di Sergio Polini

Una piccola gerarchia di classi

Brad J. Cox, nel suo Object-Oriented Programming *(Addison-Wesley, 1987), espone la gerarchia di classi implementata nell'Objective-C; l'Appendice A del libro propone delle schede con le specifiche delle diverse classi, in modo da consentire all'utente di farne l'uso più efficace. In ogni scheda fa spicco un «indice di maturità», attraverso il quale si esprime la probabilità che future ridefinizioni e/o reimplementazioni di una classe possano avere impatto sul suo uso. «Si tratta di un modo di prendere atto della realtà», spiega Cox, «e di ammettere che il software non banale matura col tempo» (p. 97). Stanley B. Lippman rincara la dose: «Il buon disegno di una gerarchia di classi è difficile. Chi ci si dedica può ben aspettarsi di dover provare e riprovare più volte. Il disegno di tali gerarchie è oggetto di attive ricerche (e di qualche controversia), e non vi è ancora accordo su un qualche insieme di regole»* (C++ Primer, *Addison-Wesley, 1969, p. 347)*

Le citazioni da Cox e Lippman ci mettono in guardia: decidere l'interfaccia e l'implementazione di una classe, comporre più classi in una gerarchia in cui l'una derivi dall'altra e ne erediti, specificandole o arricchendole, le caratteristiche, è impresa tutt'altro che banale. Con ciò non vi voglio certo scoraggiare: nella pratica, infatti, chi programma si deve curare più di trovare e usare una gerarchia già pronta (magari estendendola), che di ripartire ogni volta da zero. Il vantaggio della programmazione orientata all'oggetto è del resto proprio questo: la possibilità di procedere velocemente sulla base di codice già disponibile e facilmente adattabile a specifiche esigenze.

Quel monito va quindi inteso come un'esortazione a tenere bene a mente che quello che vediamo realizzato in un modo avrebbe potuto essere fatto anche molto diversamente. Avere una qualche idea dei diversi modi in cui una gerarchia di classi può essere costruita aiuta a cogliere i punti forti e deboli di una specifica implementazione, e quindi a farne l'uso più efficace. Per questo motivo, vi propongo una piccola gerarchia con particolari caratteristiche: non è molto ampia, per non allungare troppo una esposizione che, avendo come scopo primario quello di consentirvi di familiarizzarvi con le tecniche della OOP, deve essere sufficientemente analitica; trae ispirazione da gerarchie classiche (quella dello SmallTalk in primo luogo), ma tiene presenti anche quelle che più direttamente si propongono agli utenti dei linguaggi Borland (il Turbo Vision del Pascal e la Class Library del C++), soprattutto per mostrare vantaggi e svantaggi di scelte diverse.

```
TObject
    TUndefinedObject
    TMagnitude
        TNumber
        TString
        TAssociation
    TCollection
        TIndexedCollection
            TFixedSizeCollection
            TOrderedCollection
                TSortedCollection
        TBag
            TSet
                TDictionary
TIterator
    TIndexedCollectionIterator
    TBagIterator
```

Figura 1 - La struttura della nostra gerarchia di classi.

Lo schema generale

Nella figura 1 è tracciato uno schema della gerarchia, che potete trovare già su MC-Link (TPCLASS1.ZIP), come potete vedere, tutto deriva da una classe base *TObject* (fa eccezione la gerarchia di «iteratori», più o meno parallela a quella delle «collezioni», che svolge peraltro funzioni meramente ausiliarie). Si tratta di una impostazione molto frequente, in quanto consente di sfruttare appieno il «polimorfismo» della OOP; come abbiamo già visto, un metodo virtuale definito per una classe può essere ridefinito per tutte le classi da questa derivate, assegnando a puntatori alla classe base gli indirizzi di oggetti istanza sia di questa (se non «astratta») che delle classi derivate, si ottiene che con la «chiamata» di quel metodo mediante uno di tali puntatori venga poi eseguito il codice corrispondente alla definizione del metodo per la classe di cui l'oggetto di volta in volta puntato è istanza (se non vi fosse ancora chiaro, potreste magari tornare agli esempi con classi di animali che vi ho proposto nei mesi scorsi). Ne segue che la flessibilità che tale meccanismo consente è massima quando vi è una sola classe base.

