

*Alla mia splendida
famiglia
e al nonno Lorenzo
esempio di coraggio e umanità*

Indice

Sommario	1
Introduzione	3
1 Simulazione e ricerca	7
1.1 Costruire un modello	7
1.1.1 I sistemi complessi	9
1.1.2 La decomposizione di problemi complessi	17
1.2 Logica e strumenti	23
1.2.1 La logica	23
1.2.2 Gli strumenti	24
1.3 Lo schema E/R/A	28
2 La simulazione con Swarm	33
2.1 La programmazione ad oggetti	33
2.1.1 Precedenti linguaggi: la programmazione imperativa e logica	33
2.1.2 La programmazione Object-Oriented e le sue caratteristiche	35
2.1.3 Vantaggi della programmazione ad oggetti	42
2.2 Java	44
2.2.1 L'indipendenza dalla piattaforma	44
2.2.2 Java e l'Internet	46
2.2.3 Java e Objective C	48

2.2.4	Documentazione di Java	49
2.3	Swarm	55
2.3.1	Swarm: struttura base di una simulazione ad agenti	55
2.3.2	Struttura della simulazione in Swarm con <i>Model</i> ed <i>Observer</i> : librerie simtoolsgui e objectbase	57
2.3.3	La struttura con <i>Schedule</i> , <i>Probe</i> e <i>Liste</i> : librerie activity , objectbase e collections	61
3	Modelli di struttura aziendale	69
3.1	Considerazioni introduttive	69
3.2	La struttura aziendale	72
3.2.1	La struttura aziendale: rapporti interni	72
3.2.2	La struttura aziendale: rapporti esterni	73
3.2.3	L' <i>Information Technology</i>	77
3.3	Struttura di alcuni modelli	81
3.3.1	Multi-Agent Information System (<i>MAIS</i>)	81
3.3.2	National Industrial Information Infrastructure Protocols (<i>NIHIP</i>)	84
3.3.3	Process Life Cycle Engineering	88
4	Economia dell'informazione	93
4.1	Il valore dell'informazione	93
4.2	Il <i>lock-in</i> e gli <i>switching costs</i>	96
4.3	Business to Business	100
4.4	Modelli	104
4.4.1	DMarks II	104
5	Il modello di impresa virtuale	111
5.1	L'idea di partenza	111
5.1.1	Lo schema iniziale	112

5.1.2	Lo sviluppo del codice in Swarm	117
5.1.3	Perfezionamenti con la versione 0.3.9	125
5.1.4	I risultati ottenuti	128
5.2	Magazzini e rule master	134
5.2.1	Lo schema	134
5.2.2	Lo sviluppo del codice in Swarm	136
5.2.3	I risultati ottenuti	139
5.3	Ambiente grafico con Java	145
5.3.1	La classe <i>BarChart</i>	145
5.3.2	La classe <i>Histogram</i> di <i>Ptplot</i>	147
5.4	La gestione delle informazioni	155
5.4.1	Informazioni e <i>news</i>	155
5.4.2	Lo sviluppo del codice in Swarm	156
5.4.3	I risultati ottenuti	158
5.5	La contabilità d'impresa	160
5.5.1	La contabilità per <i>order</i> e per <i>unit</i>	160
5.5.2	Lo sviluppo del codice in Swarm	162
5.5.3	I risultati ottenuti	165
6	La simulazione	171
6.1	La prima fase di simulazioni	171
6.2	Perfezionamenti della simulazione	180
6.3	Conclusioni	185
A	Progetto <i>Ptolemy</i>	189
A.1	Modellare e progettare	189
A.1.1	Modelli <i>sincronizzati/reattivi</i>	190
A.1.2	Modelli di eventi discreti	191
A.1.3	Modelli <i>Timed CSP/PN</i>	191

A.2	Conclusioni	191
B	UML: Unified Modeling Language	195
B.1	Caratteristiche base dell'UML	196
B.1.1	Diagrammi di Casi d'Uso	197
B.1.2	Diagrammi di Interazione	198
B.1.3	Diagrammi di Classe	200
B.1.4	Diagrammi di Stato e di Attività	202
B.2	Scrivere in UML	204
B.3	L'impresa virtuale in UML	205
C	Strumenti di simulazione	213
C.1	Descrizione di alcuni ambienti	213
C.1.1	Simul8	213
C.1.2	Extend	215
C.2	Conclusioni	220
D	Listato	225
D.1	Ambienti di sviluppo ed Editor	225
D.1.1	Emacs	225
D.1.2	JBuilder	226
D.1.3	Together	227
D.1.4	Kawa	227
D.2	Il codice	228
D.2.1	StartVEFrame.java	228
D.2.2	VEFrameObserverSwarm.java	230
D.2.3	VEFrameModel.java	247
D.2.4	Unit.java	259
D.2.5	Order.java	280
D.2.6	PTHistogram.java	287

D.2.7 Accounting.data.java 295

Elenco delle figure

1.1	la logica della simulazione	23
1.2	problema di simulazione	25
1.3	simulazione su computer	26
1.4	lo schema E/R/A	28
2.1	esempio di struttura di programmazione imperativa	34
2.2	esempio di struttura di programmazione ad oggetti	35
2.3	incapsulamento	39
2.4	ereditarietà	41
2.5	programmi compilati tradizionalmente	45
2.6	programmi compilati con Java	46
2.7	esempio di link nella documentazione	54
2.8	schema base di una simulazione con Swarm	56
2.9	schema di funzionamento di una simulazione con Swarm	58
2.10	menù di avvio della simulazione in swarm	60
2.11	esempio di <i>probe</i> nell' <i>observer</i>	65
3.1	rappresentazione grafica della value chain	74
3.2	rappresentazione grafica della supply chain	75
3.3	rappresentazione grafica della supply chain network	75
3.4	rappresentazione grafica del Customer Relationship Management	76

3.5	esempio di struttura di agente nel modello MAIS	83
3.6	architettura del modello <i>NIIP</i>	87
4.1	architettura del modello <i>DMark II</i> , P. rappresenta il punto di incontro per le transazioni, D.B. il database e WWW l'internet . .	105
5.1	rappresentazione delle <i>unità produttive</i> espresse con i numeri ro- mani e del <i>componente produttivo</i> con il numero arabo	113
5.2	curve di costo e di insoddisfazione	116
5.3	rappresentazione delle tre matrici di flusso	117
5.4	<i>jveframe.scm</i>	126
5.5	probe dell'observer	128
5.6	probe del model	129
5.7	grafico del caso con 3 unità produttive ed un ordine con un vettore di max 5 cifre	130
5.8	grafico del caso con 3 unità produttive ed un ordine con un vettore di max 10 cifre	131
5.9	grafico del caso con 10 unità produttive ed un ordine con un vettore di max 5 cifre	132
5.10	grafico del caso con 3 unità produttive ed un ordine con un vettore di max 5 cifre	133
5.11	<i>probe</i> del <i>model</i> con le opzioni relative ai magazzini	140
5.12	grafico delle quantità in magazzino con 20 unità produttive, un ordine con un vettore di max 30 cifre ed una capienza massima dei magazzini di 20 pezzi	141
5.13	grafico delle quantità in magazzino con 5 unità produttive, un ordine con un vettore di max 20 cifre ed una capienza massima dei magazzini di 20 pezzi	142
5.14	10 unità produttive, vettore con max 20 cifre ed utilizzo dei ma- gazzini con capacità pari a 10	143

5.15	10 unità produttive, vettore con max 20 cifre, senza magazzini . . .	144
5.16	grafico a barre con 10 unità produttive	147
5.17	grafico a barre con 10 unità produttive, max 30 numeri nel vettore d'ordine e capienza max magazzino di 10 beni	152
5.18	finestra di modifica formato dell'istogramma in figura 5.17	153
5.19	caso senza <i>news</i> al 50° step	158
5.20	caso con <i>news</i> al 50° step	159
6.1	probe relativa al caso di un'impresa sotto dimensionata con valo- rizzazione pari a tre di tutte le fonti di ricavo	172
6.2	grafico delle liste di attesa, impresa con <i>news</i> e <i>warehouses</i>	173
6.3	grafico delle liste di attesa, impresa con <i>warehouses</i>	173
6.4	grafico delle liste di attesa, impresa senza <i>news</i> e <i>warehouses</i>	174
6.5	ratio con <i>news</i> e <i>warehouses</i>	175
6.6	ratio con <i>warehouses</i>	175
6.7	ratio senza <i>news</i> e <i>warehouses</i>	176
6.8	confronto impresa sotto dimensionata con valorizzazione dei ricavi pari a 3	177
6.9	probe relativa al caso di un'impresa sovra dimensionata con valo- rizzazione pari a tre di tutte le fonti di ricavo	178
6.10	confronto impresa sovra dimensionata con valorizzazione dei ricavi pari a 3	179
6.11	probe con le modifiche di <i>minInWarehouses</i> e di <i>inventoryEvalua- tionCriterion</i> , impresa sottodimensionata	181
6.12	ratio con <i>news</i> , <i>warehouses</i> e <i>minInWarehouses</i> pari a 3	181
6.13	grafico delle liste di attesa, impresa con <i>news</i> , <i>warehouses</i> e <i>minI- nWarehouses</i> pari a 3	182
6.14	confronto impresa sotto dimensionata con <i>inventoryEvaluationCri- terion</i> = 2	183

6.15	confronto impresa sotto dimensionata con <i>inventoryEvaluationCriterion</i> = 1	184
6.16	probe con <i>inventoryFinancialRate</i> pari a 5	185
6.17	confronto con <i>inventoryFinancialRate</i> pari a 5	187
B.1	esempio di diagramma di caso d'uso	197
B.2	esempio di diagramma di sequenza	199
B.3	esempio di diagramma di collaborazione	200
B.4	esempio di diagramma di classe	201
B.5	esempio di definizione di una classe	201
B.6	esempio di diagramma di Stato	203
B.7	esempio di diagramma di Attività	204
B.8	percorso di scrittura in UML	205
C.1	esempio di simulazione in Simul8	214
C.2	esempio di orologio in Simul8	215
C.3	esempio di processo per la simulazione di impresa	216
C.4	esempio di <i>Extend</i> per la <i>Sala Simulazione del Giubileo</i>	222

Elenco delle tabelle

1.1	confronto tra modelli tradizionali e complessi	10
2.1	caratteristiche della programmazione ad oggetti	36
3.1	ipotesi di Information Technology	79
3.2	confronto tra <i>supply chain</i> e modello con <i>Swarm</i>	82
4.1	tipologie di lock-in e switching cost a loro associati	98
5.1	calcolo del profitto nell'impresa virtuale	167
6.1	corrispondenza di valori di <i>inventoryEvaluationCriterion</i> con le variabili del modello	180

Sommario

Capitolo 1

Lo studio della complessità tramite l'applicazione di modelli ad agenti per gestire imprevisti ed imprevedibilità. La logica della simulazione e gli strumenti utilizzabili.

Capitolo 2

La programmazione ad oggetti ed i suoi vantaggi nell'applicazione sui modelli ad agenti. Librerie di Java e di Swarm per gestire la struttura e le funzionalità della simulazione.

Capitolo 3

La struttura dell'impresa ed il suo inserimento all'interno di catene del valore, catene di fornitura e di distribuzione; possibili incidenze sulla gestione organizzativa. Presentazione di alcuni modelli sui temi trattati.

Capitolo 4

Il mutato valore dell'informazione: costi, rischi e vantaggi. L'analisi dell'incidenza dell'economia dell'informazione sulla struttura tecnologica ed informativa delle imprese. Esempi e commenti su alcuni modelli.

Capitolo 5

La teoria del modello di impresa virtuale ed alcune indicazioni sulle metodologie seguite per la sua programmazione: sulla base della sequenza di modifiche ed aggiunte apportate. Grafici e codice che illustrano e spiegano il funzionamento.

Capitolo 6

Le simulazioni eseguite sull'ultima versione del modello. Indicazioni sul suo funzionamento e su possibili strade da perseguire per ulteriori miglioramenti.

Appendice A

Descrizione sul progetto *Ptolemy* ed indicazioni sul suo possibile impiego in modelli ad agenti. L'aspetto grafico ed il suo utilizzo nell'impresa virtuale.

Appendice B

Descrizione di uno *standard* per analisi di modelli e la stesura del relativo codice, l'*UML*. Rappresentazione dell'impresa virtuale secondo questa struttura tramite il *reverse engeneering*.

Appendice C

Descrizione di strumenti alternativi a *Swarm* per simulare: differenze ed indicazioni per la scelta del più adeguato alle proprie esigenze.

Appendice D

Le classi *Java* modificate in funzione del lavoro svolto.

Introduzione

L'analisi di fenomeni complessi, intesi quali sistemi dinamici per i quali è difficile giustificare e comprendere il comportamento delle singole parti che li compongono e, pertanto, ancor più prevedere gli sviluppi successivi, può essere oggetto di studio in campo sociale, economico ed anche, nel nostro caso, aziendale. L'impresa è una realtà complessa in quanto ancororché scomposta nelle sue parti costituenti risulta spesso difficile analizzare con chiarezza il perché si verifichino determinati eventi o, meglio, il perché si verifichino eventi imprevisti e imprevedibili.

L'imprevedibilità non è certo gradita da chi si occupa di gestire il sistema aziendale, in quanto può causare errori inaspettati anche se ci si comporta razionalmente. Per ovviare a questo problema si pensa di ricorrere all'utilizzo di un modello che sia in grado di ricostruire la realtà in discussione partendo dalle sue parti elementari fino a giungere alla complessa struttura finale. Il modello così elaborato potrà essere utilizzato per simulare il problema in questione cercando di prevederne gli aspetti nascosti ed indesiderati.

La scelta del modello appropriato è un passo delicato e fonte di molte discussioni; poiché però l'argomento che maggiormente caratterizza i fenomeni complessi è la loro non linearità, si può desumere come i modelli matematici non possano essere applicati, infatti il loro impiego richiederebbe eccessive forzature alle ipotesi di partenza. Più appropriati paiono essere i modelli informatici poiché forniscono strumenti più dinamici e flessibili. Inoltre nell'ambito della modellistica informatica la struttura più appropriata per questo tipo di simulazioni sembra

essere quella basati su agenti, dove si pongono gli agenti quali rappresentanti delle singole parti in cui il problema può essere scomposto ed i legami che tra loro intercorrono quali esempi delle complesse relazioni aziendali.

Gli strumenti informatici utilizzati per la costruzione del modello e per l'esecuzione della simulazione sono stati *Java* ed, integrato con esso, *Swarm*. La scelta di *Java* è stata dettata dal fatto che esso possiede le caratteristiche della programmazione ad oggetti, la quale richiama sia logicamente sia praticamente la struttura di un modello ad agenti. Inoltre, da un po' di tempo, è possibile integrarne l'utilizzo con le librerie di *Swarm* che consentono di eseguire simulazioni basate su agenti con una semplificata gestione del flusso degli eventi. In pratica è possibile riprodurre la sequenza logica delle azioni che devono essere svolte dal modello simulando il trascorrere del tempo.

Su questi basi si sta procedendo alla formulazione dell'impresa virtuale; l'obiettivo è quello di fornire uno strumento in grado di indicare quale sia la strada da percorrere per gestire correttamente un'impresa in base ai mercati in cui opera.

Gli spunti di maggiore interesse per chi opera in azienda, e pertanto anche per il modello, riguardano la struttura che la caratterizza e l'ambiente in cui viene inserita. Questo poiché se si riesce a comprendere a priori come debbano essere impostate le variabili interne dell'impresa, le uniche modificabili, è possibile: inserirsi meglio all'interno della catena del valore del bene cui si indirizza la produzione, conseguenzialmente ridurre i costi di produzione migliorando i profitti ed, infine, accrescere l'efficacia e l'efficienza della catena stessa. Naturalmente sono molte le variabili esterne che influenzano la gestione di cui occorre tener conto, ad esempio è indispensabile considerare dal lato delle risorse il comportamento della catena di fornitura e dal lato delle uscite le richieste dei canali di distribuzione.

Le decisioni in merito all'adeguatezza della struttura si complicano ulteriormente qualora si faccia riferimento all'*information technology*. Infatti, poiché l'informazione è divenuto un bene vero e proprio che può agevolare o danneggiare

L'operato di un'impresa, molta attenzione deve essere rivolta alla struttura tecnologica che si occupa della sua gestione. In particolare, bisogna domandarsi se cambiamenti strutturali dovuti all'inserimento di nuove tecnologie siano o meno necessari per migliorare il rendimento dell'azienda oppure se si rischia solo di incappare in pesanti costi di transazione non più recuperabili.

Il modello di impresa virtuale prevede di mostrare la realtà aziendale in tutti questi aspetti, siano essi strutturali, per i quali si passa dalle catene di fornitura alle scelte dei canali di distribuzione, siano organizzativi, concernenti flussi di informazione e gerarchie, o siano gestionali, per valutazioni quali le quantità di beni da immagazzinare o la valorizzazione da assegnare ai prodotti. Un ampio spettro di studio che per ora ha portato ottenere informazioni ed impressioni in merito all'accumulo di semilavorati in magazzino, alla gestione delle informazioni ed alla contabilità.

L'impresa virtuale ha per ora il compito di ottimizzare il processo produttivo in funzione degli ordini in arrivo. I casi prospettati per operare i confronti tra le simulazioni sono tre: il primo prevede un'impresa che produce solo quando arrivano ordini, il secondo consiste nella stessa impresa con l'aggiunta della possibilità di immagazzinare beni ed il terzo inserisce la gestione delle informazioni relative agli ordini in arrivo. Naturalmente l'impresa che opera senza regolamentazione corre il rischio di formare colli di bottiglia che portano ad inevitabili ritardi nelle consegne, i casi successivi cercano di ovviare proprio a questo problema anticipando la produzione, sia in modo irrazionale (caso con i soli magazzini) sia in modo razionale (caso della gestione delle informazioni). Per comprendere i risultati della simulazione viene infine impostato un sistema contabile che cerca di penalizzare i sovra dimensionamenti dell'azienda e di avvantaggiare situazioni di sotto dimensionamento opportunamente gestite.

La struttura si basa ora su dati totalmente fittizi, in quanto agevolano la comprensione del comportamento del modello, ma nel momento in cui si avrà la certezza della correttezza dei dati in uscita dalle simulazioni sarà necessa-

rio disporre di dati reali per poter valutare l'effettiva applicabilità dell'impresa virtuale.

Capitolo 1

La simulazione come strumento di ricerca sull'attività di impresa

1.1 Concetti per la costruzione di un modello di impresa virtuale

L'obiettivo di costruire un modello di impresa virtuale può essere perseguito solo dopo che si è preso coscienza dei problemi che sussistono e delle scelte che devono essere operate in merito all'organizzazione di un'impresa. Un modello, infatti, ha il compito di riprodurre in forma semplificata una realtà, in questo caso un'impresa, rispettandone le caratteristiche principali e sottolineando gli aspetti rilevanti del suo operato, ma questo può essere ottenuto solo a seguito di un'attenta analisi della realtà di base. Procedendo, pertanto, con l'analisi delle caratteristiche di un'impresa occorre valutare quale di esse risulti di maggiore interesse al fine della costruzione di un modello. La documentazione in merito all'organizzazione dell'impresa è molto vasta e spazia da teorie che centrano l'attenzione su una suddivisione scientifica del lavoro, che tutelano la produttività aziendale, a teorie comportamentistiche, che tutelano il lavoratore in quanto anima del prodotto aziendale, fino alle teorie che strutturano l'azienda in funzione delle specifiche

proprie del mercato in cui essa opera. Un modello aziendale deve prendere in considerazione tutti gli aspetti salienti che emergono dalle teorie sopra riportate cercando di non sottovalutare alcuna informazione che potrebbe modificare l'aspetto dell'impresa. Infatti, è importante che il modello rappresenti:

- le *competenze operative* dei partecipanti al prodotto finale e le *procedure* da seguire per svolgere i singoli incarichi; nell'ambito della modellizzazione è vantaggioso suddividere le competenze in singoli atti elementari, facilmente analizzabili, per poi procedere alla loro combinazione nell'ambito di una procedura finalizzata.
- le *gerarchie* relative all'organizzazione da rispettare per lo svolgimento di determinate funzioni; aspetto questo di grande utilità nel formulare un modello per schematizzare, nel modo più flessibile possibile, i flussi informativi e di beni all'interno della struttura rappresentata.
- le *relazioni con il mercato* da parte dell'impresa, sia in termini di concorrenza sia in termini di input-output di beni; questo aspetto può modificare notevolmente la struttura aziendale a seconda che essa decida di produrre interamente il prodotto con una perfetta integrazione verticale o che decida per motivi di efficienza-efficacia di appoggiarsi al mercato.

Tutti questi aspetti che si intrecciano in una struttura di agenti, relazioni e regole rendono difficile la comprensione della realtà e, di conseguenza, la sua semplificazione all'interno di un modello. La situazione da rappresentare è riconducibile all'analisi di un *problema complesso* in quanto lo spazio di ricerca è difficilmente esplorabile in modo estensivo, e questo rende la ricerca lunga e difficile, e le relazioni funzionali del problema sono solo in parte comprensibili dagli agenti stessi.

Partendo dal presupposto che il modo migliore, valuteremo se è tale, per rappresentare un sistema di questo tipo sia analizzare le sue parti elementari, occorre

prima comprendere cos'è e come si può trattare un sistema di tipo complesso, al fine di capire, innanzitutto, se l'impresa che ci accingiamo a simulare debba essere trattata come tale e, successivamente, quali siano le relazioni funzionali al suo interno per poterla scomporre senza arrecare danno al risultato globale.

1.1.1 I sistemi complessi

I modelli elaborati in riferimento a sistemi complessi si contrappongono ai modelli tradizionali, in particolare matematici, per le ipotesi che vengono assunte nel processo di modellizzazione.

La tabella 1.1 illustra le due principali correnti di pensiero, da notare come in essa venga utilizzata una terminologia molto discussa quando si parla di complessità, specie in riferimento ai termini chaos e catastrofe. Essi sono stati il punto di partenza nell'elaborazione della teoria della complessità, la quale, ora, ha soppiantato i due termini ereditandone le principali tematiche.

I concetti relativi alla complessità sono portati avanti da centri di ricerca in tutto il mondo, tra cui (Rosser 1999, pp. 2-3) la Free University di Bruxelles in Belgio, la Stuttgart University in Germania e il Santa Fe Institute in New Mexico; essi studiano sistemi complessi al fine di comprendere fenomeni *dinamici non lineari*, come, ad esempio, per rimanere in campo economico, la formazione di bolle speculative in un mercato finanziario.

I risultati finora ottenuti hanno dato vita a pareri discordanti sulla reale utilità dello studio di fenomeni complessi. Ad esempio in riferimento alle ricerche svolte dal Santa Fe Institute M. Waldrop (1992, pp. 12-13) ha affermato:

(...) they all share the vision of an underlying unity, a common theoretical framework for complexity that would illuminate nature and human kind alike ... They believe that their application of these ideas is allowing them to understand the spontaneous, self-organizing dynamics of the world in a way that no one ever has before

Tabella 1.1: confronto tra modelli **tradizionali** e **complessi**

Modello tradizionale	Modello complesso
<p>Linearità</p> <p>è possibile prevedere ogni futuro stato o comportamento attraverso una semplice equazione causa-effetto</p>	<p>Non linearità</p> <p>non c'è proporzionalità nelle relazioni causa-effetto, il futuro è incerto le reazioni del sistema sono imprevedibili</p>
<p>Unificabilità delle parti</p> <p>il tutto come somma delle parti</p>	<p>Frammentazione</p> <p>la totalità è fatta di molteplici interazioni tra le single parti che la compongono</p>
<p>Controllo</p> <p>il sistema per quanto possibile è in grado di controllare il disordine</p>	<p>Chaos</p> <p>c'è una stretta relazione tra chaos e ordine, al punto che uno porta all'altro in un processo dinamico. Non evitare il chaos, ma utilizzarlo per auto-organizzare il sistema</p>
<p>Uniformità</p> <p>il sistema non cambia in modo improvviso. Se lo facesse, qualcosa non è stato ben controllato</p>	<p>Catastrofe</p> <p>un impercettibile cambiamento può portare a comportamenti esplosivi del sistema</p>

with the potential for immense impact on the conduct of economics, business, and even politics (...)

di contro J. Horgan (1995, 1997) ha definito queste teorie con l'etichetta *chaoplexology* riferendosi ai termini cybernetics, catastrophe, chaos e complexity come terreni su cui si è sempre discusso, ma non si è mai giunti a conclusioni pratiche; le sue parole sono state:

So far chaoplexologists have created some potent metaphors: the butterfly effect, fractals, artificial life, the edge of chaos, self-organized criticality. But they have not told us anything about the world that is both concrete and truly surprising, either in a negative or in a positive sense.

Le opinioni espresse sull'argomento sono, pertanto, discordanti, ma sottolineano l'attualità dell'argomento nel settore della ricerca economica, finanziaria e sociale.

In campo economico una significativa introduzione all'argomento della complessità si trova in Marshall (1920), dove si affronta il problema economico derivandone analogie con altre scienze da quelle sociali a quelle biologiche. In particolare egli differenzia quello che può essere lo studio di una scienza fisica, nella quale il principio causa effetto genera soluzioni universalmente vere a parità di condizioni di partenza e di sviluppo del processo, dallo studio di fenomeni sociali, biologici e, perchè no, economici nei quali non ci è dato di comprendere le parti elementari che li costituiscono e, pertanto, ancor meno il comportamento che terranno. Marshall sottolinea che tali forze più che cambiare si *evolvono*, influenzando imprevedibilmente altre forze circostanti; come si deduce da Marshall (1961, p. 241):

(...) the development of the organism, whether social or physical, involves an in-

creasing subdivision of functions between its separated parts on the one hand, and on the other a more intimate connection between them. Each part gets to be less and less-sufficient, to depend for its well-being more and more on other parts, so that any disorder in any part of a highly-developed organism will affect other parts also.

L'evoluzione delle parti in gioco, nel modo di pensare e di agire, complica continuamente le relazioni che intercorrono tra gli agenti. Questo rende sempre più instabile il sistema in cui si opera, o meglio, se la stabilità viene raggiunta non è detto che essa sia mantenuta oppure possono verificarsi più stabilità contemporanee coesistenti. Tutto ciò porta a concludere che difficilmente si opera in un ambiente lineare, vedi tabella 1.1. La non-linearità del sistema provoca sia problemi di carattere matematico sia, riferendoci specificatamente alla teoria del modello di impresa virtuale, problemi di carattere economico.

Per quanto concerne i problemi di tipo matematico, in riferimento alla matematica tradizionale, la non-linearità compromette, in primo luogo, l'equilibrio del sistema, e con esso la certezza di raggiungere un risultato coerente con la realtà rappresentata, in secondo luogo, implica la *non cumulabilità* degli eventi. Di conseguenza, eventi non cumulabili implicano che lo studio di singole parti del sistema, indipendenti dalle relazioni che le legano tra di loro, non sia sufficiente per poter trarre conclusioni sulla globalità del problema. Questa limitazione crea notevoli problemi nell'applicazione pratica del problema, poichè nel modellizzare il gestore del sistema deve cercare di semplificarlo suddividendolo in parti più semplici e maggiormente gestibili, ma poichè esse non sono cumulabili corre il rischio di spezzare relazioni tra di esse che non potrà più recuperare. Nel caso particolare degli studi economici questo aspetto può essere riscontrato nella non cumulabilità di risultati di breve periodo per assurgere a conclusioni di lungo periodo. Un'ultima precisazione sulla non linearità è relativa alla sua interdipendenza con la complessità, in altre parole se una implica l'altra. Per risolvere questo problema può essere utile ricordare Day (1994), secondo il quale un sistema dinamico può

essere anche complesso se endogenamente non tende asintoticamente ad un punto fisso, ad uno spazio definito o se non esplode. Questi sistemi possono esibire comportamenti discontinui e possono essere rappresentati da equazioni differenziali, ma non tutti generano comportamenti complessi; ad esempio, una funzione esponenziale monotona crescente, che spesso indica una crescita economica, è un caso di sistema non lineare e non complesso. Da questo possiamo dedurre che se è condizione necessaria per un sistema complesso la non linearità è altresì vero che essa non è anche condizione sufficiente per la complessità.

Un'ulteriore complicazione al modello derivata nello specifico studio delle scienze economiche, che incide ancora maggiormente sulle difficoltà computazionali di un modello complesso, è causata dalla possibilità degli agenti di *prendere decisioni*. Il problema proprio di ogni schema economico è sulla modalità di azione degli agenti in base alla loro *razionalità*, limitata o perfetta, ed in base alle loro *aspettative*, razionali o irrazionali. In un modello tradizionale, caratterizzato dalla linearità, è necessario ipotizzare la perfetta capacità degli agenti di muoversi all'interno del mercato, supponendo che essi siano in grado di conoscere l'ambiente in cui si muovono e le leggi che lo regolano. Questa restrizione può portare a forzature nei risultati del modello e, pertanto, a risultati non corretti. Lo sviluppo di strumenti idonei a manipolare sistemi complessi potrebbe risolvere alcuni problemi previsionali dinamici e non lineari. Questo passo ha probabilmente spinto gli studiosi di fenomeni complessi ad interessarsi di economia, infatti è in questo punto che le scienze economiche si distaccano da quelle fisiche o biologiche.

Non resta che definire quali sono le principali condizioni che, presenti nell'ambiente analizzato, comportano tutti i problemi finora descritti. Secondo le teorie del Santa Fe Institute, elaborate da Arthur e altri (1997, pp. 3-4), la complessità è frutto dei seguenti punti:

1. interazioni disperse tra eterogenei agenti che operano localmente e contemporaneamente in uno spazio

2. non ci sono controllori globali in grado di spiegare tutte le interazioni che intercorrono tra gli agenti e quali opportunità offre il sistema
3. un'organizzazione intricata con una fitta rete di interazioni
4. continua evoluzione ed adattamento degli agenti al sistema grazie alla capacità di apprendere
5. continuo inserimento nel sistema di novità tecnologiche, mercati, istituzioni
6. la possibilità di nessun equilibrio all'interno del sistema o della coesistenza di più equilibrii, nessuno dei quali verosimilmente prossimo all'ottimo globale

Queste condizioni sono proprie di un ambiente imprevedibile nei comportamenti e nelle decisioni; in termini pratici è difficile prevedere se si formeranno bolle speculative, in campo finanziario, se, in tema sociale, un individuo che ha coraggio di protestare è in grado di provocare una rivoluzione, o se, in campo macroeconomico, possiamo attribuire ad una delle precedenti condizioni il fallimento del tentativo di raggiungere l'equilibrio in un modello keynesiano.

Implicazioni della complessità

I presupposti prima presentati e gli esempi sopra riportati portano a implicazioni prima di tutto di tipo teorico. Infatti, se si ottengono risultati con la verosimile coesistenza di più equilibrii o la mancanza totale di un equilibrio, come visto nei casi di interazione tra una moltitudine di agenti, si può dedurre che occorre interpretare l'ambiente come composto da agenti con *aspettative irrazionali*. A livello microeconomico (Colander 1998), ad esempio, questo comporta che non sempre si ottenga l'equilibrio richiesto dalla legge di mercato di Walras ¹, con conseguenti nuove teorie definite come *Post Walrasian economics*. Questo concetto viene

¹La legge di Walras a cui ci si riferisce prevede che la somma di domanda e offerta in due mercati che effettuano scambi sia identicamente uguale a zero, in virtù del fatto che si azzerano

ripreso e per la maggior parte ribadito da Sargent (1993) il quale afferma che piuttosto che parlare di aspettative non razionali sarebbe più opportuno parlare di *aspettative adattive* in un *contesto di razionalità limitata* causato dall'oggettiva difficoltà computazionale degli agenti in ambiente complesso. Sempre sullo stesso argomento Elster (1979) ribadisce l'importanza della razionalità limitata, affermando:

(...) come colui che sta salendo su un pendio, ha i suoi occhi miopi fissi al suolo, ed è incapace di prendere in considerazione ciò che accade aldilà del dosso più prossimo.

e puntando il dito su un problema che la razionalità limitata sicuramente porta ovvero la determinazione dell'ottimo per il sistema, vedi il paragrafo 1.1.2.

Un'ulteriore implicazione è di carattere empirico-metodologico, specie per l'introduzione negli ultimi anni della simulazione attuata utilizzando computer sempre più veloci e, pertanto, in grado di gestire un elevato numero di eventi. La domanda sorta è se effettivamente l'utilizzo di mezzi innovativi possa risolvere i problemi riscontrati in passato. Proprio grazie a questo miglioramento tecnologico che consente l'utilizzo di software molto elaborati anche su un normale personal computer può prendere forma la simulazione dell'azienda virtuale anche a livello accademico.

Modelli sulla complessità

Un buon numero di lavori in campo economico sul tema della complessità è partito dal *dilemma del prigioniero*. La complessità è stata inserita nel modello tramite il ripetersi di operazioni decisionali da parte di un numero di agenti superiore alla coppia prevista nel modello base e di gruppi di agenti con strategie diverse in risposta alle sollecitazioni dei *vicini*. Lo studio si è quindi incentrato

gli eccessi di domanda e di offerta; e che questo si realizza per tutti i prezzi disponibili, non solo per il prezzo di equilibrio.

sull'evoluzione nelle capacità decisionali degli agenti e se effettivamente la situazione potesse migliorare. Un modello di questo tipo è stato elaborato da Lindgren (1997), il quale ha supposto un certo numero di agenti che interagiscono con un largo numero di decisioni prese da altri e che operano in un ambiente a *nido d'ape* ove reagiscono in base agli stimoli della cella a loro vicina. Gli agenti vivono in un ambiente in grado di auto organizzarsi in base alla loro capacità di prendere decisioni e di modificarle nel tempo. I risultati di questo tipo di simulazione hanno sottolineato: in primo luogo, la formazione di nuove strategie e, in secondo luogo, la mancanza di un equilibrio globale.

Un esempio simile per costruzione e risultati è stato sviluppato all'interno del Santa Fe Institute fin dal 1989, si veda Arthur (1997), nel formulare un mercato di scambio borsistico. In questo modello vi sono numerosi agenti, numerosi analisti di mercato e processo di distribuzione del dividendo stocastico; gli agenti decidono in base ad un gruppo di aspettative casuali di cui dispongono e in base alle previsioni degli analisti di mercato, scegliendo se accettarle o meno. I risultati di questa simulazione hanno mostrato due possibili soluzioni: in una gli agenti non interrogano più gli analisti e le loro aspettative convergono in un'unica soluzione di aspettativa razionale, nell'altra emerge un mercato complesso con la formazione di tutte le strategie di mercato che si adattano al mutare della situazione. Il mercato presenta così bolle e rotture tipiche di un mercato reale.

Questi e molti altri modelli hanno preso spunto dall'idea di utilizzare la simulazione per costruire esempi di *vita artificiale* tramite l'utilizzo di *automi cellulari* che interagiscono l'uno con l'altro, crescono e si evolvono (Langton, 1989). In questo ambito Holland (1992) ha sviluppato *classifier system* che giudicano il comportamento strategico degli agenti e possono generare meccanismi adattivi attraverso algoritmi genetici che modificano ed evolvono le strategie nel corso del tempo. Appare immediatamente l'importanza questo strumento che consente di introdurre in qualunque tipo di simulazione soggetti in grado di apprendere dai loro errori e di modificare le decisioni future in funzione degli stessi.

1.1.2 La decomposizione di problemi complessi

Nel caso della formulazione del modello di azienda virtuale ci si imbatte inevitabilmente in un caso di rappresentazione di problema complesso. Pertanto è opportuno, prima di procedere nella costituzione pratica del modello, prendere coscienza delle caratteristiche che un problema di questo tipo presenta per poter operare un'analisi inerente al caso. Nuovamente facciamo ricorso ad un concetto espresso da Marshall (1920, p. 175), il quale afferma che la complessità non si ritrova solo nell'*oggetto di studio*, ma anche nella *conoscenza* utilizzata per analizzare questi processi. Con questo intendo dire che ci sono due tipologie di complessità con cui ci si scontra nell'analisi del problema: la prima inerente alla razionalità limitata dell'*analista* nel visualizzare in modo completo e corretto il problema, la seconda inerente alla razionalità limitata degli agenti costituenti l'oggetto di studio e determinanti per l'equilibrio del sistema. Quindi, l'obiettivo dell'analista nel modellizzare è riuscire nel miglior modo possibile a comprendere l'ambiente di studio per poter così ridurre al minimo la complessità che deriva dalla propria complessità cognitiva. Il primo passo da compiere è *scomporre* il problema in più sottoparti singolarmente gestibili e rappresentabili. Questo passo è forse il più complicato da svolgere, soprattutto se facciamo riferimento a quanto detto nei paragrafi precedenti; si presume, infatti, di trattare agenti con *razionalità limitata* che, per agire, come sottolinea Simon (1981):

(...) devono procedere necessariamente tramite la decomposizione di un problema complesso, computazionalmente intrattabile, in più sottoproblemi di dimensioni minori che si possano risolvere l'uno indipendentemente dall'altro (...)

Nel nostro caso l'azienda può essere vista come un aggregato multidimensionale di pratiche, regole e procedure spesso sconosciute anche a chi la amministra e, quindi, ricondotta ad un sistema complesso impenetrabile nella sua globalità.

La decomposizione del problema porta inevitabilmente a scindere alcune interdipendenze del sistema, scissione che, però, ha un costo che può risultare anche molto elevato. Infatti, il compito delle parti costituenti il problema è quello di operare alla ricerca di un risultato ottimale non tanto per se stesse, ma sempre e comunque per la globalità del sistema. Ecco quindi che la ricerca dell'ottimo globale può essere compromesso dalla mancanza di interdipendenze necessarie, mancanza che porta alla miopia della totalità e pertanto all'accettazione sì di ottimi, ma di ottimi locali. Con questo si intende che a sbagliare non è l'agente, ma l'amministratore del problema il quale ha reso l'agente stesso incapace di trovare il risultato migliore.

Le molteplici interdipendenze di un problema complesso creano molti ottimi locali; bisogna riuscire a separarle adeguatamente ed ad inviare segnali forti agli agenti cosicchè essi persistano nella ricerca dell'ottimo globale. Siamo così di fronte alle seguenti diverse situazioni, Delaini e altri (2000):

- un problema perfettamente decomposto è stato ridotto all'unione di problemi di minima complessità, mentre un problema non decomposto ha un grado di complessità massimo
- un problema il cui schema di decomposizione abbia dimensione minima può essere risolto in *tempo* lineare, mentre un problema non decomposto si può risolvere in tempo esponenziale, non accettabile neanche per un velocissimo computer
- d'altro canto uno schema non decomposto ha sempre una soluzione ottima raggiungibile, mentre uno schema perfettamente decomposto prevede soluzioni ottime solo sotto condizioni sempre più restrittive

vi è quindi un pericoloso *trade off* tra complessità ed ottimalità.

Uno schema di decomposizione è una sorta di sagoma ideale che definisce e delimita come nuove possibili soluzioni possano essere generate e testate. In ampi

spazi di ricerca, dove è possibile esplorare solo un piccolo sottoinsieme di tutte le possibili combinazioni, la strategia impiegata per generare tali nuove combinazioni gioca un ruolo cruciale nella definizione dell'insieme delle soluzioni finali ottenibili. Apprese le soluzioni locali occorre raggiungere quelle finali, seguendo un determinato percorso suggerito da Delaini e altri (2000). Assumiamo che gli agenti a razionalità limitata possano esplorare solo localmente lo spazio e solo nelle direzioni indicate dallo schema di decomposizione: nuove soluzioni sono generate e testate in un intorno della combinazione di partenza, dove i *vicini* sono nuove combinazioni ottenute modificando alcuni (e possibilmente anche tutti gli) stati degli elementi contenuti nel blocco. Una nuova soluzione sarà un *vicino* (*preferibile*) per un determinato blocco quando risulti migliore di quella di partenza secondo la relazione di preferenza del problema, cioè quando ottenga un payoff più alto; essa differirà dalla combinazione di partenza solo per gli stati assunti da elementi che appartengano al blocco in questione. Stando alla definizione del meccanismo di selezione, un *vicino* può essere raggiunto da una certa possibile soluzione tramite una singola operazione di mercato. Data una certa combinazione, avremo un insieme di *vicini* che la circonda. Da questo insieme dovremo estrapolare il *vicino ottimo*, che sarà la combinazione che ottiene il punteggio in assoluto più alto nell'insieme dei *vicini*. Spostandoci nello spazio delle possibili soluzioni ci troveremo su un ottimo locale quando l'insieme dei *vicini* (*preferibili*) della combinazione è l'insieme vuoto, vale a dire quando non potremo più spostarci in alcuna direzione consentita dal nostro schema di decomposizione senza diminuire il payoff ottenuto. Riprendendo l'esempio di Elster (1979) dello scalatore che cerca la vetta più alta in mezzo alle montagne (vedi il paragrafo 1.1.1) ora: siamo su un picco della montagna ma non sulla vetta, per arrivare alla vera cima possiamo solo scendere e poi risalire. Generalizzando dal singolo blocco all'intera decomposizione, avremo un insieme di *vicini* (*preferibili*) per lo schema di decomposizione all'interno del quale ci muoveremo per raggiungere la soluzione finale seguendo un preciso percorso che chiamiamo *sentiero di ricerca*.

Un sentiero di ricerca altro non è che una precisa sequenza di vicini (preferibili), cioè una precisa sequenza delle combinazioni che hanno ottenuto il payoff di performance globale più alto, ciascuna nel suo blocco. Il sentiero di ricerca sarà ottimo se corrisponderà ad una sequenza di *vicini ottimi*. Fra tutti gli schemi di decomposizione di un certo problema, siamo specialmente interessati a quelli per i quali il massimo globale è raggiungibile a partire da una qualunque combinazione. Una decomposizione di questo tipo, che crei sentieri di ricerca che collegano ciascuna combinazione alla soluzione ottima, esiste sempre, ed è la decomposizione che considera il problema nella sua interezza, per cui esiste un solo ottimo locale che coincide con quello globale. Ovviamente, però, noi siamo interessati a decomposizioni minori - se consentite dalla struttura del problema - ed in particolar modo a quelle minime. La decomposizione degenere rappresenta la massima estensione cui si può arrivare nella suddivisione del problema in sotto-problemi indipendenti e coordinati da un meccanismo di selezione tipo mercato, in cui tale selezione conduca alla massimizzazione da qualunque combinazione si partenga. Purtroppo per noi, invece, decomposizioni più raffinate in generale non permettono processi di selezione decentralizzata che guidino all'ottimizzazione, a meno che per puro caso non ci si trovi in partenza all'interno del bacino di attrazione del massimo globale. L'algoritmo di ricerca della soluzione ottima si può rappresentare, in maniera informale, in questo modo:

1. partire con la decomposizione minima: N blocchi atomici composti da un solo elemento
2. verificare se esiste un sentiero ottimo che conduce dalla combinazione con punteggio più basso a quella con payoff massimo. Se sì, STOP, altrimenti procedere.
3. costruire una decomposizione meno raffinata, unendo i blocchi di dimensione minore che non hanno permesso di soddisfare la condizione 2. Ritornare a 2.

Nel costruire uno schema di decomposizione aspiriamo a raggiungere la perfetta decomponibilità, nel senso che richiediamo che tutti i blocchi possano essere ottimizzati in modo completamente indipendente l'uno dall'altro. In questo modo ci è garantita la possibilità di scomporre il problema in elementi perfettamente isolati. Questa è una richiesta decisamente restrittiva: anche quando le interdipendenze sono relativamente deboli, ma diffuse fra tutti gli elementi costitutivi del problema, ci troviamo facilmente in casi in cui non esiste alcuna forma di perfetta decomponibilità. Possiamo allora indebolire la richiesta di perfetta decomponibilità con una di quasi-decomponibilità: non chiediamo più che il problema sia scomponibile in sottoproblemi non correlati. Ci basta che i sottoproblemi contengano solo le interdipendenze più influenti in termini di performance globale, mentre quelle che incidono meno possono restare sparse fra i blocchi. Per questa via l'ottimizzazione separata di ciascun sottoproblema non ci condurrà necessariamente alla scoperta dell'ottimo globale, ma ci permetterà almeno di raggiungere una delle soluzioni migliori. In altre parole costruiamo quasi-decomposizioni che ci consentono di dare una precisa misura del trade-off tra ottimalità e decentralizzazione: un maggior livello di decentramento e di coordinazione dei mercati, e quindi una più alta velocità di adattamento al sistema, sono ottenibili solo a spese dell'ottimalità delle soluzioni che si possono raggiungere. Se rinunciamo a trovare con certezza l'ottimo globale possiamo arrivare ad elevati livelli di decentralizzazione della ricerca nello spazio.

Una volta compreso il problema e, quindi, superata la complessità cognitiva del sistema, quello che le simulazioni in generale e l'azienda virtuale nel caso specifico si propongono di fare è rappresentare nuovamente in forma diversa, semplificata il problema. Questo passo viene dettato dai vincoli cognitivi della mente umana che ci impongono, secondo la psicologia del pensiero, di semplificare la realtà standardizzandola in rappresentazioni schematiche. Finora abbiamo ipotizzato che lo spazio preso in considerazione fosse determinato esogenamente e non manipolabile, ma se gli agenti non conoscono la struttura oggettiva del

problema e, pertanto, dispongono solo della rappresentazione che essi stessi si costruiscono, unicamente lo spazio relativo alla loro capacità interpretativa deve essere decomposto. Grazie a questa forte astrazione, che permette una riduzione dello spazio di analisi a dimensioni accettabili per la conoscenza umana, si possono trarre tre importanti conclusioni che fanno della ri-rappresentazione della realtà un potente strumento di ricerca:

- ogni problema ammette una rappresentazione ed uno schema di decomposizione che permettono di risolverlo
- dato un qualunque problema, esso ammette una decodificazione tale da renderlo risolvibile secondo uno schema di decomposizione di dimensione minima
- dato un qualunque problema ed una qualunque decodificazione, esiste sempre una relazione di preferenza che rende il problema risolvibile con uno schema di decomposizione di minima complessità

In sostanza, la simulazione viene eseguita nel tentativo di semplificare ciò che avviene nella realtà, economica o biologica che sia, sicuramente in modo complesso e, pertanto, difficilmente gestibile. Simulando si cerca proprio di migliorare la gestibilità di questi sistemi, inserendo l'opportunità di introdurre modifiche al fine di valutare i comportamenti degli agenti. Questo può, come ultimo passo, favorire la capacità previsionale di chi osserva dall'esterno un sistema complesso, riducendone la complessità cognitiva. Nel simulare occorre ricordare che il grado di complessità di un problema, la sua scomponibilità ed il tempo richiesto per la sua risoluzione dipendono dalla rappresentazione che ne viene fatta; la scelta della rappresentazione è arbitrariamente decisa dal gestore problema, il quale però deve sapere a priori che, in base a quello finora sostenuto, può trasformare qualunque problema in uno di complessità minima.

1.2 Logica e strumenti di una simulazione

1.2.1 La logica

La simulazione è, in base a ciò che finora è stato affermato, un metodo di ricerca scientifica per raggiungere un *obiettivo*, quale può essere sintetizzare una realtà complessa in un *modello*. Simulare è, quindi, un *metodo* tramite il quale il ricercatore intende elaborare *dati simulati* per confrontarli con *dati raccolti* tramite, ad esempio, modelli statistici. L'obiettivo di questo confronto è valutare l'affidabilità del modello nel rappresentare ciò che accade al fine di, qualora si ottenga un elevato grado di congruenza tra dati simulati e reali, utilizzare il modello in termini previsionali.

I passaggi da compiere per giungere all'obiettivo finale sono (vedi figura 1.1):

- *astrarre* il problema in un modello
- *simulare* ovvero far lavorare il modello, vedremo nel paragrafo 1.2.2 con quali strumenti, e studiarne i risultati
- *raccogliere* i dati reali sul problema proposto
- *confrontare* i dati simulati con quelli reali

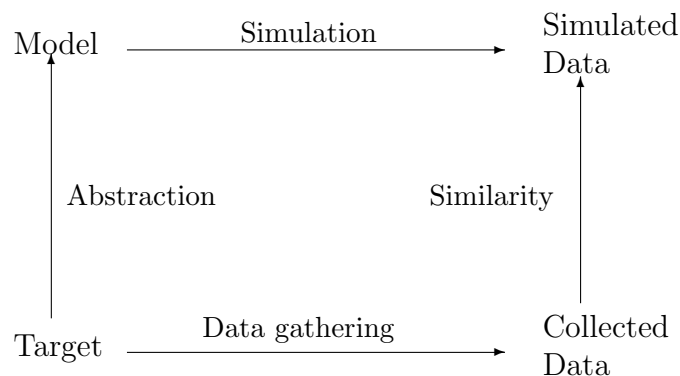


Figura 1.1: la logica della simulazione

I momenti che si riferiscono maggiormente alla parte operativa del percorso sopra descritto sono l'*astrazione* e la *simulazione*. Astrarre la realtà in un modello comporta difficoltà crescenti proporzionalmente alla crescente complessità del modello. In altri termini, poiché è obiettivo del processo formulare risultati coerenti con la realtà trattata, è opportuno che il modello, pur semplificandola, si accosti ad essa quanto più possibile rispettandone le relazioni che la caratterizzano. In questo passaggio si identificano i problemi emersi in merito alla decomposizione di problemi complessi, si veda il paragrafo 1.1.2, ovvero la necessità di semplificare l'oggetto da rappresentare nel modello rispettandone, però, le peculiarità che lo contraddistinguono. Infatti, in accordo con l'intento di utilizzare il modello per fini previsionali, l'amministratore dello stesso deve accertarsi che esso produca risultati corretti quanto meno nella forma in cui sono espressi. In altre parole egli deve essere cosciente delle ipotesi di base che vengono formulate per semplificare il modello e degli influssi che esse operano sui risultati ottenuti, in modo da valutare l'ambito in cui a senso accettare come validi i risultati della simulazione. I limiti ed i possibili errori dell'astrazione e, con essa, della simulazione sono causati in parte dall'abilità interpretativa della realtà, problema cognitivo di chi gestisce il problema complesso per il quale si veda il paragrafo 1.1.2, ma in parte dagli strumenti di cui dispone l'amministratore del processo. Con il termine *strumenti* si intende tutto ciò che viene utilizzato per costruire materialmente il modello, dall'applicazione di relazioni matematiche all'utilizzo di un programma su computer. La scelta di uno strumento piuttosto che un altro dipende dalle caratteristiche proprie dello stesso e da come esse consentano di ridurre al minimo le ipotesi sottostanti la simulazione.

1.2.2 Gli strumenti

L'interrogativo che si pone a seguito delle affermazioni finora fatte in merito alla complessità dei modelli economici ed alla difficoltà della loro trattazione è relativo

a quale strumento sia il più idoneo per una simulazione economica.

Ostrom (1988) sottolinea tre vie: la prima è una discussione verbale del problema, la seconda una modellizzazione matematica e la terza la simulazione tramite computer. Considerando che la prima difficilmente porta a conclusioni pratiche e che la seconda, vedi il paragrafo 1.1.1, si basa spesso su ipotesi poco plausibili o forzate, non resta che approfondire l'analisi della terza. La simulazione tramite computer consiste nell'elaborare un modello tramite un programma che formalizza teorie complesse, elabora esperimenti e consente di osservare l'imprevedibilità del processo. In questo modo si cerca, in un primo momento, di rappresentare il problema complesso nella sua interezza senza scindere troppe interdipendenze a causa di eccessive semplificazioni ed, in un secondo momento, di osservare come gli agenti del sistema si comportino con il trascorrere del tempo. Un modo di procedere per simulare su computer sistemi dinamici non-lineari di carattere economico è quello di costruire modelli basati su agenti che siano in grado di rappresentare la complessità e l'imprevedibilità. La situazione in cui ci troviamo è quella rappresentata in figura 1.2.

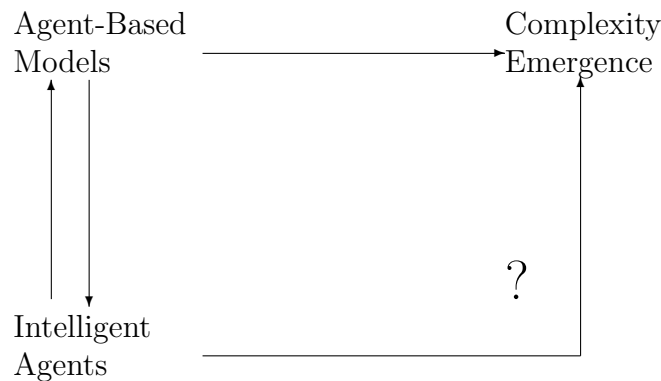


Figura 1.2: problema di simulazione

Questa rappresentazione sottolinea i due passaggi, già citati, che il gestore della simulazione intende compiere, ovvero:

- la *costruzione* del modello tramite un programma specifico su computer.

Parte pratica volta a formalizzare il passaggio da modello basato su agenti a sistema complesso ed imprevisto-imprevedibile (vedi figura 1.3).



Figura 1.3: simulazione su computer

- l'*interpretazione* dei risultati da parte di chi osserva la simulazione. Parte teorica della simulazione che si presta a molteplici interpretazioni sia relative alla costruzione del modello sia relative ai risultati prodotti dal modello. Proprio per sottolineare questa libertà di valutazione in figura 1.2 questo passaggio viene rappresentata con un punto interrogativo.

Il primo passo da compire per iniziare la simulazione è costruire un modello basato su agenti che disponga delle seguenti caratteristiche. Gli agenti devono essere considerati, si veda Gilbert e Terna (1999), come *oggetti*, intesi come parti di un programma, in grado di contenere informazioni e regole. Le regole consentono il funzionamento del meccanismo di reazione alle informazioni che provengono dall'esterno del modello. Il passo successivo consiste nell'osservare il comportamento dei singoli agenti attraverso l'andamento di variabili interne, proprie di ogni agente-oggetto, e lo sviluppo del loro comportamento collettivo. Infine, gli agenti così creati vengono inseriti in una struttura nella quale possono disporre di *informazioni pubbliche*, comuni a tutti gli agenti, e di *informazioni private*, specifiche per ogni agente; quindi, sulla base di entrambe prendere decisioni. La struttura ad agenti creata è in grado, a questo punto, di elaborare ed emettere dati sulla base del comportamento delle parti componenti il modello. I dati emessi devono essere raccolti ed analizzati da chi osserva il modello, il quale potrà interpretarli e valutare se apportare modifiche. La possibilità di modificare le componenti del sistema per studiare quali effetti esse abbiano sulla globalità del modello, con l'obiettivo di un continuo miglioramento dello stesso, è uno dei

punti di forza della simulazione su computer. Da notare che simili cambiamenti possono essere apportati solo tenendo conto di un fattore determinante per una simulazione impostata in questo modo, ovvero del **tempo**; è, infatti, necessario *sincronizzare* i tempi degli esperimenti per essere sicuri che le osservazioni fatte in aggregato e la conoscenza che emerge dalla condizione interna degli agenti siano rilevanti in termini di liste di esperimenti (experimental schedules).

Dopo aver elaborato un modello basato su agenti occorre considerarlo in relazione con la complessità del modello, come già trattato in precedenza (si veda il paragrafo 1.1.1), ma anche con l'**emergence**. Questo termine, che letteralmente significa apparizione improvvisa, è rilevante specie per la parte interpretativa dei risultati ottenuti dalla simulazione; per meglio comprendere la sua importanza è opportuno definirlo dettagliatamente nei seguenti modi:

- *unforeseen emergence*: in riferimento alla formazione di fenomeni imprevisibili, per esempio, quando ci si aspetta di raggiungere uno stato di equilibrio, trascorso un certo tempo di simulazione, ed invece si genera un comportamento ciclico determinato dalla struttura interna del modello.
- *unpredictable emergence*: in riferimento al fatto che il chaos è osservabile nelle reali scienze sociali, ma che non è facile costruire un processo inverso che porti ad esso come risultato di una simulazione basata su agenti.

L'importanza di queste definizioni deriva dal fatto che esse mettono in evidenza le due situazioni chiave che possono emergere da una simulazione, da un lato un imprevisto comportamento del modello e dall'altro l'imprevedibilità delle azioni degli agenti.

