Strumenti della simulazione

Introduzione

Il presente capitolo ha lo scopo di fornire una panoramica sugli strumenti operativi, tra cui linguaggi di programmazione, librerie di funzioni, ambienti di sviluppo e sistemi di base. Questi si sono resi utili al fine di creare il codice informatico che permette di simulare una astrazione del funzionamento dell'impresa presa in considerazione nella prospettiva dell'analisi della complessità di cui verrà trattato nel capitolo x.

La tecnologia adottata si basa essenzialmente sui seguenti software:

1. Java Development Kit, *jdk-1.3.1_01*, come linguaggio di programmazione;
2. Swarm 2.2, libreria di funzioni scritte in Java ed ObjectiveC con lo scopo di fornire strumenti adeguati allo sviluppo della strutture di simulazione ad agenti, come accennato nel capitolo y sulla complessità;
3. Ambiente Cygwin, emulatore di macchina Unix utilizzabile sotto i sistemi operativi Windows, utile per la compilazione e la gestione delle applicazioni sviluppate;
4. Editor di testo *E-macs-21.1* con *java Development Environment-2.2.5* utile al fine di scrivere e controllare il codice informatico, nonché di eseguirlo tramite l'utilizzo della apposita *bash*.

Java Development Kit

Esiste una moltitudine di linguaggi di programmazione in grado di fornire all'utente, più o meno esperto, una via per risolvere problemi di diversa natura mediante la progettazione e l'elaborazione informatica di algoritmi, ossia sequenze di istruzioni di difficoltà variabile.

Ciò che Java offre è sicuramente un linguaggio di livello avanzato in grado di formalizzare problemi da risolvere.E' possibile, grazie ad esso, controllare il computer in modo tale che si riveli uno strumento utile per effettuare azioni o per prendere decisioni.

Lo sviluppo di questo software è avvenuto da parte di Sun Microsystems, la quale lo rende disponibile gratuitamente all'indirizzo **http://java.sun.com**.

L'utilizzo di Java, oltre che per le sue caratteristiche intrinseche, si collega ad una tendenza che si riscontra generalmente in tutti gli ambienti informatici. Oggi i computer invadono tutte le aree produttive, pertanto gli ambiti di ricerca non possono assolutamente sottrarsi al loro utilizzo. In un'epoca in cui assistiamo ad un rialzo generale dei prezzi, il costo dei computer è in decisa controtendenza, offrendo ad un maggior numero di utenti la possibilità di disporre di un *hardware*.

Java è probabilmente destinato a divenire il linguaggio più importante del nuovo millennio, grazie all'implementazione di applicazioni basate sull'*Internet* e sulle *Intranet* (reti interne aziendali), nonché per lo sviluppo di software destinato alle comunicazioni in rete, come nel caso dei telefoni cellulari.

A lungo le università hanno adottato ed insegnato linguaggi di programmazione come C e C++ caratterizzati da elevata portabilità, bastava infatti avere un compilatore e i programmi potevano funzionare ovunque. Il mondo della programmazione è in continua evoluzione e la

richiesta degli utenti è rivolta sempre più verso prodotti sofisticati, che forniscano interfacce grafiche, la possibilità di sfruttare le capacità multimediali di immagini, video e suoni.Ne è esempio il *multithreading*, che permette ai programmatori di specificare che più attività debbano essere eseguite in parallelo, applicazioni che possano essere scritte rapidamente e semplicemente utilizzando la caratteristica di ereditarietà e quindi il riutilizzo del software.

L'estrema portabilità dei programmi, ossia la possibilità di eseguire le applicazioni su ogni piattaforma ,senza bisogno di modifiche, è infine caratteristica fondamentale dalla quale non si può più prescindere.

Java è inoltre molto interessante a livello universitario poichè si tratta di un linguaggio completamente orientato agli oggetti, e l'elaborazione degli errori è già contenuta nella struttura, ma non solo: anche il fatto che possa essere scaricato gratuitamente dal sito della Sun Microsystems è importante per le università, dotate di solito di budget limitati.

La programmazione orientata agli oggetti (OOP)

Bruce Eckell (2000) fa notare come tutti i linguaggi di programmazione rappresentino una certa dose di astrazione, è evidente come la complessità dei problemi che una persona è in grado di risolvere sia direttamente proporzionale al tipo ed alla qualità dell'astrazione che è in grado di fare, considerando per "tipo" ciò che si sta astraendo.

I linguaggi di tipo *assemby* rappresentano un lieve astrazione della macchina sottostante, mentre linguaggi cosiddetti imperativi, di epoche successive, come il *Fortran*, il *BASIC* ed il *C* sono astrazioni dell'*assembly*. Sicuramente questi ultimi rappresentano una notevole

evoluzione rispetto al linguaggio macchina iniziale, ma la loro astrazione richiede ancora di pensare in termini della struttura dell'*hardware*, piuttosto che al problema da risolvere.

