Swarm e jESLet con Eclipse

Università degli Studi di Bologna
Facoltà di Scienze MM. FF. NN.
Corso di Laurea Triennale in Scienze di Internet
Anno Accademico 2003-2004

Laboratorio di Sistemi e Processi Organizzativi

ECLIPSE + SWARM

- Mostriamo come è possibile scrivere un programma con Swarm nell'ambiente di sviluppo eclipseUML.
- Il progetto java che creeremo conterrà:
  - una classe Consumer
  - un modelSwarm
  - il file con il metodo main
- Nel modelSwarm sarà creata una istanza della classe Consumer, che deciderà a caso se vuole andare al mercato. In caso affermativo deciderà a caso quanto spendere.

Crazione del progetto

- Creiamo un nuovo progetto java
  - mercatoSemplice
- Importiamo la libreria di swarm
  - /usr/share/swarm/swarm.jar
- Nel workspace, sul progetto creato
  - UML -> nuovo package "mercato"
  - -> open "mercato.ucd"

ModelSwarm Class

- Creiamo la nuova classe ModelSwarm che ha superclasse SwarmImpl.
- Creiamo due nuovi attributi pubblici:
  - modelSchedule di tipo Schedule (swarm.activity.Schedule)
  - modelActions di tipo ActionGroup (swarm.activity.ActionGroup)
- Creiamo un ulteriore attributo pubblico:
  - modelTime di tipo int
  - questo attributo indicherà il tempo simulato nel modello
- aggiungiamo nel costruttore il codice: modelTime=0;

Consumer Class

- Creiamo la nuova classe Consumer che ha superclasse SwarmObjectImpl
- Creiamo i tre attributi:
  - public int myBudget;
  - public int myName;
  - public int moneySpent;
- Facciamo refactoring del costruttore in modo che, oltre il paramtro di tipo Zone, richieda anche:
  - int name;
  - int budget;
- Nel codice del costruttore aggiungiamo:
  - myName=name;
  - myBudget=budget;

ModelSwarm-Consumer

- Nell'esempio che stiamo modellando abbiamo un solo consumatore
- Creiamo un'associazione tra la classe ModelSwarm e la classe Consumer
  - dal lato del consumatore facciamo in modo che l'associazione non sia navigabile (casella Navigable vuota)
  - dal lato del modello lasciamo l'associazione navigabile e
    - chiamiamo l'attributo aConsumer
    - molteplicità 1 (abbiamo un solo consumatore)

buildObjects()

- Nella classe ModelSwarm creiamo il metodo pubblico buildObjects() che ritorna tipo Object
- Nel metodo buildObjects() si creano gli oggetti usati nella simulazione:
  - nell'esempio abbiamo bisogno solo di un consumatore, creato con il suo nome e il suo bilancio.
  - quindi scriviamo il codice:

```
int budget=10;
int name=1;
super.buildObjects();
aConsumer=new Consumer(getZone(),name,budget);
return this;
```

- Il consumatore si chiama quindi aConsumer ed è un'istanza della classe Consumer.
- E' creato usando una zona della memoria del modelSwarm (grazie a getZone()).

Metodi di Consumer

- Nella classe Consumer aggiungiamo
  - import swarm.Globals
- quindi aggiungiamo i tre metodi seguenti:
  - public goToTheMarket(): in cui il consumatore decide se andare o meno al mercato; avrà codice:

```
int k;
k = Globals.env.uniformIntRand.getIntegerWithMin$withMax(0,1);
return k;
```

  - public int spend(): che decide quanto il consumatore spende, avrà codice:

```
moneySpent=Globals.env.uniformIntRand.getIntegerWithMin$withMax(0,myB
  udget);
return moneySpent;
```

  - public int calculateRemainingBudget(): calcola il budget rimasto; avrà codice.
    - myBudget-=moneySpent;
    - return myBudget;

marketDay()

- Nella classe ModelSwarm definiamo il metodo
  - public Object marketDay()
  - che indica il momento in cui il consumatore decide se andare o meno al mercato;
  - avrà codice:

```
int go;
int spending;
go=aConsumer.goToTheMarket();
if (go==1) {
    spending=aConsumer.spend();
    System.out.println("This is time "+modelTime);
    System.out.println("I am consumer " + aConsumer.getMyName() + ", I went to the market and spent " +
    spending +".");
    System.out.println("I have " + aConsumer.calculateRemainingBudget() + " of currency left.");
} else {
    System.out.println("This is time "+modelTime);
    System.out.println("I am consumer " + aConsumer.getMyName() + ", I did not go to the market.");
    System.out.println("I have " + aConsumer.getMyBudget() + " of currency left.");
}
return this;
```


buildActions()

