SSIS VENETO
Didattica della OOP mediante la tartaruga LOGO

Specializzando: dott. Andrea Marin
Relatore: prof. Lucio Varagnolo

14/05/2004

Indice

Capitolo 1

Introduzione

In questo lavoro prendiamo in considerazione un approccio didattico alla OOP con uno strumento ideato dal prof. Renato Conte (che ringraziamo per la partecipazione attiva e per i consigli fornitici nello stendere questo lavoro) cioé una libreria C++. Essa, oltre che a proporre strutture dati ad alto livello (vettori, stringhe) rendendo il linguaggio piú consono alla didattica, fornisce uno strumento eccezionale per l'introduzione della programmazione ad oggetti. Si tratta di una classe Tartaruga che ricalca e potenzia l'idea della tartaruga Logo ideata dal prof. Papert negli anni 60. Grazie ad alcune attivitá didattiche mirate, alcune delle quali sono proposte nei capitoli che seguono, essa puó essere un supporto formidabile alla costruzione di concetti chiave della programmazione come:

- differenza tra classi ed oggetti
- passaggio di parametri per copia o per indirizzo
- ricorsione
- frattali
- ereditarietá
- polimorfismo
- stream di input e output

Con questo lavoro, oltre ad analizzare proposte didattiche che fanno uso della libreria EASYC4 (scaricabile da `www.dsi.unive.it/~conte`), propone anche una trasposizione della stessa per il linguaggio Java tentando di formulare ipotesi di lavoro e soprattutto individuando elementi di criticitá di una tale impostazione didattica.

Capitolo 2

Problemi nella didattica della OOP

In questo capitolo tenteremo di mostrare da un lato le valenze formative della programmazione ad oggetti, dall'altro di individuarne i maggiori problemi riscontrabili nella sua didattica.
C'é da anticipare che la programmazione ad oggetti é un paradigma che nasce alla fine degli anni settanta ma che trova diffusione soltanto all'inizio degli anni 90. Si pensi che per attendere una classificazione rigorosa del polimorfismo si é dovuto attendere la metá degli anni '80 (Cf. [Car85]), confrontando i risultati della ricerca sul lambda calcolo e dell'evoluzione dei linguaggi di programmazione commerciali (primo fra tutti C++). Per questi motivi la ricerca didattica sui metodi e sul valore formativo della OOP non sembra essere molto sviluppata; d'altra parte il tema appare ancora come qualcosa per specialisti e riveste una parte del curricolo (purtroppo) solo nel corso di informatica per specialisti (l'attuale sperimentazione *Abacus*).
Cercheremo di illustrare ed argomentare in questo lavoro essenzialmente due tesi:

- il valore formativo della programmazione ad oggetti
- una proposta per una didattica articolata a partire da quelli che possono essere considerati i punti critici dell'acquisizione di questo paradigma di programmazione

2.1 Valenze formative della OOP

Innanzitutto cerchiamo di rispondere alla domanda: perché affrontare in un corso di studi (anche per non specialisti) la programmazione ad oggetti?

Certo la risposta per quei casi in cui si stanno formando dei periti industriali informatici appare essere l'utilitá del paradigma, vista anche la sua enorme diffusione e gli ampi vantaggi che esso permette nello sviluppo del codice. Ma crediamo che questo giustifichi solo marginalmente una scelta tanto importante e che se riapplicato come metodo arriverebbe a giustificare l'insegnamento dei wordprocessors commerciali nei curricoli scolastici. Qual é allora il valore formativo della programmazione orientata agli oggetti?

Crediamo che per rispondere a questa domanda sia necessario fare un passo indietro e focalizzare per un momento l'attenzione sui tre pilastri che caratterizzano questo paradigma recente ed innovativo:

- incapsulamento
- ereditarietá
- polimorfismo

Aggiungiamo un'altra considerazione prima di addentrarci nell'esame del valore formativo del tema; Nell'articolo [Car85] viene proposta la seguente *equazione*:

object-oriented=data abstractions + object types + type inheritance

Il primo punto da toccare é il concetto di *data abstraction*. Definire un tipo di dato astratto significa compiere un'operazione di astrazione paragonabile a quelle richieste nei concetti piú delicati della matematica. Prendiamo ad esempio la teoria assiomatica dei gruppi; un gruppo puó essere considerato un tipo di dato astratto: esso é caratterizzato dall'esistenza di un elemento neutro, di alcune operazioni ecc... Altra cosa é invece considerare un struttura algebrica che soddisfi gli assiomi dei gruppi (cioé un modello). Laddove abbiamo un risultato che vale per la teoria dei gruppi esso varrá anche in ogni suo modello. Allo stesso modo un'interfaccia di un oggetto (sia essa considerata esplicitamente come in Java o implicitamente come in C++) definisce alcuni assiomi che tutte le sue implementazioni devono soddisfare. Naturalmente come poi effettivamente é fatto il modello non interesserá all'utilizzatore finale, interesserá soltanto l'interfaccia. Sebbene spesso si nasconda questo principio che in informatica prende il nome di *incapsulamento* gli allievi hanno giá confidenza con esso: tutta l'analisi che si svolge nelle scuole secondarie si basa sui reali assiomatizzati dalla teoria dei campi ordinati completi, e i risultati ottenuti valgono per ogni suo modello. Di certo quando un allievo studia una funzione non pensa se il modello che usa é quello basato sulle successioni di Cauchy o sulle sezioni di Dedekind!

Abbiamo allora evidenziato uno dei valori formativi del principio dell'incapsulamento (dico come si comporta una classe senza spiegare come fa a realizzarlo): sviluppare le capacitá di astrazione degli allievi e le loro competenze logiche. Come poi la comprensione di questa colonna della OOP possa incidere su un buono stile di programmazione, verrá scoperto dagli allievi con l'esperienza nella realizzazione di software articolati (preferibilmente in gruppo): infatti in questi contesti piú si riesce ad astrarre, meglio si organizza lo sviluppo del software.
É altrsí interessante di per sé questa analogia tra matematica ed informatica, discipline che spesso prendono incomprensibilmente strade molto diverse.

