di Sergio Polini

Le classi astratte

Prima di continuare il discorso sulla nostra piccola gerarchia di classi, sono costretto a ritornare sulla unit TSR pubblicata lo scorso anno, per confermare che i sorgenti pubblicati su MC sono corretti ed esenti da errori. Purtroppo copiare 720 linee di codice non è facile, ed è ben possibile che ci scappi qualche errore. Questo è quanto è capitato anche a Marco Scaglioni di Luino (Varese), che mi ha mandato la stampa del file TSRINT.ASM come da lui copiato, evidenziando due istruzioni rifiutate dal Turbo Assembler. È successo semplicemente che due «WORD PTR» sono diventati «WORD PRT». «PTR» è diventato «PRT». Tutto qui. Capisco che errori di tale natura sono difficili da scoprire, soprattutto se si insinua il dubbio che vi possa essere qualche pecca nel listato pubblicato. Ma su questo punto posso tranquillizzarvi: i listati pubblicati sono corretti. Chi voglia evitare le fatiche (e le insidie) del lavoro di copiatura manuale, può comunque ottenere i sorgenti sia chiedendone il dischetto che prelevandoli da MC-Link (TSRTP100.ZIP)

Eccoci finalmente giunti ad esaminare da vicino interfaccia ed implementazione delle classi che fanno parte della nostra piccola gerarchia. Ripeto che questa va intesa soprattutto come una introduzione «sul campo» alla programmazione orientata all'oggetto; per questo motivo, uno dei principali obiettivi che mi sono assegnato è stata la concisione: poche classi, pochi metodi, qualche approssimazione, in attesa di poter poi passare sia ad una gerarchia più complessa come quella del Turbo Vision, sia a sue estensioni che ci aiutino a realizzare programmi di buona «sostanza» oltre che di buona «apparenza».

L'oggetto e il suo contrario

Quattro sono le classi astratte della nostra gerarchia: *TObject, TMagnitude, TCollection e TIterator*. Lo scopo di una classe astratta, come ripetuto anche nell'articolo di Tommaso Masi di questo mese, è quello di consentire alle sue sottoclassi di condividere un insieme di proprietà comuni, massimizzando in questo modo il riutilizzo del codice (che non deve essere duplicato in ogni sottoclasse). Il disegno di tali classi è quindi sempre un po' arduo: si tratta di anticipare le caratteristiche di un sistema di sottoclassi, decidendo quanto deve essere comune a ciascuna di esse e cosa ognuna dovrà avere di specifico. In pratica, il disegno di una classe astratta non può procedere che per tentativi (ricordate le citazioni da Cox e da Lippman del mese scorso?).

Dal vostro punto di vista, ciò comporta che la motivazione di alcune scelte vi potrà essere pienamente chiara solo dopo che avremo discusso tutta la gerarchia. Questa è una regola generale. I vantaggi della OOP non discendono come manna dal cielo, ma richiedono un minimo di sforzo iniziale: una gerarchia di classi non è — ripeto — come una tradizionale libreria di funzioni; non si giunge lontano se ci si limita a cercare quello che serve in un indice analitico. Bisogna in primo luogo imporsi di farsi un'idea non troppo vaga almeno dei principi base del suo disegno. Non ci vuole molto, ed è un investimento che promette larghi dividendi.

Guardate ad esempio il metodo *Hash* nella figura 1: compare nella interfaccia della classe *TObject* (così come nelle corrispondenti classi di Smalltalk o dell'Objective-C) in quanto serve per agevolare l'implementazione di altre classi come *TSet*. Per ora dobbiamo limitarci a dire che ritorna una *word*; solo quando avremo esaminato le classi *TBag, TSet* e *TDictionary* potremo capire appieno perché occorre che ogni classe della gerarchia possa rispondere al messaggio *Hash*.

