UNA PICCOLA INTRODUZIONE A SWARM: ObjectiveC e Java

Preparata da Marie-Edith Bissey
Dipartimento di Politiche Pubbliche e Scelte Collettive POLIS
Università del Piemonte Orientale
email: bissey@sp.unipmn.it

7 marzo 2001

Indice

1 L'USO DELLE SIMULAZIONI

1.1 Perchè usare simulazioni e agent-based modelling

Si usano simulazioni quando una soluzione analitica al problema non esiste senza semplificazioni che togliono interesse al problema. Succede per esempio nel caso di intelligenza artificiale, di problemi con interazioni, con un numero grande di agenti, etc.

Quindi, i modelli basati sugli agenti ('agent-based modelling') sono un modo più flessibile di studiare il comportamento economico dei modelli matematici. Nei modelli basati sugli agenti, tutte le parti del sistema sociale possono essere rappresentate da algoritmi e variabili che definiscono il comportamento degli agenti virtuali, e conservano dati sulla loro evoluzione nel tempo.
Il comportamento collettivo e i componenti di un sistema possono avere effetti dinamici che cambiano l'ambito del sistema: i vincoli del modello cambiano, e questo è difficile da studiare con modelli analitici, e spariscono nell'osservazione empirica.
Quest'approccio si trova a meta fra i modelli analitici e l'osservazione empirica, nel senso che si basa sui teoremi dei modelli analitici, ma genera dati che devono essere analizzati prima di arrivare ad una conclusione. Inoltre, permette di vedere come, partendo di una base, si arriva ad un risultato specifico, il che non è possibile con l'osservazione empirica, che ha a sua disposizione solo una 'foto' del sistema a diversi tempi.
I modelli basati sugli agenti offrono dunque una visione complementarie dei sistemi di comportamento rispetto ai modelli analitici.

1.2 Alcuni casi di uso dell'agent-based modelling

L'agent-based modelling viene usato un pò dappertutto nell'ambito della scienza. Alcuni esempi (tratti dal sito di Swarm) sono:

- Scienze naturali
  - Simulazione dell'espansione di cellule batteriche
  - Modelli dinamici di ecosistemi
  - Modelli di predatore/preda usando la dinamica spaziale
- Scienze sociali
  - Il ruolo delle emozioni nelle simulazioni sociali.
  - Crescita regionale, configurazione spaziale e sistemi di trasporto
  - Trasmissione culturale fra gruppi spazialemente strutturati
- Scienze economiche
  - Modellazione degli effetti della soddisfazione dei consumatori sui profitti per diverse durate di tempo
  - (In)stabilità dei sistemi finanziari
  - Modelli di concorrenza oligopolistica
- Applicazioni commerciali
  - Sistemi di comunicazione mobile con agenti mobili
  - Amministrazione e risorse umane (programmi di formazione)
- ...

2 SWARM E LA PROGRAMMAZIONE CON OGGETTI

2.1 Cos'è Swarm

Swarm è nato nel 1995, nel Santa Fe Institute (New Mexico, USA). L'obiettivo era di creare un insieme di programmi e librerie standard, da usare per simulare ed analizzare sistemi complessi di comportamento, nell'ambito delle scienze naturali e sociali.
L'idea era di avere uno strumento che permetta ai ricercatori di sviluppare le loro applicazioni senza dovere spendere tempo su problemi tipici di programmazione informatica (per esempio, la generazione di grafici o di un'interfaccia grafica, o i modi di salvare i risultati delle simulazioni). Inoltre, Swarm provvede una struttura per simulazioni, un linguaggio comune che le rende riproductibili da altri ricercatori.

2.2 Risorse

La maggioranza di queste risorse si trova online, e in inglese.

2.2.1 Risorse generali

- The Swarm Development Group (`http://www.swarm.org/index.html`)
- Gruppo Torinese Utilizzatori Swarm (`http://eco83.econ.unito.it/swarm/`)
- Benedikt Stefansson's course at UCLA (`http://cce.sscnet.ucla.edu/swarmcourse/`)
- Support lists (`http://www.swarm.org/community-mailing-lists.html`). Sullo stesso sito, troverete anche gli archivi delle liste, e le FAQ di Swarm, che potete leggere e cercare. *Si raccomanda di guardare nell'archivio e le FAQ prima di mandare una domanda alla lista: spesso, qualcuno ha già avuto un problema simile al vostro.*
- *'Economic Simulations in Swarm: Agent-based Modelling and Object Oriented Programming', editors Francesco Luna and Benedikt Stefansson, Kluwer Academic Publishers, 2000.* Questo libro contiene rapporti su diversi lavori che usano Swarm, i cui programmi si trovano sul sito web di Swarm (`ftp://ftp.swarm.org/pub/swarm/src/users-contrib/anarchy/lunabook/`). Il capitolo 1 contiene un tutorial di Swarm, molto utile, anche se obsoleto in alcune parti (per esempio, l'uso di *ObjectLoader* che è adesso deprecato).
- Swarm by Example: bell, Ralph Stefan: è una presentazione di un programma, dalla prima idea alla compilazione. (`ftp://ftp.swarm.org/pub/swarm/src/users-contrib/anarchy/bell/HTML/bell.html`)

2.2.2 ObjectiveC

- Swarm documentation (`http://www.santafe.edu/projects/swarm/swarmdocs/set/set.html`)
- Swarm tutorial (`http://www.swarm.org/intro-tutorial.html`)
- Swarm user guide (`http://www.santafe.edu/projects/swarm/swarmdocs/userbook/userbook.html`)
- ObjectiveC (`http://www.swarm.org/resources-objc.html`)

2.2.3 Java

Le librerie di Swarm sono scritte con ObjectiveC. Però gli scrittori di Swarm forniscono un'interfaccia per scrivere i programmi con Java. Non c'è dunque un'implementazione 'nativa' di Swarm in Java (e non è prevista).

- JavaSwarm reference guide (`http://www.santafe.edu/projects/swarm/swarmdocs/refbook-java/index.html`)
- JavaSwarm tutorial (`http://www.swarm.org/csss-tutorial/frames.html`, or `http://www.swarm.org/csss-tutorial/csss-tutorial.pdf` for a `pdf` version)
  Questo tutorial è completo e semplice. Spiega anche un modello con agenti già complessi, ed è dunque da leggere, anche se preferite ObjectiveC, almeno per i metodi di programmazione (gli agenti del modello sono piu' sofisticati degli 'heatbugs' usati in altri tutorials). Questo tutorial accentua anche bene le differenze fra i due linguaggi di programmazione.
- Java (`http://java.sun.com/`)

2.3 Programmare con oggetti

2.3.1 Definizione

Una cosa importante da capire sulla programmazione con oggetti è che è un modo diverso di programmare. Ogni elemento di un programma viene messo in una classe separata che definisce variabili e comportamenti (metodi); poi, il programma principale usa esempi (instances) di queste classi e li fa interaggire insieme.
Gli stessi programmi potrebbero essere scritti con un modo di programmare più tradizionale, ma sarebbero più complicati e meno chiari.
La programmazione con oggetti ha 4 proprietà essenziali:

⇒ astrazione il programma (code) che descrive un oggetto è scritto in una classe, poi nel programma principale, gli oggetti sono creati come esempi (instances) della classe.

⇒ incapsulamento gli oggetti nascondono i loro metodi e dati, separano l'interfaccia dell'implementazione. Si sa cosa fa un oggetto, ma come lo fa si trova da un'altra parte, e può essere modificato senza dovere cambiare i programmi che usano l'oggetto.
I valori delle variabili nell'oggetto sono propri all'oggetto. Devono essere scritti per passare quest'informazione all'esterno dell'oggetto.

⇒ eredità ogni classe di oggetti può essere derivata da altre classe già scritte. In questi casi, la classe 'figlia' eredita di tutte le variabili e i metodi della classe 'madre', li può modificare e può aggiungerne altri.

⇒ polimorfismo si possono avere metodi con lo stesso nome, ma che hanno parametri diversi (si chiama anche *overloading*).
Si deve notare che Swarm non permette il polimorfismo. I metodi creati in Swarm devono dunque avere nomi unici.

2.3.2 Vocabolario

⇒ class la definizione di un oggetto e la 'fabbrica' dell'oggetto

⇒ superclass una classe dalla quale l'oggetto eredita comportamento (metodi) e stati (variabili) in maniera ricorsiva

⇒ subclass una classe che eredita comportamento e variabili da una superclass

⇒ instance, object un oggetto (esempio di una classe), che è stato creato e cui è stato dato uno spazio in memoria. un oggetto contiene 2 tipi di informazione: delle *variabili* (instance variables) e dei *metodi* (methods).

⇒ instance variable una variabile pubblica per tutti i metodi di un oggetto, che dà informazione sullo stato dell'oggetto

⇒ method una funzione (metodo) che definisce un comportamento dell'oggetto, e che può essere chiamata tramite un messaggio mandato all'oggetto

Tutto questo può sembrare abbastanza oscuro, ma lo vediamo adesso su un esempio semplice di un programma di Swarm scritto prima con l'interfaccia ObjectiveC e quindi con l'interfaccia Java.

2.4 La struttura generale di un programma

Come vedrete fra poco (par. 3, pagina 7), i files che dovete scrivere per un programma sono diversi secondo che usate ObjectiveC o Java. Ma comunque, troverete sempre gli stessi elementi:

- Un file dove è definita una funzione main(), che è fundamentale nel programma. Qui, viene inizializzato lo Swarm, e sono creati gli 'swarms' importanti nel programma (il *modelSwarm*, l'*observerSwarm* o l'*experimentSwarm*). È anche qui che viene lanciata la simulazione
- Un `Makefile` che permette di compilare il programma. Contiene i nomi dei files che compongono il programma, le relazioni fra di loro, con informazioni sulla posizione di Swarm nel filesystem.
- Un *modelSwarm*, dov'è definito il modello e sono creati gli agenti. Un programma di Swarm funzionnerebbe anche senza *modelSwarm*, dichiarando tutto nel metodo main(), ma si perderebbero molte facilità di Swarm (spezialmente gli *Schedules*).
- *(facoltativo, ma utile)* Un *observerSwarm*, dove si trovano gli oggetti e interfaccie grafici.
- *(facoltativo)* Un *experimentSwarm*, dov'è definita una simulazione. Permette di far girare il programma più volte, cambiando alcuni parametri.
- Alcuni files che definiscono gli agenti del modello.

3 UN PROGRAMMA SCRITTO CON SWARM

Questo programma non è il più semplice che si può scrivere con Swarm. Nei tutorials, corrisponde al capitolo 'simpleSwarmBug'. Contiene una classe *Consumer*, un *modelSwarm* e il file con il metodo main(). Nel *modelSwarm*, sara creata un 'instance' (esempio) della classe *Consumer*, che dovra decidere (a caso) se vuol andare al mercato. Nel caso di si, deciderà (di nuovo a caso) quanto spendere (dato il suo bilancio).

3.1 Objective C

Per ogni classe, si scrivono due files. Il file `.h` contiene le dichiarazioni sulle variabili e metodi della classe (l''interfaccia'), mentre il file `.m` contiene le definizioni sui metodi (l''implementazione'). I listings presentano prima la classe *Consumer*, poi il *modelSwarm*, e finalmente, il file `main.m` e il `Makefile`. Sono nell'ordine nel quale dovrebbero essere scritti, nel senso che nella programmazione con oggetti, si scrivono prima gli oggetti semplici, e poi le classi dove interagiscono.
Nella parte seguente, li spiegheremo nell' ordine inverso, che è più adatto per capire il modello.

3.1.1 Un mercato semplice: commenti

Il Makefile serve a definire le condizioni della compilazione del programma.