Ma la programmazione ad oggetti ha altre qualitá didattiche. Eriditarietá e polimorfismo sono due aspetti cruciali. L'ereditarietá ha un semplice corrispettivo matematico: data la teoria assiomatica dei gruppi prendo in considerazione la teoria assiomatica dei gruppi commutativi. É chiaro che in questo caso tutti i risultati che avevo ottenuto precedentemente con lo studio della teoria dei gruppi li *eredito* nella teoria dei gruppi communtativi. Quindi probabilmente, grazie al nuovo assioma, otterró nuovi risultati.
Spesso ci si limita a presentare la questione dell'eriditarietá in questo modo e, in un certo senso puó essere anche didatticamente accettabile a seconda degli scopi che l'insegnante si prefigge. L'eriditarietá diventa quindi una semplice relazione *é un*; se questo é vero vale anche il principio di sostituibilitá di Liskov[1] (Cf. [Lis88]) (ecco l'introduzione del polimorfismo[2]): dove mi aspetto un oggetto di classe A posso anche mettere un oggetto di classe A_1 purché valga la relazione $A_1 é\ A$. Prendiamo in considerazione un caso in cui la sostituibilitá presenta qualche problema. Se consideriamo la classe dei rettangoli, allora certamente la classe dei quadrati é una sua sottoclasse cioé, accettando la sostituibilitá, un suo sottotipo. Tuttavia su di un rettangolo é possibile modificare la lunghezza dei lati indipendentemente l'uno dell'altro, mentre in un quadrato no. Quindi un metodo che acquisisca come parametro un rettangolo, ma che riceva un'istanza di quadrato puó fallire. Questo é il noto paradosso del Quadrato-Rettangolo. Esso sembra condurci in una contraddizione logica, e che quindi la sua trattazione sia addirittura didatticamente controproducente. Prima peró di precipitarci su questa conclusione facciamo due osservazioni:

[1]Il principio si basa sull'idea di far coincidere, come comunemente accade, l'idea di sottoclasse con l'idea di sottotipo

[2]In questo contesto parlaremo della forma di polimorfismo caratteristica della OOP cioé del polimorfismo per inclusione. Nondimeno riteniamo che sia importante che gli allievi abbiano la possibilitá di confrontarsi con il polimorfismo ad-hoc dei linguaggi (es. operatori +)

- ci stiamo trovando di fronte ad un problema che é sí logico ed astratto ma che ha anche dei risvolti pratici enormi. A dimostrazione di questo cito il fatto che vi sono numerosi articoli che trattano questo problema in riviste non certo di stampo accademico. Infatti il problema del paradosso Qadrato-Rettangolo rischia di non far funzionare i programmi!

- l'approfondimento di questo paradosso che sembra portare ad una contraddizione l'astrazione logica tanto ricercata in informatica, con la pratica della programmazione. Perché logicamente questo paradosso non sussite?

Il problema sta nella modificabilitá dello stato degli oggetti informatici. Il paradosso pué essere riassunto nella diversitá dell'interpretazione di *x=x+1* in informatica e in matematica. Nella programmazione un oggetto non ha uno stato costante, ma puó variare durante il suo ciclo vitale. Se ad esempio ho l'oggetto *Insieme* di solito sono previste le operazioni di inserimento di un elemento e di esclusione. Queste operazioni non generano un altro oggetto lasciando inalterato il primo, ma vanno proprio a modificare lo stato di questo. Appare evidente come in generale ció in matematica non sia vero. Ecco allora che il paradosso Quadrato-Rettangolo si presenta soltanto quando una classe permette all'utente di modificare il proprio stato durante l'esecuzione del programma.

Quanto detto precedentemente voleva essere un esempio di un lavoro formativo che riteniamo utile per lo sviluppo delle competenze logiche degli allievi.

2.2 Il linguaggio e la didattica

Spesso nella didattica dell'informatica ci si trova di fronte al problema della scelta del linguaggio di programmazione da adottare. Il problema é piú delicato di quanto possa sembrare in prima istanza in quanto alcuni aspetti che si ritengono concettualmente importanti possono non trovare una corrispettiva implementazione da parte del linguaggio. Se ad esempio il docente ritiene che abbia senso trattare il tema della controvarianza nella ridefinizione dei metodi, non dovrá scegliere probabilmente Java come linguaggio di programmazione; se invece il docente ritiene che sia opportuno abituare gli

allievi ad una programmazione sicura[3], invece, la scelta probabilmente non ricadrá su C++. Un altro aspetto poi entra in gioco ed é la commerciabilitá di una competenza: se si stanno formando degli specialisti da inserire nel mondo del lavoro si dovrebbe cercare di coniugare il valore formativo di un tema con la sua utilitá e diffusione negli ambiti lavorativi.

Con queste premesse ritengo che i linguaggi che sarebbero didatticamente efficaci purtroppo non sono altettanto diffusi: come negare la buona fondatezza di lunguaggi come MODULA3, ADA, Eiffel? In questi tre casi la OOP trova un ambiente molto fertile per la sua didattica.
Tuttavia l'ultimo principio dichiarato nel paragrafo precedente, cioé la valenza commerciale del linguaggio, non é soddisfatto. La scelta quindi si confina solitamente tra il linguaggio C++ e il linguaggio Java.

2.2.1 Didattica della OOP con C++

Molte critiche sono state mosse al C++ come linguaggio per la didattica della OOP. Il controllo dei tipi é debole, non ci sono controlli (o sono ridottissimi) in fase di run time, non dispone di librerie grafiche ecc...
La flessibilitá del linguaggio, che gli permette di essere adottato per gli scopi piú diversi (dalla programmazione di droidi al software applicativo, alla scirttura dei sistemi operativi), gli é imputata, sotto un profilo didattico, come un difetto. Ciononostante ritengo che esso non possa essere escluso dalla rosa dei linguaggi adottabili perché quelli che abbiamo acconsentito di considerare difetti didattici, possono essere superati da un'attenta azione dell'insegnante. D'altra parte il principio é generale: non sempre quello che un linguaggio consente di fare é buona programmazione. Sta dunque al programmatore e, nella sua fase di acquisizione delle competenze all'insegnante, prestare attenzione che un programma non sia solo funzionante ma sia anche un buon programma.

2.2.2 Didattica della OOP con Java

Scegliere Java come linguaggio per la didattica della OOP risolve molti dei problemi citati parlando del C++. Si effettuano i controllo sugli indici dei vettori, la strutturazione é rigorosamente ad oggetti, non esiste l'artimetica dei puntatori (qulcuno si spinge a dire che non esistono i puntatori, ma francamente questa affermazione ci sembra molto azzardata) che puó introdurre

[3]per programmazione *sicura* intediamo ad esempio la capacitá che ha Java di segnalare che un indice non appartiene ad un Array, o il lanciare eccezioni in caso di uso inappropriato degli oggetti

errori difficilmente rilevabili ecc...
D'altro canto Java porta con sé un bagaglio a nostro avviso notevole di problemi sia di organizzazione didattica che strutturali:

Il problema didattico principale che mi preme segnalare é quello della transizione da un linguaggio imperativo Pascal-like verso Java. Lo studente si trova da subito di fronte a due scogli:

- l'acquisizione di una sintassi nuova, di un modo diverso di impostare i cicli
- lo scontro obbligatorio con la programmazione ad oggetti che lo obbliga ad usare una serie di parole chiave senza la cosapevolezza di quello che sta facendo. Giusto per citare un esempio, egli dovrá usare nel suo primo programma almeno le seguenti: *import, static, class, public*. Anche tutto l'input/output é naturalmente orientato agli oggetti e quindi, se per scrivere una stringa a console video si é costretti ad usare l'operatore . con l'istruzione *System.out.println(...)*, per leggera da tastiera é addirittura necessario istanziare degli oggetti con l'operatore *new*.

