei passi già espletati e la sequenza di quelli ancora da compiere per portare a termine l'obiettivo.

Ogni ordine richiede un determinato tempo (simulato) per essere portato a termine; la durata è espressa attraverso il formalismo delle ricette. Le unità produttive possono prendere in carico un solo ordine alla volta, se più ordini vengono assegnati alla stessa unità si crea una coda d'attesa. Tutte le unità produttive lavorano in modo autonomo ed in parallelo.

La simulazione simula, quindi, anche lo scorrere del tempo; deciso il livello di dettaglio che si intende utilizzare, l'utente deve associare la più piccola unità di tempo reale presa in considerazione (un secondo, un minuto, dieci minuti, un ora, ...) ad un *tick* della simulazione. Le fasi lavorative sono rappresentate come gruppi di *tick*; una volta deciso che un *tick* rappresenta un minuto (perché questo è il livello di dettaglio che intendiamo utilizzare) e se, ad esempio, si vogliono simulare turni di otto ore, un ciclo della simulazione sarà composto da 3800 *tick*.

L'impresa simulata riceve gli ordini da una "fonte"; la fonte può rappresentare il mercato per un'impresa che lavora *just in time* o può rappresentare un addetto alla programmazione della produzione per un'impresa classica. Il lancio degli ordini nella simulazione si può potenzialmente ottenere in più modi:

- generando casualmente (o con qualche algoritmo) delle ricette;
- lanciando in modo casuale (o con qualche algoritmo) delle ricette preparate dall'utente;
- lanciando secondo una sequenza le ricette preparate dall'utente;
- ...

Ad ogni ciclo della simulazione l'impresa riceve degli ordini che vengono presi in consegna da un'unità interna che svolge il compito di *Front End* verso il mercato; come si vede in figura 4.1 tale unità si occupa esclusivamente di passare ogni ordine all'unità che è in grado di svolgere il primo passo della ricetta produttiva.

Ogni unità produttiva, quando prende in carico un ordine, lo accoda nella propria lista di ordini da eseguire secondo la modalità FIFO³ (First In First Out); dopo l'esecuzione richiede all'ordine (coincidente con il modulo che contiene la ricetta produttiva), l'informazione necessaria per trasmetterlo ad una successiva unità; praticamente, l'unità che ha in carico l'ordine, una volta compiuta la sua lavorazione, controlla il passo successivo della ricetta, individua l'unità che è in grado di svolgerlo e gli passa l'ordine.

³In informatica, con la sigla FIFO si indica la logica di trattamento di una lista di informazioni, attività o richieste organizzate in una coda, che devono essere prese in carico nell'ordine d'arrivo.

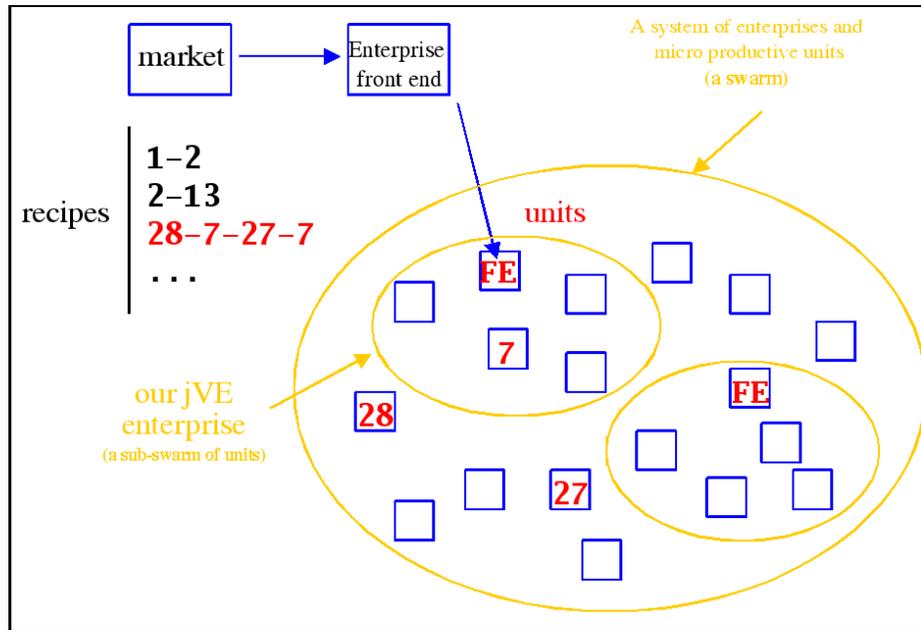


Figura 4.1. Schema generale del modello jES

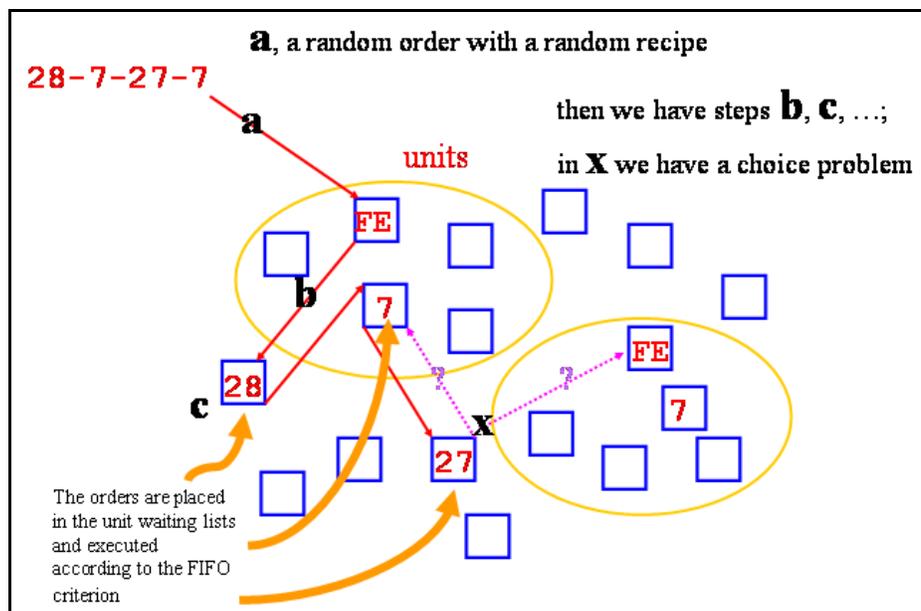


Figura 4.2. Le ricette

Esattamente come nella realtà aziendale potremmo rilevare problemi di assegnazione quando più unità produttive possono svolgere la stessa lavorazione. E' possibile, infatti, inserire nel modello anche unità produttive esterne all'impresa, simulando in questo

caso attività di fornitura o di outsourcing come riportato in figura 4.2.

Ogni volta che l'utente riuscirà a ri-costruire questi problemi di assegnazione, il modello permette di effettuare, con qualunque tecniche conosciuta in materia, ottimizzazioni di processo. Facendo nuovamente girare la simulazione con le modifiche apportate è possibile confrontare le diverse configurazioni produttive ipotizzate.

Vedremo ora nel dettaglio i due formalismi che ci permettono di descrivere cosa deve fare l'impresa (aspetto WD) e chi è in grado di farlo (aspetto DW).

4.1.2 Che cosa fare?

Il *che cosa fare* viene rappresentato con il formalismo delle ricette. Le ricette sono composte da una successione passi produttivi che rappresentano le lavorazioni. Ma le tipiche attività lavorative di un'impresa non sono, ovviamente, solo sequenziali; per questo motivo sono stati introdotti nel formalismo passi "speciali" che permettono la descrizione di ramificazioni delle ricette, atti di approvvigionamento o ancora lavorazioni per lotti.

Le ricette

Durante la descrizione WD si rappresentano le ricette attraverso un semplice formalismo numerico. Ogni passo della ricetta è espresso con una cifra col fine di rappresentare in modo univoco l'atto produttivo o organizzativo che deve essere compiuto.

Ad ogni passo produttivo si affianca il tempo necessario per il suo compimento espresso con una quantità ed una unità di tempo (giorni, minuti, secondi, ...).

7	s	20	8	m	1	...
---	---	----	---	---	---	-----

Tabella 4.1. Formalismo di una generica ricetta

In tabella 4.1 è rappresentata una generica ricetta per la descrizione di un ordine composto da due passi produttivi da compiersi nell'ordine riportato: vediamo così che il passo **7** richiede 20 secondi per essere completato ed il passo **8** un minuto.

Terminata la ricetta l'ordine è stato completato ed è pronto ad "uscire" dall'impresa (ad esempio per essere venduto). Un ordine può anche rappresentare un prodotto non

finito (un semi-lavorato) utile per portare a termine altri ordini. Un prodotto non finito deve essere depositato in una unità speciale detta *endUnit* come riportato in tabella 4.2.

3	s	2	2	s	10	e	100
---	---	---	---	---	----	---	-----

Tabella 4.2. Formalismo di una generica ricetta per un prodotto non finito

In questo esempio vediamo che l'ordine andrà depositato in una unità-magazzino con codice **100**.

I batch

Ci sono situazioni aziendali in cui non risulta realistico descrivere dei processi produttivi prendendo in considerazione singoli oggetti perché i tempi unitari risulterebbero troppo piccoli ed ininfluenti all'interno della simulazione.

Diventa naturale in questi casi ragionare per lotti di prodotti, come avviene, ad esempio, durante un'operazione di etichettatura di articoli o per una fornitura di bulloni.

Per questo motivo all'interno del modello jES sono state inserite due tipologie di passi definite *batch*.

Un *sequential batch* (espresso con il simbolo \) deve essere associato ad un passo produttivo, come riportato in tabella 4.3; in questo modo si indica che il passo **8** deve essere compiuto, da un'unica unità produttiva, contemporaneamente su 100 ordini, dopo la lavorazione *sequential batch* verranno nuovamente gestiti in modo separato i diversi singoli ordini.

7	s	20	8	m	1	\	100	...
---	---	----	---	---	---	---	-----	-----

Tabella 4.3. Formalismo di un passo *sequential batch*

L'esempio riportato potrebbe essere utilizzato per descrivere un'operazione di tipo *sequential batch* per simulare l'etichettatura di 100 prodotti, anche differenti, che avviene in un minuto.

Un *stand alone batch* (espresso con il simbolo /) può essere inserito soltanto in ricette

composte da un unico passo ed un unità di destinazione (*endUnit*) come riportato in tabella 4.4; questo formalismo è tipicamente utilizzato per rappresentare una fornitura di materie prime o semilavorati necessari per completare altri ordini.

Ipotizzando una fornitura di bulloni, compiuto il passo **7** di durata un giorno all'interno della *endUnit* **10** saranno presenti 2000 bulloni utili al completamento di ordini futuri (valvole, motori, ...).

7	<i>d</i>	1	/	2000		<i>e</i>	10
----------	----------	----------	---	------	--	----------	-----------

Tabella 4.4. Formalismo di un passo *stand alone batch*

Procurement

Affinché un componente possa essere recuperato da una *endUnit*, è previsto un passo di tipo *procurement* (espresso con il simbolo **p**); vediamo in tabella 4.5 che il passo **4** per essere portato a termine è necessaria la fornitura di due prodotti non finiti con codice **10** e **20**.

3	<i>s</i>	5		p	2	10	20	4	<i>s</i>	5		...
----------	----------	----------	--	----------	----------	-----------	-----------	----------	----------	----------	--	-----

Tabella 4.5. Formalismo di un passo *procurement*

I casi di fornitura di componenti crea in ogni azienda problemi di sincronizzazione tra l'inizio delle diverse produzioni e le relative componenti, sia che si tratti di gestione di reti produttive all'interno di una stessa azienda, sia di un sistema integrato di aziende. In questo ultimo caso l'unità che si occuperà della fornitura è considerata una scatola nera, il cui funzionamento interno non è indagato ulteriormente.

Grazie a questo formalismo per la creazione di sotto-ricette produttive è possibile, come riportato in figura 4.3, esplodere a piacere ogni processo aziendale in più sub-processi in funzione del livello di dettaglio che si desidera utilizzare.

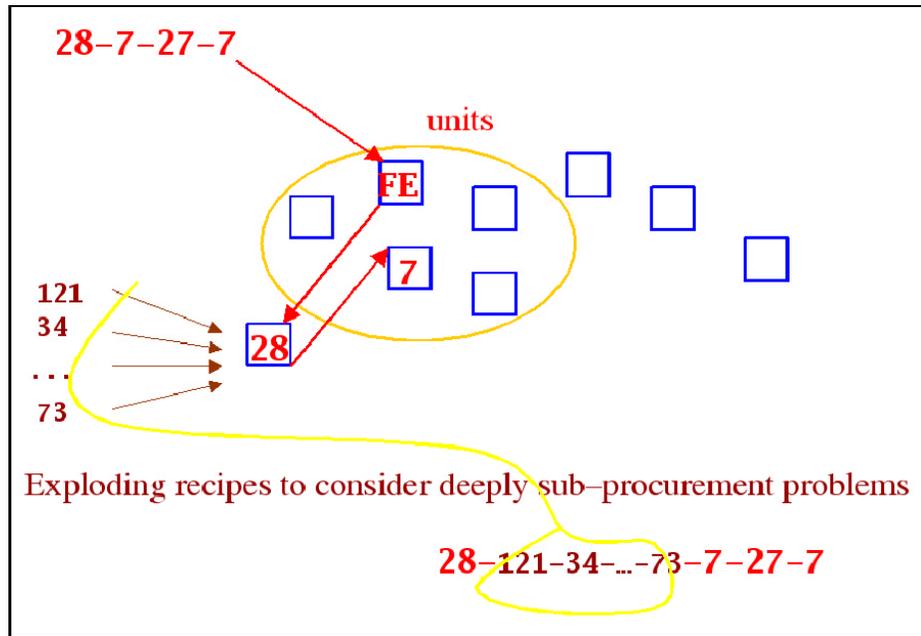


Figura 4.3. Passo di procurement

Or

E' tipico di molte aziende la possibilità di realizzare un prodotto seguendo "strade" alternative; il taglio di un materiale potrebbe, ad esempio, essere effettuato utilizzando una lama elettrica o un dispositivo laser. I due prodotti finiti risulterebbero uguali, ma i passi compiuti per realizzarli sono stati differenti.

Un passo di tipo *or* (espresso con il simbolo $||$) permette di ramificare la ricetta produttiva su due o più rami alternativi. Nell'esempio riportato in tabella 4.6 vediamo che l'ordine può essere completato in due modi alternativi: o eseguendo il passo 100 o il passo 200.

10	<i>m</i>	1	$ $	$ $	1	100	<i>s</i>	30	$ $	2	200	<i>s</i>	45	$ $	0	...
-----------	----------	---	------	------	---	------------	----------	----	------	---	------------	----------	----	------	---	-----

Tabella 4.6. Formalismo di un *or*

La scelta sul passo da compiere avviene in funzione di alcuni criteri già previsti nel modello; è possibile, infatti, impostare la simulazione in modo tale che:

- si eseguano tutti i rami;

- si esegua solo il primo ramo;
- si esegua il secondo ramo⁴;
- il ramo da eseguire sia scelto in modo casuale;
- si esegua il ramo che presenta il primo passo con la minore coda di produzione;
- il ramo sia scelto con l'ausilio di un oggetto informatico (oggetto computazionale, si veda a pagina 109);

E' su punti di questo tipo che si aprono le maggiori aree di indagine all'interno delle nostre simulazioni; potenzialmente l'utilizzatore, con l'ausilio degli (oggetti computazionali) può fin da ora introdurre sistemi di scelta fondati su reti neurali, sistemi classificatori, algoritmi genetici o facendo direttamente intervenire un umano.

4.1.3 Chi fa che cosa?

L'aspetto DW è legato alla descrizione delle risorse produttive disponibili all'interno dell'impresa, o presenti esternamente ma in qualche modo legate alla sua realtà.

Le unità

L'attività principale dell'unità produttiva consiste nell'inserire l'informazione di completamento del proprio compito e individuare a quale unità spetta il processo produttivo seguente: concettualmente si potrebbe dire che l'unità produttiva fa semplicemente passare il tempo (simulato) necessario al completamento del passo.

Le differenti unità si contraddistinguono in base ad una cifra; ad ogni unità produttiva sono associati uno o più passi che essa è in grado di svolgere.

Tutte le unità vengono descritte all'interno di un file: la descrizione comprende le seguenti informazioni:

Codice unità: si attribuisce un numero univoco ad ogni unità presente nella simulazione;

Passo associato: si indica quale passo produttivo l'unità è in grado di svolgere;

⁴I primi tre criteri sono utili in fase di test della simulazione

Utilizzo dei magazzini: si indica se l'unità presenta un magazzino ad essa associato (con un 1 o uno 0);

Costi fissi: si indicano i costi fissi che l'impresa deve registrare ad ogni ciclo della simulazione (supponiamo un giorno, mese, anno, ...)

Costi variabili: si indicano i costi variabili associati all'attività produttiva.

Il formalismo si presenta quindi simile a quello riportato in tabella 4.7.

Codice unità	Passo associato	Usa magazzino	Costi fissi	Costi variabili
1	100	1	10	1
2	200	0	15	2
...

Tabella 4.7. Formalismo delle unità

Le unità produttive descritte sono tutte quelle necessarie per giungere a completamento dei prodotti inseriti nella simulazione, indipendentemente dal fatto che esse siano interne o esterne all'impresa.

Unità complesse

E' possibile inserire nel modello unità complesse in grado di svolgere più passi produttivi. Ogni unità registra costi fissi e variabili differenti in funzione dell'attività che sta svolgendo, analogamente anche l'utilizzo del magazzino potrebbe risultare differente; si immagina a questo proposito il riattrezzaggio una una macchina per la verniciatura.

Per descrivere le unità complesse si associa al precedente formalismo una matrice, come quella riportata in tabella 4.8, contenente i singoli passi, i costi fissi, i costi variabili e l'utilizzo o meno del magazzino.

Passo associato	Costi fissi	Costi varibili	Usa magazzino
100	10	1	1
200	15	2	0
...

Tabella 4.8. Formalismo delle unità complesse

End unit

Come abbiamo visto in precedenza, i componenti, per poter essere soggetti a *procurement*, al termine delle loro lavorazioni vengono depositati in unità-magazzino dette *endUnit*.

Il formalismo per descrivere tali unità prevede di elencare in un file il loro codice univoco indicando, con un segno meno, se sono sensibili o meno ai layer (vedremo il concetto di *layer* a pagina 111). Un esempio è riportato in tabella 4.9.

Codice endUnit
1000
-2000
...

Tabella 4.9. Formalismo delle endUnit

4.1.4 Un semplice esempio

Qui di seguito si riporta un esempio di funzionamento del modello per un'ipotetica impresa simulata molto semplice; l'esempio aiuterà a comprendere come avviene all'interno di jES la gestione della produzione, ma soprattutto mostrerà quanto può essere utile lo strumento simulativo utilizzato come macchina per automatizzare gli esperimenti mentali.

L'impresa è composta da due unità produttive con codice **1**, **2** e **3** ed una *endUnit* con codice **10**. La descrizione di tipo DW è quella riportata in tabella 4.10.

Codice unità	Passo associato	Usa magazzino	Costi fissi	Costi variabili
1	1	0	10	1
2	2	0	10	1
3	3	0	10	1

Tabella 4.10. Descrizione unità dell'esempio

In tabella 4.11 sono riportate le due ricette che descrivono gli ordini da portare a termine. La prima ricetta descrive un prodotto semi-lavorato (codice **100**), la seconda un prodotto finito (codice **101**) che necessita l'approvvigionamento del semi-lavorato.

semi-lavorato	100	1	s	1	2	s	1	p	1	10	3	s	1
prodotto	101	1	s	1	e	10							

Tabella 4.11. Ricette dell'esempio

Ogni ciclo produttivo simulato è composto da un secondo (un *tick* equivale ad un secondo ed il ciclo simulativo è composto da un *tick*).

Sono lanciati 19 ordini durante 10 cicli di simulazione secondo la sequenza riportata in tabella 4.12; i pedici sono utilizzati per distinguere i diversi ordini.

Ciclo produttivo	Ordini lanciati
0	100 _a
1	101 _b , 100 _b
2	101 _c , 100 _c
3	101 _d , 100 _d
4	101 _e , 100 _e
5	101 _f , 100 _f
6	101 _g , 100 _g
7	101 _h , 100 _h
8	101 _i , 100 _i
9	101 _j , 100 _j

Tabella 4.12. Sequenza ordini dell'esempio

Con la tabella 4.13 si schematizza l'attività produttiva svolta all'interno del modello durante i 10 cicli simulati, indicando l'ordine preso in carico dalle singole unità e le relative code d'attesa.

4.2 Le funzionalità del simulatore

4.2.1 La gestione dei magazzini

Un esempio classico di organizzazione, anch'essa fonte della caratteristica complessità aziendale, è la creazione di scorte per il magazzino come vediamo rappresentato in figura 4.4.

Per funzionare, i magazzini hanno bisogno di regole, che indicano in quali circostanze

Ciclo	Unità 1		Unità 2		Unità 3		endUnit
	Coda	Produzione	Coda	Produzione	Coda	Produzione	Magazzino
0							
1	100a	100a					100a
2	101b 100b	101b					100a
3	100b 101c 100c	100b	101b	101b			100a 100b
4	101c 100c 101d 100d	101c			101b	101b[100a]	100b
5	100c 101d 100d 101e 100e	100c	101c	101c			100b 100c
6	101d 100d 101e 100e 101f 100f	101d			101c	101c[100b]	100c
7	100d 101e 100e 101f 100f 101g 100g	100d	101d	101d			100c 100d
8	101e 100e 101f 100f 101g 100g 101h 100h	101e			101d	101d[100c]	100d
9	100e 101f 100f 101g 100g 101h 100h 101i 100i	100e	101e	101e			100d 100e

Tabella 4.13. Produzione dell'esempio

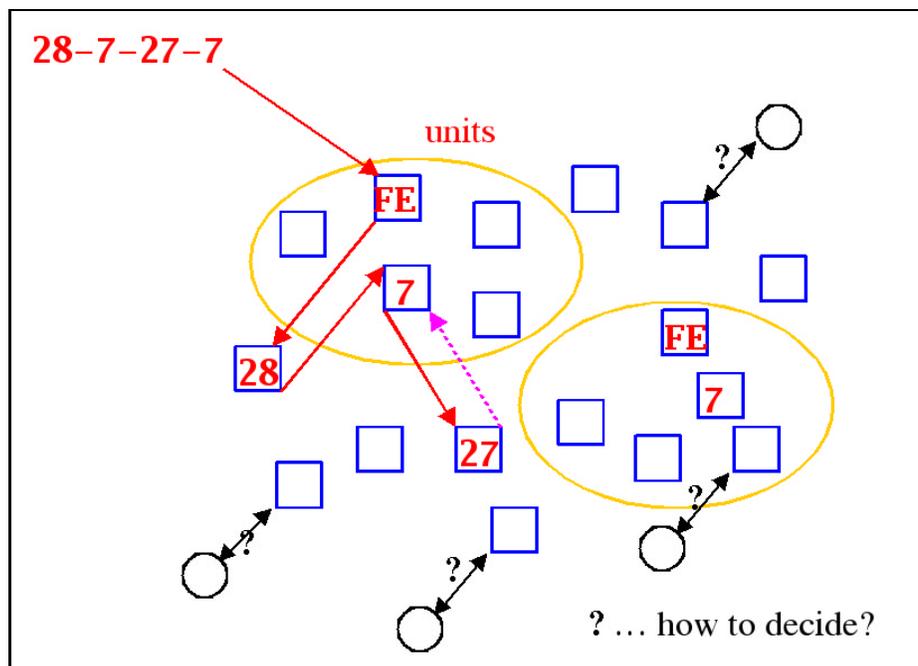


Figura 4.4. I magazzini

si devono riempire o svuotare; nel modello jES tali regole sono contenute all'interno di un oggetto informatico chiamato *RuleMaster*, in accordo con lo schema ERA già descritto.

Quando un'unità produttiva, associata ad un magazzino, non è coinvolta nella lavorazione di alcun ordine interroga il *RuleMaster* per sapere se deve produrre delle scorte ed in quale quantità.

Le regole sono scelte, attraverso la scrittura di un appropriato algoritmo informatico, dall'utente della simulazione; l'unità coinvolta potrebbe, ad esempio, controllare la ricetta associata all'ordine in arrivo e produrre per il magazzino il componente che sarà richiesto in uno dei prossimi cicli simulativi. L'unità a questo punto non dovrà produrre, sveltendo il processo e saltando di fatto un'attività produttiva o, comunque, ridurrà la coda d'attesa ad essa associata.

Il *RuleMaster* permette anche la gestione delle quantità di componenti che possono essere contenuti nei magazzini delle singole unità, simulando, in questo modo, la loro dimensione.

4.2.2 Le capacità computazionali

Il modello prevede l'inserimento di oggetti informatici in grado di eseguire algoritmi informatici specificati dall'utente. I risultati delle computazioni sono archiviate, durante lo scorrere del tempo (simulato), in matrici di memoria predisposte dal programma.

Questi oggetti con capacità computazioni, con le relative matrici di memoria, possono essere richiamati dalle ricette produttive utilizzando un formalismo come quello riportato in tabella 4.14.

6	<i>s</i>	12	c	1998	2	0	1	8	<i>m</i>	2	...
----------	----------	----	----------	------	---	---	---	----------	----------	---	-----

Tabella 4.14. Formalismo degli oggetti computazionali

Nell'esempio riportato al passo **8** è associata una computazione (simbolo **c**) con codice **1998** che richiede l'utilizzo delle due matrici **0** e **1**.

Come si può intuire, lo scopo degli oggetti computazionali è consentire all'impresa simulata di eseguire operazioni immateriali, fare previsioni, gestire aste, indirizzare procurement o archiviare informazioni in database. In questo modo è, potenzialmente, possibile anche simulare il sistema informativo dell'azienda; una simulazione utile per fare il confronto tra ciò che accade "veramente" e quello che l'azienda vede attraverso il suo sistema informatico.

Come già accennato a pagina 103 gli oggetti computazioni permettono la scelta del ramo

le comuni realtà aziendali in cui la distribuzione dell'informazione è limitata.

Nel modello jES le informazioni sono racchiuse all'interno di appositi oggetti informatici definiti *News*. Le unità produttive ad ogni ciclo decidono se inviare o meno informazioni ad altre unità produttive. La news risulta, quindi, una metafora del messaggio, dell'e-mail, della lettera, del dispaccio aziendale o della cartella consegnata ad un dato settore.

Attualmente, il programma jES permette di simulare una circolazione di informazioni molto semplificata caratterizzata da una serie di messaggi che le diverse unità, logicamente connesse, si scambiano per facilitare le decisioni relative alla produzione di scorte o alla destinazione di risorse scarse condivise tra più unità produttive.

In modo analogo alla gestione dei magazzini, le *News* richiedono regole e contenuti che potrebbero, in versioni successive del modello, essere gestite, seguendo lo schema ERA, tramite un *RuleMaster* ed un *RuleMaker*.

4.2.4 I layer

Le ricette produttive caratterizzano i diversi prodotti che l'impresa simulata è in grado di realizzare. Ogni ordine è associato ad una ricetta, ma ordini con ricette uguali potrebbero avere caratteristiche qualitative o quantitative differenti che, in qualche modo, devono essere rilevate durante la simulazione.

La ricetta produttiva per la realizzazione di penne a sfera è, ad esempio, sempre la stessa, ma potremmo avere la necessità di distinguere le penne nere da quelle blu, oppure quelle per il mercato asiatico da quelle per il mercato europeo.

I *layer* consentono di gestire in modo distinto prodotti uguali, ma riferiti a periodi di produzione o, più in generale, ad ambiti differenti; nel modello si è aggiunta la possibilità di associare ad ogni ordine un *layer*. L'assegnazione del *layer* avviene in fase di lancio degli ordini da parte di un oggetto informatico chiamato *orderDistiller*. Questa diversificazione degli ordini avrà o meno, in base alle decisioni dell'utente, affetto sulle operazioni interne alla simulazione, come *sequential batch*, *stand alone batch*, *procurement* e computazioni. E' possibile, infatti, in fase di descrizione del lato DW distinguere (con il segno -) le unità e le matrici che risultano insensibili ai layer da quelle sensibili.

4.2.5 La contabilità

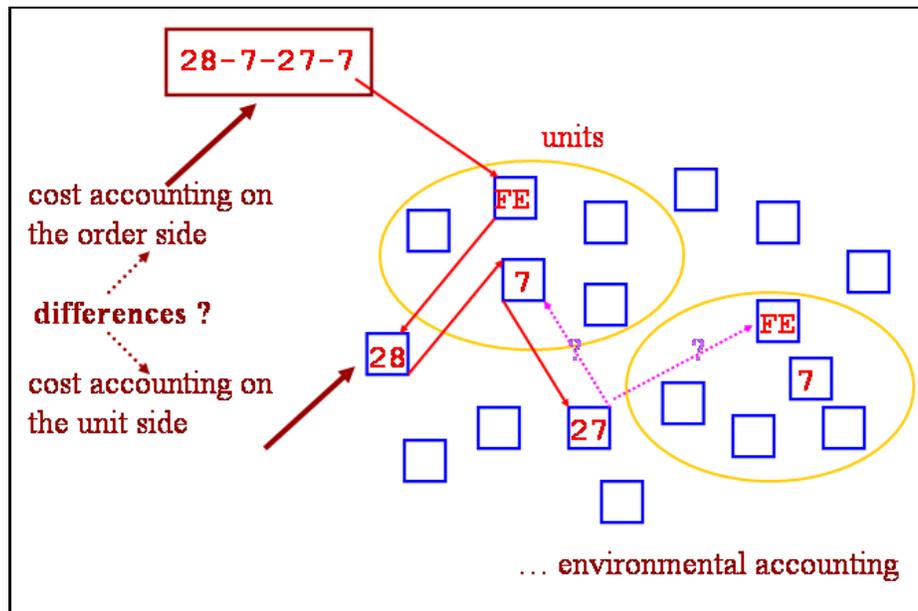


Figura 4.6. La contabilità nel modello jES

All'interno del modello la contabilità viene aggiornata in tempo reale durante il ciclo della simulazione.

