

UNIVERSITÀ DEGLI STUDI DI TORINO

Facoltà di Economia

Corso di Laurea in Economia e Commercio

Tesi di Laurea

Simulazione Agent Based in contesti d'impresa

Applicazione del modello Virtual Enterprise in JavaSwarm ad
un'azienda reale

Relatore:

prof. Pietro Terna

Correlatore:

prof. Sergio Margarita

Candidato:

Marco LAMIERI

ANNO ACCADEMICO 2001-2002

RINGRAZIAMENTI

Desidero ringraziare la BasicNet s.p.a. che, tramite la dott.sa Paola Bruschi, ha fornito i dati necessari alla realizzazione della simulazione.

Ringrazio il dott. Michele Sonnessa e Alessandro Raimondi per le utili informazioni riguardo i progetti StarLogoJve e MatLabJve.

Ringraziamenti particolari vanno a Francesco Merlo, mio compagno nella preparazione di questo lavoro, e al prof. Pietro Terna, relatore della tesi.

SOMMARIO

Capitolo 1

In questo capitolo si introducono le teorie della complessità. Si cercherà di comprendere il significato delle simulazioni fondate su agenti adattivi, e quali vantaggi portano alle scienze sociali, con particolare attenzione all'economia

Capitolo 2

Obiettivo di questo capitolo è descrivere alcuni dei principali formalismi utilizzati nelle simulazioni Agent Based, come Java,UML, Swarm e gli strumenti con i quali è stata sviluppata la Virtual Enterprise.

Capitolo 3

In questo capitolo si intendono introdurre alcune teorie dell'impresa, partendo dal lavoro di Coase, che definisce l'impresa attraverso i costi di transazione, sino all'ordine spontaneo di Hayek e la scuola austriaca in economia.

Capitolo 4

In questo capitolo si descrive l'essenza del modello di Java Virtual Enterprise (Terna, 2002), nato nel corso dell'anno 2000 all'interno del Dipartimento di Scienze economiche e finanziarie G.Prato dell'Università degli Studi di Torino.

Capitolo 5

In questo capitolo si descrive come è stato applicato il modello Java Virtual Enterprise ad un'azienda reale, la Basic Net s.p.a..

Capitolo 6

In questo capitolo si affronterà lo stato dell'arte della simulazione fondata su agenti applicata all'economia, e si cercheranno di trarre alcune, seppur provvisorie, conclusioni sul lavoro svolto.

INDICE

<i>Ringraziamenti</i>	I
<i>Sommario</i>	III
<i>1. Sistemi complessi e simulazione ad agenti in economia</i>	1
1.1 Le scienze cognitive	3
1.2 Sistemi adattivi e sistemi complessi	5
1.3 La modellizzazione della realtà	8
1.4 Agenti semplici e razionalità limitata	11
<i>Bibliografia</i>	17
<i>2. Gli strumenti per la simulazione</i>	19
2.1 La programmazione orientata agli oggetti e il linguaggio Java	20
2.1.1 Tutto è un oggetto	23
2.1.2 Java e l'indipendenza dalla piattaforma	29
2.1.3 Java in pratica	31
2.1.4 Extreme Programming	39
2.2 Swarm	42
2.2.1 Sciami di agenti	43
2.2.2 Perché Java	47
2.2.3 Swarm, un progetto libero	48
2.2.4 Creare un universo virtuale con Swarm	49

2.3	UML	71
2.3.1	Fasi dello sviluppo	76
2.3.2	Visual Modelling	80
2.3.3	Tool di Sviluppo	83
2.3.4	Vista UML di JVEFrame	88
2.4	Altri strumenti per la simulazione: StarLogo, Ascape, RePast, Extend	88
2.4.1	Starlogo	88
2.4.2	Ascape	91
2.4.3	RePast	91
2.4.4	Extend	93
	<i>Bibliografia</i>	97
3.	<i>L'azienda come sistema complesso</i>	101
3.1	La teoria dell'impresa di Coase	102
3.2	Come l'azienda crea conoscenza tacita ed esplicita	103
3.3	Hayek e la scuola austriaca	106
	<i>Bibliografia</i>	113
4.	<i>Il modello Java Virtual Enterprise</i>	115
4.1	La metafora Virtual Enterprise	116
4.1.1	Lo schema ERA	117
4.1.2	La separazione What to Do/Doing What	120
4.1.3	I magazzini	122
4.1.4	La trasmissione della conoscenza	124
4.1.5	I meccanismi contabili	126
4.1.6	Produzioni batch	128
4.1.7	I sub-processi	129

4.1.8	Le capacità di scelta tra diverse lavorazioni possibili	130
4.1.9	I layer	131
4.1.10	Gli oggetti computazionali	132
4.1.11	La scelta del codice esterno, intermedio e interno	134
4.2	Altri modelli di simulazione d'impresa	136
4.2.1	JVE in MatLab-SimuLink-StateFlow	136
4.2.2	JVE in StarLogo	138
4.2.3	NIIP	140
	<i>Bibliografia</i>	143
5.	<i>BasicJVE - Analisi della BasicNet in prospettiva di formalizzazione del modello JVEFrame</i>	145
5.1	Breve storia dell'azienda	146
5.2	La BasicNet vista dall'interno	148
5.2.1	Introduzione al modello di business	148
5.2.2	Posizionamento dei marchi	149
5.2.3	Approvvigionamento dei prodotti	150
5.2.4	Sviluppo e gestione del Network	151
5.2.5	Fasi del processo organizzativo	151
5.2.6	Ruolo delle divisioni dotcom	153
5.3	Il processo di formalizzazione: da BasicNet a BasicJve	157
5.3.1	Scopo della simulazione	157
5.4	Processo di formalizzazione delle ricette	159
5.4.1	Prima formalizzazione	160
5.4.2	Seconda formalizzazione	165
5.4.3	Terza formalizzazione	169
5.4.4	Quarta formalizzazione	173

5.4.5	Quinta formalizzazione	175
5.5	Proposte di modifica al codice	178
5.5.1	Prime proposte	178
5.5.2	Le modifiche apportate	181
5.6	BasicJve-0.9.7.30.b: BasicJVE la simulazione di BasicNet	182
5.6.1	Dibattito sull'uso dei layers	183
5.6.2	Il modello: le unità	185
5.6.3	Il modello: le ricette	190
5.6.4	Il modello: le matrici di memoria	200
5.6.5	Il modello: la sequenza degli ordini	205
5.6.6	Il modello: i parametri della simulazione	205
5.7	Le modifiche al codice	210
5.7.1	Modifiche a "OrderDistiller.java"	210
5.7.2	Modifiche a "Recipe.java"	216
5.7.3	La classe ComputationalAssembler.java	218
5.8	Esperimenti e analisi dei risultati	241
5.8.1	Primo esperimento	242
	<i>Bibliografia</i>	247
6.	<i>Conclusioni</i>	249
6.1	Considerazioni sulla simulazione BasicJVE	249
6.2	Possibili sviluppi	251
	<i>Bibliografia</i>	253
7.	<i>Appendice</i>	255
7.1	Listati	255
7.1.1	ComputationalAssembler.java	255

7.1.2	OrderDistiller.java	277
7.1.3	Recipe.java	300

ELENCO DELLE TABELLE

1.2	Risultati delle simulazioni del modello SUM con agenti dotati di mente e privi di mente in diversi contesti	14
5.1	Esempio dei file ”map” per coordinare gli articoli e le collezioni descritte in <i>orderSequences.xls</i>	177
5.2	Esempio di <i>orderSequences.xls</i>	177
5.3	Calendario stimato per le principali attività organizzative	191
5.4	Analisi del prezzo degli articoli	191
5.5	Analisi degli ordini dei singoli licenziatari in un semestre	192
5.6	Analisi degli ordini dei singoli licenziatari su una collezione grande in un semestre	193
5.7	Analisi degli ordini dei singoli licenziatari su una collezione piccola in un semestre	194
5.8	Numero articoli e quantità ordinate per le collezioni Kappa in un semestre	196
5.9	Numero articoli e quantità ordinate per le collezioni Robe di Kappa in un semestre	197
5.10	Stime dei tempi di produzione per ordine e singolo articolo	198
5.11	Stime dei tempi di produzione di ogni Trading Company per ordine e singolo articolo	198
5.12	Le ricette utilizzate nella simulazione	200
5.13	Matrice di memoria <i>designMatrix</i>	202

5.14	Matrice di memoria <i>agentMatrix</i>	203
5.15	Matrice di memoria <i>tcMatrix</i>	203
5.16	Matrice di memoria <i>basicNetMatrix</i>	204
5.17	Matrice di memoria <i>orMatrix</i>	204
5.18	Esempio di sequenza degli ordini per una collezione di 300 articoli . .	206
5.19	Esempio di sequenza degli ordini per una collezione di 30 articoli . . .	207

ELENCO DELLE FIGURE

2.1	Esempio di ereditarietà della classe <i>Shape</i> in notazione UML	27
2.2	Compilare un programma tradizionale	30
2.3	Compilare un programma Java	31
2.4	Prototipo di AB Model secondo lo schema Bottom-Up	44
2.5	Gerarchia multi-livello di una simulazione con Swarm	46
2.6	Evoluzione dell'UML	74
2.7	Diagrammi UML	81
2.8	StarLogo Window	90
2.9	Esempio di simulazione con RePast	94
2.10	Esempio di simulazione di processo con Extend	95
4.1	Schema ERA	118
4.2	Ricette e unità in JVE	120
4.3	Esempio di interazione tra ordini, ricette e unità	123
4.4	Il problema del knowledge management in JVE	125
4.5	I meccanismi contabile di JVE	127
4.6	Motore del modello JVE realizzato in MatLab-SimuLink	138
4.7	Control center del modello JVE realizzato in StarLogo	139
4.8	Finestra "plot" del modello JVE realizzato in StarLogo	140
5.1	Loghi di Proprietà Basic Net	149
5.2	Organigramma del modello di business BasicNet	156
5.3	La simulazione BasicJVE in corso	241

5.4	Rapporto tra tempi della ricetta e tempi di produzione	243
5.5	Tempi di attesa minimi, massimi e medi degli ordini	244

Capitolo 1

SISTEMI COMPLESSI E SIMULAZIONE AD AGENTI IN ECONOMIA

In questo capitolo si introducono le teorie della complessità, analizzando i contributi di Holland (1975) e Arthur, Durlauf, and Lane (1997). Si cercherà di comprendere il significato delle simulazioni fondate su agenti adattivi, e quali vantaggi portano alle scienze sociali, con particolare attenzione all'economia. Si discuterà della razionalità limitata e del contributo delle scienze cognitive alla ricerca economica. Si cercherà di descrivere lo stato dell'arte nella costruzione di modelli agent-based.

L'economia è detta la 'triste scienza'. Triste perché studia la migliore allocazione delle risorse, partendo dall'assunto che queste sono limitate. Comprende, quindi, intrinsecamente l'esistenza della povertà. Scienza perché pretende di spiegare la relazione tra variabili economiche con formule e modelli che sono considerati scientifici.

Nella formulazione di una teoria o un modello della realtà definito scientifico si deve cercare di essere oggettivi. Questo significa premunirsi contro i rischi di errori sistematici e concettuali. Gli strumenti che vengono utilizzati per garantire l'oggettività sono fondamentalmente tre: fare affidamento al carattere collettivo della ricerca, usare il linguaggio quantitativo dei numeri, e adottare il metodo degli esperimenti di laboratorio.

L'economia certamente utilizza il primo criterio, ritiene di utilizzare il secondo mentre, nell'impostazione classica, non utilizzava gli esperimenti di laboratorio per difficoltà pratiche e inadeguatezza del laboratorio per lo studio di fenomeni sociali.

Le scienze sociali, a volte, affrontano fenomeni intrinsecamente qualitativi e non quantitativi, legati al comportamento soggettivo degli attori economici, difficilmente misurabile e legabile con una relazione causa effetto, alle azioni da questi compiute.

Questo non è un piccolo problema perché, come afferma Feynman (2000):

Il principio cardine della scienza, quasi la sua definizione, è che la verifica di tutta la conoscenza è l'esperimento. L'esperimento è il solo giudice della verità scientifica.

Una buona teoria o modello per essere rigorosamente fondato deve essere:

- **Preciso:** esprimersi con il linguaggio dei numeri, e essere una teoria quantitativa.
- **Semplice:** cioè comprensibile da altri scienziati.

- **Elegante:** cioè spiegare la realtà nel modo più generale possibile.
- **Falsificabile secondo Popper:** la scienza produce solo risultati provvisori che non sono ancora stati falsificati. Un modello è fondato se può essere falsificato da un altro ricercatore.
- **Essere in grado di fare buone previsioni:** un buon modello consente di fare delle previsioni quantitative dei fenomeni, che possono essere empiricamente verificate.

Le scienze naturali possono dirsi, Colombatto (2001), scienze avanzate a pieno diritto perché formulano teorie rigorosamente fondate e plausibili, mentre le scienze sociali, tra le quali l'economia, limitano fortemente i dati empirici con i quali confrontano le loro teorie e, seguendo l'approccio classico, sono destinate ad un'inevitabile superficialità, perché spiegano la realtà costruendo sistemi di equazioni con una variabile in funzione di altre, o del tempo, ma senza spiegare come le diverse variabili sono connesse tra loro e in che modo si evolvono seguendo certe traiettorie di covariazione. Quando si passa da un livello di spiegazione macro ai comportamenti umani che li generano ci si scontra con alcuni assunti inaccettabili, che minano alla base i fondamenti stessi delle teorie economiche, primo tra tutti la razionalità olimpica e la macchina ottimizzatrice di Lord Robins.

Scienze come fisica o chimica si trovano di fronte ad un numero di variabili e ad un grado di complessità inferiori rispetto a quello in cui si incappa nello studio della mente (psicologia cognitiva), della vita (biologia evolutiva) o della società (scienze sociali).

1.1 *Le scienze cognitive*

Le scienze cognitive, secondo la definizione proposta da Legrenzi (2002), hanno come oggetto di studio la cognizione, cioè la capacità di un qualsiasi sistema, naturale

o artificiale, di comunicare a se stesso e agli altri ciò che conosce. La natura di questa capacità è stata, in vari modi, investigata da psicologi, filosofi, informatici, economisti, linguisti, antropologi e biologi. Queste discipline hanno una loro storia consolidata e metodi di studio collaudati. Le scienze cognitive non sono la semplice somma di questi saperi, bensì la confluenza degli studi di molte discipline su alcuni problemi specifici: i processi cognitivi.

E' importante la terminologia utilizzata, è diverso parlare di 'scienza cognitiva' al singolare o di 'scienze cognitive' al plurale.

Per 'scienza cognitiva' in senso stretto si intende (Legrenzi, 2002) lo studio di un qualsiasi sistema, naturale o artificiale, che sia in grado di filtrare e ricevere informazioni dall'ambiente circostante (percezione e selezione delle informazioni), di rielaborarle creandone di nuove (pensiero), di archivarle e cancellarle (ricordo e oblio), di comunicarle ad altri sistemi naturali o artificiali e, infine, di prendere decisioni e di agire nel mondo adattandosi ai suoi cambiamenti (decisione e azione) e adattando il mondo a se stesso grazie alla creazione di artefatti.

Per 'scienze cognitive' si intende un campo di studio assai più ampio che comprende tutto ciò che ha a che fare con la capacità creativa dell'uomo e con gli artefatti da lui creati.

I processi cognitivi rivestono una importanza cruciale nella comprensione delle dinamiche dei sistemi socio-economici. Proprio attraverso i processi cognitivi gli agenti economici possono apprendere le informazioni necessarie per prendere decisioni economiche. Si può parlare di apprendimento ogni qual volta un agente economico ha una comprensione imperfetta o incompleta del mondo in cui opera, per diversi motivi:

- Per la mancanza di parte dell'informazione rilevante per le sue decisioni,
- A causa di un'imperfetta conoscenza della struttura del mondo,

- Perché dispone di un repertorio di azioni limitato rispetto a quelle virtualmente accessibili ad un decisore onnisciente,
- Perché ha una conoscenza imprecisa e parziale dei propri obiettivi e delle proprie preferenze.

L'apprendimento viene definito come il processo dinamico di modificazione di tale conoscenza.

Le scienze cognitive studiano le varie forme di sapere tacito e esplicito che viene generato dalle organizzazioni. Così come l'individuo non è consapevole dei suoi meccanismi cognitivi, le organizzazioni non sempre posseggono un sapere codificato del loro modo di comportarsi e decidere. La questione è stata studiata anche in funzione dei suoi risvolti applicativi: essendo i saperi taciti¹ molto più difficili da trasferire di quelli codificati, le scienze cognitive possono fornire gli strumenti per comprendere come avviene questo difficile processo.

Le scienze cognitive, infine, non si occupano solo di studiare i meccanismi cognitivi della mente, ma hanno cercato di trasferire la nozione di adattamento, che è cruciale per comprendere l'evoluzione naturale della specie, allo studio dei sistemi complessi. Per questo motivo fanno parte delle scienze cognitive lo studio degli algoritmi genetici, dei sistemi classificatori e, più in generale, delle simulazioni agent-based.

1.2 Sistemi adattivi e sistemi complessi

I sistemi complessi adattivi sono stati studiati da Holland (1975), che li definisce:

gruppi di agenti legati in un processo di co-adattamento, in cui le mosse di adattamento di ciascuno hanno conseguenze per l'intero gruppo di individui.

¹ Per maggiori informazioni sulla conoscenza tacita e esplicita si veda la sezione 3.2

Holland (1975), mostrava che, sotto determinate condizioni, modelli semplici presentano sorprendenti capacità di auto-organizzazione.

I sistemi complessi sono strettamente correlati con i sistemi non lineari. La definizione di sistema non lineare di Holland (1975) afferma che:

un sistema non lineare è un sistema il cui comportamento non è uguale alla somma delle singole parti che lo compongono.

Se, dunque, per studiare i sistemi lineari, 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 comportamento di tali sistemi, infatti, dipende dall'interazione delle parti, più che dal comportamento delle parti stesse. Occorre quindi considerare il sistema non lineare come un tutto non uguale alla somma delle parti e, dunque, occorre focalizzarsi sulle dinamiche di interazione fra gli elementi che compongono il sistema.

I fenomeni complessi non possono essere studiati con strumenti matematici tradizionali, ma si possono analizzare osservando l'interazione degli elementi del sistema, nel tentativo di scorgere una qualche coerenza. Questo tipo di coerenze, tipiche dei sistemi complessi, vengono definite 'fenomeni emergenti'.

Il termine 'complesso' non è sinonimo di 'complicato', nell'accezione di difficile. Per complessità si intende un fenomeno matematicamente definibile o un aggregato organico e strutturato di parti tra loro interagenti, che assume proprietà non derivanti dalla semplice somma delle parti che lo compongono. Come esempio si immagini il motore di un'automobile, composto da molti meccanismi, anche sofisticati. Il motore viene definito 'macchina banale' perché il suo funzionamento, per quanto difficile e complicato, è il frutto della somma tra le parti che lo compongono, e può essere studiato e scomponendolo in queste parti. In contrapposizione al motore immaginiamo il 'formicaio', un insieme di formiche che interagendo tra loro sono in grado di mantenere, per esempio, la temperatura all'interno del formicaio su valori

costanti e con minime variazioni tra estate e inverno. Il formicaio è un sistema complesso perché ogni singola formica non conosce il meccanismo di regolazione termica dell'ambiente circostante ma, semplicemente dall'interazione di molte formiche, si manifesta un fenomeno 'complesso'. In questo secondo caso non è possibile studiare il formicaio studiando il comportamento delle singole formiche.

Mentre le scienze naturali pure sono in grado di prevedere nel dettaglio gli eventi, le scienze dei fenomeni complessi possono solo azzardare un orientamento. Bisognerebbe comunque (Hayek, 1937) mantenere la consapevolezza dello scarso potere predittivo dell'econometria che, limitando i fattori esplicativi solo a quelli misurabili, ha portato agli insuccessi dell'interventismo macroeconomico keynesiano.

I sistemi sociali sono dunque sistemi complessi, cioè sistemi nei quali molte cause concorrono a creare un solo effetto e, nei quali, le relazioni fra le cause stesse sono non lineari, nel senso che l'effetto di ogni singola causa non è indipendente da quello delle altre, per cui non può essere isolato.

I sistemi complessi sono generalmente composti da agenti che interagiscono; ogni individuo interagisce solo con un ristretto numero di altri individui e, da queste interazioni locali, emergono fenomeni globali complessi, che non sono ipotizzabili a priori, pur conoscendo i singoli elementi ed i legami di interazione che intercorrono fra loro. La cosiddetta 'teoria del tutto' non è quindi sufficiente a spiegare i fenomeni complessi perché il 'tutto' non è la semplice somma delle sue parti.

Un sistema complesso viene definito da Arthur et al. (1997) come un sistema che possieda le seguenti caratteristiche:

1. La dinamica del sistema non deve avere un equilibrio globale.
2. Deve esistere interazione diffusa tra gli agenti del modello.
3. Non deve esistere un controllo centralizzato del modello.
4. Si deve creare una gerarchia a livelli con interazioni reciproche.

5. Gli agenti devono imparare ed evolversi in funzione dell'ambiente e delle loro interazioni.

Come risultato di tali proprietà si ottiene un ambiente caratterizzato da razionalità limitata e da aspettative non razionali.

Per affrontare i 'fenomeni complessi' può essere utile la teoria dell'individualismo metodologico' (cfr. cap. 3.3). Non potendo avere informazioni complete su stato e circostanze in cui si trovano i singoli elementi sociali, mentali o biologici, si ipotizza che essi si muovano a livello micro in base a certi principi: razionalità orientata allo scopo o al valore; significatività delle connessioni tra neuroni o ricerca della sopravvivenza. Ipotizzando alcuni meccanismi causali tra i singoli micro elementi, si immagina una metodologia di ricerca che conduca a predire l'emergere a livello macro di certi tipi di 'fenomeni complessi' nella loro forma astratta di 'modelli' (patterns). Questi modelli sono i modelli simulativi.

1.3 *La modellizzazione della realtà*

I fenomeni complessi sono difficilmente studiabili con gli stessi strumenti utilizzati per i sistemi semplici, come calcoli matematico-statistici e sperimentazioni in laboratorio. Il motivo risiede nel grandissimo numero di variabili che si dovrebbero manipolare, che risulterebbe non gestibile con equazioni matematiche. I sistemi complessi hanno anche delle componenti dinamiche di imprevedibilità e caotiche, che gli strumenti tradizionali hanno difficoltà ad affrontare.

Le simulazioni, e in particolare le simulazioni condotte mediante modelli ad agenti adattivi, rappresentano il principale strumento per lo studio di fenomeni complessi tipici delle scienze sociali.

Con il termine simulazione sociale si definisce dunque l'utilizzo di modelli di teorie esplicite creati ed eseguiti con l'ausilio del computer. Lo scopo è ricreare e

studiare aspetti essenziali della socialità e delle società naturali, siano esse umane, animali o artificiali.

Per agente si intende l'unità di popolazione osservata: una popolazione può essere una società umana, una specie animale, un'azienda o, per esempio, una catena di fornitura.

La simulazione può essere divisa in due fasi distinte:

1. La '*modellizzazione*', ossia l'ideazione e la costruzione concettuale del modello da simulare.
2. La '*creazione*' vera e propria del programma di simulazione con il supporto di strumenti informatici come quelli descritti nel capitolo [2].

Le scienze tradizionali utilizzano un processo di 'analisi' dei fenomeni, mentre le simulazioni sono una tecnica di 'sintesi' dei fenomeni.

Analizzare i fenomeni significa partire dalla realtà, dividerla nei suoi componenti e poi ricostruire i fenomeni reali mettendo insieme queste componenti mediante la teoria e il ragionamento.

Le simulazioni seguono invece la via della 'sintesi' della realtà, dove sintesi vuol dire partire dalle componenti per studiare cosa emerge quando queste componenti vengono messe insieme e fatte interagire. I modelli così creati partono dalla descrizione dei singoli componenti semplici, che vengono fatti interagire nel computer, producendo il sistema complesso oggetto della teoria.

Le simulazioni non servono soltanto a scoprire quali predizioni empiriche si possono derivare da una teoria una volta che la teoria è stata formulata e realizzato il programma per eseguirla nel computer. Le simulazioni servono anche per elaborare le teorie, per esplorarne e valutarne le caratteristiche e le implicazioni mentre le si sta ancora costruendo. Più specificamente con le simulazioni diventa possibile sviluppare e valorizzare un metodo di ricerca che viene usato in modo implicito e

spesso inconsapevole in tutte le scienze: il metodo degli esperimenti mentali. La simulazione viene definita da Parisi (2001) un ‘esperimento mentale assistito dal computer’. Il computer viene utilizzato per assistere il ricercatore nella costruzione degli esperimenti mentali, ma questo non significa che lo scienziato non continui ad elaborare teorie con metodi tradizionali, per confrontarle con i risultati degli esperimenti al computer. Il ricercatore che utilizza le simulazioni finisce per elaborare la teoria insieme alla simulazione, con un continuo processo di prove ed errori, che si sviluppa in parte nella mente del ricercatore e in parte nella simulazione.

I principali vantaggi delle simulazioni al computer rispetto a tecniche sperimentali tradizionali sono:

- **La possibilità di inserire gradualmente le diverse parti della teoria nel programma.** Questo significa poter modificare i dati già presenti, verificarne la coerenza interna ed esterna in ogni momento e osservare i risultati generati dal computer. Questo vantaggio oggi può essere sfruttato appieno proprio grazie a strumenti informatici come la programmazione orientata agli oggetti e software per la simulazione agent-based come Swarm².
- **La grande flessibilità offerta dalle simulazioni allo sperimentatore.** Negli esperimenti di laboratorio tradizionali bisognava studiare con attenzione tutte le variabili ambientali prima di eseguire ogni esperimento, perché risultava lungo, difficile e costoso ricreare lo stesso esperimento in condizioni leggermente differenti. Oggi, grazie alle simulazioni di mondi artificiali, tempo e causalità sono determinati dallo sperimentatore, le caratteristiche ambientali possono essere modificate, la rilevazione dei dati è più precisa e l’esperimento può essere ripetuto a piacere. In sintesi possono essere modificati i parametri del modello che si studia in modo semplice e veloce.

² Per una panoramica sul progetto Swarm si veda 2.2

- **La possibilità di modificare comodamente i parametri di osservazione.** L'osservatore influenza sempre il fenomeno osservato. Questo principio è molto rilevante nelle scienze sociali (si pensi all'influenza dell'intervistatore sull'intervistato in un sondaggio). Rimane comunque importante nell'osservazione di qualsiasi fenomeno, anche solo perché la percezione della realtà è sempre filtrata attraverso la soggettività dell'osservatore. La possibilità di modificare tutti i parametri dell'osservatore consente di isolare e studiare in modo migliore il fenomeno. La velocità di scorrimento del tempo è particolarmente rilevante perché contribuisce a ridurre la durata della ricerca che, se svolta nella realtà, potrebbe richiedere la ripetizione periodica delle osservazioni, impegnando tempi considerevolmente lunghi. Casi particolari, ma spesso rilevanti, si manifestano difficilmente in ambiti reali, o sono difficilmente osservabili. La ripetizione della rilevazione e le variazioni delle condizioni possono facilitarne l'osservazione nel tentativo di individuare fenomeni emergenti.
- **La possibilità di tenere sotto osservazione il modello.** Le sonde³ consentono di tenere sotto controllo tutte le variabili di ogni agente della simulazione, anche senza dover preventivamente decidere quali saranno rilevanti. L'utilizzo degli agenti virtuali consente di rilevare i dati facilmente, evitando errori e senza determinare a priori quali grandezze osservare.

L'indagine su ambiti virtuali risulta, quindi, molto vantaggiosa rispetto allo studio tradizionale in laboratorio, per l'analisi dei fenomeni definiti 'complessi'.

1.4 Agenti semplici e razionalità limitata

Con modelli basati su agenti si possono ottenere risultati interessanti, grazie al loro elevato grado di indipendenza e alla capacità di adattare il loro comportamento alle

³ Le sonde sono strumenti informatici utilizzati nel progetto Swarm, cfr. 2.2

caratteristiche dell'ambiente in cui operano.

La simulazione agent-based viene utilizzata con due finalità:

1. Simulazione basata su agenti a reti neurali artificiali, o algoritmi evolutivi con regole in grado di riprodurre, anche se in maniera molto semplificata, i comportamenti umani. In questi modelli gli agenti sono dotati di capacità di apprendimento e sono capaci di modificarsi e adattarsi in base all'interazione con l'ambiente fisico o sociale.
2. Studio di fenomeni emergenti. I fenomeni emergenti vengono osservati mediante la ricostruzione di sistemi capaci di auto-organizzazione, i quali si modificano per rispondere a cambiamenti sopravvenuti nell'ambiente, contribuendo così, a modificare l'ambiente stesso.

Fenomeni complessi, di una certa importanza, risultano emergere già con l'impegno di agenti capaci di reagire, in modo prestabilito, a stimoli semplici; il comportamento di simili agenti risulta, però, ancora influenzato dalle scelte di chi redige il programma. Quando si ricercano fenomeni emergenti bisogna porre particolare attenzione alle cosiddette 'false emergenze', cioè fenomeni che sembrano mostrare una qualche forma di coerenza, che in realtà è solo apparente, e non implica una relazione o regola. Come esempio si pensi alle costellazioni: le stelle, come osservate dalla Terra, appaiono disposte in costellazioni, quindi sarebbe ipotizzabile una regola che accomuni gli astri appartenenti alla stessa costellazione. In realtà i corpi celesti sono disposti nello spazio in modo molto diverso da come appaiono dalla Terra, stelle vicine sono molto lontane tra loro e vice versa. La regolarità delle costellazioni non ha alcun fondamento nello spazio. Questo esempio è anche interessante per mostrare come il punto di vista, o più in generale l'osservatore, sia importante nell'analisi di un fenomeno, sia esso naturale o ricreato nel computer.

Quando la determinazione delle regole di comportamento avviene in base a tecniche di intelligenza artificiale, invece, l'azione di ogni agente risulta essere conseguenza del grado di apprendimento del medesimo, permettendo di garantire l'autonomia sufficiente a promuovere lo sviluppo di fenomeni innovativi. Le tecniche di intelligenza artificiale utilizzate nella generazione degli agenti artificiali fanno capo a due paradigmi: le reti neurali artificiali e i metodi evolutivi (algoritmi genetici, classifier system e genetic programming).

Il meccanismo delle reti neurali per la costruzione di agenti adattivi si basa sull'apprendimento con il quale gli agenti imparano ad essere coerenti con i target sui quali viene esercitata la rete neurale, correggendo contemporaneamente le congetture effettuate sulle azioni da compiere e sugli effetti di tali azioni. In questo modo gli agenti sviluppano una sorta di 'capacità cognitiva', anche se limitatissima, che consente loro di reagire agli stimoli dell'ambiente.

I metodi evolutivi sono una tecnica che si basa sull'imitazione del processo di evoluzione neurale: le regole sono assoggettate a procedimenti di riproduzione ed estinzione, atti a garantire la sopravvivenza di quelle che permettono di ottenere i migliori risultati. Si procede valutando ciascuna regola per stabilire quali fra esse dovranno contribuire alla formazione di nuove istanze, cioè riprodursi, e quali saranno sostituite dalle nuove regole. La ripetizione ciclica del procedimento consente un graduale perfezionamento del patrimonio normativo e le mutazioni delle condizioni ambientali.

In base a questi meccanismi ogni agente sarà in grado di interagire con l'ambiente. I processi di interazione si sviluppano attraverso effetti di variabili ambientali sull'agente, in modo da costruire una 'coerenza interna'. La capacità di agire sull'ambiente indica banalmente che l'agente deve essere integrato nel sistema, non è quindi un'entità astratta e indipendente. L'influenza deve avvenire in entrambe le

direzioni; l'agente deve essere influenzato dall'ambiente, ma anche l'ambiente deve evolversi in funzione degli agenti che lo popolano. Questo meccanismo appare chiaramente nel modello di borsa SUM. Questo modello, descritto in Terna (2000a), studia il comportamento di diverse popolazioni di agenti borsisti che operando in un mercato virtuale contribuendo, con le loro azioni, a formare il prezzo di mercato. Si è potuto osservare un interessante fenomeno emergente, semplicemente come conseguenza delle interazioni degli agenti si formano bolle inflazionistiche.

Gli agenti comunicano tra loro ad un livello locale, non è necessario il controllo da un livello superiore. Per poter costruire questo meccanismo è necessario dotare gli agenti di intelligenza. Questo problema è stato affrontato in Minsky (1987), che sostiene si possa rappresentare la mente come una società di agenti che vengono paragonati a neuroni, ognuno di questi agenti ha funzioni distinte. La mente emerge dall'interazione tra questi agenti, proprio come l'intelligenza umana è frutto dell'interazione tra i neuroni, i quali sono entità autonome ed hanno la proprietà di poter essere utilizzati in molte differenti sequenze di interazione per realizzare attività diverse. Secondo questo principio Minsky (1987) arriva a sostenere che 'l'intelligenza può emergere dalla non-intelligenza'.

	Ambiente non strutturato	Ambiente strutturato
Agenti senza mente	Possono determinare risultati aggregati complessi, con il rischio della poca plausibilità dei risultati	Possono determinare risultati aggregati complessi e plausibili
Agenti con mente	Possono determinare risultati aggregati complessi e plausibili	Possono determinare risultati aggregati complessi e plausibili, con la eventualità di risultati di rilievo a livello micro-individuale

Tab. 1.2. Risultati delle simulazioni del modello SUM con agenti dotati di mente e privi di mente in diversi contesti

Secondo Terna (2002) nei modelli ad agenti con interazione non sono necessari agenti particolarmente sofisticati per creare situazioni complesse; la capacità di learning e la struttura esterna agli agenti sono però determinanti per la qualità dei risultati. Esistono quindi, in accordo con la classificazione di Terna (2002b), diversi tipi di simulazioni, a seconda che si utilizzino agenti ‘con mente’ o agenti ‘senza mente’⁴. I risultati che si ottengono da simulazioni in ambienti strutturati e non strutturati utilizzando i due tipi di agenti sono descritti in tab. 1.2

⁴ Il termine ‘mente’ non deve essere inteso in senso cognitivo o psicologico. In questo caso per mente si intende solo la capacità di adattamento degli agenti all’ambiente prodotta dall’utilizzo di reti neurali artificiali.

BIBLIOGRAFIA

- Arthur, W., Durlauf, S., & Lane, D. (1997). *The economy as an evolving complex system*. Addison-Wesley.
- Feynman, R. (2000). *Sei pezzi facili*. Adelphi.
- Holland, J. (1975). *Adaptation in natural artificial systems*. Ann Arbor.
- Legrenzi, P. (2002). *Prima lezione di scienze cognitive*. Laterza.
- Marraffa, M. (2002). *Scienza cognitiva, un'introduzione filosofica*. Cleup.
- Minsky, M. (1987). *The society of mind*. Picador.
- Parisi, D. (2001). *Simulazioni, la realtà rifatta nel computer*. Il mulino.
- Terna, P. (2000). *Sum: a surprising (un)realistic market: Building a simple stock market structure with swarm*, presentato a cef 2000, barcelona, 5-8 giugno.
- Terna, P. (2002). *La simulazione come strumento di indagine per l'economia; paper per il workshop su scienze cognitive ed economia organizzato dalla associazione italiana di scienze cognitive, rovereto, 21 settembre*.

BIBLIOGRAFIA

Capitolo 2

GLI STRUMENTI PER LA SIMULAZIONE

La rivoluzione che è iniziata dagli anni ottanta in economia e in tutte le scienze sociali, stravolgendo i modelli classici e neoclassici, a favore dello studio di sistemi complessi, non sarebbe stata possibile senza la nascita e il veloce sviluppo degli strumenti informatici. La simulazione, come strumento dello studio dell'economia, è praticabile oggi grazie all'aumento esponenziale della velocità di calcolo degli elaboratori. In parallelo all'aumento della potenza di calcolo, sono sorti paradigmi di programmazione innovativi. Ne è un esempio il linguaggio Java, che consente di formalizzare, in modo semplice, gli agenti economici all'interno di un programma per computer. Dall'interazione degli agenti nella simulazione è possibile osservare i fenomeni emergenti. Programmare rimane comunque un lavoro lungo e difficile, con molte possibilità di errori. Per non dover trasformare gli scienziati sociali in programmatori sono nati, negli ultimi anni, una moltitudine di strumenti informatici, come Swarm, che semplificano di molto l'interazione tra il ricercatore e il computer. Obiettivo di questo capitolo è descrivere alcuni dei principali formalismi utilizzati nelle simulazioni Agent Based, come Java e UML, e gli strumenti con i quali è stata sviluppata la Virtual Enterprise.

