

I rapporti tra oggetti

di Marco Planchestainer

Nella prima parte di questa introduzione alla teoria O-O si sono esaminati i concetti di base, in questa seconda si affronterà con maggiore puntualità il codice Delphi. La struttura di una classe è il primo argomento che verrà esaminato, successivamente ci si concentrerà sui rapporti che classi ed oggetti possono intrattenere tra loro. Questi argomenti sono centrali nella teoria O-O, essi verranno spiegati tenendo presente sia la loro applicazione a programmi Delphi e perciò tenendo conto delle funzionalità di questo linguaggio, sia presentando quei concetti che si ritrovano in tanti scritti su O-O, magari più correttamente applicati ad altri linguaggi. Si parlerà genericamente di rapporti tra oggetti, intendendo con questo anche tutti quei rapporti che prima che tra oggetti vengono definiti a livello di classi (p.es. ereditarietà). In sostanza lo scopo è quello di permettere a coloro che conoscono Delphi di poter sostenere una lettura od una conversazione con programmatori o designer che provengono da più specifiche esperienze O-O (leggi Eiffel, Java, C++). Volutamente breve nel trattare certi concetti, nella speranza che il lettore disponga di almeno una certa infarinatura di Delphi, questo articolo costituisce un complemento alle nozioni di base.

Per iniziare: un esempio di class Delphi

Si prendano ad esempio le due classi TEsempio e TEsempio_Ereditato definite in **listato 1, unit Template**. Tramite esse si esamineranno i diversi tipi di visibilità, il modo di dichiarare l'ereditarietà e le differenti modalità di call dei metodi.

unit Template;

interface

type

```
TEsempio=class(TObject)
  // nessuna visibilità, dati e metodi privati
  private
  // visibilità solo alla classe stessa e classi ereditate
  protected
  // visibilità a tutti
  public
    procedure metodoNormale;
    procedure metodoVirtuale; virtual;
    procedure metodoAstratto; virtual; abstract;
end;
```

```
TEsempioEreditato=class(TEsempio)
  public
    // sostituisce totalmente quello del padre
    procedure metodoNormale;
    // viene deciso a run-time se utilizzare quello
    // del padre o quello del figlio
    procedure metodoVirtuale; override;
```

```
    // il padre è astratto (implementazione rimandata, eseguita // qui)
    procedure metodoAstratto; override;
end;
```

implementation

...

end.

Nella class TEsempio si sono definiti tre metodi, il primo (MetodoNormale) viene richiamato tramite static binding, il secondo (MetodoVirtuale) tramite un binding dinamico (utilizzando una tabella VMT, Virtual Method Table), il terzo (MetodoAstratto) usa sempre un binding dinamico (VMT) ma il metodo non viene implementato in questa classe, la sua implementazione viene posposta alle classi figlie, nel nostro caso l'implementazione sarà presente in TEsempio_Ereditato. Si è ritenuto utile aggiungere al codice la dichiarazione dei principali tipi di visibilità, commentando prima di ognuna il suo significato, in modo che questa classe possa servire come template di riferimento per il programmatore. Lo schema UML delle due classi è molto semplice, **fig.1**:

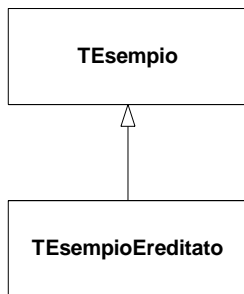


Fig.1 - Ereditarietà

Il rapporto tra le due classi si definisce di ereditarietà e verrà spiegato tra poco. E' importante esaminare comunque non soltanto le classi ma anche gli oggetti che nascono da esse, perché ai rapporti tra classi (si pensi all'ereditarietà) si aggiungono quei rapporti tra oggetti di cui si comprende la natura solo esaminando come essi interagiscano (si pensi alla composizione vs aggregazione che verranno esaminate tra poco). Un esempio di utilizzo delle due classi potrebbe essere:

Var

```
oggetto: TEsempio;  
oggettoesempio: TEsempio;  
oggettoesempioereditato: TesempioEreditato;
```