Esistono due prospettive di contabilità:

- per unità produttive;
- per prodotto;

La contabilità per unità produttiva è caratterizzata dai costi fissi e variabili che, al termine di ogni ciclo (giorno, mese, anno, ...), l'impresa simulata registra. Nella seconda prospettiva, il prodotto in ogni fase di lavorazione viene caricato dei costi fissi e di quelli variabili delle unità che lo hanno preso in carico; se un ordine prevede, nella sua ricetta, un approvvigionamento (procurement), i costi accumulati nel prodotto approvvigionato vengono interamente scaricati sull'ordine finale. Le due contabilità, quindi, non sempre coincidono rilevando in questo modo i costi fissi delle unità non utilizzate in fase di produzione. Anche le unità che producono per i magazzini registrano costi fissi e variabili.

I prodotti in giacenza nel magazzino vengono ricaricati di un costo finanziario, pari ad un tasso d'interesse stabilito dall'utilizzatore della simulazione. In modo analogo vengono trattati i prodotti presenti nelle *endUnit*.

Bibliografia

- [4.1] Parisi D., *Simulazioni. La realtà ifatta nel computer*, Il mulino, 2001.
- [4.2] Pelligra P., *Un modello di impresa virtuale in javaswarm: la circolazione delle informazioni*, Tesi di laurea in Economia e Commercio, Torino, 2001.
- [4.3] Remondino M., *Analisi dei processi aziendali mediante un modello ad agenti in Java Swarm, con scorte*, Tesi di laurea in Economia e Commercio, Torino, 2000.
- [4.4] Terna, P., *Improved java virtual enterprise (jve) in swarm*, presentazione per la SwarmFest, <http://web.econ.unito.it/terna/jes/>, 2002.
- [4.5] Terna, P., *Simulazione ad agenti in contesti di impresa*, Sistemi intelligenti, XVI (1), 2002.

Capitolo 5

L'azienda da simulare: BasicNet

5.1 Breve storia dell'azienda

Qui di seguito vengono riportate le tappe che hanno portato all'attuale configurazione del Gruppo BasicNet. A nostro giudizio, tale analisi potrà essere utile per comprendere le motivazioni del *business system* ideato ed adottato dall'azienda che, in seguito, verrà analizzata. Tutte le informazioni sono liberamente tratte da *BasicPress*, il sito web di informazione stampa del gruppo.

Nel 1983 viene fondata la *Football Sport Merchandise S.r.l.* di cui è Presidente Marco Boglione, attuale Presidente di BasicNet. *Football Sport Merchandise S.r.l.* è la prima società italiana, e tra le prime in Europa ad ottenere, dalle principali squadre di calcio, la licenza per produrre e distribuire abbigliamento per il tempo libero contrassegnato dai loro marchi e segni distintivi.

Nel 1988 *Football Sport Merchandise S.r.l.*, a fronte di una rilevante crescita del mercato dei prodotti contrassegnati dai marchi delle squadre sportive nel mondo, costituisce la *Basic Merchandise S.r.l.* L'obiettivo di Basic Merchandise, oggi *Kappa Italia*, è quello di ottimizzare i processi produttivi e la distribuzione dei prodotti in alcuni mercati esteri.

Nel 1994 *Football Sport Merchandise*, tramite *Basic Merchandise* rileva dal curatore fallimentare i marchi, il magazzino e l'immobile di proprietà del *Maglificio Calzificio Torinese S.p.A.*, società fondata nel 1916 proprietaria dei marchi **Kappa**, **Robe di Kappa**

e **Jesus Jeans**. L'obiettivo dell'acquisizione è quello di disporre di marchi conosciuti ed affermati per poter valorizzare appieno le potenzialità organizzative e commerciali di Football Sport Merchandise nel mercato dell'abbigliamento informale, per lo sport ed il tempo libero.

Nel 1996 in collaborazione con un nuovo partner strategico e in grado di contribuire alla crescita dell'attività, vengono firmati nuovi contratti di licenza per i principali mercati europei e vengono definite le strategie per la penetrazione dei marchi nei mercati degli Stati Uniti, dell'Asia e dell'Africa. Nell'occasione Football Sport Merchandise assume la nuova denominazione sociale di *Basic Properties Services S.p.A.*, in conformità alla logica di denominazione delle società del Gruppo che sposa dalla ragione sociale all'obiettivo operativo della società.

Nel 1997, coerentemente con la strategia di penetrazione del mercato USA, vengono acquisite le attività di una piccola azienda americana, fortemente specializzata nel segmento dell'abbigliamento e della calzatura per la pratica del calcio e con una struttura organizzativa e distributiva adatta a costituire la base operativa del licenziatario statunitense del Gruppo. Viene così costituita la *Kappa USA*, oggi controllata da BasicNet. Nel corso dello stesso anno la controllata Kappa Italia avvia la costruzione di un nuovo complesso immobiliare in Torino destinato ad attività di confezionamento, magazzino e spedizione.

Nel 1998 ha inizio la ristrutturazione della sede del Gruppo Basic in Torino, che prevede la realizzazione del primo progetto *Basic Village* per ospitare, oltre alle attività gestionali del gruppo, anche il rinnovato modello di punto vendita, il *Gigastore Kappa*. Il Gigastore Kappa rappresenta lo strumento con il quale il Gruppo Basic si propone di smaltire le rimanenze generate dall'attività dei Licenziatari e con il Basic Village rappresenta il prototipo funzionale per consentire al Gruppo Basic di entrare in mercati ad alto potenziale di consumo, ma a basso potere d'acquisto.

All'inizio del 1999 *Basic World N.V.*, l'attuale azionista di maggioranza relativa, decide di portare in quotazione le azioni del gruppo di Società da esso possedute, il cui cespite principale è rappresentato dai marchi Kappa e Robe di Kappa. Nel Giugno del 1999 viene deliberato di conferire in *BasicNet S.p.A.* tutte le partecipazioni nelle società attive del

Gruppo detenuto da Basic World N.V., la cui attività principale ruota, a tutti i livelli della catena dell'offerta, sui marchi di proprietà nel settore dell'abbigliamento informale e sportivo.

5.2 La BasicNet vista dall'interno

Il Gruppo BasicNet è attivo nel settore dell'abbigliamento, delle calzature e degli accessori, per lo sport, il tempo libero e per tutte le occasioni di vita sociale e professionale ove non è richiesta la formalità. Il Gruppo BasicNet opera attraverso i marchi Robe di Kappa e Kappa ed è proprietario di altri marchi che la Società intende sfruttare in futuro, fra i quali il più noto è Jesus Jeans. L'attività del gruppo BasicNet consiste nello sviluppare il valore dei marchi di cui è titolare e diffonderne i prodotti attraverso il proprio business system.

5.2.1 Introduzione al modello di business

Il gruppo BasicNet ha impostato il proprio sviluppo su un modello di impresa "a rete", identificando nel licenziatario il partner ideale per la diffusione e la distribuzione dei propri prodotti nel mondo e scegliendo di porsi nei confronti di quest'ultimo non come fornitore del prodotto in sé, ma come fornitore di un insieme integrato di servizi.

Il Business System di BasicNet ha consentito al gruppo di crescere rapidamente, pur mantenendo una struttura agile e leggera: una grande azienda fatta di tante piccole aziende collegate fra loro da un'unica piattaforma informatica completamente integrata al network tramite l'Internet e studiata per la condivisione in tempo reale e per la massima fruizione delle informazioni.

Il Business System, inoltre, è stato concepito e strutturato in modo da consentire lo sviluppo sia per linee interne (nuovi licenziatari o società), sia per linee esterne (nuovi marchi sviluppati o acquisiti, nuove linee di business).

Alla capogruppo BasicNet S.p.A. fanno capo le attività strategiche di ricerca e sviluppo prodotto, di marketing globale, lo sviluppo e coordinamento della rete di licenziatari e la

creazione di sistemi software per consentire la gestione on line di tutti i processi della catena dell'offerta.



Figura 5.1. Loghi di Proprietà Basic Net

A tal fine il gruppo BasicNet sviluppa e coordina il Network nonché presidia con gestione diretta, a garanzia di funzionalità e di redditività, i suoi centri nevralgici, rappresentati da:

- il posizionamento dei marchi e dei prodotti;
- la strategia di marketing e di comunicazione;
- la concezione e l'industrializzazione dei prodotti: creatività, ricerca e sviluppo;
- l'approvvigionamento dei prodotti: i Sourcing Centers;
- lo sviluppo e la gestione del Network.

5.2.2 Posizionamento dei marchi

I marchi del Gruppo BasicNet si posizionano nel settore dell'abbigliamento informale, mercato in forte crescita sin dalla fine degli anni '60 e che si ritiene sia destinato ad avere uno sviluppo progressivo in considerazione della "liberalizzazione" del costume a livello globale. All'interno del settore "abbigliamento informale" sono stati identificati 3 distinti posizionamenti:

- **maschile:** con connotazione di prodotto "sportivo per il tempo libero" coperti dal marchio Robe di Kappa,
- **unisex:** con connotazione di prodotto "sportivo funzionale /sport attivo" coperti dal marchio Kappa,
- **femminile:** con connotazione di prodotto "moda" che saranno coperti dal marchio Jesus Jeans al termine del 2002.

5.2.3 Approvvigionamento dei prodotti

Il gruppo BasicNet non svolge un'attività diretta nella produzione industriale dei prodotti, essendo quest'ultima affidata a soggetti terzi, ma partecipa, tuttavia, alla redditività del ciclo produttivo del Network. Infatti il gruppo BasicNet, mediante società dedicate, i *Sourcing Centers*, presidia e ottimizza tutte le fasi della produzione per conto dei Licenziatari. I Sourcing Centers (gestiti in joint venture con un gruppo asiatico) hanno il compito di individuare e coordinare le società alle quali affidare la realizzazione dei prodotti. A fronte di tale attività il gruppo BasicNet percepisce commissioni dai licenziatari sulla merce da questi acquistata tramite i Sourcing Centers.

L'attività di distribuzione all'ingrosso e di marketing locale è affidata, nei mercati in cui opera il Network, ad una rete di società licenziatarie, che riconoscono al gruppo BasicNet commissioni calcolate sulle vendite, a compenso della licenza dei marchi, del beneficio che deriva dal marketing globale sviluppato direttamente dal gruppo e del "business know-how" loro messo a disposizione. Per l'approvvigionamento dei prodotti finiti, i licenziatari sono liberi di decidere se appoggiarsi ai sourcing centers o rivolgersi ad altri fornitori al di fuori del circuito di sourcing, utilizzando le specifiche di produzione che BasicNet mette loro a disposizione.

Nell'esercizio 1999 due licenziatari erano controllati direttamente dal Gruppo BasicNet: Kappa Italia S.p.A., licenziatario in Italia, che costituisce storicamente il laboratorio di sviluppo del Network e che quindi consente di proporre ai Licenziatari il *know-how*, e Kappa USA Inc., che ha rappresentato la testa di ponte diretta del Gruppo BasicNet per penetrare nel mercato statunitense.

Il gruppo BasicNet è anche presente direttamente nella distribuzione al dettaglio attraverso punti vendita detti Gigastore.

5.2.4 Sviluppo e gestione del Network

La piattaforma informatica costituisce uno dei principali investimenti strategici del Gruppo, al quale è dedicata la massima attenzione sia in termini di risorse umane, sia di centralità nello sviluppo del Business System. La piattaforma è stata concepita e sviluppata in una

prospettiva completamente integrata al web, interpretato dal Gruppo come lo strumento ideale di comunicazione fra gli elementi che costituiscono il network. Il dipartimento di Information Technology si occupa di progettare e implementare sistemi di raccolta e trasmissione dati, sfruttando le opportunità date dalle reti dell'internet, per collegare le società del network BasicNet fra loro e con l'esterno.

In questa prospettiva, lo schema di business è stato disegnato in base a cosiddetti *e-process*, divisioni *dotcom* che eseguono ognuna un tassello del processo produttivo e lo propongono alle altre divisioni utilizzando per l'interscambio e la negoziazione esclusivamente le transazioni on line.

La sfida è utilizzare le tecnologie legate all'internet per fare business: non solo per comunicare con i clienti, ma anche con i propri partner commerciali, per gestire il magazzino, per scambiare informazioni all'interno dell'azienda

5.2.5 Fasi del processo organizzativo

Descriveremo ora in modo cronologico le fasi del processo organizzativo necessarie per la creazione di ogni collezione di abbigliamento; la durata di tale processo può variare in base alla natura ed alla grandezza della collezione da realizzare.

1. Il processo inizia indicativamente due anni prima della commercializzazione nei negozi per le collezioni principali (Autunno/Inverno o Primavera/Estate) dette anche **MegaCollection**, per le collezioni minori, dette **SMU** il processo inizia più tardi. La divisione **BasicSamples.com**, in un arco di circa 2 mesi, crea le prime bozze dei modelli che si potranno commercializzare; adottando la terminologia dell'azienda, questi vengono chiamati **Meta Samples**.
2. Successivamente i licenziatari, rappresentati dalla divisione **BasicCountry.com** scelgono il campionario che intenderanno acquistare basandosi sui Meta Samples. I licenziatari, in fase di acquisto del campionario, indicano una previsione di quantità, non vincolante, che intenderanno acquistare di ogni prodotto.
3. Viene prodotto, in base ai Meta Samples selezionati, il campionario da consegnare ai singoli Licenziatari. Attualmente vengono prodotti circa 100 campioni per ogni

articolo considerandone uno per agente (35 in Italia, 17 in Francia e i rimanenti 48 nel resto del mondo).

4. Vengono così elaborate le previsioni di produzione dalla divisione **BasicForecast.com** sulla base delle previsioni dei licenziatari.
5. Sulla base delle previsioni e dei Meta Samples scelti dai licenziatari, vengono decise dalla divisione **BasicSpecs.com** le specifiche tecniche dei prodotti.
6. A questo punto la divisione **BasicBiddings.com** indice un'asta per la scelta dei produttori (Trading Companies e Sourcing Centers). L'asta viene fatta in un arco di tempo di circa 3 mesi ma, ovviamente, deve durare il meno possibile.
7. Viene consegnato ai licenziatari il campionario. I licenziatari potranno iniziare un'attività di promozione locale (presso i negozi) che indicativamente dura 10 settimane.
8. La divisione **BasicFactory.com** raccoglie gli ordini effettivi dei licenziatari e li passa alle Trading Company che iniziano la produzione. Gli ordini avvengono in tre momenti distinti per ogni collezione (generalmente il primo di piccole quantità, il secondo grande e il terzo contenuto, a meno che il prodotto non abbia avuto particolare successo).
9. Dopo circa quattro mesi dai rispettivi ordini, viene consegnata la merce.
10. Parallelamente BasicForecast.com effettua delle previsioni di vendita, interagendo con i Licenziatari, per intraprendere azioni promozionali.
11. I Licenziatari, a questo punto, possono occuparsi della commercializzazione del prodotto a livello locale.
12. La merce non venduta può essere acquistata e venduta attraverso i Gigastore. Di questa operazione, e dell'eventuale scambio di scorte tra diversi Licenziatari, si occupa la divisione **BasicVillage.com**. Il prezzo di acquisto delle rimanenze è uguale al prezzo iniziale di vendita. Tale operazione è autorizzata esclusivamente se sotto il controllo del gruppo e viene gestita tramite un'interfaccia web. L'incidenza dell'invenduto è di circa il 10% del totale delle merci prodotte.

5.2.6 Ruolo delle divisioni dotcom

BasicTrademark.com: è proprietaria dei marchi del gruppo BasicNet;

BasicMarketing.com: effettua lo studio del marketing globale dei marchi e la supervisione dei licenziatari;

BasicForecast.com: si occupa delle previsioni di produzione e vendita interagendo con i licenziatari. Le previsioni sono sostanzialmente l'aggregato delle quantità di probabile acquisto indicate dai Licenziatari in sede di ordine dei samples. In realtà sono i Licenziatari a stimare le vendite ed assumersi tutti i rischi.

BasicSample.com: svolge l'attività di ideazione e design del campionari; quando un designer crea un nuovo Meta Sample si ipotizza il prezzo, sul mercato europeo, del capo da lui disegnato. I prezzi dei campioni da vendere ai licenziatari sono generalmente il doppio/triplo del prezzo di vendita del prodotto che andrà sul mercato Per ogni paese viene elaborato un prezzo ipotetico di mercato ma BasicNet non può ne imporre ne consigliare il prezzo di vendita al cliente finale. La vendita del campionario è una delle principali forme di remunerazione della divisione BasicSample.com.

BasicSpecs.com: determina le specifiche tecniche dei prodotti da consegnare alle Trading Company e Sourcing Center;

BasicBiddings.com: gestisce l'asta per la scelta delle aziende che dovranno produrre gli articoli per i licenziatari. L'asta è rivolta alle Trading Company (società esterne al gruppo) ed ai Sourcing Center (società gestite in joint venture con il gruppo). Le società destinatarie dell'asta sono attualmente 5.

BasicFactory.com: ha in carico la gestione della produzione; esistono indicativamente tre *deadline* per gli ordini di produzione di ogni collezione, a distanza circa di un mese una dall'altra. Una volta raccolti gli ordini la produzione viene lanciata. Le società che si occupano della produzione, generalmente asiatiche, dialogano con le fabbriche locali e forniscono adeguate garanzie finanziarie (come lettere di credito) e qualitative.

BasicCountry.com: rappresenta l'insieme di licenziatari legati al gruppo BasicNet. E' da sottolineare che i licenziatari acquistano i loro articoli non dal gruppo BasicNet ma

direttamente dalle fabbriche asiatiche. e pagano una commissione (7% del fatturato) per l'intermediazione alle Trading Company. I licenziatari, ogni 3 mesi, devono pagare una somma pari al 10% del loro fatturato per l'insieme di servizi offerti dal gruppo, tale somma andrà ripartita tra BasicTrademark.com (4%), BasicMarketing.com (4%) e BasicSample.com (2%). Ogni licenziatario riceve tre fatture distinte: una da parte della fabbrica che ha prodotto la merce, una da parte di BasicNet per le *royalties* e l'ultima da parte della Trading Company. Attualmente esistono 35 licenziatari che coprono 72 paesi nel mondo.

Legenda attività riportate nel grafico

La fig. 5.2 mostra il modello di business BasicNet.

1. Attività di collaborazione / contatto tra il gruppo BasicNet ed i licenziatari; fase di acquisto del campionario da parte dei licenziatari.
2. Realizzazione dei Meta Sample, scelta dei Meta Sample da realizzare come campionario, richiesta delle specifiche tecniche, produzione e consegna delle merci.
3. Richiesta di previsioni di vendita ai licenziatari.
4. Richiesta alle Trading Company del preventivo per la produzione degli articoli.
5. Raccolta degli ordini delle merci. Gli ordini passati alle Trading Company
6. Vendita dei prodotti sul mercato.

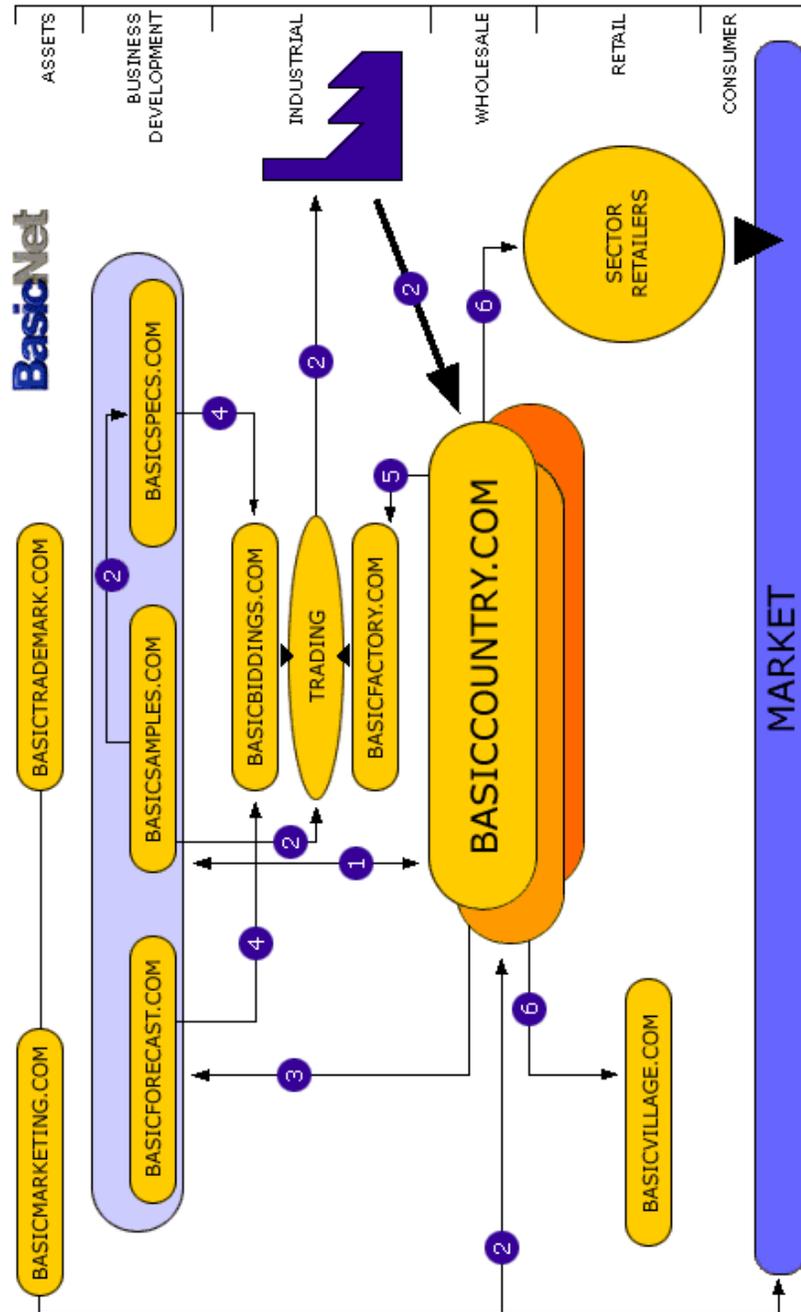


Figura 5.2. Organigramma del modello di business BasicNet

Bibliografia

- [5.1] BasicBiddings, <http://www.basicbiddings.com>, ultima visita Dicembre 2002.
- [5.2] BasicFactory, <http://www.basicfactory.com>, ultima visita Dicembre 2002.
- [5.3] BasicForecast, <http://www.basicforecast.com>, ultima visita Dicembre 2002.
- [5.4] BasicMarketing, <http://www.basicmarketing.com>, ultima visita Dicembre 2002.
- [5.5] BasicNet, <http://www.basicnet.com>, ultima visita Dicembre 2002.
- [5.6] BasicPress, <http://www.basicpress.com>, ultima visita Dicembre 2002.
- [5.7] BasicSample, <http://www.basic-sample.com>, ultima visita Dicembre 2002.
- [5.8] BasicSpecs, <http://www.basic-specs.com>, ultima visita Dicembre 2002.
- [5.9] BasicTrademark, <http://www.basic-trademark.com>, ultima visita Dicembre 2002.
- [5.10] Lamieri M., Merlo F., *BasicJVE - Analisi della BasicNet in prospettiva di formalizzazione per il modello JVEFrame*, <http://eco83.econ.unito.it/tesive/basicnet/>, (user: tesi - password: tangram).

Capitolo 6

Da BasicNet a BasicJes

Questo capitolo è stato scritto da Francesco Merlo e Marco Lamieri. Descrive il percorso seguito per applicare il modello jES alla realtà BasicNet. Vengono anche presentati gli esperimenti sul modello creato e analizzati i risultati ottenuti.

Per poter giungere ad un modello funzionante di BasicJes abbiamo provato più volte a formalizzare il processo organizzativo di BasicNet per poterlo simulare all'interno del modello jES.

Nei mesi da ottobre 2001 a maggio 2002 il nostro lavoro è stato principalmente comprendere il funzionamento del modello jES e cercare di adattare la realtà aziendale agli strumenti che avevamo a disposizione.

Fino al mese di settembre con la versione 0.7.10 non abbiamo avuto a disposizione gli oggetti computazionali, strumento essenziale per ricreare le dinamiche interne di BasicNet.

Riteniamo che la parte più importante, nonché formativa, del nostro lavoro sia stato il continuo apprendimento e dibattito durante gli incontri con il Prof. Terna e gli altri tesisti. Per questo motivo riportiamo, come in un diario, l'intero percorso di tentativi ed errori che abbiamo affrontato, reputando che ogni fase non sia stata un insuccesso ma un forte stimolo per migliorare la nostra capacità di analisi.

6.1 Il processo di formalizzazione

Con il modello che ci accingiamo a sviluppare si intende "far funzionare" un'azienda simulata, basandoci sulla struttura organizzativa del gruppo BasicNet. La nostra rappresentazione non vuole essere un'animazione creata sulla base di sequenze predeterminate di eventi; nel nostro modello gli eventi accadono in modo indipendente, generando interazioni anche imprevedibili tra atti produttivi e unità produttive, proprie della complessità.

6.1.1 Scopo della simulazione

E' del massimo interesse costruire modelli di simulazione che siano fondati su una formulazione astratta e generale di processo di produzione, ma che incorporino anche una realistica visione della realtà, come quella prima prospettata nello sviluppo dei casi aziendali, proprio per simulare processi continui di adattamento e innovazione a prova ed errore (Terna 2002).

Il primo passo è quello di sviluppare ed osservare il funzionamento della nostra azienda simulata ad un livello *macro*, per poter successivamente mettere una lente di ingrandimento sulle diverse fasi di questo processo e sulle unità organizzative, così da complicare e rendere più aderente alla realtà il modello. Il risultato finale avrà molte analogie con un video gioco, senza, per questo, dimenticare gli aspetti teorici e metodologici tipici di una simulazione ad agenti, al cui interno accadranno eventi non definiti a priori da parte del programmatore.

Scopo finale dello studio potrebbe consistere nel confrontare il modello in analisi in contesti diversi (replicabilità del modello) e saggiare la bontà del modello di business BasicNet rispetto ad una struttura organizzativa tradizionale (per esempio a produzione interna e che si assume il rischio di vendita). La simulazione economica mostrerà a questo punto la sua utilità, permettendo di valutare i vantaggi economici e competitivi dei diversi modelli di business.

jESFrame e le "ricette"

Per creare una struttura fortemente flessibile in grado di poter essere modificata continuamente in tutte le sue parti in funzione di crescenti necessità di sviluppo del modello, jESFrame si fonda sul principio della ricetta produttiva. La ricetta viene formalizzata in un vettore di numeri ciascuno dei quali rappresenta un passo nella realizzazione di un bene, merce o servizio che sia. Idealmente la ricetta indica all'impresa virtuale la metodologia da perseguire per ottenere ciò che è stato richiesto.

jESFrame e le "unità" organizzative

Una volta preparata, la ricetta verrà eseguita dalle unità produttive dell'azienda seguendo i passi necessari al suo completamento. Nel nostro caso le unità produttive dovrebbero essere chiamate più correttamente unità organizzative, e potrebbero coincidere con le divisioni *dotcom* di BasicNet. L'attività delle unità organizzative consiste nell'apprendere e modificare le informazioni contenute nei vettori ricetta. In particolare devono inserire le informazioni di completamento del proprio compito ed individuare a quale unità corrisponde il processo seguente. Le unità alle quali si fa riferimento sono tutte quelle necessarie al completamento del processo organizzativo, indipendentemente che siano interne od esterne. Nel nostro caso le unità organizzative esterne verranno impostate con costi pari a 0.

Ogni unità produttiva, quando prende in carico un ordine, lo accoda nella propria lista di ordini da eseguire secondo la modalità cosiddetta FIFO (First In First Out); dopo l'esecuzione richiede all'ordine (coincidente con il modulo che contiene la ricetta produttiva), l'informazione necessaria per trasmetterlo ad una successiva unità [...]. Le singole unità produttive possono essere autonomi micro-mechanismi operanti nel sistema economico, oppure unità integrate all'interno di imprese (Terna 2002).

Ogni unità organizzativa potrà in futuro svolgere diversi compiti proprio come avviene nella realtà aziendale; per esempio l'unità addetta alle previsioni (BasicForecast.com) si

occuperà sia di quelle sulla produzione che sulle vendite, l'unità addetta alle specifiche tecniche creerà le caratteristiche sia delle magliette che dei pantaloni. I Licenziatari potranno essere considerati unità produttive esterne. In un primo momento la multifunzionalità sarà solo implicita, e le unità saranno analizzate ad un livello macro.

Procedendo in questo modo si prevede di ottenere due risultati importanti: il coordinamento della successione degli eventi ed un continuo monitoraggio sulla loro effettiva realizzazione, tenendo sotto controllo eventuali colli di bottiglia nel processo organizzativo.

6.2 Le fasi del processo di formalizzazione

6.2.1 Prima formalizzazione

Le ricette

Il modello di Enterprise Simulator¹ era allora alla versione 0.5 e non sapeva gestire i procurement. Era in previsione lo sviluppo dei passi OR e AND. Questo primo modello è molto generale e descrittivo. E' stato utile per fissare quella che era, a quel momento, la nostra comprensione del funzionamento di jESFrame e, soprattutto, della BasicNet.

Introduciamo in questa sezione una prima formalizzazione delle ricette produttive necessarie per la simulazione del Business System di BasicNet. Descriveremo quindi le principali fasi organizzative necessarie per la realizzare un esemplare prodotto dell'azienda in esame, dal design alla commercializzazione.

Sebbene l'intero processo possa essere descritto "verbalmente" come un'unica sequenza temporale di eventi, al fine di una simulazione in prospettiva jES è stato necessario scomporre l'insieme delle fasi produttive in diverse ricette. Questo avviene poiché il soggetto della nostra descrizione in alcuni passaggi varia ad esempio nella sua natura (da meta-campione a campione) o nel suo ordine di grandezza (da singolo esemplare a lotto di produzione); in altri casi, poi, è necessario che alcune fasi (ricette) vengano eseguite più volte, per creare le condizioni necessarie per l'esecuzione della fase successiva (ad esempio nel caso delle previsioni o degli ordini).

¹Inizialmente denominato Virtual Enterprise

Nel corso di questa descrizione non si farà alcun riferimento a **CHI FA COSA (DW)**, trattando esclusivamente il **COSA FARE (WD)**. Al termine di ogni ricetta verranno riportate alcune note.

START:

[1] Creazione artistica del meta-campionario

Nota: perchè avvenga un aggancio tra una ricetta ed un'altra e per poterle differenziare in base alla tipologia del prodotto, è necessario affiancare alle ricette dei valori identificativi del prodotto e delle sue caratteristiche. In questa prima fase la ricetta restituirà una sorta di codice del prodotto, la collezione alla quale appartiene ed un prezzo indicativo. Nel seguito della descrizione useremo una notazione (provvisoria) del tipo [ID] . [COLLECTION] . [METAPRICE].

