

Swarm e Eclipse

Università degli Studi di Bologna
Facoltà di Scienze MM. FF. NN.
Corso di Laurea in Scienze di Internet
Anno Accademico 2004-2005

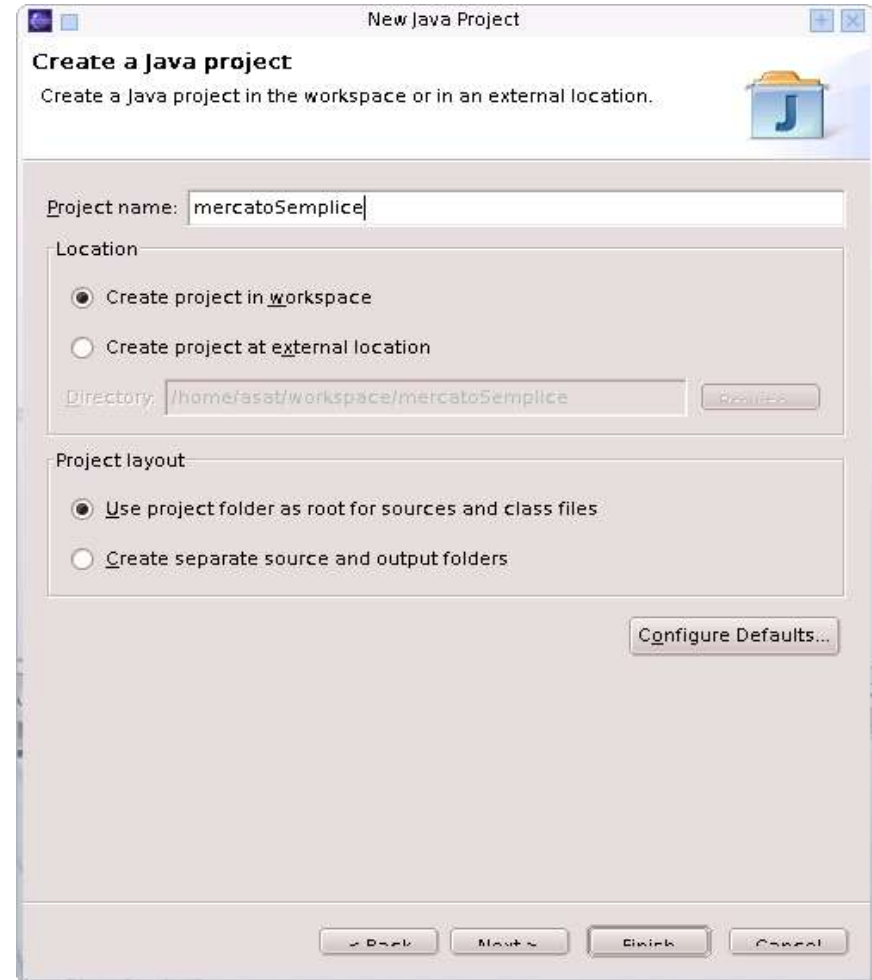
Laboratorio di Sistemi e Processi Organizzativi

ECLIPSE + SWARM

- Mostriamo come è possibile scrivere un programma con **Swarm** nell'ambiente di sviluppo **eclipse+UML**.
- 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*
- *come illustrato nella figura a destra*



