

La struttura di un oggetto

di Marco Planchestainer

In questa serie di articoli si vuol percorrere una via che porti da una conoscenza basilare di Delphi ad una più profonda consapevolezza della teoria Object Oriented. Si parlerà di programmazione, di teoria, di design e di analisi. Gli argomenti verranno affrontati sempre con l'approccio più aggiornato che è stato possibile reperire, in modo da fornire quegli strumenti che un buon programmatore dovrebbe avere.

Spesso ci si avvicina a questo mondo partendo dal manuale di un qualche linguaggio, C++ o Delphi che sia e solitamente si rimane con l'impressione che non tutto sia stato chiarito. Questo deriva dal fatto che i manuali devono spiegare un linguaggio specifico e non una teoria generica: qui si esaminerà invece il generico e generale che è alla base di tutti i linguaggi O-O. Con ciò non si vuol dare l'impressione che quanto segue si possa considerare esaustivo in materia di O-O, poiché l'ambito è in realtà così vasto che sarebbe impossibile riassumerne i punti in una serie di articoli. In realtà e semplicemente l'obiettivo è quello di presentare alcuni aspetti fondamentali con un approccio diverso, che permetta di ampliare un po' il proprio punto di vista.

Introduzione

L'approccio che si seguirà in queste pagine è basato su una visione 'semplificata', ma non semplicistica, della teoria O-O, in contrapposizione ad un atteggiamento più teorico: chi fosse interessato ad approfondire questa parte potrà farlo, p.es., su Meyer [2]. Si presuppone una conoscenza di base di Delphi, senza però richiedere molto di più di una buona infarinatura.

Si parta dalla struttura di base di ogni oggetto O-O. Un oggetto è costituito da operazioni e dati. Per *dati* si intende un insieme di strutture capaci di memorizzare dei valori numerici, alfabetici od anche di tipo più complesso come stringhe od altri oggetti.

Per *operazioni* si intendono le azioni che tale oggetto può compiere. In alcuni casi la letteratura definisce tali operazioni dei "messaggi": l'uso di tale termine deriva dal fatto che una operazione viene considerata un messaggio che l'utilizzatore dell'oggetto (il *client*) manda all'oggetto perché esso esegua una particolare azione.

Si rivela subito utile per comprendere e visualizzare mentalmente un oggetto O-O introdurre una sua rappresentazione grafica. Un oggetto può essere disegnato con diverse notazioni, una di queste è descritta tramite uno standard definito Unified Modeling Language (UML) [1]. In questo articolo si utilizzerà UML 1.1 poiché esso è considerato il più moderno mezzo per descrivere oggetti software.

Un semplice oggetto chiamato *p1* di tipo *Punto* si rappresenta come in **fig.1**.



fig.1

Questo oggetto rappresenta un punto del piano cartesiano. Si vede chiaramente la divisione dell'oggetto in parte dati (*x,y*) ed operazioni (*disegna*, *distanza_dal_centro*), con in aggiunta un nome per chiamarlo (*p1*) ed il tipo di oggetto a cui appartiene (*Punto*).

Esaminando la sua rappresentazione dunque si nota che il suo nome è *p1* ed appartiene alla *class* Punto. Per una descrizione breve ma sufficientemente completa di UML 1.1 si rimanda al terzo articolo di questa serie. Si procede ora con il concetto, appena introdotto, di classe.

Class

La *class* è il tipo dell'oggetto; quando si programma ad oggetti normalmente non si utilizza solo un oggetto, ma molti dello stesso tipo, cioè con la stessa struttura. In Delphi (ma genericamente in molti altri linguaggi) si è introdotto il concetto di *class* per consentire di descrivere genericamente un oggetto; chi utilizza gli oggetti li creerà partendo da questo 'stampo' generico che è la *class*.

La *class* è una estensione del concetto di tipo dato astratto (ADT ovvero Abstract Data Type). Per coloro che hanno una formazione matematica la *class* è il dominio di definizione degli oggetti di uno stesso tipo (l'oggetto *O* appartiene al dominio *D* dove $D = \text{class}(\text{Tipo_Oggetto_O})$).

Tornando allo schema di **fig. 1**, si nota che l'oggetto *p1* ha due dati *x* ed *y*, entrambi di tipo real (sono real p.es.: 0.12 , 145.5 e 3).

Il simbolo '+' che li precede indica (UML) che questi dati sono di *visibilità pubblica*, si parlerà della visibilità in maniera più dettagliata, per ora basti sapere che la visibilità pubblica implica che i due dati sono leggibili e scrivibili direttamente dai *client* dell'oggetto (per *client* si intendono i programmi che useranno tale oggetto).

