Introduzione alla programmazione orientata all'oggetto

seconda parte

di Sergio Polini

La volta scorsa abbiamo visto come la programmazione orientata all'oggetto consenta di estendere, mediante la definizione di «tipi di dati astratti», le possibilità di *information hiding* proprie della programmazione modulare. Per tipo astratto si intende, ripetiamo, un tipo che non venga caratterizzato semplicemente dai valori che possono assumere variabili ad esso appartenenti, e tanto meno da una particolare implementazione, che rimane «nascosta», ma piuttosto da un insieme di operazioni (va magari sottolineato che un tipo «astratto» è tale solo perché nel suo uso si prescinde — si *astrae* — dalle modalità di implementazione ma per il resto è ben concreto, è anzi tanto «reale» quanto i tipi predefiniti; è questo il motivo per cui Bjarne Stroustrup, il padre del C++, preferisce parlare di «tipo definito dall'utente»). Abbiamo fatto cenno a due vantaggi che ne derivano: si riduce in primo luogo il bisogno di variabili globali, in quanto lo «stato» di un oggetto appartenente al tipo astratto viene mantenuto in «campi» del tipo che, locali al singolo oggetto quanto una variabile locale ad una funzione, hanno per il resto la stessa persistenza delle variabili globali. Il fatto poi che i dettagli della implementazione rimangono nascosti, che l'accesso ad essi viene filtrato da apposite funzioni e procedure di interfaccia, consente di ridurre i tempi di test e di manutenzione di un programma; un cambiamento nella definizione di un tipo, infatti, purché non alteri le procedure di interfaccia (né la sintassi della chiamata, né il funzionamento), non comporta alcuna necessità di intervenire sul resto del programma.

In altra parte della rivista (rubrica Turbo Pascal) tento di fornire un esempio concreto di tali benefici, senza però ricorrere alla OOP. Mi sono avvalso delle sole possibilità offerte dalle **unit** del Turbo Pascal, un po' per mostrare gli aspetti di continuità tra la OOP e tecniche più tradizionali, un po' per ribadire che l'*incapsulamento* della rappresentazione di un tipo è solo un aspetto della OOP, è anzi quasi solo un prerequisito; c'è ben altro. Nella figura 1, comunque, vi propongo una possibile ridefinizione «orientata all'oggetto» della interfaccia del tipo *TRealMat* discusso nella rubrica: noterete che l'accesso al tipo è filtrato da un insieme di funzioni e procedure, mentre la rappresentazione (due interi e un puntatore ad un array) rimane nascosta, grazie all'uso della parola chiave **private** che il Turbo Pascal 6.0 ha mutuato dal C++.

Comunanza ed ereditarietà

Traduco con «comunanza» l'inglese *commonality*, termine con il quale si vuole descrivere la situazione in cui due o più tipi distinti hanno qualcosa in comune. Possiamo pensare agli oggetti tipici di una interfaccia utente: finestre,

```
unit Matrix1;

interface

type
  PReal      = ^real;
  PRealArray = ^TrealArray;
  TRealArray = array[1..65520 div SizeOf(real)] of real;
  PRealMat   = ^TrealMat;
  TRealMat   = object
    constructor Init(NumRighe, NumColonne: integer);
    destructor Done; virtual;
    procedure Read;                                          (* input dati *)
    procedure Write;                                         (* output dati *)
    function Elem(r, c: integer): PReal; virtual;            (* ptr a [r,c] *)
    function Det: real;                                      (* determinante *)
    procedure Inv(var Result: TRealMat);                     (* inversione *)
    function Righe: integer;                                 (* numero righe *)
    function Colonne: integer;                               (* numero colonne *)
  private
    NRighe, NColonne: integer;
    Dati: PRealArray;
  end;
```

Figura 1 - Una possibile versione «orientata all'oggetto» del tipo TRealMat proposto in altra parte della rivista. La parola chiave «private», che consente l'effettivo incapsulamento della rappresentazione del tipo, mancava nel Turbo Pascal 5.5, ma è disponibile nella versione 6.0.