- In ObjectiveC, la compilazione usa i files `.h` e `.m` per creare dei files oggetti `.o`, che sono in formato binario. Alla fine della compilazione, c'è una fase di 'linking' dove i files `.o` sono usati per creare il file `market.exe`.
- Il `Makefile` deve dunque contenere almeno i componenti seguenti:
  - Il nome del file `.exe` (alla voce `APPLICATION=`)
  - I files oggetti (`.o`) da creare (alla voce `OBJECTS=`)
  - Dove trovare l'applicazione `Makefile.appl`
  - Poi, uno dopo l'altro, i files che compongono i files oggetti. Per esempio, per creare `main.o`, il programma ha bisogno di `main.m`, ma anche di `ModelSwarm.h`. Se guardate `main.m`, vedrete che `ModelSwarm.h` è incluso all'inizio.
- Per far compilare il programma, si dà il comando `make` al prompt del terminale di Swarm (dopo aver usato il comando `cd` per andare nella cartella dov'è il programma).
- Quando il programma è stato compilato (senza errori), il file `market.exe` viene creato. Per far girare il programma, basta dunque scrivere `./market` al prompt. Notate che `'./'` è importante! Altrimenti, Swarm va a cercare nella sua cartella `bin/` se un tale programma esiste ... e non lo trova!, dandovi l'errore:

  ```
  "C:/Swarm-2.1.1/bin/bash.exe": market: command not found
  ```

- Il comando `make clean` cancella tutti i files `.o` e `.exe` creati da `make`.
- È importante salvare il `Makefile` con la `M` maiuscola e nessuna estenzione, altrimenti, il comando `make` non funziona.

Il file main.m

- Questo file non ha un corrispondente `main.h`. Infatti, questo file è anche diverso degli altri, nel senso che definisce solo una funzione `main()` nella quale sono iniziati lo Swarm e la simulazione.
- La prima dichiarazione del file `main.m` è di importare il file `modelSwarm.h` dove sono importati altri files di Swarm, e dichiarati gli oggetti usati nella simulazione.
- Poi, si dichiara la funzione main. Gli argomenti (`argc`, `argv`) servono a interpretare alcuni argomenti che si possono aggiunge al comando `make` (come per esempio `clean`).

- La prima cosa che si fa nella funzione main() è di dichiarare il modelSwarm (le dichiarazioni di variabili interni al metodo si fanno sempre prima delle altre cose, vedere par. 3.1.2, pagina 11.
- In main, si inizializza lo Swarm con il comando

  ```
  initSwarm(argc,argv);
  ```

- Poi, si crea il *modelSwarm*, e si costruisce la simulazione, con invocazioni ai metodi buildObjects, buildActions e activateIn:.
- il *modelSwarm* è creato usando lo statement:

  ```
  modelSwarm=[ModelSwarm create:globalZone];
  ```

  modelSwarm è il nome dato al modelSwarm della nostra simulazione. È un instance della classe *ModelSwarm* ed è creato usando una zona di memoria (`globalZone`).
  Una convenzione nelle notazioni è che la classe commincia con una lettera maiuscola, mentre il nome dell'instance inizia con una lettera minuscola. L'instance può anche avere un nome diverso di quello della classe. Per esempio, avremmo potuto chiamare il nostro modelSwarm *marketModelSwarm*, e in questo caso, sarebbe stato dichiarato:

  ```
  marketModelSwarm=[ModelSwarm create:globalZone];
  ```

- Quando la simulazione è pronta, gli diciamo di girare (run).

Il modelSwarm

- Il file `.h`: l'interfaccia.
  - La prima cosa è di caricare le librerie di Swarm necessarie per far girare il *modelSwarm*. Le librerie provenendo da `objectbase`, `simtools` e `activity` devono esserci. Infatti, definiscono cos'è uno Swarm e come funziona.
    La libreria `random` è utile in questa simulazione perchè abbiamo bisogno di numeri a caso.
  - Le librerie di Swarm si caricano con il comando

    ```
    #import <libraryName.h>
    ```

    I `'< >'` sono un'indicazione dato a Swarm per cercare le librerie nel suo stesso path.
  - Si carica anche il file header dov'è definita la classe *Consumer*
  - Gli headers delle classi che hanno rapporto con la simulazione si caricano con il comando:

    ```
    #import "className.h"
    ```

    I `'" "'` sono un'indicazione data a Swarm di cercare i files nel path dov'è la simulazione. In questo file, dichiariamo il *modelSwarm*. Notate che questi files vengono inclusi nel `Makefile` come componenti del file `.o`.

– Nell'interfaccia, si dichiarano le variabili globali della classe (qui, solo il `modelSchedule` e un consumatore).
– Poi, si dichiarano i 6 metodi usati dal *ModelSwarm*
– Il file finisce con `@end`

Di solito, si dichiara solo una classe per file. Ma in alcuni casi, vedrete più di una classe: basta riscrivere un altro blocco `@interface...@end` sotto il primo.

- il file `.m`: l'implementazione.
  – La prima dichiarazione è di importare il `ModelSwarm.h`, dove sono definiti variabili globali per la classe, e i metodi.
  – L'implementazione inizia con lo statement:

    ```
    @implementation ModelSwarm
    ```

    e finisce con `@end`
  – La creazione del *ModelSwarm* si fa tramite il paio di metodi createBegin: e createEnd. Questo ci permette di inizializzare alcune variabili generali del modelSwarm. Usiamo una sintassi un pò 'barbara' con la creazione di un oggetto `obj`, che sarà il nostro `modelSwarm`. Inizializzamo le variabili (come `modelTime`) in modo diretto usando

    ```
    obj->modelTime=0;
    ```

    Questo è detto 'barbaro' perchè normalmente, dovremmo usare i metodi del tipo setModelTime:. Tuttavia, sarebbe necessario scrivere molti tali metodi, che sarebbero usati solo una volta nel programma.
    Altri dettagli sulla creazione di oggetti si trovano nel par. 3.1.2, pagina 11.
  – Nel metodo buildObjects si creano gli oggetti usati nella simulazione. Qui, abbiamo solo bisogno di un consumatore, che è creato con il suo nome e il suo bilancio.
    Lo statement per creare il consumatore è:

    ```
    aConsumer=[Consumer create:[self getZone]];
    ```

    quindi, il nostro consumer si chiama *aConsumer* ed è un instance della classe *Consumer*. È creato usando una zona della memoria del modelSwarm:

    ```
    [self getZone]
    ```

    vuol dire che prendiamo una parte della zona di memoria della classe dove siamo adesso (cioè il *modelSwarm*).
    Il metodo buildObject non deve ritornare nessun risultato, ma non è neanche dichiarato `void`. In questo caso, lo statement `return self` deve finire l'implementazione del metodo.
  – Il metodo marketDay spiega cosa succede il giorno in cui c'è il mercato.

1. Il consumatore deve decidere (a caso) se va o no al mercato. La sua decisione è contenuta nella variabile `go`, risultato del metodo goToTheMarket definito nella classe *Consumer*, e quindi invocato dalla nostra instance *aConsumer*.
   `go` prende il valore 1 se il consumatore va al mercato, e 0 se resta a casa.
2. Se `go` non è 0, il consumatore deve decidere quanto spendere. Il risultato è contenuto nella variabile `spending`, risultato del metodo spend invocato dal consumatore.
3. Chiediamo al programma di scrivere sullo schermo alcuni dettagli sul consumatore, quanto ha speso al mercato, e quanto resta. Per questo, si usa il metodo fprint(). Si scrive la frase fra `" "`. Se ci sono variabili per le quale vogliamo mettere il valore, si scrive il codice del formato della variabile nella frase. Alcuni dei codici più usati sono %d per un numero intero, `%f` per un numero decimale. La frase finisce con `\n` che significa 'iniziare una nuova riga sullo schermo dopo la frase'. Poi, dopo una virgola, si mettono i nomi delle variabili che si devono includere nella frase.
4. Se il consumatore non va al mercato (`else`), scriviamo i dettagli del consumatore, e gli chiediamo di comunicarci il suo bilancio.

– Il metodo buildActions crea:

  * Un gruppo di azioni: il *modelActions*. Nell'actionGroup, troviamo tutte le azioni che si eseguono simultanemente nella simulazione. Qui, potremmo anche fare senza, dato che c'è solo un'azione da eseguire.
  * Il *modelSchedule*, un instance della classa *Schedule*. Il *Schedule* ('orario') permette di specificare gli actionGroups da eseguire ad ogni momento della simulazione. Qui, lo *Schedule* dice che al momento 0, si eseguono le *modelActions*.
    Dichiarato così, il programma girerà una volta sola. Se vogliamo ripetere la simulazione più volte (ad infinitum in questo caso, perchè non abbiamo scritto un metodo per fermarla), dobbiamo specificare un `repeatInterval` con:

    ```
    [modelSchedule setRepeatInterval: 1];
    ```

    Il `repeatInterval` è 1 perchè abbiamo solo un actionGroup da eseguire.

– Il metodo activateIn: è necessario per poter girare la simulazione, e si presenta quasi sempre com'è scritto qui:

1. Si attiva la classe "madre" del *modelSwarm*.
2. Si attiva lo schedule del *modelSwarm*.
3. Si ritorna l'attività del *modelSwarm*.

Il Consumer In questi files, si dichiarano variabili e metodi che caraterizzano il consumatore.

- setConsumerName:(int)name Budget:(int)budget permette di passare dentro la classe *consumer* i valori del suo nome e del suo bilancio.
- (int)goToTheMarket decide a caso se il consumatore va (1) o no (0) al mercato.
- (int)spend decide a caso quanto del bilancio si spende sul mercato.
- (int)calculateRemainingBudget viene usato quando il consumatore è andato al mercato, e ricalcola il bilancio meno le spese. Notate l'uso dell'operatore '$-=$', molto comune in C e

linguaggi derivati. $a- = b$ è equivalente a $a = a - b$. Una sintassi simile si può usare con gli altri operatori: $+$, $*$, $/$.

- i metodi (int)getConsumerName e (int)getBudget sono usati per passare i valori del nome e del bilancio rispettivamente ad altre classe del programma.

3.1.2 Nota: dichiarare variabili e metodi in ObjectiveC

- Le dichiarazioni sono separate nel file `.h`
- Notate che la dichiarazione di un metodo, o la sua definizione inizia con un trattino `-`. Vedrete anche ogni tanto un metodo iniziare con +(il più importante è `+createBegin:`). Questo significa un metodo che non può essere eseguito da un instance della classe.
- Bisogno dichiarare il tipo di variabili: `int variableName;`
  - `int` per numeri interi
  - `float`,`double` per numeri decimali
  - `BOOL` per booleans (valori 'vero' (1) o 'falso' (0))
  - `char *` per caratteri
  - `id` per oggetti, è assunto come tipo di default se la dichiarazione della variabile non contiene il tipo

  Notare che le variabili devono essere dichiarate all'inizio del metodo o dell'interfaccia, prima di ogni altro statement (altrimenti, ci sara un errore).
- Oggetti possono essere dichiarati con il tipo `id`:

  ```
  id aConsumer
  ```

  Possono anche essere dichiarati in maniera più precisa. Per esempio, sappiamo che `aConsumer` sarà un instance della classe *Consumer*. Dunque, lo possiamo dichiarare (come abbiamo fatto), come un oggetto di "tipo" *Consumer*.

  ```
  Consumer * aConsumer
  ```

  Il vantaggio di questo è che Swarm controlla immediatamente se la dicharazione del consumatore `aConsumer` corrisponde a quella della classe *Consumer*.
  Un altro modo di dichiarare oggetti, che abbiamo usato nel *ModelSwarm* per dichiarare il `modelSchedule` e il `modelActions`, è la notazione usando i protocolli. Un protocollo è una lista dei metodi eseguibili da un oggetto. A differenza della classe, non ci sono variabili, e non c'è la definizione dei metodi. Gli oggetti più importanti di Swarm, come gli Schedules, gli ActionGroups, ma anche le Lists e Arrays, hanno protocolli, e dunque si possonno dichiarare usando la sintassi:

  ```
  id <Schedule> modelSchedule;
  ```

  Notare che tutti questi oggetti si possonno anche dichiarare usando la sintassi della classe:

  ```
  Schedule * modelSchedule;
  ```

- Bisogno dichiarare il tipo del risultato dei metodi. Sono gli stessi tipi delle variabili. In più, esiste il tipo `void` per una funzione che non ritorna nessun risultato. Se non c'è un tipo specificato, Swarm assume che il risultato è di tipo oggetto (`id`). Nei metodi, si deve anche dichiarare, fra parentesi, il tipo degli argomenti:

  ```
  (float)methodName:(int) arg1:(char *) arg2:arg3;
  ```

  dove methodName ritorna un risultato di tipo `float`, usando argomenti di tipo `int`, `char *` e `id` (che può non essere specificato). Un altro modo di dichiarare un metodo è di usare altre keywords, per rendere le cose più chiare (in particolare il numero e la posizione degli argomenti):

  ```
  (float)methodNameUsingArg1:(int) arg1 WithArg2:(char *) arg2 AndArg3:arg3;
  ```

- il nome della funzione nel file `.m` deve essere uguale al nome della funzione nel file `.h` (il che include anche il nome degli argomenti e il loro tipo).