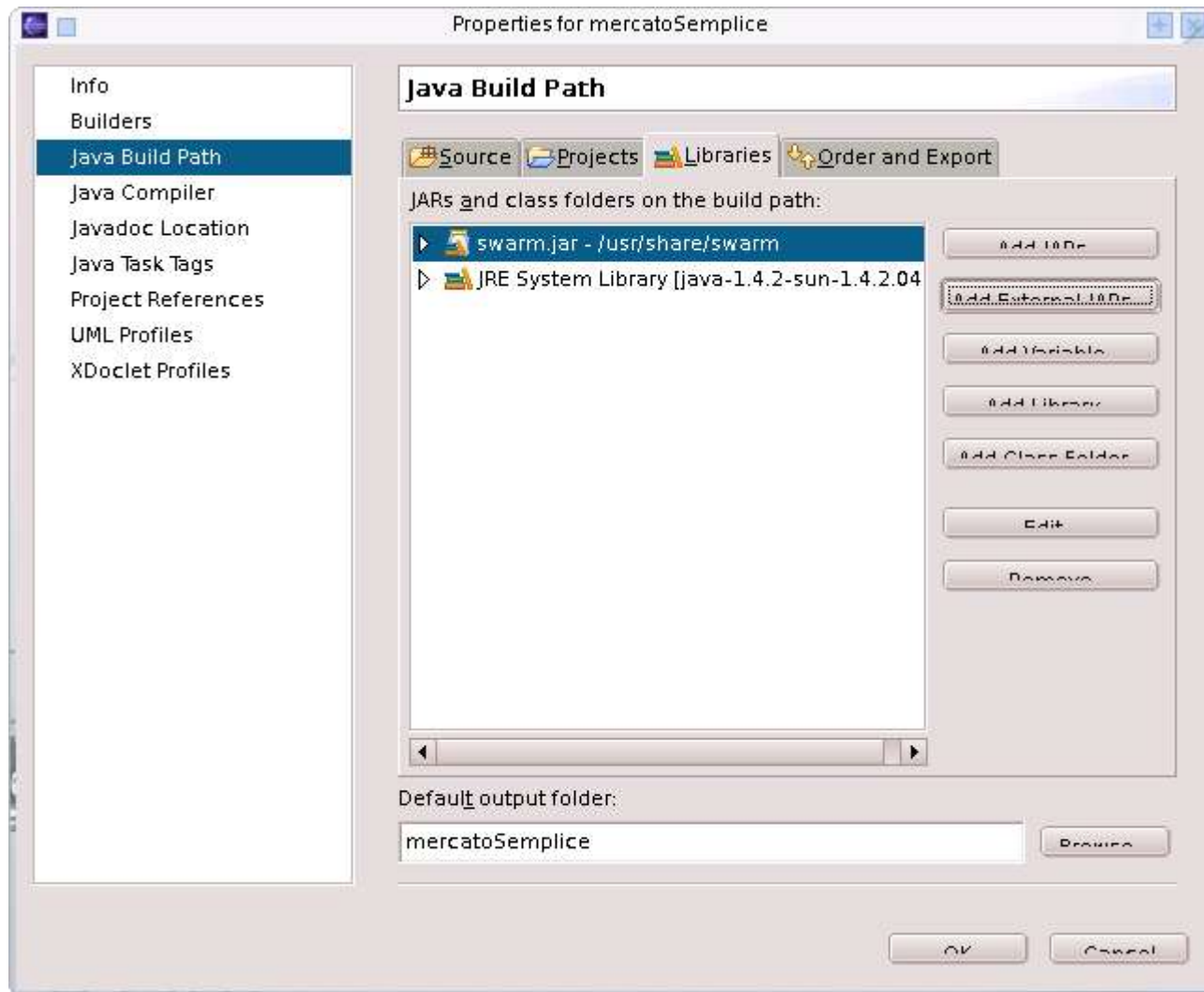
Crazione del progetto

- Importiamo la libreria di swarm **swarm.jar**
 - Nel workspace fate click con il tasto destro del mouse sulla voce *mercatoSemplice* ➡ *properties*
 - Quindi sotto la voce *Java Build Path* nel tab *Libraries* ➡ *Add External JAR* e scegliete:

`/usr/share/swarm/swarm.jar`

- *come mostrato nella slide successiva*

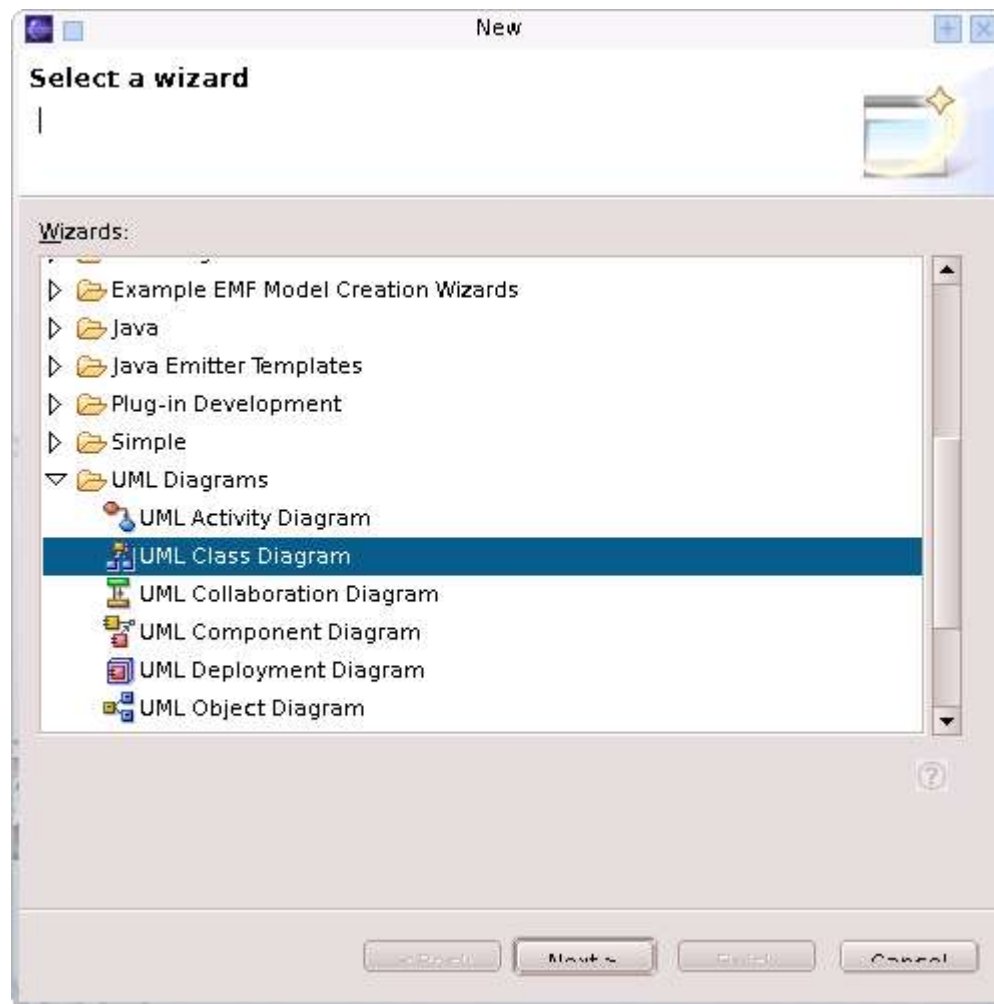
Crazione del progetto



Crazione del progetto

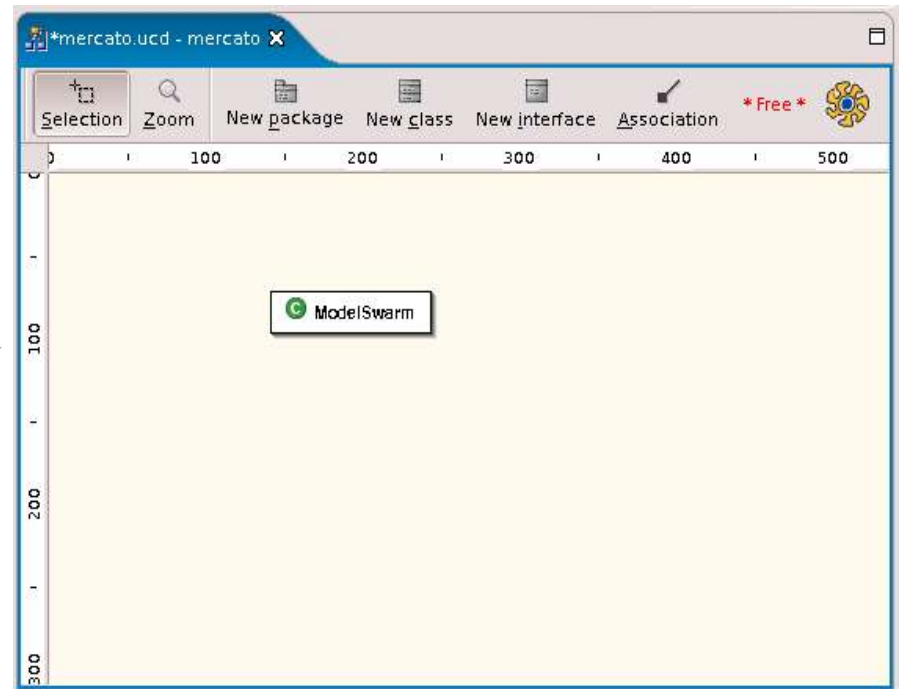
- Nel workspace, sul progetto creato
 - *mercatoSemplice* ➡ *new* ➡ *Package*
 - Nella voce *Name* indicate “*mercato*”
- Nel workspace, sul package *mercato*
 - *mercato* ➡ *new* ➡ *Other*
 - Sotto la voce *UML Diagrams* scegliete:
UML Class Diagram
come indicato nella slide successiva
- Vi sarà quindi suggerito il nome ***mercato.ucd*** per il diagramma che state creando

Crazione del progetto





ModelSwarm Class

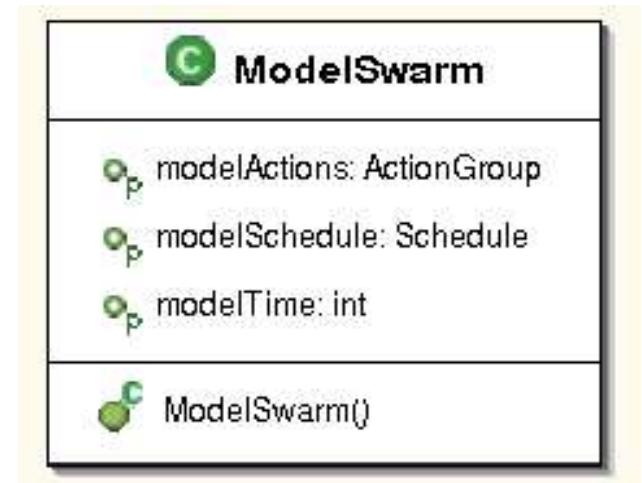
- Creiamo la nuova classe *ModelSwarm* che ha superclasse *SwarmImpl*.