Nel passaggio da un linguaggio Pascal-like a Java c'é altrsí il rischio di instaurare negli allievi quello spiacevole stile di programmazione (anche se molto in voga) che porta gli sviluppatori a dichiarare *static* tutto il possibile.

Se al paragrafo precedente abbiamo proposto delle critiche probabilmente superabili da un'attenta didattica del docente, piú problematiche risultano quelle che proponiamo adesso. Java é un linguaggio didatticamente difficile perché:

1. nel tentativo di semplificare e ridurre la sintassi, si é ridotto ad usare la stessa indicazione per concetti diversi. É il caso dell'operatore ., ad esempio: se compare dopo il nome di una classe invoca un metodo di classe, se compare dopo il nome di un oggetto invoca un metodo dell'istanza. Concettualmente le operazioni sono molto diverse (in C++ si distingue infatti l'operatore . dall'operatore ::) ma sotto un profilo sintaticco, leggendo il codice, non é possibile sapere, in generale, se un identificatore é usato come nome di classe o di oggetto[4].

[4]Per questa ragione le stesse indicazioni SUN consigliano di far cominciare i nomi delle classi con la lettera maiuscola e quelle degli oggetti con la lettera minuscola. Benché sia un consiglio estremamente valido, che va trasmesso agli allievi motivandolo, ci pare che sia un'argomentazione debole per gustificare il limite citato

2. i puntatori sono stati eliminati apparentemente, anzi direi solo sotto un profilo sintattico. L'assegnamento tra due oggetti é in realtá un assegnamento tra puntatori (*handle*); anche qui la semplificazione sintattica non va di pari passo con quella concettuale. In C++ se a e b sono puntatori ad oggetti della stessa classe allora é sia possibile eseguire l'assegnamento tra puntatori: $a = b$ che tra oggetti $*a = *b$. In C++ quindi non v'é differenza semantica tra l'operazione di assegnamento sui tipi primitivi e sugli oggetti, in Java dobbiamo ammettere lo studio dei puntatori per avere la stessa coerenza.

3. non sembrano neanche del tutto leciti i typecast impliciti di Java, in particolare ci sembrano azzardate le promozioni di tipo $int \rightarrow String$ e simili. É anche incomprensibile la ridefinizione del solo operatore + solo per le stringhe (concatenazione); perché non ridefinire anche l'operatore di confronto?

4. In Java non é previsto il passaggio dei parametri per indirizzo, causando talvolta dei problemi nell'utilizzo dei metodi ricorsivi (si pensi ai problemi correlati all'inserimento in coda nelle liste).

Il commento del creatore del linguaggio C++, Stroustrup, su Java é[5]:

> Much of the relative semplicity of Java is -like for most of new languages- party an illusion and party a function of its incompleteness. As time passes, Java will grow sgnificantly in size and complexity.

2.2.3 Conclusioni sulla scelta del linguaggio

La scelta del linguaggio appare quindi come un'operazione complessa, che deve tener conto di molti aspetti. Spesso si differenziano i linguaggi commerciali da quelli didattici, gli uni efficienti e flessibili, gli altri ben organizzati ed efficaci nella didattica. Per comprendere la portata di questa difficoltá proponiamo un estratto di un'intervista a Niklaus Wirth, autore di linguaggi di programmazione come Pascal, Modula 2 e Oberom, nonché insegnante di informatica all'ETH di Zurigo:

> Il fatto che fossi un insegnante ha avuto un'influenza decisiva nel creare linguaggi e sistemi il piú semplici possibile per poter

[5]Gran parte della relativa semplicitá di Java é -come per la maggior parte dei nuovi linguaggi- in parte un'illusione e in parte una conseguenza della sua incompletezza. Piú passa il tempo, piú Java crescerá significativamente in dimensione e complessitá

> concentrare i miei insegnamenti sull'essenza della programmazione, anziché sui dettagli dei linguaggi o delle notazioni.

Traspare esattamente l'idea di un linguaggio come strumento per far acquisire dei concetti: per questo l'uso dello strumento deve comportare meno difficoltá possibili per l'allievo in modo tale che egli possa concentrarsi sulle questioni cruciali.
L'obiezione tipicamente é che, seguendo questa filosofia, si tende a creare un insegnamento accademico, autoreferenziale, che non ha contatti reali con la comunitá dei programmatori. Riprendendo la stessa intervista, l'osservazione dell'intervistatore é stata:

> Quando ho l'occasione di parlare con un professore e con un programmatore, vedo fin troppo spesso che essi pensano al software in modi totalmente diversi. Anche una recente ricerca dell'IEEE sul futuro del software ha sostanzialmente dimostrato che non vi é alcuna intersezione tra le opinioni degli accademici e dei professionisti del software.

Obiezione alla quale Wirth risponde:

> Anche secondo me la lenta, costante separazione dell'accademia dalla pratica della programmazione é deleteria.

E che noi ci sentiamo di condividere.

Capitolo 3

Studio di un innovativo approccio alla didattica OOP

Le problematicitá dell'introduzione della programmazione ad oggetti sono molte; lo studente infatti si trova di fronte ad un paradigma di programmazione diverso, in cui l'approccio ai problemi, la progettazione del software, la rappresentazione dell'informazione si discostano dalla famosa equazione:

Programmi=Algoritmi + Strutture dati

Allora da una parte ci pare che sia importante indicare fin da subito a quali esigenze risponde l'introduzione di questa innovazione. Sicuramente non risolviamo problemi che prima erano irrisolvibili perché, sappiamo dalla teoria della calcolabilitá, la programmazione ad oggetti non é maggiormente espressiva rispetto a quella imperativa tradizionale. D'altra parte é importante che gli allievi abbiano un approccio graduale al nuovo paradigma; se disponiamo di un linguaggio che solo parzialmente[1] é orientato agli oggetti, come il C++, possiamo pensare di individuare tre fasi:

1. Inizialmente gli allievi useranno gli oggetti. Impareranno ad usarne i metodi, a creare istanze, ecc...

2. Successivamente impareranno a creare classi autonomamente. In questo caso lo scoglio da superare é il far comprendere l'importanza della rappresentazione dell'informazione, dell'incapsulamento e il concetto di stato.

3. L'ultimo passo é la comprensione dell'ereditarietá e del polimorfismo. A nostro avviso questi due pilastri della OOP sono comprensibili a fondo

[1] intendiamo con questo che permette una programmazione ibrida con l'utilizzo di oggetti all'interno di una programmazione imperativa

solo se gli allievi sono invitati a lavorare concretamente su progetti che abbiano le seguenti caratteristiche:

(a) coinvolgano tutta la classe nella realizzazione dello stesso programma: in questo modo sará apprezzabile la modularitá della OOP e il vantaggio dell'astrazione garantita dall'incapsulamento

(b) si prestino all'individuazione di classi: non tutti i progetti, anche se di grossa portata, si prestano ad essere sviluppati con un approccio orientato agli oggetti. É bene pertanto che l'insegnante stia attento a proporre il giusto problema.