La prima conseguenza è la presenza di strutture di dati un po' diverse da quelle cui siamo abituati; array, set e si-

mili nel Pascal standard sono strutture i cui elementi devono tutti inesorabilmente appartenere ad uno stesso tipo. Il Pascal con oggetti consente invece di realizzare strutture i cui elementi, in quanto puntatori alla classe base, possono contenere gli indirizzi di istanze di qualsiasi classe da quella derivata, divenendo così strutture «polimorfiche». Analogamente, si generalizza anche il concetto stesso di struttura di dati, divenendo la classe astratta «collezione» (*TCollection* nel nostro caso, dove la «T» iniziale sta per «tipo», mentre *PCollection* sarà «puntatore a collezione»): il suo compito è quello di definire i comportamenti comuni ad ogni «gruppo» di oggetti, mentre «gruppi» più specializzati possono essere ricavati semplicemente derivandoli da quello astratto. Vediamo così che da *TCollection* derivano *TIndexedCollection* e *TBag:* la prima è la classe di tutte le collezioni agli elementi delle quali si può avere accesso mediante un indice (come nei più tradizionali array), la seconda è una sorta di contenitore generico in cui tale possibilità d'accesso manca; ambedue possono contenere un numero arbitrario di elementi, limitato solo dalla memoria disponibile.

Da *TIndexedCollection* derivano *TFixedSizeCollection* e *TOrderedCollection:* la prima è più simile al normale array Pascal in quanto ha una dimensione massima, la seconda è una collezione ai cui elementi si può accedere con indici solo «in lettura»: inserimenti e cancellazioni sono invece sottoposti ad un vincolo in quanto la collezione viene detta «ordinata» nel senso che viene mantenuto l'ordine con cui i suoi elementi le sono stati aggiunti (pensate ad uno stack su cui sono consentite le sole operazioni *push* e *pop*). *TSortedCollection* è invece una collezione ordinata in cui viene mantenuto un ordine determinato dal confronto tra gli elementi via via aggiunti e quelli già presenti. Un tale tipo di ordinamento richiede ovviamente oggetti «ordinabili»: ciò vuol dire che in una *TSortedCollection* possono essere inseriti solo oggetti che siano istanza di classi derivate da *TMagnitude,* che è la classe astratta di tutti gli oggetti per cui sia definito un metodo *IsLessThan* (cioè «è minore di»). Dovrebbe essere evidente la natura delle classi *TNumber* e *TString,* sui cui comunque torneremo in un prossimo appuntamento.

La classe *TBag* non è in sé particolarmente utile, ma vale come base di *TSet,* classe di strutture (non indicizzate) in cui non sono ammesse duplicazioni: come per i set del Pascal standard, un elemento viene aggiunto solo se non risulta già presente. Vedremo subito un esempio d'uso. Il vantaggio di una gerarchia di strutture di dati risiede nella possibilità di derivarne facilmente altre. Pensate ad una collezione di record aventi ognuno un campo chiave: è chiaro che, perché il campo chiave sia tale a tutti gli effetti, non vi devono essere due record aventi uguali valori in quel campo. Bene: una possibile soluzione è la classe *TDictionary,* che altro non è che un *TSet* i cui elementi sono tutti istanza della classe *TAssociation,* ovvero di coppie «chiave-valore» (dove «valore» sta per l'insieme dei campi non chiave).