- se il metodo non ritorna nulla, deve essere dichiarato con il tipo `void`. Oppure, può essere dichiarato senza tipo di ritorno (che vuol dire `id`), e avere l'istruzione di ritorno: `return self;`. Questo vuol dire che il metodo ritorna l'oggetto al quale appartiene. Questo è usato più spesso della dichiarazione con tipo `void` nei programmi scritti con Swarm.

- instances di una classe usano il metodo create, o la coppia di metodi createBegin: e createEnd. Quindi:

  ```
  objectName=[ClassName create:aMemoryZone];
  ```

  è equivalente a

  ```
  objectName=[ClassName createBegin:aMemoryZone];
  objectName=[objectName createEnd];
  ```

  Il secondo modo di creare la classe è utile se la creazione dell'oggetto ha bisogno di usare alcuni metodi, o di inizializzare alcune variabili.

3.2 Java

La struttura del programma è uguale a quella precedente, quindi nei commenti, parliamo solo delle diferenze fra il programma scritto con ObjectiveC e quello scritto con Java. La differenza principale è che dichiarazioni e implementazioni delle classi non sono più separate fra i files `.h` e `.m`. Ogni classe di nome *ClassName* si trova in un file `ClassName.java` (notate la corrispondenza fra il nome della classe e quello del file). La funzione main() si trova nel file `ProgramName.java`, che dopo compilazione girerà con il comando:

```
javaSwarm ProgramName
```

La compilazione si fa sempre tramite il `Makefile`.

3.2.1 Un mercato semplice: commenti

Il Makefile `JAVA_SRC` specifica i files source del programma (cioè i programmi della simulazione) alla voce: `JAVA_SRC=`. Poi, vedete 2 voci:

- `all` specifica le cose da fare quando è dato il comando `make`, cioè compila il programma
- `clean` pulisce i files creati da `make` quando è dato il comando `make clean`.

`make` crea un file `.class` per ogni classe. Questi files sono usati quando gira la simulazione con il comando

```
javaSwarm StartMarket
```

(nel nostro caso).

StartMarket.java Questo file sarebbe l'equivalente del file `main.m` del programma ObjectiveC.

- Prima, si importano le librerie. Notare la differenza con l'import di ObjectiveC: qui, non abbiamo bisogno di un #, ma deve esserci un `;` alla fine dello statement.
- La libreria *swarm.Globals* definisce lo swarm, e *deve essere dichiarata in tutti i files del programma*.
- La classe *StartMarket* contiene il programma. Notare che ha il nome del programma da far girare dopo la compilazione.
- Nella classe *StartMarket*, si trova definita la funzione main(), che è dichiarata con il tipo `void`. *NON DEVE AVERE UN'ALTRO TIPO* o ci sara un errore.
- Prima di `void`, si trovano caratteristiche della funzione: `public` e `static`. Lo vedrete spesso nelle funzioni in Java. Definiscono l'ambito (scope) della funzione e se può essere classa 'madre'. Troverete piu dettagli nel par. 3.2.2, pagina 15.
- Il resto della funzione main() è esattamente come nel file `main.m`, tranne l'uso della sintassi di Java.
  In generale, in ObjectiveC, un messaggio a un oggetto è:

  ```
  [objectName methodName:argomenti];
  ```

  Con Java, si scrive:

  ```
  objectName.methodName(argomenti);
  ```

- Lo Swarm si inizializza con lo statement:

  ```
  Globals.env.initSwarm ("market","2.1.1","bissey@sp.unipmn.it",args);
  ```

  dove il primo argomento è il nome del programma, il secondo è la versione di Swarm con la quale è stato scritto, il terzo è un indirizzo dove mandare i bug reports, e il quarto rappresenta gli argomenti del comando `make`. Questi 4 argomenti sono obbligatori (se non ci sono, ci sara un errore).
- Per dichiarare un instance di una classe, si usa la keyword new:

  ```
  objectName= new classConstructor(arguments);
  ```

  Vedere par. 3.2.2, pagina 15 per altri dettagli

ModelSwarm.java

- Prima, si importano le librerie. Quelle che finiscono con `'impl'` permettono di creare nuovi oggetti con la keyword `new` (vedere par. 3.2.2, pagina 15)
  Notare che non c'è bisogno di importare i files locali (javaSwarm li trova dal `Makefile`).
- Nella dichiarazione della classe *ModelSwarm*, si deve specificare la sua classe 'madre' (*superClass*). In Java, si usa la keyword `extends`:

  ```
  public class ClassName extends SuperClass
  ```

- si dichiarano le variabili globali della classe
- il primo metodo richiesto è il costruttore della classe, che viene usato con la keyword `new` quando si crea un instance della classe. Il costruttore ha lo stesso nome della classe, e fra i suoi argomenti, uno di tipo `Zone`, per gestire la zona di memoria nella quale è creata l'instance. Notare anche che il costruttore non deve avere nessun tipo di return (come `void` ...).
- Il resto della classe è molto simile a quello che abbiamo descritto nei commenti del `modelSwarm.m`.
- Una cosa da notare è il modo di stampare dettagli sullo schermo, che è diverso di quello usato da ObjectiveC. Il metodo è chiamato con:

  ```
  System.out.println(); o System.out.print();
  ```

  dove il `'ln'` indica di andare su una nuova riga alla fine. Le parti di 'frase' sono fra `" "`. Se vogliamo introdurre il valore di una variabile, chiudiamo la parte 'frase' con `"`, et dopo un `+`, aggiungiamo il nome della variabile, e continuiamo la frase dopo un altro `+`.
- Un'altra cosa da notare è che la keyword `self` non esiste in Java. Si usa `this`.

Consumer.java Non ci sono molte differenze con la classe scritte in ObjectiveC, tranne che abbiamo approfittato del costruttore per inizializzare le variabili necessari al consumatore, e dunque non abbiamo più il metodo setConsumerName:Budget.

3.2.2 Nota: dichiarare variabili e metodi in Java

- Ci sono due modi di dichiarare un instance di una classe di Swarm:
  - si usa il costruttore della classe (preceduto dalla keyword `new`), con gli argomenti adeguati. Questo è il modo più semplice e più usato. In molte classe vedrete che ci sono infatti più di un costruttore, con diversi argomenti. Questa caratteristica di Java si chiama *Overloading*. Vuol dire che si può dichiarare più di una funzione con lo stesso nome, e diversi argomenti. Quando l'usate, Java guarda gli argomenti per sapere quale funzione usare.
    Le classi di Swarm che si possono usare in questo modo hanno nomi che finiscono con `'Impl'`.
  - si usa un modo simile a quello di ObjectiveC usando prima il metodo createBegin(), inizializzando alcune variabili dell'oggetto, e terminando con createEnd().
    Questo modo è più flessibile.

- Nella dichiarazione delle classe e dei metodi, abbiamo visto alcune keywords in Java, che non troviamo in ObjectiveC: `public`, `static` per esempio. Queste keywords danno indicazioni al compilatore (e al lettore) sul livello d'accesso di una classe o di un metodo.

  – per una classe, possiamo avere 3 keywords, che troviamo una dopo l'altra.

  ⇒ public permette alla classe di essere accessibile da tutte le altri classi. Se `public` non si trova, la classe può essere accessibile solo da classi dello stesso 'package' (una collezione di classi)

  ⇒ abstract definisce una classe che non può essere 'instanziata' (per creare un oggetto). Una tale classe definisce variabili e metodi, ma può solo essere una *SuperClass*.
  Una classe `abstract` generalmente dichiara un concetto abstratto:

  ```
  public abstract class graphic(args);
  ```

  che viene usata da classi più specifiche:

  ```
  public class histogram extends graphic(args);
  ```

  Se `abstract` non si trova, la classe può essere instanziata.

  ⇒ final dichiara una classe che non può essere 'subclassed' (cioè eredita di un'altra che può aggiungere o soprascrivere ('overload') metodi). Ci sono 2 ragioni per questo:
  - Sicurezza: soprascrivere classi è un metodo usato tradizionalmente dai crackers per cambiare il comportamento di un programma, o invadere un sistema;
  - Design: se trovate la funzione perfetta, oppure se la classe non può, dato il suo ruolo specifico, avere subclasses.

  La maggioranza delle classi che abbiamo usate sono dichiarate solo con la keyword `public`, il che vuol dire che possono essere usate da qualunque altre classe, che possono essere instanziate e avere subclasses.

  – per un metodo, abbiamo 5 keywords possibili (vedere il "Java Tutorial"), i cui più importanti sono:

  ⇒ per controllare l'accesso:
  - ⇒ private accessibile solo dalla classe nella quale è definito
  - ⇒ protected accessibile dalla classe, dalle sue subclasses, e dalle classi dello stesso package
  - ⇒ public accessibile da tutte le classi in qualunque package
  - ⇒ package accessibile dalle classi dello stesso package (ma non da subclasses in altri packages). È il default se nessuna keyword di controllo di accesso è specificata

  ⇒ static dichiara un membro della classe e non una variabile o metodo. La differenza è che queste variabili o metodi sono comuni a tutte le instances della classe. Le variabili *comuni* nel modello a tutte le istance della classe sono dichiarati `static`. Un metodo dichiarato `static` è anche privato alla classe, e per questo, non può usare le variabili globali della classe, solo quelle anche dichiarate `static`.

  ⇒ final dichiara una variabile il cui valore non cambia mai (costante). Una convenzione è di scrivere il nome di una variabile finale in lettere capitali:

  ```
  final double PI=3.1456
  ```

- Dichiarare o usare un metodo senz'argomenti: non dimenticare parentesi vuote `()`
- Il tipo `id` non esiste, e non si può lasciare vuoto il tipo di una variabile o metodo. Per un oggetto, si usa il tipo `Object`
- I metodi si possono anche dichiarare con 'più parole' per aiutare a ricordarsi gli argomenti, ma la sintassi è un pò diversa da quella di ObjectiveC. In Java, un metodo si dichiara così:

  ```
  public int methodName(int arg1, float arg2, string[12] arg3);
  ```

  oppure:

  ```
  public int methodNameUsingArg1$WithArg2$AndArg3
                 (int arg1,float arg2,string[12] arg3);
  ```

  dove il primo $ corrisponde al primo argomento, ecc.

- Messaggi e selectors:
  In ObjectiveC, i selectors si usano tramite la macro M() come in:

  ```
  [modelActions createActionTo:self message:M(marketDay)];
  ```

  In Java, una tale macro non esiste, e si deve creare un selector come in:

  ```
  new Selector(getClass(),"marketDay",false);
  ```

  dove `getClass()` ritorna la classe corrente[1](nel esempio, il *ModelSwarm*). `"marketDay"` è il nome del metodo che vogliamo usare, e `false` specifica che non usiamo lo stile di ObjectiveC per creare le instances.
  Tuttavia, non possiamo usare questo selector da solo, perchè c'è un rischio di errore, e Java protesta. Infatti, è possibile che il programma abbia da creare messaggi e selectors 'on the fly' durante una simulazione, ma la corrispondenza fra classe e messaggio non è controllata fino al runtime. Se abbiamo fatto un errore nella ortografia del messaggio, o vogliamo usare un metodo che non è definito nella classe data da `getClass()`, avremmo un errore di runtime. Se questo succede in ObjectiveC, ci sara un errore piutosto enigmatico al runtime. In Java, un selector da solo è segnalato durante la compilazione, con un messaggio abbastanza chiaro, dicendo che questo selector deve essere in un blocco catch/try. Quindi, dobbiamo usare, per ogni selector (cioè ogni volta che la macro M() sarebbe usata in ObjectiveC):

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

[1] Se vogliamo un metodo che esiste in un'altra classe, per esempio, la classe *Consumer*, della quale abbiamo l'instance aConsumer, useremmo: `aConsumer.getClass()`

3.3 Quale linguaggio scegliere?

Ambedue hanno punti positivi e negativi.

