di Sergio Polini

La sintassi della OOP

Nei numeri di novembre e dicembre vi ho proposto una breve «Introduzione alla OOP»; non una trattazione esauriente, ma piuttosto un tentativo di mostrare che può valere la pena di esplorare il nuovo terreno che ci viene reso disponibile dalla recente offerta di linguaggi «orientati all'oggetto». Ho cercato in primo luogo di raccontare la genesi del nuovo stile di programmazione, nato quasi per caso dallo sforzo di adattare l'ALGOL alle esigenze della simulazione discreta; ho però anche cercato di mostrare come i concetti introdotti per la prima volta dal Simula si siano dimostrati di utilità molto più generele: grazie ad essi, infatti, è diventato semplice applicare tecniche di sviluppo del software che permettono di ridurre i tempi di programmazione e soprattutto di test e di manutenzione. A partire da questo mese, la rubrica Turbo Pascal e la nuova rubrica C++ cercheranno di dare maggiore concretezza a quel discorso introduttivo, con esempi «sul campo» di cosa voglia dire OOP e soprattutto di cosa si possa fare con la OOP

Nelle due puntate della *Introduzione alla OOP* vi ho proposto solo brevi cenni alle soluzioni che la OOP offre a problemi che capitano molto spesso, come l'adattamento a nuove esigenze o la estensione a nuove applicazioni di codice già scritto, da noi o da altri. Ogni linguaggio, infatti, ha un suo proprio modo di essere «orientato all'oggetto», e una esposizione delle diverse caratteristiche di ognuno avrebbe inevitabilmente appesantito il discorso. La rubrica è la sede più opportuna per una esposizione più analitica che, pur facendo riferimento implicito o esplicito ad una storia e ad istanze metodologiche già raccontate nella *Introduzione,* si occupi con la dovuta attenzione anche dei dettagli e, perché no, delle idiosincrasie, della OOP come attuabile in Turbo Pascal.

Quasi come un record

Una delle caratteristiche più importanti della OOP è la particolare organizzazione che viene imposta ai tipi di dati. Non è difficile trovare programmi Pascal che iniziano con una lunga dichiarazione di tipi e di variabili globali: elencati sì con un certo ordine, nel senso che la dichiarazione del tipo «T» deve precedere quella del tipo «array di T», ma per il resto sostanzialmente alla rinfusa. L'ordine è spesso solo la conseguenza di un (sano) vincolo sintattico, che impone di dichiarare qualsiasi cosa prima di poterla usare, ma è difficile individuare anche un ordine «logico».

Ben diverso il caso della OOP. Come abbiamo accennato nella *Introduzione,* la programmazione orientata all'oggetto è tale in quanto presume di avere a che fare con oggetti «attivi», in grado cioè di rispondere secondo propri «metodi» a «messaggi» inviati dall'applicazione. Quasi come se fossero entità vive di un mondo reale invece che astrazioni sintattiche. Il paragone potrà sembrarvi (come a me sembrava le prime volte) un po' assurdo. Potrete però constatare che non si tratta di una fantasiosa metafora, ma di una vera e propria guida all'ordinato sviluppo di un programma. Ogni programma non banale, infatti, non è altro che l'implementazione di un modello della realtà, di una realtà che è effettivamente popolata da entità dotate ognuna di proprie caratteristiche e di propri comportamenti. Dovrebbe quindi essere chiara (o almeno lo sarà quando avremo acquisito maggiore familiarità con la OOP) la superiore efficacia di un ragionamento condotto in termini di modelli di tali entità piuttosto che di parametri valore e parametri variabili, di **if** e **case**, di cicli **repeat** e **while**.