A) DA META-CAMPIONE A CAMPIONE

[2] Acquisto del campione di [ID] da realizzare

[3] Previsione di vendita di [ID]

Nota: questa ricetta dovrà rilasciare il valore riguardante le previsioni di vendita effettuate su un certo prodotto e (eventualmente) chi le ha effettuate; la notazione sarà quindi [ID] . [UNIT] . [FORECAST].

B) CREAZIONE DEL CAMPIONARIO

[4] Somma delle previsioni di vendita --> SOMMA([ID] . [FORECAST])

[5] SE SOMMA([ID] . [FORECAST]) >= MIN_FORECAST

Creazione delle specifiche tecniche

ALTRIMENTI

attendi (o termina produzione)

[6] Produzione del campione di [ID]

[7] Consegna del campione di [ID]

Nota: in questo caso le ricette A e B sono state separate poichè cambia il soggetto della nostra descrizione (dal meta-campione al campione) e poichè l'esecuzione della ricetta B è condizionata da più esecuzioni della ricetta A. Il valore MIN_FORECAST indica la quantità minima richiesta per procedere alla produzione e commercializzazione di un prodotto. Anche questa ricetta, per essere agganciata alla seguente deve rilasciare dei valori. Se è andata a termine rilascerà qualcosa tipo [ID].[UNIT], ossia la corrispondenza tra il campione e l'unità che l'ha ordinato.

C) ASTA

[8] SE SOMMA([ID].[FORECAST])>=MIN_FORECAST

Offerta prezzo di produzione [RANGE]

ALTRIMENTI

attendi (o termina offerta)

[9] Scelta del produttore [FIRM] di [ID] in base a [RANGE]

Nota: la ricetta C è disgiunta dalla B poichè avviene in parallelo. In questa fase il valore [RANGE] rappresenta le fasce di prezzo offerte (corrispondenti a diverse quantità) per la produzione di un prodotto. [FIRM] identifica il soggetto che ha effettuato l'offerta.

D) ORDINE

[10] Ordine del prodotto [ID] e della quantità [QUANTITY]

Nota: questa ricetta dovrà essere accompagnata anche dell'identificativo [UNIT] di chi ha effettuato l'ordine (necessario per la sua consegna).

E) PRODUZIONE

[11] Produzione di SOMMA([ID].[QUANTITY])

[12] Vendita e consegna di [ID].[QUANTITY] a [UNIT]

Nota: provvisoriamente abbiamo inserito una separazione tra le ricette D ed E. Supponiamo, infatti, che la produzione venga avviata solo una volta raccolti tutti gli ordini arrivati entro un certo periodo.

F) COMMERCIALIZAZIONE

[13] Vendita al dettaglio di [ID]

G) GESTIONE INVENDUTO

[...] Non realizzata.

Le unità organizzative

Completiamo ora la nostra descrizione analizzando i soggetti che sono in grado di compiere i passi delle ricette. Descriveremo quindi il modello della nostra azienda intermini di **CHI FA COSA (DW)**.

100) BASICSAMPLE

[1] Creazione artistica del meta-campionario

101) LICENZIATARIO

[2] Scelta del campione di [ID] da realizzare

[3] Previsione di vendita di [ID]

[10] Ordine del prodotto [ID] e della quantità [QUANTITY]

[13] Vendita al dettaglio di [ID]

[14] Inserimento nel magazzino dell'invenduto di [ID]

103) BASICFORECAST

[4] Somma delle previsioni di vendita --> SOMMA([ID].[FORECAST])

103) BASICSPEC

[5] SE SOMMA([ID].[FORECAST])>=MIN_FORECAST

Creazione delle specifiche tecniche

ALTRIMENTI

attendi (o termina produzione)

104) BASICSAMPLEFACTORY

[6] Produzione del campione di [ID]

[7] Vendita e consegna del campione di [ID]

Nota: abbiamo creato quest'ultima unità, addetta esclusivamente alla produzione del campionario, al solo fine di tenere una contabilità separata tra chi produce il prodotto finito ed il campione. In oltre ci sembra che la produzione di questi due prodotti sia totalmente differente (il campione è fatto in modo più artigianale)

105) BASICBIDDING

[9] Scelta del produttore [FIRM] di [ID] in base a [RANGE]

105) TRADINGCOMPANY

[8] SE SOMMA([ID].[FORECAST])>=MIN_FORECAST

Offerta prezzo di produzione [RANGE]

ALTRIMENTI

attendi (o termina offerta)

[11] Produzione di SOMMA([ID].[QUANTITY])

[12] Vendita e consegna di [ID].[QUANTITY] a [UNIT]

Riflessioni sulla prima formalizzazione

La nostra prima proposta presentava sicuramente molte lacune sulla comprensione del modello jESFrame; la presenza di più ricette per descrivere un atto organizzativo "continuo" snaturava il concetto stesso di ricetta produttiva. Le nostre singole ricette si presentavano, infatti, più come una sequenza di algoritmi tipici della programmazione classica.

Le difficoltà incontrate erano sicuramente dovute anche dall'ambizioso progetto di adattare il modello jESFrame ad una realtà notevolmente differente dall'iniziale concezione dell'applicazione; dovevamo comprendere quanto potesse esserci in comune tra la descrizione

di un processo produttivo "classico" (dove le unità sono delle macchine per la produzione) e la descrizione di un processo organizzativo (dove le unità sono per lo più centri di scelta e gestione delle informazioni).

E' da sottolineare anche il fatto che il nostro lavoro, e l'applicazione in Vir, erano i primi tentativi di utilizzo del modello in un contesto reale. Il confronto parallelo sugli sviluppi delle due applicazioni è stato un continuo spunto di riflessioni sulla bontà del modello, e di proposte per arricchire le funzionalità del programma.

Conclusa questa prima fase del lavoro, ma soprattutto di comprensione, la nostra attenzione si è spostata verso una descrizione unica e continua del business system di BasicNet utilizzando nuovi strumenti, seppur non ancora operativi in quel momento, come i passi AND e gli oggetti computazionali.

6.2.2 Seconda formalizzazione

Presentiamo in questa sezione una nuova formalizzazione delle ricette organizzative per l'applicazione del modello jESFrame al caso BasicNet. La nuova proposta presenta un'unica ricetta, detta **Main**, che descrive tutti i passaggi organizzativi di un tipico prodotto dell'azienda, dall'ideazione alla vendita.

In questa sezione introduciamo anche un nuovo metodo per risolvere i problemi di tipo computazionale, come le previsioni e le somme. All'interno della ricetta sono presenti degli oggetti speciali capaci di assolvere a queste ed ad altre necessità.

La tabella qui sotto riportata descrive tutti i passi della ricetta **Main**; sotto ogni passaggio sono, eventualmente, riportati gli oggetti speciali che dovranno essere richiamati e le eventuali ricette da essi generate. Al termine della tabella è riportata una descrizione dettagliata di ogni passo ed oggetto speciale (da leggere attraverso le coordinate di riga e colonna).

A	B	C	D	E	F	G	H	J	K	L	M	N	O	
Ricetta Main	1	2	3	4	&&1 5	&&2 6	7	&&0	8	9	10	11	e	1
Oggetti Speciali		O2	O3		O5	O6	O7		O8	O9	O10	O11		2
			201		501	601				901	1001			3
		202			502						1002			4
Oggetti Speciali		O201				O601				O901	O1001	O1101		5
												O11001		6

Legenda della tabella

- Cella: B1 Creazione artistica del meta-campionario di [ID]
- Cella: C1 Pubblicazione del meta-campionario di [ID]
- Cella: D1 Somma delle previsioni di vendita di [ID]
- Cella: E1 Definizione delle specifiche tecniche di [ID]
- Cella: F1 Produzione del campionario di [ID]
- Cella: G1 Asta per la formazione del prezzo di produzione di [ID]
- Cella: H1 Formazione del prezzo di produzione di [ID]
- Cella: J1 Formazione delle deadlines di [ID]
- Cella: K1 Raccolta ordini di produzione di [ID]
- Cella: L1 Produzione di [ID]
- Cella: M1 Commercializzazione e gestione dell'invenduto di [ID]
- Cella: C2 L'oggetto speciale O2 (sampleCollector) genera ricette di tipo "201-202" e contabilizza le previsioni di vendita di [ID]
- Cella: D2 L'oggetto speciale O3 (forecastMaster) richiede a O2 le somma delle previsioni per [ID] e procede con la ricetta main se le previsioni sono maggiori del minimo richiesto, altrimenti termina la ricetta
- Cella: F2 L'oggetto speciale O5 (sampleBuilder) genera delle ricette di tipo "501-502" e si occupa della produzione dei campioni di [ID] interrogando O2 per conoscerne le quantità
- Cella: G2 L'oggetto speciale O6 (biddingMaster) genera delle ricette di tipo "601" per richiedere l'offerta di produzione di [ID]
- Cella: H2 L'oggetto speciale O7 (PriceMaker) interroga O6 per la formazione del prezzo di produzione di [ID]
- Cella: J2 L'oggetto speciale O8 (deadLinesMaker) genera le deadline di [ID] stabilendone il numero e la loro posizione nello schedule

- Cella: K2 L'oggetto speciale O9 (orderCollector) genera ricette del tipo "901" per raccogliere gli ordini di [ID]
- Cella: L2 L'oggetto speciale O10 (productionMaster) interroga O9 e se è scaduta una deadline genera ricette del tipo "1001-1002" per la produzione degli esemplari di [ID]
- Cella: M2 L'oggetto speciale O11 (wareHouseMaster) tiene traccia dell'inventario e con qualche regola lo scambia.
- Cella: C3 Acquisto del campione di [ID] da realizzare
- Cella: F3 Produzione del campione di [ID]
- Cella: G3 Richiesta prezzo di produzione di [ID]
- Cella: K3 Raccolta ordine di [ID]
- Cella: L3 Produzione di [ID]
- Cella: C4 Previsione quantità di vendita di [ID]
- Cella: F4 Consegna del campione di [ID]
- Cella: L4 Consegna di [ID]
- Cella: C5 L'oggetto speciale O201 (sampleSelector) sceglie il campionario con dei criteri stabiliti e genera le previsioni di acquisto di [ID]
- Cella: G5 L'oggetto speciale O601(biddingMaker) genera, con criteri stabiliti, il prezzo di produzione di [ID]
- Cella: K5 L'oggetto speciale O901 (orderMaker) decide la quantità di [ID] da produrre (0=non produrre).
- Cella: L5 L'oggetto speciale O1001 (withdrawMaker) effettua il ritiro degli esemplari di [ID]
- Cella: M5 L'oggetto speciale O1101 (marketing) genera ricette del tipo 11001 che rappresentano il tentativo di vendita degli esemplari di [ID]
- Cella: M6 Vendita degli esemplari di [ID]

Riflessioni sulla seconda formalizzazione

Al termine di questa fase del lavoro riuscimmo a descrivere l'intero processo organizzativo in un'unica ricetta Main. Notammo subito che con questo sistema stavamo perdendo gran parte dell'informazione necessaria per ricostruire un'azienda "virtuale", funzionante, nella prospettiva di una simulazione ad agenti.

Con questa tecnica gli eventi erano il frutto di azioni di "causa ed effetto" generati con qualche distribuzione di probabilità. Gli oggetti computazionali concatenati in questo modo non davano spazio all'emergere dell'interazione tra le unità aziendali, che creano i fenomeni complessi oggetto del nostro studio. Una simulazione di questo tipo risultava quindi troppo simile ad un'analisi di processo e perciò distante dai nostri propositi.

6.2.3 Terza formalizzazione

Con questa terza formalizzazione vogliamo rappresentare il flusso organizzativo attraverso la descrizione di più ricette concatenate da diversi codici identificativi e dall'azione di *procurement*.

La proposta è stata rielaborata attraverso progressivi miglioramenti che si possono sintetizzare in tre passaggi logici.

Proposta 3.1

In un primo momento abbiamo scritto tutte le ricette necessarie al funzionamento della simulazione legandole insieme con dei procurement e utilizzando gli oggetti computazionali.

Nell'esempio il procurement è utilizzato per legare tra loro le ricette, mentre lo scopo dell'oggetto C01 è di creare i due codici [ID] e [IC].

Creazione collezione completa 20 d 1 C01 [ID, IC] i 1001 d 180 ;

Creazione metacampionario p 1 1001 30 d 2 31 d 1 i 1003 d 30 ;

Le ricette esplorano tutti i casi possibili di interazione tra Trading Company, BasicNet e Licenziatario. Viene inoltre introdotto l'oggetto speciale C0 che, se ha in input il valore 0, interrompe la ricetta che lo ha richiamato.

Nell'esempio ogni licenziatario controlla che esista il metacampionario pubblicato sul web (P i 1003), decide l'acquisto (991x) e prevede le quantità (C10), nel caso che il licenziatario x non intenda acquistare [ID] le previsioni [FORECAST] saranno pari a 0 e la ricetta verrà bloccata dall'oggetto C0.

Il controllo di fattibilità controlla che i licenziatari abbiano ordinato (p 310099100: 1099102), viene creata la soglia minima di produzione [TRESHOLD] (C20), viene controllata la fattibilità (C30) e se non si supera la soglia C0 blocca la ricetta.

Con il passo 42 si creano le specifiche tecniche di [ID] e la ricetta viene messa in una *end-unit iterate* per 180 giorni.

Scelta campionario licenziatario 0

```
p 1 1003 9910 d 4 [ID,IC] C10 [FORECAST] [FORECAST]
C0 [0,1] i 1099100 d 180
```

Scelta campionario licenziatario 1

```
p 1 1003 9911 d 4 [ID,IC] C10 [FORECAST] [FORECAST]
C0 [0,1] i 1099101 d 180
```

Scelta campionario licenziatario 2

```
p 1 1003 9912 d 4 [ID,IC] C10 [FORECAST] [FORECAST]
C0 [0,1] i 1099102 d 180
```

Controllo fattibilità campionario

```
p 3 1099100 : 1099102 40 d 1 [NULL] C20 [THRESHOLD] 41 d 5
[FORECAST, ID, THRESHOLD] C30 [T/F, SUMMFORECAST] [T/F]
C0 [0,1] 42 d 7 i 1004 d 180
```

Proposta 3.2

Nella seconda stesura ci siamo posti il problema della coerenza tra i vari procurement e i codici identificativi dei prodotti [ID] e [IC]. Abbiamo cercato di compattare la notazione

delle ricette inserendo i codici [ID], [TC] e [LC] all'interno dei numeri del passo e del codice identificativo della ricetta. Si viene così a creare una sorta di codice prodotto che contiene al suo interno le informazioni su chi lo ha ordinato, da chi è stato prodotto e di cosa si tratta.

Ipotizziamo che l'*Order Generator* si occuperà di "compilare" in modo automatico i codici, e esplodendo in questo modo le possibilità riportate in Proposta 3.1.

Nell'esempio si eseguono le stesse operazioni descritte in precedenza con la nuova notazione dove l'asterisco significa "tutti".

Creazione collezione

```
10 d 15 i 1001(IC) d 180
```

Creazione metacampionario

```
p 1 1001(IC) 20 d 2 21 d 1  
i 1002(IC)(ID) d 30
```

Scelta campionato licenziatario (LC)

```
p 1 1002(IC)(ID) 30(LC) d 1 C10 [ ] [FC] CO [FC] [NULL]  
i 1003(IC)(ID)(LC) d 180
```

Controllo fattibilità campionato

```
40 d 1 C20 [ ] [TH] p (LC) 1003(IC)(ID)(*) 41 d 1  
C30 [FC i1003(IC)(ID)(*), TH]  
[T/F,SFC] CO [T/F] [NULL] 42 d 7 i 1004(IC)(ID) d 180
```

Proposta 3.3

Nella terza stesura abbiamo riscritto la notazione degli oggetti computazionali per renderli uniformi alle indicazioni fornite durante l'incontro tesisti. Con questa nuova notazione si agevola il compito del *parser* che dovrà vagliare la sintassi per tradurla in linguaggio interno al programma.

Nell'esempio riportiamo le stesse operazioni con la nuova notazione.

Creazione collezione

10 d 15 i 1001(IC) d 180

Creazione metacampionario e pubblicazione

p 1 1001(IC) 20 d 2 21 d 1 i 1002(IC)(ID) d 30

Scelta campionato licenziatario (LC)

p 1 1002(IC)(ID) 30(LC) d 1 c 10 in 0 out 1 FC

c 0 in 1 FC self out 1 T/F

i 1003(IC)(ID)(LC) d 180

La notazione degli oggetti computazionali utilizzata è la seguente:

c n.computazione in *n.input* NOME1 LOCAZIONE1 ... NOME_n LOCAZIONE_n

out *n.output* NOME1 LOCAZIONE1 ... NOME_n LOCAZIONE_n

Riflessioni sulla terza formalizzazione

Con questa formalizzazione concentrammo nuovamente l'attenzione non più sull'intero processo organizzativo ma sulle singole azioni necessarie per ottenere i diversi prodotti finiti.

In pratica si tornò a ragionare in termini di prodotto anziché di collezione.

Punto di svolta fu sicuramente l'introduzione del passo di procurement che, in principio, era necessario solo per il modello in Vir. A questo punto eravamo in grado di descrivere le singole azioni (ricette) che dovevano essere intraprese da parte dei soggetti della nostra simulazione (unità).

Il problema che sorgeva era l'enormità di azioni che dovevamo andare a descrivere.

6.2.4 Quarta formalizzazione

Con questa quarta proposta introduciamo l'utilizzo di un data base *Access* per la generazione automatica delle ricette descritte nella proposta 3.2. La grammatica degli oggetti speciali è stata semplificata eliminando parametri ridondanti come la lunghezza del vettore e i segnaposto per i valori in output. La generazione della moltitudine di ricette necessarie

per esplorare tutti casi possibili è stata ottenuta realizzando *query* in linguaggio SQL che creano tutte le combinazioni possibili dei dati, e i *Report* di Access per la presentazione delle ricette. Tali Report sono facilmente esportabili in formato *Excel* già formattati seguendo la il formalismo di jES.

Creazione collezione Autunno - Inverno

10 d 15 i 10012 d 180

Creazione metacampionario Maglione

p 1 10012 20 d 2 21 d 1 i 100225 d 180

Scelta licenziatario A

p 1 100225 301 d 1 c 10 i 1003251

Scelta licenziatario B

p 1 100225 302 d 1 c 10 i 1003252

Scelta licenziatario C

p 1 100225 303 d 1 c 10 i 1003253

Creazione metacampionario Camicia

p 1 10012 20 d 2 21 d 1 i 100226 d 180

Scelta licenziatario A

p 1 100226 301 d 1 c 10 i 1003261

Scelta licenziatario B

p 1 100226 302 d 1 c 10 i 1003262

Scelta licenziatario C

p 1 100226 303 d 1 c 10 i 1003263

Mail licenziatario A

p 2 1003251 1003261 40 d 1 i 100421 d 180

Mail licenziatario B

p 2 1003252 1003262 40 d 1 i 100422 d 180

Mail licenziatario C

p 2 1003253 1003263 40 d 1 i 100423 d 180

Riflessioni sulla quarta formalizzazione

La strada intrapresa era sicuramente improponibile. Il moltiplicarsi di azioni dovuto all'interazioni di più collezioni contenenti più prodotti che dovevano essere "lavorati" da almeno 35 licenziatari e 5 Trading Company portava ad una "esplosione" di ricette (nell'ordine delle 90.000).

Sebbene questo sistema potesse funzionare nella, allora, versione corrente di jESFrame, ci si sarebbe imbattuti nello studio di un "mondo" talmente grande che ogni dato avrebbe perso di significatività poiché sarebbe stato difficile determinare se gli eventi che si sarebbero osservati erano il frutto di azioni imprevedute o imprevedibili.

Era necessaria una maggiore generalità delle ricette.

6.2.5 Quinta formalizzazione

Con questa quinta proposta introduciamo l'utilizzo delle memorie all'interno delle ricette. L'utilità è distinguere i singoli prodotti, poter tenere memoria di alcune caratteristiche e lo stato del prodotto durante la lavorazione. Dal punto di vista informatico si tratta di vettori con posizioni collegate alla singole ricette e il cui contenuto è gestito dagli oggetti computazionali, nell'applicazione BasicNet queste posizioni corrispondono con i codici [ID].

Di seguito vengono riportate le ricette necessarie per giungere alla creazione del metacampionario.

creazione collezione

c 0 1 d 15 m 0 ;

creazione metacampionario

c 1 2 d 30 m 1 ;

previsioni agente 1

c 2 101 d 1 m 101 ;

previsioni agente 2

c 2 102 d 1 m 102 ;

previsioni agente 3

c 2 103 d 1 m 103 ;

[...]

previsioni agente 35

c 2 135 d 1 m 135 ;

somma previsioni

c 3 3 d 2 m 3 ;

produzione campionario

c 4 4 d 1 41 d 60 m 4 ;

Introduciamo anche un file di supporto chiamato "map" che verrà utilizzato dagli oggetti computazionali per conoscere le caratteristiche [IC] di ogni prodotto (identificato dagli [ID]). La colonna "m" indica la posizione del vettore, e corrisponde nel nostro caso a ID.

La tab.6.1 mostra un esempio di "map"

Sul modello dell'applicazione in Vir proponiamo anche un *orderSequence* che scandisce il lancio delle ricette nel tempo e che verrà letto dall'*orderDistiller*.

La sintassi è la seguente:

- [day] = primo valore di ogni riga che corrisponde al tick di lancio della ricetta,
- [FROM] ; [TO] * [RECIPE] = lancia la ricetta [RECIPE] per tutti i prodotti (nel nostro caso [ID]) da [FROM] a [TO].

La teb. 6.2 riporta un esempio di *orderSequences.xls*:

Riflessioni sulla quinta formalizzazione

La versione definitiva era vicina. La semplificazione delle ricette avvenne grazie all'uso del nuovo strumento introdotto nel modello jESFrame, i layer.

m	ID	IC
1	1	1
2	2	1
3	3	1
4	4	1
5	5	1
6	6	1
7	7	2
8	8	2
9	9	2
10	10	2
11	11	2
12	12	3
13	13	3
14	14	3
15	15	3
16	16	3
17	17	3
18	18	3
19	19	3
20	20	3

Tabella 6.1. Esempio dei file "map" per coordinare gli articoli e le collezioni descritte in *orderSequences.xls*.

1	1	;	6	*	1		
2	1	;	3	*	2		
3							
4							
5	4	;	6	*	2		
6	1	;	6	*	101	;	135
7							
8							
9							
10	1	;	2	*	3		
11	1	;	2	*	4		
12	3	;	6	*	3		
13	3	;	6	* 4			

Tabella 6.2. Esempio di *orderSequences.xls*

Le ricette erano di facile lettura e la scrittura del file *orderSequences.xls* poteva risultare molto più comprensiva. L'ultima questione da affrontare era ancora la vera natura degli oggetti computazionali, passaggio necessario per giungere ad una prima formalizzazione che potesse realmente funzionare nel modello.

6.3 Proposte di modifica al codice

In questa sezione introduciamo gli sviluppi e le nuove funzionalità necessarie per applicare il modello jES alla BasicNet.

6.3.1 Prime proposte

Le prime proposte sono state respinte perché tentavano di ricostruire un semplicissimo linguaggio di programmazione con comandi definiti dagli oggetti computazionali. La metodologia definitiva, adottata per la creazione degli oggetti computazionali, prevede, invece, la costruzione di molti metodi di una classe Java, crati *ad hoc* per gli scopi della simulazione, che sfruttano la potenza e completezza del linguaggio JavaSwarm per il loro funzionamento.

Memorie

Gestione dei vettori di memoria, per raccogliere (in modo analogo alle *endUnit*) i dati di collezioni e articoli creati, previsioni di acquisto, quantità ordinate, prezzi di produzione ecc... Ogni riga dei vettori di memoria (nel nostro caso) dovrebbe raccogliere i dati relativi ad un articolo (es: in riga uno di una memoria troviamo le previsioni del prodotto 1, in riga due del prodotto 2 ...)

Segnali

Possibilità di inserire all'interno del file Excel contenente la sequenza degli ordini (*orderSequence*) un tipo di "segnale" che permetta di attribuire le ricette produttive successive ad esso ad una determinata riga dei vettori di memoria.

Per il nostro modello potrebbe essere utile avere un doppio livello di questi "segnali" (ad esempio un primo per indicare la collezione ed un secondo per indicare il prodotto).

OR con il criterio prezzo

Possibilità di specificare i criteri dell'operatore OR; nel nostro caso si dovranno confrontare i valori contenuti in alcuni vettori memoria (quelli relativi alle offerte di produzione) e scegliere il ramo della ricetta corrispondente alla Trading Company che ha offerto il prezzo piú basso.

Oggetti computazionali comuni

Utilizzo di oggetti computazionali per poter effettuare principalmente somme e previsioni (estrazione di numeri casuali); questi oggetti potranno prelevare i dati dai vettori di memoria ed anche inserirli.

Abbiamo ipotizzato la sintassi di alcuni oggetti computazionali comuni:

Get

Sintassi: `get m k`

Legge il valore nella memoria m colonna k

Put

Sintassi: `put n m k`

Mette il valore n nella memoria k

Random

Sintassi: `r m k min max`

Inserisce nella memoria m colonna k un valore casuale uniforme compreso nell'intervallo min max

Sum

Sintassi: `sum t k n m1 m2 .. mn`

Somma gli n valori m1 m2 ... mn e inserisce il risultato nella memoria t colonna k

Subtraction

Sintassi: `sub t k m1 m2`

Sottrae i valori $m_1 - m_2$ e inserisce il risultato nella memoria t colonna k

If

Sintassi: `if m k`

Controlla che sia presente un valore nella memoria m colonna k e non lascia proseguire la ricetta fino a quando non viene rilevato.

If greater

Sintassi: `ifg m k n`

Controlla che nella memoria m colonna k sia presente un valore maggiore n e non lascia proseguire la ricetta fino a quando non viene rilevato.

If minor

Sintassi: `ifm m k n`

Controlla che nella memoria m colonna k sia presente un valore minore n e non lascia proseguire la ricetta fino a quando non viene rilevato.

Higher

Sintassi: `high t k n m1 m2 ... mn`

Sceglie il più grande tra gli n valori $m_1 m_2 \dots m_n$ e lo inserisce nella memoria t colonna k

Smaller

Sintassi: `small t k n m1 m2 ... mn`

Sceglie il più piccolo tra gli n valori $m_1 m_2 \dots m_n$ e lo inserisce nella memoria t colonna k

6.3.2 Le modifiche apportate

Le modifiche al codice descritte di seguito sono state realmente apportate alla versione 0.0.7.10 di jES per creare una prima prova di simulazione funzionante. Queste modifiche sono la base del modello finale e realistico descritto di seguito.

Per poter adattare jES versione 0.9.7.10 alla simulazione della BasicNet abbiamo dovuto modificare i files *OrderDistiller.java* e *Recipe.java* in modo da gestire la lettura dei codici "layers" dal file *orderSequences.xls*, e la lettura dei codici di riferimento agli oggetti computazionali dal file *recipes.xls*.

Per apportare queste modifiche ci siamo ispirati al lavoro realizzato dai nostri colleghi Elena Bonessa, Antonella Borra e Cristian Barreca.

Le modifiche apportate sono le seguenti:

- Seguendo l'impostazione del lavoro in Vir², abbiamo creato in terzo array chiamato "orderSequence3" contenente il codice-layer al quale un ordine appartiene. In questo modo, in fase di generazione, gli ordini verranno creati leggendo questi tre ARRAY per determinare il codice della ricetta, la quantità ed il layer.
- Per la lettura dei files OrderSequences*.xls abbiamo creato un nuovo metodo "checkForLayer(ExcelReader e)" che ad ogni lettura del file controlla prima la presenza di un nuovo segnale (l = layer) e poi legge il valore corrispondente. Un esempio di come può essere inserito il codice è presente nei files OrderSequences*.xls.
- Abbiamo corretto la lunghezza degli Array contenenti il codice degli ordini e le quantità ordinate. Attualmente avevano lunghezza "dictionaryLenght". Per la nostra simulazione questa lunghezza non è giustificata poiché in un giorno possono essere immessi ordini con lo stesso codice ma con layer diversi. La nuova lunghezza provvisoriamente è stata impostata a $maxOrderSequence = dictionaryLenght * MaxLayerNumber$
- Abbiamo aggiunto al metodo *distill* la capacità di leggere i passi computazionali e lanciare gli ordini *layerd*.
- Abbiamo creato un primo oggetto computazionale con codice 1997 che genera un numero casuale tra 0 e 5 e lo pone nella posizione 0,0 della prima matrice (matrix 0).

²Abbiamo anche corretto alcuni bug: in *OrderDistiller.java*: la variabile "firstTime" viene utilizzata per il passaggio dalla lettura del file OrderSequenceModified.xls (una volta all'inizio della simulazione) alla lettura del file OrderSequence.xls (in modo ciclico per il resto della simulazione). Nella versione precedente tale variabile era inserita nel metodo "readOrderSequence()" quindi veniva ridefinita ad ogni lettura del file. La simulazione avveniva quindi SOLO sugli ordini scritti nel primo file. Abbiamo quindi dichiarato questa variabile globale e statica. Abbiamo corretto un bug riguardante gli Array contenenti il codice degli ordini e le quantità: questo non venivano azzerati all'inizio di ogni ciclo; quindi ogni giorno si immettevano alcuni ordini dei giorni precedenti.

- Attualmente la nostra versione presenta delle stampe a video per meglio comprendere il funzionamento interno del modello.

6.4 BasicJes-0.9.7.30.b: BasicJes la simulazione di BasicNet

In questa sezione viene descritto il primo modello funzionante e realistico di simulazione della BasicNet.

6.4.1 Dibattito sull'uso dei layers

La prima proposta di BasicJes nella versione 0.9.7.21.b.2 utilizza gli oggetti computazionali e i layers per organizzare e regolare la produzione.

I layers vengono usati per separare le *collezioni* e renderle, così, indipendenti l'una dalle altre. Ipotizzando di voler simulare un anno di gestione, dovremmo prevedere un numero di circa 10 collezioni. Di queste 4 sono principali, cioè contenenti qualche migliaio di articoli, e altre 6 sono definite *SMU*, cioè collezioni minori per eventi speciali (come la collezione scuola o lo speciale per la nazionale di calcio) che contano qualche decina di articoli.

I layer sono dei contenitori di prodotti assolutamente indipendenti tra loro. jESFrame gestirà tutte le operazioni, come sequential batch, stand alone batch, procurement e computazioni, in modo indipendente per ogni layer. La ricerca dei prodotti fatta dai procurement avverrà solo tra prodotti appartenenti al layer di riferimento del procurement. Le computazioni su matrici avverranno su matrici distinte a seconda del layer di riferimento.