(c) siano riconducibili a design patter ben noti: oltre a garantire quanto richiesto dal punto precedente, il riferimento ad un design patter, come ad esempio quello *modello-vista-controllore*[2] nel caso di programmi con interfacce grafiche, ha sia il pregio di abituare a stili di programmazione standard, sia quello di guidare gli allievi verso una comprensione piú chiara di come si organizzi un programma con la OOP.

In questo capitolo mostreremo l'utilizzo di una libreria scritta per C++ dal prof. Renato Conte, basata sull'idea della vecchia tartaruga Logo, che puó risultare molto utile alla comprensione di alcuni concetti della OOP e della programmazione imperativa in generale.

3.1 Richiami alla tartaruga Logo

In this culture we belive (correction: we know) that children of all ages and from all social backgrounds can do much more than they are belived capable of doing. Just give them the tools and the opportunity. Seymour Papert.

Il Logo é un linguaggio di programmazione ideato con finalitá didattiche dal matematico ed informatico americano Seymour Papert. Finora il Logo é stato visto, specie in Italia, come un linguaggio elementare, adatto ad un primo approccio alla programmazione o alla geometria, ma nulla piú (Cf.[Laz98]). Piú in generale Logo permette l'acquisizione di concetti molti delicati della programmazione: il concetto di variabile, di sottoprogramma, di ricorsione, di iterazione.

[2]Questo design patter stabilisce la suddivisione logica tra le classi che gestiscono la rapperesentazione grafica (vista), le classi che gestiscono l'input utente (controllore) e quelle che si occupano dello stato del programma in modo astratto (modello)

Operare in ambiente Logo significa programmare i movimenti di una tartaruga stilizzata che si muove su uno schermo in risposta ai nostri comandi. La tartaruga, vista come entitá geometrica, é caratterizzada da:

1. la sua posizione nel piano
2. il suo orientamento

I comandi elementari per muovere la tartaruga sono *AVANTI, DESTRA, SINISTRA* che modificano rispettivamente la posizione della tartaruga e il suo orientamento. La tartaruga, muovendosi, lascia una traccia sulo schermo. Per approfondimenti si puó consultare [Laz98]. Per esempi di indicazioni su esperienze didattiche basate su Logo rinviamo a [Sop99], [Ric99].

La strutturazione del linguaggio Logo si basa sui risultati della ricerca pedagogica di Piaget; esso nasce con scopi fortemente didattici e si basa su alcuni assunti (Cf. [Pap99]):

- L'incontro con la programmazione é un aspetto molto importante della formazione degli allievi nella nostra cultura.
- I bambini sono in grado di programmare a partire dalla giovane etá.
- Crediamo in un approccio costruttivista dell'apprendimento.

Ci troviamo di fronte, quindi, ad uno strumento per un apprendimento costruttivista e come lo stesso Papert spiega, il linguaggio Logo cerca di favorire la *lerning by doing strategy* (Cf. [Pap99-bis]).

3.2 Descrizione dell'azione didattica

La trasposizione delle tecniche usate dal linguaggio Logo per la didattica della programmazione OOP non puó non ricalcare gli stessi principi che Papert ha enunciato nei suoi articoli.
Il prof. Renato Conte mette a disposizione dei docenti interessati uno strumento didattico che sposa la filosofia del Logo fino in fondo, ma costituisce un appoggio alla didattica della programmazione in C++. La libreria *EASYC* puó essere scaricata liberamente dal sito dell'universitá di Venezia[3] assieme alla relativa documentazione; inoltre é a disposizione il libro *Il mondo degli oggetti: programmazione in C++* di R. Conte in cui si spiega l'implementazione della classe Tartaruga (Cf. [Con96]).

[3] www.dsi.unive.it/~conte

3.3 Analisi di casi effettivamente sperimentati

Presentiamo in questo contesto alcuni casi di studio effettivamente sperimentati assieme al docente nella didattica della OOP.

3.3.1 Passaggio dei parametri

A differenza di molti altri linguaggi il C++ ammette il passaggio dei parametri per copia e per indirizzo. Vogliamo utilizzare la tartaruga per aiutare gli allievi alla comprensione della differenza tra i due modi. Facciamo osservare che sebbene C++ ammetta il passaggio per indirizzo e non C, qusto tema non é strettamente legato alla programmazione ad oggetti ed é quindi spendibile anche nelle classi terze (come infatti avviene nella pratica didattica). Sottoponiamo agli allievi questo problema.

Esempio 1 *Disegna un albero stilizzato seguendo le seguenti regole: tutti i rami sono uguali e saranno disegnati da una funzione chiamata* disegnaRamo *che avrá per parametro una Tartaruga che si trova nel posto in cui deve cominciare il ramo. La funzione* disegnaFusto *sposterá la tartaruga in linea retta e ad intervalli casuali inserirá un ramo. Proponi una soluzione in cui la tartaruga passata alla* disegnaRamo *é per copia e una in cui é per riferimento. Cosa cambia?*

Con buona probabilitá gli allievi si troveranno di fronte al problema che nel disegnare il ramo la tartaruga che ricevono per parametro cambia di stato; se il passaggio avviene per copia il problema non sussiste, se invece avviene per referenza bisogna che pongano attenzione a ripristinare lo stato iniziale del parametro prima di terminare il disegno, altrimenti la *disegnaFusto* incontra dei problemi poiché non sa dove il disegno del ramo ha lasciato la tartaruga. Da un punto di vista grafico, nell'effettuare il disegno, di fronte ad un passaggio per copia, l'oggetto Tartaruga si duplica e la nuova tartaruga é distinguibile dall'originale perché é cerchiata. Quindi l'effetto che vedrá l'allievo di fronte al disegno del ramo col passaggio per copia sará il seguente:

1. La tartaruga si duplica
2. La nuova tartaruga (cerchiata) disegna il ramo mentre la vecchia rimane ferma sul fusto
3. Al termine del disegno del ramo la copia scompare e si ricomincia a muovere la tartaruga del fusto

Mentre di fronte al passaggio per indirizzo tutto viene svolto dalla stessa Tartaruga!

3.3.2 Ricorsione

Nella mia esperienza didattica credo che la ricorsione sia uno degli aspetti piú ostici da trattare nel processo di insegnamento-apprendimento. I programmi ministeriali e la prassi didattica osservata nel tirocinio, vogliono che la ricorsione sia trattata in terza (quindi o sui vettori o sugli int). A mio avviso questo é un errore che sarebbe meglio evitare: a molti degli allievi sfugge la natura del principio di induzione per i naturali su cui si basa la ricorsione. Paradossalmente mi pare diventi piú comprensibile se la ricorsione é introdotta nel trattamento di strutture dati ricorsive (come le liste o gli alberi). Una valida alternativa a questi due modi é l'intrduzione della ricorsione del disegno con la Tartaruga di semplici frattali. Legare la ricorsione allo studio dei frattali non é una novitá anche se, dobbiamo sottolineare, in informatica trattiamo algoritmi calcolabili mediante ricorsione finita, mentre il disegno dei frattali non lo é (quindi ci accontentiamo di approssimazioni). Come al solito pensiamo sia meglio servirsi di un esempio:

Esempio 2 *Vogliamo disegnare un ramoscello d'albero di lunghezza l come segue:*

- *Un ramoscello puó essere un segmento lungo l passi*
- *Un ramoscello puó essere un segmento lungo $l/2$ con una biforcazione sulla punta dove germogliano altri due ramoscelli lunghi $l/2$.*

Scrivi una funzione C++ che disegni il ramoscello descritto con un numero di biforcazioni e una lunghezza specificabile tramite parametro.

