JVEFrame: Analisi dell'idea di partenza

Contenuti

La descrizione della versione 0.5.0 del modello jVEFrame permette di osservare l'idea di partenza, per poter cogliere gli sviluppi che seguiranno, in maniera più analitica di quanto si avrebbe con una descrizione generale. Le classi che compongono la simulazione sono:

1. StartVEFrame.java;
2. VEFrameObserverSwarm.java;
3. VEFrameModelSwarm.java;
4. Ordergenerator.java;
5. Order.java;
6. Unit.java;
7. InformationRuleMaster.java;
8. Warehouse.java;
9. InformationFlowMatrix.java;
10. News.java;
11. AccountingParameters.java;
12. SwarmUtils.java;
13. PTHistogram.java;
14. jveframe.scm.

StartVEFrame.java

Il listato comincia con la riga

import swarm.Globals;

la quale richiama (tramite il comando *import*), all'interno della classe che stiamo esaminando, un'altra classe, la classe *Globals* del package *swarm* appunto, per ereditarne le caratteristiche (si veda il paragrafo x.x.x per questa caratteristica di Java) e per rendere più rapida la stesura del codice infatti, non sarà necessario richiamare sia il package sia la classe.

La funzione fondamentale della classe *swarm.Globals* è di mettere a disposizione del client un'ampia serie di funzioni già predeterminate, sfruttate in genere con un'elevata frequenza nei programmi che utilizzano Swarm.

La struttura del lavoro segue quella usualmente adottata per le simulazioni che adottano questa libreria di funzioni: il metodo *main()* è contenuto nel programma di *Start…*esso è l'unico metodo che viene cercato dall'interprete per avviare le istruzioni contenute nel programma. Il *main()* avvia la simulazione richiamando l'*Observer* il quale a sua volta richiamerà il *Model* e tutto sarà pronto per osservare l'impresa virtuale in funzione.

La classe *StartVEFrame* contiene al suo interno un *"costruttore"* identificato con un nome identico a quello del programma, ciò avviene tramite la riga di comando

public class StartVEFrame,

la cui funzione è quella di inizializzare delle variabili cosiddette di istanza appartenenti cioè a tutta la classe e condivise da tutti metodi della stessa

public static VEFrameObserverSwarm vEFrameObserverSwarm;

public static boolean verbose.

Come detto la definizione del metodo *main()*,dà avvio alla simulazione, richiamando le "azioni" che il programma dovrà compiere

public static void main (String[] args)

Un esempio dell'utilità di aver importato la classe *swarm.Globals* si ha con la seguete riga di codice, la quale rende noto: il nome del programma, la versione, l'autore ed eventuali argomenti che si decida di passare a questo metodo in fase di compilazione

Globals.env.initSwarm ("jveframe", "v.0.4.6", "pietro.terna@unito.it", args).

Si procede quindi alla creazione di un'istanza dell'oggetto *VEFrameObserverSwarm* non tramite il metodo classico che utilizza il comando *new*, ma richiamando il file *jveframe.scm* che contiene i valori di default della simulazione e che possono essere modificati senza dover ricompilare l'intera classe ogni volta, favorendo perciò l'utilizzodella struttura del file Jar

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

si determina la posizione precisa del pannello di controllo, che sarà la medesima ad ogni avvio (altro esempio dell'utilità della suddetta classe)

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

la simulazione, dal punto di vista pratico, inizia in questo punto:

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

  vEFrameObserverSwarm.buildObjects ();

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

  vEFrameObserverSwarm.buildActions();

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

  vEFrameObserverSwarm.activateIn(null);

- con il richiamo del metodo go(), ereditato dalla classe *swarm.simtoolsgui.GUISwarmImpl,* con il compito di avviare tutte le attività, ma necessario anche per permettere all'utente di gestire manualmente la simulazione, tramite il pannello di controllo delle attività

  vEFrameObserverSwarm.go();

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

vEFrameObserverSwarm.drop();

- con

System.exit(0);

per chiudere i grafici e la simulazione.

VEFrameObserverSwarm.java

La prima riga del listato

import swarm.Globals;

è già stata analizzata, mentre per le successive, dopo aver detto che la parte di codice *import swarm* richiama il package contenente le classi di Swarm, passerò alla descrizione delle singole classi e librerie il cui nome conclude il richiamo del comando import.

La funzione del comando *import* è quella di indicare al compilatore in quale package, diverso da *Object*, si trovino metodi ed oggetti utilizzati all'interno del programma. Così facendo non è più necessario passare al compilatore, ogni volta che serva, tutto il percorso per trovare la classe o libreria corretta.

Le classi o librerie richiamate per implementare l'Observer del modello jVEFrame sono:

- Selector

Questa classe si rende necessaria perché prima che un messaggio possa essere inserito all'interno di un *ActionGroup* o di uno *Schedule* esso dovrà essere incapsulato in un altro tipo d'oggetto chiamato *Selector* appunto (si veda il paragrafo su SwarmUtils.Java);

- defobj.Zone

La libreria defobj è necessaria per fornire supporto agli oggetti di Swarm; essa contiene i metodi più importanti per disegnare *agenti* e *swarm;* viene utilizzata anche come deposito per gestire le sonde (probe).
La classe Zone è un registro d'allocazione per i dati o per gli oggetti creati in Swarm. Una zona deve essere assegnata ogniqualvolta un nuovo oggetto è creato, essa diventa la fonte da cui dati per le variabili d'istanza e per altri dati interni sono ottenuti. Nell'intenzione degli sviluppatori di Swarm si vorrebbero utilizzare zone multiple per assicurarsi che oggetti con necessità di dati ed altre caratteristiche omogenee siano allocati insieme in maniera ottimale, od in modo tale che sia possibile condividere le medesime risorse;

- activity.Activity

Per comprendere l'utilità di questa classe occorre considerare uno Swarm come un insieme di agenti e di liste di attività che costoro dovranno svolgere: è agevole immaginare, rimanendo nell'ambito della programmazione orientata agli oggetti, che anche queste liste saranno oggetti definiti all'interno della suddetta classe;

- activity.ActionGroup
- activity.Schedule

Un *ActionGroup,* così come uno *Schedule,* è un "gruppo" di azioni in precedenza incapsulate nel Selector, che devono essere eseguite, secondo un ordine definito internamente, da parte di oggetti cui verranno passati come messaggi ;

- activity.ActionGroupImpl
- activity.ScheduleImpl

Queste sono le classi contenenti i metodi necessari per implementare sia gli *ActionGroup* sia gli *Schedule;*

- objectbase.Swarm

Si tratta di un contenitore temporaneo. Uno Swarm è un insieme di agenti che condividono sia il tempo sia una certa area di memoria;

- objectbase.VarProbe

Una classe che permette all'utente del programma di ottenere informazioni circa una particolare variabile di istanza (condivisa all'interno di una classe da tutti i metodi) o locale (propria di ogni metodo);

- objectbase.MessageProbe

Una classe che permette all'utente di richiamare un determinato messaggio riguardante ogni esemplare di una classe, o riguardante un oggetto che erediti da un'altra classe;

- objectbase.EmptyProbeMapImpl

Una *CustomProbeMap* utilizzata per creare *ProbeMaps* direttamente dall'interno del programma:

➢ una P*robeMap* è una classe-contenitore per Probe, utilizzato per definire il contenuto di un *ProbeDisplay*. Ad esempio tutte le sonde relative a variabili di istanza di un Model dovrebbero essere raccolte all'interno di una ProbeMap;

➢ un *ProbeDisplay* è l'interfaccia grafica da cui è possibile osservare le *Probe;*

➢ una *CustomProbeMap* è una sottoclasse della *ProbeMap* usata per creare *Probe* che siano inizializzate in uno "empty state" od utilizzando VarProbe e MessageProbe.

- simtoolsgui.GUISwarm
- simtoolsgui.GUISwarmImpl

Queste sono le classi che si rendono necessarie per introdurre e gestire una interfaccia grafica nel modello, permettendoci di controllare i grafici, decidendo ad esempio quante volte dovranno essere aggiornati nel corso della simulazione. In particolare vedremo più avanti come lo stesso Observer estenda le caratteristiche della superclasse GUISwarmImpl;

- analysis.EZGraph;

- analysis.EZGraphImpl;

Queste classi consentono all'utente di creare facilmente grafici (da cui il nome, infatti la pronuncia di EZ ricorda il termine inglese *easy* che significa facile) di diverse misure e tipologie, a seconda delle necessità. L'utente creerà prima i grafici e poi *"Sequences"* ossia liste ordinate di elementi, che verranno rappresentate. Le *"Sequences"* sono generate basandosi su dati forniti da un singolo oggetto o da un insieme di oggetti, in corrispondenza ad uno specifico selettore. Una delle caratteristiche più interessanti è la capacità della classe EZGraph di generare automaticamente medie, totali, minimi, massimi e conteggio delle sequenze senza che l'utente debba creare ulteriori sottoclassi;

- java.util.List;

Si tratta di una classe che mette a disposizione una *"Sequences"*. L'utente ha il controllo sull'esatta locazione dei singoli elementi inseriti nella lista e può accedere direttamente a ciascuno di essi con una ricerca che si basi sul numero intero d'ordine di ciascuno.
L'*Observer* contiene il *Model* e gli oggetti grafici per la rappresentazione dell'andamento dell'impresa virtuale. Estendendo la classe *GUISwarmImpl* la classe *VEFrameObserverSwarm* ne diventa direttamente una sottoclasse, sarà quindi in grado di utilizzare tutti i metodi della superclasse ed inoltre aggiungerà altri metodi propri utilizzabili poi solo dagli oggetti di tipo *VEFrameObserverSwarm* o da riferimenti ad oggetti della stessa.

public class VEFrameObserverSwarm extends GUISwarmImpl .

