Programmazione Object Oriented in Turbo Pascal 5.5 e Quick Pascal 1.0

di Sergio Polini

Quasi contemporaneamente la Borland e la Microsoft hanno rilasciato due prodotti tanto simili quanto innovativi e interessanti. Il Quick Pascal 1.0, prendendo finalmente atto dello standard scelto dal mercato, è un compilatore interattivo che dichiara ampia compatibilità con le versioni 4.0 e 5.0 del compilatore Borland; a ciò aggiunge le funzionalità di OOP (Object Oriented Programming) dell'Object Pascal sviluppato dalla Apple per il Mac. Il Turbo Pascal 5.5 offre anch'esso l'OOP, ma si ispira sia all'Object Pascal che al C++ della ATT, ricercando così una maggiore flessibilità e una migliore efficienza. A settembre metteremo a confronto i due prodotti; intanto prepariamo il terreno con una breve chiacchierata sull'OOP, argomento nuovo per i nostri lettori (... e non solo). Cominceremo con una breve storia, intenzionalmente semplificata e parziale, dei linguaggi di programmazione.

Si parte naturalmente dal progetto del primo compilatore FORTRAN nel lontanto 1954: un linguaggio espressamente dedicato ad elaborazioni di tipo matematico, usato da matematici. La «leggibilità» dei sorgenti e la documentazione dei programmi non erano un grosso problema (un matematico sa bene come maneggiare un sistema di equazioni differenziali), ma ci si preoccupava soprattutto della efficienza del codice prodotto dal compilatore. Si trattava in fondo dei primi tentativi di imporre ad un elaboratore un linguaggio più simile a quello di noi umani che ad una poco espressiva sequenza di bit, si temeva che il codice prodotto dalla traduzione in linguaggio macchina avrebbe pesato più del lecito su un hardware piuttosto costoso. Ecco quindi gli **if** aritmetici, il controllo del flusso affidato ai GOTO, se non addirittura ai GOTO calcolati, ecc., ovvia influenza della programmazione in assembler su quella ad alto livello.

Eppure già con il FORTRAN si introdussero forme di *astrazione sul control-*

lo: un programma non era necessariamente un confuso groviglio di istruzioni, ma poteva ben essere organizzato in sottoprogrammi compilati separatamente; una volta realizzata una FUNCTION o una SUBROUTINE, si poteva ricordare solo *cosa* faceva, non anche *come* lo faceva, astraendo così dai dettagli.

Intanto gli elaboratori diventavano via via sempre più potenti e l'hardware meno costoso, mentre aumentavano invece i costi di progettazione, sviluppo e manutenzione di programmi sempre più complessi, usati nelle applicazioni più disparate. Il COBOL, voluto dal Ministero della Difesa americano (il DOD), aveva proposto dati strutturati in record, un'apposita sezione dedicata alla dichiarazione dei dati, un'attenzione quasi ossessiva alla leggibilità dei sorgenti. Ma non era abbastanza. Si ripresero quindi alcune idee dell'ALGOL 60, quali le nozioni di tipo dei dati e di *type checking*, nonché meccanismi di astrazione sul controllo quali gli ormai irrinunciabili **if..then..else e for** (che consentono di «nascondere» un bel po' di GOTO) e il blocco racchiuso dalla coppia begin-end, con le sue variabili locali.

Astrazione sul controllo

Passando attraverso ALGOL 68 e ALGOL-W (quella «W» sta per Wirth!) si arrivò così al Pascal e alla programmazione strutturata: ogni programma come sequenza di azioni dei tre tipi fondamentali (sequenza, scelta e ripetizione), ogni azione realizzata mediante blocchi aventi un unico punto d'ingresso e un unico punto d'uscita. Ci sarebbe molto da dire sulla programmazione strutturata, ma in fondo non è certo argomento nuovo per chi legge queste pagine. Ci soffermeremo quindi soprattutto su alcuni aspetti dell'astrazione sul controllo consentita dalle procedure del Pascal.

