

UNIVERSITÁ DEGLI STUDI DI TORINO
FACOLTÁ DI ECONOMIA
CORSO DI LAUREA IN ECONOMIA AZIENDALE

TESI DI LAUREA

**SIMULAZIONE DI UN'IMPRESA REALE UTILIZZANDO IL
MODELLO jES: IL CASO VIR**

Relatore:

Prof. PIETRO TERNA

Correlatore:

Prof. SERGIO MARGARITA

Candidato:
ELENA BONESSA

Anno Accademico 2001 - 2002

*Alla mia famiglia che mi ha sempre
sostenuto e a Sirio mia grande gioia.*

Ringrazio l'ing. Savino Rizzio, fondatore della Vir S.p.a., ed il figlio ing. Giovannibattista Rizzio, che ci hanno accolto nella loro azienda concedendoci l'accesso ai dati aziendali ed alle informazioni che sono state necessarie al completamento di questo lavoro.

Ringrazio inoltre l'ing. Gervasini, che ha fornito gli elementi relativi all'organizzazione dell'impresa da un punto di vista gestionale ed il sig. Camana, per la sua disponibilità ad illustrarci le problematiche tecniche di gestione della produzione e dei macchinari e per averci guidato nella visita dello stabilimento.

Indice degli argomenti

Introduzione	4
Capitolo 1 Il concetto di impresa	6
1.1 La visione neoclassica dell'impresa	6
1.2 Mises, Hayek e l'impostazione Austriaca	10
1.3 L'imprenditore di Kirzner	12
Capitolo 2 La simulazione	16
2.1 Simulazione e realtà	17
2.2 Simulazione e complessità	18
2.3 Simulazione e conoscenza	20
2.4 La conoscenza tacita	22
2.5 Micromondi e sistemi dinamici	24
Capitolo 3 Strumenti di simulazione	34
3.1 La programmazione ad oggetti (OOP)	34
3.2 Il linguaggio Java	38
3.3 Swarm	44
3.4 Il modello jES: java Enterprise Simulator	47
Capitolo 4 NIIP e la collaborazione fra imprese	62
4.1 Il progetto NIIP	63
4.2 L'architettura NIIP	65
4.3 I distretti industriali	71
Capitolo 5 Applicazione del modello JES al caso pratico VIR . . .	76
5.1 Una realtà aziendale: VIR	76
5.2 Formalizzazione per il modello JES	89
5.3 Sviluppo del codice	106
5.4 Appendice	118
5.4.1 Dettaglio articoli	119

5.4.2	Analisi delle fasi di produzione	129
5.4.3	File javadoc delle classi Order Distiller e Recipe	133
Capitolo 6	Revisione della prima formalizzazione del modello . . .	146
6.1	Sviluppi inerenti agli ordini ed alle unità	146
6.2	Tempo e quantità	147
6.3	Risultati ottenuti e problemi ancora aperti	149
Capitolo 7	UML	150
7.1	La notazione UML	150
7.2	La struttura	154
7.3	Utilizzo dell'UML nei processi di sviluppo	171
7.4	I meccanismi generali e le eccezioni	174
Bibliografia	177
Appendice - Il codice jES-VIR	181

Introduzione

Simulazione d'impresa significa rappresentare una realtà aziendale nel computer, mediante l'applicazione di un modello informatico. Nel nostro caso si tratta di un modello che utilizza la programmazione ad oggetti e simula sistemi complessi non dall'alto, mirando a risultati predeterminati, ma dal basso, descrivendo le singole parti ed osservandone le interazioni nel tempo.

La simulazione ha due principali scopi, uno teorico ed uno pratico: il primo è simulare il processo di scoperta imprenditoriale attraverso un percorso di continuo adattamento ed innovazione dell'impresa virtuale, il secondo è quello di poter disporre di un campo di prova sul quale poter applicare le scelte necessarie al miglioramento dell'impresa, come se si stesse agendo nella realtà.

Il primo capitolo si richiama al contributo della scuola Austriaca che offre l'opportunità di riflettere sul significato di impresa e di imprenditore, dall'imprenditore-funziionario neoclassico all'imprenditore-scopritore di Kirzner.

Il secondo capitolo tratta della simulazione come nuovo strumento, che si aggiunge a quelli tradizionali con cui la scienza cerca di conoscere e di capire la realtà. Con la simulazione si potranno scoprire gli effetti e soluzioni non individuati in precedenza.

L'obiettivo del terzo capitolo è quello di introdurre le nozioni alla base del paradigma Object Oriented con particolare riguardo al linguaggio Java ed alle librerie Swarm. Tale percorso si conclude con la descrizione del modello jES (java Enterprise Simulator), un frame in java-swarm, utilizzato per l'applicazione della simulazione a contesti di impresa oggetto di questa dissertazione.

Il quarto capitolo è relativo al progetto NIIP (National Industrial Information Infrastructure Protocols consortium) che estende l'analisi dell'impresa virtuale a sistemi di imprese che interagiscono scambiandosi informazioni e costruendo opportunità che singolarmente non avrebbero saputo cogliere. Questa struttura richiama quella dei distretti industriali, trattati in conclusione del capitolo.

Il quinto capitolo descrive il percorso per passare da una azienda reale ad un modello per la simulazione. Si analizza d'apprima la realtà aziendale VIR e successivamente la struttura ed il funzionamento del modello javaSwarm di impresa virtuale, utilizzato come base per il modello jES-VIR.

Il sesto capitolo tratta delle continue revisioni necessarie alla formalizzazione di un sistema complesso come può essere quello della realtà aziendale descritta nel capitolo precedente. La metodologia utilizzata è stata la continua sperimentazione, con esempi anche semplici, per comprendere i meccanismi sottostanti al modello.

Il settimo ed ultimo capitolo fornisce una panoramica generale sull'UML (Unified Modeling Language): chiarisce che cosa si intende per UML e quali siano le relative aree di applicazione e gli obiettivi. L'interesse per questo strumento scaturisce dalla ricerca di migliorare la progettazione e l'uso di jES attraverso l'adozione di una schematizzazione standard, adattabile a diversi ambienti.

Capitolo 1 – Il concetto di impresa

Il contributo della scuola Austriaca offre l'opportunità di riflettere sul significato di impresa e di imprenditore che subisce un notevole sviluppo, dall'imprenditore-“funzionario” neoclassico all'imprenditore-scopritore di Kirzner.

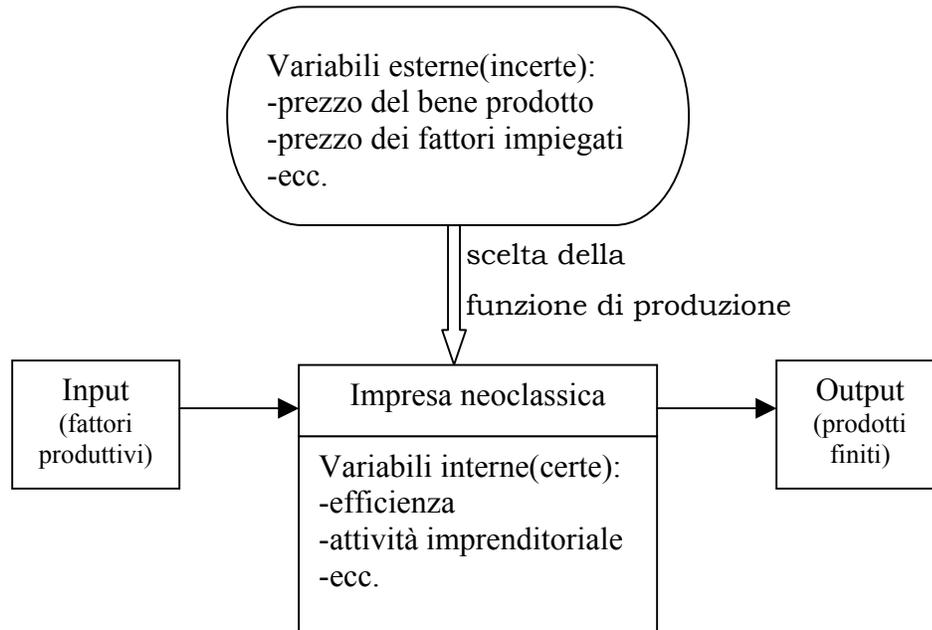
L'analisi delle diverse teorie mira a determinare se l'impresa, mutata dai continui avanzamenti tecnologici e sottoposta a diverse ricerche e simulazioni nel campo delle scienze sociali, possa ancora essere interpretata dal paradigma classico.

1.1 La visione neoclassica dell'impresa

Sin dal XX secolo, e soprattutto negli ultimi cinquant'anni, la visione neoclassica della microeconomia ha assunto un ruolo importantissimo nell'analisi economica. Questo grazie alla notevole semplicità dei suoi concetti che racchiudono una visione meccanicistica dell'economia secondo cui un problema può essere espresso sotto forma di una funzione obiettivo la quale, nel rispetto dei vincoli e dei gradi di libertà, dovrà essere massimizzata.

A questo modello di “perfezione” può essere applicato il concetto di impresa che, secondo la visione neoclassica (Colombatto 2001), è costituita da un insieme di fattori organizzati al fine di produrre e commercializzare un bene che possa essere venduto a un prezzo almeno pari alla somma delle remunerazioni corrisposte alle risorse impiegate.

Grazie alla funzione di produzione fissata e conosciuta da tutti l'impresa diventa una macchina che trasforma gli input in output. Essa è influenzata da variabili interne del tutto certe e determinate e da variabili esterne che rappresentano l'unica fonte di incertezza come ad esempio il prezzo dei beni.



La figura dell'imprenditore risulta assimilabile ad un funzionario organizzatore-controllore, che sceglie di volta in volta la funzione di produzione più adatta a realizzare l'obiettivo del "modello", ma non indirizza le sue potenzialità alla scoperta di nuovi bisogni da soddisfare.

Nel sistema economico esistono le asimmetrie informative ovvero i diversi operatori sono in possesso di informazioni diverse e quindi sono sottoposti a differenti gradi di rischio.

Alcuni studiosi sostengono che la concezione neoclassica di impresa escluda il concetto di rischio e di conseguenza di asimmetrie informative, in questo modo non potrebbero esistere imprenditori che grazie alle loro qualità organizzative risultino migliori di altri. Non vi sarebbe inoltre distinzione fra equilibrio di breve e di lungo periodo in quanto, nessun imprenditore avrebbe mai l'esigenza di modificare le proprie decisioni nel corso del tempo.

Gli esponenti della scuola Austriaca sostengono che nel contesto neoclassico l'elemento incertezza che caratterizza l'attività d'impresa potrebbe essere, in teoria, eliminabile poiché l'imprenditore avrebbe la possibilità di conoscere le distribuzioni di probabilità legate agli eventi aleatori che a lui interessano.

Va considerato però, che avere accesso alle informazioni non significa averlo senza costi e dunque se il costo dell'acquisizione delle informazioni diventa superiore al beneficio che si trae da queste (decisioni più accurate) non sarà più conveniente essere totalmente informati.

Al fine di creare imprese neoclassiche differenziate occorre che almeno una delle seguenti due condizioni sia verificata:

1. gli imprenditori dovrebbero essere differenziati fra loro, e avere quindi percezioni diverse in merito al beneficio atteso generato dall'acquisto dell'informazione marginale;
2. a parità di spesa devono corrispondere insiemi di informazioni distinte. In tale caso gli imprenditori lavorerebbero con funzioni di produzione diverse, a seconda dell'informazione di cui dispongono, fino a quando i peggiori non correggeranno i propri errori osservando e copiando i migliori.

L'incertezza non si riferisce solo alle variabili esogene, ma può anche riguardare l'impiego delle risorse produttive le quali possono non coincidere con le attese.

L'imprenditore neoclassico è sottoposto dunque a due tipi di rischio:

- rischio interno : scelta dei fattori produttivi;
- rischio esterno : carenza di informazioni sulle variabili esterne.

L'equilibrio neoclassico si basa sul concetto di ottimo paretiano, l'interazione fra le imprese porterà alla concorrenza perfetta, se

consideriamo prodotti omogenei, o alla concorrenza monopolistica, se invece i prodotti sono differenziati.

Il profitto, differenza fra prezzo e costo medio, in un mercato concorrenziale tende ad annullarsi, ma questo non sempre accade.

Se vi è incertezza, e gli imprenditori sono differenziati fra loro, si possono verificare profitti positivi.

Per quanto riguarda il medio-lungo periodo questo non vale perché gli operatori meno efficienti tenderanno ad imitare gli operatori migliori, con la conseguenza di erodere i profitti. Solo fattori esogeni, quali nuove opportunità offerte dalla tecnologia, potrebbero portare a nuovi fenomeni di differenziazione. Quest'ultima visione equivale però a frazionare il lungo periodo in tanti brevi periodi e non a considerarlo interamente.

L'imprenditore neoclassico si limita a scegliere la funzione di produzione più adatta e non può commettere errori. Egli opera in un regime concorrenziale, che determina una situazione stazionaria, turbata solo da un eventuale shock esterno impreveduto.

Alcuni studi di impronta neoclassica, della metà degli anni ottanta, hanno rielaborato e arricchito il concetto di impresa classico.

Nella letteratura neoclassica recente l'imprenditore, pur non essendo visto come innovatore, è una sorta di negoziatore walrasiano che si occupa dell'assegnazione ottima dei diritti di proprietà e della riduzione delle asimmetrie informative e dei costi di transazione.

E' già stato spiegato sopra il significato di asimmetria informativa, mentre i costi di transazione sono quei costi sostenuti per effettuare e gestire gli scambi.

Proprio l'opportunità di ridurre tali costi permette di valutare la struttura del mercato, non più attraverso le barriere all'entrata bensì, attraverso le dimensioni ottime dell'impresa.

Gli incrementi di benessere, secondo tale linea di pensiero, non avvengono grazie alla capacità imprenditoriale né, tanto meno, grazie alla potenziale entrata nel mercato di nuovi operatori ma per mezzo di quattro fattori:

- 1) scambio efficiente;
- 2) impiego ottimo dei fattori disponibili dati;
- 3) miglioramento dell'assegnazione dei diritti di proprietà fra imprese note;
- 4) un più efficace sistema di incentivi e di monitoraggio all'interno delle imprese.

Tali elementi condividono l'assenza di interazione fra le imprese già presenti sul mercato e quelle entranti, ma i primi due punti rispecchiano la teoria neoclassica tradizionale e gli altri due rispecchiano le teorie più recenti.

1.2 Mises, Hayek e l'impostazione Austriaca

Lo schema neoclassico è stato messo in discussione relativamente a due profili principali: il profilo normativo e il profilo positivo.

Con riferimento agli aspetti normativi si fa chiaramente notare che l'intervento pubblico, attuato da un decisore pubblico, tende a massimizzare l'interesse personale di questi piuttosto che perseguire il benessere collettivo: questo accade proprio perché si opera secondo modalità neoclassiche.

E' vero che tale interesse personale potrebbe coincidere con quello della collettività, ma ciò accade raramente perché è molto più probabile che un intervento pubblico sia a favore di gruppi di pressione relativamente ristretti.

Gli errori dell'operatore privato hanno un limite nel profitto negativo, mentre per l'operatore pubblico tale limite è molto più ambiguo.

E' necessario dunque porre la massima cautela nel creare operatori nuovi quali l'imprenditore pubblico e il politico con funzioni redistributive.

Per quanto riguarda il profilo positivo, la critica più incisiva allo schema neoclassico è stata senza dubbio quella Austriaca.

A partire dai fondatori stessi della Scuola, per continuare poi con Mises e Hayek misero in discussione il concetto neoclassico di economia di mercato, di concorrenza e di equilibrio concorrenziale.

Secondo Mises lo studio dell'economia consiste nell'analisi delle possibilità, in condizioni di incertezza, di allentare i vincoli tipici dell'attività economica (scarsità delle risorse a disposizione dei produttori) e di accrescere la soddisfazione dei consumatori. Quest'ultimo obiettivo può essere raggiunto aumentando il potere di acquisto dei consumatori e offrendo loro l'opportunità di acquistare nuovi beni e servizi, in grado di soddisfare meglio le loro esigenze.

Come si può notare tale concezione differisce dalla prospettiva neoclassica, la quale si basa sulla capacità di creare in modo automatico una configurazione caratterizzata da un utilizzo ottimo delle risorse disponibili: in sintesi si esaminano le condizioni di ottimo.

Un concetto che accomuna Mises e Hayek è quello secondo cui lo scopo dell'economia non è l'analisi della razionalità in una situazione nota di scarsità, bensì l'analisi delle modalità attraverso le quali l'individuo può rendere meno gravosi i vincoli di scarsità e migliorare la propria condizione per il futuro.

Secondo Hayek l'equilibrio denoterebbe una situazione in cui gli agenti nutrono attese del tutto corrette nei confronti dei comportamenti degli altri individui. Il punto essenziale non è il raggiungimento dell'equilibrio, ma l'individuazione dei meccanismi attraverso i quali gli squilibri possano correggersi automaticamente e con maggior rapidità.

Secondo gli Austriaci, la superiorità dell'economia di mercato va individuata nella sua capacità di creare incentivi tali affinché gli imprenditori si ingegnino a superare i vincoli imposti dalle funzioni di produzione esistenti. Si potrebbero sperimentare nuovi processi e nuovi prodotti, per raggiungere mercati ove soddisfare a condizioni più vantaggiose la domanda esistente.

Con tale impostazione non si esclude a priori la presenza pubblica, infatti qui la libertà di intraprendere non viene intesa come anarchia o arbitrio di calpestare la libertà altrui. E' anche vero però che si valutano con cautela quegli interventi pubblici che non sono giustificati dall'esigenza di garantire le libertà individuali fondamentali, e che potrebbero limitare o distorcere gli stimoli al miglioramento.

1.3 L'imprenditore di Kirzner

Kirzner ha una visione dell'imprenditore totalmente innovativa rispetto alle precedenti. Citando Colombatto (2001):

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

Va precisato che, secondo tale concezione, l'attività imprenditoriale consiste nella capacità di vedere realtà che, seppur sono già presenti,

non sono state ancora percepite come possibilità di crescita e di benessere. L'atto imprenditoriale non è invenzione di ciò che non c'era in precedenza o che esisteva ma non era noto, bensì, ad esempio, può riscontrarsi in innovazioni di prodotto o di processo che mettono in evidenza la capacità di combinare in modo diverso fattori produttivi noti.

Da tutto questo scaturisce anche una nuova definizione di profitto.

Kirzner per profitto non intende più il compenso che permette di remunerare le capacità esecutive e organizzative dell'imprenditore e nemmeno la remunerazione del rischio di impresa legata ai fattori produttivi capitale e lavoro.

Il concetto di profitto è legato al concetto di atto imprenditoriale, la cosiddetta "scoperta imprenditoriale", che distingue l'imprenditore-scopritore Kirzneriano dall'imprenditore-massimizzatore neoclassico.

Il conseguimento del profitto avviene, infatti, precedentemente alla realizzazione dell'attività di impresa, in quanto si identifica con la remunerazione dovuta all'abilità dell'imprenditore di percepire gli errori altrui e le collegate opportunità di miglioramento.

Per Kirzner, l'imprenditore puro è dunque privo di mezzi di produzione, poiché l'atto imprenditoriale non consiste nell'organizzare e attivare le risorse, ma nell'individuare le opportunità. La scoperta imprenditoriale nasce dal nulla; se ciò non fosse, la remunerazione di tale scoperta – il profitto – sarebbe, di fatto, remunerazione delle risorse utilizzate.

Nell'economia di mercato di Kirzner l'equilibrio è uno stato in cui le azioni degli operatori sono prevedibili e previste: una configurazione di statica, caratterizzata dall'assenza dell'attività imprenditoriale, e dove nessun agente ha notato occasioni di profitto e quindi opportunità di crescita.

I neoclassici vedono la situazione di equilibrio come il “migliore di quelli possibili”, per loro dunque ha un’accezione positiva e non sono incentivati ad allontanarsi da tale situazione.

Kirzner e la scuola Austriaca, percepiscono la situazione di equilibrio come una fase di “stallo” che definisce un contesto economico privo di vivacità, è ovvio che in questo caso l’equilibrio ha una connotazione negativa.

Lo squilibrio, e dunque la presenza di profitti, è invece considerato un segnale di vitalità imprenditoriale, di crescita e miglioramento per il benessere collettivo. La dinamica imprenditoriale prende il via da una situazione squilibrata, la quale consiste di opportunità che l’imprenditore coglie prima di altri, e dà origine ad attività produttive che riducono lo squilibrio. Più intenso sarà lo squilibrio, maggiori saranno le occasioni e le prospettive di progresso.

Con questo, naturalmente, non si intende dire che ci si debba adoperare per creare a tutti i costi situazioni di squilibrio, o che un settore con maggiori scompensi sia migliore di un settore più regolare. Si vuole solo evidenziare la maggiore possibilità di svolgere attività imprenditoriali, volte al profitto ed alla crescita, che caratterizza il primo settore rispetto al secondo.

Anche il concetto di concorrenza cambia da quello neoclassico: la concorrenza non è più solo la presenza di pari opportunità e l’assenza di rendite di posizione, ma è anche libertà di allontanarsi dalla situazione di equilibrio alla ricerca di profitti latenti; evenienza pressoché inconcepibile nel mondo neoclassico.

Concorrenza per Kirzner non significa solo libertà di accesso ai mercati dei beni, dei fattori, delle tecnologie, ma anche, e soprattutto, libertà di appropriarsi dei frutti della “scoperta”, dai profitti generati dalla capacità di non ripetere gli errori altrui.

La trattazione del concetto di impresa subisce una forte evoluzione dalla visione neoclassica alla riscoperta del ruolo dell'imprenditorialità.

La scoperta dell'imprenditore tradizionale si basa sul concetto di *ignoranza razionale*: trovare qualcosa che si presume che esista, ma del quale si ignorano i contenuti; l'imprenditore di Kirzner si basa sul concetto di *ignoranza inconsapevole*: scoprire per caso, senza averle deliberatamente cercate, imperfezioni e dunque opportunità di cui non si immaginava nemmeno l'esistenza.

Le basi della teoria neoclassica, della tradizione Austriaca e soprattutto le riflessioni apportate da Israel Kirzner permettono di riflettere su alcuni aspetti della nuova economia. Le simulazioni di imprese virtuali rappresentano un valido contributo per raffigurare processi continui di adattamento ed innovazione.

Capitolo 2 – La simulazione

La simulazione è un nuovo strumento, che si aggiunge a quelli tradizionali, con cui la scienza cerca di conoscere e di capire la realtà. Il mondo creato da una simulazione somiglierà al mondo reale e permetterà al programmatore di osservare e capire meglio la realtà che lo circonda. Non solo, la realtà virtuale mostrerà anche le potenzialità insite nel mondo reale, ma non sono ancora espresse.

Agendo sulla simulazione si potranno scoprire gli effetti degli interventi praticati e le soluzioni a problemi sui quali non ci si era soffermati in precedenza.

Se analizziamo il comportamento delle amministrazioni possiamo notare che nessuno è in grado di risolvere problemi altamente complessi in modo totalmente razionale, si procede dunque riducendo le soluzioni possibili per due principali motivi:

- 1) perché si scelgono le soluzioni che sembrano più giuste, ipotesi da migliorare;
- 2) perché quella è la prassi e “si è sempre fatto così”, ipotesi da sconfiggere.

La simulazione permette di andare verso queste direzioni.

Lo scopo della simulazione di impresa è duplice, da un lato è più improntato sulla teoria di impresa mentre dall'altro è prettamente pratico.

Iniziando dal primo la mira è quella di simulare processi continui di adattamento e innovazione. Secondo Kirzner, infatti, il processo di scoperta imprenditoriale si fonda sulla tendenza verso l'equilibrio e tale tendenza si manifesta in ogni momento, attraverso le opportunità di profitto generate da precedenti errori imprenditoriali.

Il secondo obiettivo è quello di disporre di un sistema computabile su cui provare i cambiamenti e le scelte necessarie per migliorare o semplicemente far funzionare processi complessi, come se si

lavorasse sulla realtà, tenendo conto in primo luogo della conoscenza effettivamente disponibile nel contesto studiato.

2.1 Simulazione e realtà

Si afferma (Parisi 2001) che le simulazioni, a differenza dei modelli tradizionali, non sono soltanto teorie, ma “realtà artificiale”.

Le simulazioni, infatti, soddisfano i tre criteri che definiscono la realtà:

1. la simulazione può essere osservata guardando sullo schermo di un computer, abbiamo quindi accesso ad essa con i nostri sensi;
2. attraverso i comandi che abbiamo impostato, possiamo agire su di essa e valutare i risultati delle nostre azioni;
3. la simulazione costituisce un vincolo, un limite alle nostre azioni, ma nello stesso tempo è il mezzo per svolgerle.

Il fatto che le simulazioni siano realtà costituisce una novità per la scienza. Le simulazioni, una volta costruite, diventano qualcosa di concreto ed entrano a far parte della realtà. Questo assimila la scienza, che è un'impresa volta a conoscere e capire la realtà, alla tecnologia, che è un'impresa volta a modificare la realtà e ad ampliarla.

Questo aspetto delle simulazioni è importante in quanto è alla base dello studio di fenomeni individuali ed unici.

Le normali teorie sono sempre generalizzazioni che astraggono dall'individuale mentre una simulazione può riprodurre un evento o un processo specifico.

Le simulazioni possono essere realtà anche nel senso fisico. Per capire questo è necessario rifarsi alla loro nascita.

Le simulazioni possono essere viste come la fusione di due strumenti che gli scienziati avevano già scoperto ed utilizzato in passato:

- la costruzione di modelli fisici semplificati ("modellini") della realtà;

- l'elaborazione di teorie matematiche capaci di cogliere la struttura quantitativa della realtà.

Con i primi condividono il fatto che sono anch'esse un pezzo di realtà costruita per riprodurre certi fenomeni, mentre con le seconde condividono il fatto che sono strumenti per automatizzare il processo di generazione di predizioni empiriche. Queste ultime, derivabili da una teoria espressa come simulazione, sono i risultati che la stessa simulazione produce quando gira nel computer.

L'elemento in più che le simulazioni hanno rispetto ai "modelli" fisici ed alle formule matematiche è appunto il computer. Prima di tutto, il computer permette di creare un modello fisico (virtuale) computazionale. In secondo luogo, permette di costruire modelli molto più complicati: non solo statici come quelli fisici ma dinamici, fatti di meccanismi che operano e di processi che avvengono e producono risultati. Di questi si tratterà più nello specifico nel paragrafo 5.

2.2 Simulazione e complessità

La simulazione non è uno strumento neutro in aggiunta a quelli tradizionali, infatti, non lascia la scienza immutata. Utilizzare questo metodo, come metodo di ricerca, ha delle conseguenze sul modo in cui la scienza concepisce e studia la realtà.

Una delle direzioni dello sviluppo della scienza è quella di vedere la realtà non come un insieme di sistemi semplici, ma come un insieme di sistemi complessi.

In un sistema semplice una singola causa produce un singolo effetto, per cui sapendo che si è verificata la causa possiamo prevedere che si produrrà o si è prodotto l'effetto. Quest'ultimo può anche essere prodotto da più cause che si sommano senza però interagire fra loro.

I sistemi semplici hanno anche altre caratteristiche:

- gli stati successivi sono prevedibili se si conoscono gli stati precedenti;

- le trasformazioni nel tempo sono prevedibili;
- l'effetto di una perturbazione, derivante da un evento esterno, che è commisurato all'entità della perturbazione stessa;
- le condizioni iniziali influenzano lo sviluppo del sistema nel tempo, ma condizioni iniziali molto simili di due sistemi porteranno a sviluppi pressoché uguali;
- il contesto in cui si opera non ne influenza il funzionamento;
- gli elementi non si influenzano reciprocamente;
- non fanno parte di una gerarchia di sistemi;
- il ruolo dei singoli elementi è ben individuabile.

Anche se i sistemi semplici sono i più comprensibili, la realtà è fatta soprattutto di sistemi complessi.

In un sistema complesso ci sono molte cause che producono un dato effetto e le relazioni tra le cause non sono lineari.

I sistemi complessi sono fatti di moltissimi elementi distinti che si influenzano localmente: un elemento interagisce solo con un ristretto numero di altri elementi di cui è composto l'intero sistema. Da queste numerose interazioni locali emergono proprietà globali dell'intero sistema che non sono deducibili o prevedibili anche se conosciamo alla perfezione i singoli elementi ed il loro modo di interagire.

Altre proprietà dei sistemi complessi sono le seguenti:

- gli stati futuri non sono prevedibili sulla base degli stati precedenti;
- le trasformazioni nel tempo non sono prevedibili;
- l'effetto di una perturbazione, derivante da un evento esterno, non è commisurato all'entità della perturbazione stessa;
- le condizioni iniziali ne influenzano lo sviluppo, due sistemi con condizioni iniziali appena differenti potranno avere uno sviluppo molto diverso;
- il contesto in cui si opera ne influenza il funzionamento;
- gli elementi si influenzano reciprocamente;
- tendono ad essere inseriti in una gerarchia di sistemi;

- il ruolo dei singoli elementi non è ben individuabile.

Proprio per queste caratteristiche, i sistemi complessi non possono essere abbracciati da teorie espresse nei modi tradizionali, da sistemi di equazioni matematiche o da esperimenti in laboratorio.

La simulazione è il principale strumento per analizzare i sistemi complessi in quanto la si può vedere come un laboratorio sperimentale virtuale.

2.3 Simulazione e conoscenza

La conoscenza in azienda può essere rappresentata utilizzando tre formalismi:

1. I modelli letterario-descrittivi;
2. I modelli matematico-statistici;
3. I modelli con codice informatico.

Nella letteratura economica di questi ultimi anni è cresciuta sempre più l'importanza del tema della "conoscenza".

La conoscenza è ormai considerata come una risorsa cruciale del sistema economico, assimilabile ai fattori produttivi capitale e lavoro. Essa svolge un ruolo particolarmente importante per quanto riguarda la "capacità innovativa", intesa come introduzione di nuove tecnologie e nuovi prodotti. E' principalmente su questo fattore che si basa la competitività del sistema economico ed il benessere complessivo della società.

Molti economisti interessati all'analisi del comportamento economico riferito all'innovazione hanno analizzato le caratteristiche organizzative delle imprese, cercando di capire quali fossero le forme di organizzazione in grado di favorire la creazione e l'utilizzo della conoscenza, facilitando l'introduzione di innovazioni di processo e di prodotto.

La maggior parte di questi studi di economia dell'innovazione ha, in altre parole, valutato la conoscenza come una risorsa scarsa del sistema produttivo, assimilabile al capitale ed al lavoro. I problemi di

allocazione e gestione della conoscenza vengono di conseguenza risolti applicando ad essi gli stessi schematismi sviluppati per l'allocazione delle risorse scarse. Il pericolo implicito in questo tipo di approccio è rappresentato dal rischio di incorrere in un eccesso di semplificazione: mentre è chiaro a tutti a cosa ci si riferisca quando si parla di "lavoro" e "capitale", non sempre è altrettanto chiaro a cosa ci si riferisca quando si parla di "conoscenza".

La conoscenza può essere distinta in interna ed esterna.

Per conoscenza interna si intende quella che viene sviluppata facendo ricorso esclusivamente alle risorse interne all'impresa. Nel contesto economico attuale questo tipo di conoscenza, da sola, non basta per garantire la competitività delle imprese, le quali sono sempre più spesso spinte a fare riferimento a fonti esterne, in grado di apportare le ulteriori conoscenze richieste per sostenere e alimentare i processi innovativi. Le forme di conoscenza ottenibili mediante il ricorso a Internet o ad altre tecniche di comunicazione di rete vanno considerate come "conoscenza interna", nel caso in cui l'accesso a tali risorse sia gratuito, e come "conoscenza esterna", nel caso in cui tale accesso sia a pagamento.

Nella maggior parte dei casi però, la conoscenza non può essere considerata come un semplice bene materiale che possa essere comprato e venduto sulla base di normali meccanismi di mercato. Riuscire a trasferire la conoscenza dal luogo nel quale essa è stata generata al luogo nel quale essa deve essere applicata a fini commerciali è un processo che spesso può essere alquanto complesso.

Abbiamo riscontrato questo fenomeno nell'applicazione della realtà Vir al modello di simulazione. Infatti nel reperire le informazioni necessarie, intervistando gli ingegneri ed i tecnici responsabili della Vir, ci siamo accorti che questi, pur gestendo l'azienda giorno per giorno, hanno avuto difficoltà a descrivere la loro impresa.

I problemi che il trasferimento di conoscenza comporta sono legati alle caratteristiche della conoscenza stessa, che la contraddistinguono nei confronti delle altre merci di scambio, solitamente abbastanza facili da immagazzinare e trasportare. In effetti, mentre alcune componenti conoscitive possono essere facilmente codificate e trasmesse, altre componenti non possono essere facilmente trasmesse in forma verbale o scritta. In letteratura questo tipo di conoscenza è definita *tacita*, o *implicita*.

2.4 La conoscenza tacita

La “conoscenza tacita” è una parte della conoscenza umana distinta e complementare rispetto alla conoscenza *esplicita*. Per conoscenza esplicita si intendono quelle forme di sapere che possono essere codificate in forma orale o scritta.

In molti casi accade che si riesca a portare a termine un’attività complessa, senza che si sia però in grado di spiegare in maniera dettagliata ad altre persone in che modo si sia riusciti ad eseguire tale attività:

“Poniamo il caso che io sappia andare in bicicletta, oppure stare a galla mentre nuoto. Io potrei non avere la minima consapevolezza di come sia in grado di fare ciò, oppure potrei averne un’idea totalmente sbagliata, o anche molto incompleta. Eppure, ciò non mi impedisce di riuscire a pedalare e a nuotare correttamente. In questo caso, non sarebbe nemmeno possibile sostenere che io sappia nuotare o andare in bicicletta, e al tempo stesso non sappia come coordinare i complicati meccanismi muscolari richiesti per lo svolgimento di questo tipo di attività. In effetti, io so svolgere queste attività e al tempo stesso so coordinare tutti i minimi movimenti che esse implicano, sebbene io non sia in grado di descrivere esattamente tali movimenti. Questo è dovuto al fatto che io sono solo sussidiariamente consapevole di queste

cose, e spesso la nostra consapevolezza delle cose non è sufficientemente affinata per consentirci di descriverle correttamente” (Polanyi 1966).

Questo semplice esempio permette di chiarire il concetto di conoscenza tacita. La necessità di arrivare ad una definizione di conoscenza tacita più precisa è un requisito necessario se si vuole analizzare il ruolo che la conoscenza può svolgere nei processi di trasferimento scientifico e tecnologico. Un prezioso ausilio giunge, a tal fine, dai più recenti sviluppi conseguiti nel campo delle scienze cognitive, in particolar modo in quei settori che si sono occupati della rappresentazione della conoscenza.

Le scienze cognitive, secondo la definizione proposta da Legrenzi (2002), hanno come oggetto di studio la cognizione, cioè la capacità di un qualsiasi sistema, naturale o artificiale, di comunicare a se stesso e agli altri ciò che conosce. Da tempo, all'interno della psicologia cognitiva, si tende a separare lo studio del ragionamento esplicito (o consapevole) da quello del ragionamento implicito (o inconsapevole).

Per studiare la conoscenza bisogna tener conto di diverse caratteristiche. Le distinzioni fanno riferimento da un lato ai processi di apprendimento e di rappresentazione, dall'altro alla separazione tra processi espliciti e processi impliciti. Sovrapponendo queste due dimensioni, possiamo pensare ad una classificazione di quattro ambiti separati:

- 1) processi espliciti di apprendimento della conoscenza;
- 2) processi espliciti di rappresentazione della conoscenza;
- 3) processi impliciti di apprendimento della conoscenza;
- 4) processi impliciti di rappresentazione della conoscenza.

In letteratura, gli sforzi sono stati rivolti principalmente allo studio dei problemi connessi con i processi di rappresentazione della conoscenza.

Dal lato implicito e tacito della conoscenza, spicca la notevole discrepanza tra l'abilità pratica di risolvere un problema assegnato e la capacità verbale di descrivere e trasmettere la conoscenza utilizzata per pervenire alla soluzione.

Particolarmente interessanti a questo riguardo sono gli studi di Berry e Broadbent (1988) sul controllo di sistemi complessi. Utilizzando simulazioni al computer, si richiede ai soggetti di raggiungere e mantenere determinati valori obiettivo di una variabile di output semplicemente modificando i valori assegnati ad una variabile input. I risultati mostrano che con l'acquisizione di esperienza pratica le prestazioni dei soggetti coinvolti nel problema del controllo di produzione e nel problema dell'interazione personale migliorarono significativamente. L'esperienza non ha però alcuna conseguenza sulla capacità dei soggetti di rispondere correttamente ad una serie di domande relative ai problemi da loro affrontati. I soggetti che si comportavano meglio nello svolgimento effettivo del compito tendevano ad essere quelli che rispondevano peggio alle domande scritte e viceversa.

Nell'ambito aziendale vige un comune atteggiamento che consiste nella convinzione che la maggiore difficoltà sia costituita dalla mancanza di informazioni e di dati. In realtà però il problema risiede nell'incapacità di percepire le conseguenze delle informazioni di cui si è in possesso. Un aiuto in questo senso può essere l'applicazione della metodologia *system dynamics*. I modelli «dinamici» aiutano a comprendere che lo stato attuale del sistema costituisce l'effetto di azioni e reazioni prodottesi in passato e che le decisioni presenti concorreranno a modificare la struttura stessa del sistema nel futuro.

2.5 Sistemi dinamici e micromondi

I modelli dinamici (Bianchi 2001) si basano su una prospettiva orientata all'individuazione e allo studio dei circuiti di retroazione

(feedback) caratterizzanti i fenomeni gestionali, osservati nell'ambito di sistemi chiusi. Tali modelli tendono ad esplicitare la chiave di lettura implicita della realtà che ciascun decisore detiene nella propria mente.

Un modello dinamico accoglie tre principali categorie di variabili:

1. variabili stock: indicano il livello di risorse chiave del sistema;
2. variabili flusso: rappresentano le variazioni in aumento o in diminuzione delle variabili stock;
3. variabili di input: rappresentano vincoli esogeni o leve direzionali sulle quali è possibile agire allo scopo di influenzare le variabili di stock.

Soltanto queste ultime variabili costituiscono delle informazioni riguardanti le condizioni in essere del sistema; esse rappresentano la sintesi di un processo di accumulazione alimentato dalle variabili flusso. I flussi includono quattro principali componenti:

- l'obiettivo perseguito dal decisore;
- la condizione osservata;
- lo scostamento tra obiettivo e condizione osservata;
- l'azione desiderata, orientata al raggiungimento dell'obiettivo.

Un modello dinamico di simulazione si basa sull'esplicitazione delle politiche che sono alla base del processo decisionale. La formulazione delle decisioni è osservata con un continuo processo di conversione delle informazioni in segnali suscettibili di alimentare delle azioni orientate a modificare gli stock di risorse nella direzione desiderata.

La qualità delle politiche decisionale dipende dalle informazioni prese in esame dagli individui e dal modo in cui esse sono convertite in azioni, secondo un certo assetto di "regole" esplicite o implicite.

Secondo questa prospettiva, la formulazione delle decisioni viene osservata come un processo caratterizzato da filtri organizzativi e cognitivi. Tra le tante informazioni che affluiscono da un determinato soggetto, soltanto un numero limitato viene effettivamente preso in considerazione.

Nel processo di formulazione delle decisioni si possono distinguere due tipi di circuiti di retroazione:

- positivo: descrive un circolo virtuoso o vizioso riguardante un processo di sviluppo o di involuzione (ad esempio la pubblicità);
- negativo: di tipo bilanciante.

La dominanza dei circuiti a retroazione positiva non è illimitata nel tempo, nel lungo periodo possono essere controbilanciati da circuiti a retroazione negativa, che sono una fonte di stabilità del sistema.

Ad esempio, nel lungo periodo, il feedback positivo generato dagli investimenti pubblicitari potrà essere frenato da una saturazione del mercato. In tal senso, dato un certo mercato potenziale, il flusso dei nuovi clienti che andranno ad incrementare lo stock dei clienti aziendali verrebbe successivamente a ridursi a causa di una progressiva saturazione del mercato.

Si distinguono feedback negativi di primo grado e di ordine superiore al primo, la distinzione riguarda il numero di livelli tra loro concatenati nelle relazioni causali che influenzano una determinata risorsa chiave. La dinamica generata da un feedback negativo di grado superiore al primo è assai meno prevedibile di quella generata da un feedback di primo grado.

Quando in un modello dinamico coesistono sia circuiti positivi sia negativi l'andamento delle risorse chiave può variare a seconda della predominanza di un circuito sull'altro; è questo il caso del ciclo di vita di un prodotto (figura 1). Durante i primi anni di vita di un prodotto, il flusso delle vendite aumenta progressivamente, in virtù della prevalenza del feedback positivo. Tuttavia, man mano che aumenta lo stock di clienti che già utilizzano il prodotto, una certa percentuale di essi decide di passare ad altri prodotti, facendo così ridurre lo stock di clienti aziendali (feedback negativo).

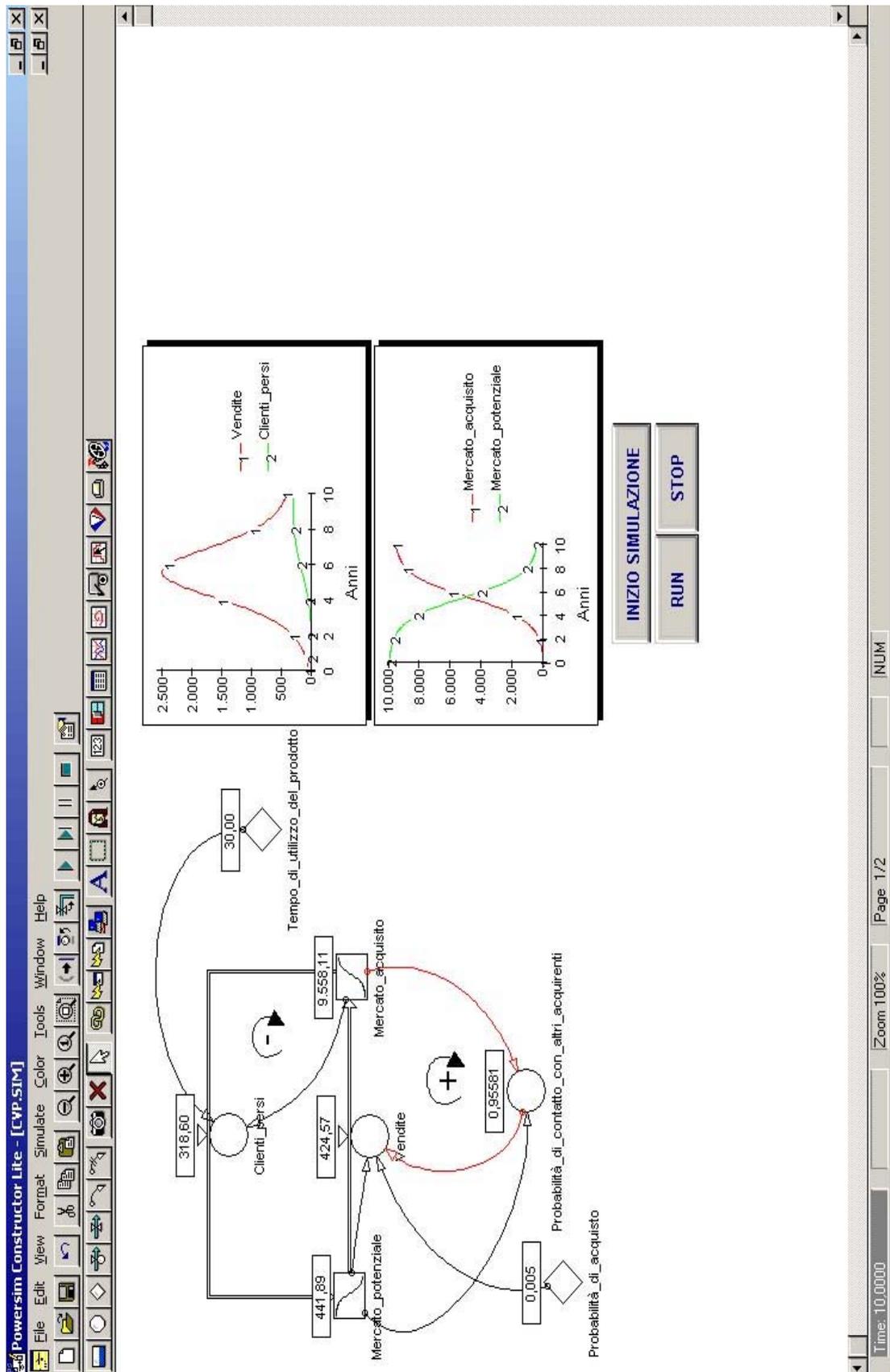


Figura 1: esempio di formalizzazione di un sistema dinamico tramite il software Powersim Constructor Lite (CD allegato al libro).

Un concetto implicito nell'esistenza di un circuito di causa effetto è quello di ritardo temporale. Un ritardo è di per sé connesso all'esistenza di uno stock e di un flusso tra loro collegati. È possibile distinguere due diverse tipologie di ritardo:

- il ritardo fisico materiale: fa riferimento ad un flusso di risorse fisiche che pervengono dopo essere transitate da uno o più stock intermedi;
- il ritardo informativo: può essere importabile sia a congetture soggettive sia alla tardiva disponibilità di informazioni.

Un esempio del primo ritardo può essere quello illustrato nella figura 2 che mostra come il flusso di prodotti consegnati ai magazzini, in un determinato arco temporale, rappresenti un ritardo fisico riferito ad un flusso di merci precedentemente immesse in lavorazione, suscettibili di diventare prodotto finito dopo un determinato tempo di produzione.

Un esempio di ritardo informativo è costituito dalla formazione di aspettative sulla domanda di mercato come mostrato in figura 3.