La sottolineatura del nome oggetto e del nome classe è una regola, arbitraria, dettata da UML. L'oggetto *p1* ha anche due operazioni, *disegna* e *distanza_dal_centro*, entrambe pubbliche: la prima probabilmente (nulla si evince dalla rappresentazione grafica) disegnerà il punto sul piano cartesiano, la seconda restituirà un numero uguale alla distanza dal centro assi del punto, ricavata p.es. con la formula di Pitagora:

$$\sqrt{x^2 + y^2}$$

Dunque abbiamo un oggetto composto di una parte dati e di due operazioni che esso può compiere: in questo modo si è creato qualcosa che può essere utilizzato per memorizzare dei valori (*x* ed *y*) e per compiere delle azioni (disegnare e calcolare una distanza). In fondo un oggetto software non è poi molto più complicato di questo.

L'implementazione degli oggetti

La trattazione di come vengano effettivamente rappresentati gli oggetti all'interno di un programma (nella RAM del computer) è solitamente relegata a testi sulla teoria dei compilatori, eppure conoscere come funzionino effettivamente un programma O-O chiarisce spesso concetti che sarebbe difficoltoso spiegare altrimenti.

Ogni singolo oggetto ha una sua parte dati che è rappresentata in maniera distinta, mentre le operazioni (*methods* o metodi) sono costituite da parti di codice eseguibile uniche per ogni classe, condivise tra tutti gli oggetti. La **fig.2** dovrebbe aiutare a capire il concetto.

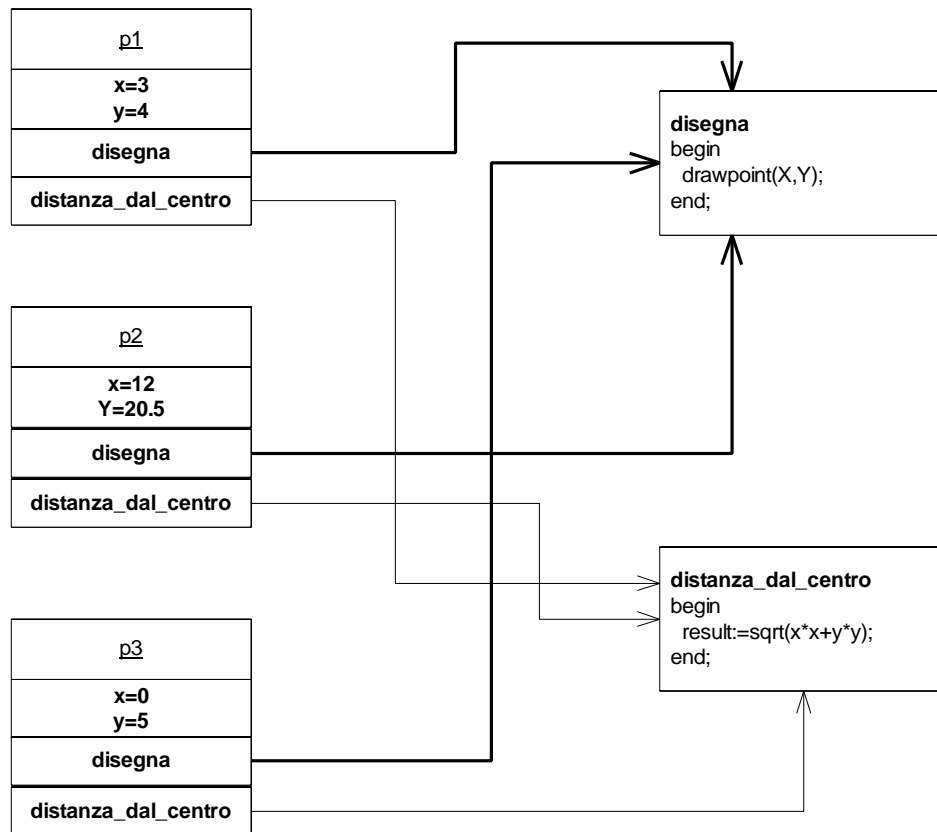


fig.2

Si vede come ogni oggetto in memoria sia rappresentato singolarmente per la parte dati ma in maniera condivisa per quanto riguarda i suoi *metodi*. Le frecce rappresentano dei puntatori (*reference*) ai metodi.

Semplificando un po', i metodi dunque sono una serie di istruzioni in codice macchina che eseguono fisicamente le operazioni su dati propri di ogni oggetto. Si ha una sola serie di implementazioni fisiche delle operazioni, sempre le stesse per ogni oggetto della stessa *class*, mentre molte istanze dei dati, una per ogni singolo oggetto creato.