L'esempio di Cox

Prima di vedere in dettaglio la definizione e l'implementazione delle nostre classi, voglio proporvi un adattamento del *Dependency Graph* di Cox, cioè dell'esempio di cui Cox si serve per illustrare i vantaggi della programmazione orientata all'oggetto rispetto a quella tradizionale. Nel sesto capitolo del suo libro mette infatti a confronto due diverse implementazioni di un «grafo» delle «dipendenze» tra i file che compongono

```
unit Nodes;
(*$X+*)
interface

uses Base, Strings, Collect, BagSet;

type
  PNode = ^TNode;
  TNode = object(TString)
    constructor Init(s: string);
    destructor Done; virtual;
    procedure AddReference(var AnotherNode: TNode);
    procedure Visit;
  private
    References: PSet;
    Visited: integer;
  end;

implementation

constructor TNode.Init(s: string);
begin
  TString.Init(s);
  References := nil;
  Visited := 0;
end;

destructor TNode.Done;
begin
  if References <> nil then Dispose(References, Done);
  TString.Done;
end;

procedure TNode.AddReference(var AnotherNode: TNode);
begin
  if References = nil then New(References, Init(10));
  References^.Add(AnotherNode);
end;

procedure TNode.Visit;
const
  Count: integer = 0;
var
  Iterator: PIterator;
begin
  Write('':Count, TNode.Str);
  if References <> nil then begin
    if Count <> 0 then Write(', che');
    Writeln(' dipende da:');
    Inc(Count,4);
    Iterator := References^.InitIterator;
    while Iterator^.More do
      PNode(Iterator^.Next)^.Visit;
    Dec(Count,4);
    Dispose(Iterator, Done);
  end
  else
    Writeln;
end;

end.
```

Figura 2 - Il file NODES.PAS, contenente definizione e implementazione della classe TNode.

un programma, cioè della struttura di dati su cui deve operare un MAKE per produrre un eseguibile a partire dai suoi sorgenti, ricompilando solo quelli che risultano modificati dopo l'ultima compilazione. Il caso ha voluto che due anni fa, per discutere di strutture di dati e di algoritmi ricorsivi, vi proposi un MiniMake che, pur se meno dotato del MAKE della Borland, riusciva a svolgere in tutta onestà (e perfino più rapidamente) il suo compito: nella sostanza, lo stesso compito svolto dal grafo di Cox. Ci vollero sei mesi (dall'aprile al settembre 1989, dopo una puntata introduttiva a marzo) per portare a termine il discorso; è vero che ciò fu dovuto in primo luogo alle frequenti divagazioni (analisi lessica-

```
program PreMake;

uses Base, Collect, BagSet, Nodes;

const
  EOL = 256;
  WRD = 257;

var
  MakeFile: text;
  Target  : TNode;
  Graph   : TSet;
  Iterator: PIterator;

function Lex(var s: string): word;
const
  OldCh: char = #0;
var
  Ch: char;
begin
  while not Eof(MakeFile) do begin
    if OldCh = #0 then Read(MakeFile, Ch)
    else begin
      Ch := OldCh;
      OldCh := #0;
    end;
    if Ch = #13 then begin
      Read(MakeFile, Ch); (* per #10 *)
      Lex := EOL;
      Exit;
    end
    else if Ch = ' ' then begin
      repeat Read(MakeFile, Ch) until Ch <> ' ';
      OldCh := Ch;
    end
    else begin
      s := '';
      repeat
        s := s + Ch;
        Read(MakeFile, Ch)
      until Ch in [#13, ' '];
      OldCh := Ch;
      Lex := WRD;
      Exit;
    end;
  end;
  Lex := 0;
end;
function Filter(var S: Tset; var st: string): PNode;
var
  Node, OldNode: PNode;
begin
  New(Node, Init(st));
  OldNode := PNode(S.Find(Node^));
  if Node^.IsEqual(OldNode^) then begin
    Dispose(Node, Done);
    Filter := OldNode;
  end
  else
    Filter := PNode(S.Add(Node^));
end;

procedure Parse;
var
  Token: word;
  TokenStr: string;
  PN1, PN2: PNode;
begin
  repeat
    Token := Lex(TokenStr);
    case Token of
      WRD: begin
             PN1 := Filter(Graph, TokenStr);
             repeat
               Token := Lex(TokenStr);
               case Token of
                 WRD: begin
                        PN2 := Filter(Graph, TokenStr);
                        PN1^.AddReference(PN2^);
                      end;
                 EOL: begin end;
                 else Exit;
               end;
             until Token <> WRD;
           end;
      EOL: begin end;
      else Exit;
    end;
  until FALSE;
end;

begin
  if ParamCount <> 1 then begin
    Writeln('Uso: PREMAKE <target>');
    Halt(1);
  end;
  Assign(MakeFile, 'MAKEFILE');
  (*$I-*) Reset(MakeFile); (*$I+*)
  if IOError <> 0 then begin
    Writeln('Errore apertura MAKEFILE');
    Halt(2);
  end;
  Graph.Init(10);
  Parse;
  Target.Init(ParamStr(1));
  PNode(Graph.Find(Target))^.Visit;
  Target.Done;
  Iterator := Graph.InitIterator;
  while Iterator^.More do
    Dispose(Iterator^.Next, Done);
  Dispose(Iterator, Done);
  Graph.Done;
  Close(MakeFile);
end.
```