La seguente variabile intera definisce il numero di volte che i grafici devono essere aggiornati nel corso della simulazione, il valore di default è 1, ciò sta ad indicare che ad ogni ciclo vi sarà un aggiornamento dei grafici

public int displayFrequency;

La scelta se essere o meno *verbose* ci serve per mostrare, se vera, le descrizioni degli avvenimenti che si susseguono, utile in particolar modo in fase di *debug*

public boolean verboseChoice;

Questa variabile di tipo ActionGroup farà da contenitore della sequenza di eventi grafici

public ActionGroup displayActions;

Questa variabile contiene un esemplare della classe Schedule che servirà per gestire le sequenze temporali nell'Observer

public Schedule displaySchedule;

vEFrameModelSwarm è lo Swarm che noi osserveremo, esso è un esemplare della classe VEFrameModelSwarm

public VEFrameModelSwarm vEFrameModelSwarm;

Le seguenti variabili rappresentano i grafici o gli output su file alternativi

public EZGraphImpl waitingListGraph, warehouseGraph, totalTimeLengthGraph,totalTimeLengthDataFile, totalDailyCostFile, totalCostFile,

totalRevenueFile, dailyRevenueFile, dailySemimanufacturedOrderRevenueFile,

totalInventoryFinancialCostFile, dailyStoredComponentValueFile,

dailySemimanufacturedOrderCostFile, finishedOrderCostFile, benefitFile,

benefitGraph;

Una variabile di tipo PTHistogram che servirà a realizzare l'istogramma delle code presso le unità ed i magazzini.

public PTHistogram ptHistogram;

Il progetto PTolemy, su cui si fonda la classe, è sviluppato dal Department of Electrical Engeneering and Computer Science, University of California Berkeley, informazioni sono reperibili presso: http://ptolemy.eecs.berkeley.edu

Ogni costruttore in Swarm deve richiamare Zone e passare alla superclasse (GUISwarmImpl in questo caso) la zona di memoria in cui è depositata;

public VEFrameObserverSwarm (Zone aZone)

questo avviene perché gli autori del progetto intendono sviluppare la classe Zone in maniera tale da poter poi gestire più zone di memoria cui fare riferimento. Allo stato attuale questa diversificazione non è ancora stata implementata, perciò il passaggio del parametro aZone è una formalità (ha infatti valore nullo)

super(aZone)

Vengono inizializzati i valori delle due variabili

displayFrequency = 1
verboseChoice=false;

Costruiamo una mappa personalizzata delle sonde, utilizzando una sottoclasse "locale" della classe EmptyProbeMapImpl. Questa produce una classe separata di nome VEFrameObserver-Swarm1VEFrameObserverProbeMap.class

class VEFrameObserverProbeMap extends EmptyProbeMapImpl {

public VarProbe probeVariable (String name) {
return
Glo-
bals.env.probeLibrary.getProbeForVariable$inClass(name,VEFram
eObserverSwarm. this.getClass ());}
public MessageProbe probeMessage (String name) {
return

```
Glo-
bals.env.probeLibrary.getProbeForMessage$inClass(name,VEFra
meObserverSwarm. this.getClass ());}
public void addVar (String name) {
addProbe (probeVariable (name));}

public void addMessage (String name) {
addProbe (probeMessage (name));}
```

Con il seguente definiamo il costruttore per le probe passando alla superclasse *EmptyProbeMapImpl* i valori di _aZone ed *aClass;* richiamando il metodo *addVar* attiviamo le sonde sia per la variabile *displayFrequancy* sia per *verbosechoice*

```
public VEFrameObserverProbeMap (Zone _aZone, Class aClass) {
super (_aZone, aClass);
addVar ("displayFrequency");
addVar("verboseChoice");}
```

S'installa la classe *probeMap* direttamente nella probeLibrary

```
Glo-
bals.env.probeLibrary.setProbeMap$For(newVEFrameObserverPro
beMap(aZone, getClass()),getClass()).
```

Il procedimento seguito, per la realizzazione della sottoclasse delle sonde, è ciò che viene definito *Java inner class*, cioè una classe java realizzata internamente ad un'altra (da cui l'utilizzo di this all'interno dei me-

todi per far riferimento alla classe cui si appartiene dopo aver utilizzato getclass() che richiama la sottoclasse).
Creiamo ora gli oggetti necessari per rappresentare a video il Model ed effettuarne le rilevazioni.

public Object buildObjects ()

La prima cosa che occorre fare nel metodo buildObjects() è inizializzare il VEFrameObserverSwarm effettuando la chiamata al metodo buildO-bjects() della "parent class"

super.buildObjects ();

super applica il metodo buildObjects() al costruttore della classe richiamando il pannello di controllo della simulazione sul quale sono presenti alcuni pulsanti:

1. Start, necessario per avviare la simulazione,
2. Stop, necessario per mettere in pausa la simulazione;
3. Next, per procedere un passo alla volta;
4. Save, per salvare lo stato della simulazione;
5. Quit, per chiudere tutte le interfacce e terminare la simulazione.

Sono creati in sequenza:

- Il vEFrameModelSwarm che vogliamo esaminare, esso è creato non direttamente con il comando *new*, ma utilizzando *LispApplArchiver* per i motivi esposti in precedenza;

vEFrameModel-Swarm=(VEFrameModelSwarm)Globals.env.lispAppArchiver.getWith Zone$key(getZone(), "vEFrameModelSwarm");

- Sono poi create le sonde, per l'analisi e la gestione dei parametri della simulazione, sia del vEFrameModelSwarm sia del vEFrameObserverSwarm

Globals.env.createArchivedProbeDisplay (vEFrameModel-Swarm,"vEFrameModelSwarm");
Globals.env.createArchivedProbeDisplay (this, "vEFrameObserverSwarm");

- Si comunica al pannello di controllo di attivare per la prima volta il comando di Stop in automatico appena è creato il pannello stesso, la simulazione non sarà avviata fino a che ciò non sia stabilito dall'utente, il quale avrà il tempo di decidere quali parametri utilizzare nella simulazione.

getControlPanel ().setStateStopped ();

- Si comunica allo StartVEFrame la decisione se mostrare tutte le descrizioni degli avvenimenti o meno

StartVEFrame.verbose=verboseChoice;

- Comunichiamo al vEFrameModelSwarm che è il momento di costruire i propri oggetti (si veda più avanti il dettaglio)

vEFrameModelSwarm.buildObjects ();

- Viene creato l'oggetto grafico per rappresentare a video il contenuto delle liste di attesa creando un oggetto waitingList del tipo EZGraphImpl passandogli come parametri il titolo del grafico, degli assi cartesiani, i valori indicati nella legenda…

waitingListGraph = new EZGraphImpl(getZone (), "Waiting lists", "Time",
"Waiting list values (avrg, min, max)","waiting_list");

- …la posizione iniziale in cui è creata sullo schermo

Globals.env.setWindowGeometryRecordName (waitingListGraph, "waitingListGraph");

- Sono creati i dati richiamando i valori presenti nell'UnitList del vEFrameModelSwarm, ottenendo così il riferimento alle singole unità; di ciascuna si leggerà la lunghezza della coda e tramite il metodo createAverageSequence della classe GUISwarm si otterrà la media che sarà passata a waitingListGraph. Stesso procedimento è adottato per il minimo ed il massimo

waitingList-
Graph.createAverageSequence$withFeedFrom$andSelector("avrg
Waiting

List",vEFrameModelSwarm.getUnitList(),SwarmUtils.getSelector("Unit","getWaitingListLength"));

waitingList-Graph.createMinSequence$withFeedFrom$andSelector("minWaitingList", vEFrameModelSwarm.getUnitList(), SwarmUtils.getSelector("Unit","getWaitingListLength"));

waitingList-Graph.createMaxSequence$withFeedFrom$andSelector("maxWaitingList", vEFrameModelSwarm.getUnitList(), SwarmUtils.getSelector("Unit", "getWaitingListLength"));

- Con i seguenti comandi si crea il grafico per rappresentare le quantità accumulate nel magazzino d'ogni unità, se presenti

if (vEFrameModelSwarm.getUseWarehouses()){
warehouseGraph = new EZGraphImpl(getZone (), "Quantities in the warehouses", "Time", "Quantities (avrg, min, max)", "warehouse_list");
Globals.env.setWindowGeometryRecordName(warehouseGraph, "warehouseGraph");
warehouseGraph.createAverageSequence$withFeedFrom$andSelector("avrgQ", vEFrameModelSwarm.getWarehouseList(), SwarmUtils.getSelector("Warehouse","getInventoryQuantity"));

```
warehouseGraph.createMinSequence$withFeedFrom$andSelector("minQ", vEFrameModelSwarm.getWarehouseList(), SwarmUtils.getSelector("Warehouse","getInventoryQuantity"));
warehouseGraph.createMaxSequence$withFeedFrom$andSelector("maxQ", vEFrameModelSwarm.getWarehouseList(), SwarmUtils.getSelector("Warehouse","getInventoryQuantity"));
```

- Con i seguenti comandi si crea il grafico per rappresentare il tempo totale di produzione, una volta che gli ordini sono terminati, come rapporto fra tempo e lunghezza della ricetta

```
totalTimeLengthGraph = new EZGraphImpl(getZone (), "Ratio total time / total lengths", "Time", "Ratio", "totalTimeLengthGraph");
Globals.env.setWindowGeometryRecordName(totalTimeLengthGraph, "totalTimeLengthGraph");
totalTimeLengthGraph.createSequence$withFeedFrom$andSelector("ratio", vEFrameModelSwarm, SwarmUtils.getSelector(vEFrameModelSwarm,"getTimeLengthRatio"));
totalTimeLengthDataFile = new EZGraphImpl(getZone(), "data");
totalTimeLengthDataFile.createSequence$withFeedFrom$andSelector("ratio", vEFrameModelSwarm, SwarmUtils.getSelector(vEFrameModelSwarm,"getTimeLengthRatio"));
```

- Si utilizza un oggetto della classe EZGraphImpl per scrivere sul file totalDailyCosts.txt l'ammontare totale giornaliero dei costi delle unità

totalDailyCostFile = new EZGraphImpl(getZone(), "Costs/totalDailyCosts");
totalDailyCost-File.createTotalSequence$withFeedFrom$andSelector("txt", vE-FrameModelSwarm.getUnitList(), SwarmU-tils.getSelector("Unit","getTotalDailyCosts"));

- Si utilizza un oggetto della classe EZGraphImpl per scrivere sul file totalCosts.txt l'ammontare totale dei costi fissi e variabili delle unità dall'inizio della simulazione

totalCostFile = new EZGraphImpl(getZone(), "Costs/totalCosts");
totalCost-File.createTotalSequence$withFeedFrom$andSelector("txt",vEFrameModelSwarm.getUnitList(), SwarmU-tils.getSelector("Unit","getTotalCosts"));

- Si utilizza un oggetto della classe EZGraphImpl per scrivere sul file totalRevenues.txt l'ammontare dei profitti dell'impresa virtuale

totalRevenueFile = new EZGraphImpl(getZone(), "Reve-nues/totalRevenues");
totalRevenueFi-le.createTotalSequence$withFeedFrom$andSelector("txt",vEFrame

ModelSwarm.getUnitList(), SwarmUtils.getSelector("Unit","getTotalSaleRevenues"));

- Si utilizza un oggetto della classe EZGraphImpl per passare dagli ordini finiti al file dailyRevenues.txt l'ammontare dei costi giornalieri dell'impresa virtuale.

dailyRevenueFile = new EZGraphImpl(getZone(), "Revenues/dailyRevenues");
dailyRevenueFile.createTotalSequence$withFeedFrom$andSelector("txt", vEFrameModelSwarm.getUnitList(), SwarmUtils.getSelector("Unit","getSaleRevenues"));