... creazione oggetti etc. ...

```
oggettoesempio.metodonormale;  
oggettoesempio.metodovirtuale;  
// manda in errore la compilazione: oggettoesempio.metodoastratto;  
oggettoesempioereditato.metodonormale;  
oggettoesempioereditato.metodovirtuale;  
// si può fare perché non è più astratto  
oggettoesempioereditato.metodoastratto;
```

```
oggetto:=oggettoesempioereditato;  
// si può fare perché a run time-viene richiamato il metodo  
// implementato dalla classe figlia TesempioEreditato  
// in quanto il metodo viene trovato tramite VMT  
// (è un metodo virtuale!)  
oggetto.metodoastratto;
```

E' interessante osservare come la definizione di una classe abbia profonde ripercussioni sull'utilizzo che si può fare degli oggetti (che sono sue istanze). Una delle difficoltà della programmazione O-O consiste proprio nella necessità di prevedere l'utilizzo che si farà degli

oggetti, in quanto prima occorre ideare le classi, a freddo in un certo senso, e solo in seguito utilizzarle gli oggetti.

L'ereditarietà

Nelle class viste precedentemente si è fatto uso della tecnica dell'ereditarietà. Insieme a composizione, aggregazione ed associazione, l'ereditarietà è alla base della teoria O-O. Per una spiegazione di base, si consiglia di scorrere velocemente la User Guide Delphi 4, qui si cerca di affrontare la cosa da un differente punto di vista.

La notazione UML dell'ereditarietà è rappresentata da una freccia che va dalla classe figlia (child) a quella genitore (parent), dove la punta della freccia è costituita da un triangolo (si veda la **fig.1**).

In Delphi:

```
TPenna=class(TObject); // TPenna eredita da TObject
```

A che serve tutto ciò? Tramite questa tecnica è possibile costruire una gerarchia di classi in cui si parte da una classe generica, i cui dati ed operazioni sono comuni a tutte le altre classi e poi si aggiungono e si specializzano dati ed operazioni nelle classi figlie.

Esaminando una buona libreria Delphi (si consiglia la VCL [Dlph4]) si nota come venga costruito tutto un castello di classi, partendo da classi semplici per poi ottenere, per ereditarietà, classi sempre più complesse, complete e potenti.

Dunque una operazione definita in una classe padre può essere disponibile in una classe figlia, di contro una classe figlia avrà alcune operazioni che valgono per essa ma non per la classe genitore. Come usare l'ereditarietà? Per esempio si supponga che una software house abbia la necessità di creare degli oggetti specializzati nel controllare/compilare e spedire ordini di materia prima. Un oggetto del genere dovrà avere un metodo per spedire un messaggio, per calcolare l'importo dell'ordine, per eseguire un controllo degli importi.

Definire da subito una classe che possa generare oggetti di questo tipo risolve il problema ma non lo struttura bene ed il lavoro fatto sarà difficilmente riutilizzabile. Più utile un approccio di questo tipo: creare una classe TRichiesta che abbia come metodi una procedura per spedire un messaggio ed una per firmarlo, poi da essa creare una classe figlia TOrdine che abbia in più i metodi per calcolare l'importo e per verificare i dati inseriti.

In tal modo, quando occorrerà creare degli oggetti di tipo Preventivo si potrà sfruttare la classe TRichiesta e da essa ereditare i metodi per spedire un preventivo. Con una struttura del genere il lavoro fatto per un oggetto si può sfruttare per costruirne altri. Si utilizzerà più avanti questo esempio che qui è stato solo accennato, per il momento è sufficiente comprendere come la tecnica dell'ereditarietà sia potente ed utilizzabile in molti modi differenti e variamente ingegnosi.

Nelle classi figlie è possibile accedere alle parti pubbliche, protette e *published* della classe padre, ma non a dati o metodi dichiarati privati (che, si faccia attenzione, sono sempre presenti ma accessibili solo a metodi della classe genitore).

Il **listato 1**, riportato sopra, mostra come si realizza in Delphi un rapporto di ereditarietà tra due classi (in sostanza tramite l'utilizzo della definizione TClasseOggettiFigli = class(TClasseOggettiPadri)). Per ora ci si accontenti di queste brevi note sull'ereditarietà, perché col tempo e con maggiori esempi tutto diventerà molto chiaro, l'ereditarietà è in fondo un concetto molto semplice.