1.3 Lo schema E/R/A

Prima di analizzare quale sia il programma opportuno per costruire un modello basato su agenti è necessario illustrare uno schema generale che può essere impiegato per simulazioni di questo tipo. Nelle simulazioni di fenomeni economico-sociali un modello generale sul quale basarsi per analizzare casi specifici è lo schema, presentato ad esempio in Terna (2000), *Environment-Rules-Agents* (E/R/A). Tale schema, illustrato in figura 1.4, ha il pregio di porre su livelli concettualmente differenti da un lato l'ambiente, che fornisce regole ed informazioni generali, dall'altro gli agenti, che decidono sulla base di informazioni private.

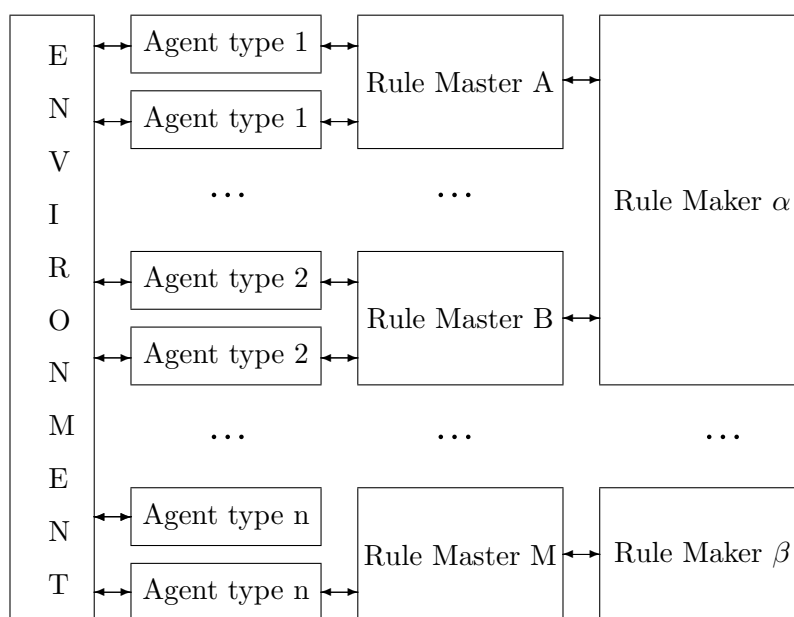


Figura 1.4: lo schema E/R/A

Il nucleo principale dello schema E/R/A è costituito dagli *agenti*; un agente può essere qualunque cosa o persona che compia un'azione rilevante per raggiungere gli obiettivi del modello. Non esistono criteri prestabiliti per la definizione di un agente: la necessità stessa di generarlo nasce nel momento in cui, astruendo la

realtà in un modello, si desidera sintetizzare una precisa azione in un determinato oggetto deputato a compierla.

Gli agenti che vengono così creati all'interno del modello necessitano di una precisa collocazione per evitare che l'azione da loro compiuta non si muova in sintonia con quelle da altri svolte. Una prima strada perseguibile consiste nell'instaurare una fitta rete di relazioni che metta in relazione tutti gli agenti tra di loro; in questo modo però si hanno due tipi di problemi: in primo luogo si rischia di aumentare esponenzialmente il numero di legami con l'aumentare del numero degli agenti, in secondo luogo il modello diventa difficilmente gestibile in caso di modifiche. La soluzione adottata, si veda figura 1.4, è quella di non far comunicare direttamente gli agenti tra di loro, ma tutti tramite un l'*Environment*. In questo modo si viene a creare un ambiente nel quale è possibile reperire tutte le informazioni di carattere pubblico di cui gli agenti desiderino disporre. La successiva aggiunta di nuove funzioni al modello non costituisce più un problema, in quanto è sufficiente renderle disponibili nell'*Environment* affinché siano percepibili da tutto il modello.

Oltre alle informazioni pubbliche che gli agenti raccolgono dall'ambiente essi dispongono di informazioni private che ottengono dai *gestori di regole* o *Rule Masters*, si veda figura 1.4. I *gestori di regole* costituiscono la conoscenza esterna ovvero sono dei fattori discriminanti nel comportamento degli agenti. Essi forniscono regole da seguire nel caso in cui gli agenti si trovino ad affrontare determinate scelte.

Occorre aggiungere che tali regole non sono specifiche per un determinato agente, ma sono relative al verificarsi di una specifica situazione. In questo modo il loro utilizzo non è vincolato, ma possono essere utilizzate da più agenti contemporaneamente.

A loro volta i *gestori di regole* possono affidare parte della loro conoscenza ai *Rule Makers*, i quali si occupano di produrre le regole da gestire. Nell'ottica del modello i *produttori di regole* hanno il compito di modificare la gestione del

comportamento degli agenti. Nel caso in cui, ad esempio, sia previsto un processo di apprendimento degli agenti dai risultati raggiunti dalla simulazione, si rende necessario l'impiego di produttori di regole; essi avranno il compito di indicare ai *Rule Masters* quali siano le nuove regole da indicare agli agenti.

L'utilizzo di uno schema di tipo E/R/A per simulare tramite personal computer vincola molto la scelta del linguaggio di programmazione da utilizzare. Infatti, si veda il capitolo 2, il processo di astrazione richiesto per formulare tale schema implica l'impiego di uno strumento molto flessibile e versatile. Caratteristiche, queste, fornite dalla programmazione ad oggetti, la quale consente di ricondurre a singoli oggetti le varie parti dello schema e di regolare lo scambio di informazioni che vi intercorre con legami facilmente scindibili e ripristinabili.

La programmazione ad oggetti è costituita da molti linguaggi, quali Java, Objective C, C++, Visual Basic e altri, tutti con caratteristiche idonee allo sviluppo di una simulazione che richieda astrazione e flessibilità. Maggiore attenzione all'argomento di modelli basati su agenti è stata rivolta dal Santa Fe Institute che ha sviluppato lo strumento denominato *Swarm*. *Swarm* consiste in un insieme di librerie con caratteristiche specifiche per simulazioni di tipo socio-economico agent-based. Poiché tali librerie sono esclusivamente supportate dai linguaggi *Java* ed *Objective C* ad essi, vedi capitolo 2, si farà riferimento.

Bibliografia

- [1] Arthur, W. Brian, S. N. Durlauf e D. A. Lane (1997), *The Economy as an Evolving Complex System II*, Addison-Wesley, pp. 1-14.
- [2] Bissey M. E. (2000), *Una piccola introduzione a Swarm: Objective C e Java*, <http://polis.unipmn.it/alex/activities/corso.html>, pp. 3-4.
- [3] Colander D. (1998), *Beyond New Keynesian Economics: Towards a Post Walrasian Macroeconomics*, in Roy J. Rotheim, *New Keynesian Economics/Post Keynesian Alternatives*, London, pp. 277-87.
- [4] Day H. (1994), *Complex Economic Dynamics, Volume I: An Introduction to Dynamical Systems and Market Mechanisms*, Cambridge, MA.
- [5] Gilbert N. e Troitzsch K. G. (1999), *Simulation for the Social Scientist*, Philadelphia.
- [6] Holland J. H. (1992), *Adaptation in Natural and Artificial Systems*, Cambridge.
- [7] Horgan J. (1995), *From Complexity to Perplexity*, in Scientific American, June, pp. 104-09.
- [8] Horgan J. (1997), *The End of Science: Facing the Limits of Knowledge in the Twilight of the Scientific Age*, New York, Broadway Books.
- [9] Langton C. G. (1989), *Artificial Life*, Redwood City.

- [10] Lindgren K. (1997), *Evolutionary Dynamics in Game-Theoretic Models*, in W. Brian Arthur, Steven N. Durlauf, e David A. Lane, eds., *The Economy as an Evolving Complex System II*, pp. 337-67.
- [11] Luna F. e Stefannson B. (2000), *Economic Simulations in Swarm: Agent-Based Modelling and Object Oriented Programming*, Kluwer Academic Publishers.
- [12] Marshall A. (1920), *Principles of Economics*, London.
- [13] Sargent T. J., *Bounded Rationality in Macroeconomics*, Oxford.
- [14] Terna P. (2000), *Economic Experiments with Swarm: a Neural Network Approach to the Self-Development of Consistency in Agents' Behaviour*, in Luna F. e Stefannson B. (2000), *Economic Simulations in Swarm: Agent-Based Modelling and Object Oriented Programming*, Chapter 3.
- [15] Waldrop M. (1992), *Complexity: The Emerging Science at the Edge of Order and Chaos*, New York, Simon & Schuster.

Capitolo 2

La simulazione con Swarm

2.1 La programmazione ad oggetti

2.1.1 Precedenti linguaggi: la programmazione imperativa e logica

Il concetto che occorre tenere presente quando si parla di linguaggi di programmazione è che programmare implica *astarre*. La differenza tra un linguaggio e l'altro risiede nel metodo di astrazione utilizzato. Conseguentemente, poiché la complessità di un problema è direttamente collegata con il tipo e la qualità di astrazione applicata, ogni tipo di programmazione sarà più o meno idonea alla risoluzione di specifici problemi.

La programmazione *imperativa*, con linguaggi come Basic, Fortran e C, è stata un primo grande passo nel processo di astrazione tramite personal computer. Essa è strutturata come una serie di istruzioni che permettono il dialogo tra programmatore e macchina. La sequenza di queste istruzioni indica la logica di ragionamento da seguire. Ad esempio, in C, le istruzioni

```
i=2;  
j=i+1;
```

indicano al computer di assegnare 2 alla zona di memoria i e successivamente di mettere in j il valore di i maggiorato di 1. Una diversa sequenza di tali istruzioni cambierebbe interamente la logica del programma.

La struttura che si crea in questo modo, come si nota in figura 2.1, è una struttura ad albero con un percorso principale dal quale si diramano una serie di istruzioni sequenziali o alternative.

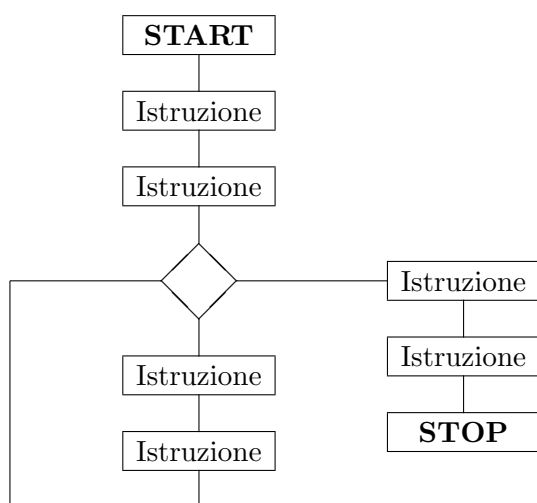


Figura 2.1: esempio di struttura di programmazione imperativa

I linguaggi di programmazione strutturati in questo modo presentano due difetti principali; in primo luogo sono basati su una struttura molto rigida che difficilmente può essere riutilizzata e che è, comunque, molto costoso aggiornare, in secondo luogo il processo di astrazione richiede al programmatore di pensare più in termini di struttura del computer piuttosto che del problema da risolvere.

Una soluzione alternativa alla programmazione tradizionale è la programmazione *logica*, la quale parte dal presupposto che è meglio modellare il problema da risolvere piuttosto che modellare la macchina in funzione di esso. Diverse soluzioni sono state proposte per raggiungere questo scopo ed ogni linguaggio ne persegue una, come: il LISP, che riduce tutti i problemi in *liste*, l'APL, che li riporta tramite *algoritmi*, e il PROLOG, che li proietta in *catene decisionali*.

Ognuno di essi è una buona soluzione per risolvere un problema specifico, ma se ci si sposta dall'ambito per il quale essi sono preposti diventano poco maneggevoli.

2.1.2 La programmazione Object-Oriented e le sue caratteristiche

La programmazione orientata ad oggetti fa un passo ulteriore fornendo al programmatore gli strumenti per rappresentare gli elementi del problema come oggetti nello spazio. Questo al fine di non legare il programma ad una particolare struttura per uno specifico problema. L'idea consiste nel formulare un supporto generale per qualsiasi tipo di problema dal quale partire per effettuare studi più dettagliati. La strada per ottenere questo obiettivo è compiere un grande sforzo di astrazione per formulare il modello del problema da risolvere in modo che esso non vincoli nessun percorso che altri intendano perseguire. Con queste premesse si ottengono programmi molto flessibili che possono essere adattati a qualunque situazione per poi essere corretti o riutilizzati senza costi eccessivi.

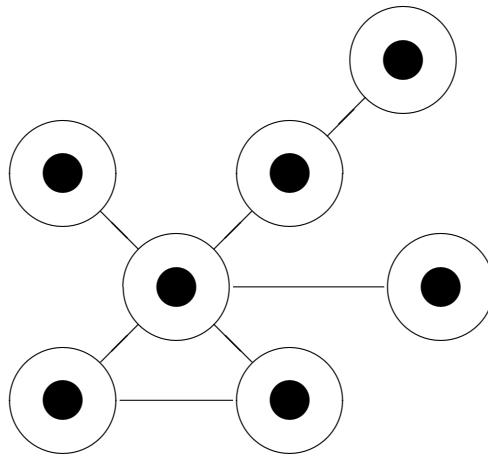


Figura 2.2: esempio di struttura di programmazione ad oggetti

La flessibilità, si veda la figura 2.2, della struttura ad oggetti è data dall'indipendenza dei singoli oggetti e dalle relazioni che li legano, le quali sono facilmente

Tabella 2.1: caratteristiche della programmazione ad oggetti

Proprietà	Object Based (es. Visual Basic)	Object Oriented (es. Java, Objective C)
Astrazione	Si	Si
Classe&Oggetto	Si	Si
Incapsulamento	Si	Si
Ereditarietà	No	Si
Polimorfismo	No	Si

scindibili e ripristinabili. In questo modo se si vogliono apportare cambiamenti ad un progetto oppure riformulare un modello per altre finalità non è detto che occorra riscrivere l'intero codice, come accade per una programmazione tradizionale, ma può essere sufficiente riutilizzare oggetti già creati in precedenza estraendoli dal contesto in cui sono inseriti per inserirli in uno nuovo .

Una struttura come quella rappresentata in figura 2.2 può essere formulata solo riferendosi alle caratteristiche che la programmazione orientata ad oggetti fornisce. Una sintesi di queste peculiarità è riportata nella tabella 2.1.

In questa appare una suddivisione tra due categorie di linguaggi utilizzabili per sviluppare il codice ad oggetti. La prima categoria è definita *Object Oriented* e si distingue perché dispone di strumenti per gestire l'ereditarietà tra gli oggetti, mentre la seconda, definita *Object Based*, si serve di interfacce per simulare il concetto di ereditarietà. Questa premessa serve per illustrare che anche all'interno della programmazione ad oggetti lo sviluppatore deve sapere che vi sono alcune differenze tra i linguaggi che gestiscono oggetti a seconda che essi appartengano alla prima categoria, come Java, Visual C++ o Objective C, piuttosto che alla seconda, come Visual Basic.

Procediamo, ora, con un'analisi più dettagliata delle caratteristiche della

programmazione Object Oriented (O.O.P.).

Astrazione

Il principio di fondo, come già specificato precedentemente, è l'astrazione, ovvero la capacità di esprimere dei concetti senza considerarne i dettagli. Astrarre in programmazione implica scrivere un programma che sia valido per un determinato argomento indipendentemente dai contenuti che dovrebbe rappresentare.

Essa si realizza nella suddivisione delle funzioni da svolgere nel programma in singoli oggetti, ognuno in grado di operare per conto proprio svolgendo il compito per cui è stato preposto. L'obiettivo finale del programma, ovvero il contenuto, si otterrà in un secondo tempo tramite le relazioni che si instaureranno tra i singoli oggetti e che consentiranno il coordinamento dell'operato dei singoli.

Classe & Oggetto

Finora abbiamo parlato di oggetti in termini teorici, riferendoci al loro utilizzo nell'ampio contesto di un modello. Occorre adesso chiarire cos'è un oggetto e come può essere generato.

Innanzitutto, l'**oggetto** è un insieme di *metodi* e di *proprietà*.

I metodi sono classiche sequenze di istruzioni che svolgono una specifica procedura, in altri termini sono l'insieme dei compiti che l'oggetto è in grado di svolgere. Essi circoscrivono una determinata funzione all'interno di un oggetto contraddistinguendolo da tutti gli altri, ciò comporta che ogni oggetto sia in grado di compiere determinate azioni qualora vengano richieste. La programmazione ad oggetti prevede quindi che i medesimi *dialoghino* tra di loro scambiandosi informazioni e dati in base alle capacità dei singoli di elaborarli; in pratica questo viene attuato con delle richieste di attivazione da parte di un oggetto dei metodi contenuti in altri oggetti in grado di svolgere la funzione di suo interesse. In questo modo con una serie di reciproci richiami a rispettivi metodi gli oggetti possono

tessere una fitta rete di legami che può facilmente riprodurre in un modello le interdipendenze di un sistema complesso. Da qui il vantaggio della simulazione di fenomeni complessi su computer, ovvero la possibilità di costruire modelli in scala con la realtà senza scindere interdipendenze tra i soggetti rappresentati con ipotesi forzate.

Le proprietà o variabili sono l'insieme dei dati che l'oggetto è in grado di comprendere e, quindi, di gestire. Le variabili possono essere comuni a tutto l'oggetto o proprie di un particolare metodo, se una variabile è comune a più metodi dell'oggetto dovrà essere dichiarata dentro l'oggetto, ma non in un metodo specifico, se, invece, una variabile è propria di un metodo in particolare dovrà essere dichiarata all'interno dello stesso.

La **classe** è la descrizione di oggetti simili, con le stesse variabili e gli stessi metodi. Gli oggetti, descritti con il codice delle classi, si ottengono *istanziando* le classi stesse ovvero creando un *esemplare* di quella classe. In sostanza, quando si programma in un linguaggio orientato ad oggetti non si definiscono oggetti singoli, ma classi di oggetti; in questo modo si otterranno tanti oggetti che differiscono l'uno dall'altro, ma che hanno tutti degli elementi in comune che li rendono riconoscibili in quanto tra di loro correlati.

Incapsulamento

L'incapsulamento è la capacità di racchiudere-incapsulare all'interno dell'oggetto le informazioni relative alle scelte di progetto. In questo modo all'esterno dell'oggetto traspare solo la sua funzionalità e non le proprietà relative alla sua costruzione. Ad essere incapsulati nell'oggetto sono gli stessi metodi e le stesse variabili già spiegati per descrivere le caratteristiche dell'oggetto, ma quello che ora si vuole sottolineare, si veda figura 2.3, è come le informazioni e i dati utili all'oggetto siano contenuti in esso ed utilizzabili dall'esterno semplicemente tramite l'utilizzo dell'oggetto. Questo aspetto è molto importante poiché l'utilizzatore di

un metodo deve interessarsi esclusivamente dell'interfaccia che gli appare e non delle sue complicazioni interne, questo anche qualora egli debba riutilizzare il metodo in altre applicazioni.

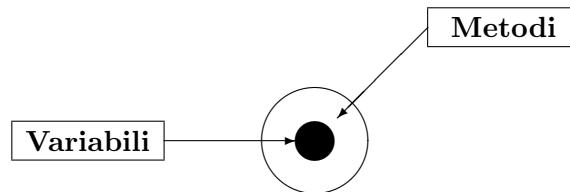


Figura 2.3: incapsulamento

Dalla figura 2.3 emerge anche meglio l'importanza di definire i metodi e variabili come privati, pubblici o protetti. Infatti, poiché essi sono incapsulati dentro l'oggetto è possibile decidere, in sede di programmazione, se consentire all'utente cliente di entrare dentro l'oggetto per modificarlo o esclusivamente di utilizzarlo senza poter modificare ciò che in esso è incapsulato. Questa scelta dipende da chi sviluppa il programma; egli deve decidere se nello sviluppo del progetto possa essere necessario apportare modifiche ad alcune istruzioni, in questo caso definirà il metodo come **public**, pubblico, ovvero gestibile da tutti, oppure se un metodo presenta caratteristiche cruciali per la struttura del programma che non devono essere passibili di modifiche, nel qual caso definirà il metodo **private**, privato, ovvero gestibile solo dall'oggetto che lo contiene. Una via di mezzo tra private e public è l'opzione **protected** che consente l'utilizzo del metodo protetto solo da parte degli oggetti che *estendono*, in base al principio di ereditarietà, l'oggetto in cui esso è contenuto.

Ereditarietà

L'ereditarietà è un sistema di relazionare due o più classi tra loro. Il suo nome deriva dalla descrizione del compito che assume ovvero dalla possibilità di alcune

classi di *ereditare* proprietà e metodi da un'altra classe. La classe dalla quale le altre ereditano è detta *classe root*, mentre le classi che ereditano sono dette *sotto-classi*. Non è raro che, per rimanere legati alla similitudine con l'eredità, la root venga definita classe padre e le sotto-classi classi figlie. Da un punto di vista puramente teorico qualunque classe può ereditare da un'altra a patto di non introdurre cicli dai quali sarebbe poi impossibile uscire. In pratica il principio dell'ereditarietà è utile per procedere ad una *specializzazione* del modello e, quindi, si utilizza quando si vogliono ereditare metodi e proprietà già elaborati, ma che si ha interesse a specializzare in funzione di un modello specifico. Infatti le classi figlie possono:

- aggiungere propri metodi ed attributi.
- modificare metodi ereditati, nel qual caso metodo ereditato e metodo ridefinito dovranno avere lo stesso nome.

In questo modo si viene una struttura ad albero, vedi figura 2.4, tramite la quale per ogni classe è possibile definire l'insieme dei suoi ascendenti, *supertipi* e dei suoi discendenti, *sottotipi*.

Molti linguaggi ad oggetti consentono solo l'*ereditarietà semplice*, per ogni classe un solo padre, che permette di formulare strutture fortemente gerarchiche, ma molto ordinate. Altri linguaggi, più recenti, permettono l'*ereditarietà multipla*, più sotto-classi per una root, che consente maggiore libertà nel modellare. La difficoltà nell'usare l'ereditarietà multipla è avere ben chiaro lo schema da seguire per non cadere in ripetizioni dannose per il programma.

Polimorfismo

Il polimorfismo è la capacità dell'oggetto di rispondere ad un comportamento generico con un'azione specifica. Infatti, a differenza della programmazione tradizionale, nella quale non è possibile avere due funzioni con lo stesso nome o

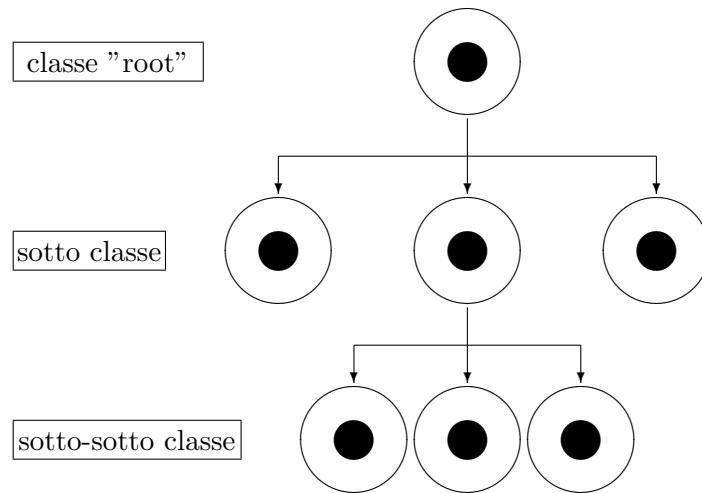


Figura 2.4: ereditarietà

contenere più dati nella stessa variabile, la programmazione ad oggetti consente di riferirsi con lo stesso nome a variabili, metodi od oggetti diversi. Questo consente al programmatore la scrittura di variabili e metodi generici, polimorfici appunto, che potranno essere utilizzati per finalità differenti. Si realizza così l'obiettivo della programmazione ad oggetti di sviluppare un'unico codice per modelli generali applicabile poi a casi specifici. In pratica il polimorfismo è permesso dal fatto che in un programma si possano avere:

- nomi uguali di variabili e metodi in classi diverse.
- funzioni od operatori con lo stesso nome, ma con differente tipo e numero di parametri.
- variabili o puntatori che si riferiscano ad oggetti diversi durante l'esecuzione del programma.

2.1.3 Vantaggi della programmazione ad oggetti

Nell'analizzare quali possano essere i vantaggi nel programmare ad oggetti occorre sottolineare quali si riferiscano ad un suo generale utilizzo e quali all'argomento specifico di simulazione di impresa virtuale.

In termini generali, in riferimento proprio a come è strutturato questo tipo di programmazione, un vantaggio considerevole è la possibilità di reimpiegare il software. La possibilità di riutilizzare classi root e di specializzarle con sotto classi implica una riduzione dei *tempi* di scrittura del codice e, quindi, dei *costi* di sviluppo del modello.

Questo aspetto è cresciuto in importanza soprattutto in questi ultimi anni in cui si è visto l'aumento della domanda di prodotto informatico da parte di molte imprese, quasi indipendentemente dal settore in cui esse operano, sia per lo sviluppo del commercio elettronico sia per la formazione di nuovi strumenti per la gestione dell'impresa. Per le aziende produttrici di software, che si occupano di soddisfare questa domanda, ciò ha significato commesse di pacchetti informatici da soddisfare possibilmente in tempi brevi per eludere la concorrenza. Proprio per esse la programmazione ad oggetti ha significato un notevole passo in avanti, infatti è stato possibile strutturare più progetti sulla stessa struttura di base senza trascurare gli interessi specifici di ogni singolo cliente.

In un secondo tempo emerge anche il vantaggio della semplicità di manutenzione del software. Infatti, consegnato un prodotto, è opportuno procedere al suo aggiornamento, qualora diventi obsoleto, ed al suo ripristino, qualora non funzioni correttamente. Tutto ciò viene agevolato dalla estrema flessibilità della struttura ad oggetti, che consente interventi isolati senza compromettere la struttura globale del prodotto. Nuovamente per le aziende che operano nel settore informatico e che gestiscono programmi molto complicati questo implica minore impiego di personale, quindi minori costi, e minore tempo di attesa del cliente, quindi migliore impatto sul mercato.

Per quanto concerne nello specifico la simulazione di impresa virtuale il vantaggio della programmazione orientata ad oggetti consiste nel fatto che si presta alla creazione di modelli. Infatti, il processo di astrazione richiesto per programmare ad oggetti è molto simile ad uno schema mentale di modellizzazione della realtà. Gli oggetti che possono dialogare tra loro, passarsi dati ed informazioni o ereditare capacità da altri oggetti si prestano ad essere associati con le parti in cui può essere decomposto un sistema complesso. Così nella simulazione di scienze sociali si possono interpretare gli oggetti come i singoli soggetti che si muovono in uno spazio, instaurano relazioni, prendono decisioni. Il tutto con una struttura molto flessibile che consente la formulazione di un modello standard applicabile ad una molteplicità di casi specifici.

2.2 Java

Assodati i vantaggi che derivano dall'utilizzo della programmazione ad oggetti per lo sviluppo di modelli da utilizzare in simulazioni, rimane da compiere la scelta del più appropriato. Una prima selezione è stata necessariamente richiesta dal fatto che per la simulazione di impresa virtuale, si veda il paragrafo 2.3, esiste un pacchetto di librerie, *Swarm*, che presenta peculiarità idonee allo sviluppo di simulazioni di fenomeni sociali agent-based. Queste librerie sono supportate da due soli linguaggi ad oggetti: Objective C e Java. Objective C è stato il primo linguaggio al quale si sono rivolti gli sviluppatori di Swarm, ma di recente è stato affiancato da Java in considerazione del notevole successo che sta vivendo su scala mondiale. Per il modello di impresa virtuale è stato utilizzato Java.

Tale scelta è stata una conseguenza dovuta sia alla maggiore conoscenza del prodotto nell'ambiente informatico, quindi alla maggiore possibilità di utilizzo del modello da parte di terzi, sia per la presenza di vaste librerie per la programmazione grafica e sull'Internet.

Di seguito sono elencate le caratteristiche principali di Java in riferimento al sistema operativo su cui può essere utilizzato, alle applicazioni che consente di sviluppare sul Web ed alle modalità di lavoro del compilatore.

2.2.1 L'indipendenza dalla piattaforma

L'indipendenza dalla piattaforma, ovvero la capacità di un programma di essere eseguito su piattaforme e sistemi operativi differenti, è un grande vantaggio di Java rispetto agli altri linguaggi di programmazione. Quando, vedi Lemay (1998), si compila un programma scritto, ad esempio in C, ma lo stesso accade per la maggior parte degli altri linguaggi, il compilatore traduce il file *sorgente*¹ in

¹Il sorgente o codice sorgente è l'insieme delle istruzioni di programmazione che il programmatore inserisce in un editor di testi quando crea un programma.

*codice macchina*². Se si compila il codice su un computer con un determinato processore il programma potrà funzionare su un altro computer solo se questo avrà lo stesso tipo di processore, in caso contrario sarà necessario trasportare il file sorgente sulla nuova macchina e ricompilare il codice. Spesso si generano ulteriori problemi causati dal fatto che il processore può richiedere alcuni cambiamenti allo stesso file sorgente. Questo procedimento è illustrato in figura 2.5.

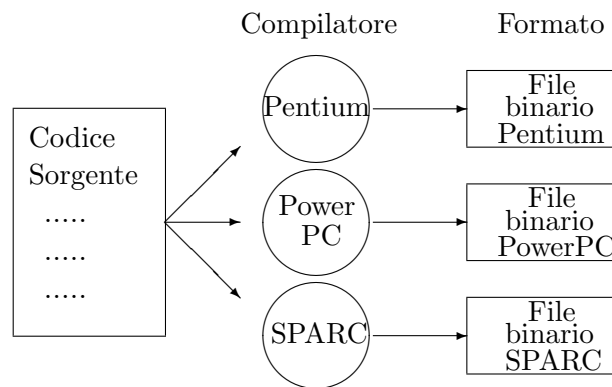


Figura 2.5: programmi compilati tradizionalmente

Java riesce ad evitare questo tipo di problemi grazie all'utilizzo della *Java Virtual Machine* (J.V.M.). La Virtual Machine esegue alcuni compiti del computer all'interno del computer stesso, ovvero preleva i programmi Java compilati e converte le relative istruzioni in comandi gestibili da parte del sistema operativo. In questo modo un programma compilato in Java può essere eseguito da qualunque piattaforma e sistema operativo che possieda una Java Virtual Machine. La J.V.M. esegue, quindi, due funzioni: la prima di codifica, trasformando il programma compilato in un formato *Bytecode* indipendente da piattaforma e processore, la seconda di interprete³ di programmi già compilati.

Il bytecode è simile al linguaggio macchina prodotto da altri linguaggi in sede

²Il codice macchina è un insieme di istruzioni comprensibili dal processore specifico che il computer sta utilizzando

³Per questo la J.V.M. è anche nota come *interprete di Java* o *runtime di Java*

di compilazione, ma non è specifico di alcun processore. In questo modo si ottiene il percorso rappresentato in figura 2.6

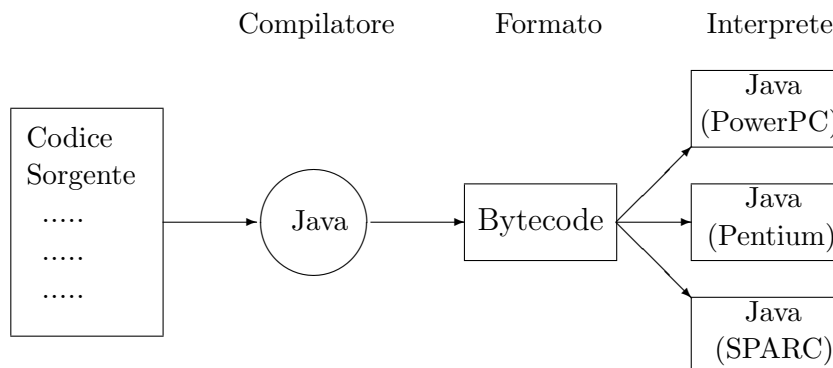


Figura 2.6: programmi compilati con Java

L'aggiunta del passaggio di codifica in bytecode causa non solo vantaggi, ma anche qualche problema. Il principale consiste nel fatto che rende più lente le esecuzioni di programmi scritti in Java rispetto a programmi che compilano per un'unica piattaforma. Quella della velocità di esecuzione è senza dubbio la critica più pesante rivolta a Java, tanto che gli sviluppatori dello stesso, per porvi rimedio, hanno sviluppato dei compilatori *just in time* che eseguono i bytecode ad una velocità decisamente superiore.

2.2.2 Java e l'Internet

Lo sviluppo delle comunicazioni, del commercio e del lavoro sull'Internet ha determinato la crescita di programmi volti a risolvere i problemi dettati dalla sicurezza delle transazioni e dalla gestione dei dati. Naturalmente si sono ottenute molte risposte alle richieste degli utilizzatori della rete da parte di tutte le case produttrici di software, le quali hanno studiato dei metodi per cercare di rendere più veloce ed efficiente il dialogo del *client* con il *server*⁴ sviluppando una pro-

⁴*Client* indica la macchina remota, l'utente che si collega all'Internet per ottenere delle informazioni, le quali gli vengono passate dal *Server*; pertanto un *Server* gestisce la connessione

grammazione sia *Client Side* sia *Server Side*. Java si colloca tra i linguaggi che consentono una programmazione dal lato cliente di interfacce utili per l'Internet grazie agli *Applet*.

L'Applet è un mini-programma che, una volta inserito all'interno di una pagina Web, viene direttamente eseguito dal *browser*⁴. In questo modo quando un client si collega ad un server per ottenere una pagina che contiene un Applet, questo viene direttamente scaricato, come se fosse un grafico, e, qualora attivato dal browser cliente, esegue un programma. In riferimento a quanto già detto in 2.2.1, l'Applet potrà essere tradotto da qualsiasi piattaforma e processore a patto che il browser sia in grado di tradurre il codice compilato dell'Applet.

Il vantaggio per il server che gestisce pagine con Applet è il minore impatto che esse hanno sul suo lavoro, infatti si tratta di un programma inserito come testo in una pagina che verrà eseguito in locale dal client. Inoltre per le versioni di Java successive alla 1.1 è possibile comprimere i file compilati nel formato *.jar* i quali possono essere inseriti come tali nelle pagine per poi essere decompressi in sede di esecuzione.

Un altro aspetto che lega Java all'Internet è relativo alla enorme fonte di librerie che la rete mette a disposizione degli utenti. Nuovamente ritorna l'importanza della flessibilità delle strutture programmate ad oggetti. Infatti, grazie alla possibilità di adattare facilmente programmi già scritti e funzionanti alle proprie esigenze è ragionevole pensare di utilizzare l'Internet come deposito di librerie prelevabili da chi ne abbia la necessità. Questo può essere utile sia per lo sviluppo di progetti comuni tramite la comunicazione diretta tra gli interessati sia per lo sviluppo di progetti diversi, ma che hanno in comune alcune difficoltà da risolvere.

al Web di più *Client*.

⁴Il browser è un software presente sulla macchina remota che consente di visualizzare ed eseguire programmi scritti con linguaggi per l'Internet quali HTML, Java, JavaScript,....

2.2.3 Java e Objective C

Java e Objective C si distinguono sia per differenze di carattere generale che si riferiscono all'operatività del linguaggio sia per differenze di tipo formale in merito alla metodologia di scrittura del codice, vedi Bissey (2000).

Occorre fin da ora precisare che Objective C è il linguaggio nativo di Swarm e che, quindi, la maggioranza delle risorse, ad esempio programmi e tutorials, sono scritte in questo linguaggio. D'altra parte Java si sta sviluppando molto velocemente, per quanto concerne il suo utilizzo con Swarm, ed, in più, offre una vasta gamma di librerie grafiche aggiuntive. Ad oggi però l'integrazione tra i due ambienti non è ancora completa.

Le principali differenze sono le seguenti:

1. In Objective C occorre creare un maggior numero di files in quanto si deve prima salvare un file *header*, interfaccia della classe con estensione *.h*, che contiene l'elenco delle variabili e dei metodi e, poi, un *file.m*, implementazione del *.h*, che contiene il codice che descrive i metodi. L'atto di implementazione avviene nel *file.h* tramite il comando *#import <nome file>.h*. In Java è sufficiente creare un unico *file.java* che contiene elenco e descrizione, occorre solo fare attenzione a salvare il file con lo stesso nome della classe.
2. Java ha una tipizzazione di tipo statico, *static binding*, ovvero richiede sempre di specificare il *tipo* dell'oggetto a cui si riferisce; questo comporta una maggiore attenzione al momento della scrittura del programma, ma anche un controllo immediato in fase di compilazione della correttezza dello stesso. Objective C consente una tipizzazione dinamica, *dynamic binding*, grazie all'utilizzo del protocollo *id* che definisce un oggetto, ma non ne indica il tipo; in questo modo si ottiene maggiore flessibilità in fase di programmazione, ma si rischia di ottenere errori in fase di esecuzione del programma non sottolineati durante la compilazione.

3. In Java è presente un *Gabage Collector* che si occupa di eliminare automaticamente gli oggetti non utilizzati, al fine di alleggerire l'esecuzione del programma, mentre in Objective C è necessario introdurre specifici comandi all'interno del codice.
4. In Java si utilizza un *costruttore* della classe che si occupa di specificare i parametri da passare all'oggetto quando questo viene costruito. Il costruttore è un normale metodo della classe caratterizzato dall'avere il suo stesso nome. L'utilizzo del costruttore rende più immediata e comprensibile la programmazione.

2.2.4 Documentazione di Java

La documentazione in Java, come negli altri linguaggi di programmazione ad oggetti, è molto importante per disporre facilmente delle caratteristiche di packages, classi, metodi e funzioni. Infatti tramite una buona descrizione di ciò che è già stato elaborato si può comodamente risalire ai campi di utilizzo degli oggetti di cui si ha la disponibilità per riutilizzarli nel proprio modello. Inoltre una buona documentazione, che prevede anche commenti descrittivi dei comandi utilizzati, facilita la comprensione dei modelli da parte di chi si occupa di studiarne i risultati.

Java prevede l'elaborazione della documentazione dei propri programmi tramite alcune specifiche dell'HTML integrate con convenzioni di scrittura del programma, che permettono la traduzione del listato in documentazione e la visualizzazione del risultato tramite un browser. La lettura risulta pertanto molto dinamica e consente l'immediato raggiungimento delle informazioni ricercate.

Un ottimo esempio dell'utilità della documentazione di Java è la *documentation* scaricabile dal sito della Sun (www.sun.com) relativa alle peculiarità del

linguaggio in base alla versione di sviluppo cui è arrivato⁵. La versione attuale è la 1.3 che sostituisce la precedente 1.2.2.

Dal momento in cui Swarm è stato sviluppato come una serie di librerie scritte in Java anche per esso è stata prodotta la documentazione relativa, da integrarsi con quella base del JDK, reperibile sul sito *www.swarm.org*.

Pertanto, in accordo con la logica degli sviluppatori di Java può essere utile apprendere come si può sviluppare la documentazione relativa al proprio programma per renderlo facilmente comprensibile da coloro che fossere interessati al suo utilizzo.

Questo può essere eseguito direttamente utilizzando il JDK installato sul proprio computer che dispone del comando *javadoc*. Attraverso *javadoc* vengono lanciate una serie di istruzioni che automaticamente producono una pagina HTML con una struttura prefissata nella quale vengono inseriti i punti chiave del codice (packages, classi, attributi, metodi) ed i commenti ad essi relativi. Il percorso⁶ da seguire prevede di posizionarsi nella cartella in cui sono posizionati i files.java che si vogliono documentare e dare il comando:

... > C : \JDK1.3\BIN\javadoc * .java -d < nome cartella di destinazione > ⁷

Nel caso in cui si voglia documentare un progetto che contiene Swarm, oltre che Java, occorre prima settare la macchina con i seguenti comandi:

⁵La versione di sviluppo di Java si evidenzia dal *JDK, Java Development Kit*, che consente la compilazione e l'esecuzione di programmi scritti in Java. In esso è, inoltre, inclusa una raccolta di librerie utili per la realizzazione di grafica, applet, connessioni e molto altro

⁶Nel descrivere la procedura di creazione della documentazione si fa riferimento all'utilizzo del *Prompt di MS-DOS*, poiché la simulazione di impresa virtuale è stata sviluppata in ambiente *Windows*.

⁷Nel comando si suppone che la versione installata del JDK sia la 1.3 e che essa sia stata installata direttamente sul disco fisso C:.

- in ambiente MS-Dos:

```
set CLASSPATH = \Swarm - 2.1.1\share\swarm\swarm.jar;
```

- in ambiente Unix:

```
export CLASSPATH = /Swarm - 2.1.1/share/swarm/swarm.jar;
```

Per commentare il codice in modo che *javadoc* possa comprendere come trasportarlo in HTML occorre seguire alcuni accorgimenti di scrittura ed inserire comandi specifici.

I commenti al codice

Per produrre commenti all'interno della documentazione del modello è necessario inserire dei commenti all'interno del codice fonte dei files .java.

Javadoc interpreta come commenti tutti i caratteri inseriti tra */***, come apertura, e **/*, come chiusura. Il testo inserito in questo spazio può essere disposto su una o più righe.

La prima frase del commento dovrebbe essere una sintesi delle principali funzioni eseguite poiché il file HTML la posiziona accanto alla parola cui fa riferimento in alto nell'elenco di attributi, funzioni e metodi. La totalità del commento viene riportata, invece, alla base della pagina dove c'è la descrizione dettagliata di ciò che è stato elencato sopra.

Considerando che i commenti sono tradotti in HTML, per essere visualizzati dal browser, è possibile ottenere una formattazione del testo in base alle proprie esigenze inserendo all'interno del commento gli appropriati *tag*⁸ HTML. Ad esempio per andare a capo si può utilizzare *< BR >* oppure per sottolineare una parola in grassetto occorre racchiuderla tra il *tag di apertura < B >* e *tag di*

⁸I *tag* sono i comandi compresi dal linguaggio HTML caratterizzati dall'essere racchiusi tra *<* e *>*.

chiusura `< /B >`. Non bisogna, però, utilizzare *tag* di intestazione come `< h1 >` o `< h2 >`, infatti poiché javadoc crea un documento già strutturato l'inserimento di altri comandi strutturali potrebbe creare contrasti.

I commenti alla classe o interfaccia

Javadoc interpreta il simbolo `@` come *tag* per il quale è già stata prevista una specifica formattazione. `@` deve essere accompagnato da alcune parole chiave che definiscono argomenti ritenuti importanti per documentare il codice. Si possono inserire i seguenti comandi:

- **@author** *name-text*: specifica chi ha scritto il codice di quella classe o interfaccia, è possibile inserire più tag di questo tipo.
- **@version** *version-text*: indica la versione di Java utilizzata in riferimento al JDK, è opportuno inserire solo un tag di questo tipo.
- **@see** *classname*: aggiunge la scritta **See also** nella documentazione con l'intento di sottolineare un argomento correlato. Dopo *classname* è possibile specificare anche gli attributi, i metodi o i costruttori a cui si fa riferimento utilizzando il tag `#`. Ad esempio se si vuole mostrare un legame con il metodo *addFirst* della classe *swarm.collections.ListImpl*, il commento sarà:

```
/**
 * @see swarm.collections.ListImpl#addFirst
 */
```

ed il risultato:

See Also:

ListImpl.addFirst(java.lang.Object)

Nel caso in cui siano presenti più metodi o costruttori con lo stesso nome è possibile sottolineare quello a cui si fa riferimento inserendo dopo il nome del metodo o costruttore la lista degli argomenti che lo contraddistinguono tra parentesi. Ad esempio:

```
/*
 *
 *@see java.lang.Object#wait(int)
 */
```

Se si facesse, invece, riferimento ad un package intero, ma con interesse solo ad alcuni files di questo si può inserire normalmente il richiamo ad un file e, dopo, tra parentesi, separati da uno spazio, l'elenco dei files desiderati. Ad esempio, qualora si vogliano mostrare esclusivamente i collegamenti a *File* e *String* del package *io* si può scrivere:

```
/*
 *
 *@see java.io.File#File(java.io.File, java.lang.String)
 */
```

Infine è possibile inserire un collegamento interno ad una pagina HTML che contenga maggiori dettagli su un argomento semplicemente utilizzando il comando HTML per i *links*. Ad esempio, se ulteriori informazioni sul codice sono contenute nella pagina *spec.html* si può dare il comando seguente:

```
/*
 *
 *@see <ahref = "spec.html" > JavaSpec </a >
 */
```

ottenendo come risultato la figura 2.7

- **@since** *since-text*: serve per indicare che modifiche e aggiornamenti al file

```
public class Unit
    extends swarm.objectbase.SwarmObjectImpl
```

See Also:
[Java Spec](#)

Figura 2.7: esempio di link nella documentazione

sono state apportate fino alla versione indicata nel *since-text*, come accade per il JDK.

- **@deprecated** *deprecated-text*: è un commento che indica che un API non è più usato. Solitamente si sottolinea anche da cosa è stato sostituito con la frase *@deprecated Replaced by*

I commenti a metodi e costruttori

Utilizzano tutti gli *@see tag* e in più:

- **@param** *parameter-name description*: aggiunge un parametro alla sezione *Parametres*, la descrizione può essere effettuata sulla linea successiva.
- **@return** *description*: aggiunge la sezione *Return* che descrive il valore di ritorno.
- **@exception** *fully-qualified-class-name description*: aggiunge la sezione *Throws* che contiene i nomi delle eccezioni che possono essere lanciate dal metodo.

Ulteriori informazioni possono essere raccolte sul sito della Sun all'indirizzo:
<http://java.sun.com/products/jdk/1.1/docs/tooldocs/win32/javadoc.html>

2.3 Swarm

Langton (1996) afferma che:

Swarm is a multi-agent software platform for the simulation of complex adaptive system.

Questa frase descrive esaurientemente *Swarm* e tutto quello per cui viene utilizzato.

Innanzitutto, è un software che, per ora, è stato scritto con due linguaggi di programmazione orientati ad oggetti: Java ed Objective C. Consiste in una raccolta di librerie nelle quali sono contenute le classi e le interfacce necessarie per strutturare modelli basati su agenti. Infatti, in Swarm l'unità base per una simulazione è uno *swarm*, ovvero uno sciame, di agenti tramite i quali è possibile costruire modelli che consentono la simulazione di sistemi complessi.

2.3.1 Swarm: struttura base di una simulazione ad agenti

La struttura base di una simulazione con Swarm si compone molto semplicemente da un insieme di *agenti* dove con agenti si intende *attori* in un *sistema*. Definizioni più dettagliate non sarebbero appropriate poiché non si impone alcun vincolo alla caratterizzazione degli agenti o dell'ambiente in cui vengono collocati, lasciando così un'ampio spettro di alternative per la costruzione di modelli in qualunque campo di studi.

Definire un agente implica determinarne il *comportamento* ovvero fissarne le regole d'azione; tali regole possono essere sia *interne*, peculiari del suo essere agente differente dagli altri, sia *esterne* dettate dall'ambiente in cui opera. In questo modo gli agenti sono sia attori "*attivi*" in quanto con le proprie decisioni possono influenzare il sistema, e con esso tutti coloro che ne fanno parte, sia attori "*passivi*" in quanto essi stessi parte del sistema. In alternativa all'essere

un contenitore di regole che rispondono a specifici stimoli un agente può anche contenere un *swarm* di altri agenti consentendo l'elaborazione di strutture ad annidamenti volte a rappresentare realtà anche molto complesse.

Una volta definiti i singoli agenti ed assegnate ad ognuno specifiche caratteristiche è possibile formulare modelli *agent based* impostando una rete di legami che mettono in relazione gli agenti tra di loro e con l'ambiente circostante.

Il passaggio da modello a simulazione richiede un passo ulteriore ovvero la definizione della successione di azioni all'interno del modello. Swarm tratta l'intercalare degli eventi tramite uno *schedule* ovvero una struttura che combina le azioni nello specifico ordine in cui esse devono essere eseguite. Grazie all'impostazione di un *orologio* che determina *quando* le azioni degli attori del sistema devono essere compiute è possibile far rispettare al modello la corretta sequenza degli eventi in analogia con quanto accade nella realtà simulata. Non solo, è anche possibile tenere sotto controllo il rapporto causa-effetto di determinate azioni con l'opportunità di valutare alternative alla sequenza impostata.

Una simulazione in Swarm impostata con queste caratteristiche di base è rappresentabile come in figura 2.8.

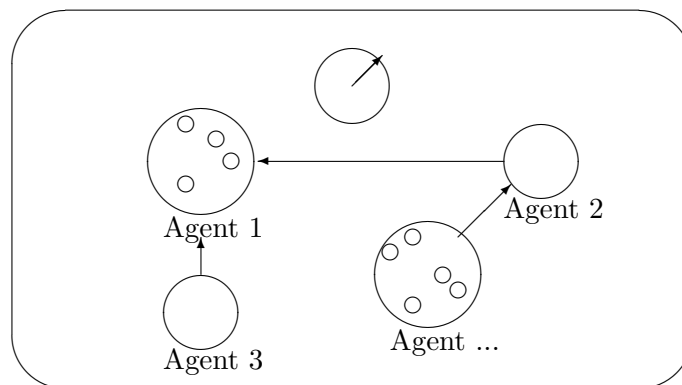


Figura 2.8: schema base di una simulazione con Swarm

Questa struttura di base può essere ulteriormente complicata con l'inserimento

di oggetti che migliorino la rappresentazione grafica e strutturale del modello elaborato. Tali oggetti sono presenti all'interno delle librerie di Swarm e, pertanto, direttamente adattabili alle proprie esigenze.

Passare ad una struttura più elaborata e dettagliata è opportuno nel caso in cui si rappresentino realtà molto complesse dove per numero di agenti, eventi e relazioni sia necessario fare chiarezza nell'elaborazione del modello. L'analisi degli strumenti di Swarm procede, pertanto, con lo studio delle principali librerie e delle loro funzioni.

2.3.2 Struttura della simulazione in Swarm con *Model* ed *Observer*: librerie *simtoolsgui* e *objectbase*.

L'aggiunta delle classi *Model* ed *Observer* consente di perfezionare la struttura della simulazione separando l'aspetto grafico da quello strettamente inerente agli eventi generati dagli agenti. L'*Observer* si occupa di gestire le interazioni tra utente e modello, *Model* che è contenuto al suo interno. In sostanza, l'idea consiste nel far partire la simulazione da un *osservatore* esterno che idealmente guarda dentro al modello per osservare le azioni degli agenti che vi operano.

Questa struttura è rappresentabile come in figura 2.9.

Model ed *Observer* non sono strettamente legati, è possibile anche generare una simulazione che preveda esclusivamente l'impiego del *modello* e trascuri l'aspetto grafico dei risultati raggiunti. Questa situazione si può ottenere semplicemente saltando il passaggio di creazione dell'*osservatore*. Nell'analisi del *main()*, metodo che si occupa di avviare la simulazione in Java, si affronteranno entrambi i casi di avvio di una simulazione con o senza *Observer*.

Il *main()*

La simulazione parte da una classe che convenzionalmente, per facilitare la lettura del codice, viene nominata con la parola *start*, ma che obbligatoriamente contiene

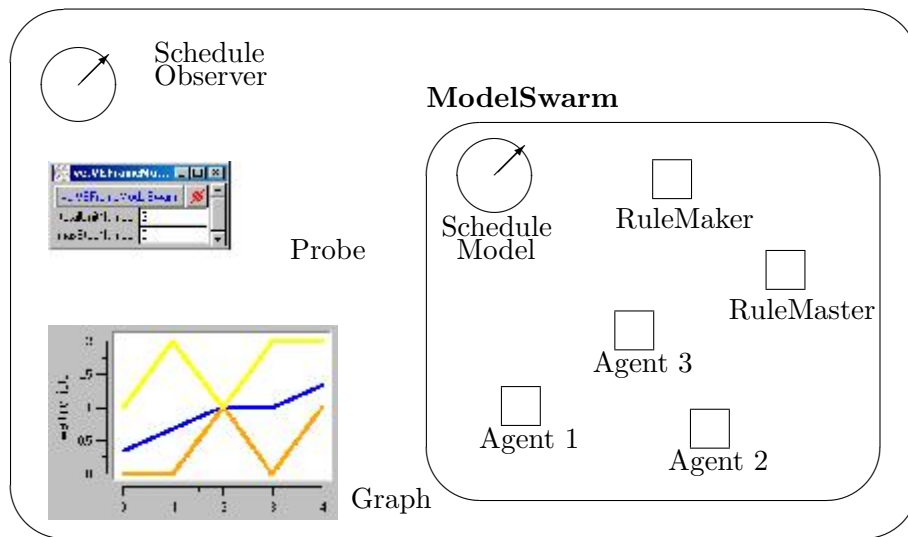
ObserverSwarm

Figura 2.9: schema di funzionamento di una simulazione con Swarm

il metodo:

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

Il *main()* è il metodo che il compilatore Java ricerca quale punto di partenza per l'esecuzione del programma. In esso occorre, nel caso di un modello elaborato con Swarm, generare gli oggetti che avviano la simulazione ed indicare la presenza dell'ambiente Swarm.

Infatti, la prima istruzione che occorre inserire all'interno del metodo *main()* è:

```
Globals.env.initSwarm ("...", "...", "...",args);
```

la quale si occupa di impostare le variabili ed i metodi "globali" dell'ambiente Swarm. Ad esso viene passato, come ultimo parametro, il vettore stringa di argomenti che consente di iniziare l'esecuzione del programma. I primi tre parametri richiesti permettono di segnalare, sotto forma di stringa: il nome del programma, la versione a cui è giunto ed un indirizzo a cui mandare eventuali segnalazioni sul funzionamento.

Dopo aver inizializzato Swarm, il *main()* genera la classe che lancia la simulazione. Le situazioni principali che possono crearsi a livello di struttura di programmazione di una simulazione con Swarm sono dovute alla presenza o meno di una classe *observerSwarm* e di una classe *modelSwarm*.

Nel caso in cui vi sia un *observerSwarm*, il *main()* dovrà *puntare* ai metodi dell'observer e poi eseguirlo con l'istruzione:

```
observerSwarm.go();
```

e fermarlo con l'istruzione:

```
observerSwarm.drop();
```

Entrambi i metodi, *go()* e *drop()*, non appartengono all'*observerSwarm*, ma vengono ereditati dalla classe *swarm.simtoolsgui.GUISwarmImpl*.

Con questa procedura, poiché l'*observerSwarm* viene costruito come contenitore esterno di tutti gli oggetti della simulazione, si eseguono tutti gli oggetti del modello.

Nel caso in cui non vi sia l'*observerSwarm*, ma solo il *modelSwarm*, la simulazione comincia con il comando:

```
(modelSwarm.getActivity()).run();
```

Il metodo *getActivity()* contiene le informazioni relative all'oggetto *Activity*, costruito nel *modelSwarm*.

L'ObserverSwarm

L'*observer*, si veda figura 2.9, è il contenitore esterno degli oggetti della simulazione; la sua funzione è quella di attivare il modello e di facilitare il dialogo dello stesso con il suo utilizzatore. Le funzioni grafiche devono essere ereditate da *GUISwarmImpl*, che appartiene alla libreria **simtoolsgui**. Essa genera automaticamente un pannello di controllo, si veda figura 2.10, e definisce un metodo *go()* che interpreta lo stato del pannello in risposta alle esigenze dell'utilizzatore.



Figura 2.10: menù di avvio della simulazione in swarm

Inoltre, in quanto sotto classe di *GUISwarm*, l'*observerSwarm* implementa lo stesso tipo di metodi per costruire oggetti, azioni ed attivarsi; tramite i comandi:

- *buildObjects()*: metodo nel quale vengono creati tutti gli oggetti utilizzati dall'*observer* per proseguire la simulazione; in esso vengono creati il *model* ed i grafici.
- *buildActions()*: metodo nel quale si richiamano le azioni degli oggetti precedentemente creati e si stabilisce in quale momento esse debbano essere eseguite; a questo livello viene inserito lo *schedule*, si veda 2.3.3, che nell'*observerSwarm* viene utilizzato per regolare l'aggiornamento dei grafici in base ai risultati raggiunti dal modello.
- *activateIn()*: metodo con la funzione di "incollare" tutto quello precedentemente fatto in modo da poter passare al *main()* un'unica istruzione da eseguire.