2.1 La programmazione orientata agli oggetti e il linguaggio Java

Lo scopo di questo capitolo è descrivere gli strumenti che si possono adoperare per gestire le simulazioni in generale, e che sono stati utilizzati per sviluppare *JVE* in particolare. Le simulazioni sono inevitabilmente complesse, questa complessità viene gestita dai programmi per computer, che possono essere paragonati a simulazioni digitali di modelli concettuali e fisici.

I linguaggi con i quali si creano i programmi non sono altro che un'astrazione e formalizzazione di processi mentali. Alcuni linguaggi (come il FORTRAN, il BASIC o il C) sono definiti *imperativi* perché l'astrazione utilizzata consiste nel pensare il problema in termini del funzionamento interno del computer, piuttosto che porre attenzione al problema sottostante. In alternativa esistono altri linguaggi (come il LISP, APL o PROLOG) che si concentrano sul problema dal punto di vista logico, scomponendolo in una catena di decisioni. Ognuno di questi linguaggi si adatta bene a specifiche tipologie di problemi da risolvere.

La storia della programmazione¹ inizia negli anni '40. L'unico metodo per programmare era il linguaggio macchina, in altre parole il lavoro del programmatore era quello di settare ogni singolo bit a 1 o 0 su enormi computer che occupavano stanze intere e pesavano decine di tonnellate; si capisce che questo tipo di procedimento, non solo era faticoso, ma era riservato ad una cerchia ristretta di persone. Pochi anni dopo il 1950, con il progresso tecnologico e la riduzione delle dimensioni e dei costi dei calcolatori, nacquero due importanti linguaggi di programmazione, il Fortran (FORmula TRANslator), il cui utilizzo era ed è prettamente quello di svolgere in maniera automatica calcoli matematici e scientifici, e l'ALGOL (ALGOrithmic Language), altro linguaggio per applicazioni scientifiche sviluppato da Backus (l'inventore del FORTRAN) e da Naur, i quali oltretutto misero a punto un metodo per

¹ cfr. <http://www.html.it> (2002)

rappresentare le regole dei vari linguaggi di programmazione che stavano nascendo. Nel 1960 venne presentato il Cobol (COmmon Business Oriented Language), ideato per applicazioni nei campi dell'amministrazione e del commercio, per l'organizzazione dei dati e la manipolazione dei file. Nel 1964 fa la sua comparsa il Basic, il linguaggio di programmazione per i principianti, che ha come caratteristica fondamentale quella di essere molto semplice e, infatti, diventa in pochi anni uno dei linguaggi più utilizzati al mondo. Intorno al 1970, però, Niklus Wirth creò il Pascal per andare incontro alle esigenze di apprendimento dei neo-programmatori, introducendo però la possibilità di creare programmi più leggeri e comprensibili di quelli sviluppati in Basic. Si può affermare con certezza che Wirth ha centrato il suo obiettivo, considerando che ancora oggi il Pascal viene usato come linguaggio di apprendimento nelle scuole. Pochi anni più tardi fa la sua comparsa il C (chiamato così perché il suo predecessore si chiamava B), che si distingueva dai linguaggi precedenti per il fatto di essere molto versatile nella rappresentazione dei dati. Il C, infatti, ha delle solide basi per quanto riguarda la strutturazione dei dati, però può apparire come un linguaggio assai povero vista la limitatezza degli strumenti a disposizione. Invece la sua forza risiede proprio in questi pochi strumenti che permettono di fare qualsiasi cosa, non a caso viene considerato 'il linguaggio di più basso livello tra i linguaggi ad alto livello', per la sua potenza del tutto paragonabile al linguaggio macchina, mantenendo però sempre una buona facilità d'uso.

Quando un programmatore scrive un programma il numero delle variabili e delle interazioni che devono essere coordinate simultaneamente diventa sempre più gravoso con l'aumentare delle dimensioni del programma. Questo è uno dei principali problemi quando lo sviluppo avviene come una serie di passi lineari, senza crearne il modello astratto. Pur essendo il modo di procedere più intuitivo ben si adatta a problemi semplici, la cui struttura può essere facilmente immaginata dalla mente del programmatore, ma causa errori e situazioni ingestibili in programmi di maggiori

dimensioni, nei quali le variabili non possono essere tenute sotto controllo contemporaneamente dallo sviluppatore, e quando è difficile crearsi un'immagine mentale unitaria del programma stesso. I problemi derivanti dalla mancanza di interfacce definite e interazioni tra le funzioni, e la necessità continua di modificare o implementare il codice senza dover necessariamente rivedere il funzionamento di tutto il programma, rendono praticamente irrealizzabile una simulazione economica con linguaggi *procedurali*.

Nel 1983 Bjarne Stroustrup introdusse una vera rivoluzione, nacque il C++ (o come era stato chiamato inizialmente C con classi) che introduceva, sfruttando come base il C, una nuova struttura, la classe.

Il concetto di classe è semplice: consiste nel dividere l'interfaccia dal contenuto. Si ottengono in questo modo tanti moduli che comunicano tra loro attraverso le interfacce, che permettono al programmatore di cambiare il contenuto di una classe, (se sono stati trovati errori o solo per introdurre delle ottimizzazioni) senza, per questo, doversi preoccupare di controllare eventuale altro codice che richiami la classe. Come si può intuire tale linguaggio ha completamente stravolto il modo di programmare, che prima si riduceva ad una programmazione procedurale, e lasciava poco spazio al riutilizzo del codice². Il concetto di classe e la filosofia OOP viene ripresa e sviluppata da uno dei linguaggi attualmente più evoluti, il *Java*.

La più idonea alternativa per costruire una simulazione economica, tenendo conto della sua natura complessa, è la programmazione *object-oriented* e in particolare il linguaggio *Java*, che è stato utilizzato per creare JVE. La programmazione orientata agli oggetti, oltre a fornire una procedura flessibile che considera gli elementi del problema come oggetti astratti, consente al programma di adattarsi al problema specifico, aggiungendo nuove classi di oggetti, in modo che il codice descriva la

² Basti pensare all'utilizzo del GOSUB nel Basic che su migliaia di righe di codice creava naturalmente confusione, o al Pascal nel quale una variabile (e il suo tipo) deve essere dichiarata prima di essere usata.

soluzione e il problema nello stesso momento (nel nostro caso la simulazione e il modello sottostante).

L' OOP (*Object-Oriented Programming*) permette di descrivere i problemi sia logicamente, essendo gli oggetti metafore del mondo reale che posseggono delle caratteristiche e dei comportamenti, sia tenendo conto della struttura del computer, poiché gli oggetti possono essere visti come piccoli computer con un loro *stato* e delle operazioni che sanno eseguire.

Aristotele fu probabilmente il primo a studiare il concetto di *tipo*. Il concetto fondamentale della OOP è la creazione di tipi astratti di dati, dette *classi*, dalle quali generare attraverso *instance* gli oggetti e manipolarli inviando loro dei messaggi.

L'idea che tutti gli oggetti, rimanendo unici, abbiano caratteristiche e comportamenti comuni è stata usata per la prima volta nel linguaggio *Simula-67* che era stato creato per sviluppare simulazioni bancarie, e ha introdotto il concetto di *classe*.

Le caratteristiche che definiscono un linguaggio puro *object-oriented* sono descritte nei paragrafi successivi.

2.1.1 Tutto è un oggetto

Un oggetto si può immaginare come una scatola nera contenente dati, alla quale si possono richiedere informazioni, e si può ordinare di eseguire operazioni su se stessa. In teoria è possibile astrarre e concettualizzare qualsiasi processo logico all'interno di un oggetto.

Un oggetto consiste in due tipi di informazioni:

Variabili: La lista delle variabili sintetizza lo stato dell'oggetto, e nel caso delle simulazioni lo stato dell'agente. Le variabili possono essere di qualsiasi tipo consentito dal C, come *integer*, *float*, *array*, *puntatori* o *istanze di classi*.

Metodi: I metodi determinano cosa un oggetto sa fare. Tipicamente esistono metodi

che raccolgono informazioni dal mondo esterno, metodi che inviano informazioni all'esterno e metodi che le processano.

Gli oggetti sono creati attraverso in processo che si definisce *instance*, tradotto in italiano in modo impreciso con *istanza*. Il codice è scritto in classi e gli oggetti sono *instance* delle classi. Le variabili che sono contenute nell'oggetto si definiscono *instance variables*, e le informazioni in esse contenute sono disponibili per tutti i metodi dell'oggetto. Se uno dei metodi dell'oggetto necessitasse di informazioni *private* che non vengano viste dagli altri è possibile creare delle *method variables*.

Gestione della memoria

Gli oggetti gestiscono una propria allocazione di memoria indipendente. In questo modo è possibile creare un oggetto da un gruppo di altri oggetti.

Tipizzazione

Gli oggetti sono tipizzati. Ogni oggetto è una *instance* di una *classe*, che rappresenta il *tipo* al quale appartiene. In questo modo è possibile costruire la complessità di un modello di simulazione nascondendola all'interno della semplicità dei singoli oggetti.

I messaggi

Un programma è un insieme di oggetti che comunicano tra loro attraverso messaggi. Per fare delle richieste agli oggetti si utilizzano i messaggi che possono essere immaginati come chiamate a funzioni appartenenti all'oggetto.

Tutti gli oggetti dello stesso tipo possono ricevere gli stessi messaggi. La tipologia dell'oggetto ne definisce la *forma* e garantisce la capacità di gestire i messaggi. Questo significa che è possibile scrivere codice che si interfacci con gli oggetti senza sapere esattamente come sono costruiti al loro interno. Questa caratteristica è uno dei concetti più importanti della OOP ed è quella che ci permette, nelle simulazioni,

di descrivere i modelli e i mondi virtuali utilizzando gli agenti senza doverli descrivere singolarmente.

Gli oggetti dialogano attraverso le interfacce

Per poter far eseguire delle operazioni agli oggetti, come disegnare qualcosa sullo schermo o svolgere delle operazioni matematiche, è sufficiente inviare dei messaggi ad un *interfaccia*, che stabilisce quali richieste si possono fare ad uno specifico oggetto.

Incapsulamento

Si pensi all'incapsulamento come ad un involucro protettivo che avvolge sia le istruzioni, sia i dati che si stanno manipolando. Questo involucro protegge i comportamenti (metodi) e i dati (variabili) da accessi arbitrari da parte di un altro programma.

In *Java* la base dell'incapsulamento è la classe. Si crea una classe che rappresenta l'astrazione di un gruppo di oggetti che hanno in comune la stessa struttura e lo stesso comportamento. L'oggetto è una singola istanza di una classe che mantiene la struttura e i comportamenti definiti dalla classe.

La rappresentazione individuale, o la struttura dei dati di una classe vengono definite da una serie di variabili. Il valore delle variabili all'interno di un oggetto rimangono private, a meno che i metodi al suo interno siano scritti per passare informazioni al di fuori dell'oggetto. Nelle metafore delle simulazioni significa che gli agenti, proprio come gli esseri umani, hanno accesso alle informazioni al loro interno e dialogano con l'esterno solo tenendo conto del loro stato.

La struttura peculiare degli oggetti *Java* divide gli sviluppatori in due categorie: i *class creator* e i *client programmer*. I primi sono coloro che costruiscono le classi, nascondono il loro funzionamento interno e rendono accessibile agli altri programmatori solo il necessario per utilizzare la classe, i *client programmer* sono invece gli

utilizzatori delle classi create da altri per creare delle loro applicazioni nel modo più rapido e semplice possibile.

Nel caso dello sviluppo di simulazioni con *Swarm*, ci si pone un po' in entrambe le vesti. Da un lato si utilizzano le classi di *Swarm* create da altri, che ci aiutano a gestire gli aspetti più tecnici della simulazione, senza dover preoccuparci troppo per esempio del funzionamento interno della memoria, o della gestione del tempo nella simulazione. D'altro canto, creando un modello della realtà completamente nuovo, che incorpora al suo interno una teoria, dobbiamo obbligatoriamente seguire un processo di astrazione e formalizzazione, e quindi scrivere una nuova classe.

Riutilizzabilità del codice

Una volta che una classe è stata creata dovrebbe, idealmente, rappresentare una unità di codice riutilizzabile. L'utilizzo di classi create da altri non è così semplice come si potrebbe sperare, il modo più semplice per farlo è utilizzare direttamente l'oggetto con una *instance* della classe, ma è anche possibile inserire uno o più oggetti in una nuova classe, creando una struttura a “scatole cinesi”. Questa procedura si definisce *composition* o *aggregation*.

Ereditarietà

Dopo aver creato una classe sarebbe un peccato dover crearne un'altra completamente nuova, per apportare delle piccole modifiche. Per ovviare a questa difficoltà è possibile clonare una classe esistente, e fare aggiunte o modifiche al clone. Se la classe originale (chiamata anche *base*, *super* o *parent class*) cambia, le modifiche si riflettono sul clone (chiamata *derivata*, *ereditata*, *sub* o *child class*). Questo principio è detto dell'ereditarietà.

Sviluppando in modo ereditario è possibile ragionare nei termini del problema,

non essendo necessari molti modelli intermedi per passare dalla descrizione del problema alla soluzione. Con questa struttura gerarchica, la *super class* è il modello astratto, dal quale si passa direttamente alla descrizione del mondo reale. Un semplice esempio per comprendere questo concetto basilare può essere la classe *Shape* (forma) utilizzata, ad esempio, in programmi grafici. Ogni forma può avere diverse caratteristiche, come la dimensione, il colore, la posizione e così via. Sulla forma si possono compiere molte operazioni: disegnarla, cancellarla, muoverla, colorarla. Dalla classe generale *Shape* è quindi necessario derivare forme specifiche con diverse caratteristiche e comportamenti. La Fig. 2.1³ rappresenta l'esempio descritto utilizzando la notazione UML

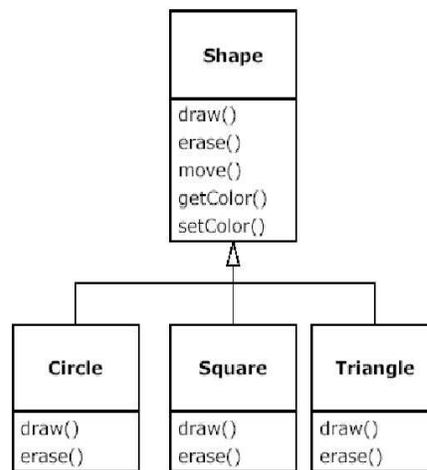


Fig. 2.1. Esempio di ereditarietà della classe *Shape* in notazione UML

Polimorfismo

Il polimorfismo permette di avere metodi con lo stesso nome, ma che hanno parametri diversi. Viene detto anche *function overloading*.

³ Tratta da (Eckel, 2000, p. 42)

Quando si lavora con classi gerarchizzate può essere necessario trattare gli oggetti, non come una tipologia specifica, ma come un modello generale. Nell'esempio di *Shape* le funzioni manipolano forme generiche, senza preoccuparsi che siano cerchi o triangoli o una qualche figura che non è ancora stata definita. È possibile disegnare qualsiasi figura semplicemente mandando un messaggio all'oggetto *Shape*. Questo è uno dei modi per estendere un programma e per permettergli di manipolare situazioni nuove.

Si deve notare che Swarm non permette il polimorfismo. I metodi creati in Swarm devono dunque avere nomi unici.

Gestione degli errori

La gestione degli errori è sempre stata uno dei problemi più difficili da risolvere, molti linguaggi di programmazione semplicemente hanno ignorato il problema. Con Java gli errori, o come si definiscono le *exceptions*, sono visti come oggetti che "scaturiscono" dall'errore e possono essere 'catturati' e gestiti adeguatamente senza poterli ignorare. In questo modo i programmi risultano molto più robusti e affidabili.

Multi Thread

Un obiettivo primario della programmazione è gestire più processi contemporaneamente. In passato si scrivevano delle *routines* di interruzione e sospensione dei processi hardware. In alcune situazioni particolari questo genere di gestione è necessaria, ma, nella maggior parte dei casi, è sufficiente dividere il programma in parti distinte, che vengono eseguite contemporaneamente. La gestione dei processi in *Java* è incorporata nel linguaggio, ed è gestita a livello degli oggetti, in modo che, ad ogni oggetto, possa corrispondere un processo distinto, e apparentemente parallelo. Si noti che il parallelismo dei processi, non solo in Java, è solo apparente perché

i microprocessori attuali lavorano comunque intrinsecamente in modo sequenziale, ma sincronizzano le operazioni in sequenza a livello della CPU.

2.1.2 Java e l'indipendenza dalla piattaforma

Il linguaggio *Java*⁴ ha un grande pregio, che lo ha reso così popolare e diffuso, in particolare per le applicazioni Internet client-server: l'indipendenza dalla piattaforma. Un programma *Java* può essere eseguito allo stesso modo su sistemi operativi diversi (Windows, Linux, Solaris e altri), e addirittura su macchine con architetture hardware diverse (Intel, Alpha, Worksation Sun e altri).

Per poter eseguire un programma che si è scritto ad esempio in C, ma lo stesso accade per la maggior parte degli altri linguaggi, è necessario compilarlo. Compilare un programma significa tradurre il file sorgente in codice macchina. Il file sorgente è il documento che contiene il programma espresso nel linguaggio di programmazione scelto; è stato scritto dal programmatore e viene compreso dal compilatore ma non dal computer. Il codice macchina viceversa sono le istruzioni comprensibili dal microprocessore che permettono di eseguire il programma. Il file in codice macchina è generato dal compilatore. E' facile intuire che se si compila il codice per un determinato microprocessore il programma potrà funzionare solo su computer con lo stesso tipo di processore.

Java utilizza un procedimento diverso. Il file sorgente Java non viene compilato dal compilatore direttamente in codice macchina, ma in un codice, che si definisce bytecode, che deve essere ulteriormente interpretato per poter divenire comprensibile dal microprocessore. Lo strumento che si occupa di interpretare il file semi compilato in bytecode si chiama Java Virtual Machine (J.V.M.), anche detta Interprete Java o Java Runtime Environment. La Virtual Machine può essere immaginata come un computer all'interno del computer stesso. In questo modo un programma compilato

⁴ Per maggiori informazioni sul linguaggio *Java* un ottimo testo che, oltre affrontare gli aspetti tecnici, avvicina alla filosofia di programmazione OOP è (Eckel, 2000)

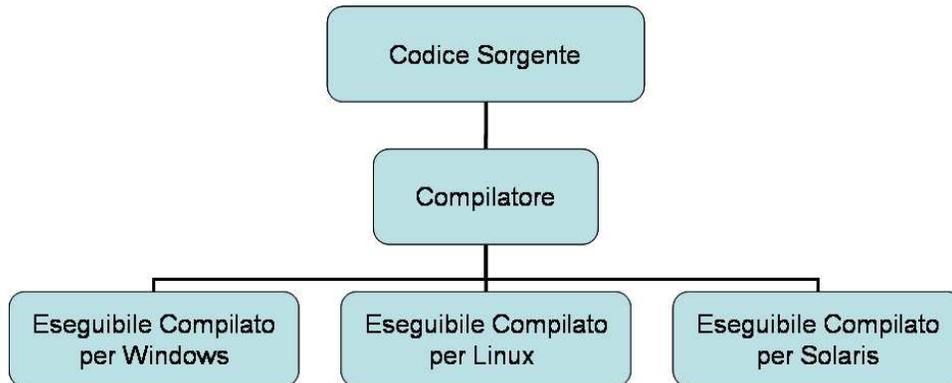


Fig. 2.2. Compilare un programma tradizionale

in Java potrà essere eseguito da qualunque piattaforma e sistema operativo che possieda una Java Virtual Machine installata.

Una Java Virtual Machine è implementata anche nei vari Browser (Come Netscape e Explorer) per poter eseguire i programmi Java incontrati nella rete, i cosiddetti Applet. Questo però, unito al fatto che Java ancora si evolve, causa degli ovvi problemi di incompatibilità: capita spesso che il più moderno Browser supporti una versione di Java precedente all'ultima rilasciata dalla Sun Microsystem. Inoltre bisogna tener presente che non tutti gli utenti di Internet navigano usando l'ultima versione di Netscape o di Explorer. Quindi, volendo creare un applet, ed inserirlo in un nostro documento HTML, dobbiamo tenere presente questi problemi, e cercare di scrivere un programma che sia compatibile con la maggior parte delle JVM implementate nei vari browser⁵.

Java esegue, quindi, due funzioni: la prima di codifica, trasformando il programma compilato in un formato Bytecode indipendente da piattaforma e processore con il comando *javac*, la seconda di interprete di programmi già compilati con la *JavaVM*.

⁵ Come esempio si pensi al progetto SUMWeb del prof. Pietro Terna <http://web.econ.unito.it/terna/sum>

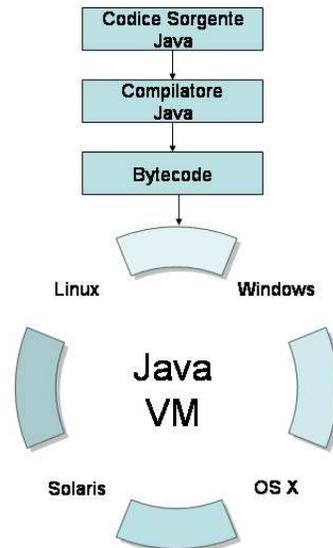


Fig. 2.3. Compilare un programma Java

2.1.3 Java in pratica

Per poter eseguire un programma Java è necessario installare sul computer JDK (Java Development Kit) che si può ottenere liberamente dal sito della Sun Microsystems <http://www.sun.com>. A questo punto dobbiamo preparare un file sorgente⁶, con un qualsiasi editor di testo (come ‘notepad’ per Windows) e salvarlo con estensione *.java*. L’esempio più semplice possibile è un comando che faccia apparire a schermo la scritta ‘Hello World’:

```
public class HelloWorldApp {
    public static void main(String [] args) {
        // Display "Hello World!"
        System.out.println("Hello World!");
    }
}
```

⁶ Esempio Hello World tratto da <http://java.sun.com/docs/books/tutorial/getStarted/cupojava/win32.html>

Salviamo il file con il nome ‘helloworld.java’.

Per compilare il file in formato bytecode è sufficiente scrivere il comando:

```
javac helloworld.java
```

Per eseguirlo all’interno della JVM è sufficiente scrivere:

```
java helloworld
```

ed ecco fatto. Questo è un semplice esempio, ovviamente una simulazione economica è infinitamente più complessa, ma il meccanismo della compilazione rimane il medesimo, aggiungendo solo al compilatore le informazioni su dove trovare le biblioteche di funzioni *Swarm*⁷.

Documentare i programmi con Javadoc

Una delle necessità del programmatore è quella di illustrare nel dettaglio il funzionamento della sua applicazione e comunicare ad altri sviluppatori il modo in cui potersi interfacciare con le classi e i metodi da lui creati. Ogni classe, metodo o campo ha proprie caratteristiche e modalità di utilizzo, la conoscenza precisa di queste è indispensabile per poter modificare o implementare nuove funzionalità nel lavoro realizzato da altri. La stesura di una precisa ed esauriente documentazione del codice di una applicazione, o di un insieme di classi, è uno dei requisiti fondamentali che ogni programmatore deve soddisfare. Serve per migliorare il rapporto tra il prodotto da lui scritto, e gli utilizzatori che intendono utilizzarlo, per far sì che l’utilizzo avvenga nel modo più aderente possibile alle intenzioni del programmatore stesso.

⁷ Le biblioteche di funzioni utilizzate da Swarm sono contenute in archivi Jar, e sono: share/swarm/swarm.jar;lib/;lib/plot.jar;lib/gui.jar;lib/xlrd.jar;lib/jveframe.jar

La documentazione è un requisito dal quale non si può prescindere nemmeno per applicazione piccole, ma cresce notevolmente al crescere del numero di classi implementate, e diventa essenziale nel caso di una simulazione come JVE. La documentazione svolge una funzione primaria in quei progetti che vengono sviluppati da un gruppo di ricercatori, e non da un singolo, ed è altresì importante per il singolo che sviluppa un grande progetto in momenti diversi. Una corretta documentazione consente di riprendere lo sviluppo del codice da dove lo si era lasciato senza dover, per questo, ristudiarlo integralmente.

La Sun stessa si manifesta pienamente convinta del ruolo fondamentale svolto, all'interno del contesto generale di produzione di un'applicazione, dalla documentazione dell'API, e lo dimostra attraverso due segnali forti e precisi:

- l'insieme di pagine HTML che accompagna il Java Development Kit sin dalla sua prima versione è voluminoso quanto dettagliato, indispensabile per orientarsi in una collezione di package sempre più numerosa ed articolata;
- javadoc, lo strumento utilizzato da Sun per produrre la documentazione ufficiale di cui sopra, viene incluso in JDK, e di conseguenza distribuito gratuitamente. In particolare, javadoc è presente a partire da JDK 1.1, ma non riporta una propria versione riconoscibile in modo indipendente, perché specificata nell'eseguibile o nel package di riferimento: la consuetudine è quella di indicare, a questo scopo, la stessa versione del JDK nel quale è contenuto (quindi, ad esempio, Java 2 SDK versione 1.2.2 contiene javadoc versione 1.2.2).

La comunità degli sviluppatori ha quindi a disposizione, da questo punto di vista, sia il migliore esempio del 'prodotto finito' ottenibile, sia lo strumento (e le indicazioni) attraverso il quale tale risultato viene raggiunto, javadoc.

In pratica, javadoc è in grado di riproporre, in un formato facilmente consultabile, lo ‘scheletro’ delle dichiarazioni interne al codice sorgente, completamente spogliato delle complicazioni superflue dello sviluppo. In termini generali, javadoc elabora le dichiarazioni e i commenti doc presenti in uno o più file sorgenti Java, producendo un set corrispondente di pagine HTML che descrivono classi, interfacce, costruttori, metodi e campi con indicatori di visibilità `public` o `protected`. Volendo, il programmatore può inserire ulteriori commenti come chiarimenti, precisazioni ed esempi d’uso, al fine di rendere la consultazione del codice davvero inutile per l’utente.

Uno dei punti di forza del meccanismo ideato da Sun è probabilmente costituito dal fatto che, al fine di realizzare le proprie elaborazioni, javadoc si basa sulla presenza del compilatore `javac`. In particolare, javadoc utilizza `javac` per compilare il codice sorgente, con lo scopo di mantenere le dichiarazioni ed i commenti doc, scartando tutte le parti di implementazione.

Ciò significa che, in teoria, sarebbe possibile far lavorare javadoc già dalle fasi di design, nel momento in cui l’interfaccia inizia a prendere corpo ma è ancora assente l’implementazione nella sua totalità, creando i vantaggi precedentemente descritti per lo sviluppo in team del progetto. L’utilizzo di `javac` da parte di javadoc è anche garanzia di produzione di documentazione esatta ed affidabile.

Tecnicamente Java gestisce sia i commenti multi linea (caratteristici di C), delimitati da `/*` e `*/`, che i commenti mono linea (caratteristici di C++), indicati da `//`. Entrambi i commenti appartenenti a queste due categorie, definibili come commenti ordinari, vengono inseriti all’interno del codice sorgente Java con lo scopo di rendere più comprensibili i dettagli implementativi del codice stesso. Queste due tipologie di commento sono ignorate da javadoc. I commenti gestiti da javadoc iniziano con `/**` (invece del normale `/*`), terminano con `*/` e sono solitamente multi linea. Vengono

inseriti nel codice sorgente, e sono riconosciuti e processati da javadoc solo se immediatamente precedenti la dichiarazione di una qualunque entità (classe, interfaccia, metodo, costruttore, campo).

Ogni commento doc è formato da due sezioni:

1. la descrizione, che inizia dopo il delimitatore iniziale `/**` e prosegue fino alla sezione dei tag. La prima frase di ogni descrizione deve essere una rappresentazione concisa ma completa dell'entità dichiarata. Essa viene infatti pensata e scritta per figurare anche da sola, in quanto usata come definizione generale ed inserita da javadoc nella sezione dal titolo 'Summary'.
2. la sezione dei tag, la quale inizia con il primo carattere `@` che si trova in testa ad una riga (escludendo asterischi e spazi).

I commenti doc sono scritti in HTML, i cui tag possono essere liberamente utilizzati per controllarne l'aspetto; può essere comodo, ad esempio, separare tramite il tag HTML `< p >` i paragrafi del commento.

Un esempio di commento doc è:

```
/**
 * Sezione descrittiva del commento doc
 *
 * @tag Commento per il tag
 */
```

Alcuni dei tag più ricorrenti nelle documentazioni sono:

`@author [nome]`

Aggiunge 'Author' seguito dal nome specificato. Ogni commento doc può contenere molteplici tag `@author`, presentati in ordine cronologico.

`@version [versione]`

Aggiunge ‘Version’ seguito dalla versione specificata.

`@param [nome del parametro] [descrizione]`

Aggiunge il parametro specificato e la sua descrizione alla sezione ‘Parameters’ del metodo corrente. Il commento doc riferito ad un certo costruttore o ad un certo metodo deve obbligatoriamente presentare un tag `@param` per ognuno dei parametri attesi, nell’ordine in cui l’implementazione del costruttore o del metodo specifica i parametri stessi.

`@return [descrizione]`

Aggiunge ‘Returns’ seguito dalla descrizione specificata. Indica il tipo restituito e la gamma di valori possibili. Può essere inserito solamente all’interno del codice sorgente Java, dove deve essere obbligatoriamente usato per ogni metodo, a meno che questo non sia un costruttore oppure non presenti alcun valore di ritorno (void).

`@throws [nome completo della classe] [descrizione]`

Aggiunge ‘Throws’ seguito dal nome della classe specificata (che costituisce l’eccezione) e dalla sua descrizione. Il commento doc riferito ad un certo costruttore o ad un certo metodo deve presentare un tag `@exception` per ognuna delle eccezioni che compaiono nella sua clausola throws, presentate in ordine alfabetico.

`@see [riferimento]`

Aggiunge ‘See Also’ seguito dal riferimento indicato.

`@since [versione]`

Utilizzato per specificare da che momento l'entità di riferimento (classe, interfaccia, metodo, costruttore, campo) è stata inserita nell'API.

La sintassi di javadoc è la seguente:

```
javadoc [opzioni] [package] [file sorgenti] [@mieifile]
```

In JVE la documentazione viene generata con i comandi:

```
set CLASSPATH=%CLASSPATH%;/Swarm-2.2/share/swarm/swarm.jar;  
lib/gui.jar;lib/plot.jar;lib/xlrd.jar;lib/jveframe.jar
```

```
md doc
```

```
javadoc *.java -author -d doc\
```

Il comando *set CLASSPATH* serve a indicare dove trovare i package necessari a Swarm e JVE in formato archivio JAR. Il comando *md* serve a creare la directory nella quale posizionare le pagine del commento, mentre il comando *javadoc* viene utilizzato con l'opzione *-author* per fare in modo che stampi l'autore delle classi.

Il Java Archive Tool

Il Java Archive Tool⁸ è un tool standard Java che permette di memorizzare in uno stesso archivio (detto appunto archivio JAR) diversi file. In realtà JAR è un tool di archiviazione e compressione general-purpose, che utilizza il formato di compressione ZIP e ZLIB, anche se è nato con lo scopo di memorizzare in uno stesso file più classi Java (i file .class) che costituiscono un applet o un'applicazione, insieme ai vari file grafici, sonori, o altri archivi utili agli applet o alle applicazioni.

⁸ Per approfondimenti sui tool java cfr.
<http://www.lorenzobettini.it/articoli/javatools/javatools.html>

La sintassi di JAR non si discosta molto dal comando tar in ambiente Unix, che in pratica svolge le stesse mansioni.

```
jar cf myjarfile *.class
```

L'opzione c sta per create ed f per file; in questo modo si memorizzano nel file myjarfile tutti i file con estensione '.class'. Per estrarre i file si utilizzerà l'opzione x (extract), mentre se si vuole solo ottenere a schermo la lista dei file contenuti nell'archivio si potrà utilizzare l'opzione t. Ad esempio

```
jar xf myjarfile
```

estrae i file contenuto nell'archivio specificato.

Come già accennato, queste opzioni sono identiche a quelle del comando Unix tar; una differenza con quest'ultimo comando è che mentre in tar, quando si crea un file, non viene applicata di default nessuna compressione, in jar, i file vengono memorizzati in formato compresso, se si volesse evitare che questo avvenga (cioè se si vuole solo effettuare uno store dei file nell'archivio) si dovrà specificare in fase di creazione dell'archivio l'opzione -O. Specificando anche -V si esegue il comando in modalità verbose, cioè si ottiene l'output di tutto quello che viene memorizzato (o estratto).

Utilizzando questi file è possibile applicare una firma digitale all'archivio, in modo da poter risalire all'autore per motivi di sicurezza e affidabilità del programma eseguito.

Nel caso di JVE è stato utilizzato un archivio jar per contenere la versione stabile x.y. Consente di confrontare il funzionamento della versione stabile rispetto alla versione modificata successivamente, senza dover ricompilare il codice. Risulta molto utile disporre di questa versione JAR di JVE soprattutto in fase di sviluppo. Per avviare JVE dall'archivio JAR è sufficiente avviare lo script linux *jdkswarm.Jve2* scrivendo nella bash cygwin:

```
make runJar
```

Per creare l'archivio JAR è sufficiente scrivere:

```
make jar
```

2.1.4 *Extreme Programming*

L'Extreme Programming⁹ è una filosofia di programmazione ed un insieme di regole da seguire.

Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, and courage. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation.¹⁰

Queste regole seguono le recenti metodologie di programmazione e si possono sintetizzare in:

Write test first: I test tradizionalmente sono relegati alla conclusione del progetto, dopo che tutto è funzionante. Hanno generalmente un'importanza minore rispetto alla programmazione, e spesso vengono svolti da persone diverse dagli sviluppatori. Secondo la filosofia Extreme Programming i test hanno la stessa priorità dello sviluppo del codice. Si raccomanda di preparare i tests prima di aver scritto il codice, e includerli nel programma stesso. Il test deve essere eseguito con successo ogni volta che si apportano delle modifiche al codice. L'effetto è una chiara definizione delle interfacce alle classi, che obbliga a specificare esattamente le caratteristiche e i comportamenti della classe. Il test

⁹ Sul concetto di extreme programming si veda <http://www.xprogramming.com>

¹⁰ Ron Jeffries (2001)

risulta il modo più semplice per mostrare il funzionamento della classe stessa ai suoi utilizzatori. Il secondo beneficio di utilizzare i test in fase di sviluppo è la presenza di un controllo sintattico in fase di compilazione, riducendo così il tempo di debugging. Quando si lavora con progetti molto grandi, proprio come JVE, è indispensabile introdurre un test che venga eseguito ogni volta che si apporta una pur piccola modifica, per controllare che questa modifica non abbia disallineato, in punti magari impensabili, il funzionamento del programma. Solo con un test che produca sempre gli stessi risultati, e che copra tutti gli aspetti del programma, si può essere sicuri che la modifica non abbia corrotto il suo funzionamento.