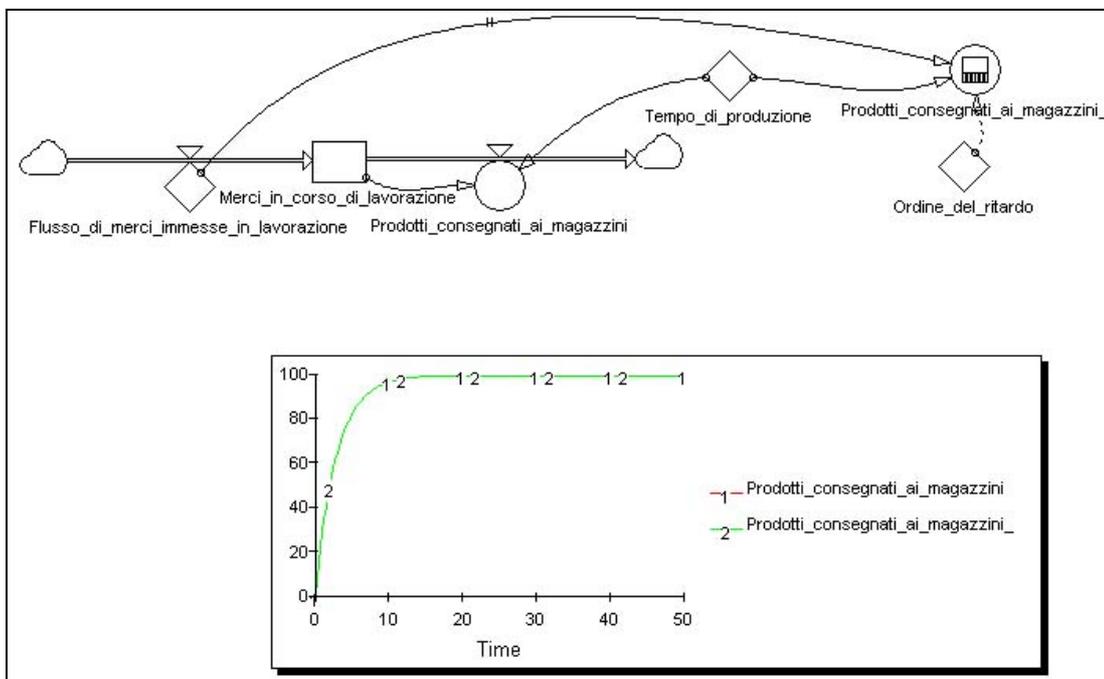


Figura 2: esempio di ritardo fisico-materiale.

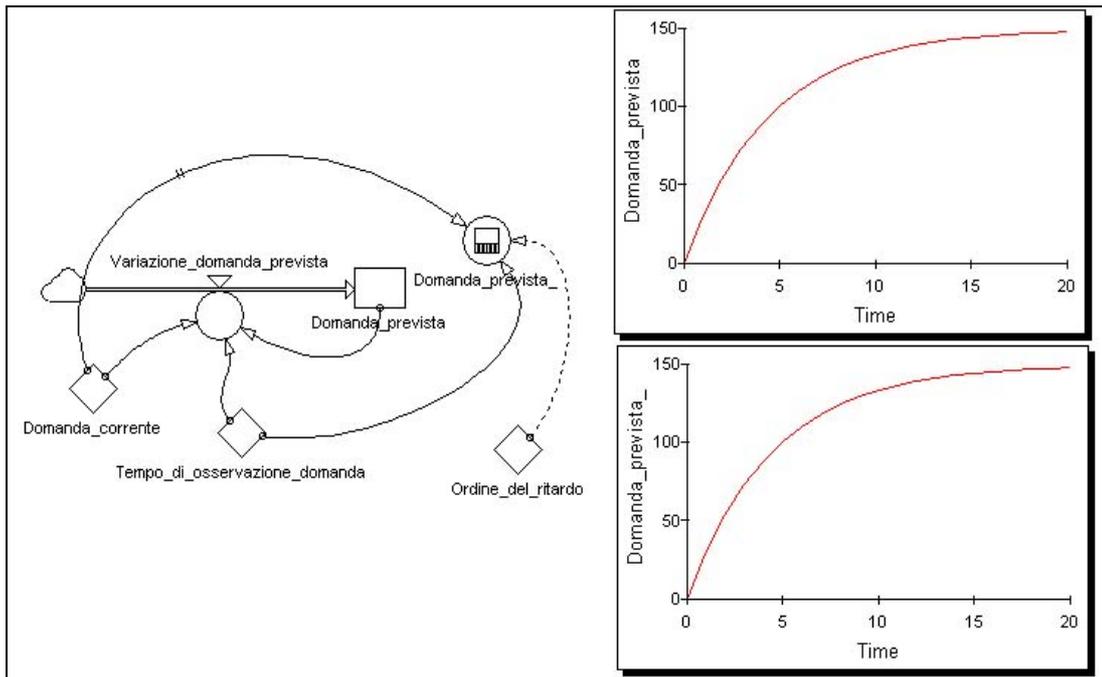


Figura 3: esempio di ritardo informativo.

Attraverso i modelli dinamici i diversi processi gestionali, in relazione ai quali si desidera comprendere le strategie e le politiche da adottare in futuro, sono rappresentati attraverso sistemi chiusi, ovvero mediante circuiti di causa effetto che inglobano le strategie e le politiche costituenti oggetto di valutazione. Un sistema può definirsi chiuso quando è influenzato dai risultati che esso stesso ha concorso a generare in passato.

Gli attori chiave dell'impresa devono comprendere la struttura del sistema in cui operano. La realizzazione del modello diventa parte di un più ampio processo di apprendimento che tende all'acquisizione di un assetto mentale che consenta all'imprenditore proprietario di cogliere cause da effetti, ritardi temporali, relazioni non lineari tra le variabili rilevanti e intervenire sulla configurazione del sistema aziendale per influenzare la dinamica futura delle risorse chiave verso la direzione desiderata.

Migliorare i modelli mentali per avvicinarli quanto più possibile alla realtà è il presupposto per un consapevole governo dello sviluppo

aziendale. Il processo di apprendimento continuo si articola lungo le seguenti fasi:

- osservazione della realtà;
- riflessione, esplicitazione e confronto della percezione di ciascun decisore in merito al sistema;
- diagnosi e condivisione di uno schema interpretativo della realtà;
- formulazione delle decisioni.

Affinché lo studio di un fenomeno possa giustificare l'adozione della metodologia della system dynamics deve presentare determinati requisiti:

- a) esistenza di relazioni di causa-effetto tra le variabili che concorrono a modificare la dinamica delle risorse-chiave;
- b) presenza di effetti di breve e lungo periodo contrastanti tra loro per una medesima variabile;
- c) andamenti contrastanti nel tempo fra diverse variabili;
- d) esistenza di significative relazioni non lineari fra diverse variabili rilevanti;
- e) il significativo peso di variabili non monetarie o anche puramente qualitative sulla dinamica delle risorse-chiave oggetto dell'analisi.

Quanto più elevata è la complessità gestionale e la discontinuità ambientale, tanto maggiore è il rischio che la realtà oggettiva venga percepita da ciascuno degli attori chiave aziendali in modo diverso. La diversità dei modelli mentali è una ricchezza per l'azienda, perché consente di osservare i fenomeni gestionali sotto differenti prospettive.



Figura 4: la circolarità del processo di modellizzazione dinamica.

I modelli dinamici hanno una natura descrittiva relativamente alla realtà rappresentata, le informazioni e i valori in essi accolti sono finalizzati ad offrire una visione di sintesi riguardo alla struttura e alla dinamica del sistema in funzione delle ipotesi adottate, con riferimento ad un determinato arco temporale. L'attitudine a rappresentare l'andamento delle risorse chiave in funzione delle politiche volte ad influire su di esse e degli scenari volitivi del sistema analizzato sono una misura della qualità del modello come supporto all'apprendimento.

Una progressiva focalizzazione dei confini del sistema rilevante consente di migliorare i modelli mentali dei decisori.

L'esplicitazione dei modelli mentali rappresenta un mezzo attraverso il quale i soggetti non individuano semplicemente le soluzioni ai problemi, ma indagano sulle cause al fine di adottare delle scelte che siano in grado di incidere sulle reali disfunzioni piuttosto che sui sintomi esterni.

Il processo di modellizzazione - apprendimento deve rendere visibili a ciascun attore chiave i limiti sottostanti ai propri "filtri informativi" ed a promuoverli in relazione al progressivo emergere dell'evidenza dei

fenomeni concreti, come risultato di un processo di diagnosi condiviso dai diversi soggetti operanti nel sistema aziendale.

Con il termine micromondi (Bianchi 2001) si intendono modelli di simulazione che tendono a riprodurre le principali caratteristiche del sistema in cui opera il decisore, consentendo a quest'ultimo di sperimentare in un ambiente protetto, individualmente o in gruppo, gli effetti derivanti dall'adozione di determinate strategie e politiche.

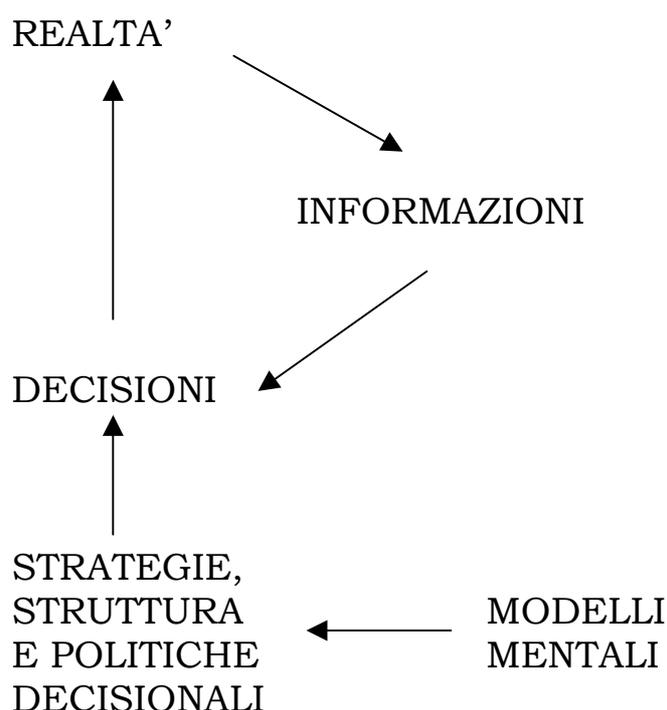


Figura 5: relazione tra decisioni e sistema di controllo.

Attraverso i micromondi basati su modelli dinamici il soggetto decisore può scorgere in anticipo i presumibili effetti sulle risorse chiave conseguenti alla manovra delle leve direzionali disponibili. Ciò può agevolare la comprensione della struttura sottostante ai processi che contraddistinguono la dinamica del sistema in cui ciascun "attore" opera, per valutare e riconoscere così i punti deboli dei propri modelli mentali. L'utilizzo dei micromondi, inoltre, in quanto facilita l'esplicitazione dei modelli mentali di ciascuno, consente di migliorare la qualità della comunicazione tra gli attori-chiave e, quindi, di

comprendere meglio i diversi possibili esiti conseguenti all'adozione delle alternative strategiche perseguibili. I micromondi sono quindi in grado di fornire una chiave di lettura adeguata ai fabbisogni conoscitivi sia per le situazioni di cambiamento incrementale, sia per quelle di cambiamento strutturale. Sono stati distinti micromondi di prima e di seconda generazione, la differenziazione riguarda il modo in cui tali strumenti sono in grado di influire sulle fasi caratterizzanti il processo di apprendimento individuale e di gruppo.



Figura 6: L'attività di programmazione e controllo come processo di apprendimento continuo attraverso il supporto di modelli dinamici.

Tali strumenti cognitivi possono essere di utile adozione a supporto dei meccanismi di programmazione e controllo, la loro realizzazione deve essere personale per ogni impresa e può convenientemente basarsi su un opportuno connubio tra modelli contabili e dinamici.

Capitolo 3 – Strumenti di simulazione

Obiettivo del presente capitolo è introdurre brevemente le nozioni alla base del paradigma Object Oriented con particolare riguardo al linguaggio Java ed alle librerie Swarm.

Tale percorso si conclude con la descrizione del modello jES, un frame in java-swarm, utilizzato per l'applicazione della simulazione a contesti di impresa.

3.1 La programmazione ad oggetti (OOP)

L'acronimo OOP in informatica sta per *Object Oriented Programming* (Programmazione Orientata agli Oggetti) e vuole denotare l'insieme delle caratteristiche a fondamento del paradigma della programmazione ad oggetti.

Le caratteristiche che deve avere un linguaggio di programmazione per essere classificato OOPL sono le seguenti:

1. insieme di operazioni astratte, associate ad un tipo;
2. stato locale;
3. ereditarietà.

La prima caratteristica, spesso associata all'acronimo ADT (Abstract Data Type), stabilisce che l'unico modo per agire sugli oggetti di un certo tipo è quello di invocare delle *operazioni* (nel gergo OOP chiamate metodi) prestabilite e note agli utenti in modo astratto, cioè indipendente dalla *rappresentazione* prescelta per la loro concreta *implementazione* (realizzazione).

La seconda caratteristica stabilisce che ogni oggetto ha un proprio stato locale che può cambiare per effetto dell'esecuzione di alcuni metodi. In pratica il valore dello stato locale ad un certo istante è stabilito dal contenuto di un insieme di celle di memoria associate all'oggetto, dette campi o attributi dell'oggetto.

L'ereditarietà stabilisce infine che il linguaggio deve prevedere un meccanismo per stabilire delle relazioni tra *classi* di oggetti. Questo

meccanismo consente in definitiva il riuso e il polimorfismo, due dei principali fattori alla base del successo della OOP.

Prima di esaminare più a fondo i concetti di ereditarietà e polimorfismo è necessario parlare di oggetti e di classi.

Gli oggetti hanno un ruolo fondamentale: sono i mattoni con i quali vengono costruiti i programmi.

Un oggetto è un'entità dotata di:

- identità;
- stato;
- comportamento.

Identità

L'identità di un oggetto è la caratteristica che lo contraddistingue da tutti gli altri. Spesso ciò è dato da un valore univoco come ad esempio può essere un codice.

Stato

Gli oggetti, tipicamente, non vengono creati per permanere in un determinato stato; al contrario, durante il loro ciclo di vita transitano in una serie di fasi. Alcuni di essi sono vincolati ad evolvere attraverso un insieme finito di fasi, mentre per altri è infinito oppure molto grande. Quindi, mentre per i primi può avere molto senso descrivere il diagramma degli stadi attraverso i quali i relativi oggetti possono transitare durante la propria vita, per gli altri l'esercizio è decisamente più complesso e non sempre fattibile e/o utile. Si consideri una classe che rappresenta una semplice lampadina, in questo caso l'insieme degli stati dei relativi oggetti prevede due soli elementi: acceso e spento. Si consideri ora una sua evoluzione, ossia una lampadina digitale con un numero ben definito di diverse intensità luminose. In questo caso l'insieme degli stati potrebbe prevedere: spenta, accesa intensità 1, accesa intensità 2, ... , accesa intensità max.

Altri oggetti, invece, durante l'arco della propria vita evolvono attraverso una serie di stati, che però non sono numerabili. Si consideri per esempio un sistema di illuminazione delle stanze la cui funzione sia accendere/spegnere i vari fari in funzione del numero di persone presenti nelle stanze. Sebbene questo numero sia delimitato, non è conveniente indicare i vari stati dell'oggetto.

L'ultima categoria è costituita dagli oggetti il cui stato è teoricamente infinito.

Lo stato di un oggetto è molto importante poiché ne influenza il comportamento futuro.

Comportamento

Una volta studiato e formalizzato lo stato di un oggetto si è effettuato un passo in avanti nel processo di astrazione, ma non è ancora sufficiente per la completa descrizione dello stesso. Molto importante è analizzare anche il comportamento.

Un oggetto non solo non viene creato per essere lasciato ozioso in uno specifico stato, ma neanche per lasciarlo "morire di solitudine". Tipicamente un oggetto interagisce con altri scambiando messaggi, ossia rispondendo agli stimoli provenienti da altri oggetti e, a sua volta inviandoli ad altri al fine di ottenere la fornitura di "sottoservizi" necessari per l'espletamento del proprio. Quindi, il comportamento di un oggetto è costituito dalle inerenti attività (operazioni) visibili e verificabili dall'esterno. "Il comportamento stabilisce come un oggetto agisce e reagisce, in termini di cambiamento del proprio stato e del transito dei messaggi." (Booch).

Per creare gli oggetti la tecnica più seguita è quella di definire prima una classe, cioè uno schema di creazione che definisca i possibili stati e i comportamenti, e di istanziare (ossia creare fisicamente) l'oggetto dalla classe. Tale istanza viene detta *costruttore* o *creatore* e determina univocamente l'identità di un oggetto e ne stabilisce lo *stato iniziale*.

Ogni oggetto è istanza di una e una sola classe. Nella fase di sviluppo si definiscono le classi, che, a meno di particolari vincoli, originano diversi oggetti.

Una rappresentazione grafica della relazione fra oggetti e classi può essere quella riportata in figura 1.

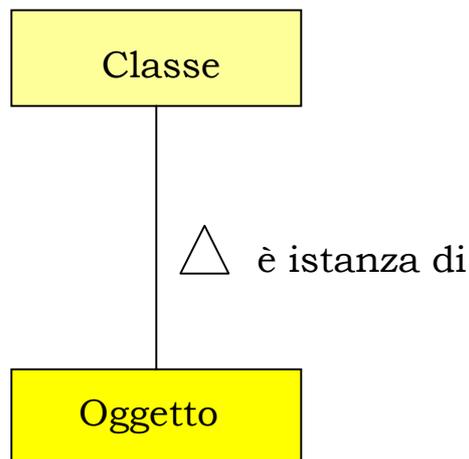


Figura 1: relazione esistente fra gli oggetti e le classi.

L'ereditarietà risulta indispensabile per processi di modellizzazione molto importanti nella moderna costruzione del software quali la *specializzazione* e la *generalizzazione*.

Nella sua forma più semplice, l'ereditarietà è un meccanismo che consente ad una classe di considerarsi erede di un'altra, detta classe padre o superclasse, con una dichiarazione che possiamo assumere del tipo:

classe <nome> eredita da <classe padre>

Così facendo la classe <nome>, detta sottoclasse (o classe derivata), eredita tutte le proprietà della <classe padre> specificata: tutti gli attributi e i metodi.

Quello che in sostanza succede con l'ereditarietà è la clonazione di una classe già esistente, al fine di ampliarne alcune caratteristiche o modificarne alcune parti. E' da tener presente che se la classe

originale cambia il clone modificato riflette anch'esso questi cambiamenti.

Con il polimorfismo è possibile progettare ed implementare sistemi facilmente estendibili. I programmi possono essere scritti per elaborare genericamente gli oggetti di tutte le classi esistenti in una gerarchia.

Il polimorfismo prevede che: *la decisione sulla quale debba essere la routine da richiamare viene presa a tempo di esecuzione*, a seconda della classe effettiva (più stretta) di appartenenza dell'oggetto rispetto a cui viene fatta la chiamata. Questa tecnica è nota come collegamento dinamico (*late o dynamic binding*) dei nomi al codice che deve essere effettivamente eseguito. Esso si contrappone al tradizionale collegamento statico (*early o static binding*) deciso dal compilatore, di norma, nel caso di chiamate non polimorfe.

3.2 Il linguaggio Java

Java è un linguaggio orientato agli oggetti e, da questo punto di vista, l'impostazione data risulta piuttosto rigorosa, dato che non è possibile fare ricorso a organizzazioni miste del codice (in parte a oggetti, in parte strutturate) come avviene in altri linguaggi, primo fra tutti il C++.

La filosofia Object Oriented di Java ha come obiettivo la creazione di un linguaggio estremamente semplice, facile da apprendere volto a eliminare quei costrutti pericolosi che portano in certi casi a situazioni non facili da gestire.

Una classe in Java si definisce per mezzo della parola chiave `class`. La creazione di una classe avviene in questo modo:

```
[accesso] class nomeclasse { ... }
```

In Java esistono tre specificatori di accesso: `public`, `private` e `protected`, con i quali si può specificare l'accessibilità di una classe, metodo o variabile, secondo le seguenti specifiche.

`public`: una proprietà o un metodo all'interno di una classe A che venga definito pubblico acquista visibilità totale. Potrà infatti essere visto al di fuori della classe, sia dalle classi che ereditano dalla classe A, sia da quelle classi che non estendono A. Anche le classi esterne al package di A potranno accedere liberamente a tali elementi.

`private`: questo specificatore è di fatto il simmetrico di `public`, dato che blocca ogni tipo di visibilità. Metodi o proprietà della classe A dichiarati privati hanno visibilità limitata alla sola classe A, nemmeno le classi derivate potranno accedervi.

`protected`: i metodi e le proprietà dichiarate protette possono essere viste da altre classi esterne solo se queste appartengono allo stesso package. All'esterno del package nessuno potrà accedere a tali elementi mentre esclusivamente le classi figlie li ereditano al loro interno.

Esiste inoltre una modalità d'accesso di default: a questo caso non corrisponde una parola chiave, trattandosi del livello di visibilità che viene assegnato automaticamente nel caso in cui non venga specificata esplicitamente nessuna modalità. In questo caso la visibilità è limitata alla gerarchia di classe e al package.

Per descrivere il corpo della classe, all'interno delle parentesi graffe si definiscono i vari metodi, variabili e in casi particolari anche altre classi.

Come già detto uno dei fondamenti base della programmazione ad oggetti, insieme al polimorfismo, è l'ereditarietà. Per specificare che la classe A eredita o deriva da B si utilizza la parola chiave `extends`, ad esempio:

```
public class A extends B {  
    ...  
    corpo della classe A  
    ...  
}
```

Le parole chiave `public`, `private` e `protected` non sono le uniche che si possono utilizzare in abbinamento a un metodo o a una classe per specificarne ulteriormente la natura.

Esiste anche `static`, che ha un significato completamente differente, e varia a seconda che si riferisca a un metodo o a una proprietà.

Nel caso si definisca una proprietà `static`, allora tale proprietà sarà condivisa fra tutte le istanze di tale classe.

Lo specificatore `static` applicato invece a un metodo indica che esso può essere invocato anche se la classe di cui fa parte non è stata istanziata.

Un caso tipico è quello del metodo `main` che può, e anzi deve, essere chiamato senza che la classe sia stata creata con una chiamata a `new`: in questo caso non potrebbe essere altrimenti, dato che tale metodo ha proprio lo scopo di far partire l'applicazione, quando nessuna classe ancora esiste.

Per quanto riguarda le regole relative ai nomi delle variabili e dei metodi, il formalismo adottato è il seguente: tutti i nomi di classi devono iniziare per lettera maiuscola (così come le lettere iniziali delle parole composte), mentre i nomi delle variabili e dei metodi avranno la prima lettera minuscola e le prime lettere delle parole composte maiuscole. Ad esempio:

```
NomeClasse          nomeMetodo()          nomeVariabile;
```

Per controllare il flusso delle operazioni all'interno dei metodi, Java mette a disposizione una serie di parole chiave e di costrutti molto simili a quelli che si trovano negli altri linguaggi di programmazione procedurale.

Costrutto `if else`

```
If(espressione booleana)
  istruzione1
else
  istruzione2
```

Si esegue il blocco istruzione 1 se l'espressione booleana assume il valore booleano true, altrimenti che si esegue il blocco istruzione 2.

Costrutto while e do while

```
While (espressione booleana)
    istruzione
```

Si esegue il blocco istruzione fino a che l'espressione booleana assume il valore booleano true.

Funzionalità analoga è offerta dal costrutto do-while:

```
do
    istruzione
while (espressione booleana)
```

La differenza fra i due costrutti è che nel primo caso il controllo viene effettuato prima di eseguire l'istruzione, mentre, nel secondo caso, dopo.

Costrutto for

```
for (espressione di inizializzazione; espressione
    booleana; espressione di incremento) {
    istruzione
}
```

Si esegue il blocco for fino a che l'espressione booleana assume valore booleano true; espressione di inizializzazione viene eseguita solo la prima volta, mentre l'espressione di incremento ad ogni iterazione.

Costrutto switch

```
switch variabile
    case valore1:
        istruzione1
    case valore2:
        istruzione2
    case valore3:
        istruzione3
    case valore4:
        istruzione4
```

default:

Si esegue un controllo sulla variabile `intera` `variabile` e si esegue il blocco di codice che segue il `case` corrispondente al valore assunto dalla variabile. Infine viene sempre seguito il blocco `default`.

Parliamo ora dei costruttori di classe, il costruttore è un particolare metodo che serve per creare una istanza della classe. L'invocazione avviene per mezzo della parola chiave `new`, come ad esempio:

```
String str = new String();
```

Per definire il costruttore della classe è necessario e sufficiente creare un metodo con lo stesso nome della classe.

Ad esempio, se si ha una classe detta `MiaClasse`, il costruttore avrà questa forma:

```
//Costruttore di default
public MiaClasse() {}

//Un altro costruttore
public MiaClasse(String str) {}
```

Il primo dei due viene detto costruttore di default, dato che non prende nessun parametro.

Normalmente il costruttore è definito pubblico, in modo che possa essere utilizzato per creare istanze di oggetti come ad esempio:

```
MiaClasse = new MiaClasse();
```

In Java ogni oggetto definito deve obbligatoriamente avere un costruttore; nel caso non lo si definisca esplicitamente, il compilatore, al momento della compilazione, ne inserisce uno vuoto che non riceve parametri in input (per questo motivo è detto di default) e che in questo caso non compie alcuna azione.

Un altro aspetto, di minore importanza, è quello relativo al modo in cui vengono istanziate le variabili di una classe.

Le variabili di tipo primitivo (ovvero quelle definite al di fuori di ogni metodo) sono istanziate al loro valore di default al momento dell'istanziamento della classe, mentre per le variabili di tipo reference in genere viene assunto il valore *null*.

Il `null` un è un valore speciale: di fatto non contiene nessun valore specifico, ma fungere da “tappo”. In pratica, il compilatore permette di utilizzare una variabile che sia stata inizializzata con `null`.

Ecco i vari tipi di variabili in Java:

tipi		Parole chiave	
primitivi	booleani		boolean
	numerici	interi	byte
			short
			int
		long	
		char	
	floating point	float	
		double	
reference	classi		class
	interfacce		interface
Null			

I principali operatori, invece, sono i seguenti:

Operatori aritmetici	
simbolo	significato
+	addizione aritmetica o concatenazione di stringhe
-	sottrazione aritmetica
*	moltiplicazione aritmetica
\	divisione aritmetica
=	assegnazione
Operatori booleani e di confronto	
simbolo	significato
&&	AND logico
	OR logico
!	negazione
==	confronto

3.3 Swarm

Swarm (www.swarm.org) è un insieme di programmi e librerie standard, utilizzabili per la simulazione e l'analisi di sistemi complessi di comportamento, nell'ambito delle scienze naturali e sociali.

Swarm nasce nel 1995, nel Santa Fe Institute (New Mexico, USA), con l'obiettivo di permettere ai ricercatori di sviluppare le loro applicazioni senza dovere spendere tempo su problemi tipici di programmazione informatica (ad esempio, la generazione di grafici o di un'interfaccia grafica, o i modi di salvare i risultati delle simulazioni).

Questa biblioteca di funzioni è sviluppata tramite classi e metodi, secondo l'impostazione della programmazione ad oggetti, che utilizza classi astratte per realizzare esemplari (*instance*) specifici; esemplari che reagiscono a ordini o messaggi (i metodi). I linguaggi di programmazione utilizzati per scrivere le applicazioni sono stati l'Objective-C e successivamente Java.

Le librerie svolgono due funzioni principali:

- possono essere utilizzate dai costruttori dei modelli per creare oggetti direttamente dalle classi in esse contenute;
- possono essere utilizzate per creare sottoclassi specializzate in particolari funzioni, in relazione al sistema che si desidera modellizzare.

Gli elementi che caratterizzano un'applicazione Swarm sono:

1. gli oggetti che devono essere costruiti all'interno del modello;
2. gli oggetti relativi alla gestione del tempo e cioè le sequenze degli eventi;
3. il caricamento congiunto degli oggetti che rappresentano gli agenti della simulazione e di quelli che gestiscono gli eventi nel tempo.

Fra gli oggetti costruiti nel modello hanno particolare rilievo gli agenti, questi, interagendo fra di loro e rispondendo agli stimoli, danno vita ad uno sciame da cui il nome Swarm.

Ogni sciame contiene un orologio indipendente da quello degli altri sciame, che può scandire in modo diverso il tempo della simulazione definito dall'orologio globale cui tutti gli agenti fanno riferimento. La tipica struttura di simulazione scritta in Swarm prevede la definizione di due livelli di astrazione: da un lato abbiamo l'osservatore che, attraverso opportuni strumenti grafici, osserva l'andamento della simulazione nel tempo; dall'altro lato il modello stesso ed il suo funzionamento.

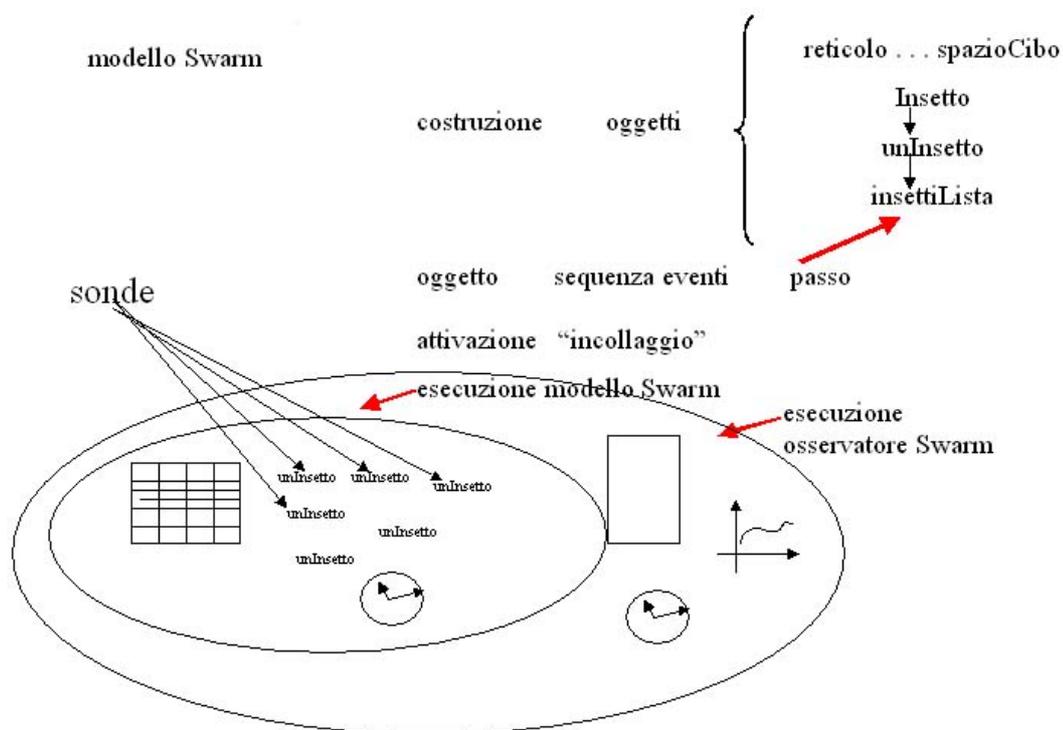


Figura 2: struttura esemplificativa di una simulazione con Swarm (Terna 2003).

Un modo per costruire gli oggetti che rappresentano gli agenti all'interno della simulazione può essere quello proposto dallo schema ERA (Environment-Rules-Agents), riportato nella figura 3.

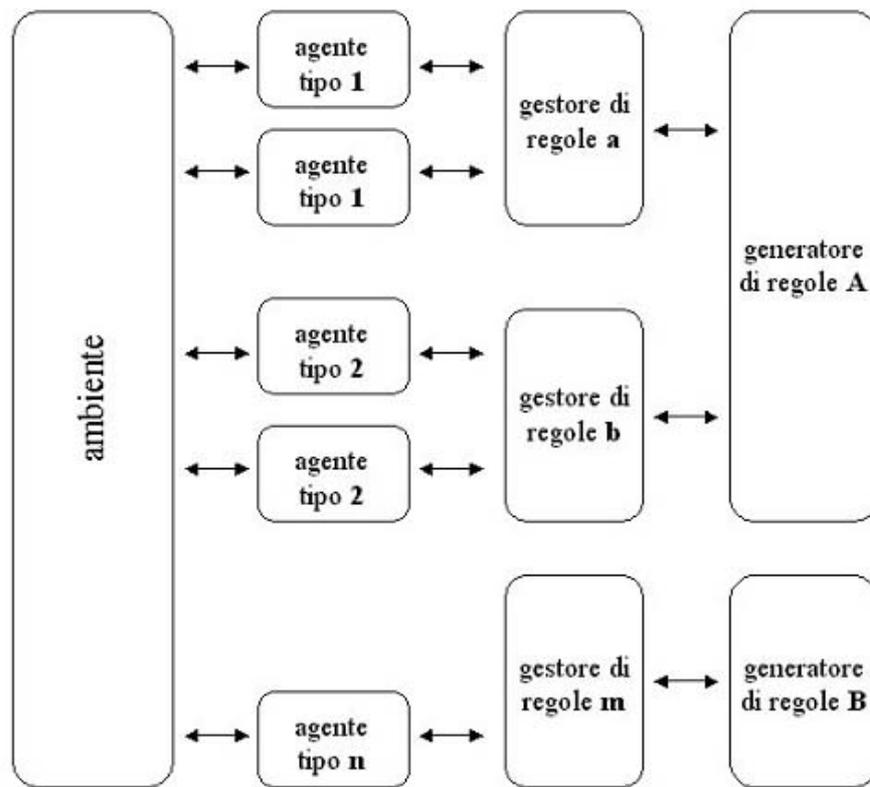


Figura 3: schema ERA, Environment-Rules-Agents (Terna 2003).

Questo schema prevede la gestione di quattro diversi strati nella costruzione del modello e degli agenti.

Primo strato: rappresenta l'ambiente in cui gli elementi sono chiamati ad interagire. Si tratta della classe ModelSwarm nella quale si definiscono gli agenti, se ne strutturano le liste, si individuano gli eventi nel tempo e si chiariscono le regole di interazione tra gli agenti (i metodi).

Secondo strato: è quello degli agenti, costruiti come esemplari di una o più classi.

Terzo strato: rappresenta le modalità attraverso cui gli agenti decidono il proprio comportamento. Gli agenti, ad ogni scelta, interrogano il gestore di regole (RuleMaster) comunicandogli i dati necessari e ottenendo le indicazioni di azione.

Quarto strato: rappresenta la costruzione delle regole. Come gli agenti interrogano i gestori di regole, questi interrogano i generatori di regole (RuleMaker) per effettuare le proprie azioni.

3.4 Il modello jES: java Enterprise Simulator

Nella simulazione di impresa gli eventi si svolgono in una sequenza di interazioni non controllata a priori dal programmatore che ha progettato il modello. E' proprio questo il significato dell'espressione simulazione "ad agenti" che si contrappone alla simulazione "di processo". Lo scopo della simulazione infatti non è quello di descrivere minutamente gli eventi a priori osservandone poi lo svolgersi nel tempo, bensì quello di fare "funzionare" effettivamente la realtà aziendale simulata grazie all'interazione fra le sequenze produttive e le unità produttive in grado di compiere le prime.

La complessità scaturisce proprio dall'intrecciarsi dei due elementi principali, totalmente indipendenti, del modello jES: le ricette e le unità.

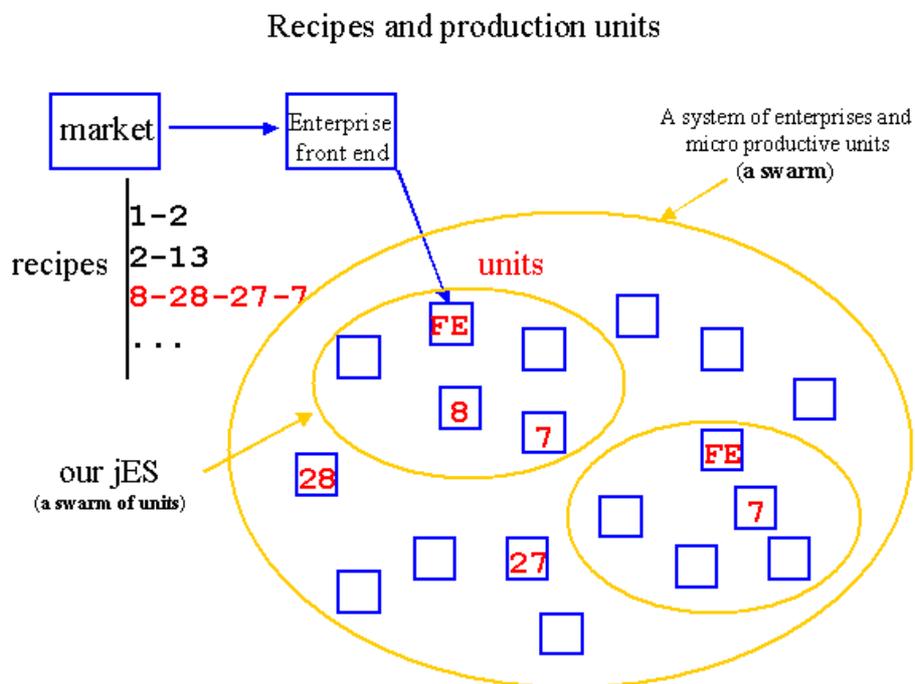


Figura 4: schema di jES (Terna 2003).

La figura 1 descrive alcuni esempi di ricette, numeri rappresentativi delle fasi di lavorazione necessarie per la produzione, e alcuni esempi di unità produttive capaci di svolgere ciascuna un passo o diversi

passi delle ricette stesse, o di operare come interlocutori del mercato: i cosiddetti front end.

Nell'ambito aziendale le unità e gli ordini non sono proprio gli agenti ma costituiscono un contesto ad oggetti. Gli agenti sono pezzi di software autonomi che sono in grado di svolgere determinate operazioni. Nel modello jES questi prenderanno vita nel momento in cui sarà possibile inserire un operatore che prenda le decisioni: un algoritmo genetico o un umano.

Il formalismo relativo alle ricette ed alle unità verrà esposto dettagliatamente nel capitolo seguente.

La simulazione inizia con la generazione casuale degli ordini di produzione, le ricette, si ottiene così una serie di vettori di numeri casuali che simboleggiano i prodotti da realizzare e costituiscono l'insieme delle operazioni da compiere. Questi consentono all'impresa di individuare il percorso da seguire per completare la produzione.

Ogni unità produttiva, quando prende in carico un ordine, lo accoda nella propria lista di ordini da eseguire secondo la modalità cosiddetta FIFO (First In First Out); dopo l'esecuzione richiede all'ordine, l'informazione necessaria per trasmetterlo ad una successiva unità.

Ogni unità produttiva è dotata di un proprio magazzino per raccogliere la produzione in eccesso, se presente.

Qualora il livello delle scorte presenti sia inferiore al quantitativo minimo stabilito l'unità produce per il magazzino fino a raggiungere il livello massimo di scorte consentito.

L'introduzione di scorte di semilavorati è stata necessaria per ridurre le code nelle liste di attesa delle unità produttive, d'altro canto un accumulo eccessivo di scorte può creare problemi di spazio e di costi finanziari, sia per la realtà aziendale che per la simulazione.

Ogni unità, infatti, è dotata di un sistema di contabilità. La unit computa per ogni ciclo produttivo un costo fisso ed un costo variabile nel seguente modo:

- i costi fissi vengono computati giornalmente, indipendentemente dall'attività svolta dall'unità;
- i costi variabili vengono computati quando l'unità produce per l'ordine o per il magazzino.

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

Ai magazzini viene imputato anche un costo finanziario, che dipende dalle quantità che giornalmente rimangono inutilizzate, sulla base di tre criteri:

1. a costi variabili;
2. a costi fissi più costi variabili;
3. con lo stesso criterio dei prodotti finiti.

La contabilità viene registrata anche dal lato degli ordini in termini di costi fissi e variabili calcolati nel momento del passaggio attraverso le unità. Seguendo questa metodologia i costi delle unità e degli ordini vengono registrati nel medesimo momento se l'impresa non fa uso di magazzini, altrimenti in due momenti separati.

Gli ordini forniscono anche l'informazione relativa ai ricavi ottenuti dall'impresa, calcolati come prodotto tra la lunghezza della ricetta ed un tasso di ricarico predefinito.

La contabilità degli ordini ci fornisce due importanti informazioni:

- quanto costa uno specifico bene;
- quanto si spende in totale per produrre;
- quanto l'impresa riesce a realizzare in termini di prodotti finiti e di semilavorati.

In questo momento, la contabilità del modello jES è ancora in fase di sviluppo.

Il modello, in questa formalizzazione, può funzionare in tre diverse versioni:

- senza magazzini;
- con i magazzini;
- con i magazzini e le news.

Le news sono state introdotte per simulare la circolazione delle informazioni all'interno dell'impresa, nella versione del modello descritta in queste pagine, sono utilizzate unicamente per fornire indicazioni riguardanti l'order, il messaggio consiste nell'indicare alle unit con chi possono o devono comunicare per anticipare il processo produttivo.

Passiamo ora alla descrizione del codice. Le varie versioni del modello sono denominate jesframe-x.y.z, dove x.y.z rappresenta il numero della versione.

Le principali classi del modello sono le seguenti:

- StartESFrame.java
- ESFrameObserverSwarm.java;
- ESFrameModelSwarm.java;
- OrderGenerator.java;
- Order.java;
- Unit.java;
- InformationRuleMaster.java;
- Warehouse.java;
- InformationFlowMatrix.java;
- News.java;
- UnitParamiters.java;
- SwarmUtils.java;
- PTHistogram.java.

E' presente anche un file denominato jesframe.scm nel quale vengono impostati i parametri che si desiderano assumere per lo svolgimento della simulazione.

Tutti gli elementi della simulazione sono rappresentati da oggetti generati dalle classi appena elencate, dalle quali ereditano tutte le caratteristiche e le capacità (metodi) di svolgere azioni.

Si possono distinguere due fasi principali nella generazione dell'ambiente di simulazione tipico delle applicazioni Swarm: una fase di creazione degli oggetti e una fase di costruzione delle azioni.

Il programma ha inizio eseguendo la classe StartESFrame.java contenente il metodo Main, che ingloba le azioni che è il programma dovrà compiere permettendo l'avvio della simulazione.

Si procede, in questa fase, alla creazione di un'istanza dell'oggetto *ESFrameObserverSwarm* richiamando il file *jesframe.scm* di cui accennato sopra. In questo modo i valori di default della simulazione possono essere modificati senza dover ricompilare l'intera classe ogni volta:

```
eSFrameObserverSwarm = (ESFrameObserverSwarm)
Globals.env.lispAppArchiver.getWithZone$key(Globals.env.globalZone,
"eSFrameObserverSwarm");
```

La simulazione inizia in questo punto con:

- il richiamo del metodo *buildobjects()* necessario per costruire tutte le parti di cui è composto l'*Observer*

```
eSFrameObserverSwarm.buildObjects ();
```

- il richiamo del metodo *buildActions()* necessario per determinare le azioni che svolgerà l'*Observer*

```
eSFrameObserverSwarm.buildActions ();
```

- il richiamo del metodo *activateIn()* necessario per attivare tutto ciò che è stato creato

```
eSFrameObserverSwarm.activateIn (null);
```

- il richiamo del metodo *go()*, ereditato dalla classe *swarm.simtoolsgui.GUISwarmImpl*, con il compito di avviare tutte le attività e permettere all'utente di gestirle tramite il pannello di controllo

```
eSFrameObserverSwarm.go ();
```

- il richiamo del metodo *drop()* per terminare le attività

```
eSFrameObserverSwarm.drop ();
```

- `System.exit(0);` per chiudere i grafici e la simulazione.

L'Observer contiene il Model e gli oggetti grafici per la rappresentazione dell'andamento dell'impresa virtuale:

- `public int displayFrequency` definisce il numero di volte che i grafici devono essere aggiornati nel corso della simulazione;
- `public boolean verboseChoice` se vera consente di mostrare le descrizioni degli avvenimenti che si susseguono;
- `public ActionGroup displayActions` questa variabile di tipo `ActionGroup` farà da contenitore alla sequenza di eventi grafici;
- `public Schedule displaySchedule` contiene un esemplare della classe `Schedule` che servirà per gestire le sequenze temporali nell'Observer;
- `public ESFrameModelSwarm eSFrameModelSwarm` è lo Swarm che noi osserveremo, è un esemplare della classe `ESFrameModelSwarm`;
- `public PTHistogram ptHistogram` è una variabile di tipo `PTHistogram` che servirà a realizzare l'istogramma delle code presso le unità ed i magazzini.

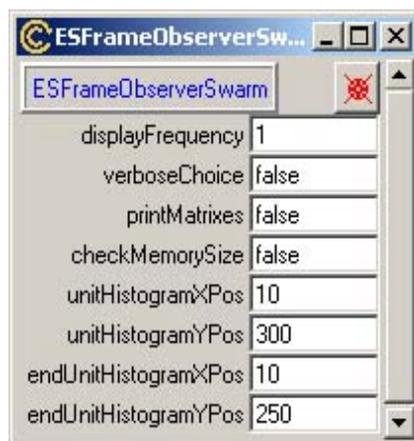


Figura 5: parametri dell'Observer.

A questo punto si creano gli oggetti per la rappresentazione a video del Model e per effettuare le rilevazioni:

```
public Object buildObjects () {
```

successivamente si creano in sequenza:

- il `eSFrameModelSwarm`

```
eSFrameModelSwarm =  
(ESFrameModelSwarm)Globals.env.lispAppArchiver.getWithZone$ke  
(getZone(), "eSFrameModelSwarm");
```

- le sonde, in per l'analisi e la gestione dei parametri della simulazione, sia del `sEFrameModelSwarm` sia del `sEFrameObserverSwarm`;

`getControlPanel().setStateStopped()`; comunica al pannello di controllo di attivare per la prima volta il comando di Stop: la simulazione sarà avviata dall'utente, una volta valutati i parametri da utilizzare, attraverso i comandi rappresentati nella figura 6.



Figura 6: pannello di controllo.

StartESFrame.verbose=verboseChoice; comunica allo `StartESFrame` la decisione se mostrare tutte le descrizioni degli avvenimenti o meno.

A questo punto il `eSFrameModelSwarm` costruisce i propri oggetti.

Al termine del metodo `ESFrameObserverSwarm.buildObjects()` si sono descritti sia gli oggetti necessari per avviare il modello sia quelli necessari ad osservarlo.

Occorre ora creare all'interno del `ESFrameObserverSwarm` le azioni per avviare la simulazione.

Una volta definite le azioni da svolgere e creato lo `Schedule` si inserisce un'istanza del gruppo di azioni nell'istanza dello `Schedule` medesimo:

```
for (i=0;i<displayFrequency;i++)
    {
        if(i==displayFrequency-1)
            displaySchedule.at$createAction (i, displayActions1);
        displaySchedule.at$createAction (i, displayActions2);
    }
return this;
```

le azioni saranno svolte dal gestore degli eventi.

Il `SEFrameModelSwarm` contiene le unità e tutti i relativi strumenti utilizzabili, come ad esempio i magazzini, necessari per dialogare con gli inventari, o le notizie.