Osserviamo in figura 3.1 il codice che si ottiene e in figura 3.2 il risultato di una sua esecuzione: Alcune osservazioni sulla valenza didattica di esercizi di questo tipo:

- La scrittura dell'algoritmo ha un corrispettivo grafico, la struttura ripetitiva del ramoscello sempre con meno arborescenze é visibile dal disegno della tartaruga. Questo facilita notevolmente gli allievi nella comprensione della ricorsione da un punto di vista matematico
- Ad ogni chiamata della funzione ricorsiva si istanzia nello stack un nuovo ambiente. Gli allievi possono notare questo vedendo le tartarughe (passate per copia) moltiplicarsi fino a raggiungere l'estremitá

```
void DisegnaRamoscello(Tartaruga t, int lung, int passi) {
    if (passi>0) {
        t.Avanti(lung/2);
        t.Sinistra(45);
        DisegnaRamoscello(t, lung/2, passi-1);
        t.Destra(90);
        DisegnaRamoscello(t, lung/2, passi-1);
    }
    else //caso base
        t.Avanti(lung/2);
}
```

Figura 3.1: Algoritmo che permette alla Tartaruga di disegnare un semplice frattale

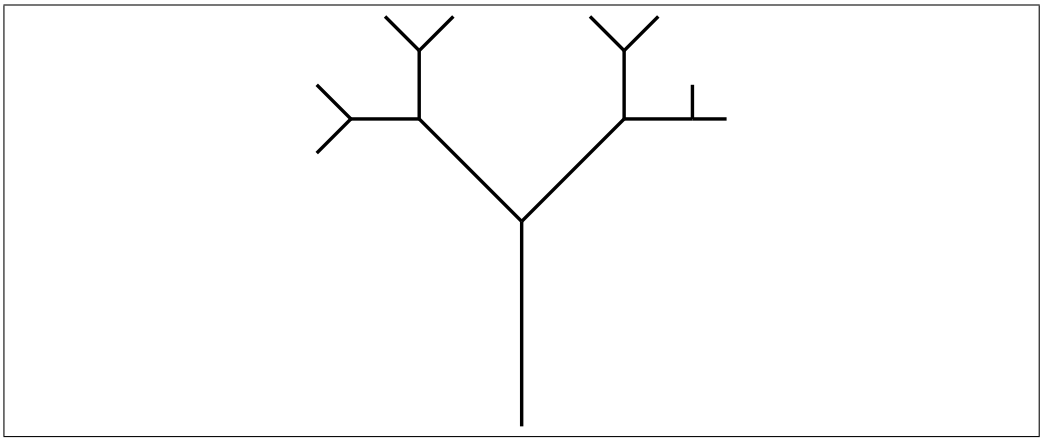

Figura 3.2: Esempio di esecuzione della funzione il cui codice é riportato in figura 3.1. La chiamata iniziale per ottenere questo disegno sará DisegnaRamoscello(t,100,3)

del ramo e poi morire in senso opposto. Questo facilita sicuramente la comprensione del funzionamento dello stack e come viene implementata la ricorsione nei linguaggi di programmazione. Uno dei misconcetti nell'uso della ricorsione é proprio il non comprendere il meccanismo di

istanziazione dell'ambiente ad ogni chiamata: pensiamo in particolare a quelle volte in cui ci si é trovati di fronte ad un errore come quello mostrato in figura 3.3

```
int fattoriale(int n) {
    int risultato=1;
    if (n>0) {
        risultato=risultato*n;
        fattoriale(n-1);
    }
    return risultato;
}
```

Figura 3.3: Errore nel calcolo del fattoriale (classe III) probabilmente dovuto alla non comprensione del funzionamento dello Stack nelle chiamate ricorsive (lo studente cerca di usare una variabile sul modello dell'accumulo). Facciamo notare che il metodo sarebbe funzionante (con qualche avvertenza) se la variabile *risultato* fosse stata dichiarata *static*. Nonostante ció si sarebbe trattato di un pessimo stile di programmazione da non considerare corretto.
.

3.3.3 Classi, oggetti, stato

Uno dei misconcetti che talvolta investe anche gli specialisti dell'informatica é la confusione tra classe ed oggetto. Non stiamo pensando ad una confusione nella definizione ma piuttosto ad una vera e propria interpretazione errata di concetti[4]. Non é d'aiuto che in alcuni linguaggi (es. Visual Basic) tale distinzione non abbia un corrispettivo sintattico.
Da un punto di vista didattico ció che gli allievi devono riuscire a capire é che:

[4]In letteratura esistono in effetti due concezioni della programmazione ad oggetti. In lunguaggi come Java o C++ la classe é assimilabile al tipo e l'oggetto all'istanza. In Galileo97 il concetto di classe é diverso: l'istanziazione avviene per clonazione di oggetti e la classe é l'insieme di istanze riconducibili al medesimo padre. Naturalmente non é questa la confusione tra classe ed oggetto alla quale pensiamo.

1. si possono avere piú istanze (oggetti) della stessa classe

2. ogni oggetto ha un suo stato che durante l'esecuzione del programma evolve indipendentemente dagli stati degli altri oggetti della medesima classe

3. l'identitá dell'oggetto non é individuata soltanto dallo stato. Due o piú oggetti possono avere lo stesso stato ma rimangono comunque distinti

4. l'invocazione di un metodo avviene su un oggetto, l'effetto della sua esecuzione dipende, in generale, dallo stato dell'oggetto.