Quando un client vuole dire all'oggetto *p1* di eseguire l'operazione di calcolo della distanza dal centro assi, allora basta che esso mandi il *messaggio* : "esegui il metodo *distanza_dal_centro*" all'oggetto *p1*.

Come si manda questo messaggio? in genere con una sintassi del tipo:

```
p1.distanza_dal_centro
```

dove il 'punto' viene usato per legare oggetto (*p1*) con operazione (*distanza_dal_centro*). Si possono mandare tali messaggi ad un qualsiasi oggetto del tipo Punto, p.es.

```
DistanzaP2:=p2.distanza_dal_centro;
DistanzaP1:=p1.distanza_dal_centro;
if DistanzaP1<DistanzaP2 then . . .
```

In effetti si può pensare alla invocazione di un metodo (al messaggio) come se si stesse eseguendo una funzione, ed essa venisse invocata con parametri o variabili globali diverse a seconda dell'oggetto che la chiama.

Interfaccia e tipo di un oggetto

Parlando di oggetti vengono spesso citati i concetti di *interfaccia* e di *tipo*, per esempio tutta la trattazione della programmazione COM parte dalla spiegazione della keyword *interface*.

Si può dare la seguente definizione: l'interfaccia di un oggetto è definita dall'insieme di tutti i metodi di cui esso dispone.

Il termine *interfaccia* è ben scelto: l'insieme di tutte le operazioni dichiarate da un oggetto è proprio il modo con cui esso si interfaccia con l'esterno, il suo collegamento con ciò che è esterno.

Per esempio si considerino le due seguenti *class* costituite come in **fig.3**:

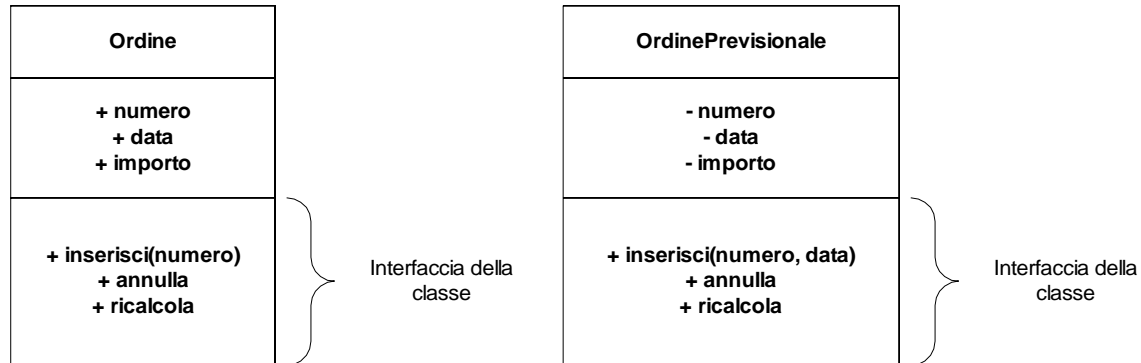


fig.3

In questo caso l'interfaccia della *class* Ordine è costituita dai tre metodi pubblici: *inserisci(numero)*, *annulla*, *ricalcola*.

L'interfaccia è determinata anche dai parametri del metodo, questo significa che un metodo presente con lo stesso nome in due oggetti ma differenti parametri genera due interfacce diverse.

Se si considerano le due *class* Ordine ed OrdinePrevisionale, è immediatamente chiaro che esse hanno due interfacce differenti, in quanto il metodo *inserisci*, presente in entrambe, nella prima viene definito come avente in lista parametri solo il *numero* ordine, nella seconda si aggiunge il parametro *data* (dell'ordine).

Il nome di un'operazione (ovvero il nome di un metodo), i suoi parametri ed il suo valore di ritorno costituiscono la sua firma (*signature*). Un altro modo, più tecnico, di descrivere l'interfaccia di un oggetto è quello di definirla come l'insieme di tutte le sue *signature*.

Il tipo (*type*) di un oggetto indica tutti gli oggetti accomunati da una stessa interfaccia.

Due oggetti con la stessa interfaccia sono dello stesso tipo, p.es. se un oggetto deriva (è figlio, ovvero viene ricavato tramite ereditarietà) da un altro, allora i due oggetti hanno in comune lo stesso tipo (a meno di override).

La visibilità

Ogni dato o metodo di un oggetto ha una proprietà detta visibilità (*scope*) che determina chi possa usare dati e metodi. Le tipologie di visibilità sono strutturate su tre livelli: parti visibili solo all'oggetto stesso, parti visibili a tutti (oggetto e client), parti visibili solo all'oggetto ed a tutti gli oggetti ricavati (*ereditati*, *inherited*) da esso.