È fatta la dichiarazione del tipo di variabile per quelli che saranno tutti i parametri della simulazione:

- *public int ticksInATimeUnit*
il numero di ticks in un giorno;
- *public int totalUnitNumber*
il numero di unità che utilizziamo nella simulazione;
- *public int totalEndUnitNumber*
il numero di end-units che utilizziamo nella simulazione;
- *public int totalLayerNumber*
il numero totale di layer;
- *public int maxStepNumber*
il numero massimo di passi da eseguire per completare un ordine;
- *public int maxStepLength*
la lunghezza massima di un passo contenuto in una ricetta;
- *public int maxTickInAUnit*

il numero massimo di ticks che un prodotto può passare in una unità;

- *public int maxInWarehouses*

la quantità massima di scorte presenti in un magazzino;

- *public int minInWarehouses*

la quantità minima di scorte presenti in un magazzino;

- *public boolean useWarehouses*

la scelta se usare (true) oppure no i magazzini e le scorte;

- *public boolean useNewses*

la scelta se usare oppure no le notizie;

- *public int infDeepness*

l'intensità della propagazione dell'informazione (quanti passi esaminiamo dopo la fase di produzione corrente);

- *public float revenuePerEachRecipeStep*

la stima di ricchezza dell'impresa ad ogni passo della ricetta;

- *public float inventoryFinancialRate*

il tasso annuo utilizzato per valutare il costo delle scorte;

- *public int nOfNewsesToProduce*

il numero di notizie necessarie per decidere la produzione di scorte;

- *public int nOfNewsesToBeCleared*

il numero di notizie che devono essere cancellate dopo la decisione di produrre per aumentare le scorte;

- *public int nOfOrdersInNewses*

il numero di ordini per cui le informazioni sono propagate da un'unità;

- *public int inventoryEvaluationCriterion*

il criterio di valutazione delle scorte;

- *public int totalProductionTime*

il tempo totale di produzione richiesto dall'ordine terminato;

- *public int totalRecipeLength*

la lunghezza totale della ricetta dell'ordine terminato;

- *public OrderGenerator orderGenerator*

il creatore d'ordini, l'order generator, che può essere rimpiazzato dall'order distiller o in futuro da una struttura più sofisticata;

- *public boolean useOrderDistiller*

la scelta se usare l'order distiller al posto dell'order generator o no;

- *public OrderDistiller orderDistiller*

uso dell'order distiller, usando archivi di ricette e di sequenze di ordini per generare gli eventi simulati;

- *public ActionGroup modelActions1, modelActions2, modelActions2b, modelActions2generator, modelActions2distiller, modelActions3*

l'oggetto di tipo ActionGroup necessario per mantenere una sequenza ordinata di eventi;

- *public Schedule modelSchedule*

lo Schedule operante nel SEFrameModelSwarm;

- *public InventoryRuleMaster inventoryRuleMaster*

il gestore di regola che controlla le scorte;

- *public InformationRuleMaster informationRuleMaster*

il gestore del flusso di informazioni (notizie);

- *public UnitParameters unitParameters*

la classe per l'immissione dei parametri delle unità;

- *public ListImpl unitList*

la lista indicante le unità produttive attivate;

- *public ListIndex unitListIndex*

gli iteratori della lista delle unità;

- *public ListImpl warehouseList*

la lista indicante i magazzini attivati;

- *public ListImpl orderList*

l'intera lista degli ordini;

- *public ListIndex orderListIndex*

gli iteratori della lista degli ordini;

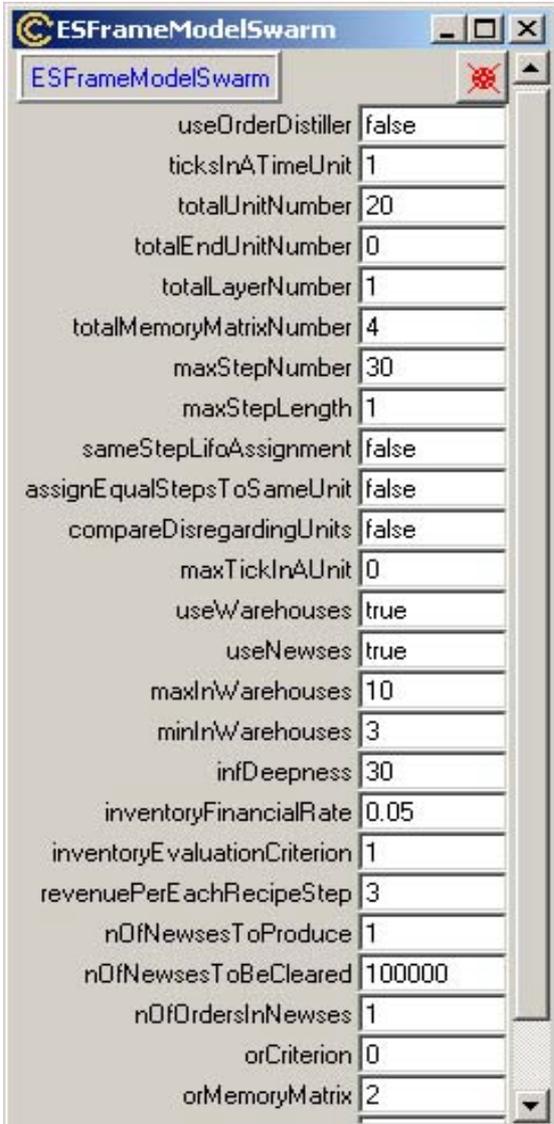
- *public ListImpl endUnitList*

la lista delle end-units.

Il costruttore della classe ESFrameModelSwarm ha il compito di definire quali siano i principali parametri da passare all'oggetto quando questo è creato.

Successivamente vengono inizializzate le variabili della simulazione con i valori di default uguali a quelli contenuti nel file jesframe.scm, che verranno visualizzati nel pannello (figura 7)

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



Parameter	Value
useOrderDistiller	false
ticksInATimeUnit	1
totalUnitNumber	20
totalEndUnitNumber	0
totalLayerNumber	1
totalMemoryMatrixNumber	4
maxStepNumber	30
maxStepLength	1
sameStepLifoAssignment	false
assignEqualStepsToSameUnit	false
compareDisregardingUnits	false
maxTickInAUnit	0
useWarehouses	true
useNewses	true
maxInWarehouses	10
minInWarehouses	3
infDeepness	30
inventoryFinancialRate	0.05
inventoryEvaluationCriterion	1
revenuePerEachRecipeStep	3
nOfNewsesToProduce	1
nOfNewsesToBeCleared	100000
nOfOrdersInNewses	1
orCriterion	0
orMemoryMatrix	2

Figura 7: parametri del Model.

Si impostano le seguenti attività:

- creazione di un'istanza della classe UnitParameters che contiene i metodi relativi alla gestione della contabilità dei costi fissi e variabili;
- lettura tramite i metodi propri della classe UnitParameters dei costi fissi e variabili dai file FixedCosts.txt e VariableCosts.txt;
- controllo del limite massimo di scorte possibili in magazzino, se pari a zero allora verrà attribuito il massimo valore intero possibile;
- creazione di un'istanza della classe ListImpl che registrerà unità (il numero assegnato alle unità) attivate;
- creazione di un'istanza della classe ListImpl che registrerà i magazzini assegnati alle unità attivate;
- creazione di un'istanza della classe ListImpl che registrerà gli ordini esistenti ad ogni passo di simulazione;
- creazione di un'istanza della classe ListImpl che registrerà la lista delle end-units;
- creazione di un'istanza della classe OrderGenerator che creerà una ricetta di lavorazioni, scegliendo casualmente sia la lunghezza sia il contenuto;
- creazione di un'istanza della classe InventoryRuleMaster che è il gestore delle regole per le scorte;
- creazione di un'istanza della classe InformationRuleMaster che è il gestore delle regole per il flusso di informazioni;
- con l'utilizzo di un ciclo iterativo sono create tante unità quanto è il valore di totalUnitNumber e se attivati anche i magazzini relativi.

Viene creato adesso un ActionGroup che comprende una lista di azioni da gestire definite nelle classi Unit e OrderGenerator:

- UnitStep1;

- UnitStep2.

Prima di procedere si fa un controllo, se si utilizza l'OrderGenerator si crea un ordine in modo casuale mentre, se si utilizza l'OrderDistiller, gli ordini verranno presi dai file relativi.

Tornando ai passi di cui sopra, UnitStep1 rappresenta il momento della produzione giornaliera, la unit controlla la lista degli order in attesa, se è piena effettua la lavorazione sul primo order della lista, utilizzando, se disponibili, le scorte, altrimenti interroga l'Inventory Rule Master circa l'opportunità di produrre per il magazzino.

In UnitStep2 la unit controlla il contenuto della propria waitingList ed invia le informazioni riguardanti il primo order che subirà la lavorazione alle unità successive interrogando l'Information Rule Master per verificare se sia possibile effettuare la comunicazione. Se invece l'order è completo è possibile dichiarare terminata la sua produzione.

Una volta che abbiamo definito le azioni da svolgere e che abbiamo creato lo Schedule l'avvio del tempo si avrà in zero considerando tutto ciò che compone modelActions, ottenendo così una parvenza di gestione parallela degli eventi in realtà definiti in sequenza.

ESFrameObserverSwarm, Model e schedule saranno attivati all'interno dell'Observer.

E' necessario introdurre le due differenti situazioni in cui possiamo far girare la simulazione:

- 1) utilizzando l'orderGenerator;
- 2) utilizzando l'orderDistiller.

Se utilizziamo l'orderGenerator, quando stiamo analizzando le code o riproducendo una situazione in cui non abbiamo informazioni in merito alle sequenze degli ordini, e dunque dobbiamo generarle in modo casuale, tutte le ricette sono generate internamente usando lo stesso intervallo di tempo (secondi, minuti, ore, ...). Coerentemente

dovrà essere usato lo stesso intervallo di tempo nelle tabelle dei file relativi alle unità per misurare i costi fissi ed i costi variabili.

Se utilizziamo l'orderDistiller, seguendo una sequenza di ordini conosciuti applicati ad un repertorio noto di ricette, potremmo avere a che fare con intervalli di tempo differenti nelle ricette: l'orderDistiller deve dunque convertire internamente tutte le misure di tempo considerate in quella più piccola. Tale misura anche in questo caso dovrà essere adottata per il calcolo dei costi fissi e di quelli variabili.

La classe Order contiene i prodotti dell'impresa virtuale che sono lavorati dalle unità: gli oggetti della classe Unit.java.

Per poter far girare la simulazione con e senza magazzini e poter gestire la presenza o meno delle news la classe Unit è dotata di più costruttori, questo è consentito dal cosiddetto polimorfismo delle classi java, descritto nel primo paragrafo di questo capitolo.

La classe UnitParameters è utilizzata per leggere i parametri relativi alle unità. Utilizzando il metodo:

```
public void readUnitData() {
```

legge i numeri delle unità, le fasi di produzione, i costi fissi e variabili dal file unitData/unitBasicData.txt, mentre legge i numeri delle end units dal file unitData/endUnitList.txt.

La classe Wharehouse è necessaria al fine dell'utilizzo sia dei magazzini sia delle scorte.

La classe InventoryRuleMaster è qui utilizzata per prendere decisioni riguardanti la gestione delle scorte.

La classe News, trattata come un oggetto, consente di trasmettere le informazioni necessarie inerenti ai passi successivi nella sequenza di produzione di un ordine.

La classe InformationRuleMaster si occupa di gestire il flusso delle informazioni nell'ambito della simulazione. Il suo compito è quello di leggere dal file `informationFlowMatrix` le informazioni e di assegnarne il valore alla matrice `informationFlowMatrix`.

L'ultima classe, PTHistogram, contiene tre costruttori per disegnare i grafici degli istogrammi (figura ...) con le code di produzione presso le singole unità e il contenuto dei magazzini.

Il primo costruttore considera la versione semplice del modello di impresa virtuale senza magazzini, il secondo la versione con i magazzini ed il terzo è stato progettato in previsione di un futuro sviluppo del modello.

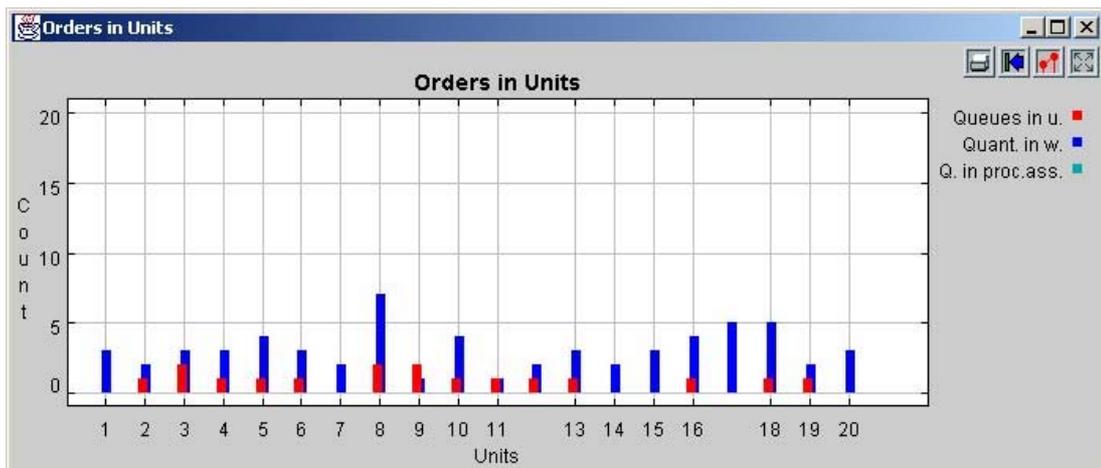


Figura 8: code nelle unità e quantità nei magazzini.

Nel capitolo successivo verrà descritta l'applicazione di questo modello ad un caso pratico.

Capitolo 4 – NIIP e la collaborazione fra imprese

Nel modello jES, descritto nel capitolo precedente, le entità che entrano in contatto tra loro sono le ricette e le unità. L'analisi era infatti incentrata su una singola azienda.

Estendendo a livello teorico la simulazione, si possono intendere le relazioni che intercorrono tra le unità produttive all'interno di un'impresa come relazioni tra imprese appartenenti ad uno stesso settore e quindi trasportare ad un livello di analisi macroeconomica le problematiche relative a questioni interne all'azienda.

Nel progetto NIIP, infatti, le entità sono rappresentate da imprese diverse che interagiscono scambiandosi informazioni e costruendo opportunità che singolarmente non saprebbero cogliere.

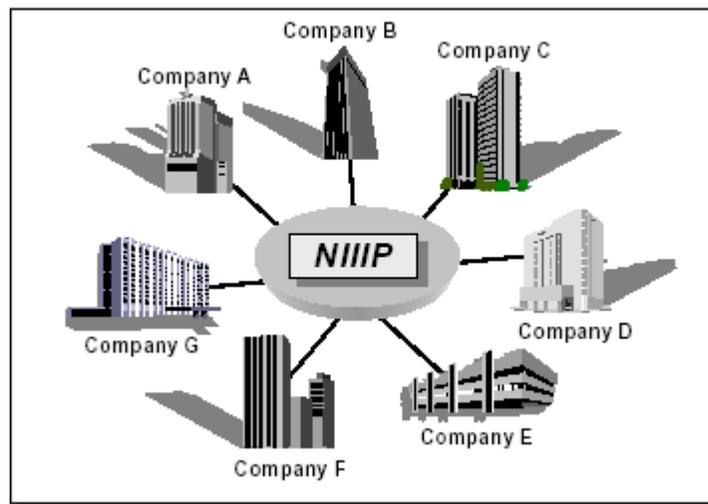


Figura 1: Virtual Enterprise Integration.

NIIP è l'esempio più noto di gestione di impresa virtuale come somma o intersezione di più imprese; con jES si può simulare il comportamento di parti di imprese che interagiscono formando una impresa virtuale.

Questa struttura richiama quella dei distretti industriali dei quali si tratterà nell'ultimo paragrafo; con jES possiamo simulare l'interazione tra imprese anche organizzate in uno spazio territoriale integrato.

L'analisi dei distretti e del progetto NIIP di azienda virtuale industriale, inteso come aggregato di aziende reali, può fornire preziose indicazioni applicabili alla simulazione di impresa, oggetto di questo lavoro.

4.1 Il progetto NIIP

The logo for NIIP Consortium features the letters 'NIIP' in a large, bold, blue, sans-serif font. To the right of 'NIIP', the word 'Consortium' is written in a smaller, black, serif font with a subtle drop shadow effect.

Il progetto NIIP (National Industrial Information Infrastructure Protocols

consortium) nasce nel 1994 da un accordo tra il Governo americano e alcune industrie volto a sviluppare un'infrastruttura informativa che favorisse la collaborazione tra imprese di uno stesso settore produttivo al fine di incoraggiare la condivisione dei costi infrastrutturali e delle informazioni relative alle tecnologie produttive, ridurre i tempi di attesa nella catena di fornitura e di reazione ai cambiamenti.

Il modello di industria virtuale, che si delinea nel progetto NIIP, ricalca il paradigma del business-to-business e si propone di essere impiegato sul mercato americano come standard di dialogo tra le imprese.

Pur rimanendo entità separate, il grado di cooperazione tra gli aderenti al progetto, può arrivare ad assomigliare a quello che si realizza nell'ambito di un'impresa verticalmente integrata e consentire anche alle piccole imprese di competere nel mercato globale.

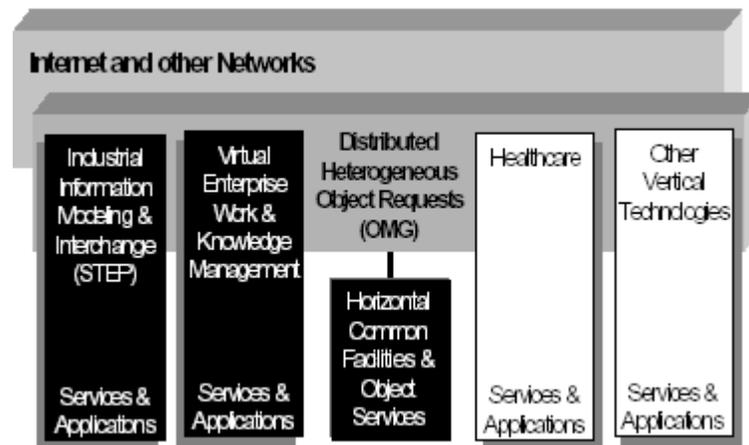


Figura 2: tecnologie ed altri segmenti verticali.

Il modello NIIP è composto da diversi progetti:

- ISEC (Integrated Shipbuilding Environment Consortium): è il progetto cui è legato lo sviluppo di componenti software, corsi di apprendimento e dimostrazioni per verificare la validità della tecnologia proposta, nell'ambito di questo progetto è previsto un sotto-progetto Electronic Commerci (EC) che si propone di valutare l'introduzione del commercio elettronico nelle catene di fornitura di componenti per la progettazione e la costruzione di navi;
- SMART (MES-Adaptable Replicable Technology): è un progetto finalizzato a consentire l'integrazione e l'interoperabilità tra i MES (Manufacturing Execution Systems);
- SPARS (Shipbuilding Partners and Suppliers): è un progetto per estendere le potenzialità dell'impresa virtuale al mondo dei cantieri navali;
- SPARS SC (Shipbuilding Partners and Suppliers Supply Chain): si propone di integrare all'interno dell'impresa virtuale le catene di fornitura del settore di produzione di navi per rendere più trasparenti le relazioni tra clienti, soci, subappaltatori e fornitori e ridurre i costi e i tempi di approvvigionamento.

4.2 L'architettura NIIP

L'obiettivo principale del National Industrial Information Infrastructure Protocols Consortium è lo sviluppo di una piattaforma tecnologica per la realizzazione della *Industrial Virtual Enterprise*.

Per perseguire tale obiettivo, il consorzio ha sviluppato un'architettura (figura 3) aperta e con tecnologia non proprietaria in cui si identificano quattro elementi fondamentali:

- Protocolli di comunicazione comuni;
- Tecnologia ad oggetti uniforme per garantire la comunicazione tra i sistemi e tra le applicazioni;
- Modello comune di scambio delle informazioni;
- Gestione cooperativa dei processi integrati della *virtual enterprise* (VE).

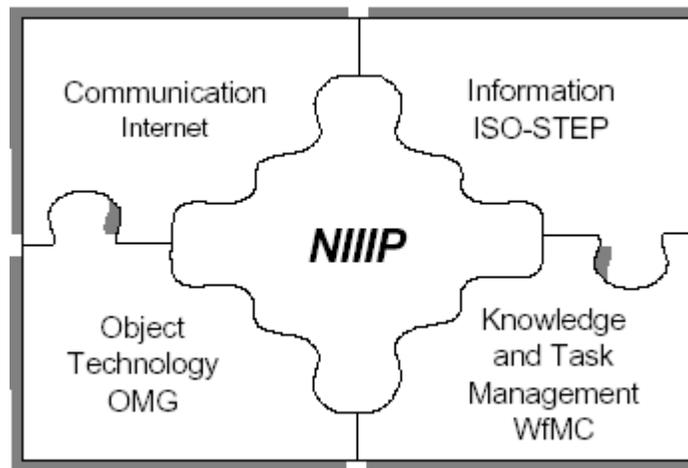


Figura 3: l'architettura NIIP.

Il primo livello di definizione delle tecnologie che consentono di dar vita alla VE è costituito dallo stato che garantisce lo scambio di informazioni. Internet è ridefinito all'interno di questo paradigma come "la super-autostrada nazionale delle informazioni".

Esso è l'elemento chiave di questa tecnologia. Sulla base del paradigma del NIIP, un punto della VE invoca un servizio con l'invio di un messaggio attraverso questa infrastruttura. Esso è implementato attraverso un componente software compatibile con la tecnologia CORBA 2.0 (Common Object Request Broker Architecture). Ogni messaggio deve essere inviato:

1. Confidenzialmente - senza divulgazione a soggetti terzi non autorizzati;
2. Integralmente - con la certezza che i dati non vengano alterati;
3. Autenticamente - la fonte deve essere identificabile in modo attendibile.

La sicurezza delle informazioni può essere garantita dai protocolli di comunicazione di Internet (TCP/IP), all'interno dei componenti software che realizzano lo strato di interscambio delle informazioni.

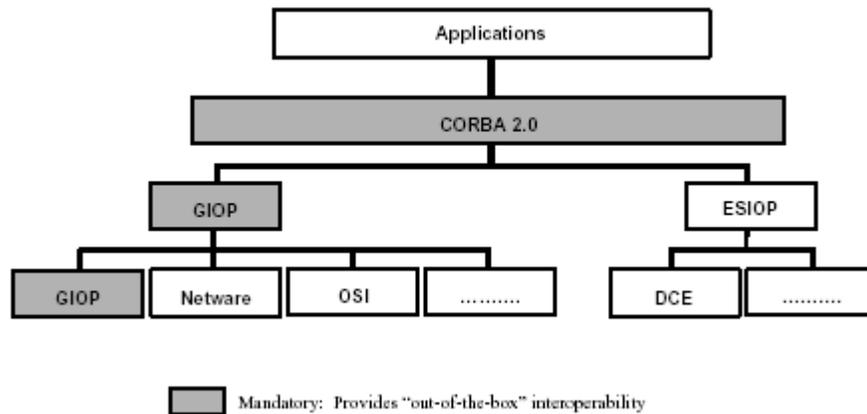


Figura 4: CORBA 2.0 Interoperability.

Il secondo livello dell'architettura definisce la tecnologia degli oggetti software. Essi devono rispondere alle specifiche denominate CORBA 2.0. Tale tecnologia permette di creare strati di software che eseguono operazioni e sono in grado di comunicare all'esterno grazie ad un

linguaggio detto IDL (Interface Definition Language). Esso fornisce agli oggetti un'interfaccia che garantisce la loro visibilità attraverso la rete. Il software adottato dal consorzio si appoggia all'Object Management Group (OMG) ed è dotato di caratteristiche di portabilità, interoperabilità e possibilità di riutilizzo.

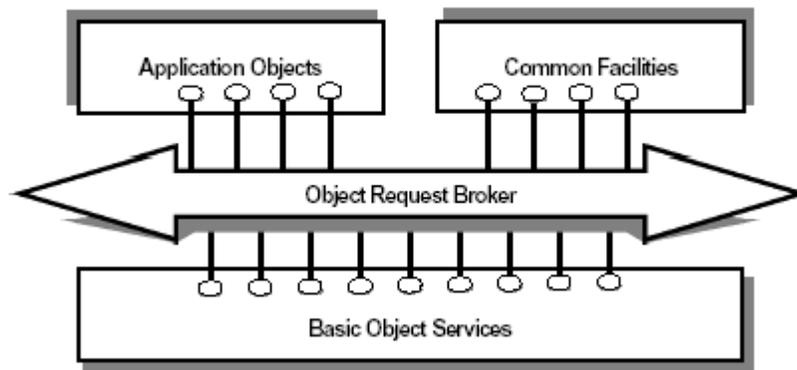


Figura 5: OMG Reference Model.

Il modello comune di scambio delle informazioni cui fa riferimento lo standard del consorzio è ISO STEP (*the international standard for exchange of products*) che richiede il rispetto di alcune specifiche.

Il protocollo stabilisce, in primo luogo, che le informazioni relative ai differenti gruppi di lavoro operino tra loro in modo da costituire un'unica struttura logica relativa al prodotto, che deve essere fornito di una propria documentazione. Tutte le applicazioni, infine, devono sviluppare i differenti aspetti del progetto, compresi i processi di produzione.

L'ultimo livello riguarda la predisposizione delle strutture di descrizione delle informazioni, che garantiscono la possibilità di collaborazione tra le aziende, attraverso la realizzazione di un modello che unifica i flussi di dati, le relazioni tra i gruppi e le applicazioni, favorendo l'integrazione fra le differenti competenze.

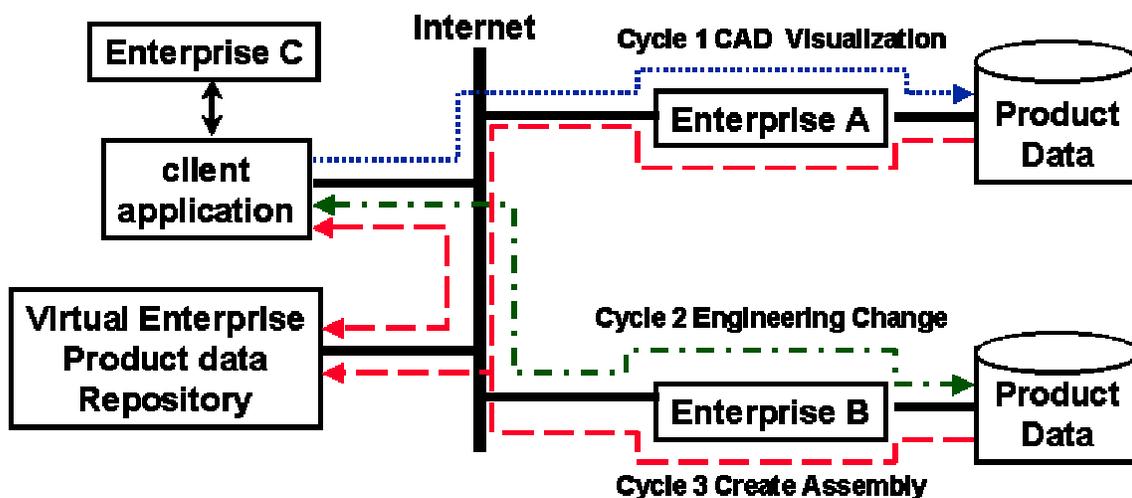


Figura 6: The challenge problem of the NIIP project from the perspective of the product data team.

Il modello è strutturato su tre cicli (esempio figura 5):

- nel primo si descrivono le risorse di ciascun componente, i dati, le associazioni tra essi, le regole;
- nella seconda fase è definito un schema globale che descrive la totalità delle risorse dell'organizzazione virtuale;
- il terzo prevede la descrizione dei meccanismi che permettono di mediare i contenuti dei singoli schemi locali nella metrica dello schema globale.

Il risultato di questo processo è la descrizione unitaria di un metamodello dell'azienda virtuale.

Il consorzio NIIP, come descritto in figura 7, svolge dodici mansioni. L'architettura di riferimento (Reference Architecture) guida lo sviluppo dei protocolli che permettono la sperimentazione con diversi sottoinsiemi di tecnologia attraverso Internet. I protocolli permettono la costruzione della pubblicità, della difesa e dei motori di ricerca.

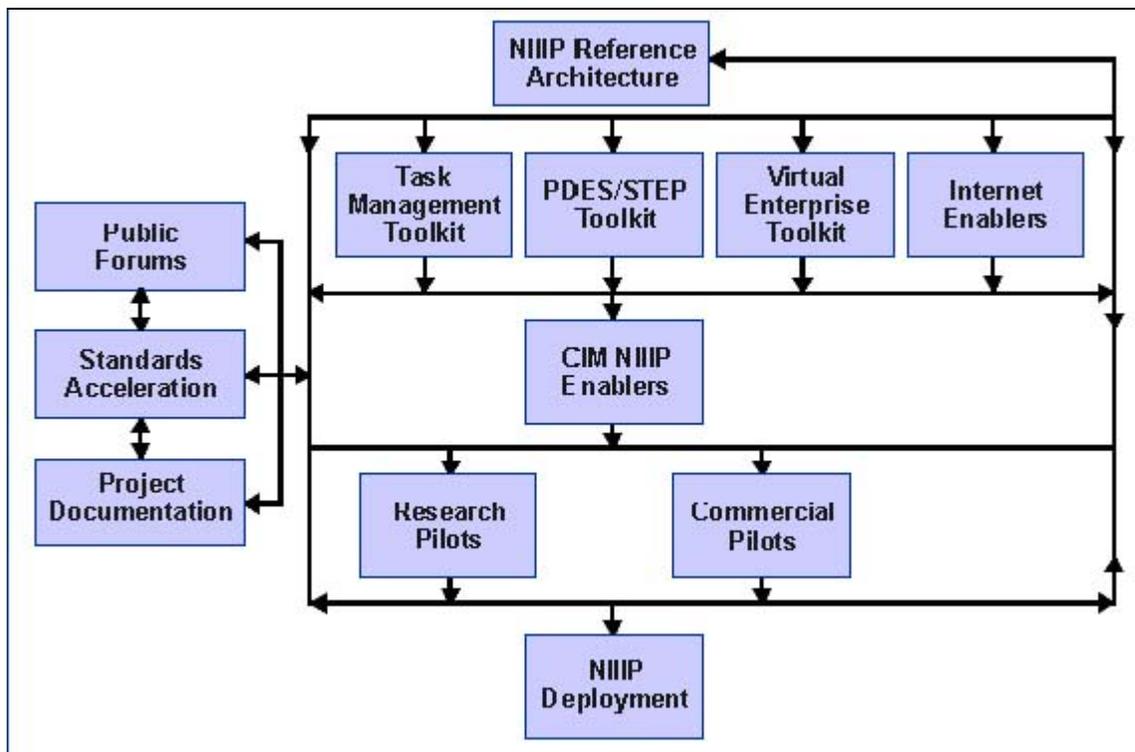


Figura 7: mansioni NIIP.

Affinché l'impresa reale prenda forma è necessario predisporre *interfaces protocols*, oggetti con attributi, stati e funzioni che consentono di definire le dotazioni dell'impresa.

Gli utilizzatori finali del progetto e gli agenti che rendono disponibili i servizi si incontrano attraverso il NIIP Desktop (figura 8), il luogo di esecuzione del lavoro dell'impresa virtuale.

I flussi di lavoro tra i componenti sono gestiti dal *workflow*, mentre un *mediator* regola il dialogo tra i vari database.

Le risorse della VE sono raccolte dal VE Member Resources, una piattaforma comune facilmente accessibile, che le rende visibili a tutti gli utilizzatori affinché possano avere una visione di insieme del sistema.

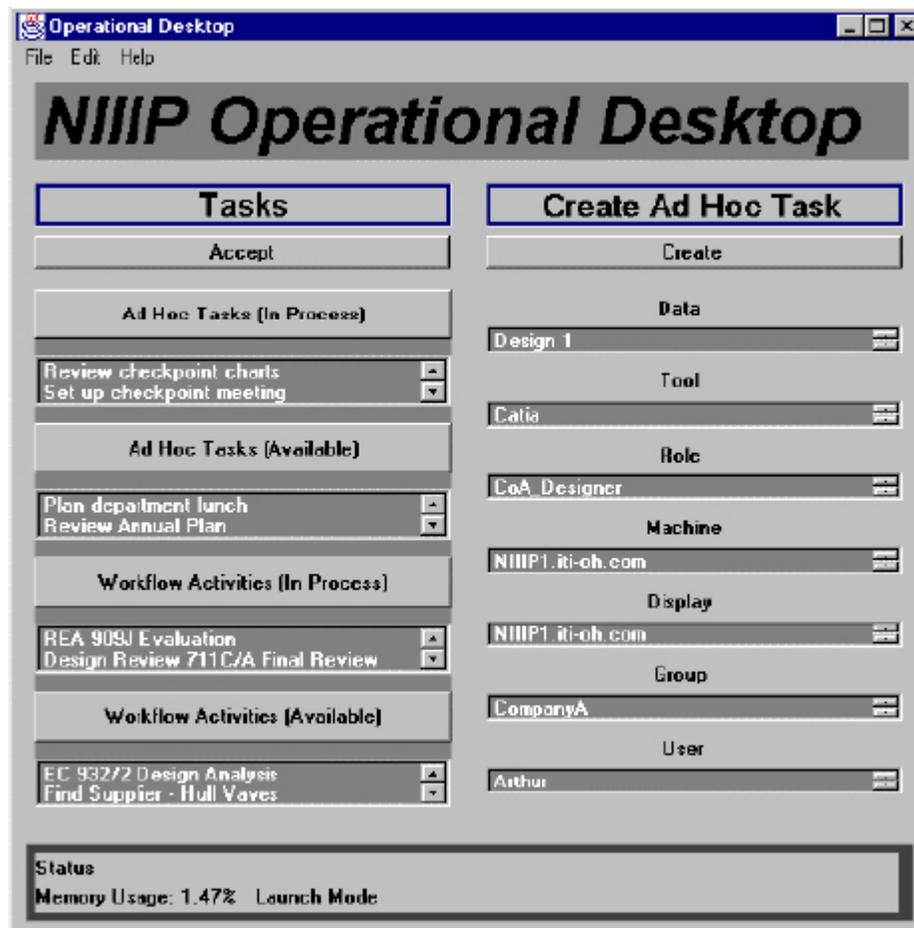


Figura 8: NIIP Desktop.

I vantaggi per le imprese che aderiscono al modello sono rilevabili in termini di competitività, accresciuta dalla collaborazione attraverso l'impresa virtuale, di facilità di accesso ai mercati senza limiti geografici e di specializzazione sul prodotto, data l'interazione tra più imprese in diverse fasi della catena del valore.

Il potenziale difetto di questa metodologia è la scarsa flessibilità del risultato derivante dalla necessità, per i progettisti del paradigma NIIP, di scegliere alcune tecnologie e definire gli standard per una realtà in continua evoluzione.

4.3 I distretti industriali

Il termine distretto industriale venne coniato da Alfred Marshall, nella seconda metà del XIX secolo, in riferimento alle zone tessili di Lancashire e Sheffield. La definizione che Marshall diede, in seguito, fu la seguente: « Quando si parla di distretto industriale si fa riferimento ad un'entità socioeconomica costituita da un insieme di imprese, facenti generalmente parte di uno stesso settore produttivo, localizzato in un'area circoscritta, tra le quali vi è collaborazione ma anche concorrenza. »

Dunque gli elementi individuati dall'economista inglese erano:

- l'individuazione di una specifica realtà sociale, oltre che economica;
- la specializzazione in una precisa categoria di prodotti;
- la concentrazione in un'area geografica;
- il particolare rapporto tra le imprese: allo stesso tempo collaborazione e concorrenza.

Se oggi torniamo ad osservare con il massimo interesse i distretti industriali e gli altri fenomeni locali, è perché abbiamo bisogno di capire in che modo si possono affrontare livelli elevati di complessità, senza pretendere di ridurla a priori con il calcolo, il comando o con la negoziazione.

La riscoperta dei distretti industriali, infatti, coincide con la riscoperta della complessità. I due fenomeni riemergono insieme, perché ambedue esprimono, su terreni differenti, il fallimento dell'idea di modernità ereditata dal fordismo e il bisogno di una nuova concezione.

Un distretto industriale prende forma quando un numero consistente di imprese, appartenenti allo stesso settore produttivo o a settori collegati, si addensa nello stesso luogo, utilizzando la contiguità territoriale come mezzo di relazione e di scambio. Il territorio, inteso come luogo in cui sono sedimentate cultura, storia, istituzioni condivise dagli operatori

locali, funziona come un *frame* relazionale e comunicativo, capace di integrare migliaia di intelligenze decentrate e interdipendenti. Queste ultime, interagendo tra loro, danno luogo ad un comportamento aggregato non solo organizzato, ma efficiente; così efficiente da risultare competitivo in numerosi settori dell'economia moderna.

La crescita dei distretti avviene mobilitando risorse che sono doppiamente anomale per la teoria standard: prima di tutto perché sono risorse locali, più uniche che riproducibili; e poi perché sono risorse complesse, che, sommando storia, cultura e relazioni dagli esiti imprevedibili, forniscono schemi di comportamento non calcolabili.

I distretti industriali fanno parte dell'economia moderna che può assumere forme complesse sfuggendo alla ragione strumentale ed al calcolo deterministico.

Per un certo periodo sembrava quasi che i distretti si sviluppassero solo in Italia, ma in realtà questo nuovo concetto incubava i semi di una rivoluzione concettuale di portata assai più grande e sicuramente non soltanto italiana.

La prima netta demarcazione teorica, che enuclea l'economia distrettuale, radicata nei luoghi, dalla classica economia dei settori, avviene nel 1961 con un lavoro di Giacomo Becattini dedicato al rapporto tra valore economico e settori produttivi (*industries*). Becattini critica la teoria del valore centrata sul concetto di industria, in quanto povera di tutti quegli aspetti concreti che invece sono rintracciabili nei processi di produzione localizzati. Guardando al di là del settore Becattini intravede un capitalismo diverso da quello suggerito dalla teoria convenzionale: un capitalismo che produce valore mobilitando l'intelligenza degli uomini e la loro capacità di relazione entro i contesti appropriati.

Il distretto industriale non è dunque soltanto una macchina di allocazione efficiente delle risorse, è anche una forma organizzata di

learning in action, che esplora la complessità e che si trasforma per effetto di questa esplorazione.

Il punto forte del distretto industriale, non è tanto la sua efficienza differenziale o il suo successo competitivo, ma il fatto che, nel distretto, diventa visibile il nesso attraverso cui la produzione economica e la cultura della società si alimentano e si condizionano a vicenda, ponendo al centro dell'analisi economica di *uomini* che danno vita alle imprese e quelli che ci lavorano. Questi uomini, anche quando sembrano seguire una logica semplicemente efficientistica e strumentale, in realtà hanno intrecciato in modo inestricabile i loro obiettivi di vita con il lavoro svolto nell'impresa creando, in questo modo, un tipo di economia che in quel luogo sembra del tutto naturale, ma che non potrebbe esistere altrove (Becattini 2000).

La difficoltà di riprodurre il distretto altrove nasce dal fatto che il materiale di cui è fatto non sono solo le istituzioni, le regole contrattuali, le fabbriche o le infrastrutture materiali, ma sono anche gli uomini in carne ed ossa, con il loro modo di vivere e di relazionarsi.

Anche per Sebastiano Brusco, il distretto è il luogo che mette in equilibrio dinamico le ragioni dell'efficienza produttiva, dell'equilibrio sociale e della produzione di conoscenza. Esso deve riprodurre non soltanto il dispositivo efficientistico da cui dipendono i costi e i ricavi delle imprese, ma anche le condizioni "di contorno" che consentono a quel dispositivo di funzionare, in quel luogo e non altrove. Queste condizioni di contorno comprendono le istituzioni, la cultura sociale, le relazioni fiduciarie tra singole persone, il sistema familiare e l'etica locale; insomma tutto ciò che la storia del distretto ha messo in campo per mantenere in equilibrio il delicato rapporto di forze tra economico e sociale, tra individuale e collettivo e tra passato e presente.

Nei distretti la storia procede secondo i ritmi e i modi dell'apprendimento evolutivo: non mancano momenti di discontinuità, in cui viene meno

qualche equilibrio fondamentale o muta bruscamente l'ambiente competitivo a cui il distretto è vincolato. Ma la risposta che il sistema locale dà, quando riesce a trovare sufficienti energie per vincere la sfida, cumula mille piccoli passi, mille esplorazioni minute che, sommandosi, producono ad un certo punto un cambiamento visibile. E' lo "sviluppo senza fratture" (Fuà e Zacchia 1983), che non vuol dire senza strappi, drammi, piccole e grandi rivoluzioni, ingrandite dalla ristrettezza dell'ambiente locale.

L'Italia è costellata da distretti industriali diversi l'uno dall'altro che, dunque, richiedono studi *ad hoc* per ciascuno.

Negli ultimi anni le problematiche attinenti ai distretti si sono internazionalizzate non solo perché i distretti costituiscono una forma organizzativa interessante in tutti i casi in cui si vuole innescare un meccanismo di propagazione di neo-imprenditorialità, ma anche come spiegazione delle forme localizzate assunte dallo sviluppo economico.

Con la teoria del clustering (Porter 1989) e con la "nuova geografia economica" di Krugman e Venables (1990, 1995) la teoria economica si avvicina al territorio con lo scopo di assimilarne la complessa geometria delle forme e delle storie.

I distretti industriali offrono un esempio pratico di come sia possibile realizzare una ragionevole sintesi tra organizzazione e complessità. Quest'ultima viene metabolizzata attraverso una dinamica di apprendimento distribuito, che mobilita l'intelligenza di molti attori autonomi capaci di comunicare e di interagire tra loro. Nessuno di questi attori ha abbastanza potere tecnico, normativo o negoziale da ordinare il comportamento degli altri in un programma coerente di azioni, ma ciascuno impara, nel corso dell'azione, ad esplorare il suo segmento di complessità facendo esperienza in proprio e, al tempo stesso, facendo tesoro dell'esperienza degli altri.

Di qui la creazione di un circuito di *self-organization* (Silverberg, Dosi e Orsenigo 1988) che progressivamente, attraverso prove ripetute, “impara” a rendere efficiente l’organizzazione interna del sistema ed a rispondere in modo flessibile alle domande dell’ambiente esterno.

Il comportamento del distretto nel suo insieme nasce dalla combinazione di scelte di ordinamento parziali, che consentono a ciascuna impresa di regolare il suo posizionamento rispetto agli altri, secondo le classiche dinamiche dello sciame (*swarm*). La somma di questi comportamenti semi-indipendenti e limitatamente razionali dà luogo ad un risultato ordinato, in molti casi di efficienza superiore a quello ottenibile con l’intervento di una regia centrale.

Capitolo 5 – Applicazione del modello jES al caso pratico VIR

Descriviamo il percorso per passare da una versione reale di azienda ad un modello della stessa, necessario per la simulazione. Questo passaggio è molto delicato e complesso: in primo luogo per la difficoltà di raccogliere informazioni che permettano di fotografare la realtà qual è, in secondo luogo per il non immediato processo di astrazione ed applicazione al modello jES.

Il duplice scopo di tale applicazione è da un lato quello di permettere un continuo sviluppo di jES, ed in tal modo affinare ed evolvere sempre più la simulazione; dall'altro lato quello di monitorare la struttura organizzativa dell'azienda e gli effetti che producono su di essa le decisioni adottate dal manager.

5.1 Una realtà aziendale: VIR

La VIR è un'impresa industriale specializzata nel settore delle valvole e rubinetterie attiva da più di 30 anni. E' situata a Valduggia, polo piemontese che insieme a quello bresciano rappresenta circa la totalità della produzione di valvole e rubinetterie italiane.

E' importante notare che la VIR è dotata di una propria fonderia interna, questo è un raro esempio nel settore e permette il controllo qualitativo sin dal primo passo produttivo.

I prodotti della VIR, proprio grazie alla grande esperienza ed al dinamismo di quest'ultima, vengono collocati non solo in Europa, che presto sarà un mercato domestico, ma anche in altri continenti nella misura del 60% dell'export.

Il prodotto valvola non è ad alto contenuto tecnologico, quindi l'azienda ha puntato su un'elevata tecnologia nei mezzi e nei metodi di produzione.

La fonderia, con i moderni forni ad induzione, il reparto lavorazioni meccaniche, dove le operazioni di carico e scarico delle macchine transfer avvengono tramite robot, ed infine, il reparto assemblaggio, con le sofisticate apparecchiature, progettate e realizzate dalla VIR sono le basi del successo ottenuto.

Le materie prime utilizzate dall'azienda sono l'OTTONE per l'88%, il BRONZO per il 6% e la PLASTICA, che è un materiale introdotto da poco, per il restante 6%. La nostra analisi prenderà in considerazione solo l'ottone, per l'importanza della produzione sul fatturato totale, nella prospettiva di una simulazione che necessariamente introduce delle semplificazioni.

La produzione della VIR si articola in tre principali famiglie di prodotti:

- Saracinesche (vedi figura 1)
- Valvole a sfera (vedi figura 2)
- Valvole a globo ed affini



Figura 1: saracinesca. Figura 2: valvola a sfera.

Le valvole a sfera si compongono di dieci parti di cui sono lavorate internamente all'azienda il corpo, il manicotto e la sfera.



Figura 3: esplosione di una valvola a sfera.

Il dettaglio di alcuni articoli è riportato in appendice.

Analogamente a quanto visto per le valvole a sfera, anche le saracinesche sono composte da dieci parti, ma soltanto il corpo, il cuneo e il vitone sono prodotti internamente



Figura 4: esplosione di una saracinesca.

L'analisi nel dettaglio le fasi di produzione di alcuni articoli è riportato nell'appendice 1.

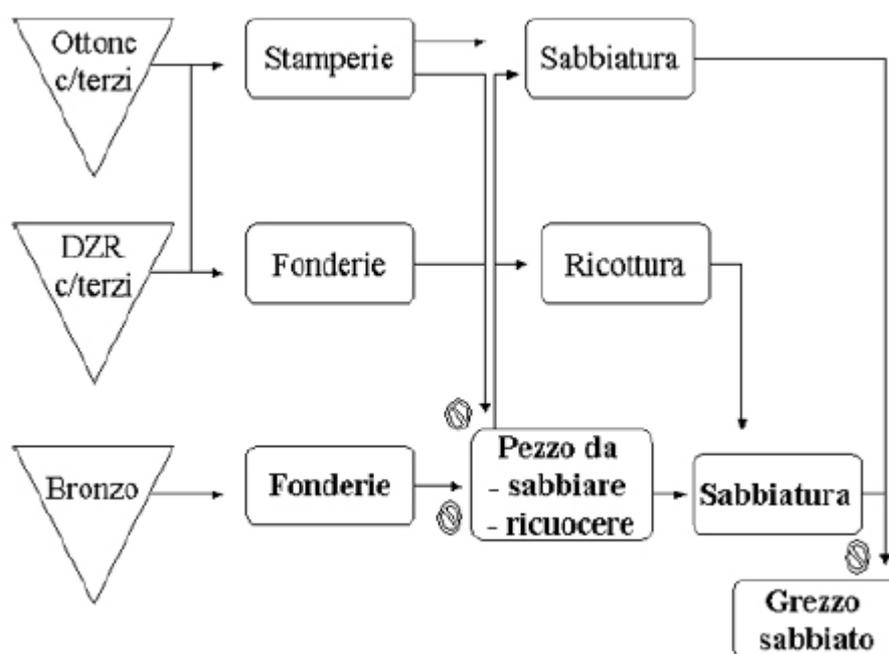
Come emerge dall'analisi del dettaglio degli articoli le due famiglie principali di prodotti sono segmentate in una molteplicità di dimensioni e sottocaratteristiche in continua evoluzione in armonia con le esigenze di mercato, ma in contrasto con le rigidità del sistema determinate dalle risorse finanziarie, dai tempi di evoluzione dei macchinari e dalle regole del mercato del lavoro.