I programmatori devono stabilire l'associazione fra il modello di computer ed il modello del problema da risolvere. Lo sforzo richiesto per produrre programmi difficili da scrivere e costosi da mantenere è notevole ed ha l'effetto collaterale di creare un'industria del "come programmare".

L'alternativa possibile alla modellizzazione della macchina è avere uno strumento che permetta di ragionare in funzione del problema da risolvere:

1. linguaggi dichiarativi, come il *LISP* e l'*APL* hanno una visione particolare del mondo, infatti "*All problems are ultimately lists*" e "*All problems are algorithmic*" rispettivamente;
2. il *PROLOG* racchiude tutti i problemi in catene di decisione;
3. altri programmi sono stati creati con l'unico obiettivo di manipolare simboli grafici.

Tutti questi sembrano avere una ottima soluzione per lo specifico tipo di problema per cui sono creati, ma diviene arduo un loro utilizzo al di fuori degli obiettivi posti dai rispettivi ideatori.

La programmazione orientata agli oggetti fa quindi un passo avanti, fornendo agli utenti degli applicativi in grado di rappresentare gli elementi del problema in maniera sufficientemente generale così che essa non sia limitata solo a determinati tipi di problemi.

Ogni elemento del problema è trattato come un *oggetto*, l'idea è quella di permettere al programma di adattarvisi aggiungendo nuovi oggetti, in modo tale che, quando si legge il codice che descrive la

soluzione, in realtà vengono visualizzate le parole che esprimono il problema. Questo è sicuramente un metodo di astrazione del linguaggio più flessibile e potente di quanto non si abbia avuto sino ad ora, perciò la *OOP* permette di descrivere il problema da risolvere nei suoi termini, piuttosto che nel linguaggio del computer che lo deve eseguire.

Ovviamente il grado di astrazione non è completo: infatti un oggetto funziona come un computer. Ha dei valori che ne definiscono lo stato ed è inoltre in grado di eseguire delle operazioni più o meno complesse rappresentate da strutture di algoritmi, in ogni casosi rivela un buon compromesso con la realtà nella quale vi sono oggetti, ognuno con propri attributi e comportamenti.

Alan Kay ha riassunto cinque caratteristiche di base di Smalltalk, il primo linguaggio OOP, sul quale è basato Java:

1. **Ogni cosa è un oggetto.** Occorre pensare ad un oggetto come una variabile speciale, che immagazzina dati, ma alla quale si può anche fare una richiesta computazionale su se stessa. Teoricamente ogni cosa faccia parte di un problema è rappresentabile come un oggetto, con le proprie caratteristiche
2. **Un programma è un insieme di oggetti che comunicano fra loro cosa fare trasmettendosi dei messaggi**. Per fare una richiesta ad un oggetto occorre mandare un messaggio, il quale può essere concretamente interpretato come una chiamata ad una funzione che appartiene all'oggetto;
3. **Ogni oggetto ha la propria memoria costituita da altri oggetti**. In altre parole la creazione di un oggetto avviene tramite l'utilizzo di pacchetti che contengono altri oggetti già

esistenti. Diviene così possibile costruire la complessità in un programma, nascondendola in semplici oggetti;

4. **Ogni oggetto ha una propria definizione di tipo**. In parole più semplici ciò significa che ogni oggetto è una *instance* (un esemplare riportante le caratteristiche generali) di una classe, dove classe è sinonimo di tipo. La distinzione più importante delle caratteristiche di una classe è basata su quale messaggio è in grado di ricevere;
5. **Infine tutti gli oggetti di una particolare classe possono ricevere gli stessi messaggi.** Ciò significa, ad esempio, che un oggetto di tipo cerchio è anche un oggetto del tipo formeGeometriche, per cui tutti i messaggi che possono essere inviati all'oggetto formeGeometriche possono essere recepiti anche dall'oggetto cerchio. Perciò diviene possibile scrivere del codice informatico che dialoghi, ossia scambi messaggi, con formeGeometriche e che automaticamente si riferisca a cerchio. Questa sostituibilità è una delle caratteristiche più importanti della programmazione orientata agli oggetti.

Il concetto per il quale un singolo oggetto sia nella sua unicità anche parte di una classe di oggetti, deriva dal primo linguaggio orientato proprio ad essi, Simula – 67, con il termine *class* che introduce un nuovo tipo di dato nel programma.

Quindi ciò che si fa realmente nella programmazione orientata agli oggetti è creare nuovi tipi di dati, che si fondano sulla descrizione di nuovi tipi o *classi* appunto.

Le funzionalità o comportamenti che ogni singolo oggetto è in grado di tenere, consistono di procedure strutturate, archiviazioni di

dati in variabili di vari tipi e scambio di messaggi. I comportamenti sono chiamati metodi ed attengono ai singoli oggetti che li richiamano.