3.3.1 A proposito di ObjectiveC

- Semplice da usare.
- È il linguaggio nel quale è stato sviluppato Swarm dall'inizio, dunque la maggioranza delle risorse (programmi, tutorials, ecc) sono scritti in objectiveC.
- Inoltre, è veloce.
- ObjectiveC permette di usare le funzioni di 'low-level' di C, tipo unioni e puntatori (che tra l'altro, può essere pericoloso per il debuttante).
- Permette 'dynamic typing'

3.3.2 A proposito di Java

- Semplice da usare, ma i nomi dei metodi sono generalmente più lunghi (dunque, un pò più da scrivere).
- Ma si scrivono meno files.
- Meno risorse (tutorials, programmi, ecc), ma si sta sviluppando
- Questo linguaggio ha un uso grande anche nella creazione di siti internet, applets, ecc.
- Incorraggia 'static typing', quindi forza a evitare alcuni errori (spezialmente di runtime), che si possono fare con ObjectiveC. Dall'altro lato, è meno flessibile in certe circostanze.
- Java fa automaticamente il 'garbage collecting' (cioè distrugge automaticamente gli oggetti non usati, prevenendo quindi i 'memory leaks').
- Java non è il linguaggio nativo di Swarm, dunque certe funzioni possono essere più lenti di quelle scritte con ObjectiveC, a causa delle necessarie traduzioni.
- L'interfaccia Java offre la possibilità di sfruttare delle numerose librerie grafiche di Java (che sono più ricche di quelle offerte da Swarm).

E per finire, secondo gli autori di: 'A tutorial introduction to Swarm' (scritto con Java):

> "In summary, while Objective C has advantages, the cultural and practical advantages of Java outweigh them in for the purposes of getting started with Swarm."

Nel resto di questo paper, presenterò sempre i programmi scritti con i 2 linguaggi.

4 GESTIRE PIÙ AGENTI E CONSERVARE LE LORO AZIONI

In questa parte, cambiamo il programma per introdurre nuovi importanti oggetti: liste (Lists), mappe (Maps), vettori (Arrays). Questi sono collezioni di oggetti. Quindi, se vogliamo usarli per conservare dati su risulati della simulazione (per esempio le spese dei consumatori) che non sono oggetti per se, dobbiamo usare 'wrappers' che sono classi specifiche, che trasformano un intero (per esempio) in un oggetto.
Usiamo il programma per illustrare questi concetti. Il programma scritto in ObjectiveC è in par. **??**, pagina **??** mentre quello scritto con Java è in par. **??**, pagina **??**. Abbiamo modificato significativamente il nostro programma, creando 3 consumatori (conservati in una *lista*). Per ognuno di loro, conserviamo un'indicazione sulla loro partecipazione al mercato in un *vettore* (`arrayOfVisits`), e un'indicazione sulle loro spese in una *mappa* (`mapOfSpending`). Abbiamo anche cambiato il buildActions del modelSwarm, aggiungendo un modo di fermare il programma dopo 6 ripetizioni (checkToStop). Abbiamo introdotto alcuni cicli (loops) (`for`, `while`) e condizioni (`if else`).

4.1 Collezioni e 'object wrappers'

4.1.1 Collezioni in Swarm

Ci sono tre tipi principali di collezioni: *Lists*, *Arrays*, *Maps*.

Lists

Definizione ed uso Il protocollo *List* di Swarm implementa una lista con dimensioni flessibili (cioè, la lista cresce quando sono aggiunti elementi, e decresce quando vengono cancellati elementi). La lista permette di elencare oggetti, e di eseguire, in modo semplice, azioni ripetitive sugli oggetti contenuti nella lista.
Una lista si crea in ObjectiveC con:

```
nameOfList=[List create:[self getZone]];
```

e in Java:

```
nameOfList= new ListImpl(getZone())
```

Metodi

⇒ addFirst, addLast aggiunge un oggetto all'inizio o alla fine della lista.

⇒ removeFirst, removeLast cancella il primo o l'ultimo elemento della lista.

⇒ getCount riporta il numero di elementi nella lista.

⇒ begin: aZone, .listBegin(getZone()) per creare un indice che può essere usato per attraversare la lista (la prima notazione è ObjectiveC, la seconda è Java).

⇒ remove: aMember, .remove(aMember) per cancellare `aMember` dalla lista.

⇒ removeAll cancella tutti gli oggetti della lista, ma non la lista stessa.

⇒ deleteAll cancella tutti gli oggetti della lista, e cancella anche la lista dalla memoria.

⇒ atOffset:i, .atOffset(i) per ritornare l'elemento che si trova in posizione `i` nella lista.
IMPORTANTE: nelle collezioni di Swarm, la numerazione degli elementi inizia da 0. Dunque, il primo elemento della lista si trova alla posizione 0, il secondo element si trova alla posizione 1, ecc.

⇒ contains: aMember, .contains(aMember) ritorna 1 se `aMember` è nella lista.

⇒ forEach:M(message), .forEach(selector) manda il messaggio message a tutti gli oggetti della lista. Vedere par. 3.2.2, pagina 15 per una discussione sui selectors in Java.

Arrays

Definizione ed uso Il vettore si usa quando abbiamo bisogno che gli oggetti siano nella collezione in un ordine specifico. La grande differenza con la lista è che il vettore viene dichiarato con la sua dimensione. Non si possono aggiungere oggetti fuori di questa dimensione (quindi non si possono usare i metodi addFirst o addLast). Anche, dato che la dimensione del vettore non si può cambiare, non si possono cancellare elementi dal vettore.
Il vantaggio del vettore sulla lista è che gli oggetti possono essere trovati molto velocemente, perchè il loro indice è un intero. Quindi, si può attraversare un array usando il loop `for` normale. Si può anche creare un indice con il metodo begin:aZone o .begin(aZone), come per la lista. Il vettore si crea usando, con ObjectiveC:

```
nameOfArray=[Array create:[self getZone] setCount:size];
```

e con Java

```
nameOfArray= new ArrayImpl(getZone,size);
```

Metodi

⇒ atOffset:i put:obj, .atOffset$put(i,obj) per mettere l'oggetto `obj` alla posizione `i` del vettore.
IMPORTANTE: ricordate che i vettori comminciano anche loro a 0.

⇒ atOffset:i, .atOffset(i) per ritornare l'oggetto che si trova alla posizione `i` del vettore.

⇒ getCount

⇒ contains:aMember, .contains(aMember)

⇒ forEach:M(message), .forEach(selector)

Maps

Definizione ed uso Conviene vedere una mappa come una tabella con due file di oggetti. Nella seconda, si trovano gli oggetti che vogliamo conservare nella collezione. Nella prima si trovano i nomi degli oggetti. Quindi, per mettere un oggetto in una mappa, dobbiamo anche specificare il suo nome (che si chiama: *key*, cioè chiave). Possiamo poi chiedere alla mappa di ritornare, o cancellare l'oggetto che corrisponde ad una chiave particolare. *IMPORTANTE: in Swarm, anche le chiavi devono essere OGGETTI*.
Una mappa si crea usando, con ObjectiveC:

```
nameOfMap=[Map create:[self getZone]];
```

e con Java

```
nameOfMap= new MapImpl(getZone())
```

Metodi

⇒ at:keyObject insert:object, .at$insert(keyObject,object) per inserire `object` con la chiave `keyObject`.

⇒ at:keyObject, at(keyObject) per ritornare l'oggetto corrispondente a `keyObject`.

⇒ getCount

⇒ contains:object, .contains(object)

⇒ forEachKey:M(message), .forEachKey(selector)

⇒ getFirst,getLast

Indexes

Descrizione ed uso Tutte le collezioni possono generare un indice (*Index*) per accedere ai loro membri. Come abbiamo visto prima, l'indice si crea con il metodo begin. Questo accede ai membri in ordine da 0. Se vogliamo accedere ai membri della collezione in un ordine a caso, possiamo creare l'indice con il metodo beginPermuted. Esempi dell'uso di indici si trovano nei programmi pi'u sotto, par. **??**, pagina **??**, per ObjectiveC, e par. **??**, pagina **??**, per Java.

Metodi

⇒ get per ritornare l'oggetto corrispondente alla posizione corrente dell'indice

⇒ next per far muovere l'indice all'oggetto accanto nella collezione, e ritornarlo

⇒ findNext:object, .findNext(object) per far si che l'indice cerchi avanti nella collezione finche non trova un oggetto uguale ad `object`.

⇒ drop per cancellare l'indice dalla memoria, quando non è più utile

4.1.2 'Object wrappers'

Abbiamo appena visto che le collezioni di Swarm richiedono che i loro elementi siano di tipo `object`. Quindi, se vogliamo conservare nelle collezioni elementi che non sono oggetti (come facciamo negli esempi per salvare le spese dei consumatori), dobbiamo trasformarli in'oggetti. Per questo, usiamo delle classi che chiamiamo *'wrappers'*. Vuol dire che queste classi non fanno altro che trasformare, per esempio, un numero intero in un oggetto che contiene il valore di questo numero intero.
Ci sono alcune differenze fra ObjectiveC e Java, che vediamo nei paragrafi seguenti.

ObjectiveC Nella versione del programma di mercato che trovate al par. **??**, pagina **??**, potete vedere che abbiamo aggiunto una classe di nome *Integer*. Questo è il nostro wrapper per numeri interi.
È una classe abbastanza semplice, con 3 metodi:

⇒ setValue: per entrare nell'oggetto il valore del numero intero che ci interessa.

⇒ getMyValue per ritornare il valore del numero intero

⇒ compare: per confrontare 2 oggetti di tipo intero: si usa il metodo getMyValue

Ci permette di includere per esempio le spese del consumatore in una mappa, ogni volta che va al mercato: per questo, usiamo la variabile `modelTime` come chiave, e la variabile *spending* come element. Tutte e due variabili sono numeri interi, quindi dobbiamo transformarli in oggetti prima di metterli nella mappa. Lo facciamo attraverso il metodo updateSpending::, che prende come argomenti i due numeri interi: la chiave e il valore.

```
-updateSpending:(int)key:(int)value
    {
    Integer * keyObject;
    Integer * valueObject;

    keyObject=[Integer create:[self getZone]];
    [keyObject setValue:key];
    valueObject=[Integer create:[self getZone]];
    [valueObject setValue:value];

    [mapOfSpending at:keyObject insert:valueObject];
    return self;
    }
```

Si dichiarano prima 2 oggetti di tipo *Integer*. Poi si creano gli oggetti, e si inserisce il valore usando il metodo setValue. Finalmente, si inseriscono gli oggetti nella mappa.
Per estrarre un oggetto dalla mappa, o controllarne il valore, usiamo nell'esempio il metodo getSpendingValue: che prende come argomento il valore (intero) della chiave e ritorna quelle dell'oggetto corrispondente:

```
-(int)getSpendingValue:(int)key
    {
    Integer * keyObject;
    Integer * element;
    keyObject=[Integer create:[self getZone]];
    [keyObject setValue:key];
    element=[mapOfSpending at:keyObject];
    return [element getMyValue];
    }
```

Vediamo dunque che prima, trasformiamo la variabile `key` nel oggetto `keyObject`. Poi usiamo `keyObject` per prendere nella mappa l'oggetto `element` corrispondente. Sappiamo che in questa mappa l'elemento `element` è un instance della classe *Integer*. Dunque, possiamo chiederlo di ritornare il suo valore intero, tramite il metodo getMyValue.

Java Nel programma che potete vedere in par. **??**, pagina **??**, non abbiamo scritto nessuna nuova classe. La ragione è che in Java, esistono già i wrappers per i numeri. Quindi, in Java, esiste già la classe *Integer*. Quindi, per mettere un numero intero nella mappa, usiamo, nel nostro caso, il metodo updateSpending(), che, tranne le differenze di sintassi fra ObjectiveC e Java, è uguale a quello descritto prima:

```
public void updateSpending(int key, int value)
    {
    Integer keyObject=new Integer(key);
    Integer valueObject=new Integer(value);
    mapOfSpending.at$insert(keyObject,valueObject);
    }
```

Per estrarre oggetti dalla mappa, o controllare i loro valori, usiamo, come prima, il metodo getSpendingValue:

```
public int getSpendingValue(int key)
    {
    Integer keyObject;
    Integer element;
    keyObject=new Integer(key);
    element=(Integer) mapOfSpending.at(keyObject);
    return element.intValue();
    }
```

La differenza con il metodo scritto in ObjectiveC è che quando prendiamo nella mappa l'elemento `element`, dobbiamo precisare il suo tipo (*Integer*). Infatti, in ObjectiveC, la collezione si 'ricorda' del tipo dei suoi membri. Al contrario, una collezione in Java sa solo che l'elemento è di tipo *Object*. Dunque, lo dobbiamo caratterizzare con più precisione, altrimenti non ci permette di usare il metodo intValue per ritornare il valore dell'elemento.