Coloro che dispongono della documentazione originale di Delphi possiedono un bel poster della VCL e potranno osservare come tale libreria sia basata sul rapporto di ereditarietà delle classi. Chi però dispone del codice sorgente della VCL, fornito con la versione Client/Server, osservi come oltre all'ereditarietà sia ugualmente e forse anche più presente un'altra tecnica, che si chiama *Composizione* e che si esamina di seguito.

La composizione, l'aggregazione e l'associazione

Il secondo fondamentale meccanismo O-O è costituito dalla composizione di oggetti e dalle sue varianti, l'aggregazione e l'associazione.

Nella composizione quello che si fa è di assemblare oggetti, dunque si ha un rapporto tra oggetti e non tra classi come nell'ereditarietà.

```

In Delphi:
TPenna=class(TObject)
Public
    Refill: TRefill;
    Clip: TClip;
    ...
end;

```

Nell'esempio si è immaginato che una penna sia composta, tra l'altro, di un refill di inchiostro e che abbia una clip. Un oggetto *Penna* di tipo *TPenna* può accedere a dati e metodi dei due oggetti che lo compongono. Esistono due modi per realizzare questa relazione di "parte di" (*part-of relationship*), nel primo compongo la penna utilizzando il suo refill e la sua clip in modo che, quando creo o distruggo l'oggetto penna anche i due oggetti che lo compongono vengono creati o distrutti. Nell'altro modo una penna è sempre composta dagli stessi oggetti ma essi esistono in maniera non strettamente legata ad essa, per cui posso distruggere l'oggetto penna, senza però distruggere anche il refill e la clip (è come se smontassi la penna, distruggendola, ma salvandone i due pezzi: la clip ed il refill). A livello implementativo ciò si traduce nel fatto che quando distruggo un oggetto *Penna*, in un caso il codice Delphi deve automaticamente distruggere anche i due oggetti che compongono la penna, nell'altro basta che si faccia un *Destroy* del solo oggetto *Penna*, lasciando in vita i due sotto-oggetti che la compongono.

Su di un piano più teorico e formale, si deve distinguere allora tra:

- aggregazione (*aggregation*, diverso lifetime), *part-of* più debole,
- composizione (*composition*, medesimo lifetime), *part-of* più forte,
- associazione (*association*, conoscenza tra oggetti).

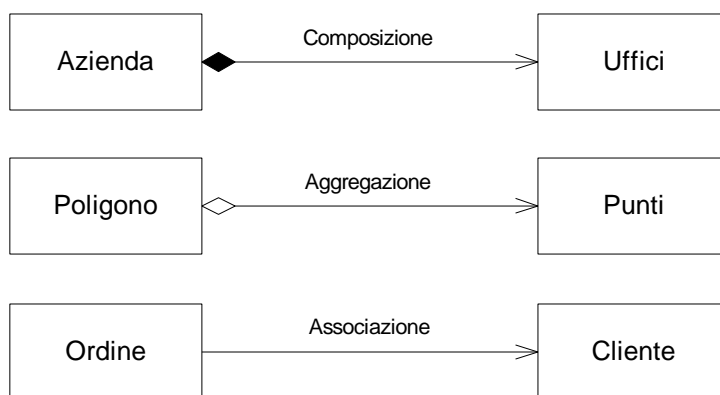


Fig.2 – Notazione UML 1.1, Composizione, Aggregazione ed Associazione

A proposito di notazione, si ricorda che diversi testi utilizzano differenti parole per definire le tipologie della composizione: qui si è scelta la notazione UML [UML]. Nel caso del rapporto *part-of* i differenti modi di chiamare dei concetti di base semplici ha reso l'argomento più ostico del necessario, qui si cercherà di spiegare ciò che conta, lasciando un po' da parte la formale tassonomia che si ritrova in molti testi.

La relazione *part-of* di oggetti più stretta è costituita dalla composizione (*composition*), in essa un oggetto ha definito come suo dato membro (*data member*) un altro oggetto, ed il periodo di vita (*lifetime*) di esso è lo stesso di quello dell'oggetto che lo contiene. Tramite questa tecnica è possibile costruire componendo più oggetti, l'uno contenuto nell'altro, cioè fare in modo che un oggetto posseda diversi altri oggetti.