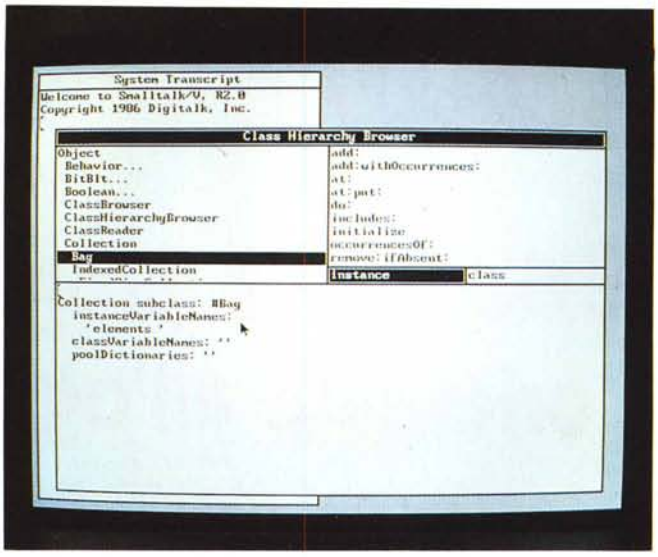

Figura 3 - Una tipica schermata SmallTalk, con il «Class Hierarchy Browser», che consente di ispezionare e modificare la gerarchia di classi, mostrata nel pannello in alto a sinistra. Nel pannello in alto a destra sono mostrati i metodi della classe Bag, nel pannello in basso la sua derivazione da Collection.

menu, dialog box, ecc. Sono diversi, ma hanno evidentemente qualcosa in comune: sono tutti rettangoli che devono poter apparire sullo schermo e poi sparire lasciando intatto quanto avevano coperto. Possiamo anche pensare alle matrici descritte nella rubrica Turbo Pascal di questo mese: un tipo implementato come array di numeri, un altro come array di puntatori per matrici «grandi», ma con una identica interfaccia e molte analogie nel codice delle diverse routine.

Può succedere di dover utilizzare in uno stesso programma oggetti analoghi, può soprattutto succedere di dover scrivere un nuovo programma con oggetti analoghi ma non identici a quelli già usati in un altro, o di estendere o modificare la funzionalità di un programma già fatto. Si tratta qui della cosiddetta *riusabilità* del codice: è chiaro che sarebbe desiderabile non dover ricominciare ogni volta da capo.

```
unit Matrix2;

interface

uses Matrix1;

type
  TPRealArray = array[1..65520 div SizeOf(pointer)] of PRealArray;
  TBigRealMat = object(TRealMat)
    constructor Init(NumRighe, NumColonne: integer);
    destructor Done; virtual;
    function Elem(r, c: integer): PReal; virtual;
  private
    Dati: TPRealArray;
  end;
```

Figura 2 - I linguaggi orientati all'oggetto consentono di derivare un tipo da un altro già definito, in modo tale che è sufficiente specificare solo quanto i due tipi hanno di diverso, in quanto il tipo derivato «eredita» automaticamente tutto quanto può rimanere invariato.

Immaginiamo di avere a disposizione una libreria di funzioni che consenta di realizzare una interfaccia utente con finestre di tipo «editor di testi» (o di lavorare su matrici implementate come array di numeri). Immaginiamo anche di non aver bisogno di righe di testo, ma di «righe di caselle», di dover cioè realizzare una interfaccia utente simile a quella di un foglio elettronico (o di dover lavorare su matrici più grandi di quanto consenta un array di numeri). Che fare? Avremmo in primo luogo bisogno del sorgente di quella libreria, senza il quale non potremmo fare altro che partire da zero. Anche disponendo dei sorgenti, tuttavia, il compito non sarebbe facile: dovremmo modificarli, ma questo richiederebbe una loro completa comprensione (non impossibile, ma ci vuole tempo!) e comporterebbe comunque il rischio di incappare in insidiosi bug. Chi lo abbia fatto almeno una volta sa bene che è un po' come cercare di accorciare una gamba di un tavolino. Anche ammesso che si riesca nell'intento in un tempo ragionevole, rimarrebbe un altro problema: se avessimo bisogno sia della vecchia che della nuova versione della libreria (sia «righe di caratteri» che «righe di caselle», sia matrici «piccole» che matrici «grandi», magari perché — facciamo l'ipotesi — queste ultime sono meno efficienti), potremmo solo scegliere tra rassegnazione ad un codice ridondante e modifiche ancora più ampie; potremmo cioè o rassegnarci alla duplicazione delle parti «comuni», o alla loro riscrittura con proliferazione di istruzioni **if, case** o **switch** per tener conto di quanto rimane di diverso in tipi molto simili ma non identici.