Durante lo sviluppo degli oggetti computazionali è stato necessario analizzare con attenzione gli strumenti a disposizione e operare una scelta. L'obiettivo è stato scrivere gli oggetti computazionali BasicNet nel modo più generale, flessibile e pulito possibile.

La maggiore difficoltà è stata individuare il corretto "punto di vista". La BasicNet, durante le diverse fasi del processo organizzativo, opera su magnitudini differenti. In molti casi le operazioni sono svolte in riferimento ai singoli articoli ma, altre volte, è necessario ragionare sulle collezioni, altre ancora sul Brand.

Questa struttura organizzativa suggerisce due possibili metodi di disegno degli oggetti

computazionali, con due significati differenti attribuiti ai *layers*:

Layer=Collezione: in questo caso i layer vengono utilizzati per l'ordine di grandezza maggiore, la collezione. I singoli prodotti sono tenuti indipendenti dagli oggetti computazionali. All'interno delle MemoryMatrix ogni colonna è un prodotto differente. Questa struttura consente grande flessibilità. Ogni oggetto computazionale potrà decidere di operare sulla singola colonna-prodotto ogni volta che viene utilizzato, portando alla generazione di tante ricette quanti sono i prodotti da lavorare. In questo modo sarà possibile fare procurement sui singoli prodotti della collezione, e osservare i dettagli per prodotto delle operazioni sui grafici. In altri casi (come per esempio durante le previsioni) non disponiamo delle informazioni relative ai singoli prodotti (come per esempio il tempo necessario al licenziatario per prevedere gli ordini di un singolo prodotto) e non avrebbe dunque senso fare delle congetture arbitrarie. In questi casi è necessario ragionare per collezione. Gli oggetti computazionali si occuperanno, con dei cicli *for*, di eseguire l'operazione su tutti i prodotti della collezione (e quindi su tutte le colonne della matrice) in una singola ricetta. Quando si ragiona per collezione, certamente, le informazioni che verranno mostrate sui grafici saranno meno rilevanti, e non si potranno eseguire dei procurement sulla ricetta costruita "per collezione".

Layer=Prodotto: un secondo modo di ragionare proporrebbe di considerare la più piccola unità di misura in gioco, il singolo articolo, e attribuire ad ogni articolo un layer differente. In questo modo il disegno degli oggetti computazionali sarebbe più pulito, rispettando una pura logica Object Oriented. La metafora di fondo, per la quale ogni articolo è intrinsecamente differente da qualunque altro, sarebbe rispettata con maggior rigore.

La scelta degli oggetti contenenti collezioni

In ragione di tutte le implicazioni, sia metodologiche sia pratiche, abbiamo optato per costruire oggetti computazionali su matrici bidimensionali, che contengano al loro interno tutte le informazioni dell'intera collezione. La collezione è organizzata con un prodotto per

ogni colonna della matrice.

La conoscenza dal punto di vista di ogni computazione viene solo spostata dall'order-Sequences agli oggetti computazionali. Questo non è un grave errore perché, in BasicNet, le operazioni organizzative avvengono su prodotti o collezioni a seconda dei casi. La stessa operazione (che per noi è rappresentata da un'oggetto computazionale) avviene sempre nello stesso modo, con lo stesso tipo di raggruppamento. Questo tipo di gestione non compromette quindi il realismo del modello. Anche i risultati saranno, se non uguali nei grafici, con lo stesso valore informativo.

D'altro canto utilizzare la metafora *Layer=prodotto* sarebbe stato più rigoroso, e avrebbe fornito sui grafici informazioni più dettagliate, ma spesso inutili. Si pensi al caso delle lavorazioni che avvengono per collezione, avrebbero considerato gli articoli della collezione come prodotti distinti, lavorandoli in sequenza, e con il conseguente formarsi di code di attesa insensate logicamente.

6.4.2 Il modello: le unità

Per far funzionare concretamente la simulazione della BasicNet abbiamo dovuto formalizzare all'interno di jES le unità e i loro attributi. Come metafora abbiamo immaginato che ogni divisione *dotcom* di basic net corrispondesse ad un unità con costi fissi e variabili pari a 1.

I licenziatari e le Trading Company corrispondevano anche ad unità distinte, ma a costo 0. Lo stesso vale per altre unità che sono necessarie al corretto funzionamento del modello, e che definiremo "ausiliarie".

Secondo la logica seguita ogni unità interna a BasicNet è stata considerata a costo 1, mentre ogni unità esterna a costo 0.

Una possibile modifica che potrà essere apportata al modello è considerare alcuni licenziatari (Kappa Italia per esempio) come se fossero interni, e quindi attribuirgli un costo. Questa ipotesi equivarrebbe a presumere una maggiore assunzione di rischio da parte di BasicNet, che potrebbe ripercuotersi in modi imprevisi sugli utili dell'azienda.

Sono state utilizzate esclusivamente unità "semplici" in grado di eseguire una sola

lavorazione. Il file contenente le informazioni sulle unità è *unitData/unitBasicData.txt*.

unit_#	_useWarehouse	prod.	phase_#	_fixed_costs	variable_costs
1	0		1	0	0
2	0		2	0	0
3	0		3	0	0
4	0		4	0	0
5	0		5	0	0
6	0		6	0	0
7	0		7	0	0
8	0		8	0	0
9	0		9	0	0
10	0		10	0	0
11	0		11	0	0
12	0		12	0	0
13	0		13	0	0
14	0		14	0	0
15	0		15	0	0
16	0		16	0	0
17	0		17	0	0
18	0		18	0	0
19	0		19	0	0
20	0		20	0	0
21	0		21	0	0
22	0		22	0	0
23	0		23	0	0
24	0		24	0	0
25	0		25	0	0
26	0		26	0	0
27	0		27	0	0

28	0	28	0	0
29	0	29	0	0
30	0	30	0	0
31	0	31	0	0
32	0	32	0	0
33	0	33	0	0
34	0	34	0	0
35	0	35	0	0
36	0	36	0	0
37	0	37	0	0
38	0	38	0	0
39	0	39	0	0
40	0	40	0	0
41	0	100	1	1
42	0	101	1	1
43	0	102	1	1
44	0	103	1	1
45	0	104	1	1
46	0	105	1	1
47	0	106	1	1

Le colonne devono essere interpretate in questo modo:

1. La prima colonna indica i codici delle unità. Le unità hanno tutte codice positivo, sono quindi tutte *layer sensitive*.
 - (a) Le unità da 1 a 35 rappresentano i licenziatari.
 - (b) Le unità da 36 a 40 rappresentano le Trading Company
 - (c) Le unità da 41 a 47 rappresentano le divisioni dotcom di BasicNet
2. La seconda colonna indica l'utilizzo dei magazzini. Nel nostro modello non vogliamo usare i magazzini, questo significa che le unità, quando non occupate da qualche

ordine, saranno inattive.

3. La terza colonna indica il passo che ogni unità è in grado di compiere:
 - (a) I passi da 1 a 35 corrispondono alle operazioni svolte dai licenziatari (fare le previsioni, fare gli ordini e restituire feedback telefonici sul mercato di riferimento).
 - (b) I passi da 36 a 40 corrispondono alle operazioni svolte dalle Trading Company (produrre i campionari, produrre gli articoli, partecipare all'asta offrendo un prezzo di produzione).
 - (c) I passi da 100 a 106 corrispondono alle operazioni svolte dalle divisioni dotcom di BasicNet.
4. La quarta colonna indica i costi fissi. Sono stati impostati a 0 per gli agenti esterni a BasicNet e a 1 per gli agenti BasicNet. Il valore 1 è solo indicativo poiché non disponiamo di informazioni sufficienti per valutare i costi reali delle unità.
5. La quinta colonna indica i costi variabili. Sono stati impostati, come i costi fissi, a 0 per gli agenti esterni a BasicNet e a 1 per gli agenti BasicNet.

Le divisioni BasicNet nel dettaglio sono:

Unità 41: Organizzazione Generale BasicNet, questa unità non è una reale divisione dotcom, ma è utile nella simulazione per contabilizzare i costi strutturali. Esegue la lavorazione 100 che corrisponde alle attività amministrative che permettono l'esistenza della BasicNet. L'unità esegue anche operazioni "ausiliarie" al modello come l'interruzione della simulazione al termine dell'orderSequence.

Unità 42: BasicSamples. Esegue la lavorazione 101 che corrisponde al disegno dei meta campionari e alla richiesta di feedback ai licenziatari.

Unità 43: BasicForecast. Esegue la lavorazione 102, che corrisponde alla raccolta e elaborazione delle previsioni fatte dai licenziatari.

Unità 44: BasicSpecs. Esegue la lavorazione 103, che corrisponde alla definizione delle specifiche tecniche dei prodotti.

Unità 45: BasicSamplesProduction, questa unità non è una reale divisione dotcom. Esegue la lavorazione 104, che corrisponde alla produzione dei campionari. Questa unità

è stata creata perché la produzione dei campionari non segue la stessa procedura dell'asta usata nella produzione degli articoli. Generalmente questo tipo di produzione viene affidata ad un'impresa italiana di fiducia.

Unità 46: BasicBidding. Esegue la lavorazione 105, che corrisponde all'asta per l'assegnazione della produzione. Viene scelta la Trading Company che offre il prezzo di produzione inferiore.

Unità 47: BasicFactory. Esegue la lavorazione 106, che corrisponde all'operazione di coordinamento delle trading Company alle quali è affidata la produzione.

Le End Unit

Nel modello sono presenti due endUnit, entrambe *layer sensitive* (e quindi indicate con codice positivo). Le endUnit sono descritte nel file *unitData/endUnitList.txt*

Le endUnit vengono utilizzate per i seguenti scopi:

EndUnit 1001: luogo dove vengono depositati i campionari prodotti. Viene fatto un procurement su questa endUnit per consegnarli ai licenziatari.

EndUnit 1002: luogo dove vengono depositati gli articoli prodotti. Viene fatto un procurement su questa endUnit per consegnarli ai licenziatari.

6.4.3 Il modello: le ricette

Per poter costruire le ricette della simulazione in modo realistico abbiamo utilizzato un Data Base contenente le informazioni relative agli ordini, consegne e tempistiche della collezione *Primavera - Estate 2002*. Da queste informazioni abbiamo estrapolato alcuni dati utili alla simulazione, riportati di seguito.

Analisi del calendario

La tabella 6.3 riporta l'ordine temporale e la durata stimata delle principali attività organizzative svolte da BasicNet. I valori indicati nella colonna "Tempo unitario" sono il risultato della formula:

$\text{TempoUnitario} = \text{OreLavorative} / 300$

dove 300 è il numero di prodotti appartenenti alla collezione.

Fase organizzativa	Codice ricetta	Da giorno	A giorno	Giorni lavoro	Ore lavoro	Tempo unitario
Disegno collezione	100	1	50	50	400	1,3
Previsione licenziatari	101-135	51	60	10	80	0,3
Produzione Campionario	150	61	70	10	80	0,3
Specifiche Tecniche	150	105	140	36	288	1,0
Consegna campionari	160	140	145	6	48	0,2
Offerta prezzi produzione	171-175	150	155	6	48	0,2

Tabella 6.3. Calendario stimato per le principali attività organizzative

Analisi dei prezzi

La tabella 6.4 riporta il calcolo statistico della media e della varianza sui prezzi di tutti gli articoli della collezione *Primavera - Estate 2002*. Questi valori sono utili per costruire in modo realistico l'oggetto computazionale C1905.

Prezzo articoli in \$			
min	max	media	varianza
1,05	19,5	6,847836795	10,33953227

Tabella 6.4. Analisi del prezzo degli articoli

Analisi degli ordini totali

La tabella 6.5 riporta il numero totale di prodotti ordinati da ogni licenziatario e la media e varianza delle quantità ordinate di ogni articolo.

Analisi degli ordini di una collezione grande

La tabella 6.6 riporta le stesse informazioni della tabella 6.5 ma solo in riferimento ad una collezione grande composta da 300 articoli.

Licenziatari	Tutte le collezioni				
	Ordini	Min	Max	Media	Varianza
Lic. 1	3056	100	648	161	16140
Lic. 2	72955	30	1910	182	29312
Lic. 3	7580	60	1780	541	225336
Lic. 4	4050	180	240	193	261
Lic. 5	840	108	192	140	1651
Lic. 6	12000	1200	4800	2400	2160000
Lic. 7	10380	50	600	346	15494
Lic. 8	209910	10	3820	721	507597
Lic. 9	3670	130	500	282	12953
Lic. 10	24820	40	2310	335	186501
Lic. 11	35814	100	840	201	23136
Lic. 12	2514	170	700	419	61134
Lic. 13	50810	70	2150	249	87470
Lic. 14	3660	40	1300	229	98958
Lic. 15	123025	20	6000	439	196425
Lic. 16	2494570	10	20000	614	931550
Lic. 17	198425	10	2200	418	138383
Lic. 18	24743	200	3000	515	217546
Lic. 19	40103	200	3500	528	304002
Lic. 20	6545	50	210	101	623
Lic. 21	476505	20	25100	529	3635987
Lic. 22	73518	60	1330	214	14032
Lic. 23	77116	15	2605	365	182357
Lic. 24	31896	30	800	175	15720
Lic. 25	32475	100	1525	373	68394
Lic. 26	117339	45	2255	371	93677
Lic. 27	12170	50	420	206	3272
Lic. 28	17368	20	550	134	12241
TOTALE	4167857	10	25100	489	911615

Tabella 6.5. Analisi degli ordini dei singoli licenziatari in un semestre

Licenziatari	Collezioni grandi				
	Ordini	Min	Max	Media	Varianza
Lic. 1	2120	100	180	125	926
Lic. 2	72895	30	1910	183	29348
Lic. 3	7080	60	1780	590	243636
Lic. 4	4050	180	240	193	261
Lic. 5					
Lic. 6					
Lic. 7	3900	300	600	355	8247
Lic. 8	105940	10	2920	555	328597
Lic. 9	1500	240	260	250	120
Lic. 10	24430	40	2310	354	194935
Lic. 11	29310	100	640	174	10818
Lic. 12					
Lic. 13	28610	100	2150	270	121887
Lic. 14	420	70	140	105	1233
Lic. 15	48910	20	1100	312	23013
Lic. 16	1709660	20	20000	612	1129950
Lic. 17	157795	10	2200	421	141386
Lic. 18	2800	300	600	350	11429
Lic. 19	2800	300	600	350	11429
Lic. 20	4055	80	115	99	34
Lic. 21	226770	20	10390	315	341421
Lic. 22	55735	60	1330	212	12359
Lic. 23	40470	35	965	247	47591
Lic. 24	16010	30	600	184	16662
Lic. 25	30915	100	1525	364	64124
Lic. 26	115989	45	2255	374	95012
Lic. 27	9030	210	210	210	0
Lic. 28	14070	20	500	117	7490
TOTALE	2715264	10	20000	439	615621

Tabella 6.6. Analisi degli ordini dei singoli licenziatari su una collezione grande in un semestre

Analisi degli ordini di una collezione piccola

La tabella 6.7 riporta le stesse informazioni della tabella 6.5 ma solo in riferimento ad una collezione SMU (piccola composta da 30 articoli).

Licenziatari	Collezioni piccole (SMU)				
	Ordini	Min	Max	Media	Varianza
Lic. 1	936	288	648	468	64800
Lic. 2	60	60	60	60	
Lic. 3	500	90	410	250	51200
Lic. 4					
Lic. 5	840	108	192	140	1651
Lic. 6	12000	1200	4800	2400	2160000
Lic. 7	6480	50	480	341	20310
Lic. 8	103970	50	3820	1040	700282
Lic. 9	2170	130	500	310	23867
Lic. 10	390	40	100	78	920
Lic. 11	6504	468	840	650	16730
Lic. 12	2514	170	700	419	61134
Lic. 13	22200	70	1030	227	50128
Lic. 14	3240	40	1300	270	127182
Lic. 15	74115	30	6000	603	371893
Lic. 16	784910	10	6550	618	495434
Lic. 17	40630	20	1645	406	128267
Lic. 18	21943	200	3000	549	253379
Lic. 19	37303	200	3500	549	334894
Lic. 20	2490	50	210	104	1659
Lic. 21	249735	20	25100	1372	15829021
Lic. 22	17783	90	780	222	19673
Lic. 23	36646	15	2605	780	438309
Lic. 24	15886	100	800	167	14890
Lic. 25	1560	500	1060	780	156800
Lic. 26	1350	100	250	225	3750
Lic. 27	3140	50	420	196	12505
Lic. 28	3298	72	550	330	30078
TOTALE	1452593	10	25100	622	1671003

Tabella 6.7. Analisi degli ordini dei singoli licenziatari su una collezione piccola in un semestre

Analisi delle collezioni Kappa

La tabella 6.8 riporta il dettaglio del numero articoli e delle quantità ordinate per le collezioni del brand "Kappa" in un semestre.

Collezione	Articoli	Ordini
Collezione 1	241	1377429
Collezione 2	54	296260
Collezione 3	36	140561
Collezione 4	28	51300
Collezione 5	26	59938
Collezione 6	20	180350
Collezione 7	17	216355
Collezione 8	16	24810
Collezione 9	16	17965
Collezione 10	14	34790
Collezione 11	12	7150
Collezione 12	11	65615
Collezione 13	9	8160
Collezione 14	9	12420
Collezione 15	7	21535
Collezione 16	7	7020
Collezione 17	6	27225
Collezione 18	6	2200
Collezione 19	5	13060
Collezione 20	5	6800
Collezione 21	5	23670
Collezione 22	5	17240
Collezione 23	5	26772
Collezione 24	3	2292
Collezione 25	3	900
Collezione 26	2	21680
Collezione 27	2	1900
Collezione 28	2	364
Collezione 29	2	200
Collezione 30	1	140

Collezione 31	1	500
Collezione 32	1	1560
Collezione 33	1	580
Collezione 34	1	1200
Collezione 35	1	760
Collezione 36	1	5256
Collezione 37	1	3035
Collezione 38	1	11750
Collezione 39	1	1920
TOTALE	584	2692662

Tabella 6.8: Numero articoli e quantità ordinate per le collezioni Kappa in un semestre

Analisi delle collezioni Robe di Kappa

La tabella 6.9 riporta il dettaglio del numero degli articoli e delle quantità ordinate per le collezioni del brand "Robe di Kappa" in un semestre.

Collezione	Articoli	Ordini
Collezione 1	144	1337835
Collezione 2	26	127885
Collezione 3	9	5725
Collezione 4	2	3550
Collezione 5	1	200
TOTALE	182	1475195

Tabella 6.9. Numero articoli e quantità ordinate per le collezioni Robe di Kappa in un semestre

Analisi dei tempi di produzione degli articoli

La tabella 6.10 riporta i valori statistici di media e varianza per i tempi di produzione di ogni singolo articolo e di ogni singolo ordine.

Durata Produzione in gg				
	min	max	media	varianza
Ordine	20	182	116,4332041	333,1870478
Unitario	0,004	12,100	0,598	0,381

Tabella 6.10. Stime dei tempi di produzione per ordine e singolo articolo

Analisi dei tempi di produzione divisi per Trading Company

La tabella 6.11 riporta i valori statistici di media e varianza per i tempi di produzione di ogni singolo articolo e di ogni singolo ordine divisi per Trading Company.

Durata Produzione per SC in gg					
SC		min	max	media	varianza
1	Ordine	36	52	47	56,767
	Unitario	0,015	0,150	0,071	0,003
2	Ordine	24	182	118	243,231
	Unitario	0,004	12,100	0,602	0,386
3	Ordine	20	115	57	566,899
	Unitario	0,014	1,875	0,406	0,119

Tabella 6.11. Stime dei tempi di produzione di ogni Trading Company per ordine e singolo articolo

Le ricette utilizzate

Le ricette descrivono le operazioni da compiere e gli oggetti computazionali da utilizzare per eseguire ogni processo organizzativo di BasicNet. Le ricette sono contenute nel file *./recipeData/recipes.xls*.

La tabella 6.12 riporta le ricette contenenti i passi necessari per ogni processo organizzativo in notazione "esterna". I tempi sono indicati con la lettera "s" anche se rappresentano nella nostra simulazione il quarto d'ora.

1 s = 15 minuti

Questa scelta è stata necessaria per mantenere la compatibilità con l'applicazione in VIR.

- La ricetta 10 descrive l'analisi delle tendenze del mercato, necessaria prima di iniziare a disegnare una nuova collezione.
- Le ricette da 21 a 55 rappresentano la richiesta di informazioni sulle condizioni del mercato da parte di BasicNet ai licenziatari. Concretamente consistono in telefonate ad ogni singolo licenziatario.
- La ricetta 100 rappresenta il disegno di un MetaSample da aprte di BasicSamples.

- Le ricette da 101 a 135 rappresentano la richiesta di previsioni di acquisto ai licenziatari per ogni articolo. Questi valori vengono generati in modo casuale e salvati in una MemoryMatrix.
- La ricetta 140 si occupa di sommare le previsioni e decidere, in base al valore ottenuto, se il metacampionario dovrà essere realizzato.
- La ricetta 150 crea le specifiche tecniche dei campionari da realizzare e li produce.
- La ricetta 160 esegue un procurement sui campionari realizzati e li consegna ai licenziatari.
- Le ricette da 171 a 175 rappresentano l'offerta del prezzo di produzione da parte di ogni Trading Company durante l'asta.
- La ricetta 180 raccoglie i prezzi offerti e sceglie la Trading Company più conveniente.
- Le ricette da 201 a 235 rappresentano gli ordini dei licenziatari per ogni articolo.
- La ricetta 250 si occupa di produrre gli articoli ordinati.
- La ricetta 260 si occupa di consegnare gli articoli prodotti utilizzando un procurement.
- La ricetta 1 serve per interrompere la simulazione alla fine del quarto anno³.

6.4.4 Il modello: le matrici di memoria

Le dimensioni di ogni matrice vengono dichiarate all'interno del file *unitData/memoryMatrixes.txt*; per la nostra simulazione è stato preparato nel modo seguente:

```
number(from_0_ordered;_negative_if_insensitive_to_layers)_rows_cols
0      1      1
1      3     301
2      3     301
3      3     301
4      3     301
```

³Si è deciso di simulare tre anni di gestione, e lasciare girare il programma per un ulteriore anno, nel quale smaltire le code formatesi

Nome ricetta	Codice	Passi									
analisi tendenze	10	101	s	32	;						
feedbacklicenziatario1	21	101	s	1	1	s	1	;			
feedbacklicenziatario2	22	101	s	1	2	s	1	;			
...			
feedbacklicenziatario35	55	101	s	1	35	s	1	;			
disegnometacampionario	100	c	1901	1	0	101	s	5	;		
previsionilicenziatario1	101	c	1902	2	0	1	1	s	1	;	
...
previsionilicenziatario2	102	c	1902	2	0	2	2	s	1	;	
previsionilicenziatario35	135	c	1902	2	0	35	35	s	1	;	
sommaprevisioni	140	c	1903	37	0	1	2	3	4	5	6
			7	8	9	10	11	12	13	14	15
			17	18	19	20	21	22	23	24	25
			27	28	29	30	31	32	33	34	35
								102	s	64	;
speceprod campionario	150	c	1904	2	0	41	103	s	4		
						104	s	1	e	1001	;
consegnacampionario	160	p	1	1001	c	1911	2	0	41		
		104	s	1	;						
offertatc1	171	c	1905	3	0	41	36	36	s	300	;
offertatc2	172	c	1905	3	0	41	37	37	s	300	;
...
offertatc5	175	c	1905	3	0	41	40	40	s	300	;
sceltatc	180	c	1906	8	0	36	37	38			
				39	40	41	42	105	s	20	;
ordinilicenziatario1	201	c	1907	3	0	1	41	1	s	9	;
ordinilicenziatario2	202	c	1907	3	0	2	41	2	s	9	;
...
ordinilicenziatario35	235	c	1907	3	0	35	41	35	s	9	;
produzione prodotto	250	c	1908	3	0	41	42	106	s	1	
			1001	36	s	9		2	37	s	9
			3	38	s	9		4	39	s	9
			5	40	s	9		0	##	s	1
								e	1002	;	
consegnaprodotto	260	p	1	1002	c	1912	2	0	41		
								106	s	1	;
theEnd	1	c	1910	1	0	100	s	0	;		

Tabella 6.12. Le ricette utilizzate nella simulazione

5 3 301
6 3 301
7 3 301
8 3 301
9 3 301
10 3 301
11 3 301
12 3 301
13 3 301
14 3 301
15 3 301
16 3 301
17 3 301
18 3 301
19 3 301
20 3 301
21 3 301
22 3 301
23 3 301
24 3 301
25 3 301
26 3 301
27 3 301
28 3 301
29 3 301
30 3 301
31 3 301
32 3 301
33 3 301

34 3 301
 35 3 301
 36 3 301
 37 3 301
 38 3 301
 39 3 301
 40 3 301
 41 6 303
 42 2 301

Riportiamo qui lo schema di matrici utilizzato nella simulazione.

designMatrix

La tabella 6.13 riporta il contenuto della MemoryMatrix 0, denominata designMatrix. In posizione (0,0) è indicato il numero di MetaSample realizzati. Questo valore è fondamentale perché verrà utilizzato da tutti gli oggetti computazionali per sapere su quanti articoli/colonne eseguire le diverse operazioni.

Matrice 0	
designMatrix	0
0	Numero di metacampionari disegnati

Tabella 6.13. Matrice di memoria *designMatrix*

agentMatrix

La tabella 6.14 riporta il contenuto delle MemoryMatrix da 1 a 35, denominate agentMatrix. Nella colonna 0 vengono creati i contatori⁴. Nella riga 1 vengono salvate le previsioni del licenziatario per ogni articolo (un articolo per colonna). Nella riga 2 vengono salvati gli ordini del licenziatario con lo stesso meccanismo.

⁴Per il funzionamento dei contatori si veda a pagina 182

Matrici 1 - 35					
agent #Matrix	0	1	2	...	300
0					
1	cont.	<i>previsioni art.1</i>	<i>previsioni art.2</i>	...	<i>previsioni art.300</i>
2	cont.	<i>ordini art.1</i>	<i>ordini art.2</i>	...	<i>ordini art.300</i>

Tabella 6.14. Matrice di memoria *agentMatrix*

tcMatrix

La tabella 6.15 riporta il contenuto delle MemoryMatrix da 36 a 40, denominate tcMatrix. Nella colonna 0 viene creato il contatore. Nella riga 1 vengono salvate i prezzi offerti dalla Trading company per ogni articolo.

Matrici 36 - 40					
tc#Matrix	0	1	2	...	300
0					
1		prezzo offerto art.1	prezzo offerto art.2	...	prezzo offerto articolo 2

Tabella 6.15. Matrice di memoria *tcMatrix***basicNetMatrix**

La tabella 6.16 riporta il contenuto della MemoryMatrix 41, denominata basicNetMatrix. Questa matrice rappresenta le informazioni possedute da BasicNet. Nella colonna 0 vengono creati i contatori. Nella riga 1 viene salvata la somma dei valori previsti dai licenziatari, utile per decidere, confrontando il valore con la soglia, se il campionario dovrà essere prodotto. Nella riga 2 viene indicato lo "status" di ogni articolo secondo la convenzione:

1. Non ha superato la soglia, da non realizzare.
2. Ha superato la soglia, da realizzare.
3. Campionario realizzato
4. Articolo realizzato

Nella riga 3 è presente il prezzo di produzione di ogni articolo da realizzare, deciso in fase di asta. Nella riga 4 è presente la somma degli ordini di tutti i licenziatari per ogni singolo articolo.

Matrice 41					
basicNetMatrix	0	1	2	...	300
0					
1		somma previsioni art.1	somma previsioni art.2	...	somma previsioni art.300
2		status art.1	status art.2	...	status art.300
3		prezzo prod. art.1	prezzo prod. art.2	...	prezzo prod. art.300
4	cont.	somma ordini art.1	somma ordini art.2	...	somma ordini art.300

Tabella 6.16. Matrice di memoria *basicNetMatrix*

orMatrix

La tabella 6.17 riporta il contenuto della MemoryMatrix 42, denominata orMatrix. Questa matrice è utilizzata internamente da jES per scegliere il "branch" (il ramo) dei passi *or* da eseguire. Il meccanismo di jES utilizza la colonna 0 per scegliere il "branch". Nelle colonne da 1 a 300 sono riportate le scelte per ogni articolo. L'oggetto computazionale C1908 copia, nel momento giusto, il valore relativo alla scelta per l'articolo in produzione nella colonna 0, così che possa essere utilizzato da jES per scegliere il ramo giusto.

Matrice 42					
orMatrix	0	1	2	...	300
0	contatore produzione				
1	ramo OR da utilizzare	<i>TC produttrice art.1</i>	<i>TC produttrice art.2</i>	...	<i>TC produttrice art.300</i>

Tabella 6.17. Matrice di memoria *orMatrix*

6.4.5 Il modello: la sequenza degli ordini

Ordini di una collezione grande

La tabella 6.18 riporta un estratto del file *recipeData/orderSequences.xls* che gestisce il lancio degli ordini scanditi nel tempo per una collezione grande di 300 articoli⁵. Ogni riga corrisponde ad uno "shift" della simulazione, che nell'applicazione BasicNet è pari a 32 tick, cioè un giorno lavorativo.

La grammatica dell'*orderSequences* è:

```
shiftN l ln recipe * quantity
```

Dove:

shiftN è il numero dello "shift" in cui verrà lanciato l'ordine.

ln è il numero del layer a cui appartiene l'ordine.

recipe è il numero della ricetta lanciata dall'ordine.

quantity è il numero di ricette che vengono lanciate contemporaneamente dall'ordine.

La nostra simulazione utilizza 1200 "shift", corrispondenti a 4 anni reali, e 101 layer in corrispondenza delle collezioni.