La notazione UML è la seguente:

- private
+ public
protected

La notazione Delphi è:

```
Ordine=class()
// dati e metodi ...
Private
// dati e metodi privati
Public
```

```

numero: integer;
data: date;
importo: real;
procedure inserisci(numero: integer);
procedure annulla;
procedure ricalcola;
Protected
  // dati e metodi protetti
End;

```

Delphi – Nota

Published: visibilità come public, vengono però create delle informazioni runtime sul tipo del dato o metodo, in modo che gli strumenti di sviluppo possano interrogare l'oggetto su tali dati o metodi.

Automated: visibilità come public, genera delle informazioni runtime per creare OLE Automation Servers.

A cosa serve specificare la visibilità di un dato? Si pensi al numero ordine della classe *Ordine*, che è stato dichiarato *public*: un client potrà leggere e modificare tale valore dall'oggetto *ordine1* di tipo *Ordine* (UML: ordine1:Ordine) nel seguente modo:

```

// leggo
numero_ordine:=ordine1.numero;
// modifico
ordine1.numero:=numero_ordine+1;

```

Il che costituisce, dal punto di vista del Design del programma, una possibile fonte di errore. Il codice è corretto, funziona a dovere, ma cosa succederà quando qualcuno deciderà di trasformare il campo numero (*field numero*) della table Ordini da numerico ad alfanumerico? Non sarà più possibile incrementare di uno il numero ordine, occorrerà modificare il codice sorgente. In ogni punto dove si incrementa il numero ordine si dovrà cambiare il codice del programma e questo non sarà certo divertente.

Una soluzione possibile è quella di avere una classe Ordine in cui *numero* sia *privato*, perciò non leggibile o modificabile dal client. Si poteva poi aggiungere un metodo pubblico *restituisce_prossimo_numero* che poteva essere implementato in Delphi con il seguente frammento di codice:

```

// parte precededente del metodo ...
restituisce_prossimo_numero:=numero+1;

```

In tal modo tutti i programmi esterni potevano utilizzare tale metodo, ecco che al cambio da numerico ad alfanumerico poteva essere sufficiente modificare il metodo *restituisce_prossimo_numero* e poco altro.

La creazione e la distruzione di un oggetto

Un oggetto deve essere creato e distrutto, questo perché occupa risorse di memoria che non si devono o possono sprecare creandolo prima del tempo o evitando di distruggerlo se non alla fine del programma. In effetti nulla vieterebbe che un linguaggio crei l'oggetto al momento stesso della sua dichiarazione, cioè quando in Pascal p.es. si fa:

```
var oggetto:Tobject;
```

il compilatore potrebbe subito dimensionare un'area di memoria che fornisca lo spazio all'oggetto. Per ragioni di efficienza, diversi linguaggi pretendono che sia il programmatore ad eseguire esplicitamente tale operazione.

Dunque deve esistere un modo per creare l'oggetto, ed ogni linguaggio ha modi differenti per farlo. In Delphi per ogni classe deve esserci un metodo di tipo *constructor* ed uno di tipo

destructor (solitamente chiamati *Create* e *Destroy*) che sono preposti alla costruzione e distruzione fisica dell'oggetto.

Create esegue: allocazione dello spazio in memoria per la parte dati, legame tra oggetto e suoi metodi, eventuali inizializzazioni necessarie.

Destroy esegue: liberazione dello spazio in memoria occupato dall'oggetto.

Nel listato 1 viene mostrato, tra l'altro, un esempio di *Create* e *Destroy*, è interessante notare che questi due metodi devono essere elencati tra quelli di visibilità pubblica, in quanto il client deve poter accedere ad essi.

Tipologie di *call* dei metodi

A che punto del codice eseguibile deve passare l'esecuzione del programma quando viene invocato un metodo? Se esistesse una sola classe (senza ereditarietà) sarebbe sufficiente generare un salto (*jump*) ad un punto fisso del codice dove inizia l'eseguibile del metodo (questo è il comportamento di default di Delphi e del C++). Quando esistono più classi, l'una figlia dell'altra, con un metodo ereditato implementato diversamente nella classe figlia, allora decidere a quale eseguibile passare l'esecuzione è più complesso.

Si consideri un oggetto *p1* della classe *C1* con metodo *m*, invocabile con *p1.m*, ed un oggetto *p2*, anche questo con metodo *m*, dove $C1 \leftarrow C2$ (*C2* è figlia di *C1*).