Il ModelSwarm

L'oggetto *modelSwarm* eredita le sue principali caratteristiche dalla classe *SwarmImpl*, appartenente alla libreria **objectbase**. *SwarmImpl* costituisce l'implementazione dell'interfaccia *Swarm* che mette a disposizione i seguenti metodi, analoghi a quelli già descritti per l'*observer*:

- *buildObjects()*: metodo nel quale vengono creati tutti gli oggetti utilizzati dall'*model* per proseguire la simulazione. In esso vengono creati gli oggetti che contengono i metodi necessari per definire le azioni della simulazione.
- *buildActions()*: metodo nel quale si definisce quali azioni la simulazione deve compiere e la sequenza da rispettare. Questo con l'utilizzo integrato dello *schedule*.
- *activateIn()*: metodo con la funzione di "incollare" tutto quello precedentemente fatto in modo da poter passare al *observer* un'unica istruzione da lanciare.

La struttura del *model* è molto simile a quella dell'*observer*, la differenza consiste nella sostanza degli oggetti trattati. L'*observer* si occupa dell'aspetto grafico, mentre il *model* gestisce la parte operativa della simulazione.

2.3.3 La struttura con *Schedule*, *Probe* e *Liste*: librerie *activity*, *objectbase* e *collections*

La gestione del tempo: lo *schedule*

Swarm, forte delle peculiarità della programmazione ad oggetti, consente una agevole gestione degli eventi nel tempo, a differenza della programmazione tradizionale nella quale era richiesta una complicata combinazione di cicli *for()*.

La libreria che provvede a fornire funzioni e metodi per una corretta gestione della simulazione è denominata **activity** e le classi principali a cui fa riferimento sono *ActionGroupImpl*, *ScheduleImpl* e *Activity*. Tramite questa libreria è possibile creare un "orologio" della simulazione; idealmente quando le "lancette" dell'orologio raggiungono determinate intervalli di tempo accadono azioni predefinite.

La creazione dell'"orologio" avviene tramite la classe *ScheduleImpl* ed l'istruzione:

```
ScheduleImpl schedule = new ScheduleImpl(getZone(), 1 );
```

In questo modo è stato creato l'oggetto *schedule* al quale vengono passati due parametri: il primo relativo alla zona di memoria da assegnargli ed il secondo, un intero, relativo agli intervalli di tempo da indicare sull'orologio. In questo caso è stato costruito uno *schedule* con un unico intervallo di tempo intercorrente tra la una esecuzione degli eventi e la successiva. Qualora si sia in presenza di una simulazione complessa in cui le azioni compiute dagli agenti si svolgano tutte in periodi differenti è sufficiente aumentare gli intervalli di tempo dello *schedule* in funzione degli eventi da eseguire.

Occorre ora precisare quali siano le azioni che il modello deve tenere in considerazione per la loro esecuzione in un determinato intervallo temoreale; si fa ora riferimento alla libreria *ActionGroupImpl* ed alle istruzioni:

```
ActionGroupImpl actions = new ActionGroupImpl(getZone ());
actions.createActionTo$message(...);
actions.createActionForEach$message(...);
...
```

con esse è possibile raggruppare nell'oggetto *actions* tutti gli eventi che caratterizzano il modello; quindi, sarà sufficiente passare l'oggetto *actions* allo *schedule* affinché tali azioni vengano eseguite. L'istruzione necessaria è la seguente:

```
schedule.at$createAction (0, actions);
```

Due sono i parametri richiesti dal metodo *at\$createAction*: il primo l'intero relativo al momento in cui eseguire le azioni ed il secondo relativo alle azioni stesse da compiere.

Nella gestione degli eventi, in particolare nel connubio tra creazione delle azioni e definizione dell'intervallo di esecuzione assume molta importanza la classe *Selector*. Questa classe, che deriva dall'impostazione dell'Objective C, è figlia della libreria principale **swarm** e serve quale contenitore di un metodo appartenente ad un'altra classe. Viene utilizzata nella gestione dello schedule per determinare quale metodo deve essere lanciato nel momento richiesto. Il costruttore di *Selector* è il seguente:

```
public Selector(Class theClass, String theMethodName, boolean theObjc-
    Flag)
```

in esso si richiede di specificare la classe di riferimento, con l'istruzione, se non ho l'istanza della classe:

```
Class.forName("<nome classe >")
```

oppure, se ho già creato un esemplare:

```
<istanza>.getClass()
```

inoltre si richiede di indicare un metodo della medesima classe sotto forma di stringa ed, infine, un parametro boolean senza una funzione specifica.

Questa procedura, medesima per ogni simulazione, consente una semplice e logica gestione degli eventi in successione, in quanto una volta stabilita la suddivisione del tempo è sufficiente imputare gli eventi ad ogni intervallo trascorso.

Le sonde di Swarm: i *probes*

Swarm consente un'agevole interazione tra utente della simulazione e modello grazie all'impiego di sonde, *probes*, che permettono l'inserimento di variabili chiave direttamente da un'interfaccia di dialogo sullo schermo.

La libreria che consente l'inserimento di sonde è **objectbase**. Metodi e funzioni sono riconducibili principalmente alla classe *EmptyProbeMapImpl* ed alle interfacce *VarProbe* e *MessageProbe*. In Java occorre definire una classe che erediti le proprietà di *EmptyProbeMapImpl*, tramite la quale, con il comando:

```
addVar (".....");
```

è possibile aggiungere variabili alla sonda. L'utente del modello può in questo inserire i valori relativi a questa variabili senza accedere al codice del programma.

Nella struttura prevista da *JavaSwarm* la sonda può essere costruita in due modi: con una sottoclasse "locale" oppure all'interno del costruttore della classe cui si riferisce.

Una sottoclasse locale è un tipo speciale di classe interna di Java. Si distingue perché è racchiusa all'interno delle parentesi graffe della classe che la contiene e perché non deve essere tipizzata come pubblica, privata o protetta, ma semplicemente deve essere definita: *class*. Nel caso della costruzione di *probes* occorre ereditare tutte le caratteristiche fornite da *EmptyProbeMapImpl* in modo da poter definire una serie di metodi *privati* che consentano l'aggiunta di variabili e messaggi. Una caratteristica relativa a questa procedura è che il compilatore di Java genera un apposito *file.class* per le classi locali.

In alternativa si può costruire un oggetto di tipo *EmptyProbeMapImpl* all'interno del costruttore della classe di riferimento ed utilizzarlo per richiamare i metodi che gestiscono le fusioni della sonda.

Le sonde, in base alla struttura di Swarm, devono essere inserite nell'*observer*, per migliorare la gestione dei grafici, e nel *model*, per modificare i valori di input

della simulazione direttamente dalle finestre sullo schermo del computer. Nulla vieta di utilizzarle in altre parti del modello per renderlo più chiaro ed interattivo.

Ad esempio all'interno dell'*observer* è opportuno inserire una sonda che regoli l'aggiornamento dei grafici, ovvero che indichi dopo quanti cicli di simulazione si intenda rappresentare a video i risultati raggiunti. Il risultato di questa operazione sarà una finestra, come quella mostrata in figura 2.11, contenente una variabile che consente di apportare le modifiche desiderate.

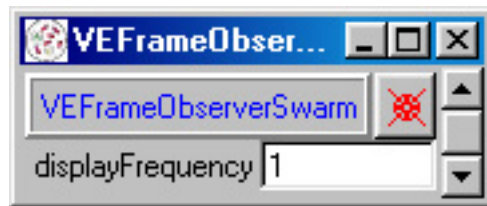


Figura 2.11: esempio di *probe* nell'*observer*

Le liste

Le liste in Swarm sono raccoglitori che svolgono una funzione analoga a quella dei vettori, per l'utilizzo dei quali occorre rifarsi al codice Java, ma con il grande vantaggio di essere *dinamiche* e di poter contenere puntatori ad oggetti.

Per la programmazione ad oggetti queste caratteristiche sono molto utili poiché consentono di creare un unico oggetto, e non la totalità, e di inserirlo più volte all'interno di una lista. Una volta completata la lista non si opera più sul singolo oggetto, ma sulla lista stessa.

Le libreria fornita da Swarm per la gestione delle liste è **collections** e precisamente dalla classe *ListImpl*.

Collections contiene le classi necessarie alla strutturazione delle liste, come i metodi di inserimento o rimozione degli oggetti dalle stesse.

Bibliografia

- [1] Bissey M. E. (2000), *Una piccola introduzione a Swarm: Objective C e Java*,
<http://polis.unipmn.it/alex/activities/corso.html>
- [2] Eckel B. (2000), *Thinking in Java: 2nd Edition*, www.BruceEckel.com.
- [3] Lemay L. e Cadenhead R. (1998), *Java 1.2: guida completa*, Milano, Apogeo.
- [4] Minar N., Burkhart R., Langton C. e Askenazi M. (1996), *The Swarm Simulation System: a Toolkit for Building Multi-Agent Simulation*,
www.swarm.org/projects/swarm

Capitolo 3

La struttura aziendale ed il suo sviluppo: implicazioni sul modello virtuale

3.1 Considerazioni introduttive

I fattori chiave del successo di un'impresa che opera su mercati produttivi o finanziari risiedono nella capacità di adattamento che essa riesce ad ottenere con l'ambiente che la circonda e, pertanto, influenza. Si parte sostanzialmente dal presupposto che la struttura aziendale deve essere pensata non solo in accordo con le necessità specifiche richieste dalla produzione del prodotto-servizio di cui si occupa, ma anche in funzione delle spinte di fattori esterni ad essa, quali rapporti di fornitura, di distribuzione e di concorrenza. Solo un'adeguato adattamento a tali condizionamenti consente di migliorare il rapporto tra costi e ricavi dell'azienda.

La difficoltà nel considerare queste influenze esterne risiede nella forte mutevolezza delle stesse che rende l'ambiente molto *dinamico* e pertanto difficile da assecondare nei suoi cambiamenti. Questo implica innanzi tutto che la struttura

aziendale deve essere *flessibile*; obiettivo non facile da perseguire e che presuppone la capacità dell'impresa di *decidere* in tempi brevi anche su questioni molto importanti.

Il *tempo* diventa così un altro fattore determinante per il successo di un'impresa. I ritmi imposti dal mercato sono serrati ed occorre rispettarli se non si vuole essere sopraffatti dalla concorrenza. Si pensi alle attuali strategie di marketing che impongono non più solo di soddisfare le richieste dei cliente, ma di creare aspettative anticipando i desideri del mercato; l'attuale concetto di competizione si basa su una gara di tempi per raggiungere il cliente, come sostengono Stalk e altri (1992) la competizione è:

(...) "*wave of movement*" in which success depends on the anticipation of markets trends and the quick response to changing customer needs.

Questa realtà complessa che permea l'ambiente in cui le imprese moderne operano complica le scelte gestionali in merito a quale sia la struttura interna più idonea all'integrazione dell'impresa nell'ambiente esterno. Il problema in sostanza risiede nell'abilità manageriale di riuscire a comprendere quale sia la struttura migliore per la propria impresa non solo in riferimento a ciò che essa produce, ma soprattutto in relazione al *target* che intende colpire ed alla *concorrenza* che deve affrontare. Da notare come spesso le uniche variabili che l'impresa è in grado di influenzare siano quelle interne ad essa, mentre le altre occorra considerarle come date ed adattarsi in funzione dei loro cambiamenti. La difficoltà di decidere in un ambiente così dinamico e vincolante ha comportato la formulazione di differenti strategie aziendali specifiche per ogni singola situazione.

A sostegno di queste scelte manageriali sono stati proposti alcuni modelli *agent-based* con lo specifico intento di simulare i possibili effetti che determinate decisioni avrebbero comportato; l'obiettivo è quello di *prevenire* risultati inattesi ed indesiderati.

La scelta sull'impostazione da seguire nella costruzione del modello varia in base al problema aziendale che si intende affrontare: alcuni modelli raccolti in Shaw e altri (1996), per i quali si veda il paragrafo 3.3, si basano su "the Order Fulfillment Process" (OFP) in quanto uno dei principali obiettivi della Supply Chain Network, altri presentati in Scacchi e Mi (1997) sono improntati su "Process Life Cycle Engineering" e quindi sulla suddivisione per processi dell'attività produttiva.

In questo contesto è maturato anche il modello di *Virtual Enterprise* il cui obiettivo racchiude molti degli aspetti presentati in questi modelli. Infatti, l'intento è quello di formulare un'impresa nella sua globalità che consideri tutti gli aspetti decisionali che possono essere coinvolti nell'attività di impresa ovvero tutti i passaggi che devono essere compiuti affinché le risorse in entrata si trasformino in prodotti in uscita. Con il risultato di valutare, pesando adeguatamente ogni passaggio, quali vantaggi o svantaggi economici possano derivare da una determinata gestione imprenditoriale.

Creare un modello virtuale in questo ambito, tenendo presente che esso non viene studiato e creato per un'impresa specifica, comporta la difficoltà di *generalizzarlo* al punto da renderlo adattabile a qualunque situazione. Occorre specificare che un modello generale non intende sminuire le caratteristiche peculiari delle singole imprese, anzi si propone con una struttura molto flessibile atta a recepire qualunque tipo di informazione riguardi il caso specifico preso in analisi.

La struttura del modello deve, pertanto, essere in grado di riprodurre le scelte in merito al collocamento dell'impresa in una catena del valore, in merito alle relazioni dell'impresa con clienti e fornitori, in merito alle scelte di integrazione verticale ed all'*Information Technology*. Il primo passo che può essere mosso verso la costruzione del modello consiste nell'attuare un'analisi delle variabili aziendali che possano incidere nel fissare la struttura del modello.

3.2 La struttura aziendale

Nell'analisi della struttura aziendale occorre distinguere quali siano le variabili interne all'impresa stessa e quali, invece, agiscano su di essa, ma provengano dall'ambiente esterno. In entrambi i casi l'attività di impresa subisce delle influenze di cui deve tener conto per operare opportune ed adeguate scelte decisionali. La differenza consiste nel fatto che: per quanto riguarda le variabili interne l'impresa ha il potere di incidere su di esse modificandone gli effetti, mentre sulle variabili esterne, se si escludono rari casi, l'impresa non ha alcun potere.

3.2.1 La struttura aziendale: rapporti interni

Per quanto concerne la caratteristiche proprie dell'azienda, gli spunti di maggiore interesse nella costruzione del modello derivano dalle seguenti operazioni:

1. scomporre l'azienda nelle sue parti principali ed evidenziare per ognuna di esse: caratteristiche, ruoli, legami e proprietà
2. valutare quali sono i metodi di comunicazione interni dell'impresa; quindi evidenziare come le singole unità emettono *output* e da chi ricevono *input*
3. valutare l'apprendimento dell'azienda durante i singoli processi aziendali

Pertanto, procedendo un po' più nel dettaglio, è opportuno comprendere che l'azienda è composta da molte unità operative, ognuna con un proprio obiettivo ed un proprio comportamento; i quali, spesso, non sono determinati dalle unità stesse, ma da coloro che operano prima o dopo di esse. Questa consapevolezza è determinante per un duplice motivo: in primo luogo, per assegnare ad ogni *agente* del modello le caratteristiche che lo rendono il miglior clone possibile della realtà che rappresenta, in secondo luogo per valutare se tali caratteristiche si integrano correttamente nel *processo produttivo* dell'azienda. Al fine di rappresentare queste unità operative all'interno del modello è opportuno definire alcuni termini

che ben sintetizzano il procedimento che si vuole descrivere. Quindi, secondo l'interpretazione di Shaw e altri (1996) consideriamo:

- *business unit*, un'unità dell'azienda che, in base al livello di astrazione utilizzato, può rappresentare dall'intera organizzazione, alto livello di astrazione, al singolo team di sviluppo, basso livello di astrazione;
- *processo*, un'attività qualunque all'interno dell'impresa, ma che rappresenti una successione logica e ben integrata di attività;
- *input - output*, le informazioni o i prodotti, in base al contesto in cui ci si trova, che entrano od escono dall'impresa;
- *relazioni - interazioni*, gli input/output che intercorrono tra le varie business unit

3.2.2 La struttura aziendale: rapporti esterni

Nel suo operare l'azienda si trova inevitabilmente a confronto con realtà diverse dalla sua, ma con le quali deve necessariamente convivere. Un adeguato rapporto con esse può permetterle una migliore organizzazione interna e, conseguentemente, maggiore *efficienza*, nelle sue funzioni operative, e maggiore *efficacia*, in termini di profitto.

E' necessario, quindi:

1. considerare il processo produttivo che il mercato prevede per uno specifico prodotto e valutare quale posizione assumere al suo interno
2. valutare coloro che operano "a monte" dell'azienda e le forniscono gli *input*: i *fornitori*
3. valutare coloro che operano "a valle" dell'azienda per i quali essa produce *output*: i *clienti*

Il modello di impresa virtuale deve essere in grado di collocare la *Virtual Enterprise* in qualunque fase del processo produttivo del prodotto ed, in funzione di questo posizionamento, di valutare il miglior compromesso nelle relazioni con clienti e fornitori.

Gli schemi che occorre considerare per rappresentare tutte queste situazioni sono: *value chain*, *supply chain* e *customer relationship management*.

Value chain

La catena del valore può essere vista come una successione di processi che trasformano in sequenza gli input in output. E' importante osservare che la catena del valore va considerata dal punto di vista del prodotto, pertanto essa non indica un'impresa o una successione di imprese, bensì quali sono i passaggi che il prodotto deve compiere per acquisire valore. Questo processo può poi essere svolto da una o più aziende in base alla *struttura organizzativa* delle stesse, ma questo non modifica la catena del valore del prodotto.

Un esempio semplificato riferito alla value chain del prodotto "personal computer" è riportato in figura 3.1.

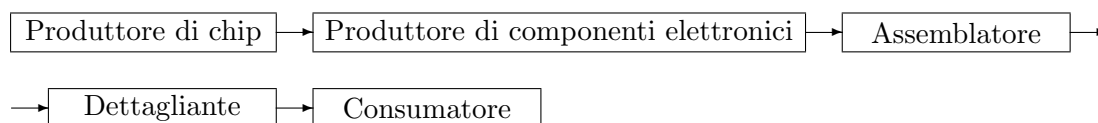


Figura 3.1: rappresentazione grafica della **value chain**

In 3.1 viene riportato un esempio lineare di catena del valore, ma non è detto che sia sempre così. Ad esempio, il produttore di circuiti potrebbe ricevere due tipi di input, I^1 e I^2 , che a seguito del processo di trasformazione potrebbero dar vita a due distinti output, O^1 e O^2 , e, quindi, a due distinte catene del valore.

Supply Chain

La *supply chain* sposta nuovamente l'ottica del problema dal prodotto al produttore, infatti prende in considerazione quale azienda si occupi di produrre un determinato *output* e da chi riceva le risorse necessarie. In sostanza, mostra quali sono i rapporti dell'azienda con i propri fornitori.

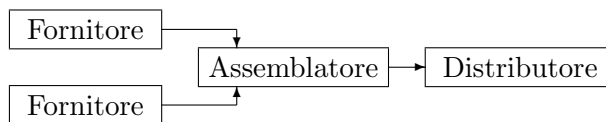


Figura 3.2: rappresentazione grafica della **supply chain**

La struttura della *supply chain* influenza la catena del valore di un prodotto in quanto mostra quali sono i rapporti tra i soggetti coinvolti in un determinato processo produttivo. In base a questo si può dedurre che l'unione di più catene di fornitura in una simbolica *supply chain network* restituirebbe la catena del valore con una rappresentazione dettagliata dei passaggi delle singole risorse. Questo accade poiché una rete di catene di forniture prevede che ogni produttore sia cliente dell'impresa che lo precede e fornitore di quella che lo segue. Un esempio di una parte di *supply chain network* è rappresentato in figura 3.2.

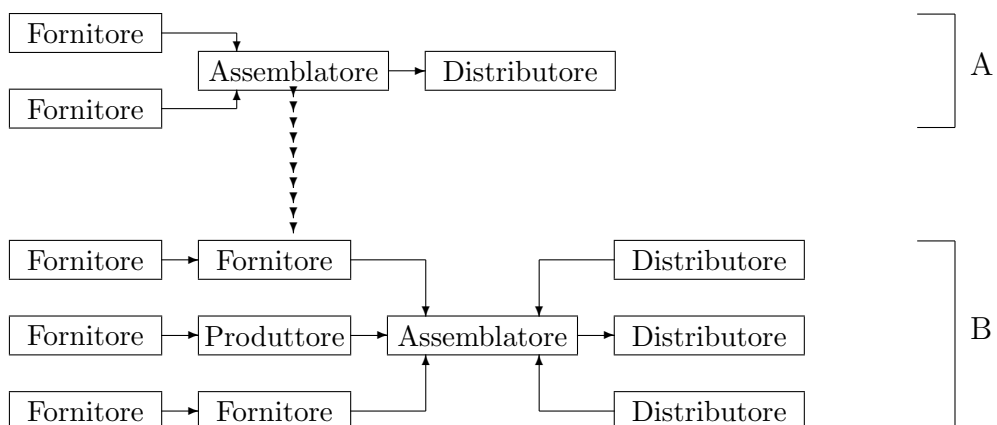


Figura 3.3: rappresentazione grafica della **supply chain network**

In figura 3.2 si può notare una semplice struttura di *supply chain* costituita da una sequenza, rappresentata nello schema *A*, che mostra soggetti nello svolgere funzioni di fornitori, altri di produttori e così via; ma nulla vieta che contemporaneamente si verifichi la situazione dello schema *B*, nel quale un assemblatore della precedente catena produttiva è ora diventato un fornitore.

Questo intende spiegare che nel costruire un modello aziendale occorre tenere in considerazione le diverse situazioni in cui si può trovare l'impresa a seconda della struttura di mercato in cui è inserita.

Customer Relationship Management

Un altro punto di vista per l'impresa è il lato delle vendite del proprio prodotto. Come si verifica dalla parte delle forniture anche dalla parte delle vendite si possono prospettare più scenari; essi non sono reciprocamente esclusivi, ma possono convivere senza problemi se la struttura dell'impresa lo consente.

Una sintesi delle situazioni in uscita può essere quella rappresentata in figura 3.4.

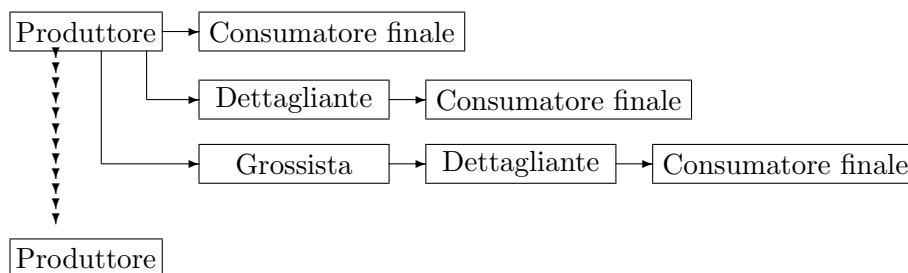


Figura 3.4: rappresentazione grafica del **Customer Relationship Managment**

La scelta di un percorso piuttosto che di un altro dipende principalmente dalle scelte strategiche di *distribuzione* del prodotto che l'azienda intende adottare. Infatti, solo uno studio specifico della clientela con cui l'impresa tratta permette di valutare se sia più opportuno commerciare direttamente con il cliente o tramite intermediari.

Ad esempio, nel caso in cui essa si trovi di fronte ad una molteplicità di singoli clienti finali si presume, ma non è detto, che deleghi le vendite ad un operatore specializzato il quale sarà in grado di considerare, consigliare, stimolare maggiormente il cliente. Mentre se il suo scopo è vendere grosse forniture di materiale ad altre aziende presumibilmente sarà essa stessa a trattare con il cliente, in previsione, magari, di un rapporto di lavoro continuativo.

Queste accortezze consentono di risparmiare notevolmente sui costi, sia *fissi*, in quanto evitano all'impresa investimenti in canali distributivi improduttivi, sia *variabili*, in quanto limitano sprechi di materiali e di tempo.

In conclusione, tali processi sono alla base della formazione di un'impresa in quanto ne influenzano direttamente la struttura organizzativa. Pertanto, è opportuno tenerne conto ed interpretarne correttamente il funzionamento per formulare una struttura generalizzata dell'azienda virtuale. Pertanto, date le nozioni di base, occorre analizzare quali sono le variabili in merito alla struttura organizzativa che maggiormente influenzano le decisioni aziendali.

3.2.3 L'Information Technology

Nell'ambito dell'architettura di un'impresa ed in considerazione di ciò che è stato finora detto è opportuno valutare come l'azienda decida di svolgere determinate funzioni o meno in una specifica catena del valore. La scelta consiste in sostanza nel *make or buy* ovvero tra produrre od acquistare risorse nel corso di un processo produttivo.

Qualora si scelga di *produrre* per buona parte o per l'intero processo si è optato per l'*integrazione verticale* dell'impresa, al contrario se si sceglie di concentrare le forze aziendali in un ambito specifico della produzione *comprando* ciò di cui si necessita si è optato per la *specializzazione*.

Entrambe le decisioni mostrano vantaggi e svantaggi. La scelta di una piuttosto che dell'altra si basa sull'analisi dell'ambiente in cui l'impresa intende operare.

Infatti, un'azienda integrata verticalmente è auto sufficiente ed indipendente, ma presenta una struttura complicata che spesso risulta difficile da gestire. Normalmente una struttura di questo tipo causa una forte burocratizzazione per l'inevitabile moltiplicarsi di reparti dirigenziali con una conseguente lentezza decisionale ed operativa. Pertanto tale architettura può portare buoni risultati o in un ambiente molto *statico* in cui raramente si presenti la necessità di ri-modellare l'azienda in funzione dei nuovi impulsi del mercato o, per forza di cose, in un ambiente in cui manchino i fornitori di risorse per l'attività di impresa.

Qualora l'azienda operi in una realtà molto *dinamica* è più opportuno optare per la *specializzazione*. In questo modo si concentrano le risorse dell'impresa verso un unico obiettivo con il vantaggio di snellire la struttura rendendola maggiormente adattabile all'ambiente circostante. Da notare come il termine specializzazione non vada confuso con *dimensione*; non è assolutamente detto che un'impresa specializzata sia una piccola impresa, la dimensione dipende dall'abilità dell'azienda di guadagnare quote di mercato indipendentemente da ciò che essa si occupi di produrre.

Lo sviluppo di imprese specializzate viene ad oggi favorito dal *progresso tecnologico* e dallo sviluppo di *standard* di produzione. La tendenza è di spezzare la catena del valore del prodotto in parti semplici per poi procedere in un momento successivo all'assemblaggio dei singoli pezzi. Questo perché: in primo luogo la produzione di oggetti complicati richiede costi fissi elevati ed in secondo luogo la necessità di soddisfare il cliente per eludere la concorrenza richiede che il prodotto sia un prodotto di qualità. Un miglioramento di entrambi questi fattori può essere, appunto, perseguito con la suddivisione del lavoro tra più aziende specializzate in un settore specifico.

Per un corretto funzionamento di un mercato composto da imprese specializzate non è però solo sufficiente che i singoli pezzi siano standardizzati e, pertanto, assemblabili, ma occorre che il commercio di prodotti tra agenti sia efficiente. Con efficienza si intende sia, da un punto di vista commerciale, che le imprese abbia-

no l'opportunità di incontrarsi sul mercato sia, da un punto di vista finanziario, che vi siano bassi *costi di transazione*. Elevati costi di transazione, infatti, non giustificerebbero la scelta di spostare la produzione all'esterno dell'azienda per ridurre i costi fissi di produzione.

Notevoli passi in avanti nell'efficienza dei mercati vengono favoriti dalla tecnologia stessa che, con l'utilizzo del commercio elettronico, può sia agevolare i contatti di compra-vendita sia ridurre i tempi di consegna dei prodotti. In particolare la possibilità di reperire informazioni più tempestivamente ed a costi contenuti, grazie ad un miglioramento delle infrastrutture della comunicazione, sta modificando molto la gestione dell'*information technology* (IT)¹.

Tabella 3.1: ipotesi di Information Technology

Industry Scenario	Old IT	New IT
Integrated and Few Firms	H1: being integrated is more successful than specialization	
Specialized and Many Firms		H1: being specialized is more successful than integration

Nel formulare un modello che intenda prendere in analisi le scelte in merito all'*information technology* occorre considerare, riducendo il problema ai minimi termini, sia la possibilità di rappresentare un'impresa integrata sia un'impresa specializzata. Uno schema possibile delle ipotesi base è quello rappresentato in tabella 3.1, si veda Langdon e Shaw (2000).

Considerando che il modello di impresa virtuale mira a conseguire una generalizzazione tale da renderlo applicabile a molteplici realtà è opportuno prendere

¹Per una trattazione più approfondita si veda il capitolo 4

in considerazione entrambe le ipotesi **H1** ed **H2** rappresentate in tabella 3.1 e formulare una struttura sufficientemente flessibile da renderle entrambe plausibili.

3.3 Struttura di alcuni modelli

3.3.1 Multi-Agent Information System (MAIS)

In Shaw e altri (1996) viene proposto un modello ad agenti relativo ad una simulazione sulle catene di fornitura. Questo progetto è rilevante per lo studio del modello di impresa virtuale poiché non solo tratta di un argomento aziendale, ma è anche stato sviluppato con il medesimo strumento, *Swarm*, il che può aiutare a comprenderne l'utilizzo.

Swarm può essere visto come un *multi agent information system* comprendente: agenti, attività, organizzazioni e infrastruttura informativa. Nel dettaglio, si veda Shaw e altri (1996):

- un *agente* è un oggetto capace di compiere azioni per raggiungere un obiettivo e di comunicare con altri agenti basandosi su una struttura organizzativa predefinita.
- le *attività* sono ciò per cui gli agenti devono lavorare; le attività possono sia essere scomposte ed affidate a più agenti sia raggruppate ed affidate ad un unico agente.
- le *organizzazioni* dentro il *MAIS* sono determinate dalle relazioni che intercorrono tra gli agenti; relazioni, quali flussi di controllo ed informativi.
- l'*infrastruttura informativa* gestisce i flussi informativi tra gli agenti e, pertanto, le interazioni della struttura organizzativa.

In particolare, una catena di fornitura può essere vista come una composizione di autonome o semi-autonome *business units* che in un modello possono essere rappresentate come uno *swarm* di agenti. Ognuna di queste unità è in grado di svolgere una determinata attività, che avrà uno specifico ruolo all'interno di un'organizzazione; ugualmente gli agenti possono svolgere il proprio compito

all'interno di uno *swarm*. Infine, è possibile riportare il flusso informativo della catena di fornitura come dialogo virtuale tra gli agenti del modello.

Questo, descritto più nel dettaglio in tabella 3.2, per dire come sia possibile formulare una connessione logica tra una realtà aziendale ed un modello con *Swarm*.

Tabella 3.2: confronto tra *supply chain* e modello con *Swarm*

Supply Chain	Swarm
<i>Business Units</i>	<i>Swarm</i> di agenti
Attività delle unità di <i>business</i>	Metodi e variabili degli agenti
Suddivisione di processi	Struttura gerarchica di agenti
Flusso informativo	Messaggi tra agenti
I/O di materiali ed informazioni	Eventi discreti regolati dallo schedule
\sum entità = risultato di gruppo	\sum agenti = <i>swarm</i>

La componente principale del modello è costituita dall'agente, il quale è elaborato su una struttura standard quale quella rappresentata in figura 3.5, tratta da Shaw e altri (1996 p. 16).

L'agente inizia il suo compito nel momento in cui riceve un *input* dal modello. Le azioni che è in grado di compiere sui dati in entrata sono descritte nell'*Action Engine*; in questa fase è definito il comportamento specifico dell'agente. I risultati ottenuti vengono:

- raccolti in un *database* in modo che possano essere letti e ricordati da tutto il modello
- considerati come nuova conoscenza dell'agente e raccolti nel *knowledge base*
- mandati in uscita dal modello

La conoscenza degli agenti potrà essere utilizzata al fine di innescare un processo di apprendimento, *Learning*, che detterà all'*Action Engine* nuovi comportamenti da tenere nelle differenti situazioni che si prospettano.

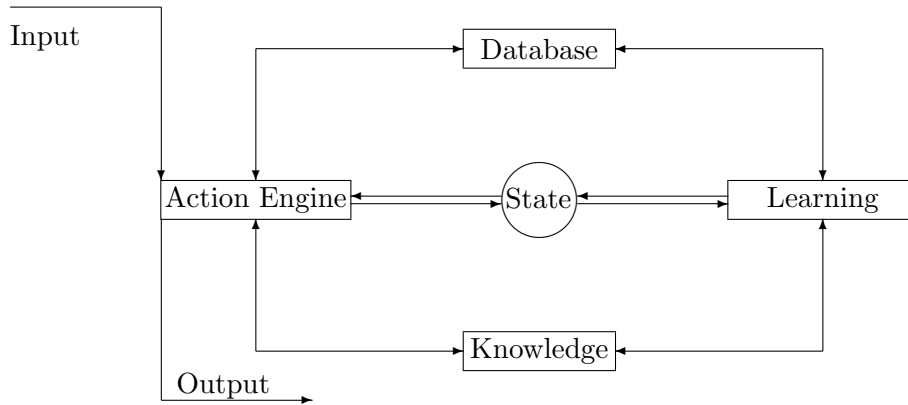


Figura 3.5: esempio di struttura di agente nel modello MAIS

All'interno del modello sono presenti molti agenti con questa struttura, ognuno con un ruolo specifico inerente al proprio compito nella catena di fornitura. Ad esempio si ha l'*Order Managment Agent* che si occupa di ricevere gli ordini e di trasmetterli al *Production Planning Agent* per la definizione del processo produttivo, in seguito i piani produttivi saranno trasmessi al *Manufacturing Agent* per essere eseguiti. In sostanza si ottiene una fedele riproduzione di una catena di fornitura in un modello basato su agenti.

In questo modo è possibile simulare:

- l'impatto dell'impresa sul mercato in termini di quantità prodotte, tipologie di prodotti presentati ed i tempi di attesa per la consegna da parte dei clienti.
- i meccanismi di controllo dell'organizzazione che determinano le interrelazioni tra le singole entità.
- il flusso informativo tra e nelle imprese; viene rivolta particolare atten-

zione per questo tema in quanto nodo centrale dell'*information technology* nell'economia dell'informazione (si veda il capitolo 4).

- le strategie di gestione in merito alla produzione, alla distribuzione ed alle scorte.
- le incertezze derivanti da un sistema complesso di stampo aziendale in merito alle variazioni della domanda di mercato, ai danni alla produzione o alla mancanza di forniture nei tempi previsti.

I principi di base del modello *MAIS* sono molto simili alle idee di partenza per la formulazione del modello di impresa virtuale, in particolare per ciò che concerne l'utilizzo di agenti e di *swarm* di agenti, il flusso degli ordini ed il flusso informativo all'interno delle organizzazioni. La differenza risiede nell'obiettivo più ampio dell'impresa virtuale di simulare non solo ciò che concerne la catena di fornitura, ma tutto il processo produttivo aziendale.

3.3.2 National Industrial Information Infrastructure Protocols (*NIIIP*)

L'interesse verso il modello *NIIIP* deriva dal fatto che esso si prefigge, vedere *NIIIP* 1998):

(...) 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 *NIIIP* technology, foster collaborative efforts and the sharing of engineering and manufacturing information.

L'impresa virtuale viene in questo contesto intesa come un'organizzazione costituita da compagnie che intendano collaborare per migliorare reciprocamente la loro posizione di mercato. Un'intento diverso da quello espresso per la *Virtual Enterprise* del modello elaborato in questa sede, che si prefigge di sviluppare un'unica impresa da interfacciare con il mercato. Un'analisi più approfondita può però essere utile per comprendere in analogia con il dialogo delle compagnie interagenti nel modello *NIIP* come far collaborare e cooperare le unità produttive esterne ed interne dell'impresa virtuale.

NIIP si presenta come un Consorzio di aziende che intende sviluppare una struttura software che consenta di migliorare la comunicazione tra le imprese stesse per ottenere un'interazione nel processo produttivo; per perseguire questo scopo devono essere raggiunti tre obiettivi:

- stabilire un protocollo standard di base che integri i vari processi di business, i dati e gli ambienti di calcolo.
- sviluppare *NIIP* sulle basi delle strutture stabilite.
- sviluppare il software per dare il via al dialogo tra le imprese.

Da questi punti emerge l'intento di creare una nuova infrastruttura di *Information Technology* volta a favorire la specializzazione sul prodotto data l'interazione tra più imprese in diverse fasi della catena del valore, si veda il paragrafo 3.2.3. Inoltre, si fissano i principi base per il commercio elettronico, basato non su uno standard di conoscenza comune, ma su un protocollo appositamente realizzato, si veda il capitolo 4.

Le compagnie dell'impresa virtuale *NIIP* devono essere in questo modo in grado di interagire e scambiarsi informazioni in tempo reale, quali fossero singole unità di un'unica impresa. L'integrazione dovrebbe così portare virtualmente ad avere una sola unità che si occupa dell'intero processo produttivo.

I vantaggi di questo modello sarebbero di produrre prodotti di elevata qualità, data la specializzazione delle singole unità, ma con le economie di scala di un'impresa interamente integrata; conseguentemente prodotti più competitivi e tempi di produzione più ristretti.

Per realizzare questa struttura si è strutturato un modello che si basa principalmente sullo sviluppo di:

- *NIIP interfaces protocols*: i protocolli sono *oggetti* con attributi, stati e funzioni. Ad esempio, alla base di tutti vi sono *Create VE*, *Register VE Resources* *Operate upon VE Resources*, con essi prendono forma l'impresa virtuale e le risorse di cui dispone.
- *NIIP Components*: con componente si intende in questo caso un "*luogo*" per raggruppare le interfacce che supportano un determinato protocollo. *NIIP* intende sviluppare un proprio ambiente di componenti come *Microsoft* ha fatto con *COM* e *Java* con *CORBA*. I principali sono:
 1. *NIIP Desktop*: è il luogo di esecuzione del lavoro dell'impresa virtuale. Qui si incontrano gli utilizzatori finali del progetto e gli agenti che rendono disponibili i servizi.
 2. *Agent*: gli agenti parlano tutti lo stesso linguaggio affinché possano comunicare tra di loro e negoziare.
 3. *DataManagement*: si occupa di gestire i dati di cui dispone l'impresa virtuale
 4. *Workflow*: gestisce i flussi di lavoro tra i componenti.
 5. *Mediator*: gestisce il dialogo tra i vari *database*.
- *VE Member Resources*: le risorse dell'impresa virtuale sono raccolte in uno schema globale, in modo che tutti gli utilizzatori possano avere una visione d'insieme.

Una rappresentazione globale del funzionamento del modello è rappresentata in figura 3.6, in essa si possono notare le interazioni dell'impresa virtuale con i protocolli e le risorse dei membri di VE (come oggetti astratti) e con le componenti del modello (come oggetti istanziati). Inoltre, appare anche il termine di *VE_Gateway* per indicare le strutture informatiche con cui interagisce l'impresa virtuale; in queste risiedono i *database* relativi a tutte le operazioni svolte da VE.

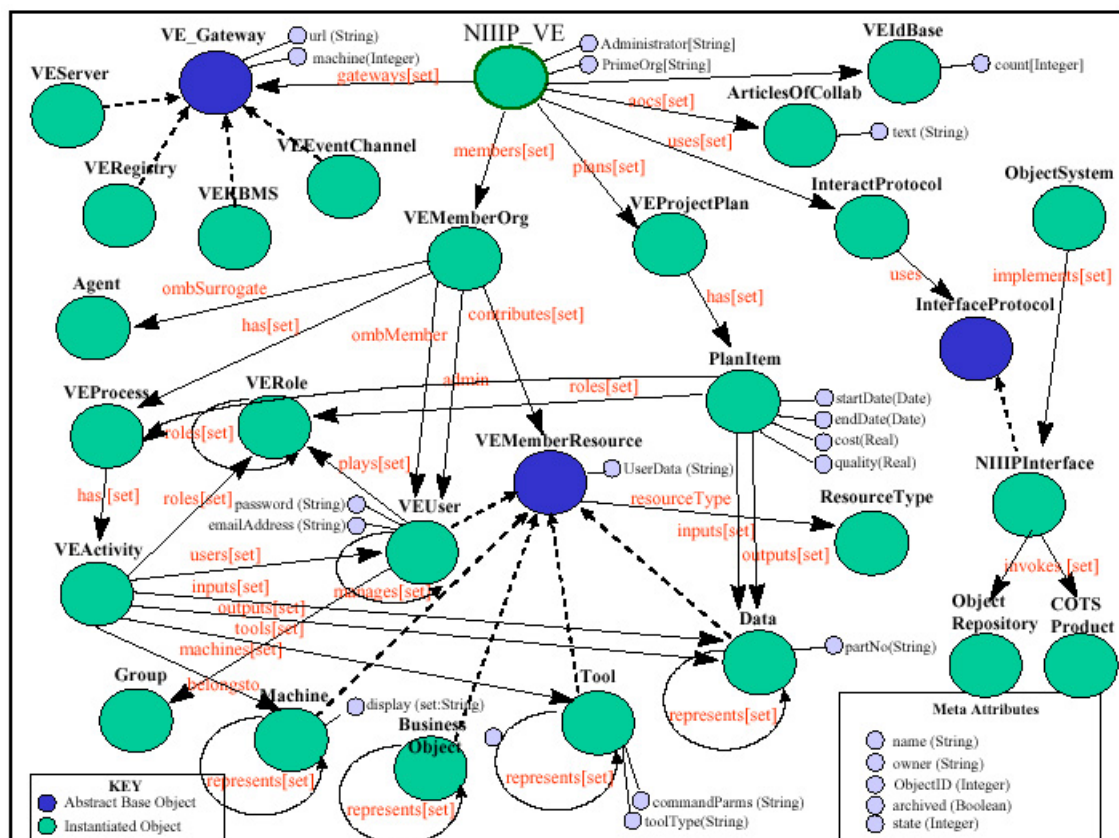


Figura 3.6: architettura del modello *NIIP*

L'obiettivo finale del modello è quello di essere impiegato sul mercato americano come standard di dialogo tra tutte le imprese, con gli stessi presupposti del *business-to-business*. La difficoltà nella realizzabilità del progetto risiede nell'enorme fase progettuale da realizzare e nei tempi che esse può richiedere. Infatti, data la dinamicità del mercato dell'informatica ed il continuo sviluppo di nuovi

standard sarebbe necessario un continuo aggiornamento del modello senza che si abbia il tempo di farne un uso effettivo.

3.3.3 Process Life Cycle Engineering

Scacchi e Mi (1997) identificano nell'*information technology* una strada da perseguire per rispondere alle pressioni competitive del mercato ed alla richiesta di continui miglioramenti dei processi produttivi. Attraverso una maggiore e migliore interazione tra le imprese, specie nei mercati elettronici, è possibile espandere le loro quote di mercato.

Il loro obiettivo consiste nel riprogettare i complessi processi organizzativi delle imprese utilizzando un ambiente di sviluppo *knowledge based*, ovvero data la conoscenza maturata sui dati ed i risultati attuali si mira ad utilizzarla per migliorare l'organizzazione dell'azienda. Per perseguire questo scopo si fa uso di un percorso denominato *Process Life Cycle Engineering* che fissa le seguenti, principali, tappe:

- *meta-modeling*: ridefinire i concetti e la logica relativi ad un processo per generare immagini dei medesimi sotto forma di classi, metodi, attributi e regole.
- *modeling*: formalizzare la descrizione del funzionamento di un processo.
- *analysis*: sviluppare le proprietà di un processo per valutarne la corrispondenza con la realtà.
- *simulation*: attivare, simbolicamente, il processo per determinare lo stato ed il flusso delle transizioni interne, in modo che possano essere analizzate, ripetute e modificate per un confronto o cambiate per prospettare nuovi scenari.

- *redesign*: riorganizzare e trasformare la struttura per ottimizzare i tempi, riducendo passaggi inutili.
- *visualization*: rendere accessibili i risultati raggiunti tramite la generazione di grafici di processi statici o dinamici.

Il percorso fin qui delineato mostra il tipico caso di analisi, creazione di un modello e simulazione di un fenomeno complesso.

Successivamente vengono descritte le fasi relative ad una miglior strutturazione del modello, ad esempio rendendolo interattivo con l'ambiente che lo circonda e registrandone i comportamenti che esso mostra nelle varie situazioni in cui si viene a trovare.

Interessanti sono gli ultimi passi descritti; il primo, definito *articulation*, fa riferimento alle proceure da seguire nel caso in cui si verifichi un rottura all'interno del processo. Questo ci riconduce all'argomento della complessità in quanto la fase di *articulation* prende in considerazione la possibilità del verificarsi di *unforeseen emergence*, si veda il paragrafo 1.2, ovvero di un fenomeno imprevisto. Nella realtà un fenomeno di rottura del processo è comune e diffuso e deve esserlo altrettanto all'interno del modello; proprio perché si tratta di un fattore non prevedibile o descivibile, complesso appunto, occorre computarlo nella simulazione.

Poi si fa riferimento all'*evolution* ovvero alla necessità che il modello migliori se stesso in funzione dei risultati raggiunti, in questo modo è possibile capitalizzare la conoscenza maturata nel corso della simulazione.

Infine, la fase di *process asset managment* che mira a raccogliere la conoscenza maturata e cumulata nel modello per progettare e riorganizzare attività reali.

Dal confronto della *Virtual Enterprise* con il *Process Life Cycle Engineering* emergono molte analogie sia concettuali che applicative. Innanzi tutto l'argomento principale di analisi è la complessità aziendale, poi per entrambi i problemi maggiori di studio sono incentrati sulle scelte da operare per la scomposizione

del problema e la sua schematizzazione in un modello. Superata questa fase si fa uso in entrambi i casi di un ambiente in cui inserire il modello e di agenti per interagire con esso. La differenza maggiore risiede forse nel punto focale di analisi del modello; mentre per il *Process Life Cycle Engineering* si procede ad un studio dettagliato del processo, l'impresa virtuale mira a simulare l'azienda nella sua globalità, comprensiva di processi e di altre variabili.

Bibliografia

- [1] Langdon C. e Shaw M. J. (2000), *Information Technology and The Vertical Organization of Industry*, Information Systems and Operation Managment Department, University of Southern California, Working Paper.
- [2] Lin F. R., Tan G. W. e Shaw M. J. (1996), *Multi Agent Enterprise Modeling*, Department of Business Administration, University of Illinois, Working Paper.
- [3] NIIIP (1998), *Introduction to NIIIP Concepts*, www.niiip.org, book0.pdf.
- [4] NIIIP (1998), *Guide to NIIIP Reference Architecture Model*, www.niiip.org, book1.pdf.
- [5] NIIIP (1998), *NIIIP Planner's Guide*, www.niiip.org, book2.pdf.
- [6] Scacchi W. e Mi P. (1997), *Process Life Cycle Engineering: A Knowledge-Based Approach and Environment*, Intelligent System in Accounting, Finance and Management, Vol. 6, pp 83-107.
- [7] Stalk G., Evans P. e Shulman L. (1992), *Competing on capabilities: the new rules of corporate strategy*, Harvard Business Review, pp 57-69.

Capitolo 4

Economia dell'informazione

Il fine ultimo per cui è stato iniziato il progetto di impresa virtuale e, pertanto, l'origine del progetto stesso consiste nel poter simulare il passaggio dal commercio tradizionale al commercio elettronico. L'obiettivo è di poter valutare, prima di agire, la convenienza o no di questo cambiamento.

L'idea è maturata in seguito all'osservazione del mutato valore dell'informazione ed alle ripercussioni che stanno maturando sulle imprese.

Questo capitolo intende esaminare perché sia importante disporre di un modello su cui simulare i cambiamenti causati dallo sviluppo dell'economia dell'informazione e come questi stiano influenzando il processo di elaborazione del modello stesso.

4.1 Il valore dell'informazione

Prima di analizzare le cause del mutato valore dell'*informazione* occorre specificare che essa viene intesa non più solo secondo il significato classico del termine, ma, facendo riferimento a Varian e Shapiro (1999, pp. 3-4), si parla di:

(...) informazione con un accezione molto ampia. Fondamentalmente tutto ciò che può essere digitalizzato, ovvero rappresentato come una sequenza di bit, è informazione. (...) Parte dell'informazione ha valore in quanto intrattenimento, e parte in quanto legata ad un'attività economica; tuttavia, a prescindere da quale sia l'origine dell'informazione, le persone sono disposte a pagare per entrarne in possesso.

Pertanto l'informazione è considerata come un bene vero e proprio che può essere prodotto, commerciato, venduto e riprodotto; un bene che assume tanta più importanza quanta riescono a trasmettergliene i canali di distribuzione tramite i quali può passare da chi lo produce a chi lo utilizza. Proprio la distribuzione di questo prodotto, che nello specifico si può ridefinire come *comunicazione*, è stata ed è tuttora motivo di molti cambiamenti strutturali nel mercato delle imprese. Le motivazioni della crescita d'importanza a livello economico del *bene informazione* sono attribuibili principalmente allo sviluppo *tecnologico* dei canali di comunicazione. L'analisi è incentrata soprattutto sull'*Information Technology* ovvero sull'infrastruttura che consente di distribuire informazioni. L'utilizzo di reti di comunicazione, quali l'*Internet* o le *Intranet*, hanno potenziato molto l'*IT* e, conseguentemente, l'informazione.

Con questo si intende che la crescita di importanza *non* è dovuta ad un miglioramento della *qualità* o della *quantità* del bene, ma ad una maggiore *reperibilità*. Infatti, dal punto di vista della qualità non è possibile trarre alcuna conclusione generale, in quanto spetta a chi lo ricerca selezionare quella di maggiore interesse e pertinenza ai propri bisogni, e del punto di vista della quantità non si è registrato alcun aumento come sottolineano Varian e Shapiro (1999, pp. 10-11):

la vera percentuale di testi utili in rete è molto minore (...) pari a circa quindicimila libri o, equivalentemente, alla metà dei libri di un buon emporio.

L'aspetto che invece ha aumentato molto il valore dell'informazione è la *di-*

sponibilità in tempi brevi e a costi contenuti. Infatti, l'informazione è un prodotto con elevati costi fissi, ma bassi costi marginali; la facilità con cui è possibile duplicarlo implica che una volta prodotto possa essere riprodotto velocemente quasi senza costi aggiuntivi. In questo modo anche coloro che producono informazione stanno cambiando la strategia di gestione del prodotto; poiché *proteggere* il loro bene con marchi o brevetti è divenuto pressoché impossibile è necessario cercare di valorizzare il loro lavoro sfruttando proprio la vasta percentuale di utenti che l'informazione è in grado di influenzare.

Le imprese devono necessariamente fare i conti con lo sviluppo di questa nuova economia sia in quanto ricettrici che in quanto produttrici di informazioni. Si consideri ad esempio il paragrafo 3.2.3, la possibilità di migliorare la struttura dell'information technology può cambiare i rapporti tra le imprese e tra imprese e clientela, con notevoli ripercussioni sulla struttura stessa dell'azienda. Cambiare, però, è costoso e non sempre i costi sostenuti per il cambiamento possono essere recuperati da vantaggi ottenuti sul mercato.

4.2 Il *lock-in* e gli *switching costs*

Cambiare la struttura dell'impresa in funzione dei nuovi principi dettati dall'economia dell'informazione comporta costi e rischi.

Le spese che un'impresa deve sostenere per cambiare la propria tecnologia ed acquisire nuovi clienti sono definite *switching costs* ovvero costi di transizione, che sono ulteriormente divisibili in costi di investimento, di interruzione, di pubblicità e di acquisizione dei clienti.

La prima valutazione che occorre fare concerne i *costi di investimento* in beni durevoli che l'impresa deve sostenere. Si tratta sostanzialmente, nel caso specifico dell'economia dell'informazione, dell'hardware che occorre acquistare per far funzionare uno specifico software ed i relativi accessori. Non solo questa è la prima valutazione che viene fatta, ma è anche la più immediata; infatti per investire occorre cercare uno o più fornitori e richiedere un preventivo per l'acquisto e l'installazione del prodotto. Fin da questo punto è possibile valutare la convenienza o meno del cambiamento di tecnologia confrontando i costi dell'investimento con i profitti stimati di un possibile incremento delle vendite. I costi da sostenere non finiscono però qui.

Infatti, variare la tecnologia di produzione implica variare il prodotto dell'impresa; cambiamento che comporta due difficoltà: in primo luogo valutare di poter fornire un prodotto compatibile con le strutture della clientela, in secondo luogo promuovere i pregi del nuovo bene con i relativi costi di marketing. Specie in questo tipo di mercato la percezione della clientela di una superiorità dell'impresa fornitrice è pari, se non superiore, ad una effettiva superiorità del prodotto.

Inoltre, occorre valutare di poter prestare il nuovo servizio continuativamente al precedente, senza interruzioni. Interrompere un servizio, anche se per un breve periodo, viene vissuto dal consumatore come un disservizio tale da compromettere la scelta di cambiare. Infatti, Varian e Shapiro (1999, p. 138) indicano nel computo dei costi di transizione anche i *costi di interruzione* del servizio presta-

to; da notare come spesso questa sia una condizione prossoché inevitabile. Ne è la prova la tesi sostenuta dai concorrenti potenziali nella telefonia americana di fronte alle autorità antitrust in merito alla difficoltà riscontrate nell'accedere ad un mercato in cui cambiare fornitore implica la perdita del servizio telefonico per qualche giorno. Ancora più complicato sarebbe il caso del commercio elettronico tra imprese in cui l'azienda cliente dovesse rallentare la propria produzione a fronte di un blocco di un servizio prestato da un fornitore. Risulta indubbiamente difficile computare questi costi in termini di fattibilità economica, poiché si tratta per lo più di saper misurare il prezzo della insoddisfazione dei clienti dovuta all'interruzione di un servizio. Ciò nonostante occorre tenerne conto come possibile variabile che gioca a sfavore dei profitti generati dal cambiamento.

Infine, ammesso che un'impresa sia disposta a sopportare tutti i costi strutturali per se stessa, i costi di marketing ed abbia risolto i problemi di interruzione del servizio deve domandarsi quanto le costa acquistare un nuovo cliente. Cambiare tecnologia all'interno dell'azienda e quindi del servizio fornito implica modificare anche la tecnologia del cliente ricevente il nuovo servizio. Le scelte possibili a questo punto sono due: o l'impresa sopporta i costi di installazione¹ presso il proprio cliente o lo incentiva a farsene carico con offerte promozionali. In entrambi i casi occorre conteggiare dei costi siano essi maggiori spese siano minori profitti.

Il complesso degli *switching cost* comporta il rischio di *lock-in* dell'impresa: il quale si verifica qualora l'analisi dei costi di transizione mostrasse un rilevante riscontro economico negativo tale da non essere sostenibile dalla finanza dell'azienda. Ovvero l'impresa si trova bloccata in una situazione di vecchia tecnologia dalla quale non riesce più ad uscire. Si pensi, ad esempio, ad un'azienda che per anni abbia investito nella marca di computer A ed un giorno decida, per convenienza o per i migliori servizi offerti, di spostarsi verso la marca di computer B. I

¹Caso delle imprese che promuovono la linea telefonica con tecnologia ADSL o delle compagnie che offrono parabola ed installazione della stessa per passare alla tecnologia satellitare (NdA).

costi di investimento saranno relativi all'hardware che occorre rinnovare, ma non solo. Tutto il software relativo alla marca di computer A diventerà inutilizzabile e pertanto occorrerà rinnovarlo ed *addestrare* il personale al suo utilizzo. Inoltre occorre interrogarsi se il nuovo sistema informativo è compatibile con quello della clientela, altrimenti tutto i nuovi prodotti risulterebbero inutilizzabili. In pratica potrebbe crearsi una situazione per la quale l'impresa si trovi impossibilitata ad apportare cambiamenti alla propria struttura.

Switching costs e *lock-in* sono, pertanto, direttamente proporzionali in quanto maggiori sono i costi che devono essere sostenuti per il cambiamento maggiore sarà la difficoltà a modificare la nuova realtà. Naturalmente il *lock-in* può essere estremamente negativo per chi vi rimane *ingabbiato*, ma estremamente positivo per chi riesce a vincolare continuativamente nel tempo i propri clienti. In entrambi i casi è opportuno riuscire a prospettarsi in anticipo le conseguenze, da un lato per preparare eventuali soluzioni alternative, dall'altro per riuscire a trarne il massimo profitto.

Tabella 4.1: tipologie di lock-in e switching cost a loro associati

Tipo di lock-in	Switching Cost
<i>Impegni contrattuali</i>	Oneri di liquidazione o di compensazione
<i>Acquisti di beni duravoli</i>	Sostituzione delle attrezzature
<i>Addestramento specifico</i>	Apprendimento di un nuovo sistema
<i>Informazione e Database</i>	Conversione dei dati ai nuovi formati
<i>Fornitori di beni specifici</i>	Finanziamento di nuovi fornitori
<i>Costi di Ricerca</i>	Costi di ricerca complessivi (del fornitore e dei clienti)
<i>Programmi di fidelizzazione</i>	Perdita dei benefici dei vecchi fornitori

Una sintesi dei principali casi di *lock-in* con i ripetitivi *switching cost* è descritta

in tabella 4.1, tratta da Varian e Shapiro (1999, p. 142).