- Si utilizza un oggetto della classe EZGraphImpl per passare dagli ordini finiti al file dailySemimanufacturedOrderRevenues.txt l'ammontare dei profitti dei semimanufatti

dailySemimanufacturedOrderRevenueFile = new EZGraphImpl(getZone(), "Revenues/dailySemimanufacturedOrderRevenues");
dailySemimanufacturedOrderRevenueFile.createTotalSequence$withFeedFrom$andSelector("txt",vEFrameModelSwarm.getOrderList(), SwarmUtils.getSelector("Order","getOrderEvaluation"));

- Si utilizza un oggetto della classe EZGraphImpl per passare la somma dei costi finanziari dei prodotti semifiniti presenti in magazzino

al file totalInventoryFinancialCosts.txt (sempre che i magazzini siano attivati)

```
if(vEFrameModelSwarm.getUseWarehouses()==true){
totalInventoryFinancialCostFile = new EZGraphImpl(getZone(),
"Costs/totalInventoryFinancialCosts");
totalInventoryFinancialCost-
File.createTotalSequence$withFeedFrom$andSelector("txt", vE-
FrameModelSwarm.getUnitList(), SwarmU-
tils.getSelector("Unit","getTotalInventoryFinancialCosts"));
```

- Si utilizza un oggetto della classe EZGraphImpl per passare il profitto dei componenti presenti in magazzino al file dailyStoredComponentValue.txt (sempre che i magazzini siano attivati)

```
if(vEFrameModelSwarm.getUseWarehouses()==true){
dailyStoredComponentValueFile = new EZGraphImpl(getZone(),
"Revenues/dailyStoredComponentValue");
dailyStoredComponentValue-
File.createTotalSequence$withFeedFrom$andSelector("txt", vE-
FrameModelSwarm.getUnitList(), SwarmU-
tils.getSelector("Unit","getDailyStoredComponentValue"));
```

- Si utilizza un oggetto della classe EZGraphImpl per passare il costo totale degli ordini dall'inizio della simulazione al file finishedOrderCosts.txt

finishedOrderCostFile = new EZGraphImpl(getZone(), "Costs/finishedOrderCosts");
finishedOrderCost-File.createTotalSequence$withFeedFrom$andSelector("txt", vE-FrameModelSwarm.getUnitList(), SwarmU-tils.getSelector("Unit","getTotalFinishedOrderCosts"));

- Si utilizza un oggetto della classe EZGraphImpl per passare il costo totale degli ordini in produzione al filedailySemimanufacturedOrderCosts.txt

dailySemimanufacturedOrderCostFile = new EZGra-phImpl(getZone(), "Costs/dailySemimanufacturedOrderCosts");

dailySemimanufacturedOrderCost-File.createTotalSequence$withFeedFrom$andSelector("txt", vE-FrameModelSwarm.getOrderList(), SwarmU-tils.getSelector("Order","getOrderTotalCosts"));

- Sono creati:
  - Il grafico dei guadagni totali dell'impresa, con i relativi dati;
  - Il file dei guadagni totali dell'impresa;

benefitGraph = new EZGraphImpl(getZone (), "Enterprise benefit", "Time", "Benefit", "benefitGraph");
Globals.env.setWindowGeometryRecordName(benefitGraph, "be-nefitGraph");

```
benefit-
Graph.createSequence$withFeedFrom$andSelector("benefitGraph"
, vEFrameModelSwarm, SwarmU-
tils.getSelector(vEFrameModelSwarm,"getEnterpriseBenefit"));
benefitFile = new EZGraphImpl(getZone(), "Benefit/benefit");
benefitFile.createSequence$withFeedFrom$andSelector("txt", vE-
FrameModelSwarm, SwarmU-
tils.getSelector(vEFrameModelSwarm,"getEnterpriseBenefit"));
```

- L'istogramma per la rappresentazione delle code di attesa di ciascuna unità e di eventuali magazzini

```
if(vEFrameModelSwarm.getUseWarehouses()==true){
ptHistogram = new PTHistogram("Orders", "Virtual Enterprise",
"Count", vEFrameModelSwarm.totalUnitNumber, 20.0, vE-
FrameModelSwarm.unitList, vEFrameMod-
elSwarm.warehouseList);}
else ptHistogram = new PTHistogram("Orders", "Virtual Enter-
prise", "Count",vEFrameModelSwarm.totalUnitNumber, 20.0, vE-
FrameModelSwarm.unitList);
return this;}
```

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

Il codice seguente serve per definire, all'interno dell'VEFrameObserverSwarm le azioni necessarie per avviare la simulazione.

public Object buildActions ()

In particolare si parte con l'inizializzazione del metodo facendo il richiamo del metodo buildActions() della superclasse

super.buildActions();

Successivamente verrà richiamato lo stesso metodo per il vEFrameModelSwarm permettendo così la costruzione dello schedule proprio di ogni Model

vEFrameModelSwarm.buildActions();

Il prossimo passo è quello di creare un ActionGroup per gestire le azioni necessarie a controllare vEFrameModelSwarm e per generare il display dello stesso. Si tratta di una serie di attività che vogliamo avvengano ad ogni passo durante la simulazione.

displayActions = new ActionGroupImpl (getZone());

Prima di tutto organizziamo l'aggiornamento per il display delle probe

displayAc-
tions.createActionTo$message(Globals.env.probeDisplayManager,
SwarmU-
tils.getSelector(Globals.env.probeDisplayManager,"update"));

Organizziamo poi l'aggiornamento degli oggetti grafici

```
displayAc-
tions.createActionTo$message(waitingListGraph,SwarmUtils.getSe
lector(waitingListGraph, "step"));

if(vEFrameModelSwarm.getUseWarehouses())
displayAc-
tions.createActionTo$message(warehouseGraph,SwarmUtils.getSe
lector(warehouseGraph, "step"));
displayAc-
tions.createActionTo$message(totalTimeLengthGraph,SwarmUtils.
getSelector(totalTimeLengthGraph, "step"));
displayAc-
tions.createActionTo$message(totalTimeLengthDataFile,SwarmUtil
s.getSelector(totalTimeLengthGraph, "step"));
displayAc-
tions.createActionTo$message(ptHistogram,SwarmUtils.getSelecto
r(ptHistogram, "addPoints"));
displayAc-
tions.createActionTo$message(totalDailyCostFile,SwarmUtils.getS
elector(totalDailyCostFile, "step"));
displayAc-
tions.createActionTo$message(totalCostFile,SwarmUtils.getSelecto
r(totalCostFile, "step"));
displayAc-
tions.createActionTo$message(totalRevenueFile,SwarmUtils.getSel
ector(totalRevenueFile, "step"));
```

```
displayAc-
tions.createActionTo$message(dailyRevenueFile,SwarmUtils.getSe
lector(dailyRevenueFile, "step"));
displayAc-
tions.createActionTo$message(dailySemimanufacturedOrderReve
nue-
File,SwarmUtils.getSelector(dailySemimanufacturedOrderRevenue
File, "step"));
if(vEFrameModelSwarm.getUseWarehouses()){
displayAc-
tions.createActionTo$message(totalInventoryFinancialCostFile,Sw
armUtils.getSelector(totalInventoryFinancialCostFile,          "step"));
    }
if(vEFrameModelSwarm.getUseWarehouses()){
displayAc-
tions.createActionTo$message(dailyStoredComponentValueFile,S
warmUtils.getSelector(dailyStoredComponentValueFile, "step"));}
displayAc-
tions.createActionTo$message(dailySemimanufacturedOrderCostF
ile,SwarmUtils.getSelector(dailySemimanufacturedOrderCostFile,
"step"));
displayAc-
tions.createActionTo$message(finishedOrderCostFile,SwarmUtils.
getSelector(finishedOrderCostFile, "step"));
displayAc-
tions.createActionTo$message(benefitFile,SwarmUtils.getSelector(
benefitFile, "step"));
```

displayAc-
tions.createActionTo$message(benefitGraph,SwarmUtils.getSelect
or(benefitGraph, "step"));

Infine occorre organizzare un aggiornamento per l'interfaccia utente nel suo complesso, questo è molto importante perché senza di esso non vi sono né aggiornamenti sui grafici né sulla possibilità di riutilizzare il pannello di controllo.

displayAc-
tions.createActionTo$message(getActionCache(),SwarmUtils.getSel
ector(getActionCache(),"doTkEvents"));

Si richiama una istanza della classe ScheduleImpl per creare effettivamente uno Schedule. In particolare la "versione" di ScheduleImpl che attiviamo, richiede l'assegnazione di aZone di tipo Zone, che viene attribuita con il richiamo getZone(), ed anche l'assegnazione di un valore di tipo int rptInterval utile per creare Schedule ripetitivi. Nel nostro caso la ripetizione delle azioni contenute nello Schedule avviene, di default, una volta ogni ciclo

displaySchedule = new ScheduleImpl (getZone (), displayFrequency);

Una volta che abbiamo definito le azioni da svolgere e che abbiamo creato lo Schedule inseriamo una istanza delle azioni nell'istanza dello Schedule medesimo…

displaySchedule.at$createAction (0, displayActions);
return this

…le azioni saranno svolte così come un tutt'uno dal gestore d'eventi

E' attivato lo schedule in modo che la simulazione sia pronta a partire. L'argomento swarmContext è la zona in cui VEFrameObserverFrame è attivato. VEFrameObserverSwarm e lo swarm al livello più elevato, cosicché esso è attivato in "null", mentre gli altri (sub) swarm e schedule saranno attivati all'interno di VEFrameObserverSwarm.

public Activity activateIn (Swarm swarmContext) {

Come in ogni fase precedente prima di tutto si richiama il metodo activateIn() della superclasse

super.activateIn (swarmContext);

Successivamente sarà avviato lo Schedule del vEFrameModelSwarm, l'utilizzo del comando this come parametro indica che il modello ha come swarmContext il vEFrameObserverSwarm medesimo,

vEFrameModelSwarm.activateIn (this).

Infine è attivato lo Schedule proprio del VEFrameObserverSwarm

displaySchedule.activateIn (this);

E' restituita l'attività di Swarm che è pronta ad essere avviata

return getActivity();

Il metodo drop() serve a chiudere l'Observer, ma ciò avviene solo dopo aver chiuso tutti i file creati via EZGraph.

public void drop()

VEFrameModelSwarm.java

Le classi importate non ancora incontrate sono

- defobj.SymbolImpl

Un simbolo è un oggetto creato con un nome fisso. Esso non ha altro comportamento se non quello di restituire il proprio nome; un *Symbol* è utilizzato normalmente per definire valori di id(entificazione) che vengono assegnati a costanti di tipo globale.

- objectbase.SwarmImpl

Si tratta di un contenitore temporaneo. Uno Swarm è un insieme d'agenti che condividono sia il tempo sia una certa area di memoria.