⁵Si riferisce alla collezione "Robe di Kappa" Primavera-Estate 2002, corrispondente al layer 1

Shift	Layer		Ordini														
			10	*	1												
51	1	1															
...																	
66	1	1	21	*	1	22	*	1	23	*	1	24	*	1	25	*	1
67	1	1	26	*	1	27	*	1	28	*	1	29	*	1	30	*	1
68	1	1	31	*	1	32	*	1	33	*	1	34	*	1	35	*	1
69	1	1	36	*	1	37	*	1	38	*	1	39	*	1	40	*	1
70	1	1	41	*	1	42	*	1	43	*	1	44	*	1	45	*	1
71	1	1	46	*	1	47	*	1	48	*	1	49	*	1	50	*	1
72	1	1	51	*	1	52	*	1	53	*	1	54	*	1	55	*	1
...																	
76	1	1	100	*	300												
...																	
126	1	1	101	*	300	102	*	300	103	*	300	104	*	300			
127	1	1	111	*	300	112	*	300	113	*	300	114	*	300			
128	1	1	121	*	300	122	*	300	123	*	300	124	*	300			
129	1	1	105	*	300	106	*	300	107	*	300	108	*	300			
130	1	1	115	*	300	116	*	300	117	*	300	118	*	300			
131	1	1	125	*	300	126	*	300	127	*	300	128	*	300			
132	1	1	109	*	300	110	*	300	133	*	300	134	*	300			
133	1	1	119	*	300	120	*	300	135	*	300	132	*	300			
134	1	1	129	*	300	130	*	300	131	*	300						
...																	
149	1	1		140	*	1											
...																	
156	1	1		150	*	300											
...																	
165	1	1		171	*	1	172	*	1	173	*	1	174	*	1	175	*
...																	
175	1	1		160	*	300											
...																	
186	1	1	180	*	1												
...																	
214	1	1	201	*	300	202	*	300	203	*	300	204	*	300	205	*	300
215	1	1	206	*	300	207	*	300	208	*	300	209	*	300	210	*	300
216	1	1	211	*	300	212	*	300	213	*	300	214	*	300	215	*	300
217	1	1	216	*	300	217	*	300	218	*	300	219	*	300	220	*	300
218	1	1	221	*	300	222	*	300	223	*	300	224	*	300	225	*	300
219	1	1	226	*	300	227	*	300	228	*	300	229	*	300	230	*	300
220	1	1	231	*	300	232	*	300	233	*	300	234	*	300	235	*	300
...																	
251	1	1	250	*	300												
...																	
305	1	1		260	*	300											

Tabella 6.18. Esempio di sequenza degli ordini per una collezione di 300 articoli

Ordini di una collezione piccola

La tabella 6.19 riporta un estratto del file *recipeData/orderSequences.xls* che gestisce il lancio degli ordini scanditi nel tempo per una collezione SMU di 30 articoli⁶.

Shift	Layer	Ordini																	
1	1	7	10	*	1														
...																			
20	1	7	100	*	30														
...																			
45	1	7	101	*	30	102	*	30	103	*	30	104	*	30					
46	1	7	111	*	30	112	*	30	113	*	30	114	*	30					
47	1	7	121	*	30	122	*	30	123	*	30	124	*	30					
48	1	7	105	*	30	106	*	30	107	*	30	108	*	30					
49	1	7	115	*	30	116	*	30	117	*	30	118	*	30					
50	1	7	125	*	30	126	*	30	127	*	30	128	*	30					
...																			
51	1	7	109	*	30	110	*	30	133	*	30	134	*	30					
52	1	7	119	*	30	120	*	30	135	*	30	132	*	30					
53	1	7	129	*	30	130	*	30	131	*	30								
...																			
64	1	7		140	*	1													
...																			
72	1	7		150	*	30													
...																			
76	1	7		171	*	1	172	*	1	173	*	1	174	*	1	175	*	1	
...																			
87	1	7		160	*	30													
...																			
99	1	7	180	*	1														
...																			
101	1	7	201	*	30	202	*	30	203	*	30	204	*	30	205	*	30		
102	1	7	206	*	30	207	*	30	208	*	30	209	*	30	210	*	30		
103	1	7	211	*	30	212	*	30	213	*	30	214	*	30	215	*	30		
104	1	7	216	*	30	217	*	30	218	*	30	219	*	30	220	*	30		
105	1	7	221	*	30	222	*	30	223	*	30	224	*	30	225	*	30		
106	1	7	226	*	30	227	*	30	228	*	30	229	*	30	230	*	30		
107	1	7	231	*	30	232	*	30	233	*	30	234	*	30	235	*	30		
...																			
125	1	7	250	*	30														
...																			
150	1	7		260	*	30													

Tabella 6.19. Esempio di sequenza degli ordini per una collezione di 30 articoli

6.4.6 Il modello: i parametri della simulazione

I parametri utilizzati per la simulazione sono descritti nel file *jveframe.scm* riportato di seguito:

```
(list
 (
  cons 'vEFrameObserverSwarm
  uuuuuuuu(
  u make - instance u 'VEFrameObserverSwarm
    #:displayFrequency 1
    #:verboseChoice #f
```

⁶Si riferisce alla collezione "Robe di Kappa" Speciale Scuola 2002, corrispondente al layer 7

```

        #:printMatrixes      #f
        #:checkMemorySize   #f
        #:unitHistogramXPos  10
        #:unitHistogramYPos  300
        #:endUnitHistogramXPos 10
        #:endUnitHistogramYPos 250
    )
)
(
    cons 'vEFrameModelSwarm
    uuuuuuu(
    make - instance 'VEFrameModelSwarm
        #:useOrderDistiller #t
        #:ticksInATimeUnit  32
        #:totalUnitNumber    47
        #:totalEndUnitNumber  2
        #:totalLayerNumber   101
        #:totalMemoryMatrixNumber 43
            #:sameUnitLifoAssignment #t
        #:maxTickInAUnit      900
        #:useWarehouses       #f
        #:useNewses           #f
        #:orCriterion         5
        #:orMemoryMatrix      42
    )
)
)
)

```

- **useOrderDistiller=true** gli ordini e le ricette vengono letti dai file *orderSequences.xls* e *recipes.xls* utilizzando la classe *OrderDistiller.java*.
- **ticksInATimeUnit=32** ogni giorno della simulazione è composto da 32 tick pari a 15 minuti ognuno. Il giorno lavorativo è considerato di 8 ore.
- **totalUnitNumber=47** il numero delle unità è 47.
- **totalEndUnitNumber=2** il numero delle endUnit è 2.
- **totalLayerNumber=101** il numero dei layer, corrispondenti alle collezioni, è 101.
- **totalMemoryMatrixNumber=43** il numero delle memoryMatrix è 43.
- **sameUnitLifoAssignment=true** la produzione all'interno di ogni unità avviene con procedura LIFO per ogni prodotto. La produzione risulta quindi sequenziale.
- **maxTickInAUnit=900** dopo che un prodotto rimane fermo per 900 tick all'interno della stessa unità viene eliminato. Questo meccanismo è utile per non appesantire

troppo la simulazione con prodotti che non superano la soglia, e che quindi rimangono bloccati dagli oggetti computazionali.

- **useWarehouses=false** non devono essere utilizzati i magazzini delle unità.
- **useNewses=false** non devono essere propagata l'informazione alle unità attraverso il meccanismo delle news.
- **orCriterion=5** il criterio di scelta dei rami del passo *or* avviene attraverso una MemoryMatrix.
- **orMemoryMatrix=42** la MemoryMatrix utilizzata per la scelta dell passo *or* è la 42.

6.5 Le modifiche al codice

Le modifiche al codice riguardano 3 files: *orderDistiller.java*, *recipes.java* e *ComputationalAssembler.java*⁷.

6.5.1 Modifiche a "OrderDistiller.java"

Le modifiche al codice sono state finalizzate alla gestione dei layers e degli oggetti computazionali. La sintassi per indicare all'interno di *orderSequence.xls* il layer di riferimento di ogni ricetta è:

```
... l x ...
```

dove *x* indica il layer al quale appartengono tutte le ricette lanciate dal distiller da quel momento fino a quando non verrà impostato un nuovo layer⁸. Nel caso non venga specificato alcun layer le ricette vengono considerate appartenenti al layer 0.

La sintassi per indicare gli oggetti computazionali all'interno di *recipes.xls* è:

```
... c x n m1 m2 ... mn ...
```

⁷Le seguenti modifiche risalgono al nomebre 2001. La versione aggiornata dei file è disponibile in appendice

⁸Se il foglio excel *orderSequences.xls* arrivasse alla fine, la lettura ripartirebbe dalla prima ricetta appartenente all'ultimo layer impostato

dove c x indica l'oggetto computazionale di tipo x . Il numero di matrici che dovrà gestire per le computazioni è pari a n , e le matrici sono indicate da $m1$ $m2$... mn .

Dal punto di vista informatico è stato necessario modificare il file *OrderDistiller.java* come segue⁹.

Listing 6.1. OrderDistiller.java: modifica 1

```
public class OrderDistiller extends OrderGenerator{

    public boolean
    worksheetOrderSequenceFileOpen = false ,
    worksheetRecipeFileOpen = false ;
    int [] orderSequence1 , orderSequence2 , orderSequence3;
```

È stato aggiunto un array di tipo *integer* denominato *orderSequence3* che contiene le indicazioni del layer di riferimento per ogni ricetta. Il vettore è azzerato all'inizio di ogni "shift"¹⁰ della simulazione. I vettori *orderSequence1* e *orderSequence2* contengono rispettivamente il numero della ricetta e la quantità che deve essere lanciata in produzione per ogni "shift".

Listing 6.2. OrderDistiller.java: modifica 2

```
public String semicolon = ";" , checkTheCell , gate = "#" , p = "p" ,
    sec = "s" , min = "m" , end = "e" , slash = "/" , backslash = "\\",
    _or_ = " || " , _u_layer_ = "l" , _u_computation_ = "c";
```

È stata aggiunta la variabile stringa *layer* contenente la lettera "l", il segno distintivo dell'impostazione dei layers all'interno di *orderSequences.xls*.

Listing 6.3. OrderDistiller.java: modifica 3

```
public static boolean firstTime = true;
```

⁹Per il listato completo del codice si veda l'appendice

¹⁰Ogni shift, o turno della simulazione, è rappresentato da una riga del file *orderSequences.xls*

E' stata inizializzata la variabile globale *firstTime = true* che prima era presente solo nel metodo *distill*. In questo modo è impostata come *true* solo all'avvio della simulazione, e non ogni volta che si invoca il metodo *distill*. Questa modifica è necessaria per il corretto funzionamento e la compatibilità con l'applicazione jES in VIR.

Listing 6.4. OrderDistiller.java: modifica 4

```
int currentLayer = 0;
int maxOrderSequence;
```

Sono state aggiunte le variabili globali necessarie alla lettura dei layer e al dimensionamento degli array.

Listing 6.5. OrderDistiller.java: modifica 5

```
public void setDictionary() {
    super.setDictionary();

    maxOrderSequence = dictionaryLength * totalLayerNumber;
    orderSequence1 = new int [maxOrderSequence];
    orderSequence2 = new int [maxOrderSequence];
    orderSequence3 = new int [maxOrderSequence];
    setRecipeContainers();
    readRecipes();
}
```

Il metodo *setDictionary()* si occupa di raccogliere i nomi di *unit* e *endUnit*, per eseguire un controllo in sede di lettura delle ricette. Abbiamo inizializzato il vettore *orderSequence3 = new int[maxOrderSequence]* di lunghezza pari a *maxOrderSequence*.

Listing 6.6. OrderDistiller.java: modifica 5

```
public void computation(ExcelReader e){
    int numberOfMatrixesForComputation = 0;
    int computationCode;

    // The computational code
    e.getIntValue();
```

```

numberOfMatrixesForComputation = e.getIntValue();
length += (3 + numberOfMatrixesForComputation);

// Skipping the Matrixes
for (int h = 0; h < numberOfMatrixesForComputation; h++)
    checkTheCell = e.getStrValue();
    checkTheCell = e.getStrValue();
}

```

Abbiamo aggiunto il nuovo metodo *computation(ExcelReader e)* dove *e* è il file di tipo *ExcelReader* che deve essere letto (nel nostro caso *recipes.xls*). Il metodo è richiamato quando nella lettura delle ricette si incontra il codice *c*, cioè un oggetto computazionale. Il metodo trasforma il codice esterno delle ricette in codice intermedio gestibile da jES. I metodi richiamati *getIntValue()* e *getStrValue()* leggono dal file *e* il valore della cella successiva e controllano che sia, rispettivamente, di tipo integer o di tipo string.

Listing 6.7. OrderDistiller.java: modifica 6

```

public void distill(){
int i, ii, iii, k, code, quantity, layerNumber;
...
anOrder.setOrderLayer(layerNumber);
if (StartVEFrame.verbose)
System.out.println("Order_#" + anOrder.getOrderNumber() +
" is generated with Name:_" + anOrder.getRecipeName() +
" and layerNumber_#" + anOrder.getOrderLayer());
}

```

All'interno del metodo *distill()* è fatto il casting delle variabili locali, tra le quali *layerNumber* di tipo integer.

Ogni volta che un ordine è "distillato", cioè tradotto dal formalismo dell'*orderSequences.xls* in codice interno del programma jES, il layer corrispondente è impostato con il comando *anOrder.setOrderLayer(layerNumber)*.

Listing 6.8. OrderDistiller.java: modifica 7

```

public void readOrderSequence(){
int i, numberOfShift;
}

```

```

if(StartVEFrame.verbose)
System.out.println("firstTime_ùis_ù" + firstTime);
i = 0;
if(! worksheetOrderSequenceFileOpen && firstTime){
    orderSequenceWorksheet = new ExcelReader
    ("recipeData/orderSequencesModified.xls");
    worksheetOrderSequenceFileOpen = true;
    firstTime = false;
    if(StartVEFrame.verbose)
    System.out.println("orderSequencesModified.xls_ùhas_ùbeen_ùopen");
}
else if(! worksheetOrderSequenceFileOpen){
    orderSequenceWorksheet = new ExcelReader
    ("recipeData/orderSequences.xls");
    worksheetOrderSequenceFileOpen = true;

    if(StartVEFrame.verbose)
    System.out.println("orderSequences.xls_ùhas_ùbeen_ùopen");
}

```

Dal metodo *readOrderSequence()* è stata tolta l'inizializzazione della variabile *firstTime* ed è stata spostata all'inizio della classe, per evitare che venisse reimpostata come *true* ogni chiamata del metodo.

Listing 6.9. OrderDistiller.java: modifica 8

```

numberOfShift = orderSequenceWorksheet.getIntValue();
if(StartVEFrame.verbose)
System.out.println("The_ùshift_ù#" + numberOfShift + "_ùhas_ùbegun_ù");

for( int ii = 0; ii < maxOrderSequence; ii++){
    orderSequence1[ii] = 0;
    orderSequence2[ii] = 0;
    orderSequence3[ii] = 0;
}

```

Abbiamo corretto l'azzeramento dei vettori *orderSequence1*, *orderSequence2* e *orderSequence3* che deve avvenire al termine di ogni turno.

Listing 6.10. OrderDistiller.java: modifica 9

```

while(! orderSequenceWorksheet.eol()){
    checkForLayer(orderSequenceWorksheet);
}

```

```

orderSequence1[i] = errorIsNotAnInteger(orderSequenceWorksheet);
orderSequence3[i] = currentLayer;

errorIsNotAString(orderSequenceWorksheet);
orderSequenceWorksheet.getStrValue();

orderSequence2[i] = errorIsNotAnInteger(orderSequenceWorksheet);

```

Abbiamo aggiunto un controllo sul cambio del layer nel file *orderSequence.xls*.

Listing 6.11. OrderDistiller.java: modifica 9

```

if (StartVEFrame.verbose)
System.out.println("The recipe #" + orderSequence1[i] + " with quantity " + orderSequence2[i]
+ " and layer " + orderSequence3[i] + " is starting production");

i++;
}

```

Sono state aggiunte alcune stampe sul terminale quando la variabile *verbose* vale *true*.

Listing 6.12. OrderDistiller.java: modifica 10

```

public void checkForLayer(ExcelReader e){

if(e.checkForLabelCell()){
checkTheCell = e.getStrValue();

if(checkTheCell.equals(layer))
currentLayer = e.getIntValue();
if(StartVEFrame.verbose)
System.out.println("current layer now is " + currentLayer);
}
}

```

Il metodo *checkForLayer(ExcelReader e)* ha come argomento un file del tipo *ExcelReader* (nel nostro caso *orderSequence.xls*) e si occupa di controllare se è presente il carattere "l" che indica il cambio di layer degli ordini che verranno lanciati, e legge il nuovo layer di riferimento.

6.5.2 Modifiche a "Recipe.java"

La classe *Recipe.java* è stata creata per trasformare la notazione esterna delle ricette contenuta in *recipe.xls*, in notazione intermedia comprensibile da jES.

Il nostro intervento sulla classe è servito per permettere la trasformazione della notazione degli oggetti computazionali dal codice esterno a quello interno.

Listing 6.13. Recipe.java: modifica 1

```
public String semicolon = ";", checkTheCell, gate = "#", p = "p", end = "e",
    sec = "s", min = "m", slash = "/", backslash = "\\", _or_ = "|",
    computation = "c";
```

Sono state dichiarate le variabili globali della classe. Abbiamo aggiunto *computation="c"*.

Listing 6.14. Recipe.java: modifica 2

```
public void computation(ExcelReader e){
    int numberMatrixesForComputation = 0;
    int computationCode, step;
    int [] matrixesForComputation;

    // Getting the computation code
    computationCode = e.getIntValue();

    // Getting the number of matrixes used for computation
    numberMatrixesForComputation = e.getIntValue();

    // Creating an array with the name of matrixes
    matrixesForComputation = new int [numberMatrixesForComputation];

    // Getting the matrixes
    for(int m = 0; m < numberMatrixesForComputation; m++)
        matrixesForComputation[m] = e.getIntValue();

    // Getting the production step
    step = e.getIntValue();

    // Getting the time unit
    checkTheCell = e.getStrValue();

    // Expanding the production step for intermediate format
    if(checkTheCell.equals(sec))
        second(e, step);
```

```

else if (checkTheCell.equals(min))
    minute(e, step);
else {
    if (StartVEFrame.verbose)
        System.out.println("Time is not expressed in minutes or seconds. Check the worksheet");
    System.exit(1);
}

orderRecipe[++j] = -1 * computationCode;

orderRecipe[++j] = numberMatrixesForComputation;

for (int h = 0; h < numberMatrixesForComputation; h++)
    orderRecipe[++j] = matrixesForComputation[h];

orderRecipe[++j] = 1000000000 + step;

j++;

}

```

Il metodo *computation(ExcelReader e)* riconosce gli oggetti computazionali (indicati con la lettera "c") in *recipes.xls* e trasforma la notazione esterna in notazione intermedia. Il metodo è richiamato ogni qual volta si incontra una "c" nella ricetta.

6.5.3 La classe "ComputationalAssembler.java"

In questa classe abbiamo descritto il funzionamento degli oggetti computazionali necessari per simulare la BasicNet. I metodi della classe possono essere richiamati dalla classe stessa grazie ad una tecnica di java definita *reflection*, il cui funzionamento è inserito in *ComputationalAssemblerBasic.java*.

Listing 6.15. ComputationalAssembler.java: modifica 1

```

import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;
import swarm.collections.ListImpl;
import swarm.collections.ListIndex;
import swarm.random.NormalDistImpl;
import java.lang.Float;

/**
 * The ComputationalAssembler class instances make
 * computational processes; we have a computaciona assembler instance for

```

```

* each unit
*
* @author Pietro Terna
*/
public class ComputationalAssembler extends ComputationalAssemblerBasic {

//Our MemoryMatrix
MemoryMatrix designMatrix , basicNetMatrix , tcMatrix , orMatrix , agentMatrix;
public NormalDistImpl normal;

/**
 * the constructor for ComputationalAssembler
 */

public ComputationalAssembler (Zone aZone)
{
// Call the constructor for the parent class.
super(aZone);

normal = new NormalDistImpl (getZone());

}

```

La classe comincia con l'import delle biblioteche di funzioni di Java e Swarm necessarie. Viene dichiarata la classe `ComputationalAssembler` in modo che estenda ed erediti da `ComputationalAssemblerBasic`. Viene fatto il casting delle variabili della classe

La classe `ComputationalAssembler.java` presenta tre metodi privati utili per svolgere funzioni comuni a tutti gli oggetti computazionali.

checkMatrixes

Tutti gli oggetti computazionali richiamano il metodo privato `checkMatrixes(int code, int numberOfMatrixes)` la cui funzione è di controllare la coerenza tra il numero delle MemoryMatrix presenti nella ricetta e quelle richieste dallo specifico oggetto. Se non c'è congruenza viene stampato un messaggio di errore e il programma viene terminato.

Listing 6.16. `ComputationalAssembler.java: checkMatrixes`

```

private void checkMatrixes(int code, int numberOfMatrixes)
{
if(pendingComputationalSpecificationSet.
getNumberOfMemoryMatrixesToBeUsed() != numberOfMatrixes)
{

```

```

System.out.println("Code_" + code + "_requires_" + numberOfMatrixes
+ "_matrix;" + pendingComputationalSpecificationSet.
getNumberOfMemoryMatrixesToBeUsed() + "_found_in_order#" +
pendingComputationalSpecificationSet.getOrderNumber());

MyExit.exit(1);
}
}

```

getCounter

Il metodo privato *getCounter(MemoryMatrix currentMatrix, int row)* permette di gestire la produzione di articoli diversi utilizzando un ricetta uguale per tutti.

Listing 6.17. ComputationalAssembler.java: getCounter

```

private int getCounter(MemoryMatrix currentMatrix, int row)
{
    //Zeroing the counter if empty
    if(currentMatrix.isEmpty(layer,row,0))
        currentMatrix.setValue(layer,row,0,0.0);

    //Here we increment the counter
    int counter = (int) currentMatrix.getValue(layer,row,0);
    counter++;

    //The counter must be less than the number of MetaSample
    if(counter <= designMatrix.getValue(layer,0,0))
        currentMatrix.setValue(layer,row,0,counter);

    return counter;
}

```

Questo metodo crea un contatore che si incrementa ogni volta che viene richiamato il metodo. Il contatore viene inserito nella posizione (row,0) della MemoryMatrix *currentMatrix*. Viene anche eseguito un controllo sulla congruenza tra il valore del contatore e il numero di metasamples disegnati in quel momento.

logOpen

Il terzo metodo privato *void logOpen(boolean eraseFile)* è utilizzato per la gestione dei file di output contenenti il fatturato di BasicJes.

Listing 6.18. ComputationalAssembler.java: logOpen

```
private void logOpen(boolean eraseFile)
{
    if(!this.grossSalesLogFileOpen && eraseFile)
    {
        try
        {
            String fileName = "log/grossSales.txt";
            File fileOut = new File(fileName);
            fileOut = new File(fileName);
            FileWriter erase = new FileWriter(fileOut, false);
            erase.close();
        }
        catch (IOException e)
        {
            System.out.println(e);
            MyExit.exit(1);
        }
    }
    try
    {
        String fileName = "log/grossSales.txt";
        File fileOut = new File(fileName);
        grossSalesLog = new FileWriter(fileOut, true);
        this.grossSalesLogFileOpen = true ;
    }
    catch (IOException e)
    {
        System.out.println(e);
        MyExit.exit(1);
    }
}
```

Il metodo cancella il contenuto esistente se *eraseFile = true*, e apre il file *log/grossSales.txt* salvando l'indirizzo di memoria nella variabile *file grossSalesLog*.

C1901

[caption=ComputationalAssembler.java: c1901] L'oggetto computazionale **C1901** simula la fase di disegno dei metacampionari.

```
public void c1901 ()
{
    checkMatrixes(1901,1);

    designMatrix=(MemoryMatrix) pendingComputationalSpecificationSet.
        getMemoryMatrixAddress(0);
    layer=pendingComputationalSpecificationSet.
        getOrderLayer();

    // Zeroing the counter if empty
    if (designMatrix.isEmpty(layer,0,0))
        designMatrix.setValue(layer,0,0,0.0);

    int counter = (int) designMatrix.getValue(layer,0,0);
    counter++;

    //Here we store the number of MetaSample produced
    designMatrix.setValue(layer,0,0,counter);

    System.out.println("MetaSample_#" + counter + "_with_layer#" + layer +
        "_has_been_drawn");

    done=true;
}
```

Ogni volta che viene richiamato esegue un controllo sulla matrice designMatrix in posizione (0,0). Se è vuota viene scritto il valore 0, altrimenti viene incrementato il valore esistente di un unità. Potremmo immaginare questo valore come un contatore dei prodotti da gestire in ogni momento della simulazione per ogni layer (bisogna ricordare che le matrici sono differenti per ogni layer).

C1902

L'oggetto computazionale **c1902** simula le previsioni di acquisto dei licenziatari.

Listing 6.19. ComputationalAssembler.java: c1902

```
public void c1902 ()
```

```

{
  checkMatrixes(1902,2);

  designMatrix=(MemoryMatrix) pendingComputationalSpecificationSet.
  getMemoryMatrixAddress(0);
  agentMatrix=(MemoryMatrix) pendingComputationalSpecificationSet.
  getMemoryMatrixAddress(1);
  layer=pendingComputationalSpecificationSet.
  getOrderLayer();

  // Making prediction for available MetaSample
  if(!designMatrix.isEmpty(layer,0,0))
  {
    //Switching to an other article
    int j = getCounter(agentMatrix,1);

    //The licensee prediction
    int forecast = (int)
    normal.getSampleWithMean$withVariance(500,900000);

    //Min value
    if (forecast < 10)
    forecast = 10;
    //Max value
    if (forecast > 25000)
    forecast = 25000;

    //Storing the prediction
    agentMatrix.setValue(layer,1,j,forecast);

    System.out.println("Licensee_" + myUnit.getUnitNumber() +
    "_forecasted_" + forecast + "_for_MetaSample_" + j +
    "_with_layer_" + layer);

    // End of predictions
    done=true;
  }
  //Waiting for some MetaSample -> done=false
}

```

La previsione viene effettuata estraendo un valore da una distribuzione normale con media 500 e varianza 900000; i valori sono compresi tra 20 e 25000. Il valore estratto viene salvato in agentMatrix(1,j) dove j indica il numero del prodotto previsto.

C1903

L'oggetto computazionale **c1903** effettua la somma delle previsioni dei licenziatari.

Listing 6.20. ComputationalAssembler.java: c1903

```

public void c1903()
{
    float sum = 0;
    int threshold = 18000;
    boolean endForecast = true;

    checkMatrixes(1901,37);

    designMatrix=(MemoryMatrix) pendingComputationalSpecificationSet.
        getMemoryMatrixAddress(0);
    basicNetMatrix=(MemoryMatrix) pendingComputationalSpecificationSet.
        getMemoryMatrixAddress(36);
    layer=pendingComputationalSpecificationSet.
        getOrderLayer();

    // Getting the number of articles
    int counter = (int) designMatrix.getValue(layer,0,0);

    for (int j=1 ; j <=counter ; j++)
    {
        for (int t=1 ; t <=35 ; t++)
        {
            MemoryMatrix agentMatrix =
                (MemoryMatrix) pendingComputationalSpecificationSet.
                    getMemoryMatrixAddress(t);

            //Checking if the licensee made a prediction for this MetaSample
            if (! (agentMatrix.getEmpty(layer,1,j)))
            {
                sum = sum + agentMatrix.getValue(layer,1,j);
            }
        }

        //Storing total predictions in basicNet Matrix
        basicNetMatrix.setValue(layer,1,j,sum);

        //Checking if MetaSample can be produced
        if (sum > threshold)
        {
            basicNetMatrix.setValue(layer,2,j,1);
            System.out.println("MetaSample_" + j + "_with_layer_" + layer +
                "_can_be_produced ,forecast=" + sum + "status=1");
        }
        else
        {
            basicNetMatrix.setValue(layer,2,j,0);
            System.out.println("MetaSample_" + j + "_with_layer_" + layer +
                "_can_NOT_be_produced ,forecast=" + sum + "status=0");
        }

        //Waiting all predictions
        for (int t=1 ; t <=35 ; t++)
        {
            MemoryMatrix agentMatrix =

```

```

    (MemoryMatrix) pendingComputationalSpecificationSet.
    getMemoryMatrixAddress(t);

    if(agentMatrix.isEmpty(layer,1,j))
        endForecast = false;

    }

    if (endForecast)
    {
        System.out.println("MetaSample_" + j + "_with_layer_" + layer +
            "_forecasted_by_all_licensee");

        done=true;
    }

    sum = 0;
}
}

```

Esiste una soglia di fattibilità del prodotto pari a 18000; se la somma delle previsioni supera la soglia il campionario potrà essere realizzato.

C1904

L'oggetto computazionale **c1904** simula la produzione dei singoli campionari.

Listing 6.21. ComputationalAssembler.java: c1904

```

public void c1904 ()
{
    float val;

    checkMatrixes(1904,2);

    designMatrix=(MemoryMatrix) pendingComputationalSpecificationSet.
        getMemoryMatrixAddress(0);
    basicNetMatrix=(MemoryMatrix) pendingComputationalSpecificationSet.
        getMemoryMatrixAddress(1);
    layer=pendingComputationalSpecificationSet.
        getOrderLayer();

    //Switching to an other article
    int i = getCounter(basicNetMatrix, 1);

    if (!designMatrix.isEmpty(layer,0,0) &&
        !basicNetMatrix.isEmpty(layer,2,i))
    {
        if(i <= designMatrix.getValue(layer,0,0) &&

```

```

basicNetMatrix.getValue(layer,2,i) == 1)
{
System.out.println("Unit_#" + myUnit.getUnitNumber() +
"produced_MetaSample_" + i + "with_layer_" + layer);
basicNetMatrix.setValue(layer,2,i,2);

done=true;
}
}
}

```

Anche in questo caso è stato utilizzato il metodo del contatore situato in posizione (1,0) della matrice di basicNet. Viene stampato sul terminale un messaggio e cambiato lo status del prodotto nella *basicNetMatrix*.

C1905

L'oggetto computazionale **c1905** simula l'offerta di prezzo di produzione da parte delle Trading Company nella fase dell'asta. Con un'unica ricetta vengono fatte le previsioni di tutti gli articoli appartenenti alla stessa collezione.