L'azienda VIR è dotata di un reparto di torneria per le lavorazioni meccaniche di asportazione del truciolo composto da quattordici macchine e di un reparto di montaggio con quaranta macchinari di cui dodici adibite all'assemblaggio di prodotti finiti.

La produzione avviene su larga scala e le macchine sono altamente automatizzate; alcune postazioni rimangono però manuali a seguito

delle difficoltà di progettazione di macchinari in grado far fronte alla complessità di assemblaggio di prodotti in continua evoluzione. Se i macchinari sono infatti versatili, in quanto in grado di produrre articoli molto diversi tra loro, sono anche poco flessibili in quanto sono determinano tempi di attrezzaggio lunghi. Le macchine sono caratterizzate da un braccetto universale pronto a ricevere l'attrezzatura e da un supporto con il porta pezzo.

Figura 5: Fase I

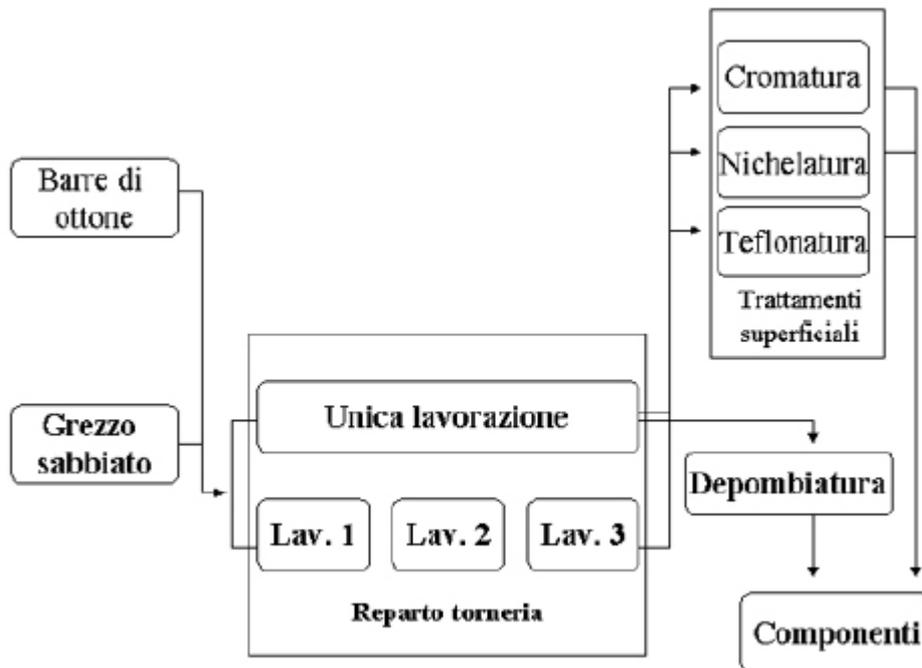


A seconda della tipologia e delle dimensioni della componente da lavorare si modifica la pinza di graffaggio e l'utensile che esegue la lavorazione.

Le fasi di lavorazione dei prodotti sono essenzialmente tre e sono comuni alle due famiglie principali.

Nella prima fase, descritta nella figura 5, si acquistano le barre e i pani di ottone che vengono prima stampati poi sottoposti a ricottura e quindi sabbiati per ottenere i pezzi grezzi da lavorare.

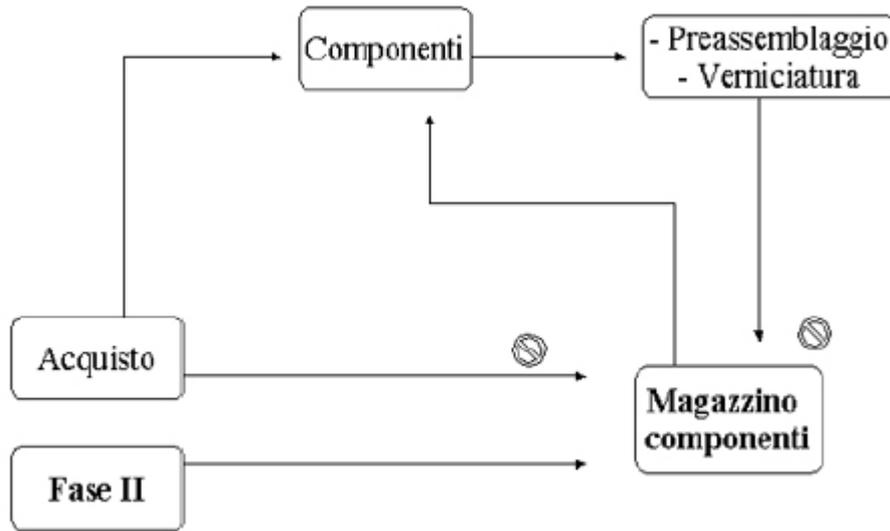
Figura 6: Fase II



Sono indicate in grassetto le lavorazioni che vengono svolte internamente all'impresa. Il simbolo che compare accanto ad alcune fasi di lavorazione indica il processo di controllo e di pesatura che viene effettuato sui pezzi da lavorare.

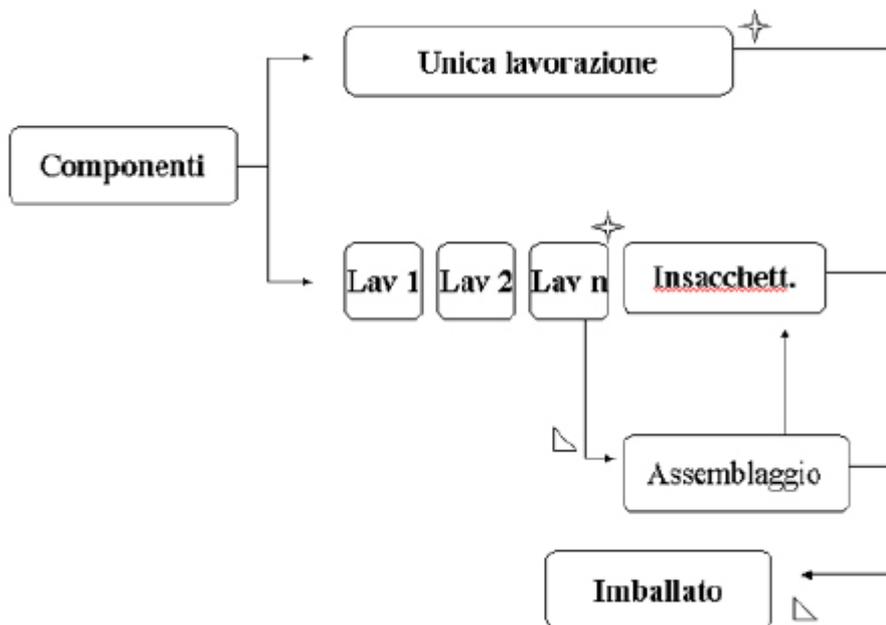
La seconda fase (figura 6) riguarda le lavorazioni meccaniche sui pezzi grezzi che interessano il reparto della torneria e i trattamenti superficiali che devono essere svolti esternamente all'azienda. In questa seconda fase il magazzino delle componenti si riempie delle parti lavorate nella fase 2 e di quelle acquistate esternamente che possono richiedere, come descritto in figura 7, la verniciatura o un preassemblaggio, entrambi svolti esternamente all'impresa.

Figura 7: Fase II b



La terza fase riguarda il reparto di montaggio in cui si assemblano le componenti lavorate internamente con quelle acquistate all'esterno. Il prodotto finito viene testato poi insacchettato ed imballato.

Figura 8: Fase III



Come mostra la figura 8 il montaggio può essere eseguito da un unico macchinario automatizzato oppure da più macchinari posti in sequenza.

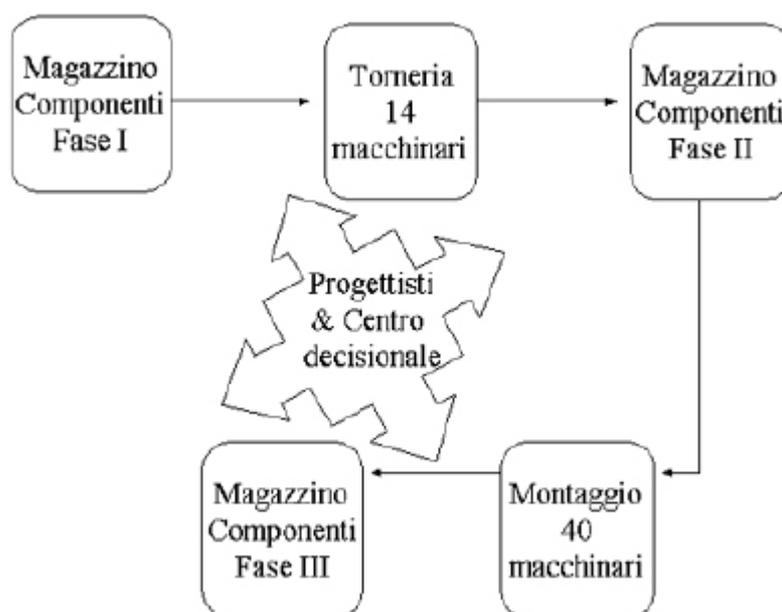
Nel primo caso il collaudo (indicato in figura 8 con una stella) avviene in linea, cioè all'ultima stazione prima dello scarico. I pezzi assemblati in questa fase vengono quindi pesati (indicato in figura 8 con un triangolo) e poi imballati.

Le sofisticate apparecchiature per l'assemblaggio ed il collaudo sono progettate e realizzate dalla VIR stessa, che in questo modo è riuscita ad ottenere impianti che si adattassero perfettamente alle esigenze di produzione e di spazio proprie dell'azienda, senza dover ricorrere ad esborsi di capitale eccessivamente onerosi.

All'interno dello stabilimento si collocano anche gli uffici dei progettisti e il centro decisionale che stabilisce le tempistiche e modalità di produzione, il vero "cuore" della VIR, che deve mettere alla prova lo schema generale del modello di impresa virtuale.

Come mostra lo schema in figura 9, la Vir è dotata di tre magazzini, due per le componenti e uno per il prodotto finito.

Figura 9: Struttura dello stabilimento della Vir

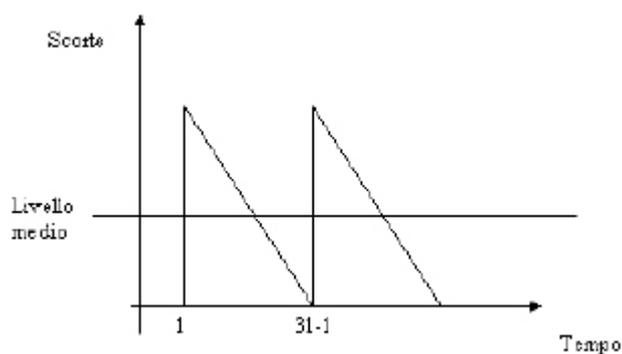


Il terzo magazzino è destinato ai prodotti imballati in attesa di spedizione; la VIR, per ragioni finanziarie e di gestione degli spazi non mantiene un livello di scorte di prodotti finiti.

Alcune componenti, comuni a più prodotti, hanno un livello di scorte superiore rispetto ad altre e sono prodotte ogni qual volta i macchinari ad esse dedicati siano poco attivati dalle necessità di produzione. In generale, però, i magazzini delle componenti sono alimentati principalmente all'inizio di ogni mese con un quantitativo stimato sufficiente per poter far fronte agli ordini che saranno evasi nel mese in questione.

L'andamento del livello delle scorte per le componenti tende ad oscillare, come descritto in figura 10, intorno ad un valore medio fissato mensilmente.

Figura 10: Andamento del livello delle scorte per le componenti



Nell'azienda VIR, il punto di partenza è l'ufficio commerciale, esso si occupa dell'acquisto dei materiali, trasmettendo gli ordini di questi ai fornitori, e della vendita dei prodotti finiti.

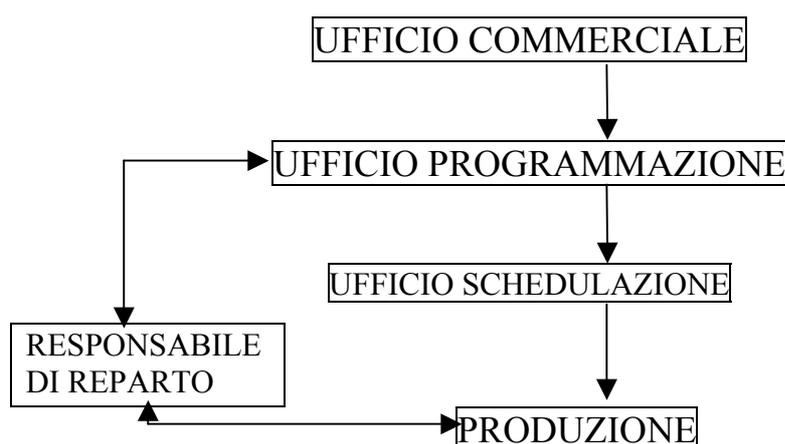
Una volta che l'ufficio commerciale ha trasmesso alla programmazione gli ordini ricevuti dai clienti, l'addetto alla programmazione, partendo dal termine della produzione ossia dalla spedizione, va a ritroso nei reparti produttivi per stilare un programma produttivo giornaliero determinando quali siano le

macchine e le risorse umane da impiegare per soddisfare le richieste provenienti dal mercato.

All'inizio di ogni mese viene stilato anche il programma di produzione sulla base degli ordini attesi e di quelli effettivamente ricevuti dall'ufficio commerciale.

A questo punto l'ufficio schedulazione si occupa della stampa e della divulgazione delle informazioni inviate alla produzione. In produzione i responsabili di reparto organizzano il lavoro e se necessario comunicano con l'ufficio di programmazione.

Figura 11: Relazioni fra gli uffici



Il programma di produzione così organizzato viene quotidianamente aggiornato sulla base delle effettive risorse disponibili e degli ordini ricevuti dall'ufficio commerciale che si possono suddividere in tre principali categorie:

1. ordini che possono essere evasi nei tempi stabiliti con il cliente anche iniziando da zero la lavorazione;
2. ordini che si possono evadere immediatamente perché vi è disponibilità di prodotti finiti grazie a previsioni stimate correttamente;

3. ordini basati su richieste che non potrebbero essere evase nei tempi stabiliti con il cliente se non si fosse prevista in anticipo una produzione di componenti che richiedono tempi particolarmente lunghi.

La correttezza delle previsioni e i tempestivi adattamenti della produzione giornaliera dipendono dal grado di comunicazione e cooperazione tra gli uffici dell'azienda, le cui relazioni sono illustrate in figura 7.

Ripercorrendo il processo produttivo a ritroso si verificheranno i seguenti passi:

1. Emissione dell'ordine di lavoro verso il reparto di assemblaggio;
2. Il reparto di assemblaggio richiederà delle componenti da montare;
3. Controllo dei magazzini componenti.

Se sono insufficienti si procede:

- a. all'acquisto all'esterno di talune parti del prodotto (bullone, maniglia, premistoppa, guarnizione premistoppa, perno, 2 seggi);
- b. all'emissione di un ordine di produzione alla torneria (per manicotto, sfera, corpo);
4. La torneria controllerà la disponibilità di pezzi grezzi da lavorare,
5. Acquisto all'esterno dei pani e delle barre di ottone.

In VIR la gestione delle risorse umane è vista come una necessità di assistenza che le singole unità lavorative hanno sia per svolgere la loro attività di produzione che per assumere le diverse “vesti” che permettono di eseguire più lavorazioni (attrezzaggi).

Sia le assunzioni sia il numero di turni vengono stabiliti sulla base dei volumi produttivi, previsti ed effettivi, e del grado di flessibilità del lavoro.

Nel reparto torneria ogni addetto è in grado di seguire più di una macchina per volta.

Il rapporto può essere di un addetto per due macchine o di due addetti per tre macchine.

Il personale è generalmente specializzato per lavorare soltanto sulle macchine cui è assegnato e nel caso sia una coppia di addetti molto spesso soltanto uno dei due è in grado di operare gli attrezzaggi.

All'interno del reparto vi sono però persone che conoscono lavorazioni e attrezzaggi di molte macchine.

Al contrario della torneria, nel montaggio non è possibile seguire contemporaneamente le lavorazioni di più macchine perché alcune stazioni di montaggio sono soltanto manuali.

Come per la torneria ogni dipendente è in grado di seguire soltanto la macchina cui è assegnato e nel caso di una coppia di addetti spesso soltanto uno è in grado di fare gli attrezzaggi

La VIR è un'azienda sottodimensionata e lo sfruttamento delle macchine a pieno regime per la torneria e solo parziale per il montaggio è condizionato dai contratti con i dipendenti e dalle impostazioni dei turni che non sono modificabili in modo repentino e ripetuto.

E' previsto inoltre il collaudo dei prodotti finiti infatti ogni valvola è sottoposta alla prova di tenuta idraulica o pneumatica prima di essere consegnata al cliente. Con le macchine automatiche di assemblaggio, il collaudo avviene in linea, cioè all'ultima stazione prima dello scarico.

Per quanto concerne la contabilità, in VIR i costi sono calcolati per processo: si determinano i costi per ogni singola fase di produzione ed i prodotti acquistano un costo pari al tempo di transito.

Si utilizza una contabilità per *centri di costo*, sono presenti centri di produzione e centri ausiliari. I costi dei centri ausiliari vengono poi “ribaltati” ai centri di produzione.

Centri di costo:

- Manodopera
- Impiegati, Dirigenti, Tecnici
- Attrezzature e Macchine
- Energia
- Combustibili

Calcolo dei singoli costi:

- Personale: si attribuiscono direttamente le ore lavorate sul centro considerato.

$\text{Salari} / \text{Ore Totali} = \text{€}/\text{h} * \text{h Centro} = \text{Costo del Pers. del Centro}$

- Impiegati: i costi sono valutati in base al valore del centro.
- Ammortamenti: si individua la quota annuale.

$\text{Quota Annuale} / \text{gg Lavorativi} = \text{€}/\text{g Costo Fisso di giorno lavorativo}$
 $\text{€}/\text{g} * \text{gg del mese} = \text{Costo mensile d'ammortamento}$

- Energia: calcolato in base alla potenza e ad un coefficiente di utilizzo.

- Combustibili: in base ai metri cubi.

Il costo totale dei centri viene imputato ai prodotti tramite un criterio di ripartizione basato sul tempo (uomo/macchina).

Per ogni pezzo si ha un tempo uomo o macchina standard, in base alla produzione ogni pezzo si moltiplica per il tempo standard.

Successivamente il costo totale del centro si divide per il risultato appena ottenuto e si determina un indice di costo. Quest'ultimo viene moltiplicato per il tempo standard del prodotto e sommando il valore dei materiali, attribuiti ai prodotti in base all'utilizzo, si arriva al costo totale del prodotto.

$$\text{Pezzi} * \text{Tempi Standard Uomo/Macchina} = \alpha$$

$$\text{Costo totale Centro} / \alpha = \text{Indice di Costo}$$

$$\text{Indice di Costo} * \text{Tempo Standard di Prodotto} = \text{Costo di Prodotto}$$

$$\text{Costo di Prodotto} + \text{Materiali} = \text{Totale Costo del Prodotto}$$

5.2 Formalizzazione per il modello jES

La realtà aziendale sopra descritta è molto complessa. Comprende infatti una molteplicità di prodotti con dimensioni e caratteristiche diverse ed un insieme di unità produttive in grado di svolgere le diverse lavorazioni con tempi ed attrezzaggi differenti.

La simulazione offre il grande vantaggio di poter descrivere sistemi complessi, come può essere un sistema aziendale, non dall'alto mirando cioè a risultati predeterminati, ma dal basso osservando le interazioni fra le parti. Ciò che è interessante infatti non è tanto il risultato finale quanto lo svolgimento del processo stesso.

Il modello jES si basa sul concetto di simulazione ad agenti e non di simulazione di processo: gli eventi non sono minutamente descritti a priori e successivamente osservati con lo scorrere del tempo, bensì

essi si svolgono in una sequenza di interazioni non controllata a priori dal programmatore che ha progettato il modello.

La complessità di jES scaturisce proprio dall'interazione fra i due principali elementi che caratterizzano tale modello: le ricette e le unità.

Nel modello le diverse fasi di lavorazione sono rappresentate da numeri, di conseguenza i prodotti sono identificati da sequenze di numeri chiamate ricette (What to Do - WD).

Nell'applicazione di jES all'azienda VIR i reali prodotti di quest'ultima non potevano essere presi in considerazione tutti e soprattutto di tutte le misure, così è stata necessaria la selezione di un campione che rappresentasse al meglio il volume di produzione reale.

La decisione è stata quella di simulare la produzione delle due famiglie di valvole principali: valvole a sfera e saracinesche differenziandole per due sole dimensioni in grandi e piccole.

Come già detto sopra questi due prodotti sono composti da dieci componenti, alcuni prodotti internamente ed altri acquistati dall'esterno.

Per effettuare la simulazione è stato necessario formalizzare le diverse fasi di lavorazione costruendone un dizionario che è riportato in appendice.

In tale schema i numeri rappresentano i possibili passi della ricetta, mentre le descrizioni indicano il tipo di lavorazione.

Vi sono alcune analogie al caso generale come le sequenze di numeri che rappresentano le componenti e le ripetizioni delle cifre relative alle fasi di lavorazione che hanno durata maggiore di uno.

Ciò che si differenzia dal modello generale è in primo luogo la generazione degli ordini: essi sono generati casualmente in jES mentre, nel caso specifico della VIR, sono contenuti in un file denominato orderSequences.xls, di cui si parlerà in seguito, prodotto in base ad una stima degli ordini realmente ricevuti dall'azienda. Un

altro problema, che emerge dall'applicazione della realtà VIR al modello generale, è quello per cui nel caso VIR le ricette sono relative a componenti che dovranno essere fra loro assemblati per concorrere a formare il prodotto finito. Tale frammentazione della ricetta principale nella sotto-ricette per le componenti determina la necessità di gestire produzioni asincrone.

Inizialmente si è valutata la possibilità di introdurre all'interno della sequenza produttiva un "valore sentinella" (ad esempio -1) seguito da un contatore ($\pm g$). Il primo serve per formalizzare la possibilità che una parte della lavorazione debba iniziare in un momento differente da quello che effettivamente le verrebbe assegnato in maniera sequenziale; il contatore viene utilizzato per determinare lo spostamento nel tempo sia in termini quantitativi, cioè il numero di unità di tempo, sia in termini qualitativi, in grado cioè di determinare la direzione sull'asse dei tempi che dovrà assumere: lo spostamento nel tempo dell'inizio della lavorazione avviene a \pm giorni dalla generazione della ricetta.

Il valore sentinella dovrebbe far fronte alla necessità di spostare nel tempo un processo produttivo riuscendo così ad ottenere il componente od il semilavorato in un momento utile al fine del rispetto della scadenza.

Con tale impostazione si possono verificare quattro diversi scenari:

- I. il segno del contatore è negativo: la lavorazione x dovrà cominciare in anticipo rispetto alla generazione della ricetta. Nella realtà VIR ciò è assimilabile al ricevimento di un ordine di prodotti composti da componenti che richiedono un tempo di produzione più lungo rispetto a quello utile per il cliente: si lavorerà su previsione sfruttando l'esperienza dell'addetto alla programmazione e si produrrà in anticipo;

Se avessimo questa sequenza produttiva...

1 - 2 - 5 - 8 - 3 - 2 - 14 - 18

...con le modifiche avremmo...

1 - 2 - 5 - (-1 -4) 8 - 3 - 2 - 14 - 18

...che significa:

8 - 8 - 8 - 8 - 8 - 8 - 8 -

1 - 2 - 5 - 8 - 3 - 2 - 14 - 18

nelle prime quattro unità di tempo si produce su previsione, poi in parallelo fino alla quarta fase

- II. il valore di g è pari a zero ovvero il segno del contatore è positivo, ma g è minore della somma dei tempi di lavorazione dall'inizio della sequenza fino al processo considerato: la lavorazione dovrà partire nel momento opportuno creando talora la necessità di avviare più lavorazioni in parallelo;

Se avessimo questa sequenza produttiva...

1 - 2 - 5 - 8 - 3 - 2 - 14 - 18

...con le modifiche avremmo...

1 - 2 - 5 - (-1 +1) 8 - 3 - 2 - 14 - 18

...che significa:

8 - 8 -

1 - 2 - 5 - 8 - 3 - 2 - 14 - 18

la quarta fase di lavorazione comincia già insieme alla seconda.

- III. il segno del contatore è positivo, ma g è maggiore della somma dei tempi di lavorazione dall'inizio della sequenza fino al processo considerato: occorre modificare la sequenza produttiva ed avviare più lavorazioni in parallelo. Nel caso VIR questa è la situazione in cui non vi è l'obbligo di rispettare la sequenza produttiva e si

preferisce effettuare una produzione di componenti dopo averne avviata un'altra che dura più a lungo. Occorrerebbe introdurre la gestione delle priorità (ad esempio spazio in magazzino);

Se avessimo questa sequenza produttiva...

1 - 2 - 5 - 8 - 3 - 2 - 14 - 18

...con le modifiche avremmo...

1 - 2 - 5 - (-1 +5) 8 - (-1 +3) 3 - 2 - 14 - 18

...che significa:

8

1 - 2 - 5 - 8 - 3 - 2 - 14 - 18

3 - 3 -

la quarta fase di lavorazione viene ritardata di una unità di tempo perché non fondamentale per eseguire la quinta fase.

IV. non ci sono "valori sentinella": nella realtà VIR è interpretabile come il punto I) di cui sopra, mentre nel modello nulla cambierebbe rispetto all'impostazione generale di jES.

Se avessimo questa sequenza produttiva...

1 - 2 - 5 - 8 - 3 - 2 - 14 - 18

...in questo modo avremmo...

(-1 0) 1 - 2 - 5 - (-1 +4) 8 - (-1 +5) 3 - 2 - 14 - 18

...che significa:

1 - 2 - 5 - 8 - 3 - 2 - 14 - 18

nella realtà nulla cambia, se non nel formalismo da adottare.

La soluzione sopra proposta non permetteva di fare fronte al problema esposto al punto I): la presenza del “valore sentinella” nella ricetta indicava la necessità che una fase di lavorazione cominciasse in anticipo, ma non era chiaro come potesse essere “lanciata” ancora prima che la ricetta stessa entrasse in produzione.

Per ovviare a tale problema è stato introdotto un sistema di procurement: grazie a questo meccanismo è possibile richiamare determinate forniture o componenti necessarie per il completamento della lavorazione in atto.

Con tale formalizzazione la sequenza della ricetta risulta strutturata nella seguente maniera:

$$n1 \text{ ts } m1 \quad p \text{ k } c1 \text{ c2 } \dots \text{ ck} \quad n2 \text{ ts } m2 \quad \dots$$

in cui i simboli sono indicativi di:

- $n1$ il numero della fase di lavorazione;
- ts il tempo che tale fase richiede (s = secondi, m = minuti, h = ore, d = giorni);
- $m1$ la quantità di s , m , h oppure d ;
- p l'indicatore di procurement;
- k il numero di componenti da reperire;
- ck la componente da reperire.

Le ricette delle componenti $c1, c2 \dots ck$ sono costruite secondo il modello tradizionale, con l'indicazione del numero della fase di lavorazione e il tempo che tale fase richiede, ma terminano tutte con una “e” ed un numero identificativo per indicare che non sono prodotti finiti, ma componenti che verranno richiamati da altre fasi.

Il formalismo è il seguente:

$$\dots n1 \text{ ts } m1 \text{ e } ck$$

Strutturando le ricette in questo modo il modello continua a funzionare in modo sequenziale: i passi della ricetta vengono eseguiti dalle unità uno dopo l'altro, per i passi che richiedono un

procurement saranno interrogate unità speciali per reperire le parti intermedie necessarie alla lavorazione.

Questa evoluzione nel formalismo delle ricette ha permesso di trattare i casi delle produzioni delle componenti che sono lavorate all'interno dell'impresa e concorreranno a formare il prodotto finale. Come accennato in precedenza, però, alcune parti delle valvole a sfera e delle saracinesche sono acquistate oppure subiscono parte delle lavorazioni all'esterno.

Nel caso della jES generale, questo aspetto non solleva alcuna difficoltà, in quanto la lavorazione esterna all'impresa è semplicemente indicata con un numero differente dalle lavorazioni svolte internamente. Questa impostazione non considera, però, i vincoli sui lotti ai quali un'impresa reale è soggetta. Nel momento in cui l'impresa si trova ad ordinare delle componenti o ad inviarne altre all'esterno per essere lavorate non può permettersi di ragionare in termini unitari, ma deve sottostare alle limitazioni sui lotti imposte dai fornitori.

Si determina quindi la necessità di una nuova evoluzione nel formalismo delle ricette che tratti il caso dei lotti, a riguardo possiamo distinguere due diverse situazioni:

◆ stand alone batch: $n_1 \text{ ts } m_1 / b_1$

i quali servono per trattare il caso di acquisto di componenti dall'esterno, la "b" rappresenta il numero di passi di produzione (pezzi) che devono essere ripetuti in un dato intervallo di tempo ed il tempo necessario per la produzione di ogni singolo pezzo è dato dalla frazione m/b , gli stand alone batch non sono seguiti da ulteriori passi;

◆ sequential batch: $n_1 \text{ ts } m_1 \quad n_2 \text{ ts } m_2 \quad \backslash \quad b_2 \quad n_3 \text{ ts } m_3$

i quali servono per trattare il caso di lavorazioni svolte esternamente, ad esempio i trattamenti superficiali descritti in precedenza nella fase II (figura 6), la "b" rappresenta il numero di passi di produzione (pezzi) che devono essere ripetuti in un

dato intervallo di tempo ed il tempo necessario per la produzione di ogni singolo pezzo è dato dalla frazione m/b .

E' da notare che per quanto riguarda gli stand alone batch è sufficiente lanciare un solo ordine della relativa ricetta per ottenere una quantità di produzione pari a "b". Per arrivare allo stesso risultato con i sequential batch invece è necessario lanciare una quantità di ordini della relativa ricetta pari a "b", questo avviene perché l'unità che effettua tale lavorazione rimane sospesa fino a che non trova nella sua waiting list un numero di ordini simili pari a "b".

Vi sono ancora due importanti formalismi da descrivere, il primo riguarda la gestione di lavorazioni diverse che portano ad ottenere lo stesso prodotto mentre la seconda si riferisce a lavorazioni che possono essere svolte in parallelo.

Il formalismo che tratta la prima situazione (OR) è così strutturato:

data la ricetta

$n_1 \text{ ts } m_1 \quad n_2 \text{ ts } m_2 \quad n_3 \text{ ts } m_3 \quad n_{22} \text{ ts } m_{22} \quad n_4 \text{ ts } m_4$

se le lavorazioni

$n_2 \text{ ts } m_2 \quad n_3 \text{ ts } m_3 \quad n_{22} \text{ ts } m_{22}$

sono tra loro alternative il formalismo della ricetta sarà:

$n_1 \text{ ts } m_1 \quad ||1 \quad n_2 \text{ ts } m_2 \quad n_3 \text{ ts } m_3 \quad ||2 \quad n_{22} \text{ ts } m_{22} \quad ||0 \quad n_4 \text{ ts } m_4$

Per determinare quale ramo scegliere sono stati introdotti diversi criteri:

- orCriterion = 0 si eseguono in sequenza entrambi i rami;
- orCriterion = 1 si esegue il primo ramo;
- orCriterion = 2 si esegue il secondo ramo;
- orCriterion = 3 si opera una scelta casuale tra i due rami;
- orCriterion = 4 si sceglie il ramo in cui l'unità che può effettuare il primo passo di produzione ha la lista di attesa più corta.

Nel caso di lavorazioni parallele (AND) avremo invece:

data la ricetta

n1 ts m1 n2 ts m2 n3 ts m3 n22 ts m22 n4 ts m4

se le lavorazioni

n2 ts m2 n3 ts m3 n22 ts m22

possono essere lavorate in parallelo il formalismo della ricetta sarà:

n1 ts m1 &&1 n2 ts m2 n3 ts m3 &&2 n22 ts m22 &&0 n4 ts m4

Le ricette che si ottengono con l'introduzione dei formalismi descritti sono raccolte nel file recipes.xls che risulta così strutturato:

- ✓ nella prima colonna il segno # indica la presenza di una riga di commento;
- ✓ nella seconda colonna viene indicato il nome della ricetta contenuta nella riga;
- ✓ la terza colonna riporta il codice della ricetta;
- ✓ dalla quarta colonna in poi sono indicate le fasi che compongono la ricetta, secondo le strutture precedentemente descritte;
- ✓ ogni riga si conclude con un “;”.

```
# LAVORAZIONI GENERICHE ;
acquistopaniottone 1000001 100001 s 173 / 500000 e 1 ;
stampaggio 1000002 p 1 1 27001 s 0 100002 s 173 \ 500000 e 2 ;
sabbiatuainterna 1000003 p 1 2 28001 s 0 100003 s 1 e 3 ;
ricotturaesterna 1000004 p 1 3 29001 s 0 100004 s 432 \ 20000 e 4 ;
sabbiaturaesterna 1000005 p 1 2 30001 s 0 100005 s 432 \ 20000 e 5 ;
preassemblaggioesterno 1000006 100006 s 432 / 5000 e 6 ;
acquistobarreottone 1000007 100007 s 173 / 500000 e 7 ;
stampaggiosfera 1000008 p 1 7 31001 s 0 100011 s 173 \ 500000 e 11 ;
```

Figura 12: file recipes.xls lavorazioni generiche.

```

# COMPONENTI ACQUISTATE ALL'ESTERNO ;
# per valvola a sfera ;
bullone 1000009 100014 s 1728 / 20000 e 14 ;
leva 1000010 100015 s 1296 / 10000 e 15 ;
perno 1000011 100016 s 2592 / 80000 e 16 ;
premistoppa 1000012 100017 s 2592 / 175000 e 17 ;
guarnizionepremistoppa 1000013 100018 s 2592 / 175000 e 18 ;
seggio 1000014 100019 s 1728 / 115000 e 19 ;
rivestimentoplastico 1000015 100020 s 1296 / 10000 e 20 ;
# per saracinesca ;
asta 1000016 100021 s 2592 / 15000 e 21 ;
dadopremistoppa 1000017 100022 s 1728 / 36000 e 22 ;
guarnizione 1000018 100023 s 2592 / 25000 e 23 ;
ghiera 1000019 100024 s 2592 / 33000 e 24 ;
volantino 1000020 100025 s 1728 / 13000 e 25 ;
dado 1000021 100026 s 1728 / 34000 e 26 ;

```

Figura 13: file recipes.xls componenti.

```

# TORNERIA ;
# per valvola a sfera grande ;
manicottopiccolo 1000022 p 1 4 100 s 3 100008 s 1728 \ 10000 e 1100 ;
manicottogrande 1000023 p 1 4 101 s 16 100008 s 1728 \ 10000 e 1101 ;
corpopiccolo 1000024 p 1 5 102 s 6 100008 s 1728 \ 10000 e 1102 ;
corpogrande 1000025 p 1 5 103 s 14 100008 s 1728 \ 10000 e 1103 ;
sferapiccola 1000026 p 1 11 104 s 5 100008 s 1728 \ 10000 e 1104 ;
sferagrande 1000027 p 1 11 105 s 18 100008 s 1728 \ 10000 e 1105 ;
cuneogrande 1000028 p 1 4 106 s 15 e 1106 ;
cuneopiccolo 1000029 p 1 5 107 s 6 e 1107 ;
vitonegrande 1000030 p 1 4 108 s 13 e 1108 ;
vitonepiccolo 1000031 p 1 5 109 s 4 e 1109 ;
corposaracinescagrande 1000032 p 1 4 110 s 17 e 1110 ;
corposaracinescapiccolo 1000033 p 1 5 111 s 5 e 1111 ;

```

Figura 14: file recipes.xls torneria.

è dunque indicativo del turno, il numero a sinistra dell'asterisco è identificativo del tipo di ricetta e quello a destra della quantità.

```
1 1000001 * 1 1000002 * 50000 1000003 * 25 1000004 * 20000
```

Figura 15: inizio della prima riga del file orderSequences.xls.

Nell'organizzare il file orderSequences.xls si è dovuto tenere conto di diversi fattori:

- volumi giornalieri di produzione suddivisi poi in due turni;
- lotti diversi per le diverse componenti;
- produzioni per il magazzino;
- tempistiche degli ordinativi.

In base alla descrizione dei quantitativi di produzione, emersa durante la visita agli stabilimenti, si è cercato di ripartire il volume reale della produzione totale tra i due prodotti scelti per la simulazione.

La produzione della VIR virtuale è costituita per il 60% da valvole a sfera (di cui 60% piccole e 40% grandi) e per il 40% da saracinesche (di cui 70% piccole e 30% grandi).

Nei primi giorni di ogni mese la produzione delle valvole a sfera è quasi doppia perché si devono alimentare i magazzini delle componenti, le saracinesche non prevedono accumuli di scorte per le componenti.

Nel corso del mese, con tempistiche regolari, devono essere lanciate le ricette per i prodotti intermedi, che saranno richiamati attraverso i procurement, e devono essere eseguiti gli acquisti delle componenti esterne.

La gestione della produzione può essere monitorata grazie al grafico ad istogrammi Procurement (int. or ext.), in esso si può osservare la produzione delle componenti o dei prodotti intermedi che verranno successivamente utilizzati per arrivare ai prodotti finiti. Questi ultimi, ossia la produzione finale vera e propria di valvole e saracinesche,

possono invece essere osservati nel file di log denominato concludedOrderLog.txt.

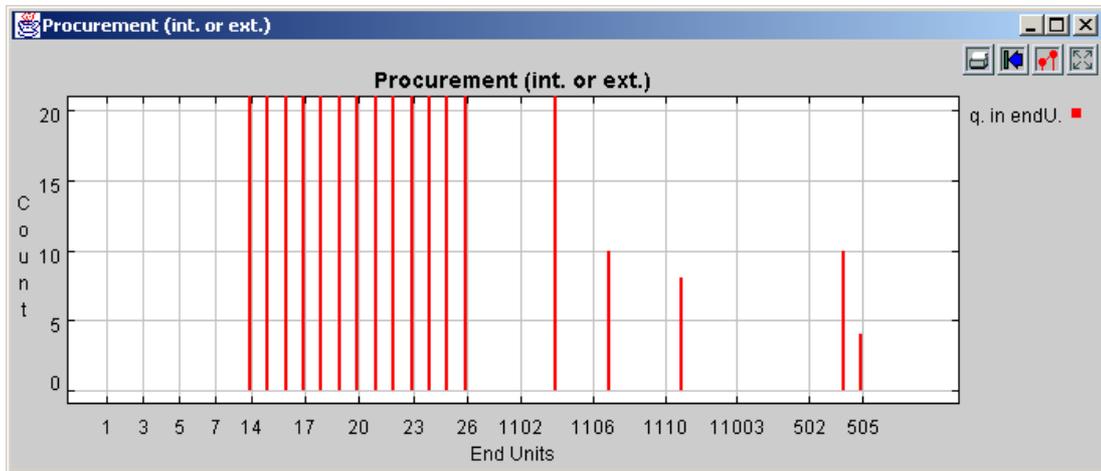


Figura 16: grafico dei Procurement (int. or ext.).

```

Each line contains: final time unit; tick in the final time unit;
                    recipe name; order layer; order number;
                    starting time unit; tick in the s. time unit
                    (number of steps); the recipe steps
                    { the units that have been doing the various steps of the
                    order (-1=step not executed, 'or' sequence) }
                    [total cost of the order]; multiplicity
1 22 imballaggiosaracinescapiccola 0 2541 1 0 (2) 10000 10000 { 80 80 } [4.0] 1
1 25 imballaggiosaracinescapiccola 0 1776 0 0 (2) 10000 10000 { 84 84 } [4.0] 1
2 25 imballaggiosaracinescapiccola 0 3316 2 0 (2) 10000 10000 { 80 80 } [4.0] 1
3 9 imballaggiosaracinescapiccola 0 3318 2 0 (2) 10000 10000 { 80 80 } [4.0] 1
3 25 imballaggiosaracinescapiccola 0 4083 3 0 (2) 10000 10000 { 80 80 } [4.0] 1
4 9 imballaggiosaracinescapiccola 0 4465 4 0 (2) 10000 10000 { 80 80 } [4.0] 1
8 10 imballaggiovalvolapiccola 0 1723 0 0 (2) 9998 9998 { 78 78 } [4.0] 1
8 19 imballaggiovalvolapiccola 0 2494 1 0 (2) 9998 9998 { 78 78 } [4.0] 1
8 21 imballaggiovalvolapiccola 0 2495 1 0 (2) 9998 9998 { 78 78 } [4.0] 1
8 25 imballaggiovalvolapiccola 0 3268 2 0 (2) 9998 9998 { 78 78 } [4.0] 1
9 1 imballaggiovalvolapiccola 0 4423 4 0 (2) 9998 9998 { 78 78 } [4.0] 1
9 5 imballaggiovalvolapiccola 0 4424 4 0 (2) 9998 9998 { 78 78 } [4.0] 1
9 7 imballaggiovalvolapiccola 0 4426 4 0 (2) 9998 9998 { 78 78 } [4.0] 1
9 10 imballaggiovalvolapiccola 0 4781 5 0 (2) 9998 9998 { 78 78 } [4.0] 1
9 12 imballaggiovalvolapiccola 0 4784 5 0 (2) 9998 9998 { 78 78 } [4.0] 1
9 14 imballaggiovalvolapiccola 0 5166 6 0 (2) 9998 9998 { 78 78 } [4.0] 1
9 16 imballaggiovalvolapiccola 0 5167 6 0 (2) 9998 9998 { 78 78 } [4.0] 1
9 18 imballaggiovalvolapiccola 0 5519 7 0 (2) 9998 9998 { 78 78 } [4.0] 1
9 23 imballaggiovalvolapiccola 0 5898 8 0 (2) 9998 9998 { 78 78 } [4.0] 1
9 25 imballaggiovalvolapiccola 0 5899 8 0 (2) 9998 9998 { 78 78 } [4.0] 1
9 25 imballaggiovalvolapiccola 0 1724 0 0 (2) 9998 9998 { 82 82 } [4.0] 1
9 27 imballaggiovalvolapiccola 0 6256 9 0 (2) 9998 9998 { 78 78 } [4.0] 1
10 3 imballaggiovalvolapiccola 0 6634 10 0 (2) 9998 9998 { 78 78 } [4.0] 1
10 5 imballaggiovalvolapiccola 0 6635 10 0 (2) 9998 9998 { 78 78 } [4.0] 1

```

Figura 17: esempio di file concludedOrderLog.txt.

Fino ad ora si sono descritte le ricette ossia le sequenze delle lavorazioni ma c'è un altro elemento molto importante nella struttura di jES: le unità (which is Doing What).

Se ripercorriamo nel dettaglio la descrizione delle due famiglie di prodotti principali, riportata sopra, è semplice notare come si ricada facilmente nella simulazione di processo assegnando ad ogni fase di lavorazione il macchinario attraverso cui il pezzo transita realmente in azienda.

Affinché la complessità, invece, emerga dall'interazione tra gli agenti, tra le ricette e le unità, è necessario descrivere queste ultime compiutamente. Nel modello generale di impresa virtuale le unità erano in grado di svolgere un'unica lavorazione, il numero distintivo della fase descritta nella ricetta corrispondeva al numero dell'unità che si occupava di quella fase.

Nel caso Vir ci sono diversi tipi di unità, alcune di esse sono semplici, possono effettuare una sola lavorazione, ed altre sono complesse in quanto sono in grado di svolgere più di una lavorazione.

L'elenco di tutte le unità è contenuto nel file unitBasicData.txt (figura 18), in esso vi è la descrizione di tutte le unità semplici che si distinguono in quanto nella terza colonna hanno il numero della fase di lavorazione che sono in grado di svolgere, seguono i costi fissi e variabili ad esse associati; quando nella terza colonna c'è il numero 0 vuol dire che quelle unità sono complesse e saranno descritte nel file units.xls. In quest'ultimo file ogni unità è associata ad una "linguetta" contrassegnata da un numero corrispondente a quello dell'unità stessa, ogni "linguetta" identifica un foglio di lavoro in cui sono riportate le specifiche organizzate secondo lo schema riportato in figura 19.

In alto a sinistra si indica il numero di lavorazioni che l'unità è in grado di eseguire. Di seguito, per ogni lavorazione, vengono riportati i costi fissi e i costi variabili.