Figura 3 - Il programma PREMAKE che, usando la unit NODES insieme ad altre della gerarchia di figura 1, esegue il lavoro preparatorio di un MAKE: la costruzione del grafo delle dipendenze tra i file.

le e sintattica dell'input, ricorsività, procedura Exec, ecc.), ma l'intricato diagramma che potete ritrovare a pagina 210 del numero di aprile 1989 dà un'idea della complessità della struttura di dati che allora fu necessario costruire.

Supponiamo invece ora di avere a disposizione una gerarchia di classi come quella della figura 1; non vi troviamo immediatamente quello che ci serve, ma possiamo facilmente estenderla. Ragioniamo un attimo: un grafo «è un» insieme di nodi, ognuno dei quali può dipendere da altri; ogni nodo, quindi, oltre ad altre caratteristiche, «è (anche) un» insieme degli altri nodi da cui dipende. Quanto al resto, un nodo «è (anche) un» file, o meglio un nome, cioè una *TString*, con cui venga denotato un file, e deve avere sia metodi per aggiungere le dipendenze da altri file che per poi rispondere l'insieme di tali dipendenze. Un nodo deve cioè essere istanza di una classe come definita nella figura 2.

Discuteremo in altra occasione alcuni aspetti sintattici propri del Pascal 6.0 (come la direttiva $X e la parola chiave **private**, facilmente eludibili da chi usi ancora la versione 5.5). Per ora limitiamoci ad osservare che, a differenza di quanto accadde due anni fa, l'implementazione del nodo è ora estremamente semplice: il groviglio delle dipendenze di un nodo dall'altro viene risolto mediante un puntatore a *TSet, References*, i cui metodi possono essere usati nonostante il fatto che la gerarchia di classi non aveva il minimo sentore di poter essere impiegata anche per realizzare grafi dei nostri nodi.

Può comunque essere utile sottolineare alcuni meccanismi. Come spesso accade, il constructor chiama prima di tutto quello della classe da cui *TNode* deriva, mentre il destructor provvede a chiamare quello di *TString* solo dopo aver rilasciato la memoria eventualmente allocata per *References*. Il metodo *Visit*, inoltre, si incarica di visualizzare gli altri nodi da cui un nodo dipende, e ricorsivamente quelli da cui questi a loro volta dipendono; per far ciò è necessario percorrere gli insiemi puntati dal campo *References* di ogni nodo, ma ciò non può essere fatto con metodi «normali». Sappiamo bene che i **set** del Pascal standard non sono «iterabili», non si può cioè operare iterativamente sui loro elementi, in quanto non esiste un «primo», un «secondo», un «ultimo» elemento. La nostra gerarchia, tuttavia, è affiancata da una gerarchia ausiliaria di «iteratori» che implementa quanto necessario per tutte le classi derivate, direttamente o indirettamente, da *TCollection;* vedremo in particolare che *TSet* eredita l'iteratore di *TBag*.

```
mmake.exe mmalex.tpu mmsim.tpu mmparser.tpu upstr.obj mmake.pas
mmalex.tpu mmalex.pas
mmsim.tpu mmsim.pas
mmparser.tpu mmalex.tpu mmsim.tpu mmparser.pas
upstr.obj upstr.asm
print
```

Figura 4 - Un esempio di MAKEFILE per il programma PREMAKE.