Listing 6.22. ComputationalAssembler.java: c1905

```

public void c1905()
{
float offerPrice, random;

checkMatrixes(1905,3);
designMatrix =(MemoryMatrix) pendingComputationalSpecificationSet.
getMemoryMatrixAddress(0);
basicNetMatrix =(MemoryMatrix) pendingComputationalSpecificationSet.
getMemoryMatrixAddress(1);
tcMatrix =(MemoryMatrix) pendingComputationalSpecificationSet.
getMemoryMatrixAddress(2);
layer=pendingComputationalSpecificationSet.
getOrderLayer();

//Making an offer-price for available articles
if (!designMatrix.getEmpty(layer,0,0))
{
int counter = (int) designMatrix.getValue(layer,0,0);

for (int i=1; i<=counter; i++)
{
if (!basicNetMatrix.getEmpty(layer,2,i))
{

```

```

    if(basicNetMatrix.getValue(layer,2,i) > 0)
    {
        //The offer-price
        offerPrice = (int) normal.getSampleWithMean$withVariance(7,10);

        //Min value
        if (offerPrice <1)
            offerPrice = 1;
        //Max value
        if (offerPrice >20)
            offerPrice=20;

        //Storing the offer-price of the TC
        tcMatrix.setValue(layer, 1,i, offerPrice);

        System.out.println("Unit \u0026;" + myUnit.getUnitNumber() +
            "\u0026;offer\u0026;price\u0026;for\u0026;MetaSample\u0026;" + i + "\u0026;with\u0026;layer\u0026;" + layer +
            "\u0026;is\u0026;$" + offerPrice);
    }
    else
        System.out.println("Unit \u0026;" + myUnit.getUnitNumber() +
            "\u0026;can\u0026;not\u0026;make\u0026;an\u0026;offer\u0026;for\u0026;Metasample\u0026;" + i + "\u0026;with\u0026;layer\u0026;"
            + layer );
    }
}
done=true;
}
// Waiting for some MetaSample -> done=false
}

```

Come nel caso delle previsioni dei licenziatari, anche qui abbiamo deciso di estrarre un valore da una distribuzione normale con media 7 e varianza 10; i valori sono compresi tra 1 e 20. I valori offerti vengono salvati nella `tcMatrix(1,i)` dove `i` è il numero del prodotto per il quale è stata fatta l'offerta.

C1906

L'oggetto computazionale **c1906** effettua la scelta della Trading Company che dovrà produrre gli articoli; la scelta è fatta sul minor prezzo di produzione offerto.

Listing 6.23. ComputationalAssembler.java: c1906

```

public void c1906 ()
{
    MemoryMatrix choiceMatrix;
}

```

```

checkMatrixes(1906,8);
designMatrix =(MemoryMatrix) pendingComputationalSpecificationSet.
    getMemoryMatrixAddress(0);
basicNetMatrix =(MemoryMatrix) pendingComputationalSpecificationSet.
    getMemoryMatrixAddress(6);
orMatrix =(MemoryMatrix) pendingComputationalSpecificationSet.
    getMemoryMatrixAddress(7);
layer =pendingComputationalSpecificationSet.
    getOrderLayer();

// Checking MetaSamples
if(!designMatrix.getEmpty(layer,0,0))
{
    int counter = (int) designMatrix.getValue(layer,0,0);

    for (int i=1; i<=counter; i++)
    {
        int choice = 0;
        float offer=0;

        //Checking the status
        if(!basicNetMatrix.getEmpty(layer,2,i))
        {
            if(basicNetMatrix.getValue(layer,2,i) > 0)
            {
                //Here we make the bidding
                for (int t=1 ; t<=5 ; t++)
                {
                    MemoryMatrix tcMatrix =
                        (MemoryMatrix) pendingComputationalSpecificationSet.
                            getMemoryMatrixAddress(t);

                    if(!tcMatrix.getEmpty(layer,1,i))
                    {
                        if(tcMatrix.getValue(layer,1,i) < offer || offer==0)
                        {
                            offer = tcMatrix.getValue(layer,1,i);
                            choice = t;
                        }
                    }
                }

                //Checking the TC chosen
                if(!basicNetMatrix.getEmpty(layer,2,i))
                {
                    if(basicNetMatrix.getValue(layer,2,i)>0 && !(choice==0))
                    {
                        //Getting its offer price ...
                        choiceMatrix =
                            (MemoryMatrix) pendingComputationalSpecificationSet.
                                getMemoryMatrixAddress(choice);

                        float offerPrice = choiceMatrix.getValue(layer,1,i);

                        //... storing it in basicNet Matrix ...
                        basicNetMatrix.setValue(layer,3,i,offerPrice);
                    }
                }
            }
        }
    }
}

```

```

//... and in the orMatrix
orMatrix.setValue(layer,1,i,choice);

System.out.println("Trading Company#" +
orMatrix.getValue(layer,1,i) +
"has been chosen for MetaSample#" + i + "with layer#" +
layer + "price" + basicNetMatrix.getValue(layer,3,i));
}
}
}
done=true;
}
// Waiting for some MetaSample -> done=false
}

```

L'oggetto computazionale deve anche gestire la matrice *orMatrix* utile successivamente in fase di produzione. Il prezzo scelto viene anche salvato in *basicNetMatrix* per essere utilizzato dagli oggetti 1911 e 1912 che si occupano della contabilità.

C1907

L'oggetto computazionale **c1907** simula, in modo analogo alle previsioni, gli ordini dei licenziatari per ogni singolo articolo. Utilizziamo anche qui un contatore in posizione (2,0) della matrice *agentMatrix* per la gestione dei singoli articoli.

Listing 6.24. ComputationalAssembler.java: c1907

```

public void c1907 ()
{
checkMatrixes(1907,3);

designMatrix=(MemoryMatrix) pendingComputationalSpecificationSet.
getMemoryMatrixAddress(0);
agentMatrix=(MemoryMatrix) pendingComputationalSpecificationSet.
getMemoryMatrixAddress(1);
basicNetMatrix=(MemoryMatrix) pendingComputationalSpecificationSet.
getMemoryMatrixAddress(2);
layer=pendingComputationalSpecificationSet.
getOrderLayer();

if(!designMatrix.getEmpty(layer,0,0))
{
int j = getCounter(agentMatrix,2);

```

```

if (!basicNetMatrix.isEmpty(layer,2,j))
{
if (basicNetMatrix.getValue(layer,2,j)>0)
{
int random= (int) normal.getSampleWithMean$withVariance(500,900000);

//Min value
if (random<10)
random = 10;
//Max value
if (random>25000)
random = 25000;

agentMatrix.setValue(layer,2,j,random);

if (basicNetMatrix.isEmpty(layer,4,j))
basicNetMatrix.setValue(layer,4,j,0.0);

float totOrder = basicNetMatrix.getValue(layer,4,j) + random;
basicNetMatrix.setValue(layer,4,j,totOrder);

System.out.println("Licensee_#" + myUnit.getUnitNumber() +
"ordered_" + random + "for_item_" + j + "with_layer_" + layer
+"totOrder_" +totOrder);

done=true;
}
// The article can not be ordered -> done=false
}
}
}
}

```

Gli ordini eseguiti dal licenziatario per ogni articolo vengono sommati a quelli degli altri licenziatari per lo stesso articolo (variabile *totOrder*) e salvati nella *basicNetMatrix* per eseguire la contabilità.

C 1908

L'oggetto computazionale **c1908** gestisce la produzione degli articoli che deve essere effettuata dalla Trading Company vincitrice dell'asta. Utilizziamo un contatore in posizione (0,0) nella matrice *orMatrix* per distinguere i singoli prodotti.

Listing 6.25. ComputationalAssembler.java: c1908

```

public void c1908()
{

```

```

checkMatrixes(1908,3);

designMatrix=(MemoryMatrix) pendingComputationalSpecificationSet.
    getMemoryMatrixAddress(0);
basicNetMatrix=(MemoryMatrix) pendingComputationalSpecificationSet.
    getMemoryMatrixAddress(1);
orMatrix=(MemoryMatrix) pendingComputationalSpecificationSet.
    getMemoryMatrixAddress(2);
layer=pendingComputationalSpecificationSet.
    getOrderLayer();

if(!designMatrix.getEmpty(layer,0,0))
{
    //Switching to an other article
    int i = getCounter(orMatrix, 0);

    if(i <= designMatrix.getValue(layer,0,0))
    {
        if(!basicNetMatrix.getEmpty(layer,2,i))
        {
            if(basicNetMatrix.getValue(layer,2,i) > 1)
            {
                float branch = orMatrix.getValue(layer,1,i);

                orMatrix.setValue(layer,1,0,branch);

                System.out.println("Branch_00#" + orMatrix.getValue(layer,1,0) +
                    "produced_item#" + i + "with_layer#" + layer);

                basicNetMatrix.setValue(layer,2,i,3.0);

                done=true;
            }
        }
        // The article can not be produced -> done=false
    }
}
}

```

L'oggetto copia nella posizione (1,0) il valore presente nella cesella (1,i) di riferimento di ogni articolo per consentire al meccanismo dell'*orCriterion = 5* di scegliere il "branch" per la produzione.

C1910

L'oggetto computazionale **c1910**, posto in una ricetta al termine di orderSequences.xls, ha lo scopo di interrompere la simulazione e stampare un messaggio sul terminale.

Listing 6.26. ComputationalAssembler.java: c1910

```

public void c1910 ()
{
    checkMatrixes (1910 ,1);
    System.out.println ("*****_THIS_IS_THE_END!_*****");
    StartVEFrame.vEFrameObserverSwarm.getControlPanel().setStateStopped ();
}

```

C1911

L'oggetto computazionale **c1911** esegue la contabilità del fatturato derivante dalla vendita dei campionari.

Listing 6.27. ComputationalAssembler.java: c1911

```

public void c1911 ()
{
    checkMatrixes (1911 ,2);

    designMatrix=(MemoryMatrix) pendingComputationalSpecificationSet.
    getMemoryMatrixAddress (0);
    basicNetMatrix=(MemoryMatrix) pendingComputationalSpecificationSet.
    getMemoryMatrixAddress (1);
    layer=pendingComputationalSpecificationSet.
    getOrderLayer ();

    float samplePrice = (float) normal.getSampleWithMean$withVariance (35,6);

    int counter = getCounter (basicNetMatrix ,4);

    this.logOpen (true);

    String text = Globals.env.getCurrentTime () + "_|_" + samplePrice + "_\n";
    try
    {
        grossSalesLog.write (text);
        grossSalesLog.flush ();
    }
    catch (IOException e)
    {
        System.out.println (e);
        MyExit.exit (1);
    }
    done = true;
}

```

Il prezzo del campionario viene estratto casualmente da una distribuzione normale con media 35 e varianza 6. Il valore contabilizzato viene salvato nel file *log/grossSales.txt*.

C1912

L'oggetto computazionale **c1912** viene utilizzato per calcolare il fatturato dell'azienda derivante dalle royalty sulla vendita degli articoli. Il log dell'operazione viene salvato all'interno del file *log/grossSales.txt*.

Listing 6.28. ComputationalAssembler.java: c1912

```
public void c1912()
{
    checkMatrixes(1912,2);

    designMatrix=(MemoryMatrix) pendingComputationalSpecificationSet.
        getMemoryMatrixAddress(0);
    basicNetMatrix=(MemoryMatrix) pendingComputationalSpecificationSet.
        getMemoryMatrixAddress(1);
    layer=pendingComputationalSpecificationSet.
        getOrderLayer();

    int counter = getCounter(basicNetMatrix,5);

    if(!basicNetMatrix.isEmpty(layer,2,counter) &&
        !basicNetMatrix.isEmpty(layer,3,counter) &&
        !basicNetMatrix.isEmpty(layer,4,counter))
    {
        if (basicNetMatrix.getValue(layer,2,counter)==3.0)
        {
            float itemPrice = basicNetMatrix.getValue(layer,3,counter)*
                basicNetMatrix.getValue(layer,4,counter);

            itemSales = (float) (itemPrice * 0.08);
        }
    }

    String text = Globals.env.getCurrentTime() + "|_|" + itemSales + "\n";
    logOpen(false);
    try
    {
        grossSalesLog.write(text);
        grossSalesLog.flush();
    }
    catch (IOException e)
    {
        System.out.println(e);
        MyExit.exit(1);
    }
}
```

```
done = true;  
}
```

Le royalty sono impostate all'8% del valore dei prodotti venduti. Il valore dei prodotti venduti viene calcolato moltiplicando il prezzo di produzione per le quantità ordinate da tutti i licenziatari.

6.6 Esperimenti e analisi dei risultati

Abbiamo condotto un primo esperimento per valutare la bontà del modello BasicJes. Intendiamo verificare, prima di tutto, la coerenza del sistema di business di BasicNet con la nostra ricostruzione virtuale all'interno di jES.

In seguito condurremo altri esperimenti apportando delle modifiche alle variabili del modello, per poter effettuare un'analisi comparata di diversi scenari¹¹. L'immagine 6.1 riporta un esempio di ciò che può essere osservato in fase di simulazione.

In questa immagine sono riportati i quattro grafici sui quali abbiamo focalizzato maggiormente la nostra attenzione.

Gli esperimenti sono tutti fondati su di un modello base comune, con parametri che rimangono invariati nelle diverse prove. Questi parametri fissi possono essere così riassunti:

- Sono stati simulati 3 anni di produzione riportati nel file *orderSequences.xls*; per ogni anno abbiamo considerato 300 giorni lavorativi (25 per mese) composti da 8 ore.
- Abbiamo considerato, in analogia a quanto ci è stato descritto, 35 licenziatari rappresentati da altrettante unità produttive;
- Sono state considerate 5 Trading Company rappresentate da altrettante unità produttive.
- Vengono realizzate in questi 3 anni 6 collezioni standard di prodotti composte da 300 articoli l'una. Queste rappresentano le principali collezioni (per ordine di grandezza)

¹¹Per la descrizione di questi esperimenti si rimanda al sito <http://eco83.econ.unito.it/tesive/basicnet> (user: tesi - password: tangram).

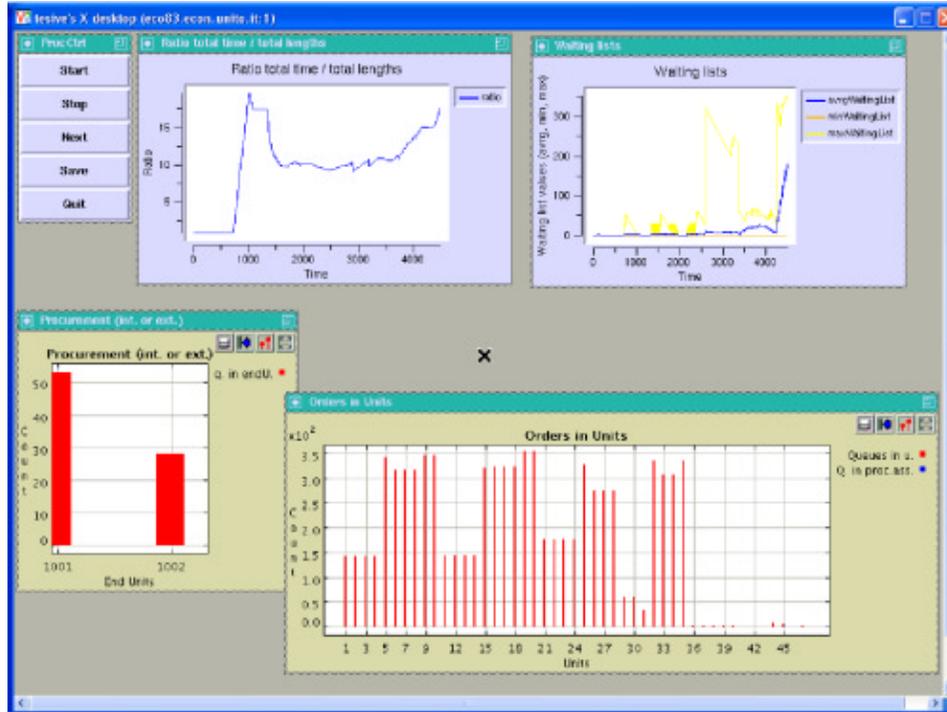


Figura 6.1. La simulazione BasicJVE in corso

del marchio Kappa.

- Abbiamo ipotizzato 90 collezioni piccole composte da 30 articoli, distribuite nell'arco di questi 3 anni. Rappresentato le collezioni di tipo SMU legate ai marchi Kappa e Robe di Kappa.
- Il numero totale di articoli da produrre all'interno della simulazione è di 4500.

Il file *orderSequences.xls* non è quindi stato modificato da un esperimento all'altro; vogliamo analizzare come questo carico di lavoro possa essere gestito in contesti diversi.

6.6.1 Primo esperimento

Il primo esperimento è stato condotto utilizzando le impostazioni fino a questo punto descritte; servirà come punto di partenza per le nostre riflessioni sulla bontà delle stime sui tempi e costi dell'azienda fatte.

Il grafico 6.2 riporta la media nel tempo del rapporto tra la durata descritta nella ricetta ed il tempo effettivo di produzione dell'ordine. Si può osservare che questo rapporto, dopo un periodo di crescita rapida, si assesta su un valore pari a 40.

Mediamente, quindi, un ordine richiede un tempo di circa 40 volte superiore a quello descritto nella ricetta per essere evaso; un abbattimento di questo rapporto implicherebbe un miglioramento delle performance dell'azienda, che in questo esperimento non appaiono particolarmente buone.

Il grafico 6.3 riporta i tempi di attesa minimi, massimi e medi degli ordini in fase di produzione. L'andamento di queste serie risulta molto ciclico e con punte piuttosto alte.

Oltre ad un'analisi dei grafici che jES mette a disposizione, le nostre considerazioni sono basate sull'analisi dei files *concludedOrderLog.txt* e *grossSales.txt* prodotti dalla simulazione.

Dai file di *log* possiamo constatare che molti articoli non sono stati prodotti poiché non hanno superato la soglia di previsioni richiesta.

In conclusione, unendo tutti i dati che abbiamo a disposizione, ci rendiamo conto che il processo organizzativo presenta un carico di lavoro eccessivo per i licenziatari; molti articoli non vengono prodotti poiché i licenziatari non riescono ad effettuare le previsioni

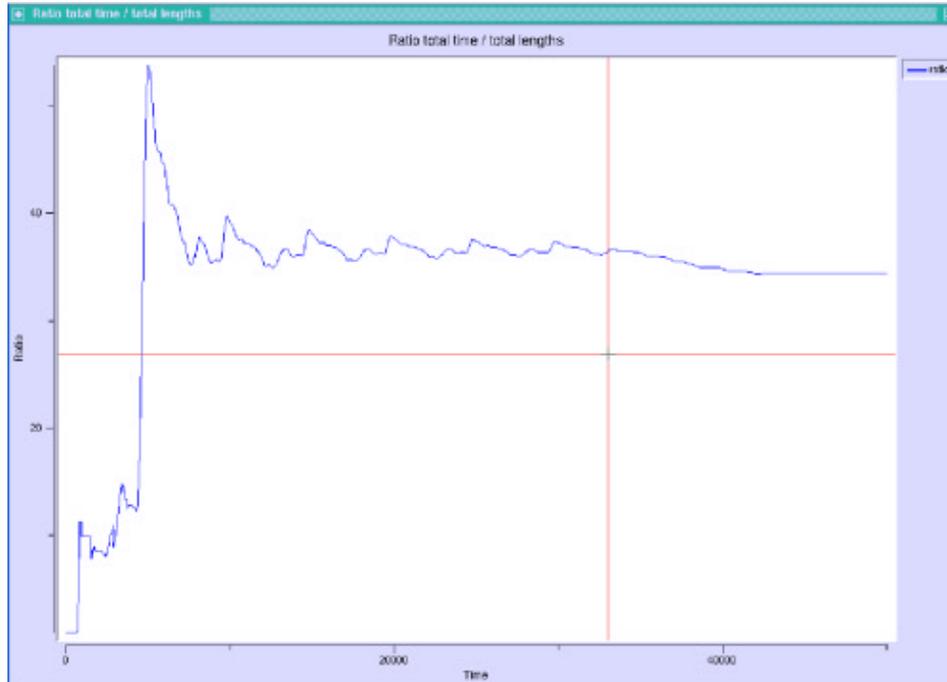


Figura 6.2. Rapporto tra tempi della ricetta e tempi di produzione

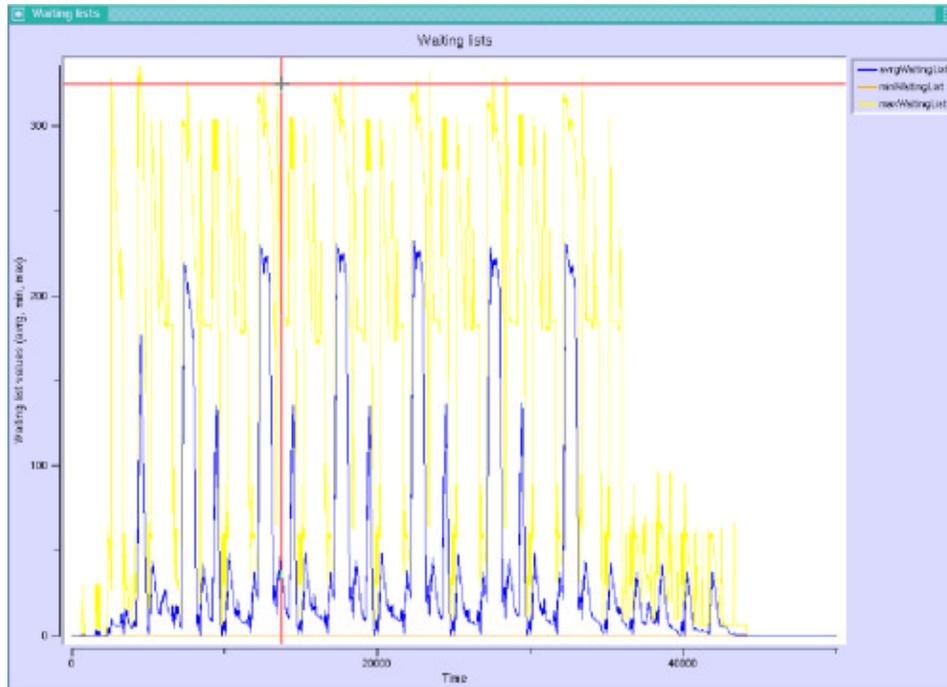


Figura 6.3. Tempi di attesa minimi, massimi e medi degli ordini

entro le scadenze richieste. Il dubbio che ci poniamo al termine di questo primo esperimento riguarda la verosimiglianza dei tempi ipotizzati all'interno delle ricette.

In un secondo esperimento intendiamo modificare i tempi di produzione con un semplice stratagemma, considerare 64 *tickInADay* invece degli originali 32.

Un'altra ipotesi potrebbe essere impostare a 0 i tempi necessari ai licenziatari per fare le previsioni (ora sono pari a 1) in considerazione del fatto che si tratta di agenti esterni dei quali non ci interessa indagare il funzionamento.

Per la descrizione di questi esperimenti si rimanda al sito <http://eco83.econ.unito.it/tesive/basicnet> (user: tesi - password: tangram).

6.6.2 Secondo esperimento

Sebbene¹² io e Marco fossimo riusciti a ricostruire, in modo coerente, il *business system* della BasicNet, i risultati ottenuti dalla prima simulazione non erano ancora realistici.

¹²Questo esperimento è stato condotto da Francesco Merlo nel febbraio 2003

Troppi prodotti non potevano essere realizzati a causa di irrealistiche code d'attesa nelle unità rappresentative dei licenziatari.

Ulteriori confronti con la dottoressa Bruschi mi hanno permesso di riformulare un modello di simulazione più aderente alla realtà. La prima scelta che ho fatto è stata la riduzione del periodo di tempo da simulare al fine di rendere più semplice la lettura (e le modifiche) del file *orderSequence.xls*. Il periodo simulato passa, così, da 1200 giorni lavorativi a 500.

In merito alle irrealistiche code d'attesa registrate nel primo esperimento, ho potuto constatare che l'introduzione di due unità produttive, con compiti differenti, per ogni licenziatario non sarebbe stata una rappresentazione molto distante dalla realtà. Tutti i licenziatari lavorano, infatti, con almeno due sotto-divisioni: una per effettuare le previsioni ed una per gli ordini.

Mi sono reso conto che, per ri-costruire l'azienda BasicNet all'interno del computer, era necessario mettere la "lente di ingrandimento" su alcuni aspetti trascurati in una prima analisi. Operando in questo modo, ho deciso di introdurre una seconda importante modifica: la divisione BasicSample in BasicJes, che in precedenza era rappresentata da una singola unità produttiva, è stata divisa in quattro sotto-divisioni.

In BasicSample, infatti, i disegnatori lavorano in modo separato ed autonomo su quattro tipologie di prodotti:

- abbigliamento con marchio Kappa;
- abbigliamento con marchio Robe di Kappa;
- calzature;
- abbigliamento tecnico sportivo.

L'introduzione di queste quattro unità ha permesso, inoltre, di eliminare una semplificazione introdotta nel primo esperimento. In questa versione, su 500 giorni lavorativi, è prevista una collezione principale per ogni tipo di prodotto, rispetto all'unica collezione rappresentativa della precedente simulazione.

Le modifiche al modello

Nella nuova simulazione saranno prodotti 2140 articoli suddivisi in 16 collezioni, come riportato in tabella 6.20.

Layer	Collezione	Articoli
1	Kappa	300
2	Robe di Kappa	300
3	Calzature	150
4	Sport	200
5	Kappa	300
6	Robe di Kappa	300
7	Calzature	150
8	Sport	200
9	Kappa SMU	30
10	Robe di Kappa SMU	30
11	Calzature SMU	30
12	Sport SMU	30
13	Kappa SMU	30
14	Robe di Kappa SMU	30
15	Calzature SMU	30
18	Sport SMU	30

Tabella 6.20. Secondo esperimento: collezioni introdotte

Le singole collezioni sono lanciate seguendo una scala dei tempi semplificata, ma non troppo irrealistica, come riportato in tabella 6.21.

In seguito ai cambiamenti introdotti, il modello ora prevede 97 unità produttive, suddivise nel seguente modo:

Codice da 1 a 70: due unità per ognuno dei 35 licenziatari;

Codice da 71 a 75: le Trading Company;

Codice da 76 a 79: le sotto-divisioni di BasicSample;

Codice 80: la divisione BasicForecast;

Codice da 81 a 84: la divisione BasicSpecs;

Codice da 85 a 88 : un'ipotetica Trading Company per la produzione del campionario;

Giorno	Collezione (layer)
1	1
1	2
1	3
1	4
90	9
120	10
150	11
180	12
185	5
185	6
185	7
185	8
210	13
240	14
270	15
300	16

Tabella 6.21. Secondo esperimento: lancio delle collezioni

Codice 89: la divisione BasicBidding;

Codice da 90 a 93: la divisione BasicFactory;

Codice da 94 a 97: le unità rappresentative del network BasicNet.

Come si può vedere, alcune divisioni sono state rappresentate con quattro unità produttive rispetto all'unica utilizzata nel primo esperimento; questa scelta è dettata dal fatto che, dato il numero di collezioni grandi, il "carico" di lavoro è circa quattro volte maggiore.

Nel file *unitData/unitBasicData.txt* sono riportati i passi che le diverse unità sono in grado di compiere, come riportato (in modo sintetico) in tabella 6.22.

Avendo inserito delle nuove unità produttive, è stato necessario rivedere anche le ricette riportate nel file *recipeData/recipes.xls*. Le nuove ricette sono riportate in tabella 6.23.

Gli ordini sono lanciati, nel corso della simulazione, seguendo un intervallo di tempo molto preciso. Le collezioni principali (quelle da 150-300 articoli) seguono i tempi riportati in tabella 6.24.

Le collezioni SMU (da 30 articoli) hanno un tempo di realizzazione minore (circa la

Unità	Passo
1	1010
2	1011
3	1020
4	1021
5	1030
...	...
67	1340
68	1341
69	1350
70	1351
71	2010
72	2020
73	2030
74	2040
75	2050
76	3010
77	3011
78	3012
79	3013
80	3020
81	3030
...	...
84	3030
85	3040
...	...
88	3040
89	3050
90	3060
...	...
93	3060
94	4000
...	...
97	4000

Tabella 6.22. Secondo esperimento: le unità produttive

metà di una collezione principale), quindi seguono dei tempi come quelli riportati in tabella 6.25.

Date le precedenti sequenze di ordini, anche la variabile *maxTickInAUnit* presente nel file *jesframe.scm* è stata cambiata ed impostata a 2080. Questo significa che gli ordini sono eliminati dopo 65 giorni di permanenza in una coda d'attesa.

Ricetta	Codice	Passi									
disegnoKappa	101	c	1901	1	0	3010	s	6			
disegnoRdK	102	c	1901	1	0	3011	s	6			
disegnoScarpe	103	c	1901	1	0	3012	s	6			
disegnoSport	104	c	1901	1	0	3013	s	6			
previsionilicenziatario1	201	c	1902	2	0	1	1010	s	1		
...											
previsionilicenziatario35	235	c	1902	2	0	35	1350	s	1		
sommaprevisioni	300	c	1903	37	0	1	2	3	4	5	6
			7	8	9	10	11	12	13	14	15
			16	17	18	19	20	21	22	23	24
			25	26	27	28	29	30	31	32	33
						34	35	41	3020	s	32
speceprodcampionario	301	c	1904	2	0	41	3030	s	1		
							3040	s	1	e	101
consegnacampionario	302	p	1	101	c	1911	2	0	41		
									4000	s	1
offertatc1	401	c	1905	3	0	41	36	2010	s	32	
...											
offertatc5	405	c	1905	3	0	41	40	2050	s	32	
sceltatc	500	c	1906	8	0	36	37	38	39	40	41
								42	3050	s	32
ordinilicenziatario1	701	c	1907	3	0	1	41	1011	s	1	
...											
ordinilicenziatario35	735	c	1907	3	0	35	41	1351	s	1	
produzioneprodotto	800	c	1908	3	0	41	42	3060	s	1	
								1001	2010	s	1
								2	2020	s	1
								3	2030	s	1
								4	2040	s	1
								5	2050	s	1
								0	3060	s	0
										e	102
consegnaprodotto	900	p	1	102	c	1912	2	0	41		
									4000	s	1

Tabella 6.23. Secondo esperimento: ricette produttive

Giorno lavorativo	Ricetta produttiva	Codice ricetta
1	Disegno metacampionario	101-104
60	Previsione licenziatari	201-235
95	Somma previsioni	300
95	Specifiche tecniche e produzione campionato	301
100	Offerta Trading Company	401-405
135	Scelta Trading Company	500
140	Consegna campionato	600
145	Ordini licenziatari	701-735
185	Produzione prodotti	800
250	Consegna prodotti	900

Tabella 6.24. Secondo esperimento: sequenza del lancio degli ordini di una collezione principale

Giorno lavorativo	Ricetta produttiva	Codice ricetta
1	Disegno metacampionario	101-104
30	Previsione licenziatari	201-235
50	Somma previsioni	300
50	Specifiche tecniche e produzione campionato	301
55	Offerta Trading Company	401-405
75	Scelta Trading Company	500
80	Consegna campionato	600
85	Ordini licenziatari	701-735
115	Produzione prodotti	800
150	Consegna prodotti	900

Tabella 6.25. Secondo esperimento: sequenza del lancio degli ordini di una collezione SMU

Analisi dei risultati

Come è possibile osservare dal grafico in figura 6.4, dai tempi d'attesa massimi è possibile individuare le diverse fasi del processo organizzativo di BasicJes; le punte di circa 1000 ordini in attesa corrispondono alle fasi di previsione ed ordine degli articoli da parte dei licenziatari.