- collections.ListImpl

Una lista è un insieme d'elementi che rispettano una posizione, a loro assegnata dall'esterno della lista stessa. La sequenza è determinata dal punto in cui ciascun elemento è aggiunto: esso può essere aggiunto all'inizio della lista, alla fine o in qualunque altro punto.
Il VEFrameModelSwarm contiene le unità e tutti i relativi strumenti utilizzabili, come ad esempio i magazzini, necessari per dialogare con gli inventari, o le news.
Estendendo la classe SwarmImpl la classe VEFrameModelSwarm ne diventa direttamente una sottoclasse, sarà quindi in grado di utilizzare tutti i metodi della superclasse. Inoltre altri metodi saranno aggiunti, utilizzabili , o da riferimenti ad oggetti, di tipo VEFrameModelSwarm.

public class VEFrameModelSwarm extends SwarmImpl

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

public int totalUnitNumber

il numero totale di unit che utilizziamo nella simulazione;

public int maxStepNumber;

il numero massimo di passi da eseguire per completare un ordine

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 (false) i magazzini e le scorte

public boolean useNewses;

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

public int infDeepness;

quanto in profondità l'informazione è propagata (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 float enterpriseBenefit;

il guadagno dell'impresa

public int nOfNewsesToProduce;

il numero di notizie necessarie per decidere di produrre scorte

public int nOfNewsesToBeCleared;

il numero di notizie che devono essere cancellate dopo la decisione di incrementare le scorte

public int nOfOrdersInNewses;

il numero d'ordini per i quali le notizie 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'orderGenerator, che per il momento opera con una selezione casuale sia della lunghezza sia del contenuto delle ricette

public ActionGroup modelActions;

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

public Schedule modelSchedule;

lo Schedule operante nel VEFrameModelSwarm

public InventoryRuleMaster inventoryRuleMaster;

il gestore di regole che controlla le scorte

public InformationRuleMaster informationRuleMaster;

il gestore del flusso d'informazioni (notizie)

public AccountingParameters accountingParameters;

la variabile necessaria per il conteggio di costi fissi e variabili

public ListImpl unitList;

la lista indicante le unità produttive attivate

public ListImpl warehouseList;

la lista indicante i magazzini attivati

public ListImpl orderList;

l'intera lista degli ordini.
Il costruttore della classe VEFrameModelSwarm ha il compito di definire quali siano i principali parametri da passare all'oggetto quando questo è creato

public VEFrameModelSwarm (Zone aZone)

Viene passata la zona di memoria in cui registrare questo model al costruttore della superclasse SwarmImpl

super (aZone);

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

totalUnitNumber = 3;
maxStepNumber=5;
useWarehouses=true;
maxInWarehouses=3;
minInWarehouses=3;
useNewses=true;

```
infDeepness=3;
inventoryFinancialRate= (float) 0.1;
inventoryEvaluationCriterion=1;
revenuePerEachRecipeStep=2;
nOfNewsesToProduce=1;
nOfNewsesToBeCleared=100000;
nOfOrdersInNewses=1;
```

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

```
class VEFrameModelProbeMap extends EmptyProbeMapImpl

public VarProbe probeVariable (String name) {
return
Globals.env.probeLibrary.getProbeForVariable$inClass(name,    VE-
FrameModelSwarm.this.getClass ());}
public MessageProbe probeMessage (String name) {
return
Globals.env.probeLibrary.getProbeForMessage$inClass(name,
VEFrameModelSwarm.this.getClass ());}
public void addVar (String name) {
addProbe (probeVariable (name));}
public void addMessage (String name) {
addProbe (probeMessage (name));}
public VEFrameModelProbeMap (Zone _aZone, Class aClass) {
super (_aZone, aClass);
```

```
addVar ("totalUnitNumber");
addVar ("maxStepNumber");
addVar ("useWarehouses");
addVar ("useNewses");
addVar ("maxInWarehouses");
addVar ("minInWarehouses");
addVar ("infDeepness");
addVar ("inventoryFinancialRate");
addVar ("inventoryEvaluationCriterion");
addVar ("revenuePerEachRecipeStep");
addVar ("nOfNewsesToProduce");
addVar ("nOfNewsesToBeCleared");
addVar ("nOfOrdersInNewses");}}
Globals.env.probeLibrary.setProbeMap$For(new VEFrameModel-
ProbeMap (aZone, getClass ()), getClass ());}
```

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

```
public Object buildObjects ()
```

La prima cosa che occorre fare nel metodo buildObjects() è inizializzare il VEFrameModelSwarm tramite il metodo buildObjects() della "parent class"

```
super.buildObjects();
```

Si impostano le seguenti attività

- Creazione di un'istanza della classe AccountingParameters che contiene i metodi relativi alla gestione della contabilità dei costi fissi e variabili, passandole la zona di memoria in cui è situato il Model e il numero totale d'unità

accountingParameters = new AccountingParameters(getZone(), totalUnitNumber);

- Lettura tramite i metodi propri della classe AccountingParameters dei costi fissi e variabili dai file FixedCosts.txt e VarialeCosts.txt

accountingParameters.readFixedCosts();
accountingParameters.readVariableCosts();

- Controllo del limite massimo di scorte possibili in magazzino, così come asssegnato o in jVEFrame.scm o direttamente dal client. Se pari a 0 allora verrà attribuito il massimo valore integer possibile (2.147.483.647)

if (maxInWarehouses==0)
{maxInWarehouses=Integer.MAX_VALUE;
System.out.println("Warning, maxInWarehouses has been changed from 0 to Integer.MAX_VALUE"); }

- è creata un'istanza della classe ListImpl che registrerà le unità (il numero assegnato alle unità) attivate;

unitList = new ListImpl (getZone());

- è creata un'istanza della classe ListImpl che registrerà i magazzini assegnati alle unità attivate;

warehouseList = new ListImpl (getZone());

- è creata un'istanza della classe ListImpl che registrerà gli ordini esistenti ad ogni passo di simulazione

orderList = new ListImpl(getZone());

- è creata un'istanza della classe OrderGenerator che creerà una ricetta di lavorazioni, scegliendone casualmente sia la lunghezza sia il contenuto

orderGenerator = new OrderGenerator (getZone(), maxStepNumber, unitList, orderList, this);

- è creata un'istanza della classe InventoryRuleMaster che è il gestore delle regole per le scorte

inventoryRuleMaster = new InventoryRuleMaster(getZone(),maxInWarehouses, minInWarehouses, nOfNewsesToProduce, nOfNewsesToBeCleared);

- è creata un'istanza della classe InformationRuleMaster che è il gestore delle regole per il flusso d'informazioni

```
if(useWarehouses && useNewses)
informationRuleMaster = new InformationRuleMaster(getZone(),
totalUnitNumber);
```

- con l'utilizzo di un ciclo iterativo sono create tante unit quanto è il valore di totalUnitNumber e, se attivati, anche i magazzini corrispondenti

```
for (i = 1; i <= totalUnitNumber; i++){
Warehouse aWarehouse=null;
```

si assegna il valore "null" per evitare errori quando il magazzino non è attivato

```
News aNews=null;
```

si assegna anche in questo caso il valore "null", per evitare errori quando non sono attivate le news

```
Unit aUnit;
```

Distinguiamo tre casi:

- il caso in cui i magazzini sono attivati, ma non lo sono le news

```
if(useWarehouses && ! useNewses){
aWarehouse =  new Warehouse (getZone(), i);
```

il nuovo oggetto aWarehouse richiede il passaggio di parametri necessari sia per conoscere in quale zona è allocato (getZone()) sia per conoscere il proprio "nome" (i)

aUnit = new Unit (getZone(), i, unitList, this, inventoryRuleMaster, aWarehouse, accountingParameters.getFixedCosts(i), accountingParameters.getVariableCosts(i), orderList);

il nuovo oggetto aUnit per essere istanziato correttamente richiede il passaggio di parametri necessari per conoscere in quale zona è allocato (getZone()), per conoscere il proprio "nome" (i), per conoscere quali altre unità esistono (unitList), a quale vEFrameModelSwarm far riferimento (this), a quale gestore di regole delle scorte far riferimento (inventoryRuleMaster), a quale magazzino viene associata (aWarehouse), l'ammontare dei costi fissi ad ogni passo (accountingParameters.getFixedCosts), l'ammontare dei costi variabili ad ogni passo di produzione (accountingParameters.getVariableCosts) ed infine la lista degli ordini in produzione (orderList).

- il caso in cui siano usati sia i magazzini sia le news :

aUnit = new Unit (getZone(), i, unitList, this, inventoryRuleMaster, aWarehouse, informationRuleMaster, infDeepness, nOfOrdersInNewses, accountingParameters.getFixedCosts(i), accountingParameters.getVariableCosts(i), orderList);

rispetto al precedente, l'oggetto aUnit con magazzini e news, richiede anche di essere messo al corrente di quale sia il gestore del flusso di in-

formazioni (informationRuleMaster), della profondità fino alla quale le news circolano (infDeepness) ed il numero di ordini per i quali le notizie sono propagate da un'unità (nOfOrdersInNewses).

- il caso in cui non siano usati né i magazzini né le news

aUnit = new Unit (getZone(), i, unitList, this, accountingParameters.getFixedCosts(i), accountingParameters.getVariableCosts(i), orderList);

rispetto ai precedenti l'oggetto aUnit non richiederà alcun'informazione circa i magazzini o le news.

- Sia la nuova unità sia, eventualmente, il nuovo magazzino vengono aggiunti alle rispettive liste in ultima posizione utilizzando il metodo addLast() della classe ListImpl

unitList.addLast (aUnit);
if(useWarehouses) warehouseList.addLast (aWarehouse);

- Di seguito, come altrove all'interno del programma, la scelta di verità della variabile verboseChoice determina (se true) la visualizzazione di messaggi di vario tipo

if(! useWarehouses) if(StartVEFrame.verbose)
System.out.println("No warehouses generated");

- Il metodo restituirà il proprio contenuto.

return this;

Il metodo buildActions(), come già descritto nel VEFrameObserver-Swarm, serve per descrivere le azioni necessarie per avviare la simulazione e per costruire lo Schedule che gestisce la simulazione del tempo nel modello.

public Object buildActions ()

Il metodo inizializza anzitutto lo stesso buildActions() della superclasse

super.buildActions();

Il passo successivo è quello di creare un ActionGroup per gestire le azioni che pongono in essere realmente la simulazione. In particolare si tratta delle azioni definite nelle classi Unit ed OrderGenerator:

- UnitStep1;
- CreateRandomOrderWithNSteps;
- UnitStep2;
- UnitStep3.

Questi richiami ai metodi permettono di eseguire la produzione, aumentare o ridurre le scorte, archiviare nuove richieste di beni o servizi, eventualmente comunicare alle unità l'arrivo di nuovi ordini.