Ora proviamo a sognare. Abbiamo una libreria di funzioni, ma non ne abbiamo i sorgenti; ci servono tipi di dati analoghi a quelli disponibili, ma un po' diversi; riusciamo a implementarne sia la rappresentazione interna che il comportamento semplicemente descrivendo quanto vogliamo sia diverso; possiamo usare contemporaneamente, senza problemi, sia i tipi «vecchi» che quelli «nuovi». Per fortuna non è un sogno, ma ordinaria amministrazione per un linguaggio «orientato all'oggetto».

Già in Simula era possibile derivare una classe da un'altra, caratteristica che ritroviamo poi in SmallTalk, C++, Turbo Pascal. Nella figura 2 vediamo come si possa derivare un tipo da un altro in Turbo Pascal 6.0: se le funzioni e procedure dell'oggetto *TRealMat* (figura 1) accedono al campo *Dati* solo attraverso la funzione *Elem*, proprio come ho fatto nel codice che vi propongo nella rubrica, è possibile dichiarare un tipo *TBigRealMat* come **object** derivato da *TRealMat* ridefinendo solo il **constructor** *Init* (speciale procedura che si occupa della inizializzazione di un oggetto, corrispondente all'*IniMat* che trovate nella rubrica), il **destructor** *Done* (corrispondente a *FreeMat*), la funzione *Elem* (corrispondente a *MatElem*) e il campo *Dati*. Gli altri campi, sia campi-dati che campi-procedura, vengono automaticamente *ereditati*: si potrà invertire una matrice *Mat* di tipo *TBigRealMat* con una istruzione *Mat.Inv* nonostante che nella unit della figura 2 non vi sia traccia di una procedura *Inv*; basta che ve ne sia una nella dichiarazione del tipo base *TRealMat*.

Un po' di gergo

Ogni linguaggio orientato all'oggetto ha il proprio gergo; in Turbo Pascal, ad esempio, i tipi definiti come nelle figure 1 e 2 vengono detti **object**, in C++ **class**, come in Simula e in SmallTalk. Volendo ora parlare in generale, dovre-

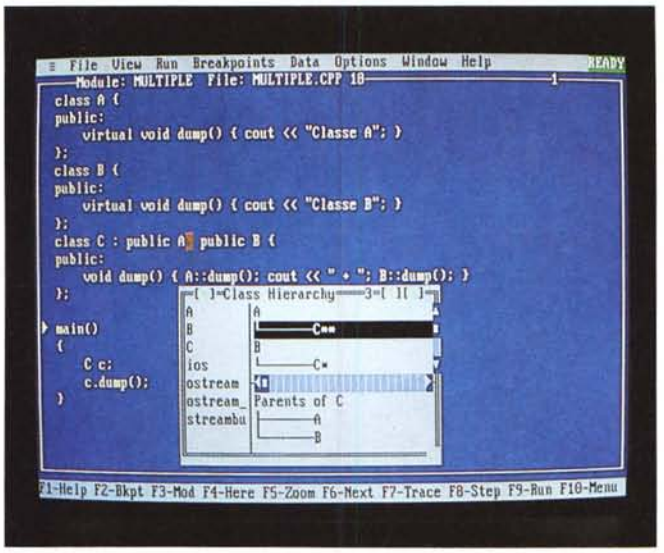

Figura 4 - Oltre alle gerarchie a forma di «albero» dello SmallTalk e del Turbo Pascal, il C++ dà la possibilità di definire gerarchie a forma di «grafo», in quanto è possibile derivare una classe da due o più altre classi (si parla di «ereditarietà multipla»). Qui vediamo tale situazione come visualizzata dal Turbo Debugger.