ModelSwarm Class

- Con il tasto destro sulla *classe ModelSwarm* appena creata,  *New*  *Attribute*, 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
```



ModelSwarm Class

- Noterete che il *debugger* di **Eclipse** vi segnala un errore:




L'attributo chiamato *modelTime* nel diagrama UML è stato implementato con il nome *time*

- Correggiamo l'errore, ad esempio scrivendo: *setModelTime(0)*;

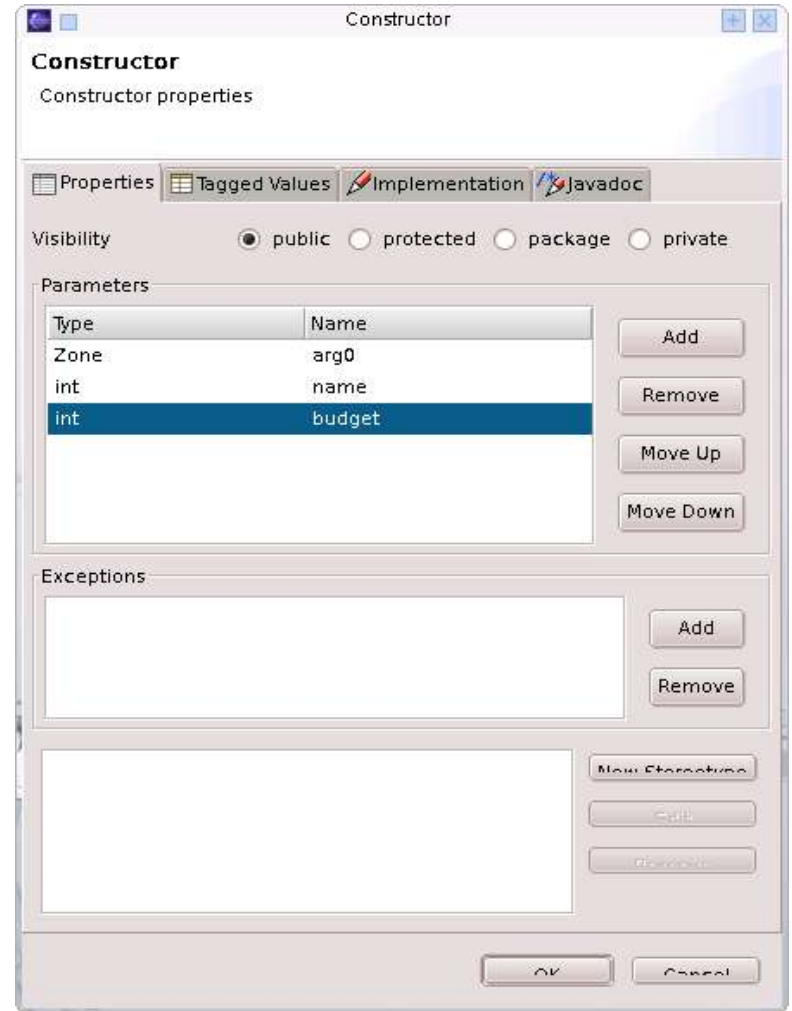
ottenendo quindi:

```
public ModelSwarm(Zone aZone) {
    super(aZone);
    setModelTime(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*;
- Con il click destro su *Consumer()*  *Properties*, facciamo **refactoring** del costruttore (cliccando sul pulsante *add*) in modo che, oltre il parametro di tipo *Zone*, richieda anche:
 - int *name*;
 - int *budget*;
- *Come mostrato nella slide seguente*

Consumer Class



Consumer Class

- Vogliamo aggiungere al *costruttore* le seguenti righe:
 - `myName=name;`
 - `myBudget=budget;`
- Ma otterremmo l'errore visto nella slide precedente.
- Aggiungiamo quindi al *costruttore* le righe seguenti:
 - `this.name=name;`
 - `this.budget=budget;`
- Essendo *name* e *budget* i nomi implementati nel codice a partire rispettivamente dai nomi UML *myName* e *myBudget*