Un programma Pascal è un grande blocco contenente dichiarazioni di costanti, tipi, variabili, funzioni e procedure; funzioni e procedure hanno una struttura praticamente identica a quella del programma: possono tra l'altro contenere a loro volta funzioni e procedure «figlie». In ogni procedura (lo stesso vale per le funzioni) sono «incapsulati» dati e codice, che vengono così «nascosti» al resto del programma: alle variabili locali e alle istruzioni di una procedura possono accedere solo le sue istruzioni e quelle delle procedure figlie, le quali possono peraltro «sovrapporre» proprie variabili a quelle con lo stesso nome dichiarate nelle procedure madri (comprese le variabili globali del programma). Si realizza così un'ampia indipendenza tra blocchi diversi del programma, al punto che, entro certi limiti, è possibile sostituire interamente una procedura con un'altra che faccia la stessa cosa in modo diverso (ad esempio: un Quicksort invece che un Bubblesort) senza per questo dover modificare il resto del programma. Ciò agevola sia il debugging (se l'output non è ordinato devo solo controllare la procedura di sort) che eventuali modifiche.

La flessibilità del codice è aumentata dalla possibilità di scrivere blocchi di istruzioni «polifunzionali» , che si adattino cioè a particolari valori di alcuni dati: basti pensare a come una istruzione **case** consente di regolare il comportamento di una procedura secondo il valore del suo selettore.

Astrazione sui dati

Programma, funzioni e procedure comunicano però tra loro mediante variabili globali o parametri — che devono appartenere a tipi ben determinati — e possono essere compilati solo tutti insieme (prescindiamo naturalmente dai parametri variabile senza tipo e dalle unit del Turbo Pascal). Ne segue che la più piccola modifica, per quanto localizzata, costringe a ricompilare tutto il programma, mentre l'aggiunta o la modifica di un tipo di dati porta a dover riscrivere buona parte del tutto. A tali inconvenienti hanno tentato di ovviare sia Wirth con il Modula-2 che il DOD con Ada.

I moduli dell'uno e le unit dell'altro consentono di scomporre un programma in unità ampiamente indipendenti, con propri dati e proprio codice. Consentono soprattutto di seguire metodologie di sviluppo quali il «mascheramento delle informazioni» (information hiding): ogni unità dl programma «nasconde» i dettagli della implementazione delle sue strutture di dati, fornendo solo funzioni di accesso a queste; il resto del programma non ha modo di accedervi direttamente, neppure attraverso variabili globali. Ne risulta un codice più leggibile, facilmente sviluppabile da un team di programmatori, soprattutto più facilmente modificabile e manutenibile.

Vi sono tuttavia dei limiti. I «tipi opachi» del Modula-2 possono essere realizzati solo mediante puntatori, ma soprattutto le strutture cui questi puntano impongono comunque una dipendenza dal tipo: si può lasciare «opaca» l'imple-

```
DEFINITION MODULE GestioneTabelle;
   TYPE Tabella;  (* tipo opaco *)
   PROCEDURE InitTabella(VAR t: Tabella);
   PROCEDURE Inserisci(t: Tabella; VAR x: ARRAY OF CHAR; n: INTEGER);
   PROCEDURE WriteTabella(t: Tabella)
END GestioneTabelle.
```

Il DEFINITION MODULE GestioneTabelle di un programma di cross-reference (adattato dal «Programming in Modula-2» di Wirth). Con la procedura Inserisci si aggiunge alla tabella una stringa insieme al numero della riga in cui si trovava, con WriteTabella si visualizza la tabella. Il tipo di questa è «opaco» , cioè nascosto in un IMPLEMENTATION MODULE, nel quale si possono usare array, liste, alberi, ecc., senza modificare il DEFINITION MODULE e quindi senza dover intervenire su altre parti del programma.

mentazione di un dato strutturato (array, lista, albero, ecc.), ma va comunque determinato il tipo dei dati elementari; si può nascondere l'implementazione di una tabella, ma questa dovrà comunque essere o una tabella di interi, o una tabella di stringhe, o una tabella di record. Non si può definire una tabella di «qualsiasi cosa»: se ho bisogno di aggiungere un nuovo tipo di dati alla mia tabella (inserire numeri reali in una tabella che fino a ieri poteva limitarsi a contenere interi) devo comunque riscrivere tutte le parti del programma che con questa tabella hanno a che fare, anche quelle che si limitano a «usare» il modulo in cui avevo sperato di poterla nascondere.