Anche il rapporto (medio) tra la lunghezza delle ricette ed il tempo impiegato per portare a termine l'ordine, riportato in figura 6.5, presenta un ordine di grandezza coerente con le aspettative.

E' interessante anche osservare, attraverso la tabella 6.26, quanti articoli, rispetto a quelli programmati, sono stati effettivamente disegnati, e quanti di questi hanno passato

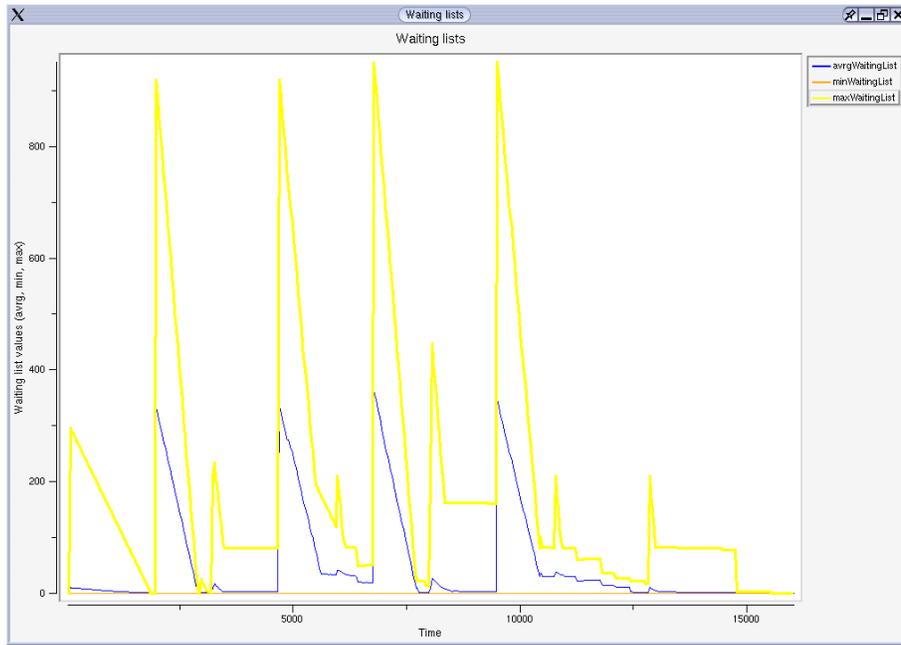


Figura 6.4. Secondo esperimento: tempi di attesa minimi, massimi e medi degli ordini

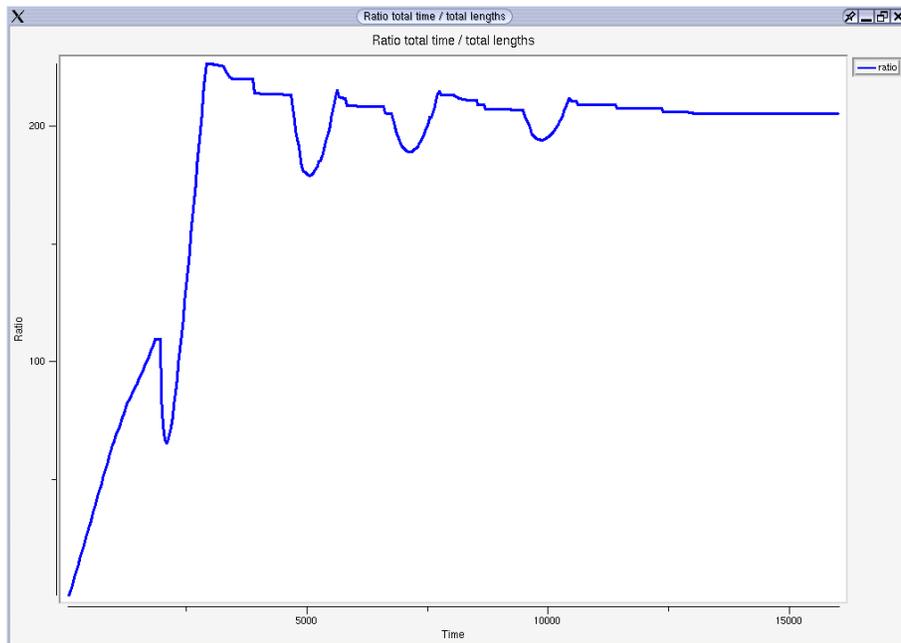


Figura 6.5. Secondo esperimento: rapporto tra tempi della ricetta e tempi di produzione

la soglia minima di previsioni per essere prodotti.

Layer	Programmati	Disegnati	Prodotti
1	300	298	267
2	300	298	275
3	150	150	140
4	200	200	188
5	300	298	278
6	300	298	275
7	150	150	140
8	200	200	179
9	30	30	28
10	30	30	29
11	30	30	30
12	30	30	29
13	30	30	29
14	30	30	27
15	30	30	28
16	30	30	30

Tabella 6.26. Secondo esperimento: articoli programmati, disegnati e prodotti

Al termine della simulazione, è possibile anche ottenere il fatturato dell'azienda.

Considerazioni sulla contabilità

Il fatturato dell'azienda è calcolato per mezzo oggetti computazionali con codice **1911** e **1912**. Il fatturato deriva dalla vendita dei metasample (rappresentata dalla ricetta 302) e dall'incasso delle *royalties* pari all'8% sul totale degli acquisti dei licenziatari (ricetta 900). In questo secondo esperimento il fatturato risulta di circa 11.500.000.

Conosciuti gli ordini di grandezza dei costi, si è così deciso di "calibrare" la simulazione in modo tale da portare BasicJes in una situazione di pareggio al termine della simulazione.

In tabella 6.27 sono riportati gli ordini di grandezza dei costi delle diverse divisioni.

Ottenuti i costi che le singole divisioni dovrebbero registrare al termine del periodo simulato, è stata effettuata una stima sulla ripartizione in costi fissi e variabili come riportato in tabella 6.28.

I costi fissi e variabili sono stati successivamente divisi per il numero di *tick* durante i

	Ripartizione dei costi	
BasicSample	20,00%	2.300.000
BasicForecast	3,00%	345.000
BasicSpecs	15,00%	1.725.000
BasicBidding	2,00%	230.000
BasicFactory	10,00%	1.150.000
BasicNet	50,00%	5.750.000

Tabella 6.27. Secondo esperimento: ripartizione dei costi tra le divisioni

	Ripartizione in CF e CV			
	CF		CV	
BasicSample	25,00%	575.000	75,00%	1.725.000
BasicForecast	75,00%	258.750	25,00%	86.250
BasicSpecs	25,00%	431.250	75,00%	1.293.750
BasicBidding	75,00%	172.500	25,00%	57.500
BasicFactory	25,00%	287.500	75,00%	862.500
BasicNet	50,00%	2.875.000	50,00%	2.875.000

Tabella 6.28. Secondo esperimento: ripartizione dei costi in fissi e variabili per ogni divisione

quali devono essere registrati, come riportato in tabella 6.29.

	Costi per tick			
	CF		CV	
	tick	costo	tick	costo
BasicSample	16000	35,93	12840	134,34
BasicForecast	16000	16,17	512	168,45
BasicSpecs	16000	26,95	2140	604,55
BasicBidding	16000	10,78	512	112,30
BasicFactory	16000	17,96	2140	403,03
BasicNet	16000	179,68	4280	671,72

Tabella 6.29. Secondo esperimento: ripartizione costi per tick

A questo punto i soli costi fissi devono ancora essere suddivisi nelle diverse unità produttive di BasicJes, come riportato in tabella 6.30.

	Costi per unità	
	Unità	CF
BasicSample	76	8,98
	77	8,98
	78	8,98
	79	8,98
BasicForecast	80	16,17
BasicSpecs	81	6,73
	82	6,73
	83	6,73
	84	6,73
BasicBidding	89	10,78
BasicFactory	90	4,49
	91	4,49
	92	4,49
	93	4,49
BasicNet	94	44,92
	95	44,92
	96	44,92
	97	44,92

Tabella 6.30. Secondo esperimento: Ripartizione dei costi fissi per unità produttiva

Terminata la simulazione possiamo ricavare un grafico come quello riportato in figura 6.6 attraverso il quale è possibile osservare l'andamento, giornaliero dei costi totali, del

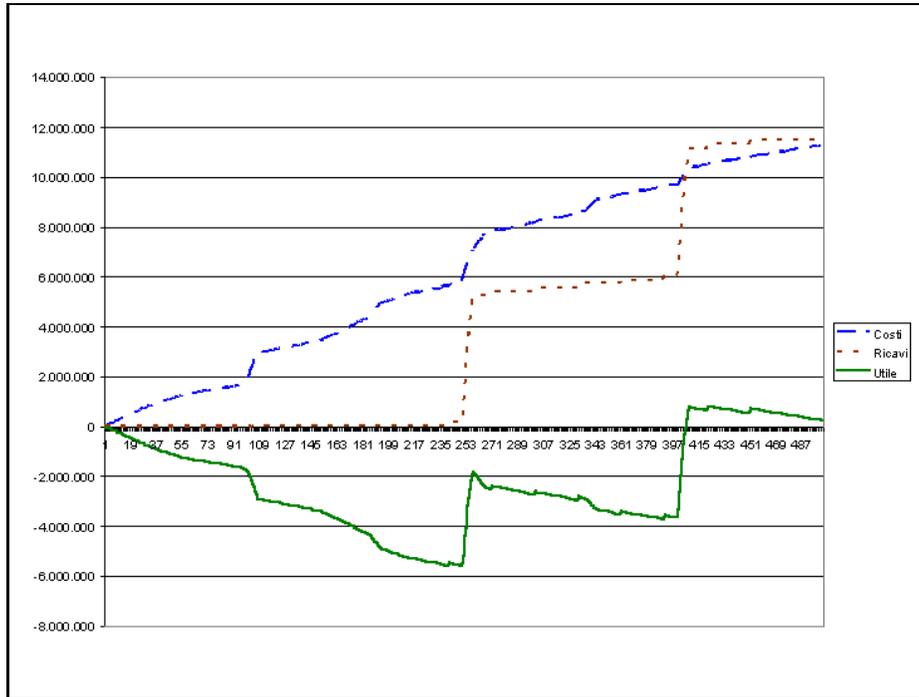


Figura 6.6. Secondo esperimento: costi, ricavi e utile totali

fatturato e dell'utile.

Bibliografia

- [6.1] Barreca C., *Simulazioni ad agenti in JavaSwarm: applicazione ad un caso aziendale*, Tesi di laurea in Economia e Commercio, Torino, 2002.
- [6.2] BasicBiddings, <http://www.basicbiddings.com>, ultima visita Dicembre 2002.
- [6.3] BasicFactory, <http://www.basicfactory.com>, ultima visita Dicembre 2002.
- [6.4] BasicForecast, <http://www.basicforecast.com>, ultima visita Dicembre 2002.
- [6.5] BasicMarketing, <http://www.basicmarketing.com>, ultima visita Dicembre 2002.
- [6.6] BasicNet, <http://www.basicnet.com>, ultima visita Dicembre 2002.
- [6.7] BasicPress, <http://www.basicpress.com>, ultima visita Dicembre 2002.
- [6.8] BasicSample, <http://www.basicssample.com>, ultima visita Dicembre 2002.
- [6.9] BasicSpecs, <http://www.basicsspecs.com>, ultima visita Dicembre 2002.
- [6.10] BasicTrademark, <http://www.basictrademark.com>, ultima visita Dicembre 2002.
- [6.11] Borra A., *Generalizzazione del modello javaswarm di simulazione di impresa per l'applicazione ad un caso concreto*, Tesi di laurea in Economia e Commercio, Torino, 2002.
- [6.12] Lamieri M., Merlo F., *BasicJVE - Analisi della BasicNet in prospettiva di formalizzazione per il modello JVEFrame*, <http://eco83.econ.unito.it/tesive/basicnet/>, (user: tesi - password: tangram).
- [6.13] Lamieri M., *A Simulazione Agent Based in contesti d'impresa: applicazione del modello Virtual Enterprise in JavaSwarm ad un'azienda reale*, Tesi di laurea in Economia e Commercio, Torino, 2002.
- [6.14] Terna, P., *Simulazione ad agenti in contesti di impresa*, Sistemi intelligenti, XVI (1), 2002.

Parte III

Conclusioni e appendici

Conclusioni

La simulazione al computer è indubbiamente una rivoluzione nel metodo. Consente di elaborare modelli complessi e spiegare fenomeni che difficilmente potrebbero essere compresi altrimenti.

Nelle scienze sociali la simulazione dovrebbe essere ancora più utile perché permette di studiare le interazioni tra gli esseri umani, troppo difficili da spiegare con i metodi letterari e matematici. Nel campo dell'economia si possono costruire modelli per spiegare ogni genere di fenomeno, da una semplice economia di scambio ad un complesso mercato di borsa.

Considerazioni sulla simulazione BasicJes

Con l'occasione delle nostre tesi, io e Marco Lamieri, abbiamo avuto la piacevole opportunità di applicare uno strumento di simulazione per ri-costruire un'azienda reale che si è dimostrata disponibile ed interessata nei nostri confronti. La parte sicuramente più stimolante, a mio giudizio, è stata l'attività più "teorica" legata al modello jES; essendo infatti la nostra una delle prime applicazioni su casi concreti (parallelamente all'esperienza di altri tesisti in VIR s.p.a.) il modello di simulazione risultava sotto alcuni aspetti ancora immaturo, rendendo così necessario un'interessante confronto tra noi ed il professore Terna su quali potessero essere le soluzioni migliori.

L'introduzione di funzionalità, non tanto tecniche ma metodologiche, come i *layer*, gli *oggetti computazionali* o l'*orderDistiller* ha richiesto molto tempo; questo perché fin dal principio lo sviluppo è stato svolto con la prospettiva di riutilizzare ogni innovazione anche

nelle applicazioni future. Ultimamente, infatti, si stanno aprendo nuove tesi nel campo della sanità: sarà interessante scoprire se le nostre analisi al modello jES risulteranno utili alla ri-costruzione di un pronto soccorso.

Durante le diverse formalizzazioni per realizzare BasicJes, ci siamo resi conto che la bontà dei risultati raggiunti dipende dalla verosimiglianza del modello all'azienda, e quindi dal numero di dettagli che si riescono ad inserire nella simulazione. Il primo obiettivo che abbiamo raggiunto è stata la ri-costruzione del *business system* dell'intero network BasicNet, infatti, già dal primo esperimento, possiamo constatare la coerenza tra gli eventi simulati e quelli reali che ci sono stati descritti. I processi organizzativi si susseguono all'interno della simulazione in modo logico e coerente, i meccanismi di coordinamento dell'azienda sui Licenziatari e le Trading Company vengono trasformati, nella simulazione, in una sorta di coerenza interna, che permette al modello di funzionare.

Nella costruzione del modello abbiamo seguito una procedura "dal basso verso l'alto"; la nostra azienda simulata nasce dalla descrizione delle singole unità produttive (chi fa che cosa) e le singole operazioni che devono essere intraprese nel tempo (che cosa fare). Solo quando la simulazione *gira* è possibile giudicare la bontà delle nostre descrizioni. Questo procedimento è formalmente rigoroso e ci ha offerto la possibilità di verificare la nostra comprensione dei meccanismi e delle relazioni che esistono in BasicNet.

Abbiamo così potuto constatare che osservando l'azienda nel suo insieme, come viene fatto nei modelli tradizionali, spesso è difficile cogliere i dettagli di alcune relazioni ed interazioni. Attraverso la simulazione, invece, si comprende che questi dettagli sono fondamentali nella realtà osservata.

Partendo dal modello letterario-descrittivo costruito "dall'alto" che ci ha fornito la BasicNet, abbiamo incontrato alcune difficoltà nella stima dei particolari operativi essenziali per costruire una simulazione più vicina alla realtà. Nel corso del primo esperimento, i dati generati dalla simulazione sul carico di lavoro delle unità sono stati uno spunto di riflessione interessante; alcune unità non riuscivano a sostenere il carico di lavoro richiesto.

Questo fenomeno (imprevisto) nasce appunto dal confronto tra i due metodi di descrizione. Nel modello "letterario" dell'azienda sono presenti passaggi logici che non possono

essere tradotti direttamente in un modello di "simulazione" come jES. Si richiederebbero semplificazioni troppo forti e la simulazione risulterebbe una semplice rappresentazione "animata" della realtà che ci è stata descritta.

A nostro giudizio, alcuni di questi passaggi dovrebbero essere indagati nuovamente sotto un diverso punto di vista. Un esempio del problema possiamo trovarlo nella fase di disegno dei campionari: supponiamo che i tempi necessari per realizzare una collezione (o addirittura il singolo campionario) siano la chiave per avvicinarsi alla realtà. Per rendere coerente il carico di lavoro e i tempi di produzione si dovrebbe così indagare in modo più approfondito sulle singole azioni che intraprendono le varie divisioni e sotto-divisioni.

Una prima conclusione raggiunta è che il continuo confronto tra il modello letterario e quello simulato può portare dei miglioramenti ad entrambi. Il pregio della prima fase del lavoro è stato la creazione di un "frame" generale che descrive, pur presentando alcune semplificazioni, la BasicNet nel suo insieme.

Una seconda conclusione deriva dal secondo esperimento. Essendo BasicJes una riproduzione abbastanza realistica è possibile, da questo momento in poi, utilizzarla come un laboratorio virtuale dove, in condizioni controllate, è possibile sperimentare alcune varianti. Oltre a permettere la costruzione di "scenari" reali, le simulazioni consentono di ipotizzare "scenari" possibili, situazioni alternative o non ancora affrontate.

Il secondo esperimento è stato sviluppato in questa direzione. Insieme alla dottoressa Bruschi sono stati stimati i costi di ogni divisione interna: la stima è stata fatta per ordini di grandezza e valutando quali divisioni avessero più costi fissi e quali più costi variabili. L'esperimento è stato successivamente "calibrato" in modo tale da ottenere un'azienda che chiude in pareggio al termine della simulazione. A partire da questa situazione ipotetica è possibile effettuare dei confronti con strutture diverse del network BasicNet, immaginando ad esempio una maggiore o minore internalizzazione dei processi produttivi.

I prossimi studi potrebbero essere nella direzione di creare uno o più modelli di business diversi da quello attuale, nei quali si potrebbe decidere di internalizzare o meno parte dei costi e, soprattutto, dei rischi d'impresa. Si può sperimentare una BasicJes che gestisce direttamente la produzione, oppure una configurazione nella quale il maggiore licenziatario,

Kappa Italia, operi come una divisione vendite. Si potrebbe, infine, sperimentare una struttura d'azienda ancora più virtuale, completamente esternalizzata, nella quale anche i MetaSample sono prodotti all'esterno, riducendo al minimo i rischi d'impresa

Possibili sviluppi in BasicJes

Essendo BasicNet un'azienda "virtuale", dove la maggior parte delle operazioni sono immateriali (come ordini, previsioni, offerte di prezzo, . . .), uno dei possibili sviluppi di BasicJes potrebbe essere la simulazione del sistema informativo attualmente utilizzato.

Un ampliamento del modello in questa direzione sarebbe molto utile alla nostra simulazione poiché l'attività principale dell'azienda è di tipo organizzativo, dove l'informazione è l'elemento essenziale.

In questo modo sarebbe possibile osservare la BasicNet ri-costruita sotto due diversi punti di vista: da una parte si vedrebbe ciò che realmente accade al suo interno (le "vere" operazioni materiali e non) e dall'altra qual'è la rappresentazione che l'azienda ha di se stessa. Una "duplice" simulazione risulterebbe interessante poiché è attraverso il sistema informativo che sono prese le decisioni in merito all'organizzazione interna.

Una seconda direzione di sviluppo del progetto BasicJes potrebbe riguardare un ulteriore approfondimento sulla divisione BasicSample, che reputiamo il punto chiave di tutta l'azienda; gli unici beni concretamente prodotti da BasicNet sono, infatti, i *MetaSamples*. Il modello di business è studiato in modo tale per cui sarebbe possibile esternalizzare anche questa fase ma, per una scelta strategica, è stato deciso di realizzarli direttamente. Per questo motivo, e per aver compreso che BasicSample è il maggior centro di costo, riteniamo che meriterebbe un'analisi più approfondita. Nel secondo esperimento la divisione BasicSample è già stata divisa in quattro sotto-divisioni con compiti ben specifici, ma l'insieme di relazioni che si sviluppano al suo interno (a livello di personale e di organizzazione del lavoro) sono al momento ignorate. Per eventuali approfondimenti futuri, potrebbe risultare utile l'analisi di questa divisione come un'azienda indipendente.

Possibili sviluppi in jES

Propongo, in conclusione, due ulteriori possibilità di sviluppo per l'applicazione jES; una "teorica", nel senso che riguarda il modello al quale si appoggia, ed una "pratica", relativa all'attuale programma scritto in JavaSwarm.

La proposta "teorica" riguarda i formalismi attualmente utilizzati per descrivere l'azienda simulata. Al momento, come si è già visto, esistono i due formalismi distinti riguardanti le unità e le ricette produttive.

In questo mio periodo di lavoro e di sviluppo mi sono accorto che, affinché un'azienda ri-costruita da un caso reale (come BasicNet o VIR) possa funzionare, è fondamentale essere rigorosi e realistici in una terza descrizione: il "quando fare che cosa".

Durante le molte simulazioni che io e Marco Lamieri abbiamo fatto "girare", ci siamo resi conto che i risultati che ottenevamo potevano risultare molto differenti in base a piccole variazioni nella sequenza del lancio degli ordini. E' normale pensare che un'azienda non sia in grado di affrontare situazioni anomale o impreviste, ma la stessa organizzazione interna "emerge" anche dalla cadenza degli eventi che essa deve affrontare. Pochi esempi possono chiarire il concetto: l'organizzazione di un maglificio è legata ai tempi necessari per ottenere la nuova lana, l'attività di un'enoteca dipende in ultima analisi dai periodi di vendemmia, mentre un panettiere è vincolato dai tempi di lievitazione del pane.

Nei modelli "letterari", come quelli proposti dalle aziende, risulta spesso difficile individuare lo stretto legame di interdipendenza tra il "che cosa fare" (WD) "chi fa che cosa" (DW) e "quando fare che cosa" (WDW, When Doing What). Io e Marco Lamieri, nel nostro caso, abbiamo affrontato molte difficoltà nel formalizzare in modo coerente la sequenza degli avvenimenti, partendo comunque dall'ottima descrizione "verbale" che ci è stata fatta dalla dottoressa Bruschi.

Nell'attuale versione di jES esiste già questo aspetto che, più tecnicamente, si affronta durante la scrittura di *orderSequences.xls*. In questo file l'utente deve infatti riportare, con un altro formalismo, la sequenza del lancio degli ordini di ogni "giorno" simulato. Anche i *layer*, riportati nello stesso file, sono fortemente legati con lo scorrere del tempo: nella simulazione BasicJes sono stati utilizzati per distinguere le diverse collezioni perché,

logicamente, non era possibile confondere gli ordini relativi ad una collezione primavera-estate da quelli di una collezione autunno-inverno.

Al termine della mia esperienza e in base a queste riflessioni, ritengo che strumenti di simulazione come jES permettano una rigorosa verifica sulla conoscenza che si ha della realtà aziendale oggetto di studio.

La seconda e ultima proposta di sviluppo è più tecnica; nasce dall'esperienza fatta in questo periodo sul programma scritto in JavaSwarm. Il secondo esperimento di BasicJes, riportato nell'ultimo capitolo, richiede un tempo di elaborazione di circa sei ore su un computer da ufficio con un processore a 1000 Mhz. In alcuni casi, credo, potrebbe essere utile "congelare" l'azienda simulata in un determinato momento, salvando in qualche modo (anche su semplici file di testo) lo stato delle attività (le code d'attesa, gli ordini in lavorazione, la contabilità, ...).

In questo modo sarebbe possibile far girare la simulazione ad intervalli, evitando così di ripetere da capo la computazione quando accadono degli eventi spiacevoli. Il vantaggio maggiore, però, sarebbe la possibilità di ripetere l'animazione di alcuni passaggi che si ritengono importanti senza dovere ri-eseguire l'intera simulazione. Alcuni esempi potrebbero essere la formazione di impreviste code d'attesa o dei periodi di inattività di alcune unità produttive; la stessa porzione di simulazione potrebbe essere successivamente analizzata, come in un laboratorio, introducendo delle varianti come nuove unità o nuovi ordini in arrivo.

Appendici

ComputationalAssembler.java

Si riporta il codice sorgente del file *ComputationalAssembler.java*, aggiornato alla versione utilizzata in *BasicJes-0.9.7.52*.

Listing 6.29. ComputationalAssembler.java

```
// ComputationalAssembler.java
import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;
import swarm.collections.ListImpl;
import swarm.collections.ListIndex;
import swarm.random.NormalDistImpl;
import java.lang.Float;
import java.io.*;
import java.util.*;
/**
 * The ComputationalAssembler class instances make
 * computational processes; we have a computationa assembler instance for
 * each unit
 *
 *
 * @author<br/>
 * Marco Lamieri<a href="mailto:lamieri@econ.unito.it"></a></br>
 * Francesco Merlo<a href="mailto:merlo@econ.unito.it"></a></br>
 * @version 0.9.7.31.b (BasicJVE 1.0)
 */
public class ComputationalAssembler extends ComputationalAssemblerBasic
{
    //Our MemoryMatrices
    MemoryMatrix designMatrix, basicNetMatrix, tcMatrix, orMatrix, agentMatrix;
    //A normal distribution for random numbers
    NormalDistImpl normal;
    //grossSales File management
    public boolean grossSalesLogFileOpen = false;
```

```

FileWriter grossSalesLog=null;
int samplesN=0;
float sampleSales = 0;
float itemSales = 0;
public MyWriter report=null;
/**
 * the constructor for ComputationalAssembler
 */
public ComputationalAssembler (Zone aZone)
{
    // Call the constructor for the parent class.
    super(aZone);
    normal = new NormalDistImpl (getZone());
}
//*****
// BasicJES Computational codes
//*****

/** computational operations with code -1901 <br><br>
 *
 * this computational code increment 1 by 1 position 0,0 of the
 * unique received matrix and set the status to done
 */
public void c1901()
{
    checkMatrixes(1901,1);

    designMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(0);
    layer = pendingComputationalSpecificationSet.getOrderLayer();

    // Zeroing the counter if empty
    if(designMatrix.getEmpty(layer,0,0))
        designMatrix.setValue(layer,0,0,0.0);

    int counter = (int) designMatrix.getValue(layer,0,0);
    counter++;

    //Here we store the number of MetaSample produced
    designMatrix.setValue(layer,0,0,counter);

    System.out.println("MetaSample_" + counter + "_with_layer_" + layer + "has been drawn");

    done=true;
}
// end c1901

/** computational operations with code -1902 <br><br>
 *
 * this computational code dial with licensee's predictions.
 */
public void c1902()
{
    checkMatrixes(1902,2);

    designMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(0);

```

```

agentMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(1);
layer=pendingComputationalSpecificationSet.getOrderLayer();

// Making prediction for available MetaSample
if (!designMatrix.isEmpty(layer,0,0))
{
    //Switching to an other article
    int j = getCounter(agentMatrix,1);

    //The licensee prediction
    int forecast = (int) normal.getSampleWithMean$withVariance(500,900000);

    //Min value
    if (forecast < 10)
        forecast = 10;
    //Max value
    if (forecast > 25000)
        forecast = 25000;

    //Storing the prediction
    agentMatrix.setValue(layer,1,j,forecast);

    System.out.println("Licensee_" + myUnit.getUnitNumber() + "_forecasted_" + forecast +
        "_for_" + metaSample + "_" + j + "_with_" + layer + "_");

    // End of predictions
    done=true;
}
//Waiting for some MetaSample -> done=false
}
// end c1902

/** computational operations with code -1903 <br><br>
 *
 * this computational code sums up predictions .
 *
 */
public void c1903()
{
    float sum = 0;
    int threshold = 18000;
    boolean endForecast = true;

    checkMatrixes(1901,37);

    designMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(0);
    basicNetMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(36)
        ;
    layer = pendingComputationalSpecificationSet.getOrderLayer();

    // Getting the number of articles
    int counter = (int) designMatrix.getValue(layer,0,0);

    for (int j=1 ; j <=counter ; j++)
    {
        for (int t=1 ; t <=35 ; t++)

```

```

{
    MemoryMatrix agentMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.
        getMemoryMatrixAddress(t);
    //Checking if the licensee made a prediction for this MetaSample
    if(! ( agentMatrix.isEmpty(layer,1,j)))
    {
        sum = sum + agentMatrix.getValue(layer,1,j);
    }
}
//Storing total predictions in basicNet Matrix
basicNetMatrix.setValue(layer,1,j,sum);
//Checking if MetaSample can be produced
if (sum > threshold)
{
    basicNetMatrix.setValue(layer,2,j,1);
    System.out.println("MetaSample_" + j + "_with_layer_" + layer +
        "_can_be_produced ,forecast=" + sum + "_status=1");
}
else
{
    basicNetMatrix.setValue(layer,2,j,0);
    System.out.println("MetaSample_" + j + "_with_layer_" + layer +
        "_can_NOT_be_produced ,forecast=" + sum + "_status=0");
}
//Waiting all predictions
for (int t=1 ; t<=35 ; t++)
{
    MemoryMatrix agentMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.
        getMemoryMatrixAddress(t);

    if(agentMatrix.isEmpty(layer,1,j))
        endForecast = false;
}

if (endForecast)
{
    System.out.println("MetaSample_" + j + "_with_layer_" + layer + "_forecasted_by_all_
        licensee");
    done=true;
}
sum = 0;
}
}
// end c1903

/** computational operations with code -1904 <br><br>
*
* this computational code check treshold and if above
* set 1 to row 2 matrix in pos 1 * and set done=true.
* 1 cell for each number of row read in matrix in pos 0 (0,0),
* whit a counter in position (0,1)
*/
public void c1904()
{
    float val;