Diciamo che viviamo in un mondo popolato da «oggetti». Gli oggetti del mondo reale presentano moltissime relazioni tra di loro: un melo *è un* albero proprio come un pero, un'automobile *è un* mezzo di trasporto come un treno o un aeroplano, un italiano *è un* europeo come un francese, una cambiale *è un* titolo di credito come un assegno, una dialog box *è una* finestra come un menu pull-down, un cerchio *è una* figura geometrica come un triangolo, e così via. Parliamo qui di oggetti aventi una certa «struttura», cioè «caratteristiche» tali da rendere possibili considerazioni del tipo: gli oggetti A e B hanno in comune le «caratteristiche» *alfa* e *beta,* in più A possiede la «caratteristica» *gamma,* B la «caratteristica» *delta*; A e B sono dunque diversi, ma ambedue *sono* C, se C è un oggetto avente le sole «caratteristiche» *alfa* e *beta.*

La possibilità di stabilire che un oggetto, per certi aspetti, *è un* altro, consente di attribuire un ordine «logico» ai tipi di dati di un programma: dai tipi più generali vengono derivati tipi via via più specifici (A e B «derivano da» C), nel modo che vedremo un po' alla volta. La necessità che si tratti di tipi «strutturati» ci riporta invece ad un concetto tradizionale del Pascal (e di altri linguaggi), quello di **record**: quelle che abbiamo genericamente chiamato «caratteristiche» sono infatti molto simili ai campi di un **record**. Sono poche, anche se molto importanti e molto potenti, le variazioni sintattiche che ci portano ad un «Pascal con oggetti».

Tipo = object(tipo-padre)

Per descrivere gli «oggetti», il Turbo Pascal riprende dall'Object Pascal la parola chiave **object.** Avrei preferito qualcosa come **class**, analogamente a quanto avviene in altri linguaggi (SmallTalk e C++, ad esempio); il motivo è che, come vedremo in seguito, con dichiarazioni di **object** si definiscono in realtà tipi inseriti in una «gerarchia di classi» molto simile ad un albero genealogico o alle classificazioni care ai biologi, mentre i veri e propri «oggetti» sono detti istanze di una classe più o meno come una variabile viene detta appartenente ad un tipo. Un tipo **object** è quindi la *classe* di tutti gli *oggetti* che ne possono essere istanza. Poco male: l'importante è capirsi.

La sintassi della dichiarazione di un **object** è molto simile a quella di un **record** (figura 1). Una prima differenza risiede nella possibilità di indicare tra parentesi, subito dopo la parola chiave **object**, il tipo-padre da cui quello dichiarato deriva: si tratta del meccanismo mediante il quale vengono esplicitate quelle relazioni «è un» cui abbiamo fatto cenno prima. Notiamo poi che, accanto ai familiari campi-dati, è possibile elencare anche i *metodi*: si tratta di quelle funzioni e procedure che implementano il «comportamento» dell'oggetto. Più esattamente, il comportamento di un oggetto è definito da un insieme di operazioni (i metodi) che hanno accesso al suo stato (i campi-dati). Ciò comporta che lo stato di un oggetto dovrebbe rimanere, a rigore, nascosto; dovrebbe essere possibile accedervi solo mediante appositi metodi.

Ricorderete che nella *Introduzione* ho affermato che la programmazione orientata all'oggetto consente di realizzare efficacemente quel «mascheramento delle informazioni» *(information hiding)* proposto da David Parnas fin dai primi anni '70: in OOP si parla di *incapsulamento*, nel senso che la rappresentazione di un tipo viene «incapsulata» in una definizione che fornisce anche un insieme di funzioni di interfaccia. La sintassi degli **object** del Turbo Pascal 5.5 fornisce solo alcuni degli strumenti a ciò necessari: vengono sì affiancati i metodi ai tradizionali campi-dati, e le **unit** consentono di ben «nascondere» il codice dei metodi nella sezione **implementation,** lasciando nella **interface** la sola intestazione; ma l'accesso ai campi dati è sempre e comunque libero. Il manuale si limita a «raccomandare» di non leggere o modificare direttamente il valore di un campo-dato, ma di accedere ad esso solo mediante opportuni metodi (ad esempio: *GetX* per leggere il campo *X*, *SetX* per modificarne il valore). Sarebbe tuttavia molto meglio se esistessero delle vere e proprie barriere. Se tornate per un attimo all'esempio delle matrici che vi ho proposto a dicembre, riconoscerete che la possibilità di usare indifferentemente diverse rappresentazioni (array di numeri o array di puntatori) in uno stesso programma sussiste solo se in esso non si tenta alcun accesso diretto all'una o all'altra rappresentazione.

```
TipoStrutturato = record
  CampoDati1 : TipoDelCampo1;
  CampoDati2 : TipoDelCampo2;
  ...
  CampoDatiN : TipoDelCampoN;
end;

TipoClasse = object (TipoPadreOpzionale)
  CampoDati1 : TipoDelCampo1;
  CampoDati2 : TipoDelCampo2;
  ...
  CampoDatiN : TipoDelCampoN;
  Metodo1;
  Metodo2;
  ...
  MetodoN;
end;
```