Pair programming: Secondo questo principio il codice dovrebbe essere scritto da due persone per ogni workstation. Questa pratica assicura che tutto il codice sia controllato da almeno un altro programmatore, e risulti un migliore disegno. Potrebbe sembrare inefficiente avere due programmatori che svolgono lo stesso lavoro, ma ricerche mostrano che, in questo modo, si riesce a produrre di più riducendo il tempo di debugging. Questa tecnica, oltre migliorare i risultati, contribuisce allo scambio di conoscenze tra i programmatori. In questo modo i *pair programmers* riescono ad affrontare problemi più complicati, che individualmente avrebbero richiesto molto tempo per essere risolti. Nel mondo delle simulazioni il principio del *pair programming* è molto utile e viene applicato in un modo un po' diverso. Le simulazioni sono l'espressione, attraverso un programma per computer, di una teoria. Per la loro natura generalmente vengono sviluppate da un solo ricercatore, colui che ha creato la teoria. Il principio di *pair programming* viene applicato da un altro ricercatore che cerca di ricostruire da zero la stessa simulazione, generalmente con strumenti diversi. Se questa seconda simulazione, che incorpora la stessa teoria di quella principale, dà i medesimi risultati, si ha una ulteriore garanzia di correttezza

del funzionamento del programma, e di bontà del modello.

2.2 Swarm

Per realizzare una simulazione economica ad agenti si pongono una serie di difficoltà pratiche, legate a problemi tecnici ed informatici. Gli scienziati sociali spesso non sono anche programmatori, e non posseggono le conoscenze necessarie per sviluppare un'applicazione informatica complessa come una simulazione. Per ovviare a queste difficoltà tecniche, e per permettere ai ricercatori di concentrarsi sul modello che intendono simulare, esistono diversi applicativi e progetti. Per sviluppare JVE si è scelta la strada più difficile, ma più rigorosa e flessibile, utilizzare il progetto Swarm¹¹. Per poter utilizzare Swarm è necessario conoscere un linguaggio di programmazione a basso livello, Java, ma in contropartita si ha la possibilità di creare qualsiasi tipo di simulazione *Agent Based* in piena libertà. Con Swarm si può realizzare praticamente qualsiasi modello ad agenti relativo a fenomeni chimici, fisici, economici, dovuti ad interazione delle leggi naturali, antropologiche, di modelli ecologici o relativi alle scienze politiche e sociali. Ciò è possibile poiché tutto il codice informatico relativo al comportamento degli agenti deve essere integralmente scritto dallo sperimentatore; Swarm fornisce soltanto funzioni generiche che possono facilitare il lavoro di scrittura del codice, come la gestione dei tempi, il controllo dei cicli di funzionamento e l'interazione grafica con l'utente.

Il progetto Swarm è nato nel 1994 da un'idea di Chris Langton al Santa Fe Institute¹² ed è distribuito con una licenza GNU-GPL. L'obiettivo è stato creare sia un linguaggio, sia un set di strumenti informatici, per lo sviluppo di modelli di simulazione multi agente (ABM, Agent Based Model).

Swarm viene definito dai suoi autori, cit.(Swarm Development Group, 2000):

¹¹ La home page del progetto è <http://www.swarm.org>

¹² Cfr. <http://santafe.edu> (2002) in New Mexico. Si basa su un'organizzazione no-profit, lo *Swarm Development Group*

The whole idea of Swarm is to provide an execution context within which a large number of objects can live their lives and interact with one another in a distributed, concurrent manner.

In sostanza Swarm è una biblioteca di funzioni e il protocollo per utilizzarle. Le sue principali caratteristiche sono:

Linguaggio Object-Oriented: Le librerie di Swarm sono scritte in *Objective-c*, un linguaggio che segue la filosofia OOP. In Swarm gli agenti sono costruiti all'interno di oggetti. Le popolazioni di agenti sono le classi, e un particolare agente è un'istanza della classe. Ogni oggetto contiene internamente le proprie variabili di stato, ma il suo comportamento è definito dai metodi della classe. Ogni azione dell'agente richiama un metodo che viene eseguito dallo specifico agente.

Programmi Gerarchici: Le applicazioni hanno una struttura che si sviluppa in modo rigido e gerarchico. Al livello più alto è creato l'*Observer*, l'osservatore dell'esperimento, che genera i grafici e gestisce gli output della simulazione. Il livello sottostante contiene il *Model Swarm*, il modello che si vuole simulare. Il *Model Swarm* a sua volta genera i singoli agenti, gestisce gli eventi nel tempo, raccoglie informazioni su ciò che avviene nel modello e le passa all'*Observer*.

Tools: Swarm comprende numerosi tool che facilitano la creazione della simulazione. Queste parti di codice si occupano, per esempio, della gestione della memoria, di maneggiare le liste, controllare lo scorrere del tempo e programmare le azioni in una sorta di calendario definito *Schedule*.

2.2.1 Sciami di agenti

Johnson and Lancaster (2000b) affermano che:

Swarm is designed to help researchers build models in which low-level actors interact (often called complex systems). One research goal is to discern overall patterns that emerge from these detailed behaviors at the individual level.

La Fig. 2.4¹³ mostra lo schema di una simulazione con swarm che nasce dal basso, cioè dalla descrizione dei singoli agenti.

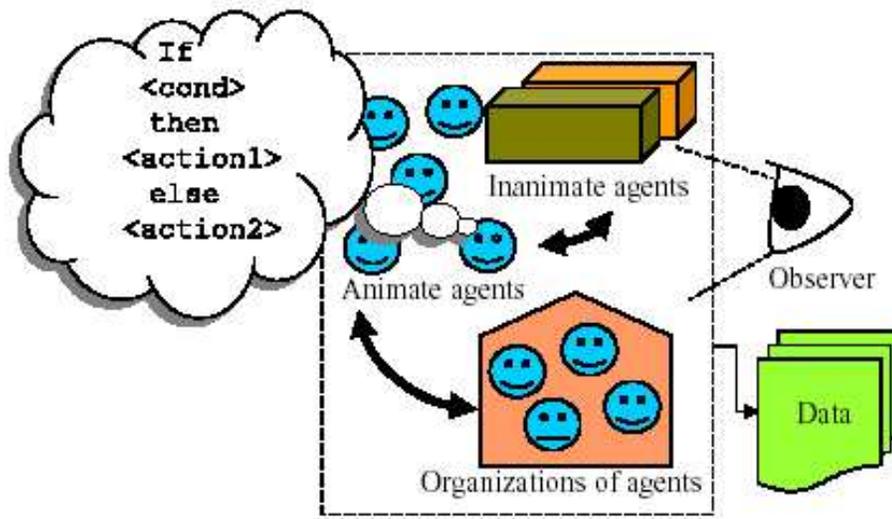


Fig. 2.4. Prototipo di AB Model secondo lo schema Bottom-Up

Come afferma Terna (1998):

...we have to emphasise the meaning of the arrow linking ABMs to the complexity and emergence paradigms ... We have also to explain the more complicated meanings both of the broken arrow going from Intelligent Agents to Complexity and Emergence and of the question mark accompanying it. It is possible and useful to employ Intelligent Agents as metaphors of economic agents and of their behaviour, considering the

¹³ Immagine tratta da Johnson and Lancaster (2000b)

possibility of interpreting the outcomes of the actions of these pieces of intelligent software as economic or social acts.

In una simulazione con Swarm un gran numero di oggetti vivono vita propria all'interno del modello, e interagiscono uno con l'altro in maniera distribuita e concorrenziale.

Il componente principale delle simulazioni ad agenti è un oggetto detto 'swarm', letteralmente 'sciame', ma sciame di che cosa? In generale di agenti, in particolare dipende dalla ricerca che si vuole svolgere; potrebbero essere formiche, agenti di borsa, conigli e coyote che si inseguono, o unità produttive di un'azienda come nel caso di JVE. Lo sciame contiene anche un calendario degli eventi che accadono nel mondo simulato. Per esempio lo swarm potrebbe essere una colonia di insetti che muovendosi nello spazio crea calore¹⁴, e le azioni potrebbero essere i movimenti per cercare di raggiungere una certa temperatura. Lo swarm in questo caso rappresenta l'intero modello, sia gli agenti che le loro azioni. Oltre che collezioni di agenti gli swarms possono essere agenti a loro volta. Un agente è tipicamente definito da un set di regole e di risposte agli stimoli (nel caso del nostro esempio la ricerca del calore come risposta alla temperatura del mondo esterno). I singoli agenti potrebbero essere swarms a loro volta, in questo caso il comportamento degli agenti è definito dai fenomeni emergenti dalla simulazione. L'approccio gerarchico utilizzato consente di creare una simulazione strutturata su più livelli, creando Swarms e SubSwarms, che si può rappresentare come nella Fig. 2.5¹⁵.

La possibilità di costruire modelli multi livello è molto potente, consente di rappresentare esplicitamente un fenomeno emergente, nel quale un gruppo di agenti, (che si definisce *popolazione*) possono comportarsi coerentemente come un singolo individuo.

¹⁴ Questo esempio è stato realmente realizzato con il nome 'heatbugs', ed disponibile all'indirizzo <http://www.swarm.org>

¹⁵ Tratta da Johnson and Lancaster (2000b)

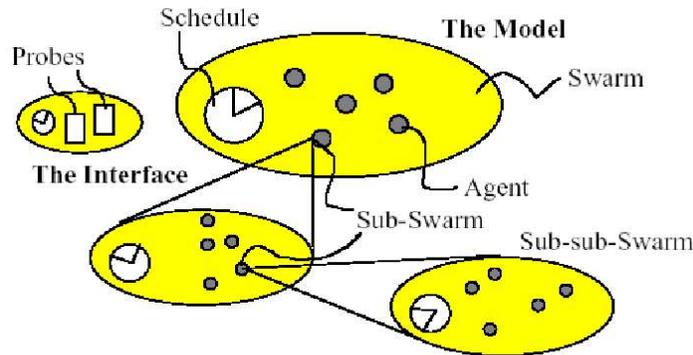


Fig. 2.5. Gerarchia multi-livello di una simulazione con Swarm

Il procedimento sperimentale

In un esperimento con Swarm si segue la procedura dettagliatamente descritta dal manuale Johnson and Lancaster (2000a), che si può sintetizzare con:

1. Creare un universo virtuale costituito da spazio, tempo e oggetti, che possono essere collocati in certe posizioni spazio-temporali dell'universo. Gli oggetti hanno dei comportamenti in accordo con il loro stato interno e lo stato dell'universo.
2. Creare gli strumenti per l'osservazione del modello. Questi oggetti servono a registrare e analizzare i dati prodotti dagli agenti nell'universo artificiale.
3. Eseguire la simulazione, attivando contemporaneamente il modello e gli strumenti di osservazione di questo.
4. Interagire con l'esperimento attraverso i dati prodotti dagli strumenti di osservazione, per eseguire dei controlli sulla simulazione.
5. In funzione dei risultati ottenuti tornare al punto precedente modificando la simulazione per verificare nuove ipotesi, sorte dall'osservazione del modello.

6. Pubblicare i risultati ottenuti, avendo cura di includere le specifiche tecniche dell'esperimento, così da renderlo replicabile e falsificabile.

2.2.2 *Perché Java*

Il *tool* di sviluppo *Swarm* potrebbe essere utilizzato sia in *Objective-C*, sia in *Java*.

Entrambi i linguaggi di programmazione seguono la filosofia *objectoriented*, ma la scelta per realizzare JVE è ricaduta su *Java*, perché oggi probabilmente è il più diffuso ed evoluto¹⁶.

I vantaggi di utilizzare *Java* rispetto a *Objective-C* sono:

- Semplice da usare, ma i nomi dei metodi sono generalmente più lunghi.
- Si scrivono meno files, in *Objective-C* ogni classe è costituita da 2 file, in *Java* ne basta uno.
- Incoraggia lo 'static typing', quindi forza a evitare alcuni errori (specialmente di runtime), che si possono fare con *Objective-C*.
- *Java* fa automaticamente il 'garbage collecting', cioè distrugge automaticamente gli oggetti non usati, prevenendo quindi i 'memory leaks'.
- L'interfaccia *Java* offre la possibilità di sfruttare delle numerose librerie grafiche di *Java* (che sono più ricche di quelle offerte da *Swarm*).
- È più utile imparare *Java*, perché è molto diffuso, soprattutto per la creazione di siti Internet e Applet.

Utilizzare *Java* ha anche alcuni svantaggi:

- Esistono meno risorse, tutorials e programmi per *Swarm* in *Java*.

¹⁶ Per maggiori informazioni cfr. Bissey (2001)

- *Java* non è il linguaggio nativo di *Swarm*, dunque certe funzioni possono essere più lenti di quelle scritte con *Objective-C*, a causa delle necessarie traduzioni.
- L'utilizzo della Java Virtual Machine, che garantisce la compatibilità dei programmi Java con tutti i sistemi operativi, rallenta ulteriormente l'esecuzione.

2.2.3 *Swarm*, un progetto libero

Altra caratteristica importante è che sia *Swarm* sia *Java* sono progetti liberi, o se si preferisce *Open Source*¹⁷. Questo significa che tutti i sorgenti sono disponibili e visionabili dagli sviluppatori, e il loro utilizzo è completamente gratuito.

La filosofia del *free software*¹⁸ si contrapposizione alla filosofia del software proprietario (della quale la *Microsoft* è il principale esponente), che concepisce il codice come un segreto industriale coperto da copyright. *Swarm* è rilasciato con una licenza GNU-GPL¹⁹. Il modello di sviluppo definito dal *free software* è particolarmente adatto a *Swarm*, sia per motivi teorici sia pratici²⁰:

Completa osservabilità: Grazie alla completa disponibilità del codice sorgente, se necessario, gli utilizzatori possono controllare l'esecuzione della simulazione a basso livello, fino al sistema operativo stesso. Questo aspetto è molto importante per la perfetta replicabilità degli esperimenti, e garantisce la possibilità di dimostrare, la coerenza interna del modello.

Condivisione delle conoscenze: Essendo *swarm open source* i ricercatori possono liberamente identificare bug, scrivere correzioni, implementare nuove funzioni e, in generale, contribuire all'evoluzione del progetto.

¹⁷ Per una definizione di *Open Source* si veda <http://www.opensource.org/osd.html> (2002)

¹⁸ Cfr. <http://www.gnu.org/philosophy/freesw.html> (2002)

¹⁹ GNU è una associazione che si occupa della tutela e diffusione del software libero, l'acronimo significa Gnu is Not Unix. Per GPL si intende General Public License cfr. <http://www.gnu.org/copyleft/gpl.html> (2002)

²⁰ In accordo con la documentazione *Swarm* di Johnson and Lancaster (2000b)

Queste sono le stesse ragioni che hanno consentito al sistema operativo GNU/Linux di crescere velocemente, e diventare molto robusto. Lo stesso sistema operativo GNU/Linux è la piattaforma per la quale era stato sviluppato originariamente *Swarm*, oggi purtroppo le ultime release²¹ risultano difficilmente installabili sotto linux, e la compilazione del codice sorgente di *Swarm* è problematica²². L'alternativa che viene utilizzata è l'ambiente *cygwin*²³. Si tratta di un emulatore Linux che apre in Windows una finestra, definita *Bash*²⁴, sul mondo Linux.

L'uso di tecnologie *Open Source* è un'ulteriore garanzia di scientificità e trasparenza della ricerca, perché si consente a chiunque (ma in particolare a qualunque altro ricercatore) di visionare il codice e comprendere o criticare il modello. Viene così rispettato il requisito della *falsificabilità*, necessario affinché la teoria incorporata nel modello si possa definire scientifica.

La strategia *open source* è utilizzata anche per incoraggiare i contributi della comunità degli utenti e sviluppatori *Swarm*. Oggi molte librerie sono state create al di fuori dello *Swarm Development Group*, e esiste una vivace lista di discussione²⁵ che permette di far conoscere i propri lavori.

2.2.4 Creare un universo virtuale con Swarm

La realizzazione di un modello di simulazione, secondo la metodologia introdotta dagli autori di Swarm, prevede la realizzazione di tre passi successivi:

1. Generazione del *ModelSwarm*.

²¹ Con Particolare riferimento a Swarm 2.2 pretest 4

²² Tentando di compilare i sorgenti con GCC vengono evidenziati dei 'warnings' che non consentono la corretta compilazione. Per poterla eseguire è necessario applicare una 'patch' al GCC, che elimina il controllo che genera i 'warnings', ma corrompe il corretto funzionamento del compilatore

²³ Un emulatore Linux prodotto dalla *Red Hat* cfr. <http://www.cygwin.com> (2002)

²⁴ La 'Bash' è un prompt dei comandi Linux testuale, il cui acronimo significa Bourne Again SHell, essendo la seconda versione della Shell, letteralmente 'conchiglia' cioè ambiente operativo, creata da Bourne

²⁵ Per potersi iscrivere alla lista cfr. <http://www.swarm.org> (2002)

- (a) Definizione delle caratteristiche di un agente.
 - (b) Generazione della popolazione di agenti.
 - (c) Creazione dello ‘Spazio’ dell’universo.
 - (d) Creazione dello *Schedule* del modello.
2. Generazione dell’*ObserverSwarm*.
- (a) Creazione dei Grafici
 - (b) Creazione delle *Probe*
3. Incollare insieme le parti della simulazione e attivarle²⁶

Le classi di Swarm

Swarm contiene un set di classi²⁷ scritte in *Ojective-C*, che sono:

defobj: La classe principali della gerarchia. Definisce i metodi per la creazione e la distruzione degli oggetti. Contiene anche classi che permettono di generalizzare instance, e creare instance basate su dati in diversi formati, così da poter generare semplicemente popolazioni di agenti.

objectbase: Contiene due classi *SwarmObject* e *Swarm*. La prima viene usata dalle sottoclassi di agenti per tenere sotto controllo alcune variabili. Attraverso le *Probe* vengono visualizzati i valori graficamente. La seconda è usata dalle sottoclassi di agenti per gestire la creazione e distruzione delle popolazioni. Controlla anche la sequenza degli eventi nella simulazione.

activity: Crea lo *Schedule*, cioè il calendario degli eventi, e aggiorna i grafici nella GUI.

²⁶ Questa procedura serve a Swarm per allocare correttamente la memoria e si definisce *Activate* cfr. Johnson and Lancaster (2000a)

²⁷ Per maggiori dettagli sul funzionamento delle classi cfr. Luna and Stefansson (2000)

collections: Gestisce le liste, i vettori e le collezioni di oggetti. I metodi di questa classe permettono di inviare messaggi a tutti gli oggetti di una collezione. Gli oggetti possono essere aggiunti rimossi o ordinati.

space: Questa classe consente di inserire o controllare la presenza di dati e oggetti su di una superficie bidimensionale strutturata come una griglia e definita *discrete 2d lattice*. Sono anche presenti funzione che consentono di leggere dati da files e applicare algoritmi agli oggetti disposti nello spazio.

gui: Permette la creazione dell'interfaccia grafica (Graphic User Interface). Definisce molti tipi di grafici, e gestisce la risposta del programma alle interazioni dell'utente.

analysis: Contiene metodi per la creazione di grafici su serie di tempo. Si occupa anche di gestire l'aggiornamento in tempo reale degli output grafici.

random: Una biblioteca di funzioni per la generazione di numeri casuali, e pseudo-casuali. Si basa su numerose distribuzioni statistiche (normale, gamma, beta, gaussiana). L'utente specifica e salva la distribuzione utilizzata, per poterla riutilizzare in test successivi, che diventano così confrontabili.

simtools: Contiene gli strumenti per interagire con files esterni, leggendo o salvando informazioni.

Un esempio applicativo, la nascita di JVE

Per far comprendere il funzionamento pratico di Swarm, descriverò sinteticamente il processo di creazione di un'applicazione. Per poterlo fare nel modo più semplice possibile mi aiuterò con degli esempi tratti del codice sorgente di JVE²⁸, con il

²⁸ Il codice utilizzato è tratto da jveframe-0.9.7.01

duplice scopo di far comprendere il funzionamento di *Swarm* in generale, e avvicinare il lettore al modello JVE, che sarà oggetto di approfondimento nei prossimi capitoli.

StartVEFrame.java

Per creare un modello si inizia costruendo la prima classe con un nome esplicativo, in modo da far capire che si tratta di quella iniziale. Per convenzione si utilizza nel nome della classe la parola ‘start’. Nel modello jve si usa ‘StartVEFrame.java’

```
import swarm.Globals;  
import java.io.*;
```

La prima riga richiama (tramite il comando *import*), all’interno della classe che stiamo esaminando, la classe *Globals* del package *swarm* per ereditarne le caratteristiche.

```
public class StartVEFrame{
```

Viene definito il costruttore

```
public static VEFrameObserverSwarm vEFrameObserverSwarm;
```

All’interno della classe vengono dichiarate le variabili. L’Observer sarà *vEFrameObserverSwarm*.

```
public static void main (String [] args) {
```

La funzione *main* si trova al livello più alto ed è da dove tutto ha inizio.

```
Globals.env.initSwarm ("jveframe",  
"v.0.9.7.01.a",  
"pietro.terna@unito.it", args)
```

Attraverso questa stringa viene inizializzato Swarm, al quale si indica il nome dell'applicazione, la versione e l'indirizzo email dell'autore.

```
vEFrameObserverSwarm =  
(VEFrameObserverSwarm)  
Globals.env.lispAppArchiver.getWithZone$key(  
Globals.env.globalZone, "vEFrameObserverSwarm");  
VEFrameObserverSwarm (Globals.env.globalZone);
```

Queste righe servono a creare l'ObserverSwarm che è definito

vEFrameObserverSwarm. La creazione avviene utilizzando i parametri contenuti nel file *jveframe.scm*, un archivio in formato LISP che viene gestito da *Swarm*, e definisce i valori di default della simulazione. Mantenendo i parametri in un file esterno al codice si ha il vantaggio di non dover ricompilare i sorgenti ogni volta che si modificano i valori.

```
Globals.env.setWindowGeometryRecordName  
(vEFrameObserverSwarm, "vEFrameObserverSwarm");
```

Con questi comandi viene salvata la posizione delle finestre sullo schermo.

```
vEFrameObserverSwarm.buildObjects ();
```

Vengono creati in memoria gli oggetti legati all'Observer.

```
vEFrameObserverSwarm . buildActions ( ) ;
```

Vengono predisposte le azioni che svolgerà l'Observer.

```
vEFrameObserverSwarm . activateIn ( null ) ;
```

Tutto ciò che è stato creato viene incollato insieme e predisposto per funzionare.

```
vEFrameObserverSwarm . go ( ) ;
```

Il metodo *go()* ha il compito di avviare tutte le attività, e permettere all'utente di gestire manualmente la simulazione tramite il pannello di controllo. I tasti sono 'Start' per avviarla, 'Stop' per interromperla, 'Next' per procedere tick per tick, e 'Quite' per chiudere il programma.

```
try {  
    if (vEFrameObserverSwarm . vEFrameModelSwarm .  
        concludedOrderLog != null)  
vEFrameObserverSwarm . vEFrameModelSwarm .  
        concludedOrderLog . close ( ) ;  
vEFrameObserverSwarm . vEFrameModelSwarm .  
concludedOrderLog=null ;  
}  
catch (IOException e) {  
    System . out . println (e) ;  
    System . exit (1) ;  
}
```

In un blocco *try-catch* viene gestita la chiusura dei file di log generati dalla simulazione.

```
vEFrameObserverSwarm . drop ( ) ;  
VEFrameModelSwarm ourModel =  
vEFrameObserverSwarm . vEFrameModelSwarm ;  
ourModel . drop ( ) ;
```

Vengono chiusi e tolti dalla memoria l'*Observer*, il *Model* e alcuni istogrammi generati al di fuori delle funzioni *Swarm*²⁹

```
System . exit ( 0 ) ;
```

Questa è la conclusione del programma.

VEFrameObserverSwarm.java

In questa classe viene definito l'*Observer*. Il file inizia importando tutte le classi di *Swarm* già descritte nella Sezione 2.2.4 per poterle utilizzare nella classe. In aggiunta viene importata la biblioteca di primitive Java *java.util*. La sintassi è la seguente.

```
import swarm . nomeClasse ;  
import java . util . * ;
```

²⁹ Per generare gli istogrammi di JVE è stato utilizzato PTHistogram del progetto Ptolemy <http://ptolemy.eecs.berkeley.edu> (2002)

L'Observer contiene il Model e gli oggetti grafici per la rappresentazione dell'andamento dell'impresa virtuale. Estendendo la classe GUISwarmImpl la classe VEFrameObserverSwarm ne diventa direttamente una sottoclasse, sarà quindi in grado di utilizzare tutti i metodi della superclasse ed inoltre aggiungerà altri metodi propri utilizzabili poi solo dagli oggetti di tipo VEFrameObserverSwarm o da riferimenti ad oggetti della stessa.

```
public class VEFrameObserverSwarm extends GUISwarmImpl {
```

Vengono di seguito definite tutte le variabili della simulazione che riguardano l'osservatore, e che sarà possibile definire sia dallo schermo quando viene avviato il modello, sia modificando il file *jveframe.scn*.

```
/** update frequency */  
public int displayFrequency;  
  
/** displaying all messages on the console */  
public boolean verboseChoice;  
  
/** displaying memory information on the console */  
public boolean checkMemorySize;
```

Vengono definite le variabile che gestiscono le sequenze degli eventi grafici:

```
public ActionGroup displayActions1 , displayActions2;
```

Viene definita la variabile che gestisce lo Schedule:

```
public Schedule displaySchedule;
```

In fine la cosa più importante: viene creata un'istanza del ModelSwarm.

```
public VEFrameModelSwarm vEFrameModelSwarm;
```

Il codice seguente gestisce l'output su files dei grafici.

```
public EZGraphImpl  
waitingListGraph=null, warehouseGraph=null,
```

Il codice seguente genera gli istogrammi realizzati utilizzando il progetto Ptolemy³⁰, e ne definisce i parametri di osservazione.

```
public PTHistogram pTHistogram1, pTHistogram2;  
public int unitHistogramXPos, unitHistogramYPos,  
endUnitHistogramXPos, endUnitHistogramYPos;
```

Vengono definite le sonde.

```
Object observerProbe;
```

Ecco il costruttore, che richiama la super classe³¹, e inizializza le variabili.

³⁰ Il progetto Ptolemy, è curato dal Department of Electrical Engineering and Computer Science, University of California Berkeley, <http://ptolemy.eecs.berkeley.edu>

³¹ In Swarm ogni costruttore deve richiamare Zone e passare alla superclasse la zona di memoria dell'oggetto

```

public VEFrameObserverSwarm (Zone aZone) {
super(aZone);
displayFrequency = 1;
verboseChoice=false;
checkMemorySize=false;
unitHistogramXPos = 10;
unitHistogramYPos = 300;
endUnitHistogramXPos= 10;
endUnitHistogramYPos= 250;

```

Viene costruita una mappa delle sonde usando una instance locale di *EmptyProbeMapImpl*. Questa operazione produce diversi files class con nome ‘*VEFrameObserverSwarm1 VEFrameObserverProbeMap.class*’ per ogni sonda. Vengono anche indicate le variabili dell’Observer che le sonde devono controllare.

```

class VEFrameObserverProbeMap extends EmptyProbeMapImpl {
public VarProbe probeVariable (String name) {
return
Globals.env.probeLibrary.getProbeForVariable$inClass
(name, VEFrameObserverSwarm.this.getClass ());
}
public MessageProbe probeMessage (String name) {
return
Globals.env.probeLibrary.getProbeForMessage$inClass
(name, VEFrameObserverSwarm.this.getClass ());
}
public void addVar (String name) {
addProbe (probeVariable (name));
}
public void addMessage (String name) {
addProbe (probeMessage (name));
}

```

```

}
public VEFrameObserverProbeMap (Zone _aZone, Class aClass) {
super (_aZone, aClass);
addVar ("displayFrequency");
addVar ("verboseChoice");
addVar ("checkMemorySize");
addVar ("unitHistogramXPos");
addVar ("unitHistogramYPos");
addVar ("endUnitHistogramXPos");
addVar ("endUnitHistogramYPos");
} }

```

Installiamo la nostra mappa delle sonde direttamente in probeLibrary.

```

observerProbe=new VEFrameObserverProbeMap(aZone, getClass());
Globals.env.probeLibrary.setProbeMap$For
  ((VEFrameObserverProbeMap) observerProbe, getClass());
}

```

Vengono creati gli oggetti che dovranno essere visualizzati.

```

public Object buildObjects () {
super.buildObjects ();
}

```

Viene creato il ModelSwarm, come un subswarm dell'Observer. Viene creato attraverso l'archivio lisp che fa riferimento al file jveframe.scm, che contiene i parametri del modello.

```

vEFrameModelSwarm =

```

```
(VEFrameModelSwarm) Globals.env.lispAppArchiver.  
getWithZone$key(  
    getZone(), "vEFrameModelSwarm");
```

Poi vengono create le sonde.

```
Globals.env.createArchivedProbeDisplay  
(vEFrameModelSwarm,  
    "vEFrameModelSwarm");  
Globals.env.createArchivedProbeDisplay (this,  
    "vEFrameObserverSwarm");
```

Poi viene creato il pannello di controllo. La simulazione è in attesa degli input dei parametri da parte dell'utente in stato *Stopped*.

```
getControlPanel().setStateStopped();
```

Vengono creati gli oggetti del modello.

```
vEFrameModelSwarm.buildObjects();
```

Vengono creati i grafici delle liste d'attesa attraverso la classe Swarm *EZGraphImpl*, e viene salvata la posizione delle finestre.

```
waitingListGraph = new EZGraphImpl  
(getZone(), "Waiting_lists", "Time",  
    "Waiting_list_values_(avrg,min,max)", "waiting_list");  
Globals.env.setWindowGeometryRecordName (waitingListGraph,
```

```
"waitingListGraph");
```

Vengono creati i dati reattivi ai grafici delle liste d'attesa. Nello stesso modo verranno creati i dati e i grafici dei magazzini, il tempo di produzione, i costi totali giornalieri, i ricavi.

```
waitingListGraph.createAverageSequence$withFeedFrom$andSelector
    ("avrgWaitingList", vEFrameModelSwarm.getUnitList(),
    SwarmUtils.getSelector("Unit", "getWaitingListLength"));
waitingListGraph.createMinSequence$withFeedFrom$andSelector
    ("minWaitingList", vEFrameModelSwarm.getUnitList(),
    SwarmUtils.getSelector("Unit", "getWaitingListLength"));
waitingListGraph.createMaxSequence$withFeedFrom$andSelector
    ("maxWaitingList", vEFrameModelSwarm.getUnitList(),
    SwarmUtils.getSelector("Unit", "getWaitingListLength"));
```

Di seguito vengono salvati i risultati del tempo di produzione su file, con il comando *createSequencewithFeedFromandSelector*. Nello stesso modo verranno salvati i dati riguardanti i costi, i ricavi.

```
totalTimeLengthDataFile = new EZGraphImpl
    (getZone(), "data");
totalTimeLengthDataFile.createSequence$withFeedFrom$andSelector
    ("ratio", vEFrameModelSwarm,
    SwarmUtils.getSelector(vEFrameModelSwarm, "getTimeLengthRatio"));
```

Poi viene creato l'istogramma Ptholemy che visualizza le liste d'attesa delle unità. Allo stesso modo verrà creato quello per controllare la lista d'attesa delle end unit.

```
pTHistogram1 = new PTHistogram(unitHistogramXPos , unitHistogramYPos ,
    "Orders_in_Units" ,
    "Units" , "Count" ,
    vEFrameModelSwarm.totalUnitNumber ,
    20.0 ,
    vEFrameModelSwarm.unitList ,
    "Queues_in_u." ,
    vEFrameModelSwarm.warehouseList ,
    "Quant_in_w." ,
    vEFrameModelSwarm.
    procurementAssemblerList ,
    "Q_in_proc_ass.");
```

Si costruisce lo Schedule e le Azioni del mondo simulato.

```
public Object buildActions () {
super.buildActions ();
```

Poi nascono le Azioni del Model.

```
vEFrameModelSwarm.buildActions ();
displayActions1 = new ActionGroupImpl (getZone());
displayActions2 = new ActionGroupImpl (getZone());
```

Vengono descritte le azioni tutte con la stessa sintassi. In questo esempio si costruisce uno step del grafico *waitingList*.

```
displayActions.createActionTo$message
    (waitingListGraph ,
```

```
SwarmUtils.getSelector(waitingListGraph, "step"));
```

Bisogna aggiornare costantemente le sonde con i seguenti comandi.

```
displayActions2.createActionTo$message
(Globals.env.probeDisplayManager,
SwarmUtils.getSelector(Globals.env.probeDisplayManager, "update"));
```

Finalmente si avviano gli eventi grafici del Observer con il messaggio *doTkEvents*

```
displayActions2.createActionTo$message(getActionCache(),
SwarmUtils.getSelector(getActionCache(), "doTkEvents"));
```

Viene impostata la velocità di refresh delle immagini. Si imposta il valore di return del metodo.

```
displaySchedule = new ScheduleImpl (getZone (), displayFrequency);
return this;
}
```

Il metodo *activateIn* attiva le Activity, l' Observer, il Model e lo Schedule, in modo che la simulazione sia pronta a partire.

```
public Activity activateIn (Swarm swarmContext) {
super.activateIn (swarmContext);
vEFrameModelSwarm.activateIn (this);
displaySchedule.activateIn (this);
return getActivity ();
}
```

Alla fine della simulazione viene fatto il drop di tutti i grafici con la stessa sintassi (come esempio viene ripostato solo *waitingListGraph*), del pannello di controllo e della super classe.

```
public void drop() {
getControlPanel ().setStateStopped ();
waitingListGraph.drop ();
...
super.drop ();
}
```

VEFrameModelSwarm.java

Il modelSwarm, a differenza dell' Observer contiene gran parte del codice riguardante il disegno del modello specifico che si vuole simulare, in ogni caso è possibile individuare alcune procedure standard di Swarm.