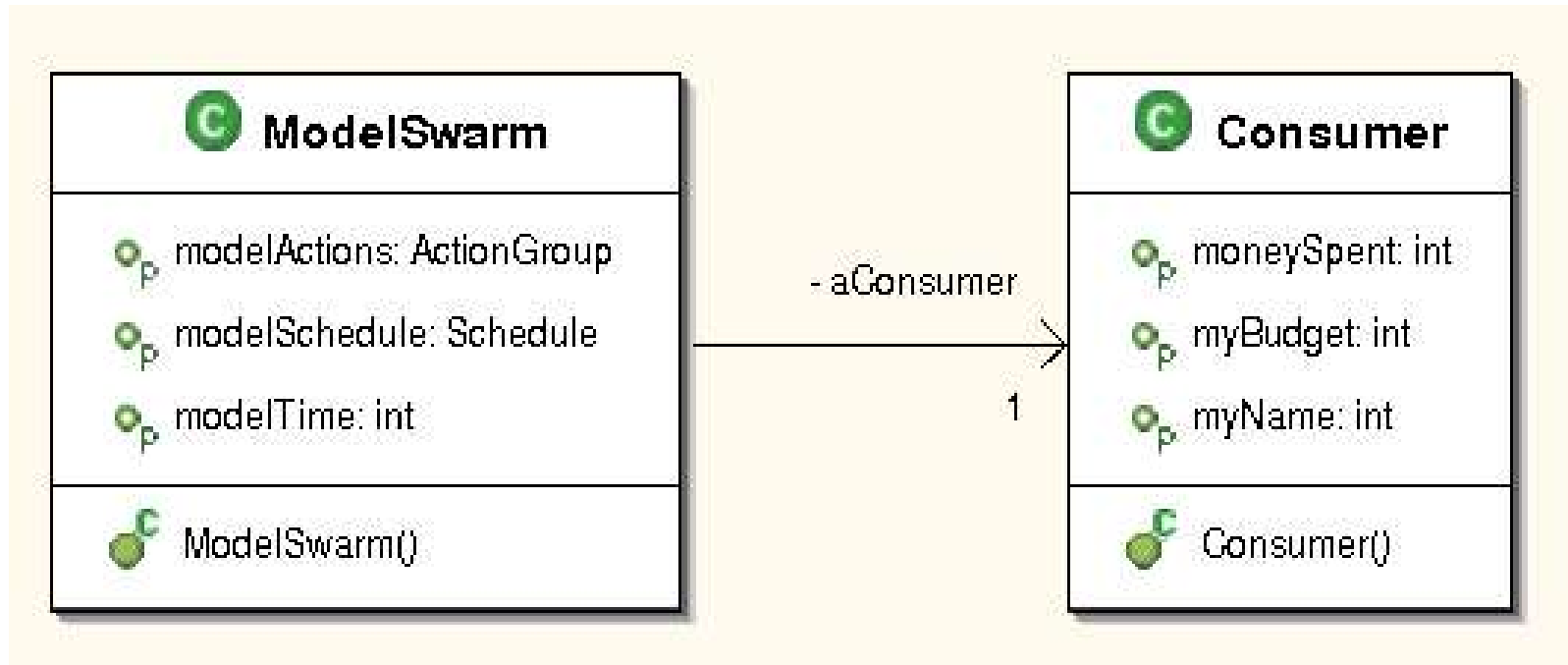
ModelSwarm-Consumer

- 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)

ModelSwarm-Consumer

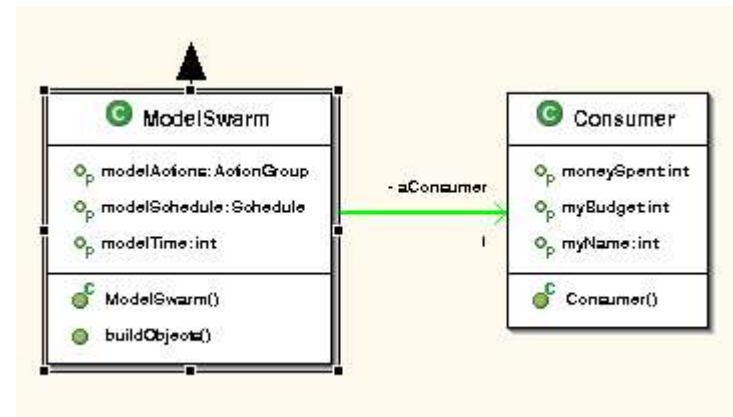
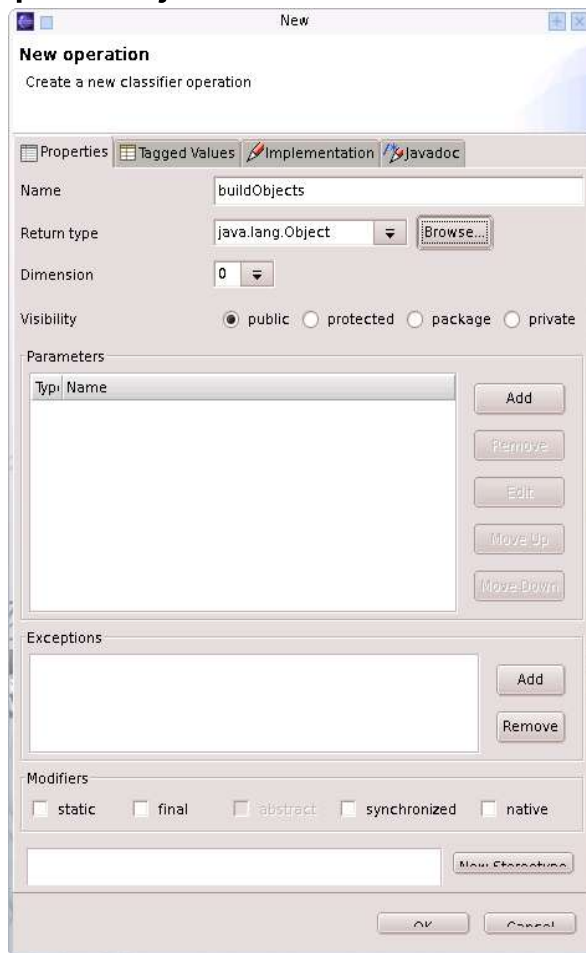
The screenshot shows the 'Association' dialog box in Eclipse. The title bar reads 'Association'. Below the title bar, there is a 'Properties' section with the instruction 'Set the association properties'. A tabbed interface at the top includes 'Properties', 'Tagged Values', '1st Association End', '2nd Association End', and 'Router'. The '1st Association End' tab is active. The 'Navigable' checkbox is checked. The 'Name' field contains 'aConsumer'. Below the name field is a large empty text area with 'New Stereotype', 'Call...', and 'Reset...' buttons. The 'Type' field contains 'mercato.Consumer' with a 'Browse...' button. The 'Dimension' dropdown is set to '0'. The 'Multiplicity' dropdown is set to '1'. The 'Ordered' checkbox is unchecked. The 'Association type' dropdown is set to 'None'. The 'Qualifier' section has a checkbox that is unchecked, with a text field containing 'java.util.Map' and a 'Browse...' button. Below it, the 'Attribute' section has a text field containing 'Object key' and a 'Browse...' button. The 'Attribute' section includes a 'Visibility' group with radio buttons for 'public', 'private' (selected), 'protected', and 'package'. There is an 'Initial value' text field. The 'Modifiers' section has checkboxes for 'static', 'final', 'abstract', 'transient', and 'volatile', all of which are unchecked. At the bottom, there are 'OK' and 'Cancel' buttons.