L'accesso ai metodi, così come ai diversi tipi di variabili, può essere limitato tra le classi per evitare che avvengano modifiche involontarie, è infatti sufficiente ricorrere a tre delimitatori dell'accesso ai metodi o alle variabili di una classe:

1. **public**, i membri di questo tipo sono accessibili da una qualunque posizione di un programma in cui ci sia riferimento a quel tipo di superclasse o ad uno dei suoi tipi di sottoclasse;
2. **protected**, i membri sono accessibili solo dai metodi della superclasse cui fanno riferimento, dalle sottoclassi e dai metodi di altre classi dello stesso package;
3. **private**, i membri di questo tipo sono accessibili solo ai membri della superclasse di appartenenza.

L'accesso ai metodi: metodi Set e Get

L'utilizzo delle variabili di istanza di tipo *private* all'interno delle classi limita, quindi, le modifiche di coloro che riutilizzano il software da altri creato e chiunque faccia uso di una classe già esistente può apportare modifiche alle variabili.

La caratteristica fondamentale dei tipi *private* è quella di rendere possibile l'accesso esclusivamente ai metodi della classe stessa, nella struttura, però, le classi forniscono sovente dei metodi *public*,in grado invece di permettere l'accesso anche dall'esterno della specifica classe, permettendo così di assegnare dei valori (*set*) od ottenere dei valori (*get*) dalle variabili di istanza *private.*

Ovviamente il fatto di chiamare set o get questo tipo di metodi è una convenzione adottata in particolar modo nella prospettiva del riutilizzo del software, in quanto questa convenzione, anche in assenza di variabili di tipo private, consente una elevata leggibilità del programma.

L'ereditarietà

L'idea di utilizzare un oggetto è una applicazione molto funzionale: infatti questo concetto permette di creare pacchetti di dati e funzioni che stanno insieme concettualmente; diviene così possibile rappresentare l'idea del problema da risolvere, piuttosto che proporre lo stesso come un idioma della macchina sottostante.

Questi concetti stanno alla base della programmazione ad oggetti quando si introduce l'utilizzo della parola chiave *classe.* Ad uno sguardo più attento ed in un successivo passo di astrazione dell'impostazione può sembrare una forzatura il fatto di impostare una classe per risolvere un problema, con tutte le difficoltà della sua realizzazione, per poi doverne progettare un'altra con simili funzionalità. Sarebbe meglio se si potesse prendere una classe già esistente, clonarla ed ampliarne le caratteristiche o modificarne alcune in parte.

Questo è ciò che viene effettivamente fatto con l'ereditarietà, con l'eccezione che se la classe originale, chiamata superclasse o classe genitore, è cambiata, il clone modificato, chiamato sottoclasse, riflette anch'esso questi cambiamenti.

Un tipo di dato ha sicuramente più caratteristiche della semplice descrizione delle capacità di un certo oggetto; esso ha anche relazioni con altri dati. Due tipi possono avere caratteristiche e comportamenti in

comune, ma un tipo può contenere più caratteristiche di un altro oltre alla possibilità di gestire più messaggi o gestirli in maniera differente.

L'ereditarietà esprime questa analogia fra i tipi usando il concetto di dato base e dato derivato: il tipo di base è creato per rappresentare il fulcro dell'idea nel concetto, da esso se ne derivano altri per esprimere i differenti modi in cui questo concetto fondamentale può essere modificato. L'esempio classico è il tipo formeGeometriche, che ha una forma, un colore ed una posizione. Ogni oggetto di questo genere può poi essere disegnato, cancellato, colorato e così via. Da questa generica forma poi è possibile creare specifici *sottotipi* che possono possedere proprie ed esclusive caratteristiche e funzionalità, come il metodo richiesto per il calcolo dell'area o del perimetro o del volume, ove si estendesse anche il numero di dimensioni. La gerarchia dei tipi incorpora così sia le uguaglianze che le differenze fra le forme.

Fino a quando sia la classe di base sia la classe derivata possiedono la stessa interfaccia, ci deve essere un'implementazione comune che permetta di richiamarne i metodi. Ciò permette all'oggetto di processare determinati messaggi che rappresentano un punto in comune fra tutte le tipologie di oggetti derivati da una determinata superclasse. Avviene così che qualora si definisca una sottoclasse ereditando da un tipo base senza attribuire ulteriori o nuovi comportamenti o funzionalità, la classe ottenuta non sarà altro che un clone del tutto simile alla classe da cui sta ereditando.

Esistono due modi per differenziare la nuova sottoclasse da quella di origine :

1. il primo consiste nell'aggiungere funzioni nuove che prima non esistevano, queste non fanno parte dell'interfaccia base della classe. Ciò accade specialmente in Java dove la parola

chiave per ereditare è in realtà *extends*, ossia con tale comportamento siamo in grado di estendere le funzionalità di una classe;

2. il secondo metodo consiste nel modificare il comportamento di una funzione della sottoclasse riscrivendo il metodo della superclasse. Questa tecnica è detta *overriding*, sovrascrittura appunto.