5 SCHEDULES E ACTIONS

In questa sessione, diamo un pò più di dettagli sugli schedules e azioni, che sono una delle caratteristiche importanti di Swarm. Infatti, permettono di integrare le azioni, ecc. di vari agenti in diversi livelli della simulazione.

5.1 Schedules

Uno schedule è come un ordine del giorno, dove si scrive cosa fare ad ogni ora del giorno. Uno schedule è dunque un altro oggetto di Swarm nel quale possiamo inserire messaggi individuali, o gruppi di messaggi (*ActionGroups*). Un argomento importante dello schedule è l'intervallo fra le azioni. Generalmente, il `repeatInterval` vale 1, il che vuol dire che le azioni sono ripetute ogni periodo.
Notate che in ObjectiveC il `repeatInterval` si inizializza fra il createBegin e il createEnd dello schedule, tramite il metodo setRepeatInterval. In Java, invece, il valore del `repeatInterval` è il secondo argomento del costruttore dello schedule.
Una volta creato lo schedule (generalmente, questo si fa nel metodo buildActions), abbiamo tutti gli oggetti della simulazione pronti. Adesso li dobbiamo attivare.

5.2 La classe Activity

Durante l'attivazione, le librerie di Swarm integrano le varie azioni degli oggetti della simulazione. Questo si fa tramite i metodi activateIn. Il metodo main() segnala allo Swarm più alto nella gerarchia (nel nostro programma, il *modelSwarm*) di attivarsi in `nil` (ObjectiveC) o `null` (Java), dopo aver creato le sue azioni e i suoi schedules. Questo fa si che il *modelSwarm* segnala al prossimo livello di Swarm di attivarsi, ecc. In questo modo, vengono sincronizzate le attività dei vari livelli della simulazione.
I metodi activateIn ritornano oggetti della classe *Activity*. Questa classe è vitale in Swarm. Sul piano pratico, risponde a metodi come run, stop, next, terminate, ecc.
Ma generalmente, non userete spesso questa classe *Activity* direttamente. Vedremo nei *GUISwarms* il *ControlPanel* che vi permetterà di gestire la simulazione dal mouse, e che nasconde gli oggetti della classe *Activity*.

5.3 Schedules dinamici

Un'altra caratteristica di Swarm, che però non vedremo nei nostri esempi semplici, è la possibilità di avere schedules dinamici. Troverete esempi nell'applicazione *Mousetrap*, per esempio.
In poche parole, per avere uno schedule dinamico, si crea l'oggetto `schedule` nel metodo buildActions (come abbiamo fatto per il `modelSchedule` nel nostro piccolo programma), ma si lascia vuoto. Poi, si scrive un metodo per specificare le azioni nello schedule, queste azioni possono dipendere dai risultati della simulazione. In questo modo, lo schedule diviene dinamico: le azioni eseguite in ogni periodo dipendono dallo stato della simulazione in questo periodo.

6 IMPORTARE PARAMETRI DA FILES

In questa sessione, salviamo i parametri necessari all'inizializzazione del *modelSwarm* in un file `parameters.scm`. Il vantaggio di questo è che diviene più semplice modificare questi parametri, perchè sono in un file separato (invece di cercarli nel `ModelFile.m` o `ModelFile.java`. Un altro vantaggio è che non c'è bisogno di ricompilare il programma se lo vogliamo far girare con nuovi valori dei parametri. Basta cambiare il file `parameters.scm`, salvarlo, e far girare il programma (con `./market` in ObjectiveC o `javaswarm StartMarket` in Java).
Nella documentazione di Swarm, troverete i metodi e documenti di aiuto sotto la voce 'serialization'. La classe che si usa per importare/salvare parametri si chiama *LispArchiver*
Swarm può importare e salvare dati in formato *Lisp* (un formato 'human-readable') o in formato *HDF5* (un formato binario, di tipo database, che può essere letto da programmi statistici tipo *R* (`http://www.ci.tuwien.ac.at/R/`) o *S-Plus* (`http://www.mathsoft.com/splus/`)).
NOTATE: in alcune applicazioni, vedrete i files importati usando le classi *ObjectLoader* e *ObjectSaver*, che prendono e producono semplici files in formato *ascii*. Il vostro programma funzionerà con queste classi, ma avrete un sacco di 'warnings' durante il running della simulazione. Infatti, l'uso di queste classi è adesso deprecato in Swarm, e possibile per adesso solo per ragioni di compatibilità.

6.1 Il file di dati

Listing 1: Il file parameters.scm

```
(list
    (cons 'modelSwarm
        (make-instance 'ModelSwarm
                    #:modelTime 0
                    #:maxTime 5
                    #:numberOfConsumers 3
                    #:startBudget 0 ;no endowment
                    #:maxBudget 10
                    #:notFinished 1 ))
)
```

È scritto in formato *Lisp*. Commincia con una dichiarazione di una `list`. Il secondo argomento, `cons 'modelSwarm` è la chiave alla quale appartengono i parametri, che corrispondono ad un instance della classe *ModelSwarm* (introdotta dalla keyword `make-instance`).
Ogni parametro si trova su una riga che commincia con `#:` seguito dal nome del parametro e del valore. Uno `;` introduce un commento.
Notate che i parametri nel file devono corrispondere a parametri dichiarati nel corrispondente file `.h` in ObjectiveC, o all'inizio della classe in Java.
Se volete passare altri parametri per un'altra classe (diciamo un *ObserverSwarm*) senza scrivere un nuovo file `.scm`, dovete:

1. togliere una `)` dall fine del file
2. aggiungere un blocco del tipo

```
(cons 'observerSwarm
    (make-instance 'ObserverSwarm
                #:myParameter 45
                #:myProbability 0.0F0 ;segnala un numero di tipo float
                )))
```

Finalmente, dato che questo file è scritto in *Lisp*, non ci sono differenze fra il file `.scm` scritto per una simulazione in ObjectiveC, e un file `.scm` scritto per una simulazione in Java.
Vediamo adesso cosa cambia nel nostro programma. Dato che cambiano solo cose nel metodo main() e nel *modelSwarm*, includiamo solo le parti rilevanti del programma.

6.2 Cambiamenti nel programma in ObjectiveC

Ci sono cambiamenti nel file `main.m`, e nel metodo createBegin nel file `modelSwarm.m`. Dunque, il file `main.m` diviene: import the file main.m

Listing 2: Il file main

```
// main.m

#import "ModelSwarm.h"

int main(int argc, const char ** argv)
    {
    ModelSwarm * modelSwarm;
    id <LispArchiver> archiver;
```

```
    initSwarm(argc, argv);

// create the modelSwarm
// load the parameters from file: first create an archiver instance,
// then load the parameters, with a check that the file exists, or that
// the key (modelSwarm) is in the file. The LispArchiver also uses
// the createBegin and createEnd methods of the modelSwarm
    archiver=[LispArchiver create:globalZone setPath:"parameters.scm"];
    if((modelSwarm=[archiver getWithZone:globalZone key:"modelSwarm"])
        ==nil)
        {
        raiseEvent(InvalidOperation,"can't find file or key\n");
        }
    [archiver drop];

// set the seed of the random generator so results can be reproduced
// (this is facultative)
    [randomGenerator setStateFromSeed:100000];
    [modelSwarm buildObjects];
    [modelSwarm buildActions];
    [modelSwarm activateIn: nil];
    [[modelSwarm getActivity] run];
    return 0;
    }
```

- all'inizio, si crea un nuovo oggetto `archiver` di tipo *LispArchiver*
- si inizializza `archiver`, dandogli informazioni su una zona della memoria e la locazione del file dei parametri:

  ```
  archiver=[LispArchiver create:globalZone setPath:"parameters.scm"];
  ```

- finalmente, si chiede all'`archiver` di creare un instance dell'oggetto per il quale abbiamo parametri (il *modelSwarm*), usando la chiave apposita:

  ```
  if((modelSwarm=[archiver getWithZone:globalZone key:"modelSwarm"])==nil)
      {
      raiseEvent(InvalidOperation,"can't find file or key\n");
      }
  ```

  La condizione `if` permette di individuare i casi nei quali il file `.scm` non esiste, o non contiene la chiave richiesta. In questo caso, la compilazione si ferma con il messaggio d'errore indicato.
  È importante notare che il metodo getWithZone:key instanzia il *modelSwarm*, dunque, usa direttamente i metodi createBegin e createEnd del *ModelSwarm*.

L'atro cambiamento nel programma si trova appunto nel metodo createBegin della classe *ModelSwarm*, dove abbiamo cancellato gli statements corrispondenti alla creazione degli parametri. Il metodo createBegin diviene:

Listing 3: ObjectiveC: Il metodo createBegin del ModelSwarm

```
+createBegin: (id) aZone
    {
    ModelSwarm * obj;

// call the createBegin method of the superClass
    obj = [super createBegin: aZone];
    return obj;
    }
```

6.3 Cambiamenti nel programma in Java

Tranne i cambiamenti di sintassi sono gli stessi di quelli della versione in ObjectiveC. Dunque, il metodo main() della classe *StartMarket* diviene: import the file main.m

Listing 4: Java: Il file StartMarket.java

```
// StartMarket.java

import swarm.Globals; // no # but a ; at the end
import swarm.defobj.ZoneImpl;
import swarm.defobj.LispArchiverImpl;

//import "modelSwarm.java";

public class StartMarket
    {
    public static void main(String[] args) // the main function MUST be
          'void'
        {
// declare modelSwarm
        ModelSwarm modelSwarm;

        LispArchiverImpl archiver;

// initialise Swarm: need the 4 strings!!!
        Globals.env.initSwarm ("market","2.1.1","bissey@sp.unipmn.it",
            args);

// create the archiver object
        archiver=new LispArchiverImpl(Globals.env.globalZone,"
            parameters.scm");

// import the parameters (note the casting as ModelSwarm). The archiver
// will also use automatically the constructor for the ModelSwarm
        modelSwarm=(ModelSwarm)archiver.getWithZone$key(Globals.env.
            globalZone,"modelSwarm");

// get the simulation running
        Globals.env.randomGenerator.setStateFromSeed(100000);
        modelSwarm.buildObjects();
        modelSwarm.buildActions();
        modelSwarm.activateIn(null);
```

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

Da notare:

- dobbiamo importare la libreria `swarm.defobj.LispArchiverImpl`, mentre non era necessario importare nuove librerie in ObjectiveC
- l'oggetto `archiver` è del tipo `LispArchiverImpl`
- è possibile omettere il secondo argomento del costruttore LispArchiverImpl() (che corrisponde alla locazione del file dei parametri). Il default è che il file da importare abbia lo stesso nome dal primo argomento del metodo initSwarm(), con l'estensione `.scm` (dunque, nel nostro caso, il file dovrebbe chiamarsi `market.scm`)

Il costruttore ModelSwarm() della classe *ModelSwarm* diviene:

Listing 5: Java: Il costruttore del ModelSwarm

```
public ModelSwarm (Zone aZone)
    {
    super(aZone);
    }
```

7 L'INTERFACCIA GRAFICA DI SWARM

7.1 GUISwarms e sonde (probes)

L'*ObserverSwarm* è la classe che ci permette di osservare e controllare l'andamento della simulazione. La caratteristica principale è il *ControlPanel*, che contiene 5 bottoni

⇒ start Inizia la simulazione, e si ferma alla fine

⇒ stop Pausa la simulazione

⇒ next Avanza la simulazione di un'iterazione

⇒ save Salva la disposizione delle finestre (se esiste lo statement apposito nel programma)

⇒ quit Esce dalla simulazione (anche se non finita)

Le funzioni dell'*ObserverSwarm* ci permettono di creare delle sonde (*Probes*) per vedere i valori dei parametri degli oggetti della simulazione. Abbiamo la possibilita di cambiarli prima di lanciare la simulazione (basta cambiare il valore nella finestra e premere il tasto 'return'). Questo ci fa anche vedere bene i vantaggi della programmazione con gli oggetti: non è più difficile fare girare il nostro programma con 3 o 10 agenti!
Finalmente, nei *GUISwarms*, possiamo creare grafici usando i risultati della simulazione. Vedremo in questa sessione gli *EZGraphs* che permettono di creare grafici semplici. Nel par. 8, pagina 31, vedremo come possiamo anche gestire lo spazio (il mondo) dei nostri consumatori, e come rappresentarlo.
Per implementare l'*ObserverSwarm*, dobbiamo dunque creare una nuova classe (cioè due nuovi

files in ObjectiveC, un nuovo file in Java), con le caratteristiche dell'*ObserverSwarm*: i suoi parametri, le loro sonde, i grafici, e lo schedule dell'*ObserverSwarm*.
Di consequenza, ci sono stati parecchi cambiamenti anche negli altri files. Li commenteremo fra poco, ma possiamo riassumere quelli principali adesso:

- L'*ObserverSwarm* è lo Swarm più elevato nella gerarchia. Dunque, diviene quello inizializzato nella funzione main(). Il *ModelSwarm* sarà creato e inizializzato nell'*ObserverSwarm*.
- Nell'*ObserverSwarm*, creiamo una sonda anche per il *ModelSwarm*. Nel *ModelSwarm*, vedremo come personalizzare questa sonda in modo tale che faccia vedere solo i parametri che ci interessano (e che vogliamo lasciare cambiare all'utente).
- Troverete alcuni nuovi metodi 'get', per passare dati da un livello della gerarchia all'altro.