mo stabilire delle convenzioni. Chiameremo quindi *classe* il tipo «astratto» definito dall'utente come una struttura comprendente sia *variabili d'istanza* che *metodi*, cioè sia «campi-dati» che «campi-procedura» (in SmallTalk e in C++ ci sono anche *variabili* e *metodi di classe* ma di questo vi parlerà a suo tempo Corrado Giustozzi); chiameremo invece *oggetti* le «istanze» di una classe, cioè le variabili appartenenti ad un tipo-classe: l'oggetto *fileMenu* come istanza della classe *PullDownMenu*, gli oggetti *mat1, mat2* e *mat3* come istanze della classe *TRealMat*.

Si parla di classi di oggetti, in quanto l'insieme delle «cose» su cui il programma agisce (finestre, file, matrici, ecc.) non viene semplicemente «elencato» o «aggregato» più o meno alla rinfusa (come spesso capita di vedere nell'header file di un programma C o nella sezione **var** di un programma Pascal), ma «classificato»; dai tipi più generali vengono derivati tipi via via più specifici, in modo da costituire una *gerarchia di classi* simile, nell'aspetto, a un albero geneaologico o ad una classificazione di quelle care ai biologi. E si dice che un oggetto è istanza di una classe più o meno come si può dire che Bjarne Stroustrup è una istanza dell'homo sapiens. Classi e oggetti sostituiscono così tipi e variabili tradizionali; si dice che un oggetto è istanza di una classe come prima si diceva che una variabile appartiene a un tipo.

La motivazione del cambiamento va ricercata soprattutto nel fatto che un oggetto «incapsula» non solo dati ma anche funzioni e procedure; alle variabili d'istanza vengono affidati la rappresen-

Un po' di titoli, con due parole di commento

La OOP va di moda. Se questo può far piacere a chi sappia apprezzarne gli aspetti positivi, comporta però anche il fiorire di libri e articoli che, approfittando della situazione, cercano di farsi leggere più per l'argomento trattato che per la validità dei contenuti. Ritengo quindi utile proporvi un po' di titoli «sicuri».

Carlo GHEZZI e Mehdi JAZAYERI, *Concetti dei linguaggi di programmazione*, Franco Angeli, 1989 (propone efficacemente concetti rigorosi per comprendere e confrontare la definizione e l'implementazione dei diversi linguaggi; attribuisce la dovuta importanza ai tipi di dati astratti e più in generale alle caratteristiche che aiutano nella realizzazione e nella manutenzione di progetti ampi e complessi).

Ellis HOROWITZ, *Fundamentals of Programming Languages*, Computer Science Press, 1984 (paragonabile al precedente, comprende un intero capitolo dedicato allo SmallTalk e alla OOP).

BORLAND International, *Turbo Pascal 5.5 — Object-Oriented Programming Guide*, 1989 (uno dei manuali forniti con il compilatore, è stato definito dal *Dr. Dobb's Journal* come la più chiara introduzione alla OOP).

BORLAND International, *Turbo C++ 1.0 — Getting started*, 1990 (uno dei manuali del compilatore, comprende un capitolo che è in pratica la traduzione in C++ del precedente).

Bjarne STROUSTRUP, «What is Object-Oriented Programming?», *Proc. of the USENIX C++ Workshop*, 1987 (un bell'articolo, da cui ho tratto qualche ispirazione per questa «Introduzione»; è stato riprodotto nel materiale fornito dalla Borland ai partecipanti all'OOP World Tour 1990; chi avesse mancato l'edizione 1990 del Tour, potrebbe magari tenersi pronto per la prossima primavera...).