- Nella classe ModelSwarm creiamo il metodo
  - public Object buildActions()
  - crea un gruppo di azioni. In generale nell'actionGroup troviamo tutte le azioni che si eseguono simultaneamente nella simulazione;
  - un'istanza della classe Schedule (modelSchedule) in cui risiede la tabella dei tempi in cui sono inseriti tutti gli actionGroup.

buildActions() code

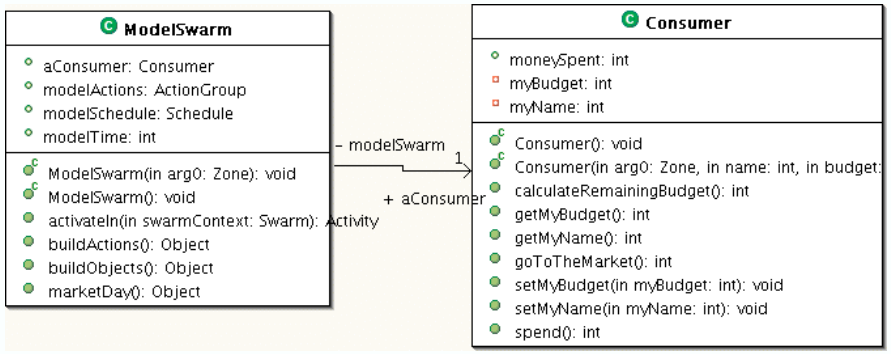
```
modelActions=new ActionGroupImpl(getZone());
try {
modelActions.createActionTo$message(this,new
  Selector(getClass(),"marketDay",false));
} catch (Exception e) {
e.printStackTrace(System.err);
System.exit(1);
}


modelSchedule=new ScheduleImpl(getZone());
modelSchedule.at$createAction(0,modelActions);
return this;
```


activateIn

- Nella classe ModelSwarm creiamo il metodo
  - public Activity activateIn(Swarm swarmContext)
  - necessario per poter eseguire la simulazione;
  - segue la struttura:
    - attiva la classe madre del modelSwarm
    - attiva lo schedule del modelSwarm
    - ritorna l'attività del modelSwarm
  - avrà codice:

```
super.activateIn(swarmContext);
modelSchedule.activateIn(this);
return this.getActivity();
```

Class Diagram



StartMarket

- Creiamo una nuova classe StartMarket con metodo main in cui
  - dichiariamo il modello:
    - ModelSwarm modelSwarm;
  - inizializziamo lo swarm:
    - Globals.env.initSwarm ("market","2.2","abologne@cs.unibo.it",args);
  - creiamo il modelSwarm
    - modelSwarm=new ModelSwarm(Globals.env.globalZone);

Run StartMarket

- impostiamo il seme generatore per i numeri casuali

```
Globals.env.randomGenerator.setStateFromSeed(934850934);
```

- facciamo partire la simulazione dopo aver creato gli oggetti e le azioni

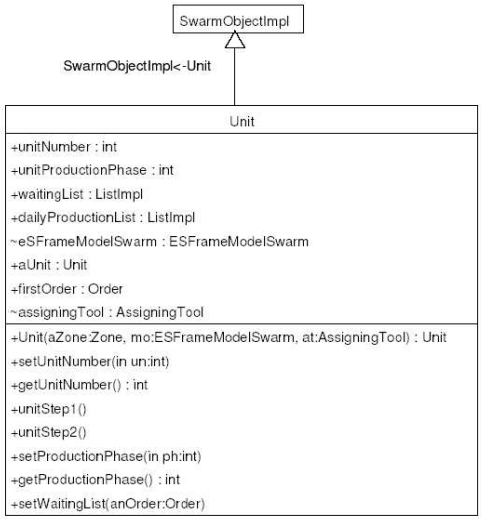
```
modelSwarm.buildObjects();
modelSwarm.buildActions();
modelSwarm.activateIn(null);
(modelSwarm.getActivity()).run();
```

- Per far eseguire la simulazione:
  - Dal menu Run scegliamo Run..,
  - Java Application -> New -> Run