Molto interessante è la struttura del richiamo dei metodi, ad esempio per attivare unitStep3(). Si procede come segue: viene assegnato a ciascun elemento della lista delle unità un messaggio, il quale è un richiamo al metodo getSelector della classe *SwarmUtils.java* (interno al package jVEFrame 0.5.0). A sua volta il metodo getSelectorpassa un

messaggio incapsulato a unitStep3(), dopo aver trovatouna classe di nome “Unit” (si veda più avanti per la descrizione della classe *SwarmUtils.java*).

```
modelActions = new ActionGroupImpl (getZone ());
modelActions.createActionForEach$message(unitList,SwarmUtils.getSelector("Unit","unitStep1"));
modelActions.createActionTo$message(orderGenerator,SwarmUtils.getSelector(orderGenerator, "createRandomOrderWithNSteps"));
modelActions.createActionForEach$message(unitList,SwarmUtils.getSelector("Unit","unitStep2"));
modelActions.createActionForEach$message(unitList,SwarmUtils.getSelector("Unit","unitStep3"));
```

Infine è creato lo Schedule che serve per eseguire il modelActions della classe ActionGroup, che da sola non avrebbe modo di gestire il tempo. Per eseguire tutte le azioni quando necessarie lo Schedule richiede una definizione di Zone di sua competenza e si attiverà una volta ogni ciclo.

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

Una volta che abbiamo definito le azioni da svolgere e che abbiamo creato lo Schedule inseriamo una domanda del modelActions nell’istanza dello Schedule medesimo, 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

modelSchedule.at$createAction (0, modelActions);

E' attivato lo schedule in modo che la simulazione sia pronta a partire. L'argomento swarmContext è la zona in cui VEFrameModelFrame è attivato. Essendo VEFrameModelSwarm un (sub) swarm di VEFrameObserverSwarm, sarà attivato all'interno dell'ambiente dell'Observer.
La differenza fondamentale con VEFrameObserverSwarm sta nel fatto che non si attiva, ma direttamente il suo Schedule

modelSchedule.activateIn (this).

Seguono ora una lista di metodi utilizzati all'interno del package JVEFrame nella sua interezza, oltre che in questa classe:

getUnitList()

restituisce il contenuto della lista delle unità;

getWarehouseList ()
restituisce il contenuto della lista dei magazzini;

getOrderList()

restituisce il contenuto della lista degli ordini;

getUseWarehouses()

restituisce vero se i magazzini sono usati e falso nel caso opposto;

setProductionTimeAndRecipeLength(int pt, int rl)

registra il tempo totale di produzione e la lunghezza della ricetta. Richiede, come argomenti, il valore intero sia del tempo di produzione sia della lunghezza dell'ordine eseguito

getTimeLengthRatio()

restituisce totalProductionTime/totalRecipeLength se totalRecipeLength è diverso da 0 altrimenti restituisce 1, valore questo considerato un buon compromesso essendo 0 la lunghezza totale della ricetta nel primo giorno di simulazione ed avendo comunque impiegato una unità di tempo per concludere il primo ordine

getInventoryFinancialRate()

restituisce il tasso finanziario da applicare alle scorte

getInventoryEvaluationCriterion()

restituisce il valore scelto per la valutazione delle scorte

getRevenuePerEachRecipeStep()

restituisce il valore della ricchezza d'ogni passo della ricetta

getEnterpriseBenefit()

restituisce il profitto dell'impresa come differenza tra:

- la sommatoria dei ricavi dovuti ai prodotti finiti e, qualora fossero attivati i magazzini, quelli derivanti dai semilavorati e dalle rimanenze;
- la sommatoria dei costi fissi e variabili e, qualora fossero attivati i magazzini, quelli derivanti dal costo delle scorte

OrderGenerator.java

L'OrderGenerator è utilizzato per simulare il mercato, per questa versione gli ordini da evadere sono ottenuti dalla generazione casuale sia dei contenuti sia della lunghezza delle ricette, il cui limite massimo è rappresentato dalla quantità di Unit presenti

public class OrderGenerator extends SwarmObjectImpl

Le variabili d'istanza sono

public int maxStepNumber;

il numero massimo di passi presenti nella ricetta

public int orderCount;

utilizzato per registrare il numero d'ordini generati

public Order anOrder;

un ordine

public ListImpl unitList;

la lista contenente tutte le unità

public ListImpl orderList;

la lista contenente tutti gli ordini

VEFrameModelSwarm vEFrameModelSwarm;

Il Model cui si applica il guadagno derivante dall'ordine

public int[] orderRecipe;

il vettore contenente una ricetta
Il costruttore della classe

public OrderGenerator(Zone aZone, int msn, ListImpl ul, ListImpl ol, VEFrameModelSwarm mo)

innanzitutto i parametri assegnati nel VEFrameModelSwarm sono utilizzati per inizializzare il costruttore della classe; viene anche inizializzato il vettore orderRecipe che sarà il contenitore effettivo dei valori rappresen-

tativi le fasi necessarie per ottenere il prodotto finito (siano esse sequenze di lavorazioni di un'azienda metalmeccanica, dei servizi, del credito…)

Di seguito la spiegazione dei metodi propri della classe:

public void createRandomOrderWithNSteps()

serve per determinare casualmente sia la lunghezza della ricetta sia il contenuto d'ogni passo, che avviene tramite l'utilizzo di routine per la selezione casuale di numeri interi compresi tra il primo argomento (richiamato con getIntegerWithMin) ed il secondo (richiamato con withMax). Nel primo caso la scelta avverrà tra il valore uno ed il massimo numero di passi impostati con valore di default nel Model, nel secondo caso avverrà tra uno ed il numero d'unità presenti nella relativa lista

randomStepNumber = Globals.env.uniformIntRand.getIntegerWithMin$withMax(1, maxStepNumber);
orderRecipe[i] = Globals.env.uniformIntRand.getIntegerWithMin$withMax(1, unitList.getCount());

è creata un'istanza della classe Order, vale a dire un ordine cui è stata passata la zona di riferimento, il numero d'ordine, il momento di "ricezione", la lunghezza dello stesso, il contenuto ed il riferimento allo swarm

anOrder = new Order(getZone(), orderCount, Globals.env.getCurrentTime(), randomStepNumber,orderRecipe, vEFrameModelSwarm);

Successivamente l'ordine è aggiunto in coda alla lista corrispondente e sarà eliminato quando terminato

orderList.addLast(anOrder);

L'ordine è infine mandato alla prima unità di produzione: per ogni elemento della lista delle unità, quando c'è corrispondenza fra il primo passo di produzione, contenuto del nuovo ordine, e la fase di lavorazione di aUnit (che per il momento coincide con il numero dell'unità stessa), l'ordine è aggiunto alla sua coda di produzione.

```
operatingUnitNotFound=true;
for (i=0;i<unitList.getCount() && operatingUnitNotFound;i++){
aUnit=(Unit) unitList.atOffset(i);
if  (anOrder.getNextStep () == aUnit.getProductionPhase()){
operatingUnitNotFound=false;
aUnit.setWaitingList(anOrder);}}
```

Order.java

Un order è:

…the form containig our metaphoric object (Terna 2001)

le classi da importare e da cui eredita le proprie caratteristiche sono già state descritte in precedenza, si faccia riferimento a quelle per chiarimenti.

Le variabili d'istanza definite sono le seguenti:

public int stepNumber;

il numero di passi nella ricetta di produzione

public int orderNumber;

il numero ordinale di quest'ordine

public int startingDay;

il "giorno in cui è stato generato"

public int[] orderRecipe;

il vettore d'interi contenente i passi della ricetta

public int[] orderState;

il vettore d'interi contenente le fasi di lavorazione già effettuate

public int numberOfTheStepForWhichANewsIsSent;

il numero sequenziale di passo della ricetta per il quale è mandata una news

public float cumulatedFixedCost, cumulatedVariableCost;

i costi fissi e variabili dell'ordine

public float orderTotalCosts = 0;

il costo totale dell'ordine

VEFrameModelSwarm vEFrameModelSwarm;

il Model cui facciamo riferimento
La classe Order inizia definendo il suo costruttore e richiamando la superclasse

public Order (Zone aZone, int n, int sd, int sn, int [] r, VEFrameModelSwarm model).

Le variabili locali vengono inizializzate con i valori passati al momento della creazione di ogni singola istanza all'interno di OrderGenerator, come esposto in dettaglio precedentemente.

```
super(aZone);
orderNumber=n;
startingDay=sd;
stepNumber=sn;
vEFrameModelSwarm = model;
```

Le seguenti vengono inizializzate localmente, in particolare orderRecipe[] conterrà i passi della ricetta di lavorazione:

```
orderRecipe= new int[stepNumber];
orderState = new int[stepNumber];
int i
```

Si attribuiscono ad orderRecipe[] i passi della ricetta di lavorazione e si annuncia alla console (se verboseChoice è vera) la presenza del nuovo ordine, con relativo numero

```
for (i=0;i<stepNumber;i++)orderRecipe[i]=r[i];
if(StartVEFrame.verbose)System.out.println("New order (#" + orderNumber + ")");
```

Si determina la condizione delle fasi di lavorazione nel vettore di stato (0=empty; 1=done) e la si annuncia

```
if(StartVEFrame.verbose)System.out.print("orderState  vector ");
for (i=1;i<=stepNumber;i++) {
orderState[i-1]=0;
if(StartVEFrame.verbose)System.out.print(orderState[i-1] + " ");}
if(StartVEFrame.verbose)System.out.println();
if(StartVEFrame.verbose)System.out.print("orderRecipe vector ");
for (i=1;i<=stepNumber;i++)
if(StartVEFrame.verbose)System.out.print(orderRecipe[i-1] + " ");
if(StartVEFrame.verbose)System.out.println();}
```

Seguono ora una lista di metodi utilizzati per la gestione del package JVEFrame nella sua interezza:

getNextStep()

Il metodo restituisce la cifra assegnata ad un determinato passo della ricetta, se non è ancora stato assegnato ad un'unità (stato questo rilevato nel vettore orderState)

setDoneStep()

cerca il primo elemento di orderState[i] pari a zero e lo imposta su uno.

getOrderNumber()

restituisce il numero d'ordine dell'order cui si riferisce

getStepNumber()

restituisce il numero di passi

getStartingDay()

restituisce il momento ("giorno") di ricevimento dell'order

getHowManyStepsToBeDone()

restituisce quanti passi devono essere fatti per terminare la produzione. Per far ciò determina il contenuto della variabile stepsToBeDone come differenza fra il totale dei passi nella ricetta ed il valore di un contatore

che si incrementa di una unità ogni volta che un elemento del vettore orderState[] è uguale ad uno

getProductionToBeDoneAtStepAheadNumber(int sn)

il metodo richiede il valore parametrico indicante il numero sequenziale di passo nella ricetta. Posiziona il cursore al primo valore d'indice diverso da uno nel vettore orderState (primo passo della ricetta ancora da completare), per ottenere numberOfTheStepForWhichANewsIsSent come risultato della somma tra il valore dell'indice ed il numero passato in argomento diminuito di uno.
Il metodo restituisce la fase di lavorazione da compiere allo specifico passo considerato per interrogare le altre unità e sapere quale di loro la eseguirà.

getNumberOfTheStepForWhichANewsIsSent()

restituisce il numero sequenziale della ricetta con riferimento al quale sarà mandata una news

getOrderEvaluation()

restituisce il valore della produzione determinato come prodotto tra i passi effettivi che sono stati compiuti ed il ricavo registrato per ogni passo della ricetta.

setFixedAndVariableOrderCosts(float fc, float vc)

determina la somma dei costi fissi, dei costi variabili, dei costi totali sulla base dei parametri che sono passati come argomenti (costi fissi, costi variabili)

getOrderTotalCosts()

restituisce il costo totale dell'ordine.