7.2 Cambiamenti del programma

7.2.1 Nel Makefile

Dobbiamo segnalare al `Makefile` l'esistenza della nuova classe. In Java, basta scriverla sulla prima linea. In ObjectiveC, dobbiamo dichiarare un nuovo oggetto `ObserverSwarm.o`, e segnalare i legami con le altre classi.

7.2.2 Nel file dei parametri

Abbiamo aggiunto un nuovo elemento nella lista, che corrisponde alla chiave `'observerSwarm`. Contiene 2 parametri: `displayFrequency` controlla quando aggiorniamo i grafici (qui, è 1, cioè ogni iterazione); `displayConsumerName` definisce il consumatore che viene rappresentato nel grafico `consumerGraph`.

7.2.3 Nella funzione main()

Il cambiamento principale è che creiamo l'*ObserverSwarm*, invece del *ModelSwarm*. Tranne questo, troverete 2 novità:

- la macro SET_WINDOW_GEOMETRY_RECORD_NAME(observerSwarm) in ObjectiveC, e Globals.env.setWindowGeometryRecordName(observerSwarm,"observerSwarm") in Java. Permettono di salvare la posizione della finestra delle sonde dell'*ObserverSwarm* sullo schermo (usanto il tasto 'save' del *ControlPanel*).
- Alla fine di main(), diamo al *ObserverSwarm* il comando `go` (invece di `run` per il *ModelSwarm*). Questo trasferisce l'andamento della simulazione al *ControlPanel*.

7.2.4 Descrizione dell'ObserverSwarm

- Il primo metodo è il +createBegin, che viene chiamato automaticamente dal *LispArchiver* quando l'instance `observerSwarm` è creata nella funzione main().
  In questo metodo, viene personalizzata la *ProbeMap*. La *ProbeMap* stessa viene creata nel metodo buildObjects, e se non facciamo nulla, mostra tutti i parametri e i metodi dell'*ObserverSwarm*. Qui, usiamo una *ProbeMap* personalizzata, che fa vedere solo i due parametri dell'*ObserverSwarm*.
  Dunque, creiamo la variabile `probeMap` come un instance di una *EmptyProbeMap* (una

mappa di sonde vuota), nella classe corrente (`self` in ObjectiveC, getClass() in Java).
Per aggiungere i parametri che vogliamo mettere nella `probeMap`, usiamo il metodo addProbe, che prende una sonda come argomento. La sonda si crea con una chiamata alla *probeLibrary*, con il metodo getProbeForVariable:inClass in ObjectiveC (getProbeForVariable$inClass in Java). Il primo argomento è il nome della variabile fra `" "`, il secondo è un riferimento alla classa corrente (`[self class]` o getClass()).
Una volta che ci sono tutte le variabili nella `probeMap`, la aggiungiamo nella *probeLibrary* con il metodo setProbeMap.

- Il metodo buildObjects è, come sempre, usato per costruire gli oggetti necessari a questo livello della simulazione. Nel nostro caso, creiamo il *ModelSwarm* (e li chiediamo di creare i suoi oggetti), le sonde, e i grafici.
  - Per creare le sonde, usiamo il metodo CREATE_ARCHIVED_PROBE_DISPLAY(className), o Globals.env.createArchivedProbeDisplay(className,"windowName").
  - Per creare un *EZGraph* in ObjectiveC, usiamo i metodi createBegin/createEnd del grafico. Fra questi metodi, inizializziamo alcune caratteristiche nel grafico: il titolo (setTitle:) e i nomi degli assi (setAxisLabelsX:Y).
    Poi, segnaliamo al grafico cosa deve rappresentare, usando i metodi:
    * createSequence:"nameOfSequence" withFeedFrom:simulationObject andSelector:M(methodReturningTheVariableToDraw)
    * createTotalSequence:withFeedFrom:andSelector, dove il `simulationObject` è una collezione, e viene rappresentata la somma delle variabili per tutti gli oggetti della collezione
    * createAverageSequence:withFeedFrom:andSelector, per rappresentare la media della variabile
    * createMinSequence:withFeedFrom:andSelector, per rappresentare i valori minimi della variabile
    * createMaxSequence:withFeedFrom:andSelector, per rappresentare i valori massimi della variabile
  - Per creare un *EZGraph* in Java, troviamo poche differenze con ObjectiveC:
    * Il costruttore rendre la creazione del grafico più semplice, includendo tutti gli elementi (assi, titolo, ecc) nello stesso commando:
      ```
      spendingGraph=new EZGraphImpl(getZone(),"Agents' spending",
              "Timex2","Spending","spendingGraph");
      ```
    * I metodi createSequence richiedono l'uso di *Selector*, e devono essere dichiarati fra un blocco `try/catch`.

- Nel metodo buildActions, definiamo le azioni della simulazione, e il loro schedule. Chiediamo prima alla *superClass* e al `modelSwarm` di costruire le loro azioni. Poi definiamo le azioni dell'*ObserverSwarm*. Prima, mandiamo ai grafici il messaggio di eseguire il loro metodo step (che combina il metodo update e il metodo outputGraph). Poi, chiediamo all'*ObserverSwarm* di controllare se deve fermarsi o no. Finalmente, mandiamo all'*actionCache* il messaggio M(doTkEvents), che gestisce gli oggetti grafici della simulazione.
  Queste azioni si trovano in un *ActionGroup*, `displayActions`, che viene poi messo nel `displaySchedule`

- Dopo l'activateIn, che è simile a quelli che abbiamo visti prima, viene definito il metodo observerCheckToStop, che controlla il metodo checkToStop del *ModelSwarm*. Se la simulazione è finita, chiediamo al *ControlPanel* di fermarsi (setStateStopped). Possiamo poi uscire della simulazione premendo il tasto 'quit'. Non usiamo il metodo terminate usato prima, perchè ci porterebbe al prompt di Swarm, chiudendo tutte le finestre della simulazione (il che non ci permette di osservare i grafici, per esempio).

7.2.5 Nel ModelSwarm

- Come abbiamo fatto nell'*ObserverSwarm*, usiamo una sonda personalizzata per far vedere i parametri che possono essere cambiati prima di far girare la simulazione.
- Il metodo checkToStop è stato modificato per ritornare un intero, che viene passato all'*ObserverSwarm* per controllare la fine della simulazione.
- Abbiamo aggiunto tre metodi par passare all'*ObserverSwarm* alcuni oggetti: getListOfConsumers ritorna la lista dei consumatori; getConsumer:name ritorna il consumatore di nome 'name'; getCurrentTime ritorna il `modelTime`

7.2.6 Nello Schedule

Adesso, nella simulazione, facciamo 2 cose: osserviamo i consumatori che vanno al mercato, e disegniamo alcuni risultati. Per questo abbiamo cambiato lo *Schedule* in modo tale che, in ogni ripetizione della simulazione, troviamo chi val al mercato e quanto spende in un primo tempo, e in un secondo tempo, disegniamo i risultati. Quindi vedete che il `repeatInterval` dello `modelSchedule` e dello `displaySchedule` è adesso fissato a 2. Al tempo 0, si esegue il `modelSchedule`, mentre al tempo 1, si esegue il `displaySchedule`.
L'unico svantaggio di questo tipo di schedule si vede nei labels orizzontali degli *EZGraph*, che rappresentano il tempo in termini di azioni nella simulazione. Quindi, con un `repeatInterval` uguale a 2, comparono nei grafici 2 periodi di tempo per un unità di `modelTime` (per questo, il label è stato cambiato da 'Time' a 'timex2'.

7.2.7 Nel Consumer

Poche cose sono cambiate in questa classe.

- Abbiamo aggiunto una variabile `currentTime`, che viene inizializzata nel metodo updateVisits. Serve per i metodi getVisit e getSpending
- I metodi getVisit e getSpending servono a ritornare risultati dei metodi getVisitValue e getSpendingValue rispettivamente, corrispondenti al valore della variabile `currentTime`. Questi metodi sono necessari, perchè il metodo M() nel metodo createSequence nei grafici non prende argomenti. Usiamo dunque i metodi getVisit e getSpending nel `consumerGraph` dove dobbiamo dire al metodo createSequence di prendere nel vettore o nella mappa il valore corrispondente all'ultima iterazione della simulazione.

8 METTERE GLI AGENTI IN UNO SPAZIO

Usiamo questa sessione per presentare un altro grafico: il *ZoomRaster*, che permette di organizzare gli agenti in uno spazio, e di osservare le loro interazioni, e come si muovono. Per questo, abbiamo

dovuto cambiare i files del consumatore, del modelSwarm e dell'observerSwarm. Abbiamo aggiunto i metodi per creare e disegnare il *ZoomRaster*, e per gestire la posizione degli agenti nello spazio, e i loro movimenti. Presentiamo prima i nuovi programmi, in ObjectiveC e Java, poi alcuni dettagli dei cambiamenti nei files.

8.1 Cambiamenti nel programma

8.1.1 Nell'ObserverSwarm

Abbiamo aggiunto le variabili per creare il "raster" e gli elementi da far vedere. Nel nostro caso, vogliamo un fondo nero, con lo spazio riservato al mercato materializzato con un riquadro giallo, nel centro del grafico. All'inizio, tutti i consumatori sarano fuori del mercato, rappresentati con punti rossi. Ad ogni iterazione, cambieranno posto: se vanno nel mercato, sarano rappresentati nello spazio del mercato, e diventerano azzuri; altrimenti, sarano rappresentati nel resto del mondo, sempre in rosso.
Questo si fa nel buildObjects del *ObserverSwarm*:

⇒ zoomFactor questa variabile determina la dimensione di un 'punto' nel raster

⇒ colorMap determina i colori che usiamo nel raster. Ci sono diversi modi di definirla. Qui abbiamo usato quello più semplice, che associa un codice (per esempio 0) al nome di un colore (per esempio `"black"`). Altri modi, che permettano di avere colori più 'raffinati' usano la quantità (da 0 a 255) di azzuro, verde e rosso nel colore

```
[colorMap setColor:1 ToName:"blue"];
[colorMap setColor:4 ToRed:255 Green:255  Blue:255]; // per bianco
[colorMap setColor:backGroundColor ToRed:0 Green:0  Blue:0]; // per nero
```

Una cosa da notare è che in Java, i codici dei colori sono del tipo `byte`, che deve essere precisato, se usate un codice numerale

```
colorMap.setColor$ToName((byte)0,"black");
```

⇒ worldRaster definisce lo spazio, che è un instance della classe *ZoomRaster*. I metodi setColormap:, setZoomFactor:, setWidth:Height:, setWindowTitle: permettono di definire le sue caratteristiche. Il metodo pack lo inizializza e lo fa vedere (vuoto) sullo schermo.