Dopo queste veloci puntualizzazioni, rileviamo che *Visit* si limita a visualizzare sullo schermo il grafo delle dipendenze, senza verificare le date dei vari file e senza procedere a compilazioni. Non sarebbe difficile aggiungere tali funzionalità, ma non faremmo altro che riportare nell'esempio qualche riga del MiniMake. Meglio passare subito a vedere come usare la classe *TNode* in un programma.

Statico e dinamico

Nella figura 3 troviamo appunto un programma che, usando la unit NODES insieme ad altre della gerarchia, costruisce quel grafo delle dipendenze che costituisce la premessa di un MAKE. La cosa più importante da notare è la gestione della memoria. A differenza di altre (ad esempio quella del Turbo Vision), la gerarchia che vi propongo implementa le collezioni in modo tale che è possibile aggiungervi sia elementi allocati dinamicamente con *New* o *GetMem* che normali variabili globali o locali. Come avremo modo di vedere meglio in seguito, ciò comporta che i destructor delle collezioni liberano sì la memoria allocata per le collezioni stesse, ma non quella allocata per i loro elementi (in quanto, appunto, potrebbero non essere «dinamici»). Vi deve provvedere il programmatore, come ho fatto con le ultime istruzioni di PREMAKE, da «Target.Done» a «Graph.Done». Ciò non sarebbe stato necessario, peraltro, se i vari nodi non fossero stati creati con *New*. Ci si imbatte qui in sottili problemi su cui torneremo più ampiamente in seguito; tanto per darvi un'idea, provate a considerare che uno stesso nodo può appartenere, oltre che a *Graph*, anche agli insiemi puntati dal campo *References* degli altri nodi da cui quello dipende: una «deallocazione automatica» (in quanto operata dai destructor) potrebbe portare a tentare di rilasciare più volte una stessa area di memoria, con conseguenze catastrofiche. Ora sarà sufficiente chiarire che la procedura *Filter*, proprio in quanto tutti i nodi sono dinamici, «filtra» i nodi da aggiungere al grafo nel senso che aggiunge ad esso i nodi che non vi siano già contenuti, mentre elimina (rilasciando la memoria da loro occupata) gli altri.

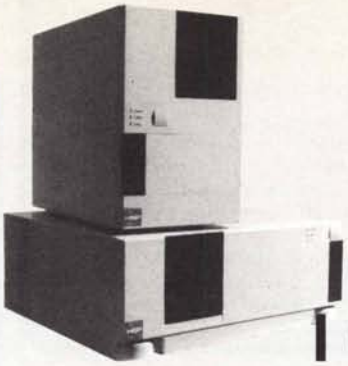
```
mmake.exe dipende da:
    upstr.obj, che dipende da:
        upstr.asm
    mmalex.tpu, che dipende da:
        mmalex.pas
    mmake.pas
    mmparser.tpu, che dipende da:
        mmparser.pas
        mmalex.tpu, che dipende da:
            mmalex.pas
        mmsim.tpu, che dipende da:
            mmsim.pas
    mmsim.tpu, che dipende da:
        mmsim.pas
```

Figura 5 - L'output di PREMAKE con un makefile come quello della figura 4.

PREMAKE richiede la presenza di un file chiamato «MAKEFILE» come quello della figura 4: un normale file testo in cui ogni riga contenga il nome di un file seguito da quelli da cui dipende (il fatto che si tratti dei file che componevano il MiniMake di due anni fa non è casuale...). La funzione *Lex* percorre il file alla ricerca dei singoli nomi di file, che assegna l'uno dopo l'altro al suo parametro variabile. La procedura *Parse* aggiunge i nomi al *TSet Graph* mediante la funzione *Filter* e aggiunge ad ogni nome che sia il primo di una riga i nomi che lo seguono sulla stessa riga. Completati così i preliminari, si inizializza il nodo *Target* con il parametro dato sulla riga comando: con «PREMAKE MMAKE.EXE» si indica che si intende esaminare il grafo delle dipendenze di MMAKE.EXE, con «PREMAKE MMPARSER.TPU» quello della unit MMPARSER.