ModelSwarm-Consumer



buildObjects()

- Nella classe ModelSwarm creiamo il metodo pubblico buildObjects() che ritorna tipo Object



buildObjects()

- 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*.
- È 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 int 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,budget);  
return moneySpent;
```

- `public int calculateRemainingBudget():` calcola il budget rimasto; avrà codice.

```
budget-=moneySpent;  
return budget;
```

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 "+time);
    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 "+time);
    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.

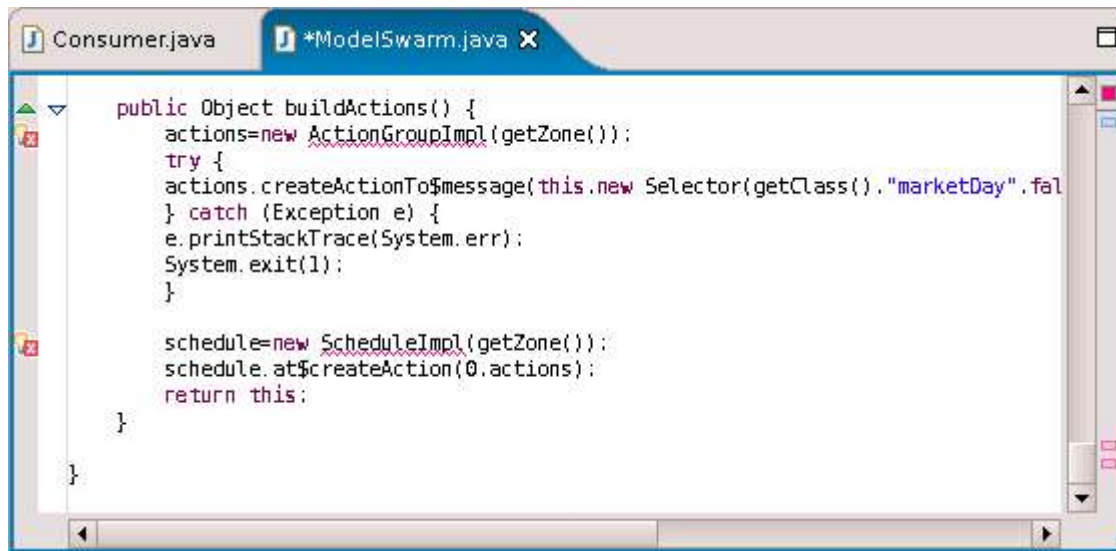
buildActions() code

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

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

buildActions()

- Una volta scritto il codice vi accorgete che il *debugger* di *Eclipse* vi indica alcuni errori:



```
Consumer.java  *ModelSwarm.java x
public Object buildActions() {
    actions=new ActionGroupImpl(getZone());
    try {
        actions.createActionTo$message(this.new Selector(getClass().`marketDay`.fal
    } catch (Exception e) {
        e.printStackTrace(System.err);
        System.exit(1);
    }