La tecnica di codifica di una *composition* è quella che si ritrova per esempio nel codice Delphi creato per una *Form*:

```

type
  TFormPrincipale = class(TForm)
    Titolo: TLabel;
    Bottone: TButton;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

```

In questo caso la classe TFormPrincipale contiene due altri oggetti, Titolo e Bottone. Questi sono creati alla creazione della form e distrutti quando la form viene distrutta, perciò hanno lo stesso lifetime dell'oggetto di tipo TFormPrincipale che li contiene.

Il rapporto è di contenimento, un oggetto contiene letteralmente l'altro, non si limita a possedere un riferimento (un puntatore) all'altro ma dispone proprio dello spazio per contenerlo **fig.3, caso A**.

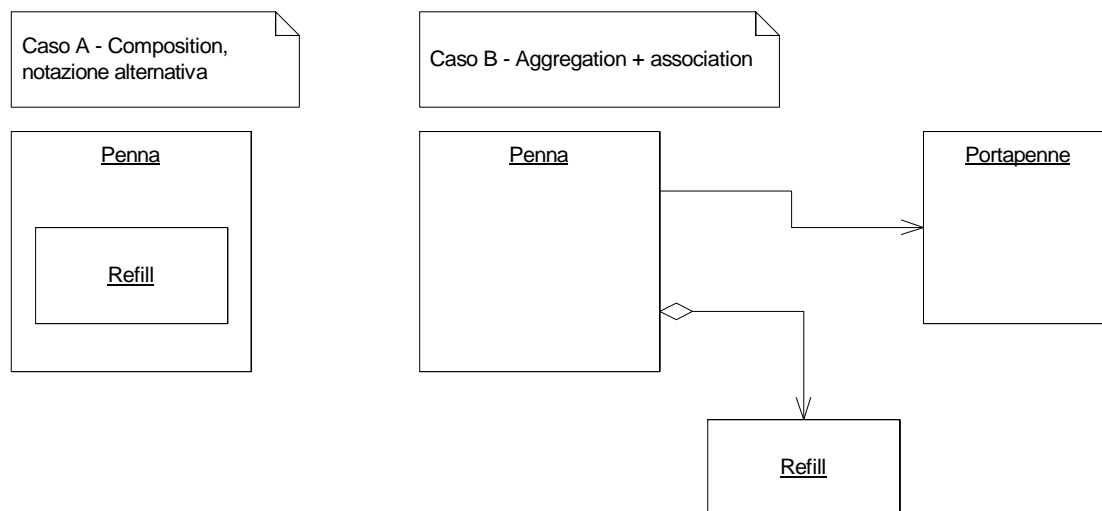


Fig.3 – Composition vs Aggregation

Il secondo metodo è definito aggregazione (*aggregation*), in questo caso un oggetto non contiene l'altro, ma si limita a disporre di un riferimento (reference, spesso implementato con un puntatore) ad esso. Il lifetime dei due oggetti è, generalmente, diverso: non necessariamente vengono creati e/o distrutti assieme. Per un esempio pratico si veda il **caso B, fig.3**.

Rimane da esaminare la Associazione (*association*): con essa si indica una relazione tra istanze di classi, p.es. il fatto che una persona lavori per una ditta, ovvero che una lampada è collegata ad una presa di corrente.

La differenza tra associazione e composizione risiede nel fatto che mentre la composizione esprime il concetto che un oggetto è il risultato dell'assemblaggio di altri oggetti, l'associazione si limita a fotografare i rapporti che esistono tra oggetti che devono 'conoscersi', ma non sono l'uno parte dell'altro. Da un punto di vista implementativo anche l'associazione si realizza tramite reference (volgarmente e semplificando un po': puntatori).