La figura 5 riproduce l'output del programma nel primo caso: mi pare un risultato più che apprezzabile per poche decine di righe di codice. È vero che non includiamo nel conto i sorgenti delle unit della gerarchia, ma c'è un valido motivo; la unit MMSIM del MiniMake costruiva una struttura di dati utilizzabile solo per quel programma, mentre la unit BAGSET della nostra gerarchia è di uso assolutamente generale: può essere ri-usata così com'è in ogni programma in cui vi sia qualcosa di cui possa dirsi che «è un» insieme.

MS



COMPUTER
HSP
COMPUTER

I CAMPIONI DI POTENZA

DESIGNER - 21
AT 16/21 MHz
da L. 550.000
512K FDD1.2 RS232 PRINTER

DESIGNER SX
386 SX
da L. 949.000
512K FDD1.2 RS232 PRINTER

CAD-25
386 25 MHz
da L. 1.299.000
1MB FDD1.2 RS232 PRINTER

PROCAD-33
386 33 MHz
da L. 1.890.000
64K CACHE 1MB FDD1.2 RS232

IPERCAD-486
486 25 MHz
da L. 3.790.000
1MB FDD1.2 RS232 PRINTER

COPROCESSORI

80287-10	L.	265.000
80387-SX	L.	490.000
80387-25	L.	680.000
80387-33	L.	890.000

HARD DISK

20MB 40ms 3,5" MFM	L.	279.000
40MB 24-28ms 3,5" IDE	L.	350.000
80MB 15ms 3,5" IDE	L.	690.000
120MB 16ms 3,5" IDE	L.	830.000
210MB 15ms 3,5" IDE		TELEF.
330MB 16ms 5,25" SCSI		TELEF.
660MB 16ms 5,25" SCSI		TELEF.
1200MB 16ms 5,25" SCSI		TELEF.
CTRL 2 FDD+2HDD IDE	L.	45.000
CTRL 2 FDD+2HDD SCSI		TELEF.

SCHEDE GRAFICHE

SK. VGA 15 BIT 256K	L.	99.000
SK. VGA 16 BIT 512K	L.	195.000
SK. VGA 16 BIT 1MB TSENG	L.	239.000
SK.U VGA TMS34010		TELEF.
SK. UVGA COMP. 8514/A		TELEF.

OFFERTISSIMA

S.G.VGA 16 BIT 1MByte
con MON. 14" VGA 1024x768
L. 799.000

SOFTWARE

APPLICATIVI PER WINDOWS

WINDOWS 3.0	ING.	L.	189.000
WORD PER WINDOWS	ING.	L.	390.000
EXCEL 2.2 PER WINDOWS	ING.	L.	570.000
TOOLBOOK PER WINDOWS	ING.	L.	540.000

WORD PROCESSOR

WORD 5	ING.	L.	390.000
WORDSTAR 6.0	ING.	L.	490.000
QUATTRO PRO 2.0	ING.	L.	590.000

LINGUAGGI

TURBO BASIC	ING.	L.	120.000
QUICK BASIC	ING.	L.	115.000
TURBO C	ING.	L.	249.000
QUICK C 2.5	ING.	L.	115.000
TURBO PASCAL 5.5	ING.	L.	180.000
TURBO PROLOG	ING.	L.	180.000

DATA BASE

DATA BASE 4.2	ING.	L.	842.000
DBASE IV 1.1	ING.	L.	890.000
FOX BASE + 286 2.1	ING.	L.	349.000
FOX BASE + 386 2.1	ING.	L.	549.000
PARADOX 3.0	ING.	L.	839.000

CAD

AUTO SKETCH V. 2.0	ING.	L.	160.000
AUTODESK ANIMATOR	ING.	L.	430.000
DESIGN CAD 2D 4.0	ING.	L.	275.000
DESIGN CAD 3D 3.0	ING.	L.	365.000
EASY CAD 2.3	ING.	L.	189.000

COMMUNICATIONS

CARBON COPY PCPLUS	ING.	L.	169.000
CROSS TALK XVI	ING.	L.	195.000
LAPLINK III	ING.	L.	165.000
PRO COM PLUS 1.18	ING.	L.	119.000

DESKTOP PUBLISHING

OMNIPAGE 386	ING.	TELEF.
PAGEMAKER 3.01	ING.	TELEF.