Successivamente sono indicate due matrici contenenti rispettivamente i costi e i tempi di attrezzaggio attraverso il quale l'unità passa da uno stato ad un altro.

unit_#	useWarehouse	prod.phase_#	fixed_costs	variable_costs
1	1	0	0	0
2	1	0	0	0
3	1	0	0	0
4	1	0	0	0
5	1	0	0	0
6	1	0	0	0
7	1	0	0	0
8	1	0	0	0
9	1	0	0	0
10	1	0	0	0
11	1	101	1	1
12	1	103	1	1
13	1	0	0	0
14	1	0	0	0
15	1	0	0	0
16	1	0	0	0
17	1	0	0	0
18	1	0	0	0
19	1	0	0	0
20	1	0	0	0
21	1	0	0	0
22	1	0	0	0
23	1	0	0	0
24	1	0	0	0
25	1	0	0	0
26	1	0	0	0
27	1	0	0	0
28	1	0	0	0
29	1	0	0	0
30	1	0	0	0
31	1	0	0	0
32	1	0	0	0
33	1	0	0	0
34	1	0	0	0
35	1	0	0	0
36	1	0	0	0
37	1	0	0	0
38	1	0	0	0
39	1	0	0	0
40	1	0	0	0
41	1	0	0	0
42	1	0	0	0
43	1	0	0	0
44	1	0	0	0
45	1	0	0	0
46	1	0	0	0
47	1	0	0	0
48	1	0	0	0
49	1	0	0	0
50	1	0	0	0
51	1	0	0	0
52	1	0	0	0

53	1	0	0	0
54	1	0	0	0
55	1	100001	1	1
56	1	100002	1	1
57	1	100003	1	1
58	1	100004	1	1
59	1	100005	1	1
60	1	100006	1	1
61	1	100007	1	1
62	1	100011	1	1
63	1	100014	1	1
64	1	100015	1	1
65	1	100016	1	1
66	1	100017	1	1
67	1	100018	1	1
68	1	100019	1	1
69	1	100020	1	1
70	1	100021	1	1
71	1	100022	1	1
72	1	100023	1	1
73	1	100024	1	1
74	1	100025	1	1
75	1	100026	1	1
76	1	100008	1	1
77	1	9997	1	1
78	1	9998	1	1
79	1	9999	1	1
80	1	10000	1	1
81	1	9997	1	1
82	1	9998	1	1
83	1	9999	1	1
84	1	10000	1	1
85	1	27001	1	1
86	1	40001	1	1
87	1	29001	1	1
88	1	30001	1	1
89	1	31001	1	1
90	1	32001	1	1
91	1	33001	1	1
92	1	34001	1	1
93	1	35001	1	1
94	1	36001	1	1
95	1	37001	1	1
96	1	38001	1	1
97	1	100008	1	1

Figura 18: file unitBasicData.txt.

nOfPhasesToDealWith								
phase 1	fixed costs 1	variable costs 1	inventories in production 1					
phase 2	fixed costs 2	variable costs 2	inventories in production 2					
...								
phase n	fixed costs n	variable costs n	inventories in production n					
sc 1 1	sc 1 2	...	sc 1 n					
sc 2 1	sc 2 2	...	sc 2 n					
...					
sc n 1	sc n 2	...	sc n n					
st 1 1	st 1 2	...	st 1 n					
st 2 1	st 2 2	...	st 2 n					
...					
st n 1	st n 2	...	st n n					
Remarks								
hopefully, fixed costs are the same for all phases,								
anyway, when the unit state is undefined (no production made) we use the fixed costs of the first row								
inventory production can be 0 (no) or 1 (yes); normally, only one row is set to 1								
if we find more than one row set to 1, the first one is chosen								
sc i j = setup costs from state i to state j					st i j = setup time from state i to state j			

Figura 19: schema generale del file units.xls.

Fra le unità semplici ci sono anche delle particolari unità denominate shadow units, esse sono state introdotte per far sì che gli elementi richiamati dal procurement possano essere recepiti da una lavorazione contenente un batch.

Oltre a questi due tipi di unità c'è n'è un terzo tipo denominato end units: esse sono unità speciali, presso le quali sono reperibili le parti intermedie di produzione asincrona.

Tutte le lavorazioni intermedie e le componenti andranno dunque in una end unit, da questa verranno poi richiamate grazie al meccanismo del procurement già spiegato sopra. Tali unità sono identificate con un codice corrispondente a quello identificativo dei prodotti intermedi le cui ricette terminano con una "e".

Le end units sono elencate nel file di testo endUnitList.txt riportato nella figura 20.

```
end_unit_#;_use_positive_code_for_layer_sensitive_end_unit;_negative
_for_insensitive_(codes_cannot_be_duplicated,_not_even_with_a_differ
ent_sign)
1
2
3
4
5
6
7
11
14
15
16
17
18
19
20
21
22
23
24
25
26
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
11001
11002
11003
11004
500
501
502
503
504
505
506
507
```

Figura 20: file endUnitList.txt.

5.3 Sviluppo del codice

Per applicare il caso VIR al modello di impresa virtuale è stato necessario apportare diverse modifiche al codice, proprio per questo

ci sono numeri progressivi di versioni. All'inizio della preparazione di questa dissertazione di laurea e di quelle collegate, la versione di jES era la 0.4.5, ora siamo arrivati alla versione 0.9.7.50.

Il primo passaggio è stato dalla versione 0.4.5 alla 0.5.0, quest'ultima ha introdotto importanti novità di struttura ed in particolare un archivio *jesframe.jar* delle classi già compilate. In questo modo si è potuto sperimentare l'introduzione di nuove classi, per l'adattamento del modello al caso VIR, senza alterare il modello di base.

L'utilizzo del *Jar* non implica modifiche del codice, quanto piuttosto modifiche inerenti il package nella sua interezza.

Ecco come si articola la struttura delle cartelle all'interno di jES:

- *src*: contiene i file della versione corrente;
- *classes*: contiene i file *.class* relativi alle classi in *src* e i file usati per generare l'archivio jar:
 - ⇒ *makefile*: per richiamare l'archivio jar;
 - ⇒ *jarJES*;
 - ⇒ *MANIFEST.MF*: necessario per dare la possibilità al compilatore di trovare il file *StartVEFrame.class* di avvio della simulazione;
- *lib*: contenente l'archivio jar generato, gli archivi *gui.jar*, *plot.jar* e le sottodirectory di PT utilizzate per creare gli istogrammi.

Grazie a questa nuova impostazione è possibile compilare ed eseguire il programma con diverse opzioni:

- a. *make run* per mandare in esecuzione jES come è;
- b. *make compileBasic* per compilare e *make runFromClasses* per eseguire il programma con gli stessi effetti del punto precedente;
- c. *make jar* per rigenerare l'archivio jar;
- d. *make* e *make compile* per aggiungere o modificare classi;
- e. *make runJar* per usare la versione base o in alternativa al punto a.

La versione 0.5.1 segna una nuova tappa, in questo essa sono introdotti i primi cambiamenti significativi al codice, la modifica più importante riguarda l'Order Generator.

Si introduce la possibilità di generare ricette con passi di produzione di durate diverse. Le informazioni relative ai tempi di lavorazione sono contenute nelle ricette e non nelle unità produttive per due principali ragioni:

- esaminando la ricetta è possibile determinare il tempo standard di produzione, vale a dire il tempo che occorre per ottenere il prodotto finito nel caso in cui siano disponibili tutte le componenti;
- inserire le informazioni presso le unità significherebbe confondere due elementi di conoscenza distinti WD e DW di cui si è parlato precedentemente.

La modifica è individuabile, a livello di listato, nelle seguenti righe:

```
* orderRecipe1      contains  1 12  7  3
* stepLengthInOrder contains  1  3  2  2
*
* orderRecipe2      contains  1 12 12 12  7  7  3  3
*/
// generating the length of each step

// putting 0 in the vector is not strictly necessary, but may be
// useful for future software modifications
for (i=0;i<maxStepNumber;i++)stepLengthInOrder[i]=0;

if (maxStepLength>1)
{
    for (i=0;i<randomStepNumber;i++)
        stepLengthInOrder[i]=
            Globals.env.uniformIntRand.getIntegerWithMin$withMax
            (1, maxStepLength);
}
else
{
    for (i=0;i<randomStepNumber;i++)stepLengthInOrder[i]=1;
}
// putting (i) sequence and (ii) length, together
count=0;
for (i=0;i<randomStepNumber;i++)
    for (ii=0;ii<stepLengthInOrder[i];ii++)
    {
        orderRecipe2[count]=orderRecipe1[i];
        count++;
    }
```

Le fasi di produzione sono ripetute tante volte quanto è indicato nel vettore dei tempi corrispondente: `stepLengthInOrder`.

Questa ripetizione determina la necessità di gestire fasi di lavorazione consecutive tra loro uguali in modo LIFO e non FIFO.

In questa versione del modello si introduce anche la scansione del tempo in giorni introducendo nello schedule il valore di `ticksInADay`.

```
// 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
modelSchedule = new ScheduleImpl (getZone (), ticksInADay);

modelSchedule.at$createAction (0, modelActions1);

for (i=0;i<ticksInADay;i++)
modelSchedule.at$createAction (i, modelActions2);

modelSchedule.at$createAction (ticksInADay-1, modelActions3);

return this;
}
```

Per quanto riguarda le unità, le prime modifiche significative si hanno con la versione 0.5.2. In essa, infatti, sono introdotte unità in grado di svolgere più di una lavorazione.

Le descrizioni delle unità semplici sono contenute nel file `unitBasicData.txt` (figura 18, paragrafo 5.2), quelle delle unità complesse sono raccolte nel file `recipes.xls` (figura 19, paragrafo 5.2) le cui strutture sono state analizzate nel paragrafo stesso.

Per leggere le informazioni contenute in questi file è stata introdotta la classe `ExcelReader`, che semplifica l'uso della libreria `excelread` (<http://www.andykhan.com/excelread/>).

A questo punto, con l'introduzione di unità complesse, viene meno la corrispondenza tra i numeri delle fasi di lavorazione contenuti nelle ricette e quelli indicativi delle unità in grado di eseguire la lavorazione.

Le modifiche sono state apportate nel file `UnitParameters` di cui segue la parte di listato:

```

/** read the unit number, production phase, fixed cost and variable
cost
* parameters from the unitBasicData.txt file*/
public void readUnitData(){
try{
    unitNumber = new int[totalUnitNumber];
    simpleUnitProductionPhase = new int[totalUnitNumber];
    fixedCosts = new float[totalUnitNumber];
    variableCosts = new float[totalUnitNumber];
    useWarehouseLocalChoice = new int[totalUnitNumber];

    // read the file
    BufferedReader fileUnitBasicData = null;
    StringTokenizer t;
    fileUnitBasicData = new
        BufferedReader(new FileReader("unitData/unitBasicData.txt"));

    // reading a control line (titles)
    String line = " ", lineTest =

"unit_#__useWarehouse____prod.phase_#____fixed_costs____variable_co
sts";

    line=fileUnitBasicData.readLine();
    if (! line.equals(lineTest))
    {
        System.out.println(
            "First line in unitData/unitBasicData.txt must contain:");
        System.out.println(lineTest);
        System.exit(1);
    }
    line=" ";

    // starting with a void line; the actual
    // reading is done in MyReader, dealing also
    // with the missing data case
    t = new StringTokenizer(line, " ");

    for(i = 0; i < totalUnitNumber; i++)
    {
        t=MyReader.check("unitData/unitBasicData.txt",
            fileUnitBasicData, line, t);
        unitNumber[i] = Integer.parseInt(t.nextToken());
        useWarehouseLocalChoice[i] = Integer.
            parseInt(t.nextToken());
        simpleUnitProductionPhase[i]
            = Integer.parseInt(t.nextToken());
        fixedCosts[i] = Float.parseFloat(t.nextToken());
        variableCosts[i] = Float.parseFloat(t.nextToken());
    }

    fileUnitBasicData.close();
}
catch (FileNotFoundException fnfe)
{
    System.out.println(fnfe);
    System.exit(1);
}
catch (IOException e)
{

```

```

        System.out.println("IOException:" + e.toString());
    }
}

/** setting the parameters of units */
public void setParametersTo(int i, Unit aUnit){
String fileName;
float fixedCosts, variableCosts;
int unitProductionPhase, nOfPhasesToDealWith, phaseInventories;

aUnit.setUnitNumber(getUnitNumber(i));
aUnit.setPhaseInventories(getUseWarehouseLocalChoice(i));
aUnit.setProductionPhase(getSimpleUnitProductionPhase(i));
aUnit.setFixedAndVariableCosts(getFixedCosts(i),
getVariableCosts(i));

// complex unit case, with a sheet in a workshhet file related to
// the each unit
if(getSimpleUnitProductionPhase(i) == 0) // i.e. no unique phase
{
// open (once) the Excel file containing special unit data */
if(! worksheetUnitFileOpen)
specialUnitWorksheet =
new ExcelReader("unitData/units.xls");
worksheetUnitFileOpen = true;

specialUnitWorksheet.selectSheet(String.valueOf(i));

if(StartVEFrame.verbose)
System.out.println("Unit " + getUnitNumber(i)
+ " is a special one.");

// the unit is able to deal with nOfPhasesToDealWith
// different production phases
nOfPhasesToDealWith=specialUnitWorksheet.getIntValue();
if(StartVEFrame.verbose)
System.out.println("special unit " +
getUnitNumber(i) + " nOfPhasesToDealWith " +
nOfPhasesToDealWith);

aUnit.setNOfPhasesToDealWith(nOfPhasesToDealWith);

for (ii=1; ii<=nOfPhasesToDealWith; ii++)
{
// production phases (one or more)
unitProductionPhase=specialUnitWorksheet.getIntValue();
aUnit.setProductionPhase(unitProductionPhase);

// costs for each phase

fixedCosts = (float)
specialUnitWorksheet.getDblValue();
variableCosts = (float)
specialUnitWorksheet.getDblValue();
if(StartVEFrame.verbose)
System.out.println("special unit " +
getUnitNumber(i) + " fixed costs " +
fixedCosts + " variable costs " +
variableCosts);
}
}
}

```

```

        aUnit.setFixedAndVariableCosts (fixedCosts,
                                         variableCosts);
        phaseInventories =
            specialUnitWorksheet.getIntValue();
        aUnit.setPhaseInventories (phaseInventories);
    }
}

```

Una volta definite le unità secondo le caratteristiche descritte è stato necessario modificare l'Order Generator affinché interrogasse le units e creasse un dizionario delle fasi di lavorazione da inserire, per il momento ancora in modo casuale, nelle ricette.

```

/** building the dictionary containing the steps to be included in
the recipes */
public void setDictionary()
{
    int i, ii, iii;
    // how many words in the dictionary?

    // starting point: the no. of endUnits
    dictionaryLength=endUnitList.getCount();

    for (i=0;i<unitList.getCount();i++)
    {
        aUnit=(Unit) unitList.atOffset(i);
        dictionaryLength += aUnit.getNOOfPhasesToDealWith();
    }
    if(StartVEFrame.verbose)
        System.out.println(
            "Length of the dictionary (# of codes identifying steps in
            recipes): "
            + dictionaryLength);

    //creating the dictionary
    dictionary = new int[dictionaryLength];

    // loading terms into the dictionary (from endUnits and then units)
    i=0;
    for (ii=0;ii<endUnitList.getCount();ii++)
    {
        anEndUnit=(EndUnit) endUnitList.atOffset(ii);
        dictionary[i++]=anEndUnit.getUnitNumber();
    }
    for (ii=0;ii<unitList.getCount();ii++)
    {
        aUnit=(Unit) unitList.atOffset(ii);
        for (iii=0;iii<aUnit.getNOOfPhasesToDealWith();iii++)
            dictionary[i++]=aUnit.getProductionPhase(iii);
    }

    if(StartVEFrame.verbose)
        for (i=0;i<dictionaryLength;i++)
            System.out.println(i + " " + dictionary[i]);
}

```

Nel listato si può notare che l'Order Generator interroga, tra le unità, anche le end units introdotte con la versione 0.6.2 per simulare il caso di produzioni asincrone. Le end units sono unità speciali che contengono i prodotti intermedi necessari per le lavorazioni di altre units "tradizionali".

Il numero che le identifica corrisponde al codice con cui si concludono le ricette dei prodotti intermedi corrispondenti. L'elenco delle end units è contenuto nel file: endUnitList.txt (figura 20, paragrafo 5.2).

La classe delle end units è costruita sfruttando la proprietà dell'ereditarietà delle classi (Unit è la *parent class*), propria della programmazione ad oggetti.

```
* the constructor for EndUnit
*/
public EndUnit (Zone aZone, VEFrameModelSwarm mo)
{
// Call the constructor for the Unit parent class.
super(aZone);

// the model (for future use)
VEFrameModelSwarm = mo;

// creating the internal list temporary the items temporary in hold
// hold
temporaryList = new ListImpl (getZone());
}
```

Per poter attivare queste unità speciali è stato necessario introdurre nelle ricette un nuovo formalismo. La versione 0.6.3 introduce nelle ricette i passi di procurement (p).

Ecco il listato dell'Order Generator:

```
// introducing "p" or "c" sequences

// the "p" structure is
// p k a1 a2 .... ak
// where k is the # of item to be procured and
// a1, a2, ..., ak are the code of those items
// (corresponding to "e" or endUnit code in sub-recipe
// related to internal or external sub-processes

// the recipe scheme is
// external recipe (with the time specifications and
```

```

// a tick corresponding to 1 sec.)

// 11 s 3 p 3 101 102 103 22 s 2

// internally, in orderGenerator, with p reported as -1

// 11 11 11 -1 3 101 102 103 22 22

// into anOrder

// 11 11 11 -22 22
// with a procurementSpecifications object having the same
// # of the order and a position parameter relative to the
// recipe sequence (here position = 4)

// and, when anOrder comes back from procurementAssembler,
// 11 11 11 22 22
// always with the procurementSpecifications object

if(endUnitList.getCount() > 0)
{
    if (ii==0 && 0.05 >= Globals.env.uniformDblRand.
        getDoubleWithMin$withMax(0,1)){

        // length of the "p" or "c" step
        k=Globals.env.uniformIntRand.
            getIntegerWithMin$withMax
            (1, endUnitList.getCount());

        // increase the length of orderRecipe2 vector
        orderRecipeTmp = new int[orderRecipe2.
            length+2+k];
        System.arraycopy(orderRecipe2,0,
            orderRecipeTmp,0,
            orderRecipe2.length);
        orderRecipe2=orderRecipeTmp;

        //the code of process p
        orderRecipe2[count++]= -1;

        // the length
        orderRecipe2[count++]=k;

        // k codes, chosen randomly
        for (iii=0;iii<k;iii++)
            orderRecipe2[count++]=
                ((EndUnit) endUnitList.
                    atOffset(
                        Globals.env.uniformIntRand.
                            getIntegerWithMin$withMax
                            (0, endUnitList.getCount()-1)
                            )).getUnitNumber();
    }
}

// copying normal steps
orderRecipe2[count++]=orderRecipe1[i];
}

```

```

if(StartVEFrame.verbose)
{
    for (iii=0; iii<count;iii++)
        System.out.print(orderRecipe2[iii]+ " ");
    System.out.println(" internal recipe for order " + orderCount);
}

```

Contemporaneamente sono state create due nuove classi:

- ProcurementSpecificationSet: contiene le sottoricette;
- ProcurementAssembler: cerca le componenti da procurare.

La struttura delle ricette necessita ancora di una modifica per gestire le produzioni batch, il cui formalismo è stato già descritto nel precedente paragrafo.

Sono state create per questo scopo due classi per ogni tipologia di batch:

- StandAloneBatchSpecificationSet
- StandAloneBatchAssembler
- SequentialBatchSpecificationSet
- SequentialBatchAssembler

in analogia con le classi generate per gestire i procurements.

Un ulteriore aspetto da considerare in materia di formalismo delle ricette riguarda la possibilità di trattare produzioni alternative.

Infatti nella versione 0.9.0 sono stati inseriti i criteri di “OR”, già esaminati nel paragrafo precedente, con la creazione della classe Or.java.

```

// applying the 'or' criterion
if(orCriterion == 0) return;

chosenNode=0;
recipe = (int []) o.getRecipeVector();
state = (int []) o.getStateVector();

if(orCriterion == 1) chosenNode=1;
if(orCriterion == 2) chosenNode=2;
if(orCriterion == 3) chosenNode=Globals.env.uniformIntRand.
    getIntegerWithMin$withMax(1, spec.getNodeNumber());

if(orCriterion == 4)
{
    length=1000000000;
    // looking for the first unit in each branch
    for (node=1; node<=spec.getNodeNumber();node++)

```

```

    {
        length0=length;
        unitNotFound=true;
        for (i=0;i<unitList.getCount() && unitNotFound;i++)
        {
            aUnit=(Unit) unitList.atOffset(i);
            for (iii=0;iii<aUnit.getNOOfPhasesToDealWith()
                && unitNotFound;iii++)
            {
                if(recipe[spec.getNodePosition(node)]==
                    aUnit.getProductionPhase(iii))
                {
                    unitNotFound=false;
                    length=aUnit.
                        getWaitingListLength();
                    if(length<length0)chosenNode=
                        node;
                }
            }
        }
    }
}

```

La successiva versione 0.9.1 ha permesso di avere dei processi di procurement all'interno dei rami "or".

Nella versione 0.9.7.01 sono stati introdotti i "layer", nome utilizzato per indicare i vari livelli cui la stessa ricetta può appartenere.

Gli ordini possono essere ora distinti secondo il layer a cui appartengono e quindi, nel procurement si può attingere dalle end unit solo prodotti intermedi appartenenti a quel layer; è però possibile indicare alcune end unit (o tutte) come indifferenti ai layer (se ad esempio si tratta di approvvigionamenti che sono indipendenti dai layer).

Anche nella gestione dei sequential batch si tiene conto - per formare i batch - del fatto che gli order appartengano allo stesso layer; però è possibile fare in modo che alcune unit (o tutte) non tengano conto dei layer nel formare i sequential batch.

Per introdurre i layer è stato necessario modificare il file Order.java ed il file OrderGenerator.java.

```

public void setOrderLayer(int la){
orderLayer=la;
}

```

```

/**
 *
 */
public int getOrderLayer(){
return orderLayer;
}

// choosing the layer of the order(s) generated
// (from 0 to totalLayerNumber-1)
if(totalLayerNumber>1)
    if(0.03>Globals.env.uniformDblRand.
        getDoubleWithMin$withMax(0,1))
        layerNumber=Globals.env.uniformIntRand.
            getIntegerWithMin$withMax(0, totalLayerNumber-1);

/*      // to be activate to generate comparative test with one or
more
        // layers and unsensitive units
        // this second layer generator has no effect, but in case of
only one
        // layer it reproduces the same sequence of random numbers
if(totalLayerNumber==1)
    if(0.03>Globals.env.uniformDblRand.
        getDoubleWithMin$withMax(0,1))
        Globals.env.uniformIntRand.
            getIntegerWithMin$withMax(0, 1); // don't change
this
                                                    // 0-1 interval
*/

```

Il modello di simulazione di impresa virtuale, per essere adattato al caso VIR, richiede una evoluzione non soltanto dal lato del formalismo delle ricette e della complessità delle unità, ma anche nell'aspetto della generazione degli ordini.

Per introdurre la possibilità di gestire un archivio storico (figura 15, paragrafo 5.2) da cui estrarre le ricette da mandare in produzione è stato necessario generare due nuove classi:

- OrderDistiller;
- Recipe;

che sostituissero la generazione casuale degli ordini operata dall'Order Generator con una produzione aderente a quella della realtà aziendale simulata.

I file javadoc relativi alle due classi in questione sono riportati in appendice.

I metodi definiti nelle due classi consentono di effettuare una lettura controllata dei dati da due file:

- OrderSequences.xls per la classe Order Distiller;
- Recipe.xls per la classe Recipe.

Nella fase di raccolta dei dati e di inizializzazione dei vettori si verifica, infatti, che la struttura definita per i due file sia rispettata e non si siano verificati errori nel corso dell'inserimento delle informazioni.

Seguiamo ora idealmente il percorso di formazione di un ordine di produzione:

- all'inizio di ogni ciclo l'Order Distiller legge dal foglio di lavoro orderSequences.xls e registra i numeri identificativi delle ricette nel vettore orderSequences1 e delle quantità nel vettore orderSequences2;
- la classe Recipe legge dal file Recipe.xls e registra i passi che compongono ciascuna delle ricette e il loro codice identificativo nel vettore orderRecipe;
- l'Order Distiller è quindi in grado di generare l'ordine e di inviarlo alla prima unità produttiva, simulando il ruolo del Front End.

In questo momento si avvia la produzione di jES-VIR, che, in termini di volumi, eguaglia quella dell'impresa reale.

La formalizzazione presentata sopra è molto articolata, ma presenta ancora alcune imperfezioni che saranno trattate e sviluppate nel capitolo successivo.

5.4 Appendice

Riportiamo in calce alcune descrizioni permettendo così una più fluida lettura dei concetti esposti.

5.4.1 Dettaglio articoli

Ecco il dettaglio di alcune valvole a sfera:

famiglia 340

1/2 pollice

corpo

1. acquisto dei pani di ottone;
2. stampaggio all'esterno;
3. sabbiatura all'esterno;
4. lavorazione meccanica per asportazione del truciolo:
 1. hP A
 1. C5
 2. hP B
 1. B1
 3. hP C
 1. C1
 4. hP D
 1. C2
 5. hP E
 1. all'esterno
5. cromatura all'esterno;
6. deposito nel magazzino componenti;

manicotto

1. acquisto dei pani di ottone;
2. stampaggio all'esterno;
3. sabbiatura all'esterno;
4. lavorazione meccanica per asportazione del truciolo:
 1. hP A
 1. A4

2. hP B
 1. B5
3. hP C
 1. all'esterno
5. cromatura all'esterno;
6. deposito nel magazzino componenti;

sfera

1. acquisto delle barre di ottone;
2. lavorazione meccanica per asportazione del truciolo:
 1. hP A
 1. D1
3. cromatura all'esterno;
4. deposito nel magazzino componenti;

bullone, maniglia, premistoppa, guarnizione premistoppa, perno, 2
seggi

1. Acquisto all'esterno;
2. Verniciatura all'esterno;
3. deposito nel magazzino componenti;

Assemblaggio:

1. hP A
 1. S1 unico percorso di montaggio (dalle componenti alla scatola confezionata);
 2. deposito nel magazzino prodotti finiti.
2. hP B
 1. P6 dalle componenti al prodotto finito;
 2. "STAZIONE DI LAVORO" insachettamento ed inscatolamento manuale;
 3. deposito nel magazzino prodotti finiti.

3. hP C

1. P1 seggio nel corpo;
2. L1 seggio nel manicotto;
3. L5 corpo + manicotto + sfera ;
4. L6 assemblaggio finale;
5. "STAZIONE DI LAVORO" insachettamento ed
in scatolamento manuale;
6. deposito nel magazzino prodotti finiti.

3/4 pollice

corpo

1. acquisto dei pani di ottone;
2. stampaggio all'esterno;
3. sabbiatura all'esterno;
4. lavorazione meccanica per asportazione del truciolo:
 1. hP A
 1. C5
 2. hP B
 1. B1
 3. hP C
 1. C1
 4. hP D
 1. C2
 5. hP E
 1. all'esterno
5. cromatura all'esterno;
6. deposito nel magazzino componenti;

manicotto

1. acquisto dei pani di ottone;
2. stampaggio all'esterno;

3. sabbiatura all'esterno;
4. lavorazione meccanica per asportazione del truciolo:
 1. hP A
 1. A4
 2. hP B
 1. B5
 3. hP C
 1. all'esterno
5. cromatura all'esterno;
6. deposito nel magazzino componenti;

sfera

1. acquisto delle barre di ottone;
2. lavorazione meccanica per asportazione del truciolo:
 1. hP A
 1. D1
 2. hP B
 1. C4
3. cromatura all'esterno;
4. deposito nel magazzino componenti;

bullone, maniglia, premistoppa, guarnizione premistoppa, perno, 2
seggi

1. Acquisto all'esterno;
2. Verniciatura all'esterno;
3. deposito nel magazzino componenti;

Assemblaggio:

1. hP A
 1. S1 unico percorso di montaggio (dalle componenti alla scatola confezionata);

2. deposito nel magazzino prodotti finiti.
2. hP B
 1. P6 dalle componenti al prodotto finito;
 2. "STAZIONE DI LAVORO" insachettamento ed inscatolamento manuale;
 3. deposito nel magazzino prodotti finiti.
3. hP C
 1. P1 seggio nel corpo;
 2. L1 seggio nel manicotto;
 3. L5 corpo + manicotto + sfera ;
 4. L6 assemblaggio finale;
 5. "STAZIONE DI LAVORO" insachettamento ed inscatolamento manuale;
 6. deposito nel magazzino prodotti finiti.

1 pollice

corpo

1. acquisto dei pani di ottone;
2. stampaggio all'esterno;
3. sabbiatura all'esterno;
4. lavorazione meccanica per asportazione del truciolo:
 1. hP A
 1. A2
 2. hP B
 1. C1
5. cromatura all'esterno;
6. deposito nel magazzino componenti;

manicotto

1. acquisto dei pani di ottone;
2. stampaggio all'esterno;

3. sabbiatura all'esterno;
4. lavorazione meccanica per asportazione del truciolo:
 1. hP A
 1. A4
 2. hP B
 1. B3
5. cromatura all'esterno;
6. deposito nel magazzino componenti;

sfera

1. acquisto delle barre di ottone;
2. lavorazione meccanica per asportazione del truciolo:
 1. hP B
 1. C4
3. cromatura all'esterno;
4. deposito nel magazzino componenti;

bullone, maniglia, premistoppa, guarnizione premistoppa, perno, 2
seggi

1. Acquisto all'esterno;
2. Verniciatura all'esterno;
3. deposito nel magazzino componenti;

Assemblaggio:

1. hP A
 1. L4b seggio nel corpo, seggio nel manicotto, corpo + manicotto + sfera ;
 2. L4 assemblaggio finale;
 3. "STAZIONE DI LAVORO" insachettamento ed inscatolamento manuale;
 4. deposito nel magazzino prodotti finiti.

famiglia 342

3/4 pollice

corpo

1. acquisto dei pani di ottone;
2. stampaggio all'esterno;
3. sabbiatura all'esterno;
4. lavorazione meccanica per asportazione del truciolo:
 1. hP A
 1. C1 prima fase;
 2. C2 seconda fase;
 2. hP B
 1. C2 att2 prima fase;
 2. C2 seconda fase;
5. cromatura all'esterno;
6. deposito nel magazzino componenti;

manicotto

1. acquisto dei pani di ottone;
2. stampaggio all'esterno;
3. sabbiatura all'esterno;
4. lavorazione meccanica per asportazione del truciolo:
 1. hP A
 1. A4
 2. hP B
 1. B5
 3. hP C
 1. all'esterno
5. cromatura all'esterno;

6. deposito nel magazzino componenti;

sfera

1. acquisto delle barre di ottone;
2. lavorazione meccanica per asportazione del truciolo:
 1. hP A
 1. D1
 2. hP B
 1. C4
3. cromatura all'esterno;
4. deposito nel magazzino componenti;

bullone, maniglia, premistoppa, guarnizione premistoppa, perno, 2
seggi

1. Acquisto all'esterno;
2. Verniciatura all'esterno;
3. deposito nel magazzino componenti;

Assemblaggio:

1. hP A
 1. P1 seggio nel corpo;
 2. L1 seggio nel manicotto;
 3. L5 corpo + manicotto + sfera ;
 4. L6 assemblaggio finale;
 5. "STAZIONE DI LAVORO" insachettamento ed
inscatolamento manuale;
 6. deposito nel magazzino prodotti finiti.

Ecco il dettaglio di alcune saracinesche:

famiglia 42

3/4 pollice

corpo

1. acquisto dei pani di ottone;
2. stampaggio all'esterno;
3. sabbiatura all'esterno;
4. lavorazione meccanica per asportazione del truciolo:
 1. hP A
 1. B1
5. deposito nel magazzino componenti;

vitone

1. acquisto dei pani di ottone;
2. stampaggio all'esterno;
3. sabbiatura all'esterno;
4. lavorazione meccanica per asportazione del truciolo:
 1. hP A
 1. A4
 2. hP A
 1. B3
5. deposito nel magazzino componenti;

cuneo

1. acquisto dei pani di ottone;
2. stampaggio all'esterno;
3. sabbiatura all'esterno;
4. lavorazione meccanica per asportazione del truciolo:
 1. hP A
 1. B4
5. deposito nel magazzino componenti;

dado, volantino, dado premistoppa, 2 ghiera, guarnizione premistoppa, asta

1. Acquisto all'esterno;
2. deposito nel magazzino componenti;

Assemblaggio:

1. hP A
 1. O4
 2. O5
 3. "STAZIONE DI LAVORO" insachettamento ed inscatolamento manuale;
2. spedizione

1 pollice

corpo

1. acquisto dei pani di ottone;
2. stampaggio all'esterno;
3. sabbiatura all'esterno;
4. lavorazione meccanica per asportazione del truciolo:
 1. hP A
 1. B1
5. deposito nel magazzino componenti;

vitone

1. acquisto dei pani di ottone;
2. stampaggio all'esterno;
3. sabbiatura all'esterno;
4. lavorazione meccanica per asportazione del truciolo:
 1. hP A
 1. A4

2. hP A

1. B3

5. deposito nel magazzino componenti;

cuneo

1. acquisto dei pani di ottone;

2. stampaggio all'esterno;

3. sabbiatura all'esterno;

4. lavorazione meccanica per asportazione del truciolo:

1. hP A

1. B4

5. deposito nel magazzino componenti;

dado, volantino, dado premistoppa, 2 ghiera, guarnizione premistoppa,

asta

1. Acquisto all'esterno;

2. deposito nel magazzino componenti;

Assemblaggio:

1. hP A

1. O4 att2

2. O5 att2

3. "STAZIONE DI LAVORO" insachettamento ed
in scatolamento manuale;

2. spedizione

5.4.2 Analisi delle fasi di produzione

Dizionario delle fasi di lavorazione:

Valvole a sfera

Corpo

1. stampaggio all'esterno;
2. sabbiatura all'esterno;
3. ricottura esterna;
100. sabbiatura interna;
4. cromatura all'esterno;
5. teflonatura all'esterno;
6. nichelatura all'esterno;

Manicotto

1. stampaggio all'esterno;
7. sabbiatura all'esterno;
8. ricottura esterna;
101. sabbiatura interna;
9. cromatura all'esterno
10. teflonatura all'esterno;
11. nichelatura all'esterno;

Sfera

12. sabbiatura all'esterno;
102. sabbiatura interna;
13. cromatura all'esterno

Parti

14. bullone;
15. maniglia;
16. premistoppa;
17. guarnizione premistoppa;
18. perno;
19. seggio (ne servono sempre 2);

20. verniciatura all'esterno;

21. sfera;

Saracinesche

Corpo

1. stampaggio all'esterno;
22. ricottura all'esterno;
23. sabbiatura all'esterno;

Cuneo

1. stampaggio all'esterno;
24. sabbiatura all'esterno;
25. ricottura esterna;
103. sabbiatura interna;

Vitone

1. stampaggio all'esterno;
26. ricottura;
27. sabbiatura all'esterno;
104. sabbiatura interna;

Parti

28. Dado
29. Asta
30. Volantino
31. Guarnizione

Torneria

Lavorazione meccanica per asportazione del truciolo

Componente	Grande	Piccolo
Manicotto	101	100
Sfera	105	104
Corpo (valvole a sfera)	103	102
Corpo (saracinesche)	110	111
Cuneo	106	107
Vitone	108	109

Montaggio

Prodotto finito	Grande	Piccolo
Imballaggio valvole a sfera	9997	9998
Imballaggio saracinesche	9999	10000

Preassemblato	Grande	Piccolo
<i>Valvole a sfera:</i>		
Seggio + Manicotto (A)	500	503
Seggio + Corpo (B)	501	504
A + B + Sfera + Perno (C)	502	505
C + Maniglia + Bullone + Dado + Premistoppa + Prova	10001	10002
<i>Saracinesche:</i>		
Corpo + Cuneo + Vitone + Asta + Guarnizione	506	507
Volantino + Dado + Prova	10003	10004

5.4.3 File javadoc delle classi Order Distiller e Recipe

Si riportano qui di seguito i file javadoc relativi alle classi Order Distiller e Recipe:

Class OrderDistiller

OrderGenerator

|
+-**OrderDistiller**

public class **OrderDistiller**

extends OrderGenerator

This class is used to read data from two worksheets. The first one contains the list of recipes of our virtual enterprise. The second one contains a sequence of orders to be launched, shift by shift, in order to make the daily production activities.

Field Summary	
Order	<u>anOrder</u> a specific order
java.lang.String	<u>backslash</u> Flags to operate checks while reading
java.lang.String	<u>checkTheCell</u> Flags to operate checks while reading
java.lang.String	<u>computation</u> Flags to operate checks while reading
java.lang.String	<u>end</u> Flags to operate checks while reading
swarm.collections.ListImpl	<u>endUnitList</u> the list containig the end units
static boolean	<u>firstTime</u>
java.lang.String	<u>gate</u> Flags to operate checks while reading
java.lang.String	<u>layer</u> Flags to operate checks while reading
java.lang.String	<u>min</u> Flags to operate checks while reading
java.lang.String	<u>or</u> Flags to operate checks while reading
Int	<u>orderCount</u> used to record the number of generated orders

swarm.collections.ListImpl	<u>orderList</u> the list containig all the orders
java.lang.String	<u>p</u> Flags to operate checks while reading
swarm.collections.ListImpl	<u>recipeList</u> the list containig all the orders
java.lang.String	<u>sec</u> Flags to operate checks while reading
java.lang.String	<u>semicolon</u> Flags to operate checks while reading
java.lang.String	<u>slash</u> Flags to operate checks while reading
swarm.collections.ListImpl	<u>unitList</u> the list containig the operating units
boolean	<u>worksheetOrderSequenceFileOpen</u> A flag to check if the orderSequences worksheet file is open
boolean	<u>worksheetRecipeFileOpen</u> A flag to check if the orderSequences worksheet file is open

Fields inherited from class OrderGenerator

dictionary, dictionaryLength, layerNumber, maxStepLength,
maxStepNumber, orderRecipe1, orderRecipe2, orderRecipeTmp,
stepLengthInOrder, totalLayerNumber, useOor1

Constructor Summary

[OrderDistiller](#)(swarm.defobj.Zone aZone, int msn, int msl,
swarm.collections.ListImpl ul, swarm.collections.ListImpl eul,
swarm.collections.ListImpl ol, int tln, VEFrameModelSwarm mo,
AssigningTool at)

Method Summary

void	<u>calculateLength</u> (java.lang.String cTC) This method is used to calculate the length of each row after a strong check of the elements
void	<u>checkForComments</u> (java.lang.String cTC) This method is used to check if there are comments in the cells
void	<u>checkForLayer</u> (ExcelReader e) This method is used to check the presence of a new layer
int	<u>checkTheExistence</u> (int c) This method is used to check the corrispondence with the dictionary of production phases

void	computation (ExcelReader e) This method dial with the computational choice
void	distill () This is the method containing the iterator needed to launch the daily production of recipes.
void	end (ExcelReader e) This method dial with the end choice
int	errorIsNotAnInteger (ExcelReader e) This method is used to check if a type error occurs
void	errorIsNotAString (ExcelReader e) This method is used to check if a type error occurs
int	getOrderSequence1 (int j) This method is used to obtain the elements of the orderSequence1 array
int	getOrderSequence2 (int j) This method is used to obtain the elements of the orderSequence2 array
int	getOrderSequence3 (int j) This method is used to obtain the elements of the orderSequence3 array
void	minute (ExcelReader e)
void	number (ExcelReader e) This method dial with normal or batch choice
void	oR (ExcelReader e) This method dial with the or choice
void	procurement (ExcelReader e) This method dial with the procurement choice
void	readOrderSequence () This method reads from the worksheet containing, shift by shift, the sequence of orders to be launched and fills in the orderSequence1 with the ID codes of recipes and the orderSequence2 with the quantities of each recipe.
void	readRecipes () This is the method needed to read and store the recipes.
void	second (ExcelReader e)
void	setDictionary () This method is used to collect the names of the units and of the end units, so that it can operate the check of correspondency between the production phases required by recipes and the phases of production the units can do.
void	setRecipeContainers () This method is used to set the length of each recipe

Methods inherited from class OrderGenerator

createRandomOrderWithNSteps

Methods inherited from class swarm.objectbase.SwarmObjectImpl

compare, describe, describeID, drop, getCompleteProbeMap, getDisplayName, getName, getProbeForMessage, getProbeForVariable, getProbeMap, getTypeName, getZone, perform, perform\$with, perform\$with\$with, perform\$with\$with\$with, respondsTo, setDisplayName, xfprintf, xfprintfid, xprint, xprintid

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

worksheetOrderSequenceFileOpen

public boolean **worksheetOrderSequenceFileOpen**

A flag to check if the orderSequences worksheet file is open

worksheetRecipeFileOpen

public boolean **worksheetRecipeFileOpen**

A flag to check if the orderSequences worksheet file is open

semicolon

public java.lang.String **semicolon**

Flags to operate checks while reading

checkTheCell

public java.lang.String **checkTheCell**

Flags to operate checks while reading

gate

public java.lang.String **gate**

Flags to operate checks while reading

p

public java.lang.String **p**

Flags to operate checks while reading

sec

public java.lang.String **sec**

Flags to operate checks while reading

min

public java.lang.String **min**
Flags to operate checks while reading

end

public java.lang.String **end**
Flags to operate checks while reading

slash

public java.lang.String **slash**
Flags to operate checks while reading

backslash

public java.lang.String **backslash**
Flags to operate checks while reading

or

public java.lang.String **or**
Flags to operate checks while reading

layer

public java.lang.String **layer**
Flags to operate checks while reading

computation

public java.lang.String **computation**
Flags to operate checks while reading

orderCount

public int **orderCount**
used to record the number of generated orders

firstTime

public static boolean **firstTime**

anOrder

public Order **anOrder**
a specific order

unitList

```
public swarm.collections.ListImpl unitList  
    the list containig the operating units
```

endUnitList

```
public swarm.collections.ListImpl endUnitList  
    the list containig the end units
```

orderList

```
public swarm.collections.ListImpl orderList  
    the list containig all the orders
```

recipeList

```
public swarm.collections.ListImpl recipeList  
    the list containig all the orders
```

Constructor Detail

OrderDistiller

```
public OrderDistiller(swarm.defobj.Zone aZone,  
    int msn,  
    int msl,  
    swarm.collections.ListImpl ul,  
    swarm.collections.ListImpl eul,  
    swarm.collections.ListImpl ol,  
    int tln,  
    VEFrameModelSwarm mo,  
    AssigningTool at)
```

Method Detail

setDictionary

```
public void setDictionary()
```

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

Overrides:

setDictionary in class OrderGenerator

setRecipeContainers

```
public void setRecipeContainers()
```

This method is used to set the length of each recipe

readRecipes

```
public void readRecipes()
```

This is the method needed to read and store the recipes. The procedure follows this steps:It opens the worksheet file recipes.xls, in which are stored the sequences of steps of all the recipes; It makes some check of the routines; It search the object containig the same code of the recipe we are considering; Finally it substitutes the values of the array of that object with the new ones.

calculateLength

```
public void calculateLength(java.lang.String cTC)
```

This method is used to calculate the length of each row after a strong check of the elements

computation

```
public void computation(ExcelReader e)
```

This method dial with the computational choice

procurement

```
public void procurement(ExcelReader e)
```

This method dial with the procurement choice

oR

```
public void oR(ExcelReader e)
```

This method dial with the or choice

end

```
public void end(ExcelReader e)
```

This method dial with the end choice

number

```
public void number(ExcelReader e)
```

This method dial with normal or batch choice

second

```
public void second(ExcelReader e)
```

minute

```
public void minute(ExcelReader e)
```

distill

```
public void distill()
```

This is the method containing the iterator needed to launch the daily production of recipes. It take a look at the orderSequence arrays to determine which recipes must be done and how many times. A

request for which unit can do the first production phase of each recipe will be done to units or endUnits.

readOrderSequence

public void **readOrderSequence**()

This method reads from the worksheet containing, shift by shift, the sequence of orders to be launched and fills in the orderSequence1 with the ID codes of recipes and the orderSequence2 with the quantities of each recipe.

checkTheExistence

public int **checkTheExistence**(int c)

This method is used to check the correspondence with the dictionary of production phases

checkForLayer

public void **checkForLayer**(ExcelReader e)

This method is used to check the presence of a new layer

checkForComments

public void **checkForComments**(java.lang.String cTC)

This method is used to check if there are comments in the cells

errorIsNotAnInteger

public int **errorIsNotAnInteger**(ExcelReader e)

This method is used to check if a type error occurs

errorIsNotAString

public void **errorIsNotAString**(ExcelReader e)

This method is used to check if a type error occurs

getOrderSequence1

public int **getOrderSequence1**(int j)

This method is used to obtain the elements of the orderSequence1 array

getOrderSequence2

public int **getOrderSequence2**(int j)

This method is used to obtain the elements of the orderSequence2 array

getOrderSequence3

public int **getOrderSequence3**(int j)

This method is used to obtain the elements of the orderSequence3 array

Class Recipe

Recipe

public class **Recipe**

extends swarm.objectbase.SwarmObjectImpl

This class is used to record the recipes and their referring number (ID), so we can assign them to a List

Field Summary

java.lang.String	<u>backslash</u> Flags to operate checks while reading
java.lang.String	<u>checkTheCell</u> Flags to operate checks while reading
java.lang.String	<u>computation</u> Flags to operate checks while reading
java.lang.String	<u>end</u> Flags to operate checks while reading
java.lang.String	<u>gate</u> Flags to operate checks while reading
java.lang.String	<u>min</u> Flags to operate checks while reading
java.lang.String	<u>or</u> Flags to operate checks while reading
java.lang.String	<u>p</u> Flags to operate checks while reading
java.lang.String	<u>sec</u> Flags to operate checks while reading
java.lang.String	<u>semicolon</u> Flags to operate checks while reading
java.lang.String	<u>slash</u> Flags to operate checks while reading

Constructor Summary

[**Recipe**](#)(swarm.defobj.Zone aZone, int c, int l, java.lang.String rN)

Method Summary

void	<u>computation</u> (ExcelReader e)
------	---

	This method dial with the computation choice
void	<u>end</u> (ExcelReader e) This method dial with the end choice
int	<u>errorIsNotAnInteger</u> (ExcelReader e) This method is used to check if a type error occur
void	<u>errorIsNotAString</u> (ExcelReader e) This method is used to check if a type error occur
java.lang.String	<u>getCheckTheCell</u> ()
int	<u>getCodeNumber</u> ()
int	<u>getLength</u> ()
int[]	<u>getOrderRecipe</u> ()
java.lang.String	<u>getRecipeName</u> ()
void	<u>minute</u> (ExcelReader e, int step)
void	<u>number</u> (ExcelReader e) This method dial with normal or batch choice
void	<u>oR</u> (ExcelReader e) This method dial with the or choice
void	<u>procurement</u> (ExcelReader e) This method dial with the procurement choice
void	<u>second</u> (ExcelReader e, int step)
void	<u>setSteps</u> (java.lang.String cTC, ExcelReader recipeWorksheet)

Methods inherited from class `swarm.objectbase.SwarmObjectImpl`

compare, describe, describeID, drop, getCompleteProbeMap, getDisplayName, getName, getProbeForMessage, getProbeForVariable, getProbeMap, getTypeName, getZone, perform, perform\$with, perform\$with\$with, perform\$with\$with\$with, respondsTo, setDisplayName, xfprintf, xfprintfid, xprint, xprintid

Methods inherited from class `java.lang.Object`

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

semicolon

```
public java.lang.String semicolon  
    Flags to operate checks while reading
```

checkTheCell

```
public java.lang.String checkTheCell  
    Flags to operate checks while reading
```

gate

```
public java.lang.String gate  
    Flags to operate checks while reading
```

p

```
public java.lang.String p  
    Flags to operate checks while reading
```

end

```
public java.lang.String end  
    Flags to operate checks while reading
```

sec

```
public java.lang.String sec  
    Flags to operate checks while reading
```

min

```
public java.lang.String min  
    Flags to operate checks while reading
```

slash

```
public java.lang.String slash  
    Flags to operate checks while reading
```

backslash

```
public java.lang.String backslash  
    Flags to operate checks while reading
```

or

```
public java.lang.String or  
    Flags to operate checks while reading
```

computation

```
public java.lang.String computation
```

Flags to operate checks while reading

Constructor Detail

Recipe

```
public Recipe(swarm.defobj.Zone aZone,  
              int c,  
              int l,  
              java.lang.String rN)
```

Method Detail

getCodeNumber

```
public int getCodeNumber()
```

getOrderRecipe

```
public int[] getOrderRecipe()
```

getLength

```
public int getLength()
```

getRecipeName

```
public java.lang.String getRecipeName()
```

setSteps

```
public void setSteps(java.lang.String cTC,  
                    ExcelReader recipeWorksheet)
```

getCheckTheCell

```
public java.lang.String getCheckTheCell()
```

errorIsNotAnInteger

```
public int errorIsNotAnInteger(ExcelReader e)  
    This method is used to check if a type error occur
```

errorIsNotAString

```
public void errorIsNotAString(ExcelReader e)  
    This method is used to check if a type error occur
```

computation

```
public void computation(ExcelReader e)  
    This method dial with the computation choice
```

procurement

public void **procurement**(ExcelReader e)
This method dial with the procurement choice

oR

public void **oR**(ExcelReader e)
This method dial with the or choice

end

public void **end**(ExcelReader e)
This method dial with the end choice

number

public void **number**(ExcelReader e)
This method dial with normal or batch choice

second

public void **second**(ExcelReader e,
int step)

minute

public void **minute**(ExcelReader e,
int step)

Capitolo 6 – Revisione della prima formalizzazione del modello

La formalizzazione di un sistema complesso come può essere quello di un'azienda, ed in particolare della realtà aziendale descritta nel capitolo precedente, necessita di continue revisioni. Il nostro caso non fa eccezione: analizzando in modo sempre più approfondito l'applicazione VIR al modello si sono riscontrati diversi adattamenti e miglioramenti.