A livello architetturale è possibile, con una certa disinvoltura, utilizzare quasi sempre il rapporto di associazione, a detrimento di aggregazione e composizione: esso è il rapporto più generico, che può sostituire in fase di studio/analisi gli altri due, che poi ricompariranno mano mano che si definiscono i dettagli dell'analisi e del design. In **fig.2** si vede che un esempio tipico di associazione è quello tra classe ordine e classe cliente, dove ogni ordine possiede una reference all'oggetto cliente a cui si riferisce: questa è una associazione che indica, tramite la freccia, che esiste una navigabilità da ordine a cliente (dove mancasse la freccia si intendeva

associazione con navigabilità non specificata). L'implementazione di una associazione in Delphi è, come facilmente prevedibile, effettuata tramite l'utilizzo di un reference all'oggetto puntato dalla freccia, nel caso in questione:

```
TOrdine=class
  Protected Cliente: TCliente;
End;
```

Dunque le differenze tra le varie notazioni non appaiono tanto nel codice sorgente Delphi, che fondamentalmente è lo stesso, ma nel modello di sistema che si costruisce. Il fatto di sapere cosa si vuol realizzare, associazione, composizione od aggregazione, permetterà in fase di design e programmazione di effettuare le scelte più appropriate.

Si badi che quelle che possono sembrare sottigliezze di programmazione o inutili distinguo, non lo sono affatto. Il modello è importante, su di esso ci si basa: la realtà ovvero il modello di essa che si vuole implementare in un programma è fatta proprio a questo modo, ci sono oggetti che ne contengono altri in tutto e per tutto, ed oggetti che possiedono solo un riferimento ad altri.

E' interessante notare che dal punto di vista della dichiarazione o interfaccia della classe il codice Delphi può essere, praticamente, il medesimo. Ovviamente per le parti di creazione e distruzione dell'oggetto (implementazione della classe) esso sarà sicuramente diverso. Quello che una classe possiede dichiarando una *aggregation* non è un oggetto, ma una reference ad un oggetto: in realtà ogni oggetto che è composto di altri oggetti possiede dei reference ad essi (dei puntatori), tuttavia nel caso della *aggregation* esiste il riferimento ma non la gestione di creazione e distruzione, nel caso della *composition* le operazioni di creazione e distruzione vengono gestite dall'oggetto principale.

Un altro esempio di *association* potrebbe essere costituito da una classe TUtente, utilizzata per gestire le caratteristiche dell'utente di un programma, che possiede un riferimento ad una finestra in cui si inserisce ID e Password, per la convalida dell'identità. Dopo che la persona ha inserito i suoi dati, la finestra è inutile e può essere distrutta, tuttavia l'oggetto Utente ha ancora una sua ragione di esistere (per gestire lungo tutta la durata dell'esecuzione del programma i diritti di accesso per utente). In questo caso la relazione non è di *part-of*, la finestra non fa parte dell'oggetto Utente, al contrario è un oggetto indipendente a cui Utente accede per ottenerne dei servizi.

Rimane da dire che il confine tra *aggregation* e *composition* è abbastanza labile in un linguaggio come Delphi, ed essendo le due tecniche strutturalmente molto simili, si parlerà ove possibile di composizione senza specificarne il tipo. Per inciso è bene chiarire che altri linguaggi (p.es. Eiffel) possiedono semantiche più potenti, che permettono di distinguere con chiarezza tra aggregazione e composizione, nonché di utilizzare operatori specifici nei diversi casi. Questo per far capire come concetti generali, che dovrebbero essere validi per tutti i linguaggi, vengano implementati in maniera diversa e con livelli di completezza più o meno estesi: nel caso di Delphi si può parlare di sufficiente potenza, di un giusto livello di innovazione, compatibilità con il passato e semplicità.

In **fig.4** si rappresenta un oggetto in cui è presente un rapporto di composizione ed un altro di aggregazione.