Ada offre di più: è effettivamente possibile definire una tabella di «qualsiasi cosa». Nella preparazione di un package (o anche di una procedura) **generic**, si può definire **private** un certo tipo, lasciandolo così indeterminato fino al momento della compilazione; il codice che ne risulta non può essere però usato direttamente, in quanto si tratta di qualcosa di simile ad una macro: il package o la procedura devono essere «istanziati» con un **new**, in modo da originare una compilazione in cui al tipo privato (potremmo dire «rimandato») viene sostituito un tipo ben determinato. Si tratta cioè da un lato di strumenti molto potenti, dall'altro solo di rimandare dalla scrittura del codice alla sua compilazione le scelte relative ai tipi; ciò comporta ad esempio che, a meno delle solite acrobazie (quelle con cui si fa comunque tutto con tutto), non è possibile approntare librerie di funzioni generiche che possano essere utilizzate così come sono, solo linkandole al resto del programma. Un serio limite alla «riusabilità» del codice: si può riusare il sorgente, non sempre il codice oggetto.

```
-- package specification
generic
  type ELEMENTO is private;
package GESTIONE_TABELLE is
  type TABELLA is limited private;
  procedure INSERISCI(T: in out TABELLA; E: in ELEMENTO);
  procedure WRITE_TABELLA(T: in TABELLA);
private
  type TABPTR;
  type TABELLA is
      record
        DATO: ELEMENTO;
        NEXT: TABPTR;
      end record;
  type TABPTR is access TABELLA;
end GESTIONE_TABELLE;

-- package body
package body GESTIONE_TABELLE is
-- ...
-- codice per le funzioni e procedure esportate
-- (INSERISCI e WRITE_TABELLA) ed eventuali
-- dati, funzioni e procedure "ausiliarie".
-- ...
end GESTIONE_TABELLE;

-- creazione di istanze del package generico
package INTTAB is new GESTIONE_TABELLE(INTEGER);
package STRTAB is new GESTIONE_TABELLE(STRING);
```

I package «generic» di Ada offrono una limitata indipendenza dal tipo: il package GESTIONE TABELLE implementa tabelle come liste aventi un campo DATO di tipo qualsiasi; può però essere usato solo dopo averne creato istanze in cui venga definito il tipo del campo DATO. L'indipendenza dal tipo finisce quindi con la compilazione; durante l'esecuzione l'istanza INTTAB non potrà essere altro che una tabella di INTEGER.

Altri stili di programmazione

Per trovare una soluzione bisogna complicare un po' la storia dei linguaggi, in quanto quello che abbiamo brevemente descritto è solo uno dei filoni principali, quello della programmazione *imperativa* o *orientata all'istruzione*. Si tratta dello stile di programmazione ancora dominante in quanto basato sull'architettura di Von Newman, quella da cui derivano quasi tutti i moderni calcolatori elettronici. Una macchina di Von Newman può essere schematizzata mediante quattro blocchi: Ingresso, Uscita, una singola CPU, una Memoria; il suo funzionamento tipico prevede la lettura di variabili (nomi simbolici per locazioni di memoria) e la loro modifica mediante un operatore di assegnazione. Nella rubrica Turbo Pascal del mese scorso abbiamo accennato alla possibilità di programmare senza assegnazioni, dando esempi di calcolo della somma e del prodotto di due numeri; abbiamo cioè dato un piccolissimo esempio di un mondo completamente diverso, quella della programmazione *funzionale* o *applicativa*, che vede nell'APL e soprattutto nel LISP (ambedue nati nel 1956) i suoi campioni. È più recente la proposta di una programmazione *logica* o *dichiarativa* (basti pensare al PROLOG), ma fin dagli anni '60 c'era chi in Norvegia aveva gettato le premesse per un completo capovolgimento della programmazione orientata alle istruzioni: non azioni attive eseguite su oggetti passivi, ma oggetti attivi che rispondono a messaggi; in breve, programmazione *orientata all'oggetto*, che la Borland definisce a buon diritto come lo stile di programmazione degli anni '90. Non a caso vi sono già diverse versioni *object oriented* di linguaggi imperativi, logici e funzionali.