Ciò che ha permesso di rilevare i cambiamenti necessari è stata la continua sperimentazione, grazie anche ad esempi molto semplificati, ma efficaci per quanto riguarda la comprensione dei meccanismi sottostanti al modello. Di fondamentale importanza è stato osservare lo svolgimento della simulazione e monitorarne il funzionamento.

Qui di seguito verrà riportata una seconda formalizzazione del modello scaturita dalla risoluzione di alcuni problemi pratici e concettuali.

6.1 Sviluppi inerenti agli ordini ed alle unità

Un primo sviluppo è stato quello dell'assegnazione dei passi di una lavorazione alla stessa unità. I sequential batch non riuscivano a formare i lotti da mandare in lavorazione perché per controllare che i prodotti fossero uguali viene utilizzato un criterio di eguaglianza basato sui vettori di stato delle ricette. Nel vettore di stato viene inserito il valore 0 se il passo è ancora da compiere, altrimenti viene inserito il codice dell'unità che lo ha svolto.

Prendiamo in considerazione un esempio concreto, riferito all'azienda VIR, per capire meglio il problema.

Se ci occupiamo, ad esempio, della ricetta 1000023 si individua la lavorazione 101 che serve per la produzione del manicotto grande. I sedici tick relativi a questa lavorazione possono essere svolti da cinque

unità differenti: la numero due, tre, quattro, dieci e undici. Nel vettore di stato saranno dunque registrati codici sempre diversi.

manicottogrande	1000023	p	1	4	101	s	16	100008	s	2	\	10	e	1101	;
-----------------	---------	---	---	---	-----	---	----	--------	---	---	---	----	---	------	---

Per ovviare a questo problema i passi della lavorazione in corso di produzione vengono assegnati sempre alla stessa unità.

In questo caso i sedici passi della lavorazione 101 vengono riassegnati alla prima unità, fra quelle citate sopra, che risponderà alla chiamata.

Un altro sviluppo è stato quello della gestione di gruppi di ordini che utilizzano la stessa unità. Ecco un esempio:

101	s	5	101	s	5	101	s	5
-----	---	---	-----	---	---	-----	---	---

I passi che risultano formalmente distinti saranno trattati come se fossero uniti: in questo caso 101 s 15.

Per fare questo è stato inserito un flag, nella probe del model, denominato sameStepLifoAssignment, se è 'true' gli ordini che tornano alla stessa unit da cui provengono vanno in testa alla waitingList.

6.2 Tempo e quantità

Nel modello jES-VIR si prendono in considerazione due turni al giorno da 8 ore, ogni turno di 8 ore è dunque composto da 480 minuti e da 28.800 secondi.

Ogni tick nella simulazione è uguale ad un secondo quindi ogni turno dovrebbe durare 28800 ticks. La simulazione, strutturata in questo modo, era troppo pesante come tempo di calcolo. Si sono così adottate delle semplificazioni: d'apprima si è portato il numero dei ticks per ogni turno a 288 con la conseguenza che un tick diventava uguale a 100

secondi, ma anche così la simulazione era troppo lenta; si è giunti così alla situazione attuale nella quale ci sono 29 ticks per turno ed ogni tick è uguale a 1000 secondi.

Adottando questa semplificazione, erano state effettuate le seguenti modifiche:

- conformazione del file orderSequence.xls dividendo per 1000 le quantità degli ordini lanciati;
- inserimento nel parametro ticksInATimeUnit del valore 29;
- ricette invariate.

In questo modo però c'era un problema di sincronia fra secondi e ticks proprio perché le ricette erano rimaste invariate.

Consideriamo adesso un caso pratico:

bullone	1000059	100014	s	433	/	200	e	14	;
---------	---------	--------	---	-----	---	-----	---	----	---

per ottenere, ad esempio, i bulloni (ricetta 1000059) per cui era necessaria una lavorazione di 433 secondi, e che quindi con 29.000 secondi per turno avrebbero dovuto essere pronti all'inizio del primo turno (un turno va da 0 a 29 ticks), non ci voleva solo qualche frazione di tick come sarebbe stato giusto che fosse ma si vedevano spuntare i suddetti bulloni (corrispondenti alla end unit 14 nel grafico Procurement int. or ext.) dopo circa 433 ticks.

La soluzione temporaneamente adottata è stata quella di intervenire direttamente sulle ricette dividendo per 1000 i secondi.

6.3 Risultati ottenuti e problemi ancora aperti

Si ha ora un'azienda virtuale funzionante che produce e della quale verrà analizzato l'andamento per determinare quali possano essere gli eventuali "colli di bottiglia" o le eventuali disfunzioni.

I risultati e l'analisi degli stessi è riportata sul sito ospitato dal Dipartimento di scienze economiche e finanziarie "G. Prato" eco83.econ.unito.it/tesive. Qui si troverà anche la versione aggiornata del programma di base che diventerà un pacchetto unificato con il caso aziendale VIR.

I problemi ancora aperti sono quelli relativi ai tempi ed alle quantità dei quali si è parlato sopra e quelli relativi alla contabilità.

Per quanto riguarda i primi verranno gestiti in futuro da un OrderDistiller "intelligente" in grado di trattare scale di tempi e di quantità differenti e di convertire automaticamente i secondi in minuti e così via.

La contabilità è tuttora in fase di sviluppo per tenere conto dei formalismi recentemente introdotti, come i procurement per i quali è stato necessario considerare la ricorsività, ossia la computazione di tutti i passi richiamati, i batch per i quali è necessario tener conto dei costi dei lotti e così via.

Capitolo 7 – UML (Unified Modeling Language)

Obiettivo di questo capitolo è fornire una panoramica generale sull'UML: chiarire cosa si intende per UML e quali siano le relative aree di competenza e gli obiettivi. In particolare viene esposta l'organizzazione in viste e vengono presentati molto brevemente i diagrammi di cui ciascuna di essa è composta.

Per lo sviluppo di un modello di simulazione applicato ad un caso concreto, è di fondamentale importanza la collaborazione fra gli analisti-sviluppatori del codice e l'impresa stessa. Un mezzo per superare questo gap, e quindi evitare incomprensioni fra le specifiche dell'azienda reale e la realizzazione del modello, è proprio l'adozione di una notazione UML.

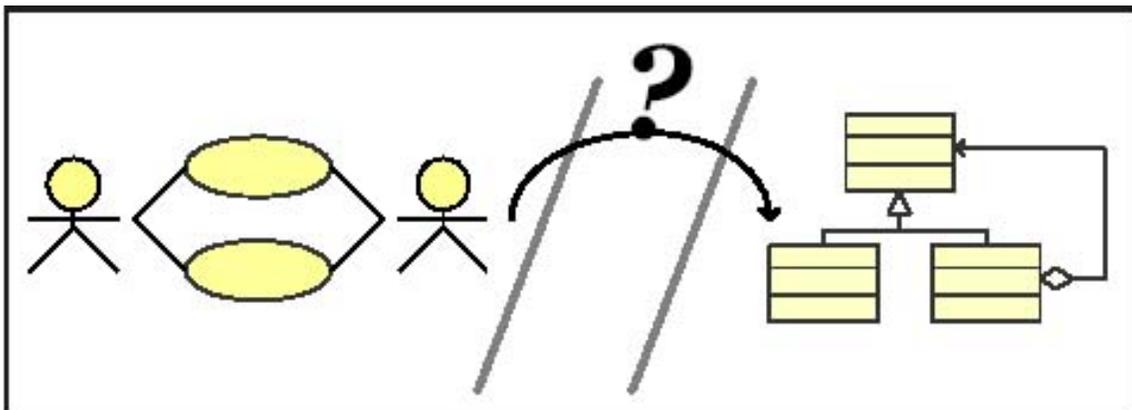


Figura 1: gap della progettazione del software.

L'interesse per questo strumento scaturisce dalla ricerca di migliorare la progettazione e l'uso di jES attraverso l'adozione di una schematizzazione standard, adattabile a diversi ambienti.

7.1 La notazione UML

UML (Unified Modeling Language) è un metodo per descrivere l'architettura di un sistema e costituisce l'evoluzione e l'unificazione (da

cui il nome) di tre nozioni precedentemente esistenti: Booch (dell'omonimo autore), OMT (di Rumbaugh) e OOSE (di Jacobson).

Questo strumento di analisi molto versatile, nasce per risolvere le problematiche connesse alla progettazione Object Oriented del software, ma si adatta bene ad essere utilizzato negli ambienti più disparati.

Attualmente è disponibile la versione 1.4 e si sta lavorando alla versione 2.0. L'evoluzione dell'UML è mostrata in figura 1, attraverso il diagramma dei componenti, uno degli strumenti messi a disposizione del linguaggio.

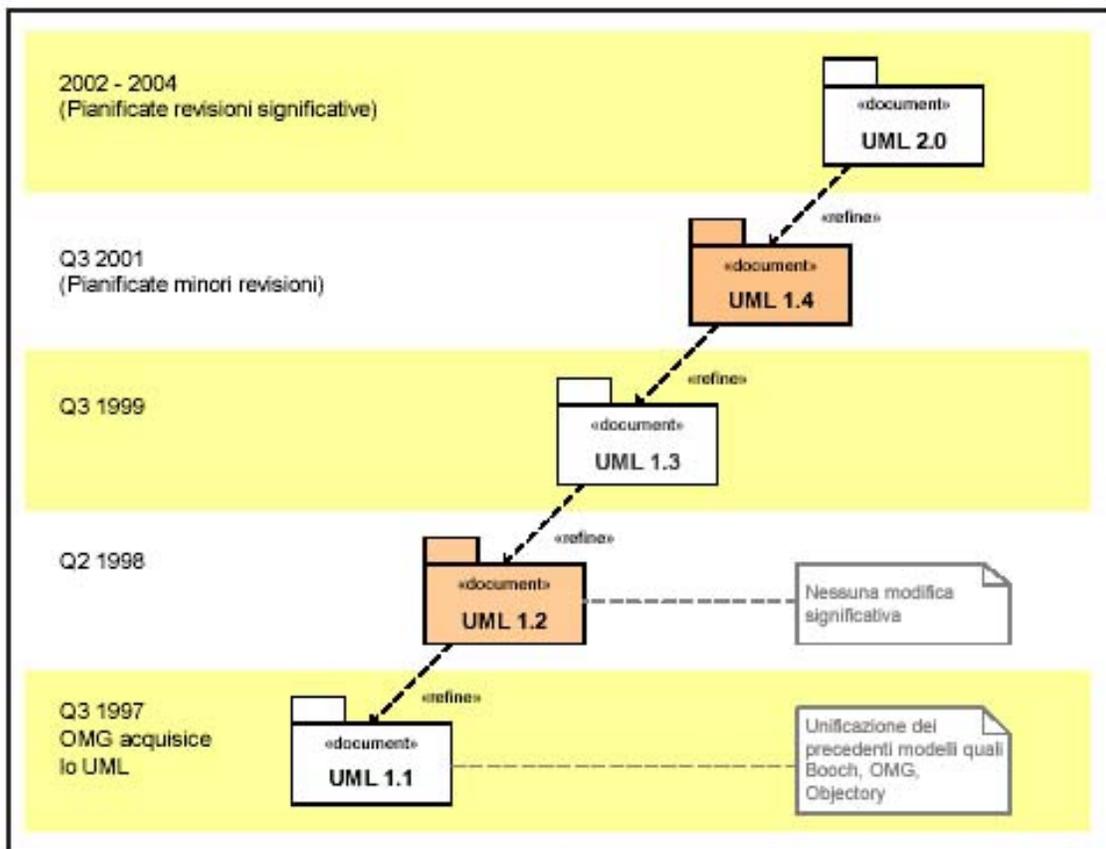


Figura 2: diagramma dell'evoluzione dell'UML.

I principali obiettivi che ci si prefigge di raggiungere attraverso lo sviluppo dell'UML sono i seguenti:

- fornire agli utenti un linguaggio di modellazione visuale pronto ad essere utilizzato per sviluppare e scambiare modelli espressivi. L'UML ha consolidato un insieme nucleare di concetti di modellazione, necessari in molte applicazioni reali, presenti in molti metodi di progettazione e tool di modellazione disponibile sul mercato;
- fornire meccanismi di estensione e specializzazione al fine di accrescere i concetti presenti nel nucleo. Un punto fermo di questo lavoro è stato quello di realizzare un linguaggio di modellazione quanto più generale possibile, in modo da non relegarne l'utilizzo ad un dominio specifico;
- supportare specifiche che risultino indipendenti da un particolare linguaggio di programmazione o processo di sviluppo. Chiaramente l'UML risulta particolarmente indirizzato a linguaggio di programmazione Object Oriented, ma è stato reso idoneo all'utilizzo nei più svariati ambienti ed inoltre si è fatto in modo di preservare la sua adattabilità a sviluppi futuri;
- fornire le basi formali per comprendere il linguaggio di modellazione che deve necessariamente essere preciso e al contempo offrire una complessità contenuta. L'UML, infatti, fornisce una definizione formale del modello, utilizzando un metamodello espresso attraverso il diagramma delle classi dello stesso UML;
- incoraggiare la crescita del mercato dei tool Object Oriented attraverso un linguaggio di modellazione standard che permettesse di superare alcuni problemi quali: la difficoltà per le aziende di selezionare standard da adottare, la frustrazione da parte dei tecnici di fronte alla proliferazione dei metodi, la difficoltà di circolazione dei vari modelli prodotti, ecc.;

L'UML permette di visualizzare, per mezzo di un formalismo rigoroso, "manufatti" dell'ingegneria, consentendo di illustrare idee, decisioni prese e soluzioni adottate.

Tale linguaggio favorisce, inoltre, la divulgazione delle informazioni, in quanto standard internazionale non legato alle singole imprese. In teoria, un qualunque tecnico, di qualsivoglia nazionalità, dipendente della più ignota delle software house, con un minimo di conoscenza dell'UML dovrebbe essere in grado di leggere il modello del progetto e di comprenderne ogni dettaglio senza troppa fatica e, soprattutto senza le ambiguità tipiche del linguaggio naturale.

Lo Unified Modeling Language non è unicamente una notazione standard per la descrizione di modelli Object Oriented di sistemi software; si tratta bensì di un metamodello definito rigorosamente che, a sua volta è istanza di un meta-metamodello definito altrettanto formalmente. L'UML (metamodello) permette di realizzare diversi modelli Object Oriented (modelli definiti dall'utente). Il metamodello dell'UML definisce la struttura dei modelli UML. Un metamodello non fornisce alcuna regola su come esprimere i concetti del mondo OO, quali ad esempio classi, interfacce, relazioni e così via, ma esso rende possibile avere diverse notazioni che si conformano alla metamodello stesso.

Nella figura 2 vengono illustrate graficamente quanto le relazioni esistenti tra il meta-metamodello, il metamodello e il modello dell'utente. Scorrendo il diagramma dall'alto verso il basso si assiste ad una graduale diminuzione del livello di astrazione: se si fosse rappresentato un ulteriore livello, si sarebbero trovate le istanze della classe del modello dell'utente (gli oggetti).

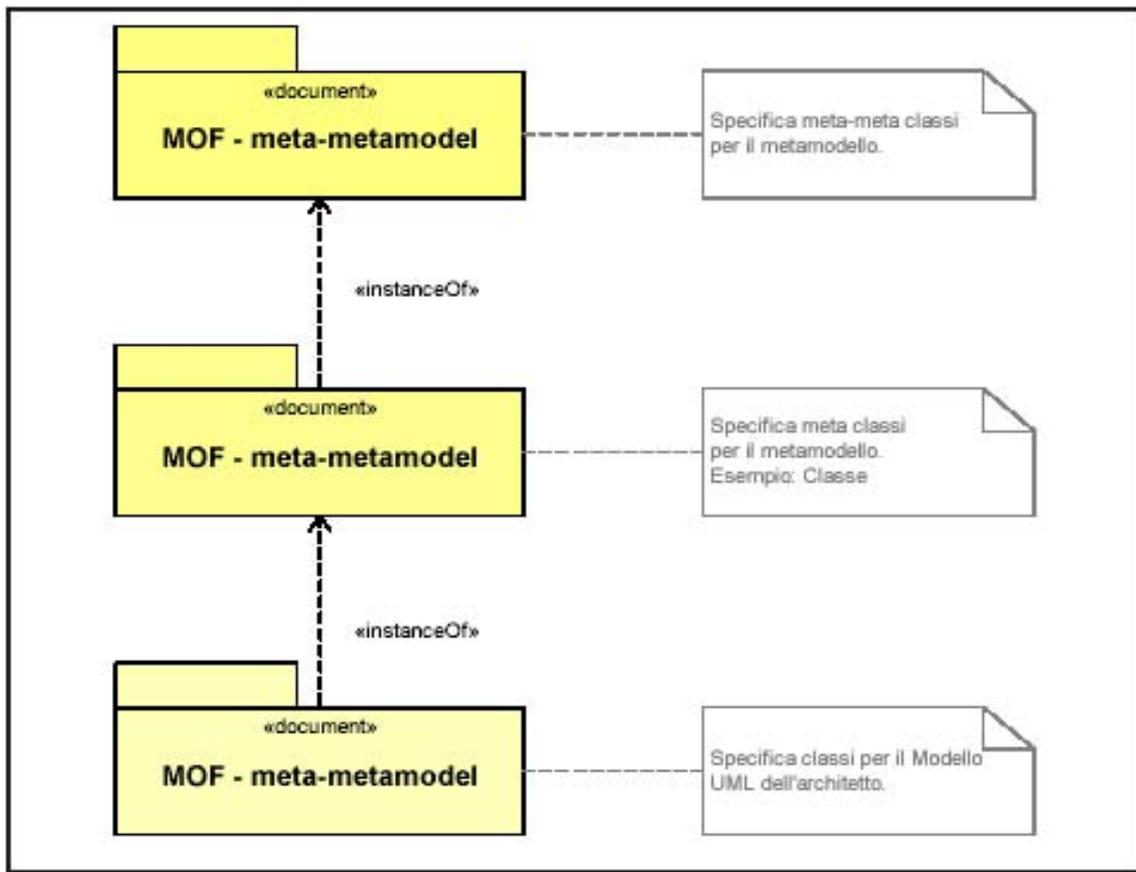


Figura 3: meta-metamodello, metamodello e modello UML.

7.2 La struttura

L'UML presenta una tipica struttura a strati; procedendo dall'esterno verso l'interno, essa è costituita da:

1. viste;
2. diagrammi;
3. elementi del modello.

Le viste mostrano i diversi aspetti di un sistema per mezzo di un insieme di diagrammi. Si tratta di astrazioni, ognuna delle quali analizza il sistema da modellare secondo un'ottica diversa e ben precisa (funzionale, non funzionale, organizzativa, ecc.), la cui totalità fornisce il quadro d'insieme.

I diagrammi permettono di descrivere graficamente le viste logiche. L'UML prevede ben nove tipi di diagrammi differenti, ognuno dei quali è particolarmente appropriato ad essere utilizzato in particolari viste.

Per ciò che concerne gli elementi del modello, essi sono i concetti che permettono di realizzare i vari diagrammi. Alcuni esempi di elementi sono: attori, packages, oggetti, e così via. In UML sono definiti e standardizzati un certo numero di elementi base invariati (core, nucleo), ritenuti fondamentali, e sono forniti un insieme di altri meccanismi atti ad estendere la semantica del linguaggio (stereotypes) per aggiungere documentazione, note, vincoli, ecc..

Le viste

In prima analisi l'UML è organizzato in viste, ed in particolare è costituito dalle viste Use Case, Design, Implementation, Component e Deployment come illustrato in figura 4.

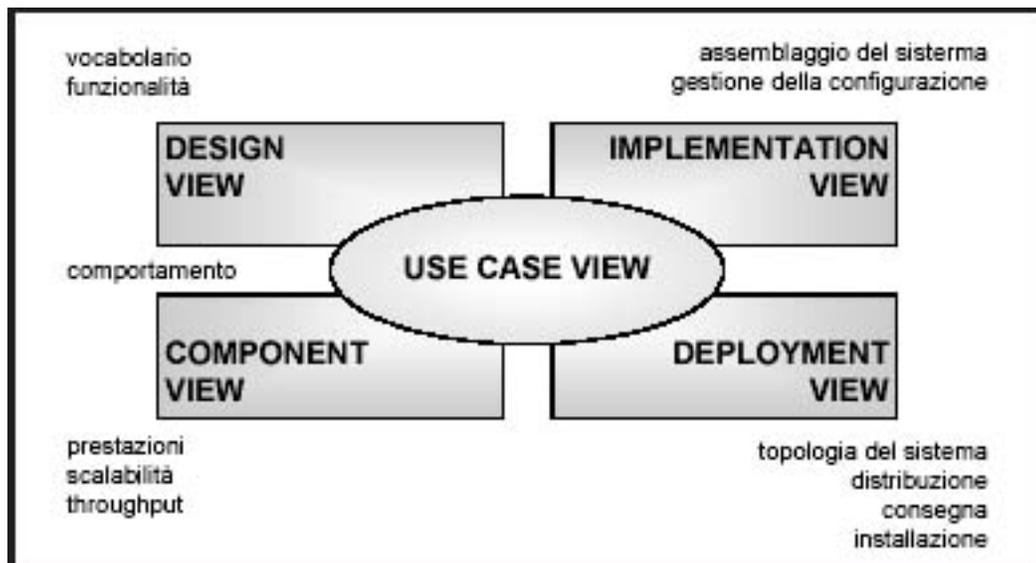


Figura 4: diagramma delle viste dell'UML.

In breve, la prima vista, Use Case View (vista dei casi d'uso), è utilizzata per analizzare i requisiti utente: specifica le funzionalità del sistema come vengono percepite dalle entità esterne al sistema stesso dette attori. Dunque, si tratta di una vista ad alto livello di astrazione e di importanza fondamentale, nella quale è necessario individuare tutti gli attori, i casi d'uso e le relative associazioni.

La vista dei casi d'uso è costituita da due proiezioni: quella statica catturata dai diagrammi dei casi d'uso e quella dinamica rappresentante le interazioni tra gli attori e il sistema.

La Design View (vista di disegno, talune volte indicata come Logical View, vista logica), descrive come le funzionalità del sistema debbano essere realizzate, in altre parole si analizza il sistema dall'interno. Anche questa vista è composta sia dalla struttura statica (diagramma delle classi e diagramma degli oggetti), sia dalla collaborazione dinamica dovuta alle interazioni tra gli oggetti che lo costituiscono (diagrammi di comportamento del sistema).

La Implementation View (vista implementativa, detta anche Component View, vista dei componenti) è la descrizione di come il codice (classi per i linguaggi object oriented) debba essere accomunato in opportuni moduli (package) evidenziandone le reciproche dipendenze.

La Process View (vista dei processi, detta anche Concurrency View, vista della concorrenza), rientra nell'analisi degli aspetti non funzionali del sistema e consiste nell'individuare i processi e i processori. Ciò sia al fine di dar luogo ad un utilizzo efficiente delle risorse, sia per poter stabilire l'esecuzione parallela degli oggetti (almeno quelli più importanti), sia per gestire correttamente eventuali eventi asincroni, e così via.

La Deployment View (vista di "dispiegamento"), mostra l'architettura fisica del sistema e fornisce l'allocazione delle componenti software nella struttura stessa.

I diagrammi

I diagrammi definiti dall'UML sono quelli riportati in figura 5.

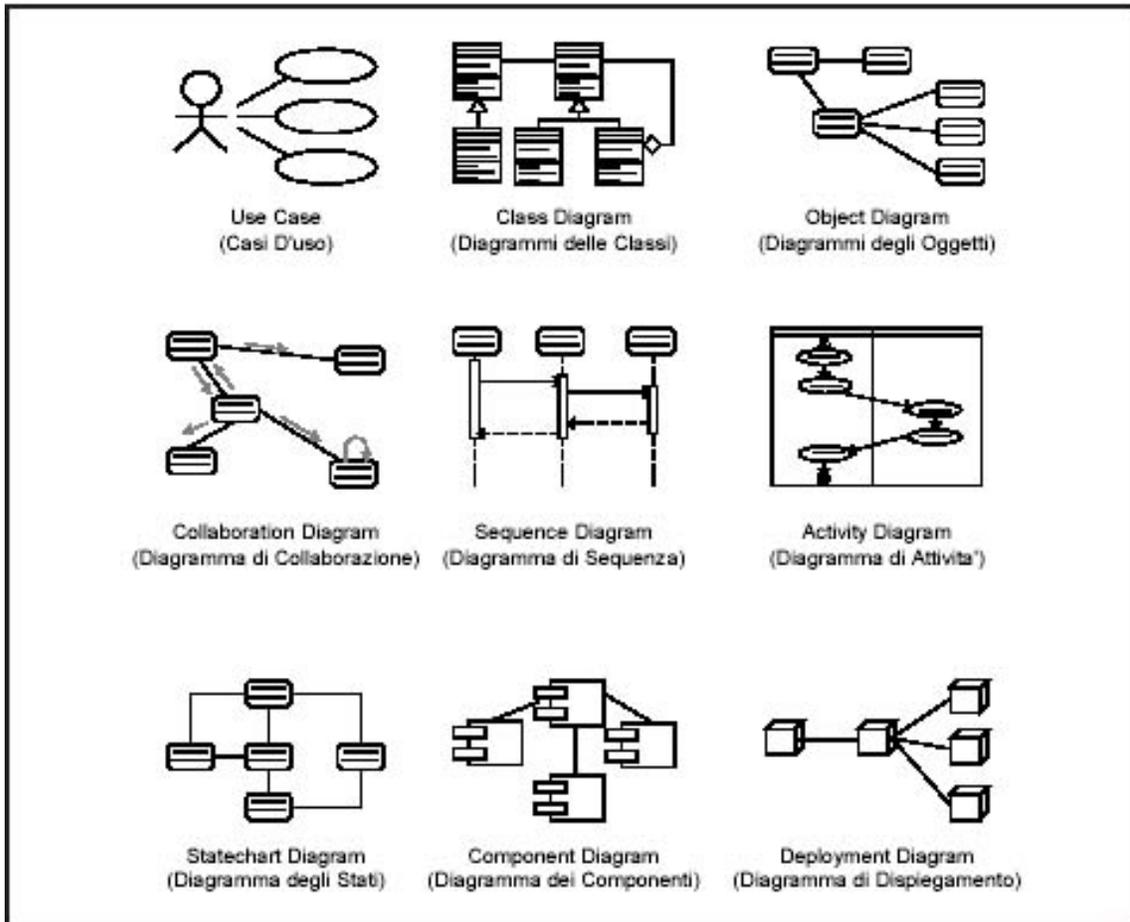


Figura 5: diagrammi dell'UML.

Diagrammi strutturali (logico):

- Use Case Diagram (diagramma dei casi d'uso, da non confondersi con la relativa vista)
- Class Diagram (diagramma delle classi)
- Object Diagram (diagramma degli oggetti)

Diagrammi di comportamento:

- Statechart Diagram (diagramma degli stati)
- Activity Diagram (diagramma delle attività)

Diagrammi di interazione:

- Sequence Diagram (diagramma di sequenza)
- Collaboration Diagram (diagramma di collaborazione)

Diagrammi di implementazione:

- Component Diagram (diagramma dei componenti)
- Deployment Diagram (diagramma di “dispiegamento”)

Per meglio comprendere la struttura dei diagrammi analizziamo di seguito, prendendo in considerazione un unico esempio, un sistema di commercio elettronico di un teatro.

Use Case Diagram

I diagrammi dei casi d'uso mostrano un insieme di entità esterne al sistema, detta attori, associati con le funzionalità, dette a loro volta *Use Case* (casi d'uso), che il sistema dovrà realizzare. L'interazione tra gli attori e i casi d'uso viene espressa per mezzo di una sequenza di messaggi scambiati tra gli attori e il sistema. L'obiettivo dei diagrammi dei casi d'uso è definire un comportamento coerente senza rivelare la struttura interna del sistema.

Nel contesto UML, un attore è una qualsiasi entità esterna al sistema che interagisce con esso: può essere un operatore umano, un dispositivo fisico qualsiasi, e così via.

In figura 6 viene riportato lo Use Case Diagram relativo alla vendita dei biglietti teatrali. Analizzando tale diagramma si può notare che, in primo luogo, gli attori possono essere suddivisi in due macrocategorie: quelli umani (Addetto allo sportello e Cliente) e quelli non umani (Credit Card Authority e Sistema di back office). La suddivisione degli attori umani del sistema in clienti e addetti allo sportello è dovuta alle modalità con cui sarà possibile fruire del sistema: attraverso qualsiasi stazione remota Internet e per mezzo di

appositi postazioni ubicate nelle varie biglietterie. E' presente, inoltre, anche un attore astratto, denominato Utente, atto a raggruppare il comportamento comune di altri attori. Dall'analisi degli attori non umani, si può notare che:

- l'acquisto on-line di biglietti è fruibile solo ai possessori di carta di credito e le relative transazione sono subordinate all'autorizzazione fornita da un apposito sistema: Credit Card Authority;
- il sistema rappresenta una sorta di front office, mentre tutte le attività di contabilità, amministrative, ecc. vengono effettuate da un apposito sistema: Sistema di back office (BCS). I due sistemi dialogano tra loro attraverso l'invio di opportuni messaggi.

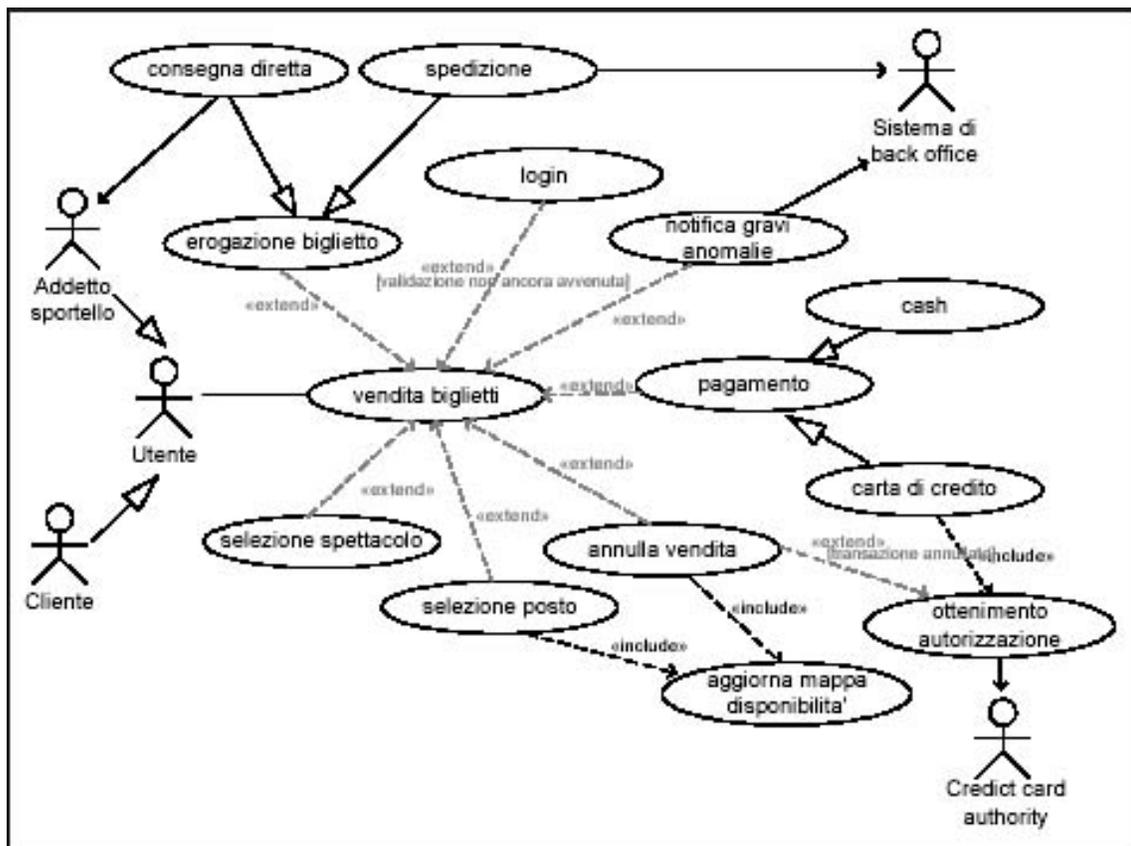


Figura 6: diagramma dei casi d'uso della vendita dei biglietti.

In primo luogo il caso d'uso di vendita biglietti prevede l'autenticazione dell'utente (login). Si tratta di un extend - e quindi di un comportamento opzionale - in quanto è necessario eseguire il caso d'uso solo se l'utente non sia già stato precedentemente riconosciuto nell'arco della stessa sessione. Se il riconoscimento fallisce, l'esecuzione termina, e quindi tutte le altre funzioni non devono essere eseguite.

Nel caso in cui tutto proceda bene, l'utente seleziona lo spettacolo desiderato e quindi i posti desiderati tra quelli disponibili. La selezione del posto include l'esecuzione della funzione di aggiornamento della mappa di disponibilità: i posti "prenotati" non devono ovviamente poter essere selezionati da altri utenti. Selezionati anche i posti, è necessario effettuare il pagamento. Nel caso la fruizione del sistema avvenga da uno sportello vendita biglietti, il pagamento può avvenire o tramite contante o per mezzo di carta di credito, in tutti gli altri casi solo per mezzo di carta di credito.

Nel caso in cui il cliente opti per un pagamento tramite carta di credito è necessario richiedere l'autorizzazione ad un apposito sistema esterno, indicato genericamente con il nome di Credit Card Authority. Nel caso in cui l'autorizzazione venga negata, è necessario annullare la vendita e quindi rendere nuovamente disponibili i posti precedentemente prenotati.

Effettuato con successo anche il pagamento, è possibile effettuare l'erogazione del biglietto. Nel caso dello sportello è possibile fornirlo direttamente al cliente. Nel caso in cui il cliente stia fruendo del sistema attraverso le altre modalità, potrà decidere se richiedere il recapito presso l'indirizzo impostato o ritirarlo in uno degli uffici oppure direttamente a teatro.

Nel caso in cui si verificano dei problemi gravi è prevista immediata notifica al sistema di back office.

Class Diagram

Il diagramma delle classi (figura 7) definisce gli elementi base del sistema e la sua visione statica con l'individuazione delle classi e delle relazioni tra loro.

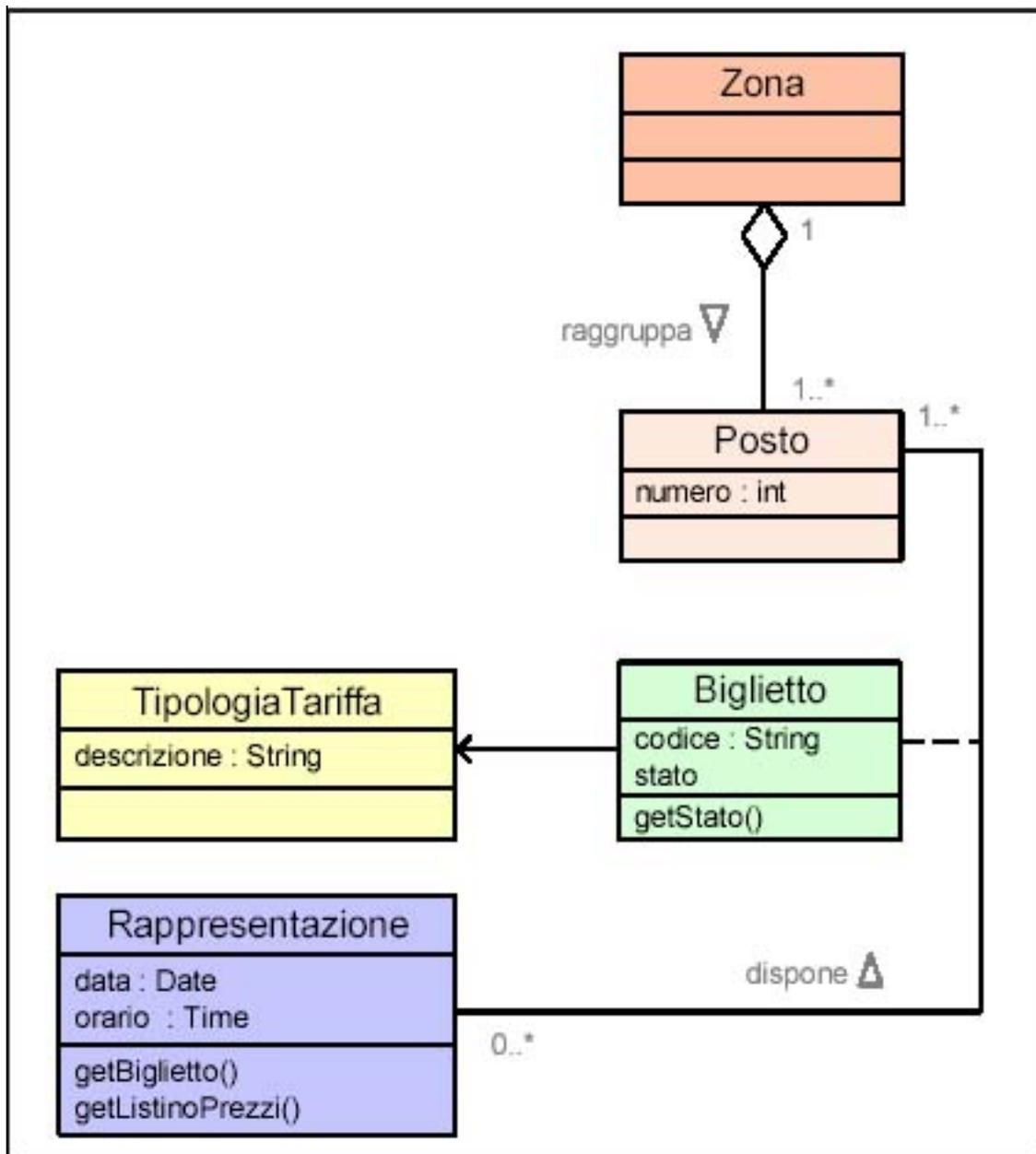


Figura 7: diagramma parziale delle classi relativo ai biglietti ed alle loro tariffe.

In UML una classe è composta da tre parti:

1. nome;
2. attributi (lo stato) sono caratteristiche delle classi che devono essere definite in modo preciso;
3. metodi (il comportamento).

Nel caso preso in considerazione (biglietti e tariffe) abbiamo cinque classi:

- Zona
- Posto
- Tipologia Tariffa
- Biglietto
- Rappresentazione

Il numero accanto alle classi è indicativo della molteplicità ovvero il numero minimo e massimo di oggetti che possono essere relazionati ad un altro oggetto e segnala se l'associazione è obbligatoria oppure no. Le relazioni possono essere descritte tramite la cardinalità, il nome e un particolare elemento grafico che ne chiarisca il tipo (Adornments).

Object Diagram

I diagrammi degli oggetti rappresentano una variante dei diagrammi delle classi, tanto che anche la notazione utilizzata è pressoché equivalente con le sole differenze che i nomi degli oggetti vengono sottolineati e le relazioni vengono dettagliate.

Il diagramma degli addetti proposto in figura 8, mostra una porzione di due ipotetici listini prezzi applicabili al teatro, denominati rispettivamente Alta Stagione (AS) e Bassa Stagione (BS).

Come si può notare, entrambi applicano lo stesso template previsto per la suddivisione in fasce tariffarie, denominato standard e composto dagli oggetti istanza della classe FasciaTariffaria: Super, Lusso, Ordinaria e Scontata.

A ciascuno di questi oggetti si può associare un particolare oggetto TemplateTariffe in funzione anche del listino che si prende in considerazione. A tal fine è prevista la classe (associazione) ApplTipologie che appunto associa un listino ad un opportuno TemplateTariffe.

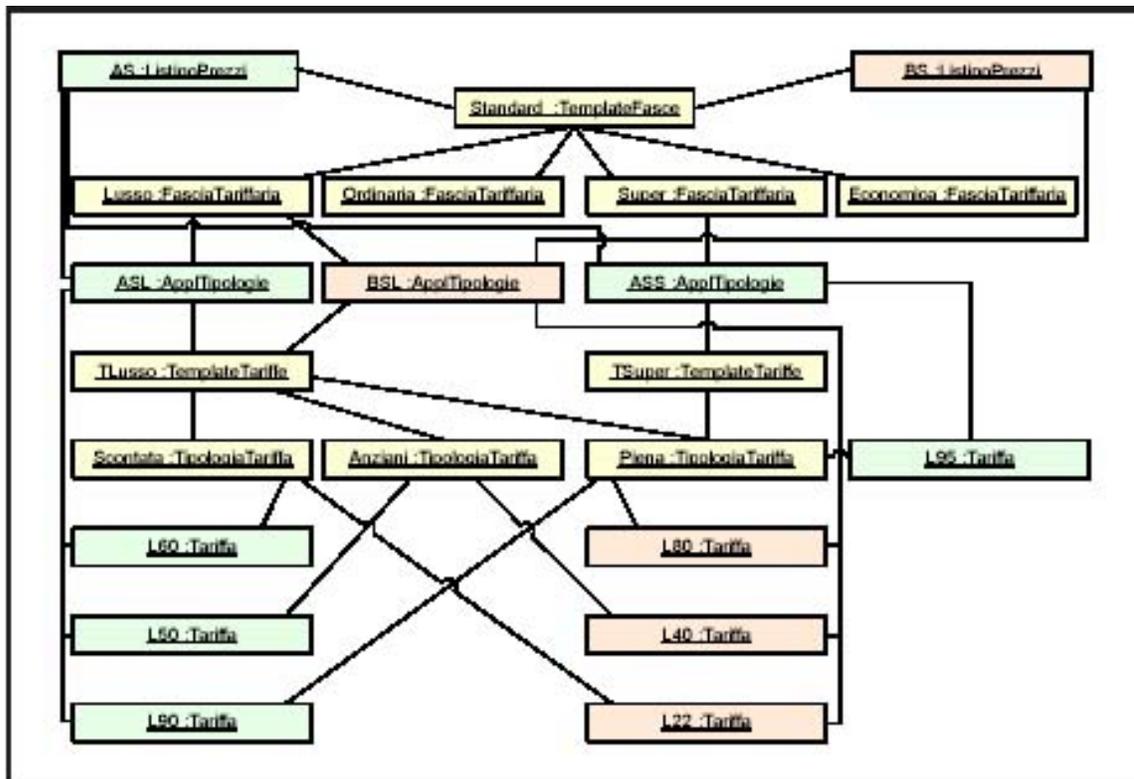


Figura 8: diagramma degli oggetti relativo al listino prezzi.

Statechart Diagram

I diagrammi di stato essenzialmente descrivono automi a stati finiti e pertanto sono costituiti da un insieme di stati, transazioni tra di essi, eventi e attività. Ogni stato rappresenta un periodo di tempo ben delimitato della vita di un oggetto durante il quale l'oggetto stesso soddisfa precise condizioni.

I diagrammi degli stati pertanto, modellano la possibile storia della vita di un oggetto, vengono utilizzati principalmente come completamento

della descrizione delle classi: concorrono a modellarne il comportamento dinamico.

Prendendo in considerazione l'esempio riportato in figura 9, un biglietto per un preciso spettacolo può trovarsi nello stato *Disponibile* (non è stato ancora acquistato o prenotato che lo spettacolo non è andato in scena), *Prenotato* (il biglietto è stato riservato e di tempo a disposizione per acquistarlo non è scaduto), *Acquistato* (l'importo del biglietto è stato versato) e *Annullato* (il tempo limite per l'acquisto è trascorso).