⇒ marketDisplay quest'instance della classe *Value2dDisplay* gestisce la rappresentazione del mercato nello spazio.
Deve sapere dove si deve disegnare (il `worldRaster`), e quali colori usare (setDisplayWidget:colormap:), e cosa rappresentare (il `market`, definito nel modelSwarm) (setDiscrete2dToDisplay:)
Come spesso, in Java, esiste un costruttore che rende più corta la dichiarazione dell'instance:

```
marketDisplay=new Value2dDisplayImpl(getZone(),worldRaster,
        colorMap,modelSwarm.getMarket());
```

⇒ worldDisplay gestisce il display dei consumatori nello spazio. I metodi necessari sono setDisplayWidget: (per indicare che si deve disegnare nel `zoomRaster`), setDiscrete2dToDisplay: (per indicare che deve disegnare il `world` definito nel *ModelSwarm*), setObjectCollection:

(per indicare che gli oggetti da disegnare sono gli consumatori contenuti nella `listOfConsumers`), setDisplayMessage:M() (per indicare che si deve usare il metodo drawSelfOn: dei consumatori).
Il costruttore in Java di nuovo semplifica la creazione del `worldDisplay`. Notate che contiene un *Selector* (dovuto al messaggio per il metodo drawSelfOn), e che quindi deve essere incluso in un blocco `try/catch`.

```
worldDisplay=new Object2dDisplayImpl(getZone(),worldRaster,
    modelSwarm.getWorld(),
    new Selector(Class.forName("Consumer"),"drawSelfOn",false));
worldDisplay.setObjectCollection(modelSwarm.getListOfConsumers());
```

⇒ sonde per gli elementi nel raster finalmente, chiediamo al `worldRaster` di creare sonde per gli agenti del `worldDisplay`, quando viene premuto il bottone destro del mouse (`ButtonRight` in ObjectiveC, e 3 (terzo bottone) in Java)

⇒ display del raster chiediamo poi al raster di disegnarsi con tutti i suoi elementi, quando inizia il programma:

```
worldRaster.erase();
marketDisplay.display();
worldDisplay.display();
worldRaster.drawSelf();
```

Nel buildActions dell' *ObserverSwarm*, aggiungiamo al `displayActions` i messaggi per disegnare il raster (gli stessi di sopra).

8.1.2 Nel ModelSwarm

Di nuovo, i cambiamenti principali si trovano nel metodo buildObjects. Le dimensioni del mondo (`worldWidth`, `worldHeight`) e la dimensione del mercato (che è quadrato), sono parametri del file `parameters.scm`, e possono anche essere modificati nella sonda per il *ModelSwarm* all'inizio della simulazione.

- Definizione della posizione del mercato al centro dello spazio: si creano le variabili `xMin`, `xMax`, `yMin`, `yMax`, che definiscono i limiti del mercato nello spazio.
- Creazione della variabile `world`, un instance della classe *Grid2d*, che è inizializzata con oggetti `nil` (o `null` in Java), in attesa di contenere i consumatori.
- Creazione del `market`, un instance della classe *Discrete2d*, nella quale mettiamo, fra i limiti del mercato, il valore '2' (che corrisponde al colore giallo nella `colorMap`).
- Aggiungiamo alla creazione dei consumatori la definizione della loro posizione iniziale nello spazio, usando il metodo findPositionInWorld:For:ExcludeMarket:. Siccome sono stati appena creati, e non hanno ancora gioccato, il valore per `ExcludeMarket` è 1 (dunque, non li mettiamo nel mercato).
- Creiamo anche una sonda per i consumatori, che fa vedere il loro nome.

La posizione dei consumatori nello spazio viene determinata dal metodo findPositionInWorld:For:ExcludeMarket:. L'idea è di trovare una posizione a caso, che sia nella parte dello spazio voluta (cioè nel mercato o fuori), in un punto in cui non c'è già qualcuno. Prima di tutto, si cancella il consumatore dallo spazio (usando l'oggetto `nil` o `null`), poi si cerca la nuova posizione. I valori di `trialX` e `trialY` adeguati vengono entrati nelle caratteristiche del consumatore, usando il metodo setPositionX:Y:. Il consumatore è entrato nel `world` in questa posizione (putObject:atX:Y:).
L'ultimo cambiamento si trova nel metodo marketDay, che consente anche di modificare la posizione degli agenti nello spazio in base alla loro partecipazione o meno al mercato.

8.1.3 Nel Consumer

Abbiamo aggiunto una variabile `marketGoer`, che viene inizializzata a 0 durante la creazione dell'agente, e che poi prende il valore di `k` del metodo goToTheMarket.
I consumatori ricordano la loro posizione nello spazio usando il metodo setPositionX:Y:, e la comunicano attraverso i metodi getPositionX e getPositionY.
Un metodo fondamentale è drawSelfOn: che gestisce la loro rappresentazione sul `worldRaster`, dandogli informazione sulla posizione del consumatore nello spazio, e il colore nel quale deve essere disegnato.
Nel nostro caso, questo colore è determinato dal metodo getStrategyColor, che assicura che i consumatori che non vanno al mercato sono disegnati in rosso (colore 3), e che quelli che vanno al mercato sono disegnati in azzuro (colore 1).

9 ESPERIMENTI

In questa sessione, vediamo come far girare un esperimento in Swarm. Per esperimento, intendiamo che la simulazione viene eseguita più volte, con parametri che cambiano. Nel nostro piccolo esempio, il parametro che cambia è `startBudget`. Finora, era fissato a 0. Adesso, lo facciamo cambiare da 0 a 10, con incrementi di 1.
Il risultato dell'esperimento è la spesa media per tutti i consumatori, in ogni ripetizione della simulazione.
In questo programma, viene introdotta un nuovo Swarm: *ExperSwarm*, che è un GUISwarm di alto livello. Per questo, non può convivere in una simulazione con l'*ObserverSwarm*. Dunque, l'*ObserverSwarm* è scomparso.
Come prima, il *ModelSwarm* viene creato nello Swarm di alto livello, cioè qui l'*ExperSwarm*. Il *modelSwarm* è creato in un ciclo (loop) con alcuni parametri che cambiano durante l'esperimento, e distrutto alla fine di ogni iterazione. La nuova classe *SimulationData* gestisce i parametri dell'esperimento e come vengono passati al *ModelSwarm*.

9.1 Cambiamenti nel programma

Spiego i cambiamenti principalmente usando il programma scritto in ObjectiveC. Il programma scritto in Java è quasi uguale, tranne la parte per rappresentare graficamente i risultati, che viene spiegata nel par. 9.1.4, pagina 36.

9.1.1 2 nuove classi: ExperSwarm e SimulationData

ExperSwarm gestisce l'esperimento e il display dei risultati. *SimulationData* gestisce il passare dei dati fra *ExperSwarm* e *ModelSwarm*.

L'*ExperSwarm* diviene lo Swarm di livello piú alto, e prende il posto dell' *ObserverSwarm*. É anche lui un *GUISwarm*, quindi abbiamo ancora il ControlPanel, e la possibilità di rappresentare risultati graficamente.

ExperSwarm: il metodo createBegin: si crea la class *ExperSwarm*, e si definisce la `probeMap`, con le variabili che possono essere cambiateprima di far girare la simulazione.

ExperSwarm: il metodo buildObjects: si crea il display della probeMap, e si crea il grafico per far vedere alcuni risultati. Nel caso dell'esperimento, vogliamo disegnare la spesa media di tutti i consumatori ad ogni ripetizione (cioè per i 10 periodi della simulazione). In questo caso, non usiamo l'*EZGraph*, ma l'oggetto *Graph*, che è molto più flessibile. Definiamo prima le caratteristiche della finestra grafica (titolo, larghezza, altezza, nomi degli assi).
Poi creiamo l'oggetto *spending*, di tipo *GraphElement*. Dentro, salveremo i dati da disegnare sul grafico. *spending* viene dunque creato come un 'elemento' di *spendingGraph*. Gli diamo un nome (setLabel:) e un colore (setColor:).

ExperSwarm: il metodo run: dov'è definito l'andamento dell'esperimento. Il ciclo `for` definito all'inizio del metodo fa girare la simulazione (il *ModelSwarm*) per ogni valore possibile del bilancio iniziale, da `minStartBudget` a `maxStartBudget` con incrementi di `incStartBudget`.
I valori di queste tre variabile appaiono nella probeMap dell'*ExperimentSwarm* e possono essere cambiati prima di far girare l'esperimento. La variabile `setStartBudget` contiene il valore corrente del bilancio iniziale.
La prima cosa da fare è di salvare `setStartBudget` in un file di parametri (`loop.scm`) che sarà poi importato dal *ModelSwarm* via il *LispArchiver*. Per questo, abbiamo creato una nuova classe *SimulationData*, che trasforma, tramite il metodo initPara:, `setStartBudget` in un oggetto `simLoop`, che contiene il suo valore, e il suo nome nel *ModelSwarm* (`startBudget`).
`outFile` è un *LispArchiver*, che viene usato per salvare `simLoop` in un file lisp, usando i metodi putShallow:object e sync per creare il file `loop.scm`.
Una volta definito il parametro dell'esperimento, creiamo il *modelSwarm*, caricando i parametri del file `parameters.scm`.
Poi carichiamo i parametri della simulazione (dal file `loop.scm`), e li integriamo al *modelSwarm* usando il metodo setPara della classe *SimulationData*. È importante che questo venga eseguito dopo l'inizializzazione del *modelSwarm* (dai parametri di `parameters.scm`), perchè i parametri dell'esperiment cambiano spesso i valori di alcuni parametri iniziali della simulazione. Per esempio, nel file `parameters.scm`, la variabile `startBudget` ha un valore di 0, mentre nell'esperiment, il valore di `startBudget` cambia ad ogni iterazione.
Possiamo adesso cancellare gli oggetti `outFile`, `simLoop` e `archiver` che non servono più a niente. È utile ripetere che cancellare (drop) oggetti inutili permette di conservare più spazio in memoria.
Adesso, il *modelSwarm* è stato creato, e gli chiediamo di costruire i suoi oggetti (buildObjects) e le sue azioni (buildActions). Poi lo facciamo girare normalmente, finchè non si fermi (nel nostro esempio, il *modelSwarm* contiene 3 consumatori, e gira per 10 periodi). Una volta girato in *modelSwarm*, calcoliamo la spesa media per tutti i consumatori in tutti i periodi del *modelSwarm*. Aggiungiamo questo risultato all'oggetto `spending`, che sarà poi disegnato sul grafico `spendingGraph`.
Una volta terminato il ciclo sui valori di `setStartBudget`, chiediamo al grafico di disegnarsi (`[spendingGraph pack]`, e mettiamo il *ControlPanel* in stato di pausa.

La classe SimulationData: è una classe scritta aposta per l'esperimento. Serve a trasformare in oggetti i parametri che cambiano ad ogni iterazione dell'esperimento, in modo da poter salvarli in un file di tipo lisp (metodo initPara). Il metodo setPara: invece, prende come argomento la classe nella quale sono caricati i parametri (nel nostro caso, l'argomento del metodo setPara: sarà *modelSwarm*).

9.1.2 Cambiamenti nel ModelSwarm

Poche cose sono cambiate qui. Abbiamo creato 2 nuovi metodi: setSimulationParameters prende il parametro dell'esperimento come argomento, e viene usato nel metodo setPara della classe *SimulationData*. Il metodo getAllSpending permette di mettere in una lista tutte le spese di tutti i consumatori in ogni periodo della simulazione. Il risultato viene usato per compilare la spesa media che viene disegnata alla fine dell'esperimento.
Vedrete anche alcuni cambiamenti nel metodo marketDay, dovuti al fatto che abbiamo sostituito la `mapOfSpending` con una `listOfSpending`, per comodità del grafico finale.

9.1.3 Cambiamenti nella classe Consumer

Anche qui sono pochi. Il nuovo metodo getAllSpending permette di riportare tutte le spese del consumatore durante la simulazione. Gli altri cambiamenti sono dovuti alla sostituizione della `mapOfSpending` con la `listOfSpending`. Vengono modificati dunque i metodi createMapOfSpending (che diventa createListOfSpending, updateSpending (che prende solo il valore della spesa come argomento, e l'aggiunge all'inizio della lista) e getSpendingValue (che non ha più argomenti: basta prendere il valore del primo elemento della lista).