Brad J. COX, *Object Oriented Programming. An Evolutionary Approach*, Addison-Wesley, 1987 (a Cox dobbiamo non solo l'Objective-C, «cuore» del NeXT di Steve Jobs, ma anche forse la migliore esposizione della OOP, delle caratteristiche e dei benefici, nonché della importanza di disporre di una libreria di classi ben fatta).

Keith E. GORLEN, Sanford M. ORLOW e Perry S. PLEXICO, *Data Abstraction and Object-Oriented Programming in C++*, John Wiley & Sons, 1990 (illustrazione della famosa libreria di classi di Gorlen, modellata su quella di SmallTalk, ma anche ottima introduzione alla programmazione «seria» in C++).

Michele MARCHESI, *SmallTalk e la programmazione «Object-Oriented»*, Gruppo Editoriale Jackson, 1989 (è quasi impossibile trovare testi dedicati allo SmallTalk/V della Digitalk, o trovare testi che espongano chiaramente, con esempi di programmazione, il paradigma MVC, cioè l'approccio con cui si scrivono applicazioni «event-driven» in SmallTalk e da cui hanno tratto ispirazione il Mac, Windows, il Turbo Vision, ecc.; il testo di Marchesi è l'uno e l'altro, ed è pure molto ben fatto).

Bjarne STROUSTRUP, *The C++ Programming Language*, Addison-Wesley, 1986 (obbligatorio per «fare conoscenza» con il padre del C++ e con la prima versione del linguaggio, anche se è un po' datato e soprattutto non riesce ad essere il K&R del C++: non è sempre di facile lettura, e lascia molti punti interrogativi; ma va letto!).

Stanley B. LIPPMAN, *C++ Primer*, Addison-Wesley, 1989 (viene considerato, a ragione, come il testo più valido attualmente disponibile per avvicinarsi al C++).

Stephen C. DEWHURST e Kathy T. STARK, *Programming in C++*, Prentice-Hall, 1989 (fa coppia col precedente: è meno completo, ma approfondisce meglio alcuni aspetti).

Margaret A. ELLIS e Bjarne STROUSTRUP, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990 (scelto come base di lavoro per la standardizzazione del C++ in corso ad opera dell'ANSI, non è un testo per principianti, ma espone in modo esemplare le diverse caratteristiche del linguaggio, illustrandone le motivazioni).

Peter COAD e Edward YOURDON, *Object-Oriented Analysis*, Yourdon Press, 1990 (dopo aver proposto con Larry Constantine lo «Structured Design», Yourdon sposa con convinzione le nuove metodologie; illuminante, il capitolo «Selecting CASE for OOA», in cui si contrappone la semplicità ed efficacia della OOA alla pesantezza di metodi più tradizionali).

Bertrand MEYER, *Object-Oriented Software Construction*, Prentice-Hall, 1988 (benché faccia riferimento al linguaggio Eiffel, messo a punto dall'autore, si propone agli utenti di qualsiasi linguaggio orientato all'oggetto come il testo che forse meglio di ogni altro illustra contemporaneamente e con pari efficacia gli aspetti e i vantaggi di OOD e OOP).

```
1. [Livello piu' alto di astrazione]

   Risolvi un caso completo del problema


2. [Primo raffinamento]

   Leggi i valori in input
   Calcola i risultati
   Produci i risultati
```

Figura 5 - Le prime due fasi dello sviluppo di un programma da eseguire in batch con il metodo «top-down», mediante approssimazioni successive («stepwise refinement»).

Figura 6 - Le prime due fasi dello sviluppo della versione interattiva dello stesso programma della figura 5, con lo stesso metodo «top-down», mediante approssimazioni successive (le figure 5 e 6 sono adattate da Bertrand Meyer, Object-Oriented Software Construction, Prentice-Hall, 1988).

```
1. [Livello piu' alto di astrazione]

   Tratta una transazione


2. [Primo raffinamento]

   if l'utente vuole immettere nuovi dati then
       immetti i dati
       registrali in memoria
   else if vengono chiesti dati gia' immessi then
       cerca i dati richiesti
       visualizzali
   else if viene chiesto un risultato then
       if l'informazione e' gia' disponibile then
           cerca il risultato richiesto
           visualizzalo
       else
           chiedi conferma della richiesta
           if confermata then
               cerca i dati necessari
               calcola il risultato richiesto
               visualizzalo
           end
       end
   else ... (ecc.)
```