Il problema del *lock-in* ha assunto importanza crescente nell'economia dell'informazione, in quanto il continuo cambiamento delle tecnologie in direzioni pressoché imprevedibili rende sempre più difficile valutare il ciclo vitale del prodotto di cui si fa uso. E' possibile così trovarsi legati all'utilizzo di un bene che avrà una vita economica molto breve, con il rischio di non poterlo sostituire a causa dei costi di transizione troppo elevati.

4.3 Imprese ed ottimizzazione dell'informazione

I vantaggi dell'economia dell'informazione, consolidati con il miglioramento delle tecnologie di comunicazione, sono stati accolti dalle imprese come una possibile via per migliorare il commercio. Commercio inteso sia nei confronti dei consumatori sia tra le imprese stesse. La comunicazione in entrambi i casi è stata notevolmente facilitata con l'impiego di reti quali l'*Internet* e le *Intranet* che permettono, grazie all'utilizzo di immagini e grafica, un avvicinamento tra acquirenti e venditori anche fisicamente molto distanti.

Le imprese intravedono la possibilità di utilizzare questi strumenti per migliorare i rapporti con i propri fornitori da un lato e con clienti dall'altro, tramite l'ottimizzazione dei costi di produzione e commerciali. Come illustrato nel paragrafo 4.2, i costi ed i rischi del cambiamento sono molti, ma i vantaggi che può offrire possono convincere a sostenerli. Le vie da percorrere sono due: la prima concerne la costruzione di un nuovo canale di distribuzione tra imprese e utilizzatori finali (*business to consumer*) e la seconda riguarda la formulazione di un nuovo protocollo di dialogo per le forniture delle imprese (*business to business*). I primi risultati del cambiamento sono già visibili sul mercato e possono essere analizzati.

Il commercio automatizzato tra imprese e consumatori finali ha dato buoni risultati solo per alcuni prodotti; i rapporti tra i due contraenti, infatti, possono essere facilmente compromessi, come sottolineano Borenstein e Saloner (2001), dalla necessità dei clienti di *toccare, annusare, provare o sentire* alcuni prodotti. Esistono beni che possono essere commerciati elettronicamente ed altri per i quali alle imprese conviene utilizzare canali di distribuzione alternativi.

Maggiore successo è registrabile per il commercio tra imprese; per esse il dialogo è più semplice e più ampio in quanto:

- possono disporre di esperti in grado di definire nei minimi dettagli il pro-

dotto che intendono acquistare o vendere, la competenza tecnica agevola il dialogo e la fiducia nei rapporti con la controparte.

- possono definire degli *standard* che, una volta operativi, possono essere ammortizzati con una durata continuativa del rapporto.

Grazie a questi fattori il *business to business*, inteso proprio come l'insieme delle transazioni tra imprese quali acquisto o la vendita di servizi, risorse, tecnologie e semilavorati, è in forte sviluppo; ne è la riprova la mole di prodotti commerciati tramite canali elettronici. Reiley e Spulber (2001) citano alcune stime relative all'ammontare del commercio elettronico tra imprese negli Stati Uniti:

By 2005, Jupiter expects the on-line component to represent \$6.3 trillion out of a total of \$15.1 trillion. A bit more modestly, Goldman Sachs (2000) projects B2B e-commerce transactions to reach \$4.5 trillion world wide by 2005.

Date le dimensioni previsionali del fenomeno e l'attenzione che vi rivolge l'ambiente imprenditoriale è verso questo aspetto che mira ad ottenere risultati il modello di impresa virtuale. L'interesse emerge dalle possibilità che un modello può offrire ad un'impresa di valutare i vantaggi che un tale cambiamento può fruttare a fronte dei costi che lo stesso comporta, per i quali si veda il paragrafo 4.2. In particolare, un modello di impresa che simuli il passaggio al commercio elettronico è in grado di mostrare nella pratica quali interventi strutturali ed organizzativi siano necessari per ottimizzare i vantaggi dell'impiego dell'economia dell'informazione.

La maggior parte di vantaggi che è possibile elencare in merito all'impiego dell'elettronica nel commercio concerne la riduzione dei tempi per il passaggio delle informazioni e la riduzione dei costi per le transazioni. Ad esempio è possibile ridurre i costi di ricerca di fornitori, i costi burocratici delle transazioni, i costi di comunicazione con la controparte.

Ciò che, però, incide in larga misura sull'economicità dell'utilizzo del *business to business* riguarda la riduzione dei costi per l'ottimizzazione della struttura interna ed esterna all'impresa. Si pensi ai processi di realizzazione di un prodotto ed alle varie fasi che esso deve superare per giungere a completamento, la produzione è spesso articolata in complicate catene di fornitura, specie nei settori in cui è presente un'elevato grado di specializzazione. Questo comporta inevitabili intoppi al processo produttivo che si riflettono sul costo del prodotto e sui tempi di attesa del cliente. Si rischia così di innescare circoli viziosi per i quali o si crea un'inadeguata attesa da parte del cliente, con il rischio di perderlo, o si accumulano preventivamente beni in magazzino, con un conseguente aumento del prezzo del prodotto. Con l'aumento della specializzazione dei mercati ed i perfezionamenti della tecnologia il problema si complica ulteriormente a causa della necessità di immagazzinare una maggiore varietà di beni con il rischio che questi diventino inutilizzabili. Per soddisfare la clientela le imprese sostengono, come affermano Borenstein e Saloner (2001, pp. 7):

(...) the tremendous cost of maintaining inventories in a wide variety of products across geographically dispersed outlets, coupled with an high search and transportation costs for consumers, which results at present in a great deal of compromising on product attributes.

L'obiettivo è di automatizzare tramite l'elettronica il trasferimento di informazioni tra le imprese che operano in successione sulla medesima catena di fornitura per ottimizzare i tempi di produzione e ridurre i beni immagazzinati in attesa di essere lavorati.

Ad esempio, può essere interessante valutare di elaborare un sistema previsionale della produzione relativo ad un determinato periodo dell'anno che automaticamente invia ai fornitori ordini di acquisto per le materie prime necessarie. Non appena l'ordine viene ricevuto, il fornitore commissionato può valutare i propri

criteri produttivi ed approvvigionarsi del materiale necessario. Se questo criterio fosse seguito da tutta la catena di fornitura si potrebbe: ridurre i tempi di attesa da parte dei cliente e produrre *just in time* senza ricorrere a dispendiose scorte di magazzino.

Conseguentemente a queste modifiche nei mercati si potrà notare un altro cambiamento inerente alle imprese che si occupano di intermediazione. La nuova struttura di impresa incentrata sul dialogo diretto le proprie controparti pare dare avvio ad un processo di disintermediazione dei mercati. Reiley e Spulber (2001) preferiscono però parlare di una modifica dei compiti degli intermediari piuttosto che di una loro scomparsa, infatti sostengono che:

(...) less expensive intermediation and lower transaction costs do not necessarily mean fewer intermediaries.

Questo perché l'utilizzo del commercio elettronico e le modifiche dell'*Information Technology* stanno favorendo una maggiore specializzazione delle imprese, quindi il trasferimento di molte attività finora interne all'esterno e, quindi, la necessità di aumentare i rapporti con altre realtà. In questo contesto si possono inserire i nuovi rapporti di intermediazione, che devono essere volti a migliorare e favorire il dialogo tra i singoli.

4.4 Modelli sull'economia dell'informazione

Due esempi molto interessanti di interpretazione e realizzazione in un modello degli aspetti concernenti il commercio elettronico sono *NIIP* e *DMarks II*.

Il primo riguarda la formulazione di un protocollo di comunicazione tra le imprese per migliorarne il commercio; da esso emergono difficoltà e priorità nella pratica del *business to business*. Anche se *NIIP* punta a realizzare un'impresa virtuale su tutto il mercato americano, le simulazioni che sviluppa consentono di apprendere concetti importanti anche per casi più ristretti (vedere il paragrafo 3.3.2).

Il secondo, *DMarks II*, mira a simulare un mercato sul quale gli agenti comunicano ed acquistano o vendono tramite l'utilizzo di strumenti automatizzati; a seguire si analizza il modello nel dettaglio.

4.4.1 A Decentralized Agent-Based Platform for Automated Trade and its Simulation (DMarks II)

A Decentralized Agent-Based Platform for Automated Trade and its Simulation (DMarks II), si veda Polani ed altri (2000), è stato strutturato con l'obiettivo di progettare un mercato in cui operino agenti autonomi in grado di comunicare ed operare transazioni. Il modello si basa su una tipica struttura *multi-agent* costituita da acquirenti e venditori e la sua simulazione mira a mostrare la formazione dei prezzi sul mercato e le differenti strategie di vendita.

La struttura del modello

Il modello è costituito da agenti, ognuno di essi dispone di una *conoscenza* che può o meno utilizzare all'interno del modello. In sostanza gli agenti possono o meno prendere parte alla simulazione rendendo così il modello estremamente dinamico.

La loro *conoscenza* si basa su informazioni per comunicare e commerciare; essa viene raccolta all'interno di *database* o direttamente dal *World Wide Web*.

Ogni singolo agente può prendere parte ad un *forum*, ovvero ad un luogo di discussione, che suddivide l'ambiente globale in un sotto spazio di agenti. Il *forum* costituisce il punto di incontro tra i partecipanti alla simulazione, accedendovi si rendono disponibili le proprie informazioni agli altri, ma al contempo se ne ottengono di nuove.

In questo modo ogni agente può operare la ricerca della controparte adeguata per instaurare una comunicazione ed eventualmente una transazione.

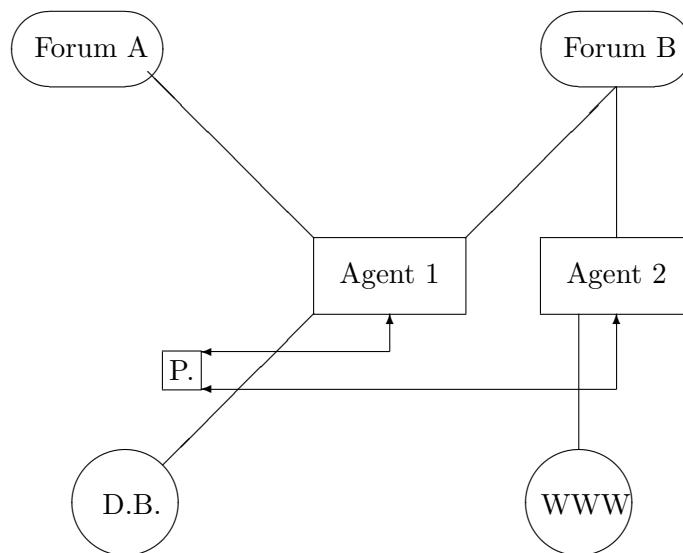


Figura 4.1: architettura del modello *DMark II*, P. rappresenta il punto di incontro per le transazioni, D.B. il database e WWW l'internet

Comunicare implica semplicemente l'invio di un'informazione ad un punto di scambio con l'indicazione del mittente, dell'indirizzo e dell'argomento di conversazione.

Eseguire una transazione è più complicato in quanto è necessario instaurare un contratto tra le parti. I contratti sono elaborati nella forma di moduli bi-laterali

che per essere effettivi necessitano di essere firmati da ambo le parti. Dopo che il secondo contraente ha firmato la propria parte il contratto viene eseguito. Nella simulazione non appaiono forme speciali volte a garantire la reciproca fiducia delle parti, si presumono solo rapporti basati sulla correttezza.

Per garantire la sicurezza delle transazioni si fa uso di una struttura *decentralizzata*, ovvero ogni rapporto avviene in luoghi sicuri esterni agli agenti contraenti.

Un esempio del modello può essere osservato in figura 4.1; in essa appaiono due agenti che si incontrano in un *forum* e, sulla base dei reciproci interessi sorti dalle informazioni di un database e dell'internet, contrattano in *P*.

Requisiti tecnici

Il linguaggio di programmazione utilizzato per la simulazione è *Java*, scelta dettata dall'interesse degli autori di far proprio l'utilizzo del metodo *RMI*. Il *Remote Method Invocation* consente di eseguire applicazioni decentralizzate, fondamentali per la struttura di *DMark II*. Infatti, è possibile costruire agenti come oggetti remoti, ognuno con proprie specifiche relative alle informazioni di cui dispone, su macchine tra loro separate. Le informazioni che ogni macchina invia al *server* della simulazione vengono gestite in contemporanea con quelle provenienti da altri agenti. La successione degli eventi o meglio la gestione simultanea di più eventi è garantita da *threads* che si occupano di coordinarle singolarmente.

La teoria della simulazione

Il modello consiste in una riproduzione di uno scenario di mercato in cui *produttori* e *consumatori* scambiano singoli beni con valuta.

I *produttori* generano unità di beni che vengono immagazzinate fino al momento in cui esse non debbano essere vendute. Produrre ed immagazzinare implica costi, quindi per ottimizzare il profitto è necessario regolare la produzione in ba-

se ad un adeguato rapporto prezzo/quantità. Sul mercato viene anche fissato un grado di concorrenza, che rende più o meno difficile collocare i prodotti alle condizioni desiderate.

I *consumatori* dispongono di una funzione di domanda la cui valorizzazione viene determinata in base alla quantità di moneta di cui dispongono per un certo periodo. L'acquisto avviene qualora ci sia un'intersezione tra la curva relativa a quanto si è disposti a spendere per una certa quantità, il *bid level*:

$$b(q_0 ; q)$$

e la curva di soddisfazione:

$$r(q_0 ; q ; p)$$

per rimanere su questa curva ad ogni passo della simulazione viene assegnata al consumatore una quota di moneta che gli consenta di *soddisfare* i propri bisogni.

Quando avviene una transazione sui contratti di acquisto e vendita viene registrata la quantità scambiata ed il prezzo pattuito. Dal loro prodotto emerge il valore da aggiungere al bilancio del produttore e da sottrarre all'ammontare a disposizione del consumatore. Il prodotto acquistato viene consumato immediatamente e ciò implica una modifica del livello di soddisfazione dei consumatori, sulla base della seguente relazione:

$$r(q_0 ; q ; p) = \int_{q_0}^{q_0+q} b(s)ds - p \cdot q$$

La scelta dei consumatori rispetto a quale venditore rivolgersi per acquistare si basa su un confronto tra il prezzo richiesto dal venditore e la quantità che si desidera ottenere, questa scelta diventa tanto più complicata quanto aumenta il livello di concorrenza sul mercato.

I risultati della simulazione

La simulazione di differenti impostazioni del modello ha consentito di illustrare i perfezionamenti che possono essere introdotti nelle strategie di produzione dei

venditori. In particolare, sono stati simulati i seguenti casi:

- *un mercato con offerta costante*: ovvero un mercato in cui i venditori regolano i propri profitti solo attraverso variazioni di prezzo. In questo caso si è notato che il prezzo diventa estremamente legato al grado di concorrenza che si imposta nel mercato, se viene aumentata la concorrenza i venditori devono abbassare sensibilmente il prezzo per riuscire a vendere la quantità che gli consente di ottenere un profitto. Questo esperimento mostra che produrre una certa quantità ed immagazzinarla per poi cercare di venderla al prezzo di mercato non porta a buoni risultati per le imprese.
- *un mercato con offerta variabile in funzione della domanda*:
 1. *strategia di apprezzamento fisso*: i venditori continuano ad aumentare il prezzo dei prodotti fino a quando il loro profitto continua ad aumentare; ovvero seguono una strategia di formulazione di un prezzo monopolistico. In uno scenario concorrenziale non si riesce mai a raggiungere il livello desiderato, anzi può accadere che si resti a livello dei costi di produzione.
 2. *strategia di informazione*: le imprese possono ottenere informazioni dal mercato in base alle quantità da produrre ed alle attese di profitto. Su queste basi possono decidere di produrre più o meno a seconda del prezzo che pensano di poter imporre sul mercato.

Bibliografia

- [1] Borenstein S. e Saloner G. (2001), *Economics and Electronic Commerce*, Journal of Economic Perspectives, Volume 15, Number 1, pp. 3-12.
- [2] Goldman Sachs (2000), *Technology: Internet-Commerce, United State*, Global Equity Reseach.
- [3] Jupiter Communications (2000), *U.S. Business to Business Internet Trade Projections*, www.nmm.com/reports/bbc/b2b_projections_reg.asp.
- [4] Polani D., Kutschinski E. e Uthmann T. (2000), *A Decentralized Agent-Based Platform for Automated Trade and its Simulation*, working paper.
- [5] Reiley D. L. e Spulber D. F. (2001), *Business-to-Business Electronic Commerce*, Journal of Economic Perspectives, Volume 15, Number 1, pp. 55-68.
- [6] Shapiro C. e Varian H. R. (1999), *Information Rules*, Etas, Milano.

Capitolo 5

Il modello di impresa virtuale

5.1 L'idea di partenza e la sua realizzazione in Swarm

Lo sviluppo della simulazione di impresa virtuale prevede come primo passo l'ideazione di un modello astratto in grado di costituire una solida base sulla quale poter elaborare la complessa struttura di un'impresa. In questa prima fase saranno introdotte forti semplificazioni rispetto alla realtà immaginabile di impresa che opera come sistema complesso in una realtà altrettanto complessa; semplificazioni che sono opportune sia, in termini pratici, per facilitare la stesura del codice sia, in termini teorici, per facilitare la comprensione e l'elaborazione delle singole parti del progetto.

La difficoltà di questa prima fase consiste nell'ideare una struttura fortemente flessibile, in grado di poter essere modificata continuamente in tutte le sue parti in funzione delle crescenti necessità di sviluppo del modello. E' pertanto opportuno tenere sempre ben presente l'obiettivo che si intende raggiungere ed in relazione ad esso l'ampio spettro di alternative che si propongono per ogni scelta che si intende compiere. Solo in questo modo si può procedere senza precludere nessuna strada per futuri miglioramenti del modello.

5.1.1 Lo schema iniziale

Gli ordini

La simulazione parte con l'emissione di un ordine di acquisto da parte dei clienti dell'impresa virtuale.

La scelta di partire nella simulazione dal lato della *domanda* è stata fatta in considerazione del fatto che, per adeguare i tempi e le procedure di produzione e di immagazzinamento dei prodotti, occorre avere ben presente quale siano le richieste degli stessi e su di esse modellare tutta la struttura aziendale. Questo al fine di ridurre i tempi morti o, al contrario, uno spreco di beni; in breve per poter formulare un'offerta adeguata alla domanda.

In termini pratici, in questa prima fase del progetto, all'impresa arrivano ordini di produzione composti da una sequenza di *numeri*, per ora generati casualmente dal computer. Si ottiene così una serie di vettori di numeri casuali che idealmente simboleggiano un prodotto da realizzare, ma praticamente costituiscono un insieme di informazioni delle operazioni da compiere.

La sequenza dei numeri indica all'impresa la metodologia da perseguire per ottenere ciò che le è stato richiesto; essa è da intendersi come una specie di *ricetta* che informa il produttore sul *percorso* da seguire per giungere al completamento del bene e progressivamente sullo *stato di completamento* raggiunto. Tali indicazioni possono riassumersi con due vettori: il primo relativo all'ordine ed il secondo, parallelo al primo, relativo al livello di produzione. Sarà così sufficiente modificare questi due vettori per migliorare e modificare l'informazione contenuta nella ricetta dell'ordine. Procedendo in questo modo si ottengono due risultati molto importanti: il coordinamento della successione degli eventi ed un continuo monitoraggio sull'effettiva realizzazione degli eventi stessi.

Del tutto casualmente avviene anche la scelta della lunghezza del vettore di numeri casuali poiché si assume che vi siano prodotti più o meno laboriosi da realizzare. In questo modo, se si ha che per ogni ciclo di produzione viene

accorciato il vettore di un'unità, un vettore di tre numeri potrà considerarsi finito in tre cicli, mentre un vettore di nove dovrà rimare in produzione per altri sei cicli.

Il Front End e le unità produttive

Gli ordini che giungono all'impresa vengono presi in consegna da unità interna che svolge il compito di *Front End* verso il mercato e che si occupa esclusivamente di smistare tali ordini alle altre unità dell'impresa. Il Front End può essere paragonato all'ufficio vendite di un'impresa reale, mentre le altre unità all'apparato produttivo. Per ora tali unità si contraddistinguono solo in base ad un numero e non in base alla funzione svolta. Questa semplificazione permette di mantenere il modello astratto e di agevolare la gestione degli ordini da parte delle unità. Infatti si può assumere che ogni unità sia deputata a produrre lo stesso numero che la contraddistingue. Uno schema della disposizione interna all'impresa è rappresentato in figura 5.1.

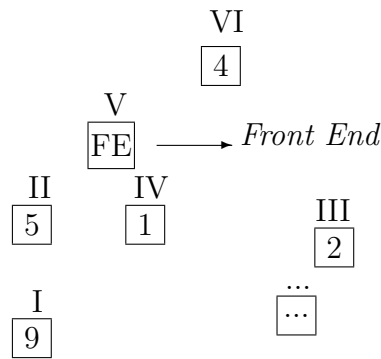


Figura 5.1: rappresentazione delle *unità produttive* espresse con i numeri romani e del *componente produttivo* con il numero arabo

L'attività dell'unità produttiva consiste nell'apprendere e modificare le informazioni contenute nei vettori dell'ordine. In particolare devono inserire l'informazione di completamento del proprio compito e individuare a quale unità spetta il processo produttivo seguente.

Le unità produttive a cui si fa riferimento sono tutte quelle necessarie per giungere a completamento del prodotto, indipendentemente dal fatto che esse siano interne o esterne all'impresa. Il caso di utilizzo di una catena di fornitura e una di distribuzione in realtà non si distacca molto da questo schema, se non concettualmente, infatti sarà sufficiente distinguere alcune unità per alcune caratteristiche, ma il dialogo con il modello rimarrà invariato. Quello a cui occorrerà porre attenzione sono i tempi ed i costi che distinguono l'utilizzo di una *supply chain* rispetto ad un'integrazione verticale dell'azienda. Per procedere in queste considerazioni occorre però prima analizzare la contabilità dell'impresa e la sua politica dei magazzini.

Le liste di Swarm come strumento di gestione

Nello sviluppare il processo produttivo appare subito il problema di gestire eventuali code di ordini di produzione. Questo problema in Swarm può essere superato tramite l'utilizzo delle *liste* che agiscono come raccoglitori di oggetti dai quali è possibile estrarre a piacere quello a cui si intende far riferimento.

Poiché è prevedibile che più ordini giungano contemporaneamente o in periodi molto ravvicinati alla stessa unità produttiva è necessario dotare ogni unità di una *lista di attesa* nella quale vengono inseriti gli ordini in arrivo e dalla quale vengono prelevati gli ordini da produrre. In questo modo sarà possibile affrontare ogni ciclo produttivo singolarmente senza sacrificare alcuna informazione.

La scelta del metodo da applicare per estrarre un oggetto da produrre piuttosto che un altro da una lista riporta alle strategie di gestione dei prodotti LIFO e FIFO. La scelta di uno o dell'altro per ora è ancora del tutto ininfluyente, ma è indicativa della flessibilità delle liste per la gestione delle code all'interno dell'azienda.

Le liste delle unità forniscono importanti informazioni anche sull'andamento dell'impresa. In particolare la lunghezza delle medesime indica una sovra o sotto

stima della capacità produttiva. Un problema che può causare un intasamento del processo produttivo generando colli di bottiglia che compromettono tutti i tempi di produzione.

Le liste possono così indicare se sia opportuno potenziare l'impresa, trasferire risorse al suo interno o decidere di affidarsi ad un produttore esterno. Tutte scelte di gestione molto importanti che sicuramente meritano un'analisi più approfondita, specie per quanto concerne i costi, ma che fin da questo passo possono emergere e, pertanto, devono essere sottolineate.

Il magazzino dei semilavorati

L'analisi di partenza dell'impresa indica anche la necessità di introdurre uno o più magazzini in grado di raccogliere la produzione in eccesso e di conservarla per un eventuale utilizzo futuro. Questo passo è indubbiamente necessario per generalizzare il modello in funzione delle molteplici esigenze aziendali.

Diverse politiche aziendali si basano su di un differente utilizzo del magazzino: le imprese con produzione stagionale, ad esempio, devono forzatamente, per un periodo dell'anno, produrre per il magazzino, mentre le imprese che si rivolgono al commercio elettronico cercano, attraverso una migliore gestione dei tempi, di ridurre al minimo i depositi in azienda.

Strategie diverse per necessità differenti, la cui profittabilità è misurabile solo in termini di costi sostenuti per la gestione di un magazzino in relazione alla soddisfazione del cliente. La situazione che si prospetta, vedi figura 5.2, si basa sul migliore compromesso che l'impresa riesce ad ottenere tra il tempo di attesa del cliente e la propria contabilità interna.

Nella prima fase non esiste un vero e proprio magazzino, ma la necessità del suo inserimento si desume dal comportamento delle liste di attesa delle unità produttive. Infatti, anche se del tutto casualmente, si può notare come spesso si formino code in alcune liste, mentre altre risultano completamente vuote. Sinto-



Figura 5.2: curve di costo e di insoddisfazione

mo di una errata gestione delle risorse che spesso risultano sprecate. Il magazzino è una possibile soluzione a questo tipo di problema che va sicuramente tenuta in considerazione.

I flussi aziendali

Un altro problema da prendere in esame sono le modalità con le quali comunicano le singole unità dell'azienda.

Nel sistema aziendale circolano principalmente decisioni, beni ed informazioni. I flussi che identificano questi passaggi devono essere predeterminati secondo una ben precisa logica aziendale per garantire il fluido operare dell'impresa.

Nel modello questa situazione può essere elaborata tramite l'utilizzo di tre matrici tra loro correlate, vedi figura 5.3.

Per ogni matrice si devono indicare le unità presenti in azienda sia sulle righe che sulle colonne. All'interno della matrice si indica la relazione che lega tra di loro tali unità, ad esempio lo 0 potrebbe indicare una mancanza di legame mentre l'1 il passaggio di un flusso. Questo ragionamento deve essere ripetuto per i flussi gerarchici, di beni e di informazioni.

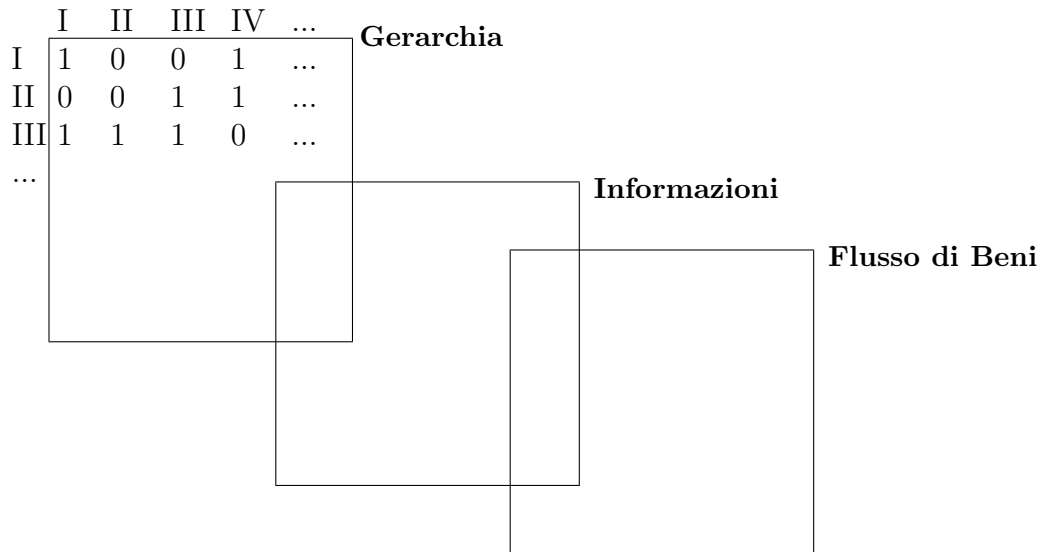


Figura 5.3: rappresentazione delle tre matrici di flusso

La prima fase del programma prevede che il passaggio di questi flussi sia racchiuso interamente negli spostamenti dell'ordine di produzione. Esso possiede le coordinate di spostamento nei vettori di produzione e di completamento che indicano precisamente per ogni ciclo come l'ordine deve spostarsi. In questo modo, l'ordine porta con sé il flusso di beni ed in qualche misura il flusso informativo, mancano invece le relazioni gerarchiche tra le unità.

5.1.2 Lo sviluppo del codice in Swarm

Il programma Java di partenza nella versione 0.3.3, la prima funzionante in modo integrato, si compone dei files:

- StartVEFrame.java
- VEFrameObserverSwarm.java
- VEFrameModelSarm.java
- Order.java

- OrderGenerator.java
- Unit.java

StartVEFrame.java

Questa classe viene costruita per lanciare la simulazione; le istruzioni che devono essere prese in considerazione sono relative alla costruzione di *VEFrameObserverSwarm* con i corrispettivi metodi che aprono l'interfaccia di dialogo dell'utente con il modello.

```
VEFrameObserverSwarm vE = new VEFrameObserverSwarm(Globals.env.globalZone);
vE.buildObject();
vE.buildActions();
vE.activateIn(null);
vE.go();
vE.drop();
```

In questa fase si dice all'oggetto *vEFrameObserverSwarm* di costruire gli oggetti (*buildObjects*), le azioni (*buildActions*) e di attivarsi (*activateIn*). Infine vengono richiamati due metodi, *go* e *drop*, che non sono propri della classe *VEFrameObserverSwarm*, ma che vengono ereditati da *swarm.simtoolsgui.GUISwarmImpl* e consentono di far partire e di fermare la simulazione.

VEFrameObserverSwarm.java

VEFrameObserverSwarm, in questa fase, compie le seguenti operazioni:

1. creazione degli oggetti nel metodo *buildObjects()*, in particolare:

- crea l'oggetto *vEFrameModelSwarm* con il comando

```
VEFrameModelSwarm vE = new VEFrameModelSwarm(getZone());
```

nel rispetto dello schema di procedimento suggerito da Swarm

- crea il grafico che rappresenta le liste di attesa delle unità produttive del modello, con il comando

```
EZGraphImpl waitingListGraph = new EZGraphImpl(getZone(),
"waiting_list", "time", "waiting list", "waiting_list");
```

indicando, nel costruttore, la zona di memoria da assegnare al grafico, il titolo della finestra, l'etichetta dell'ordinata e dell'ascissa ed, infine, il titolo del grafico

- crea una sonda ¹ con un'unica variabile, *displayFrequency*, volta a regolare l'aggiornamento del grafico sui risultati del programma

2. creazione delle azioni nel metodo *buildActions()* ed in particolare:

- permette a *vEFrameModelSwarm* di costruire il suo schedule:

```
vEFrameModelSwarm.buildActions();
```

- costruisce un oggetto di tipo *ActionGroup* per regolare lo schedule dei grafici:

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

- regola le azioni del grafico *waitingListGraph* sulle basi delle istruzioni proprie dell'*ActionGroup*:

```
displayActions.createActionTo$message
(waitingListGraph,
new Selector(waitingListGraph.getClass(), "step", true));
```

- costruisce lo schedule e lo imposta sulla base della "frequenza" di scatti, decisi tramite sonda dall'utente con la variabile *displayFrequency*, e del momento in cui vengono create le azioni, in questo caso al momento zero:

¹Occorre osservare che la sonda non fa parte del *buildActions()*, ma di una classe locale, si veda il paragrafo 2.3.

```
displaySchedule = new ScheduleImpl(getZone(), displayFrequen-
    cy);
displaySchedule.at$createAction(0, displayActions);
```

VEFrameModelSwarm.java

Secondo la procedura descritta nel paragrafo 2.3, *VEFrameModelSwarm* si compone di una prima fase di creazione degli oggetti e di una seconda di regolamentazione della tempistica dei loro compiti. Le operazioni da compiere saranno le seguenti:

1. costruzione degli oggetti:

- creazione dell'oggetto *orderGenerator* con il comando

```
OrderGenerator orderGenerator = new OrderGenerator(
    Globals.env.globalZone, totalUnitNumber, maxStepNumber, uni-
    tList);
```

orderGenerator è l'oggetto che genera i vettori di numeri casuali, il *model* si occupa di passargli l'indicazione sul numero di unità produttive da generare (*totalUnitNumber*), la lunghezza dei vettori (*totalUnitNumber*) e le liste delle unità (*unitList*)

- costruzione di una lista di oggetti (*Unit*) con la seguente istruzioni:

```
ListImpl unitList = new ListImpl(Globals.env.globalZone);
```

- richiamo dell'oggetto unità produttiva che lo aggiunge alle liste precedentemente costruite, le istruzioni sono:

```
Unit aUnit;
unitList.addLast(aUnit);
```

a questo punto avviene la scelta di costruire liste secondo un criterio F.I.F.O., come in questo caso in cui scegliamo di aggiungere oggetti alla

fine della lista(*addLast()*), o L.I.F.O., nel qual caso avremmo dovuto utilizzare il comando *addFirst()*

- creazione di una sonda con due variabili, *totalUnitNumber* e *maxStepNumber*, che passa i valori direttamente a *orderGenerator*

2. costruzione delle azioni. *VEFrameModelSwarm*:

- costruisce un oggetto di tipo *ActionGroup* per regolare lo schedule del modello:

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

- costruisce lo schedule e lo imposta con zero come momento di esecuzione dell'evento e con uno come intervallo di attesa prima di eseguire il successivo:

```
displaySchedule = new ScheduleImpl(getZone(), 1);
```

```
displaySchedule.at$createAction(0, modelActions);
```

- richiama la classe *Unit* e, nello specifico, il metodo *unitStep1()*, che esegue la prima sequenza di operazioni impostate dal programma:

```
sel = new Selector(Class.forName("Unit"),
```

```
"unitStep1", false);
```

```
modelActions.createActionsForEach$message(unitListStatic, sel);
```

- successivamente ordina all'oggetto *orderGenerator* di eseguire il metodo che genera numeri casuali:

```
sel = new Selector(orderGenerator.getClass(),
```

```
"createRandomOrderWithNSteps", false);
```

```
modelActions.createActionsTo$message(orderGenerator, sel);
```

- infine, come ultima azione, richiama ancora la classe *Unit* e con essa il metodo *unitStep2()*:

```
sel = new Selector(Class.forName("Unit"),
```

```

    "unitStep2", false);
    modelActions.createActionsForEach$message(unitListStatic, sel);

```

OrderGenerator.java

L' *OrderGenerator*, come dice il nome stesso, si occupa essenzialmente di produrre ordini; in una metafora della realtà rappresenta il *Front End* sul mercato ovvero l'ufficio a cui pervengono gli ordini dei clienti. Gli ordini si compongono di numeri casuali che vengono generati dal metodo *createRandomOrderWithNSteps()*.

In questo metodo si decide in un primo momento quanto deve essere lungo il vettore di numeri costituente l'ordine²:

```

    randomStepNumber=Globals.env.uniformIntRand.getIntegerWithMin$withMax
        (1, maxStepNumber);

```

in un secondo tempo, noto il valore casuale *randomStepNumber*, si costruisce, altrettanto casualmente, la ricetta dell'ordine:

```

    for (i=1;i<=randomStepNumber;i++)
    {
        orderRecipe[i-1]= Globals.env.uniformIntRand.getIntegerWithMin$withMax
            (1, unitNumber);
    }

```

Infine, l'ordine viene inviato alla lista di attesa della prima unità produttiva indicata nella sua ricetta:

```

    aUnit.setWaitingList(anOrder);

```

²Si è stabilito di generare ordini di diversa lunghezza in quanto possono pervenire domande di prodotti più o meno complicati da produrre.

Order.java

Order è l'oggetto metaforico rappresentante l'ordine, ad esso il modello chiede quasi esclusivamente di indicare, una volta prodotta una componente dell'ordine stesso, quale sia la successiva per inviarla alla corrispondente unità produttiva. Questo procedimento viene eseguito con il metodo *getNextStep()*, così strutturato:

```
public int getNextStep () {
    int i, stepN;
    stepN=0;
    for (i=0;i< stepNumber && stepN==0;i++)
        if (orderState[i]==0)stepN=orderRecipe[i];
    return stepN;
}
```

Per poter ricordare a quale livello della produzione dell'ordine si è giunti viene anche aggiornato il vettore relativo allo stato di completamento (*orderState[]*), sostituendo degli uno agli zero iniziale; con il metodo *setDoneStep()*:

```
public void setDoneStep () {
    int i, done;
    done=0;
    for (i=0;i<= stepNumber && done==0;i++)
        if (orderState[i]==0)
            {orderState[i]=1;
            done=1;
            }
}
```

Unit.java

La classe *Unit* rappresenta l'apparato produttivo dell'impresa virtuale; si compone essenzialmente di due metodi *unitStep1()* e *unitStep2()* che simboleggiano le

due fasi che in una giornata lavorativa³ devono essere svolte.

In *unitStep1()* si ricevono gli ordini e si passano alla produzione; nel dettaglio l'unità guarda nella propria *waitingList* e, se è presente un ordine,:

```
if(waitingList.getCount() > 0)
```

lo trasferisce in una lista di produzione giornaliera:

```
dailyProductionList.addLast(firstOrder);4
```

In *unitStep2()* si termina la produzione e si passa l'ordine all'unità produttiva successiva; in sostanza si rimuove l'ordine dalla lista di produzione giornaliera:

```
firstOrder=(Order)dailyProductionList.removeFirst();
```

e lo si interroga sul passo successivo da compiere. Se il passo seguente è nullo:

```
if (firstOrder.getNextStep()== 0)
```

allora l'ordine esce dalla produzione:

```
firstOrder.drop();
```

altrimenti viene passato alla fase produttiva successiva che corrisponde al numero della ricetta da eseguire:

```
if (firstOrder.getNextStep() == aUnit.getProductionPhase())  
{  
operatingUnitNotFound=false;  
aUnit.setWaitingList(firstOrder);  
}
```

³Per ora le unità produttive impiegano tutto il medesimo tempo, un tick dello schedule, per svolgere le proprie funzioni. In seguito i tempi produttivi saranno differenziati in funzione delle caratteristiche componente da produrre

⁴Continuando nella scelta di produrre con il metodo FIFO si utilizza *addLast* nell'inserimento nella lista

5.1.3 Perfezionamenti con la versione 0.3.9

La variabile *lispAppArchiver* e il *file.scm*

Per il passaggio al modello di valori o informazioni si sono finora mostrate due vie possibili: la prima tramite codice del programma e la seconda tramite le sonde. Una terza strada percorribile con l'impiego di Swarm consiste nello scrivere i dati all'interno di un file di tipo *scheme*. I *file.scm* hanno la particolarità di essere scritti con linguaggio *lisp*, ma per il loro utilizzo in questo ambito è sufficiente saper gestire una struttura semplice come quella mostrata in figura 5.4 (tratta da *jveframe.scm*).

In figura 5.4 si può notare che si richiede di creare le istanze di *vEFrameObserverSwarm* e di *vEFrameModelSwarm* per poi elencare le rispettive variabili con i valori collegati. Da notare che esse sono le medesime rappresentate nelle sonde e che quindi non esistono contrasti nell'impiego simultaneo dei due metodi. Una eventuale variazione dei valori nelle sonde al momento dell'esecuzione non sarebbe compromessa dai valori espressi nel *file.scm*.

Per integrare queste informazioni con il codice Java si utilizza un diverso costruttore delle classi che intendono richiamare il *file.scm*, in particolare si utilizza la variabile *lispAppArchiver* che appartiene all'*Environment* di *Swarm*. Nel caso dell'impresa virtuale i costruttori di *Observer* e *Model* devono essere così strutturati:

```
vEFrameObserverSwarm = (VEFrameObserverSwarm)
Globals.env.lispAppArchiver.getWithZone$key( Globals.env.globalZone, "vE-
FrameObserverSwarm");
vEFrameModelSwarm = (VEFrameModelSwarm)
Globals.env.lispAppArchiver.getWithZone$key( getZone(), "vEFrameModel-
Swarm");
```

```

(list
  (
    cons 'vEFrameObserverSwarm
      (
        make-instance 'VEFrameObserverSwarm
          #:displayFrequency 1
          #:verboseChoice #f
        )
      )

    (
      cons 'vEFrameModelSwarm
        (
          make-instance 'VEFrameModelSwarm
            #:totalUnitNumber 3
            #:maxStepNumber 3
            #:useWarehouses #t
            #:maxInWarehouses 10
          )
        )
      )
  )

```

Figura 5.4: *jveframe.scm*

La classe *SwarmUtils*

Questa classe è stata sviluppata per una questione di maggior ordine e chiarezza nella gestione del *Selector* all'interno dei *buildActions()* dell'*Observer* e del *Model*. Questo per evitare ripetizioni di codice ed, in particolare, di gestire le singole eccezioni generate dalla classe *Selector*.

Il metodo da richiamare per utilizzare il *Selector* è: *getSelector(Object obj, String method)*. I parametri richiesti dal metodo, come accadeva per la classe stessa, sono:

- l'oggetto da inserire nello *schedule* per simularne l'attività
- il metodo che contiene le istruzioni relative alle azioni da compiere

La variabile *verbose*

Verbose è stata introdotta per consentire all'utente di scegliere se visualizzare o meno tutti i risultati ottenuti dalla simulazione. Una volta controllata la correttezza dei dati elaborati dal programma, infatti, conviene scegliere di non mostrare tutto quello che il programma dovrebbe portare a video per ottenere un'esecuzione più agevole e veloce.

Questa variabile viene gestita dall'*Observer*; tramite la sonda è possibile decidere *true* se si intendono mostrare tutti i risultati o *false* se si vuole avere solo una rappresentazione grafica.

Da notare che, per fare in modo che *verbose* sia compresa da tutte le classi del modello essa è stata definita come *static* ed in quanto statica deve essere inserita in una classe a sua volta statica che sia vista da tutte le altre classi. Quindi, la scelta *verboseChoice* operata nell'*Observer* deve essere trasmessa a *StartVEFrame* che contiene il metodo statico *main()*; con il comando:

```
StartVEFrame.verbose = verboseChoice;
```

Il grafico *Ratio total time / total lenghts*

Questo grafico integra l'informazione fornita dai grafici delle liste di attesa, infatti esprime il rapporto tra il tempo totale di produzione del bene e la lunghezza della ricetta di produzione del bene stesso.

La rappresentazione avviene tramite le funzioni di *EZGraph*, come per i grafici delle liste di attesa, che gestiscono i dati prodotti dal metodo *getTimeLenghtRatio()* all'interno di *vEFrameModelSwarm*. I valori prodotti sono frutto della seguente relazione:

```
return ((float) totalProductionTime)/totalRecipeLength;
```

Un'impresa che riesce a gestire correttamente la produzione senza creare liste di attesa e, quindi, ritardi dovrebbe fissare questo rapporto sull'1. Nel caso in cui la curva diventi crescente si avrà che i tempi impiegati per produrre cresceranno rispetto al dovuto e che quindi l'impresa non è in grado di soddisfare adeguatamente i clienti.

Questo rapporto è valido fino a che il tempo di produzione sarà pari ad 1 per ogni unità; nel caso in cui i tempi si differenziassero non bisognerà più considerare la lunghezza della ricetta, ma la somma dei tempi di produzione delle unità coinvolte dall'ordine.

5.1.4 I risultati ottenuti

Quando si esegue il programma vengono generati i due probes che consentono all'utente di interagire con la simulazione ed il "pannello di controllo" tipico di Swarm, si veda il paragrafo 2.3.

La prima sonda si riferisce all'observerSwarm, si veda figura 5.5; in essa si indica ogni quanti cicli dello schedule è richiesto di aggiornare i grafici.

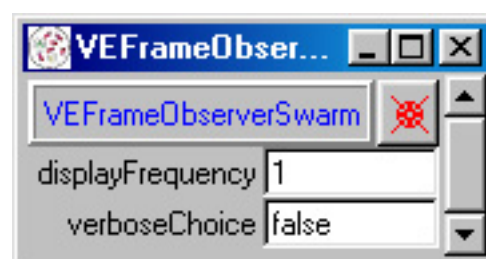


Figura 5.5: probe dell'observer

Per ora si richiede di ridisegnare con una frequenza (*displayFrequency*) pari ad uno, ovvero di aggiornare il grafico ad ogni ciclo. Questo si può fare nel caso

in cui il programma non sia ancora troppo complicato e quindi la simulazione non risulti rallentata per questioni grafiche.

La seconda sonda si riferisce al `mdelSwarm`, si veda figura 5.6; in essa si possono modificare: il numero di unità produttive presenti nell'impresa e la dimensione massima che può raggiungere il vettore degli ordini.

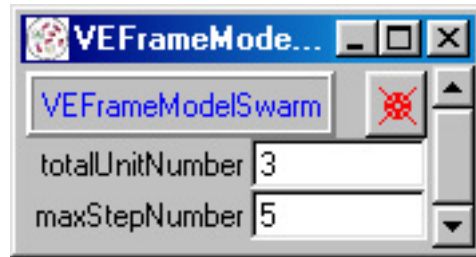


Figura 5.6: probe del model

In figura 5.6 si hanno 3 unità produttive per ordini con al massimo un codice di 5 cifre. Questa situazione genera il grafico descritto in figura 5.7

Nel grafico si possono notare tre linee, tutte in riferimento alla lista di attesa delle unità produttive. La linea arancione indica la lista di attesa più corta tra tutte le unità, senza specificare a quale unità si riferisca, la linea blu indica una media tra le liste e quella gialla le liste con il maggior numero di ordini in attesa.

Nella figura 5.7 si nota come ponendo la lunghezza del vettore dell'ordine maggiore rispetto alle unità produttive si generi una coda di ordini monotona crescente. Questo accade poiché nel vettore di ordini prodotto casualmente è molto probabile che si generino sequenze di numeri doppi, ad esempio qualora si ottenesse il vettore 311123 si avrebbe l'unità 1 costretta a svolgere 3 cicli solo per questo ordine senza considerare eventuali altri arrivi successivi.

Questo fenomeno ancora del tutto teorico può essere ricondotto alla realtà di unità produttive che svolgono ruoli primari o secondari all'interno dell'impresa e che quindi siano caratterizzate dal dover svolgere un maggiore od un minore lavoro. Il caso dell'unità 1 prima esemplificato indica in essa un sovra carico di

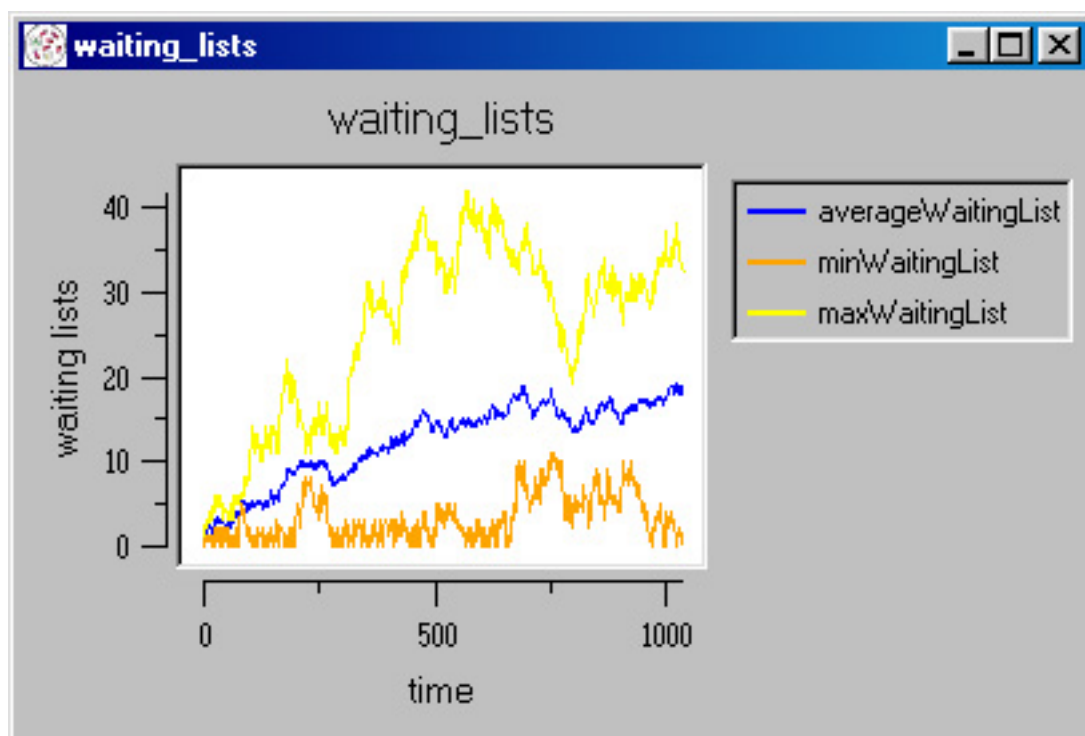


Figura 5.7: grafico del caso con 3 unità produttive ed un ordine con un vettore di max 5 cifre

lavoro che l'impresa non è pronta a gestire. Quindi con la figura 5.7 si mostra il caso di un'impresa sotto dimensionata e che necessita di una migliore gestione delle risorse, ad esempio tramite l'introduzione di un magazzino.

Questo fenomeno viene ulteriormente esasperato in figura 5.8, nella quale si hanno sequenze di ordini composti fino ad un massimo di 10 cifre per tre sole unità produttive. La situazione è immediatamente più critica, infatti mentre nel caso precedente con un vettore massimo di 5 numeri il massimo della media attesa delle unità produttive, raggiunto dopo 1000 cicli temporali, è di circa 20 ora con un vettore massimo di 10 numeri è di circa 200. Si ha quindi un peggioramento di ben 10 volte nella lunghezza media delle liste di attesa.

Caso opposto ai precedenti è quello rappresentato in figura 5.9 nel quale si hanno 10 unità produttive per ordini con al massimo un vettore di 5 cifre.

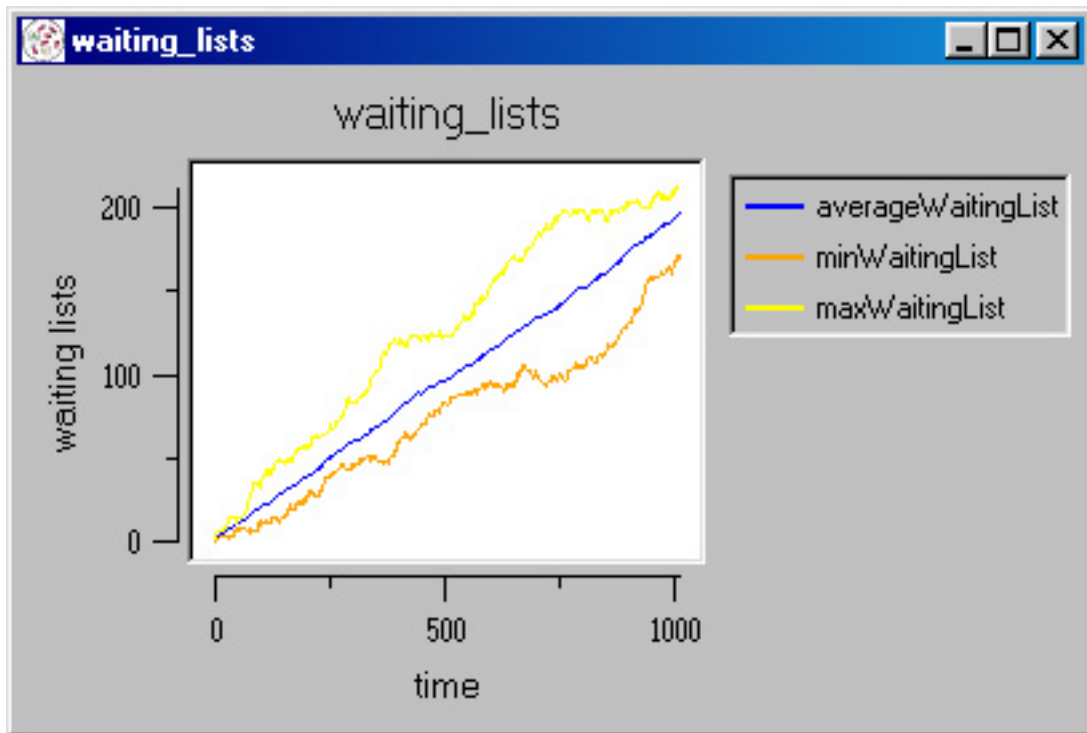


Figura 5.8: grafico del caso con 3 unità produttive ed un ordine con un vettore di max 10 cifre

Come era ragionevole attendersi le linee in questo caso si mantengono basse ad indicare che le liste di attesa vengono assorbite dall'impresa stessa. Questo è sicuramente un bene dal punto di vista di fornitura al cliente, ma non è detto che rappresenti una buona gestione dell'impresa. Infatti, se si rivolge l'attenzione alla linea arancione questa è sempre posizionata sullo 0, ad indicare che è presente qualche unità che non sta svolgendo alcun compito. Per tali unità, spesso improduttive, l'azienda sostiene costi fissi che incidono negativamente sul bilancio. Sarebbe opportuno, quindi, o ottenere maggiore lavoro oppure ridurre le dimensioni aziendali dirottando le funzioni delle unità improduttive su altre unità funzionanti correttamente.

Un ulteriore indice dell'andamento dell'impresa è mostrato dal grafico *Ratio total time / total lengths* il quale dovrebbe confermare le ipotesi maturate

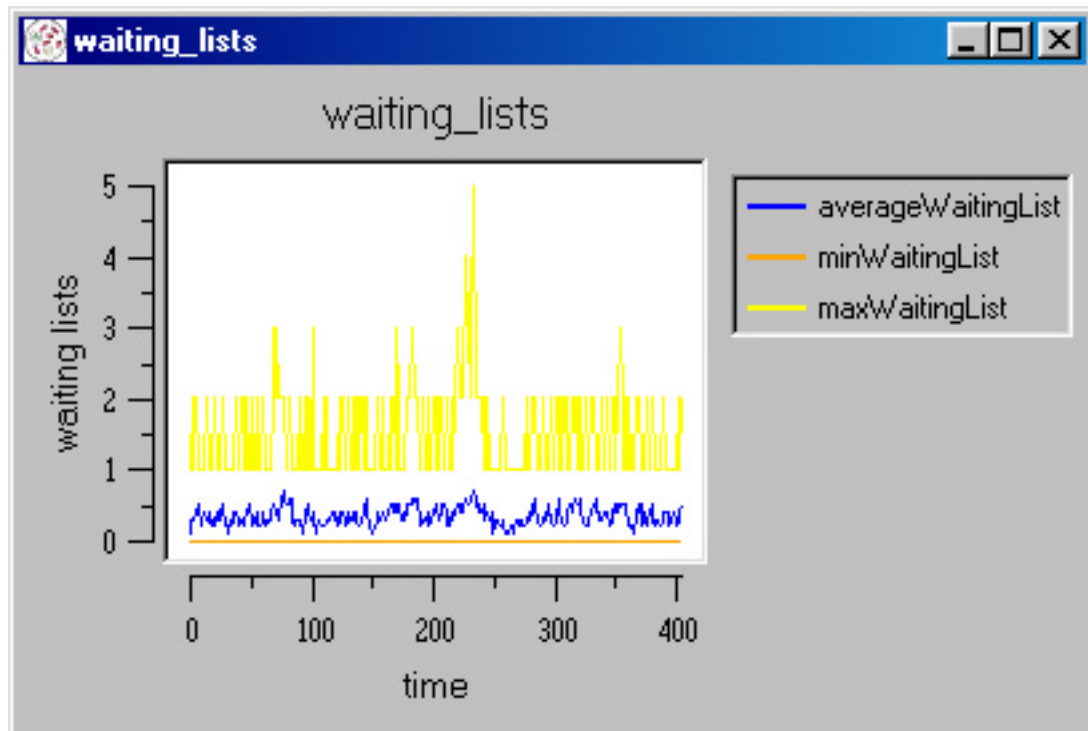


Figura 5.9: grafico del caso con 10 unità produttive ed un ordine con un vettore di max 5 cifre

dall'osservazione delle liste di attesa.

Se consideriamo il caso rappresentato in figura 5.7 con 3 unità produttive ed un vettore "ricetta" dell'ordine pari a 5 in cui si aveva un aumento costante della lunghezza della lista di attesa e riproduciamo la stessa situazione per il grafico di "Ratio" si ottiene il grafico rappresentato in figura 5.10.