Classi, eredità e polimorfismo

Il Simula 67 dei norvegesi Dahl e Nygaard, derivato dall'ALGOL 60 e dedicato alla descrizione e simulazione di sistemi, introdusse il concetto di *classe*: accanto ai blocchi dell'ALGOL 60, annidati l'uno dentro l'altro, proponeva classi di oggetti tra di loro indipendenti, che venivano definite incapsulando le istruzioni insieme alle dichiarazioni dei dati, e quindi fuori dalla struttura gerarchica del programma (da ciò derivarono poi sia i moduli di Modula-2 che le unit di Ada); offriva soprattutto la possibilità di combinare precedenti classi in una nuova: dopo aver definito una classe «tabella generica», si poteva concatenare a questa una sottoclasse «tabella di record», per definire la quale era sufficiente descrivere quanto si voleva specifico di una tabella di record; il resto veniva «ereditato» dalla classe precedente.

Lo Smalltalk, sviluppato negli anni '70

```
Input                              Output
-----                              ------

lista <- List new
lista addFirst: 2/3
lista addLast: 10
lista addFirst: 'abc'
lista addLast: $Z
lista print                        List ( 'abc' 0.6666667 10 $Z )
lista first                        abc
lista last                         $Z

vettore <- Array new: 4
vettore at: 2 put: 2/3
vettore at: 3 put: 10
vettore at: 1 put: 'abc'
vettore at: 4 put: $Z
vettore print                      #( 'abc' 0.6666667 10 $Z )
vettore at: 2                      0.6666667
```

Una breve sessione con Little Smalltalk (disponibile su MC-Link come ST.ARC), implementazione di pubblico dominio di un subset dello Smalltalk-80. Sia Array che List sono derivazioni della classe Collection. List è una Collection con un numero indefinito di elementi, mentre Array ha un numero di elementi pari all'argomento del messaggio «new:». In entrambi i casi gli elementi possono essere di qualsiasi tipo.

presso il PARC (Palo Alto Research Center) della Xerox, ha fatto del concetto di classe il cardine di tutto un nuovo modo di programmare, interfaccia utente compresa. In Smalltalk una variabile non è il nome simbolico di una locazione di memoria, ma designa un oggetto «attivo» allocato dinamicamente: in luogo di funzioni attive (istruzioni della CPU) che operano su oggetti passivi (celle di memoria), si hanno oggetti che, in quanto modelli del mondo reale, sono attivi e in grado di rispondere ai messaggi spediti da altri oggetti. Tali messaggi non sono legati a tipi di dati: in una tradizionale chiamata di procedura vanno passati parametri ognuno appartenente ad un ben determinato tipo, così come ben determinato deve essere il tipo del valore ritornato da una funzione. In Smalltalk invece i controlli di tipo vengono eseguiti durante l'esecuzione, nel senso che l'unico controllo concerne la possibilità dell'oggetto destinatario di un messaggio di rispondere a questo secondo un suo proprio *metodo*. Se un oggetto ha — o eredita da classi superiori rispetto a quella di cui è istanza — un suo metodo per eseguire una certa azione X, posso semplicemente dirgli «fai X» , e lui lo farà senza che io debba preoccuparmi del suo tipo. I messaggi sono cioè «generici», o meglio *polimorfici*.

Si tratta di concetti talmente nuovi per molti di noi, che può essere utile proporre un esempio facile facile, anche se un po' forzato (non molto in verità: sarà solo un po' arbitrario parlare di polimorfismo invece che di *overloading*, ma la differenza tra i due concetti non è così rilevante finché ci si limita a linguaggi imperativi). In Pascal esiste una implicita classe **numero**, che è caratterizzata dall'avere i suoi elementi appartenenti ad un certo range, ordinati, ecc. e dalla possibilità di eseguire su tali elementi le quattro operazioni. Da tale classe derivano le sottoclassi **integer** e **real** che *ereditano* quelle caratteristiche, aggiungendo ognuna qualcosa di suo: sono diverse ad esempio la divisione di due integer e quella di due real, l'operatore **mod** è definito solo per gli integer, ecc. Se voglio il quadrato di un numero gli mando il messaggio *Sqr()*, che accetta indistintamente argomenti di tipo integer o real: il codice che produce il quadrato è ben diverso nei due casi, il tipo del risultato cambia secondo il tipo dell'argomento, ma il messaggio è sempre lo stesso; è un messaggio *polimorfico*, in grado cioè di adattarsi al tipo dell'oggetto cui è inviato, di tener conto che oggetti integer e oggetti real hanno diversi *metodi* per elevarsi al quadrato. Un carattere non ha un metodo analogo e non può quindi «rispondere» al messaggio. Tutte cose di cui magari non siamo consapevoli, ma che ci sono ben familiari.

```
unit Liste;

interface

type
  NodoPtr = ^Nodo;
  Nodo = object
    Next: NodoPtr;
    procedure Init;
  end;
  Lista = object
    Primo: NodoPtr;
    procedure Init;
    procedure Aggiungi(N: NodoPtr);
  end;
```