Unit.java

Come le definisce il suo autore:
The Unit class instances are micro unit in our virtual enterprise; i.e. the units where the steps required to build a product (to fulfill an order) are done. (Terna 2001)
Le variabili d'istanza sono:

public int unitNumber;

il numero che identifica la unit

public boolean useWarehouses;

la scelta circa l'utilizzo o meno dei magazzini,

public boolean useNewses;

la scelta sull'utilizzo delle news

public boolean madeProduction;

un segnale sull'utilizzo ripetuto del magazzino

public ListImpl unitList;

la lista di tutte le unità esistenti nella nostra simulazione

public ListImpl waitingList;

la lista degli ordini in coda presso l'unità

public ListImpl dailyProductionList;

la lista contenente gli ordini evasi in una giornata, se presenti, anche considerando quelli evasi tramite l'uso delle scorte in magazzino

VEFrameModelSwarm vEFrameModelSwarm;

il Model da cui ricaviamo informazioni circa il tempo di produzione, la lunghezza della ricetta, ecc…

public Unit aUnit;

l'indirizzo di memoria temporaneo cui far riferimento per le Unit

public Unit receivingUnit;

l'indirizzo di memoria temporaneo cui far riferimento per la unità alla quale mandare una news

public Warehouse myWarehouse;

il magazzino, se attivato

public Order firstOrder;

il primo ordine da evadere

public Order incomingOrder;

un ordine inserito in una news

public News aNews;

una news da mandare

public News toBeDroppedNews;

la news da cancellare all'arrivo dell'ordine

public InventoryRuleMaster inventoryRuleMaster;

il gestore delle regole delle scorte

public InformationRuleMaster informationRuleMaster;

il gestore delle regole del flusso di informazioni

public ListImpl newsList;

la lista delle news

public int informationDeepness;

per quanti passi dopo l'attuale fase di produzione divulgheremo le news

public int nOfOrdersInNewses;

numero di ordini per i quali viene mandata una news

public ListImpl wholeOrderList;

la lista completa degli ordini

public float fixedCosts, variableCosts;

costi fissi e variabili

public float totalDailyVariableCosts;

costi variabili giornalieri totali

public float totalDailyCosts;

costi totali giornalieri

public float totalCosts = 0;

costi sin dall'inizio della simulazione

public float saleRevenues;

l ricavo giornaliero derivante dagli ordini completi

public float totalSaleRevenues = 0;

i ricavi sin dall'inizio della simulazione

public float totalFinishedOrderCosts;

il costo totale dell'ordine posrtato a termine

public float dailyStoredComponentValue = 0;

la valutazione giornaliera dei componenti immagazzinati

float totalInventoryFinancialCosts = 0;

il costo cumulato delle scorte

Di seguito sono elencati i costruttori della classe. I parametri assegnati nel VEFrameModelSwarm sono utilizzati per inizializzare il costruttore della classe:

- con i magazzini ma senza le news

  public Unit (Zone aZone, int un, ListImpl ul, VEFrameModelSwarm mo, InventoryRuleMaster rm, Warehouse wh, float fc, float vc, ListImpl ol)

- Senza né magazzini né news

  public Unit (Zone aZone, int un, ListImpl ul, VEFrameModelSwarm mo, float fc, float vc, ListImpl ol)

- Sia con i magazzini sia con le news

  public Unit (Zone aZone, int un, ListImpl ul, VEFrameModelSwarm mo, InventoryRuleMaster rm, Warehouse wh, InformationRuleMaster irm, int idp, int noin, float fc, float vc, ListImpl ol)

All'inizio di ogni "ciclo" le unità cercano uno o più ordini da eseguire. Al momento il ciclo di produzione consente un ordine per periodo, ma potremmo anche utilizzare le scorte…

unitStep1()

Innanzitutto sono annullati i costi variabili e totali giornalieri poi, se si utilizzano i magazzini, dopo aver effettuato uno o più passi della produ-

zione, tramite l'unità o tramite le scorte, l'ordine è assegnato ad una lista di produzione giornaliera, utilizzata nel metodo unitStep2 per diffondere i risultati della produzione.

Nel dettaglio la produzione avviene come segue:

- Se la lista delle code presso una unità contiene qualche ordine, il primo di essi viene spostato in una lista di produzione giornaliera :
  - Se i magazzini sono attivati
    - se il magazzino contiene delle scorte, lo si riduce di una unità;
    - altrimenti, se non vi è nulla, si effettuerà la produzione e cioè si aggiungono i costi variabili giornalieri all'ammontare totale dei costi variabili.
- successivamente si aggiorna il vettore orderState[], segnalando l'esecuzione della fase appena terminata.

- Se la lista delle code presso una unità non contiene alcun ordine si effettua il controllo per produrre per il magazzino:
  - Se si usano le news:
    - avviene il controllo con il gestore delle scorte che comunicherà se vi sono prodotti da preparare sulla base di quanto è richiesto via news ed in caso affermativo procederà in tal senso. Comunque si acconsentirà alla produzione per il magazzino solo se il livello delle scorte è minore del massimo e contemporaneamente c'è un ordine in arrivo (informazione ricevuta grazie alle news) oppure se il livello delle scorte è inferiore al minimo. Successivamente è aggiornato il costo variabile totale;
  - Se non si usano le news:
    - Si consentirà la produzione per il magazzino se il livello delle scorte è minore del massimo e se, contemporaneamente, il livello delle scorte è inferiore al minimo. . Successivamente è

vello delle scorte è inferiore al minimo. . Successivamente è aggiornato il costo variabile totale;

- Infine, se si utilizzano i magazzini, si potranno completare contemporaneamente (nello stesso ciclo) tutti i passi in coda presso una determinata unità, sino ad esaurimento delle scorte. Allo stesso tempo vengono addebitati alla singola ricetta tutti i costi di produzione relativi alle fasi effettuate, viene aggiornato il vettore orderState[], se il passo successivo di produzione può essere fatto in questa unità esso verrà assegnato immediatamente ed allo stesso tempo evaso se disponibile in magazzino, altrimenti viene inserito nella lista di produzione giornaliera.

unitStep2()

- La seconda fase di attività del nostro modello di simulazione è quella di azzerare i ricavi giornalieri ed i costi totali dell'ordine precedente. Poi fino a che la lista di produzione giornaliera non sarà vuota controllerà le ricette per sapere se i passi siano stati effettuati tutti.
- Per gli ordini terminati si informa l'ambiente (vEFrameModelSwarm) del tempo trascorso per evadere un determinato ordine e del numero di passi nella ricetta. L'ordine verrà rimosso dalla lista generale e successivamente verranno registrati i ricavi, i costi e verrà chiuso l'oggetto.
- Gli ordini non ancora terminati, verranno assegnati alla unità di produzione che risponderà dichiarando di essere in grado di effettuare tale fase di lavoro, con lo stesso procedimento presentato alla nascita dell'istanza anOrder

- Vengono registrati i valori della contabilità descritti più avanti in questa stessa classe

unitStep3()

Fino a che contemporaneamente:

- il contatore n è minore del numero di ordini per i quali le news verranno propagate;
- ci sono ordini, in attesa che un passo della loro ricetta venga terminato
- vi è utilizzo delle news;
  - si calcolano i passi ancora da fare prima di terminare la ricetta del primo ordine in coda;
  - rispettando sia il numero di passi della ricetta che devono ancora essere effettuati, sia la profondità massima alla quale l'informazione può essere diffusa, registriamo la cifra indicativa del passo da eseguire, in *receivingUnitNumber* ;
  - Si richiama l'informationRuleMaster per controllare la possibilità di mandare una news all'unità successiva, ci interroghiamo per sapere se l'unità in cui ci troviamo non sia già in grado di effettuare tale passo della ricetta. Se entrambi i controlli danno esito positivo allora si procede;
  - Si controlla tutta la lista delle unità;
  - Se il numero di unità corrisponde con la fase di lavorazione (per adesso la fase di lavorazione restituisce il numero della unità stessa) viene generata una istanza della classe News, la quale viene aggiunta alla lista dell'unità che dovrà lavorare il passo indicato

dalla stssa, dopo essersi assicurati che la News non sia stata ripetuta.

public int getProductionPhase()

restituisce il numero dell'unità, che per il momento coincide con la fase di produzione;

public void setWaitingList(Order anOrder)

Asssegna in coda alla lista dell'unità il nuovo ordine.

waitingList.addLast(anOrder);

Se si utilizzano le news e contemporaneamente la lista delle stesse non è vuota, si scorre l'intera lista delle news che devono essere eliminate, continuando a spostare in ultima posizione quelle inerenti ad ordini diversi da quello appena giunto alla Unit, sino a quando si trovino news contenenti informazioni circa l'ordine inserito in lista e le si cancellano.

```
if (useNewses && newsList.getCount()>0)
for (i=1;i<=newsList.getCount();i++){
toBeDroppedNews=(News)newsList.removeFirst();
if(toBeDroppedNews.incomingOrderInNews.getOrderNumber()==a
nOrder.getOrder Number()){
toBeDroppedNews.drop();
if(StartVEFrame.verbose)
```

```
System.out.println("Dropping the news announcinig the order n° "+
toBeDroppedNews.incomingOrderInNews.getOrderNumber());}
else
newsList.addLast(toBeDroppedNews);}}
```

public int getWaitingListLength()

restituisce la lunghezza della lista cui il metodo fa riferimento

public void setNewsList(News aNews)

Fa in modo che la Unit accetti un ordine contenuto in una determinata News se ,dopo aver effettuato il controllo delle news all'interno della rispettiva lista, non trova altre News che contengano lo stesso Order con lo stesso passo da effettuare (si tratterebbe di una duplicazione della stessa News all'interno della lista).

```
if (newsList.getCount()>0){
for (i=0;i<newsList.getCount() && ! duplicatedNews;i++){
controlNews=(News) newsList.removeFirst();
newsList.addLast(controlNews);
if(toBeCheckedNews.incomingOrderInNews.getOrderNumber()==co
ntrol-
News.incomingOrderInNews.getOrderNumber()&&toBeCheckedNe
ws.getNumberOfTheStepForWhichA NewsIs-
Sent()==controlNews.getNumberOfTheStepForWhichANewsIsSent()
)
duplicatedNews=true;}
```

if (! duplicatedNews){
newsList.addLast(toBeCheckedNews);
if(StartVEFrame.verbose)
System.out.println("Unit " + unitNumber + " accept a news containing order # "+ toBeCheckedNews.incomingOrderInNews. getOrderNumber());