Fatta questa necessaria premessa, osserviamo che la unit BASE contiene le definizioni delle classi *TObject* e *TUndefinedObject*. Come per ogni altra classe, definiamo accanto al tipo anche un puntatore al tipo; dovrebbe essere infatti chiaro che, essendo puntatori e parametri variabile gli strumenti principali del polimorfismo, faremo un ampio uso dei puntatori. Vediamo poi che, accanto al «misterioso» metodo *Hash*, ne abbiamo altri che ci rispondono il «nome» della classe, che verificano l'eguaglianza di due oggetti, che producono una immagine in output di un oggetto.

Per dare un po' di generalità a tale ultimo metodo, l'interfaccia della unit dichiara una variabile di tipo «file di testo» *StdOut*, analoga a quella ben familiare a chi programmi in C, inizializzata nell'apposita sezione della unit. Si ottiene così che, passando al metodo *PrintOn* l'argomento *StdOut* si ha un output su video, ma passando un altro file di testo si può avere l'output su disco. Il metodo *PrintOn* è anche l'unico ad avere una implementazione «concreta»; gli altri si limitano a chiamare la procedura *Abstract* che, come l'omonima del Turbo Vision, provoca un errore di esecuzione con codice 211. Fa eccezione il destructor, su cui torneremo tra un attimo.

La classe *TUndefinedObject* ha un'unica istanza, allocata nella sezione di inizializzazione e «puntata» dal puntatore *Null*, una sorta di variante del familiare **nil**. Lo scopo è quello di poter manipolare anche oggetti «nulli», ad esempio per inizializzare un array di oggetti più o meno come un array di interi potrebbe essere inizializzato con tanti zeri. In questo caso, come vedremo, mandando ad un array vuoto il messaggio *PrintOn*, otterremo in output il «nome» dei suoi «oggetti nulli», cioè tanti «Nil» quanti sono i suoi elementi. *TUndefinedObject* eredita infatti da *TObject* il metodo *PrintOn*, che scrive il risultato del metodo *Name*, il quale ritorna «Nil» per istanze di *TUndefinedObject*.

Due parole su constructor e destructor

I manuali del Turbo Pascal 5.5 e 6.0 ammoniscono che ogni classe per la quale sono dichiarati metodi virtuali de-

ve avere un constructor, ma le nostre classi astratte non ne hanno alcuno. In realtà, se si ricorda a cosa serve un constructor (ne abbiamo parlato a marzo), si può ben intendere che una classe astratta potrebbe anche non averne in quanto non può avere istanze. Un po' diverso il caso del destructor. Sempre a marzo abbiamo visto che è bene che i destructor siano virtuali; dichiarare un destructor *Done* virtuale per la classe base della gerarchia può essere utile in quanto forza a dichiarare virtuali tutti i destructor con lo stesso nome delle classi derivate.

La classe *TUndefinedObject* si distingue inoltre da tutte le altre della gerarchia, in quanto ha un costructor **privato**. Ricordo che chi abbia ancora il Pascal 5.5 può fare a meno della nuova parola chiave, semplicemente portando all'inizio della dichiarazione della classe gli eventuali campi-dati (le variabili d'istanza) che apparissero dopo la keyword **private**. Ma questa può essere utile. Nel nostro caso, ad esempio, ci consente di prevedere che vi sia un'unica istanza della classe *TUndefinedObject:* essendo il constructor inaccessibile da parte di codice che non sia quello della unit BASE, solo in essa è possibile creare e inizializzare istanze di *TUndefinedObject*. Proprio per questo motivo il metodo *IsEqual di TUndefinedObject* può limitarsi a verificare il tipo del suo argomento: perché un oggetto sia uguale ad un'istanza di *TUndefinedObject,* è sufficiente che sia anch'esso istanza della stessa classe, in quanto vi è una sola istanza di tale classe.