    schedule=new ScheduleImpl(getZone());
    schedule.at$createAction(0.actions);
    return this;
}
}
```

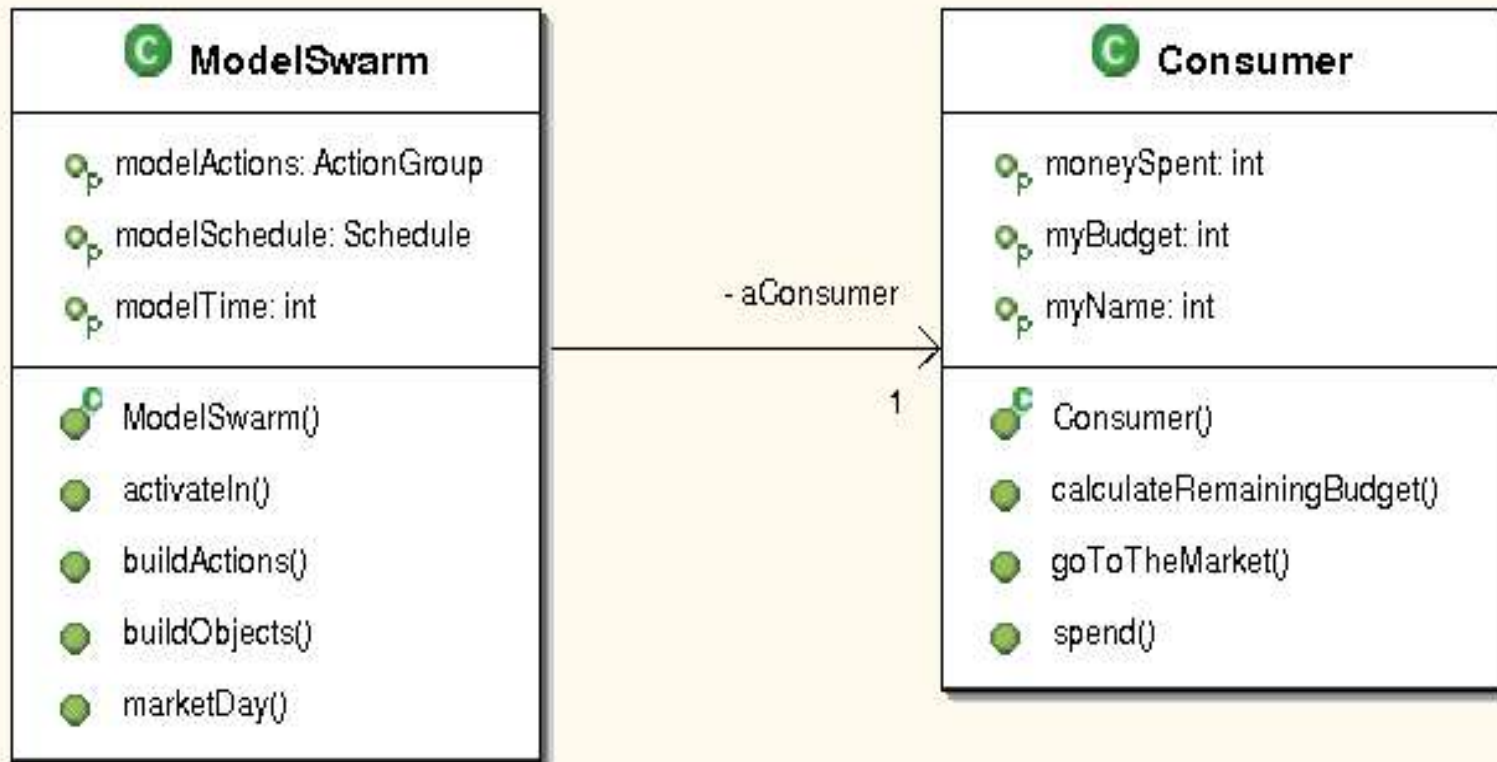
- Per correggerli basta cliccare sulla icona rossa a forma di lampadina a sinistra e introdurre i necessari *import*

activateIn

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

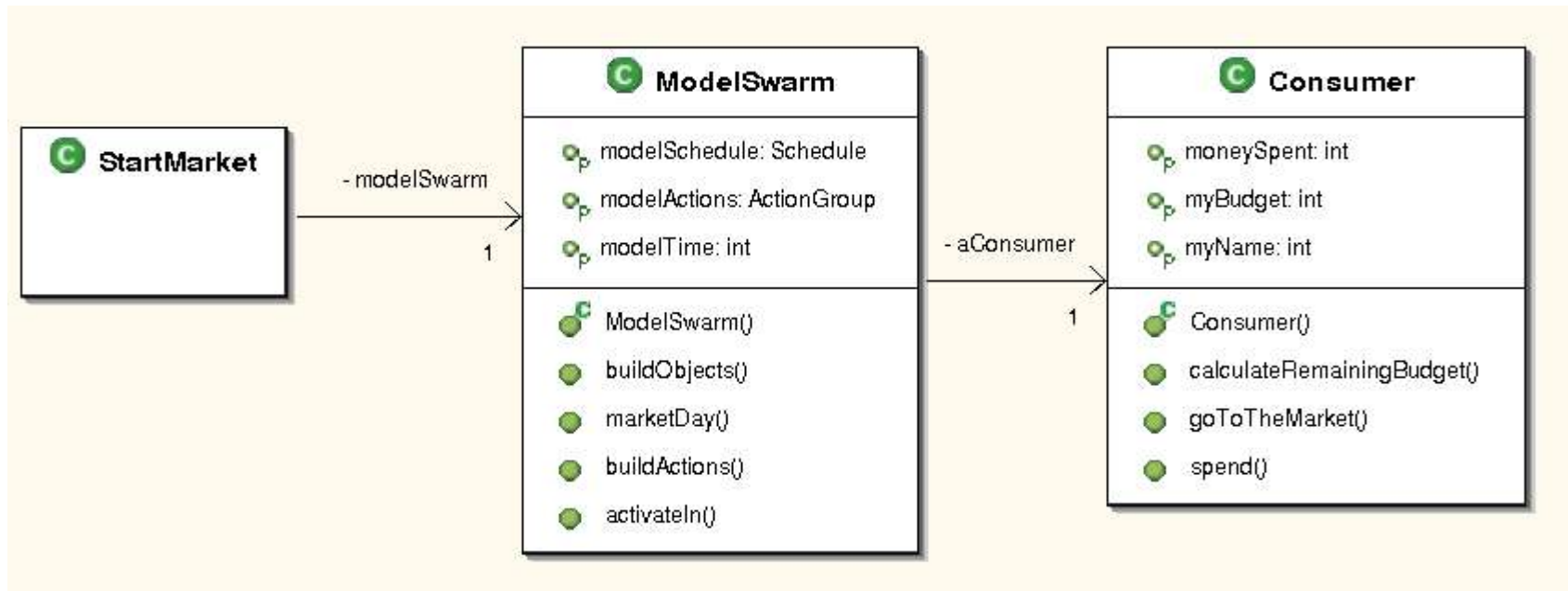
```
super.activateIn(swarmContext);  
schedule.activateIn(this);  
return this.getActivity();
```


Class Diagram



StartMarket

- Creiamo una nuova classe *StartMarket* con metodo main
- Creiamo una associazione tra questa e la classe ModelSwarm 1-1



StartMarket

- Nel metodo main:
 - inizializziamo lo swarm:

```
Globals.env.initSwarm("market", "2.2", "abologne@cs.unibo.it", args);
```

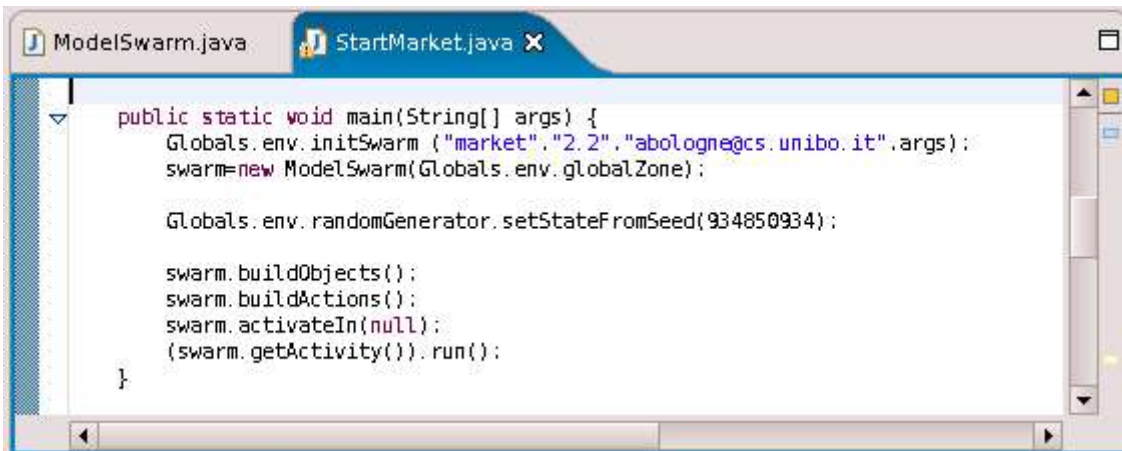
- creiamo il modelSwarm

```
swarm=new ModelSwarm(Globals.env.globalZone);
```

StartMarket metodo main()

- 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

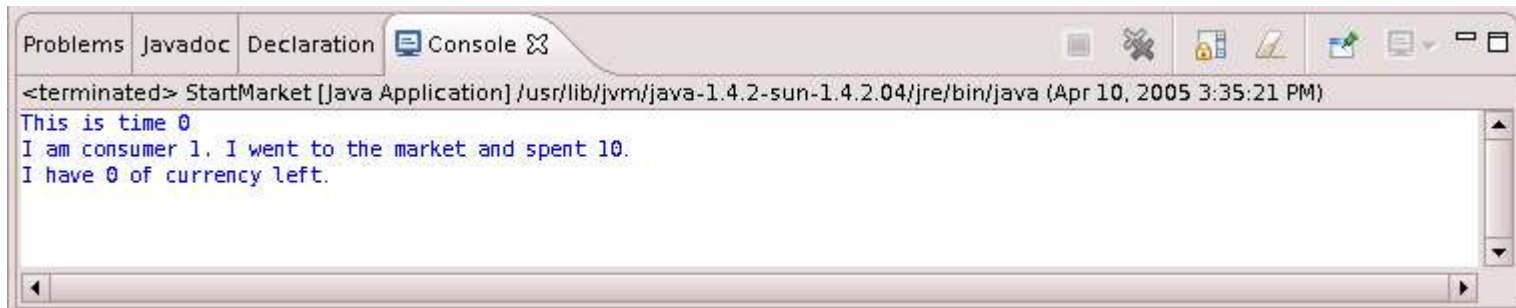
```
swarm.buildObjects();  
swarm.buildActions();  
swarm.activateIn(null);  
(swarm.getActivity()).run();
```



```
public static void main(String[] args) {  
    Globals.env.initSwarm ("market", "2.2", "abologna@cs.unibo.it", args);  
    swarm=new ModelSwarm(Globals.env.globalZone);  
  
    Globals.env.randomGenerator.setStateFromSeed(934850934);  
  
    swarm.buildObjects();  
    swarm.buildActions();  
    swarm.activateIn(null);  
    (swarm.getActivity()).run();  
}
```

Run StartMarket

- Per far eseguire la simulazione:
 - Dal menu **Run** scegliamo *Run...*,
 - *Java Application* ➔ *New* ➔ *Run*
- Il risultato mostrato nel tab console è:



```
<terminated> StartMarket [Java Application] /usr/lib/jvm/java-1.4.2-sun-1.4.2.04/jre/bin/java (Apr 10, 2005 3:35:21 PM)
This is time 0
I am consumer 1. I went to the market and spent 10.
I have 0 of currency left.
```