Altrimenti la News non viene accettata

System.out.println("Unit " + unitNumber +" doesn't accept a news containing order # "+ toBeCheckedNews.incomingOrderInNews.getOrderNumber() +" which just exists for the same step");}

La News viene anche accettata in ogni caso in cui la lista sia vuota.

else{
newsList.addLast(toBeCheckedNews);
if(StartVEFrame.verbose)
System.out.println("Unit " + unitNumber +" accepts a news containing order N° "+ toBeCheckedNews.incomingOrderInNews.getOrderNumber()+" because newsList is empty");}}

public void addDailyVariableCosts()

determina il valore dei costi variabili giornalieri

public void setTotalDailyCosts()

determina il valore totale di costi fissi e costi variabili giornalieri

public void setTotalCosts()

determina il valore totale dei costi giornalieri

public float getTotalDailyCosts()

restituisce i costi totali giornalieri

public float getTotalCosts()

restituisce i costi totali

public float getSaleRevenues()

restituisce i profitti di un ordine concluso

public void setTotalSaleRevenues()

determina i profitti totali degli ordini terminati

public float getTotalSaleRevenues()

restituisce i profitti totali

public float getTotalFinishedOrderCosts()

restituisce i costi totalidegli ordini terminati

public void setTotalInventoryFinancialCosts(){

determina i costi finanziari delle scorte considerando un indice stimato su base annua considerando duecento giorni di lavoro nell'anno

totalInventoryFinancial-
Costs+=(myWarehouse.getInventoryQuantity()(vEFrameModel*
Swarm. getInventoryFinancialRate()/200));}

public float getTotalInventoryFinancialCosts()

restituisce il costo finanziario totale delle scorte

public void setStoredComponentDailyValue()

Per ciò che riguarda le scorte di componenti queste sono valutate giornalmente sula base di tre criteri :

1. il primo attribuisce a ciascuna il valore di profitto ottenuto per ogni passo della ricetta;

if(vEFrameModelSwarm.getInventoryEvaluationCriterion()==3)
dailyStoredComponentValue = myWarehou-
*se.getInventoryQuantity()*vEFrameModel*
Swarm.getRevenuePerEachRecipeStep();

2. il secondo attribuisce la somma tra costi fissi e variabili di competenza;

```
else if(vEFrameModelSwarm.getInventoryEvaluationCriterion()==2)
dailyStoredComponentValue = myWarehouse.getInventoryQuantity()*(fixedCosts+ variableCosts);
```

3. la terza attribuisce unicamente i costi variabili di competenza

```
else if(vEFrameModelSwarm.getInventoryEvaluationCriterion()==1)
dailyStoredComponentValue = myWarehouse.getInventoryQuantity()*variableCosts;
```

4. al di là di quei criteri viene comunicato l'errore di impostazione e si ha la chiusura del metodo

```
elseif (vEFrameModelSwarm.getInventoryEvaluationCriterion()!=1&&vEFrameModelSwarm. getInventoryEvaluationCriterion()!=2&& vEFrameModelSwarm.getInventoryEvaluation Criterion() !=3){
System.out.println("your criterion doesn't exist:the variable"+"inventoryEvaluation Criterion must 1, 2 or 3");
System.exit(0);}}

public float getDailyStoredComponentValue()
```

restituisce il costo delle scorte accumulate "quotidianamente".

AccountingParameters.java

La classe *java.io.** mette a disposizione classi utili per la lettura e la scrittura di dati da e su file mediante la tecnica dello streaming su disco; la classe *java.util.** viene richiamata per la presenza al suo interno della classe *StringTonizer* utile per poter distinguere i singoli caratteri di una stringa e poterli così leggere od utilizzare più agevolmente.

public class AccountingParameters extends SwarmObjectImpl

il vettore dei costi fissi per ogni passo o giorno di ogni unità

public float[] fixedCosts;

il vettore dei costi fissi per ogni passo o giorno di ogni unità

public float[] variableCosts;

il numero totale delle unità utilizzato in questo caso per dimensionare i vettori di cui sopra

public int totalUnitNumber;

Il costruttore della classe richiede, oltre alla zona di memoria il numero delle unità presenti

public AccountingParameters(Zone aZone, int un) {

super(aZone);
totalUnitNumber = un; }

Il metodo seguente legge i dati presenti nel file *CostParameters\fixedCosts.txt* e li utilizza al fine del calcolo dei costi fissi:

public void readFixedCosts()

si ricorre all'utilizzo del blocco try{}...catch{} principalmente per evitare che un errore di battitura nel path del file contenente i dati possa bloccare tutta la simulazione rendendo molto colmplesso il debugging

try

occorre anzitutto inizializzare la dimensione del vettore che raccoglierà i dati contenuti nel file *CostParameters\fixedCosts.txt*

fixedCosts = new float[totalUnitNumber];
BufferedReader fileInputFixedCosts = null;

La classe *BufferedReader* velocizza lo streaming dei dati da disco aprendo un contenitore temporaneo in cui i dati vengono depositati a blocchi dopo esser stati letti dal'oggetto della classe *FileReader*

fileInputFixedCosts = new BufferedReader(new FileReader ("CostParameters /fixed Costs.txt"));

La variabile t di tipo StringTokenizer riceverà in input gruppi di caratteri.
La classe *StringTokenizer* ci permette di assegnare a t tutta la riga letta sapendo già che l'oggetto cui fa riferimento *StringTokenizer* sarà in grado di distinguere i caratteri tra uno spazio e l'altro. Successivamente, poiché si tratta di cifre, potremo effettuare parsing per trasformare tale valore in un valore di tipo float da utilizzare per le nostre rilevazioni.
In particolare il file in questione contiene una colonna di soli numeri rappresentativi dei costi fissi delle singole unità.

```
StringTokenizer t;
for(i = 0; i < fixedCosts.length; i++){
String line = fileInputFixedCosts.readLine();
t = new StringTokenizer(line, " ");
fixedCosts[i] = Float.parseFloat(t.nextToken());}
```

Nel caso in cui il nome e/o il percorso del file fosse inesatto verrebbe restituito un messaggio di errore particolare che ci permetterebbe facilmente di individuare il punto in cui si è verificata l'anomalia.

```
catch (FileNotFoundException fnfe){
System.out.println(fnfe);
System.exit(0);}
catch (IOException e){
System.out.println("IOException:" + e.toString());}
```

Identiche considerazioni valgono nel caso della lettura dei dati dal file CostsParameters\variableCosts.txt, contenente i costi variabili.

```
public void readVariableCosts(){
try{
variableCosts = new float[totalUnitNumber];
BufferedReader fileInputVariableCosts = null;
fileInputVariableCosts=new BufferedReader(new FileRea-
der("CostParameters/variableCosts.txt"));
StringTokenizer t;
for(i=0; i<variableCosts.length; i++){
String line = fileInputVariableCosts.readLine();
t = new StringTokenizer(line, " ");
variableCosts[i] = Float.parseFloat(t.nextToken());}}
catch (FileNotFoundException fnfe){
System.out.println(fnfe);
System.exit(0);}
catch (IOException e){
System.out.println("IOException:" + e.toString());}}
```

```
public float getFixedCosts(int i)
```

restituisce il costo fisso di una Unit determinata il cui costo fisso è registrato in maniera ordinata nel vettore *fixedCosts[]*

```
public float getVariableCosts(int i)
```

restituisce il costo variabile di una Unit determinata il cui costo fisso è registrato in maniera ordinata nel vettore *variableCosts[]*

Warehouse.java

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

```
public class Warehouse extends SwarmObjectImpl
```

```
public int warehouseNumber;
```

il numero di riferimento del magazzino

```
public int inventoryCounter;
```

il contatore della quantità di scorte
Il costruttore della classe richiede, oltre alla zona di memoria di competenza, il numero identificativo del magazzino

```
public Warehouse (Zone aZone, int n) {
super(aZone);
warehouseNumber=n;
inventoryCounter = 0;
if(StartVEFrame.verbose)
System.out.println("Warehouse number " + warehouseNumber + "
has been created."); }
```

```
public int getInventoryQuantity()
```

restituisce la quantità di pezzi presenti nel magazzino

public int getWarehouseNumber()

restituisce il numero identificativo del magazzino

public void increaseInventoryCounterValue()

serve per incrementare le scorte presenti nel magazzino

public int decreaseInventoryCounterValue()

qualora il magazzino fosse pieno viene decrementato di una unità e viene restituito 1, altrimenti viene restituito 0

public boolean empty()

si controlla se il magazzino è vuoto, nel qual caso viene restituito il valore true, altrimenti si restituisce false

InventoryRuleMaster.java

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

public class InventoryRuleMaster extends SwarmObjectImpl

public int unitNumber;

il numero dell'unità che vuole comunicare con l'InformationRuleMaster

public int maxInWarehouses;

la quantità massima di scorte che possono essere tenute in magazzino

public int minInWarehouses;

la quantità minima di scorte che devono essere tenute in magazzino

public Warehouse theUnitWarehouse;

il magazzino dell'Unità che vuole comunicare con l'InfomationRuleMaser

public int nOfNewsesToProduce;

il numero di News necessarie per stabilire che sia opportuno produrre per il magazzino

public int nOfNewsesToBeCleared;

il numero di News che devono essere cancellate, dopo aver preso la decisione di produrre per incrementare le scorte di magazzino

public ListImpl newsList;

la lista delle news

public News toBeClearedNews;

una news non ancora cancellata dall'InventoryRuleMaster
Il costruttore per questa classe richiede che vengano passati in fase di creazione di un oggetto del tipo di quelli definiti dalla stessa: la zona sdi memoria cui far riferimento, il massimo numero di scorte presenti in magazzino, il minimo numero di scorte, il numero di news da produrre ed il numero di News che devono essere cancellate

```
public InventoryRuleMaster (Zone aZone, int max, int min, int nop,
int noc){
super(aZone);
maxInWarehouses=max;
minInWarehouses=min;
nOfNewsesToProduce=nop;
nOfNewsesToBeCleared=noc; }
```

Il metodo seguente risponde alla chiamata della Unit, la quale vuole sapere se dovrà produrre per il magazzino.

public boolean increaseInventories(int un, Warehouse aw)