L'interface di una unit in cui vengono dichiarati gli oggetti Nodo e Lista, limitatamente alle caratteristiche più generali (ogni nodo ha un puntatore al nodo successivo, ad ogni lista bisogna poter aggiungere nodi, ecc.). In questa e nelle seguenti figure si seguono le convenzioni del Turbo Pascal 5.5, in quanto il Quick Pascal ci è arrivato poco prima dell'ultimo giorno utile per andare in macchina.

E tuttavia classi, eredità e polimorfismo, pur se implicitamente presenti in un compilatore Pascal, non lo rendono un linguaggio orientato all'oggetto perché non è possibile definire nuovi oggetti che ereditino le caratteristiche di altri, non è possibile definire tipi di dati che si comportino «attivamente» come i tipi numerici predefiniti. È facile definire un tipo complesso come un record di un parte intera e una parte immaginaria, ma non è possibile sommare due numeri complessi con un «+» o verificarne l'eguaglianza con un «=». È possibile definire tipi di dati anche molto ricchi e potenti, ma non è possibile scrivere programmi che consentano di trattare tali dati con la stessa flessibilità che il compilatore offre per i tipi predefiniti.

La programmazione orientata all'oggetto richiede che si ripetano anche per l'astrazione sui dati i progressi raggiunti nell'astrazione sul controllo: ad ambedue è comune *l'incapsulamento* delle dichiarazioni di dati e di istruzioni nella dichiarazione di un oggetto o di una procedura, o nella **interface** di una **unit**; occorre però anche che un oggetto possa avere oggetti «figli» così come una procedura può avere altre procedure al suo interno; occorre che gli oggetti figli possano ereditare dal padre, così come una procedura nidificata eredita le variabili di quella in cui è inserita; occorre che gli oggetti figli possano sostituire ai metodi del padre propri metodi, così come una procedura nidificata può sostituire alle variabili di quella esterna proprie variabili con lo stesso nome; occorre che un stesso messaggio possa dar luogo all'attivazione di diversi metodi,

```
unit LsIntStr;
(* LiSta di INTeger e STRing *)
interface

uses Liste;

type

  NodoGenPtr = ^NodoGen;
  NodoGen = object(Nodo)
    procedure Print; virtual;
  end;
  NodoIntPtr = ^NodoInt;
  NodoInt = object(NodoGen)
    Valore: integer;
    constructor Init(V: integer);
    procedure Print; virtual;
  end;
  NodoStrPtr = ^NodoStr;
  NodoStr = object(NodoGen)
    Valore: ^string;
    constructor Init(V: string);
    destructor Done; virtual;
    procedure Print; virtual;
  end;
  ListaGen = object(Lista)
    procedure Print;
  end;

implementation

(* dichiarazione dei metodi per *)
(* NodoGen, NodoInt e NodoStr   *)

procedure ListaGen.Print;
var
  P: NodoGenPtr;
begin
  P := NodoGenPtr(Primo);
  while P <> nil do begin
    P^.Print;
    P := NodoGenPtr(P^.Next);
  end;
end;
```

La unit LISTE può essere usata per derivarne nodi di tipo particolare (integer e string, nel nostro caso) e per aggiungere il metodo Print alla lista «astratta». In ListaGen.Print si passa attraverso tutti i nodi, mandando ad ognuno il messaggio «Print»; ogni nodo risponderà secondo il suo «metodo», senza che il programmatore debba preoccuparsi minimamente del tipo dei vari nodi.

così come una istruzione case può causare l'esecuzione di codice diverso secondo il valore del suo selettore.