La classe *TObject* consente di creare oggetti confrontabili solo quanto alla loro uguaglianza, ma ciò non è sempre sufficiente; abbiamo già accennato, ad esempio, alla presenza di collezioni «ordinate». Abbiamo quindi bisogno di oggetti confrontabili anche secondo un criterio che ci dica quale viene «prima» di quale altro. La unit della figura 2 ci propone una classe astratta *TMagnitude* che aggiunge a *TObject* solo un metodo *IsLessThan* che verrà poi precisato nelle classi derivate, ognuna delle quali stabilirà i propri criteri per determinare se un'instanza «è minore di» un'altra.

```
unit Base;

interface

type
  PObject = ^TObject;
  TObject = object
    destructor Done; virtual;
    function Name: string; virtual;
    function Hash: word; virtual;
    function IsEqual(var o: TObject): boolean; virtual;
    procedure PrintOn(var f: text); virtual;
  end;
  PUndefinedObject = ^TUndefinedObject;
  TUndefinedObject = object(TObject)
    function Name: string; virtual;
    function Hash: word; virtual;
    function IsEqual(var o: TObject): boolean; virtual;
  private
    constructor Init;
  end;

var
  StdOut: text;
  Null: PUndefinedObject;

procedure Abstract;

implementation

procedure Abstract;
begin
  RunError(211);
end;

destructor TObject.Done;
begin
end;

function TObject.Name: string;
begin
  Abstract;
end;

function TObject.Hash: word;
begin
  Abstract;
end;

function TObject.IsEqual(var o: TObject): boolean;
begin
  Abstract;
end;

procedure TObject.PrintOn(var f: text);
begin
  Write(f, Name);
end;

constructor TUndefinedObject.Init;
begin
end;

function TUndefinedObject.Name: string;
begin
  Name := 'Nil';
end;

function TUndefinedObject.Hash: word;
begin
  Hash := 0;
end;

function TUndefinedObject.IsEqual(var o: TObject): boolean;
begin
  IsEqual := TypeOf(o) = TypeOf(TUndefinedObject);
end;

begin
  Assign(StdOut, ''); Rewrite(StdOut);
  New(Null, Init);
end.
```

Figura 1 - La unit BASE.PAS, contenente la definizione delle classi TObject e TUndefinedObject.

```
unit Magnitud;

interface

uses Base;

type
  PMagnitude = ^TMagnitude;
  TMagnitude = object(TObject)
    function IsLessThan(var m: TMagnitude): boolean; virtual;
  end;

implementation

function TMagnitude.IsLessThan(var m: TMagnitude): boolean;
begin
  Abstract;
end;

end.
```

Figura 2 - La unit MAGNITUD.PAS, contenente la definizione della classe astratta TMagnitude, da cui derivano le classi degli oggetti confrontabili.

Le collezioni

La figura 3 ci propone invece la unit COLLECT, dove viene definita la classe *TCollection*.

Accanto ad essa troviamo un tipo *TObjPArray*, per array di puntatori ad oggetti, che ci sarà utile per l'implementazione delle collezioni. A dire il vero, in altre gerarchie viene usata una più ampia articolazione delle classi: vi sono classi come *ArrayOb* (nella «class library» di Gorlen) o *IdArray* (Objective-C) che svolgono funzione analoga a *TObjPArray*, ma, come ho detto prima, la concisione è ora per noi un obiettivo fondamentale, anche a prezzo di qualche approssimazione.

Torniamo quindi a *TCollection*. Si tratta di una classe centrale in qualsiasi gerarchia (ne ritroviamo una anche nel Turbo Vision), in quanto racchiude in sé le caratteristiche e il comportamento comuni ad ogni «gruppo» di oggetti. Deve necessariamente trattarsi di una classe molto generale, in quanto può dirsi «collezione», nel gergo della OOP, qualsiasi aggregato di qualsiasi numero di oggetti che possa essere considerato come un tutto. Possono far parte di una collezione i file contenuti in una directory, gli ordini pervenuti da un dato cliente, le barre di un istogramma, le caselle di uno spreadsheet. Sono però anche collezioni, come abbiamo già visto, le strutture di dati tipiche della programmazione tradizionale, come array, liste, code, stack, alberi, grafi, dizionari, ecc. Il compito di una classe astratta *TCollection* è quello di racchiudere in sé quanto vi può essere di comune a tutte le collezioni. In considerazione della nostra esigenza di concisione, ci limiteremo proprio all'essenziale, anche perché avremo modo di esaminare in seguito la più ricca omonima classe del Turbo Vision.