La risposta sarà true se il magazzino contiene meno scorte del livello massimo, mentre sarà false nel caso contrario.

```
unitNumber=un;
theUnitWarehouse=aw;
```

```
if(StartVEFrame.verbose)
System.out.println("* Unit #" + unitNumber + " is calling the inventoryRuleMaster");
if(StartVEFrame.verbose)
System.out.println("* Warehouse #" +theUnitWarehouse.getWarehouse Number() + " contains " + theUnitWarehouse.getInventoryQuantity() +" inventory units");
if (theUnitWarehouse.getInventoryQuantity()<maxInWarehouses){
if(StartVEFrame.verbose)
System.out.println("* Unit #" + unitNumber +" has to increase inventories");
return true;}
else if(StartVEFrame.verbose)
System.out.println("* The warehouse of unit #" + unitNumber + " is already full.");
return false;}
```

Il metodo seguente risponde alla chiamata della Unit, la quale vuole sapere se dovrà produrre per il magazzino. A differenza del caso precedente, questo contempla anche l'utilizzo delle News.

```
public boolean increaseInventoriesUsingNewses(int un, Warehouse aw, ListImpl nl)
```

Se il livello delle scorte è inferiore al massimo ed è presente una News che segnala l'arrivo di un passo di produzione, oppure se il livello delle scorte è inferiore al minimo occorrerà produrre per il magazzino.

```
int i;
int count;
int countCleared;
newsList= nl;
unitNumber=un;
theUnitWarehouse=aw;
if(StartVEFrame.verbose)
System.out.println("* Unit #" + unitNumber + " is calling the inventoryRuleMaster using newses");
if(StartVEFrame.verbose)
System.out.println("* Warehouse #" + theUnitWarehouse.getWarehouseNumber() + " contains " + theUnitWarehouse.getInventoryQuantity() + " inventory units");
count=0; countCleared=0;
for (i=0;i<newsList.getCount();i++){
toBeClearedNews = (News) newsList.atOffset(i);
if(toBeClearedNews.getNotCleared()){
count++; countCleared++;
if(countCleared<=nOfNewsesToBeCleared)
toBeClearedNews.setClearedByInventoryRuleMaster();
if(StartVEFrame.verbose)
System.out.println("the InventoryRuleMaster " + "is clearing a news in unit " + unitNumber);}}
if(count >= nOfNewsesToProduce && theUnitWarehouse. getInventoryQuantity() < maxInWarehouses){
if(StartVEFrame.verbose)
System.out.println("* Unit #" + unitNumber + " is increasing inventories via newses");
```

```
return true;}
else if(count < nOfNewsesToProduce && theUnitWarehouse.getInventoryQuantity() < minInWarehouses){
if(StartVEFrame.verbose)
System.out.println("* Unit #" + unitNumber + " is increasing inventories via minInWarehouses");
return true;}
else{
if(StartVEFrame.verbose)
System.out.println("* The warehouse of unit #" + unitNumber + " is already full or newsList is empty" + " and we are over minInWarehouses quantity");
return false;}}}
```

News.java

La classe News.java contiene la descrizione di un oggetto necessario per trasmettere le informazioni (News appunto) riguardanti i passi successivi nella sequenza di produzione di un Order

```
public class News extends SwarmObjectImpl{

public int sendingUnitNumber;
```

il numero della Unit che manda la News

```
public int receivingUnitNumber;
```

il numero della Unit che riceve la News

public Order incomingOrderInNews;

l'Order annunciato nella News

boolean notClearedByInventoryRuleMaster;

un segnale indicante la possibilità che sia L'InventoryRuleMaster a cancellare la News

public int numberOfTheStepForWhichANewsIsSent;

il passo in cui la News è mandata (può essere mandata a due o più passi successivi)

Il costruttore della classe necessita oltre alla zona di memoria, anche del numero di unità che manda la News, il numero di chi la riceve ed il numero dell'Order

```
public News (Zone aZone, int su, int ru, Order or){
super(aZone);
int i;
sendingUnitNumber=su;
receivingUnitNumber=ru;
incomingOrderInNews= or;
notClearedByInventoryRuleMaster=true;
numberOfTheStepForWhichANewsIs-
Sent=incomingOrderInNews.getNumberOfTheStepFor WhichA-
NewsIsSent();
if(stasIsSent()
if(Sent.Ferrume.verbose)
```

```
System.out.println("News: unit# ° " + sendingUnitNumber + " is an-
nouncing to unit #  " + receivingUnitNumber+ " order #  " + incomin-
gOrderInNews.getOrderNumber() + "with recipe and state:");
if(StartVEFrame.verbose){
for (i=1;i<=incomingOrderInNews.getStepNumber();i++)
System.out.print(incomingOrderInNews.orderRecipe[i-1] + " ");
System.out.println();
for(i=1;i<=incomingOrderInNews.stepNumber;i++)
System.out.print(incomingOrderInNews.orderState[i-1] + " ");}}

public void setClearedByInventoryRuleMaster()
```

Il metodo stabilisce che sia l'InventoryRuleMaster a cancellare le News nella lista presso la Unit

```
public boolean getNotCleared()
```

restituisce il valore appena impostato

```
public int getNumberOfTheStepForWhichANewsIsSent ()
```

restituisce il numero sequenziale della ricetta in riferimento alla quale la News è stata mandata

InformationRuleMaster.java

La classe InformationRuleMaster è utilizzata per gestire il flusso dell'informazione all'interno della impresa virtuale.

public class InformationRuleMaster extends SwarmObjectImpl

public int sendingUnit;

il numero di Unit che vuole comunicare con l'InformationRuleMaster

public int receivingUnit;

il numero della Unit alla quale abbiamo mandato una News

public int [][] informationFlowMatrix;

la matrice contenente le regole di gestione del flusso di informazione
Il costruttore della classe richiede oltre alla zona di memoria, anche il numero di unità

```
public InformationRuleMaster (Zone aZone, int un){
super(aZone);
int i;
int j;
int unitNumber = un;
String line=null;
BufferedReader data = null;
informationFlowMatrix = new int[unitNumber][unitNumber];
```

Le informazioni contenute nel file *informationFlowMatrix.txt* sono lette ed assegnate ai singoli elementi della matrice *informationFlowMatrix[][]*. I dati sono raccolti come descritto per la classe *AccountingParameters.java*

e sono rappresentati da valori 1 (il flusso dell'informazione scorre) oppure da valori 0 (l'informazione si ferma).
Nuovamente in questo caso si utilizza il blocco try…catch

```
try{
data = new BufferedReader(new FileReader("informationFlowMatrix.txt"));}
catch (FileNotFoundException fnfe){
System.out.println(fnfe);
System.exit(0);}
if(StartVEFrame.verbose)
System.out.println("informationFlowMatrix " + unitNumber + " x " + unitNumber);
for (i=0;i<unitNumber;i++){
try{
line=data.readLine();}
catch (IOException ioe) {
System.out.println(ioe);
System.exit(0);}
for (j=0;j<unitNumber;j++){
informationFlowMatrix [i][j] = Integer.parseInt(line.substring(j,j+1));
if(StartVEFrame.verbose)
System.out.print(+informationFlowMatrix[i][j]+" ");}
if(StartVEFrame.verbose)System.out.println();}}

public boolean checkToSendNews(int s, int r)
```

la Unit che manda la News controlla se il flusso dell'informazione fino alla Unit ricevente è libero oppure no

```
sendingUnit=s;
receivingUnit=r;
if(StartVEFrame.verbose)
System.out.println(" the unit # " + sendingUnit + " is checking the informationRuleMaster");
if (informationFlowMatrix[sendingUnit-1][receivingUnit-1]==1){
if(StartVEFrame.verbose)
System.out.println("the unit # " + sendingUnit + " can send a news to the unit # " + receivingUnit + " (informationFlowMatrix value= " + informationFlowMatrix[sendingUnit-1][receivingUnit-1] + ")");
return true; }
else{
if(StartVEFrame.verbose)
System.out.println("the unit # " + sendingUnit + " cannot send a news to unit # " + receivingUnit + " (informationFlowMatrix value= " + informationFlowMatrix[sendingUnit-1][receivingUnit-1] + ")");
return false; }}}
```

SwarmUtils.java

Questo file è un estratto da:

jSIMPLEBUG: a Swarm tutorial for Java Charles P. Staelin, Smith College Northampton, MA April 2000

…based on ObjectiveC code and original text by Chris Langton & Swarm development team Santa Fe Institute, NM.

public class SwarmUtils

I due metodi seguenti creano un Selector, come già accennato si tratta di un oggetto usato per incapsulare messaggi da mandare ad altri oggetti. In questo caso il messaggio è composto dagli argomenti da passare al costruttore *getSelector().* Essi saranno: il nome della classe cui far riferimento, in parte dal metodo, della stessa,che ci interessa richiamare.
Il metodo getSelector() è un metodo sovraccarico, infatti può essere richiamato passando, come primo argomento, sia una stringa contenente il nome della classe, sia un oggetto della classe desiderata.
Il valore boolean, presente nell'argomento dell'oggetto Selector, indicala possibilità di usare la sintassi di tipoObjectiveC, se impostato su true. Indica, invece, la possibilità di usare la sintassi Java, se impostato su false. *(Charles P. Staelin)*

```
public static Selector getSelector(String name, String method){
Selector sel;
try{
sel = new Selector(Class.forName(name), method, false); }
catch (Exception e){
System.err.println("There was an error in creating a Selector for
method " + method + "\nin Class " + name + ".");
System.err.println (name + "." + method + " returns " +
e.getMessage());
System.err.println("The process will be terminated.");
System.exit(1);
return null;}
return sel;}
```

```
public static Selector getSelector(Object obj, String method){
Selector sel;
try{
sel = new Selector(obj.getClass(), method, false);}
catch (Exception e){
System.err.println("There was an error in creating a Selector for
method " + method + "\nin Class " + (obj.getClass()).getName() +
".");
System.err.println ((obj.getClass()).getName() + "." + method + " re-
turns " + e.getMessage());
System.err.println("The process will be terminated.");
System.exit(1);
return null;}
return sel;}}
```

PTHistogram.java

La classe PTHistogram. Java contiene tre costruttori per disegnare i grafici degli istogrammi necessari per mostrare sia le code di produzione presso le singole Unit sia il contenuto dei magazzini. Il primo costruttore considera la versione semplice del modello di impresa virtuale Warehouse; il secondo si attiva nel caso in cui anche i Warehouse siano attivi ed il terzo è stato progettato in previsione di un futuro sviluppo del modello *(Landini 2000).*

Bibliografia

Terna, P. (2001), *jVEFrame – Java Virtual Enterprise: un frame in JavaSwarm.*

Deitel, H. M. e Deitel, P. (2000), *Java. Fondamenti di programmazione.* Milano, Apogeo S.r.l.

Eckel, B. (2000), *Thinking in Java.* www.BruceEckel.com

Staelin, C. P. (2000), *jSimpleBug: a Swarm tutorial for Java.* Smith College, Northampton, MA