In conclusione, si tratta di meccanismi e di un modo di programmare ben diversi da quelli abituali, ma che in fondo possiamo fare nostri semplicemente ripetendo su un altro piano quel processo di apprendimento che ci ha portato a saper scrivere programmi strutturati in Pascal.

```
program ListDemo;

uses Liste, LsIntStr;

type
  Complex = record
    R, I: real
  end;
  NodoComPtr = ^NodoCom;
  NodoCom = object(NodoGen)
    Valore: Complex;
    constructor Init(V: Complex);
    procedure Print; virtual
  end;
var
  L: ListaGen;
```

Un programma che usi sia LISTE che LSINTSTR può dichiarare ulteriori tipi particolari di nodi, nonché variabili dei tipi dichiarati nelle unit usate. Una istruzione «L.Print» chiamerà la procedura ListaGen.Print dichiarata nella unit LSINTSTR.PAS, la quale chiamerà a sua volta le procedure Print dichiarate per i vari tipi di nodi, compresa quella per i nodi di tipo Complex (nonostante tale tipo sia dichiarato successivamente), senza bisogno né di modificare né di ricompilare la unit LSINTSTR.

Smalltalk, Object-Pascal, C++

In Smalltalk tutto è oggetto, anche le classi: la programmazione imperativa e orientata all'istruzione è semplicemente impossibile. Ne segue una notevole potenza e flessibilità (nessun problema con tabelle i cui elementi non siano tutti dello stesso tipo), ma anche una sintassi inconsueta. È inoltre un linguaggio interpretato, in cui gli oggetti vengono sempre allocati dinamicamente. In conclusione, è adatto più alla ricerca che alla realizzazione di applicazioni «reali». Accanto quindi allo Smalltalk e ad altri linguaggi specializzati, sono state sviluppate molte estensioni orientate all'oggetto di linguaggi tradizionali: Neon (basato sul Forth), Objective-C, Objective-Logo, Flavors (basato sul LISP), ecc. Tra questi ci interessano particolarmente l'Object Pascal e il C++.

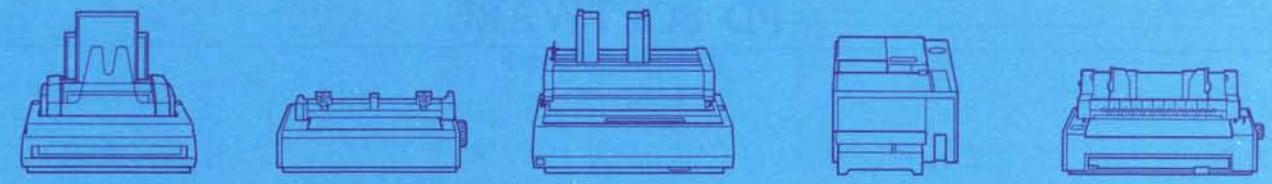
Dopo l'esperienza del Clascal, scomparso insieme al computer Lisa, l'Apple chiese aiuto a Wirth per una nuova versione orientata all'oggetto del Pascal. Ne derivò l'Object Pascal, in cui gli oggetti vengono definiti come record con campi di dati e campi di metodi, i quali sono normali procedure che possono essere chiamate premettendo il nome dell'oggetto e un punto al loro nome, proprio come avviene per i campi di un record.

Più o meno nello stesso periodo, Bjarne Stroustrup seguiva una strada analoga per pervenire ad una estensione object oriented del C: classi simili alle **struct**, con dati e funzioni (nel C++, derivato dal Simula più che dallo Smalltalk, si parla di funzioni **friend** e **member** piuttosto che di metodi). Aggiunse tuttavia delle possibilità ulteriori: con alcune volle rendere potente e flessibile il suo linguaggio, con altre volle evitare una perdita d'efficienza rispetto al C normale. Tra le prime troviamo ad esempio il «sovraccarico» (*overloading*) di funzioni e operatori: si possono definire in una stessa classe più metodi dallo stesso nome ma con argomenti diversi, come anche usare gli operatori aritmetici e relazionali su operandi definiti dall'utente. Tra le seconde troviamo una keyword virtual, che consente di rendere polimorfiche (capaci cioè di operare su oggetti di vario tipo, essendo la determinazione del tipo effettivo rimandata al momento dell'esecuzione) solo le funzioni che si vuole siano effettivamente tali, mentre quando ciò non serve si può ottenere codice più efficiente lasciando una funzione non virtuale. Le funzioni virtuali possono poi essere implementate come funzioni *inline*, in modo da non rinunciare né al polimorfismo né all'efficienza.