MONITOR

VGA 14" MONO PW	L.	200.000
VGA 14" COLORE	L.	530.000
VGA 1024x768		TELEF.
NEC 3D 1024x768	L.	990.000
NEC 4D		TELEF.
NEC 5D		TELEF.
VGA 17" 1024x763 FLAT	L.	1.500.000

ACCESSORI

MOUSE 800DPI ITALIANO	L.	59.000
HANDY SCANNER LOGITECH da	L.	290.000
HANDY SCANNER COLORI		TELEF.
SCANNER MONO PIANO FISSO A4		TELEF.
SCANNER COLORI PIANO FISSO A4		TELEF.
TAVOLETTE GRAFICHE 12"x12"	L.	390.000
TAVOLETTE GRAFICHE 12"x18"	L.	750.000
DISCHETTI 1,44MB	L.	1.600
DISCHETTI 720KB	L.	900
FAX SAMSUNG CON TELEFONO	L.	890.000

NOTEBOOK
A4 kg. 2,8

286	IMB HD20 CGA	L.	2.200.000
286	IMB HD20 VGA	L.	3.050.000
286	IMB HD40 VGA	L.	3.200.000
386SX	IMB HD20 VGA	L.	3.300.000
386SX	IMB HD40 VGA	L.	3.500.000

MODEM

SK. 300/2400	L.	159.000
EST. 300/2400	L.	210.000
SK. MOD. 2400 FAX G2	L.	290.000
SK. MOD. 2400 FAX G3	L.	390.000

STAMPANTI

CITIZEN 1200 PLUS	80C 120S 9A	L.	269.000
EPSON LX400	80C 150S 9A	L.	~~350.000~~
SAMSUNG	80C 300S 9A	L.	380.000
EPSON LX1050	80C 162S 9A	L.	650.000
CITIZEN MSP15E	136C 160S 9A	L.	480.000
CITIZEN PRODOT9X	136C 3000S 9A	L.	750.000
EPSON FX 1050	136C 220S 9A	L.	843.000
CITIZEN 124D	80C 144S 9A		TELEF.
CITIZEN SWIFT 24	80C 192S 24A		TELEF.
EPSON LQ400	80C 150S 24A	L.	~~546.000~~
PANASONIC KX 1123 24A		L.	480.000
NEC P2 PLUS	80C 192S 24A	L.	570.000
EPSON LQ860	80C 200S 24A	L.	1.206.000
NEC P60	80C 268S 24A	L.	990.000
SAMSUNG 136C	220C 24A	L.	990.000
CITIZEN SWIFT 24X	136C 192S 24A		TELEF.
EPSON LQ1050+	136C 220S 24A	L.	1.206.000
NEC P70	136C 268C 24A	L.	1.300.000
EPSON LQ1060	136C 300C 24A	L.	1.519.000
TEXAS INSTRUMENTS A4 6PPM LASER		L.	1.990.000
EPSON EPL7100 A4 6PPM LASER		L.	1.750.000
EPSON EPJ200 A3 2PPM INK JET		L.	2.058.000

INFO.SIST.

CONCESSIONARIO SU ROMA

CENTRO ASSISTENZA
TECNICA PC.
PROGETTAZIONE
RETI LOCALI

Via Malta 8 • 00198 Roma
Tel. (06) 8842378/8411987/8450338
dal Lun. al Sab. 9.00-13.00 / 16.30-19.30

GARANZIA 12 MESI - PREZZI IVA ESCLUSA