Figura 1 - Un confronto tra la dichiarazione di un record e quella di un object nel Pascal 5.5; in quest'ultima è possibile indicare, accanto ai campi-dati, anche «metodi», cioè le funzioni e procedure che implementano il «comportamento» dell'oggetto; si può anche indicare il tipo-padre da cui quello dichiarato deriva, «ereditandone» campi-dati e metodi.

```
TipoClasse = object (TipoPadreOpzionale)
  CampoDati1 : TipoDelCampo1;
  CampoDati2 : TipoDelCampo2;
  ...
  CampoDatiN : TipoDelCampoN;
  Metodo1;
  Metodo2;
  ...
  MetodoN;
private
  CampoDatiPrivato1 : TipoDelCampoPrivato1;
  CampoDatiPrivato2 : TipoDelCampoPrivato2;
  ...
  CampoDatiPrivatoN : TipoDelCampoPrivatoN;
  MetodoPrivato1;
  MetodoPrivato2;
  ...
  MetodoPrivatoN;
end;
```

Figura 2 - Il Turbo Pascal 6.0 permette di nascondere in una sezione «private» alcuni campi-dati e metodi, l'accesso ai quali è possibile solo nella unit in cui sono dichiarati.

Il nuovo Turbo Pascal 6.0 introduce finalmente un meccanismo di protezione, mediante la parola chiave **private** (mutuata dal C++). Nella versione 5.5 la dichiarazione dei campi-dati deve precedere quella dei metodi, ma nella 6.0 è possibile spostare i campi-dati, come anche alcuni metodi, in una sezione «privata» posta in fondo alla dichiarazione dell'**object** (figura 2). Campi-dati e metodi «privati» sono accessibili solo dal codice della **unit** in cui il relativo **object** viene dichiarato.

Un semplice esempio

Nelle figure 3 e 4 vi propongo un esempio (piuttosto scheletrico...) dell'uso della nuova parola chiave. Approfitto dell'occasione per darvi anche una dimostrazione di come una classe derivata da un'altra ne possa «ereditare» le caratteristiche.

Nella **unit** ESEMPIO.PAS viene dichiarata in primo luogo una semplice *ClasseBase*, niente più che un campo-dato intero e due metodi di accesso: *Assegna* e *Stampa* per assegnare un valore al campo-dato e per poi visualizzarlo. Segue la dichiarazione di una *ClasseDerivata* che, accanto a metodi

```
Unit Esempio;

interface

type
  ClasseBase = object
    procedure Assegna(x: integer);
    procedure Stampa;
  private
    i: integer;
  end;
  ClasseDerivata = object(ClasseBase)
    procedure Assegna(x, y: integer);
    procedure Stampa;
  private
  (* i: integer;       "Error 4: Duplicate Identifier" *)
    j: integer;
  end;

implementation

procedure ClasseBase.Assegna(x: integer);
begin
  i := x;
end;

procedure ClasseBase.Stampa;
begin
  Writeln('i: ', i);
end;

procedure ClasseDerivata.Assegna(x, y: integer);
begin
  i := x;          (* Assegnazione della variabile "ereditata" *)
  j := y;
end;

procedure ClasseDerivata.Stampa;
begin
  Writeln('i: ', i);
  Writeln('j: ', j);
end;

end.
```