In ambedue i casi si tratta di linguaggi compilati, in cui *eredità* e *polimorfismo* assicurano una notevole *estensibilità* e *riusabilità* del codice: si tratta di abituarsi a individuare una gerarchia di astrazioni successive, dichiarando prima gli oggetti con le caratteristiche più generali (tutti i nodi di una lista hanno un puntatore al nodo successivo, tutte le figure geometriche hanno una posizione sullo schermo, ecc.), per derivarne poi altri oggetti via via più specifici, fino a quelli su cui vogliamo concretamente lavorare. Proprio come dividiamo un programma in unit, funzioni e procedure, articolando sempre più le azioni che vogliamo eseguire. Il beneficio sarà la possibilità di riusare le unit con le dichiarazioni più generali, semplicemente aggiungendo quanto vogliamo in più e modificando quanto vogliamo di diverso: ma per modificare un metodo ereditato basta dare lo stesso nome ad un nuovo metodo, senza bisogno di intervenire su quanto già fatto. Le unit contenenti gli oggetti «padri» possono essere quindi usate senza che si debba né riscriverle né ricompilarle.

Vedremo il mese prossimo quanto poco occorra con i nuovi compilatori per ottenere tanto.

MC



Economia e Professionalità
Classe "low-end" con prestazioni professionali a prezzi contenuti. Versione ad aghi, a colori, con velocità fino a 200 cps.
DM100 S - DM105 S

Multifunzionalità
Soluzioni professionali per tutte le esigenze dell'ufficio. Matrice di stampa a 9 o 18 aghi, velocità da 240 a 400 cps, gestione speciale della carta.
PR24 - PR24 L - DM410

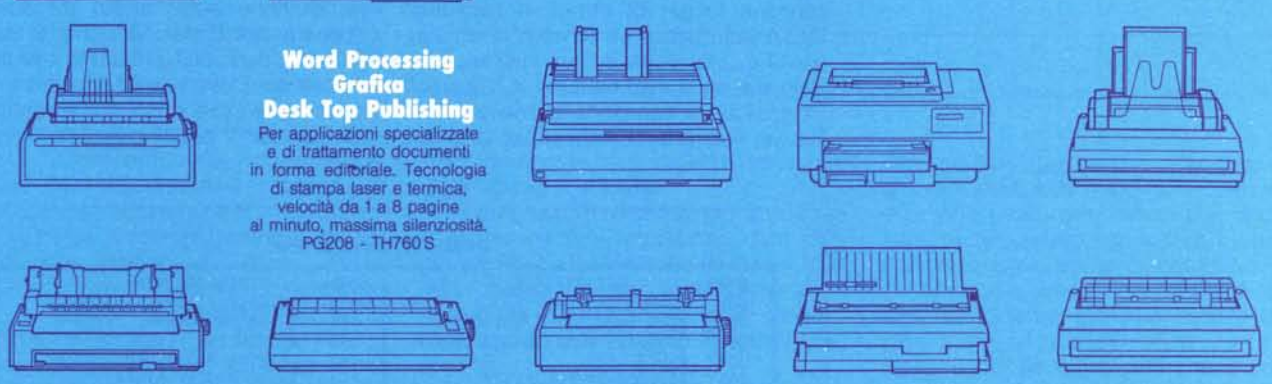
Versatilità
Classe "general purpose" che risponde efficacemente a tutte le esigenze dell'utente professionale. Versioni ad aghi, con velocità di stampa di 200 cps.
DM282 - DM292

Alta qualità
Classe indirizzata ad applicazioni di office automation che richiedono alta velocità e sofisticato trattamento carta. Matrice di stampa a 24 aghi, velocità da 200 a 330 cps.
DM600 S - DM250 L

STAMPANTI OLIVETTI.

SETTE FAMIGLIE, DECINE DI MODELLI GARANTITI DAL MAGGIOR PRODUTTORE EUROPEO.

Word Processing Grafica Desk Top Publishing
Per applicazioni specializzate e di trattamento documenti in forma editoriale. Tecnologia di stampa laser e termica, velocità da 1 a 8 pagine al minuto, massima silenziosità.
PG208 - TH760 S



Heavy duty
Massima velocità ed affidabilità per alti volumi di stampa e per lavori multicopie. Velocità fino a 400 cps, gruppo stampa e trascinamento carta specializzati, bar-code.
DM400 - DM717

Specializzazione
Grande varietà di modelli per applicazioni specifiche di: sportello bancario, POS, invalidazione, marcaggio ottico e magnetico, bar-code, con tecnologia ad aghi e laser.

olivetti