Si abbia inoltre un oggetto *px*, non istanziato (un puntatore), di classe *C1*, allora se *px:=p1*; *px.m* invoca il metodo *m* della classe *C1*, al contrario se *px:=p2*; *px.m* invoca il metodo *m* di *C2*.

Come fa lo stesso oggetto *px* ad invocare prima il metodo *m* di *C1* e poi il metodo *m* di *C2*? La risposta è *dynamic binding*, ovvero chiamate dinamiche che vengono risolte a run time.

Statiche:

per default Delphi genera delle *static calls* (chiamate statiche) per i metodi dichiarati in una classe. Questo significa che il compilatore determina quale sia il metodo da richiamare in maniera definitiva in fase di compilazione e genera una reference (riferimento o puntatore) al codice da eseguire per ciascun metodo statico. Il significato di tutto ciò si chiarisce se spiegato all'interno dei concetti di ereditarietà, per ora basti sapere che essendo l'indirizzo di un metodo statico fissato in fase di compilazione, ogni volta che manda un messaggio all'oggetto chiedendo l'esecuzione del metodo, viene sempre eseguito lo stesso codice anche se talvolta verrebbe naturale aspettarsi che venga eseguito il codice di un metodo appartenente ad una *class* ereditata.

Virtuali (*virtual*):

Un metodo si dice *virtuale* (*virtual*) quando il tipo di classe od oggetto coinvolto nella *call* determina quale implementazione del metodo utilizzare. Definendo un metodo *virtual* il compilatore genera una tabella dove scrive l'indirizzo da utilizzare per richiamare il metodo, in esecuzione il programma accede a tale tabella usando come indice il tipo di oggetto coinvolto nella *call*, in modo da identificare il metodo da eseguire. A cosa serve tutto ciò? In effetti questo modo di invocare un metodo è fondamentale per la teoria OO, in quanto in tal maniera è possibile utilizzare completamente il meccanismo dell'ereditarietà. In Delphi è necessario, per sfruttare questo meccanismo di *call*, dichiarare di tipo *override* ogni metodo figlio interessato.

Dinamiche (*dynamic*):

I metodi dinamici sono funzionalmente identici a quelli virtuali, differiscono solo nel meccanismo di generazione del codice in quanto il compilatore utilizza ... generando codice più compatto ma meno veloce rispetto a quanto possibile con i metodi virtuali.

In linea di massima è meglio lavorare con metodi virtuali

Astratte (*abstract*):

Un metodo astratto deve sempre essere dichiarato anche virtuale (o dinamico) e si utilizza per determinare quei metodi la cui implementazione non viene realizzata nella class che lo dichiara ma in class da essa derivate per ereditarietà. Nella classe derivata comparirà perciò lo stesso metodo dichiarato come *override*, ed esso sarà lì implementato (badando di non utilizzare la tecnica dell'ereditarietà in quanto non potrebbe ovviamente funzionare).

Finora si è parlato di oggetti in maniera abbastanza generica, è il momento di affrontare il codice Delphi, ma per questo si rimanda alla parte seconda: "I rapporti tra oggetti".

Conclusioni

Sono stati esaminati i concetti di:

- Class : concetto di classe dell'oggetto
- UML : rappresentazione grafica per OOP e OOD
- Method : operazione eseguita da un oggetto
- Message : vedi method
- Object implementation : rappresentazione in memoria di un oggetto
- Interface : insieme dei metodi di un oggetto
- Type : insieme di equivalenza definito tramite l'uguaglianza delle interfacce
- Signature : firma di un metodo
- Scope : visibilità di metodi e dati
- Public : visibilità totale
- Private : visibilità solo dall'interno dell'oggetto
- Protected : visibilità solo all'oggetto e suoi derivati
- Constructor e destructor: tipi di procedure speciali per creare e distruggere oggetti
- Static call : chiamata di default, fissa per uno stesso oggetto
- Virtual call : chiamata tramite tabella, dipende dall'oggetto in questione
- Dynamic call : come virtual
- Abstract call : virtual e non implementata, segnaposto per classi ereditate

Bibliografia

- [UML11] Rational - "UML 1.1", Rational, 1997
- [Mey97] B.Meyer - "Object Oriented software construction", Prentice-Hall, 1997
- [GoF95] E.Gamma - "Design patterns", Prentice-Hall, 1995
- [TiJ98] B.Eckel - "Thinking in Java", Prentice-Hall, 1998
- [DLH3] Vari - Manuali Delphi 3, Borland, 1997
- [Mul] Peter Müller - Introduction to Object-Oriented Programming Using C++
Globewide Network Academy (GNA) www.gnacademy.org/

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>.