Come prima cosa viene fatto l'import delle classi, poi il casting delle variabili. All'interno del costruttore viene richiamata la superclasse. Estendendo la classe SwarmImpl la classe VEFrameModelSwarm ne diventa direttamente una sottoclasse, sarà quindi in grado di utilizzare tutti i metodi della superclasse. Inoltre altri metodi saranno aggiunti, utilizzabili , o da riferimenti ad oggetti, di tipo VEFrameModelSwarm.

```
public class VEFrameModelSwarm extends SwarmImpl
...
{
    public VEFrameModelSwarm (Zone aZone) {
super (aZone);
    }
```

Successivamente vengono inizializzate le variabili della simulazione con i valori di default uguali a quelli contenuti nel file `jveframe.scm`.

```
useOrderDistiller=false ;
ticksInATimeUnit = 1;
totalUnitNumber = 10;
totalEndUnitNumber = 3;
totalLayerNumber=1;
totalMemoryMatrixNumber=4;
maxStepNumber=30;
maxStepLength=1;
useWarehouses=true ;
maxInWarehouses=3;
minInWarehouses=3;
useNewses=true ;
infDeepness=3;
inventoryFinancialRate=(float) 0.1;
inventoryEvaluationCriterion=1;
revenuePerEachRecipeStep=2;
nOfNewsesToProduce=1;
nOfNewsesToBeCleared=100000;
nOfOrdersInNewses=1;
orCriterion=0;
orMemoryMatrix=2;
unitCriterion=0;
distillerMultiplicity=1;
```

Anche in questo caso le Probe sono costruite come classe interna, estendendo la classe `EmptyProbeMapImpl`, ottenendo la classe `VEFrameModelSwarm1VEFrameModelProbeMap.class`

```

class VEFrameModelProbeMap extends EmptyProbeMapImpl {
    public VarProbe probeVariable (String name) {
    return
        Globals.env.probeLibrary.getProbeForVariable$inClass
        (name, VEFrameModelSwarm.this.getClass ());
    }
    public MessageProbe probeMessage (String name) {
    return
        Globals.env.probeLibrary.getProbeForMessage$inClass
        (name, VEFrameModelSwarm.this.getClass ());
    }
    public void addVar (String name) {
addProbe (probeVariable (name));
    }
    public void addMessage (String name) {
addProbe (probeMessage (name));
    }
    public VEFrameModelProbeMap (Zone _aZone, Class aClass) {
super (_aZone, aClass);
addVar ("useOrderDistiller");
addVar ("ticksInATimeUnit");
addVar ("totalUnitNumber");
    addVar ("totalEndUnitNumber");
addVar ("totalLayerNumber");
addVar ("totalMemoryMatrixNumber");
addVar ("maxStepNumber");
addVar ("maxStepLength");
addVar ("useWarehouses");
addVar ("useNewses");
addVar ("maxInWarehouses");
addVar ("minInWarehouses");
addVar ("infDeepness");
addVar ("inventoryFinancialRate");

```

```

addVar ("inventoryEvaluationCriterion");
addVar ("revenuePerEachRecipeStep");
addVar ("nOfNewsesToProduce");
addVar ("nOfNewsesToBeCleared");
addVar ("nOfOrdersInNewses");
addVar ("orCriterion");
addVar ("orMemoryMatrix");
addVar ("unitCriterion");
addVar ("distillerMultiplicity");
    }
}

```

La mappa delle sonde viene inserita direttamente nella ProbeLibrary

```

modelProbe=new VEFrameModelProbeMap (aZone, getClass ());
Globals.env.probeLibrary.setProbeMap$For
((VEFrameModelProbeMap) modelProbe, getClass ());
}

```

La costruzione vera e propria del VEFrameModelSwarm ha inizio in questo punto con la definizione di quali siano gli oggetti che determinano il comportamento della classe, cosa dovrà fare quando sarà attivata

```

public Object buildObjects ()
{
    ...
    super.buildObjects ();
}

```

A questo punto vengono inizializzati tutti gli oggetti del modello utilizzando il loro costruttore. La sintassi è simile per tutti gli oggetti, di seguito riporto la creazione delle unità.

```
unitList = new ListImpl (getZone());
unitListIndex = unitList.listBegin(getZone());
...
```

Il metodo `buildActions()`, come già descritto nel `VEFrameObserverSwarm`, serve per descrivere le azioni necessarie per avviare la simulazione e per costruire lo `Schedule` che gestisce la simulazione del tempo nel modello. Il metodo inizializza anzitutto lo stesso `buildActions()` della superclasse. Vengono quindi creati gli ‘ActionGroups’ per gestire le azioni che pongono in essere realmente la simulazione. Nel nostro caso sono sei: `modelActions1` `modelActions2` `modelActions2generator` `modelActions2distiller` `modelActions2b` `modelActions3`. Creare diversi ‘ActionGroups’ permette di dividere le azioni in gruppi omogenei e eseguirle in tempi diversi. Per esempio le azioni riguardanti `modelActions1` verranno eseguite solo all’avvio della simulazione, mentre quelle di `modelActions2` ad ogni tick della ‘giornata’ simulata.

```
public Object buildActions ()
{
    int i, ii;
    super.buildActions();

    modelActions1 = new ActionGroupImpl (getZone ());
    modelActions2 = new ActionGroupImpl (getZone ());
    modelActions2generator = new ActionGroupImpl (getZone ());
    modelActions2distiller = new ActionGroupImpl (getZone ());
    modelActions2b = new ActionGroupImpl (getZone ());
```

```
modelActions3 = new ActionGroupImpl (getZone ());
```

Poi vengono create tutte le azioni per ogni ‘ActioGroup’ seguendo la stessa sintassi. Nell’esempio si azzerà la contabilità all’inizio della simulazione.

```
modelActions1.createActionForEach$message( unitList ,
SwarmUtils.getSelector("Unit","unitStep0"));
...
```

Infine è creato lo Schedule che serve per eseguire il modelActions della classe ActionGroup, che da sola non avrebbe modo di gestire il tempo. Per eseguire tutte le azioni quando necessarie lo Schedule attiverà le ‘ActionGroups’ in momenti diversi. Per esempio la modelAction2 verrà attivata ad ogni tick. La sintassi generale è `modelSchedule.at$createAction (0, modelActions)`.

```
modelSchedule = new ScheduleImpl (getZone () , ticksInATimeUnit);

modelSchedule.at$createAction (0, modelActions1);

for ( i=0;i<ticksInATimeUnit;i++)
{
  modelSchedule.at$createAction (i, modelActions2);
  if(! useOrderDistiller)modelSchedule.
    at$createAction (i, modelActions2generator);
  if( useOrderDistiller && i==0)
    for(ii=0;ii<distillerMultiplicity;ii++)
      modelSchedule.
        at$createAction (i, modelActions2distiller);
  modelSchedule.at$createAction (i, modelActions2b);
}
```

```
modelSchedule.at$createAction (ticksInATimeUnit -1, modelActions3);  
  
return this;  
}
```

E' attivato lo schedule in modo che la simulazione sia pronta a partire. L'argomento *swarmContext* è la zona in cui *VEFrameModelFrame* è attivato. Essendo *VEFrameModelSwarm* un subswarm di *VEFrameObserverSwarm*, sarà attivato all'interno dell'ambiente dell'Observer. Poi verrà attivato lo Schedule del model e in fine verrà restituito lo stato della simulazione per permettere di gestire gli errori.

```
public Activity activateIn (Swarm swarmContext) {  
super.activateIn (swarmContext);  
modelSchedule.activateIn (this);  
return getActivity ();  
}
```

2.3 UML

Secondo la definizione dell' *Object Management Group* contenuta all'interno dell' *OMG Unified Modeling Language Specification*³²

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.

UML è un formalismo che permette, tramite l'utilizzo di modelli visuali, di analizzare, descrivere, specificare e documentare un sistema software anche complesso. UML è versatile ed espressivo, ciò consente di avere una visione chiara del sistema anche alle persone con figure professionali molto diverse tra loro.

Il crescere delle dimensioni e della complicazione delle applicazioni rende sempre più difficile costruire e gestire applicazioni di alta qualità in tempi brevi. Come risultato di tale difficoltà e dalla esigenza di avere un linguaggio universale per modellare gli oggetti, che potesse essere utilizzato da ogni industria produttrice di software, fu inventato il linguaggio UML, il cui acronimo significa Unified Modeling Language.

In termini pratici potremmo dire che il linguaggio UML rappresenta, in qualche modo, un progetto di ingegneria del software.

L'UML è, dunque, un metodo per descrivere l'architettura di un sistema in dettaglio. Come è facile intuire, utilizzando una sorta di progetto sarà molto più facile costruire o mantenere un sistema, e assicurarsi che il sistema stesso si presterà senza troppe difficoltà a futuri cambiamenti dei requisiti.

³² Citazione tratta da ObjectManagementGroup (1999)

La forza dell'Unified Modeling Language consiste nel fatto che il processo di disegno del sistema può essere effettuato in modo tale che i clienti, gli analisti, i programmatori e chiunque altro sia coinvolto nel sistema di sviluppo, possa capire ed esaminare in modo efficiente il sistema e prendere parte alla sua costruzione in modo attivo.

Durante gli anni '90 furono introdotte nel mercato dell'Information Technology parecchie metodologie per il disegno e la progettazione di sistemi software. Ognuna di queste tecnologie aveva il suo insieme proprio di notazioni e simboli, che differiva, a volta in modo rilevante, dalle altre. In particolare, eccellevano tre di queste metodologie:

- OMT di Rumbaugh
- Booch dell'autore omonimo
- OOSE e Objectory di Jacobson

Uno degli obiettivi che hanno caratterizzato il lavoro dell'OMG è stato realizzare un linguaggio di modellazione che unificasse e incorporasse le caratteristiche migliori dei linguaggi esistenti intorno agli anni Novanta. In particolare, come citato esplicitamente nel documento delle specifiche ObjectManagementGroup (1999), l'UML è scaturito principalmente dalla fusione dei concetti presenti nei metodi di Grady Booch, OMT (Object Modeling Technique di cui Rumbaugh era uno dei principali fautori) e OOSE (Object Oriented Software Engineering /Objectory di cui Ivar Jacobson era uno dei più importanti promotori). Agli inizi degli anni Novanta, ossia poco prima dell'avvio dei lavori per l'UML, i suddetti metodi erano probabilmente quelli più apprezzati, a livello mondiale, dalla comunità Object Oriented. Brevemente, il metodo di Booch, diventato famoso nel settore con il nome di 'metodo delle nuvolette'. Esso definisce una notazione in cui il sistema viene analizzato e suddiviso in un insieme di viste, ciascuna costituita dai diagrammi di modello. Il

metodo non si limita a un linguaggio di modellazione, ma contiene anche un processo, in base al quale, il sistema viene studiato attraverso micro e macro viste di sviluppo, secondo un classico schema incrementale e iterativo. Il metodo di Booch, a detta degli esperti, sembrerebbe molto efficace soprattutto in fase di disegno e di codifica, mentre presenterebbe qualche lacuna nelle varie fasi di analisi. L'OMT è un linguaggio di modellazione, sviluppato presso la General Electric, rilevatosi particolarmente efficace nella fase di esplorazione dei requisiti. In maniera speculare al metodo precedente, quest'ultimo sembrerebbe essere carente nella fase della formulazione delle soluzioni, mentre risulterebbe particolarmente accurato nell'esplorazione delle specifiche del problema. L'OMT prevede che il sistema venga descritto attraverso un preciso numero di modelli che si completano vicendevolmente. Grazie alla sua accuratezza nella fase di analisi dei requisiti, il modello fornisce un valido ausilio anche alla fase di test. Infine i metodi OOSE e Objectory (in gran parte dovuti al lavoro di Jacobson) si basano, quasi interamente, sugli Use Case (casi d'uso). Poiché gli Use Case permettono di definire i requisiti iniziali del sistema, così come vengono percepiti da un attore esterno, i metodi OOSE e Objectory si sono dimostrati particolarmente efficaci nell'analisi del problema. Anche questi metodi forniscono una serie di informazioni e linee guida su come passare dall'analisi dei requisiti al disegno del sistema. Altri metodi degni di essere menzionati sono Fusion (elaborato presso i laboratori della HewlettPackard) e il metodo Coad/Yourdon (noto anche come OOA/OOD – Object Oriented Analysis / Object Oriented Design) che vanta il primato di essere stato uno dei primi metodi di analisi e disegno di sistemi Object Oriented. Negli anni intercorsi tra il 1989 ed il 1994, oltre ai metodi appena citati, considerati tra i più importanti, se ne contarono addirittura una cinquantina. Questi andarono ad alimentare quella che fu definita 'guerra dei metodi', con la conseguente incomunicabilità. Ovviamente ciascuno di questi presentava punti di forza, lacune, un proprio formalismo, una nomenclatura proprietaria e un peculiare

processo di sviluppo. Tutto ciò finiva per minare alla base lo sviluppo di processi di progettazione più accurati e rendeva difficoltosa la realizzazione di opportuni tool di supporto. Altri inconvenienti, sempre frutto dell'incomunicabilità dei metodi, erano legati all'impossibilità di far circolare informazioni tra team di aziende diverse e alla difficoltà di rendere rapidamente produttive nuove risorse allocate al progetto. Ognuno di questi metodi aveva, naturalmente, i suoi punti di forza e i suoi punti deboli.

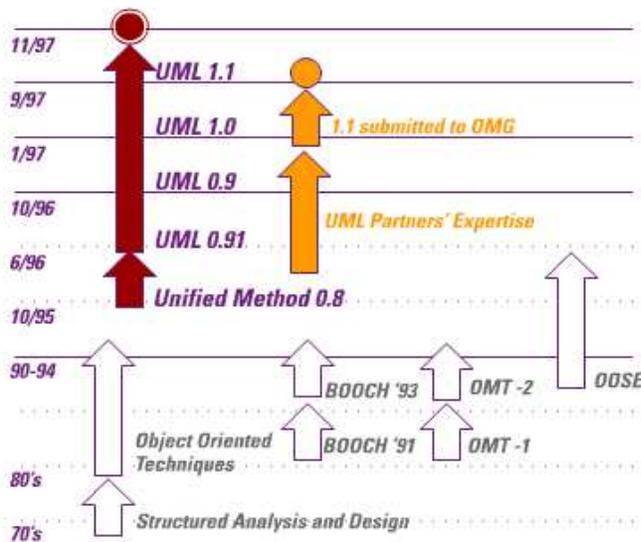


Fig. 2.6. Evoluzione dell'UML

Successivamente, Booch scrisse il suo secondo libro, che adottava i principi di analisi utilizzati da Rumbaugh e Jacobson nelle loro rispettive metodologie. A sua volta, Rumbaugh pubblicò una serie di articoli (conosciuti come OMT-2) che descrivevano parecchie delle tecnologie di disegno di Booch.

In sostanza, i tre metodi stavano convergendo verso un'unica visione che incorporasse le qualità migliori che ognuno di essi aveva mostrato. L'unico problema che restava era il fatto che ogni metodo portava ancora con sé la propria notazione. Tale

problema non era da sottovalutare, in quanto, l'uso di simbologia differente portava facilmente a confusione sul mercato, a causa del fatto che un determinato simbolo poteva avere un significato differente per analisti e disegnatori differenti.

Fu così che, nell'Ottobre del 1995, nacque la prima bozza dell'UML, ovvero l'unificazione delle notazioni e delle idee prodotte da Booch, Rumbaugh e Jacobson per modellare un sistema software. La prima versione ufficiale, prodotta dall'OMG (Object Management Group) fu rilasciata nel Luglio del 1997 e nel Novembre dello stesso anno l'UML venne adottato come standard.

Se a tutto ciò si aggiunge il fatto che il linguaggio è non proprietario si può ben capire che la sua affermazione era garantita sin dall'inizio della sua invenzione. Si tratta infatti di un linguaggio completamente aperto, tanto che le aziende sono incoraggiate a integrarlo nei propri metodi di sviluppo del software e che le imprese produttrici di CASE possono liberamente produrre software di supporto allo UML.

I benefici derivanti dall'utilizzo del linguaggio UML sono:

- Un sistema software, grazie al linguaggio UML, viene disegnato professionalmente e documentato, ancor prima che ne venga scritto il relativo codice, da parte degli sviluppatori. Si sarà così in grado di conoscere in anticipo il risultato finale del progetto su cui si sta lavorando.
- Poiché, come detto, la fase di disegno del sistema precede la fase di scrittura del codice, ne consegue che la scrittura del codice stesso è resa più agevole ed efficiente. In tal modo è più facile scrivere del codice riutilizzabile in futuro. I costi di sviluppo, dunque, si abbassano notevolmente con l'utilizzo del formalismo UML.
- È più facile prevedere e anticipare eventuali buchi nel sistema. Il software che si scrive, si comporterà esattamente come ci si aspetta, senza spiacevoli sorprese finali.

- L'utilizzo dei diagrammi UML fornisce una chiara idea, a chiunque sia coinvolto nello sviluppo, di tutto l'insieme che costituisce il sistema. In questo modo si potranno sfruttare al meglio anche le risorse hardware, in termini di memoria ed efficienza, senza sprechi inutili o, al contrario, rischi di sottostima dei requisiti di sistema.
- Grazie alla documentazione del linguaggio UML diviene ancora più facile effettuare eventuali modifiche future al codice. Questo, ancora, a tutto beneficio dei costi di mantenimento del sistema.
- La comunicazione e l'interazione tra tutte le risorse umane che prendono parte allo sviluppo del sistema è molto più efficiente e diretta. Parlare la stessa lingua aiuta ad evitare rischi di incomprensioni e quindi sprechi di tempo.

2.3.1 Fasi dello sviluppo

Un Sistema informatico è una combinazione di componenti software e hardware che fornisce una soluzione ad un'esigenza del mondo reale (business problem). Per far sì che un Sistema sia efficiente, sarà necessario recepire bene le richieste del cliente e fare in modo che esse vengano condivise e recepite come requisiti indispensabili dall'intero team di lavoro. Successivamente, si useranno tali requisiti per generare il codice necessario alla costruzione del Sistema assicurandosi, andando avanti nella scrittura del codice, che non si perda di vista l'obiettivo finale. Tale procedimento prende il nome di Modeling (o, se preferite, modellazione). Il vecchio metodo di modeling dei sistemi, conosciuto anche come Metodo a Cascata (Waterfall Method), si basava sull'assunto che i vari passi che costituiscono il processo di sviluppo di un sistema fossero sequenziali (l'inizio di uno avveniva soltanto dopo il termine di un altro). Quello che accadeva seguendo il waterfall method era che le persone del team si trovavano a lavorare su task differenti e, spesso, non avevano alcun modo di

comunicare, a discapito di importanti questioni inerenti il Sistema stesso. Un altro punto debole del waterfall method era il fatto che esso prevedeva che si concedesse la maggior parte del tempo alla scrittura del codice togliendolo, in tal modo, all'analisi ed al disegno, che invece rappresenta la base per un solido progetto.

Il Visual Modeling, che ha soppiantato ormai il waterfall method, altro non è che il processo che prevede la visualizzazione grafica di un modello, utilizzando un insieme ben definito di elementi grafici. Nel formalismo UML sono rappresentati dai 9 Diagrammi di base.

Il processo per la costruzione di un Sistema coinvolge molte persone: Prima di tutto il cliente, ovvero la persona che desidera una soluzione ad un problema ,o ad un'esigenza. Poi un analista, che ha il compito di documentare il problema del cliente, e trasmettere tali informazioni agli sviluppatori, i quali costituiscono la parte del team che si occupa di implementare il codice, testarlo (con il supporto di un team apposito per il test) e installarlo sull'hardware dei computer. Questo tipo di suddivisione dei compiti è necessario poiché al giorno d'oggi i sistemi sono diventati davvero molto complessi e la conoscenza è divenuta molto più specializzata, tale da non poter essere gestita soltanto da una persona.

Il Visual Modeling prevede una continua interazione tra i vari membri del team di lavoro. Analisti e disegnatori, per esempio, necessitano di un continuo scambio di vedute per permettere che i programmatori possano basarsi su una solida informazione. A loro volta i programmatori devono interagire con gli analisti ed i disegnatori per condividere le loro intuizioni, modificare i disegni, e consolidare il codice. Il vantaggio di tale metodo di lavoro è che la conoscenza generale del team cresce notevolmente, ognuno è in grado di capire a fondo il sistema, e il risultato finale non può che essere un sistema più solido.

Un progetto per lo sviluppo di un sistema non può basarsi sull'improvvisazione ma deve seguire un iter tecnico-logico che consenta di arrivare in tempi brevi alla

soluzione migliore.

Il RAD rappresenta proprio una sorta di tracciato del progetto che viene incontro a tale esigenza. Il RAD (Rapid Application Development), ovvero lo sviluppo rapido di applicazioni consiste, fondamentalmente, di cinque sezioni:

Raccolta delle informazioni: In questa fase, gli analisti studiano i processi di business del cliente servendosi di colloqui specifici con il cliente stesso e chiedendogli di analizzare a fondo i processi rilevanti, passo per passo. Di solito, al termine di tale attività vengono abbozzati uno o più Activity Diagrams.

Analisi: L'analista intervista il cliente con lo scopo di capire bene le entità principali del problema del cliente. Alla conversazione tra l'analista ed il cliente prende parte anche un altro membro del team, il quale prende accuratamente nota di tutto. In particolare, si cerca di porre particolare attenzione ai termini chiave del problema che vengono appuntati come possibili Classi. Al termine di tale analisi, alcuni termini dell'analisi diverranno attribuiti mentre determinate azioni, che sono ritenute di rilevante importanza, saranno etichettate come metodi delle classi. Si produce un Class Diagram ad alto livello ed una buona serie di appunti. È questo il momento in cui vengono prodotti gli Use Case Diagrams. In una sessione JAD (Joint Application Development) con i potenziali utenti, il team di sviluppo lavora con gli utenti per definire gli actors³³ che interagiscono con ogni singolo use case sia come beneficiari che come iniziatori dello stesso use case. I passi principali dell'analisi di un progetto sono:

- Comprensione dell'utilizzo del sistema
- Creare gli Use Cases
- Analizzare i cambi di stato negli oggetti

³³ Con il termine 'actor' può intendere sia una persona che un sistema

- Analizzare l'integrazione del sistema da sviluppare con gli altri sistemi preesistenti

Disegno: A questo punto, il team è pronto per andare a fondo sui risultati ottenuti dalla raccolta delle informazioni e approfondire, dunque, la conoscenza del problema. Alcune delle azioni di Analisi iniziano, in realtà, proprio durante la fase di raccolta delle informazioni, quando viene abbozzato uno schizzo del Class Diagram. Durante la fase di disegno, il team lavora in base ai risultati che sono stati forniti dall'analisi al fine di disegnare la soluzione. Il Disegno e l'Analisi necessitano, a questo punto, di essere aggiornati vicendevolmente fin quando non si reputa che la fase di disegno sia completata. Le fasi del disegno sono:

- Sviluppare e rifinire gli Object Diagrams
- Sviluppare i Component Diagrams
- Pianificare il Deployment del sistema
- Disegnare e creare un prototipo dell'interfaccia utente
- Costruire i test
- Iniziare la documentazione

Sviluppo: Se le precedenti fasi di analisi e disegno sono state eseguite con accuratezza e scrupolo, la fase dello sviluppo non potrà che scorrere velocemente e senza grossi intoppi. Con i Class Diagrams, gli Object Diagrams, gli Activity Diagrams e i Component Diagrams a disposizione, i programmatori implementano il codice per il sistema. Ogni nuova implementazione del codice sarà, necessariamente, interfacciata con una opportuna fase di test. Si può, praticamente, dire che le due fasi camminano di pari passo fin quando il codice supera tutti

i livelli di test. Vengono prodotti in questa fase i risultati del Test. Viene completata la documentazione.

Deployment: Quando la fase di sviluppo è completata, il sistema viene installato sull'hardware appropriato ed integrato con i sistemi esistenti. Questa fase prende il nome di Deployment. Dopo aver installato il software sulle macchine appropriate, il team di sviluppo testa il sistema.

2.3.2 Visual Modelling

Il linguaggio UML contiene svariati elementi grafici che vengono messi insieme durante la creazione dei diagrammi. Poiché l'UML è un formalismo, esso utilizza delle regole per combinare i componenti del linguaggio di programmazione per creare i diagrammi. L'obiettivo dei diagrammi è quello di costruire molteplici viste di un sistema, tutte correlate tra di loro. L'insieme di tali viste costituirà quello che abbiamo definito Visual Modeling. Passiamo ora in rassegna, brevemente, tutti i diagrammi UML, prima di analizzarli più dettagliatamente in seguito.

Il linguaggio UML consiste di nove diagrammi di base, ma si tenga presente che è assolutamente possibile costruire e aggiungere dei diagrammi differenti dagli standard (che vengono definiti ibridi).

Class Diagram: I Class Diagrams forniscono le rappresentazioni utilizzate dagli sviluppatori delle classi.

Object Diagram: Gli Object Diagram forniscono una rappresentazione degli oggetti, e vengono anch'essi usati dagli sviluppatori.

Use Case Diagram: Uno Use Case (caso d'uso) è una descrizione di un comportamento particolare di un sistema dal punto di vista dell'utente. Per gli sviluppatori, gli use case diagram rappresentano uno strumento notevole: infatti

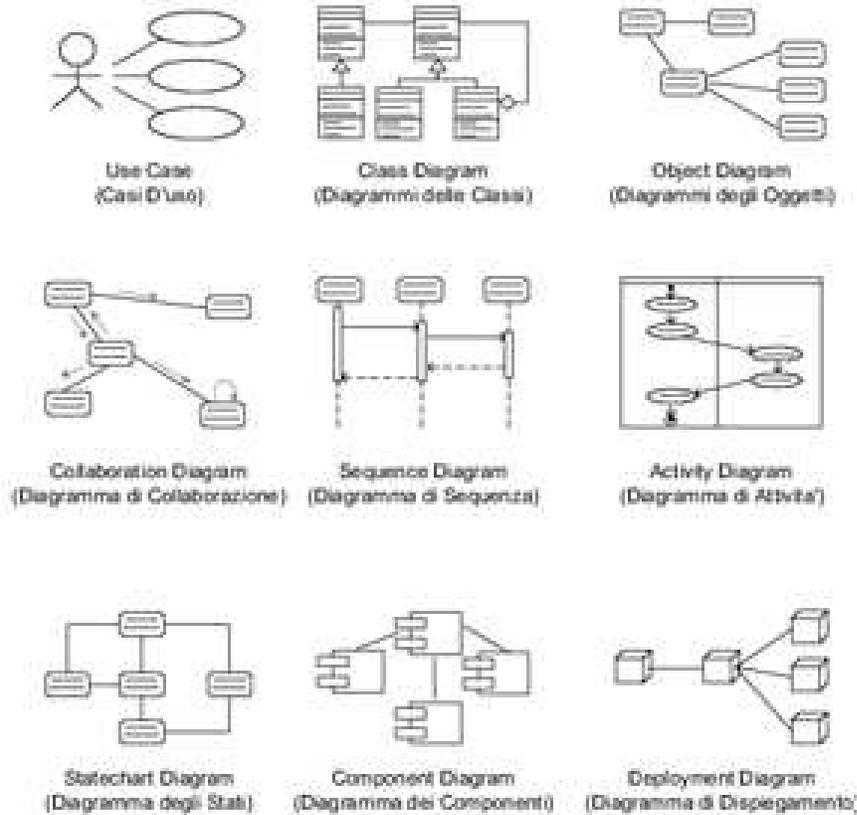


Fig. 2.7. Diagrammi UML

tramite tali diagrammi, essi possono agevolmente ottenere una idea chiara dei requisiti del sistema dal punto di vista dell'utente e quindi scrivere il codice senza timore di non aver recepito bene lo scopo finale. Nella rappresentazione grafica, viene utilizzato un simbolo particolare per l'actor³⁴.

State Diagram: Ad un determinato istante, durante il funzionamento del sistema, un oggetto si trova in un particolare stato. Gli State Diagrams rappresentano tali stati, ed i loro cambiamenti nel tempo. Ogni state diagram inizia

³⁴ L'actor' è l'entità che interagisce con uno use case facendo partire la sequenza di azioni descritte dallo use case stesso e, eventualmente, ricevendo delle precise risposte dal sistema.. Puo' essere una persona o anche un altro sistema.

con un simbolo che identifica lo stato iniziale (Start State) e termina con un altro simbolo che rappresenta lo stato finale (End State). Per esempio, ogni persona può essere identificato dai seguenti stati: neonato, infante, bambino, adolescente, adulto, anziano.

Sequence Diagram: I class diagrams e gli object diagrams rappresentano informazione statica. In un sistema funzionante, tuttavia, gli oggetti interagiscono l'uno con l'altro, e queste interazioni avvengono in relazione al trascorrere del tempo. Il sequence diagram mostra le dinamiche, basate sul tempo, delle varie interazioni tra gli oggetti.

Activity Diagram: Le attività che si riscontrano all'interno di use case o all'interno del comportamento di un oggetto accadono, tipicamente, in una sequenza ben definita. Tale sequenza viene rappresentata con gli activity diagrams.

Collaboration Diagram: Gli elementi di un sistema lavorano insieme per realizzare e soddisfare le necessità del sistema. Un linguaggio di modellazione deve avere un modo per rappresentare tale cooperazione. Il Collaboration Diagram nasce proprio per questa ragione.

Component Diagram: Oggi, nell'ingegneria del software viene sempre più utilizzato il modello di organizzazione secondo il quale ognuno nel team di lavoro si occupa di lavorare su un componente differente. Il component diagram descrive questa importante caratteristica.

Deployment Diagram: Il Deployment Diagram mostra l'architettura dal punto di vista fisico e logistico di un sistema. Tale diagramma può descrivere i computer e i vari dispositivi presenti, mostrare le varie connessioni che intercorrono tra di essi e, ancora, il software che è installato su ogni macchina.

Naturalmente non è sempre necessario sviluppare tutti i diagrammi possibili, ma bisognerà operare una scelta in base agli aspetti critici della simulazione.

2.3.3 Tool di Sviluppo

Come ogni linguaggio che si rispetti anche l'UML necessita di tools appropriati che ne agevolino l'utilizzo. Ci sono tanti tools che trattano l'UML. I tools commerciali più diffusi sono Together e RationalRose, mentre nel mondo *open source* esiste il progetto Argo UML dal quale deriva Poseidon for UML CE.

Together

Together, sebbene sia un ottimo prodotto, può correre il rischio di generare delle perplessità. Il motivo risiede proprio e paradossalmente in quella che dovrebbe esserne una peculiarità: la fusione del processo di disegno con quello di implementazione. Questa possibilità finisce per essere una tentazione troppo forte, che porta ad abbandonare prematuramente la fase di disegno per tuffarsi in quella di codifica. Chiaramente si tratta di uno strumento e quindi, se viene utilizzato in modo errato non si può attribuire la colpa all'applicativo.

In ogni modo si tratta di un tool di comprovato successo, particolarmente apprezzato per la relativa cura conferita all'aspetto grafico e ai dettagli. Anche la funzione di reverse engineering (o meglio di sincronizzazione in tempo reale tra disegno e codifica) risulta ben congegnata e Together rappresenterebbe effettivamente uno strumento imbattibile. Il passaggio dal modello di disegno alla relativa traduzione in codice non è un problema del processo di sviluppo del software; anzi il tutto è quasi immediato specie con linguaggi di programmazione come Java. La vera fase critica è invece passare dalla vista dei casi d'uso a quella di disegno. Un altro piccolo problema è che il tool è molto Java oriented, e ciò lo porta, talune volte,

lontano dalle specifiche formali dell'UML. Attualmente la TogetherSoft distribuisce un'ottima evoluzione del prodotto denominata Together Control Center.

Rational Rose

Un discorso a parte va fatto per il software della Rational: è, o comunque è stato, il punto di riferimento degli altri, realizzati a sua immagine e somiglianza. Si tratta indubbiamente di un altro livello, anche dal punto di vista economico (tipicamente, costa un'ordine di grandezza in più rispetto agli altri). La relativa esosità, non sempre giustificata, forse ne costituisce la seccatura principale. Probabilmente per sistemi di una certa dimensione esistono poche alternative, ma per una simulazione economica è sconsigliabile. La cosa meno comprensibile è che, nonostante Rational Rose sia fornito dall'azienda che ha prodotto le prime versioni dell'UML, non sembra aderire sempre alle relative specifiche. Attualmente è disponibile l'evoluzione denominata XDE (eXtended Development Environment, ambiente di sviluppo esteso), nella quale si è cercato di muoversi verso le fasi più di dettaglio del processo di sviluppo del software.

Argo UML

Un altro tool che merita particolare menzione è Argo UML ³⁵, frutto di un progetto realizzato presso l'Università della California (Computer Science University of California, Irvine). Si tratta veramente di un tool accattivante e ben studiato, soprattutto dal punto di vista dell'interfaccia utente. Propone soluzioni decisamente innovative e completamente diverse rispetto all'approccio classico su cui sono basati gli altri software. Altro grosso vantaggio è che si tratta di un sistema open source e quindi i sorgenti sono di pubblico dominio. Eventualmente è possibile personalizzarlo sulle proprie esigenze, modificandone direttamente il codice. Come

³⁵ <http://www.ArgoUML.org>

Together consente di passare comodamente dal disegno al codice e viceversa, con il rischio di indurre l'utente a trascurare la parte progettuale. Qualche pecca è imputabile alla non sempre perfetta aderenza alla direttive standard dell'UML e alla latenza nell'incorporamento dei cambiamenti di specifiche dovuti alle varie versioni dell'UML.

Trattandosi di un prodotto open source, non esiste un'azienda che ne faccia un prodotto di punta e quindi non viene effettuata alcuna operazione di marketing, non vi sono dipendenti (stipendiati) per curare il supporto della clientela, lo sviluppo di nuove funzionalità e il miglioramento. Ciò, d'altro canto, offre altri vantaggi, relativi soprattutto al fatto che il prodotto non è vincolato alle leggi di mercato. Per esempio vengono realizzate funzionalità relative ad aree che dal punto di vista del mercato risultano di minore importanza per via del numero di clienti che potrebbero esservi interessati, il ciclo di vita del prodotto è basato su criteri canonici della progettazione, è possibile avvalersi della collaborazione di personale veramente esperto dislocato nelle più svariate parti del mondo.

Poseidon for UML

A integrazione e modifica di quanto appena riportato, va considerato però che recentemente la software house Gentleware distribuisce una versione di ArgoUML, battezzata Poseidon. Poseidon è disponibile sia nella versione professional a pagamento che nella versione Community Edition gratuita, ma con alcune limitazioni, come la possibilità di stampare i grafici e importare archivi JAR esterni. Per analizzare JVE secondo il formalismo UML è stato utilizzato Poseidon for UML CE.

2.3.4 Vista UML di JVEFrame

Il progetto JVE è stato sviluppato in JavaSwarm senza l'ausilio del paradigma UML quindi non sono ancora stati creati molti dei grafici. Attraverso Poseidon for UML è comunque possibile avere una vista UML delle classi Java di JVE e delle loro relazioni.

2.4 Altri strumenti per la simulazione: *StarLogo*, *Ascape*, *RePast*, *Extend*

2.4.1 *Starlogo*

StarLogo is a programmable modeling environment for exploring the workings of decentralized systems – systems that are organized without an organizer, coordinated without a coordinator.

StarLogo³⁶ è un ambiente di sviluppo per analizzare il comportamento di sistemi distribuiti, come stormi di uccelli, ingorghi di automobili o colonie di formiche. Starlogo è stato costruito appositamente per gli studenti. Lo scopo è cercare di far emergere i percorsi e le regole di comportamento in un sistema distribuito senza un controllo centralizzato.

StarLogo è un estensione del linguaggio di programmazione Logo. Con il Logo tradizionale è possibile creare animazioni, comandando i movimenti di una tartaruga virtuale sullo schermo. StarLogo completa questa idea, consentendo il controllo di molte tartarughe che si muovono in parallelo. In aggiunta StarLogo ha reso il mondo sul quale si muovono le tartarughe attivo. E' possibile programmare programmare le conformazione dell'ambiente sul quale le tartarughe si muovono. Le tartarughe

³⁶ Per maggiori informazioni sul progetto StarLogo cfr.<http://education.mit.edu/starlogo> (2002)

e l'ambiente possono interagire tra loro. Per esempio, è possibile programmare le tartarughe per muoversi a velocità diverse a seconda del tipo di terreno calpestato.

StarLogo contiene tre tipi principali di 'characters'.

Turtles: Gli abitanti dei mondi StarLogo. E' possibile utilizzare le 'Turtles' per rappresentare praticamente qualsiasi tipo di oggetto: una macchina nel traffico, una formica, una molecola di gas o un'unità produttiva della Virtual Enterprise. Ogni tartaruga ha una posizione, una testa e un colore, ma è possibile aggiungere ulteriori proprietà utili al modello. Le tartarughe si muovono in parallelo.

Patches: Le 'Patches' sono parti del mondo sul quale si muovono le tartarughe. Il mondo è diviso in una griglia, e ogni quadrato corrisponde ad una diversa 'Patch'. Queste parti del mondo sono attive, nel senso che possono interagire con le tartarughe e eseguire i comandi di StarLogo.

Observer: L'"Observer", come in Swarm, osserva quello che accade nel mondo dall'alto. L'"Observer" può anche creare nuove tartarughe e interagire con la simulazione.

L'interfaccia di StarLogo utilizza alcune finestre grafiche:

StarLogo window: Questa finestra è dove si muovono e disegnano le tartarughe. La zona bianca contiene le interfacce con l'utente. In questa zona sono presenti bottoni, sliders, e display, che permettono di interagire con la simulazione.

Control Center window: L'area 'Control Center' è la parte dello schermo dedicata al prompt dei comandi StarLogo. La 'Procedures Area' è dove si possono inserire le routine di istruzioni. Cliccando sulle tartarughe e su 'Observer' è possibile specificare a chi sono riferite le procedure.

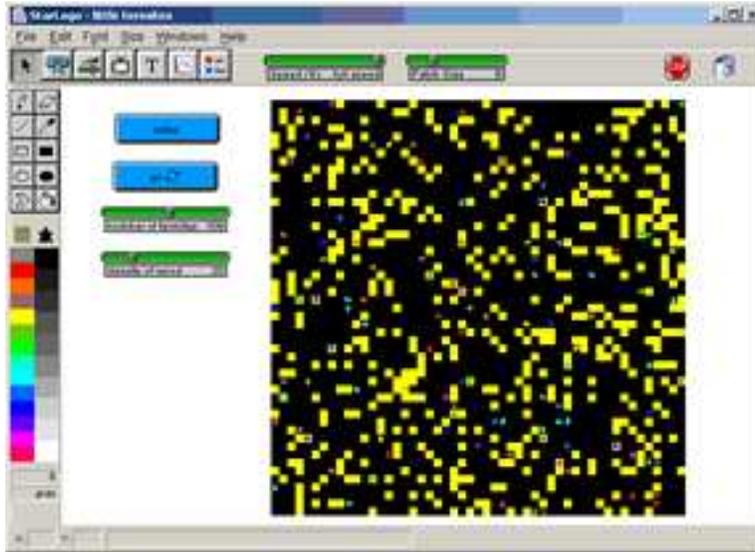


Fig. 2.8. StarLogo Window

Altre Finestre: La barra dei menu in alto sul 'Control Center' consente di accedere ad alcune altre finestre.