Figura 3 - Una unit Turbo Pascal 6.0 nella quale vengono definite due classi, l'una derivata dall'altra, ambedue con campi-dati privati. Notate che non è possibile ridichiarare il campo-dato «i» in ClasseDerivata, in quanto automaticamente ereditato da ClasseBase.

```
Program Demo;

uses Esempio;

type
  ClasseDerivata2 = object(ClasseDerivata)
    procedure Assegna(x,y,z: integer);
    procedure Stampa;
  private
    k: integer;
  end;

procedure ClasseDerivata2.Assegna(x,y,z: integer);
begin
  ClasseDerivata.Assegna(x,y);
  (* "i := x" e "j := y" genererebbero l'errore "Unknown identifier" *)
  k := z;
end;

procedure ClasseDerivata2.Stampa;
begin
  ClasseDerivata.Stampa;
  (* l'accesso a i o j genererebbe l'errore "Unknown identifier" *)
  Writeln('k: ', k);
end;

var
  b : ClasseBase;
  d : ClasseDerivata;
  d2: ClasseDerivata2;

begin
  b.Assegna(1);
  b.Stampa;
  (* "b.i := 10" o "Writeln('b.i: ', b.i)" genererebbero *)
  (* l'errore "Field identifier expected" *)
  d.Assegna(2, 3);
  d.Stampa;
  d2.Assegna(4, 5, 6);
  d2.Stampa;
end.
```

Figura 4 - Un breve demo che, usando la unit della figura precedente, mostra sia l'inaccessibilità dei campi-dati dichiarati «privati», sia come sia comunque possibile un accesso mediante appositi metodi. Ciò consente di cambiare la dichiarazione della ClasseBase, ad esempio rendendo real il campo «i», circoscrivendo le variazioni al solo codice della unit.

analoghi a quelli di *ClasseBase,* «aggiunge» al campo-dato *i* un altro campo *j*. «Aggiunge» vuol dire che *ClasseDerivata,* essendo derivata da *ClasseBase,* si porta con sé tutti i campi-dati di questa, al punto che non è possibile dichiararne altri con lo stesso nome. Ciò non impedisce, peraltro, l'accesso in lettura e in scrittura al campo ereditato, come si può agevolmente verificare esaminando l'implementazione dei metodi. In ogni caso, possiamo vedere che *ClasseDerivata* «è una» *ClasseBase,* nel senso che, pur avendo qualcosa in più (il campo-dati *j*) e qualcosa di diverso (vengono ridefiniti i metodi), ha qualcosa in comune (il campo *i*); un po' come una zebra «è un» cavallo con qualcosa in più (le strisce bianche e nere) e qualcosa di diverso (vive in altri ambienti), ma anche molto in comune.

La sezione **implementation** svela anche come sia possibile dotare oggetti diversi di metodi con lo stesso nome. A differenza di quanto accade nelle normali **unit** l'intestazione del codice che implementa un metodo non riproduce quella anticipata nella **interface,** in quanto vi compare anche il nome della classe. La notazione è identica a quella usata per l'accesso ai campi di un **record:** il nome dell'**object**, un punto, il nome del metodo. In questo modo diventa facile distinguere l'implementazione del metodo *Assegna* di *ClasseBase* da quella dello stesso metodo di *ClasseDerivata.*

Nel programma DEMO.PAS si può vedere come la stessa notazione venga usata per «chiamare» i metodi, ma su questo ritorneremo tra un attimo. Ora ci soffermiamo sulla dichiarazione di *ClasseDerivata2,* in quanto costituisce un esempio di uso di una classe con una sezione **private:** i campi-dati vengono comunque ereditati, ma non è più possibile accedervi se non attraverso metodi «pubblici» (non privati). Questo scarno esempio si propone infatti anche di rimarcare l'importanza dei metodi di accesso: per salvaguardare le possibilità di riusabilità ed estensibilità del codice tipiche della OOP, è consigliabile nascondere la rappresentazione in memoria delle classi rendendola **private;** per rendere agevole la dichiarazione di classi derivate da altre di un'altra **unit,** è opportuno dotare queste ultime di un completo set di metodi di accesso (compresi quelli del tipo *GetX* e *SetX*). Le prime volte che ci si cimenta con la OOP tali metodi sembrano rendere inutilmente pesante la dichiarazione delle classi, ma si deve anche con-

```
type
  ClasseBase = record
    i: integer;
  end;

var
  b: ClasseBase;

procedure ClasseBaseAssegna(var istanza: ClasseBase; x: integer);
begin
  istanza.i := x;
end;

begin
  ClasseBaseAssegna(b, 1);
end.
```