tazione in memoria dell'oggetto e il compito di tenere costantemente nota del suo «stato», ai metodi il compito implementare il «comportamento» dell'oggetto. È proprio questo il punto principale: così come un tipo astratto viene definito mediante un set di operazioni invece che esplicitando la sua implementazione, un oggetto non si presenta più come un insieme di locazioni di memoria da cui possiamo leggere o in cui possiamo scrivere valori; si presenta quasi come un qualcosa di esterno a noi, dotato di un proprio *modus operandi*, capace di rispondere a modo suo — con un suo *metodo* — ai nostri *messaggi*. E già, perché in OOP non si parla di chiamate di procedura, ma di «invio di messaggi».

Estensibilità e polimorfismo

Per capire i motivi di un gergo così «antropomorfico», occorre considerare che la ereditarietà non viene da sola. Data una classe base *Window*, posso derivare da questa le classi *Menu* ed *EditWindow*, e poi da *Menu* le classi *PullDownMenu* e *PopUpMenu*. Ogni classe derivata eredita tutte le variabili d'istanza della sua classe base, come anche tutti i metodi; può però ridefinire i metodi della classe base, onde poter assumere comportamenti diversi da quelli ereditabili e più coerenti con la propria natura: se invio il messaggio «Chiuditi!» ad un menu, lo vedrò semplicemente scomparire; se lo invio ad una finestra di editing, mi verrà probabilmente prima chiesto se intendo salvare il risultato del mio lavoro.

Immaginiamo ora che il mio programma mantenga una lista di tutti gli oggetti istanza delle classi derivate da *Window* e che, al termine dell'esecuzione, quella lista venga percorsa per chiudere tutto ciò che risulta aperto.

Ovviamente la lista conterrà nodi di tipo diverso; potrei quindi usare una istruzione **case** (Pascal) o **switch** (C) per adottare i provvedimenti necessari a chiudere nel modo appropriato i diversi tipi di finestre. Nulla di strano. Ci sarebbero tuttavia un paio di problemi. In primo luogo, dovrei dotare ogni classe di una variabile d'istanza che mi dicesse, per ogni oggetto in cui mi imbatto, a quale classe appartiene. Soprattutto, però, la chiusura di finestre al termine di un programma è solo una delle situazioni del genere in cui potrei capitare; ogni volta che dovessi fare i conti con una stessa azione da eseguire su oggetti appartenenti a classi diverse, dovrei «fissare» in una (e quindi in tante) istruzioni **case** o **switch** la gamma di oggetti usati da un programma. Ma che succederebbe se in un secondo tempo volessi «estendere» questa gamma, ad esempio derivando da *Window* anche *DialogBox*? Dovrei modificare tutte quelle istruzioni e ricompilare da capo. E guai a dimenticarne una!

I linguaggi orientati all'oggetto offrono una soluzione molto più semplice e molto meno soggetto ad errori. Abbiamo detto che in una classe derivata si possono ridefinire i metodi ereditati dalla classe base; se i metodi ridefiniti vengono dichiarati *virtual* e opero mediante puntatori alla classe base, posso assegnare a tali puntatori l'indirizzo di qualsiasi classe derivata ed ottenere che, chiamando tramite quei puntatori un metodo virtuale, venga eseguito per ogni oggetto il codice della specifica versione del metodo che è stata ridefinita per quell'oggetto. In altri termini, se *wPtr* è un puntatore a *Window*, chiamando *wPtr^.Close* (o *wPtr->close()* con la notazione del C++) verrà eseguito *PullDownMenu.Close* o *EditWindow .Close* secondo la classe dell'oggetto il cui indirizzo ho assegnato a *wPtr*, sia esso una finestra di editing o un menu.