Sicuramente alcune di queste consapevolezze possono essere acquisite con le attivitá precedentemente descritte: pensiamo ad esempio all'osservazione di ció che accadeva alle tartarughe passate per copia, come lo stato della copia si modificasse indipendentemente dall'originale.
Tuttavia possiamo pensare a delle attivitá mirate proprio per raggiungere questo obiettivo. Il docente accogliente chiedeva agli allievi a questo pro di realizzare il seguente gioco:

Esempio 3 *Vogliamo realizzare un gioco nel quale cinque tartarughe si sfidano in una gara podistica. Il giocatore deve poter scommettere su una delle cinque tartarughe; se questa arriva prima delle altre allora il giocatore vince la puntata altrimenti la perde.*

Agli allievi é richiesto di lavorare con un vettore di tartarughe. In un ciclo ogni tartaruga viene spostata di una distanza casuale (in modo da dare l'idea della velocitá), a seconda del primo oggetto che raggiunge una certa posizione si determina il vincitore.
Per risolvere questo piacevole esercizio gli allievi devono acquisire i concetti espressi precedentemente. Appare evidente quindi che la classe é la stessa ma vi sono molti oggetti della stessa classe. É altresí importante evidenziare il fatto che il comando *AVANTI(20)* invocato su un oggetto ha un effetto diverso a seconda dello stato della tartaruga: se questa si trova al centro dello schermo e orientata a Nord verrá disegnata una certa linea, altrimenti la linea sará diversa. In figura 3.4 si mostra un codice esemplificativo in cui si creano piú tartarughe che vanno a zonzo sullo schermo.

3.3.4 Incapsulamento

Il principio dell'incapsulamento non solo ha un'importanza notevole nella pratica della programmazione perché permette una progettazione ad alto livello del software, ma anche ha un valore formativo estremamente elevato

```
void AZonzo() {
    cont N=7;
    Tartaruga T[N];
    for (int i=0; i<N; i++) {
        T[i].TempoPasso(4); //rallenta le tartarughe
        T[i].UsaPennelloLargo();
    }
    do //Muovi le tartarughe{
        for (int i=0; i<N; i++) {
            T[i].Avanti(random(50));
            if (T[i].FuoriSchermo())
                T[i].Indietro(55);
            if (random(5)==0)
                T[i].Destra(random(100));
        }  while (!kbhit()); //aspetta pressione tasto
}
```

Figura 3.4: Funzione che sposta N=7 tartarughe a caso sullo schermo. Aiuta gli allievi a comprendere l'indipendenza dello stato degli ogetti della stessa classe, e il fatto che gli effetti dell'invocazione di un metodo su un oggetto dipendono dallo stato dell'oggetto stesso. Spesso l'insegnante chiede di integrare l'esercizio con un mirino grafico spostato da mouse col quale bisogna centrare le tartarughe (in questo modo gli allievi devono anche interrogare lo stato degli oggetti).

perché consente all'insegnante di trattare il delicato tema della rappresentazione dell'informazione. Sappiamo nella fattispecie che lo stato della tartaruga del Logo é dato dalla sua posizione nel piano e dal suo orientamento[5]. Tuttavia l'allievo che usa l'oggetto ignora come tali informazioni vengano memorizzate; giusto per esemplificare possiamo poporre agli allievi alcune rappresentazioni dello stato della tartaruga:

- una coppia di *int* puó rappresentare le coordinate della tartaruga nel schermo, e un *float* puó rappresentare un angolo in gradi.
- Le coordinate della posizione potrebbero essere polari invece che cartesiane (coppia di *float*), e l'orientamento in radianti.

Ma le combinazioni possono essere davvero tante! pensiamo semplicemente all'arbitrarietá della scelta dei punti di riferimento!
Il principio dell'incapsulamento afferma che l'utente di una classe (cioé il programmatore che ne crea istanze) non é tenuto a conoscere la rappresentazione dell'informazione sullo stato degli oggetti, tanté che l'utilizzo degli oggetti tartarughe si é proprio basato su quell'implicito assunto.

3.3.5 Input, output: i flussi

La ridefinizione degli operatori concessa dal C++ permette che gli allievi entrino in contatto con il modello di input/output dello *stream*. Sebbene anche con l'introduzione della tartaruga lo standard input rimanga la tastiera e lo standard output rimanga lo schermo, é sicuramente importante far cogliere alcune differenze. Nell'istruzione:

```
cout<<"Prova";
```

chiedo di stampare il flusso di caratteri *Prova* allo standard output. Se t é un'istanza di una tartaruga allora l'istruzione:

```
t<<"Prova";
```

chiede ad una tartaruga di gestire il flusso di caratteri *Prova*. Come giá visto, l'esecuzione del comando dipenderá dallo stato della tartaruga (posizione sullo schermo, spessore del pennello, direzione). Analogamente vengono ridefinite le operazioni di input (supponiamo x una variabile int):

```
t>>x;
```

[5]L'oggetto Tartaruga della libreria che consideriamo é piú sofisticato e ha uno stato piú articolato. Ad esempio deve memorizzare se la tartaruga é visibile o no, se il pennello é alzato o abbassato

La programmazione ad oggetti assume allora anche un ruolo importante nell'affrontare i modelli astratti di gestione dell'input e dell'output; l'utilizzo di questo semplice oggetto grafico fornisce l'opportunitá di prendere in considerazione dispositivi non standard in modo relativamente semplice.

3.3.6 Ereditarietá e polimorfismo

Le altre due colonne portanti della OOP sono appunto l'ereditarietá e il polimorfismo. Anche questi concetti ci sembrano essere molto delicati, per almeno le seguenti ragioni:

- il polimorfismo viene spesso associato soltanto alla programmazione ad oggetti. Questo é un errore concettuale abbastanza diffuso, pochi insegnanti rendono consapevoli gli allievi che il polimorfismo s'incontra anche in linguaggi non orientati agli oggetti.
  Per esempio l'operatore di addizione (+) in molti linguaggi di programmazione é polimorfo: esegue la somma tra *int* ma anche tra *double*. La procedura di libreria *Write* del Pascal é polimorfa in quanto riceve parametri di tipi diversi (anche non compatibili).

- Ma anche nella programmazione ad oggetti puó capitare di trovare esempi di ereditarietá che male si sposano con il principio del polimorfismo. In un noto libro di testo per la didattica della OOP nella scuola secondaria superiore si trova l'esempio illustrato in figura (3.5).

La trattazione di questi argomenti é importante perché permette agli allievi di acquisire notevoli bagagli teorici che potrebbero favorire il livello d'astrazione. Riteniamo che sia insufficiente pertanto la proposta didattica che formuliamo con l'oggetto Tartaruga, e che le problematiche della covarianza, della controvarianza, dei paradossi debbano diventare patrimonio culturale degli allievi: la motivazione é di ordine epistemologico; gli studenti spesso vivono nella convizione che informatica sia sinonimo di tecnologia e, sebbene la tecnologia costituisca certo un supporto a questa scienza, dimenticano il fondamentale impianto teorico, dove oltretutto la ricerca appare piú interessata.
Riteniamo che si possa trattare l'ereditarietá dopo che gli allievi hanno avuto esperienza nel realizzare delle classi, anche semplici, in maniera autonoma. Vediamo un esempio di esercizio in cui gli allievi possono acquisire il concetto di ereditarietá:

Esempio 4 *Vogliamo realizzare un gioco che permetta una gara di automobili. L'utente deve guidare un'automobile in un tracciato senza toccarne i*