In primo luogo, considereremo quelle proprietà che competono alle collezioni per la loro derivazione dalla classe base *TObject*. Non vengono reimplementati né il destructor né il metodo *Name*, in quanto questi possono essere ereditati così come sono (ovviamente non sarà così nelle classi «concrete» che deriveremo da *TCollection*). Il metodo *Hash* ritorna ora zero, mentre *IsEqual* e *PrintOn* vengono più sostanzialmente riscritti, come vedremo tra un attimo.

Quattro sono i metodi che distinguono una collezione da un altro oggetto: *Add* aggiunge un oggetto al «gruppo», *Remove* lo toglie, *Find* ci permette di vedere se un dato oggetto è presente o meno nella collezione; c'è poi *InitIterator*, con cui si inizializza un meccanismo attraverso il quale percorrere la collezione oggetto per oggetto. A differenza di *Add* e *Remove*, che rimangono per ora astratti come *InitIterator*, i metodi *IsEqual*, *PrintOn* e *Find* sono funzionanti a tutti gli effetti. Qui bisogna capirsi: non è possibile creare istanze di *TCollection*, in quanto, ad esempio, il tentativo di aggiungervi un oggetto mediante il metodo *Add* farebbe scattare un errore di esecuzione 211. Le classi derivate da *TCollection* potranno tuttavia ridefinire il metodo *Add* (e tutti quelli che chiamano la procedura *Abstract*), e avvalersi dei metodi *IsEqual*, *PrintOn* e *Find* semplicemente ereditandoli implicitamente da *TCollection*. In questo senso si tratta di metodi «funzionanti».

Gli iteratori

C'è qualcosa di comune ai metodi *IsEqual*, *PrintOn* e *Find:* usano tutti variabili locali di tipo *PIterator*. Si tratta di qualcosa di un po' diverso da quanto si trova nella programmazione tradizionale, o meglio di più generale. Ricordavamo la volta scorsa che i **set** del Pascal non sono «iterabili», nel senso che non è possibile percorrere gli elementi di un **set** dal primo all'ultimo; «primo» e «ultimo» non hanno anzi alcun senso con riguardo agli elementi di un set. È invece sempre possibile «iterare» attraverso una collezione.

Nell'articolo di questo mese della ru-

```
unit Collect;

interface

uses Base;


type
  PObjPArray = ^TObjPArray;
  TObjPArray = array[1..65520 div SizeOf(PObject)] of PObject;
  PIterator = ^TIterator;
  TIterator = object
    destructor Done; virtual;
    function More: boolean; virtual;
    function Next: PObject; virtual;
  end;
  PCollection = ^TCollection;
  TCollection = object(TObject)
    function Hash: word; virtual;
    function IsEqual(var o: TObject): boolean; virtual;
    procedure PrintOn(var f: text); virtual;
    function InitIterator: PIterator; virtual;
    function Add(var o: TObject): PObject; virtual;
    function Remove(var o: TObject): PObject; virtual;
    function Find(var o: TObject): PObject; virtual;
  end;


implementation

function TCollection.Hash: word;
begin
  Hash := 0;
end;
```

```
function TCollection.IsEqual(var o: TObject): boolean;
var
  This, Other: PIterator;
  Result: boolean;
begin
  if (TypeOf(o) <> TypeOf(Self)) then begin
    IsEqual := FALSE;
    Exit;
  end;
  This := InitIterator;
  Other := PCollection(@o)^.InitIterator;
  Result := TRUE;
  while This^.More and Other^.More and Result = TRUE do
    if not This^.Next^.IsEqual(Other^.Next^) then
      Result := FALSE;
  if This^.More or Other^.More then Result := FALSE;
  Dispose(This, Done);
  Dispose(Other, Done);
  IsEqual := Result;
end;

procedure TCollection.PrintOn(var f: text);
var
  Iterator: PIterator;
begin
  Iterator := InitIterator;
  Write(f, Name, '(');
  while Iterator^.More do begin
    Iterator^.Next^.PrintOn(f);
    if Iterator^.More then Write(f, ',');
  end;
  Writeln(f, ')');
  Dispose(Iterator, Done);
end;

function TCollection.InitIterator: PIterator;
begin
  Abstract;
end;

function TCollection.Add(var o: TObject): PObject;
begin
  Abstract;
end;

function TCollection.Remove(var o: TObject): PObject;
begin
  Abstract;
end;

function TCollection.Find(var o: TObject): PObject;
var
  Iterator: PIterator;
  P: PObject;
begin
  Iterator := InitIterator;
  while Iterator^.More do begin
    P := Iterator^.Next;
    if P^.IsEqual(o) then begin
      Dispose(Iterator, Done);
      Find := P;
      Exit;
    end;
  end;
  Dispose(Iterator, Done);
  Find := Nil;
end;

destructor TIterator.Done;
begin
end;

function TIterator.More: boolean;
begin
  Abstract;
end;

function TIterator.Next: PObject;
begin
  Abstract;
end;

end.
```