«Estensibilità» vuol dire che, quando aggiungerò al mio programma delle dialog box, il codice che contiene l'istruzione *wPtr^.Close* non dovrà essere né modificato né ricompilato, in quanto il metodo da eseguire dipende dalla classe di cui è istanza l'oggetto il cui indirizzo viene di volta in volta assegnato a *wPtr*, e quale sia questa classe viene automaticamente riconosciuto dal programma durante l'esecuzione (questo viene detto *late binding*), e non già durante la compilazione (che quindi non c'è bisogno di ripetere).

Ecco perché si parla di messaggi invece che di chiamate di procedura. Non è necessario scrivere procedure cui passare come parametro l'oggetto su cui si deve agire (come invece sono stato costretto a fare nelle routine in rubrica). Tramite un puntatore, viene inviato un *messaggio* a qualsiasi cosa possa risiedere nell'indirizzo cui quello punta; sarà poi compito dell'oggetto «puntato» rispondere secondo il proprio *metodo*.

L'invio del messaggio tramite un puntatore prescinde dalla natura dell'oggetto destinatario; può quindi essere effettuato nei confronti di oggetti di varie classi, anche di classi non ancora neppure pensate; questo si intende per *polimorfismo*.

OOA, OOD, OOP

Potrei a questo punto descrivere i meccanismi attraverso i quali avvengono tali magie, ma, considerando anche che si tratterebbe di entrare nei dettagli dei diversi linguaggi, vi rimando alle rubriche C++ e Turbo Pascal. Come vi ha già detto Corrado il mese scorso, da gennaio entreremo nel vivo della programmazione orientata all'oggetto, con esempi «sul campo» nei due linguaggi più diffusi. Ora vorrei soprattutto evitare che pensaste che la OOP serve solo a riusare o a estendere cose già fatte, come se fosse di poca utilità quando si deve partire da zero nella realizzazione di un progetto interamente nuovo. Non è così, in quanto la disponibilità di linguaggi orientati all'oggetto consente di applicare tecniche più semplici e più efficaci anche al lavoro che va condotto prima di inziare a scrivere un programma. Si parla a questo riguardo di *Object Oriented Analysis* (OOA) e di *Object Oriented Design* (OOD).

Sarebbe un discorso lungo, che devo necessariamente rinviare ad altra occasione. Non posso fare a meno, tuttavia, di riproporvi pari pari un efficace esempio di Bertrand Meyer, il padre del linguaggio Eiffel, tratto dal suo *Object-Oriented Software Construction* (Prentice-Hall, 1988). Considerate, lui dice, un programma che abbia due versioni, una «batch», in cui tutto avviene in una botta sola sempre nello stesso modo, e una interattiva, in cui ogni sessione sia una sequenza di diverse transazioni. Potete pensare a un compilatore disponibile sia in un ambiente integrato che in versione separata, o a quei programmi scientifici che consentono sia lunghi calcoli notturni che sessioni del tipo «proviamo a vedere subito cosa succede se cambio qualche dato».

Le figure 5 e 6 illustrano i primi due passi di una tradizionale scomposizione funzionale secondo il metodo *top-down*, prima per la versione batch poi per quella interattiva. Vedete subito che ne vengono fuori cose del tutto diverse, tanto diverse che si perde completamente di vista il fatto che quelli che sembrano due programmi distinti sono in realtà solo due versioni di uno stesso programma. Questo accade perché si è posto l'accento su «ciò che il programma deve fare» invece che su «ciò su cui il programma deve operare». Badando in primo luogo agli oggetti invece che alle funzioni, si possono mettere a punto gli insiemi di operazioni applicabili ai diversi oggetti, rimandando solo ad una ultima fase le decisioni circa l'ordine in cui tali operazioni devono essere effettuate.

Come disse una volta P. J. Plauger, il metodo *top-down* va bene solo quando si sa già come fare quello che si deve fare. Per arrivare a capire cosa e come fare, partire dagli oggetti è il modo migliore. OOA, OOD e OOP sono quindi gli strumenti più efficaci anche per affrontare progetti interamente nuovi.

MC