Figura 5 - Una traduzione in «normale» Pascal di quello che accade quando viene inviato un «messaggio» all'istanza di una classe: la procedura da eseguire viene individuata grazie alla classe cui appartiene l'istanza cui il messaggio viene inviato, e viene passato come parametro anche l'indirizzo dell'istanza (in quanto parametro variabile).

siderare che lo «smart linker» del Turbo Pascal è in grado di eliminare dal programma eseguibile molti dei metodi che non risultino effettivamente utilizzati.

Il parametro invisibile

Dicevamo che la notazione con il punto viene usata anche per la «chiamata» dei metodi. In realtà non si tratta di normali chiamate di procedura, proprio perché quello che viene eseguito è codice appartenente alla classe di cui una variabile è istanza: nel programma DEMO vengono eseguiti tre metodi *Assegna* e tre metodi *Stampa*, ma, nonostante i nomi comuni, viene eseguito ogni volta codice diverso e, come se non bastasse, ad *Assegna* vengono passati prima uno, poi due e infine tre parametri. Ricorderete che, nel gergo della OOP, si dice che viene inviato alle istanze delle tre classi un «messaggio», al quale ognuna risponde secondo il proprio «metodo». Quello che accade dietro le quinte non è poi così misterioso: la notazione con il punto viene tradotta dal compilatore in una chiamata della procedura definita per la classe di cui è istanza la variabile alla sinistra del punto, passando come primo parametro l'indirizzo della variabile: in questo modo può avvenire che, mandando a *b* il messaggio *Assegna(1)*, venga eseguito il metodo *Assegna* di *ClasseBase* e venga assegnato 1 al campo *i* di *b* (la figura 5 propone una traduzione in «normale» Pascal del tutto).

```
procedure ClasseBase.Assegna(x: integer);
begin
  Self.i := x;
end;
```

Figura 6 - Una implementazione del metodo Assegna di ClasseBase, con esplicitazione del parametro Self. Il ricorso a Self è peraltro quasi sempre superfluo e, come tale, da evitare a meno che non risulti necessario per evitare ambiguità.

Questo parametro, pur se invisibile, ha un nome: viene chiamato *Self* (come in SmallTalk), e può anche essere usato nel codice dei metodi, come si vede nella figura 6. Va detto, però, che l'uso esplicito di *Self* dovrebbe di norma essere evitato, proprio perché quasi sempre superfluo. Vi si può tuttavia far ricorso per evitare ambiguità, come quella che si avrebbe se la unit ESEMPIO definisse anche una variabile *i* al di fuori delle classi.

È facile vedere cosa accadrebbe se non fosse previsto questo meccanismo all'apparenza un po' bizzarro: saremmo costretti ad inventare diversi nomi per procedure che, pur con qualche differenza, farebbero sostanzialmente la stessa cosa; se adottassimo convenzioni come quella della figura 5 (nome della procedura dato dalla descrizione dell'azione immediatamente preceduta dal nome del tipo), potrebbe poi capitare che, se *ClasseDerivata* non ridefinisse il metodo *Assegna* ma si limitasse ad «ereditarlo», ci troveremmo a chiamare *ClasseBaseAssegna* passandole come parametro l'indirizzo di una variabile appartenente a *ClasseDerivata*.

Inconvenienti più fastidiosi che gravi, e in realtà quel parametro invisibile ha ben altre potenzialità di quanto abbiamo potuto vedere in questo primo appuntamento. Ne riparleremo.

MC