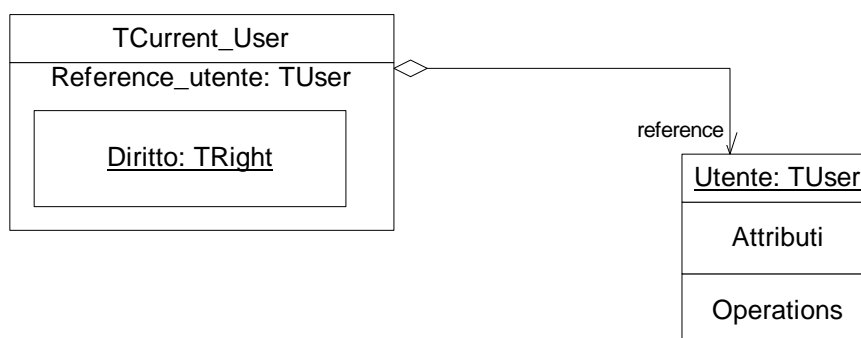


Fig. 4 – Notazioni e significati differenti per Composizione ed Aggregazione

Nel caso dell'accesso ad Utente, l'oggetto di tipo TCurrent_User sfrutta una reference ad un oggetto esistente al di fuori di esso, nel caso del reperimento di dati e metodi riguardanti i diritti dell'utente corrente, si accede ad un sub-oggetto Diritto, compreso entro l'oggetto di tipo TCurrent_User.

La class Delphi potrebbe essere:

```
TCurrent_User=class(TObject)
Public
  Reference_utente: TUser;
  Diritto: TRight;
  ...
end;
```

Si ricorda che un'associazione viene rappresentata come una linea che collega due classi in un diagramma UML. In un prossimo articolo si approfondiranno diversi aspetti di UML, in maniera da spiegare alcune notazioni che qui si possono solo presentare.

Non si insisterà mai troppo sull'importanza della composizione vs l'ereditarietà.

E' bene imparare ad utilizzare entrambe, mescolandone bene i punti di forza ed amalgamando nel proprio design la giusta dose di questo e di quello, poco di questo e molto di quello ("Chorus Line"). In sostanza coloro che programmano ad oggetti tendono ad avere conoscenza e senso dell'esistenza di uno strumento (l'ereditarietà), mentre spesso dimenticano o non riconoscono l'altro (la composizione). E' come se un progettista meccanico utilizzasse come elemento di fissaggio solo chiodi, dimenticando l'esistenza e l'utilità di viti e bulloni.

I Design Patterns

Nelle bibliografie degli articoli viene spesso citato un libro: Gamma *et al.* - Design Patterns - Elements of reusable Object-Oriented Software, Addison-Wesley, 1995. Gli autori sono Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, e sono anche detti "The Gang of Four" ("La banda dei quattro", da cui l'acronimo "GoF" che ricorre spesso come richiamo bibliografico). La pubblicazione di questo libro ha suscitato un grande, profondo interesse sia per il valore della teoria ivi esposta e sia per la precisa, penetrante qualità del testo.

Gli argomenti esposti riguardano aspetti fondamentali della programmazione e del design O-O, il problema per coloro che conoscono Delphi è che tutti gli esempi sono scritti in C++, e non tutti riescono a seguire od almeno leggere tale linguaggio.

Con queste premesse si spiega il perché di ProjectOO (<http://members.tripod.com/mplank>), un'iniziativa volta a colmare una lacuna nella formazione dei programmatori Delphi. Una parte del sito infatti tratta in generale la teoria dei patterns di design ed in particolare alcune implementazioni ed esempi in Delphi, in modo che anche coloro che non possono leggere direttamente il libro riescano almeno ad avere chiari i concetti fondamentali che vi sono esposti.

Conclusioni

Sono stati esaminati i concetti di:

- Ereditarietà: rapporto padre/figlio tra oggetti
- Composizione: creazione di oggetti usando altri oggetti, *part-of*, (medesimo lifetime)
- Aggregazione: creazione di oggetti usando altri oggetti, *part-of*, (diverso lifetime)
- Associazione: relazione di conoscenza tra oggetti, *reference*

Bibliografia

- | | |
|---------|---|
| [Dlph4] | Vari - Manuali Delphi 4 - Borland, 1998 |
| [GoF95] | Gamma <i>et al.</i> - Design Patterns - Addison-Wesley, 1995 |
| [UML] | Rational <i>et al.</i> - UML Notation Guide - www.rational.com , 1997 |
| [Fow97] | Fowler - UML distilled - Addison-Wesley, 1997 |
| [Mey97] | B.Meyer - "Object Oriented software construction", Prentice-Hall, 1997 |

Marco Planchestainer è un ingegnere elettronico che lavora come capo progetto in un'azienda della provincia di Vicenza. Si occupa di aspetti sia tecnici che organizzativi, soprattutto riguardanti le problematiche gestionali della produzione industriale. Si può contattare all'indirizzo <m.plank@usa.net>.