Figura 3 - La unit COLLECT.PAS, contenente la definizione sia della classe TCollection che di TIterator, classe base della gerarchia «parallela» degli iteratori.

brica Smalltalk, Tommaso Masi vi propone i diversi tipi di «iteratori» di quel linguaggio; avremo modo di vedere che — pur con le inevitabili differenze — i metodi *ForEach* e *FirstThat* delle collezioni del Turbo Vision presentano una qualche somiglianza con gli iteratori *collect* e *detect* dello Smalltalk. Qui ho preferito mostrarvi una impostazione diversa, quale si trova nell'Objective-C o nella gerarchia di classi del Borland C++: una gerarchia di iteratori più o meno parallela a quella delle collezioni.

La classe base degli iteratori, *TIterator,* non deriva da *TObject* in quanto deve rispondere a due soli messaggi: *More* e *Next.* Il primo risponde TRUE o FALSE secondo che vi sono o no ancora elementi attraverso cui iterare, il secondo ritorna un puntatore all'oggetto «successivo».

Essendo *TIterator* una classe astratta, non viene dichiarato un constructor e il destructor virtuale ha lo stesso scopo di quello di *TObject;* i metodi *More* e *Next,* inoltre, non fanno altro che chiamare la procedura *Abstract.* Ciò non impedisce tuttavia di usare gli iteratori nei metodi di *TCollection*: *IsEqual* percorre con due iteratori la collezione che riceve il messaggio e quella passata come argomento per verificare la loro eguaglianza; *PrintOn* usa un iteratore per mandare lo stesso messaggio, uno dopo l'altro, a tutti gli oggetti della collezione; *Find* percorre con un iteratore la collezione in cerca di un oggetto uguale a quello passato come argomento.

L'implementazione dei metodi può essere vista come un esempio dell'uso degli iteratori, ma soprattutto come un esempio di codice che, pur non essendo usabile con le classi per cui è definito (in quanto classi astratte) può essere ereditato e usato senza modifiche da qualsiasi classe derivata.

Ciò può accadere perché il metodo *InitIterator* di *TCollection* ritorna un puntatore a *TIterator,* come faranno i metodi omonimi delle collezioni derivate.

Grazie al polimorfismo, mandando il messaggio *PrintOn* ad un oggetto istanza della classe *TBag,* per fare un esempio, verrà chiamato il metodo *InitIterator* come ridefinito per questa classe, che ritornerà un puntatore ad un iteratore che sarà istanza della classe *TBagIterator.*

Ricorderete, infatti, che è possibile assegnare ad un puntatore ad una classe anche l'indirizzo dell'istanza di una classe da questa derivata, come abbiamo già fatto con le classi «zoologiche» dei mesi scorsi. Ne vedremo presto altri esempi.

MC