Si può ora notare che trascorsi 1000 scatti dell'orologio (se si vuole 1000 giorni lavorativi) si sono formate liste di attesa che causano un rallentamento della produzione. In particolare un ordine per essere prodotto impiega circa 10 volte più tempo che in una situazione normale. Nuovamente appare la necessità di potenziare l'impresa per non compromettere la soddisfazione dei cliente sui tempi di consegna.

Una prima scelta potrebbe essere quella di duplicare le unità produttive

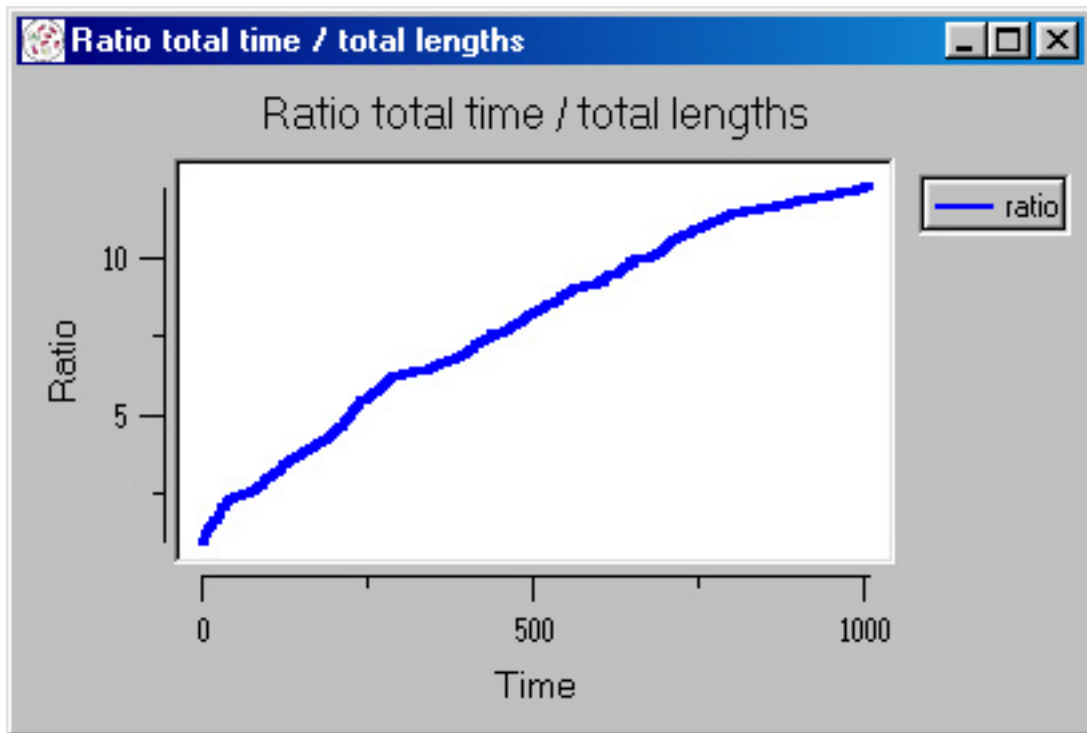


Figura 5.10: grafico del caso con 3 unità produttive ed un ordine con un vettore di max 5 cifre

che non riescono ad adempiere ai propri compiti oppure potenziare la capacità produttive di quelle già esistenti.

Alternativamente è possibile consentire alle unità di produrre beni anche qualora non ricevano ordini e di immagazzinarli in attesa di future domande. In questo modo si avrebbe un aumento dei costi variabili delle unità, piuttosto che se fossero inoperative, ma da un lato una migliore gestione dei costi fissi e dall'altro una maggiore soddisfazione del cliente.

5.2 L'introduzione dei magazzini e di un gestore di regole

5.2.1 Lo schema

Il primo passo da compiere dopo la realizzazione della struttura di base è consistita, si veda Remondino (2001), nell'introdurre dei magazzini che assorbano la produzione delle unità produttive qualora esse siano in attesa di ricevere ordini. In questo modo è possibile aumentare sia l'efficienza che l'efficacia dell'impresa virtuale; da un lato, infatti, si mantiene l'operatività delle unità anche nei periodi in cui esse sarebbero prive di lavoro, dall'altro si prevengono ordini futuri che saranno così immediatamente soddisfatti.

Da notare che, per ora, l'analisi è incentrata sul lato della produzione di beni, sull'*output*, e che, quindi, quando si parla di magazzini si fa riferimento a magazzini di semilavorati. Non viene ancora preso in considerazione l'aspetto delle risorse che pervengono alle unità produttive, quindi ai rispettivi magazzini di materie prime, in quanto si suppone che esse possano produrre quanto desiderino senza limiti economici, ma solo con limiti di tempo. Per poter compiere questo passo aggiuntivo è necessario attendere che l'impresa virtuale si completi ulteriormente.

Nell'analizzare i risultati prodotti dall'inserimento dei magazzini dei semilavorati occorre precisare che si procede esclusivamente in linea teorica dandosi che solo un caso specifico può chiarire che tipo di strategia debba perseguire una determinata azienda. Ad esempio un'impresa con prodotto fortemente stagionale produrrà sicuramente per il magazzino per buona parte dell'anno e si occuperà delle vendite nel periodo di interesse verso il suo prodotto. Altresì un'impresa che lavora su commessa difficilmente riuscirà a prevenire le richieste dei clienti e pertanto ad immagazzinare prodotti. In accordo con il principio di generalità enunciato all'inizio di questo capitolo, lo schema di *Virtual Enterprise* non com-

promette nessuna di queste ipotesi se non per il fatto che occorre impostarlo nel modo corretto.

Ad esempio si può scegliere se simulare un'impresa *con* o *senza* i magazzini, la scelta viene operata direttamente dalla *probe* del *model* sul video. Nel caso in cui esistano è possibile decidere quale sia la capienza massima che gli viene assegnata. In futuro potrà essere interessante valutare quale sia la capienza ottimale in termini di costi.

Prima di descrivere il codice che li realizza occorre spiegare come operano. I magazzini ricevono un bene tutte le volte che, eseguito un giro dell'orologio dello *schedule*, le unità non hanno alcun ordine da eseguire. In questo modo l'unità continua a produrre senza spreco di *costi fissi*. Nel momento in cui arriva un ordine di produzione viene eseguito o con beni accumulati precedentemente in magazzino oppure direttamente dall'unità se il magazzino fosse vuoto. Nel caso in cui arrivi un ordine di produzione con una sequenza di compiti da svolgere da parte della stessa unità questa soddisfa ciò che può con le scorte accumulate e, poi, se c'è ancora necessità di produrre con il proprio lavoro altrimenti ricomincia a produrre per il magazzino.

La produzione per il magazzino avviene fino a quando questi non raggiungono la capienza massima decisa dall'utente⁵.

Con l'introduzione della prima fase decisionale del modello consistente nella scelta tra produrre o meno per il magazzino è stato necessario, in accordo con lo schema E/R/A descritto nel paragrafo 1.4, aggiungere anche un gestore di regole. La gestione della capienza, infatti, non viene decisa dal magazzino, ma da un *RuleMaster* che contiene il parametro decisionale relativo alla capienza massima.

⁵Quale sia la capienza ottimale, tale da non fare crescere i costi del magazzino più del costo di attesa del cliente, non è ancora dato poiché manca ancora l'aspetto contabile.

5.2.2 Lo sviluppo del codice in Swarm

A partire dalla versione 0.3.9 il programma prevede l'utilizzo di magazzini e di un *RuleMaster* che gestisce le regole di funzionamento dei magazzini stessi. Questo passaggio ha comportato l'aggiunta di due nuovi file.java:

- Warehouse.java
- InventoryRuleMaster.java

e la modifica di altri file esistenti, in particolare:

- Unit.java
- VEFrameModelSwarm.java
- VEFrameObserverSwarm.java

Warehouse.java

Warehouse è sostanzialmente un contatore di beni; quando l'unità produce per il magazzino il contatore aumenta di 1 il suo valore, quando riceve ordini ed ha dei beni in magazzino si riduce di 1.

La classe si compone sostanzialmente di due metodi che svolgono le funzioni sopra descritte, il primo che aumenta il contatore:

```
public void increaseInventoryCounterValue() {  
    inventoryCounter++;  
    return;  
}
```

ed il secondo che lo riduce:

```
public int decreaseInventoryCounterValue(){  
    if(getInventoryCounterValue() > 0)
```

```

{
    inventoryCounter--;
    return 1;
}
return 0;
}

```

InventoryRuleMaster.java

L'*InventoryRuleMaster* controlla qual è la quantità di beni in magazzino, se essa non supera il massimo prestabilito ordina di produrre per il magazzino, altrimenti lo vieta. La struttura è pertanto molto semplice e si basa sul metodo *checkToIncreaseInventories(int uN, Warehouse aW)* che verifica se il magazzino ha ancora spazio a disposizione:

```
if (theUnitWarehouse.getInventoryCounterValue() < maxInWarehouses)
```

allora produrre:

```
theUnitWarehouse.increaseInventoryCounterValue();
```

altrimenti indicare che il magazzino è pieno.

Unit.java

I maggiori cambiamenti causati dall'introduzione dei magazzini sono avvenuti a livello di unità produttive.

Innanzitutto è stato introdotto un secondo costruttore della classe, per poter elaborare sia il caso *con* che il caso *senza* magazzini. In questo modo si vincola la creazione degli oggetti di tipo *Warehouse* ed *InventoryRuleMaster* ad un loro effettivo utilizzo da parte delle unità produttive. Questo è stato possibile grazie al *polimorfismo* delle classi *Java* descritto nel paragrafo 2.1.2.

In *unitStep1()* si inizia la *giornata virtuale* di produzione controllando se ci sono ordini in lista di attesa; in caso di risposta affermativa si prende il primo della lista, lo si inserisce in una lista di prodotti giornalieri (*dailyProductionList*) e si indica all'ordine che la produzione è stata eseguita:

```

if(waitingList.getCount() > 0){
    firstOrder=(Order)waitingList.removeFirst();
    dailyProductionList.addLast(firstOrder);
    firstOrder.setDoneStep();
}

```

altrimenti se non ci sono ordini, ma ci sono dei magazzini, l'unità produce per loro:

```

else if(useWarehouses)
    inventoryRuleMaster.checkToIncreaseInventories(unitNumber, myWarehouse);

```

Se, inoltre, vengono usati i magazzini e la lista di attesa non è vuota si indica di continuare a diminuire i beni in magazzino fino a soddisfare gli ordini in questione:

```

if (useWarehouses){
    while(waitingList.getCount()>0 && ! myWarehouse.empty()){
        myWarehouse.decreaseInventoryCounterValue();
        firstOrder=(Order)waitingList.removeFirst();
        firstOrder.setDoneStep();
    }
}

```

In questo modo un'unità che in passato ha ottenuto poche ordinazioni può mettere comunque a frutto il lavoro svolto velocizzando la produzione attuale. Nel caso in cui il passo successivo dell'ordine sia diverso dalla produzione dell'unità in cui si trova si indica all'unità di prenderlo e metterlo nella *dailyProductionList*.

In *unitStep2()*, idealmente la seconda parte della giornata lavorativa, si procede alla distribuzione degli ordini alle successive unità produttive; il procedimento consiste nello scorrere la *dailyProductionList* estraendo ordine per ordine ed interrogandolo sul successivo passo che deve compiere.

VEFrameModelSwarm.java

Nel *vEFrameModelSwarm* la prima modifica è consistita nel porre una condizione sul costruttore delle *Unit* all'interno del *buildObjects()*, in particolare:

```
if(useWarehouses)
```

e *useWarehouse* è *true* vengono anche generati magazzini e gestore di regole, in caso contrario si costruiscono solo le unità produttive.

La seconda modifica è stata di introdurre due nuove variabili nella sonda: *useWarehouses* per specificare se si desidera simulare l'impresa con o senza magazzini e *maxInWarehouses* per indicare la capienza dei magazzini.

VEFrameObserverSwarm.java

Il *vEFrameObserverSwarm* è stato modificato per aggiungere il grafico relativo alla produzione per i magazzini. E' stata utilizzata la classe *EZGraphImpl* per generare tre curve rispondenti alla massima quantità di beni presente, alla media delle quantità ed al minimo, tutte in relazione al trascorrere del tempo.

5.2.3 I risultati ottenuti

La sonda del *model* consente ora di specificare, oltre al numero di unità che si intende creare ed alla lunghezza del vettore ordine, se si intende utilizzare dei magazzini e quale capienza questi debbano avere; come mostrato in figura 5.11.

A questo punto se si lancia la simulazione appare un nuovo grafico sulle *Quantities in the Warehouses* che mostra quanti beni vengono immagazzinati dalle unità produttive che non hanno alcun ordine da eseguire.

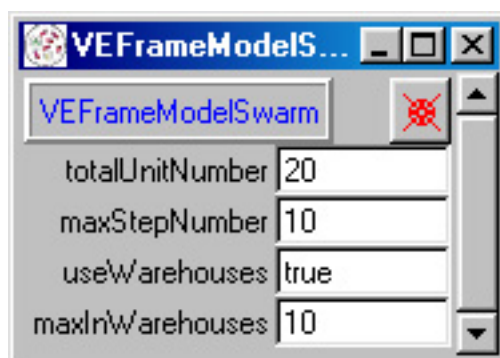


Figura 5.11: *probe* del *model* con le opzioni relative ai magazzini

Due sono le situazioni chiave che è interessante analizzare: la prima relativa al caso in cui l'impresa riesca a soddisfare a pieno i suoi doveri produttivi e trovi il tempo per immagazzinare ulteriori riserve, la seconda relativa ad una situazione di sotto dimensionamento dell'azienda che non le consente di immagazzinare alcun bene in quanto in ritardo con il lavoro quotidiano.

Se ad esempio fissiamo le unità pari a 20, la lunghezza dell'ordine pari 30 e la capacità dei magazzini pari a 20, si verifica la situazione mostrata in figura 5.12. Questo grafico mostra che fin dai primi passi della simulazione ci sono state unità che hanno potuto riempire i magazzini fino al massimo della loro capienza (*linea gialla* del grafico) e che questa situazione è valida in media per tutte le unità produttive, infatti anche la *linea blu* si posiziona su livelli molto alti. Una situazione interessante si scorge verso 400° passo dove una unità ha ridotto di molto la capienza del suo magazzino, si osservi la *linea arancione*, fino ad un minimo di circa 2-3 beni; indice che la produzione del bene specifico di tale unità subisce un incremento di domanda in quel periodo dell'”anno”. Questo grafico mostra anche che superata la prima fase in cui l'impresa ha i magazzini vuoti e, pertanto, molte possibilità produttive in più in un secondo tempo la situazione tende a stabilizzarsi in un *trend* costante dal quale si può uscire solo nel caso di nuovi e diversi impulsi esterni.

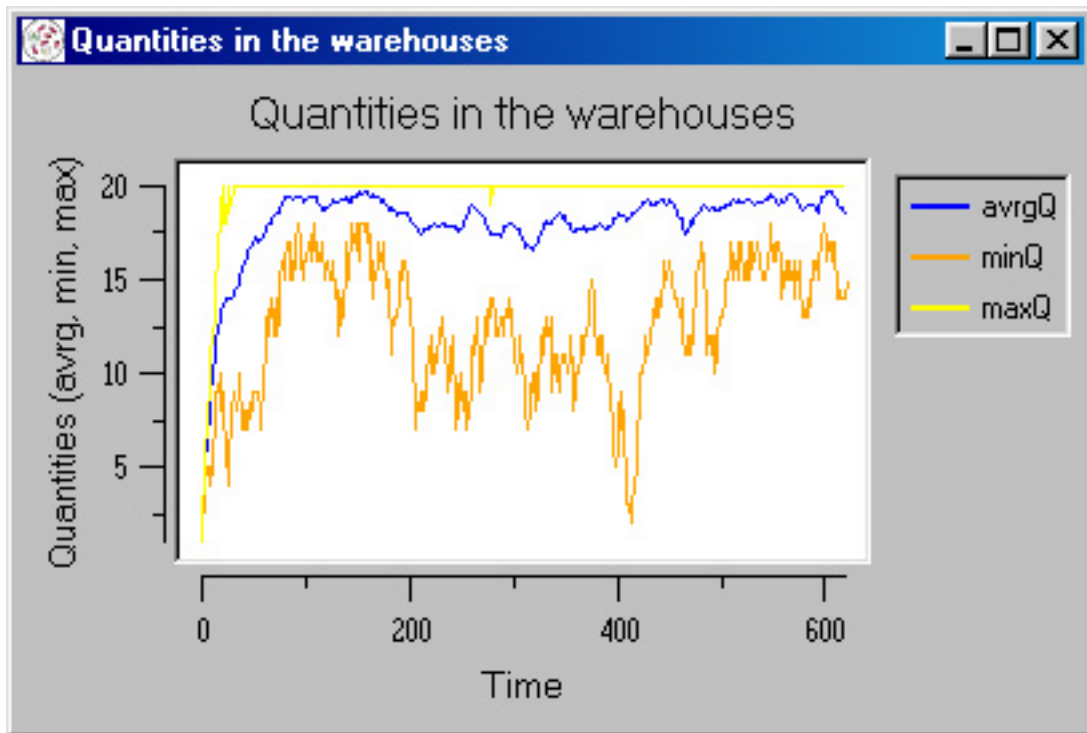


Figura 5.12: grafico delle quantità in magazzino con 20 unità produttive, un ordine con un vettore di max 30 cifre ed una capienza massima dei magazzini di 20 pezzi

La situazione opposta si verifica nel caso in cui si imposti il modello con 5 unità produttive, il vettore degli ordini pari a 20 e la capienza dei magazzini anch'essa pari a 20, come mostrato in figura 5.13. Nella prima fase della simulazione alcune unità produttive sono riuscite anche a produrre per il magazzino, fino ad un massimo di 8 beni immagazzinati, segno che la struttura d'impresa riusciva ancora a sopperire ai propri impegni. Alcune unità sicuramente si sono trovate obbligate a produrre solo per il mercato, e lo indica la linea relativa al minimo immagazzinamento che non si muove mai dallo 0, mentre altre sono riuscite anche ad immagazzinare alcuni prodotti. Con il trascorrere del tempo e l'incessante arrivo di ordini la situazione si modifica spostando tutte le curve sullo 0, segno che l'impresa non riesce più a far fronte alla domanda.

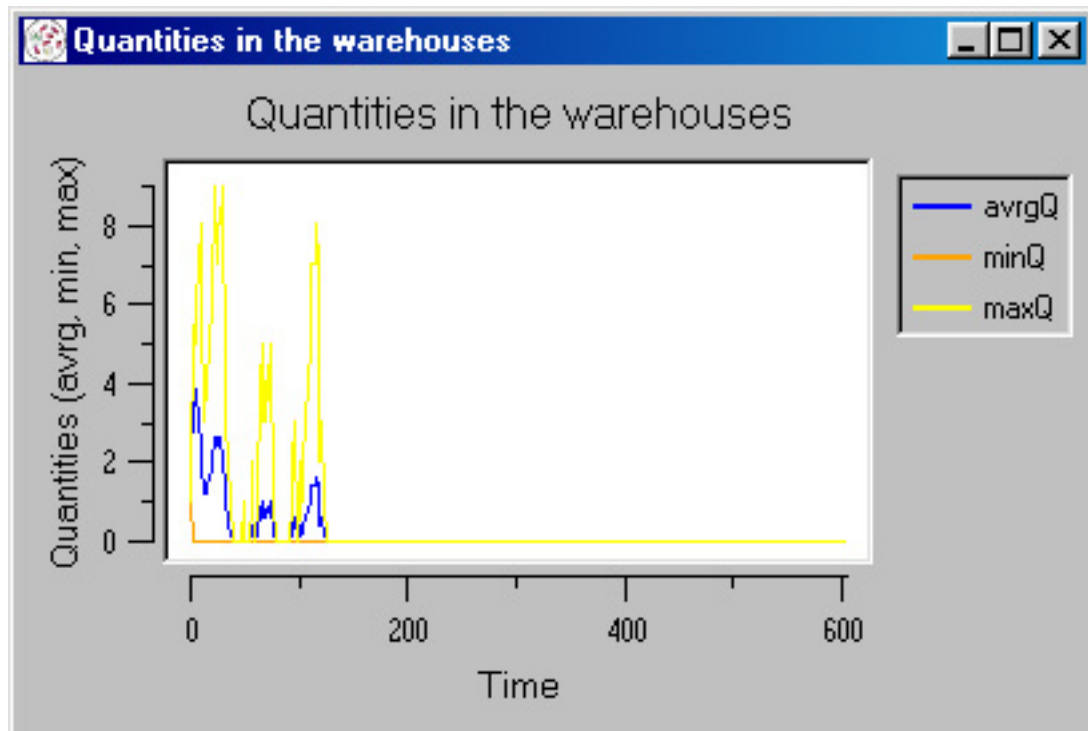


Figura 5.13: grafico delle quantità in magazzino con 5 unità produttive, un ordine con un vettore di max 20 cifre ed una capienza massima dei magazzini di 20 pezzi

Questi grafici, benché facilitino la visione di eventuali problemi nel funzionamento aziendale, non consentono all'utente di valutare *a quale unità* attribuire la causa di eventuali ritardi nella produzione oppure *quale unità* non esegua un operato proporzionale a ciò che si attende. Una valutazione dettagliata dell'andamento dell'impresa viene affrontato nel paragrafo 5.3.

Per operare un confronto tra la simulazione d'impresa con e senza l'utilizzo dei magazzini è utile ricondursi al grafico *Ratio total time / total lengths* il quale ci consente di valutare l'ottimizzazione della produzione in relazione agli ordini pervenuti. Poiché in base a questo grafico è ritenuta ottimale una situazione per la quale il rapporto tra il tempo trascorso e la lunghezza degli ordini pervenuti vale 1, considerando che per ora l'unità produttiva impiega sempre uno scatto dell'orologio per svolgere il proprio compito, possiamo notare che nel grafico di

figura 5.14 questa condizione viene rispettata fino all'80° ciclo produttivo. In altre parole fino all'ottantesima giornata lavorative l'impresa è riuscita a soddisfare adeguatamente i propri clienti.

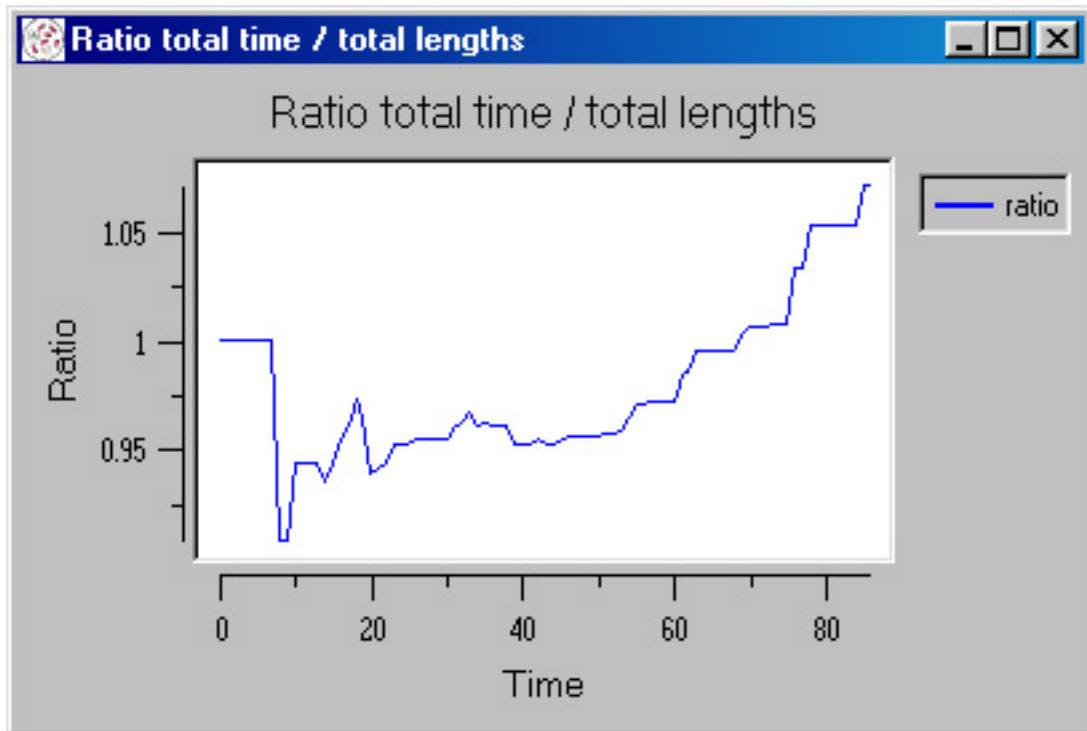


Figura 5.14: 10 unità produttive, vettore con max 20 cifre ed utilizzo dei magazzini con capacità pari a 10

Se si considera il caso senza magazzini, a parità di condizioni iniziali, la situazione, si veda la figura 5.15, si propetta molto diversa. A partire dal 10° processo produttivo l'impresa incomincia a mostrare ritardi nella produzione, con inevitabili liste di attesa per gli ordini. La curva diventa monotona crescente in quanto continua ad accumulare ritardi nella produzione ed all'80° ciclo produttivo presenta una situazione per cui si impiega il doppio del tempo previsto per produrre.

Questo indica la capacità dei magazzini di razionalizzare la produzione, almeno fino a quando non si ottiene o una saturazione degli stessi oppure un accumulo

eccessivo di ritardi da parte dell'azienda che non consente alle unità di dedicare parte della produzione all'immagazzinamento.

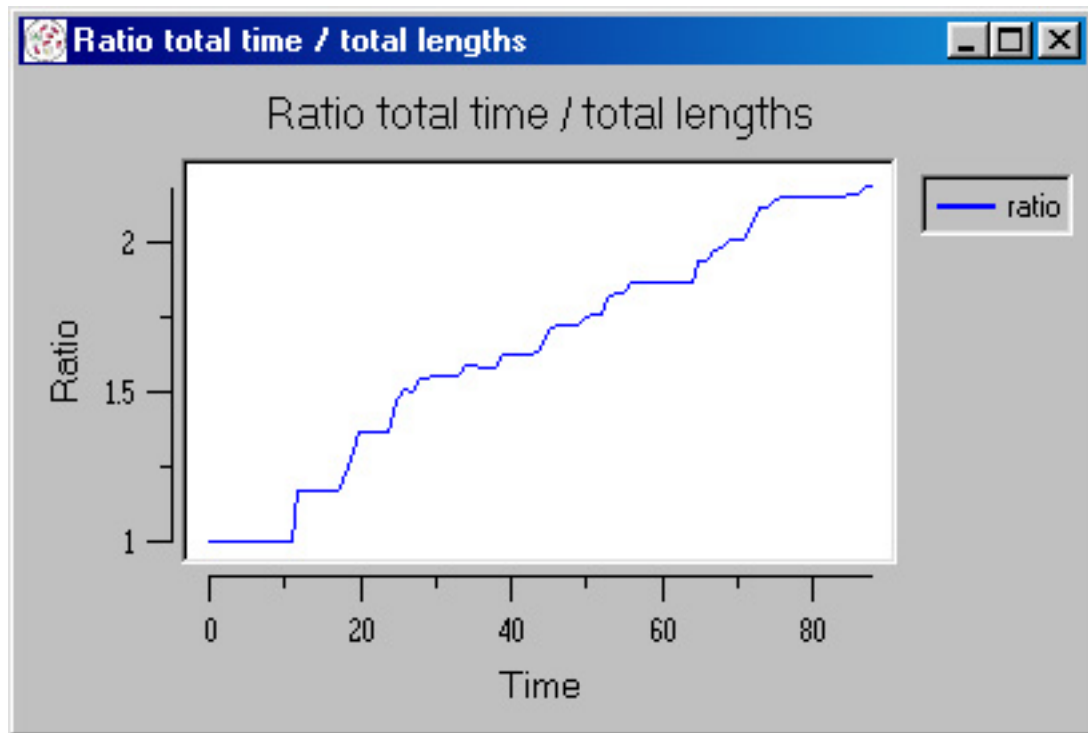


Figura 5.15: 10 unità produttive, vettore con max 20 cifre, senza magazzini

5.3 Lo sviluppo dell'ambiente grafico con Java

Nell'analisi della prima fase di sviluppo del modello è apparso l'elemento cruciale di studio delle liste di attesa delle unità produttive. L'aspetto che, però, non è ancora stato posto in evidenza riguarda quanti siano gli ordini in attesa per ogni unità produttiva. In termini economici questo implica valutare se vi siano all'interno dell'impresa unità sotto o sovra dimensionate e, nel caso, quali siano.

Il metodo migliore per operare questo tipo di analisi è rappresentare graficamente le liste di attesa per unità produttive; nello specifico produrre un diagramma a barre con le unità elencate sull'asse delle ascisse e la lista di attesa delle unità stesse sull'asse delle ordinate.

Il problema nel realizzare questo passaggio nasce poiché nel passare dalla scrittura in ObjectiveC alla scrittura in Java non sono state riportate tutte le funzioni grafiche di *Swarm*, tra le quali le classi relative ai grafici a barre. Si è reso così necessario integrare l'ambiente di *JavaSwarm* con oggetti di *Java* "puro": in particolare sono state perseguite le due strade qui di seguito illustrate.

5.3.1 La classe *BarChart*

Alcuni dettagli sul codice

La classe *BarChart*, per il cui codice si veda l'appendice D, si basa interamente sulle librerie grafiche di Java: *javax.swing.** e *java.awt.**. La sua struttura è incentrata sulle classi *JFrame* e *JPanel*, di tipo *swing*, per la costruzione della finestra in cui rappresentare il grafico e sulla classe *Graphics*, di tipo *awt*, per la rappresentazione degli elementi del grafico.

Nella versione attuale la classe, in origine elaborata da Horstman (1999) e Sonnessa (2000), prevede il seguente costruttore di oggetti:

```
BarChart(String windowTitle, double[] v, String[] n, String t, int minVal-  
ue, int maxValue);
```

Esso consente di costruire oggetti specificando: il titolo della finestra, i valori delle barre, le etichette sull'asse delle x, il titolo del grafico e le scale di valori degli assi stessi. Il suo utilizzo all'interno dell'impresa virtuale permette, in particolare, di specificare sull'asse delle ascisse il numero di unità produttiva che viene rappresentata, con *String[] n*, e sull'asse delle ordinate l'altezza delle singole barre rappresentanti la lista di attesa delle unità, con *double[] v*.

La prima modifica rispetto alla versione originale è consistita nell'aggiungere i valori *int minValue* e *int maxValue* che permettono di mantenere le proporzioni tra le barre durante la loro crescita o decrescita, esse infatti risultavano sfalsate dal ridimensionamento continuo del grafico.

La seconda modifica, puramente di carattere grafico, è consistita nell'aggiungere al metodo *paintComponent(Graphics g)* le istruzioni per disegnare una barra laterale con una scala di valori che quantifichi l'altezza delle barre. La scala di valori è suddivisa nei quattro quarti dell'altezza totale delle barre (il *maxValue* riportato nel costruttore), ogni quarto riporta un valore dinamico in relazione alla crescita delle barre. Questa modifica ha richiesto essenzialmente l'utilizzo dei metodi *drawLine*, per disegnare le linee, e *drawString*, per disegnare i valori sull'asse, entrambi della classe *Graphics*.

L'inserimento della classe in Swarm

In quanto classe puramente *Java* il suo inserimento in *Swarm* prevede alcune modifiche rispetto alla struttura classica dell'*ObserverSwarm*. Infatti, l'oggetto viene costruito nel *buildObjects()*, ma per poter passare i valori al *buildActions()* è necessario prima aggiungere un metodo che si occupi di elaborare i dati da passare alla classe *BarChart* e poi aggiungere un secondo metodo che gestisca il passaggio tra risultati ottenuti e *schedule*. Il primo metodo è stato chiamato *updateWaitingList()* e si occupa di scorrere la lista delle unità produttiva per estrarre le singole unità ed interrogarle sulla lunghezza della loro lista di attesa.

L'intero metodo viene poi gestito da un secondo, chiamato `_update_()`, che sarà richiamato nel *Selector* del *DisplayActions*.

Il risultato ottenuto seguendo questo procedimento è rappresentato in figura 5.16.



Figura 5.16: grafico a barre con 10 unità produttive

5.3.2 La classe *Histogram* di *Ptplot*

Ptplot ed il suo utilizzo

Ptplot è un pacchetto di librerie grafiche scritte in Java; si tratta di un aspetto relativo ad un progetto di modellizzazione sviluppato dall'*University of California, Berkeley*. Il progetto prende il nome di *Ptolemy* ed è ormai giunto ad un secondo stadio del suo sviluppo, si veda l'appendice A.

L'accostamento del progetto di impresa virtuale agli aspetti grafici sviluppati in *Ptplot* è derivato dalla ricerca di nuovi oggetti scritti in Java che consentissero di sviluppare un istogramma per rappresentare le liste di attesa per unità. *ptolemy.plot* è un *package* che fornisce molte classi di tipo grafico ed, in particolare, la classe *Histogram*.

Prima di procedere occorre osservare che tutte le classi grafiche del pacchetto *Ptplot5.1p1*, ultima versione ad oggi, si basano quasi interamente sulle librerie *java.swing* ed in particolare per la classe *Histogram* su: *java.swing.JComponent* e *java.swing.JPanel*. Pertanto per un corretto utilizzo di questa versione di *ptplot* è consigliabile l'utilizzo di *jdk 1.3*.

La classe *Histogram* eredita le sue principali funzioni da *PlotBox*, una classe che si occupa di integrare gli aspetti grafici di Java con le peculiarità dei grafici *ptplot*, ed aggiunge alcuni metodi utili a disegnare l'istogramma. In particolare, il metodo *addPoint(int dataset, double value)* consente di aggiungere, per un determinato *dataset*, un punto al valore desiderato; ovvero consente di disegnare punto per punto la barra relativa ad uno specifico valore.

Utilizzando questo metodo emerge l'importante concetto di *dataset*; con esso si ha la possibilità di disegnare più istogrammi sovrapposti e, quindi, di procedere ad un confronto immediato dei dati. Ogni *dataset*, rappresentato da un intero, indica un gruppo di barre che verranno rappresentate con il medesimo colore. Nel modello di impresa virtuale è possibile utilizzare questa funzione per sovrapporre l'istogramma delle liste di attesa delle unità produttive con il grafico delle quantità di beni immagazzinati, ottenendo così una panoramica completa del funzionamento dell'azienda.

L'obiettivo che queste funzionalità permettono di raggiungere consiste nell'inserire una nuova classe nel progetto di impresa virtuale che consenta di rappresentare l'attività di unità produttive e magazzino con la grafica fornita da *ptplot*.

Lo sviluppo del codice in Java ed in Swarm

L'inserimento nel progetto di impresa virtuale ha comportato l'aggiunta di un nuovo file.java:

- PTHistogram.java

e la modifica di:

- `VEFrameObserverSwarm.java`

PTHistogram eredita da *Histogram* tutte le funzioni grafiche di *ptplot* e le applica al modello di impresa virtuale.

La sua struttura si basa su tre costruttori, due attualmente in uso ed il terzo in previsione di futuri sviluppi, che contengono molte informazioni comuni e poche differenze dovute al polimorfismo della classe.

Le parti comuni per la realizzazione del grafico sono:

- la costruzione del *frame* entro cui mostrare il grafico, in particolare si fa riferimento alla classe *JFrame* della libreria `javax.swing` nel *jdk 1.3*, le cui caratteristiche consentono di impostare il titolo della finestra, di dimensionare la finestra posizionandola sullo schermo dove si preferisce, di aggiungere il grafico entro la finestra così disegnata ed infine di mostrarla a video:

```
myFrame = new JFrame();  
myFrame.setTitle(title);  
myFrame.setBounds(10,300,windowWidth>windowHigh);  
myFrame.getContentPane().add(this);  
myFrame.show();
```

- la dimensione degli assi delle ordinate e delle ascisse con le rispettive etichette:

```
setXLabel(xLabel);  
setYLabel(yLabel);  
setXRange(0,xRange);  
setYRange(0,yRange);
```

- la dimensione delle barre ed il loro posizionamento sull'asse delle ascisse.

Da notare come opera il metodo `setBinOffset()`; per il quale ogni barra è

posizionata nel *range*: $(x - \frac{w}{2} ; x + \frac{w}{2})$, dove w è la larghezza della barra e x è il valore di riferimento sulle ascisse. Passando così il valore 1 al metodo si ottiene di posizionare la barra centrata sul valore di riferimento:

```
setBinOffset(1);
setBinWidth(0.5);
```

- la legenda che definisce un'etichetta per ogni *dataset* disegnato:

```
addLegend(int dataset, String legend);
```

Le parte che differenzia i tre costruttori consiste nel numero di liste che vengono passate alla classe *PTHistogram*, rispettivamente una, due e tre. Per ora la classe, come si vedrà nelle modifiche a *VEFrameObserverSwarm*, utilizza i primi due costruttori passando o solo la lista di attesa delle unità produttive oppure anche la lista dei beni in magazzino.

Queste liste vengono utilizzate dal metodo *addPoints()* che si occupa di gestire i dati contenuti nelle liste in concomitanza con le proprietà di *Histogram* di *ptplot*.

Per quanto concerne la rappresentazione delle barre è necessario scorrere la lista delle unità ed estrarre gli oggetti contenuti:

```
for (int i=0;i<fL.getCount();i++){
    unit = (Unit) fL.atOffset(i);
```

interrogare i singoli oggetti e, se le rispettive liste non sono nulle, disegnare tanti punti quanti corrispondono alla lunghezza della lista di attesa:

```
if(unit.getWaitingListLength() != 0){
    for(int j=0;j<unit.getWaitingListLength();j++){
        addPoint(0, (double) i);
    }
}
```

Per il grafico relativo ai magazzini il procedimento è analogo, con la condizione di essere applicato solo se i magazzini vengono utilizzati.

Infine, occorre sottolineare l'importanza dei metodi:

- *clear(boolean format)*: che aggiorna il grafico ogni volta che viene richiamato; se gli si attribuisce il valore *false* aggiorna il grafico lasciando la formattazione in vigore, con *true* cancella ogni volta tutti i valori relativi al *frame*.
- *repaint()*: è indispensabile per il funzionamento di *clear()*.
- *setButtons(boolean visible)*: aggiunge alla finestra del grafico quattro bottoni che servono per formattare il grafico o stamparlo durante la simulazione.

VEFrameObserverSwarm è stato modificato nel *buildObjects()* con l'aggiunta del costruttore della classe *PTHistogram*, ma poiché si tratta di un costruttore polimorfico è stato necessario sottoporlo alla condizione di esistenza dei magazzini:

```
if(vEFrameModelSwarm.getUseWarehouses()==true)
```

nel qual caso si utilizza il costruttore con due liste, altrimenti il costruttore con una lista sola (per le unità).

Nel *buildActions()* si procede con il passaggio del metodo *addPoints()* al *Selector* dello *schedule*.

I risultati ottenuti

Il risultato ottenuto corrisponde a quello mostrato in figura 5.17.

La rappresentazione è relativa ad un'impresa con 10 unità produttiva con i rispettivi magazzini con capacità massima pari a 10 beni. Sull'asse delle ascisse sono riportate le unità produttive corrispondenti, secondo lo schema base, a numeri; infatti per ora le parti dell'impresa non hanno ancora funzioni specifiche.

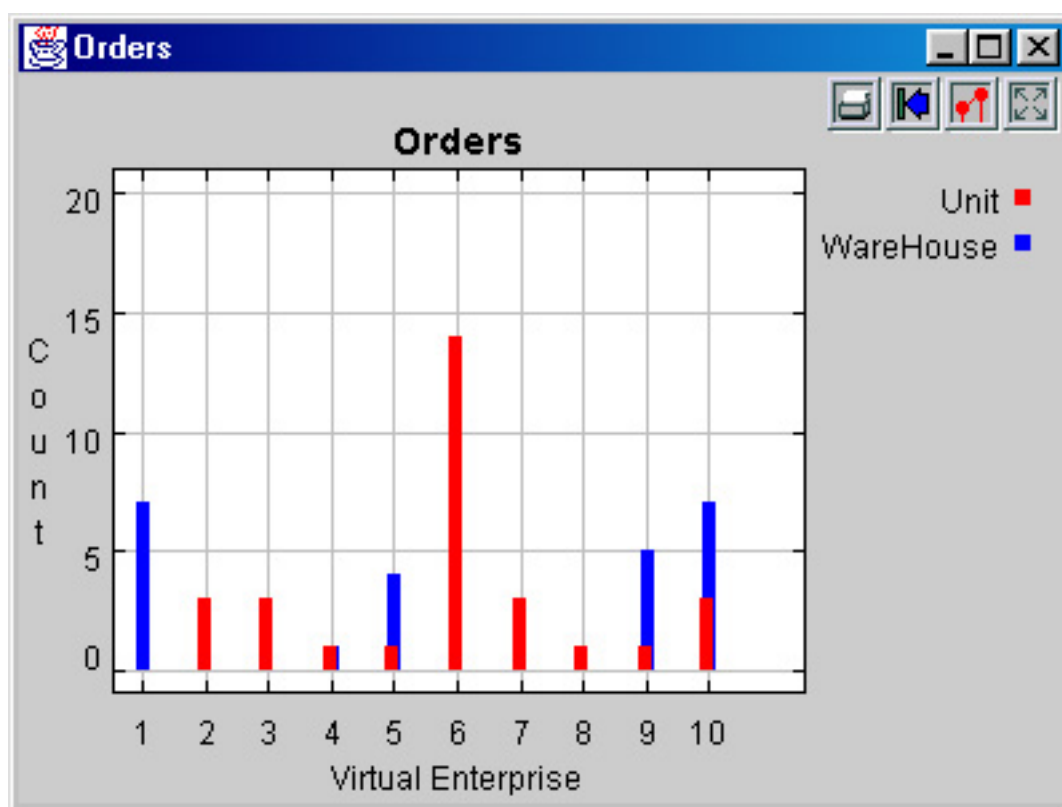


Figura 5.17: grafico a barre con 10 unità produttive, max 30 numeri nel vettore d'ordine e capienza max magazzino di 10 beni

Le barre rosse, come mostrato in legenda, corrispondono alle unità mentre quelle blu ai magazzini. La situazione rappresentata nel grafico mostra che alcune unità dell'impresa sono in forte ritardo con la produzione, l'unità 6 ad esempio non riesce più a produrre per il magazzino ed ha una lista di attesa pari a circa 14 ordini. Le motivazioni di questo ritardo non possono ovviamente essere esaminate in sede teorica, ma potrebbero rappresentare, in un caso pratico, un'inadeguatezza strutturale della sesta unità o una non equa distribuzione di risorse all'interno dell'impresa. Questa seconda ipotesi potrebbe essere sostenuta anche dal fatto che vi sono altre unità produttive quale la numero 1 che non ha alcuna lista di attesa e sta producendo esclusivamente per il magazzino. Naturalmente occorrerebbe considerare la dinamica della produzione nell'arco di un anno lavorativo,

ancorché virtuale, per poter parlare più approfonditamente, ma questo esempio è volto a mostrare l'utilità di monitorare in tempo reale la posizione delle singole parti dell'impresa.

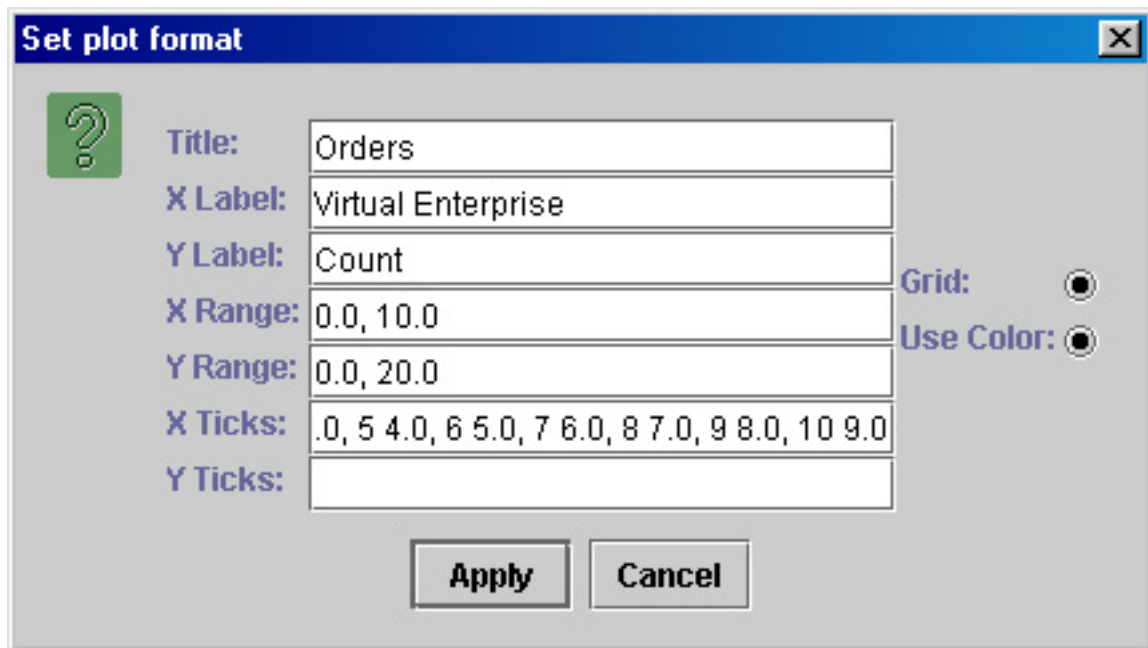


Figura 5.18: finestra di modifica formato dell'istogramma in figura 5.17

Un'ultima osservazione per sottolineare le possibilità grafiche di *ptplot* deriva dalla funzionalità dei bottoni posti sulla finestra con il metodo *setButtons(true)*. Se ad esempio si preme il tasto relativo alla formattazione del grafico appare una finestra, come quella mostrata in figura 5.18, che consente di modificare le impostazioni fatte con il codice del programma.

Problemi dei grafici Java con Swarm

L'integrazione tra l'ambiente Java e Swarm presenta ancora qualche problema.

I grafici costruiti con classi Java ed inseriti all'interno dell'*observer* non vengono bloccati dal comando *Quit* del pannello di controllo in quanto non vengono correttamente gestiti dal metodo *drop()* di Swarm. La soluzione adottata per ot-

tenere comunque l'arresto del programma nel momento desiderato è stata quella di aggiungere, nel *main()* il comando:

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

che blocca direttamente l'esecuzione del programma nel sistema.

Un secondo problema consiste nel fatto che sui grafici Java non ha effetto il comando *Save* del pannello di controllo. La soluzione adottata in questo caso è stata quella di utilizzare per la classe *PTHistogram* il metodo *setBounds()* che consente di specificare dove posizionare sullo schermo la finestra. Soluzione sicuramente meno interattiva, ma con qualche risultato.

5.4 La gestione delle informazioni

5.4.1 Informazioni e *news*

La circolazione delle informazioni tra le *business unit* di un'impresa costituisce un aspetto molto importante dell'*information technology* e dell'organizzazione aziendale. Infatti, l'impostazione delle variabili che regolano il flusso di notizie tra i settori dell'azienda possono o meno favorire i tempi e l'efficienza della produzione.

Nel modello di impresa virtuale, per ora, questo aspetto non è ancora stato sviluppato ed, infatti, tutte le *unit* sono in grado liberamente di ricevere ed inviare qualunque tipo di informazione. La necessità di vincolarlo e gestirlo nasce dal fatto che nella realtà d'azienda sia per carenze nei flussi informativi sia per problemi organizzativi non tutti sono a conoscenza di tutte le informazioni circolanti.

Per simulare questo problema sono stati posti dei vincoli al dialogo tra le unità produttive, ovvero le unità che intendono comunicare tra di loro devono prima interrogare un gestore di regole che indica se possono farlo o meno.

Nel modello le informazioni sono rappresentate metaforicamente da oggetti, quali può essere nella realtà una cartella di documenti, denominati *news*; questi contengono le notizie che fluiscono in azienda. Per ora l'unica informazione di cui si necessita è relativa alla produzione degli ordini e la *news* si occupa proprio di trasportare alle *unit* le notizie relative alle fasi di lavorazione degli stessi.

Il messaggio consiste nell'indicare alle *unit* con chi possono o devono comunicare per continuare il processo produttivo. Questa informazione è resa disponibile dall'utente che gestisce la simulazione tramite una matrice, *informationFlowMatrix*, (i, j) in cui sia il numero delle righe che il numero delle colonne corrisponde alle unità dell'impresa. Se all'incrocio tra la riga *i-esima* e la colonna *j-esima* vi è un 1 allora le corrispondenti *unit* possono comunicare, se vi è uno 0 non possono farlo.

Questa matrice viene gestita dal gestore di regole *InformationRuleMaster* che si occupa di leggerne le indicazioni.

5.4.2 Lo sviluppo del codice in Swarm

La gestione delle informazioni all'interno dell'impresa virtuale ha comportato l'aggiunta di due nuove classi:

- News.java
- InformationRuleMaster.java

e la modifica di:

- Unit.java
- Order.java
- VEFrameModelSwarm.java

News.java

La *news* porta tutte le notizie concernenti il suo utilizzo all'interno del costruttore, in cui si indica l'unità che invia la notizia, l'unità che la riceve e l'ordine che viene trasmesso:

```
public News (Zone aZone, int senderUN, int receiverUN, Order incOrder)
```

InformationRuleMaster.java

La fase di impiego di questa classe è relativa alla lettura dei dati dall'*InformationFlowMatrix*

```
data = new DataInputStream(new  
    FileInputStream("informationFlowMatrix.txt"));
```

ed alla loro registrazione all'interno di una matrice:

```

for (j=0;j<tUN;j++)
{
    InformationFlowMatrix [i][j] = Integer.parseInt(line.substring(j,j+1));
}

```

Unit.java

In *unit* sono stati aggiunti:

- un costruttore per passare alle unità l'indirizzo del *informationRuleMaster* e la "profondità" cui l'unità stessa è in grado di leggere nella ricetta dell'ordine per inviare notizie alle altre.
- in *unitStep1()*, qualora vengano utilizzati i magazzini, l'unità legge le *news* che ha ricevuto e produce, immagazzinandoli, i beni che successivamente dovrà impiegare.
- in *unitStep2()*, la *unit* guarda a quali unità successive dovrebbe inviare le *news* e chiede al gestore di regole se può farlo. Nel caso di risposta affermativa iscrive una *news* alla *newsList* con tutte le informazioni che la riguardano.

Order.java

L'ordine costituisce il contenuto della notizia che viene passata alle unità interessate. Le informazioni che porta con sé sono relative ai numeri di passi che occorre ancora compiere, quali sono i successivi e quali unità riceveranno l'ordine.

Questi passaggi sono eseguiti tramite due metodi:

- *getHowManyStepsToBeDone ()*
- *getProductionToBeDoneAtNextStepNumber (int SN)*
- *getDestinationUnitActivatedInStepN ()*

VEFrameModelSwarm.java

Dal *vEFrameModelSwarm* vengono passate, tramite costruttore, le informazioni richieste dalle unità produttive: l'indirizzo del gestore di regole ed il valore dell'*infDeepness* (quanto viene propagata l'informazione di una notizia).

5.4.3 I risultati ottenuti

Una buona rappresentazione del vantaggio relativo all'impiego delle *news* si può ottenere con il caso di un'impresa che non ha liste di attesa così lunghe da non rendere mai possibile la produzione per il magazzino o che, viceversa, non abbia mai sovraccarichi di lavoro tali da rendere inutile l'impiego di scorte.

Un caso ideale si verifica con la situazione di: 20 unità, lunghezza massima di "ricetta" dell'ordine 20, capienza magazzino illimitata e capacità di diffusione delle informazioni pari al massimo. Inoltre, ci poniamo per semplicità nel caso in cui tutte le unità sono in grado di comunicare tra loro.

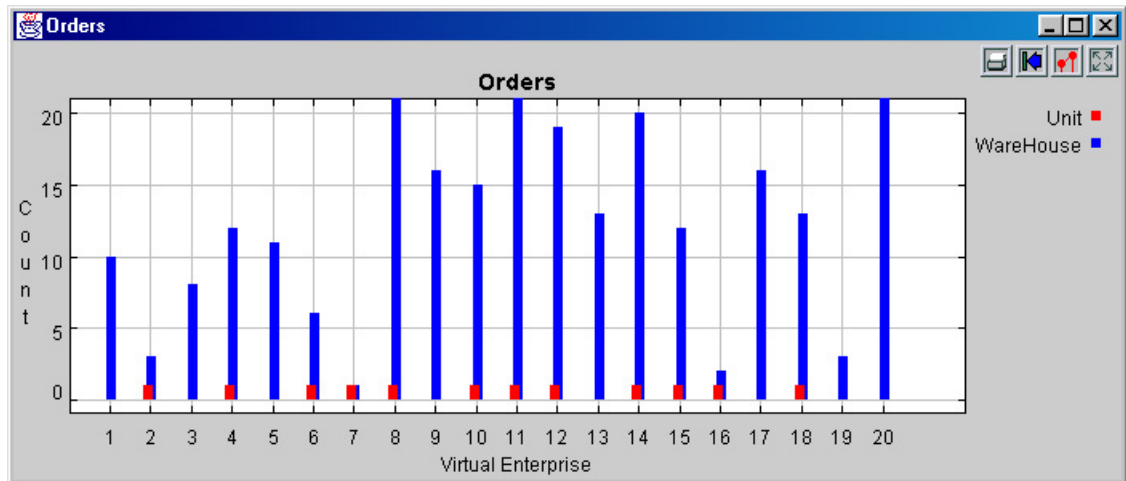


Figura 5.19: caso senza *news* al 50° step

La figura 5.19 mostra il caso con i magazzini, ma senza sistema informativo; si può notare come i magazzini siano continuamente carichi di prodotti e come

questa tendenza diventi una costante nel corso della simulazione. Questo implica maggiori costi, a causa di un eccessivo ed inutile accumulo di risorse in magazzino.

La figura 5.20 mostra il medesimo caso con l'impiego del sistema informativo. Si può chiaramente notare il migliore impiego di risorse, grazie all'immagazzinamento di quelle che è già noto saranno utilizzate nel futuro ed all'utilizzo immediato di quelle precedentemente prodotte.

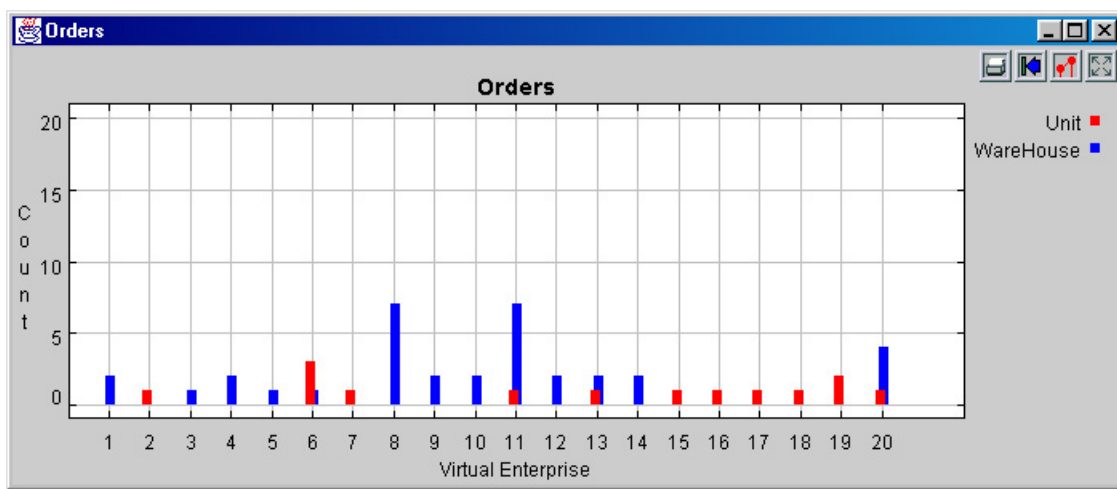


Figura 5.20: caso con *news* al 50° step

5.5 La contabilità d'impresa

5.5.1 La contabilità per *order* e per *unit*

Per valutare l'andamento dell'impresa virtuale è stato necessario introdurre la contabilità, quale metro di misura dei vantaggi o svantaggi determinati dalle scelte gestionali. Finora infatti è stato possibile osservare graficamente quali conseguenze provocassero determinate scelte, quale l'introduzione dei magazzini o del sistema informativo, sulle quantità di ordini in attesa di essere prodotti. L'utilizzo dei magazzini ha mostrato una via per migliorare il comportamento dell'impresa qualora essa si trovi troppo spesso in ritardo di produzione; situazione migliorata ulteriormente con l'introduzione del sistema informativo, che consente di anticipare l'immagazzinamento dei componenti degli ordini in arrivo.