9.1.4 Grafici in Java

Nella versione dell'*ExperimentSwarm* in Java, non è possibile usare il grafico come spiegato nella versione in ObjectiveC. La versione standard di Swarm in Java contiene infatti solo pochi grafici perchè il linguaggio Java è ricco di applicazioni grafiche che possono essere incluse direttamente nel programma. *Ptplot* di Ptolemy è un'esempio (`http://ptolemy.eecs.berkeley.edu/java/ptplot/`).
In questo caso, dovrete includere una voce `CLASSPATH=` nel `Makefile`, per indicare al compilatore dove trovare la classe.
Non ho avuto molto tempo per provare questi, quindi, nella versione Java dell' ExperimentSwarm, non ci sono grafici!.

10 MESSAGGI DI ERRORE E ALTRI DETTAGLI

10.1 Cose da sapere

Sono caratteristiche di Swarm, o di C, che possono creare confusione.

- In C e Java, la divisione di 2 interi dà un intero. Per esempio, dividere $50/100$ vi dà 0, non 0.5. La soluzione in Swarm è di assicurarsi che il risultato sia un float (o un double): `result=(float)50/100;` dà 0.5.
- Supponiamo che volete trovare un numero di agenti, usando la popolazione totale (un intero) e una proporzione (un float). Il risultato è un float, ma volete un intero. In C, `(int)proporzione*popolaz`

dà come risultato l'intero *inferiore* più vicino. Per esempio, $57*0.8 = 45.6$ ma `(int)57*0.8;` ha come risultato 45. Per avere 46, dovete scrivere: `(int)(57*0.8)+0.5;`

- I files `.tar.gz` in Windows: sono files zippati, ed è un formato molto usato per gli esempi di programmi di Swarm che troverete in rete. Avete 2 possibilità per aprirli:

⇒ in `WinZip`: può darsi che vi chieda di "entrare il nome completo del file contenuto nell'archivio". Basta generalmente aggiungere `tar` al nome suggerito da `WinZip`, e rispondere 'si' alla domanda seguente di espanderlo e aprilo.

⇒ nel terminale di Swarm: basta dare il comando:

```
tar xzvf tarfile.tar.gz
```

dove `tarfile.tar.gz` è il nome del file.

10.2 Messaggi d'errore di ObjectiveC

10.2.1 Errori di compilazione

Questi errori sono 'semplici'. Swarm si ferma, ci dice il file dove ha trovato un errore, e la riga dove c'e l'errore. Al solito, ci fa vedere molti errori, ma badare al primo è generalmente la soluzione a tutti.

Errori:

⇒ 'methodName' undeclared (first use of this function)
State provando a usare un metodo che Swarm non riconosce. La soluzione:

1. verificare l'ortografia: forse methodName si scrive mEthodName?
2. verificare che è stato importato il file header dove è definito methodName (sia un file header di Swarm o uno vostro).

⇒ 'variableName' undeclared (first use of this function)
Come sopra, dovete verificare l'ortografia di `variableName`, se è stato importato il file header, oppure, se la variabile è stata dichiarata nel file header.

⇒ parse error before 'something'
Il più probabile: manca un `;`, una `(`, un `]` subito prima di '`something`' o alla riga precedente.

⇒ cannot find protocol declaration for 'ProtocolName'
Avete dichiarato un oggetto come: `id <ProtocolName> ObjectName` ma Swarm non trova nessun protocollo con il nome ProtocolName

1. controllate l'ortografia!
2. avete importato il file header giusto?
3. esiste il protocollo ProtocolName?

Warnings: Il default per Swarm è di considerare anche i warnings come errori. In alcuni casi, il programma funzionerebbe comunque, ma Swarm scopre una fonte di confusione, o un utilizzo della memoria inutile; lo fa notare, e si ferma.

⇒ warning:local declaration of 'variableName' hides instance variable
Nello stesso file, avete dichiarato come variabile `variableName`. State anche usando `variableName` nella dichiarazione di un metodo.
La soluzione: cambiare uno dei nomi.

⇒ warning:ModelSwarm does not respond to 'methodName::'
State provando a usare un metodo in un modo diverso da quello definito.

1. controllate l'ortografia!
2. controllate la dichiarazione nel file header
3. controllate che sia chiamato con lo stesso numero di parametri della dichiarazione

⇒ warning: method 'methodName' not implemented by protocol
State provando a usare un metodo non dichiarato in un protocollo

⇒ warning: control reaches end of non-void function
Avete dichiarato un metodo che deve ritornare qualcosa, ma Swarm non ha trovato il comando return

1. se il metodo deve ritornare qualcosa, aggiungere il return opportuno
2. se il metodo non ritorna niente, ci sono due possibilità:
   - aggiungere `return self;`
   - dichiarare il metodo come `(void) methodName:...`

⇒ warning: conflicting types for 'methodName:' (nel file .m)

⇒ warning: previous declaration of '(int)methodName:' (nel file .h)
Questi due warnings si vedono insieme: nel file .h, avete detto che methodName deve ritornare un int, ma nel file .m, è dichiarat il ritorno di un oggetto.

⇒ warning: 'variableName' might be used uninitialized in this function
Generalmente, state usando le condizioni `if` e `else if` per inizializzare la variabile. Swarm vede che la variabile non è inizializzata fuori dall' `if` e segnala che potrebbe anche non essere mai inizializzata. Soluzione: inizializzare la variabile fuori dall' `if`, oppure trasformare l'ultimo `else if` in un `else`.

⇒ warning: unused variable 'variableName'
Avete dichiarato una variabile in un metodo, ma non è mai stata usata.

⇒ warning:passing arg2 of 'methodName' makes pointer from integer without a cast
Dalla dichiarazione del methodo methodName, Swarm aspettava un oggetto come secondo argomento. Invece, trova un intero.

1. controllare il tipo di argomento richiesto dal metodo, e cambiare la dichiarazione se ha bisogno di un intero
2. se invece vuol veramente un oggetto (per esempio, methodName è usa l'argomento 2 in una lista, allora si deve usare un 'wrapper' per trasformare l'intero in un oggetto (vede par. 4.1.2, pagina 21)

10.2.2 Errori di runtime

Questi errori succedono dopo la compilazione, quando state usando il vostro programma. Non è sorprendente averli, anche se il programma ha compilato bene, perche Swarm fa molti controlli anche quando funziona. Il problema con questi errori, che hanno spesso che fare con problemi di memoria, è che sono piutosto oscuri, e non danno indicazioni su dove accadono. La soluzione per identificarli è di eseguire il programma riga per riga, oppure chiedergli di scrivere qualcosa ad ogni riga. Così, è possibile vedere l'ultimo comando che ha funzionato prima dell'errore.
Segue un elenco delle cause possibili di errori di runtime:

- Una variabile è stata dichiarata, ma non inizializzata. Per esempio, avete dichiarato una lista: `id <List> aList`, e state provando di aggiungere un oggetto alla lista `[aList addFirst:anObject]` prima di avere creato la lista con `aList=[List create:[self getZone]]`.

- State provendo a mettere un oggetto in un array, ma siete fuori delle dimensioni dichiarate per quest'array (se l'errore di runtime parla di `'offset'`, ha quasi sempre origine in un array, o in una lista).

- State usando un metodo apparamente bene, ma una delle funzione che usa non è scritta come lo aspetta. Per esempio, il metodo QSort ha bisogno di una funzione di confronto nella classe che usa. Ci sara un errore di runtime se non la trova e la funzione di default di Swarm non riesce a ordinare gli oggetti che volete ordinare. Ci sara anche un errore se la funzione di confronto esiste, ma non è scritta come QSort l'aspetta.

- Ci possono essere errori di runtime con ZoomRaster quando il `zoomFactor` è troppo grande rispetto alle dimensioni del mondo (il grafico è troppo grande), o se il `zoomFactor`, se calcolato come una funzione delle dimensioni del mondo, risulta minore di 1.

- Ci sara anche un errore di runtime se state provando a usare un metodo della classe A con un oggetto della classe B. Per esempio, nel mercato che abbiamo usato all'inizio, il metodo marketDay è del modelSwarm, non del consumer. Dunque `[modelSchedule at: 0 createActionTo: aConsumer message:M(marketDay)];` è falso, ma Swarm lo noterà solo in runtime[2]. Il comando giusto è: `[modelSchedule at: 0 createActionTo: self message:M(marketDay)];`

- Un'altra causa di errori di runtime è un 'mismatch' fra un messaggio e il metodo nella classe. Riprendendo l'esempio sopra, se scrivete: `[modelSchedule at: 0 createActionTo: self message:M(marketday)];` avrete un errore di runtime, perche Swarm non trova il metodo marketday nella classe modelSwarm (il metodo che volete è marketDay). È una caratteristica di Swarm che queste relazioni vengono controllate solo nel runtime.

10.3 Messaggi d'errore di Java

Sono di 2 tipi:

- 'syntax': errori di sintassi

- 'semantic': errori nel uso di funzioni di Java o javaSwarm, a volte collegati ad errori di sintassi.

[2] I blocchi try/catch di Java permettono di evitare questo tipo di errore di runtime, e anche quello seguente.

Ci sono anche 2 grandi differenze dai messaggi di errori di ObjectiveC:

- I messaggi di errore sono generalmenti chiari, con proposte di soluzioni. Java dà indicazioni sul file e la riga dove ha trovato un errore.
- I messaggi di errore sono collegati, nel senso che risolvere uno risolve anche gli altri. La differenza con ObjectiveC è che l'errore da risolvere non è sempre il primo indicato. Sembra una buona strategia correggere gli errori di sintassi prima.

Alcuni errori che ho trovati sono:

⇒ Error: No match was found for constructor "constructorName(list of arguments)"

⇒ Error: No match was found for method "methodName(list of arguments)"
Controllate che gli argomenti siano del tipo voluto, nell'ordine voluto, e che ci siano tutti (se il costruttore o il metodo è dichiarato per avere 4 argomenti (come *initSwarm()* per esempio),avrete questo messaggio di errore se ne usate meno (o troppi)). Notare che alcuni costruttori o metodi possono essere dichiarati con diversi numeri di argomenti (overloading).

⇒ SyntaxError: "ClassName.methodName;" : "AssignmentOperator AssignmentExpression" inserted to complete statement expression
methodName non ha argomenti, ma ha bisogno di `()` vuote. Le `()` indicano a Java che methodName è una funzione.

⇒ SyntaxError: "public type methodName": ";" inserted to complete FieldDeclaration
è un errore simile al precedente. Anche nella dichiarazione di un metodo senz'argomenti, si devono mettere le `()`.

⇒ Error: the method "java.lang.Object methodName()" must contain a return statement compatible with type Object
methodName() non è stato dichiarato come `void` ma Java non ha trovato uno statement di `return`. La soluzione è sia di cambiare la dichiarazione da un tipo `Object` a un tipo `void`, sia di aggiungere lo statement `return this;`, sia di aggiungere lo statement di return per un altro oggetto della simulazione nel caso sia stato dimenticato.

⇒ Error: The constructor "Selector" can throw the checked exception "Swarm/SignatureNotFoundException", but its invocation is neither enclosed in a try statement that can catch that exception, nor in the body of a method or constructor that "throws" this exception
Questo messaggio di errore è abbastanza chiaro (vedere par. 3.2.2, pagina 15).

Riferimenti bibliografici

Cederman, L.-E. and Stefansson, B. (1999). Slides for "programming for social scientists". Course code: POL SCI 209-1, held at UCLA.

Daniels, M., Lancaster, A., and Stefansson, B. (2000). *A Tutorial Introduction to Swarm*.

Johnson, P., Lancaster, A., and Stefansson, B. (2000). *Swarm User Guide*.

Lanton, C. and Team, S. D. (1997). *SIMPLEBUG, Swarm tutorial*.

Luna, F. and Stefansson, B., editors (2000). *Economic Simulations in Swarm: Agent-Based Modelling and Object Oriented Programming*. Kluwer Academic Publishers.

Staelin, C. P. (2000). *jSIMPLEBUG: a Swarm tutorial for Java*. based on ObjectiveC code and original text by C. Lanton and Swarm Development Team.