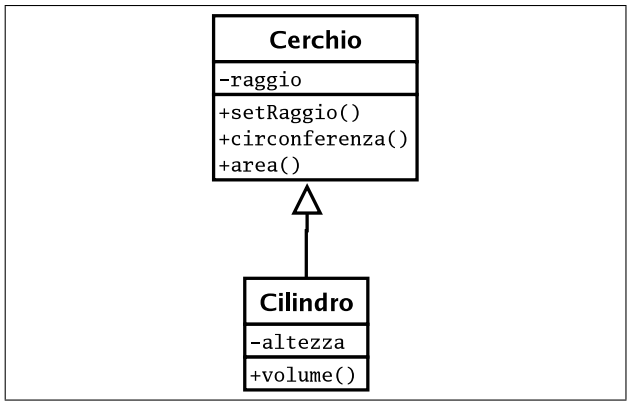

Figura 3.5: Brutto esempio di ereditarietá. Un cilindro non puó essere considerato un particolare cerchio, un metodo che si aspetta un cerchio come parametro puó ricevere un cilindro. Scambiando la gerarchia si ricade nel paradosso del *Quadrato-rettangolo*. Esempio tratto da [Lor01]
.

bordi, con la possibilitá di accelerare, frenare, sterzare a destra e sinistra. La sfida avviene contro una macchina comandata dal computer, un cronometro deve misurare i tempi.

Naturalmente gli allievi vengono guidati nella progettazione delle classi, e il pattern di riferimento sará quello mostrato in figura 3.6. La questione del polimorfismo emerge da subito nella stesura del programma: il metodo che controllerá se le auto urtano i bordi del percorso non cambiano nel caso di auto del computer o dell'utente, per esempio. Anche la ridefinzione dei metodi é coinvolta in questo progetto: pensiamo soltano al disegno dell'automobile che deve rimpiazzare quello della tartaruga[6]. Naturalmente lo svolgimento del progetto richiede una paziente ed attenta fase di progettazione iniziale che sará verificata dal docente prima dell'implementazione; gli allievi capiscono che non si puó programmare senza progettare, in modo particolare se il paradigma di riferimento é orientato agli oggetti.

[6]Ridefinizione o sovraccarico? Anche questo é un tema molto delicato e probabilmente merita di essere affrontato con la classe, anche se a nostro avviso ha una rilevanza cognitiva inferiore ai problemi connessi col polimorfismo. In ogni caso non crediamo che sia tanto importante che gli allievi ricordino quando avviene ridefinizione e quando avviene sovraccarico, ma piuttoso che comprendano il concetto e le differenze logiche. Infatti le regole cambiano da linguaggio a linguaggio, ma l'idea no.

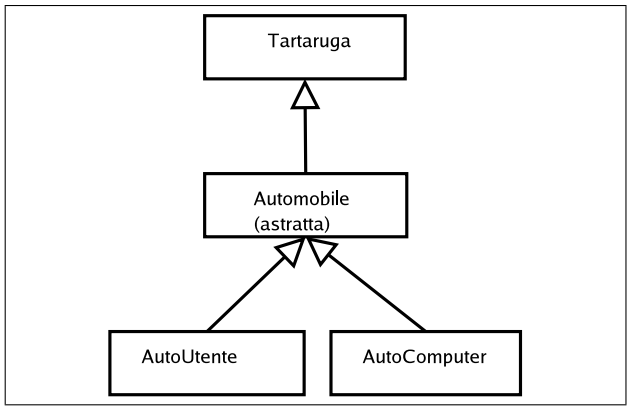

Figura 3.6: Pattern per la progettazione della gara tra tartarughe. Una classe astratta specializza la Tartaruga ad essere anche una macchina, due sottoclassi specializzano la Macchina in modo di dotarla di intelligenza artificiale (quella del computer) e di gestione dell'input dall'utente (quella del giocatore).

3.4 La libreria e la buona programmazione

Una caratteristica importante e da non sottovalutare della libreria EASYC4 é la seguente: al termine dell'esecuzione del programma, appaiono delle indicazioni, dei warning. Essi possono riguardare i seguenti aspetti non altrimenti rilevabili dal compilatore:

- Un oggetto della libreria é stato allocato in memoria dinamica senza mai venire deallocato.
- Si sono passati per copia degli oggetti di grossi dimensioni; appare conveniente, ove possibile, il passaggio per indirizzo con il modificatore *const*.
- Si sono dichiarati degli oggetti della libreria globali. Segno questo di cattivo stile di programmazione.

La libreria mette inoltre a disposizione array e stringhe Java-like in cui si effettua il controllo sugli indici degli array, si ridefiniscono gli operatori tra stringhe. Questo, unito al fatto che la sintassi richiesta per l'uso degli oggetti in C++ é abbastanza semplice (possono essere usati anche senza l'allocazione in memoria dinamica), puó costituire un'ottima base per l'introduzione alla programmazione, nel momento in cui ci si concentra maggiormente su una programmazione a livello alto.

Capitolo 4

Trasposizione dell'approccio a Java

Abbiamo pensato che sarebbe potuto risultare interessante proporre lo stesso approccio visto per C++ in Java. É stata pertanto progettata un tartaruga per il linguaggio Java. Rispetto all'originale C++ sono riscontrabili alcuni vantaggi ma anche molti svantaggi difficilmente superabili. I vantaggi riguardano sicuramente l'accattivante aspetto grafico che puó dare molte soddisfazioni agli studenti, e la progettazione della libreria (ricalca infatti il modello Vista-Controllo-Modello) che puó diventare essa stessa oggetto di studio. Tuttavia non tutte le attivitá didattiche pensate per il C++ possono essere trasposte in Java, sia per limitazioni interne del linguaggio (es. non esiste il passaggio per indirizzo), sia per difficoltá nella gestione della grafica.

4.1 La tartaruga in Java

Il progetto di trasposizione della libreria in Java, tutt'ora in corso e al quale stiamo attivamente collaborando, permette un'organizzazione dell'attivitá didattica diversa rispetto al tradizionale approccio verso Java. Abbiamo giá detto nell'introduzione che solo per scrivere una stringa sulla Console video, o leggerla, é necessario che gli allievi facciano un uso massiccio di costrutti legati alla programmazione ad oggetti, e questo puó risultare difficile se provengono da un linguaggio di programmazione imperativo. Per passare poi all'uso della grafica, la situazione diventa oltremodo complessa, e spesso si devono attendere mesi prima che gli allievi sviluppino gradevoli applicazioni grafiche. In figura 4.1 mostriamo un esempio di un risultato ottenuto con la libreria e in figura 4.2 il relativo programma.