Override di un metodo e costruttori multipli

Java permette di definire più metodi col medesimo nome a patto che questi si differenzino per un insieme di parametri diversi. La distinzione potrà avvenire in base al tipo, al numero od all'ordine dei parametri. Questa tecnica non va confusa con la precedente, anche se presenta lo stesso nome, ossia *override di un metodo.* Esiste quindi la concreta possibilità che all'interno di un medesimo programma siano presenti più funzionalità dello stesso oggetto a cui ci si riferisce col medesimo nome.

La scelta che è operata in fase di compilazione tiene conto dei tipi di dati e dell'ordine in cui questi sono passati, attraverso un *messaggio*, al metodo. Più precisamente il compilatore effettua la chiamata del metodo corretto analizzando il numero, i tipi e l'ordine degli argomenti della chiamata.

Poiché in sostanza anche i costruttori sono dei metodi della classe, è possibile che una classe abbia più costruttori con il medesimo nome, differenti però per quanto riguarda i parametri.

Il polimorfismo

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

Le classi che non esistono durante lo sviluppo del programma, possono essere aggiunte con poche modifiche alla parte generica dello stesso, a patto che dette classi facciano parte della gerarchia che viene elaborata. Le uniche parti di un programma che devono essere modificate sono quelle che richiedono una conoscenza diretta della particolare classe aggiunta alla gerarchia.

Immaginiamo di avere un gruppo di classi, tutte derivate da formeGeometriche, chiamate Cerchio, Triangolo, Quadrato e così via. Nella *OOP* a ciascuna è data la possibilità di disegnarsi, tramite un metodo interno, anche se con caratteristiche leggermente diverse. Grazie alla funzionalità di *binding dinamico* per disegnare ciascuna forma è sufficiente richiamare il metodo di disegno proprio della classe generica formeGeometriche, sarà il programma a determinare automaticamente quale metodo utilizzare per disegnare la forma esattamente richiesta.

Java sceglie in modo polimorfico il corretto metodo *overridden* nell'appropriata sottoclasse associata all'oggetto, quando un riferimento di una superclasse chiede di usare un metodo. In pratica quando un messaggio è mandato ad un oggetto, questo è in grado di rispondere perfettamente: lo stesso messaggio verrà interpretato in una moltitudine di modi differenti, ciascuno per ogni forma e con un collegamento che dipende direttamente dagli argomenti passati e che indirizzano l'operazione di *binding* dinamico.

I Java tools

Java, oltre ad avere una notevole quantità di caratteristiche che lo rendono molto adeguato allo sviluppo di applicazioni per lo studio dei problemi sociali complessi, mette a disposizione anche un numero elevato di strumenti per la gestione, la presentazione, lo scambio e la descrizione dei lavori svolti. Di seguito ne illustro, nelle loro caratteristiche più generali, alcuni che si sono rivelati molto utili nella stesura del lavoro e nella sua documentazione.

Jar

Il jar è una applicazione che archivia e comprime più file in un unico blocco di programmi Java (Java Archive), basato sul formato Zip.

La sua funzione principale è quella di facilitare l'impacchettamento (packaging) di applet od applicazioni in un singolo archivio di file con estensione *java* già compilati. Quando le componenti di una applet o di una applicazione (file di tipo .class, immagini, suoni, ecc…) sono riunite in un solo archivio, esse possono essere scaricate con un browser in una unica sessione, consentendo quindi un notevole risparmio di tempo.

Tutto questo permette inoltre l'autenticazione dei programmi da parte dell'autore.

I tre tipi di file che sono richiesti per creare e gestire un Jar sono:

1. Il file manifest (opzionale);
2. Il file jar di destinazione;
3. I file che devono essere archiviati.

Per poter creare questo archivio viene utilizzando il comando

*% jar cf nomejarfile *.class*

che indica all'applicazione jar di inserire tutti i file con estensione *.class* nel file di nome *nomejarfile*. Viene allo stesso tempo generato un file con nome *manifest.mf*, recante le informazioni riguardanti l'archivio.

Un file di tipo *manifest* può essere composto da una sezione principale contenente impostazioni di sicurezza e di configurazione riguardanti il file manifest medesimo, ed altre sezioni che definiscono i vari attributi per i packages o per i file contenuti nel Jar file.

Javadoc

L'applicazione Javadoc utilizza i commenti al codice scritto, per produrre un insieme di pagine HTML contenenti la descrizione delle classi public e protected, delle inner class, dei costruttori, dei metodi e delle variabili siano esse d'istanza o locali.