Occorre ora misurare questi cambiamenti e valutare se convengano, economicamente parlando, all'azienda. Per questo scopo sono stati sviluppati due sistemi contabili paralleli: uno relativo ai costi delle unità e l'altro relativo agli ordini.

Le unità registrano costi variabili e dei costi fissi, nel seguente modo:

- i costi fissi vengono computati giornalmente, indipendentemente dall'attività svolta dall'unità
- i costi variabili vengono computati quando:
 1. arriva un ordine e l'unità in questione deve produrlo
 2. non arriva un ordine, ma l'unità deve produrre per il magazzino (questo passaggio implica l'utilizzo dei magazzini ed eventualmente delle *news*)

In questo modo è possibile calcolare i costi giornalieri, fissi e variabili, dell'impresa ed i costi totali come somma dei costi giornalieri.

Nel caso in cui l'impresa utilizzi dei magazzini viene imputato anche un costo finanziario dovuto all'immagazzinamento dei prodotti, il quale dipende dalle quantità che giornalmente rimangono inutilizzate.

Gli ordini registrano costi fissi e variabili nel momento in cui passano attraverso le unità e completano quella fase produttiva. Pertanto, i costi dell'unità ed degli ordini vengono registrati nel medesimo momento in un'impresa che non faccia uso dei magazzini, altrimenti in due momenti separati. L'unità contabilizza i costi variabili quando *produce*, che sia per il magazzino o per un ordine è indifferente. Invece l'ordine si fa carico dei costi solo quando viene completato sia che il pezzo sia stato prodotto in quel momento sia che fosse già stato prodotto in passato ed immagazzinato.

Gli ordini portano con sé anche l'informazione relativa ai ricavi ottenuti dall'impresa, tali ricavi sono calcolati come un prodotto tra la lunghezza della ricetta dell'ordine, metaforicamente la complessità relativa alla produzione di quello specifico prodotto, ed un tasso di *ricarica* predefinito. Poiché però è interessante conoscere in continuazione il valore dei beni in produzione, ovvero dei semilavorati, è stata introdotta una lista contenenti gli ordini attualmente in circolazione all'interno dell'impresa. Quando un ordine viene generato viene iscritto a questa lista, per essere rimosso quando la sua produzione è stata completata. Durante la simulazione la lista è in grado di restituire costi e ricavi pertinenti agli ordini in quel preciso momento.

Con la contabilità degli ordini è possibile sapere quanto costa uno specifico prodotto e quanto si spende in totale per produrre, nonché quanto l'impresa riesce a valorizzare in termini di prodotti finiti e di semilavorati.

Per entrambi i sistemi contabili si utilizzano dati di costi fissi e variabili prelevati dai files: *FixedCost.txt* e *VariableCost.txt*⁶.

I risultati vengono registrati su *file.txt* tramite un'apposita istruzione di *Swarm*.

⁶Per ora entrambi i files contengono esclusivamente valori 1 poiché in questo modo è più facile valutare la correttezza dei risultati raggiunti.

5.5.2 Lo sviluppo del codice in Swarm

L'inserimento nel codice della contabilità ha comportato l'aggiunta del file:

- `AccountingData.java`

e la modifica di:

- `Unit.java`
- `Order.java`
- `OrderGenerator.java`
- `VEFrameObserverSwarm.java`
- `VEFrameModelSwarm.java`

AccountingData.java

L'oggetto *accountingData* serve per leggere da file i valori relativi ai costi fissi e variabili. La lettura da file avviene tramite le classi *:FileInputStream* che riconosce il percorso del file su cui sono registrati i dati, *BufferedInputStream*, che posiziona il file letto all'interno del *buffer* per facilitarne una successiva lettura e *DataInputStream* che legge i dati veri e propri sul file. I valori così ottenuti vengono registrati in vettori di dimensione pari al numero di unità produttive dell'impresa (1 costo per ogni unità), numero che viene passato tramite il costruttore della classe:

```
public AccountingData(int tUN) {  
    totalUnitNumber = tUN;  
}
```

Infine, vengono restituiti i valori fissi e variabili dei costi per ogni singola unità tramite i metodi: *getFixedCost(int i)* e *getVariableCost(int j)*.

Unit.java

Le unità produttive ricevono i costi fissi e variabili tramite il costruttore e li impiegano:

- per computare i costi variabili dell'unità con:

```
addDailyVariableCost();
```

quando o produce od aumenta le risorse in magazzino:

```
myWarehouse.increaseInventoryCounterValue();
```

- per passarli all'ordine:

```
firstOrder.setFixedVariableOrderCost(fixedCost, variableCost);
```

ogni volta che ne completano una fase di lavorazione:

```
firstOrder.setDoneStep();
```

Inoltre, le *unit* registrano i ricavi generati dalla vendita dell'ordine nel momento in cui esso viene completato:

```
saleRevenue= firstOrder.getOrderPrice();  
firstOrder.drop();
```

In questo modo le unità possono restituire i costi fissi e variabili, giornalieri e totali, ed i ricavi totali.

Infine, i costi finanziari dovuti all'immagazzinamento sono cumulati nella variabile *totalInventoryFinancialCost* in base al tasso fissato dall'utente tramite la sonda del *model* e la quantità presente in magazzino:

```
totalInventoryFinancialCost +=  
(myWarehouse.getInventoryCounterValue()*  
vEFrameModelSwarm.getInventoryFinancialRate());
```

Order.java

Nell'*order* vi sono i metodi:

- *getOrderPrice()*, per calcolare il valore attuale dell'ordine in produzione, semilavorato, ed il valore del prodotto finito, nel momento in cui la ricetta di stato indichi il suo completamento. Per ottenere questo si esegue una somma dei valori del vettore di stato⁷ e la si moltiplica per un tasso scelto dall'utente.
- *setFixedVariableOrderCost(int fC, int vC)*, per ottenere dalle unità i valori dei costi fissi e variabili e cumularli in variabili specifiche dell'ordine:

```
public void setFixedVariableOrderCost(int fC, int vC){
    cumulatedFixedCost += fC;
    cumulatedVariableCost += vC;
}
```

- *getTotalCostForOrder()*, che restituisce i costi totali, fissi più variabili, e cumulati.

VEFrameObserverSwarm.java

L'*observer* viene utilizzato per stampare la contabilità su *files.txt*. Per ottenere questo scopo si è fatto uso della classe di *Swarm EZGraphImpl* ed in particolare del metodo *createTotalSequencewithFeedFromandSelector()*. Tramite questo metodo oltre che specificare l'origine dei dati, si indica anche l'estensione del file su cui si intende stamparli.

VEFrameModelSwarm.java

Il *model* è stato utilizzato per due scopi:

⁷La somma degli 1 che indicano i passi già compiuti e degli 0 relativi ai passi ancora da svolgere

- passare tramite sonda il tasso finanziario con cui calcolare i costi di permanenza in magazzino, *inventoryFinancialRate*.
- introdurre la lista degli ordini, *OrderList*, in modo che possa essere passata sia all'*OrderGenerator*, per inserirvi i nuovi ordini, sia alle *Unit*, per eliminare quelli terminati.

OrderGenerator.java

Il generatore di ordini si occupa essenzialmente di aggiungere alla lista i nuovi ordini:

```
orderList.addLast(anOrder);
```

5.5.3 I risultati ottenuti

Nella cartella corrente del progetto vi sono due *directories* denominate *Revenue* e *OutputCost* in cui vengono registrati ricavi e costi dell'azienda. In particolare:

- nella cartella *Revenue* ci sono i *files.txt*:
 1. *dailyRevenue*: registra i ricavi degli ordini prodotti per ogni step; il valore viene azzerato all'inizio di ogni giorno. Il ricavo si ottiene dal prodotto della variabile *revenueForEachRecipeStep* con la lunghezza della ricetta dell'ordine.
 2. *totalRevenue*: cumula i valori di *dailyRevenue*, quindi si ottiene il valore dei ricavi dall'inizio della simulazione.
 3. *dailyStoredComponentValue*: restituisce il valore dei prodotti in magazzino che non sono ancora stati utilizzati nel processo produttivo, la sua contabilizzazione è giornaliera in quanto ogni giorno le quantità in magazzino variano e quelle non più presenti vengono già registrate tra i semilavorati od i prodotti finiti. La sua valorizzazione avviene

sulla base del prodotto del valore indicato nella sonda del *model* con le quantità immagazzinate.

4. *SemimanufacturedProductRevenue*: restituisce il valore dei semilavorati, passo per passo; ad ogni *step* dello *schedule* si interroga l'*orderList* per sapere lo stato di completamento dell'ordine e si moltiplica la ricetta finora realizzata con il valore *revenueForEachRecipeStep*.

- nella cartella *OutputCost* ci sono i *files.txt*:
 1. *totalDailyCost*: registra i costi (fissi più variabili) giornalieri; il valore viene azzerato ad ogni passo.
 2. *totalCost.txt*: cumula i valori di *totalDailyCost*, quindi si ottiene il valore dei ricavi dall'inizio della simulazione.
 3. *OrderCostProduced*: registra il costo degli ordini terminati; giornalmente può accadere che nessun ordine sia completato e, quindi, apparire uno 0, ma può altresì accadere che siano terminati più ordini contemporaneamente e, quindi, sia stampato il loro valore cumulato.
 4. *totalSemimanufacturedProductCost*: rappresenta il costo cumulato dei semilavorati, ovvero giornalmente si somma il costo dei semilavorati nella *orderList* di cui finora si sono fatti carico.
 5. *totalInventoryFinancialCost*: registra il costo finanziario delle quantità immagazzinate, il suo valore è cumulativo di tutti i costi finanziari che l'impresa ha sostenuto giorno per giorno a fronte delle quantità che ha detenuto in magazzino.
- nella cartella *Return* compare il *file return.txt*: calcola i profitti dell'impresa sommando i ricavi, dovuti ai prodotti finiti, ai semilavorati ed alle rimanenze di magazzino, e sottrae i costi fissi, variabili e quelli finanziari dovuti all'immagazzinamento (vedere tabella 5.1). I dati relativi ai profitti sono

Tabella 5.1: calcolo del profitto nell'impresa virtuale

totalRevenueOfEnterprise	+
totalSemimanufacturedRevenue	+
dailyStoredComponentRevenue	+
totalCostOfEnterprise	-
totalInventoryFinCost	-
returnOfEnterprise	=

registrati grazie all'utilizzo di *EZGraphImpl* che consente sia la scrittura su file sia una rappresentazione grafica.

Bibliografia

- [1] Ormezzano N.(2001), *Un modello ad agenti di impresa virtuale in JavaSwarm con capacità contabili*, Tesi di Laurea (in collaborazione).
- [2] Pelligra P.(2001), *Un modello ad agenti di impresa virtuale in JavaSwarm: la circolazione delle informazioni*, Tesi di Laurea (in collaborazione).
- [3] Remondino M. (2001), *Analisi dei processi aziendali mediante un modello ad agenti in JavaSwarm, con scorte*, Tesi di Laurea (in collaborazione).
- [4] Sonnessa M. (2000), *La previsione degli ordini attraverso la simulazione di una catena di fornitura: un caso aziendale*, Tesi di Laurea.

Capitolo 6

La simulazione ed i risultati

In questa sezione si analizzano i risultati derivati da simulazioni con il modello di impresa virtuale sulla base dei grafici e dei dati contabili registrati. Lo scopo non consiste nel valutare se l'impresa sia in grado di guadagnare o meno sul mercato, non ancora perlomeno. L'obiettivo è di valutare l'adeguatezza del modello a situazioni plausibili nella realtà al fine di poterlo utilizzare, in futuro, per applicazioni pratiche. Occorre pertanto affermare che i dati inseriti nel modello non hanno alcun valore reale, ma consentono una facile lettura dei risultati ed, eventualmente, miglioramenti al modello stesso.

6.1 La prima fase di simulazioni

L'inizio della valutazione dei risultati emessi dall'impresa virtuale è consistito nel provare a forzare il modello per ottenerne risposte in merito alla struttura più idonea in base alle condizioni di mercato in cui si trovi.

Il primo esperimento¹ ha visto l'impostazione delle variabili così come vengono riportate in figura 6.1.

¹Le simulazioni sono tutte eseguite per un periodo di circa 200 *step*, i giorni lavorativi annuali; la scelta è totalmente arbitraria ed eventuali scelte differenti sono segnalate nel corso degli esempi.

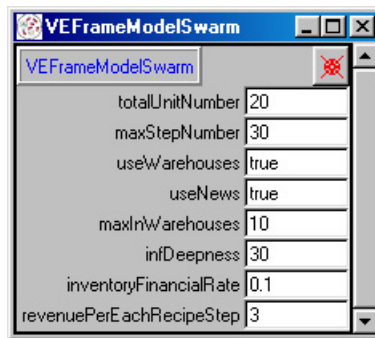


Figura 6.1: probe relativa al caso di un'impresa sotto dimensionata con valorizzazione pari a tre di tutte le fonti di ricavo

Si è pertanto nel caso di un'impresa lievemente sotto dimensionata in quanto possono arrivare ordini di lunghezza 30 a fronte di una capacità produttiva aziendale di 20 unità.

Da notare anche l'impiego del valore 3 per ricaricare sia i prodotti finiti, sia i semilavorati e sia i beni immagazzinati pur senza alcuna richiesta di produrli. L'interesse nei confronti di questo valore emerge dal fatto che ad ogni *step* le unità contabilizzano sicuramente un costo fisso e, se producono, un costo variabile, che sia esso per il magazzino o per un ordine in arrivo. Pertanto, a fronte di un costo massimo giornaliero di 2, si impone una maggiore valorizzazione del bene prodotto pari al 50%.

Un aspetto che genera ancora qualche dubbio è inerente al valore che assume *inventoryFinancialRate*; esso costituisce per ora l'unico costo aggiuntivo per le imprese che decidano di produrre per il magazzino. Nel caso sia sottovalutato è probabile che a fronte di un ricavo potenziale di 3, per un bene immagazzinato, non si riesca nel tempo a pareggiare il valore dei costi cumulati per il suo immagazzinamento e quindi l'impresa ottenga un profitto insperato.

I casi esaminati per un confronto dei risultati sono tre: il primo analizza un'impresa con i magazzini che gestisce le informazioni, il secondo un'impresa con i soli magazzini ed il terzo un'impresa senza *news* e *warehouses*.

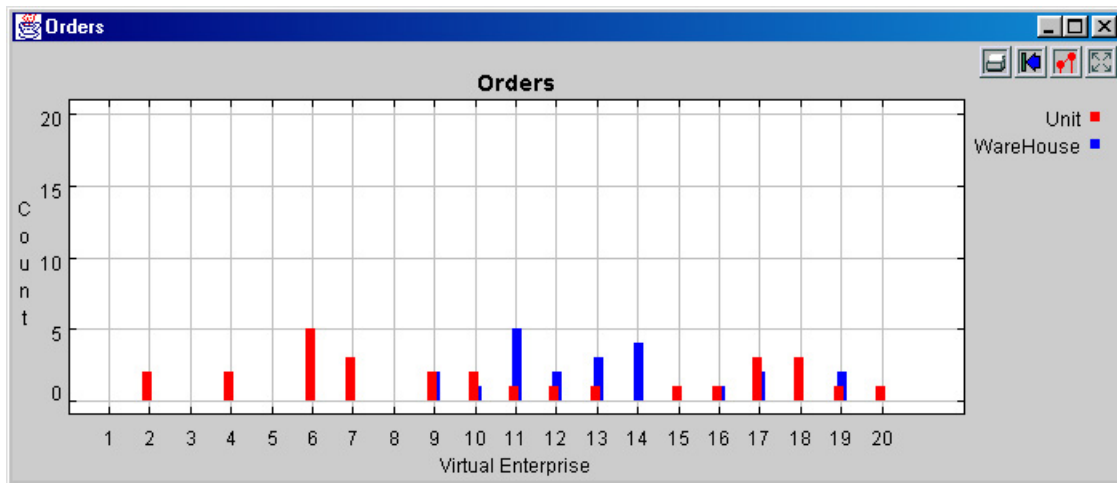


Figura 6.2: grafico delle liste di attesa, impresa con *news* e *warehouses*

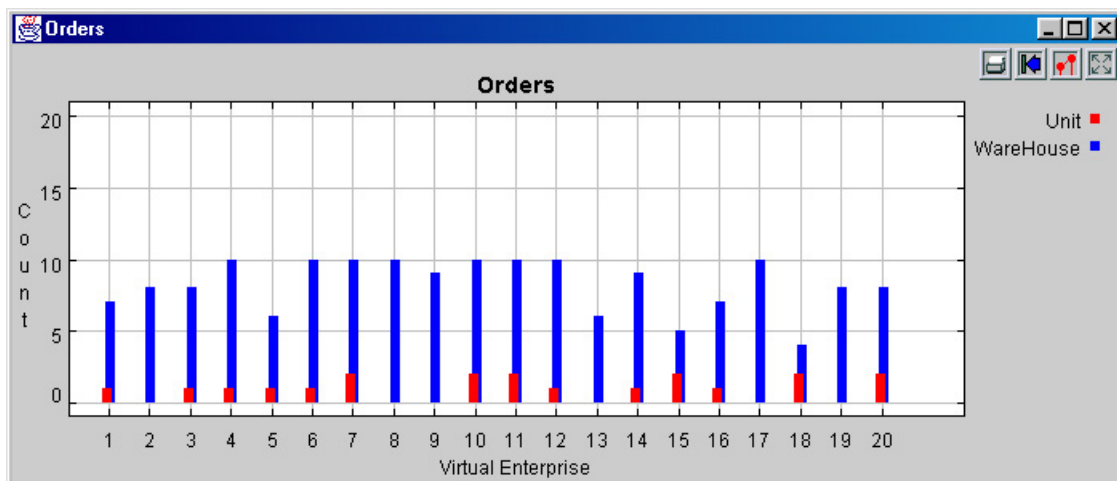


Figura 6.3: grafico delle liste di attesa, impresa con *warehouses*

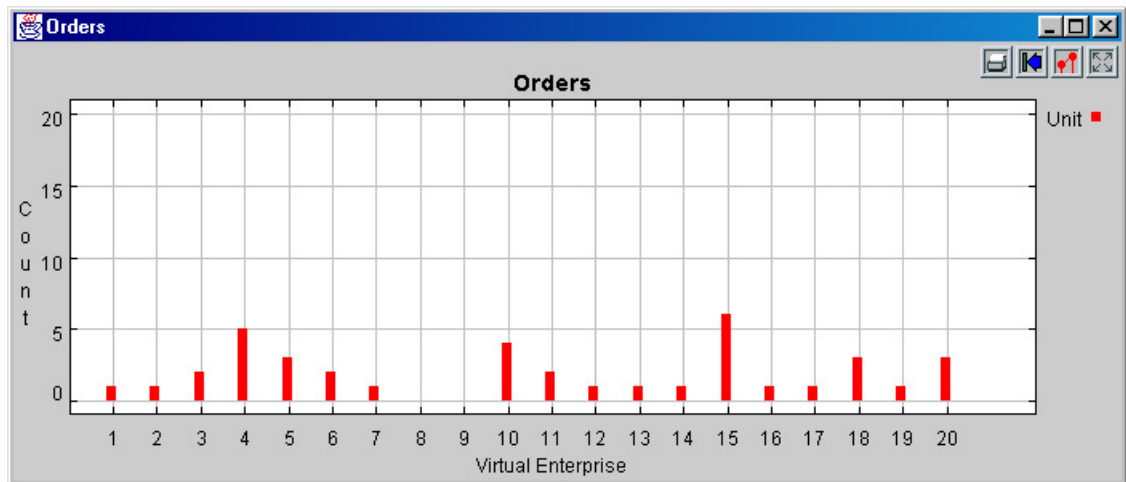


Figura 6.4: grafico delle liste di attesa, impresa senza *news* e *warehouses*

Dai grafici relativi alle liste di attesa degli ordini ed alle quantità immagazzinate si possono già notare i diversi cambiamenti dell'impresa a seconda dell'impostazione stabilita.

In figura 6.2 si può notare come l'utilizzo delle informazioni consenta all'impresa di regolare le quantità da immagazzinare rispetto agli ordini pervenuti. Per l'unità 11, ad esempio, i 5 pezzi immagazzinati, barra blu, servono per soddisfare le richieste in arrivo, quali quella in lista di attesa rappresentata dalla barra rossa. Altre unità quale la 3 o hanno appena soddisfatto tutte le attese con le scorte di cui disponevano oppure sono in attesa di ricevere l'informazione di produrre per gli ordini in arrivo, in questo momento pertanto registrano solo costi fissi.

Nel caso in cui l'impresa utilizzi solo i magazzini e non le informazioni, si veda figura 6.3, la situazione appare molto diversa. Tutte le unità hanno i magazzini saturi o quasi, fino al limite *maxInWarehouses* fissato a 10. La scelta del valore è determinata dal fatto che è opportuno limitare la possibilità di produrre se non si è effettivamente ricevuto un ordine di acquisto, altrimenti l'impresa potrebbe ottenere molti ricavi potenziali solo per il fatto che produce per se stessa. Le quantità immagazzinate consentono di tenere sotto controllo le liste di attesa del-

l'unità e la produzione è talmente impegnata che nemmeno i magazzini riescono ad essere sempre saturi.

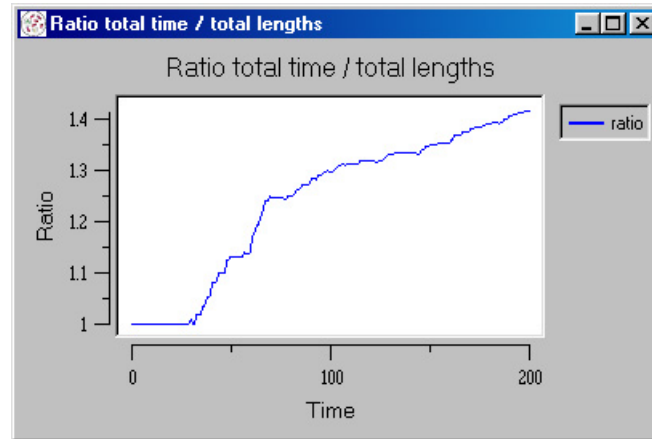


Figura 6.5: ratio con *news* e *warehouses*

L'ultimo caso rappresentato in figura 6.4 indica un'impresa senza informazioni e magazzini, che pertanto deve attendere di produrre gli ordini nel momento in cui arriva la richiesta effettiva. Le liste di attesa sono ovviamente più lunghe e pertanto si riesce a completare un minor numero di prodotti.

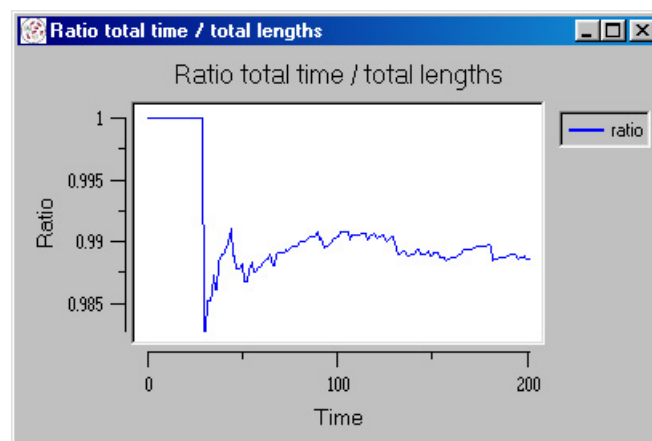


Figura 6.6: ratio con *warehouses*

Le informazioni desumibili dai grafici delle liste di attesa sono confermate dal

comportamento del *ratio*, ovvero dal rapporto che indica il tempo impiegato per produrre un ordine.

In figura 6.5, con l'utilizzo delle *news*, si può notare come dopo 200 step la produzione di un ordine impieghi $1.4 \cdot \text{tempo previsto}$; nettamente migliore la situazione relativa al caso di soli magazzini, in cui si soddisfa il cliente addirittura in anticipo rispetto ai tempi previsti.

Il caso peggiore riguarda l'impresa che opera irrazionalmente, la quale impiega quasi 2 volte il tempo necessario per produrre.

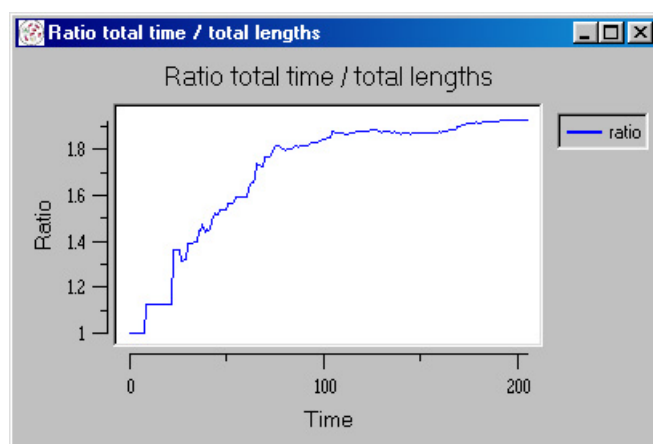


Figura 6.7: ratio senza *news* e *warehouses*

In ultima analisi del caso in questione occorre valutare contabilmente quanto economicamente convenga all'impresa una situazione piuttosto che l'altra. Un riscontro grafico, vedere figura 6.8, consente di illustrare il confronto tra i profitti delle tre situazioni sopra prospettate.

Il caso migliore è rappresentato dall'utilizzo dei soli magazzini (*with_W*), come i grafici precedenti lasciavano intuire; la curva assume un andamento crescente fin dall'inizio della simulazione, per poi mantenerlo durante tutto il percorso. Questo avviene poiché fin dall'inizio vengono contabilizzati dei ricavi potenziali per le scorte di magazzino, non solo ma essi vengono anche valorizzati ad un valore superiore al costo sostenuto per produrli (precisamente 3 contro 2).

La curva con le *news*(with_W&N) è sotto la precedente, pare, proprio perché all'inizio non accumula indiscriminatamente prodotti in magazzino, ma si limita a valutare ciò di cui effettivamente necessita; così, nel prosieguo della simulazione, si trova incapace di colmare il *gap* che si è formato. Tale difficoltà deriva dall'inadeguatezza del costo finanziario attribuito alle scorte, ovvero si penalizza troppo poco il fatto che l'impresa accumuli materiale senza alcuna logica. Occorre ribadire che la scelta dei parametri, allo stato attuale del progetto, non consente alcuna logica paragonabile con la realtà.

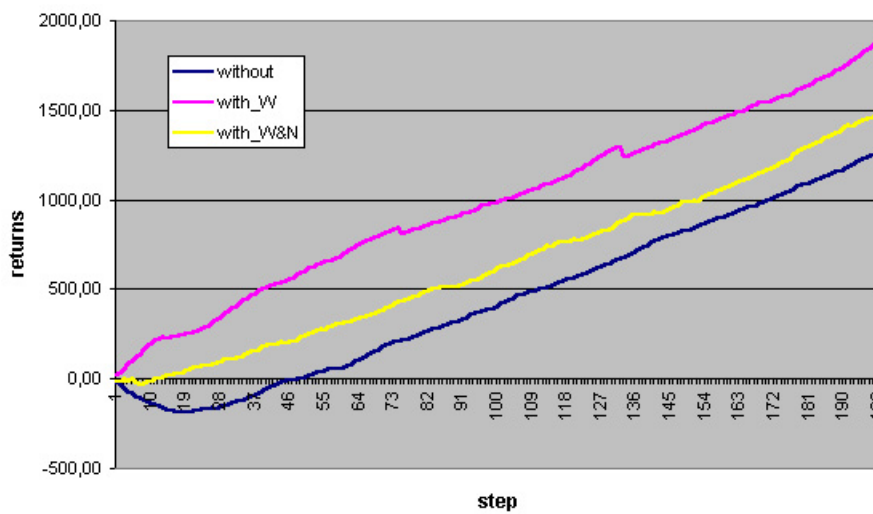


Figura 6.8: confronto impresa sotto dimensionata con valorizzazione dei ricavi pari a 3

Il caso che mostra i risultati peggiori è quello relativo all'impresa sprovvista di informazioni e magazzini. In una prima fase si può notare la formazione di una perdita dovuta ai costi sostenuti per *iniziare* l'attività e non ammortizzabili direttamente con l'accumulo di beni sotto forma di scorte. Poi si nota la formazione di utili anche in questa situazione, non appena incominciano ad essere venduti dei prodotti. Liste di attesa più lunghe e incapacità di immagazzinare risorse rendono impossibile un recupero sulle altre situazioni.

In antitesi con l'esempio precedente si può ora valutare il caso di un'impresa

sovra dimensionata rispetto agli ordini in arrivo; supponiamo ora di disporre sempre di 20 unità produttive, ma di ordini con ricette lunghe solo 10 passi (vedere figura 6.9).

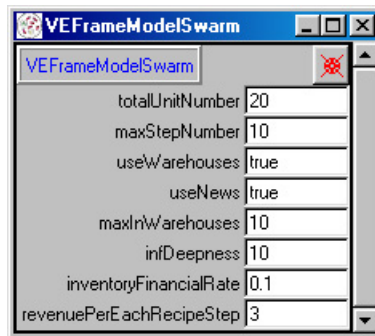


Figura 6.9: probe relativa al caso di un'impresa sovra dimensionata con valorizzazione pari a tre di tutte le fonti di ricavo

Anche in questo caso teniamo invariato il valore relativo alla valorizzazione dei ricavi pari a 3 ed il valore massimo delle scorte rimane fisso a 10 unità. Infatti se, come prevedibile, l'impresa avrà molto tempo per produrre per il magazzino, poiché non riceve ordini, occorre impedire che ne tragga anche un vantaggio.

I risultati più interessanti sono mostrati dal grafico di confronto tra i profitti delle tre situazioni, si veda figura 6.10. Rispetto al caso precedente vengono mantenute le posizioni, il caso migliore è sempre quello con i magazzini, ma la sua posizione è incrementata solo dal fatto che in un primo tempo può contabilizzare le rimanenze di magazzino. Il caso con le *news*, che prevede di immagazzinare meno specie se non ci sono ordini in arrivo, segue molto di più l'andamento dell'impresa senza valorizzazioni. In pratica nel caso in cui l'impresa sia più ampia del dovuto non ottiene vantaggi dall'aver immagazzinato beni, sia razionalmente sia irrazionalmente, anzi probabilmente con il tempo subirà il ritorno dei costi finanziari.

Inoltre, è da notare come in questo caso neanche una valorizzazione pari a 3

dei prodotti riesce a coprire i costi fissi dell'impresa che, pertanto, risulta sempre in perdita.

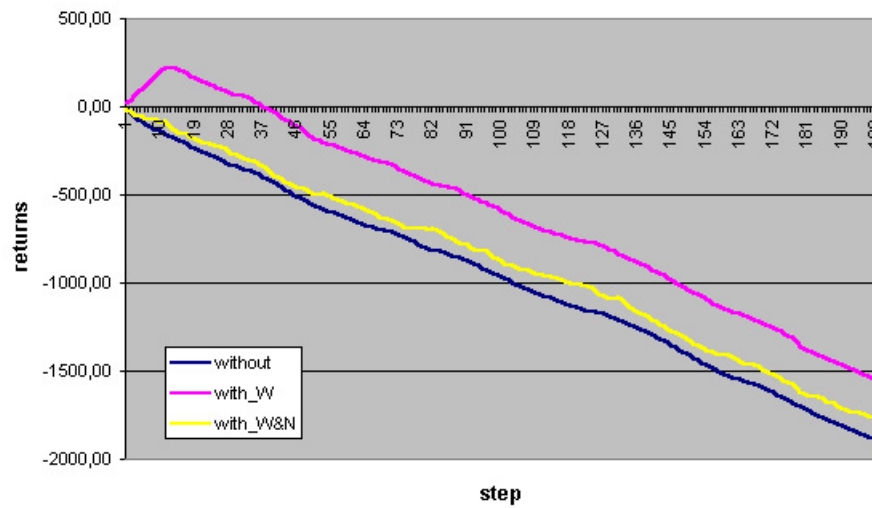


Figura 6.10: confronto impresa sovra dimensionata con valorizzazione dei ricavi pari a 3

6.2 Perfezionamenti della simulazione

In questa seconda parte di simulazioni si è cercato di risolvere due problemi: l'eccessiva rigidità con cui si calcolava la valorizzazione delle risorse in magazzino e la disparità, spesso troppo evidente, che si nota tra le risorse immagazzinate nel caso con *news* rispetto al caso senza. Le soluzioni adottate sono state le seguenti:

- introdotta la variabile *inventoryEvaluationCriterion* che fissa i valori da utilizzare per valorizzare i beni immagazzinati, vedere tabella 6.1.

Tabella 6.1: corrispondenza di valori di *inventoryEvaluationCriterion* con le variabili del modello

3	= revenuePerEachRecipeStep
2	= (fixedCost + variableCost)
1	= variableCost
$\neq 1, 2, 3$	= message of error and System.exit(0)

Questi valori vengono giornalmente moltiplicati per le quantità presenti in magazzino.

- introdotta la variabile *minInWarehouses* che, nel caso con *news*, indica al modello di continuare a produrre per il magazzino, almeno fino al minimo, anche se non ci sono notizie in arrivo.

I risultati dovuti alle modifiche si notano immediatamente dal confronto del caso proposto in figura 6.1 con quello qui riproposto nella nuova versione, si veda figura 6.11.

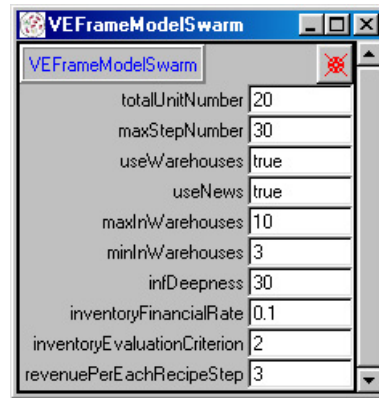


Figura 6.11: probe con le modifiche di *minInWarehouses* e di *inventoryEvaluationCriterion*, impresa sottodimensionata

Innanzitutto è possibile verificare, nel caso si gestiscano le informazioni, se effettivamente l'introduzione di un minimo di produzione per i magazzini consenta di migliorare i risultati dell'impresa.

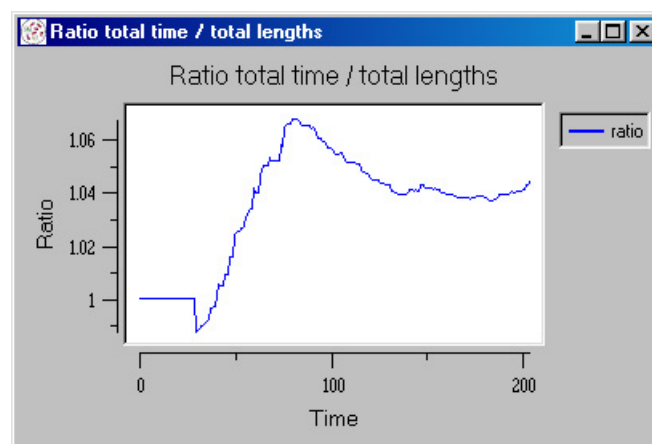


Figura 6.12: ratio con *news*, *warehouses* e *minInWarehouses* pari a 3

Un buon riscontro lo si ottiene con il grafico del tempo medio di produzione (*ratio*). Se si confronta la figura 6.5 con la 6.12, si può notare come nel caso senza un minimo di scorte prefissato il ritardo accumulato nella produzione sia superiore di un valore pari a circa 0.4, con un conseguente peggioramento dei rendimenti. Questo accade, come mostra la figura 6.13, in quanto l'impresa riesce a tutelarsi da eventuali sequenze di ordini molto lunghe.

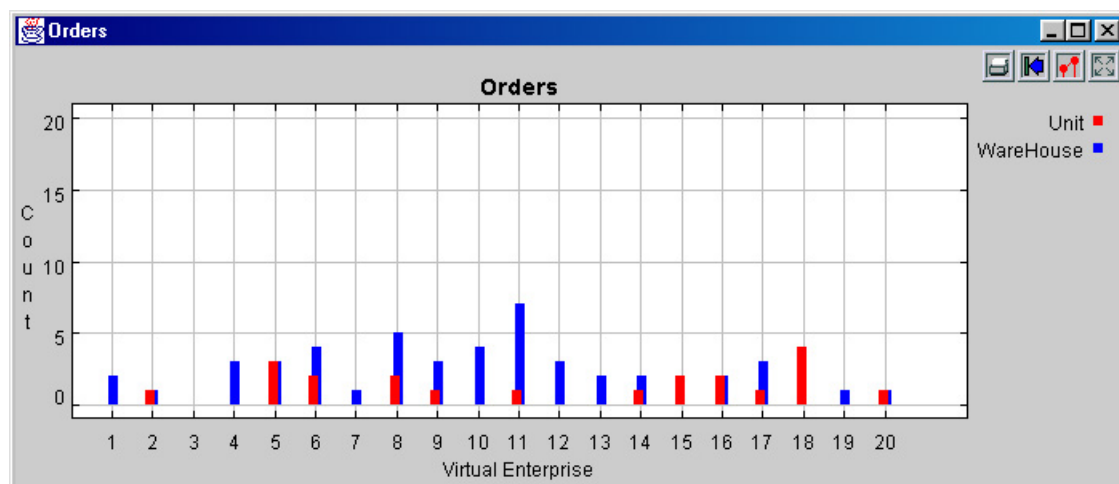


Figura 6.13: grafico delle liste di attesa, impresa con *news*, *warehouses* e *minInWarehouses* pari a 3

Infatti, nonostante questa posizione cautelativa ci sono alcune unità (quali la 3^a) che sono sempre a rischio di formare liste di attesa.

Questa prima modifica pare aver dato buoni risultati nell'ambito del miglioramento del comportameto dell'impresa, qualora utilizzi la gestione delle informazioni.

Per ciò che concerne la seconda modifica, inerente alla valorizzazione dei beni immagazzinati, occorre confrontare i risultati ottenuti in figura 6.8, che presupponavano una valorizzazione sempre pari a quella richiesta sul mercato per i prodotti

finiti ed i semilavorati (nello specifico 3)², con le due nuove ipotesi formulate.

Il primo esempio preso in esame prevede l'utilizzo di *inventoryEvaluationCriterion* = 2, caso in cui si valorizzino i prodotti per il magazzino pari alla somma di costi fissi e variabili. Sostanzialmente non si esegue alcuna ricarica rispetto al loro costo effettivo.

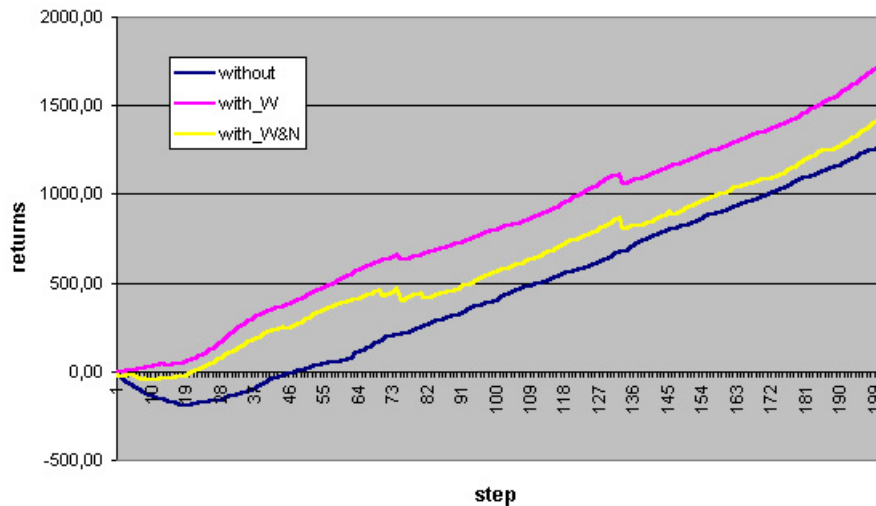


Figura 6.14: confronto impresa sotto dimensionata con *inventoryEvaluationCriterion* = 2

La figura 6.14 mostra come i valori relativi al caso dei soli magazzini si siano abbassati e, in particolare, se si osserva la prima fase della curva nel caso di valorizzazione 3 si nota che si impenna molto più rapidamente che nel caso ora analizzato. Risultato atteso in quanto non si permette all'impresa di contabilizzare in eccesso la prima fase di sola produzione per il magazzino.

Ancora più evidente è il caso di figura 6.15 in cui si valorizzano le rimanenza per il solo costo variabile. La produzione per il magazzino a questo punto non conviene più ed, infatti, la curva che la rappresenta coincide per larga parte con quella delle *news*. Entrambe risentono della perdita subita ad ogni passo, fino a

²Il caso formulato nel paragrafo 6.1 corrisponde a definire *inventoryEvaluationCriterion* = 3 nell'attuale versione.

quando non vengono quasi raggiunte dalla curva dell'impresa senza particolarità. Da notare come risenta di questa influenza anche il caso con le *news*, problema dovuto all'inserimento di *minInWarehouses* pari a 3; la curva azzurra mostra come la situazione possa migliorare se, in questo caso si fissa il minimo da immagazzinare pari a zero, ovvero la gestione delle informazioni è più efficiente se è possibile avere anche delle unità inoperative.

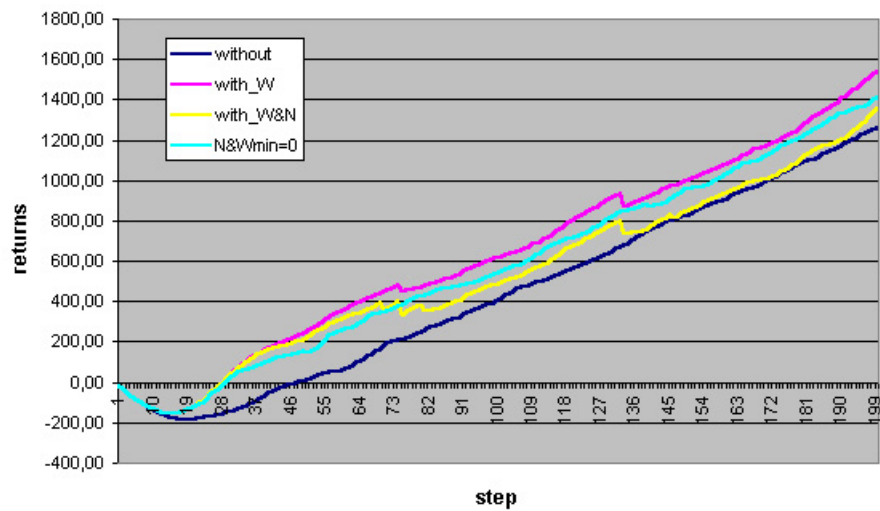


Figura 6.15: confronto impresa sotto dimensionata con *inventoryEvaluationCriterion* = 1

6.3 Esperimenti sul *Finacial Rate*: conclusioni

Al termine della serie di esperimenti eseguita è apparso determinante l'utilizzo dell'*inventoryFinancialRate*, in quanto da esso dipende l'andamento economico dei casi con magazzini e con magazzini ed informazioni. In particolare, si è potuto notare che molto spesso l'impresa non viene penalizzata dall'eccessivo ricorso all'utilizzo delle scorte; specie quando non ne ha un effettivo bisogno. Poiché l'unica variabile in grado di disincentivare questa prassi e favorire una razionalizzazione del processo è *inventoryFinancialRate* si è pensato di modificarla per trovare un limite di convenienza, per l'uno o per l'altro caso, all'impresa.

La struttura base utilizzata è sempre quella riferita al caso di impresa leggermente sotto dimensionata, in quanto ha dato prova di essere significativo per ottenere comportamenti caratteristici.

Inoltre si è scelto di fissare l'*inventoryEvaluationCriterion* al caso 2 al fine di parificare i tre esempi di impresa eliminando il vantaggio economico che può derivare dall'avere in magazzino scorte molto numerose.

La struttura completa proposta è mostrata in figura 6.16

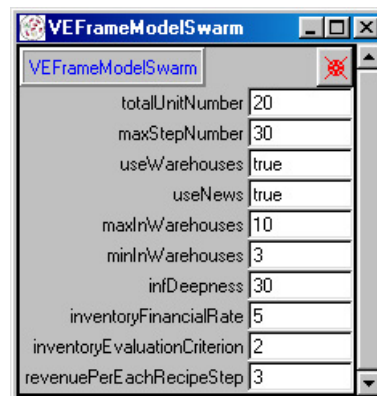


Figura 6.16: probe con *inventoryFinancialRate* pari a 5

Il caso che viene proposto mostra un valore del tasso finanziario pari a 5.

Prima di procedere occorre spendere alcune parole su cosa può indicare tale

valore per l'azienda. Ad oggi, il programma cumula giorno per giorno i costi finanziari che provengono da questo calcolo:

$$\frac{\text{quant. in magazzino} \cdot \text{inventoryFinancialRate}}{200}$$

ovvero il numero di scorte viene moltiplicato per un tasso finanziario ed il tutto diviso per 200. La scelta di 200 deriva dal semplice ragionamento di dividere il tasso per il numero di giorni lavorativi all'anno. Imporre 5 come *inventoryFinancialRate* dal *probe* del model implica quindi attribuire un costo finanziario giornaliero alle scorte pari a:

$$\frac{\text{quant. in magazzino} \cdot 5}{200}$$

, ovvero:

$$\text{quant. in magazzino} \cdot 2.5\%$$

.

Il valore di questa variabile non è per il momento significativo in quanto incide su valori di costo e di ricavo ancora totalmente fittizi, quindi verrà considerato solo per la sua influenza sul modello e non per il suo effettivo valore finanziario.

Sulla base di queste premesse si sono ottenuti i risultati rappresentati in figura 6.17.

Da essa si può notare come il caso di impresa con i soli magazzini, finora in termini di reddito sempre migliore, rispetto agli altri, sia preferibile solo nella prima parte della simulazione. Infatti, non appena i costi finanziari cumulati superano il rendimento che può derivare dall'immagazzinare scorte (intorno al 70° passo) la curva dei profitti comincia a crescere meno rapidamente delle altre, fino a quando non viene superata.

Molto più costante è l'andamento della curva d'impresa senza alcuna stra-

tegia, la quale, dopo la prima fase di perdita dovuta all'accumulo di costi fissi, incomincia ad assumere il suo tipico *trend* monotono.

Interessanti osservazioni emergono anche dall'osservazione del caso con la gestione delle informazioni. Con esso, proprio come si voleva dimostrare, si verifica che, nonostante la curva abbia lo stesso andamento del caso con i soli magazzini, dopo il 100° passo mostra un profitto maggiore. Questo accade poiché una più efficace gestione delle scorte consente di ridurre il costo finanziario delle stesse e quindi di aumentare il rendimento globale dell'impresa.

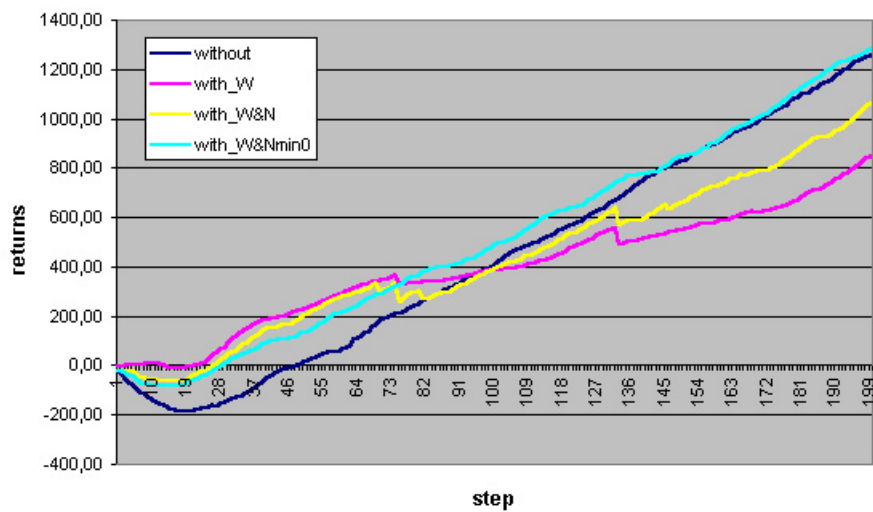


Figura 6.17: confronto con *inventoryFinancialRate* pari a 5

Inoltre, in questa situazione, fissare un limite inferiore alla quantità di beni immagazzinati risulta più un ostacolo che un vantaggio per l'azienda. Ne è la prova la linea azzurra del grafico in figura 6.17 che mostra il caso in cui l'utilizzo delle *news* dipende esclusivamente dalle informazioni che ottengono e non da altre impostazioni. In particolare, la variabile *minInWarehouse* viene fissata a zero invece che a tre.

I risultati sono immediatamente osservabili, si ottiene una curva che offre un rendimento superiore a tutte le altre grazie alla costante razionalizzazione del lavoro.

Appendice A

Progetto *Ptolemy*

Il progetto Ptolemy viene sviluppato dal *Department of Electrical engineering and Computer Science, University of California Berkeley*; l'obiettivo consiste nel fornire strumenti per modellare e progettare. Poiché sono presenti anche metodologie di gestione del tempo e di sequenze di eventi, le analogie con il progetto *Swarm* sono molte e consentono un approfondimento. Tutte le informazioni relative al progetto sono reperibili presso: <http://ptolemy.eecs.berkeley.edu>.

A.1 Modellare e progettare

Una parte del progetto " *Tolomeo* " è relativa agli studi specifici del dipartimento che lo sviluppa e, pertanto, distante dagli obiettivi del modello di impresa virtuale. Infatti, si occupa di elettronica analogica e digitale, di hardware e software e di strumenti elettronici e meccanici.

Un maggiore interesse matura in relazione alla parte che si occupa di *sistemi complessi*; intesi come, precisa Lee (1999):

(...) in the sense that they mix widely different operations, such as signal processing, feedback control, sequential decision making, and user interfaces.

In particolare i concetti relativi all'attività di modellare si accostano molto ai principi su cui si basa il progetto *Swarm* e, conseguentemente, l'impresa virtuale. Infatti, Lee (1999) indica due possibili strade per modellare: la prima matematica e la seconda computazionale.

Se si percorre la strada matematica è possibile derivare le proprietà del sistema modellato, quali le funzionalità e le dimensioni fisiche.

Mentre se si percorre la strada computazionale, costruttiva, è possibile descrivere il *comportamento* del sistema in relazione agli stimoli dell'ambiente esterno ad esso. *Progettare* è l'atto di definire uno o più modelli interattivi di un sistema fino a che non si ottengono le funzionalità desiderate.

Il progetto intende fornire gli strumenti necessari, in Java, per poter progettare e, quindi, *simulare* azioni e reazioni di un modello. In particolare sono di interesse come confronto con il progetto di impresa virtuale i modelli sincronizzati/reattivi, modelli di eventi discreti e i modelli *Timed CSP/PN*. L'interesse deriva dalle scelte operate in merito alla gestione del tempo.

A.1.1 Modelli *sincronizzati/reattivi*

Nei modelli sincronizzati/reattivi (SR) i valori sono allineati con gli scatti di un orologio globale, per ogni scatto ci sono valori discreti che procedono nel modello come input o come output. Procedimento analogo a quello studiato per lo *schedule* di *Swarm*. Questo in linea puramente teorica; infatti, nella descrizione del progetto, Lee (1999) sottolinea le funzionalità di utilizzare un metodo di modellizzazione piuttosto che un altro per un preciso obiettivo di carattere elettronico. In questo caso si consiglia di utilizzare i modelli (SR) per le simulazioni in tempo reale di strumenti che necessitino di sincronizzazione per maturare un confronto, in altre parole i risultati della simulazione emergono dal confronto di eventi simultanei.

A.1.2 Modelli di eventi discreti

Diversamente dai precedenti per i modelli di eventi discreti non si parla più di una sequenza di eventi, ma di eventi che nascono e terminano in un periodo. Non è necessario un orologio globale, in quanto non vi è più la necessità di sincronizzare gli eventi, ma si ha solo una nozione complessiva del tempo. Ovvero si considera il tempo come un periodo entro il quale può maturare un evento.

A.1.3 Modelli *Timed CSP/PN*

Innanzitutto modelli *Timed CSP/PN* sono basati su: *communicating sequential processes* (CSP) e *process network* (PN). Nel primo caso i processi comunicano tra loro grazie al coordinamento delle simultanee possibilità di dialogo; ad esempio se due processi devono comunicare ed uno raggiunge l'obiettivo prima dell'altro questo deve fermarsi fino a quando l'altro non è pronto. Nel secondo caso non c'è simultaneità di operato, ma sequenzialità; come in una "rete" un processo si attiva solo se il precedente ha terminato il proprio compito. Questi modelli si basano sull'utilizzo di *threads*¹ per gestire sia la simultaneità che la sequenzialità. In questo caso si perde totalmente la concezione del tempo in quanto non più determinante per decidere la successione degli eventi. L'attenzione è incentrata esclusivamente sul coordinamento dei processi indipendentemente dal tempo che essi impiegano per essere compiuti.

A.2 Conclusioni

Dall'analisi del progetto *Ptolemy* emerge da un lato un notevole parallelismo di intenti con *Swarm*, sia per quanto riguarda gli strumenti sia per quanto concerne

¹I *threads* sono una parte del programma predisposta per essere eseguita autonomamente mentre il resto del programma esegue qualcos'altro. Questo sistema è detto *multitasking* poiché il programma tratta più *task* contemporaneamente.

la formalizzazione dei modelli. Questo si può notare ad esempio nella gestione del tempo prevista dai modelli SR, si veda A.1.1.

Dall'altro quando si analizzano gli oggetti messi a disposizione dal progetto emerge come essi siano fortemente improntati su stampo ingegneristico e, pertanto, difficilmente adattabili ad un modello economico.

Ciò non toglie che alcuni oggetti siano strutturati in modo da essere utilizzabili per perseguire obiettivi indipendenti dall'ambito in cui essi siano applicati. Questo è il caso del *package plot* che serve per disegnare grafici bi-dimensionali, si veda il paragrafo 5.3.2.

Bibliografia

- [1] Lee A. E. (1999), *Overview of the Ptolemy Project*,
<http://ptolemy.eecs.berkeley.edu/publications/papers/98/Overview/overview.pdf>

Appendice B

UML: Unified Modeling Language

La programmazione ad oggetti richiede al programmatore, come spiegato nel capitolo 2, una forte capacità di astrarre i concetti da rappresentare nel modello elaborato. Non sempre, peraltro, un concetto viene modellizzato nel medesimo modo da persone differenti; infatti posto un problema questo può essere analizzato e simulato in modi completamente diversi, ma tutti concettualmente corretti. Questo crea dei problemi in sede di elaborazione di progetti complicati in cui interagiscono più soggetti singolarmente impegnati a sviluppare una parte di codice. Diventa necessario coordinare il lavoro per fare in modo che le parti separatamente sviluppate possano essere, in un secondo tempo, facilmente integrabili tra di loro.

Molte imprese sono oggi interessate ad informatizzare la propria struttura sia per quanto riguarda il sistema informativo sia per tutto ciò che concerne la parte produttiva e, per fare questo passo, commissionano grossi progetti ad imprese specializzate nella produzione di software. Sono quest'ultime ad essere maggiormente colpite dalla necessità di trovare un metodo di gestione coordinata di grossi progetti, al fine di poter suddividere il lavoro in più parti favorendone

uno sviluppo in parallelo.

L'UML vuole essere una risposta a queste necessità ponendosi come linguaggio comune che facilita il dialogo e le interazioni tra le parti coinvolte nel progetto.

B.1 Caratteristiche base dell'UML

L'Unified Modeling Language (UML) è un linguaggio formale per visualizzare, definire, realizzare e documentare un sistema software. Deriva dall'unione di tre suoi predecessori orientati alla rappresentazione della programmazione ad oggetti, Booch, OMT e OOSE, dai quali ha ereditato le principali caratteristiche.