Importare jESLet in Eclipse

- Creare un nuovo progetto java
- Importare le librerie
  - swarm.jar (/usr/share/swarm/swarm.jar)
- Nella cartella src importare da file la cartella
  - jeslet-0.1/src
  - il file jeslet.scm deve risiedere nella root del progetto
- Nel progetto creare una cartella chiamata unitData e importare il file
  - jeslet-0.1/unitData/unitData.txt

Unit Class diagram



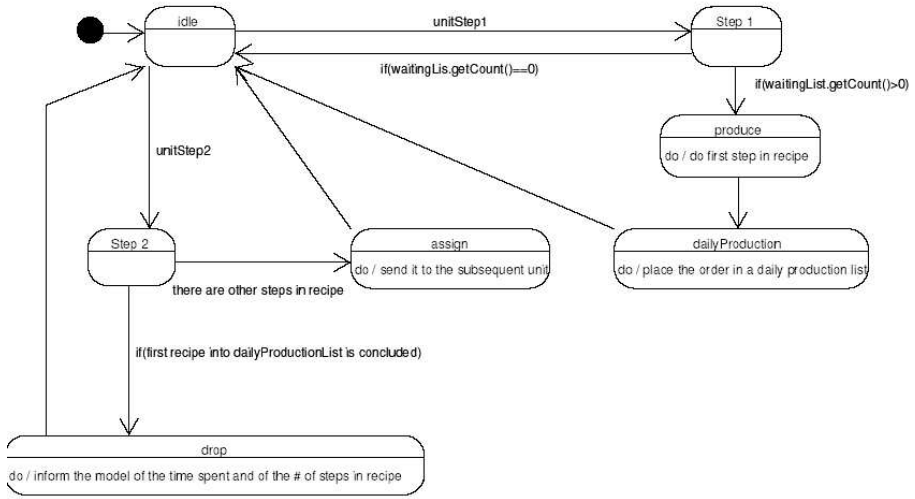
Le Unità

- Un'unità è identificata dal numero univoco
  - unitNumber
- Ed è indicata dalla fase produttiva che compie
  - unitProductionPhase
- Il vettore waitingList contiene gli ordini che devono essere elaborati
- Il vettore dailyProductionList contiene gli ordini eseguiti in un giorno

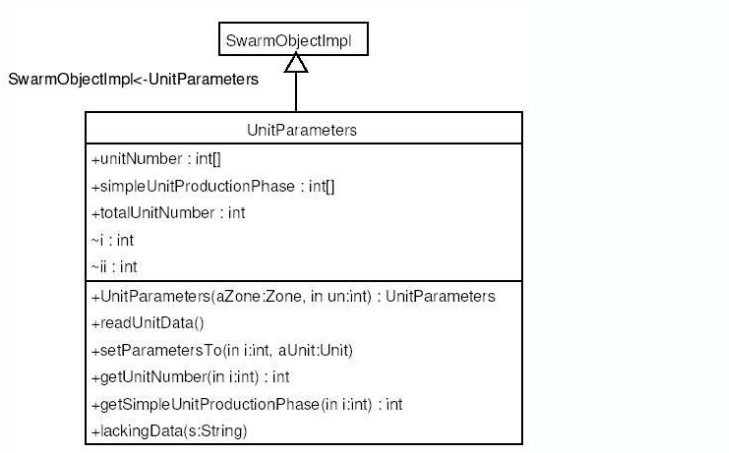
Le Unità

- unitNumber e ProductionPhase sono incapsulati nei rispettivi metodi set e get;
- il metodo setWaitingList ha come parametro un oggetto anOrder di tipo Order
  - inserisce l'oggetto anOrder nella lista waitingList
  - segue un criterio FIFO
- I metodi unitStep1() e unitStep2() sono i metodi principali che caratterizzano la classe Unit