Il suo funzionamento, in termini ampiamente generali, si fonda sul compilatore java, il quale viene richiamato per le dichiarazioni dei commenti, ignorando però l'implementazione dei metodi e del codice interno.

Questo applicativo crea anzitutto la struttura gerarchica delle classi e la struttura delle relazioni, l'affidabilità della rappresentazione è garantita dall'intervento, come già detto, del compilatore.

Swarm – 2.2

Swarm è nato nel 1995, nel Santa Fe Institute (New Mexico, USA) con l'obiettivo di mettere a disposizione un insieme di programmi e librerie standard, per simulare ed analizzare sistemi complessi di comportamento, nell'ambito delle scienze naturali e sociali.

La necessità avvertita dagli studiosi era quella di avere uno strumento che permettesse di sviluppare applicazioni di simulazione senza preoccuparsi di problemi tipici di programmazione informatica, quali la generazione di grafici o di un'interfaccia grafica o i modi di

salvare i risultati delle simulazioni. Inoltre Swarm provvede una struttura comune che le rende riproducibili da altri ricercatori, essendo fondata su un linguaggio ad oggetti quale l'*ObjectiveC* e Java.

La struttura generale di un programma

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

I files che occorre scrivere per un programma sono diversi secondo che si usi ObjectiveC o Java, ma comunque, si trovano sempre gli stessi elementi:

1. Un file dove è definita una funzione main(), che `e fondamentale nel programma. Qui, viene inizializzato lo Swarm, e sono creati gli 'swarms' importanti nel programma (il model- Swarm, l'observerSwarm). `E anche qui che viene lanciata la simulazione;
2. 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;
3. Un observerSwarm, dove si trovano gli oggetti e interfacce grafici, dov'è definita una simulazione. Permette di far girare il programma più volte, cambiando alcuni parametri;

4. Un modelSwarm, dov'e definito il modello e sono creati gli agenti. Un programma di Swarm funzionerebbe anche senza modelSwarm, dichiarando tutto nel metodo main(), ma si perderebbero molte facilitazioni di Swarm (specialmente gli Schedules).
5. Alcuni files che definiscono gli agenti del modello.

Gestione degli agenti

La creazione degli agenti in Swarm è notevolmente semplificata dall'utilizzo della programmazione orientata agli oggetti la quale, come detto in precedenza, permette di esprimere un problema nei suoi termini e di gestire esemplari, *intances*, con differenti funzionalità.

Una notevole facilitazione deriva poi dalla possibilità di impostare lo schema *Environment – Rules – Agent*, descritto nel capitolo xx. La gestione delle relazioni fra agenti e il dell'applicazione di comportamenti dei singoli agenti, è decisamente favorita dall'utilizzo delle *collezioni* messe a disposizione da Swarm.

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

Lists

Le principali liste sono importate nel programma con il richiamo delle seguenti classi:

1. *collections.ListImpl*, una lista è un insieme d'elementi che rispettano una posizione, a loro assegnata dall'esterno della lista stessa. La sequenza è determinata dal punto in cui

ciascun elemento è aggiunto: esso può essere aggiunto all'inizio della lista, alla fine o in qualunque altro punto.

2. *java.util.List*, si tratta di una classe che mette a disposizione una *"Sequences"*. L'utente ha il controllo sull'esatta locazione dei singoli elementi inseriti nella lista e può accedere direttamente a ciascuno di essi con una ricerca basata sul numero intero d'ordine di ciascuno.

Il protocollo List di Swarm implementa una lista con dimensioni flessibili (la lista cresce quando sono aggiunti elementi, decresce quando vengono cancellati ,permettendo così di elencare oggetti, e di eseguire, in modo semplice, azioni ripetitive sugli oggetti contenuti nella lista.

Una lista si crea in Java con *nameOfList = new ListImpl(getZone())*, mentre le funzionalità a disposizione sono:

1. addFirst, addLast aggiunge un oggetto all'inizio o alla fine della lista;
2. removeFirst, removeLast cancella il primo o l'ultimo elemento della lista;
3. getCount riporta il numero di elementi nella lista;
4. listBegin(getZone()) per creare un indice che può essere usato per attraversare
5. la lista;
6. remove(aMember) per cancellare aMember dalla lista;
7. removeAll cancella tutti gli oggetti della lista, ma non la lista stessa;
8. deleteAll cancella tutti gli oggetti della lista, e cancella anche la lista dalla memoria;

9. atOffset(i) per ritornare l'elemento che si trova in posizione i nella lista.
10. contains(aMember) ritorna 1 se aMember è nella lista;
11. forEach(selector) manda il messaggio message a tutti gli oggetti della lista.

Arrays

Il vettore conserva dati ed oggetti in maniera statica, ossia viene dichiarato con la sua dimensione a priori, ed è ordinato. Il vantaggio del vettore sulla lista è che gli oggetti possono essere trovati molto velocemente, perché il loro indice è un intero. Quindi, si può effettuare una ricerca usando un ciclo iterativo qualunque.