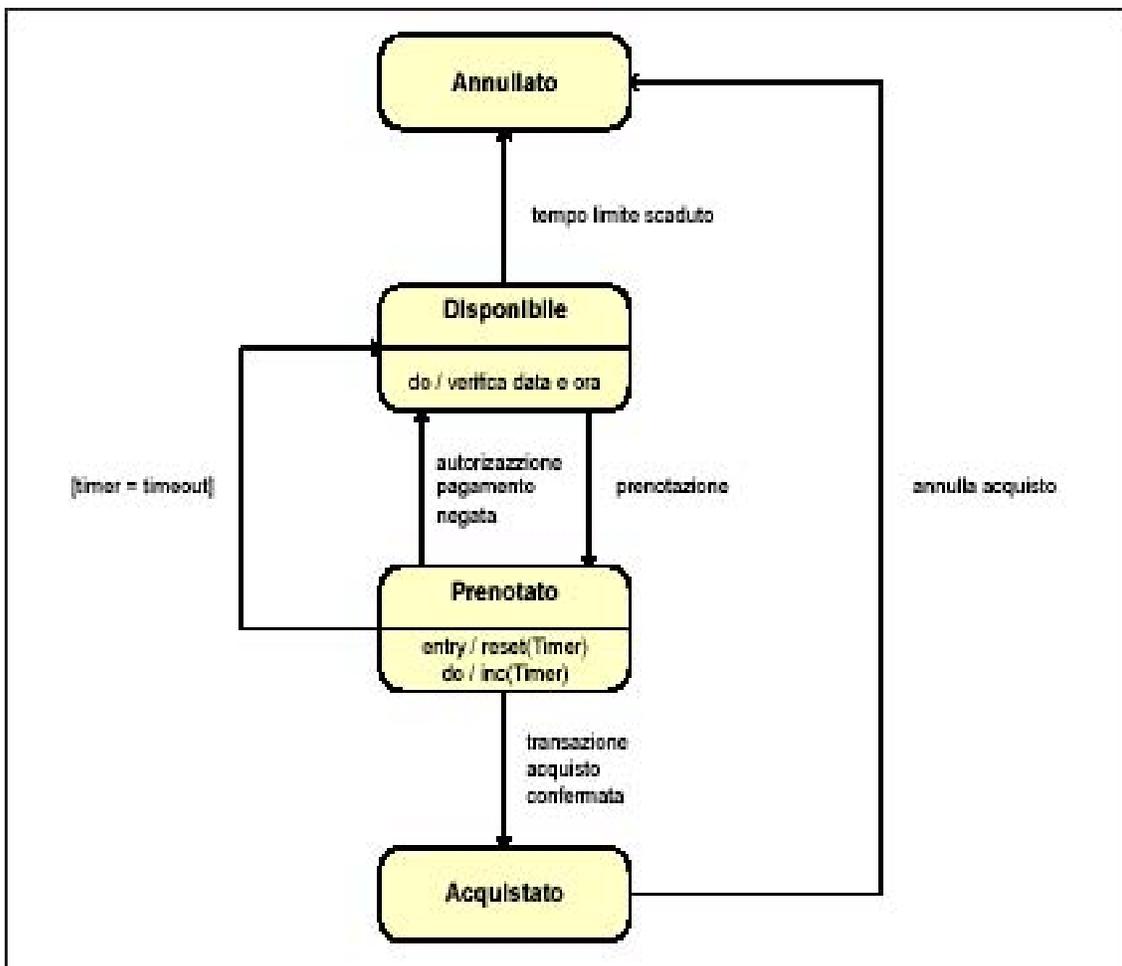


Figura 9: diagramma di stato del ciclo di vita di un biglietto.

Come si può notare esistono due tipi di transazioni:

- provocate dall'esterno (per esempio il Credit Card Authority autorizza la transazione di acquisto e quindi provoca la transizione del biglietto nello stato *Acquistato*);
- che scaturiscono internamente (per esempio scade il tempo a disposizione per poter acquistare il biglietto, e lo stesso transita nello stato di *Annullato*).

Activity Diagram

I diagrammi delle attività mostrano l'evoluzione di un flusso di attività, ognuna delle quali è definita come un'esecuzione continua non atomica all'interno di uno stato: un diagramma di attività mostra una procedura. Formalmente essi sono molto simili ai diagrammi di flusso e proprio per questo risultano un valido ausilio per documentare funzionalità da sottoporre al vaglio di personale non tecnico: clienti e utenti.

Si tratta di una variante dei diagrammi di stato in cui ogni stato rappresenta l'esecuzione di un'opportuna attività e la transizione da uno stato a quello successivo è generata dal completamento dell'attività stessa.

Gli stati evidenziati negli Activity Diagram sono essenzialmente di due tipologie: Activity State (stati di attività) e Action State (stati di azione). I primi consistono in stati che eseguono una computazione la quale, una volta ultimata, genera la transizione allo stato successivo; uno stato di azione, invece, è uno stato atomico (ossia non può essere interrotto).

I diagrammi di attività permettono di enfatizzare la sequenzialità e la concorrenza degli step di cui si compone una particolare procedura.

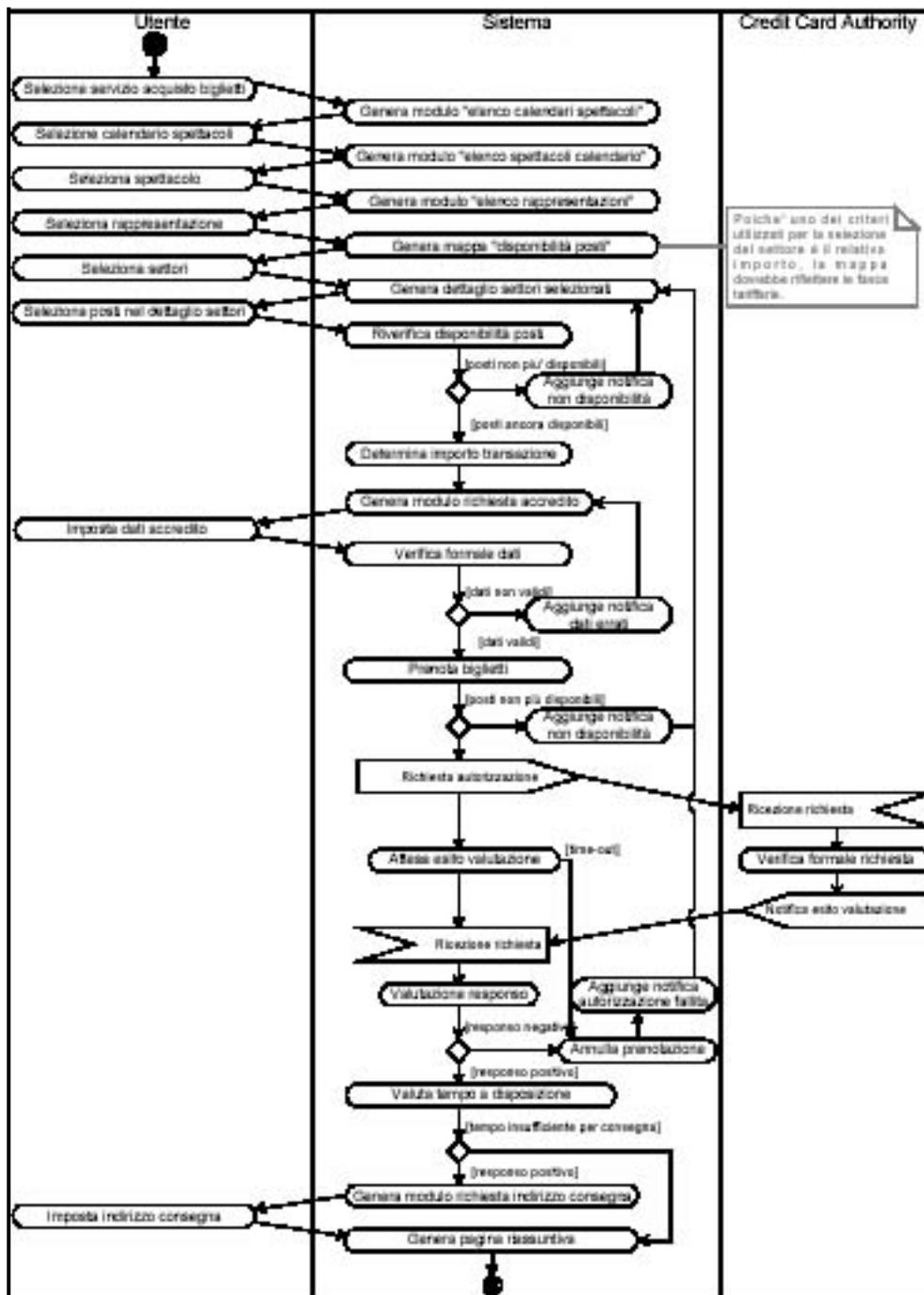


Figura 10: diagramma delle attivit  dell'acquisto dei biglietti.

Interaction Diagrams

I diagrammi di sequenza e di collaborazione - detti anche di interazione in quanto mostrano le interazioni tra oggetti che costituiscono il sistema

e attori esterni allo stesso - vengono utilizzati per modellare il comportamento dinamico del sistema.

I due diagrammi risultano molto simili e si può passare agevolmente dall'una all'altra rappresentazione.

Sequence e Collaboration si differenziano per via dell'aspetto dell'interazione a cui conferiscono maggior rilievo: i diagrammi di sequenza focalizzano l'attenzione sull'ordine temporale dello scambio di messaggi, i diagrammi di collaborazione mettono in risalto l'organizzazione degli oggetti che si scambiano messaggi. Possono essere utilizzati con diversi livelli di astrazione in funzione dell'utilizzo che se ne vuole fare.

In figura 11 viene riportato un esempio di utilizzo del sequence diagram nel modello del disegno: mostra come gli oggetti istanze delle classi collaborino tra loro per realizzare un determinato servizio, in questo caso la verifica della disponibilità di un posto per una specifica rappresentazione.

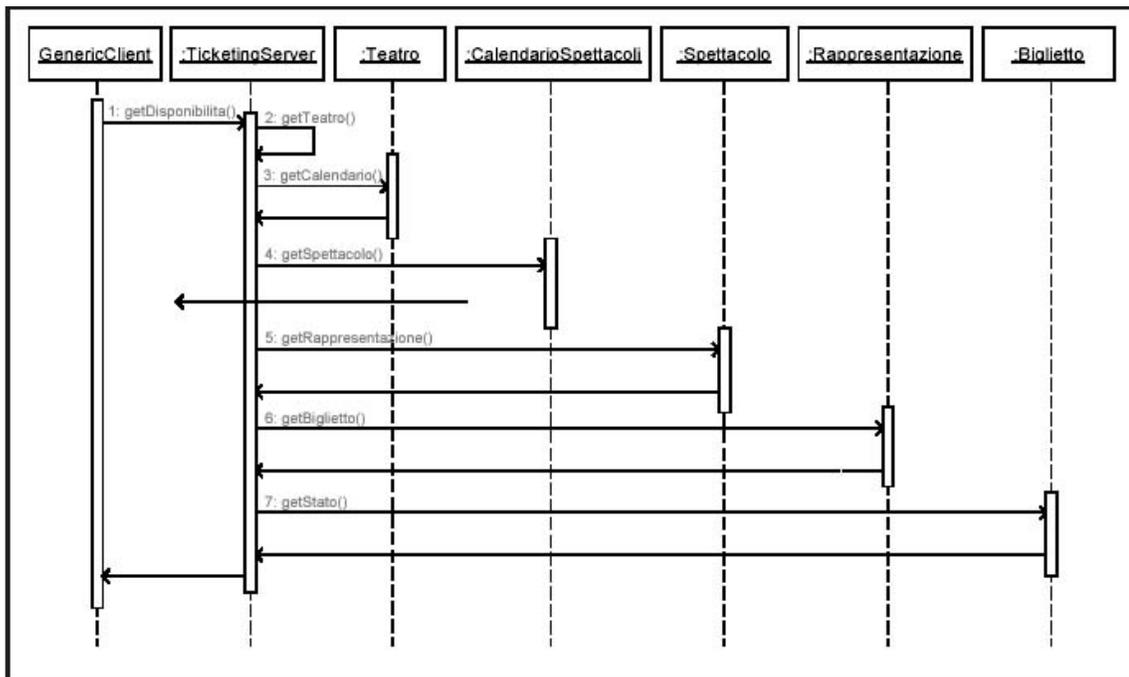


Figura 11: Sequence Diagram a livello del modello di disegno.

Il grande vantaggio offerto, come si riscontra facilmente, è legato alla semplicità di lettura e comprensione; pertanto il diagramma di sequenza si presta ad essere utilizzato per illustrare dei comportamenti da sottoporre all'attenzione del cliente. In tal caso i diagrammi devono possedere un elevato livello di astrazione.

Nei diagrammi di sequenza, nella prima riga vengono riportati gli oggetti che partecipano all'interazione. Da ciascuno di essi parte una linea tratteggiata verticale che rappresenta il trascorrere del tempo, mentre le varie frecce rappresentano lo scambio esplicito dei messaggi.

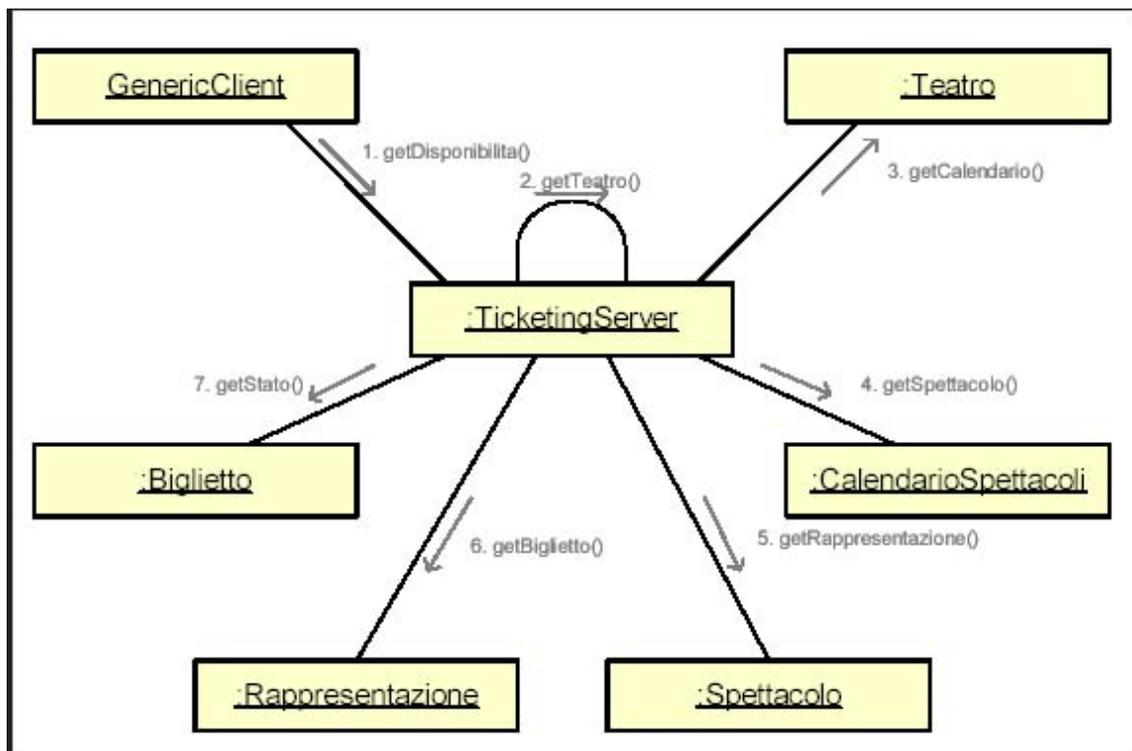


Figura 12: Collaboration Diagram della funzione disponibilità.

Il Collaboration Diagram riportato in figura 12 illustra la stessa funzionalità del precedente Sequence Diagram, solo che in questo caso l'aspetto a cui viene conferita maggiore enfasi è la sequenzialità dei messaggi nel contesto dell'organizzazione strutturale degli oggetti. Per

questa caratteristica, e per la capacità di mostrare senza grandi problemi molti oggetti nell'ambito di uno stesso diagramma senza disordinarlo, i Collaboration Diagram vengono preferiti in fase di disegno.

Component Diagram

I Component Diagram mostrano la struttura fisica del codice in termini di componenti e di reciproche dipendenze. I diagrammi dei componenti illustrano la proiezione statica dell'implementazione del sistema e pertanto, sono strettamente connessi ai diagrammi delle classi. Ciascun componente rappresenta una parte del sistema, sostituibile, che incapsula implementazione ed espone un insieme di interfacce. Un componente è tipicamente costituito da un insieme di elementi (interfacce, classi, ecc.) che risiedono nel componente stesso. Un certo numero di questi ne definisce esplicitamente le interfacce esterne, ossia la definizione dei servizi esposti e quindi forniti dal componente.

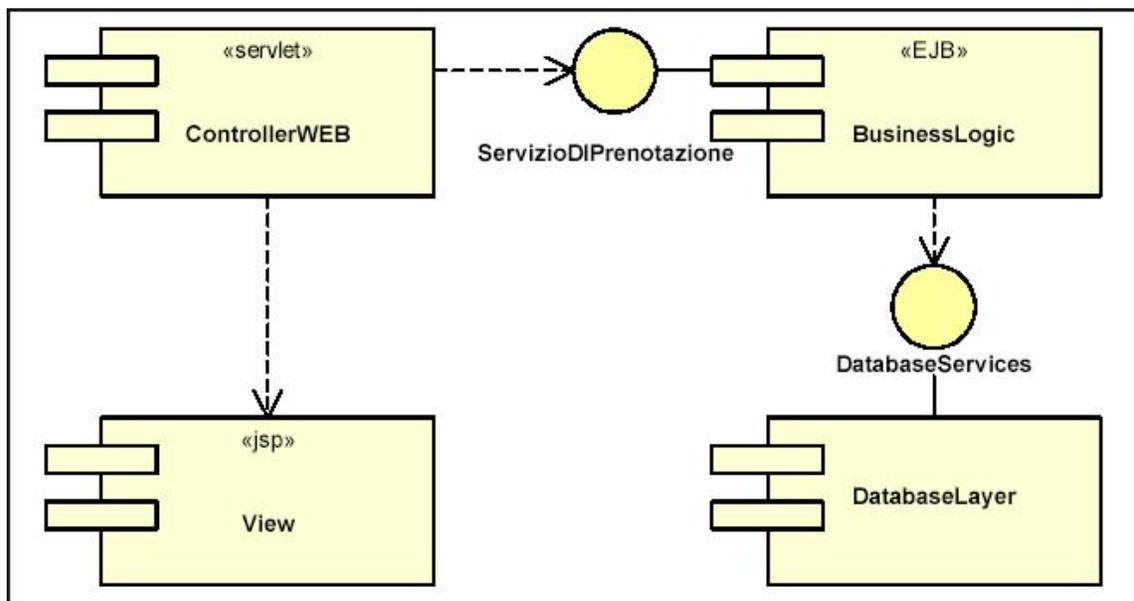


Figura 13: Component Diagram del distributore automatico.

Nella figura 13 è illustrato il diagramma dei componenti (concettuale) del sistema di ticketing del teatro.

Deployment Diagram

I diagrammi di dispiegamento mostrano l'architettura hardware e software del sistema: ne vengono illustrati gli elementi fisici (computer, reti, dispositivi fisici, ...) opportunamente interconnessi e i moduli software allocati su ciascuno di essi. In sintesi viene mostrato il dispiegamento del sistema a tempo di esecuzione sulla relativa architettura fisica, in termini di componenti e relative allocazioni nelle istanze dei nodi.

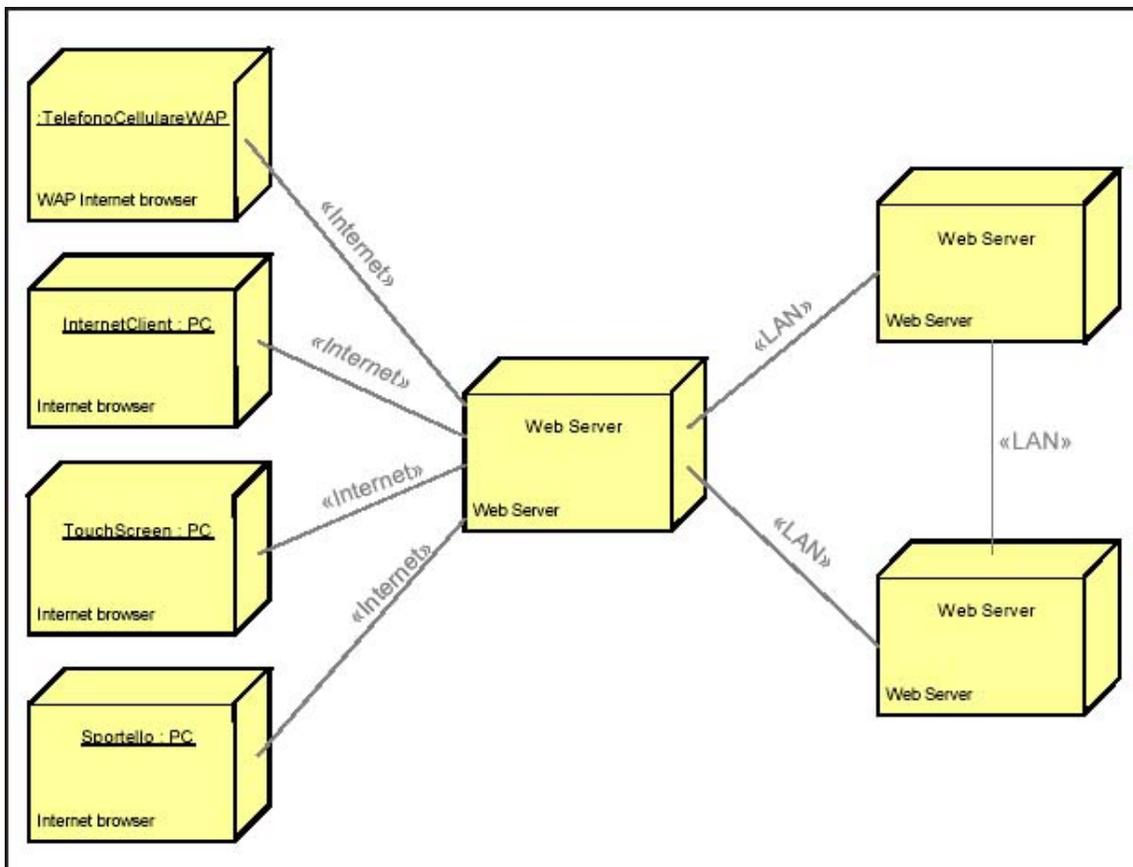


Figura 14: esempio di diagramma di dispiegamento.

Dall'analisi della figura 14 è possibile individuare una serie di nodi, rappresentati attraverso parallelepipedi, collegati tra loro per mezzo di opportune connessioni (associazioni).

I nodi vengono suddivisi in processi (*processor*) e dispositivi (*device*): la differenza risiede nel fatto che i primi dispongono di capacità elaborative (possono eseguire dei componenti), mentre i secondi no e, tipicamente, vengono utilizzati per rappresentare elementi di interfacciamento con il mondo esterno.

7.3 Utilizzo dell'UML nei processi di sviluppo

L'UML non si identifica con i processi di sviluppo, infatti, al contrario dei processi, non fornisce alcuna direttiva su come far evolvere il progetto attraverso le varie fasi.

I processi devono essere sia adattati alle singole organizzazioni in funzione di molti parametri, come per esempio le dimensioni, le diverse figure professionali disponibili e così via, sia dimensionati rispetto alle caratteristiche dei progetti.

Una schematizzazione plausibile dei vari processi può essere quella riportata in figura 15. Come emerge dal diagramma i diversi modelli prevedono due proiezioni: una statica ed una dinamica.

I vari modelli da produrre come risultato delle varie fasi di un processo sono quelli riportati di seguito.

Modello Business

Viene generato come risultato della fase di modellazione della realtà oggetto di studio (business modeling). L'obiettivo è capire cosa bisognerà realizzare (requisiti funzionali e non), quale contesto bisognerà automatizzare (studiarne struttura e dinamiche), comprendere l'organigramma dell'organizzazione del cliente, valutare l'ordine di grandezza del progetto e così via.

Modello dei requisiti

Questo modello viene prodotto a seguito della fase comunemente detta analisi dei requisiti. Scopo di questa fase è produrre una versione più tecnica dei requisiti del cliente evidenziati nella fase precedente.

Modello di analisi

Questo modello viene prodotto come risultato della fase di analisi i cui obiettivi sono di produrre una versione dettagliata e molto tecnica della Use Case View, accogliendo anche le direttive provenienti dalle prime versioni del disegno dell'architettura del sistema.

Modello di disegno

Anche in questo caso il modello di disegno è il prodotto dell'omonima fase, in cui ci si occupa di plasmare il sistema, trasformare i vari requisiti forniti nel modello dei casi d'uso di analisi in un modello direttamente traducibile in codice.

Modello fisico

Il modello fisico, a sua volta, è composto essenzialmente da due modelli: Deployment e Component. Non si tratta quindi del prodotto di una fase ben specifica, bensì sono risultati di rielaborazioni effettuate in diverse fasi. Il modello dei componenti mostra le dipendenze tra i vari componenti software che costituiscono il sistema. Il modello di dispiegamento descrive la distribuzione fisica del sistema in termini del modo in cui le funzionalità sono ripartite sui nodi dell'architettura fisica del sistema.

Modello di test

Nella produzione di sistemi, con particolare riferimento a quelli di medio/grandi dimensioni, è opportuno effettuare test in tutte le fasi. L'obiettivo è quello di eliminare eventuali errori o lacune il più presto possibile onde evitare gli effetti delle relative ripercussioni sul sistema.

Modello implementativo

Questo modello è frutto della fase di implementazione il cui obiettivo è tradurre in codice i manufatti prodotti nella fase di disegno e quindi realizzare il sistema in termini di componenti.

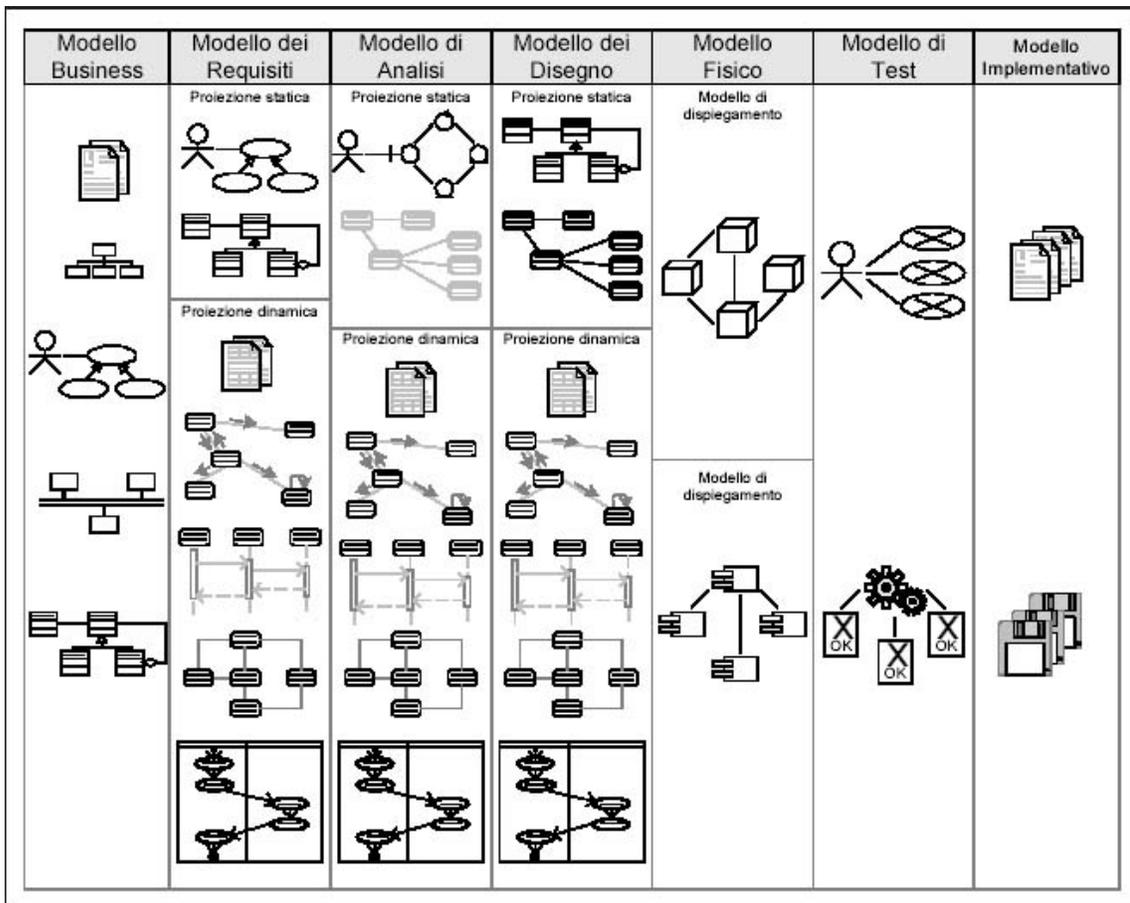


Figura 15: modelli di cui un processo è composto.

7.4 I meccanismi generali e le estensioni

Sebbene di elementi del nucleo dell'UML permettano di formalizzare molti aspetti di un sistema, è impossibile pensare che da soli siano sufficienti per illustrarne tutti i dettagli. Pertanto, al fine di mantenere il linguaggio semplice, flessibile e potente allo stesso tempo, l'UML è stato corredato sia di meccanismi generali utilizzabili per aggiungere informazioni supplementari difficilmente standardizzabili (per esempio le note), sia di veri e propri meccanismi di estensione (come per esempio gli stereotipi).

I meccanismi generali forniti dall'UML sono:

- Notes (note): informazioni supplementari, scritte con un formalismo che può variare dal linguaggio di programmazione, allo pseudocodice al testo in linguaggio naturale e pertanto difficilmente standardizzabili;
- Adornments (ornamenti): elementi grafici che permettono di aggiungere semantica al linguaggio al fine di fornire al lettore una cognizione diretta di aspetti particolarmente importanti di specifici elementi (come per esempio gli indicatori di relazione raffigurati in figura 16);
- Specifications (specificazioni): elementi di testo che aggiungono sintassi e semantica agli elementi dell'UML, per esempio l'elenco degli attributi e metodi presenti appartenenti alle classi;
- Common Divisions (divisioni comuni): nella modellazione di sistemi Object Oriented, ogni elemento reale può essere diviso almeno in due modi diversi: l'astrazione e il relativo corpo; si pensi al rapporto che esiste tra classi e oggetti.

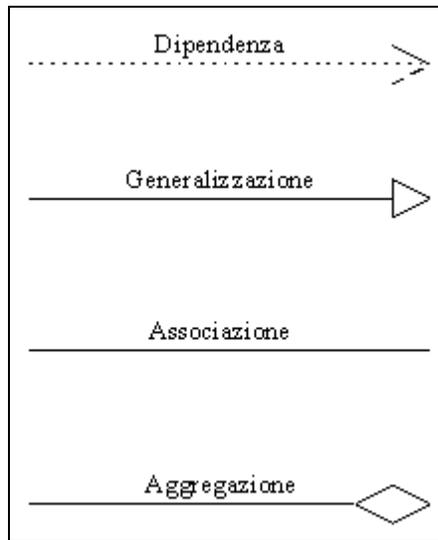


Figura 16: principali relazioni.

I meccanismi di estensione forniti dall'UML sono:

- Stereotypes (stereotipi): permettono di estendere il vocabolario dell'UML aggiungendo nuovi elementi, specifici per il problema oggetto di studio, ottenuti estendendo opportunamente quelli esistenti (un esempio è quello di figura 17 sempre relativo al teatro nel quale si fa riferimento ad un'architettura gerarchica costituita da un'installazione centrale alla quale afferiscono sia gli utenti Internet, sia tutta una serie di biglietterie remote);
- Tagged Value (valori etichettati): estendendo le proprietà degli elementi dell'UML, aggiungendo nuove informazioni costituite dalla coppia nome-valore;
- Constraints (vincoli): rappresentano restrizioni di uno o più valori (o parti) di un modello Object Oriented o di un sistema. Un vincolo può essere rappresentato attraverso qualsiasi formalismo.

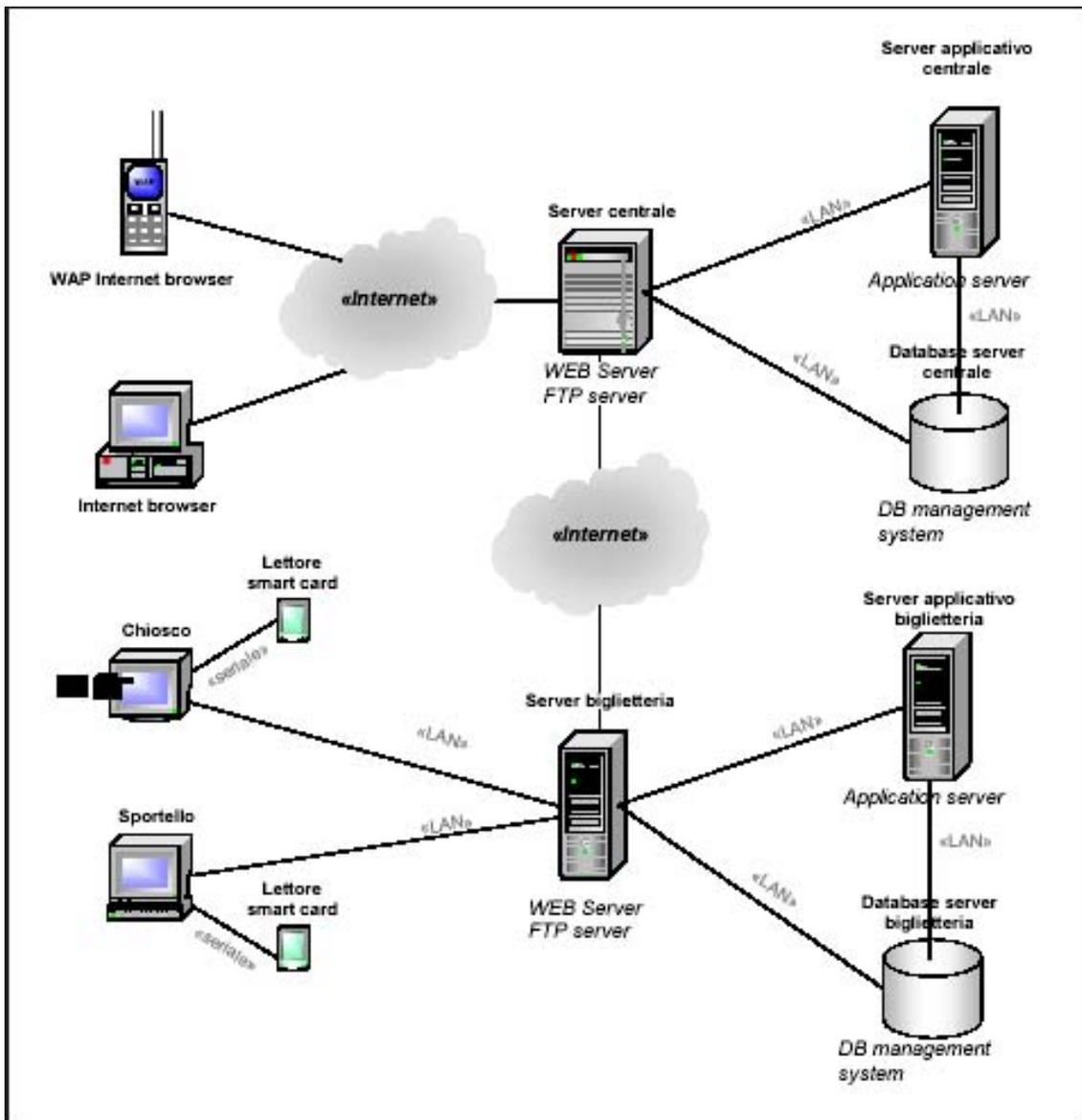


Figura 17: esempio di deployment con l'utilizzo di opportuni stereotipi.

Durante il processo di disegno di sistemi che utilizzano particolari tecnologie piuttosto ricorrenti nei progetti, è necessario avvalersi di una serie di estensioni dell'UML. Ciò al fine di renderlo in grado di rappresentare adeguatamente le caratteristiche peculiari di suddette tecnologie. L'OMG (Object Management Group) ha deciso pertanto standardizzare questi insiemi di regole, detti profili, come veri e propri plug-in dell'UML. Particolarmente utili sono i profili CORBA ed EJB.

Bibliografia

Vir S.p.a. il sito dell'azienda oggetto della simulazione:

<http://www.moredata.it/anima/vir>

Colombatto, E.(2001), *Dall'impresa dei neoclassici all'imprenditore di Kirzner*, in *Economia Politica*, n.2, pp.157-179

Kirzner, I.M. (1997), *Entrepreneurial Discovery and the Competitive Market Process: An Austrian Approach*, *Journal of Economic Literature* Vol. XXXV (March 1997), pp. 60–85

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

Terna P. (2000a), *Economic Experiments with Swarm: a Neural Network Approach to the Self-Development of Consistency in Agents' Behavior*, in F. Luna and B. Stefansson (eds.), *Economic Simulations in Swarm: Agent-Based Modelling and Object Oriented Programming*. Dordrecht and London, Kluwer Academic.

Bianchi C. (2001), *Processi di apprendimento nel governo dello sviluppo della piccola impresa – Una prospettiva basata sull'integrazione tra modelli contabili e di system dynamics attraverso i micromondi*. Milano, Giuffrè

Chen J., Lauschke J. J. (july 2001), *What Makes Knowledge Tacit? On the Evolution of knowledge-based Industrial Clusters*, Department of Finance and Accounting National University of Singapore

Boyd D. E., Spekman R. E. (2001), *Internet Usage Within B2B Relationships and Its Impact on Value Creation: A Conceptual Model and Research Propositions*, Darden Graduate School of Business Administration, Working Paper n. 01-17

Terna P. (2003), *How to use jES - An introduction to an enterprise simulator*, February 2003.

Trento S., Wrglien M., *Nuove tecnologie e cambiamenti organizzativi: alcune implicazioni per le imprese italiane*, Temi di discussione Servizio studi banca d'Italia n.428, dicembre 2001

Hunt J. (2000), *The Unified Process for Practitioners-Object Oriented Design, UML and Java*. London, Springer

Notazione UML

<http://www.uml.org/>

<http://argouml.tigris.org/>

Autori vari, *Manuale pratico di Java*, Hops 2001

<http://www.mokabyte.it/>

UML e ingegneria del software

<http://www.mokabyte.it/>

Garicano L., and kaplan S. N. (Novembre 2000), *The Effects of Business-to-Business E-Commerce on Transaction Costs*, in NBER Working Paper, n. W8017

NIIP Consortium, 1998, NIIP Reference Architecture, (<http://www.niip.org>)

Parisi D. (2001), *Simulazioni – La realtà rifatta nel computer*, Bologna, Il Mulino

Pozzali A., Viale R., *Cognizione e Conoscenza Tacita nei Processi Innovativi*, Università degli Studi di Milano – Bicocca e Fondazione Rosselli

Axtel R., *Why agents? On the varied motivations for agent computing in the social sciences*, Center on Social and Economic Dynamics, Working Paper n. 17

Rullani E., *Il distretto industriale come sistema adattivo complesso*, in *Complessità e distretti industriali: dinamiche, modelli, casi reali*, Fondazione Montedison, Milano 19-20 giugno 2001

Trento S., Wrglien M., *Nuove tecnologie e cambiamenti organizzativi: alcune implicazioni per le imprese italiane*, Temi di discussione Servizio studi banca d'Italia n.428, dicembre 2001

Kimbrough S., Wu D.J., Zhong F. (2001), *Computers Play the Beer Game: Can Artificial Agents Manage the Supply Chain?*, in R.H. Sprague, Jr. (Ed.), *Proceedings of the Thirty-fourth Annual Hawaii International Conference on System Sciences*, Los Alamitos, California, IEEE Computer Society Press

Ganeshan R., Harrison T.P., 1995, *An Introduction to Supply Chain Management*, Penn State University, Working Paper.

Eymann T., Padovan B., Schoder D. (1998), *Artificial Coordination – Simulation Organizational Change with Artificial Life Agents*, Proceedings of the IFAC Symposium on Computation in Economics, Finance, and Engineering: Economic Systems (CEFES '98), Cambridge, June 29-July 1, 1998

Eyman T., Schoder D., Padovan B. (1998), *The Living Value Chain - Coordinating Business Processes with Artificial Life Agents*, Proceedings of the Third International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi – Agents, March 23-25, London

VonKortzfleisch H.F.O., Al-Laham A. *Structurization and Formalization of Knowledge Management in Virtual Organizations: The Case of a Medium-Sized Consulting Company*, Proceedings of the 33rd Annual Hawaii International Conference on System Sciences, 4-7 January 2000, Maui, Hawaii, IEEE Computer Society, 2000

Wu D.J. (2000), *Artificial Agents for Discovering Business Strategies for Network Industries*, in International Journal of Electronic Commerce, Vol. 5, No. 1, pp. 9-36

Wu D.J., Sun Y., Zhong F. (November 2000), *Organizational Agent Systems for Intelligent Enterprise Modeling*, in International Journal Electronic Markets, Vol. 10, No. 4, pp. 272–281

Appendice – Il codice jES-VIR (le nuove classi introdotte nel modello jES)

```
//OrderDistiller.java modified by Pietro Terna (look below at rows
signed //pp)
//OrderDistiller.java modified by Marco Lamieri and Francesco Merlo
//(look below at rows signed //lm)

import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

import swarm.collections.ListImpl;

/**
 * OrderDistiller.java
 *
 *
 * Created: Wed May 08 15:29:12 2002
 *
 * @author<br/>
 * Cristian Barreca, Elena Bonessa and Antonella Borra</br>
 * Modified by:</br>
 * Marco Lamieri and Francesco Merlo
 * @version 0.9.7.31.b
 */

/**
 * This class is used to read data from two worksheets. <br/>The first
one contains the
 * list of recipes of
 * our virtual enterprise.<br/>The second one contains a sequence of
orders to be launched,
 * shift by shift, in order to make the daily production activities.*/

public class OrderDistiller extends OrderGenerator{

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

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

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

```

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

/**Flags to operate checks while reading
 */

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

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

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

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

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

/** the list containig the operating units
 */
public ListImpl unitList;

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

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

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

```

```

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

    // units
    Unit aUnit;
    EndUnit anEndUnit;

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

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

    int currentLayer = 0;

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

    int maxOrderSequence;

    AssigningTool assigningTool; //pt

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

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

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

}

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

```

```

    super.setDictionary();

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

maxOrderSequence = dictionaryLength * totalLayerNumber;

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

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

/**This method is used to set the length of each recipe
*/
public void setRecipeContainers(){

    recipeList = new ListImpl(getZone());

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

    checkTheCell = recipeWorksheet.getStrValue();

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

    worksheetRecipeFileOpen = false;
}

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

    if(! worksheetRecipeFileOpen){

```

```

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

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

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

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

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

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

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

        switch(choice){
            case 1:
                if(StartESFrame.verbose)
                    System.out.println("OrderDistiller: the choice is for
a Procurement");
                procurement(recipeWorksheet);
                break;

```

```

        case 2:
            if(StartESFrame.verbose)
                System.out.println("OrderDistiller: the choice is for
an Or ");
            oR(recipeWorksheet);
            break;
        case 3:
            if(StartESFrame.verbose)
                System.out.println("OrderDistiller: the choice is for
an End ");
            end(recipeWorksheet);
            break;
        case 4:
            if(StartESFrame.verbose)
                System.out.println("OrderDistiller: the choice is for
a Computation ");
            computation(recipeWorksheet);
            break;
        case 5:
            if(StartESFrame.verbose)
                System.out.println("OrderDistiller: the choice is for
a Number");
            number(recipeWorksheet);
            break;
        default:
            if(StartESFrame.verbose)
                System.out.println("OrderDistiller: no matches were found
reading the worksheet, check for errors inside it");
                System.exit(1);
    }
}

if(! recipeWorksheet.eof())
    checkTheCell = recipeWorksheet.getStrValue();

aRecipe = new Recipe(getZone(), code, length, recipeName);
if( StartESFrame.verbose)
    System.out.println("OrderDistiller: a Recipe named "+
recipeName+ " with code " + code + "was born");
    recipeList.addLast(aRecipe);
}

//lm
// *****
// A new method for the computational step
// *****

/**This method dial with the computational choice
*/
public void computation(ExcelReader e){
    int numberOfMatrixesForComputation = 0;

```

```

int computationCode;

// The computational code
e.getIntValue();

numberOfMatrixesForComputation = e.getIntValue();
length += (3 + numberOfMatrixesForComputation);

// Skipping the Matrixes
for(int h = 0; h < numberOfMatrixesForComputation; h++)
checkTheCell = e.getStrValue();
checkTheCell = e.getStrValue();

}

/**This method dial with the procurement choice
*/
public void procurement(ExcelReader e){
int numberOfStepsForProcurement = 0;

numberOfStepsForProcurement = e.getIntValue();
length += (2 + numberOfStepsForProcurement);
for(int h = 0; h < numberOfStepsForProcurement; h++) checkTheCell
= e.getStrValue();
checkTheCell = e.getStrValue();

}

/**This method dial with the or choice
*/
public void or(ExcelReader e){

length +=2;
e.getStrValue();
checkTheCell = e.getStrValue();

}

/**This method dial with the end choice
*/
public void end(ExcelReader e){

length++;
e.getStrValue();
checkTheCell = e.getStrValue();

}

/**This method dial with normal or batch choice
*/
public void number(ExcelReader e){

checkTheCell = e.getStrValue();
if(checkTheCell.equals(sec)) second(e);
else if(checkTheCell.equals(min))minute(e);

```

```

        else{
            if(StartESFrame.verbose)
                System.out.println("Time is not expressed in minutes or
seconds. Check the worksheet");
            System.exit(1);
        }
    }

    public void second(ExcelReader e){
        int numberOfSteps = 0;

        numberOfSteps = e.getIntValue();
        if(numberOfSteps == 0){length ++;
        checkTheCell = e.getStrValue();}
        else{
            checkTheCell = e.getStrValue();

            if(checkTheCell.equals(slash) ||
checkTheCell.equals(backslash)){

                length +=4;
                e.getStrValue();
                checkTheCell = e.getStrValue();

            }

            else {
                length += numberOfSteps;
            }

        }
    }

    public void minute(ExcelReader e){
        int numberOfSteps = 0;

        numberOfSteps = (e.getIntValue() * 60);
        if(numberOfSteps == 0){length ++;
        checkTheCell = e.getStrValue();}
        else{
            checkTheCell = e.getStrValue();

            if(checkTheCell.equals(slash) ||
checkTheCell.equals(backslash)){

                length +=4;
                e.getStrValue();
                checkTheCell = e.getStrValue();

            }

            else {
                length += numberOfSteps;
            }

        }
    }

```

```

    }
}

/**This is the method containing the iterator needed to launch the
daily production of recipes.
 * It take a look at the orderSequence arrays to determine which
recipes must be done and how many times.
 * A request for which unit can do the first production phase of
each recipe will be done to units or endUnits.
 */
public void distill(){
    //local counters
    int i, ii, iii, k, code, quantity, layerNumber;
    boolean recipeCodeNotFound;
    i=0;

    readOrderSequence(); //See below

    while(i < maxOrderSequence && orderSequence1[i] != 0){
        code = getOrderSequence1(i);
        quantity = getOrderSequence2(i);
        layerNumber = getOrderSequence3(i);

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

            orderCount++;
            // creating an order
            anOrder = new Order(getZone(), orderCount,
                Globals.env.getCurrentTime(),
                aRecipe.getLength(),
aRecipe.getOrderRecipe(),
                eSFrameModelSwarm, endUnitList);

            anOrder.setRecipeName(aRecipe.getRecipeName());

            //lm

            //*****
            // setting the layer (from 0 to totalLayerNumber-1)
            //*****

            anOrder.setOrderLayer(layerNumber);
            if(StartESFrame.verbose)
                System.out.println("Order #" + anOrder.getOrderNumber() +

```