L'obiettivo principale dell'UML è facilitare il lavoro delle imprese nello sviluppo di progetti fornendo strumenti *visivi* per l'inter-operabilità nel modellare. Le caratteristiche base per perseguire questo scopo sono:

- la definizione formale di strumenti comuni per l'analisi e la rappresentazione di un modello ad oggetti, sia esso un modello statico o comportamentale
- la definizione dei legami per rappresentare le relazioni che intercorrono tra gli oggetti
- la definizione di una sintassi grafica facilmente leggibile, ma che consenta di sintetizzare tutto ciò di cui un programmatore necessita per lavorare

Questi passaggi vengono rappresentati visivamente dall'UML tramite diverse tipologie di diagrammi volte ad illustrare il modello in tutte le sue specifiche. I diagrammi che gli sviluppatori del linguaggio hanno ritenuto necessari per un'adeguata comprensione del modello sono i seguenti:

- diagrammi di casi d'uso
- diagrammi di interazione: di sequenza e di collaborazione
- diagrammi di classe

- diagrammi di stato e attività

B.1.1 Diagrammi di Casi d'Uso

I diagrammi di casi d'uso sono composti dagli utenti del sistema, dai casi d'uso, ovvero dalle parti in cui il problema è stato scomposto, e dalle relazioni che vi intercorrono. Attori, casi e relazioni vengono tutti racchiusi in un diagramma come quello mostrato nell'esempio di figura B.1.

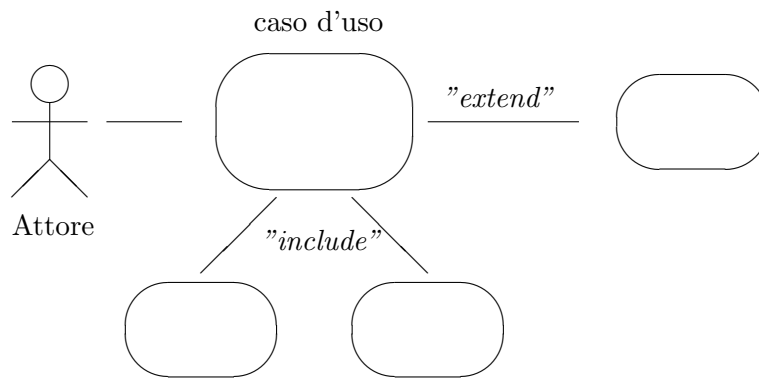


Figura B.1: esempio di diagramma di caso d'uso

Questo diagramma è il primo passo per scrivere in UML poichè permette una prima sintesi generale del percorso da compiere per giungere alla stesura del modello.

Gli utenti, che in UML vengono chiamati *attori* e simboleggiano il confine tra il modello e chi interagisce con esso, sono collegati con uno o più casi d'uso. Questa relazione indica attraverso quale caso l'utente deve passare per accedere al modello. In sostanza dall'esterno non è possibile, poiché scarsamente pratico, dialogare con tutti i casi d'uso, ma è sufficiente dialogare con uno di essi per disporre di tutto il sistema. Ogni caso dispone di proprie relazioni con gli altri ai quali passa o dai quali riceve informazioni. Per ognuno di essi è prevista una documentazione della funzione che esso svolge all'interno del sistema e delle rela-

zioni di cui dispone, al fine di fare chiarezza qualora un progetto molto complicato portasse ad una rete di funzioni troppo fitta per risultare comprensibile.

In accordo con la terminologia della programmazione ad oggetti le relazioni che si possono instaurare in questo diagramma sono di due tipi: la relazione *include* che mostra un comportamento comune a più oggetti e la relazione *extends* che mostra l'estensione di un comportamento tra gli oggetti.

Questo tipo di diagramma ripercorre, in sostanza, il percorso di astrazione richiesto per programmare ad oggetti e si può dire che offre una visione globale del problema indicando a chi modella come deve suddividerlo e quali siano le categorie di oggetti da costruire. Altri diagrammi specificheranno meglio come devono essere costruiti gli oggetti e le relazioni tra di essi.

B.1.2 Diagrammi di Interazione

Rispetto ai diagrammi di casi d'uso entrano maggiormente nel merito, come si evince dal nome stesso, delle interazioni che passano tra utente ed oggetti e tra gli oggetti stessi. Vengono suddivisi in due tipi di diagrammi differenti che privilegiano o il tempo in cui si realizzano queste interazioni, diagrammi di sequenza, o la posizione che assumono le interazioni tra gli oggetti, diagrammi di collaborazione. Entrambi hanno un'importanza specifica in base all'ambito cui viene applicato il diagramma.

Diagrammi di Sequenza

Un diagramma di sequenza ha due dimensioni: una verticale che rappresenta il *tempo* ed una orizzontale che rappresenta gli *oggetti*. L'importanza di questo diagramma, si veda l'esempio in figura B.2, risiede soprattutto nella distribuzione del tempo più che nella posizione degli oggetti, meglio elaborata dal diagramma di collaborazione. Distribuzione del tempo che può sia essere effettuata in misura

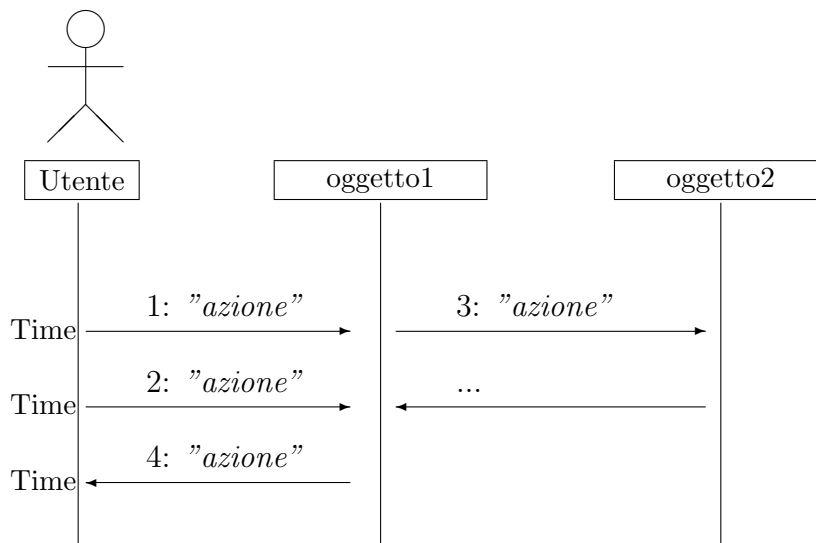


Figura B.2: esempio di diagramma di sequenza

di tempo reale se l'applicazione lo richiede sia essere solo una rappresentazione fittizia del susseguirsi degli eventi.

Come si nota in figura B.2, le azioni che il modello esegue vengono rappresentate nel diagramma come una *sequenza* di eventi. Per ogni evento viene preso in considerazione il tempo in cui un oggetto lancia un messaggio, *sendTime*, o riceve un messaggio, *receiveTime*, e, dai vettori che intercorrono tra i casi d'uso, si segnala anche quale oggetto ha lanciato o ricevuto un messaggio. Possono essere prese in considerazione anche altre funzioni relative al tempo quali: *elapsedTime*, *executionStartTime*, *queuedTime* o *handledTime*.

Diagrammi di Collaborazione

Un diagramma di collaborazione mostra l'organizzazione del sistema sotto l'aspetto delle interazione e dei legami tra le parti. Diversamente dal diagramma di sequenza mostra le relazioni che intercorrono tra gli oggetti, ma non il tempo in cui queste relazioni vengono eseguite; di conseguenza queste vanno specificate

con una numerazione sequenziale per tenere conto dell'ordine in cui esse vengono eseguite. Questo diagramma è funzionale per la comprensione della posizione che gli oggetti assumono nel progetto, ma se il modello opera in tempo reale la rappresentazione migliore rimane quella del diagramma di sequenza.

Un esempio di questo tipo di diagramma è riportato in figura B.3.

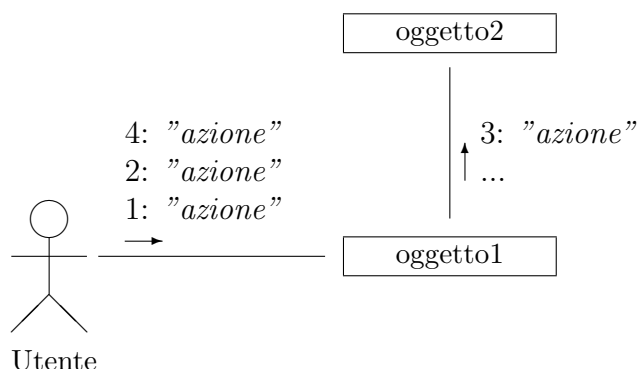


Figura B.3: esempio di diagramma di collaborazione

B.1.3 Diagrammi di Classe

Definiti gli oggetti del modello e la loro posizione il linguaggio prevede la descrizione delle classi da cui gli oggetti vengono creati. Chi scrive in UML deve pertanto avere già ben chiare le proprietà di ogni singolo oggetto e le funzioni che dovrà svolgere e sintetizzarle in uno schema come quello proposto in figura B.4. La classe viene rappresentata con un rettangolo a tre scompartimenti: nel primo vi è il nome della classe, da scegliere coerentemente con il dominio che essa ricopre, nel secondo le proprietà o attributi della classe e nel terzo i metodi.

Le classi vengono identificate partendo dai diagrammi di sequenza e collaborazione ovvero dagli oggetti che sono stati pensati per il funzionamento del modello. Ricordando, però, che più oggetti possono derivare dalla stessa classe occorre valutare quali svolgono funzioni similari all'interno del progetto per rac-

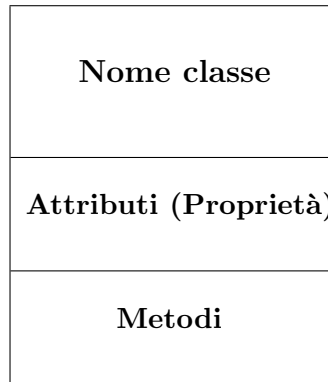


Figura B.4: esempio di diagramma di classe

coglierli in una classe sola. A questo punto si definiscono gli attributi e i metodi che contraddistinguono l'oggetto, si veda figura B.5.

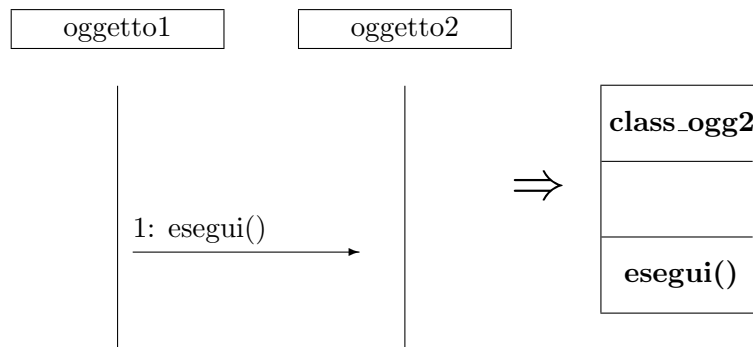


Figura B.5: esempio di definizione di una classe

I diagrammi di classe sono utili al fine di specificare le *relazioni* che intercorrono tra gli oggetti ed indicano un sentiero di navigazione tra gli oggetti. Tali relazioni possono essere dedotte dai diagrammi di interazione seguendo il percorso logico che sta alla base del modello. Nel diagramma di classe si pone maggiore attenzione al tipo di relazione che intercorre e si cerca di specificare come essa incide sul comportamento degli oggetti. Ad esempio si può rilevare nel diagramma di collaborazione che due oggetti sono posti in posizione gerarchica, in questo caso nello schema delle classi bisogna evidenziare con una precisa simbologia che vi è

un rapporto di ereditarietà tra le classi e che risulta inutile specificare nuovamente alcuni metodi o proprietà.

Vi sono poi relazioni di associazione che sottolineano quando due classi sono connesse da un rapporto biunivoco che lega le caratteristiche di una a quelle dell'altra o relazioni di aggregazione che mostrano un legame di contenuto/re/contenuto tra le classi.

L'UML precisa anche per la definizione delle relazioni una simbologia ben precisa che sinteticamente descrive concetti anche dettagliati per la costruzione del modello. Si cerca di non lasciare nulla alla libera interpretazione per essere sicuri che quando le relazioni saranno integrate non vi saranno comportamenti imprevisti degli oggetti, ciclici o ricorsivi.

B.1.4 Diagrammi di Stato e di Attività

Il diagramma di Stato viene utilizzato per descrivere il comportamento degli oggetti e delle interazioni del modello come conseguenza degli stimoli ai quali sono sopposti. Precisamente consiste in una descrizione globale che mostra tutto il modello specificando le azioni che svolge tramite metodi e funzioni. In un diagramma di questo tipo devono essere segnalati i punti di accesso al modello e i punti di uscita, oltre a tutti i percorsi percorribili dai dati durante la loro elaborazione, si veda figura B.6.

In esso vengono ripresi i diagrammi dei casi d'uso dettagliando maggiormente sia le relazioni che intercorrono, suggerite dai diagrammi di sequenza e collaborazione, sia le classi utilizzate. Come si può notare dalla figura B.6, più si scende nel dettaglio più il linguaggio fornisce strumenti per rendere chiara e completa la descrizione. A questo livello si possono, pertanto, mostrare cicli o condizioni che determinano la logica del percorso delle informazioni all'interno del sistema; sono sottolineate le scelte che l'oggetto opera sui dati che gli vengono passati, se elaborarle lui stesso ed in quale modo farlo o se passarle ad un altro oggetto.

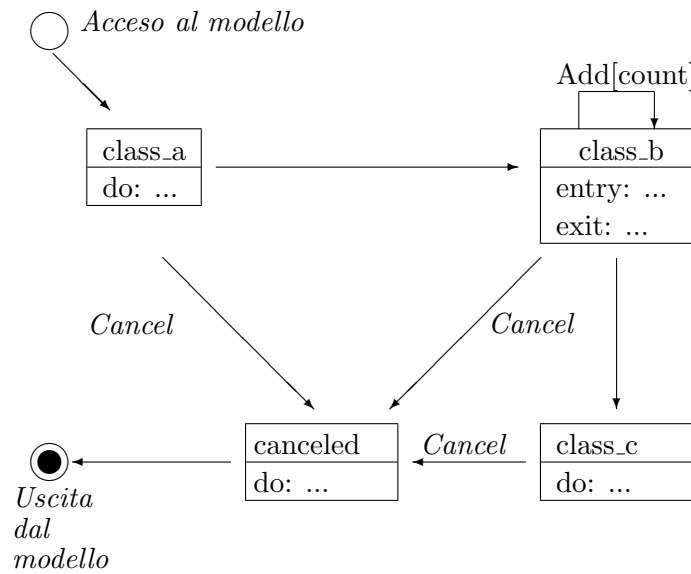


Figura B.6: esempio di diagramma di Stato

Compito del programmatore sarà osservare questo grafico e tradurlo con funzioni, quali cicli *for* o condizioni *if*, del linguaggio di programmazione ad oggetti utilizzato affinché il modello possa funzionare in pratica, compito molto semplificato poiché depurato del passaggio dell'astrazione.

Il diagramma delle Attività è una variazione di quello di Stato. La sua funzione è quella di mostrare l'azione che l'oggetto deve svolgere, senza scendere nel merito dei metodi che utilizza e degli attributi che lo contraddistinguono. L'obiettivo è di porre l'attenzione sullo scorrere delle informazioni nel modello in contrapposizione a come esse si muovono fuori dal modello. Questo diagramma è utile per mostrare come si ha intenzione di porre sotto controllo il processo di elaborazione dati del programma, si veda figura B.7.

Per certi aspetti somiglia molto al diagramma di stato, infatti i due possono considerarsi complementari, la scelta dell'uno piuttosto che dell'altro dipende da ciò che si intende mostrare ovvero i flussi delle azioni o le influenze che esse hanno sugli stati.

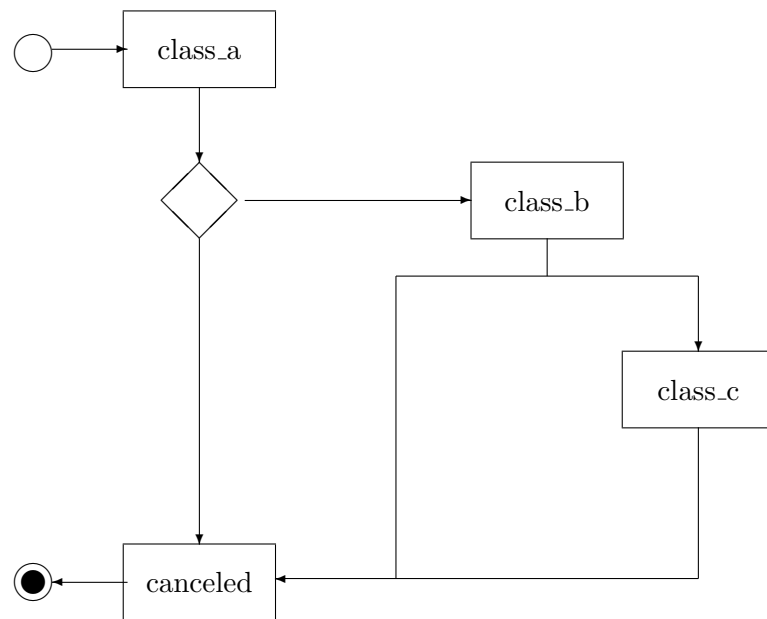


Figura B.7: esempio di diagramma di Attività

B.2 Scrivere in UML

Gli analisti di società produttrici di software dovranno essere in grado di utilizzare l'UML in modo da rappresentare fedelmente il modello da realizzare e da rendere comprensibile ciò che intendono realizzare a chi programma. Per fare questo oltre a conoscere gli strumenti che il linguaggio offre, i diagrammi descritti, dovranno essere capaci di comprendere cosa il committente intende realizzare, quale obiettivo intende raggiungere e con quali mezzi, e tradurlo in termini di modello.

Innanzitutto è opportuno operare uno studio del *dominio* ovvero osservare lo scenario in cui il sistema dovrà operare; questo per fare in modo che non vi siano problemi nelle interazioni tra utenti e sistema. Spesso risulta utile produrre un dizionario della nomenclatura utilizzata nel progetto per evitare incomprensioni sia da parte degli analisti in fase di stesura del modello sia da parte degli utenti in fase di utilizzazione.

In seguito occorre definire le *responsabilità* del sistema per capire fino a dove

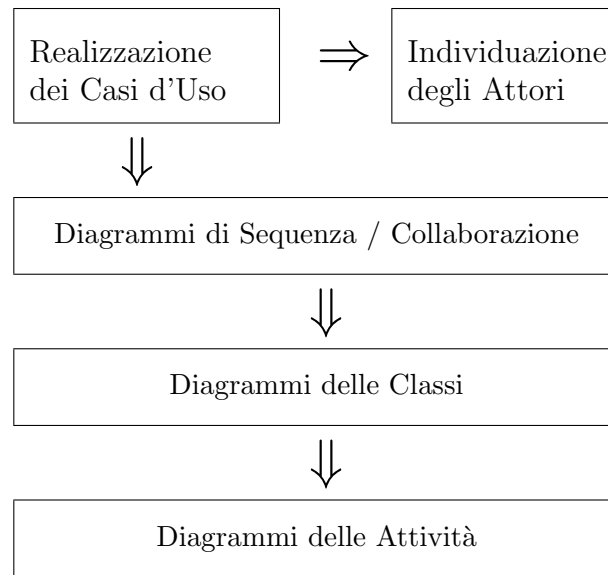


Figura B.8: percorso di scrittura in UML

può operare senza l'intervento di utenti esterni e dove, invece, sono richiesti interventi correttivi o interattivi. In sostanza occorre valutare quali situazioni possono essere valutate e risolte direttamente dal sistema e quali richiedono l'intervento di utenti esterni.

A questo punto, compresa la situazione generale, si potrà scrivere tramite i diagrammi il percorso da seguire per la realizzazione pratica del programma, si veda figura B.8.

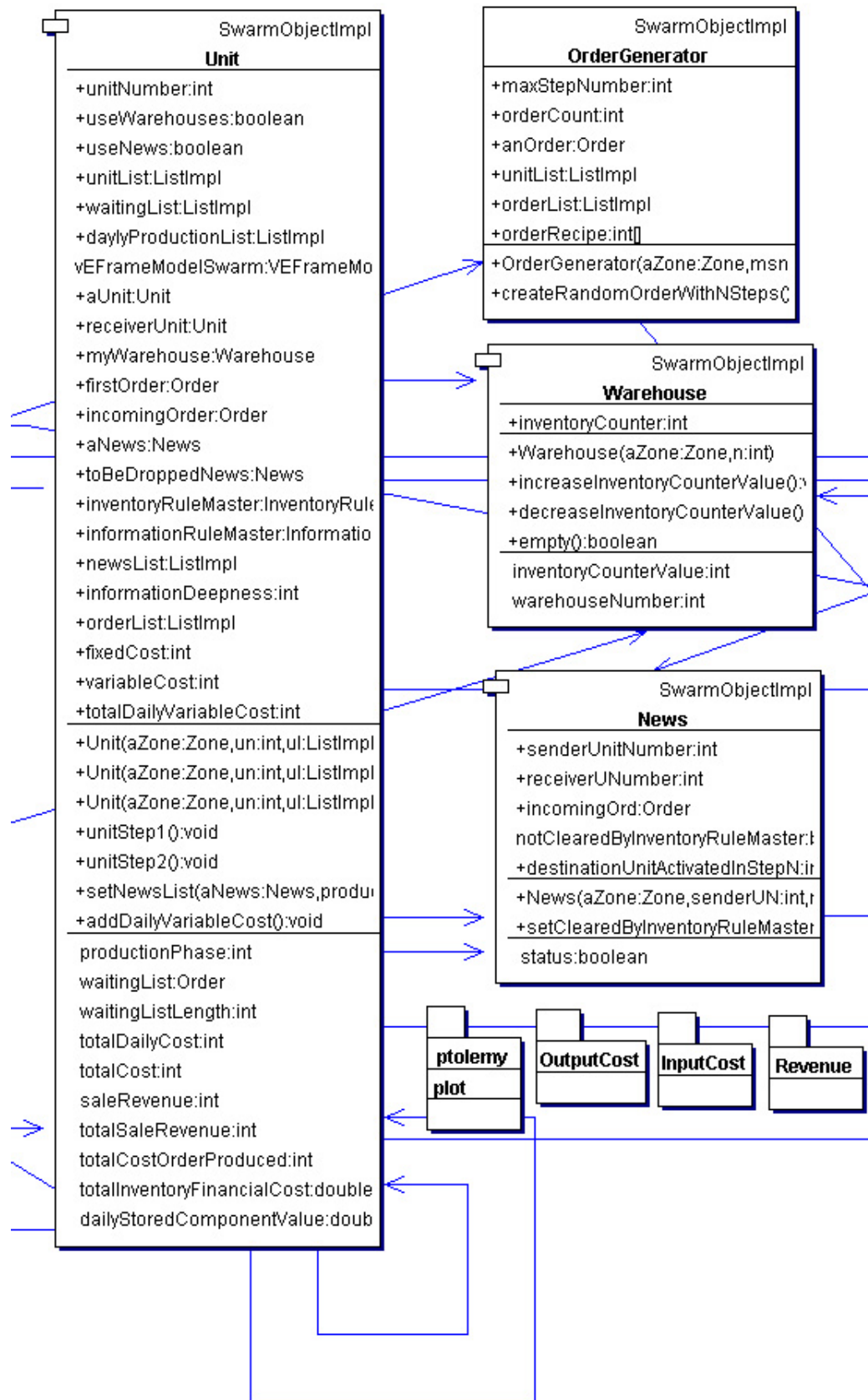
Il percorso portato come esempio in figura non è rigidamente definito, ed è pertanto passibile di modifiche, ma risulta chiaro poiché parte da una descrizione generale del modello per poi scendere sempre più nel dettaglio delle caratteristiche richieste dalla programmazione ad oggetti.

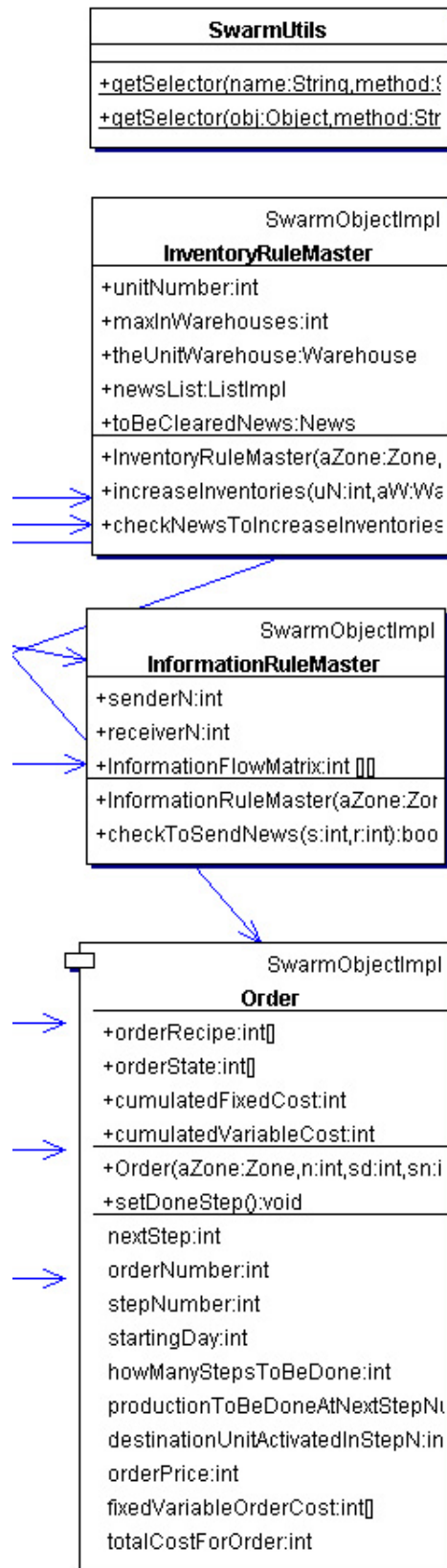
B.3 L'impresa virtuale in UML

Grazie all'impiego dell'ambiente *Together*, si veda D.1.3, prodotto dalla *TogetherSoft* è stato possibile operare il *Reverse Engineering* del codice Java relativo

all'impresa virtuale. Questo software riproduce il progetto di *Java Virtual Enterprise* attraverso i diagrammi di stato, che ripercorrono sostanzialmente lo scorrere dei flussi di informazioni all'interno del modello.







Bibliografia

- [1] OMG (1999), *Unified Modeling Language Specification*, version 1.3,
<http://www.rational.com/media/uml/post.pdf>

Appendice C

Strumenti di simulazione

La necessità crescente da parti di imprese e istituti di ricerca di simulare eventi, prima di affrontarne le difficoltà direttamente nella realtà, è sottolineata dallo sviluppo di strumenti specifici per queste esigenze.

L'impresa virtuale è stata sviluppata attraverso l'utilizzo *Swarm* scritto in *Java*; una scelta maturata dal confronto delle potenzialità offerte da questo strumento con le caratteristiche proprie di altri ambienti di sviluppo; quali *Simul8*, *Extend*. In altre situazioni la scelta è caduta su altri strumenti, poiché gli obiettivi della simulazione erano differenti; ad esempio per la simulazione del flusso di pellegrini verso la città di Roma in occasione del *Giubileo* dell'anno 2000 si è fatto uso di *Extend*.

Attraverso una breve analisi di alcuni di questi ambienti si cercherà di giustificare la scelta di *Swarm* per la simulazione dell'impresa virtuale.

C.1 Descrizione di alcuni ambienti

C.1.1 Simul8

Simul8 è un ambiente di lavoro che appare molto logico ed intuitivo sia per chi programma una simulazione sia per chi ne legge i risultati. Infatti, il primo

aspetto che colpisce chi lo osserva in funzione è la dinamicità dei modelli creati e la fedeltà con cui essi possono rappresentare un processo. Si veda ad esempio la figura C.1 in cui si rappresenta il funzionamento di un'impresa che si occupa di imbottigliare.

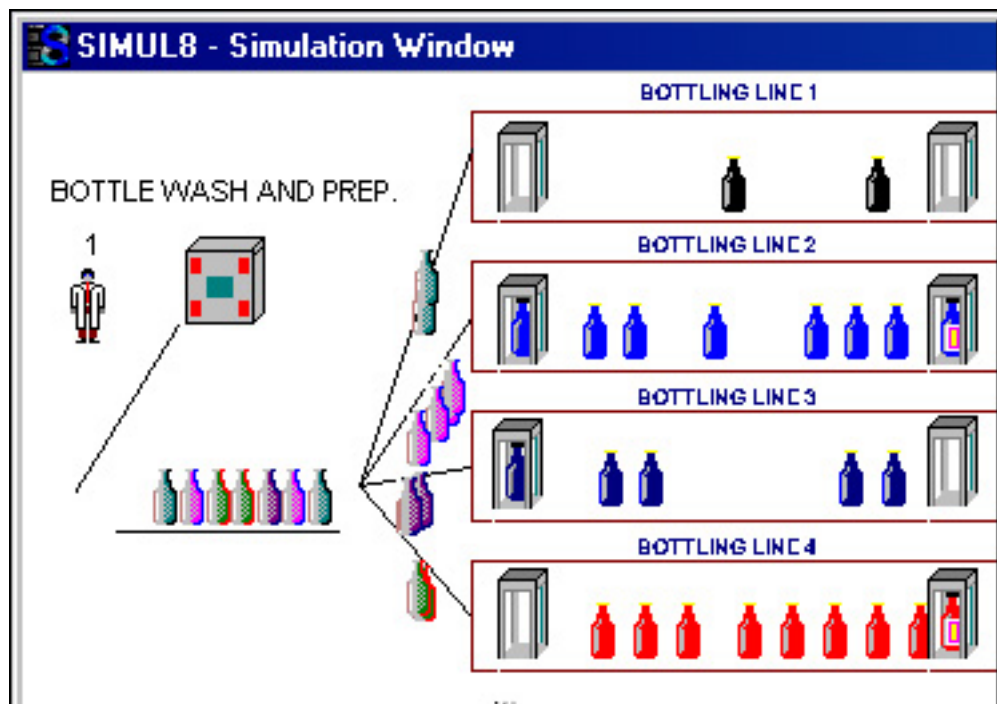


Figura C.1: esempio di simulazione in Simul8

L'aspetto grafico incide molto a favore di questo strumento in quanto è immediatamente comprensibile cosa sta accadendo, anche per chi non conosce le caratteristiche del problema. Si possono subito notare: la fase di pulizia delle bottiglie, la fase di distribuzione alle quattro linee di produzione ed infine la fase di imbottigliamento.

Il percorso viene predefinito tramite l'impiego di *link* unidirezionali o pluridirezionali tra:

- *work entry point*, il punto da cui parte la simulazione
- *storage*, luogo in cui si raccolgono i dati di partenza, un magazzino di risorse

- *work center*, centro in cui si elaborano i dati
- *work exit point*, punto di uscita della simulazione

I dati necessari al funzionamento del modello, ad esempio le quantità di bottiglie che un *work center* può imbottigliare in una giornata, possono essere raccolti in un foglio di lavoro *Microsoft Excel*. Simul8 è, infatti, in grado di interagire con i fogli di lavoro *.xls* tramite l'impostazione corretta di alcune macro. Parimenti è possibile scrivere i risultati ottenuti in formato *.xls*, con il vantaggio della trasportabilità.

La gestione dei tempi relativi al passaggio dei dati all'interno del modello si basa su un orologio impostato sulle ore lavorative settimanali, come quello mostrato in figura C.2.

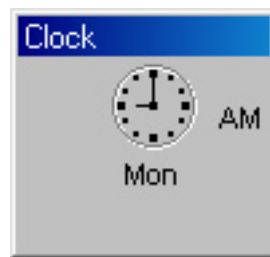


Figura C.2: esempio di orologio in Simul8

Il vantaggio di questo orologio consiste nell'essere impostato in modo da dare immediatamente idea dello scorrere del tempo in forma reale. Questo rende la simulazione facilmente comprensibile da chi la osserva ed i risultati immediatamente paragonabili con una situazione pratica.

Ulteriori informazioni possono essere raccolte all'indirizzo: www.simul8.com.

C.1.2 Extend

Per descrivere le funzionalità di *Extend* si farà uso: della presentazione di questo strumento da parte del prof. Di Leva (Dipartimento di Informatica, Università di

Torino) del 21/06/2000 presso la facoltà di Economia di Torino e della descrizione relativa alla Sala Simulazione per il Giubileo 2000 da parte del prof. Berchi nella medesima occasione.

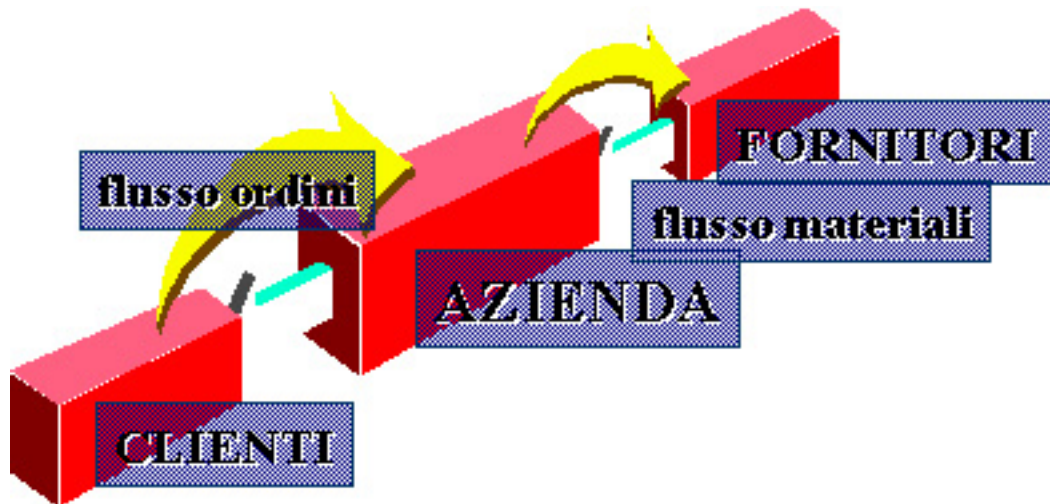


Figura C.3: esempio di processo per la simulazione di impresa

La presentazione del prof. Di Leva è stata incentrata sull'argomento *impresa* a dimostrare come *Extend* possa essere utilizzato in campo economico; in particolare si parla d'azienda, intesa come *sistema complesso* e *aleatorio* che può essere descritto solo con l'utilizzo di strumenti molto *dinamici*. Anche in questo ambito si parla di modelli, base di partenza per qualunque simulazione, poiché indispensabili per (tratto da):

- analizzare i processi in termini di prestazioni, flussi, obiettivi e vincoli, costi
- valutare l'impatto di eventi esterni sui processi
- individuare e verificare soluzioni migliorative
- controllare e gestire i processi, se integrati in una architettura operativa con strumenti di work flow, analisi, visualizzazione, consuntivazione e gestione dati, ecc...

Nel caso di una visione globale dell'impresa i modelli, analogamente a quanto sviluppato per il modello di impresa virtuale, devono consentire di simulare i flussi di *input/output* in azienda, come rappresentato in figura C.3

Questa situazione può essere riprodotta seguendo in una prima fase una metodologia definita, dal prof. A. Di Leva, come *M*Complex* ed, in una seconda fase formalizzata utilizzando strumenti *object-oriented*. *M*Complex* si occupa di rappresentare i sistemi complessi attraverso l'aspetto: organizzativo, comportamentale, informativo e delle risorse. Definiti questi passaggi nel modello occorre disporre di uno strumento, quale *Extend*, che consenta di rappresentarli con la dovuta pertinenza.

In particolare, un processo può essere descritto tramite l'utilizzo di:

- nodi di inizio o fine



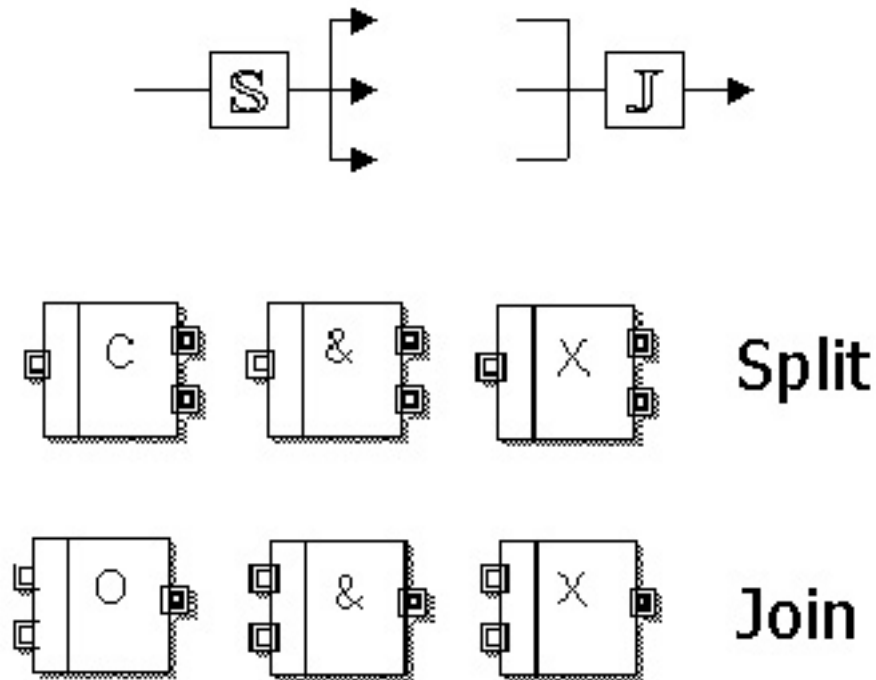
- attività



- archi di controllo



- connettori e giunzioni



I medesimi strumenti sono stati utilizzati per sviluppare una simulazione su argomenti concettualmente differenti, ma sempre riconducibili al filone della complessità. In particolare, si può analizzare il caso de "La Sala Situazione Del Giubileo" quale strumento di gestione della complessità di un sistema urbano.

Le funzioni della *Sala Situazione* erano relative a:

- la raccolta ed elaborazione di dati e informazioni relativi a tutti gli eventi previsti in calendario;
- la predisposizione di tutti gli scenari connessi agli avvenimenti ordinari e predisposizione dei piani per fronteggiare situazioni di emergenza;
- il coordinamento, durante i grandi eventi, delle iniziative e delle attività dei vari enti interessati, nel rispetto delle specifiche competenze e responsabilità istituzionali facenti capo agli enti stessi;

- il collegamento diretto con tutte le centrali operative "istituzionali" già esistenti

Questo ha comportato, innanzi tutto, la raccolta di tutte le informazioni relative alla manifestazione ed ai ruoli svolti dagli enti coinvolti. Poi è stato necessario valutare la funzionalità degli enti sulla base della planimetria della città, ottimizzando i vari accessi alle strutture e predisponendo servizi nei luoghi ritenuti critici. Ad esempio, è stato opportuno valutare la facilità di accesso a strutture ospedaliere ed alla quantità di persone che esse sono in grado di accogliere. Tutte le variabili che sono proprie di un sistema cittadino sono state prese in considerazione per la simulazione, con le mancanze e le eccezioni che possono verificarsi nel trascorrere dell'anno di riferimento.

Infine, tutti questi dati, previsionali e simulati, relativi ai tempi che la città di Roma avrebbe reso possibili nell'anno 2000 sono stati applicati al percorso previsto del visitatore. Così è stato possibile valutare in anticipo le code ed i problemi che ogni singolo giorno si sarebbero formate, anticipando le misure di sicurezza.

La simulazione è stata svolta con *Extend*, di cui si riporta un esempio in figura C.4 a pagina 222 relativa allo schema del percorso che i turisti avrebbero seguito. Ogni giunzione o connettore del grafico cela l'applicazione di tutte le ipotesi maturate dallo studio dei dati previsionali e le conseguenti variazioni o incidenze sul percorso dei visitatori.

C.2 Conclusioni

L'impressione, derivata dall'osservazione di modelli, relativa all'impiego degli strumenti sopra descritti è: che essi possano essere utilizzati per simulare situazioni reali, magari molto complicate, in cui si desideri, date una serie di informazioni di partenza, valutare cosa può accadere dopo un certo periodo di tempo. Sicuramente importante è la capacità previsionale di un modello così strutturato, infatti è possibile valutare a priori come impostare un problema e cosa richiedere alle singole parti. Nel caso, ad esempio, della simulazione sul *Giubileo* è stato possibile valutare con anticipo il flusso di persone che la città di Roma avrebbe dovuto sopportare in quel periodo e, conseguentemente, preparare le strutture della città.

L'intento dell'impresa virtuale si basa però su un ulteriore problema: la gestione dell'emergenza imprevista ed imprevedibile che emerge dall'analisi di un fenomeno complesso, si veda 1.2.2.

L'obiettivo non è solamente di carattere previsionale, ma di ricerca di quei fenomeni imprevisti ed imprevedibili che possono emergere con la complessità. Pertanto, nasce l'esigenza di formulare un modello strettamente personalizzato in grado di centrare di volta in volta il problema sulla base dei risultati raggiunti.

Swarm fornisce gli strumenti per un'analisi di questo tipo, ma lascia libera l'impostazione del lavoro, l'analisi del problema e dei risultati. Il modello può essere così strutturato in qualunque campo di studio con le caratteristiche specifiche richieste da quel preciso fenomeno di ricerca.

Nel caso dell'impresa virtuale, quale modello economico del fenomeno complesso di gestione d'impresa, si è ritenuto più idoneo l'utilizzo di *Swarm* proprio per gli intenti ricercati. In particolare, la mancanza di un caso d'analisi specifico, in quanto si è cercato di generalizzare il modello alla globalità delle imprese, ha richiesto l'utilizzo di uno strumento molto flessibile che consentisse di essere impostato, in un secondo tempo, in funzione del caso di utilizzo.

Ad esempio, la possibilità di impostare di volta in volta il modello tramite le sonde con più o meno unità produttive, con o senza l'utilizzo dei magazzini, con o senza l'utilizzo del flusso informativo ha consentito:

- in termini pratici di poter simulare qualsivoglia tipo di impresa.
- in termini teorici di operare un immediato confronto sui risultati raggiunti.

Modellare con *Swarm* significa anche tener presente queste possibilità offerte dallo strumento e farne uso al fine di risolvere la complessità del problema.

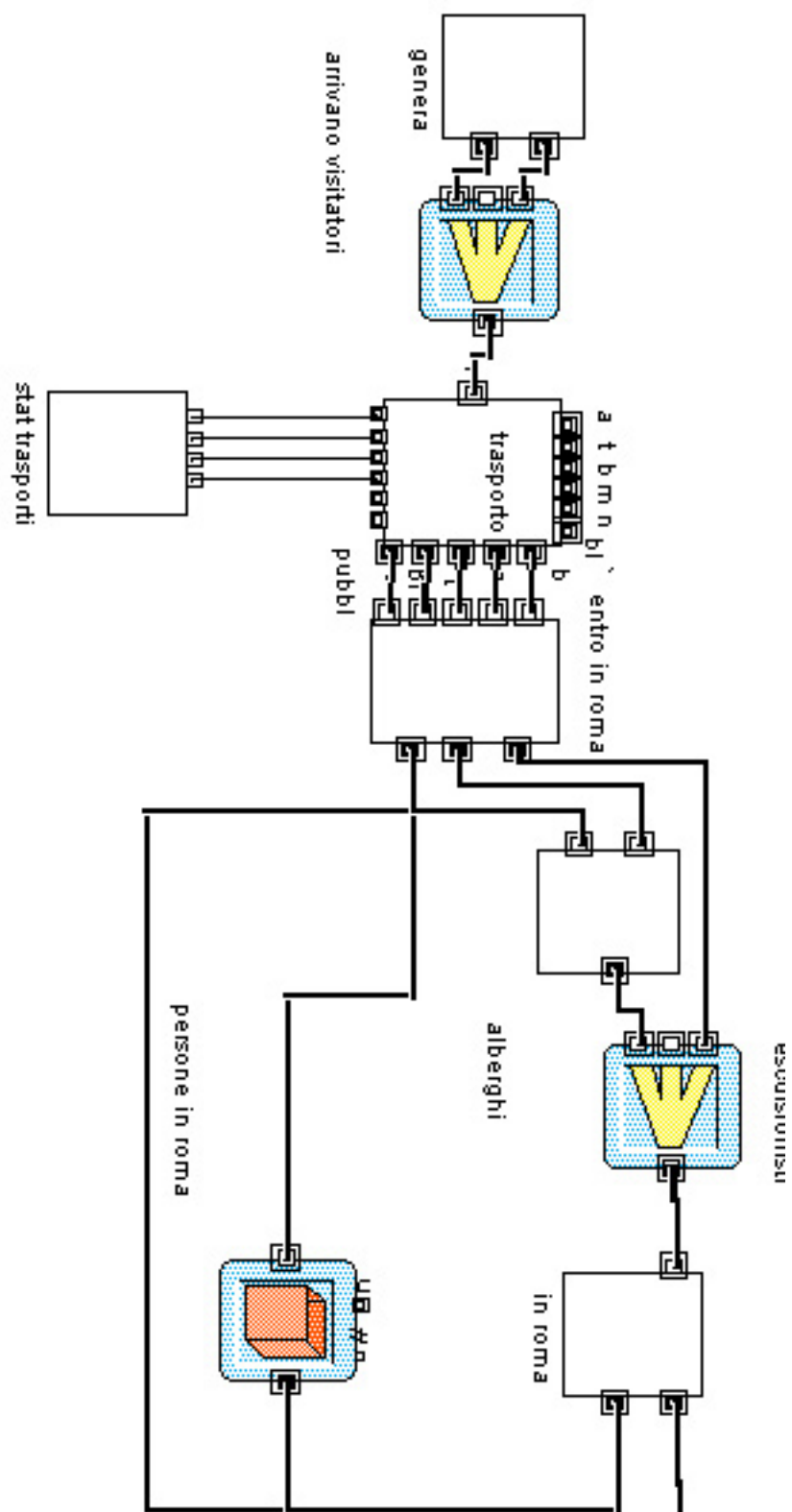


Figura C.4: esempio di *Extend* per la Sala Simulazione del Giubileo

Bibliografia

- [1] *software dimostrativo*, www.simul8.com
- [2] Berchi (2000), *Sala Situazione per il Giubileo*, presentazione presso la facoltà di Economia, Università di Torino.
- [3] Di Leva A. (2000), *Analisi di Processi Aziendali*, presentazione presso la facoltà di Economia, Università di Torino.

Appendice D

Listato

D.1 Ambienti di sviluppo ed Editor

Per scrivere il codice in Java sono disponibili diversi *Editor*, sviluppati da differenti case produttrici, con caratteristiche più o meno specifiche a seconda delle finalità per cui il Java viene impiegato.

Alcuni di essi sono stati utilizzati per l'elaborazione del modello di impresa virtuale ed è interessante analizzare le principali funzioni di ognuno di essi.

D.1.1 Emacs

Emacs è, come lo definisce il suo sviluppatore R. Stallman:

(...) the extensible, customizable, self-documenting, real time display editor.

Si tratta, infatti, di un editor polivalente in grado di comprendere moltissimi linguaggi di programmazione e di fornire per ognuno di essi le necessarie funzioni. Tutte le sue caratteristiche sono descritte all'indirizzo:

www.gnu.org/software/emacs/emacs.html.

Emacs nasce dal progetto *GNU* (Gnu is Not Unix) che viene sviluppato dalla *Free Software Foundation*. Le nozioni relative al progetto sono reperibili al sito: www.gnu.org. Questa fondazione si fa promotrice del principio dell'*Open Source*,

ovvero dall'idea di concedere all'utente del software la possibilità di modificare e, quindi, migliorare lo strumento che sta utilizzando. Proprio da questo principio sorgono le grandi potenzialità di *Emacs*, il quale dopo molteplici variazioni e perfezionamenti ha ora raggiunto un notevole stadio di sviluppo ed una vasta gamma di impieghi.

Essendo Java un linguaggio compreso da *Emacs* è stato possibile utilizzarlo per l'elaborazione dell'impresa virtuale. Inoltre, questo editor è consigliato da *Swarm* che lo promuove insieme alla versione corrente del progetto.

Per poter compilare in Java da *Emacs* è necessario affiancargli il *JDK* reperibile al sito della *Sun*: www.sun.com.

D.1.2 JBuilder

JBuilder è un ambiente *Java* sviluppato dalla *Borland* e reperibile all'indirizzo: www.borland.com.

L'attuale versione è la 4.0 che di recente ha sostituito la 3.5 perfezionandone alcuni aspetti relativi all'impostazione del progetto ed ha incluso il *JDK1.3* in luogo del *JDK1.2.2*.

In quanto ambiente di sviluppo, *JBuilder* contiene un editor Java, gli strumenti per la documentazione e gli strumenti per disegnare finestre e grafici (utili specie per lo sviluppo di *Applet*).

Per programmare è prevista l'impostazione di un progetto nel quale si possono indicare le cartelle di provenienza dei file.java, le cartelle di destinazione dei file.class e le librerie aggiuntive richieste dal progetto. Ad esempio nel caso dell'impresa virtuale è stato necessario aggiungere le librerie *swarm.jar* per le funzioni relative a *Swarm* e *plot.jar*, *gui.jar* per le funzioni relative ai grafici di *Ptolemy*.

Per compilare ed eseguire è sufficiente dare il comando *RUN* alla classe contenente il *main()* del programma.

D.1.3 Together

*TogetherUML*¹ è un ambiente di sviluppo elaborato dalla *TogetherSoft* e reperibile all'indirizzo: www.togethersoft.com.

Molto simile a *JBuilder* come impostazione di lavoro e per gli strumenti forniti; anch'esso prevede la formulazione di un progetto di lavoro con tutte le informazioni relative alla compilazione ed esecuzione del programma. Un'aggiunta degna di nota è inerente alla possibilità di sviluppare un progetto con *Java* parallelamente ad *UML*² sia in direzione di *Engineering* sia di *Reverse-Engineering*.

D.1.4 Kawa

Kawa è più simile ad un *editor* che ad un ambiente di sviluppo; infatti, ad esempio, più "leggero" prevede un'installazione a parte del *JDK*. Per *Java* è prevista la possibilità di definire un progetto di lavoro con tutte le opzioni specifiche relative all'*input* ed all'*output* del programma.

¹Per l'utilizzo di questo strumento la *TogetherSoft* ha concesso una licenza gratuita per *solì fini di studio* al *Dipartimento di Scienze Economiche e Finanziarie (G. Prato)* della Facoltà di Economia di Torino.

²Si veda l'appendice B.