Il vettore si può creare usando, *nameOfArray = new ArrayImpl(getZone(),size)* mentre le funzionalità a disposizione sono:

1. atOffset$put(i,obj) per mettere l'oggetto obj alla posizione i del vettore;
2. atOffset(i) per ritornare l'oggetto che si trova alla posizione i del vettore;
3. getCount;
4. contains(aMember);
5. forEach(selector).

Maps

Una mappa è una tabella con due file di oggetti. Nella prima si trovano i nomi degli stessi, nella seconda, invece, si trovano gli oggetti che vogliamo conservare nella collezione. Quindi, per mettere un oggetto in una mappa, dobbiamo necessariamente specificare il suo nome (key, chiave). Possiamo in seguito chiedere alla mappa di ritornare sull' oggetto che corrisponde ad una chiave particolare o di cancellarlo. Una mappa si crea usando, *nameOfMap = new MapImpl(getZone())*mentre le funzionalità a disposizione sono:

1. at$insert(keyObject,object) per inserire object con la chiave keyObject.
2. at(keyObject) per ritornare l'oggetto corrispondente a keyObject.
3. getCount;
4. contains(object);
5. forEachKey(selector);
6. getFirst,getLast;

Gli elementi di una collezione di Swarm devono essere 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 tali.

Per questo, usiamo delle classi che chiamiamo *wrappers*, con il compito di trasformare, per esempio, un numero intero in un oggetto che contiene il suo valore.

Questa è la soluzione che abbiamo adottato per gestire l'insieme delle ricette di sequenze produttive; l'implementazione della classe Recipe.java, si fonda sulla necessità di archiviare dei vettori in una lista.

Sapendo che i vettori non sono degli oggetti puri abbiamo creato una classe di tipo *wrapper*.

Gestione degli eventi

Prima di trattare la gestione degli eventi in Swarm, occorre soffermarsi sulla modalità con la quale i messaggi contenenti le azioni da eseguire sono trattati all'interno di questa libreria.

Con la classe *SwarmUtils* (Charles P. Staelin) di tipo public, si definiscono dei metodi atti a creare un Selector; si tratta di un oggetto usato per incapsulare messaggi da inviare ad altri oggetti.

In questo caso il messaggio è composto dagli argomenti da passare al costruttore *getSelector().* Essi saranno: in parte il nome della classe cui far riferimento, in parte il metodo della stessa che ci interessa richiamare.

Il metodo getSelector() è un metodo sovraccarico: infatti può essere richiamato passando, come primo argomento, sia una stringa contenente il nome della classe, sia un oggetto della classe desiderata.

Il valore boolean, presente nell'argomento dell'oggetto Selector, indica, se impostato su *true*, la possibilità di usare la sintassi di tipoObjectiveC. Indica, altrimenti, la possibilità di usare la sintassi Java, qualora impostato su false.

Per ciò che riguarda invece le azioni e l'organizzazione di queste, nella sintassi si fa riferimento rispettivamente agli *schedules* ed alle *activity*, che rappresentano una delle caratteristiche più importanti di Swarm. Esse permettono di integrare le azioni di vari agenti in diversi livelli della simulazione.

Anche se generalmente non si usa spesso questa classe *Activity* per comprenderne l'utilità, occorre considerare uno Swarm come un insieme di agenti e di liste di attività che costoro dovranno svolgere

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 *null,* dopo aver creato le sue azioni e i suoi *schedule*. Questo fa si che il modelSwarm segnali al prossimo livello di Swarm di attivarsi. In questo modo, vengono sincronizzate le attività dei vari livelli della simulazione.

In realtà un miglior uso è fatto all'interno dei GUISwarms del ControlPanel, il quale permette di gestire la simulazione dal mouse.

Alcuni importanti esempi dell'uso di questa classe, sono rappresentati da *activity.ActionGroup* e da *activity.Schedale;* un *ActionGroup,* così come uno *Schedule,* è un "gruppo" di azioni in precedenza incapsulate nel Selector, che devono essere eseguite, secondo un ordine definito internamente, da parte di oggetti cui verranno passati come messaggi. Le classi *activity.ActionGroupImpl* e *activity.ScheduleImpl* sono le classi contenenti i metodi necessari per implementare sia gli *ActionGroup* sia gli *Schedule.*

Altra caratteristica fondamentale di Swarm risiede nell'esistenza di uno schedale che può essere interpretato come un ordine del giorno, in cui si scrive cosa fare ad ogni istante della simulazione. Uno *schedule* è un oggetto di Swarm nel quale si possono inserire messaggi individuali o gruppi di messaggi (gli ActionGroups visti in precedenza).

Una caratteristica fondamentale dello *schedule* è il tempo di latenza fra azioni. Generalmente, il *repeatInterval* (così è denominato il

metodo che gestisce tale intervallo di azioni nel modello) vale uno, il che vuol dire che le azioni sono ripetute ogni periodo.