```

```

checkMatrixes(1904,2);

designMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(0);
basicNetMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(1);
layer = pendingComputationalSpecificationSet.getOrderLayer();

//Switching to an other article
int i = getCounter(basicNetMatrix, 1);

//check the number of items and the status
if(!designMatrix.getEmpty(layer,0,0) && !basicNetMatrix.getEmpty(layer,2,i))
{
    if(i <= designMatrix.getValue(layer,0,0) && basicNetMatrix.getValue(layer,2,i) == 1)
    {
        System.out.println("Unit_#" + myUnit.getUnitNumber() + "_produced_MetaSample_#" + i
            + "_with_layer_" + layer + ",_status=2");

        basicNetMatrix.setValue(layer,2,i,2);

        done=true;
    }
    else
    {
        System.out.println("Unit_#" + myUnit.getUnitNumber() + "_NOT_produced_MetaSample_#" + i
            + "_with_layer_" + layer + ",_status=1" );
    }
}
}
// end c1904

/** computational operations with code -1905 <br><br>
 *
 * This computational code dials with TC offer-price
 */
public void c1905()
{
    float offerPrice, random;

    checkMatrixes(1905,3);

    designMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(0);
    basicNetMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(1);
    tcMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(2);
    layer = pendingComputationalSpecificationSet.getOrderLayer();

    //Making an offer-price for available articles
    if(!designMatrix.getEmpty(layer,0,0))
    {
        int counter = (int) designMatrix.getValue(layer,0,0);

        for (int i=1; i<=counter; i++)
        {
            if(!basicNetMatrix.getEmpty(layer,2,i))
            {
                if(basicNetMatrix.getValue(layer,2,i) > 0)
                {

```

```

//The offer-price
offerPrice = (int) normal.getSampleWithMean$withVariance(7,10);
//Min value
if (offerPrice <1)
    offerPrice = 1;
//Max value
if (offerPrice >20)
    offerPrice=20;
//Storing the offer-price of the TC
    tcMatrix.setValue(layer , 1 , i , offerPrice);

System.out.println("Unit_#" + myUnit.getUnitNumber() + "_offer_price_for_MetaSample_" + i
+ "_with_layer_" + layer + "is$" + offerPrice);
}
else
    System.out.println("Unit_#" + myUnit.getUnitNumber() + "_can_not_make_an_offer_for_
    Metasample_" + i
+ "_with_layer_" + layer );
}
}
done=true;
}
// Waiting for some MetaSample -> done=false
}
// end c1905

/** computational operations with code -1906 <br><br>
* This computational code deal with the bidding phase
*/
public void c1906()
{
    MemoryMatrix choiceMatrix;

    checkMatrixes(1906,8);

    designMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(0);
    basicNetMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(6);
    orMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(7);
    layer = pendingComputationalSpecificationSet.getOrderLayer();

    // Checking MetaSamples
    if (!designMatrix.getEmpty(layer,0,0))
    {
        int counter = (int) designMatrix.getValue(layer,0,0);

        for (int i=1; i<=counter; i++)
        {
            int choice = 0;
            float offer=0;
            //Checking the status
            if (!basicNetMatrix.getEmpty(layer,2,i))
            {
                if (basicNetMatrix.getValue(layer,2,i) > 0)
                {
                    //Here we make the bidding

```

```

for (int t=1 ; t<=5 ; t++)
{
MemoryMatrix tcMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.
    getMemoryMatrixAddress(t);

if(!tcMatrix.isEmpty(layer ,1 ,i))
{
if(tcMatrix.getValue(layer ,1 ,i) < offer || offer==0)
{
offer = tcMatrix.getValue(layer ,1 ,i);
choice = t;
}
}
}
//Checking the TC chosen
if(!basicNetMatrix.isEmpty(layer ,2 ,i))
{
if(basicNetMatrix.getValue(layer ,2 ,i) >0 && !(choice==0))
{
//Getting its offer price ...
choiceMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress
    (choice);

float offerPrice = choiceMatrix.getValue(layer ,1 ,i);
//... storing it in basicNet Matrix ...
basicNetMatrix.setValue(layer ,3 ,i ,offerPrice);
//... and in the orMatrix
orMatrix.setValue(layer ,1 ,i ,choice);

System.out.println("Trading Company#" + orMatrix.getValue(layer ,1 ,i) +
    " has been chosen for MetaSample#" + i + " with layer#" + layer + " price"
    + basicNetMatrix.getValue(layer ,3 ,i));
}
}
}
}
done=true;
}
// Waiting for some MetaSample -> done=false
}
// end c1906

/** computational operations with code -1907 <br><br>
*
* this computational code dial with licensee's orders.
*
*/
public void c1907()
{
checkMatrixes(1907,3);

designMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(0);
agentMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(1);
basicNetMatrix=(MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(2);
layer = pendingComputationalSpecificationSet.getOrderLayer();
}

```

```

if (!designMatrix.isEmpty(layer, 0, 0))
{
    int j = getCounter(agentMatrix, 2);
    if (!basicNetMatrix.isEmpty(layer, 2, j))
    {
        //Check if the licensee received the related MetaSample
        if (basicNetMatrix.getValue(layer, 2, j) > 1)
        {
            int random = (int) normal.getSampleWithMean$withVariance(500, 900000);
            //Min value
            if (random < 10)
                random = 10;
            //Max value
            if (random > 25000)
                random = 25000;

            agentMatrix.setValue(layer, 2, j, random);

            if (basicNetMatrix.isEmpty(layer, 4, j))
                basicNetMatrix.setValue(layer, 4, j, 0.0);

            float totOrder = basicNetMatrix.getValue(layer, 4, j) + random;
            basicNetMatrix.setValue(layer, 4, j, totOrder);

            System.out.println("Licensee_" + myUnit.getUnitNumber() + "_ordered_" + random + "_for_item_"
                + j + "_with_layer_" + layer + "_totOrder_" + totOrder);

            done=true;
        }
        // The article can not be ordered -> done=false
    }
}
}
// end c1907

/** computational operations with code -1908 <br><br>
 *
 * this computational code dial with or branch choice.
 *
 */
public void c1908()
{
    checkMatrixes(1908, 3);

    designMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(0);
    basicNetMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(1);
    orMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(2);
    layer = pendingComputationalSpecificationSet.getOrderLayer();

    if (!designMatrix.isEmpty(layer, 0, 0))
    {
        //Switching to an other article
        int i = getCounter(orMatrix, 0);
    }
}

```

```

if(i <= designMatrix.getValue(layer,0,0))
{
  if(!basicNetMatrix.getEmpty(layer,2,i))
  {
    //Check if the item can be produced (status>1)
    if(basicNetMatrix.getValue(layer,2,i) > 1 && !orMatrix.getEmpty(layer,1,i))
    {
      float branch = orMatrix.getValue(layer,1,i);
      orMatrix.setValue(layer,1,0,branch);

      System.out.println("Branch_# " + orMatrix.getValue(layer,1,0) +
        " produced item # " + i + " with layer # " + layer);

      basicNetMatrix.setValue(layer,2,i,3.0);

      done=true;
    }
  }
  // The article can not be produced -> done=false
}
}
//end c1908

/** computational operations with code -1910 <br><br>
 * This computational code prints the Report and stops the simulation.
 */
public void c1910()
{
  checkMatrixes(1910,2);

  designMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(0);
  basicNetMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(1);
  layer = pendingComputationalSpecificationSet.getOrderLayer();

  try
  {
    report = new MyWriter("log/report.txt");
    report.write("Layer_# MetaSampleN_# Status0_# Status1_# Status3_#\n");
  }
  catch (IOException e)
  {
    System.out.println(e);
    MyExit.exit(1);
  }

  for(int ln=1; ln <=16; ln++)
  {
    int metaSamplesN=0;
    int [] status;
    status = new int [4];

    for(int s=0; s <=3; s++)
      status[s]=0;
  }
}

```

```

if(!designMatrix.isEmpty(ln, 0, 0))
    metaSamplesN= (int) designMatrix.getValue(ln, 0, 0);

for(int item=1; item<=300; item++)
{
    if(!basicNetMatrix.isEmpty(ln, 2, item))
        status[(int) basicNetMatrix.getValue(ln, 2, item)]++;
}
String text = ln + "|_" + metaSamplesN + "|_" + status[0] + "|_" + status[1] + "|_" +
    status[2]
    + "|_" + status[3] + "\n";

try
{
    report.write(text);
}
catch(IOException e)
{
    System.out.println(e);
    MyExit.exit(1);
}
}
try
{
    report.close();
}
catch(IOException e)
{
    System.out.println(e);
    MyExit.exit(1);
}

System.out.println("*****_THIS_IS_THE_END!_*****");
StartESFrame.eSFrameObserverSwarm.getControlPanel().setStateStopped();

done = true;
}
/* computational operations with code -1911 <br><br>
*
* This computational code calculate gross sales from samples.
*
*/
public void c1911()
{
    checkMatrixes(1911,2);

    designMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(0);
    basicNetMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(1);
    layer = pendingComputationalSpecificationSet.getOrderLayer();

    float samplePrice = (float) normal.getSampleWithMean$withVariance(35,6);
    int counter = getCounter(basicNetMatrix,4);

    logOpen(true, "log/grossSales.txt");
    String text = Globals.env.getCurrentTime() + "|_" + samplePrice + "_\n";

```

```

    try
    {
        grossSalesLog.write(text);
        grossSalesLog.flush();
    }
    catch(IOException e)
    {
        System.out.println(e);
        MyExit.exit(1);
    }
    done = true;
}
//end c1911

/** computational operations with code -1912 <br><br>
 *
 * This computational code calculate gross sales from items.
 *
 */
public void c1912()
{
    checkMatrixes(1912,2);

    designMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(0);
    basicNetMatrix = (MemoryMatrix) pendingComputationalSpecificationSet.getMemoryMatrixAddress(1);
    layer = pendingComputationalSpecificationSet.getOrderLayer();

    int counter = getCounter(basicNetMatrix,5);

    if(!basicNetMatrix.getEmpty(layer,2,counter) && !basicNetMatrix.getEmpty(layer,3,counter) &&
        !basicNetMatrix.getEmpty(layer,4,counter))
    {
        if (basicNetMatrix.getValue(layer,2,counter)==3.0)
        {
            float itemPrice = basicNetMatrix.getValue(layer,3,counter) * basicNetMatrix.getValue(layer,4,
                counter);
            itemSales = (float) (itemPrice * 0.08);
        }
    }
    String text = Globals.env.getCurrentTime() + "\u|w" + itemSales + "\u\n";
    logOpen(false, "log/grossSales.txt");

    try
    {
        grossSalesLog.write(text);
        grossSalesLog.flush();
    }
    catch (IOException e)
    {
        System.out.println(e);
        MyExit.exit(1);
    }
    done =true;
}
//end c1912

```

```

//*****
// Private functions
//*****

//The counter is used in some computational code in order to take actions
// on different article with only a recipe
private int getCounter(MemoryMatrix currentMatrix, int row)
{
    //Zeroing the counter if empty
    if(currentMatrix.getEmpty(layer, row, 0))
        currentMatrix.setValue(layer, row, 0, 0.0);
    //Here we increment the counter
    int counter = (int) currentMatrix.getValue(layer, row, 0);
    counter++;
    //The counter must be less than the number of MetaSample
    if(counter <= designMatrix.getValue(layer, 0, 0))
        currentMatrix.setValue(layer, row, 0, counter);

    return counter;
}

private void checkMatrixes(int code, int numberOfMatrixes)
{
    if(pendingComputationalSpecificationSet.getNumberOfMemoryMatrixesToBeUsed() != numberOfMatrixes)
    {
        System.out.println("Code_" + code + "_requires_" + numberOfMatrixes + "_matrix;_"
            + pendingComputationalSpecificationSet.getNumberOfMemoryMatrixesToBeUsed() + "_found in order"
            + "#_" +
            pendingComputationalSpecificationSet.getOrderNumber());

        MyExit.exit(1);
    }
}

private void logOpen(boolean eraseFile, String fileName)
{
    File fileOut = new File(fileName);
    if(!this.grossSalesLogFileOpen && eraseFile)
    {
        try
        {
            {
                FileWriter erase = new FileWriter(fileOut, false);
                erase.close();
            }
            catch (IOException e)
            {
                System.out.println(e);
                MyExit.exit(1);
            }
        }
        try
        {
            grossSalesLog = new FileWriter(fileOut, true);
            this.grossSalesLogFileOpen = true;
        }
    }
}

```

```

catch (IOException e)
{
    System.out.println(e);
    MyExit.exit(1);
}
}
}

```

OrderDistiller.java

Si riporta il codice sorgente del file *OrderDistiller.java*, aggiornato alla versione utilizzata in *BasicJes-0.9.7.52*.

Listing 6.30. *ComputationalAssembler.java*

```

//OrderDistiller.java modified by Pietro Terna (look below at rows signed //pp)
//OrderDistiller.java modified by Marco Lamieri and Francesco Merlo (look below at rows signed //
    lm)
import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;
import swarm.collections.ListImpl;

/**
 * OrderDistiller.java
 *
 *
 * Created: Wed May 08 15:29:12 2002
 *
 * @author<br/>
 * Cristian Barreca<a href="mailto:dgbarrec@libero.it"></a></br>
 * Elena Bonessa<a href="mailto:elena.bonessa@infinito.it"></a></br>
 * Antonella Borra<a href="mailto:anborra@libero.it"></a></br>
 * Modified by:</br>
 * Marco Lamieri<a href="mailto:lamieri@econ.unito.it"></a></br>
 * Francesco Merlo<a href="mailto:merlo@econ.unito.it"></a>
 * @version 0.9.7.31.b
 */

/**
 * This class is used to read data from two worksheets. <br/>The first one contains the
 * list of recipes of
 * our virtual enterprise.<br/>The second one contains a sequence of orders to be launched,
 * shift by shift, in order to make the daily production activities.*
 */

public class OrderDistiller extends OrderGenerator
{

```

```

ExcelReader orderSequenceWorksheet ,
    recipeWorksheet;

String semicolon = ";",
    gate = "#",
    layer="1",
    computation="c",
    orderSequences1 = "recipeData/orderStartingSequence.xls",
    orderSequences2 = "recipeData/orderSequence.xls",
    recipeFile = "recipeData/recipes.xls",
    currentOrderSequenceWorksheet;

Recipe aRecipe;

public int orderCount = 0, currentLayer = 0;

public boolean firstTime = true, orderSequenceWorksheetOpen = false;

public Order anOrder;

public ListImpl unitList ,
    endUnitList ,
    orderList ,
    recipeList;

int [] currentOrder;

AssigningTool assigningTool; //pt

Unit aUnit;

// CONSTRUCTOR
public OrderDistiller (Zone aZone , int msn, int msl, ListImpl ul,
    ListImpl eul, ListImpl ol, int tln, ESFrameModelSwarm mo, AssigningTool at)
{ //pt
    super(aZone, msn, msl, ul, eul, ol, tln, mo, at);

    unitList=ul;
    endUnitList=eul;
    orderList=ol;
    assigningTool=at; //pt
}

/**This is the method containing the iterator needed to launch the daily production of recipes.
 * It take a look at the orderSequence arrays to determine which recipes must be done and how
    many times.
 * A request for which unit can do the first production phase of each recipe will be done to
    units or endUnits.
 */
public void distill()
{
    if(firstTime == true)
    {
        currentOrderSequenceWorksheet = orderSequences1;

        firstTime = false;
    }
}

```

```

else
    currentOrderSequenceWorksheet = orderSequences2;

if (!orderSequenceWorksheetOpen)
{
    orderSequenceWorksheet = new ExcelReader(currentOrderSequenceWorksheet);
    orderSequenceWorksheetOpen = true;
}

checkForComment();
int shiftNumber = orderSequenceWorksheet.getIfInteger();

boolean isEndOfShift = false;

while (!isEndOfShift)
{
    if (orderSequenceWorksheet.checkForLabelCell())
    {
        String aString = orderSequenceWorksheet.getStrValue();
        if (aString.equals("semicolon"))
        {
            isEndOfShift = true;
            break;
        }
        else
        {
            orderSequenceWorksheet.goBack();
        }
    }
}

readOrderFrom(orderSequenceWorksheet);

boolean recipeCodeFound = false;
for (int r = 0; r < recipeList.getCount() && !recipeCodeFound; r++)
{
    aRecipe = (Recipe) recipeList.atOffset(r);
    if (aRecipe.getRecipeCode() == currentOrder[0])
        recipeCodeFound = true;
}

if (recipeCodeFound == false)
{
    System.err.println("OrderDistiller error: There is no Recipe with code'" + currentOrder
        [0] +
        "'_check_" + recipeFile + "'_and_" + currentOrderSequenceWorksheet + "'");
    MyExit.exit(1);
}

for (int q = 0; q < currentOrder[1]; q++)
{
    orderCount++;

    anOrder = new Order(getZone(), orderCount, Globals.env.getCurrentTime(),
        aRecipe.getLength(), aRecipe.getRecipeSteps(), eSFrameModelSwarm, endUnitList);

    anOrder.setRecipeName(aRecipe.getRecipeName());
}

```

```

// setting the layer (from 0 to totalLayerNumber-1)
anOrder.setOrderLayer(currentLayer);
// add the active orders to the general order list (they will be
// eliminated when dropped in a unit, being finished); this
// list has been introduced for accounting purposes [may be it would
// be better substitute it with a get to the units to know their
// waiting lists]
orderList.addLast(anOrder);
// sending the order to the first production unit (we are acting as the Front End of the ES)
assigningTool.assign(anOrder);

if(StartESFrame.verbose)
{
    System.out.println("OrderDistiller :_Order_" + anOrder.getOrderNumber() +
        "_with_Name_" + anOrder.getRecipeName() +
        "_and_layerNumber_" + anOrder.getOrderLayer() + "_is_starting_production");
}
}
}
if(orderSequenceWorksheet.eof())
    orderSequenceWorksheetOpen = false;
}
/**
 * This method is used to collect the names of the units and of the end units, so that it can
 * operate the check of
 * correspondency between the production phases required by recipes and the phases of production
 * the units can do.
 */
public void setDictionary()
{
    super.setDictionary();
    //The List of Recipes
    recipeList = new ListImpl(getZone());

    recipeWorksheet = new ExcelReader(recipeFile);

    while(! recipeWorksheet.eof())
    {
        aRecipe = new Recipe(getZone());
        aRecipe.setRecipeFrom(recipeWorksheet);
        recipeList.addLast(aRecipe);
    }

    if(StartESFrame.verbose)
    {
        System.out.println("OrderDistiller:_ " + recipeList.getCount() + "recipes_readed.");
        for (int i = 0; i < recipeList.getCount(); i++)
        {
            aRecipe = (Recipe) recipeList.atOffset(i);
            System.out.println("Recipe_" + i + "_with_name_" + aRecipe.getRecipeName() +
                "_and_code_" + aRecipe.getRecipeCode() + "");
        }
    }
}
}
// *****
// PRIVATE FUNCTIONS

```

```

// *****
private int [] readOrderFrom (ExcelReader e)
{
    orderSequenceWorksheet = e;
    currentOrder = new int [2];

    checkForComment ();
    checkForLayer ();
    currentOrder [0] = orderSequenceWorksheet .getIfInteger ();
    orderSequenceWorksheet .getIfString ();
    currentOrder [1] = orderSequenceWorksheet .getIfInteger ();

    return currentOrder;
}

private void checkForLayer ()
{
    if (orderSequenceWorksheet .checkForLabelCell ())
    {
        String isLayer = orderSequenceWorksheet .getStrValue ();

        if (isLayer .equals (layer))
        {
            currentLayer = orderSequenceWorksheet .getIntValue ();

            if (StartESFrame .verbose)
                System .out .println ("OrderDistiller :_Current_Layer_now_is_#" + currentLayer);
        }
        else
        {
            orderSequenceWorksheet .goBack ();
        }
    }
}

private void checkForComment ()
{
    if (orderSequenceWorksheet .checkForLabelCell ())
    {
        String isGate = orderSequenceWorksheet .getStrValue ();
        if (isGate .equals (gate))
        {
            if (StartESFrame .verbose)
                System .out .println ("OrderDistiller :_There_is_a_comment ");

            while (orderSequenceWorksheet .eol ())
            {
                String comment = orderSequenceWorksheet .getStrValue ();
                System .out .println (">>>" + comment + "<<<");
            }
        }
        else
        {
            orderSequenceWorksheet .goBack ();
        }
    }
}

```

```
}  
} // OrderDistiller
```

Recipe.java

Si riporta il codice sorgente del file *Recipe.java*, aggiornato alla versione utilizzata in *BasicJes-0.9.7.52*.

Listing 6.31. Recipe.java

```
import swarm.defobj.Zone;  
import swarm.objectbase.SwarmObjectImpl;  
/**  
 * Recipe.java  
  
 * @author<br/>  
 * Cristian Barreca<br/>  
 * Elena Bonessa<br/>  
 * Antonella Borra<br/>  
 * Modified by:<br/>  
 * Marco Lamieri<br/>  
 * Francesco Merla</a>  
 * @version 0.9.7.5  
 */  
/**This class is used to record the recipes and their referring number (ID), so we can assign  
 them to a List*/  
public class Recipe extends SwarmObjectImpl  
{  
    private String  
        semicolon = ";",  
        gate = "#",  
        p = "p",  
        end = "e",  
        sec = "s",  
        min = "m",  
        slash = "/",  
        or = "||",  
        computation = "c",  
        recipeName,  
        currentCell,  
        timeUnit,  
        //backslash = "\\|";  
  
    private ExcelReader recipeWorkSheet;  
  
    private int
```

```

timeNumber,
stepNumber,
recipeCode,
recipeLength;

private int []
steps,
recipeSteps;

//CONSTRUCTOR
public Recipe(Zone aZone)
{
    super(aZone);
    recipeSteps = new int [0];
}
// *****
// PUBLIC METHODS
// *****
public void setRecipeFrom(ExcelReader e)
{
    recipeWorkSheet = e;
    currentCell="";

    recipeName = recipeWorkSheet.getIfString();
    recipeCode = recipeWorkSheet.getIfInteger();

    while(! currentCell.equals(semicolon))
    {
        if(recipeWorkSheet.checkForLabelCell())
        {
            currentCell = recipeWorkSheet.getStrValue();

            if(currentCell.equals(p))
                procurement();
            else if(currentCell.equals(or))
                or();
            else if(currentCell.equals(end))
                end();
            else if(currentCell.equals(computation))
                computation();
            else if(currentCell.equals(semicolon))
                break;
            else
            {
                System.err.println("Recipe_error: unknown sign " + currentCell + ".");
                MyExit.exit(1);
            }
        }
        else
        {
            currentCell = recipeWorkSheet.getStrValue();
            number();
        }
    }

    if(StartESFrame.verbose)
    {

```

```

        System.out.println("Recipe : intermediate format is");
        for(int s=0; s<steps.length; s++)
            System.out.print(steps[s] + "\n");
        System.out.println();
    }

    addSteps();

}

if(StartESFrame.verbose)
{
    System.out.println("Recipe : the new recipe is");
    for(int s=0; s<recipeSteps.length; s++)
        System.out.print(recipeSteps[s] + "\n");
    System.out.println();
}
}

public String getRecipeName()
{
    return recipeName;
}

public int getRecipeCode()
{
    return recipeCode;
}

public int [] getRecipeSteps ()
{
    return recipeSteps;
}

public int getLength()
{
    return recipeSteps.length;
}

// *****
// PRIVATE FUNCTIONS
// *****
private void addSteps()
{
    int [] newRecipeSteps;
    int newRecipeLength = recipeSteps.length + steps.length;
    newRecipeSteps = new int [newRecipeLength];
    System.arraycopy(recipeSteps, 0, newRecipeSteps, 0, recipeSteps.length);
    System.arraycopy(steps, 0, newRecipeSteps, recipeSteps.length, steps.length);
    recipeSteps = new int [newRecipeLength];
    System.arraycopy(newRecipeSteps, 0, recipeSteps, 0, newRecipeLength);
}

private int [] procurement()
{
    int numberOfStepsForProcurement = recipeWorkSheet.getIfInteger();
}

```

```

steps = new int [ numberOfStepsForProcurement + 2];

steps [0] = -1;
steps [1] = numberOfStepsForProcurement;

for (int p=0; p<numberOfStepsForProcurement; p++)
    steps [p+2]= recipeWorkSheet . getIfInteger ();

System . out . print ("Recipe :_the_step_is_'p" + numberOfStepsForProcurement + "_");
for (int p=2; p<steps . length ; p++)
    System . out . print (steps [p]+"_");
System . out . println ("");

return steps;
}

private int [] number ()
{
    stepNumber = Integer . parseInt (currentCell);
    //stepNumber = currentCell;
    timeUnit = recipeWorkSheet . getIfString ();
    timeNumber = recipeWorkSheet . getIfInteger ();

    String isBatch = recipeWorkSheet . getStrValue ();

    if (isBatch . equals (slash))
    {
        int items = recipeWorkSheet . getIfInteger ();
        String isEnd = recipeWorkSheet . getIfString ();
        if (!isEnd . equals (end))
        {
            System . err . println ("Recipe_error :_there_is_no_endUnit_in_" + recipeName);
            MyExit . exit (1);
        }
        int endUnit = recipeWorkSheet . getIfInteger ();
        steps = new int [5];
        steps [0] = -2;
        steps [1] = timeNumber;
        steps [2] = items;
        steps [3] = stepNumber;
        steps [4] = endUnit;

        System . out . println ("Recipe :_the_step_is_" + stepNumber + "_" + timeUnit + "_" + timeNumber +
            "_/" + items + "_e_" + endUnit + "'");
    }
    else if (isBatch . equals (backslash))
    {
        int items = recipeWorkSheet . getIfInteger ();
        steps = new int [4];
        steps [0] = -3;
        steps [1] = timeNumber;
        steps [2] = items;
        steps [3] = stepNumber;

        System . out . println ("Recipe :_the_step_is_" + stepNumber + "_" + timeUnit + "_" + timeNumber +
            "_\\_" + items + "'");
    }
}

```

```

}
else
{
    recipeWorkSheet.goBack();
    System.out.println("Recipe : the step is " + stepNumber + " " + timeUnit + " " + timeNumber +
        "");
    if(timeUnit.equals(min))
        timeNumber = timeNumber*60;
    else if(!timeUnit.equals(sec))
    {
        if(StartESFrame.verbose)
            System.out.println("Time is not expressed in minutes or seconds . Check the worksheet");
        System.exit(1);
    }
    if(timeNumber==0)
    {
        steps = new int [1];
        steps[0] = 1000000000 + stepNumber;
    }
    else
    {
        steps = new int [timeNumber];
        for(int s=0; s<timeNumber; s++)
            steps[s] = stepNumber;
    }
}
return steps;
}

private int [] computation()
{
    // Getting the computation code
    int computationCode = recipeWorkSheet.getIntInteger();
    // Getting the number of matrixes used for computation
    int numberOfMatrixesForComputation = recipeWorkSheet.getIntInteger();
    // Creating an array with the name of matrixes
    int [] matrixesForComputation = new int [numberOfMatrixesForComputation];
    // Getting the matrixes
    for(int m = 0; m < numberOfMatrixesForComputation; m++)
        matrixesForComputation[m] = recipeWorkSheet.getIntInteger();
    // Getting the production step
    int stepNumber = recipeWorkSheet.getIntInteger();
    // Getting the time unit
    String timeUnit = recipeWorkSheet.getString();
    int timeNumber = recipeWorkSheet.getIntInteger();

    System.out.print("Recipe : the step is 'c" + computationCode + " " +
        numberOfMatrixesForComputation + " " );
    for(int c=0; c<numberOfMatrixesForComputation; c++)
        System.out.print(matrixesForComputation[c] + " ");
    System.out.println(stepNumber + " " + timeUnit + " " + timeNumber + "'");

    if(timeUnit.equals(min))
        timeNumber = timeNumber*60;
}

```

```

if(timeNumber==0)
{
    steps = new int [numberOfMatrixesForComputation + 4];
    steps [0] = 1000000000 + stepNumber;
    steps [1] = -1 * computationCode;
    steps [2] = numberOfMatrixesForComputation;
    for(int m = 0; m < numberOfMatrixesForComputation; m++)
        steps [3+m] = matrixesForComputation[m];
    steps [numberOfMatrixesForComputation + 3] = 1000000000 + stepNumber;
}
else
{
    steps = new int [timeNumber + numberOfMatrixesForComputation + 3];
    for(int s=0; s<timeNumber; s++)
        steps [s] = stepNumber;

    steps [timeNumber] = -1 * computationCode;
    steps [timeNumber+1] = numberOfMatrixesForComputation;
    for(int m = 0; m < numberOfMatrixesForComputation; m++)
        steps [timeNumber+2+m] = matrixesForComputation[m];
    steps [timeNumber + numberOfMatrixesForComputation + 2] = 1000000000 + stepNumber;
}
return steps;
}

private int [] or()
{
    steps = new int [2];
    steps [0] = -10;
    steps [1] = recipeWorkSheet . getIfInteger ();

    System.out.println("Recipe: the step is " + steps [1] + "");

    return steps;
}

private int [] end()
{
    steps = new int [1];
    steps [0] = recipeWorkSheet . getIfInteger ();

    System.out.println("Recipe: the step is 'e'" + steps [0] + "");

    return steps;
}
} //Recipe.java

```

Modifiche a `ExcelReader.java`

Si riportano le modifiche al codice sorgente del file `ExcelReader.java`, sviluppato da Michele Sonnessa, utili al funzionamento delle classi `OrderDistiller` e `Recipe`. Le modifiche sono aggiornate alla versione `BasicJes-0.9.7.52`.

Listing 6.32. `ExcelReader.java`: modifica 1

```
public int getIfInteger()
{
    if (checkLabelOfCell(currentX))
    {
        System.err.println("ExcelReader_Error: the cell should contain an integer value!");
        System.err.println("Sheet '" + currentSheet.getName() + "' at row=" + currentY + ": col=" +
            currentX);
        MyExit.exit(1);
    }
    return readIntValue(currentX);
}
```

Listing 6.33. `ExcelReader.java`: modifica 2

```
public String getIfString()
{
    if (!checkLabelOfCell(currentX))
    {
        System.err.println("ExcelReader_Error: the cell should contain a string value!");
        System.err.println("Sheet '" + currentSheet.getName() + "' at row=" + currentY + ": col=" +
            currentX);
        MyExit.exit(1);
    }
    return readStrValue(currentX);
}
```

Listing 6.34. `ExcelReader.java`: modifica 3

```
public void goBack()
{
    if (currentX == 0)
    {
        changeRow(currentY - 1);
    }
}
```

```
    currentX = maxX;
  }
  else
    currentX = currentX - 1;
}
```