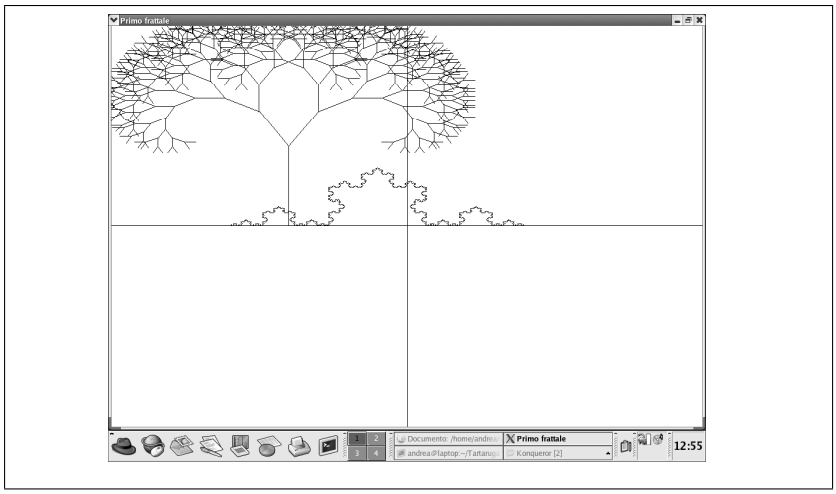

Figura 4.1: Frattali ottenuti con la libreria Tartaruga per Java

4.2 Valutazione delle problematicitá dell'intervento

Abbiamo giá accennato all'inzio che la trasposizione dell'idea della tartaruga a Java non ha lo stesso successo, a nostro avviso, rispetto al C++; cerchiamo di riassumerne i motivi:

- In ogni caso vi é uno scontro con la sintassi caratteristica della OOP in Java. In C++ posso istanziare un oggetto Tartaruga semplicemente con:

  Tartaruga t;

  In Java, invece, siamo costretti a ricorrere alla dichiarazione e quindi all'istanziazione dinamica con l'operatore new.

- In Java il passaggio avviene sempre per copia. Tuttavia la sensazione che ha lo studente é che la tartaruga venga passata per indirizzo, in quanto il suo stato viene modificato da metodi che l'acquisiscono come parametro. Appare quindi difficile affrontare proficuamente il tema della generazione degli ambienti dei metodi in questo modo, perché il passaggio di un riferimento per copia non é facile da capire[1].

[1]Nonostante tutta la documentazione della SUN sottolinei questo, molti testi affermano che in Java i tipi semplici sono passati per copia, mentre gli oggetti per indirizzo. Naturalmente questo é un errore che puó portare a misconcezioni molto gravi.

```
static private void albero (Tartaruga t, int n, int max) {
    if (n¡max) {
        t.avanti(200.0/(n+1));
        t.sinistra(30.0);
        albero(t, n+1, max);
        t.sinistra(-60.0);
        albero(t, n+1, max);
        t.sinistra(30.0);
        t.su();
        t.avanti(-200.0/(n+1));
        t.giu();
    }
}
static private void vonCock(Tartaruga t, int n, double lung) {
    double terzo = lung / 3.0;
    if (n==0)
        t.avanti(lung);
    else {
        vonCock(t, n-1, lung/3.0);
        t.sinistra(60.0);
        vonCock(t, n-1, lung/3.0);
        t.sinistra(-120.0);
        vonCock(t, n-1, lung/3.0);
        t.sinistra(60.0);
        vonCock(t, n-1, lung/3.0);
    }
}
```

Figura 4.2: Esempio d'uso della libreria Tartaruga Java. Codice dei metodi usati per realizzare i frattali di figura 4.1.

- Le animazioni nelle finestre Java non sono semplici come quelle che facciamo in C++, per cui la rappresentazione grafica degli spostamenti della tartaruga puó creare problemi.
- L'impossibilitá di ridefinire o sovraccaricare gli operatori ci impedisce

di trattare in modo approfondito il polimorfismo. Oltretutto Java non gestisce né i template[2] né la covarianza dei codomini, quindi lo studente é costretto ad usare le forzature di tipo ad ogni copia (*clone*) dell'oggetto tartaruga.

- Gli stessi metodi che useranno la tartaruga, sebbene possano essere pensati come metodi di classe[3], fanno parte di una classe e quindi puó essere diffocoltoso per gli allievi capire quello che effettivamente stanno facendo.

[2]Dalle anteprima della Sun sembra che questo limite verrá superato dalle prossime versioni

[3]Anche su questa strategia didattica non ci troviamo molto d'accordo

Capitolo 5

Conclusioni

Con questo lavoro abbiamo mostrato un'idea innovativa per l'introduzione della programmazione ad oggetti, sia essa in Java o in C++. L'idea geniale di Papert assume quindi una nuova vitalitá ma ne condivide ampiamente le idee costruttiviste fondanti, la filosofia del learning by doing, il sostegno motivazionale agli allievi che realizzano software piacevoli pur senza concentrare l'attenzione sullo studio di complesse librerie grafiche.
Sotto il profilo della sperimentazione didattica possiamo affermare che i risultati ottenuti sono stati davvero notevoli; l'osservazione nelle classi durante il tirocinio ha davvero dato risultati incoraggianti: le classi venivano coinvolte in progetti sui quali investivano molte energie.
Da questa esperienza é sorta l'idea di sperimentare lo stesso approccio per la didattica della OOP con Java, e quindi si é cominciato a lavorare in una libreria simile a quella realizzata dal prof. Renato Conte per C++. Purtroppo non v'é ancora stata occasione di provarla sul campo ma, come abbiamo argomentato nel capitolo precedente, prevedibilmente si inseriranno delle problematicitá che in C++ non sussistevano.
Oggi la maggior parte degli insegnanti opta per la didattica della OOP con Java ma, dall'esperienza osservata sul campo, ritengo che tale scelta non sia sempre ben ponderata a meno che non si utilizzino strumenti di facilitazione didattica di natura diversa. Il software BlueJ, realizzato da un consorzio di universitá, puó probabilmente essere di aiuto per la comprensione di alcuni aspetti peculiari della OOP, ma sicuramente non riesce a semplificare l'approccio alla programmazione come invece fa la tartaruga Logo.
Naturalmente attenderemo una sperimentazione in classe per poter convalidare o refutare queste ipotesi.

Bibliografia

[Car85] Luca Cardelli, Peter Wegner, *On understanding types, data abstractions, and polymorphism*, ACM Computer Surveys, 1985

[Laz98] Paolo Lazzarini, *Introduzione al MSW-Logo*, versione ipertestuale, 1998.

[Lis88] Barbara Liskov, *Data abstraction and Hierarchy*, ACM SIGPLAN Notices, 1988.

[Lor01] Agostino Lorenzi, *Il linguaggio Java*, ATLAS, 2001.

[Pap99] Seymour Papert, *What is Logo, And who needs it?*, Logo Computer System inc. 1999.

[Pap99-bis] Seymour Papert, *Papert on Piaget*, Time Magazine's special issue on The Century's Greatest Minds, page 105, 29 Marzo, 1999.

[Sop99] Sergei Soprunov, Elena Yakovleva, *The russian school system and the Logo approach*, Logo Computer System inc. 1999.

[Ric99] Jeff Richardson, *Logo in Australia: a vision of their own*, Logo Computer System inc. 1999.

[Con96] Renato Conte, *Il mondo degli oggetti: programmazione in C++*, edizioni Progetto Padova, 1996.