Una volta creato lo schedule abbiamo tutti gli oggetti della simulazione pronti e, grazie al metodo *ActivateIn()* , in grado di essere attivati.

Un'altra caratteristica di Swarm, che rende lo schedale uno strumento decisamente potente, è la possibilità di avere *schedules dinamici*, ossia un oggetto di tipo schedule vuoto.Le azioni di quest' ultimo vengono descritte in un metodo adatto alla loro specificazione; in tal modo esse sono eseguite in base allo stato della simulazione in un determinato periodo.

Gestione della simulazione

La caratteristica fondamentale di una simulazione è la possibilità di sperimentare le conseguenze di una diversa visione della realtà rappresentata. Ciò è reso possibile dall'utilizzo di parametri che solitamente sono interni al modello, in Swarm è invece possibile salvarli in un file *nomeFile.scm*, col vantaggio che diviene più semplice modificarli, perché si trovano in un file separato ed inoltre non c'è bisogno di ricompilare il programma se lo vogliamo far girare con nuovi valori dei parametri stessi. Basta cambiare il file *nomeFile.scm*, salvarlo e far girare il programma.

La classe che si usa per importare/salvare parametri si chiama LispArchiver, infatti 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/)).

GUISwarms e sonde (probes)

L'ObserverSwarm è la classe che ci permette di osservare e controllare l'andamento della simulazione.

1. La caratteristica principale è il ControlPanel, che contiene 5 bottoni:
2. start Inizia la simulazione, e si ferma alla fine
3. stop Pausa la simulazione
4. next Avanza la simulazione di un'iterazione
5. save Salva la disposizione delle finestre (se esiste lo statement apposito nel programma)
6. quit Esce dalla simulazione (anche se non finita)

Per ottenere informazioni circa lo Swarm che si osserva o per impostare parametri esistono diverse classi per poter effettuare tali impostazioni. Alcuni esempi sono:

1. objectbase.VarProbe, una classe che permette all'utente del programma di ottenere informazioni circa una particolare variabile di istanza (condivisa all'interno di una classe da tutti i metodi) o locale (propria di ogni metodo);
2. objectbase.MessageProbe, una classe che permette all'utente di richiamare un determinato messaggio riguardante ogni esemplare di una classe, o riguardante un oggetto che erediti da un'altra classe;
3. objectbase.EmptyProbeMapImpl, una *CustomProbeMap* utilizzata per creare *ProbeMaps* direttamente dall'interno del programma. Una P*robeMap* è una classe-contenitore per

Probe, utilizzato per definire il contenuto di un *ProbeDisplay.* Ad esempio tutte le sonde relative a variabili di istanza di un Model dovrebbero essere raccolte all'interno di una ProbeMap. Un *ProbeDisplay* è l'interfaccia grafica da cui è possibile osservare le *Probe*. Una *CustomProbeMap* è una sottoclasse della *ProbeMap* usata per creare *Probe* che siano inizializzate in uno *empty state* od utilizzando VarProbe e MessageProbe.

Rappresentazione grafica della simulazione

Esistono anche rappresentazioni della simulazione che favoriscono l'interazione fra l'uomo e la simulazione. Abbiamo in particolare delle classi come *simtoolsgui.GUISwarm* e *simtoolsgui.GUISwarmImpl* che sono le classi che si rendono necessarie per introdurre e gestire una interfaccia grafica nel modello, permettendoci di controllare i grafici e decidendo, ad esempio, quante volte dovranno essere aggiornati nel corso della simulazione. In particolare lo stesso Observer estende le caratteristiche della superclasse GUISwarmImpl, risultando così essere in realtà uno schema grafico intuitivo che permette l'interazione con la macchina e la predisposizione dei parametri per osservare la simulazione.

Per ciò che invece attiene alla rappresentazione grafica dei risultati, molto utili si rivelano le classi *analysis.EZGraph* e *analysis.EZGraphImpl*, che consentono all'utente di creare facilmente grafici (da cui il nome, infatti la pronuncia di EZ ricorda il termine inglese *easy* che significa facile) di diverse misure e tipologie, a seconda

delle necessità. L'utente creerà prima i grafici e poi le *"Sequences"* , liste ordinate di elementi, che verranno quindi rappresentate.

Le *"Sequences"* sono generate basandosi su dati forniti da un singolo oggetto o da un insieme di oggetti, in corrispondenza ad uno specifico selettore. Una caratteristica interessante è la possibilità che la classe EZGraph generi automaticamente medie, totali, minimi, massimi ed effettui il conteggio delle sequenze, senza che l'utente debba creare ulteriori sottoclassi.