Statechart per le Unità



UnitParameters Class diagram



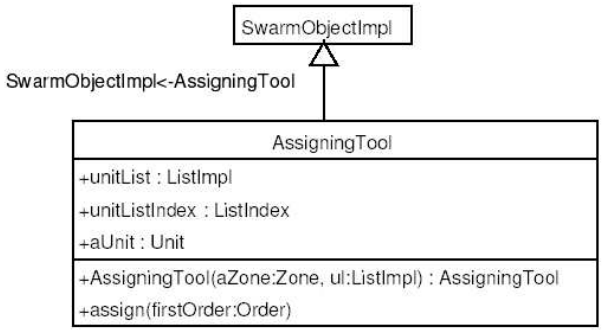
UnitParameters

- Il metodo readUnitData() legge dal file unitData/unitBasicData.txt e creadue vettori
  - int[0..totalUnitNumber-1] unitNumber
  - int[] simpleUnitProductionPhase
- Il metodo getUnitNumber(int i) ritorna il numero dell'unita che ha posizione i-esima nell'enumerazione delle unità produttive
- Il metodo setParametersTo(int i, Unit aUnit) imposta i parametri dell'agente unità aUnit presente all'i-esima posizione nel vettore unitNumber

unitData.txt

- In ogni riga contine
  - il numero identificativo dell'Unità di produzione
  - il numero che indica la sua capacità produttiva
- In jESLet
  - per semplicità, il numero identificativo dell'Unità e quello che indica la sua produzione sono gli stessi
  - sono riportate 10 unità, se è necessario simulare più di 10 unità si devono aggiungere altre righe di codice
  - se più di una unità ha la stessa capacità produttiva, solo la prima viene usata

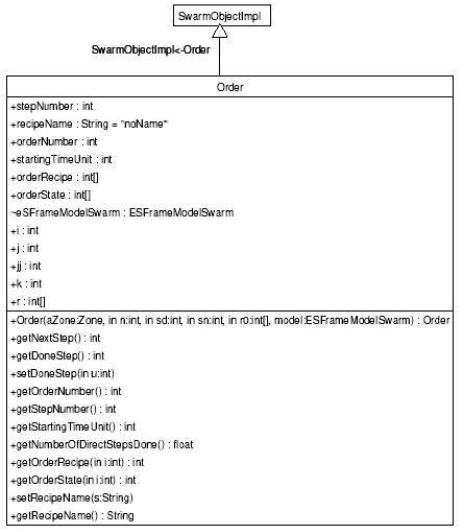
AssigningTool Class diagram



AssigningTool

- Il vettore unitList rappresenta la lista delle unità di produzione normali esistenti nel modello
- Il metodo assign(Order firstOrder) cerca tra le unità quella che ha fase di produzione uguale al prossimo passo nella ricetta di firstOrder

Order Class diagram



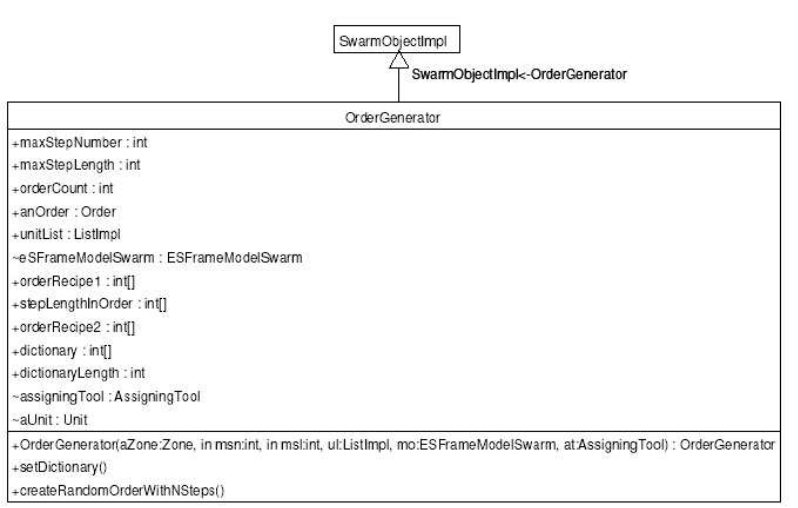
Order Class

- Le istanze della classe Order sono gli oggetti in cui vengono indicati i passi per completare degli ordini
- Il vettore di interi orderRecipe contiene i passi della ricetta rappresentata dall'oggetto order
- Il vettore orderState indica i passi compiuti nella ricetta
- Il metodo getNextStep() restituisce il passo successivo da eseguire nella ricetta

Order Class

- Il metodo getDoneStep() ritorna l'indice relativo all'ultimo passo di produzione compiuto:
  - se l'ordine è finito ritorna l'ultimo passo della ricetta
  - se non è stato compiuto alcun passo ritorna il valore -999999
- Il metodo setDoneStep(int u) cerca il primo passo non compiuto e lo imposta al valore u dell'unità.
- Il metodo getNumberOfStepsDone() ritorna il numero di passi compiuti