```

+         " is generated with Name: " + anOrder.getRecipeName()
+         " and layerNumber #" + anOrder.getOrderLayer());

// add the active orders to the general order list (they
will be // eliminated when dropped in a unit, being finished); this
// list has been introduced for accounting purposes [may be
it would // be better substitute it with a get to the units to know
their // waiting lists]
orderList.addLast(anOrder);

/**
 * sending the order to the first production unit
 * (we are acting as the Front End of the VE)
 */
assigningTool.assign(anOrder);
    }
    i++;
}

}

/**This method reads from the worksheet containing, shift by shift,
the sequence of orders to be launched and fills
 * in the orderSequence1 with the ID codes of recipes and the
orderSequence2 with the quantities of each recipe.
 */

public void readOrderSequence(){
    //local variables

    // lm
    //*****
    // boolean firstTime has been moved to global variables
    //*****
    int i, numberOfShift;

    if(StartESFrame.verbose)
        System.out.println("firstTime is " + firstTime);

    i = 0;

    if(! worksheetOrderSequenceFileOpen && firstTime){
        orderSequenceWorksheet = new
ExcelReader("recipeData/orderStartingSequence.xls");
        worksheetOrderSequenceFileOpen = true;
        firstTime = false;
        if(StartESFrame.verbose)

```

```

        System.out.println("orderStartingSequence.xls has been
open");
    }

    else if(! worksheetOrderSequenceFileOpen){
        orderSequenceWorksheet = new
ExcelReader("recipeData/orderSequence.xls");
        worksheetOrderSequenceFileOpen = true;
        //lm
        if(StartESFrame.verbose)
            System.out.println("orderSequence.xls has been open");
    }

    //lm
    /******
*
// Read the shift number, added some prints for debugging purpose
//*****
*

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

    //lm
    /******
*
// A new bug. Zeroing orderSequenceX vectors
//*****
*

    for( int ii = 0; ii < maxOrderSequence; ii++){
        orderSequence1[ii] = 0;
        orderSequence2[ii] = 0;
        orderSequence3[ii] = 0;
    }

    while(! orderSequenceWorksheet.eol()){

        //lm
        /******
//Our new method used to check if there is a new layer
//*****
        checkForLayer(orderSequenceWorksheet);

        orderSequence1[i] =
errorIsNotAnInteger(orderSequenceWorksheet);

        orderSequence3[i] = currentLayer;

        errorIsNotAString(orderSequenceWorksheet);
        orderSequenceWorksheet.getStrValue();

```

```

        orderSequence2[i] =
errorIsNotAnInteger (orderSequenceWorksheet);

        //lm
        //*****
        // Some prints for debugging purpose
        //*****
        if(StartESFrame.verbose)
            System.out.println("The recipe #" + orderSequence1[i] + "
with quantity " + orderSequence2[i]
            + " and Layer " + orderSequence3[i] +" is starting
production ");

        i++;
    }

    errorIsNotAString (orderSequenceWorksheet);
    orderSequenceWorksheet.getStrValue ();

    if(orderSequenceWorksheet.eof()){
        // System.out.println("The simulation finishes here. It is
not the right and normal way, but is only for explanation. Thank you
for your attention and interest.");
        // System.exit(1);
        worksheetOrderSequenceFileOpen = false;
    }
}

/**This method is used to check the corrispondence with the
dictionary of production phases
*/
public int checkTheExistence(int c){
    int i, check;
    boolean dictionaryVoice = false;

    check = c;

    for(i = 0; i < dictionaryLength; i++){
        if(check == dictionary[i])dictionaryVoice = true;
    }

    if(check > 1000000000){
        System.out.println("The value found is not a valid number
because it is greater than a billion");
        System.exit(1);
    }

    if(dictionaryVoice == false){
        System.out.println("The value found is not a valid number of
production");
        System.exit(1);
    }
}

```

```

    return check;
}

//lm
// *****
// Our new method
// *****

/**This method is used to check the presence of a new layer
*/
public void checkForLayer(ExcelReader e){

    if(e.checkForLabelCell()){
        checkTheCell = e.getStrValue();

        if(checkTheCell.equals(layer))
            currentLayer = e.getIntValue();

        //lm
        // *****
        // Some prints for debugging purpose
        // *****
        if(StartESFrame.verbose)
            System.out.println("currentLayer now is " + currentLayer);

    }

}

/**This method is used to check if there are comments in the cells
*/
public void checkForComments(String cTC){

    checkTheCell = cTC;
    while(checkTheCell.equals(gate)){
        row++;
        if(StartESFrame.verbose)
            System.out.println("The row # " + row + " contains a
comment");

        recipeWorksheet.getStrValue();
        checkTheCell = recipeWorksheet.getStrValue();
        if(!
checkTheCell.equals(semicolon))errorIsNotAString(recipeWorksheet);
        checkTheCell = recipeWorksheet.getStrValue();
    }
}

/**This method is used to check if a type error occurs
*/
public int errorIsNotAnInteger(ExcelReader e){

    if(e.checkForLabelCell())

```

```

        {
            System.out.println("The cell should contain an Integer
Value");
            System.exit(1);
        }
        return e.getIntValue();
    }

    /**This method is used to check if a type error occurs
    */
    public void  errorIsNotAString(ExcelReader e){

        if(e.checkForLabelCell());
        else {
            System.out.println("The cell should contain a String Value");
            System.exit(1);
        }
    }

    /**This method is used to obtain the elements of the orderSequencel
array
    */
    public int getOrderSequencel(int j){

        return orderSequencel[j];

    }

    /**This method is used to obtain the elements of the orderSequence2
array
    */
    public int getOrderSequence2(int j){

        return orderSequence2[j];

    }

    /**This method is used to obtain the elements of the orderSequence3
array
    */
    public int getOrderSequence3(int j){

        return orderSequence3[j];

    }
} // OrderDistiller

```

```

import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

/**
 * Recipe.java
 *
 *
 * Created: Wed May 13 15:29:12 2002
 *
 *
 * @author<br/>
 * Cristian Barreca, Elena Bonessa and Antonella Borra</br>
 * Modified by:</br>
 * Marco Lamieri and Francesco Merlo
 * @version 0.9.7.31.b
 */

/**This class is used to record the recipes and their referring number
(ID), so we can assign them to a List*/
public class Recipe extends SwarmObjectImpl{
    //INSTANCE VARIABLES
    /**the array containing the recipe*/
    int[] orderRecipe;

    /**the referring number of the recipe in the worksheet file*/
    int code;

    /**The length of the array
    */
    int length;

    /**Flags to operate checks while reading
    */
    public String semicolon = ";", checkTheCell, gate = "#", p = "p",
end = "e",
    sec = "s", min = "m", slash = "/", backslash = "\\ ", or = "||",
    computation="c";
    /** The name of the recipe
    */
    String recipeName;

    /** the dictionary to be used to choose the steps to be included in
    * a recipe
    */
    int[] dictionary;

    /** length of the dictionary */
    int dictionaryLength;

    OrderDistiller orderDistiller;

    int j, choice, step;
    //CONSTRUCTOR
    public Recipe(Zone aZone, int c, int l, String rN){

```

```

    super(aZone);

    code = c;
    orderRecipe = new int[1];
    for(int j = 0; j < 1; j++)orderRecipe[j] = 0;
    length = 1;
    recipeName = rN;
}

public int getCodeNumber(){

    return code;

}

public int[] getOrderRecipe(){

    return orderRecipe;

}

public int getLength(){

    return length;

}

public String getRecipeName(){

    return recipeName;

}

public void setSteps(String cTC, ExcelReader recipeWorksheet){

    checkTheCell = cTC;

    checkTheCell = recipeWorksheet.getStrValue();
    j = 0;
    if(StartESFrame.verbose)
        System.out.println(checkTheCell + " The length of the recipe n° "
+ getCodeNumber() + " is " + getLength());

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

        switch(choice){
            case 1:
                if(StartESFrame.verbose)

```

```

        System.out.println("Recipe: the choice is for a
Procurement");
        procurement(recipeWorksheet);
        break;
    case 2:
        if(StartESFrame.verbose)
            System.out.println("Recipe: the choice is for
an Or ");
        oR(recipeWorksheet);
        break;
    case 3:
        if(StartESFrame.verbose)
            System.out.println("Recipe: the choice is for
an End ");
        end(recipeWorksheet);
        break;
    case 4:
        if(StartESFrame.verbose)
            System.out.println("Recipe: the choice is for a
Computation ");
        computation(recipeWorksheet);
        break;
    case 5:
        if(StartESFrame.verbose)
            System.out.println("Recipe: the choice is for a
Number");
        number(recipeWorksheet);
        break;
    default:
        if(StartESFrame.verbose)
            System.out.println("Recipe: no matches were found reading
the worksheet, check for errors inside it");
            System.exit(1);
    }
}

for(int h = 0; h < length; h++)
    if(StartESFrame.verbose)
        System.out.println("The recipe contains " + orderRecipe[h] +
" in position " + h);
    if(! recipeWorksheet.eof())
        checkTheCell = recipeWorksheet.getStrValue();
}

public String getCheckTheCell(){

    return checkTheCell;

}

/**This method is used to check if a type error occur
*/
public int errorIsNotAnInteger(ExcelReader e){

```

```

        if(e.checkForLabelCell())
            {
                if(StartESFrame.verbose)
                    System.out.println("The cell should contain an Integer
Value");
                System.exit(1);
            }
        return e.getIntValue();
    }

/**This method is used to check if a type error occur
*/
public void errorIsNotAString(ExcelReader e){

    if(e.checkForLabelCell());
    {
        if(StartESFrame.verbose)
            System.out.println("The cell should contain a String Value");
            System.exit(1);
    }
}

/**This method deal with the computation choice
*/
public void computation(ExcelReader e){
    int numberMatrixesForComputation = 0;
    int computationCode, step;
    int [] matrixesForComputation;

    // Getting the computation code
    computationCode = e.getIntValue();

    // Getting the number of matrixes used for computation
    numberMatrixesForComputation = e.getIntValue();

    // Creating an array with the name of matrixes
    matrixesForComputation = new int[numberMatrixesForComputation];

    // Getting the matrixes
    for(int m = 0; m < numberMatrixesForComputation; m++)
        matrixesForComputation[m] = e.getIntValue();

    // Getting the production step
    step = e.getIntValue();

    // Getting the time unit
    checkTheCell = e.getStrValue();

    // Expanding the production step for intermediate format
    if(checkTheCell.equals(sec))
        second(e, step);
    else if(checkTheCell.equals(min))
        minute(e, step);
    else{
        if(StartESFrame.verbose)

```

```

        System.out.println("Time is not expressed in minutes or
seconds. Check the worksheet");
        System.exit(1);
    }

    orderRecipe[++j] = -1 * computationCode;

    orderRecipe[++j] = numberMatrixesForComputation;

    for(int h = 0; h < numberMatrixesForComputation; h++)
        orderRecipe[++j] = matrixesForComputation[h];

    orderRecipe[++j] = 1000000000 + step;

    j++;
}

/**This method dial with the procurement choice
*/
public void procurement(ExcelReader e){
    int numberOfStepsForProcurement = 0;

    orderRecipe[j] = -1;
    numberOfStepsForProcurement = e.getIntValue();
    orderRecipe[++j] = numberOfStepsForProcurement;
    for(int h = 0; h < numberOfStepsForProcurement; h++)
        orderRecipe[++j] = e.getIntValue();
    j++;
    checkTheCell = e.getStrValue();
}

/**This method dial with the or choice
*/
public void oR(ExcelReader e){

    orderRecipe[j] = -10;
    orderRecipe[++j] = e.getIntValue();
    j++;
    checkTheCell = e.getStrValue();
}

/**This method dial with the end choice
*/
public void end(ExcelReader e){

    orderRecipe[j] = e.getIntValue();
    checkTheCell = e.getStrValue();
}

/**This method dial with normal or batch choice
*/
public void number(ExcelReader e){

```

```

        step = Integer.parseInt(checkTheCell);
        checkTheCell = e.getStrValue();
        if(checkTheCell.equals(sec)) second(e, step);
        else if(checkTheCell.equals(min))minute(e, step);
        else{
            if(StartESFrame.verbose)
                System.out.println("Time is not expressed in minutes or
seconds. Check the worksheet");
            System.exit(1);
        }
        j++;
    }
}

```

```

public void second(ExcelReader e, int step){
    int numberOfSteps = 0;

    numberOfSteps = e.getIntValue();
    if(numberOfSteps == 0){
        checkTheCell = e.getStrValue();
        orderRecipe[j] = 1000000000 + step;
    }
    else {
        checkTheCell = e.getStrValue();

        if(checkTheCell.equals(slash) ||
checkTheCell.equals(backslash)) {

            if(checkTheCell.equals(slash)){
                orderRecipe[j] = -2;
                orderRecipe[++j] = numberOfSteps;
                orderRecipe[++j] = e.getIntValue();
                orderRecipe[++j] = step;
            }

            else if(checkTheCell.equals(backslash)){
                orderRecipe[j] = -3;
                orderRecipe[++j] = numberOfSteps;
                orderRecipe[++j] = e.getIntValue();
                orderRecipe[++j] = step;
            }
            checkTheCell = e.getStrValue();
        }

        else {
            orderRecipe[j] = step;
            for(int jj = 0; jj < numberOfSteps - 1; jj++)
                orderRecipe[++j] = step;
        }
    }
}

```

```

public void minute(ExcelReader e, int step){
    int numberOfSteps = 0;

```



```

// ProcurementAssembler.java

import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

import swarm.collections.ListImpl;
import swarm.collections.ListIndex;

/**
 * The ProcurementAssembler class instances are unit
 * assembling the elements of a procurement process
 *
 * @author Pietro Terna
 */
public class ProcurementAssembler extends SwarmObjectImpl implements
    PTHistogramPlottable{

    /** the unit we are assembling for */
    Unit myUnit;
    /** the list of the order to be assembled with the procurements */
    public ListImpl waitingList;
    /** the list of end units */
    public ListImpl endUnitList;
    /** its iterator */
    public ListIndex endUnitListIndex;
    /** the vector of the quantities in the end units */
    public int [] quantitiesInEndUnits;
    /** the vector of the labels fo the end units */
    public int [] endUnitLabels;
    /** a procurement specification set */
    ProcurementSpecificationSet pendingProcurementSpecificationSet;

    /**
     * the constructor for ProcurementAssembler
     */
    public ProcurementAssembler (Zone aZone, ListImpl eul)
    {
        // Call the constructor for the parent class.
        super(aZone);
        int i;

        // the end unit list
        endUnitList = eul;
        // its iterator
        endUnitListIndex = endUnitList.listBegin(getZone());

        quantitiesInEndUnits = new int[endUnitList.getCount()];
        endUnitLabels = new int [endUnitList.getCount()];

        if(endUnitList.getCount()>0){
            endUnitListIndex.setOffset(0);
            for (i=0; i<endUnitList.getCount(); i++){
                endUnitLabels[i]= ((EndUnit) endUnitListIndex.get()).

```

```

        getUnitNumber();
        endUnitListIndex.next();
    }
}

// the list procurement processes to be assembled
waitingList = new ListImpl (getZone());

}

/** setting the unit we are assembling for */
public void setUnit(Unit u){
    myUnit = u;
}

/** adding an order to the waitingList */
public boolean setProcurementWaitingList(Order anOrder){
    waitingList.addLast(anOrder);
    return true;
}

/** verifying procurements and eliminating the orders from our
 * waitingList */
public void checkingProcurementsAndFreeingOrders(){
    int n, i, ii, nmax;
    boolean completed;
    Order anOrder;

    nmax = waitingList.getCount();
    if(nmax>0)
        for (n=0;n<nmax;n++)
            {

                anOrder = (Order) waitingList.removeFirst();

                if(StartESFrame.verbose)
                    System.out.println(" ");
                // create the table of the quantities into the end
units
                for (i=0; i<endUnitList.getCount();i++){
                    quantitiesInEndUnits[i]=
                        ((EndUnit) endUnitList.atOffset(i)).
                            getTemporaryListCount(anOrder.getOrderLayer());
                if(StartESFrame.verbose)
                    System.out.print("endUnit " + endUnitLabels[i] + " "
+
                                quantitiesInEndUnits[i] + " ");
                }
                if(StartESFrame.verbose)
                    System.out.println(
                        " as seen in procurementAssembler of unit "
                        + myUnit.getUnitNumber());

                pendingProcurementSpecificationSet =
                    (ProcurementSpecificationSet)
                        anOrder.getPendingProcurementSpecificationSet();
            }
}

```

```

        if(pendingProcurementSpecificationSet == null){
            System.out.println("Internal error in" +
                " procurementAssembler of unit " +
                myUnit.getUnitNumber() +
                " - order # " +
                anOrder.getOrderNumber() +
                "waiting for procurement "+
                "has no procurement specifications.");
            MyExit.exit(1);
        }

    if(StartESFrame.verbose)
    {
        System.out.print("proc. spec. of order " +
            pendingProcurementSpecificationSet.
            getNumber() + " at pos. " +
            pendingProcurementSpecificationSet.
            getPositionInRecipe() + " items ");
        for (i=0;i<pendingProcurementSpecificationSet.
            getNumberOfItemsToBeProcured();i++)
            System.out.print(pendingProcurementSpecificationSet.
                getItemToBeProcured(i) + " ");
        System.out.println(" ");
    }

    // checking endUnits for the required procurements
    for (i=0;i<pendingProcurementSpecificationSet.
        getNumberOfItemsToBeProcured();i++)
        for (ii=0; ii<endUnitList.getCount(); ii++)
            if(endUnitLabels[ii]==
                pendingProcurementSpecificationSet.
                getItemToBeProcured(i))
                quantitiesInEndUnits[ii]--;
    if(StartESFrame.verbose)
        for(ii=0; ii<endUnitList.getCount(); ii++)
            System.out.print("endUnit " + endUnitLabels[ii] + " "
+
                quantitiesInEndUnits[ii] + " ");
    if(StartESFrame.verbose)
        System.out.println(
            " as residual in procurementAssembler of unit "
            + myUnit.getUnitNumber());

    completed=true;
    for (ii=0; ii<endUnitList.getCount(); ii++)
        if(quantitiesInEndUnits[ii]<0) completed=false;

    if(completed)
    {
    if(StartESFrame.verbose)
        System.out.println("completed");
        for (i=0;i<pendingProcurementSpecificationSet.
            getNumberOfItemsToBeProcured();i++)
            for (ii=0; ii<endUnitList.getCount(); ii++)
                if(endUnitLabels[ii]==

```

```

        pendingProcurementSpecificationSet.
            getItemToBeProcured(i))
        // set the procured items into
        // anOrder
        anOrder.setProcuredItemList((Order)
            ((EndUnit) endUnitList.atOffset(ii)).
                removeFromTemporaryList(
                    anOrder.getOrderLayer() ));
    }

    else waitingList.addLast(anOrder);

}

/**
 * return the waiting list length
 */
public int getWaitingListLength()
{
    return waitingList.getCount();
}

/**
 * checking if an order is in the waitingList
 */
public boolean thisOrderIsInTheWaitingList(Order o){

    // the result is true if o is a member of waitingList; false in
    // the opposite case
    return waitingList.contains(o);
}

/** removing an order form the waintingList */
public void removeThisOrderFromTheWaitingList(Order o){
    waitingList.remove(o);
}

/**
 * PTHistogramPlottable interface method: getLabel()
 */
public String getLabel(){
    return "";
}

/**
 * PTHistogramPlottable interface method: getValueToPlot()
 */
public double getValueToPlot(){
    return (double) getWaitingListLength();
}
}

```

```

// ProcurementSpecificationSet.java

import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

/**
 * The container generated by an order and containing the specifications
 about
 * procurement necessities
 *
 * @author Pietro Terna
 */
public class ProcurementSpecificationSet extends SwarmObjectImpl{

    /** number (i.e. that of the order which has generated this object
 */
    public int myOrderNumber;
    /** the position of this sequence in the recipe */
    public int pos;
    /** the specifications */
    public int [] specifications;

    /**
     * constructor for ProcurementSpecificationSet
     */
    public ProcurementSpecificationSet (Zone aZone, int n, int p) {

        // Call the constructor for the parent class
        super(aZone);

        myOrderNumber=n;
        pos=p;

        if(StartESFrame.verbose)
            System.out.println("Proc. spec. created " +
                "(by order " + myOrderNumber + " at pos. " +
                pos + "; NB first pos. is 0)");
    }

    /** setting the procurement specifications */
    public void setSpecificationSet(int jj, int [] r)
    {
        int i;
        specifications = new int[r[jj+1]];

        for (i=0; i<specifications.length; i++)
            specifications[i] = r[jj+2+i];

        if(StartESFrame.verbose)
        {
            System.out.print("item(s) to be procured ");
            for (i=0; i<specifications.length; i++)
                System.out.print(specifications[i] + " ");
        }
    }
}

```

```

        System.out.println(" ");
    }

}

/** returning the number of items to be procured */
public int getNumberOfItemsToBeProcured(){
    return specifications.length;
}

/** returning the item to be procured at pos. i of the vector
 *  containg the items (0<=i<specifications.length) */
public int getItemToBeProcured(int i){
    return specifications[i];
}

/**
 * returning the number of the order of this procurement
specification set
 */
public int getNumber () {
    return myOrderNumber;
}

/**
 * returning the position of this procurement specification set
 * into its recipe
 */
public int getPositionInRecipe() {
    return pos;
}

}

```

// **StandAloneBatchAssembler.java**

```

import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

import swarm.collections.ListImpl;

/**
 * The StandAloneBatchAssembler class instances are unit
 * assembling stand alone batch processes
 *
 * @author Pietro Terna
 */
public class StandAloneBatchAssembler extends SwarmObjectImpl
implements
    PTHistogramPlottable{

    /** the unit we are assembling for */

```

```

Unit myUnit;
/** the list of the order to be assembled with a stand alone batch
    process */
public ListImpl waitingList;

/**
 * the constructor for StandAloneBatchAssembler
 */
public StandAloneBatchAssembler (Zone aZone)
{
    // Call the constructor for the parent class.
    super(aZone);

    // the list of the stand alone batch processes to be assembled
    waitingList = new ListImpl (getZone());
}

/** setting the unit we are assembling for */
public void setUnit(Unit u){
    myUnit = u;
}

/** adding an order to the waitingList */
public boolean setStandAloneBatchWaitingList(Order anOrder){
    waitingList.addLast(anOrder);
    return true;
}

/** verifying stand alone batches and eliminating the orders from
our
 * waitingList in scheduling */
public void checkingStandAloneBatchAndFreeingOrders(){
    // empty method, included for simmetry reasons and for future use
}

/**
 * return the waiting list length
 */
public int getWaitingListLength()
{
    return waitingList.getCount();
}

/**
 * checking if an order is in the waitingList
 */
public boolean thisOrderIsInTheWaitingList(Order o){

    // the result is true if o is a member of waitingList; false in
    // the opposite case
    return waitingList.contains(o);
}

/**
 * removing the order from the waitingList
 */

```

```

public void removeThisOrderFromTheWaitingList(Order o){
    waitingList.remove(o);
}

/**
 * PTHistogramPlottable interface method: getLabel()
 */
public String getLabel(){
    return "";
}

/**
 * PTHistogramPlottable interface method: getValueToPlot()
 */
public double getValueToPlot(){
    return (double) getWaitingListLength();
}
}

// StandAloneBatchSpecificationSet.java

import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

/**
 * The container generated by an order and containing the specifications
 * about
 * a stand alone batch process
 *
 * @author Pietro Terna
 */
public class StandAloneBatchSpecificationSet extends SwarmObjectImpl{

    /** number (i.e. that of the order which has generated this object
 */
    public int myOrderNumber;
    /** the position of this sequence in the recipe */
    public int pos;
    /** the specifications */
    public int productionTimeInTicks;

    /**
     * constructor for StandAloneBatchSpecificationSet
     */
    public StandAloneBatchSpecificationSet (Zone aZone, int n, int p) {

        // Call the constructor for the parent class
        super(aZone);
    }
}

```

```

myOrderNumber=n;
pos=p;

    if(StartESFrame.verbose)
        System.out.println("Stand alone b. spec. created " +
            "(by order " + myOrderNumber + " at pos. " +
            pos + "; NB first pos. is 0)");
}

/** setting the stand alone batch specifications */
public void setSpecifications(int k)
{
    productionTimeInTicks = k;

    if(StartESFrame.verbose)
        System.out.println("Stand alone batch production time in ticks "
+
            productionTimeInTicks);
}

/** decrease production time */
public int decreaseProductionTime(){

    if(StartESFrame.verbose)
        System.out.println(
from"+
            "Residual production time in stand alone production set
            " order " + myOrderNumber+" at pos " + pos +": " +
            productionTimeInTicks);
    productionTimeInTicks--;
    return productionTimeInTicks;
}

/**
 * returning the number of the order of this procurement
specification
 */
public int getNumber () {
    return myOrderNumber;
}

/**
 * returning the position of this procurement specification into
its recipe
 */
public int getPositionInRecipe() {
    return pos;
}
}

```

```

// SequentialBatchAssembler.java

import java.util.Arrays;

import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

import swarm.collections.ListImpl;
import swarm.collections.ListIndex;

/**
 * The SequentialBatchAssembler class instances are unit
 * assembling sequential batch processes
 *
 * @author Pietro Terna
 */
public class SequentialBatchAssembler extends SwarmObjectImpl
implements
    PTHistogramPlottable{

    /** the unit we are assembling for */
    Unit myUnit;
    /** the list of the order to be assembled with a sequential batch
    process */
    public ListImpl waitingList;
    /** its iterators */
    public ListIndex waitingListIndex_i;
    public ListIndex waitingListIndex_ii;
    /** in sequential batch two orders can stay in the same batch
    * if several criteria match; one of them is that of having the
    * same state vector, which contains the number of the units
    * that made the various steps; if we consider more simply the
state
    * as 0 or non 0, we disregard the unit number (also the 'or',
case
    * with -1 values in not used branches, receives the same
treatment) */
    public boolean compareDisregardingUnits;

    /**
    * the constructor for SequentialBatchAssembler
    */
    public SequentialBatchAssembler (Zone aZone, boolean c)
    {
        // Call the constructor for the parent class.
        super(aZone);

        compareDisregardingUnits=c;

        // the list of the sequential batch processes to be assembled
        waitingList = new ListImpl (getZone());
        waitingListIndex_i = waitingList.listBegin(getZone());
        waitingListIndex_ii = waitingList.listBegin(getZone());
    }
}

```

```

/** setting the unit we are assembling for */
public void setUnit(Unit u){
    myUnit = u;
}

/** adding an order to the waitingList */
public boolean setSequentialBatchWaitingList(Order anOrder){
    waitingList.addLast(anOrder);
    return true;
}

our /** verifying sequential batches and eliminating the orders from
    * waitingList in scheduling */
public void checkingSequentialBatchAndFreeingOrders(){
    // empty method, included for simmetry reasons and for future use
}

/**
 * return the waiting list length
 */
public int getWaitingListLength()
{
    return waitingList.getCount();
}

/**
 * checking if an order is in the waitingList
 */
public boolean thisOrderIsInTheWaitingList(Order o){

    // the result is true if 'o' is a member of waitingList; false in
    // the opposite case
    if(StartESFrame.verbose)
        if( waitingList.contains(o))
            System.out.println("SequentialBatchAssembler:time"
                +Globals.env.getCurrentTime()+
                " order "+o.getOrderNumber()+
                " trapped from unit "+
                myUnit.getUnitNumber());

    return waitingList.contains(o);
}

/**
 * removing the order from the waitingList
 */
public void removeThisOrderFromTheWaitingList(Order o){

    waitingList.remove(o);
}

/**
 * if the whole batch exists, we can clear it to the production
 */

```

```

public void clearToProduce()
{
    int i, ii, count, countNeeded, countToBeFound, listCount;
    boolean batchNotFound;
    Order orderToBeChecked;

    if(waitingList.getCount(>0){
        waitingListIndex_i.setOffset(0);
        for (i=0;i<waitingList.getCount();i++){
            ((Order)
waitingListIndex_i.get()).setInSequentialBatch(false);
            waitingListIndex_i.next();
        }
    }

    if(waitingList.getCount(<2) return; //nothing to do
    batchNotFound=true; countToBeFound=0;

    waitingListIndex_i.setOffset(0);
    for (i=0;i<waitingList.getCount()-1 && batchNotFound;i++)
    {
        count=0;
        waitingListIndex_ii.setOffset(i+1);
        for (ii=i+1;ii<waitingList.getCount();ii++)
        {
            if(sameOrderInSequentialBatch(
                (Order) waitingListIndex_i.get(),
                (Order) waitingListIndex_ii.get()
            ))count++;
            waitingListIndex_ii.next();
        }

        // count is the number of order equals to the first one
        countNeeded=((SequentialBatchSpecificationSet)
            ((Order) waitingListIndex_i.get()
                .getPendingSequentialBatchSpecificationSet())
                .getQuantityInSequentialBatch()-1;

        if(count>=countNeeded)
        {
            batchNotFound=false;
            countToBeFound=countNeeded;
            ii=i+1;
            waitingListIndex_ii.setOffset(ii);
        }

        if(count>=countNeeded)
            while(countToBeFound>0){
                if(sameOrderInSequentialBatch(
                    (Order) waitingListIndex_i.get(),
                    (Order) waitingListIndex_ii.get()
                ))
                {
                    ((Order)
                        waitingListIndex_ii.get()).

```

```

        setInSequentialBatch(true);
        countToBeFound--;
    }
    waitingListIndex_ii.next();
}
if(! batchNotFound) ((Order)
    waitingListIndex_i.get()).
    setInSequentialBatch(true);

    waitingListIndex_i.next();
}

// eliminating from the local list the order found to complete a
batch
// eliminating them also from the unit waitingList and setting
them
// together in the list at the current position
listCount=waitingList.getCount();
for (i=0;i<listCount;i++){
    orderToBeChecked=(Order) waitingList.removeFirst();
    // locally
    if(! orderToBeChecked.getInSequentialBatch())
        waitingList.addLast(orderToBeChecked);
    // in my unit
    if(orderToBeChecked.getInSequentialBatch()){
        myUnit.removeFromWaitingList(orderToBeChecked);
        myUnit.setWaitingList(orderToBeChecked);
    }

}

}

/**
 * if the whole batch is produced, we can clear it to the
propagation
 */
public void clearToPropagate()
{
}

/**
 * checking if two orders are in the same batch
 */

public boolean sameOrderInSequentialBatch(Order o1, Order o2)
{
    boolean equalDisregardingUnits; int i;

    equalDisregardingUnits=true;
    if(compareDisregardingUnits)
        if(o1.getStateVector().length==o2.getStateVector().length)
            for (i=0;i<o1.getStateVector().length;i++)
                if(!( o1.getStateVector()[i]==0 &&
                    o2.getStateVector()[i]==0) ||
                    (o1.getStateVector()[i]!=0 &&

```

```

        o2.getStateVector()[i]!=0) ))
        equalDisregardingUnits=false;

    if(
        o1.getRecipeName()==o2.getRecipeName() &&
        (
            o1.getOrderLayer()==o2.getOrderLayer() ||
            myUnit.getSensitiveToLayers()==false
        )
        Arrays.equals(o1.getRecipeVector(), o2.getRecipeVector()) &&
        (
            Arrays.equals(o1.getStateVector(), o2.getStateVector()) ||
            (equalDisregardingUnits && compareDisregardingUnits)
        ) &&
        ((SequentialBatchSpecificationSet)
            o1.getPendingSequentialBatchSpecificationSet()).
            getProductionTimeInTicks() ==
        ((SequentialBatchSpecificationSet)
            o2.getPendingSequentialBatchSpecificationSet()).
            getProductionTimeInTicks() &&
        ((SequentialBatchSpecificationSet)
            o1.getPendingSequentialBatchSpecificationSet()).
            getQuantityInSequentialBatch()
    ==
        ((SequentialBatchSpecificationSet)
            o2.getPendingSequentialBatchSpecificationSet()).
            getQuantityInSequentialBatch()
    &&
        (! o1.getInSequentialBatch()) && (! o2.getInSequentialBatch())
    )
        return true;
    else
        return false;
}

/**
 * PTHistogramPlottable interface method: getLabel()
 */
public String getLabel(){
    return "";
}

/**
 * PTHistogramPlottable interface method: getValueToPlot()
 */
public double getValueToPlot(){
    return (double) getWaitingListLength();
}
}

```

```

// SequentialBatchSpecificationSet.java

import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

/**
 * The container generated by an order and containing the specifications
 * about
 * a sequential batch process
 *
 * @author Pietro Terna
 */
public class SequentialBatchSpecificationSet extends SwarmObjectImpl{

    /** number (i.e. that of the order which has generated this object
    */
    public int myOrderNumber;
    /** the position of this sequence in the recipe */
    public int pos;
    /** the specifications */
    public int productionTimeInTicks;
    public int quantityInSequentialBatch;

    /**
     * constructor for SequentialBatchSpecificationSet
     */
    public SequentialBatchSpecificationSet (Zone aZone, int n, int p) {

        // Call the constructor for the parent class
        super(aZone);

        myOrderNumber=n;
        pos=p;

        if(StartESFrame.verbose)
            System.out.println("Sequential b. spec. created " +
                               "(by order " + myOrderNumber + " at pos. " +
                               pos + "; NB first pos. is 0)");
    }

    /** setting the sequential batch specifications */
    public void setSpecifications(int k1, int k2)
    {
        productionTimeInTicks = k1;
        quantityInSequentialBatch = k2;

        if(StartESFrame.verbose)
        {
            System.out.println(
                "Sequential batch production time in ticks " +
                productionTimeInTicks);
            System.out.println(
                "Quantity in sequential batch production " +

```

```

        quantityInSequentialBatch + " orderNumber
"+myOrderNumber);
    }
}

/** ???????? decrease production time */
public int decreaseProductionTime(){

    if(StartESFrame.verbose)
        System.out.println(
            "Residual production time in stand alone production set
from"+
            " order " + myOrderNumber+" at pos " + pos +": " +
            productionTimeInTicks);
    productionTimeInTicks--;
    return productionTimeInTicks;
}

/**
 * returning the number of the order of this procurement
specification
 */
public int getNumber () {
    return myOrderNumber;
}

/**
 * returning the position of this procurement specification into
its recipe
 */
public int getPositionInRecipe() {
    return pos;
}

/** returning production time in ticks (original or residual) */
public int getProductionTimeInTicks(){
    return productionTimeInTicks;
}

/** setting production time in ticks (residual) */
public void setProductionTimeInTicks(int t){
    productionTimeInTicks=t;
}

/** returning quantity in sequential batch */
public int getQuantityInSequentialBatch(){
    return quantityInSequentialBatch;
}
}

```

```

// Or.java

import swarm.Globals;
import swarm.collections.ListImpl;

/**
 * An Or manager for our orders<br><br>
 *
 * orCriterion<br>
 *      0 - all 'or' branches in sequence<br>
 *      1 - choosing first branch<br>
 *      2 - choosing second branch<br>
 *      3 - a random branch<br>
 *      4 - the branch whose first step
 *          has the shortest waiting list<br>
 *      5 - the branch whose number is stored in (x,0)
pos in
 *          memoryMatrix designated by orMemoryMatrix
in
 *          ESFrameModelSwarm.java; the matrix
 *          may be sensitive or insensitive to layers;
 *          range of the branch number: from 1 to
 *          branchNumber <br>
 *          x is 0 if the first node in 'or' sequence
is
 *          numbered 1; is kk if the first node is
numbered
 *          10kk with kk 00 to 99<br><br>
 * @author Pietro Terna
 */
public class Or
{
    // the class is used in a static way, so we have no constructors
    // here and we have to declare static all the methods

    /** or criterion to be applied */
    public static int orCriterion;
    /** the list of the operating units */
    public static ListImpl unitList;
    /** memory matrix addresses to be used in orCriterion 5 */
    public static MemoryMatrix mm;

    /** setting the 'or' criterion */
    public static void setOrCriterion(int k){
        orCriterion=k;
        if(orCriterion<0 || orCriterion > 6){
            System.out.println("Invalid OrCriterion value.");
            MyExit.exit(1);
        }
    }

    /** setting the list of the units */
    public static void setUnitList(ListImpl ul){

```

```

    unitList=ul;
}

/** setting the address of memory matrix */
public static void setMemoryMatrix(MemoryMatrix m){
    mm=m;
}

/** applying 'or' */
public static boolean applyOr(Order o){
    OrSpecificationSet spec;
    Unit aUnit;
    boolean unitNotFound;
    int i, iii, node, chosenNode, length, length0, tmp;
    int [] recipe, state;

    // the or specification set to be used
    spec=(OrSpecificationSet) o.getPendingOrSpecificationSet();
    if(spec == null){
        System.out.println("Internal error: found an order x in " +
            "Or.applyOr(x) without 'or' specifications");
        MyExit.exit(1);
    }

    // applying the 'or' criterion
    if(orCriterion == 0) return true;

    chosenNode=0;
    recipe = (int []) o.getRecipeVector();
    state = (int []) o.getStateVector();

    if(orCriterion == 1) chosenNode=1;
    if(orCriterion == 2) chosenNode=2;
    if(orCriterion == 3) chosenNode=Globals.env.uniformIntRand.

getIntegerWithMin$withMax(1, spec.getNodeNumber());

    if(orCriterion == 4)
    {
        length=1000000000;
        // looking for the first unit in each branch
        for (node=1; node<=spec.getNodeNumber();node++)
        {
            length0=length;
            unitNotFound=true;
            for (i=0;i<unitList.getCount() && unitNotFound;i++)
            {
                aUnit=(Unit) unitList.atOffset(i);
                for (iii=0;iii<aUnit.getNOOfPhasesToDealWith()
                    && unitNotFound;iii++)
                {
                    if(recipe[spec.getNodePosition(node)]==
                        aUnit.getProductionPhase(iii))
                    {
                        unitNotFound=false;
                    }
                }
            }
        }
    }
}

```

```

length=aUnit.
    getWaitingListLength();
    if (length<length0) chosenNode=
        node;
    }
}
}
}
}

if(orCriterion == 5)
{
    if(mm.getEmpty(
        o.getOrderLayer(),
        ((OrSpecificationSet) o.getPendingOrSpecificationSet()).
            getOrRow(),
        0))
        return false;

    tmp=(int) mm.getValue(
        o.getOrderLayer(),
        ((OrSpecificationSet) o.getPendingOrSpecificationSet()).
            getOrRow(),
        0);

    if(tmp < 1 || tmp > spec.getNodeNumber()){
        System.out.println("Invalid branch number in Or
process"+
                                " in order # " + o.getOrderNumber());
        MyExit.exit(1);
    }

    chosenNode=tmp;
}

if(StartESFrame.verbose)
    System.out.println("Applying Or to order number "+
        o.getOrderNumber()+
        " at pos. "+spec.getPositionInRecipe());

for (node=1; node<=spec.getNodeNumber();node++)
{
    for(i=spec.getNodePosition(node);
        i<spec.getNodePosition(node+1);i++)
    {
        if(chosenNode != node) state[i]=-1;
    }
}

return true;
}
}
}

```

```

// OrSpecificationSet.java

import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

/**
 * The container generated by an order and containing the specifications
 about
 * an or step in a process
 *
 * @author Pietro Terna
 */
public class OrSpecificationSet extends SwarmObjectImpl{

    /** number (i.e. that of the order which has generated this object
 */
    public int myOrderNumber;
    /** the position of this sequence in the recipe */
    public int pos;
    /** the specification */
    public int nodeNumber;
    /** the vector of node positions in 'or' sequence (max 100) */
    public int[] nodePosition;
    /** the row used in the memory matrix invoked by an 'or' process
 using
    * orCriterion 5 (look at Or.java) */
    public int orRow;

    /**
    * constructor for OrSpecificationSet
    */
    public OrSpecificationSet (Zone aZone, int n, int p) {

        // Call the constructor for the parent class
        super(aZone);

        nodePosition = new int[100];
        myOrderNumber=n;
        pos=p;

        if(StartESFrame.verbose)
            System.out.println("Or spec. created " +
                "(by order " + myOrderNumber + " at pos. " +
                pos + "; NB first pos. is 0)");
    }

    /** setting the number of nodes */
    public void setNodeNumber(int k)
    {
        int i;
        nodeNumber = k;
    }
}

```

```

        if(StartESFrame.verbose)
        {
            System.out.print("In orSpecificationSet of order # "+
                myOrderNumber+
                ", number of # "+nodeNumber+
                "; node positions:");
            for (i=1;i<=nodeNumber+1;i++)
                System.out.print(" "+getNodePosition(i));
            System.out.println(" ");
        }
    }

    /** setting the node position */
    public void setNodePosition(int node, int position){

        if(node>100){
            System.out.println("More than 100 nodes in an 'or' sequence
in "+
                "orden # "+myOrderNumber);
            MyExit.exit(1);
        }
        nodePosition[node-1]=position;
    }

    /** setting the orRow */
    public void setOrRow(int rr){
        orRow=rr;
    }

    /** getting the orRow */
    public int getOrRow(){
        return orRow;
    }

    /** getting the node position; position of node # 1 is in
    * nodePosition[0] etc.
    */
    public int getNodePosition(int node){
        return nodePosition[node-1];
    }

    /**
    * returning the number of the order of this procurement
specification
    */
    public int getNumber () {
        return myOrderNumber;
    }

    /**
    * returning the position of this procurement specification into
its recipe
    */

```

```

    public int getPositionInRecipe() {
        return pos;
    }

    /**
     * returning the number of nodes (branches) in this 'or' sequence
     */
    public int getNodeNumber() {
        return nodeNumber;
    }
}

}

// EndUnit.java

import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

import swarm.collections.ListImpl;
import swarm.collections.ArrayImpl;

/**
 * The EndUnit class instances represent virtual or actual
 * places or warehouse were temporary finished sub-recipes (produced
 * internally
 * or obtained as procurements) wait to be used in other recipes via a
 * "p"
 * (procurement) step<br><br>
 *
 * an endUnit has its internal number equal to the concluding code
 * used to finish/identify a temporary finished sub-recipe
 *
 * @author Pietro Terna
 */
public class EndUnit extends SwarmObjectImpl implements
PTHistogramPlottable
{

    /** the number identifying the unit which is also its end code */
    public int unitNumber;
    /** the number of layers in the simulation */
    public int totalLayerNumber;
    /** declaring unit sensitive to layers */
    public boolean sensitive=true;
    /** the array of temporary lists, if layers are > 1 and
sensitive==true */
    public ArrayImpl temporaryListArray;
    /** the list containing order temporary kept in this virtual (or
actual)
 * place */
    public ListImpl temporaryList;
    /** the modelSwarm (for future uses) */

```

```

ESFrameModelSwarm eSFrameModelSwarm;

/**
 * the constructor for EndUnit
 */
public EndUnit (Zone aZone, ESFrameModelSwarm mo, int tln)
{
    // Call the constructor for the parent class.
    super(aZone);

    int i;

    // the model (for future use)
    eSFrameModelSwarm = mo;

    // the number of layers
    totalLayerNumber=tln;

    // creating the array of internal lists, with at least one
element
    temporaryListArray = new ArrayImpl(getZone(), totalLayerNumber);

    // creating the internal lists to keep the items temporary in
hold
    // (one list for each layer; if our unit will be declared to be
    // insensitive to the layers, then we will have unused lists)
    for (i=0;i<totalLayerNumber;i++){
        temporaryList = new ListImpl (getZone());
        temporaryListArray.atOffset$put(i,temporaryList);
    }
}

/** set unit number */
public void setUnitNumber(int un)
{
    // the unit id number.
    unitNumber=un;
    // announce the unit to the console
    if(StartESFrame.verbose)
        System.out.println("End unit number " + unitNumber +
            " has been created.");
}

/** get unit number */
public int getUnitNumber()
{
    // the unit id number.
    return unitNumber;
}

/**
 * Putting an order in the temporaryList of the unit, in a
 * given layer
 */

```

```

public void setTemporaryList(Order anOrder)
{
    int layer;

    layer=anOrder.getOrderLayer();
    if(! sensitive)layer=0;

    temporaryList=(ListImpl)temporaryListArray.atOffset(layer);
    temporaryList.addLast(anOrder);
    if(StartESFrame.verbose)
        System.out.println("The end unit " + unitNumber +
            " has received the order # "
            + anOrder.getOrderNumber());
}

/**
 * Removing an order from the temporaryList of the unit
 * and returning its address of the order or of its clone if
 * the original order has multiplicity > 1
 */
public Object removeFromTemporaryList(int cl)
{
    int chosenLayer;
    Order removedOrder, orderClone;

    chosenLayer=cl;
    if(! sensitive)chosenLayer=0;

    temporaryList=(ListImpl)
temporaryListArray.atOffset(chosenLayer);

    removedOrder = (Order) temporaryList.removeFirst();
    if(StartESFrame.verbose)
        System.out.println("The end unit " + unitNumber +
            " has removed the order # "
            + removedOrder.getOrderNumber());

    if(removedOrder.getMultiplicity() == 1) return removedOrder;

    else
    {
        if(StartESFrame.verbose)
            System.out.println("A clone of the order # " +
                removedOrder.getOrderNumber() +
                " has been created; the original order has now " +
                "multiplicity " + (removedOrder.getMultiplicity()-1));
        orderClone = (Order) removedOrder.clone();

        removedOrder.setMultiplicity(removedOrder.getMultiplicity()-1);
        temporaryList.addFirst(removedOrder);
        return orderClone;
    }
}

```

```

    /**
     * return the count of the orders in all the lists, with
multiplicity
     */
    public int getAllTemporaryListCount()
    {
        int i, ii, c;

        c = 0;
        for(i=0;i<totalLayerNumber;i++){
            temporaryList=(ListImpl) temporaryListArray.atOffset(i);
            for (ii=0; ii< temporaryList.getCount();ii++)
                c += ((Order)
temporaryList.atOffset(ii)).getMultiplicity();
        }

        return c;
    }

    /**
     * return the count of the orders in a specific list, with
multiplicity
     */
    public int getTemporaryListCount(int cl)
    {
        int i, c, chosenLayer;

        chosenLayer=cl;
        if(! sensitive)chosenLayer=0;
        c = 0;

        temporaryList=(ListImpl)
temporaryListArray.atOffset(chosenLayer);
        for (i=0; i< temporaryList.getCount();i++)
            c += ((Order) temporaryList.atOffset(i)).getMultiplicity();

        return c;
    }

    /** declaring the end unit insensitive to layers (its number is
     * reported as negative in unitData/endUnitList.txt)
     */
    public void setNotSensitiveToLayers(){
        sensitive=false;
        totalLayerNumber=1; // number of layers to be used internally
    }

    /**
     * PTHistogramPlottable interface method: getLabel()
     */
    public String getLabel(){
        return "" + getUnitNumber();
    }
}

```

```
/**
 * PTHistogramPlottable interface method: getValueToPlot()
 */
public double getValueToPlot(){
    return (double) getAllTemporaryListCount();
}
}
```