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

Gestione della memoria e dei processi interni

Come in ogni libreria di funzioni, di qualunque linguaggio si tratti, devono essere previste delle funzionalità per la gestione della memoria. La classe *objectbase.SwarmImpl* è un contenitore temporaneo, uno Swarm è, invece, un insieme d'agenti che condividono sia il tempo sia una certa area di memoria.

La libreria defobj è necessaria per fornire supporto agli oggetti di Swarm; essa contiene i metodi più importanti per disegnare *agenti* e *swarm,* viene pertanto utilizzata anche come deposito per gestire le sonde (probe).

La classe Zone è un registro d'allocazione per i dati o per gli oggetti creati in Swarm. Una zona deve essere assegnata ogni volta che si crea un nuovo oggetto, essa diventa la fonte da cui i dati per le variabili d'istanza e per altri dati interni sono ottenuti.

Infine, occorre tenere in considerazione quali potranno essere le eventuali evoluzioni future del progetto; è sicuramente di grande interesse l'intenzione degli sviluppatori di Swarm di utilizzare zone multiple memoria in cui allocare gli swarm. Così facendo saranno in grado di assicurarsi che oggetti con necessità di dati ed altre caratteristiche omogenee, siano allocati insieme in maniera ottimale, od in modo tale che sia possibile condividere le medesime risorse. Il risultato di tale previsione potrebbero essere strutture di simulazioni non solo con molti swarm, ma essi opererebbero contemporaneamente su zone di memoria differenti, cosa che garantirebbe un ulteriore passo avanti nella rappresentazione della complessità.

E-macs-21.1 e Cygwin

Un breve accenno va fatto anche agli strumenti operativi con i quali è stato possibile creare il codice informatico nonché testarlo, correggerlo ed adeguarlo alle versioni del lavoro che si sono succedute.

Editor

Lo strumento di editor che personalmente ho ritenuto più funzionale è stato E-macs nella sua release 21.1

E-macs, come si vantano i suoi autori, rappresenta un editor dotato di alcune caratteristiche fondamentali:

1. Estensibile;

2. avanzato
3. personalizzabile;
4. auto documentante;
5. con *display editor* in tempo reale.

La definizione di *display editor* è data perché normalmente il testo è visibile sullo schermo ed è aggiornato automaticamente man mano che si scrive. Il display è aggiornato con molta frequenza, così come rapidamente i blocchi di codice sono identati, evidenziati o colorati e ciò definisce la caratteristica di *real – time.*

I suoi autori definiscono Emacs un procedimento avanzato per il fatto che esso va oltre il semplice inserimento e la cancellazione: esiste infatti la possibilità di controllare i sottoprocessi, identificare i programmi, vedere e lavorare su più file allo stesso tempo. Vi è anche la possibilità di scrivere testi formattati e gestire parole, frasi, simboli, pagine o righe in differenti linguaggi di programmazione.

Auto – documentante significa che ogni volta che un carattere speciale dell'editor è richiamato con una combinazione di tasti, si accede ad una funzione (ctrl – h) in grado di offrire la documentazione di tale carattere.

Personalizzabile si attiene alla possibilità che si decida di modificare le definizioni dei comandi di Emacs

Estensibile si riferisce alla possibilità che tale strumento offre di creare dal nulla nuovi comandi o programmi scritti in linguaggio Lisp e che possono poi essere interpretati da Emacs stesso

Quando eseguito sotto un sistema di tipo Windows, Emacs fornisce propri menù ed attribuisce proprie funzioni ai bottoni del mouse, ma è possibile personalizzare sia quelle, sia estenderlo alle

caratteristiche di altri linguaggi di programmazione, come succede con l'installazione del modulo jde-2.2.8 e dei relativi applicativi.

Ambiente di esecuzione

Gli strumenti applicativi forniti da Cygwin fanno parte degli strumenti di sviluppo e delle *utility* della *GNU* progettate per funzionare sotto i sistemi operativi Windows NT/9x. Essi funzionano mediante l'utilizzo di una libreria che fornisce sia un ambiente di tipo UNIX sia le chiamate di sistema che si rendono necessarie.

A seguito dell'installazione di questo ambiente di sviluppo, i programmatori possono scrivere applicazioni di tipo *GUI* o interfacce per Win32 sfruttando gli standard Microsoft API o Cygwin API. Diviene così possibile portare da un sistema ad un altro molti programmi basati su UNIX senza la necessità di effettuare notevoli modifiche al codice informatico. Ciò rappresenta sicuramente un vantaggio per la diffusione dei programmi e per la condivisione di programmi di simulazione che, come detto, è fondamentale per lo sviluppo della conoscenza fra gli studiosi della simulazione ad agenti.

Bibliografia

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

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

Bissey, M.E. (2001), *Una piccola introduzione a Swarm: ObjectiveC e Java*, Dipartimento di Politiche Pubbliche e Scelte Collettive POLIS, Università del Piemonte Orientale, bissey@sp.unipmn.it

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