OrderGenerator Class diagram



OrderGenerator Class

- orderRecipe1:
  - vettore di interi che contiene una specifica ricetta di produzione in una forma che non comprende i tempi necessari per la produzione di ciascun passo
- stepLengthInOrder:
  - indica i tempi di produzione per ciascun passo
- orderRecipe2:
  - rappresenta la ricetta di produzione in cui sono esplicitati i tempi di produzione di ciascun passo

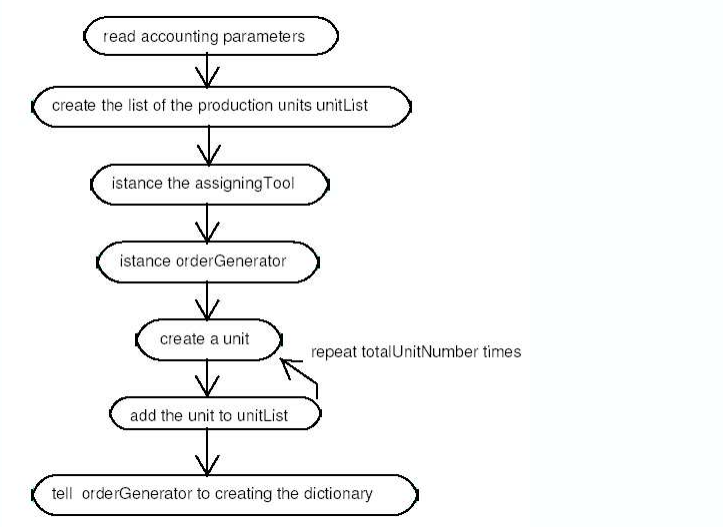
orderRecipe example

- orderRecipe1 = {1, 12, 7, 3}
- stepLengthInOrder = {1, 3, 2, 2}
- orderRecipe2 = {1, 12, 12, 12, 7, 7, 3, 3}

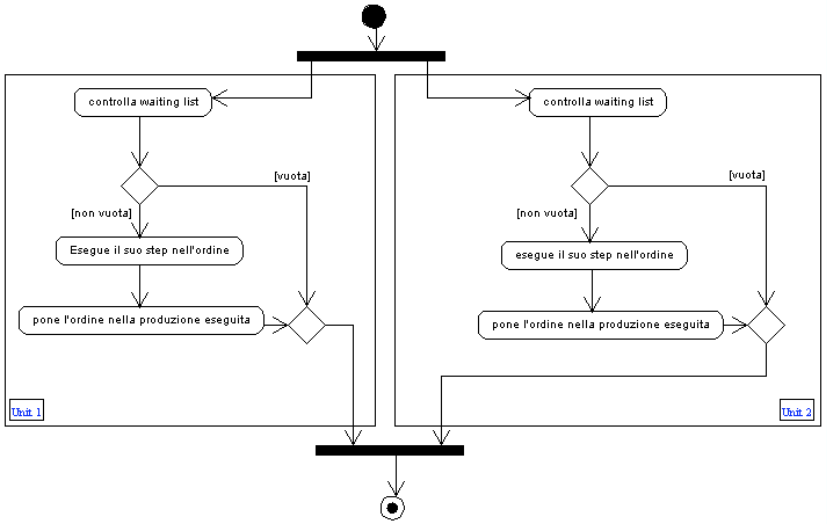
OrderGenerator Class

- setDictionary()
  - crea il vettore dictionary
  - per ogni unità listata in unitList aggiunge a dictionary un'entrata contenente un intero che rappresenta la fase di produzione di quella unità
- createRandomOrderWithNSteps()
  - sceglie casualmente la lunghezza della ricetta
  - riempie ogni step della ricetta con un elemento di dictionary e imposta casualmente la durata degli step.
  - Crea l'ordine e lo assegna all'unita di produzione tramite assigningTool

buildObjects() Activity diagram



Activity diagram unitStep1



Modellare con UML

- Per disegnare grafici UML potete usare
  - An Interaction Diagram Tool by Winkoff
    - http://goanna.cs.rmit.edu.au/~winikoff/**auml**/index.html
  - Eclipse
  - argoUML
  - Umbrello
  - ...