Information Window: si può immaginare come un tutorial contenente note, commenti e spiegazioni sull'applicazione.

Output Window: è utilizzata per visualizzare e salvare i dati generati dell'applicazione. Il comando *print* stampa a video i dati.

Plot Windows: la finestra nella quale si possono inserire grafici dinamici riguardanti la simulazione.

Turtle Monitors: possono essere paragonati alle sonde di Swarm. Se si clicca su di una tartaruga apparirà un 'Turtle Monitor'. Nel monitor verranno visualizzate le variabili di quella tartaruga, e sarà possibile per l'utente modificarle. I valori sono aggiornati in tempo reale mentre si esegue la simulazione.

Per utilizzare *StarLogo* non è necessario conoscere alcun linguaggio di programmazione è consente un rapido sviluppo di modelli semplici, con un buon impatto grafico.

StarLogo è stato utilizzato dal dott. Michele Sonnessa per creare una versione semplificata di *Virtual Enterprise* che mostra, in modo grafico, il funzionamento del modello. Questo progetto viene descritto in dettaglio nella sezione 4.2.2.

2.4.2 *Ascape*

Ascape is a software framework for developing and analyzing agent-based models. In *Ascape*, agent objects exist within scapes ; collections of agents such as arrays and lattices. These scapes are themselves agents, so that typical *Ascape* models are made up of collections of collections of agents. Scapes provide a context for agent interaction and sets of rules that govern agent behavior. *Ascape* manages graphical views and collection of statistics for scapes and provides mechanisms for controlling and altering parameters for scape models³⁷.

Ascape è un tool di sviluppo per simulazioni agent based. Come in *StarLogo* gli elementi principali sono gli agenti e l'ambiente, che in *Ascape* viene definito 'Scape'. L'ambiente possiede delle proprietà simili a quelle degli agenti, ed è lo 'Scape' ad occuparsi di tutta la gestione grafica del modello.

2.4.3 *RePast*

Our goal with *RePast* is to move beyond the representation of agents as discrete, self-contained entities in favor of a view of social actors

³⁷ Cit. <http://www.brook.edu/dybdocroot/es/dynamics/models/ascape> (2002)

as permeable, interleaved and mutually defining, with cascading and recombinant motives³⁸.

RePast³⁹ è un progetto molto simile a Swarm. Si tratta di una biblioteca di oggetti Java. Il suo principale vantaggio è di essere stato scritto direttamente con primitive Java e non, come nel caso di Swarm, in Objective-C.

RePast nasce alla *University of Chicago's Social Science Research* ed è un framework per creare simulazioni Agent Based usando il linguaggio Java⁴⁰. Contiene una biblioteca di classi per creare, eseguire, disegnare e gestire i dati di una simulazione. In aggiunta RePast può prendere snapshot (immagini dallo schermo) delle simulazione in esecuzione, e creare filmati in formato quick time. RePast è definibile ‘Swarm Like’ perché ha preso in prestito molti dei meccanismi di funzionamento di Swarm. In aggiunta RePast include alcune caratteristiche aggiuntive, come la possibilità di modificare i parametri della simulazione in run-time attraverso l’interfaccia grafica, proprio come succede in Ascape.

RePast concepisce la simulazione come una *state-machine* costituita dagli stati di tutti i suoi componenti. Questi componenti possono essere divisi in due gruppi, infrastrutture e e rappresentazioni.

Infrastrutture: le infrastrutture rappresentano i vari meccanismi che vengono attivati nella simulazione, come mostrare grafici, raccogliere dati e così via. Lo *stato* delle infrastrutture è quindi lo stato dei grafici, dell’oggetto che raccoglie i dati, e tutti gli aspetti più ‘tecnici’.

Rappresentazioni : le rappresentazioni sono ciò che viene costruito dal ricercatore, cioè il modello stesso della simulazione. Lo *stato* delle rappresentazioni è lo

³⁸ Cit. <http://repast.sourceforge.net> (2002)

³⁹ Il nome RePast è l’acronimo di REcursive Porous Agent Simulation Toolkit. Per maggiori informazioni si veda la home page del progetto <http://repast.sourceforge.net/>

⁴⁰ Per poter utilizzare RePast è richiesto JDK 1.3 o superiore

stato di ciò che viene modellizzato, come il valore dei parametri degli agenti, il valore delle variabili dell'ambiente nel quale gli agenti si muovono, così come lo stato di qualunque altro oggetto modellato dal ricercatore.

La storia della simulazione, dal punto di vista software, è la storia di entrambi i componenti. Dal punto di vista del modello simulato, comprende solo gli stati delle rappresentazioni.

In *RePast*, come in *Swarm*, ogni cambiamento di stato degli oggetti appartenenti alla rappresentazione avviene attraverso uno *Schedule*. In breve *RePast* consente di costruire la simulazione come una *state-machine* nella quale i cambiamenti di stato vengono programmati nello *Schedule*. Questo meccanismo consente massima flessibilità sia al creatore della simulazione, sia a chi volesse eventualmente modificarla. Per il resto *RePast* segue lo stesso funzionamento di *Swarm*.

L'obiettivo degli sviluppatori è simulare sistemi complessi, organizzazioni e istituzioni sfruttando la potenza del calcolo ricorsivo. La versione corrente di *RePast* è 1.4. L'obiettivo finale del progetto, quando raggiungerà la maturità, è fornire uno strumento che consenta facilmente di creare un modello, e alterarne le proprietà secondo il principio 'What if?'.⁴¹

2.4.4 *Extend*

*Extend*⁴¹ è una famiglia di prodotti per la simulazione di processi aziendali che permette di creare modelli semplici per simulare le più svariate problematiche dell'impresa come: individuazione dei migliori investimenti, ottimizzazione delle risorse a disposizione, riduzione dei costi di produzione.

Si tratta di un ambiente dinamico e multi piattaforma basato su icone, è dotato di un sistema di sviluppo incrementale che permette la definizione e la simulazione di

⁴¹ <http://www.imaginethatinc.com> (2002)

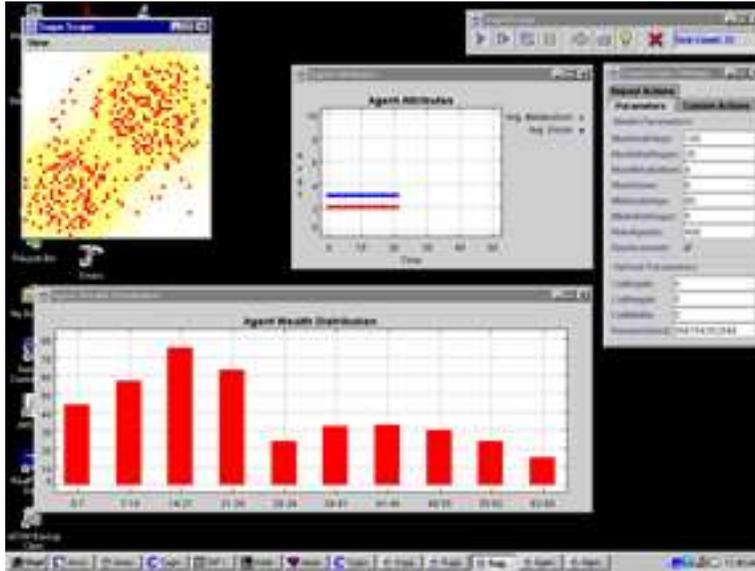


Fig. 2.9. Esempio di simulazione con RePast

sistemi ad eventi discreti, di sistemi continui e di sistemi misti. Le sue principali caratteristiche sono: gerarchizzazione, simulazione animata, rappresentazione Object-Oriented, interfaccia grafica multiplatforma, Activity-Based Costing, Export and Report.

Questo tool fornisce alcune librerie contenenti i blocchi necessari per la costruzione di tutti i tipi di modelli; inoltre è possibile programmare nuovi blocchi in base alla tipologia dell'impresa e, quindi, alle singole esigenze di rappresentazione del modello.

BPR: supporta la reingegnerizzazione dei processi business e consente di modellare ipotesi alternative dal punto di vista dell'organizzazione e di effettuare analisi what if? in ambiente virtuale (false safe ed economico) prima di metterle in opera.

Manufacturing: è utile per individuare colli di bottiglia, ridimensionare buffers e magazzini, valutare lead times e temporizzazioni. Consente di ottimizzare e

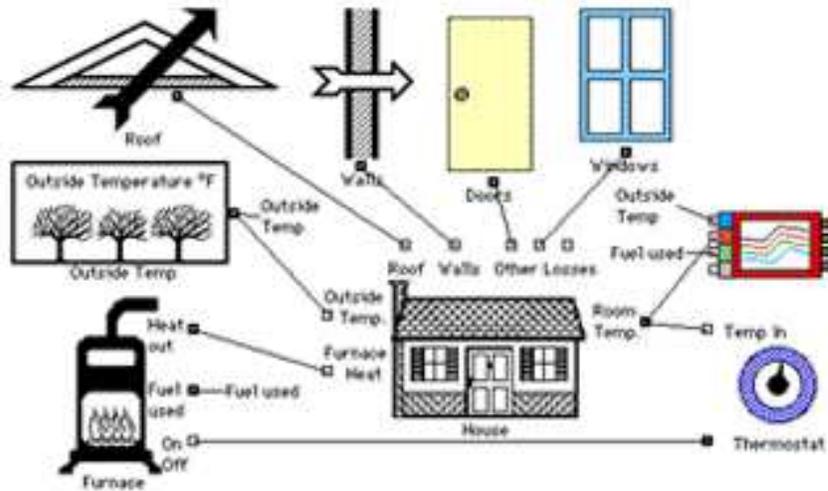


Fig. 2.10. Esempio di simulazione di processo con Extend

verificare l'apporto di cambiamenti significativi nei processi operativi di tipo industriale e commerciale, analizzando le alternative e documentando le procedure, valutando e giustificando anzitempo i costi.

BIBLIOGRAFIA

- Bissey, M. (2001). *Una piccola introduzione a swarm: Objectivec e java*.
- Eckel, B. (2000). *Thinking in java* (Second ed.). Prentice-Hall.
- Flanagan, D. (2000). *Java in a nutshell* (3rd ed.). O'Reilly.
- <http://education.mit.edu/starlogo>. (2002). *Starlogo project homepage*.
- <http://java.sun.com/j2se/1.4/docs/tooldocs/win32/javadoc.html>. (2002). *Javadoc - the java api documentation generator*.
- <http://java.sun.com/j2se/javadoc>. (2002). *Javadoc tool home page*.
- <http://java.sun.com/j2se/javadoc/faq.html>. (2002). *Javadoc faq*.
- <http://java.sun.com/j2se/javadoc/writingdoccomments/index.html>. (2002). *How to write doc comments for javadoc*.
- <http://ptolemy.eecs.berkeley.edu>. (2002). *Ptolemy project homepage*.
- <http://repast.sourceforge.net>. (2002). *Repast project homepage*.
- <http://santafe.edu>. (2002). *Santa fe institute*.
- <http://www.brook.edu/dybdocroot/es/dynamics/models/ascape>. (2002). *Ascape project homepage*.
- <http://www.cygwin.com>. (2002). *Cygwin project homepage*.

- <http://www.gnu.org/copyleft/gpl.html>. (2002). *General public license*.
- <http://www.gnu.org/philosophy/freesw.html>. (2002). *Free software*.
- <http://www.html.it>. (2002). *Html.it website*.
- <http://www.imaginethatinc.com>. (2002). *Extend by imagine that inc. homepage*.
- <http://www.lorenzobettini.it/articoli/javatools/javatools.html>. (2002). *Java archive tool*.
- <http://www.opensource.org/osd.html>. (2002). *Open source*.
- <http://www.redhat.com>. (2002). *Red hat homepage*.
- <http://www.swarm.org>. (2002). *Swarm project homepage*.
- <http://www.xprogramming.com>. (2002). *Xp homepage*.
- Johnson, P., & Lancaster, A. (2000a). *Documentation set for swarm 2.1.1*.
- Johnson, P., & Lancaster, A. (2000b). *Swarm user guide*.
- Luna, F., & Stefansson, B. (2000).
economic simulation in swarm: Agent-based modelling and object oriented programming. KluwerAcademic Publishers Boston, Dordrecht, London.
- Manzini, M. (2001). Javadoc documentare il codice java. *MokaByte*,
<http://www.mokabyte.it/2001/03/javadoc.htm>, no.50.
- Minar, N., Burkhart, R., Langton, C., & Askenazi, M. (1996). *The swarm simulation system: A toolkit for building multi agent simulations*.
- Naughton, P. (1996). *Il manuale java*. MacGraw-Hill, Italia.
- ObjectManagementGroup. (1999). *Omg unified modeling language specification*.

Terna, P. (1998). Simulation tools for social scientists: Building agent based models with swarm. *Journal of Artificial Societies and Social Simulation*, <http://www.soc.surrey.ac.uk/JASSS/1/2/4.html>, vol. 1 (no.2).

BIBLIOGRAFIA

Capitolo 3

L'AZIENDA COME SISTEMA COMPLESSO

In questo capitolo si intendono introdurre alcune teorie dell'impresa, partendo dal lavoro di Coase (1995), che definisce l'impresa attraverso i costi di transazione, sino all'ordine spontaneo di Hayek (1937) e la scuola austriaca in economia. Si analizzeranno consonanze e disaccordi tra la scuola austriaca e le moderne teorie della complessità. Particolare attenzione verrà riservata alla generazione di conoscenza tacita ed esplicita all'interno dell'impresa, in considerazione delle possibilità di simulazione di questi fenomeni offerte da JVE.

3.1 *La teoria dell'impresa di Coase*

Coase si è per primo occupato di definire in modo organico una 'Teoria dell'impresa'.

Come afferma Coase (1995):

(...) Nostro compito è scoprire perché nasce l'impresa in un'economia di scambi specializzata(...) La principale spiegazione del perché è vantaggioso creare un'impresa sembra essere il fatto che l'utilizzo del meccanismo dei prezzi di mercato comporta un costo. Il costo più evidente dell'organizzare la produzione tramite il meccanismo dei prezzi è quello di scoprire quali sono i prezzi che interessano (...) Il funzionamento del mercato comporta un certo costo e, formando un'organizzazione e permettendo a un'autorità (un imprenditore) di dirigere le risorse, possono essere risparmiati taluni costi di contrattazione (...) Un'impresa consiste dunque in un sistema di rapporti che vengono in essere quando la direzione dell'uso delle risorse dipende da un imprenditore (...) L'approccio finora sviluppato sembra offrire un vantaggio consistente nel fatto che è possibile attribuire un significato scientifico a ciò che si intende quando si afferma che un'impresa cresce o si riduce di dimensioni. Un'impresa si espande quando altre ulteriori transazioni (che possono essere scambi di mercato coordinati tramite il meccanismo dei prezzi) vengono organizzate dall'imprenditore e diventa più piccola quando quest'ultimo abbandona l'organizzazione di queste transazioni (...) Un'impresa tenderà ad espandersi fino a che i costi di organizzare una transazione in più all'interno dell'impresa diventano uguali ai costi di effettuare la stessa transazione mediante uno scambio di mercato, o ai costi di organizzare un'impresa diversa (...)

Il concetto principale che viene introdotto è quello dei 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 costi in quanto generano 'frizioni' decisionali. L'esistenza di questi costi di contrattazione spinge gli agenti economici ai comportamenti 'opportunistic' tipici delle economie di mercato. Secondo Coase è proprio l'opportunismo esistente nel mercato a spingere gli agenti economici a ricreare una struttura all'interno della quale cooperare e condividere le conoscenze. Questa struttura viene definita da Coase 'impresa'.

Il contributo di Coase (1995) è rilevante nello studio del modello Virtual Enterprise perché è il primo lavoro organico e completo con il quale si studia la nascita dell'impresa. Con il modello di simulazione JVE si intende indagare in due direzioni: da un lato cercare coerenza tra le previsioni della simulazione e la realtà aziendale, come nel caso dell'applicazione di JVE alla BasicNet; d'altro canto si persegue l'obiettivo teorico di cercare di giustificare la nascita dell'impresa. E' proprio in riguardo a questo secondo obiettivo che risulta rilevante l'analisi di Coase sulla nascita dell'impresa, come punto di partenza e spunto, per arrivare ad ottenere un modello di simulazione che spieghi ciò che Coase (1995) descrive con un modello letterario-descrittivo.

3.2 *Come l'azienda crea conoscenza tacita ed esplicita*

Il termine conoscenza è usato con due accezioni distinte. La conoscenza può essere l'insieme di nozioni tecniche e teoriche necessarie eseguire lavori più o meno complessi, ma è anche conoscenza l'insieme delle norme, in larga misura implicite, che regolano il comportamento umano (in questo caso si può parlare più precisamente di 'cultura in senso sociologico'). E' necessario fare una distinzione tra conoscenza interna ed esterna alle imprese. Per conoscenza interna si intende la conoscenza che viene sviluppata facendo ricorso esclusivamente a risorse interne dell'impresa,

mentre per conoscenza esterna si intende quella che proviene da istituzioni esterne (prima tra tutte l'università).

Per poter trasformare le conoscenze da esterne a interne all'impresa si svolge un processo di 'trasferimento' della conoscenza. Questo processo può essere più o meno complicato a seconda che si voglia trasferire conoscenza esplicita o tacita. Per conoscenza esplicita si intendono tutte quelle forme di sapere che è possibile codificare in forma orale o scritta. Queste forme di conoscenza sono, in genere, facilmente trasferibili. Un esempio potrebbe essere un libro di testo di economia, che contiene informazioni elaborate dall'autore e codificate in linguaggio descrittivo o matematico.

La conoscenza tacita risulta più difficile da definire. Per comprenderne il significato è necessario fare ricorso alla separazione tra diversi tipi di processi cognitivi proposta da una nuova branca interdisciplinare della scienza, le 'scienze cognitive'. Questa divisione distingue quattro categorie di processi cognitivi:

1. Processi espliciti di apprendimento della conoscenza.
2. Processi espliciti di rappresentazione della conoscenza.
3. Processi impliciti di apprendimento della conoscenza
4. Processi impliciti di rappresentazione della conoscenza.

Il processo cognitivo 3, definito apprendimento implicito, è quello che è in grado di generare la conoscenza tacita dell'individuo.

La conoscenza tacita viene definita da Pozzali and Viale (2002):

Queste forme di conoscenza sono acquisite grazie all'esperienza, che mette i soggetti nella condizione di riconoscere determinate similarità nell'andamento dei fenomeni naturali, ma non portano di per sé alla sistematizzazione delle intuizioni soggettive in una trattazione univoca. Inoltre,

spesso la traduzione in forma esplicita di queste forme di conoscenza, accumulate dai soggetti sulla base dell'esperienza, richiede notevoli sforzi in termini di tempo e di risorse cognitive.

Un esempio banale di acquisizione di conoscenza tacita è il processo con il quale si impara ad andare in bicicletta. Molto spesso le forme tacite di conoscenza, rappresentate in modo particolare da un sapere pratico acquisito in base all'esperienza, possano assumere un ruolo rilevante, almeno nelle fasi iniziali, nel processo di innovazione svolto all'interno dell'impresa.

La conoscenza tacita distingue ulteriormente al proprio interno diversi aspetti che coesistono:

Componente tecnica: e quella parte di conoscenza tacita che serve a svolgere attività manuali. Consiste nella capacità di compiere fisicamente una serie di predefinita di azioni al fine di svolgere un'attività complessa.

Componente cognitiva: si riferisce ai processi di apprendimento delle attività intellettuali. Per apprendere nuovi concetti è necessario attribuire un significato stabile ai dati studiati. A questo scopo si deve sviluppare un 'modello mentale' che formi nella mente dell'individuo delle categorie interpretative generali della realtà, all'interno delle quali classificare le informazioni apprese. Questi schemi e queste categorie, tuttavia, non sono pienamente compresi razionalmente, né vengono interamente tradotti mediante il linguaggio, per questo fanno parte della conoscenza tacita.

Il problema maggiore si presenta nel momento in cui si vuole trasferire la conoscenza tacita. La conoscenza tacita deve essere convertita in uno schema codificato, costruito facendo ricorso ai mezzi e alle risorse linguistiche, all'interno del quale componenti tacite e esplicite sono legate tra loro. Questo processo può essere visto

(Pozzali & Viale, 2002) come la creazione di un codice, ossia di un meccanismo, che consenta di tradurre gli schemi mentali in linguaggio descrittivo e matematico. Il trasferimento della componente tecnica avviene principalmente in forma non verbale, e richiede l'imitazione dei comportamenti dei soggetti esperti da parte dei principianti. Il trasferimento della componente cognitiva presenta invece problemi relativi alla diversità dei codici e modelli di ragionamento utilizzati all'interno delle differenti comunità di ricercatori. Il trasferimento può avvenire solo se ambedue i soggetti condividono (almeno in parte) una comune cultura di riferimento.

Lo schema Environment Rules Agents (ERA) descritto nella sezione 4.1.1 gestisce il trasferimento dell'informazione. Questo meccanismo può ricreare, nella simulazione, i processi di trasferimento interno della conoscenza. Il progetto JVE contiene al suo interno anche un meccanismo di circolazione delle informazioni (che viene definito gestione delle news, per maggiori dettagli si veda la sezione 4.1.4) che consente di simulare l'impatto della conoscenza sull'organizzazione. Uno dei possibili sviluppi di JVE potrebbe proprio essere implementare la simulazione del sistema informativo aziendale.

3.3 Hayek e la scuola austriaca

Per gli economisti neoclassici l'elemento saliente dell'economia di mercato è la sua capacità di creare in modo automatico una configurazione caratterizzata da un utilizzo ottimo delle risorse disponibili e da un consumo efficiente dei beni prodotti. Il verificarsi di malfunzioni non è escluso, ma relegato a fenomeno marginale, in ogni caso è possibile riallineare il mercato su situazioni di ottimo attraverso idonee politiche economiche.

Questa visione dell'economia nata da Marshall, Chamberlin e Robbins si può definire, Colombatto (2001), 'perfettista', descrivendo il mondo con una concezione meccanicista, che culmina nella nozione di equilibrio concorrenziale statico, e

considera un male ciò che separa il mercato dalla perfezione, del quale la politica economica è la cura . Questa visione estremamente positiva ha generato modelli complessi solo formalmente, ma intrinsecamente semplici, o forse meglio semplicistici, verso un'economia della perfezione . Gli economisti classici formalizzano i problemi in termini di funzione obiettivo, che dovrà essere massimizzata nel rispetto di un insieme di vincoli definiti nel modello, sfruttando i gradi di libertà a disposizione del decisore. Le teorie così formulate sono solo apparentemente eleganti e obiettive, nel tentativo di avvicinare la ricerca sociale alla perfezione metodologica delle scienze naturali, ma intrinsecamente irreali. Queste consentono all'economista di formulare ipotesi precise, applicabili con interventi normativi, così da porsi nella veste di decisore della politica economica. Viene invece frequentemente trascurata la scarsa aderenza alla realtà delle previsioni, e l'irrilevanza delle conclusioni , in sintesi la mancanza di realismo in economia.

Molti economisti hanno inteso il realismo come un obiettivo non primario nell'interpretazione della realtà. Si ritiene che l'economia sia in grado di fare previsioni verosimili sugli aspetti rilevanti della realtà anche senza comprenderne a fondo il funzionamento e i nessi causali.

Alla base della teoria economica hayekiana e, più in generale, della concezione delle scienze sociali, vi è una prospettiva che muove dalla considerazione dei limiti con cui gli individui devono fare i conti quando pensano e agiscono. Questi limiti (Colombatto, 2001) sono una conseguenza del fatto che essi operano in condizioni di non completa libertà, che i criteri, le norme e le leggi ai quali essi si riferiscono sono imperfetti, che la conoscenza di cui ogni singolo uomo può disporre è limitata, scarsa, fallibile e dunque revocabile.

Secondo Kant nella sua 'Critica della ragion pura':

Il complesso di tutti gli oggetti possibili è, per la nostra conoscenza, come una superficie piana, che ha il suo orizzonte apparente, quello, cioè,

che abbraccia tutto l'ambito di essi, ed è stato detto da noi il concetto razionale della totalità incondizionata. Raggiungerlo empiricamente è impossibile, e tutti i tentativi per determinarlo a priori secondo un certo principio sono stati vani. Intanto tutte le questioni della nostra ragion pura mirano a ciò che può essere fuori di questo orizzonte, o in ogni caso sulla linea del suo confine.

In Hayek il ruolo di coordinamento del mercato poggia sulla creazione (spontanea) di un sistema di regole astratte che si sostituisce alle specifiche e concrete relazioni degli scambi individuali. Prezzi, moneta e standards di qualità da un lato, interesse individuale, libertà di competere e diritti di proprietà dall'altro, rappresentano quei codici e norme che consentono alla conoscenza di diffondersi liberamente nella società. I diversi e dispersi piani individuali possono così confrontarsi e incontrarsi senza il bisogno di alcuna superiore mente ordinatrice.

Per Hayek lo scienziato sociale formula teorie nel senso di schemi di classificazione delle azioni individuali e individua archetipi di comportamento. Costruisce i modelli e classifica le azioni individuali sulla base del postulato che possiede una mente simile a quella degli individui il cui comportamento sta studiando. Per Hayek la mente è chiusa dentro 'una struttura tradizionale e impersonale di regole apprese', che evolve nel tempo, e la sua capacità di classificazione è fondata su archetipi che le vengono forniti dalla tradizione culturale del passato.

L'individualismo metodologico è una tesi fondamentale della Scuola austriaca. Questa teoria si oppone al collettivismo metodologico, cioè alla concezione che sostiene che ai concetti collettivi (Stato, partito, popolo, nazione, classe, elettorato, ecc.) 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 concetti, costruzioni artefatte dagli individui stessi. L'individualismo metodologico restituisce sostanza empirica, fatti, alle scienze

sociali. Solo gli individui pensano, ragionano e agiscono: è questo il nucleo teorico dell'individualismo metodologico di Menger, Mises e Hayek. Le azioni intenzionali degli individui comportano di necessità conseguenze inattese, inintenzionali.

Da qui scaturisce la persistente avversione degli Austriaci nei confronti del costruttivismo. Il costruttivismo sostiene che tutte le istituzioni sociali e tutti gli eventi storici sono esiti di piani intenzionali, risultati di progetti elaborati, voluti e realizzati. La scuola austriaca, soprattutto con Menger e Hayek e poi pure con Popper, ha inteso sottolineare, in una sempre più elaborata teoria evolutiva delle istituzioni sociali, che, sebbene tutte le istituzioni e tutti gli eventi sociali siano frutto dell'azione umana, poche di esse sono esiti di umani progetti deliberati e consapevoli.

Uno dei temi affrontati da Hayek è il ruolo della conoscenza nel processo di scoperta dell'equilibrio di mercato. In particolare, questo processo vede l'equilibrio come un meccanismo evolutivo in cui i partecipanti al mercato acquisiscono sempre maggiori informazioni, accurate e complete fino a sviluppare quindi una *'mutual knowledge'* della domanda e della offerta potenziale. La scuola austriaca introduce anche l'idea di *'asimmetria informativa'* considerando la *sheer ignorance*, che viene ignorata nei modelli neoclassici.

Nelle teorie hayekiane non si presume una situazione iniziale di equilibrio dei modelli, ma si pone in risalto il fatto che, da una situazione iniziale di non equilibrio, le forze tendano verso una situazione equilibrata. Da queste considerazioni emerge la convinzione che il modello dell'equilibrio generale Walrasiano, fallisca nell'offrire uno schema teorico generale che permetta di comprendere gli eventi e gli accadimenti nelle economie di mercato. Da questa osservazione si può desumere una critica alla microeconomia classica.

Secondo Kilpatrick (2001) la teoria della complessità ha notevoli parallelismi con quella dell'ordine spontaneo di Friedrich von Hayek.

Kilpatrick (2001) considerare la teoria dell'ordine spontaneo come l'anticipazione

della nozione di sistema complesso adattivo. Differenza fondamentale tra le teorie hayekiane e gli studi sulla complessità rimane lo scarso uso di metodi quantitativi e computazionali da parte degli austriaci.

Le teorie sulla complessità si allontanano sempre più dalla scuola austriaca quando si analizzano fenomeni di *path dependency* e *lock in* tecnologici. Secondo gli studiosi della complessità l'ordine spontaneo crea imperfezioni e distorsioni che dovrebbero essere corrette dalle istituzioni, mentre nella visione austriaca il mercato dovrebbe operare liberamente.

Anche l'impresa è stata rivalutata dalla scuola austriaca. Nella visione neoclassica dell'impresa vengono avanzate alcune ipotesi criticate dagli austriaci:

1. Viene esclusa l'incertezza e la presenza di durature asimmetrie informative.
2. L'accesso alle informazioni si presume gratuito.

Secondo gli austriaci, se venisse accettata la prima ipotesi, non sarebbe possibile la presenza di imprese migliori di altre, con imprenditori in grado di avvalersi di una funzione di produzione in modo più efficace e produrre così a costi inferiori rispetto ai concorrenti. Come conseguenza si assumerebbe una funzione di produzione fissa, che eliminerebbe anche i concetti di equilibrio di breve e di lungo periodo, che verrebbero a coincidere, poiché nessun operatore desidera o ha la necessità di alterare il proprio comportamento. Come estrema conseguenza di questo assunto neoclassico, si arriverebbe ad affermare che l'incertezza che caratterizza l'attività d'impresa potrebbe essere, in teoria, eliminabile poiché l'imprenditore avrebbe la possibilità di conoscere le distribuzioni di probabilità legate agli eventi aleatori che lo interessano.

Se l'accesso all'informazione fosse gratuito si dovrebbe supporre che l'acquisizione di informazioni da parte dell'imprenditore non abbia termine fino all'eliminazione completa del rischio. In realtà l'acquisizione di informazioni termina quando il costo

di acquisizione diventa superiore al beneficio che porterebbe una decisione meglio documentata e consapevole.

Hayek accoglie la visione dell'imprenditore proposta da Schumpeter. Una visione innovativa dell'imprenditore è invece proposta da Kirzner. Citando Colombatto (2001):

L'imprenditore di Kirzner è il protagonista dell'economia di mercato intesa come un sistema attraverso il quale si manifestano gli stimoli necessari per progredire. Costui non è più mero esecutore di uno schema preordinato - la funzione di produzione - ma diventa invece lo scopritore per eccellenza, cogliendo le opportunità che altri avevano trascurato e rendendo possibile la definizione di nuove produzioni; rinnovando prodotti, specificazioni funzionali o valori assunti dai parametri in funzioni già note.

L'impresa viene definita da Kirzner (1997) 'scoperta imprenditoriale'. L'imprenditore assume quindi il ruolo di 'scopritore', la scoperta è la giustificazione dell'utile d'impresa. L'"imprenditore scopritore" di Kirzner è quindi molto diverso dall'"imprenditore massimizzatore" di Lord Robbins.

Per gli Austriaci (da Schumpeter a Kirzner) 'imprenditorialità' significa allontanarsi dalla situazione di equilibrio, alla ricerca di profitti. Da questo discende dalla casualità della scoperta imprenditoriale Kirzneriana. Da questa considerazione Kirzner arriva ad affermare che chiunque può essere imprenditore, a condizione che sia in grado di allontanarsi da situazioni di equilibrio o sfruttare le imperfezioni del mercato. La concezione del mercato di Kirzner è simile a quella di Hayek. Il mercato imperfetto è caratterizzato da squilibri, l'imprenditore è motivato alla scoperta imprenditoriale dall'esistenza di informazioni alle quali altri agenti economici non possono accedere.

BIBLIOGRAFIA

- Coase, R. (1995). La natura dell'impresa. *Impresa, mercato e diritto*, pp. 199-259.
- Colombatto, E. (2001). Dall'impresa dei neoclassici all'imprenditore di kirzner. *Economia politica, XVIII*(2), pp.157-179.
- Hayek, F. A. von. (1937). Economics and knowledge. *Economica, new series, IV*(13), pp. 33-54.
- Kilpatrick, E. (2001). Complexity, spontaneous order and friedrich hayek: are spontaneous order and complexity essentially the same thing? *Complexity, VI*(4), pp.16-20.
- Kirzner, I. (1997). Entrepreneurial discovery and the competitive market process: An austrian approach. *Journal of Economic Literature, XXXV*(3), pp. 60-85.
- Pozzali, A., & Viale, R. (2002). Cognizione e conoscenza tacita nei processi innovativi. *Sistemi Intelligenti*(1).

BIBLIOGRAFIA

Capitolo 4

IL MODELLO JAVA VIRTUAL ENTERPRISE

In questo capitolo si intende descrivere l'essenza del modello di Java Virtual Enterprise (Terna, 2002), nato nel corso dell'anno 2000 all'interno del Dipartimento di Scienze economiche e finanziarie G.Prato dell'Università degli Studi di Torino. Per comprendere a fondo il funzionamento della Virtual Enterprise sarà necessario studiare la metafore che risiede alla base del modello, il mondo virtuale viene descritto attraverso la contrapposizione di due aspetti: 'come fare cosa?' e 'chi fa che cosa?'. Una volta compresa la metafora si studieranno gli strumenti informatici messi a disposizione per creare simulazioni realistiche. Particolare attenzione verrà riservata alla comprensione dei 'layer' e degli 'oggetti computazionali'.

In questo capitolo intendo descrivere l'essenza del modello di Java Virtual Enterprise (Terna, 2002). Il progetto è nato nel corso dell'anno 2000 all'interno del Dipartimento di Scienze economiche e finanziarie G.Prato dell'Università degli Studi di Torino. Lo scopo inizialmente era studiare l'impatto sulle aziende della transizione da old a new economy. Oggi lo studio è orientato alla simulazione di aziende reali, per meglio capirne il funzionamento, ottimizzarne i processi produttivi, e ricercare fenomeni emergenti previsti e imprevedibili. Questo tipo di strumento di simulazione è stato realizzato in JavaSwarm ad oggetti. Per la sua grande flessibilità e la generalità della metafora utilizzata, è idoneo a studiare possibili scenari futuri con metodologia 'what... if?'.

In Terna (2002c):

jVEFrame (Java Virtual Enterprise Frame, written in Swarm) is built to simulate the effects of deep modifications that may occur in the life of a firm - for example, the adoption of B2B and B2C strategies - and to investigate the role of knowledge into this kind of economic organizations.

Il modello di JVE, è stato utilizzato e adattato per simulare la VIR s.p.a., un'azienda tradizionale che produce valvole idrauliche e, come si descriverà ampiamente nel Capitolo 5, per simulare la BasicNet s.p.a.

4.1 *La metafora Virtual Enterprise*

Per comprendere a fondo il funzionamento della Virtual Enterprise è necessario studiare la metafore che risiede alla base della simulazione. Per poter simulare molti e diversi processi aziendali è necessari ricorrere ad una forte astrazione, e quindi costruire una metafora della realtà sulla quale sia possibile lavorare.

La metafora della Virtual Enterprise nasce dalla divisione della realtà aziendale in due elementi: 'chi fa che cosa' e 'come fare cosa'.

In Virtual Enterprise le produzioni aziendali sono una sequenza numerica, che viene definita ricetta.

Ogni numero della sequenza rappresenta un passo nella realizzazione del prodotto o, come nel caso di BasicNet, del processo organizzativo.

Il modello è creato dall'incontro di due formalismi: da un lato le operazioni da compiere, (siano esse produzioni o processi organizzativi) e dall'altro le unità che compiono le operazioni descritte (chi fa che cosa). In questo modo, è possibile ricreare una situazione reale, costruendo, per esempio, la BasicNet o ipotizzare una struttura astratta, per valutare la bontà di nuove configurazioni organizzative.

L'azienda virtuale così ottenuta può essere oggetto di indagine, esattamente come se si trattasse di un esperimento creato in laboratorio.