D.2 Il codice

D.2.1 StartVEFrame.java

```
// StartVEFrame.java - jveframe application
// Pietro Terna (October 2000-June 2001)
// Dipartimento di Scienze economiche e finanziarie G.Prato
// Università di Torino
// Some building ideas and many comments come
// from jHeatbugs (Java Heatbugs application.
// Copyright © 1999-2000 Swarm Development Group.)

import swarm.Globals; /**
 * The StartVEFrame class contains main().
 * We follow here the typical Swarm structure with main()
 * (in Start... as a convention) generating the Observer
 * and the Observer generating the Model.<br>
 * @author Pietro Terna
 */

public class StartVEFrame
{
    /** the observer in our application */
    public static VEFrameObserverSwarm vEFrameObserverSwarm;

    /** being verbose? */
    public static boolean verbose;

    /**
```

```
* The main() function is the top-level place where
    * everything starts.
*/
public static void main (String[] args) {

    // Swarm initialization
    Globals.env.initSwarm
    ("jveframe", "v.0.4", "pietro.terna@unito.it", args);

    // creating the VEFrameObserverSwarm via lispAppArchiver,
    // which refers to the jveframe.scm file where we
    // can modify parameters
    // in a stable way without recompile the interested class
    vEFrameObserverSwarm =
    (VEFrameObserverSwarm)
        Globals.env.lispAppArchiver.getWithZone$key(
        Globals.env.globalZone, "vEFrameObserverSwarm");
    // instead of using the direct constructor way
    // vEFrameObserverSwarm = new
    //      VEFrameObserverSwarm (Globals.env.globalZone);

    // to save control panel position
    Globals.env.setWindowGeometryRecordName
        (vEFrameObserverSwarm, "vEFrameObserverSwarm");

    // build objets into the obsever
    vEFrameObserverSwarm.buildObjects ();
    // verbose now is true/false, following the choice made
```

```

// in the Observer

// build actions into the observer
vEFrameObserverSwarm.buildActions ();

// activate
vEFrameObserverSwarm.activateIn (null);

// go() in vEFrameObserver is a method inherited from class
// swarm.simtoolsgui.GUISwarmImpl
// go() start the activity running, and also handle
// user requests via the control panel.
vEFrameObserverSwarm.go ();

// concluding the activity we drop the observer
vEFrameObserverSwarm.drop ();
    // to quit also the executions of java graphs
    System.exit(0);
}
}

```

D.2.2 VEFrameObserverSwarm.java

```

import swarm.Globals; import swarm.Selector; import
swarm.defobj.Zone;

import swarm.activity.Activity;

```

```
import swarm.activity.ActionGroup; import
swarm.activity.ActionGroupImpl; import swarm.activity.Schedule;
import swarm.activity.ScheduleImpl;

import swarm.objectbase.Swarm; import swarm.objectbase.VarProbe;
import swarm.objectbase.MessageProbe; import
swarm.objectbase.EmptyProbeMapImpl;

import swarm.simtoolsgui.GUISwarm; import
swarm.simtoolsgui.GUISwarmImpl;

import swarm.analysis.EZGraph; import swarm.analysis.EZGraphImpl;

import java.util.List;

/**
 * The Observer contains the Model and the graphic widgets
 * @author  Pietro Terna
 */
public class VEFrameObserverSwarm extends GUISwarmImpl {

    /** update frequency */
    public int displayFrequency;
    /** displaying all messages on the console */
    public boolean verboseChoice;
    /** ActionGroup for sequence of GUI events */
    public ActionGroup displayActions;
    /** the single Schedule instance */
    public Schedule displaySchedule;
```

```

/** the Swarm we're observing */
public VEFrameModelSwarm vEFrameModelSwarm;
/** our graphics */
public EZGraphImpl waitingListGraph, warehouseGraph,
    totalTimeLengthGraph, totalTimeLengthDataFile,
    totalDailyCostFile, totalCostFile, totalRevenueFile,
    dailyRevenueFile, totalInProductionRevenueFile,
    totalInventoryFinancialCostFile,
    dailyStoredComponentValueFile,
    totalOrderCostFile, dailyOrderCostFile,
    returnFile, returnGraph;

/** Java graphics*/
public PTHistogram ptHistogram;
/** Constructor for class */
public VEFrameObserverSwarm (Zone aZone) {
    super(aZone);
    // Fill in the relevant parameters (only two, in this case).
    displayFrequency = 1;
    verboseChoice=true;
    // Build a customized probe map using a 'local' subclass
    // (a special kind of Java 'inner class') of the
    // EmptyProbeMapImpl class.
    // This building operation produces a separated class
    // file, with name
    //VEFrameObserverSwarm$1$VEFrameObserverProbeMap.class
    class VEFrameObserverProbeMap extends EmptyProbeMapImpl {
        private VarProbe probeVariable (String name) {
            return
                Globals.env.probeLibrary.getProbeForVariable$inClass

```

```

        (name, VEFrameObserverSwarm.this.getClass ());
    }
    private MessageProbe probeMessage (String name) {
        return
            Globals.env.probeLibrary.getProbeForMessage$inClass
            (name, VEFrameObserverSwarm.this.getClass ());
    }
    private void addVar (String name) {
        addProbe (probeVariable (name));
    }
    private void addMessage (String name) {
        addProbe (probeMessage (name));
    }
}

public VEFrameObserverProbeMap (Zone _aZone, Class aClass){
    super (_aZone, aClass);
    addVar ("displayFrequency");
addVar("verboseChoice");
    }
}

// Install our custom probeMap class directly into the
// probeLibrary
Globals.env.probeLibrary.setProbeMap$For
    (new VEFrameObserverProbeMap (aZone, getClass ()),
                                     getClass ());
}

/** Create the objects used to display of the model. */

```

```
public Object buildObjects () {

    super.buildObjects ();

    // First, we create the model that we're actually observing.
    // The model is a subswarm of the observer.

    // creating the VEFrameModelSwarm via lispAppArchivers,
    // which refers to the jveframe.scm file where we can
    // modify parameters in a stable way without recompile
    // the interested class
    VEFrameModelSwarm =
    (VEFrameModelSwarm)Globals.env.lispAppArchiver.
        getWithZone$key(getZone(), "vEFrameModelSwarm");
    // instead of using the direct constructor way
    // vEFrameModelSwarm = new VEFrameModelSwarm (getZone ());

    // Now create probe objects on the model and ourselves.
    // This gives a simple user interface to let the user
    // change parameters.

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

    // Instruct the control panel to wait for a button event: we
    // halt here until someone hits a control panel button so the
    // user can get a chance to fill in parameters before the
```



```

// simulation runs
getControlPanel ().setStateStopped ();

// to change the verboseChoice via probe you have to wright
// 'true' or 'false'; not simply 't' or 'f'

// OK - the user has specified all the parameters for the
// simulation. Now we're ready to start.

// setting the 'verbose' value in StartVEFrame
StartVEFrame.verbose=verboseChoice;

// First, let the model swarm build its objects.
vEFrameModelSwarm.buildObjects ();

// Create the graph widget to display the waiting
// list content
waitingListGraph = new EZGraphImpl
(getZone (), "Waiting lists", "Time",
"Waiting list values (avrg, min, max)","waiting_list");

Globals.env.setWindowGeometryRecordName (waitingListGraph,
"waitingListGraph");

// create the data for the waitingListGraph
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"));

// Create the graph widget to display the quantities stored
// in each unit warehouse

    if (vEFrameModelSwarm.getUseWarehouses())
        // if useWarehouses in true, draw the graph
    {
warehouseGraph = new EZGraphImpl
    (getZone (), "Quantities in the warehouses", "Time",
        "Quantities (avrg, min, max)", "warehouse_list");

Globals.env.setWindowGeometryRecordName
    (warehouseGraph, "warehouseGraph");

// the data for the warehouseGraph
warehouseGraph.
    createAverageSequence$withFeedFrom$andSelector
    ("avrgQ", vEFrameModelSwarm.getWarehouseList(),
        SwarmUtils.getSelector("Warehouse",
                                "getInventoryCounterValue"));
warehouseGraph.createMinSequence$withFeedFrom$andSelector

```

```

        ("minQ", vEFrameModelSwarm.getWarehouseList(),
        SwarmUtils.getSelector("Warehouse",
                                "getInventoryCounterValue"));
warehouseGraph.createMaxSequence$withFeedFrom$andSelector
    ("maxQ", vEFrameModelSwarm.getWarehouseList(),
    SwarmUtils.getSelector("Warehouse",
                            "getInventoryCounterValue"));
}

// Create the graph widget to display the total
// production time for all the done orders divided
// by their total recipe lengths

totalTimeLengthGraph = new EZGraphImpl
(getZone (), "Ratio total time / total lengths", "Time",
 "Ratio", "totalTimeLengthGraph");

Globals.env.setWindowGeometryRecordName
(totalTimeLengthGraph, "totalTimeLengthGraph");

// the data for the totalTimeLengthGraph
totalTimeLengthGraph.createSequence$withFeedFrom$andSelector
    ("ratio", vEFrameModelSwarm,
    SwarmUtils.getSelector(vEFrameModelSwarm,
                            "getTimeLengthRatio"));

// the same, in a file ('data.ratio')
    totalTimeLengthDataFile = new EZGraphImpl
    (getZone(), "data");
totalTimeLengthDataFile.createSequence$withFeedFrom$andSelector

```

```

        ("ratio", vEFrameModelSwarm,
        SwarmUtils.getSelector(vEFrameModelSwarm,
                                "getTimeLengthRatio")));
// the amount of totalDailyCost of Units in a file
// (totalDailyCost.txt)
totalDailyCostFile = new EZGraphImpl
    (getZone(), "OutputCost/totalDailyCost");
totalDailyCostFile.createTotalSequence$withFeedFrom$andSelector
    ("txt", vEFrameModelSwarm.getUnitList(),
    SwarmUtils.getSelector("Unit","getTotalDailyCost"));
// the cumulated amount of fixed and variable costs of Units
// from the beginnig of the simulation in a file
// (totalCostFile.txt)
totalCostFile = new EZGraphImpl
    (getZone(), "OutputCost/totalCost");
totalCostFile.createTotalSequence$withFeedFrom$andSelector
    ("txt", vEFrameModelSwarm.getUnitList(),
    SwarmUtils.getSelector("Unit","getTotalCost"));
// the revenue of the virtual enterprise in a file
// (totalRevenue.txt)
totalRevenueFile = new EZGraphImpl
    (getZone(), "Revenue/totalRevenue");
totalRevenueFile.createTotalSequence$withFeedFrom$andSelector
    ("txt", vEFrameModelSwarm.getUnitList(),
    SwarmUtils.getSelector("Unit","getTotalSaleRevenue"));
// the daily revenue from the orders selling
dailyRevenueFile = new EZGraphImpl
    (getZone(), "Revenue/dailyRevenue");
dailyRevenueFile.createTotalSequence$withFeedFrom$andSelector

```

```

        ("txt", vEFrameModelSwarm.getUnitList(),
        SwarmUtils.getSelector("Unit","getSaleRevenue"));
// the revenue of the in production order
// (totalInProductionRevenue.txt)
totalInProductionRevenueFile = new EZGraphImpl
(getZone(), "Revenue/totalSemimanufacturedProductRevenue");
totalInProductionRevenueFile.
createTotalSequence$withFeedFrom$andSelector
    ("txt", vEFrameModelSwarm.getOrderList(),
    SwarmUtils.getSelector("Order","getOrderPrice"));
// the cumulated financial costs of the semi manufactured
// in warehouses in a
// file (totalInventoryFinancialCost.txt)
if(vEFrameModelSwarm.getUseWarehouses()==true){
    totalInventoryFinancialCostFile = new EZGraphImpl
    (getZone(), "OutputCost/totalInventoryFinancialCost");
totalInventoryFinancialCostFile.
createTotalSequence$withFeedFrom$andSelector
    ("txt", vEFrameModelSwarm.getUnitList(),
    SwarmUtils.getSelector("Unit",
        "getTotalInventoryFinancialCost"));
}
//the revenue of the components stored for each day
if(vEFrameModelSwarm.getUseWarehouses()==true){
    dailyStoredComponentValueFile = new EZGraphImpl
    (getZone(), "Revenue/dailyStoredComponentValue");
dailyStoredComponentValueFile.
createTotalSequence$withFeedFrom$andSelector
    ("txt", vEFrameModelSwarm.getUnitList(),

```

```

        SwarmUtils.getSelector("Unit",
                                "getDailyStoredComponentValue"));
    }

    // the cost of the orders from the beginning of simulation
    // (totalOrderCost.txt)
    dailyOrderCostFile = new EZGraphImpl
        (getZone(), "OutputCost/orderCostProduced");
    dailyOrderCostFile.
createTotalSequence$withFeedFrom$andSelector
    ("txt", vEFrameModelSwarm.getUnitList(),
        SwarmUtils.getSelector("Unit",
                                "getTotalCostOrderProduced"));

    // the daily cost of the orders in production
    // (dailyOrderCost.txt)
    totalOrderCostFile = new EZGraphImpl
        (getZone(), "OutputCost/totalSemimanufacturedProductCost");
    totalOrderCostFile.
        createTotalSequence$withFeedFrom$andSelector
            ("txt", vEFrameModelSwarm.getOrderList(),
                SwarmUtils.getSelector("Order", "getTotalCostForOrder"));

    // the data for the returnGraph
    returnGraph = new EZGraphImpl
        (getZone(), "Return of Enterprise", "Time",
        "Returns", "returnGraph");
    Globals.env.setWindowGeometryRecordName
        (returnGraph, "returnGraph");
    returnGraph.createSequence$withFeedFrom$andSelector

```

```

        ("returnGraph", vEFrameModelSwarm,
        SwarmUtils.getSelector(vEFrameModelSwarm,
                                "getReturnOfEnterprise"));

// the same in a file
returnFile = new EZGraphImpl
(getZone(), "Return/return");
returnFile.createSequence$withFeedFrom$andSelector
("txt", vEFrameModelSwarm,
    SwarmUtils.getSelector(vEFrameModelSwarm,
                            "getReturnOfEnterprise"));

// construct the Histogram for the WaitingList of Units
// and for the inventories in Warehouse
if(vEFrameModelSwarm.getUseWarehouses()==true){
    ptHistogram = new PTHistogram("Orders", "Virtual Enterprise",
        "Count", vEFrameModelSwarm.totalUnitNumber, 20.0,
        vEFrameModelSwarm.unitList,
        vEFrameModelSwarm.warehouseList);
}
else
    ptHistogram = new PTHistogram("Orders", "Virtual Enterprise",
        "Count", vEFrameModelSwarm.totalUnitNumber, 20.0,
        vEFrameModelSwarm.unitList);

return this;
}

/**
Create the actions necessary for the simulation. This is where
the schedule is built (but not run!) Here we create a display
schedule - this is used to display the state of the world and
check for user input. */

```

```

public Object buildActions () {
    super.buildActions();
    // First, let our model swarm build its own schedule.
    vEFrameModelSwarm.buildActions();
    // Create an ActionGroup for display: a bunch of things that
    // occur in a specific order.
    displayActions = new ActionGroupImpl (getZone());
    // Add the methods to the ActionGroup
    // Schedule the update of the probe displays
    displayActions.createActionTo$message
        (Globals.env.probeDisplayManager,
        SwarmUtils.getSelector(Globals.env.probeDisplayManager,
                                "update"));

    // Schedule the update of the graphic widgets
    displayActions.createActionTo$message
        (waitingListGraph,
        SwarmUtils.getSelector(waitingListGraph, "step"));
    if(vEFrameModelSwarm.getUseWarehouses())
    displayActions.createActionTo$message(
        warehouseGraph,SwarmUtils.getSelector(warehouseGraph,
                                                "step"));

    displayActions.createActionTo$message
        (totalTimeLengthGraph,
        SwarmUtils.getSelector(totalTimeLengthGraph, "step"));
    displayActions.createActionTo$message
        (totalTimeLengthDataFile,
        SwarmUtils.getSelector(totalTimeLengthGraph, "step"));
    //Schedule to update Java histogram

```



```

displayActions.createActionTo$message
(ptHistogram,SwarmUtils.getSelector(ptHistogram,
                                     "addPoints"));

//schedule to update the EZGraph file
displayActions.createActionTo$message
    (totalDailyCostFile,
     SwarmUtils.getSelector(totalDailyCostFile, "step"));
displayActions.createActionTo$message
    (totalCostFile,
     SwarmUtils.getSelector(totalCostFile, "step"));
displayActions.createActionTo$message
    (totalRevenueFile,
     SwarmUtils.getSelector(totalRevenueFile, "step"));
displayActions.createActionTo$message
    (dailyRevenueFile,
     SwarmUtils.getSelector(dailyRevenueFile, "step"));
displayActions.createActionTo$message
    (totalInProductionRevenueFile,
     SwarmUtils.getSelector(totalInProductionRevenueFile,
                             "step"));
if(vEFrameModelSwarm.getUseWarehouses()){
    displayActions.createActionTo$message
        (totalInventoryFinancialCostFile,
         SwarmUtils.getSelector(totalInventoryFinancialCostFile,
                                 "step"));
}

if(vEFrameModelSwarm.getUseWarehouses()){
    displayActions.createActionTo$message

```

```

        (dailyStoredComponentValueFile,
        SwarmUtils.getSelector(dailyStoredComponentValueFile,
                                "step")));
    }

    displayActions.createActionTo$message
        (totalOrderCostFile,
        SwarmUtils.getSelector(totalOrderCostFile, "step"));
    displayActions.createActionTo$message
        (dailyOrderCostFile,
        SwarmUtils.getSelector(dailyOrderCostFile, "step"));
    displayActions.createActionTo$message
        (returnFile,
        SwarmUtils.getSelector(returnFile, "step"));
    displayActions.createActionTo$message
        (returnGraph,
        SwarmUtils.getSelector(returnGraph, "step"));

    // Finally, schedule an update for the whole user
    // interface code. This is crucial: without this, no
    // graphics update and the control panel will be
    // dead. It's best to put it at the end of the display
    // schedule
    displayActions.createActionTo$message(getActionCache(),
        SwarmUtils.getSelector(getActionCache(), "doTkEvents"));

    // The display schedule. Note the repeat interval: display
    // is frequently the slowest part of a simulation,

```

```
// so redrawing less frequently can be a help.
displaySchedule = new ScheduleImpl(getZone (),
                                     displayFrequency);

// insert ActionGroup instance on the repeating Schedule
// instance
displaySchedule.at$createAction (0, displayActions);

return this;
}

/**
*activateIn: - activate the schedules so they're ready to run.
*The swarmContext argument has to do with what we were activated
*in*. Typically the ObserverSwarm is the top-level Swarm, so
*it's activated in "null". But other Swarms and Schedules and
*such will be activated inside of us. */

public Activity activateIn (Swarm swarmContext) {
    // First, activate ourselves (just pass along the context).
    super.activateIn (swarmContext);

    // Activate the model swarm in ourselves. The model swarm is a
    // subswarm of the observer swarm.
    vEFrameModelSwarm.activateIn (this);

    // Now activate our schedule in ourselves. This arranges for
    // the execution of the schedule we built.
    displaySchedule.activateIn (this);
}
```

```

// Activate returns the swarm activity - the thing that's
// ready to run.
return getActivity();
}

```

```

/** drop the Observer, but after having dropped
 * totalTimeLengthDataFile
 * output to file, to close the file */
public void drop() {
totalTimeLengthDataFile.drop();
totalTimeLengthDataFile=null;
    totalDailyCostFile.drop();
    totalDailyCostFile=null;
    totalCostFile.drop();
    totalCostFile=null;
    totalRevenueFile.drop();
    totalRevenueFile=null;
    dailyRevenueFile.drop();
    dailyRevenueFile=null;
    totalInProductionRevenueFile.drop();
    totalInProductionRevenueFile=null;
    if(vEFrameModelSwarm.getUseWarehouses()==true){
        totalInventoryFinancialCostFile.drop();
        totalInventoryFinancialCostFile=null;
    }
    if(vEFrameModelSwarm.getUseWarehouses()==true){
        dailyStoredComponentValueFile.drop();
        dailyStoredComponentValueFile=null;
    }
}

```

```
    }  
    totalOrderCostFile.drop();  
    totalOrderCostFile=null;  
    dailyOrderCostFile.drop();  
    dailyOrderCostFile=null;  
    returnFile.drop();  
    returnFile=null;  
  
    super.drop();  
}  
}
```

D.2.3 VEFrameModel.java

```
import swarm.Globals; import swarm.Selector; import  
swarm.defobj.Zone; import swarm.defobj.SymbolImpl;  
  
import swarm.activity.Activity;  
  
import swarm.activity.ActionGroup; import  
swarm.activity.ActionGroupImpl; import swarm.activity.Schedule;  
import swarm.activity.ScheduleImpl;  
  
import swarm.objectbase.Swarm; import  
  
swarm.objectbase.SwarmImpl;  
  
import swarm.objectbase.VarProbe; import
```

```

swarm.objectbase.MessageProbe; import
swarm.objectbase.EmptyProbeMapImpl;

import swarm.collections.ListImpl;

/**
 * The Model contains the Units and all the related tools,
 * like the warehouses which are necessary to deal with
 * the inventories.
 * @author Pietro Terna
 */
public class VEFrameModelSwarm extends SwarmImpl {
    // simulation parameters

    /** the total number of Units we are using*/
    public int totalUnitNumber=0;
    /** max number of steps to be done to complete an order*/
    public int maxStepNumber=0;
    /** max number of units (inventories) in a warehouse */
    public int maxInWarehouses;
    /** choosing if we use or not warehouses to manage the
     * inventories */
    public boolean useWarehouses;
    /** choosing if we use or not news */
    public boolean useNews;
    /** How deep information is propagated (how many steps after
     * the current productin phase)*/
    public int infDeepness;
    /** the revenue of the enterprise for each step done

```

```
* of the order recipe */
public double revenuePerEachRecipeStep;
/** the daily rate to evaluate the financial costs of
 * pieces stored in warehouses */
public double inventoryFinancialRate;
/** the value to estimate the stored products */
public double inventoryEvaluationCriterion;
/** the total production time used by done orders */
public int totalProductionTime;
/** the total length of the recipes of the done products */
public int totalRecipeLength;
/** order generator, to be replaced in the future by more
    sophisticated structures (a swarm of enterprises etc.) */
public OrderGenerator orderGenerator;
/** ActionGroup for holding an ordered sequence of action */
public ActionGroup modelActions;
/** the Schedule operating in the Model*/
public Schedule modelSchedule;
/** the inventoryRuleMaster managing inventories */
public InventoryRuleMaster inventoryRuleMaster;
/** informationRuleMaster managing information system */
public InformationRuleMaster informationRuleMaster;
/** the I/O class */
public AccountingData accountingData;
/** the return of our enterprise */
public double returnOfEnterprise;
/** the first list of the units
 * (modified by scheduled events) */
public ListImpl unitList;
```

```

    /** the list of the warehouses */
    public ListImpl warehouseList;

    /** the list of the orders */
    public ListImpl orderList;

    public VEFrameModelSwarm (Zone aZone) {
        super (aZone);

        // Now fill in various simulation parameters with default values.
        totalUnitNumber = 3;
        maxStepNumber=5;
        useWarehouses=true;
        maxInWarehouses=3;
        useNews=true;
        infDeepness=3;
        inventoryFinancialRate=0.1;
        revenuePerEachRecipeStep=2;
        inventoryEvaluationCriterion=2;

        // Build a customized probe map using a 'local' subclass
        // (a special kind of Java 'inner class') of the
        // EmptyProbeMapImpl class.

        // This building operation produces a separated class file,
        // with name  VEFrameModelSwarm$1$VEFrameModelProbeMap.class
        class VEFrameModelProbeMap extends EmptyProbeMapImpl {
            private VarProbe probeVariable (String name) {
                return
                    Globals.env.probeLibrary.getProbeForVariable$inClass
                        (name, VEFrameModelSwarm.this.getClass ());
            }

            private MessageProbe probeMessage (String name) {

```



```

        return
        Globals.env.probeLibrary.getProbeForMessage$inClass
        (name, VEFrameModelSwarm.this.getClass ());
    }

    private void addVar (String name) {
        addProbe (probeVariable (name));
    }

    private void addMessage (String name) {
        addProbe (probeMessage (name));
    }

    public VEFrameModelProbeMap (Zone _aZone, Class aClass) {
        super (_aZone, aClass);
        addVar ("totalUnitNumber");
        addVar ("maxStepNumber");
        addVar ("useWarehouses");
        addVar ("useNews");
        addVar ("maxInWarehouses");
        addVar ("infDeepness");
        addVar ("inventoryFinancialRate");
        addVar ("revenuePerEachRecipeStep");
        addVar ("inventoryEvaluationCriterion");
    }
}

// Install our custom probeMap class directly into the
// probeLibrary
Globals.env.probeLibrary.setProbeMap$For
(new VEFrameModelProbeMap (aZone,getClass ()),getClass ());
}

```

```

/** Build the model objects. */
public Object buildObjects ()
{
    int i;
    // allow our parent class to build anything.
    super.buildObjects();

    //create the class to read data
    accountingData = new AccountingData(totalUnitNumber);
    //call the method to read
    accountingData.readFixedData();
    accountingData.readVariableData();

    //check maxInWarehouses valie and if 0 change it to the max
    //integer value
    if (maxInWarehouses==0)
    {
        maxInWarehouses=Integer.MAX_VALUE;
        // no verbose control here
        System.out.println(
            "Warning, maxInWarehouses has been changed"+
            "from 0 to Integer.MAX_VALUE");
    }

    // the lists of the production units
    unitList = new ListImpl (getZone());
    warehouseList = new ListImpl (getZone());
    // the list of the order
    orderList = new ListImpl(getZone());
    // the orderGenerator
    orderGenerator = new OrderGenerator
        (getZone(), maxStepNumber, unitList, orderList, this);

```



```

        informationRuleMaster,infDeepness,
        accountingData.getFixedCost(i),
        accountingData.getVariableCost(i),
        orderList);

    }

    else aUnit = new Unit  (getZone(), i, unitList, this,
        accountingData.getFixedCost(i),
        accountingData.getVariableCost(i), orderList);

    //Add the unit and the warehouse to the end of the lists
    unitList.addLast (aUnit);
    if(useWarehouses) warehouseList.addLast (aWarehouse);
    }

    if(! useWarehouses)if(StartVEFrame.verbose)
        System.out.println("No warehouses generated");
    return this;
    }

    /**
     * Here is where the model schedule is built,
     * the data structures
     * that define the simulation of time in the model.
     * The core is an actionGroup that has a list of actions.
     * That's then put in a Schedule.  */
    public Object buildActions ()
    {
        super.buildActions();
    }

```

```

modelActions = new ActionGroupImpl (getZone ());
modelActions.createActionForEach$message(unitList,
    SwarmUtils.getSelector("Unit","unitStep1"));

// Order generation and order propagation (step 2)
// are both after step1 (production)
modelActions.createActionTo$message(orderGenerator,
    SwarmUtils.getSelector(orderGenerator,
        "createRandomOrderWithNSteps"));
modelActions.createActionForEach$message(unitList,
    SwarmUtils.getSelector("Unit","unitStep2"));

//at the end of the day the enterprise computes its return
//modelActions.createActionTo$message(this,
//SwarmUtils.getSelector(this,"getReturnOfEnterprise"));

// Then we create a schedule that executes the
// modelActions. modelActions is an ActionGroup, by itself it
// has no notion of time. In order to have it executed in
// time, we create a Schedule that says to use the
// modelActions ActionGroup at particular times. This
// schedule has a repeat interval of 1, it will loop every
// time step. The action is executed at time 0 relative to
// the beginning of the loop.

modelSchedule = new ScheduleImpl (getZone (), 1);
modelSchedule.at$createAction (0, modelActions);

return this;

```

```

}

/**
 * Now set up the model's activation. swarmContext
 * indicates where we're being started in - typically,
 * this model is run as a subswarm of an observer swarm.*/
public Activity activateIn (Swarm swarmContext) {
    // First, activate ourselves via the superclass
    // activateIn: method. Just pass along the context: the
    // activity library does the right thing.
    super.activateIn (swarmContext);
    // Now activate our own schedule.
    modelSchedule.activateIn (this);
    // Finally, return our activity.
    return getActivity ();
}

/** the method returns the list of the units */
public Object getUnitList(){
    return unitList;
}

/** the method returns the list of the warehouses */
public Object getWarehouseList (){
    return warehouseList;
}

/** the method returns the list of the orders */
public Object getOrderList(){
    return orderList;
}

/** this get method returns true if warehouses

```

```

    * are used, false otherwise */
    public boolean getUseWarehouses() {
        return useWarehouses;
    }

    /** record total production time and total recipe length */
    public void setProductionTimeAndRecipeLength(int pt, int rl){
        totalProductionTime += pt;
        totalRecipeLength    += rl;
        if(StartVEFrame.verbose)
            System.out.println("totalProductionTime "
                               + totalProductionTime +
                               " totalRecipeLength " + totalRecipeLength +
                               " ratio " + ((float)
                                             totalProductionTime)/totalRecipeLength);
    }
    return;
}

/** report the ratio totalProductionTime/totalRecipeLength*/
public float getTimeLengthRatio() {
    if(totalRecipeLength != 0)
        return ((float) totalProductionTime)/totalRecipeLength;
    else return (float) 1;

    // 1 is a good choice, being 0 totalRecipeLength
    // in the first simulation days and being 1
    // the practical actual ratio of the first
    // concluded orders
}

/** the method to pass the financial rate to the unit */
public double getInventoryFinancialRate(){

```

```

        return inventoryFinancialRate;
    }

    /** the method to pass the revenue for the order recipe */
    public double getRevenuePerEachRecipeStep(){
        return revenuePerEachRecipeStep;
    }

    /** the method to pass the value to evaluate stored products */
    public double getInventoryEvaluationCriterion(){
        return inventoryEvaluationCriterion;
    }

    /** the method to record the return of the enterprise */
    public double getReturnOfEnterprise(){
        returnOfEnterprise = 0;
        double totalRevenueOfEnterprise=0;
        double totalCostOfEnterprise=0;
        double totalSemimanufacturedRevenue=0;
        double dailyStoredComponentRevenue=0;
        double totalInventoryFinCost=0;
        Unit aUnit;
        Order anOrder;

        for(int i=0; i<unitList.getCount(); i++){
            aUnit=(Unit)unitList.atOffset(i);
            totalRevenueOfEnterprise += aUnit.getTotalSaleRevenue();
            totalCostOfEnterprise += (double)aUnit.getTotalCost();
            if(useWarehouses){
                dailyStoredComponentRevenue +=
                    aUnit.getDailyStoredComponentValue();
                totalInventoryFinCost +=

```



```

        aUnit.getTotalInventoryFinancialCost();
    }
}

for(int j=0; j<orderList.getCount(); j++){
    anOrder = (Order)orderList.atOffset(j);
    totalSemimanufacturedRevenue+=anOrder.getOrderPrice();
}

returnOfEnterprise = totalRevenueOfEnterprise +
    dailyStoredComponentRevenue +
    totalSemimanufacturedRevenue -
    totalCostOfEnterprise - totalInventoryFinCost;

return returnOfEnterprise;
}
}

```

D.2.4 Unit.java

```

import swarm.Globals; import swarm.defobj.Zone; import
swarm.objectbase.SwarmObjectImpl;

import swarm.collections.ListImpl;

/**
 * The Unit class instances are micro unit in our
 * virtual enterprise; i.e. the units where the steps

```

```

* required to build a product (to fulfill an order) are done
* @author Pietro Terna (with a Marco Remondino contribution)
*/
public class Unit extends SwarmObjectImpl{

    /** the number that identifies the unit*/
    public int unitNumber;

    /** choosing if we use or not the warehouses and so if
    we can or not to provide inventories of our product */
    public boolean useWarehouses;

    /** choosing if we use or not the news */
    public boolean useNews;

    /** a flag about using pieces in warehouse */
    public boolean madeProduction;

    /** the list of all the units existing in our environment*/
    public ListImpl unitList;

    /** the list of the orders to be executed */
    public ListImpl waitingList;

    /** the list contained the order (if any) performed in a day,
    also using the content of our warehouse */
    public ListImpl dailyProductionList;

    /** the model (to which we apply the
        * setProductionTimeAndRecipeLength(...) method */
    VEFrameModelSwarm vEFrameModelSwarm;

    /** the temporary address of an existing unit */
    public Unit aUnit;

    /**the temporary address of a unit to which send a news*/
    public Unit receiverUnit;

    /** our warehouse, if any */

```

```
public Warehouse myWarehouse;
/** the first oder to be executed */
public Order firstOrder;
/**first oder of the waitingList inserted in the news*/
public Order incomingOrder;
/** the news to be sent */
public News aNews;
/** the news to be deleted a the arrive of the
 * respecctive order */
public News toBeDroppedNews;
/** our rule master about inventories */
public InventoryRuleMaster inventoryRuleMaster;
/** our rule master about information system */
public InformationRuleMaster informationRuleMaster;
/** the list of the news */
public ListImpl newsList;
/** How deep information is propagated (how many steps
 * after the current productin phase)*/
public int informationDeepness;
/** the list of the orders */
public ListImpl orderList;
/** our fixed and variable costs in input */
public int fixedCost, variableCost;
/** our total daily variable costs */
public int totalDailyVariableCost;
/** our total daily fixed and variable costs */
public int totalDailyCost;
/** the costs from the begginning of the simulation */
public int totalCost = 0;
```

```

/**revenue coming from the seeling of dropped order */
public double saleRevenue;
/** the comulated revenue from the beginning of
 * the simulation */
public int totalSaleRevenue = 0;
/**cost coming from the production of dropped order */
public int totalCostOrderProduced = 0;
/** the the value of the stored component */
public double dailyStoredComponentValue = 0;
/** the comulated inventory financial costs generated
 * from storing pieces in warehouses
 */
double totalInventoryFinancialCost = 0;

/**
 * the first Constructor for Unit
 * (with warehouses in simulation)
 */
public Unit (Zone aZone, int un, ListImpl ul,
             VEFrameModelSwarm mo, InventoryRuleMaster rm,
             Warehouse wh, int fC, int vC, ListImpl ol)
{
// Call the constructor for the Unit parent class.
super(aZone);

useWarehouses = true;
    // if we are using this constructor the
    // warehouses are activated
useNews = false;

```

```
// if we are using this constructor the
//news are not activated
    inventoryRuleMaster = rm; // our InventoryRuleMaster
myWarehouse = wh;      // our warehouse address

// the production waiting list of the unit
waitingList = new ListImpl (getZone());
// the unit id number.
unitNumber=un;
// the list of all the units
unitList=ul;
// the model
vEFrameModelSwarm = mo;
// the list of orders
    orderList = ol;
    // our fixed and variable costs
    variableCost = vC;
    fixedCost = fC;
dailyProductionList= new ListImpl(getZone());

// Announce the unit to the console
if(StartVEFrame.verbose)
    System.out.println("Unit number " + unitNumber +
        " has been created.");
}

/**
 * the second Constructor for Unit
 * (without warehouses in simulation)
 */
```

```
public Unit (Zone aZone, int un, ListImpl ul,
            VEFrameModelSwarm mo, int fC, int vC,
            ListImpl ol) {

    // Call the constructor for the Unit parent class.
    super(aZone);

    useWarehouses = false;
    // if we are using this constructor the
    //warehouses are not activated
    useNews = false;
    // if we are using this constructor the
    // news are not activated
    inventoryRuleMaster = null;
    myWarehouse = null;

    // the production waiting list of the unit
    waitingList = new ListImpl (getZone());
    // the unit id number.
    unitNumber=un;
    // the list of all the units
    unitList=ul;
    // the model
    vEFrameModelSwarm = mo;
    // the daily list
    dailyProductionList= new ListImpl(getZone());
    orderList = ol;
    // our fixed and variable costs
    variableCost = vC;
```

```

        fixedCost = fC;
// Announce the unit to the console.
if(StartVEFrame.verbose)
    System.out.println("Unit number " + unitNumber +
        " has been created.");
}
/**
 * the third Constructor for Unit
 * (with warehouses and news in simulation)
 */
public Unit (Zone aZone, int un, ListImpl ul,
            VEFrameModelSwarm mo, InventoryRuleMaster rm,
            Warehouse wh, InformationRuleMaster irm,
            int IDeepness, int fC, int vC, ListImpl ol)
{
// Call the constructor for the Unit parent class.
super(aZone);

useWarehouses = true;
useNews = true;
inventoryRuleMaster = rm;
informationRuleMaster= irm;
myWarehouse = wh;

// the production waiting list of the unit
waitingList = new ListImpl (getZone());
// the unit id number.
unitNumber=un;
// the list of all the units

```

```

unitList=ul;
// the model
vEFrameModelSwarm = mo;
// the dayly list
dailyProductionList= new ListImpl(getZone());
    //the news list
    newList= new ListImpl(getZone());
    // How deep information is propagated
    informationDeepness= IDeepness;
    orderList = ol;
    // our fixed and variable costs
    variableCost = vC;
    fixedCost = fC;
// Announce the unit to the console
if(StartVEFrame.verbose)
    System.out.println("Unit number " + unitNumber +
        " has been created.");
}

/** at the beginning of the 'day' each unit looks
 *  for the order(s) (if any)
 *  to be executed today; at present the
 *  production cycle is 'one order
 *  a day, but we have also to account for the
 *  use of inventories in our warehouse
 */
public void unitStep1 ()
{
    //add the beginnig of the day we have not yet

```



```

    }

    // having made no production, check to increase inventories
    if(! madeProduction && useWarehouses && useNews==false){

        if(inventoryRuleMaster.increaseInventories
            (unitNumber, myWarehouse)){
            myWarehouse.increaseInventoryCounterValue();
            //recording a variable cost
            addDailyVariableCost();
        }
    }

    // having made no production, check to news increase inventories

    if(! madeProduction && useWarehouses && useNews){
        if(inventoryRuleMaster.checkNewsToIncreaseInventories
            (unitNumber, myWarehouse, newsList)){
            myWarehouse.increaseInventoryCounterValue();
            addDailyVariableCost();
        }
    }

    // the use of inventories
    if (useWarehouses){
        while(waitingList.getCount()>0 && ! myWarehouse.empty()){
            myWarehouse.decreaseInventoryCounterValue();
            firstOrder=(Order)waitingList.removeFirst();
            firstOrder.setDoneStep();
            //set the cost to the order in production
            firstOrder.setFixedVariableOrderCost(fixedCost,

```

```

                                variableCost);

    // is the next step in our unit? If yes, assign the order
    // immediately to ourselves (if there is room in the
    // inventories we can make the production step in this
    // same cycle)
    if(getProductionPhase()==firstOrder.getNextStep())
        waitingList.addLast(firstOrder);
    // If not, put the order
    // in the daily production list
    else dailyProductionList.addLast(firstOrder);
    }
}

}

/**
 * sending the orders to the subsequent production units*/
public void unitStep2 () {
    int i;
        int x;
        int m;
    boolean operatingUnitNotFound;
        int receiverUnitNumber;
        int referringStepNumber;
        int StepsToBeDone;
        boolean acceptNews;
        saleRevenue = 0;

    if(StartVEFrame.verbose)
        System.out.println("Step 2 in unit " + unitNumber +
            " dailyProductionList count " +

```

```

        dailyProductionList.getCount());

while(dailyProductionList.getCount()>0)
{
    firstOrder=(Order)dailyProductionList.removeFirst();

    // do we have concluded the production step?
    // if yes ... we drop the object
    if (firstOrder.getNextStep()==0)
    {
        if(StartVEFrame.verbose)
            System.out.println("drop order # " +
                firstOrder.getOrderNumber() +
                " startingDay " +
                firstOrder.getStartingDay() +
                " endDay " +
                Globals.env.getCurrentTime() +
                " stepNumber " +
                firstOrder.getStepNumber() +
                " ratio " +
                ( (float)
                (Globals.env.getCurrentTime() -
                firstOrder.getStartingDay()))/
                firstOrder.getStepNumber());

        // the production is concluded, so we inform
        // our environment (the model) of the time spent
        // related to the time necessary
        vEFrameModelSwarm.setProductionTimeAndRecipeLength(

```

```

    Globals.env.getCurrentTime() -
    firstOrder.getStartingDay(),
    firstOrder.getStepNumber());
if(StartVEFrame.verbose)
    System.out.println("Order #" +
        firstOrder.orderNumber +
        " has been completed");
//remove the order dropped from the orderList
orderList.remove(firstOrder);
    //recording the revenue from the selling
    //of the finished order
    saleRevenue = firstOrder.getOrderPrice();
    //recording the cost of orders
    totalCostOrderProduced = firstOrder.
        getTotalCostForOrder();
    // now we drop the object (also to save memory space)
    firstOrder.drop();
}

// if not ... we send it to the subsequent production unit
else
{
    operatingUnitNotFound=true;
    for (i=0;i<unitList.getCount() &&
        operatingUnitNotFound;i++){
        aUnit=(Unit)unitList.atOffset(i);
        if (firstOrder.getNextStep() ==
            aUnit.getProductionPhase()){
            operatingUnitNotFound=false;

```

```

        aUnit.setWaitingList(firstOrder);
    }
}
}
}

if(waitingList.getCount()>0 && useNews)
{
    //we get the first Order of the waitingList
    //and we get how many steps still remain
    // to terminate its production
    incomingOrder=(Order)waitingList.getFirst();
    StepsToBeDone=incomingOrder.getHowManyStepsToBeDone();

    // we ask the order about the potential
    // subsequent production units
    // according to the information deepness
    // and to the InformationRuleMaster.
    // 1 is the current step, so we start
    // from next step (n°2)
    for (i=2;i<=StepsToBeDone;i++)
        if(i<=informationDeepness+1)
        {
            receiverUnitNumber=incomingOrder.
            getProductionToBeDoneAtNextStepNumber(i);
            if(StartVEFrame.verbose)

System.out.println(" Receiver Unit N°: "
+ incomingOrder.getProductionToBeDoneAtNextStepNumber(i));
            if (informationRuleMaster.checkToSendNews(unitNumber,

```

```

receiverUnitNumber)
&& unitNumber!= receiverUnitNumber)
{
    operatingUnitNotFound=true;
    //Searching Units able to make the next productionPhases
    // and sending to them news containing one order for each

for (m=0;m<unitList.getCount() && operatingUnitNotFound;m++){
    receiverUnit=(Unit)unitList.atOffset(m);
    //receiverUnit=(Unit)unitList.removeFirst();
    //unitList.addLast(receiverUnit);
    if (receiverUnitNumber==
        receiverUnit.getProductionPhase()){
        aNews=new News ( getZone(), unitNumber,
            receiverUnit.getProductionPhase(), incomingOrder);
        receiverUnit.setNewsList(aNews,
            receiverUnit.getProductionPhase());
        operatingUnitNotFound=false;
        }
    }
}

}

// at the end of the day we set the accounting data
setTotalDailyCost();
setTotalCost();
setTotalSaleRevenue();
if(useWarehouses){

```

```

        setTotalInventoryFinancialCost();
        setDailyStoredComponentValue();
    }
}

/**
 * Return the production phase of the unit
 */
public int getProductionPhase()
{
    return unitNumber;

    // at present unitNumber and production phase are
    // the same; this is a temporary situation
}

/**
 * Putting an order in the production waitingList
 * of the unit and deleting the respective news
 */
public void setWaitingList(Order anOrder)
{
    boolean newsToBeDroppedNotFound;
    int i;
    waitingList.addLast(anOrder);
    if(StartVEFrame.verbose)
        System.out.println("The unit " + unitNumber +
            " has received the order # "
            + anOrder.getOrderNumber());
    if (useNews && newsList.getCount()>0)

```



```

        //looking for a news containing information about
        // the order just arrived in the unit
        for (i=1;i<=newsList.getCount();i++)
        {
            toBeDroppedNews=(News)newsList.removeFirst();

            if (toBeDroppedNews.incomingOrd.getOrderNumber()==
                anOrder.getOrderNumber()){
                toBeDroppedNews.drop();
                if(StartVEFrame.verbose)
                    System.out.println("Drop news containig order n°"+
                        toBeDroppedNews.incomingOrd.getOrderNumber());
            }
            else

                newsList.addLast(toBeDroppedNews);
        }
    }

    /**
     * Returning the waiting list length
     */
    public int getWaitingListLength()
    {
        return waitingList.getCount();
    }

    /**
     * Putting a news in the newsList of the unit after
     * checking list to avoid assimilable news
     */

```

```

    public void setNewsList(News aNews, int productionPhase)
    {
        int i;
        int prodPhase;
        boolean notAcceptedNews;
        // a single news of the newsList
        News anotherNews;
        // the news just arrived
        News toBeCheckedNews;
        prodPhase=productionPhase;
        toBeCheckedNews=aNews;
        notAcceptedNews=false;

        if (newsList.getCount(>0)){
            for (i=1;i<newsList.getCount() &&
                    notAcceptedNews==false;i++){
                anotherNews=(News) newsList.removeFirst();
                newsList.addLast(anotherNews);
                if (toBeCheckedNews.incomingOrd.getOrderNumber()==
                    anotherNews.incomingOrd.getOrderNumber()
                    && toBeCheckedNews.destinationUnitActivatedInStepN==
                    anotherNews.destinationUnitActivatedInStepN)
                    notAcceptedNews=true;
            }

            if (notAcceptedNews==false){
                newsList.addLast(toBeCheckedNews);
                if(StartVEFrame.verbose)
                    System.out.println("Unit " + prodPhase +
                        " accept a news containing order N° ")
            }
        }
    }

```

```

        + toBeCheckedNews.incomingOrd.getOrderNumber());
    }

    else

        if(StartVEFrame.verbose)

            System.out.println("Unit " + prodPhase +
" doesn't accept a news containing order N° "
+ toBeCheckedNews.incomingOrd.getOrderNumber()+
" because referring to same
destinationUnitActivatedInStepN");
        }

        else{

            newsList.addLast(toBeCheckedNews);

            if(StartVEFrame.verbose)

                System.out.println("Unit " + prodPhase +
" accepts a news containing order N° "
+ toBeCheckedNews.incomingOrd.getOrderNumber()+
" because newsList is empty");
        }
    }

}

/** method to increase the value of daily variable cost*/
public void addDailyVariableCost(){

    totalDailyVariableCost += variableCost;

}

/** method to increase the sum of fixed and variable cost*/
public void setTotalDailyCost(){

    totalDailyCost = fixedCost + totalDailyVariableCost;

}

/** the method to cumulated the total daily cost*/

```

```
public void setTotalCost(){
    totalCost += totalDailyCost;
}

public int getTotalDailyCost(){
    return totalDailyCost;
}

/** the return of totalCost */
public int getTotalCost(){
    return totalCost;
}

/** the method to record the generation of a revenue
 * from a dropped order */
public double getSaleRevenue(){
    return saleRevenue;
}

/** the method to record the generation of a revenue
 * from a dropped order */
public void setTotalSaleRevenue(){
    totalSaleRevenue += saleRevenue;
}

/** the method to return the totalRevenue */
public double getTotalSaleRevenue(){
    return totalSaleRevenue;
}

/** the method to record the generation of a cost
 * from a dropped order */
public int getTotalCostOrderProduced(){
    return totalCostOrderProduced;
}
```

```
/** the method to record the cumulated costs of semi
 * manufactured stored in warehouses */
public void setTotalInventoryFinancialCost(){
    totalInventoryFinancialCost +=
        (myWarehouse.getInventoryCounterValue()*
         (vEFrameModelSwarm.getInventoryFinancialRate()/200));
}

/** method to return the financial cost of warehouses */
public double getTotalInventoryFinancialCost(){
    return totalInventoryFinancialCost;
}

/** method to record the daily value of components stored
 * in warehouses */
public void setDailyStoredComponentValue(){
    if(
        vEFrameModelSwarm.getInventoryEvaluationCriterion()==3)
        dailyStoredComponentValue =
            myWarehouse.getInventoryCounterValue()*
            vEFrameModelSwarm.getRevenuePerEachRecipeStep();

    else if(
        vEFrameModelSwarm.getInventoryEvaluationCriterion()==2)
        dailyStoredComponentValue =
            myWarehouse.getInventoryCounterValue()*
            (fixedCost + variableCost);

    else if(
        vEFrameModelSwarm.getInventoryEvaluationCriterion()==1)
        dailyStoredComponentValue =
```

```

myWarehouse.getInventoryCounterValue()*
variableCost;

else if(
    vEFrameModelSwarm.getInventoryEvaluationCriterion()!=1 &&
    vEFrameModelSwarm.getInventoryEvaluationCriterion()!=2 &&
    vEFrameModelSwarm.getInventoryEvaluationCriterion()!=3)
{
    System.out.println("your criterion doesn't exist:
the variable"+
    " inventoryEvaluationCriterion must be included
between 1 and 3");
    System.exit(0);
}
}

public double getDailyStoredComponentValue(){
    return dailyStoredComponentValue;
}
}

```

D.2.5 Order.java

```

import swarm.Globals; import swarm.defobj.Zone; import
swarm.objectbase.SwarmObjectImpl; /**
 * The form containing our Order metaphoric object
 *
 * @author Pietro Terna
 */

```

```

public class Order extends SwarmObjectImpl{

    /** the number of steps in the product recipe */
    public int stepNumber;
    /** the number of this order */
    public int orderNumber;
    /** the day in which the order as been generated */
    public int startingDay;
    /** integer vector, containing the recipe steps */
    public int[] orderRecipe;
    /** integer vector, recording the performed phases */
    public int[] orderState;
    /** the sequential number of the recipe referring to
        * wich a news will be sent*/
    private int destinationUnitActivatedInStepN;
    /** the fixed and variable costs of orders */
    public int cumulatedFixedCost, cumulatedVariableCost;
    /** the sum of variable and fixed costs of orders */
    public int totalCostForOrder = 0;
    /** the model (to which we apply mark up of
        * the order revenue */
    VEFrameModelSwarm vEFrameModelSwarm;

    /**
    * constructor for Order
    */
    public Order (Zone aZone, int n, int sd, int sn,
        int [] r, VEFrameModelSwarm model) {

```

```

// Call the constructor for the Order parent class.
super(aZone);

orderNumber=n;
stepNumber=sn;
startingDay=sd;
orderRecipe= new int[stepNumber];
orderState = new int[stepNumber];
// the model
vEFrameModelSwarm = model;

int i;

for (i=0;i<stepNumber;i++)orderRecipe[i]=r[i];

// announcing the order presence to the console
if(StartVEFrame.verbose)
System.out.println("New order (#" + orderNumber + ")");
// setting the state vector (0=empty; 1=done)
if(StartVEFrame.verbose)System.out.print("orderState
vector ");

for (i=1;i<=stepNumber;i++) {
orderState[i-1]=0;
if(StartVEFrame.verbose)System.out.print(orderState[i-1]
+ " ");
}

if(StartVEFrame.verbose)System.out.println();
// announcing the recipe vector
if(StartVEFrame.verbose)System.out.print(

```



```

        "orderRecipe vector");
    for (i=1;i<=stepNumber;i++)
    if(StartVEFrame.verbose)System.out.print(
        orderRecipe[i-1]+" ");
    if(StartVEFrame.verbose)System.out.println();
}
/**
 * production next step in Order
 * (it is in the orderRecipe vector,
 * at the same index value in which we find the first
 * 0 in orderState vector)
 */
public int getNextStep () {
    int i, stepN;
    stepN=0;
    for (i=0;i< stepNumber && stepN==0;i++)
        if (orderState[i]==0)stepN=orderRecipe[i];
    return stepN;
}
/**
 * Done next step in Order
 */
public void setDoneStep () {
    int i, done;
    done=0;
    for (i=0;i<= stepNumber && done==0;i++)
        if (orderState[i]==0)
            {orderState[i]=1;
            done=1;

```

```
        }

    }

    /**
     * returning orderNumber of this order
     */
    public int getOrderNumber () {
        return orderNumber;
    }

    /**
     * returning the stepNumber
     */
    public int getStepNumber () {
        return stepNumber;
    }

    /**
     * returning the startingDay
     */
    public int getStartingDay () {
        return startingDay;
    }

    /**
     * returning how many steps remain to terminate
     * the production
     */
    public int getHowManyStepsToBeDone ()
{
    int i, stepsToBeDone;
    for (i=0;i< stepNumber && orderState[i]==1;i++);
    stepsToBeDone=stepNumber-i;
}
```

```

        if(StartVEFrame.verbose)
System.out.println("stepsToBeDone:  " + stepsToBeDone+
                    " ORDER N°"+ orderNumber);

return stepsToBeDone;
}

/**
 * returning the production phase at a single step
 */

public int getProductionToBeDoneAtNextStepNumber (int SN) {

int i;
int productionToBeDone;
int StepToBeDone;
    StepToBeDone= SN;
for (i=0;i< stepNumber && orderState[i]==1;i++);

destinationUnitActivatedInStepN=i+StepToBeDone-1;
productionToBeDone=orderRecipe[destinationUnitActivatedInStepN];

return productionToBeDone;
}

/**
 * returning the sequential number of the recipe
 * referring to wich a news will be sent
 */

public int getDestinationUnitActivatedInStepN () {

return destinationUnitActivatedInStepN;

```

```
}

/** returning the value of the production already done */
public double getOrderPrice(){
    int sumOrderState = 0;
    double price = 0;
    for(int i=0; i<orderState.length; i++){
        sumOrderState += orderState[i];
    }
    price = sumOrderState *
        vEFrameModelSwarm.getRevenuePerEachRecipeStep();
    return price;
}

/** the method to set the fixed and variable orders costs
 *  in production or finished */
public void setFixedVariableOrderCost(int fC, int vC){
    cumulatedFixedCost += fC;
    cumulatedVariableCost += vC;
    totalCostForOrder = (cumulatedFixedCost +
                        cumulatedVariableCost);
}

/** the method to get the total of cost of the order */
public int getTotalCostForOrder(){
    return totalCostForOrder;
}
}
```

D.2.6 PTHistogram.java

```
/**
 * PTHistogram contains three constructors to develop
 * histogram graphs
 * @author Dario Landini
 */

import swarm.Globals;

import swarm.collections.ListImpl;

import javax.swing.JFrame;

import ptolemy.plot.Histogram;

public class PTHistogram extends Histogram{

    public boolean useWarehouses;

    private JFrame myFrame = null;

    /** The Lists passed through the constructors*/

    ListImpl fL; ListImpl sL; ListImpl tL;

    /** the temporary address of an existing unit */

    Unit unit;
```

```

/** the temporary address of warehouses, if any */

Warehouse warehouse ;

/**
 * First class constructor
 * @param title the title of the histogram
 * @param xLabel the label on the X axis
 * @param yLabel the label on the Y axis
 * @param xRange the right dimension of the histogram
 * @param yRange the top dimension of the histogram
 * @param firstList a data list
 */
public PTHistogram(String title, String xLabel,
                   String yLabel, double xRange, double yRange,
                   ListImpl firstList) {

    //create the Frame for the histogram graph
    myFrame = new JFrame();
    //establish the title of graph with ptolemy
    setTitle(title);
    //pass the same title to the window's frame
    myFrame.setTitle(title);
    //establish the axes' labels with ptolemy
    setXLabel(xLabel);
    setYLabel(yLabel);
    //the lenght of x & y axes
    setXRange(0,xRange);
    setYRange(0,yRange);

```

```
//the right position of bars their widht
setBinOffset(1);
setBinWidth(0.5);
//the legend
addLegend(0, "Unit");

fL = firstList;
//create the "fill" button to resize graph
setButtons(true);
//fix the window's dimensions according
//to x & y range
int windowHeight = 400;
int windowHigh = 300;
    if(xRange > 18){
        windowHeight = 700;
    }
    if(xRange<5){
        windowHeight = 300;
    }
myFrame.setBounds(10,300,windowWidth,windowHigh);

//add the histogram to the frame
myFrame.getContentPane().add(this);
//show the frame
myFrame.show();
}

/**
 * Second class constructor
```

```
* @param title the title of the histogram
* @param xLabel the label on the X axis
* @param yLabel the label on the Y axis
* @param xRange the right dimension of the histogram
* @param yRange the top dimension of the histogram
* @param firstList a first data list
* @param secondList a second data list
*/
public PTHistogram(String title, String xLabel,
                    String yLabel, double xRange,
                    double yRange, ListImpl firstList,
                    ListImpl secondList) {

    useWarehouses = true;
    // if we are using this constructor the
    // warehouses are activated
    myFrame = new JFrame();
    setTitle(title);
    myFrame.setTitle(title);

    setXLabel(xLabel);
    setYLabel(yLabel);
    setXRange(0,xRange);
    setYRange(0,yRange);

    setBinOffset(1);
    setBinWidth(0.5);

    addLegend(0, "Unit");
```



```
        addLegend(1, "WareHouse");

        fL = firstList;
        sL = secondList;

        setButtons(true);

        int windowHeight = 400;
        int windowHigh = 300;
        if(xRange > 18){
            windowHeight = 700;
        }
        if(xRange<5){
            windowHeight = 300;
        }
        myFrame.setBounds(10,300,windowWidth,windowHigh);

        myFrame.getContentPane().add(this);
        myFrame.show();
    }

/**
 * Third class constructor
 * @param title the title of the histogram
 * @param xLabel the label on the X axis
 * @param yLabel the label on the Y axis
 * @param xRange the right dimension of the histogram
 * @param xRange the top dimension of the histogram
 * @param firstList a first data list
```

```
* @param secondList a second data list
* @param thirdList a third data list
*/

public PTHistogram(String title, String xLabel,
                   String yLabel, double xRange,
                   double yRange, ListImpl firstList,
                   ListImpl secondList,
                   ListImpl thirdList) {

    myFrame = new JFrame();
    setTitle(title);
    myFrame.setTitle(title);

    setXLabel(xLabel);
    setYLabel(yLabel);
    setXRange(0,xRange);
    setYRange(0,yRange);

    setBinOffset(1);
    setBinWidth(0.5);

    addLegend(0, "Unit");
    addLegend(1, "WareHse");
    addLegend(2, "");

    fL = firstList;
    sL = secondList;
    tL = thirdList;
```

```
        setButtons(true);

        int windowHeight = 400;
        int windowHigh = 300;
        if(xRange > 18){
            windowHeight = 700;
        }
        if(xRange<5){
            windowHeight = 300;
        }
        myFrame.setBounds(10,300,windowWidth,windowHigh);

        myFrame.getContentPane().add(this);
        myFrame.show();
    }

    /**
     * this method should be called by the Observer
     * to paint histogram bars,at every schedule.
     * It paints a point for each number contained
     * in the lists of the single object.
     */
    public void addPoints(){
        //it clears graph to repaint it, but
        //it doesn't forget the window instructions
        clear(false);
        //to scroll the first list in all its lenght
        for (int i=0;i<fL.getCount();i++){
```

```

        //put in unit the object of the first list imported
        unit = (Unit) fL.atOffset(i);
        //to transform an int into a string
        Integer f = (new Integer (unit.unitNumber));
        String singleLabels = f.toString();
        //add the labels on x axis
        addXTick(singleLabels , i);

        if(unit.getWaitingListLength() != 0){
            //addPoint in according to the waitingListLength
            for(int j=0;j<unit.getWaitingListLength();j++){
                addPoint(0, (double) i);
            }
        }
    }

    if(useWarehouses){
        //to scroll the second list in all its lenght
        for (int i=0;i<sL.getCount();i++){
            //put in warehouse the object of the second list imported
            warehouse = (Warehouse) sL.atOffset(i);
            if(warehouse.getInventoryCounterValue() != 0){
                //addPoint in according to the getInventoryCounterValue()
                for(int j=0;j<warehouse.getInventoryCounterValue();j++){
                    addPoint(1, (double) i);
                }
            }
        }
    }
}

```

```

        //repaint the window at each time and
        //execute clear instruction
        repaint();
    }
}

```

D.2.7 Accounting.data.java

```

/**
 * This class is done to read the data of variable and
 * fixed in input of each unit
 *
 * @author Dario Landini and Nicola Ormezzano
 */
import java.io.*;

public class AccountingData {

    /** the vector of fixed cost of units */
    public int[ ] fixedCost;

    /** the vector of variable cost of units */
    public int[ ] variableCost;

    /** the number of the units to size the vectors */
    public int totalUnitNumber;

    /**the constructor of AccountingData, with a parameter
     * to obtain the total unit number from the modelSwarm
     */
    public AccountingData(int tUN) {

```

```
        totalUnitNumber = tUN;
    }

    /** the method to read the fixed data from FixedCost.txt */
    public void readFixedData(){
        try{
            try{
                fixedCost = new int[totalUnitNumber];
                //read the file
                FileInputStream fileInputFixedCost =
                    new FileInputStream("InputCost/FixedCost.txt");
                //put the file in the buffer
                BufferedInputStream buffInputFixedCost =
                    new BufferedInputStream(fileInputFixedCost);
                //read data from buffer
                DataInputStream dataInputFixedCost =
                    new DataInputStream(buffInputFixedCost);

                for(int i=0; i<fixedCost.length; i++){
                    String line = dataInputFixedCost.readLine();
                    fixedCost[i] = Integer.valueOf(line).intValue();
                }

            } catch (FileNotFoundException fnfe){
                System.out.println(fnfe);
                System.exit(0);
            }

        }catch(IOException e){
```

```
        System.out.println("IOException:" + e.toString());
    }
}

/** the method to read the variable data
 * from VariableCost.txt */
public void readVariableData(){
    try{
        try{
            variableCost = new int[totalUnitNumber];
            //read the file
            FileInputStream fileInputVariableCost =
                new FileInputStream("InputCost/VariableCost.txt");
            //put the file in the buffer
            BufferedInputStream buffInputVariableCost =
                new BufferedInputStream(fileInputVariableCost);
            //read data from buffer
            DataInputStream dataInputVariableCost =
                new DataInputStream(buffInputVariableCost);

            for(int j=0; j<variableCost.length; j++){
                String line = dataInputVariableCost.readLine();
                variableCost[j] = Integer.valueOf(line).intValue();
            }

        } catch (FileNotFoundException fnfe){
            System.out.println(fnfe);
            System.exit(0);
        }
    }
}
```

```
    }catch(IOException e){
        System.out.println("IOException:" + e.toString());
    }
}

/** returning the vector of fixed costs */
public int getFixedCost(int i){
    return fixedCost[i-1];
}

/** returning the vector of variable costs */
public int getVariableCost(int j){
    return variableCost[j-1];
}
}
```