4.1.1 *Lo schema ERA*

Il progetto utilizza uno schema di relazioni fra l'agente, l'ambiente e i gestori di regole, che viene definito Struttura Environment Rules Agents. Questo schema è nato per gestire creatori di regole tipici dell'intelligenza artificiale.

Lo schema ERA, rappresentato graficamente in fig. 4.1, prevede la trasmissione delle informazioni nel modello attraverso quattro strati distinti, sia concettualmente, sia tecnicamente (esistono classi Java diverse per ogni strato).

I quattro strati vengono descritti in Terna (2002c):

1. Un primo strato rappresenta l'ambiente in cui gli elementi sono chiamati ad interagire. L'ambiente, o environment, è costituito dal Model Swarm nell'ambito del protocollo Swarm. Questo è il contesto all'interno del quale si definiscono gli agenti, se strutturano le liste, si individuano gli eventi nel tempo e si chiariscono le regole di interazione tra gli agenti grazie ai metodi (interpretabili come messaggi che gli agenti sono in grado di gestire, anche reagendo con azioni e informazioni) definiti all'interno degli oggetti creati dal Modello.

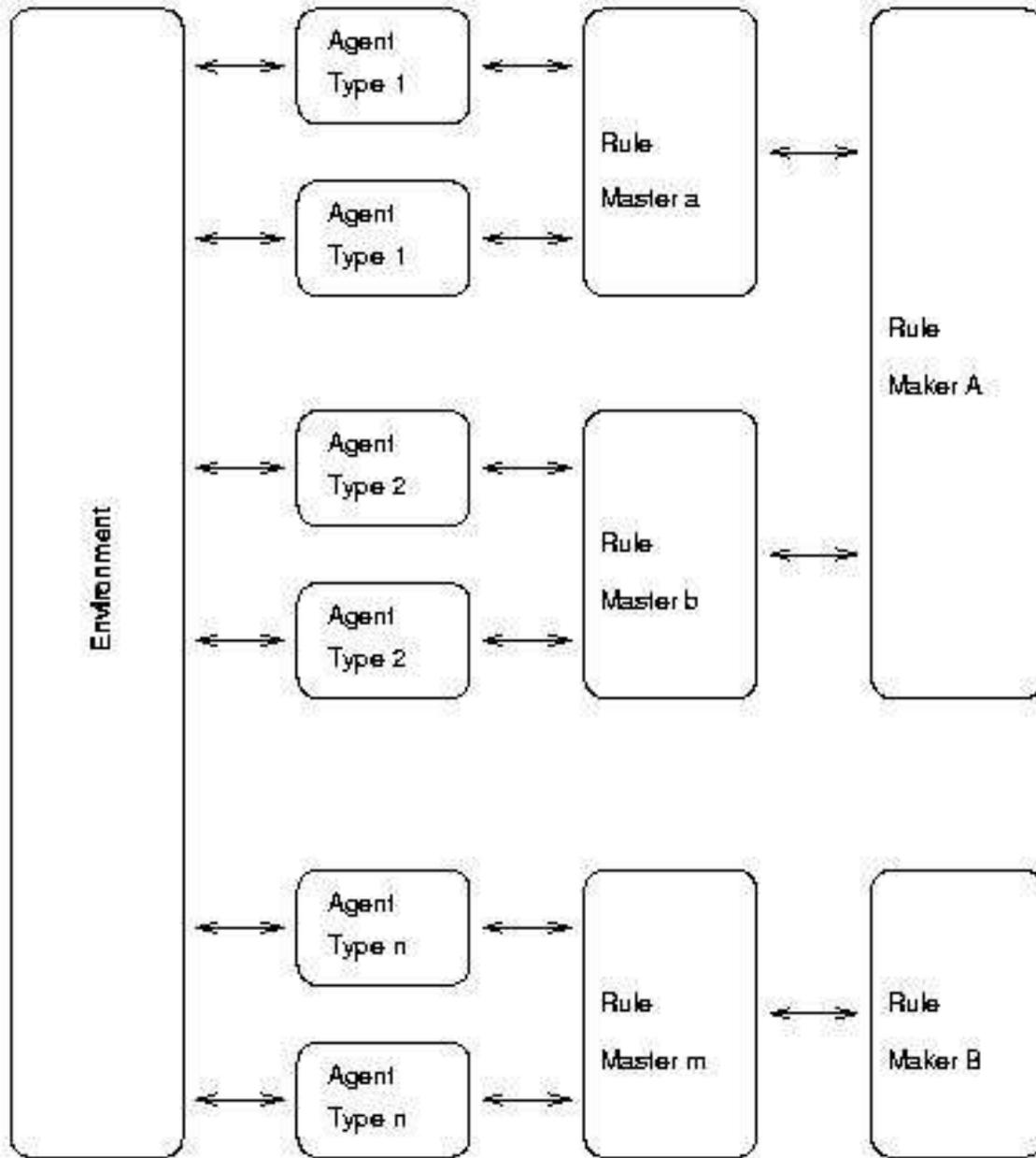


Fig. 4.1. Schema ERA

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

L'utilizzo di questo schema, che potrebbe apparire rigido, ha, in realtà, un duplice vantaggio: tenere ordinato il codice, e gestire separatamente i generatori di regole dagli agenti utilizzatori. Questa separazione consente di modificare le tecniche di generazione di regole a seconda di chi sarà ad utilizzarle, e consentirà grande pulizia e leggibilità del codice.

Questo schema è stato ideato per lo sviluppo di generiche simulazioni agent-based in Swarm, e ben si adatta alla simulazione d'impresa. Il meccanismo di generazione delle regole è molto simile ad una struttura aziendale dove la circolazione dell'informazione avviene proprio dal centro alla periferia. La direzione (RuleMaker) trasferisce informazioni verso i responsabili delle aree (RuleMaster) che a loro volta indicano agli agenti il comportamento da tenere.

Un altro indubbio vantaggio dello schema ERA è di tipo tecnico. Risulta molto più semplice modificare il codice di uno solo degli strati RuleMaster o RuleMaker, per modificare alcuni comportamenti, piuttosto che modificare direttamente la struttura degli agenti.

4.1.2 La separazione What to Do/Doing What

Il modello di Java Virtual Enterprise nasce dalla distinzione tra ‘what to do’ (WD) e ‘which is doing what’ (DW), ossia la contrapposizione tra quale sia lo scopo di vita dell’azienda e cosa essa deve fare per ottenere il risultato sperato.

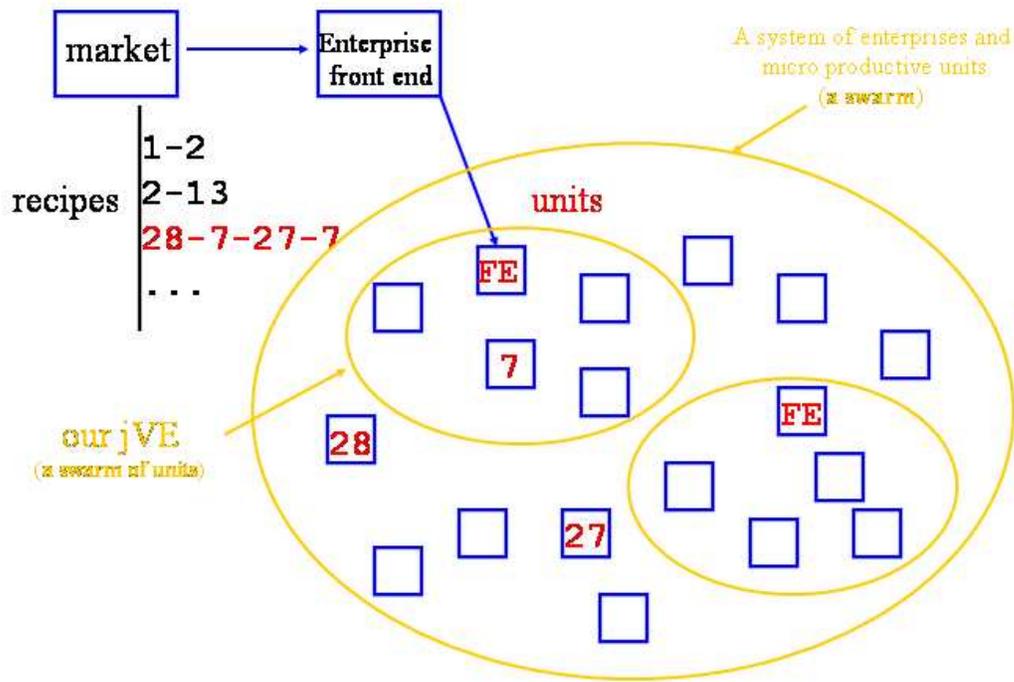


Fig. 4.2. Ricette e unità in JVE

Il WD è rappresentato da una ‘ricette’ produttiva.

Si può immaginare la ricetta proprio come una ricetta di una torta, nella quale si descrivono i passi da fare per ottenere il dolce (procurarsi gli ingredienti, mescolarli, infornare e così via).

Tecnicamente la ricetta è una sequenze di numeri interi, generalmente positivi.

Questa sequenza rappresenta i passi da eseguire per giungere all’ottenimento del prodotto finito o , come nel caso di BasicNet del servizio, che rappresenta lo scopo dell’impresa. Le ricette contengono quindi l’intelligenza, cioè le informazioni su come compiere un dato processo produttivo o organizzativo, ma non sono in grado di realizzarlo concretamente perché non posseggono gli strumenti.

In contrapposizione alle ricette si trovano le ‘unità’ che dovranno compiere le lavorazioni necessarie per l’ottenimento del servizio finale e che eseguiranno le attività concretamente.

Le unità produttive sono gli oggetti che sanno compiere i passi descritti nelle ricette. Nella metafora della torta le unità produttive saranno gli strumenti necessari per la preparazione (il forno, la teglia e così via).

Le unità sono le risorse produttive o organizzative dell’azienda. All’interno delle unità saranno contenute informazioni quali le operazioni che l’unità è in grado di svolgere, i costi fissi e variabili di utilizzo della risorsa, l’utilizzo del magazzino e la possibilità di essere *layer sensitive* o *insensitive*¹. Le unità produttive possono fare delle operazioni semplici, ma non conoscono il procedimento per giungere alla realizzazione di un prodotto finito, che rimane nelle ricette.

Sia le unità, sia le ricette posseggono una parte di conoscenza, o meglio definita intelligenza: le unità hanno capacità di svolgere attività, mentre le ricette sanno determinare quali siano le attività da svolgere e in quali tempi. L’intelligenza risulta in questo modo distribuita.

L’impresa virtuale comunica con l’ambiente esterno, come si può osservare in fig.4.2, per mezzo di unità definite Front End (FE), le quali ricevono ordini sotto forma di vettori numerici (28 8 65 45 2 nell’esempio).

Gli ordini sono oggetti informatici che rappresentano i beni da produrre. Gli ordini contengono le informazioni tecniche per la produzione (la ricetta) e le quantità

¹ Per il concetto di layer si veda 4.1.9

che dovranno essere prodotte.

Le unità che hanno in carico tali ordini vanno a interrogare le altre unità e se stesse per sapere quale di loro è in grado di eseguire il prossimo passo della ricetta, e mandano quindi l'ordine alla prima unità che risponde. Questo meccanismo si ripete sino ad aver ultimato la produzione. Anche gli ordini, come le ricette, posseggono parte dell'informazione distribuita nel sistema, e operano in modo decentrato.

Una volta inserito l'ordine nella Virtual Enterprise questo segue un percorso autonomo da qualunque altro elemento. L'ordine si sposta fisicamente da una unità all'altra, (ricordiamo che gli ordini sono veri e propri oggetti Java che vengono mossi all'interno della simulazione come oggetti reali) a seconda delle code di produzione che si sono formate, dei parametri del modello, della circolazione dell'informazione e di tutte le interazioni che nascono tra ordini, ricette e unità. E' proprio da queste interazioni, che si possono osservare in fig.4.3, che emergono i fenomeni complessi oggetto di studio, ed è da questi fenomeni imprevedibili che si spera possano verificarsi le così le cosiddette 'emergenze'.

All'interno delle Virtual Enterprise le assegnazioni avvengono in ordine First In First Out (FIFO).

Esattamente come nella realtà aziendale, rileviamo problemi di assegnazione quando più unità produttive possono svolgere la stessa lavorazione, all'interno o all'esterno della realtà dell'impresa che stiamo simulando. Questi problemi di assegnazione, se ricreati in modo realistico nella ricostruzione di un'azienda reale al computer, rendono possibile ipotizzare ottimizzazioni di processo e, simulandone l'effetto, comprendere quale sia la migliore configurazione produttiva.

4.1.3 I magazzini

Un problema rilevante nella realtà aziendale è la gestione dei magazzini. In particolare ci si pone il problema di come si debbano comportare le unità in assenza

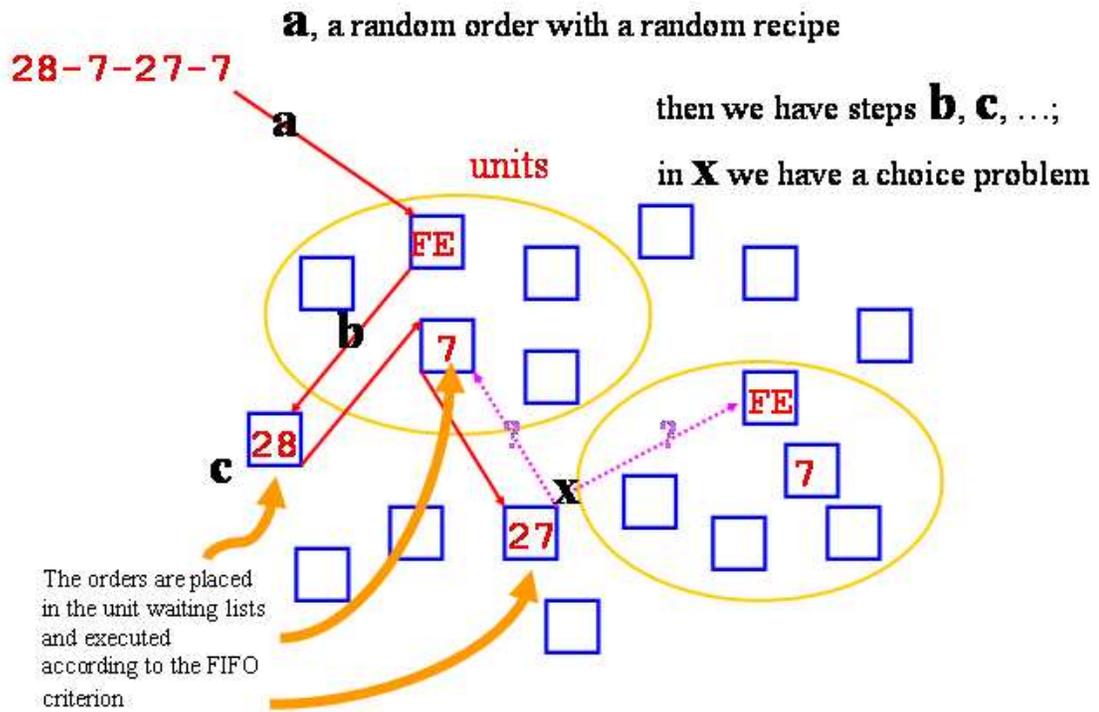


Fig. 4.3. Esempio di interazione tra ordini, ricette e unità

di ordini, cioè momentaneamente non occupate. Si immagina che ogni unità abbia un proprio magazzino privato, nel quale depositare i prodotti dell'unità. Quando questa unità è non richiesta da una ricetta si può ipotizzare che non faccia davvero nulla, oppure si può pensare che produca per i magazzini. In questo secondo caso l'unità è sempre attiva e, se non produce per degli ordini, allora produce per i magazzini. Quando arriverà all'unità un nuovo ordine non sarà necessario produrre il componente, ma questo verrà prelevato dal magazzino dell'unità.

La quantità di componenti che possono essere contenuti nei depositi delle unità varia all'interno di limiti determinati nei parametri della simulazione, ed è amministrata da un gestore delle regole del magazzino: la classe *InventoryRuleMaster.java*. A quest'ultima è affidato il compito di gestire le scorte e di inviare internamente degli ordini nel caso, ad esempio, in cui il limite superiore venga superato.

4.1.4 La trasmissione della conoscenza

Un aspetto essenziale nella realtà aziendale è la trasmissione della conoscenza. Il semplice fatto di poter prevedere quali saranno i prossimi passi produttivi necessari per completare l'ordine in lavorazione modifica profondamente il processo produttivo.

JVE consente di scegliere se propagare o no, alle unità interessate, le informazioni relative a quali saranno i prossimi passi produttivi necessari per completare l'ordine in lavorazione.

La questione della circolazione delle informazioni è ancora più rilevante se si considera la possibilità di scegliere la destinazione delle risorse scarse condivise tra più unità produttive, aspetto molto interessante da sperimentare. E' possibile che, in presenza di risorse condivise e di vincoli di impiego tra più unità produttive, emergano interazioni di rilevante complessità. La circolazione delle informazioni, infine, è un problema molto realistico nella realtà aziendale ed è in grado di determinare il

maggior o il minor successo dei risultati economici simulati dal funzionamento del modello.

Mandare o no l'informazione all'unità è un problema di cooperazione, abitudini, accordi taciti e espliciti, in definitiva il fulcro dello studio dell'organizzazione aziendale.

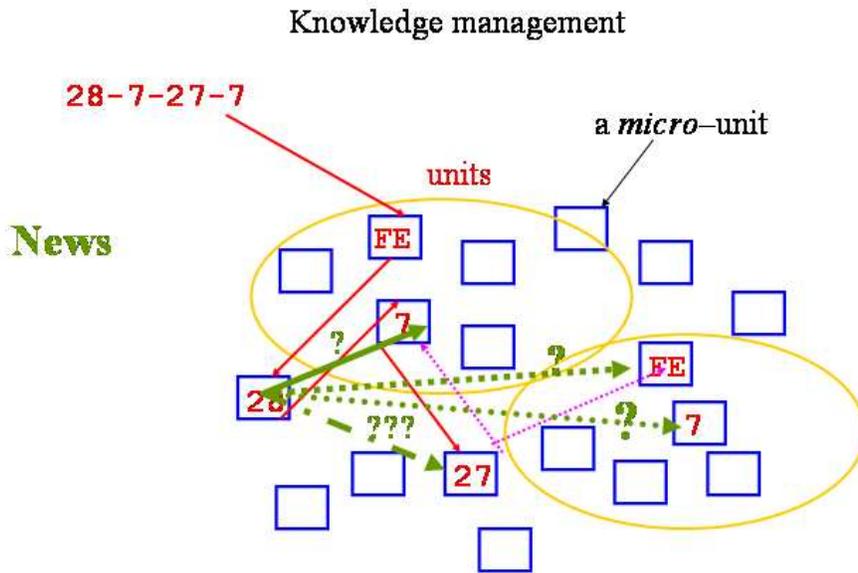


Fig. 4.4. Il problema del knowledge management in JVE

Da una prospettiva puramente teorica risulta interessantissimo poter simulare gli effetti dell'informazione sull'organizzazione. Teorici dell'organizzazione aziendale come Nonaka hanno studiato il processo di socializzazione della conoscenza² nell'impresa. Proprio queste simulazioni potrebbero essere una conferma di quelle teorie, specialmente oggi, quando la trasmissione dell'informazione accelera velocemente

² L'argomento è stato approfondito nel capitolo [3]

grazie alle nuove tecnologie, e molti modelli di business (come quello BasicNet) si fondano esclusivamente sulla trasmissione dell'informazione. Non solo ma, da un altro punto di vista, la possibilità offerta da uno strumento di indagine come la simulazione, consente di osservare anche vie non tentate precedentemente dall'imprenditore. La capacità di adattamento che questo sistema potrebbe sviluppare pongono in essere quella figura dell'imprenditore innovatore e ricercatore che sfrutta le opportunità derivanti dagli errori per ottenere un profitto. Risulta molto forte, quindi, anche il raccordo tra la teoria dell'ordine spontaneo di Hayek e la complessità emergente dalle simulazioni ad agenti in contesti d'impresa.

4.1.5 I meccanismi contabili

JVE è in grado di tenere una semplice contabilità. Ogni unità, infatti, è dotata di un sistema contabile. Per ogni ciclo produttivo viene imputato un costo fisso ed un costo variabile ad ogni unit.

Grazie a questo meccanismo è possibile computare i costi giornalieri, fissi e variabili, della virtual enterprise ed i costi totali come somma di quelli giornalieri.

- **I costi fissi** vengono computati giornalmente, indipendentemente dallo svolgimento di qualche attività da parte dell'unità.
- **I costi variabili** vengono computati solo se l'unità produce (anche se per i magazzini).

Ai magazzini viene imputato anche un costo finanziario, che dipende dalle quantità che giornalmente rimangono inutilizzate, sulla base di un criterio che può essere deciso nel model.

Il modello tiene un aggiornamento continuo della contabilità dei processi attivi sia con una prospettiva tradizionale di contabilità per unità produttive, con costi fissi e costi variabili, sia con una prospettiva analitica della contabilità secondo i

prodotti. In questo modo è possibile operare anche scelte di carattere economico e non solo organizzativo.

La contabilità viene registrata anche dal lato degli ordini in termini di costi fissi e variabili calcolati nel momento del passaggio attraverso le unità.

Grazie a questo meccanismo contabile è anche possibile conoscere i ricavi ottenuti, calcolati come prodotto tra la lunghezza della ricetta ed un tasso di ricarica predefinito, come rappresentato in fig.4.5.

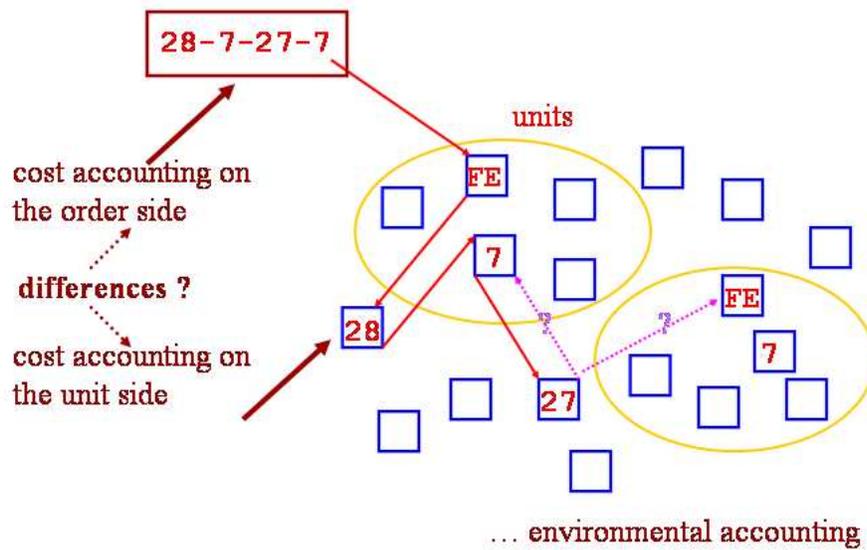


Fig. 4.5. I meccanismi contabile di JVE

La contabilità degli ordini ci fornisce due importanti informazioni:

- il costo di uno specifico bene;
- il costo totale della produzione di un ordine ;

- la valorizzazione del magazzino in un dato momento in termini di prodotti finiti e di semilavorati.

4.1.6 Produzioni batch

In alcuni casi le produzioni unitarie avvengono in tempi ridottissimi e a grande velocità. In questi casi non avrebbe senso ragionare sul singolo prodotto, ma appare più logico considerare lotti di prodotti omogenei. I lotti vengono gestiti in JVE con i meccanismi denominati *sequential batch* e *stand alone batch*.

Sequential batch

Un processo *sequential batch* lavora simultaneamente un insieme di ordini uguali tra loro. L'unità che deve svolgere la fase produttiva associata al *sequential batch* deve attendere che 'b' ordini di quella ricetta siano tutti arrivati alla fase produttiva del *sequential batch*. Dopo la lavorazione *sequential batch* vengono mantenuti separati tra loro i singoli ordini.

Come esempio la ricetta

```
1 s 1 2 s 1 3 s 5 \ b 4 s 1
```

esegue i passi 1 e 2 ma, per poter eseguire il passo di *sequential batch* 3, deve attendere che siano disponibili altre 'b-1' (considerando che un ordine esiterà certamente) ordini che presentino un *sequential batch* associato alla produzione 3, e che siano giunti tutti alla produzione 3.

Stand alone batch

Lo Stand alone batch produce simultaneamente un lotto di ordini.

Con un solo ordine, formato da un unico passo produttivo *stand alone batch*, si ottengono 'b' semilavorati destinati ad una *endUnit*.

Nell'esempio

1 s 5 / b e 1000

il passo 1 in 5 secondi produce 'b' semilavorati che verranno depositati nella *endUnit* 1000

4.1.7 I sub-processi

Il meccanismo delle sub-forniture viene gestito da JVE introducendo dei sub processi, cioè vere e proprie forniture, che devono intervenire all'interno della catena di produzione.

Un meccanismo di sub-forniture crea problemi di sincronizzazione dell'inizio delle diverse produzioni e delle relative componenti, sia che si tratti di gestione di reti all'interno di una stessa azienda, sia di un sistema integrato di aziende. In questo ultimo caso l'unità che si occuperà della fornitura verrà considerata una 'scatola nera', il cui funzionamento interno non verrà indagato ulteriormente. Grazie al formalismo delle ricette produttive è possibile esplodere a piacere, in modo ricorsivo, questo tipo di analisi e studiare, per ogni sub-processo, il livello di dettaglio desiderato.

Tecnicamente i sub-processi vengono concatenati tra loro attraverso un meccanismo che si definisce *procurement*. Il *procurement* è un passo speciale della ricetta produttiva che procura alcuni prodotti semilavorati, i quali rappresentano quindi sub processi. E' necessario che tutti i *procurement* richiesti siano stati eseguiti per poter proseguire con il passo successivo della ricetta.

I prodotti semilavorati, per poter essere soggetti a *procurement*, al termine delle loro lavorazioni vengono depositati in unità speciali, che hanno la funzione di magazzini, e che sono definite *endUnit*. Le *endUnit* possono essere sensibili o insensibili ai layer, come descritto in 4.1.9.

Un semplice esempio di sub- processo potrebbe essere:

```
subprocesso 1 s 1 2 s 1 e 1000 ;  
assemblaggio p 1 1000 3 s 1 ;
```

In questo esempio la ricetta *subprocesso* produce il semilavorato con i passi 1 e 2 entrambi della durata di secondi 1. Il semilavorato viene depositato nella *endUnit 1000*. La ricetta *assemblaggio* esegue il procurement di 1 prodotto dalla *endUnit 1000* e lo completa con il passo 3 della durata di 1 secondo.

4.1.8 Le capacità di scelta tra diverse lavorazioni possibili

La Virtual Enterprise è anche in grado di scegliere tra più lavorazioni definite possibili. Immaginiamo che per realizzare un prodotto esistano più modi possibili, alternativi tra loro. La ricetta produttiva di questo prodotto, nel modello JVE, conterrà tutti i metodi produttivi conosciuti, e deciderà di volta in volta quale utilizzare a seconda del criterio selezionato nel ModelSwarm.

I possibili criteri di scelta del ramo produttivo da utilizzare sono 5:

Criteria 0: vengono eseguiti tutti i rami in sequenza (viene spesso utilizzato in fase di test).

Criteria 1: viene sempre scelto il primo ramo.

Criteria 2: viene sempre scelto il secondo ramo.

Criteria 3: viene scelto il ramo in modo casuale (questo criterio è utilizzato quando si vuole creare un bilanciamento sul carico di lavoro delle unità produttive coinvolte nei sotto processi).

Criteria 4: viene scelto il ramo la cui coda di produzione del primo passo è più breve.

Criteria 5: viene utilizzato il risultato di un oggetto computazionale per scegliere il ramo (per maggiori dettagli si veda la sezione 4.1.10).

Le lavorazioni alternative vengono indicate nelle ricette con il simbolo ‘||’ seguite dal numero del ramo, l’ultimo ramo si conclude con ‘||0’. Un esempio di ricetta con lavorazioni alternative è:

1 s 1 ||1 10 s 2 11 s 1 ||2 20 s1 21 s 4 ||0 30 s 1

In questo esempio viene eseguita la lavorazione 1 poi si presentano 2 alternative:

1. primo ramo: eseguire i passi 10 e 11
2. secondo ramo: eseguire i passi 20 e 21

L’ordine, dopo aver intrapreso il primo o il secondo ramo, esegue la lavorazione 30.

Nel caso si utilizzi il criterio 5 per la scelta del ramo, e quindi siano gli oggetti computazionali a indicare il percorso da seguire come descritto nella sezione 4.1.10, il codice del primo ramo (che nell’esempio è 1) sarebbe 10xx dove xx indica il numero della riga della matrice che dovrà essere letta per decidere come procedere nella produzione.

4.1.9 I layer

I layer sono dei contenitori di prodotti assolutamente indipendenti tra loro. Ogni ordine appartiene ad un layer (il layer 0 se non diversamente specificato).

Se si specificano layer diversi per ordini diversi JVEFrame 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.

L'utilizzo dei layer è assolutamente automatico all'interno di JVE. L'unica interazione richiesta all'utente è indicare il layer di riferimento per ogni ordine all'interno dell'*orderSequence*³. Per escludere questa divisione è necessario indicare l'unità che si vuole rendere insensibile ai layer con un numero negativo. Anche le matrici utilizzate negli oggetti computazionali possono essere rese insensibili ai layer con lo stesso stratagemma, indicandole con un numero negativo (ovviamente questo non è possibile nella matrice 0).

Per il corretto funzionamento dei layer è sufficiente indicare all'interno del ModelSwarm il numero massimo di layer utilizzato (*maxLayerNumber*).

4.1.10 Gli oggetti computazionali

Gli oggetti computazionali sono metodi JavaSwarm che vengono richiamati da passi speciali della ricetta produttiva, e che compiono delle computazioni su valori contenuti all'interno di matrici definite *memoryMatrix*.

Lo scopo degli oggetti computazionali è consentire al modello di eseguire operazioni immateriali, fare previsioni, gestire aste, indirizzare procurement o qualunque altra operazione si ritenga utile alla simulazione.

Questo strumento conferisce grande flessibilità al modello, e consente di simulare non solo imprese tradizionali (come la V.I.R. s.p.a.) ma un qualunque processo produttivo o organizzativo (proprio come nel caso BasicNet).

Le matrici che vengono utilizzate per contenere i valori sui quali fare le computazioni, e le matrici dentro le quali salvare i risultati ottenuti, sono di regola sensibili ai layer. Questo significa che ne esistono diverse, con lo stesso codice di riferimento, a seconda del layer di appartenenza dell'ordine. Per rendere le matrici insensibili ai

³ Per dettagli sul funzionamento dell'*orderSequence* si veda la sezione 5.7

layer è sufficiente indicarne il codice identificativo con un numero negativo. Le matrici che vengono utilizzate da ogni oggetto computazionale vengono indicate nella ricetta a fianco dell'oggetto computazionale di riferimento secondo la sintassi:

```
c codice n m1 ... mn
```

Dove 'c' è il codice che identifica gli oggetti computazionali in generale, 'codice' è il codice specifico dell'oggetto computazionale che si vuole utilizzare, 'n' è il numero di matrici che devono essere usate e 'm1 ... mn' sono i codici di queste matrici.

Un esempio potrebbe essere:

```
ricetta 1 s 1 c 1901 2 10 11 2 s 1
```

L'ordine 'ricetta' esegue il passo 1 della durata di un secondo e il passo 2 della durata di un secondo. Il passo 2 consiste in una computazione dal codice 1901. Vengono utilizzate 2 memoryMatrix, che sono la 10 e la 11.

I metodi che gli oggetti computazionali ereditano, e che consentono di operare sulle memoryMatrix, sono:

- setValue(layer, (int) row, (int) col, (double) value) oppure
setValue(layer, (int) row, (int) col, (float) value)
- (float) getValue(layer, (int) row, (int) col)
- setEmpty(layer, (int) row, (int) col)
- (boolean) getEmpty(layer, (int) row, (int) col)

Gli oggetti computazionali offrono un ulteriore strumento: consentono di bloccare le ricette. All'interno di ogni oggetto computazionale esiste una variabile booleana denominata *done*. Fino a quando questa variabile non verrà impostata sul valore *true* l'oggetto non considererà terminata la computazione, e terra quindi bloccato

l'ordine al passo corrente. Questo strumento si rivela indispensabile nel caso della simulazione BasicNet per interrompere la produzione dei campionari che non sono stati ordinati. Dalla versione jveFrame-0.9.7.30 esiste un parametro del *ModelSwarm* chiamato *maxTickInAUnit*, che definisce il numero massimo di tick di permanenza degli ordini nelle unità. Trascorso questo numero di tick se l'ordine risulta ancora bloccato nell'unità verrà cancellato. Se il valore è impostato a 0 questa funzionalità non è attiva.

Attraverso gli oggetti computazionali è anche possibile gestire i passi *OR* descritti nella sezione 4.1.8. Nel *ModelSwarm* viene indicata la 'orMemoryMatrix', una memoryMatrix speciale, che viene utilizzata dalla Virtual Enterprise per decidere quale ramo dell'*OR* scegliere. La prima colonna di questa matrice conterrà i valori numerici che indicano in modo sequenziale quale ramo scegliere per ogni passo OR. Il ramo che dovrà scegliere, per esempio, il passo OR numero 1001 sarà contenuto nella posizione (1,0) della matrice, mentre la scelta per il ramo numero 1002 sarà contenuta in posizione (2,0).

Il contenuto della matrice verrà inserito dagli oggetti computazionali.

4.1.11 La scelta del codice esterno, intermedio e interno

JVE adotta un linguaggio interno differente. Le ricette così come sono state descritte fino ad ora devono quindi essere trasformate. Consideriamo come esempio la ricetta

```
1 s1 c 1999 3 0 1 3 2 s 2 3 s 2 ;
```

il cui significato è:

- esegui il passo 1 della durata di secondi 1
- richiama l'oggetto computazionale *c 1999* che utilizza 3 memoryMatrix, la 1, al 3 e la 2;

- esegui il passo 2 della durata di secondi 2;
- esegui il passo 3 della durata di secondi 2.

Questo tipo di notazione, facilmente leggibile da noi umani, per poter essere interpretata dal programma JVE deve essere trasformata in un formato definito ‘intermedio’. La ricetta usata come esempio in formato intermedio diventerà:

```
1 2 2 -1999 3 0 1 3 10000002 3 3 ;
```

il cui significato è:

- esegui il passo 1;
- esegui il passo 2 per 2 tick;
- esegui l’oggetto computazionale c1999 che assume codice intermedio -1999 ed è collegato al passo 10000002 definito per convenzione a durata 0 tick⁴.
- esegui il passo 3 per 2 tick.

Per trasformare le ricette dal formato esterno, cioè come scritte nel file *recipe.xls*, al formato intermedio, vengono utilizzate le classi *recipe.java* e *orderDistiller.java*. Queste classi sono state modificate da Francesco Merlo e me per gestire i layer e gli oggetti computazionali. Per maggiori informazioni si faccia riferimento alla sezione 5.7.

Il modello JVE interpreta il formato intermedio, ma gestisce le operazioni trasformando ulteriormente le ricette in un formato definito ‘interno’. Questo tipo di notazione è difficilmente comprensibile dagli umani, ma veloce da eseguire e semplice per il computer.

⁴ I passi con codice maggiore di 10000000 sono interpretati da JVE come a tempo 0

4.2 Altri modelli di simulazione d'impresa

4.2.1 JVE in MatLab-SimuLink-StateFlow

Per le informazioni contenute in questo capitolo ringrazio Alessandro Raimondi. La descrizione del funzionamento di MatLab, SimuLink e StateFlow è liberamente tratta da Raimondi (2002).

MatLab è un ambiente di lavoro interattivo, un linguaggio di programmazione e un ambiente di sviluppo. Può essere utilizzato per analizzare e visualizzare dati, implementare algoritmi, modellare e simulare o sviluppare applicazioni ed è fondato sulle matrici. Gli elementi di una matrice possono consistere sia di numeri sia di caratteri, e i dati sono array dimensionati in modo dinamico. L'ambiente è interpretato, non compilato, però compilabile.

SimuLink è un pacchetto aggiuntivo di MatLab per modellare, simulare ed analizzare sistemi dinamici, sviluppare sistemi lineari e non lineari simulati nel continuo, nel discreto o ibridi. La modellizzazione avviene attraverso un'interfaccia grafica che permette di costruire diagrammi a blocchi.

I modelli hanno una struttura gerarchica: si possono dunque costruire modelli che abbiano una struttura che si sviluppa dal basso verso l'alto e viceversa. Possiamo così visualizzare il sistema ad alto livello, e da qui entrare nei blocchi e scendere verso i livelli inferiori, aumentando il dettaglio di analisi del modello.

Questo modo di lavorare consente di padroneggiare al meglio la struttura del modello e di evidenziare le interazioni fra le varie parti che lo costituiscono. Una volta definito il modello parte la simulazione, che può essere gestita sia da SimuLink sia da MatLab. Utilizzando sonde e altri blocchi di visualizzazione è possibile vedere i risultati in tempo reale, inoltre è possibile variare i parametri e verificarne immediatamente l'effetto. I risultati possono essere inviati al Workspace di MatLab per effettuarne l'analisi con gli innumerevoli strumenti offerti da questo applicativo.

StateFlow è un pacchetto aggiuntivo di MatLab si fonda sulla teoria delle macchine a stati finiti (FSM). Una FSM è la rappresentazione di un sistema guidato dagli eventi: questo sistema effettua una transizione da uno stato ad un altro stato predefinito, se è realizzata la condizione che definisce la transizione.

Un diagramma di StateFlow è la rappresentazione grafica di una FSM in cui stati e transizioni formano i blocchi fondamentali di costruzione del sistema. StateFlow crea un blocco da inserire in un modello SimuLink: un insieme di blocchi StateFlow in SimuLink costituiscono una macchina StateFlow.

Alessandro Raimondi ha realizzato per la sua tesi di laurea una versione semplificata del modello di Virtual Enterprise in MatLab-SimuLink-StateFlow. La simulazione assume l’aspetto di un diagramma a blocchi.

Lo scopo di questo esperimento è rendere comprensibile in modo intuitivo i principi del funzionamento del modello JVE.

Si intende anche studiare il rapporto tra la simulazione ad agenti e la simulazione di processo. Si vuole vedere se è possibile ridurre il modello JVE ad un flusso di processi, e se questo porta ai medesimi risultati della simulazione Agent-Based in Swarm.

L’obiettivo è stabilire se possono nascere *emergenze* anche in una simulazione in cui le relazioni fra gli elementi sono predeterminate.

La fig.4.6 mostra in basso a sinistra il grafico SimuLink del modello. Le ricette sono rappresentate dai vettori [1,2,3] e [2,1,2]. Lo ‘Switch’ decide quale ricetta debba essere mandata alla produzione, ovvero in ‘Units’.

Il blocco ‘Units’ rappresenta il vero e proprio motore della Virtual Enterprise ed è realizzato con StateFlow. Si può osservarne il funzionamento nel grafico in alto a destra. Il meccanismo studiato da Raimondi legge la posizione i del vettore-ricetta e esegue la produzione corrispondente. L’indice i viene incrementato e il motore esegue quindi la produzione successiva. Il processo continua fino al termine della

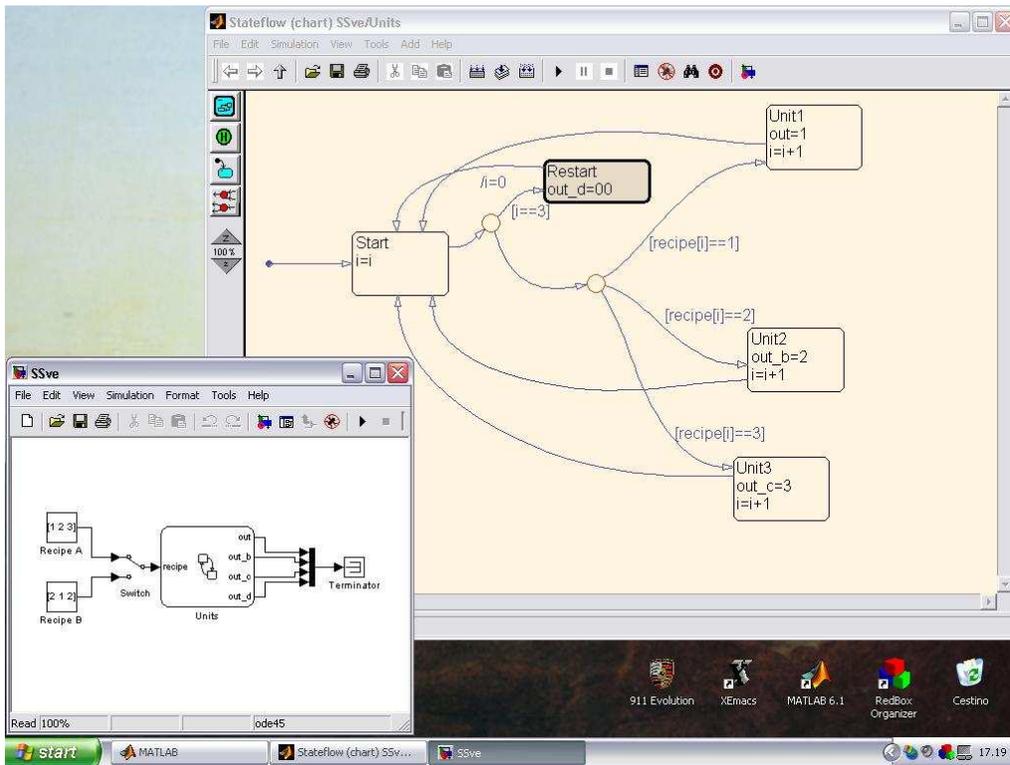


Fig. 4.6. Motore del modello JVE realizzato in MatLab-SimuLink

ricetta, quando si attiva la condizione di uscita (in questo esempio si ipotizza che le ricette possano essere lunghe al più tre passi, e quindi quando $i == 3$) viene interrotta la produzione.

4.2.2 JVE in StarLogo

Per le informazioni in questo capitolo ringrazio il dott. Michele Sonnessa. Nel suo progetto StarLogoVE (StarLogo Virtual Enterprise) ha ricreato in modo semplificato il modello di Virtual Enterprise in StarLogo. StarLogo è uno strumento per la creazione di modelli simulativi di sistemi complessi Agent-Based descritto nella sezione 2.4.1.

Lo scopo della simulazione è spiegare, in modo grafico, il funzionamento del

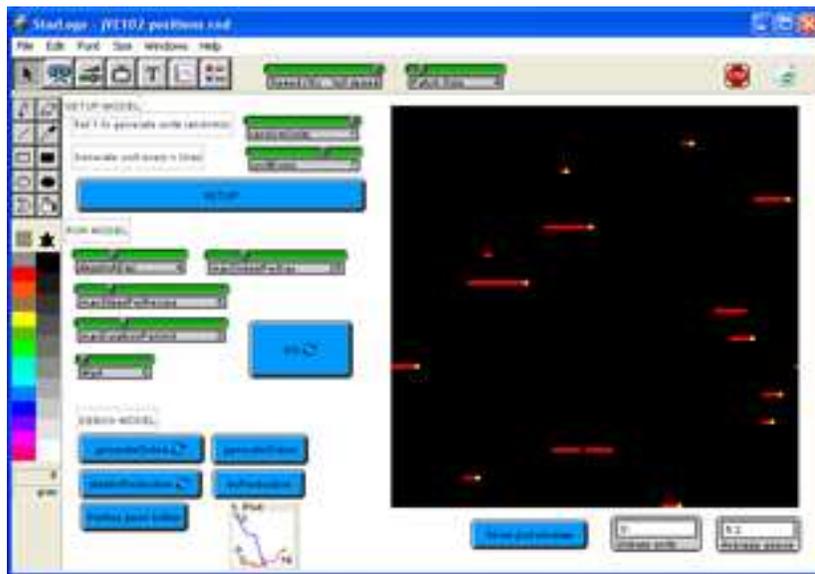


Fig. 4.7. Control center del modello JVE realizzato in StarLogo

modello.

La finestra principale in fig.4.7 mostra le unità produttive, ognuna con la propria coda di produzione. Premendo il tasto *startdoProduction* le unità iniziano a produrre, le code diminuiscono e i prodotti si spostano da un'unità all'altra.

Nella metafora JVE ogni ordine è descritto da una ricetta produttiva che ne indica i passi di produzione in sequenza, nel modello StarLogo questa ricetta è generata in modo casuale, e determina gli spostamenti dei puntini/prodotti da un'unità all'altra.

Con il tasto *generateOrders* vengono generati nuovi ordini di produzione. Graficamente dal centro dello schermo partono dei punti colorati che vanno ad accodarsi alle unità produttive.

La fig.4.8 mostra:

- **Tratto blu:** il numero di unità *unBusy*, cioè che non stanno producendo. Si può notare che il valore diminuisce all'aumentare degli ordini nel modello.

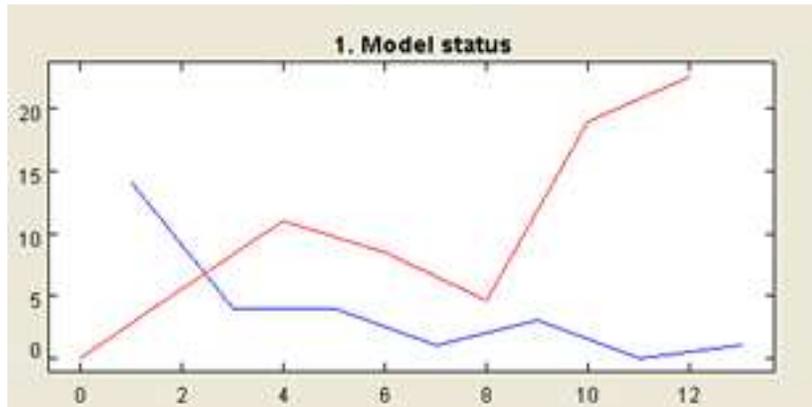


Fig. 4.8. Finestra ‘plot’ del modello JVE realizzato in StarLogo

- **Tratto rosso:** il numero medio di prodotti in coda. Il valore aumenta con l’aumentare degli ordini in lavorazione.

4.2.3 NIIP

Il progetto NIIP⁵ (National Industrial Information Infrastructure Protocols consortium) nasce nel 1994 da un accordo tra il Governo americano e alcune industrie, per sviluppare un protocollo software che consenta di migliorare la comunicazione tra le imprese e ottenere così un’interazione nel processo produttivo. Gli obiettivi del progetto sono descritti in <http://www.niip.org> (2002):

to develop, demonstrate, and transfer into widespread use the technology to enable Industrial Virtual Enterprises. A Virtual Enterprise is a temporary organization of companies that come together to share costs and skills to address business opportunities that they could not undertake individually. Industrial Virtual Enterprises, with NIIP technology, foster collaborative efforts and the sharing of engineering and manufacturing information.

⁵ Per maggiori informazioni si veda la homepage del progetto <http://www.niip.org> (2002)

A differenza del modello JVE, che intende simulare una singola impresa, l'obiettivo finale di NIIP è quello di essere impiegato sul mercato americano come standard di dialogo tra tutte le imprese. Trattandosi di un progetto di enormi dimensioni richiede una fase progettuale lunga e i risultati sono difficilmente prevedibili.

BIBLIOGRAFIA

<http://www.mathworks.com>. (2002). *Mathworks homepage*.

<http://www.niiip.org>. (2002). *Niiip project homepage*.

<http://www.teoresi.it>. (2002). *Teoresi homepage*.

Raimondi, A. (2002). *Matlab, simulink e stateflow, una breve introduzione alla simulazione*. (Presentazione presso la facoltà di Economia di Torino del 15 Gennaio 2002)

Terna, P. (2002a). *Improved java virtual enterprise (jve) in swarm, presentazione per la swarmfest 2002, seattle*.

Terna, P. (2002b). Simulazione ad agenti in contesti di impresa. *Sistemi intelligenti*, XVI(1), pp.33-51.

BIBLIOGRAFIA

Capitolo 5

BASICJVE - ANALISI DELLA BASICNET IN PROSPETTIVA DI FORMALIZZAZIONE DEL MODELLO JVEFRAME

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

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 <http://www.basicpress.com> (2002), 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 detenute 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 ¹.

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

¹ Tratto da <http://www.basicpress.com> (2002)

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



Fig. 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 né imporre né 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.

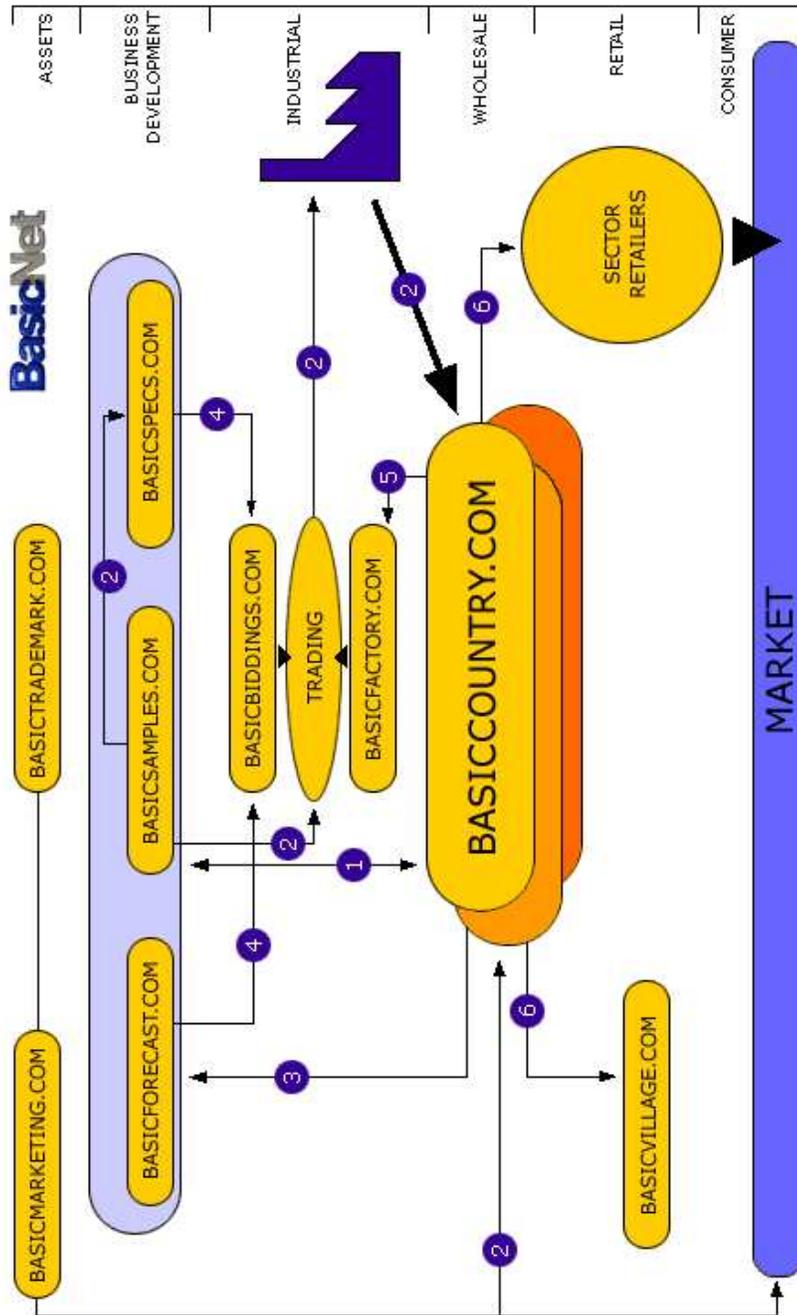


Fig. 5.2. Organigramma del modello di business BasicNet

5.3 Il processo di formalizzazione: da BasicNet a BasicJve

5.3.1 Scopo della simulazione

Simulazione ad agenti e JVEFrame

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 pre-determinate di eventi; nel nostro modello gli eventi accadono in modo indipendente, generando interazione anche imprevedibili tra atti produttivi e unità produttive, proprie della complessità.

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, 2002d)

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.

JVEFrame 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, JVEFrame 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.

JVEFrame 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-meccanismi operanti nel sistema economico, oppure unità integrate all'interno di imprese (Terna, 2002d)

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.

5.4 *Processo di formalizzazione delle ricette*

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

Nei mesi da ottobre a maggio il nostro lavoro è stato principalmente comprendere il funzionamento del modello JVE 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.

5.4.1 Prima formalizzazione

Le ricette

La Virtual Enterprise 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 JVEFrame e, soprattutto, della BasicNet.

Introduciamo in questa sezione una prima formalizzazione delle ricette produttive necessarie per la simulazione del Business System di Basic.Net. Descriveremo quindi le principali fasi organizzative necessarie per 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 JVE 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).

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]

ALTRIMENI

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] indentifica 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) COMMERCIALIZZAZIONE

[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 JVEFrame; 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 JVEFrame 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 JVEVir, 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.

5.4.2 Seconda formalizzazione

Presentiamo in questa sezione una nuova formalizzazione delle ricette organizzative per l'applicazione del modello JVEFrame 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 viene 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	5	6	7	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'inventario 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 la 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.

5.4.3 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 viene 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]
CO [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 campionario

```
40 d 1 C20 [ ] [TH] p (LC) 1003(IC)(ID)(*) 41 d 1
C30 [FC i1003(IC)(ID)(*), TH]
[T/F,SFC] C0 [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 campionario 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 JVEVir. 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.

5.4.4 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 grammatica di JVE.

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 JVE-Frame, 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.

5.4.5 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.5.1 mostra un esempio di *map*

Sul modello dell'applicazione *JVEVir* 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. 5.2 riporta un esempio di *orderSequences.xls*:

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

Tab. 5.1. 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			

Tab. 5.2. 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 JVEFrame, i layer.

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.

5.5 Proposte di modifica al codice

5.5.1 Prime proposte

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

Le prime proposte sono state respinte perché tentano 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 più 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 m1- m2 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 ni 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 ni 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

5.5.2 Le modifiche apportate

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

Per poter adattare JVE 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 di JVEVir², 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

² Abbiamo 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.

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 (1 = 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).
- Attualmente la nostra versione presenta delle stampe a video per meglio comprendere il funzionamento interno del modello.

5.6 BasicJve-0.9.7.30.b: BasicJVE la simulazione di BasicNet

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

5.6.1 Dibattito sull'uso dei layers

La prima proposta di BasicJVE 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. JVE-Frame 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'orderSequences 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.

5.6.2 Il modello: le unità

Per far funzionare concretamente la simulazione della BasicNet abbiamo dovuto formalizzare all'interno di JVE 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* (per informazioni sul funzionamento dei layer si veda la sezione 4.1.9).
 - (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 (per informazioni sul funzionamento dei magazzini si veda la sezione 4.1.3).
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, quest'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’

```
end_unit_#;  
_use_positive_code_for_layer_sensitive_end_unit;  
_negative_for_insensitive_  
  
1001  
1002
```

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.

5.6.3 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 5.3 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

Tab. 5.3. Calendario stimato per le principali attività organizzative

Analisi dei prezzi

La tabella 5.4 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

Tab. 5.4. Analisi del prezzo degli articoli

Analisi degli ordini totali

La tabella 5.5 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 5.6 riporta le stesse informazioni della tabella 5.5 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

Tab. 5.5. 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

Tab. 5.6. Analisi degli ordini dei singoli licenziatari su una collezione grande in un semestre

Analisi degli ordini di una collezione piccola

La tabella 5.7 riporta le stesse informazioni della tabella 5.5 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

Tab. 5.7. Analisi degli ordini dei singoli licenziatari su una collezione piccola in un semestre

Analisi delle collezioni Kappa

La tabella 5.8 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

Tab. 5.8: Numero articoli e quantità ordinate per le collezioni Kappa in un semestre

Analisi delle collezioni Robe di Kappa

La tabella 5.9 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

Tab. 5.9. Numero articoli e quantità ordinate per le collezioni Robe di Kappa in un semestre

Analisi dei tempi di produzione degli articoli

La tabella 5.10 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

Tab. 5.10. Stime dei tempi di produzione per ordine e singolo articolo

Analisi dei tempi di produzione divisi per Trading Company

La tabella 5.11 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

Tab. 5.11. 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 5.12 riporta le ricette contenenti i passi necessari per ogni processo organizzativo in notazione ‘esterna’³. 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 il progetto JVir

- 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.

³ Per il funzionamento dei diversi tipi di notazione in JVE si veda 4.1.11

- 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 anno⁴.

Nome ricetta	Codice	Passi									
analisitendenze	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	;			
disegnetacampionario	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	;
speceprodacampionario	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	;
produzioneprodotto	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	;		

Tab. 5.12. Le ricette utilizzate nella simulazione

5.6.4 Il modello: le matrici di memoria

Le dimensioni di ogni matrice vengono dichiarate all'interno del file *unitData/memoryMatrices.txt*; per la nostra simulazione è stato preparato nel modo seguente:

```
number(from_0_ordered;_negative_if_insensitive_to_layers)_rows_cols
```

⁴ Si è deciso di simulare tre anni di gestione, e lasciare girare il programma per un ulteriore anno, nel quale smaltire le code formatesi

0	1	1
1	3	301
2	3	301
3	3	301
4	3	301
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 5.13 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	<i>Numero di metacampionari disegnati</i>

Tab. 5.13. Matrice di memoria *designMatrix*

agentMatrix

La tabella 5.14 riporta il contenuto delle MemoryMatrix da 1 a 35, denominate agentMatrix. Nella colonna 0 vengono creati i contatori⁵.

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.

Matrici 1 - 35					
agent#Matrix	0	1	2	...	300
0					
1	cont.	previsioni art.1	previsioni art.2	...	previsioni art.300
2	cont.	ordini art.1	ordini art.2	...	ordini art.300

Tab. 5.14. Matrice di memoria *agentMatrix*

tcMatrix

La tabella 5.15 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

Tab. 5.15. Matrice di memoria *tcMatrix*

basicNetMatrix

La tabella 5.16 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:

⁵ Per il funzionamento dei contatori si veda 5.7.3

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		<i>somma previsioni art.1</i>	<i>somma previsioni art.2</i>	...	<i>somma previsioni art.300</i>
2		<i>status art.1</i>	<i>status art.2</i>	...	<i>status art.300</i>
3		<i>prezzo prod. art.1</i>	<i>prezzo prod. art.2</i>	...	<i>prezzo prod. art.300</i>
4	cont.	<i>somma ordini art.1</i>	<i>somma ordini art.2</i>	...	<i>somma ordini art.300</i>

Tab. 5.16. Matrice di memoria *basicNetMatrix*

orMatrix

La tabella 5.17 riporta il contenuto della MemoryMatrix 42, denominata *orMatrix*. Questa matrice è utilizzata internamente da JVE per scegliere il ‘branch’ dei passi *or* da eseguire. Il meccanismo di JVE 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 JVE per scegliere il ‘branch’ 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>

Tab. 5.17. Matrice di memoria *orMatrix*

5.6.5 Il modello: la sequenza degli ordini

Ordini di una collezione grande

La tabella 5.18 riporta un estratto del file *recipeData/orderSequences.xls* che gestisce il lancio degli ordini scanditi nel tempo per una collezione grande di 300 articoli⁶. 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/-collezioni.

Ordini di una collezione piccola

La tabella 5.19 riporta un estratto del file *recipeData/orderSequences.xls* che gestisce il lancio degli ordini scanditi nel tempo per una collezione SMU di 30 articoli⁷.

5.6.6 Il modello: i parametri della simulazione

I parametri utilizzati per la simulazione sono descritti nel file *jveframe.scm* riportato di seguito:

⁶ Si riferisce alla collezione ‘Robe di Kappa’ Primavera-Estate 2002, corrispondente al layer 1

⁷ Si riferisce alla collezione ‘Robe di Kappa’ Speciale Scuola 2002, corrispondente al layer 7

Shift	Layer		Ordini															
				*	1													
51	1	1	10	*	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	*	1
...																		
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												

Tab. 5.18. Esempio di sequenza degli ordini per una collezione di 300 articoli

Shift	Layer		Ordini															
			10	*	1													
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												

Tab. 5.19. Esempio di sequenza degli ordini per una collezione di 30 articoli

```
(list
  (
    cons 'vEFrameObserverSwarm
    □□□□□□□□(
    □make-instance□'VEFrameObserverSwarm
      #:displayFrequency      1
      #:verboseChoice         #f
      #:printMatrixes        #f
      #:checkMemorySize      #f
      #:unitHistogramXPos    10
      #:unitHistogramYPos    300
      #:endUnitHistogramXPos 10
      #:endUnitHistogramYPos 250
    )
  )
  (
    cons 'vEFrameModelSwarm
    □□□□□□□□(
    □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 del passo *or* è la 42.

5.7 Le modifiche al codice

Le modifiche al codice riguardano 3 files: *orderDistiller.java*, *recipes.java* e *ComputationalAssembler.java*.

5.7.1 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 *orderSequences.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.xlsx* è:

```
... c x n m1 m2 ... mn ...
```

dove $c x$ indica 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 $m1 m2 \dots mn$.

Dal punto di vista informatico è stato necessario modificare il file *orderDistiller.java* come segue⁹.

```
public class OrderDistiller extends OrderGenerator {  
  
    public boolean  
    worksheetOrderSequenceFileOpen = false ,  
    worksheetRecipeFileOpen = false ;  
    int [] orderSequence1 , orderSequence2 , orderSequence3 ;  
}
```

⁸ Se il foglio excel *orderSequences.xls* arrivasse alla fine, la lettura ripartirebbe dalla prima ricetta appartenente all'ultimo layer impostato

⁹ Per il listato completo del codice si veda l'appendice

È stato aggiunto un array di tipo *integer* denominato *orderSequence3* che contiene le indicazioni del layer di riferimento per ogni ricetta. Il vettore viene 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.

```
public String semicolon = ";", checkTheCell, gate = "#", p = "p",
sec = "s", min = "m", end = "e", slash = "/", backslash = "\\",
or_ = " || ", layer = "l", computation = "c";
```

Viene aggiunta la variabile stringa *layer* contenente la lettera l, il segno distintivo dell'impostazione dei layers all'interno di *orderSequences.xls*.

```
public static boolean firstTime = true;
```

Viene inizializzata la variabile globale *firstTime = true* che prima era presente solo nel metodo *distill*. In questo modo viene impostata come *true* solo all'avvio della simulazione, e non ogni volta che viene eseguito il metodo *distill*. Questa modifica è necessaria per il corretto funzionamento e la compatibilità con l'applicazione JVEVir.

```
int currentLayer = 0;
int maxOrderSequence;
```

Vengono aggiunte le variabili globali necessarie alla lettura dei layer e al dimensionamento degli array.

```
public void setDictionary () {
```

¹⁰ Ogni shift, o turno della simulazione, è rappresentato da una riga del file *orderSequences.xls*

```
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 parse delle ricette. Abbiamo inizializzato il vettore *orderSequence3 = new int[maxOrderSequence]* di lunghezza pari a *maxOrderSequence*.

```
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 viene 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 JVE¹¹. 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*.

```
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()* viene fatto il casting delle variabili locali, tra le quali *layerNumber* di tipo *integer*.

Ogni volta che viene distillata, cioè letta nell'*orderSequences.xls* una ricetta, viene imposto il numero del layer della ricetta con il comando *anOrder.setOrderLayer(layerNumber)*. Se nell'*JVEFrameObserverSwarm* è selezionato *verbose = true* viene stampato un report a video sulla nascita dell'ordine.

```
public void readOrderSequence() {

    int i, numberOfShift;

    if (StartVEFrame.verbose)
        System.out.println("firstTime_is_" + firstTime);
}
```

¹¹ Per la differenza tra notazione esterna, intermedia e interna si veda la sezione 4.1.11

```

i = 0;
if (! worksheetOrderSequenceFileOpen && firstTime) {
    orderSequenceWorksheet = new ExcelReader
    ("recipeData/orderSequencesModified.xls");
    worksheetOrderSequenceFileOpen = true;
    firstTime = false;
    if (StartVEFrame.verbose)
    System.out.println ("orderSequencesModified.xls has been open");
}
else if (! worksheetOrderSequenceFileOpen) {
    orderSequenceWorksheet = new ExcelReader
    ("recipeData/orderSequences.xls");
    worksheetOrderSequenceFileOpen = true;

    if (StartVEFrame.verbose)
    System.out.println ("orderSequences.xls has been open");
}

```

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 volta che viene richiamato il metodo. Grazie a questo accorgimento la prima volta che si richiama il metodo verrà aperto il file *orderSequencesModified.xls* (necessario per la compatibilità con l’applicazione JVir) e al ciclo di lettura successivo¹² verrà aperto il file *orderSequences.xls*.

```

numberOfShift = orderSequenceWorksheet.getIntValue();
if (StartVEFrame.verbose)
System.out.println ("The shift#" + numberOfShift + " has begun");

```

¹² Ricordiamo che quando termina la lettura del file, il processo viene iniziato nuovamente e la lettura riprende dall’inizio

```

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.

```

while (! orderSequenceWorksheet.eol()) {
    checkForLayer (orderSequenceWorksheet);

    orderSequence1 [ i ] = errorIsNotAnInteger (orderSequenceWorksheet);
    orderSequence3 [ i ] = currentLayer;

    errorIsNotAString (orderSequenceWorksheet);
    orderSequenceWorksheet.getStrValue ();

    orderSequence2 [ i ] = errorIsNotAnInteger (orderSequenceWorksheet);
}

```

Abbiamo aggiunto un controllo sulla modifica dei layer. Se questi vengono modificati viene richiamato il metodo `checkForLayer`.

```

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*.

```
public void checkForLayer(ExcelReader e){  
  
    if(e.checkForLabelCell()){  
        checkTheCell = e.getStrValue();  
  
        if(checkTheCell.equals(layer))  
            currentLayer = e.getIntValue();  
        if(StartVEFrame.verbose)  
            System.out.println("currentLayer_□now_□is_□" + currentLayer);  
    }  
}
```

Il metodo *checkForLayer(ExcelReader e)* ha come argomento un file del tipo *ExcelReader* (nel nostro caso *orderSequences.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.

5.7.2 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 JVE.

Il nostro intervento sulla classe è servito per permettere la trasformazione della notazione degli oggetti computazionali come descritto in 4.1.11.

```
public String semicolon = ";", checkTheCell, gate = "#", p = "p", end  
    = "e",  
    sec = "s", min = "m", slash = "/", backslash = "\\", □or□="||",  
    □computation="c";
```

Vengono dichiarate le variabili globali della classe. Abbiamo aggiunto *computation=c*.

```
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 viene richiamato ogni qual volta si incontra una c nella ricetta.

5.7.3 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*.

```

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 computationally 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.

```
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.

```
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 BasicJVE.

```
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

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.

```
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.

```

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.getEmpty(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. La soglia è stata impostata a 18000.

C1904

L'oggetto computazionale **c1904** simula la produzione dei singoli campionari.

```
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.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);
    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.

```

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.isEmpty(layer,0,0))
{
int counter = (int) designMatrix.getValue(layer,0,0);

for (int i=1; i<=counter; i++)
{
if (!basicNetMatrix.isEmpty(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
}

```

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.

```

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.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

```

```
}

```

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.

```
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))
        {
            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.

```
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.isEmpty(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_#"+ 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.

```

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.

```
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*.

```
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.

5.8 Esperimenti e analisi dei risultati

Abbiamo condotto un primo esperimento per valutare la bontà del modello BasicJVE. Intendiamo verificare, prima di tutto, la coerenza del sistema di business di BasicNet con la nostra ricostruzione virtuale all'interno di JVE.

In seguito condurremo altri esperimenti apportando delle modifiche alle variabili del modello, per poter effettuare un'analisi comparata di diversi scenari¹³. L'immagine 5.3 riporta un esempio di ciò che può essere osservato in fase di simulazione.

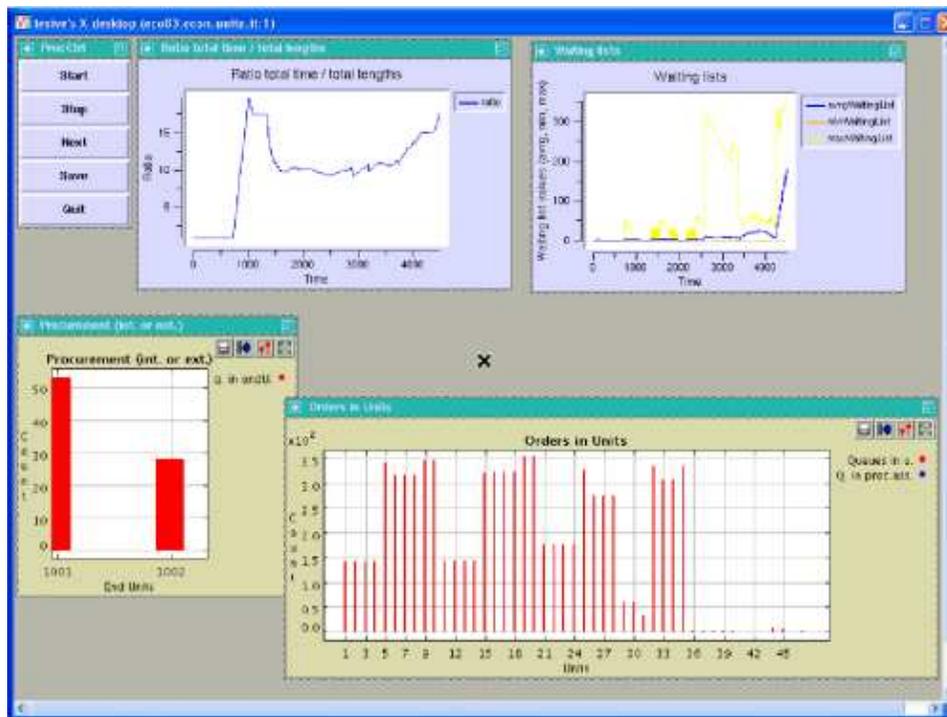


Fig. 5.3. La simulazione BasicJVE in corso

In questa immagine vengono 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:

¹³ Per la descrizione di questi esperimenti si rimanda al sito <http://eco83.econ.unito.it/tesive/basicnet> (user: tesi - password: tangram).

- 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) 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.

5.8.1 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 5.4 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

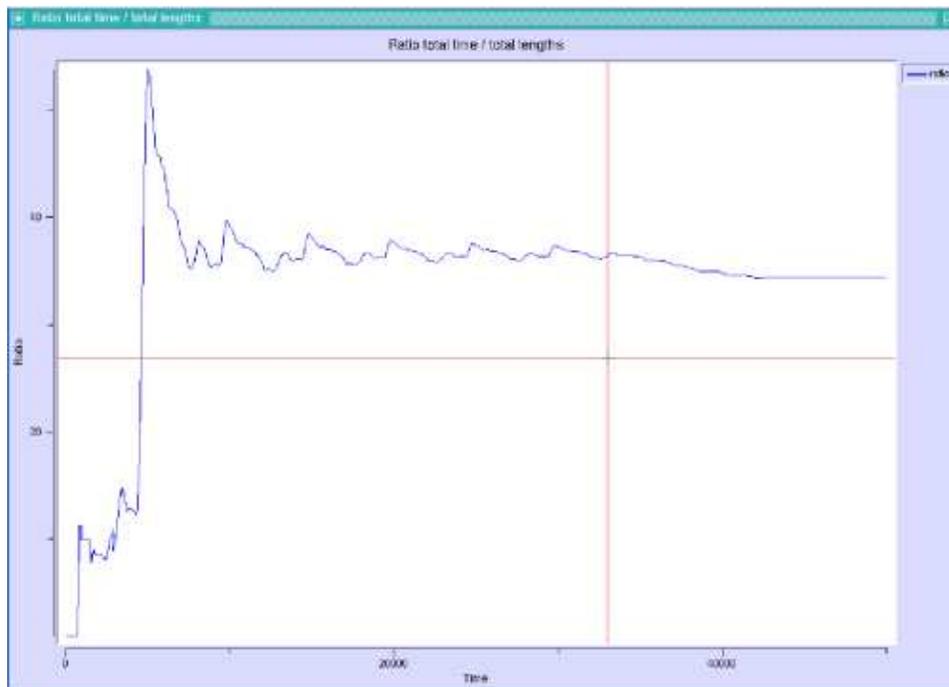


Fig. 5.4. Rapporto tra tempi della ricetta e tempi di produzione

un miglioramento delle performance dell'azienda, che in questo esperimento non appaiono particolarmente buone.

Il grafico 5.5 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.

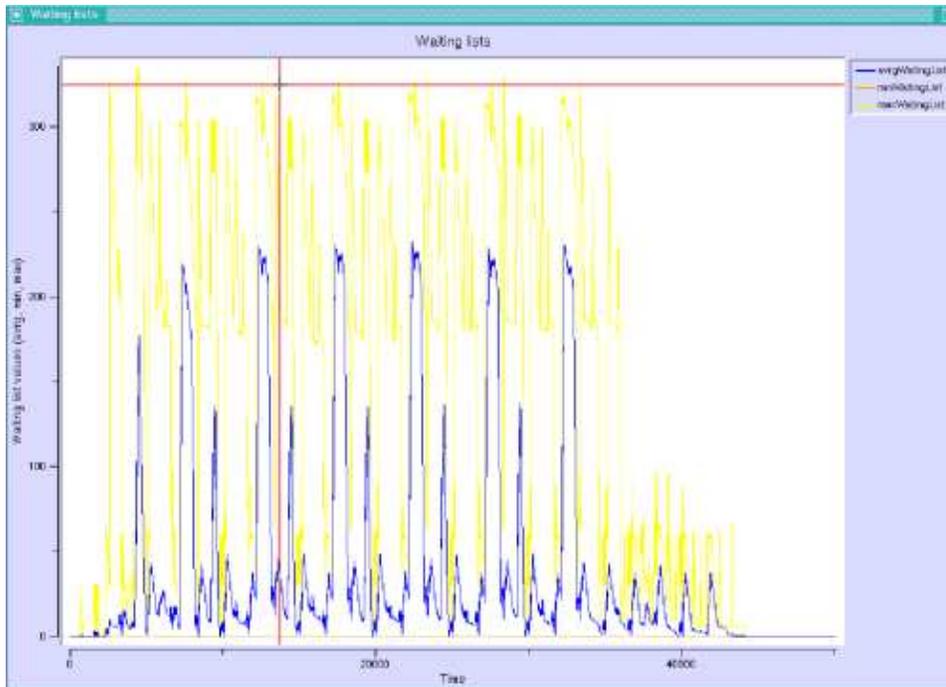


Fig. 5.5. Tempi di attesa minimi, massimi e medi degli ordini

Oltre ad un'analisi dei grafici che JVE 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 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).

BIBLIOGRAFIA

- <http://www.basicbiddings.com>. (2002). *Basicbiddings*.
- <http://www.basicfactory.com>. (2002). *Basicfactory*.
- <http://www.basicforecast.com>. (2002). *Basicforecast*.
- <http://www.basicmarketing.com>. (2002). *Basicmarketing*.
- <http://www.basicnet.com>. (2002). *Basicnet*.
- <http://www.basicpress.com>. (2002). *Basicpress*.
- <http://www.basicssample.com>. (2002). *Basicssample*.
- <http://www.basicsspecs.com>. (2002). *Basicsspecs*.
- <http://www.basictrademark.com>. (2002). *Basictrademark*.
- Lamieri, M., & Merlo, F. (2002). <http://eco83.econ.unito.it/tesive/basicnet>. (BasicJVE - Analisi della BasicNet in prospettiva di formalizzazione per il modello JVEFrame (user: tesi - password: tangram))
- Terna, P. (2002). Simulazione ad agenti in contesti di impresa. *Sistemi intelligenti*, XVI(1), pp.33-51.

BIBLIOGRAFIA

Capitolo 6

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.

Nella biologia per esempio una simulazione al computer è stata utilizzata per studiare la forma delle proteine (<http://folding.stanford.edu>); grazie a complicatissimi calcoli è stato possibile ricostruire la forma delle proteine, che non sono nient'altro che catene di amminoacidi, come effetto delle forze di attrazione e repulsione a livello intramolecolare. La simulazione ha spiegato una forma complessa sulla base di regole elementari.

Nelle scienze sociali la simulazione dovrebbe essere ancora più utile perché permette di studiare le interazioni tra gli esseri umani. Nel campo dell'economia si sono ottenuti ottimi risultati, per esempio con modelli del mercato borsistico (Terna, 2000) oppure simulazioni che hanno spiegato la nascita dei distretti industriali (Boero, Castellani, & Squazzoni, 2002). Da questi esperimenti è emerso chiaramente che fenomeni considerati complicati e difficili da spiegare, come la formazione di bolle inflazionistiche, sono semplicemente il frutto dell'interazione di agenti semplicissimi (le bolle inflazionistiche, ad esempio, si presentano addirittura con agenti che decidono in modo casuale).

6.1 Considerazioni sulla simulazione *BasicJVE*

Grazie al lavoro appena concluso, Francesco Merlo ed io, abbiamo avuto l'opportunità di sperimentare la costruzione di un'impresa virtuale partendo da dati reali. All'inizio

del nostro lavoro ci siamo domandati se questo fosse possibile, ora che il progetto ha preso forma sappiamo che non solo è possibile, ma ha anche una grande utilità pratica: consente di ottenere una rappresentazione esplicita dei processi aziendali, che sarebbe molto difficoltosa con modelli letterari.

Durante la formalizzazione per applicazione del modello JVE alla realtà di Basic-Net 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.

Per quanto riguarda il meccanismo di funzionamento dell'azienda da noi ri-costruito 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.

Il procedimento seguito nella costruzione del modello viene definito 'bottom-up'. La nostra azienda virtuale, come tutte le simulazioni agent-based nelle quali si vogliono osservare fenomeni emergenti, è costruita partendo 'dal basso' cioè dalla descrizione delle singole unità produttive e le loro interazioni, per giungere solo dopo, attraverso la simulazione, ad osservare l'azienda nel suo insieme. Questo procedimento è formalmente rigoroso e offre la possibilità di comprendere i meccanismi e le relazioni che si instaurano nell'azienda. Osservando l'azienda nel suo insieme, come viene fatto dai modelli tradizionali, è difficile cogliere le interazioni che nascono tra le varie divisioni. La maggior difficoltà che abbiamo incontrato è stata stimare i particolari operativi essenziali per costruire la simulazione, partendo da modelli letterario-descrittivi costruiti 'dall'alto'.

I dati generati dalla simulazione sul carico di lavoro delle unità sono stati uno spunto di riflessione interessante. Apparentemente le unità sembra non riescano a sostenere il carico di lavoro richiesto. Questa situazione è conseguenza dell'emergere di un fenomeno imprevisto: il passaggio, attraverso la simulazione, dalla conoscenza fattuale, fondata sul saper fare, alla rappresentazione formale, non è banale, e richiede un'attenta rielaborazione

dei dati. Nel modello ‘letterario’ dell’azienda abbiamo riscontrato, ad esempio, passaggi logici che non potevano essere tradotti direttamente in un modello di simulazione e che, quindi, dovevano essere indagati sotto un diverso punto di vista. I tempi necessari per realizzare una collezione sono la chiave per il realismo del modello. Per rendere coerente il carico di lavoro e i tempi di produzione, e quindi rendere la simulazione sempre più aderente a quello che succede nella realtà, si potrebbe pensare di indagare in modo più approfondito sulle singole divisioni. Potrebbe essere utile, ad esempio, chiedere ai disegnatori i tempi necessari a realizzare i singoli MetaSample, oppure verificare la coerenza tra le previsioni dei licenziatari e gli ordini effettivi.

In questo periodo ci siamo resi conto che il lavoro di ri-costruzione virtuale è innanzi tutto utile per poter paragonare due modi differenti di vedere la realtà. Il continuo confronto tra il modello letterario e quello simulato può portare dei miglioramenti ad entrambi. Ci auguriamo quindi che la nostra prima fase di indagine sia utile anche alla BasicNet, per poter migliorare la loro conoscenza e rappresentazione dell’azienda.

L’obiettivo che ci siamo posti è di poter effettuare un confronto tra la Basic Net reale ed un’azienda simulata simile con alcuni processi internalizzati, come la gestione della produzione o della vendita dei prodotti. E’ stato quindi necessario ricostruire l’intero processo, dal disegno dei campionari fino alla vendita dei singoli articoli. Riteniamo che il pregio del nostro lavoro sia stato aver creato un ‘frame’ generale che descrive la BasicNet nel suo insieme e che, grazie alla sua generalità, potrà essere utile per futuri approfondimenti.

6.2 *Possibili sviluppi*

Uno dei possibili sviluppi di JVE potrebbe essere implementare la simulazione del sistema informativo. Un ampliamento del modello in questa direzione sarebbe molto utile alla simulazione BasicJVE, proprio perché non si tratta di simulare un processo produttivo, ma un processo organizzativo dove l’informazione è l’elemento essenziale. Poter simulare il sistema informativo consentirebbe di confrontare l’immagine che l’azienda ha di se stessa, con quello che realmente accade.

Una direzione di sviluppo del progetto BasicJVE potrebbe essere mettere la lente d'ingrandimento sulla divisione BasicSamples, che reputiamo il punto chiave di tutta l'azienda. L'unico elemento che viene concretamente prodotto da BasicNet sono 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 BasicSamples è il maggior centro di costo, riteniamo che meriterebbe un'analisi più approfondita. Attualmente la divisione BasicSamples viene rappresentata da una singola unità produttiva di JVE; l'insieme di interrelazioni che si sviluppano al suo interno sono quindi al momento ignorate. Proponiamo, per eventuali sviluppi futuri, di analizzare questa divisione come se fosse un'azienda indipendente.

I prossimi studi su BasicJVE 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. Negli esperimenti svolti sinora abbiamo utilizzato dei costi stimati, ma comunque utili per effettuare delle prime considerazioni. Se si riuscirà ad indagare più a fondo i costi di ogni fase produttiva sarà possibile, attraverso la simulazione, creare diversi scenari ipotetici. Si potrebbe immaginare che BasicNet gestisca direttamente la produzione, oppure che il maggiore licenziatario, Kappa Italia, diventi divisione vendite di BasicNet. Si potrebbe, infine, pensare ad un modello virtuale di azienda completamente esternalizzata, nella quale anche i MetaSample vengano fatti produrre all'esterno, riducendo al minimo i rischi d'impresa.

BIBLIOGRAFIA

Boero, R., Castellani, M., & Squazzoni, F. (2002). *La distrettualizzazione delle imprese come processo cognitivo; simulazione di un sistema distrettuale tramite un prototipo computazionale basato su agenti; paper per il workshop su scienze cognitive ed economia organizzato dalla associazione italiana di scienze cognitive, rovereto, 21 settembre.*

<http://folding.stanford.edu>. (2002). *Folding distributed computing project homepage.*

Terna, P. (2000). *Sum: a surprising (un)realistic market: Building a simple stock market structure with swarm, presentato a cef 2000, barcelona, 5-8 giugno.*

BIBLIOGRAFIA

Capitolo 7

APPENDICE

7.1 Listati

7.1.1 *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 computatiiona 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 MemoryMatrixes
    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;

    /**
     * the constructor for ComputationalAssembler
     */

    public ComputationalAssembler (Zone aZone)
    {
        // Call the constructor for the parent class.
    }
}
```

```
super(aZone);

normal = new NormalDistImpl (getZone());
}

//*****
// BasicJVE 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();

// Makeing 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
}
// 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.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
        +
        " cannot be produced, forecast=" + sum + " status=0");
}

//Waiting all predictions
for (int t=1 ; t<=35 ; t++)
{
    MemoryMatrix agentMatrix =
        (MemoryMatrix) pendingComputationalSpecificationSet.
        getMemoryMatrixAddress(t);

    if(agentMatrix.getEmpty(layer,1,j))
        endForecast = false;

}

if (endForecast)
{
    System.out.println("MetaSample#" + j + " with layer#" + layer
        +
        " forecasted by all licensees");

    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);

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);
    basicNetMatrix.setValue(layer,2,i,2);

    done=true;
}
}
}
// 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.isEmpty(layer, 0, 0))
{
    int counter = (int) designMatrix.getValue(layer, 0, 0);

    for (int i=1; i<=counter; i++)
    {
        if (!basicNetMatrix.isEmpty(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);
}

```

```
        + 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.isEmpty(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
}
// 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))
    {
        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
}
}
}
// 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.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 &&
            !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 -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);

  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;
}
//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() + " | | " + itemSales + " | \
    n";
logOpen(false);
try
{
    grossSalesLog.write(text);
    grossSalesLog.flush();
}
catch (IOException e)
{
    System.out.println(e);
    MyExit.exit(1);
}

done =true;
}
//end c1912

public void c1910()
{
    checkMatrixes(1910,1);
    System.out.println("***** | THIS | IS | THE | END | *****");
}
```

```

    StartVEFrame.vEFrameObserverSwarm.getControlPanel().setStateStopped
        ();
}

//*****
// Private functions
//*****

//The counter is used in some computationl 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.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;
}

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)
{
    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);
}
}
}
```

7.1.2 OrderDistiller.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{

    /** INSTANCE VARIABLES
    *A flag to check if the orderSequences worksheet file is open
    */
    public boolean worksheetOrderSequenceFileOpen = false ,
        worksheetRecipeFileOpen = false ;

    /**The arrays containing: the sequence of orders to be done in the
        related shift ,
    * the respective quantity and layer
    */

    //lm
    //*****
    // Added third array containing layers info
    //*****
    int [] orderSequence1 , orderSequence2 , orderSequence3 ;

    /**The variable referring to the orderSequences worksheet file
    */
    ExcelReader orderSequenceWorksheet = null , recipeWorksheet = null ;
    /** The name of the recipe
    */
    String recipeName ;

    /**Flags to operate checks while reading
    */

```

```

//lm
//*****
// Added String "layer" and "computation"
//*****
public String semicolon = ";", checkTheCell, gate = "#", p = "p",
    sec = "s", min = "m", end = "e", slash = "/", backslash = "\\",
    or="|" , layer="l", computation="c";

/**An object to record the array containing the steps of the
    recipes in a List
    */
Recipe aRecipe;

/**used to record the number of generated orders
    */
public int orderCount=0;

//lm
//*****
//used to decide if read "orderSequencesModified.xls"(true) or
// "orderSequences.xls"(false) using readOrderSequence method.
//It is useful for compatibility with Vir application.
//*****
public static boolean firstTime=true;

/**a specific order
    */
public Order anOrder;

/**the list containig the operating units
    */

```

```
public ListImpl unitList;

    /** the list containig the end units
     */
    public ListImpl endUnitList;

    /** the list containig all the orders
     */
    public ListImpl orderList;

    /** the list containig all the orders
     */
    public ListImpl recipeList;

    /**The array containing the steps of the recipe and the steps of
     procurement
     */
    int[] orderRecipe;

    // units
    Unit aUnit;
    EndUnit anEndUnit;

    /* public boolean unitNotFound;*/ //pt
    int j, row, length, choice;

//lm
//*****
// Added variable for current layer
//*****

int currentLayer = 0;
```

```
//lm
//*****
// Added variable for orderSequenceX dimension
//*****

int maxOrderSequence;

    AssigningTool assigningTool; //pt

// CONSTRUCTOR
public OrderDistiller (Zone aZone, int msn, int msl, ListImpl ul,
    ListImpl eul, ListImpl ol, int tln,
    VEFrameModelSwarm mo, AssigningTool at){ //pt

super(aZone, msn, msl, ul, eul, ol, tln, mo, at);

unitList=ul;
endUnitList=eul;
    orderList=ol;
assigningTool=at; //pt

}

/**This method is used to collect the names of the units and of the
    end units, so that it can operate the check of
    * corrispondency between the production phases required by recipes
    and the phases of production the units can do.
```

```

    */
    public void setDictionary(){

        super.setDictionary();

        /**
         * Added "orderSequence3" array containing layers info //lm
         */

        maxOrderSequence = dictionaryLength * totalLayerNumber;

        /** It will contain the ID codes of the recipes worksheet file
         * orderSequence1 = new int [maxOrderSequence];
         * // It will contain the quantity of each recipe to be done
         * orderSequence2 = new int [maxOrderSequence];
         * // It will contain the layer of each recipe to be done
         * orderSequence3 = new int [maxOrderSequence];

         * setRecipeContainers(); // See Below
         * readRecipes(); // See Below
         */

        /** This method is used to set the length of each recipe
         */

        public void setRecipeContainers(){

            recipeList = newListImpl(getZone());

            if(!worksheetRecipeFileOpen){
                recipeWorksheet = new ExcelReader("recipeData/recipes.xls");
                worksheetRecipeFileOpen = true;
            }

```

```
checkTheCell = recipeWorksheet.getStrValue();

row = 0;
while(! recipeWorksheet.eof()){
    checkForComments(checkTheCell);
    calculateLength(checkTheCell);
}

worksheetRecipeFileOpen = false;
}

/** This is the method needed to read and store the recipes. The
    procedure follows this steps:<br/> It opens the worksheet file
    recipes.xls, in which are stored the sequences of steps of all
    the recipes;<br/>It makes some check of the routines; <br/>It
    search the object containig the same code of the recipe we are
    considering;<br/>
    * Finally it substitutes the values of the array of that object
    with the new ones.
    */
public void readRecipes(){
//local variables
int code = 0, i;
boolean recipeCodeNotFound;

if(! worksheetRecipeFileOpen){
    recipeWorksheet = new ExcelReader("recipeData/recipes.xls");
    worksheetRecipeFileOpen=true;
}
row=0;
checkTheCell=recipeWorksheet.getStrValue();
while(!recipeWorksheet.eof()){
```

```
        checkForComments(checkTheCell);
        code = errorIsNotAnInteger(recipeWorksheet);

        recipeCodeNotFound=true;
        for (i = 0; i < recipeList.getCount() && recipeCodeNotFound; i++)
        {
            aRecipe = (Recipe) recipeList.atOffset(i);
            if(aRecipe.getCodeNumber() == code)recipeCodeNotFound = false;
        }

        aRecipe.setSteps(checkTheCell, recipeWorksheet);
        checkTheCell = aRecipe.getCheckTheCell();
    }
}

/**This method is used to calculate the length of each row after a
    strong check of the elements
    */
public void calculateLength(String cTC){
int code = 0;
checkTheCell = cTC;
recipeName = checkTheCell;
code = errorIsNotAnInteger(recipeWorksheet);
row++;
length = 0;
checkTheCell = recipeWorksheet.getStrValue();

while(! checkTheCell.equals(semicolon)){
    if(StartVEFrame.verbose)
        System.out.println("CheckTheCell contains "+checkTheCell);
```

```
        if(checkTheCell.equals(p))          choice = 1;
        if(checkTheCell.equals(or))        choice = 2;
        if(checkTheCell.equals(end))        choice = 3;
        if(checkTheCell.equals(computation)) choice = 4;
        if(recipeWorksheet.checkForLabelCell() && !checkTheCell.equals(
            semicolon)) choice = 5;

        switch(choice){
        case 1:
            if(StartVEFrame.verbose)
                System.out.println("OrderDistiller: the choice is for a Procurement"
                    );
            procurement(recipeWorksheet);
            break;
        case 2:
            if(StartVEFrame.verbose)
                System.out.println("OrderDistiller: the choice is for an Or ");
            oR(recipeWorksheet);
            break;
        case 3:
            if(StartVEFrame.verbose)
                System.out.println("OrderDistiller: the choice is for an End ");
            end(recipeWorksheet);
            break;
        case 4:
            if(StartVEFrame.verbose)
                System.out.println("OrderDistiller: the choice is for a Computation
                ");
            computation(recipeWorksheet);
            break;
        case 5:
            if(StartVEFrame.verbose)
```

```

    System.out.println("OrderDistiller: the choice is for a Number");
        number(recipeWorksheet);
        break;
        default:
        if(StartVEFrame.verbose)
            System.out.println("OrderDistiller: no matches were found reading the
                worksheet, check for errors inside it");
    System.exit(1);

}
}

if(!recipeWorksheet.eof())
    checkTheCell=recipeWorksheet.getStrValue();

aRecipe=newRecipe(getZone(),code,length,recipeName);
if(StartVEFrame.verbose)
    System.out.println("OrderDistiller: a Recipe named "+recipeName+
        " with code "+code+"was born");
recipeList.addLast(aRecipe);
}

//lm
//*****
//A new method for the computational step
//*****

/**This method dial with the computational choice
*/

```

```
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();

    }

    /**This method dial with the procurement choice
    */
    public void procurement(ExcelReader e){
    int numberOfStepsForProcurement = 0;

    numberOfStepsForProcurement = e.getIntValue();
    length += (2 + numberOfStepsForProcurement);
    for(int h = 0; h < numberOfStepsForProcurement; h++) checkTheCell = e.
        getStrValue();
    checkTheCell = e.getStrValue();

    }

    /**This method dial with the or choice
    */
```

```
public void oR(ExcelReader e){

length +=2;
e.getStrValue();
checkTheCell = e.getStrValue();

}

/**This method dial with the end choice
*/
public void end(ExcelReader e){

length++;
e.getStrValue();
checkTheCell = e.getStrValue();

}

/**This method dial with normal or batch choice
*/
public void number(ExcelReader e){

checkTheCell = e.getStrValue();
if(checkTheCell.equals(sec)) second(e);
else if(checkTheCell.equals(min))minute(e);
else{
    if(StartVEFrame.verbose)
        System.out.println("Time is not expressed in minutes or seconds.
        Check the worksheet");
}
}

}
```

```
    public void second(ExcelReader e){
int numberOfSteps = 0;

numberOfSteps = e.getIntValue();
if(numberOfSteps == 0){length ++;
checkTheCell = e.getStrValue();}
else{
    checkTheCell = e.getStrValue();

    if(checkTheCell.equals(slash) || checkTheCell.equals(backslash)){

length +=4;
e.getStrValue();
checkTheCell = e.getStrValue();

    }

    else {
length += numberOfSteps;
    }

}

}

public void minute(ExcelReader e){
    int numberOfSteps = 0;

    numberOfSteps = (e.getIntValue() * 60);
if(numberOfSteps == 0){length ++;
checkTheCell = e.getStrValue();}
else{
```

```
checkTheCell = e.getStrValue();

if(checkTheCell.equals(slash) || checkTheCell.equals(backslash)){

    length +=4;
    e.getStrValue();
    checkTheCell = e.getStrValue();

}

else {
    length += numberOfSteps;
}

}

}

/**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(){
//local counters
int i, ii, iii, k, code, quantity, layerNumber;
boolean recipeCodeNotFound;
i=0;

readOrderSequence(); //See below
```

```

while(i < maxOrderSequence && orderSequence1[i] != 0){
    code = getOrderSequence1(i);
    quantity = getOrderSequence2(i);
    layerNumber = getOrderSequence3(i);

    recipeCodeNotFound = true;
    for (ii = 0; ii < recipeList.getCount() && recipeCodeNotFound; ii
        ++)
    {
        aRecipe = (Recipe) recipeList.atOffset(ii);
        if(aRecipe.getCodeNumber() == code)recipeCodeNotFound = false;
    }
    for(k = 0; k < quantity; k++){

orderCount++;
// creating an order
anOrder = new Order(getZone(), orderCount,
    Globals.env.getCurrentTime(),
    aRecipe.getLength(), aRecipe.getOrderRecipe(),
    vEFrameModelSwarm, endUnitList);

anOrder.setRecipeName(aRecipe.getRecipeName());

//lm
//*****
// setting the layer (from 0 to totalLayerNumber-1)
//*****

anOrder.setOrderLayer(layerNumber);
if(StartVEFrame.verbose)
System.out.println("Order #" + anOrder.getOrderNumber() +
    "" is generated with Name: " + anOrder.getRecipeName() +
    "" and layerNumber #" + anOrder.getOrderLayer());

```

```
// 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 VE)
 */
assigningTool.assign(anOrder);

    }

    i++;
}

}

/**This method reads from the worksheet containing, shift by shift
    , the sequence of orders to be launched and fills
 * in the orderSequence1 with the ID codes of recipes and the
    orderSequence2 with the quantities of each recipe.
 */

public void readOrderSequence(){
//local variables

// lm
```

```

//*****
// boolean firstTime has been moved to global variables
//*****
int i, numberOfShift;

if(StartVEFrame.verbose)
System.out.println("firstTime is "+firstTime);

i=0;

if(!worksheetOrderSequenceFileOpen&&firstTime){
    orderSequenceWorksheet=newExcelReader("recipeData/
        orderSequencesModified.xls");
    worksheetOrderSequenceFileOpen=true;
    firstTime=false;
    if(StartVEFrame.verbose)
    System.out.println("orderSequencesModified.xls has been open");
}

elseif(!worksheetOrderSequenceFileOpen){
    orderSequenceWorksheet=newExcelReader("recipeData/
        orderSequences.xls");
    worksheetOrderSequenceFileOpen=true;
    //lm
    if(StartVEFrame.verbose)
    System.out.println("orderSequences.xls has been open");
}

//lm
//*****
//Read the shift number, added some prints for debugging purpose

```

```

_//*****

numberOfShift = orderSequenceWorksheet.getIntValue();
if(StartVEFrame.verbose)
    System.out.println("The shift #" + numberOfShift + " has begun ");

_//lm
_//*****
_//_A_new_bug._Zeroing_orderSequencesX_vectors
_//*****
_for(_int_ii=_0;_ii<_maxOrderSequence;_ii++){
    _orderSequence1[ii]=_0;
    _orderSequence2[ii]=_0;
    _orderSequence3[ii]=_0;

_}
_while(!_orderSequenceWorksheet.eol()){

    _//lm
    _//*****
    _//_Our_new_method_used_to_check_if_there_is_a_new_layer
    _//*****
    _checkForLayer(orderSequenceWorksheet);

    _orderSequence1[i]=_errorIsNotAnInteger(orderSequenceWorksheet);

    _orderSequence3[i]=_currentLayer;

    _errorIsNotAString(orderSequenceWorksheet);
    _orderSequenceWorksheet.getStrValue();

```

```

orderSequence2[i] = errorIsNotAnInteger(orderSequenceWorksheet);

//lm
//*****
// Some prints for debugging purpose
//*****
if(StartVEFrame.verbose)
System.out.println("The recipe #"+orderSequence1[i]+
    " with
    quantity "+orderSequence2[i]
    + " and Layer "+orderSequence3[i]+
    " is starting production ");

i++;
}

errorIsNotAString(orderSequenceWorksheet);
orderSequenceWorksheet.getStrValue();

if(orderSequenceWorksheet.eof()){
    //System.out.println("The simulation finishes here. It is not
    the right and normal way, but is only for explanation. Thank you
    for your attention and interest.");
    //System.exit(1);
    worksheetOrderSequenceFileOpen=false;
}
}

/**This method is used to check the corrispondence with the dictionary
of production phases
*/

```

```
public int checkTheExistence(int c){
    int i, check;
    boolean dictionaryVoice = false;

    check = c;

    for(i = 0; i < dictionaryLength; i++){
        if(check == dictionary[i])dictionaryVoice = true;
    }

    if(check > 1000000000){
        System.out.println("The value found is not a valid number because
            it is greater than a billion");
        System.exit(1);
    }

    if(dictionaryVoice == false){
        System.out.println("The value found is not a valid number of
            production");
        System.exit(1);
    }

    return check;
}

//lm
//*****
//Our new method
//*****

/**This method is used to check the presence of a new layer
**/
public void checkForLayer(ExcelReader e){
```



```
    }  
    }  
  
    /**This method is used to check if a type error occurs  
    */  
    public int errorIsNotAnInteger(ExcelReader e){  
  
        if(e.checkForLabelCell())  
        {  
            System.out.println("The cell should contain an Integer Value");  
            System.exit(1);  
        }  
        return e.getIntValue();  
    }  
  
    /**This method is used to check if a type error occurs  
    */  
    public void errorIsNotAString(ExcelReader e){  
  
        if(e.checkForLabelCell());  
        else {  
            System.out.println("The cell should contain a String Value");  
            System.exit(1);  
        }  
    }  
  
    /**This method is used to obtain the elements of the orderSequence1  
    array  
    */  
    public int getOrderSequence1(int j){
```

```
    return orderSequence1[j];

}

/**This method is used to obtain the elements of the orderSequence2
    array
    */
public int getOrderSequence2(int j){

return orderSequence2[j];

}

/**This method is used to obtain the elements of the orderSequence3
    array
    */
public int getOrderSequence3(int j){

return orderSequence3[j];

}

} // OrderDistiller
```

7.1.3 *Recipe.java*

```
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

/**
 * Recipe.java
 *
 *
 * Created: Wed May 13 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 record the recipes and their referring number
    (ID), so we can assign them to a List*/
public class Recipe extends SwarmObjectImpl{
    //INSTANCE VARIABLES
    /**the array containing the recipe*/
    int [] orderRecipe;

    /**the referring number of the recipe in the worksheet file*/
    int code;
```

```

    /**The length of the array
     */
    int length;

    /**Flags to operate checks while reading
     */
    public String semicolon = ";", checkTheCell, gate = "#", p = "p",
        end = "e",
        sec = "s", min = "m", slash = "/", backslash = "\\","or="|" |",
        computation="c";
    /**The name of the recipe
     */
    String recipeName;

    /**the dictionary to be used to choose the steps to be included in
     a recipe
     */
    int [] dictionary;

    /**length of the dictionary*/
    int dictionaryLength;

    OrderDistiller orderDistiller;

    int j, choice, step;
    //CONSTRUCTOR
    public Recipe (Zone aZone, int c, int l, String rN){

        super(aZone);

        code=c;
        orderRecipe=new int [l];
        for(int j=0; j<l; j++) orderRecipe[j]=0;

```

```
    length = 1;
    recipeName = rN;
}

    public int getCodeNumber(){

return code;

}

    public int[] getOrderRecipe(){

return orderRecipe;

}

    public int getLength(){

return length;

}

    public String getRecipeName(){

return recipeName;

}

    public void setSteps(String cTC, ExcelReader recipeWorksheet){

checkTheCell = cTC;
```

```
checkTheCell = recipeWorksheet.getStrValue();
j = 0;
if (StartVEFrame.verbose)
System.out.println(checkTheCell + " The length of the recipe °n " +
    getCodeNumber() + " is " + getLength());

while (!checkTheCell.equals(semicolon)) {
    if (checkTheCell.equals(p)) choice = 1;
    if (checkTheCell.equals(or)) choice = 2;
    if (checkTheCell.equals(end)) choice = 3;
    if (checkTheCell.equals(computation)) choice = 4;
    if (recipeWorksheet.checkForLabelCell() && !checkTheCell.equals(
        semicolon)) choice = 5;

    switch (choice) {
        case 1:
            if (StartVEFrame.verbose)
                System.out.println("Recipe: the choice is for a Procurement");
            procurement(recipeWorksheet);
            break;
        case 2:
            if (StartVEFrame.verbose)
                System.out.println("Recipe: the choice is for an Or ");
            oR(recipeWorksheet);
            break;
        case 3:
            if (StartVEFrame.verbose)
                System.out.println("Recipe: the choice is for an End ");
            end(recipeWorksheet);
            break;
        case 4:
            if (StartVEFrame.verbose)
                System.out.println("Recipe: the choice is for a Computation ");
```

```
        computation(recipeWorksheet);
        break;
        case 5:
            if(StartVEFrame.verbose)
                System.out.println("Recipe: the choice is for a Number");
        number(recipeWorksheet);
        break;
        default:
            if(StartVEFrame.verbose)
                System.out.println("Recipe: no matches were found reading the
                worksheet, check for errors inside it");
        System.exit(1);

    }
}

for(int h=0; h<length; h++)
    if(StartVEFrame.verbose)
        System.out.println("The recipe contains "+orderRecipe[h]+" in
        position "+h);
    if(!recipeWorksheet.eof())
        checkTheCell=recipeWorksheet.getStrValue();

}

public String getCheckTheCell(){

    return checkTheCell;

}

/**This method is used to check if a type error occur
```

```
        */
        public int errorIsNotAnInteger(ExcelReader e){

            if(e.checkForLabelCell())
                {
                    if(StartVEFrame.verbose)
                        System.out.println("The cell should contain an Integer Value");
                }
            System.exit(1);
        }
        return e.getIntValue();
    }

    /**This method is used to check if a type error occur
    */
    public void errorIsNotAString(ExcelReader e){

        if(e.checkForLabelCell());
        {
            if(StartVEFrame.verbose)
                System.out.println("The cell should contain a String Value");
            System.exit(1);
        }
    }

    /**This method deal with the computation choice
    */
    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");
}

orderRecipe[orderRecipe.length-1] = computationCode;

orderRecipe[orderRecipe.length] = numberMatrixesForComputation;
```

```
for(int h=0; h<numberMatrixesForComputation; h++)
    orderRecipe[++j] = matrixesForComputation[h];

orderRecipe[++j] = 1000000000 + step;

j++;

}

/**This method dial with the procurement choice
 */
public void procurement(ExcelReader e){
int numberOfStepsForProcurement = 0;

orderRecipe[j] = -1;
numberOfStepsForProcurement = e.getIntValue();
orderRecipe[++j] = numberOfStepsForProcurement;
for(int h = 0; h < numberOfStepsForProcurement; h++)
    orderRecipe[++j] = e.getIntValue();
j++;
checkTheCell = e.getStrValue();

}

/**This method dial with the or choice
 */
public void oR(ExcelReader e){

orderRecipe[j] = -10;
orderRecipe[++j] = e.getIntValue();
j++;
checkTheCell = e.getStrValue();
```

```
    }

    /**This method dial with the end choice
     */
    public void end(ExcelReader e){

orderRecipe[j] = e.getIntValue();
checkTheCell = e.getStrValue();

    }

    /**This method dial with normal or batch choice
     */
    public void number(ExcelReader e){

step = Integer.parseInt(checkTheCell);
checkTheCell = e.getStrValue();
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);
}
j++;
}

public void second(ExcelReader e, int step){
    int numberOfSteps = 0;

    numberOfSteps = e.getIntValue();
```

```
if(numberOfSteps==0){
    checkTheCell = e.getStrValue();
    orderRecipe[j] = 1000000000 + step;
}
else {
    checkTheCell = e.getStrValue();

    if(checkTheCell.equals(slash) || checkTheCell.equals(backslash)){

if(checkTheCell.equals(slash)){
    orderRecipe[j] = -2;
    orderRecipe[++j] = numberOfSteps;
    orderRecipe[++j] = e.getIntValue();
    orderRecipe[++j] = step;
}

else if(checkTheCell.equals(backslash)){
    orderRecipe[j] = -3;
    orderRecipe[++j] = numberOfSteps;
    orderRecipe[++j] = e.getIntValue();
    orderRecipe[++j] = step;
}
checkTheCell = e.getStrValue();
}

else {
orderRecipe[j] = step;
for(int jj = 0; jj < numberOfSteps - 1; jj++)
    orderRecipe[++j] = step;
}

}
}
```

```
public void minute(ExcelReader e, int step){
    int numberOfSteps = 0;

    numberOfSteps = e.getIntValue();
    numberOfSteps = numberOfSteps * 60;
    if(numberOfSteps == 0){
checkTheCell = e.getStrValue();
orderRecipe[j] = 1000000000 + step;
    }
    else {
checkTheCell = e.getStrValue();

if(checkTheCell.equals(slash) || checkTheCell.equals(backslash)){

    if(checkTheCell.equals(slash)){
orderRecipe[j] = -2;
orderRecipe[++j] = numberOfSteps;
orderRecipe[++j] = e.getIntValue();
orderRecipe[++j] = step;
    }

    else if(checkTheCell.equals(backslash)){
orderRecipe[j] = -3;
orderRecipe[++j] = numberOfSteps;
orderRecipe[++j] = e.getIntValue();
    orderRecipe[++j] = step;
    }
    checkTheCell = e.getStrValue();
}

else {
orderRecipe[j] = step;
```

```
for(int jj = 0; jj < numberOfSteps - 1; jj++)
    orderRecipe[++j] = step;
}

}

}

} //Recipe.java
```

