Introduzione alla programmazione object oriented

Concetti di base

Che cosa sono gli oggetti

Un "oggetto" e' costituito da un insieme di strutture dati e da un insieme di operazioni che agiscono su queste strutture.

Una caratteristica fondamentale degli oggetti e' che essi possono provvedere una interfaccia uniforme alle varie componenti di un sistema: tutte le interazioni con un oggetto possono avvenire attraverso l'esecuzione di una delle operazioni (*metodi*, usando la terminologia Java) dell'oggetto.

L'idea fondamentale per sfruttare questa caratteristica degli oggetti e' quella di rendere un oggetto responsabile di un insieme di compiti collegati tra di loro. Se un oggetto deve poter richiedere l'esecuzione di un compito che non e' sotto la sua responsabilita', allora deve poter accedere ad un oggetto tra le cui responsabilita' vi e' quel compito. Il primo oggetto chiedera' al secondo oggetto di portare a termine quel compito invocando l'esecuzione di un metodo del secondo oggetto.

Tutte le comunicazioni tra oggetti dovrebbero avvenire attraverso invocazioni di metodi, ovvero: un oggetto non dovrebbe mai poter manipolare direttamente i dati interni di un altro oggetto.

L'uso degli oggetti permette quindi un disegno modulare del software: ogni componente puo' essere vista come un oggetto, e quindi realizzata provvedendo una interfaccia (l'insieme dei metodi dell'oggetto) al resto del sistema e nascondendo i dettagli di implementazione (le strutture dati e il codice dei metodi). In questo modo tutte le modifiche ad un oggetto che non comportano modifiche all'interfaccia non richiedono di modificare gli

altri oggetti del sistema.

I meccanismi di base dei linguaggi di programmazione object oriented

I linguaggi di programmazione object-oriented supportano la definizione e l'uso di oggetti. In particolare, i moderni linguaggi di programmazione object-oriented provvedono i seguenti meccanismi di base.

- *Aggregazione*: un oggetto e' una entita' che combina dati e operazioni.
- *Incapsulamento*: alcuni dati e metodi associati ad un oggetto potrebbero essere pubblici (accessibili da chi usa un oggetto), mentre altri saranno privati (accessibili solo all'interno dell'oggetto). Di norma i dati dovrebbero essere sempre privati, e ogni interazione con un oggetto dovrebbe avvenire attraverso l'uso di uno dei metodi dell'oggetto.
- *Subtyping e sostitutivita'*: a prescindere da come siano definiti i tipi degli oggetti il principio di sostitutivita' associato alla nozione di subtyping gioca un ruolo centrale nei linguaggi object oriented. Il tipo C' e' un sottotipo del tipo C se ogni operazione su oggetti di tipo C e anche una operazione su oggetti di tipo C'. Quindi se C' e' un sottotipo di C, allora un oggetto di tipo C' puo' essere usato al posto di un oggetto di tipo C in qualunque contesto.
- *Ereditarieta'*: l'implementazione di un oggetto puo' essere definita come la modifica incrementale dell'implementazione di un altro oggetto.

La metodologia di progetto object-oriented

La locuzione "object orientation", oltre ad indicare una caratteristica di linguaggi di programmazione e sistemi di gestione della basi di dati, indica anche una metodologia di progetto che ha lo scopo di modellare il software e di disciplinarne lo sviluppo in modo da facilicitare la costruzione di sistemi complessi a partire da componenti individuali. Tale metodologia puo' essere riassunta in quattro passi.

1. Identificare gli oggetti ad un dato livello di astrazione.
2. Indentificare quali sono le operazioni associate a questi oggetti.
3. Identificare le relazioni rilevanti che intercorrono tra gli oggetti.
4. Implementare questi oggetti.

Si tratta di un procedimento iterativo basato sull'associare oggetti con sottocomponenti o concetti in un dato sistema. Il procedimento e' iterativo perche', in genere, un oggetto e' implementato usando un certo numero di "sotto-oggetti" (analogamente a quello che avviene con l'approccio top-down alla programmazione, dove una procedura e' implementata usando un certo numero di "sotto-procedure").

Data la descrizione di un problema, una semplice regola (che chiameremo "regola dei nomi e dei verbi") per identificare gli oggetti e' quella di considerare i **nomi** che occorrono nella descrizione. Le operazioni sugli oggetti, invece, corrisponderanno ai *verbi*.

Naturalmente si tratta di una regola approssimativa: solo l'esperienza potra' aiutarci a decidere quali nomi e quali verbi saranno importanti quando, di volta in volta, dovremo individuare degli oggetti nell'ambito di un particolare problema.

Linguaggi "class-based" e linguaggi "delegation-based"

Possiamo distinguere due diversi paradigmi per linguaggi OO: il paradigma *class-based* e il paradigma *delegation-based*.

- *Class-based*. Nei linguaggi class-based gli oggetti sono descritti per mezzo di "classi". Una classe descrive l'implementazione di un oggetto, e un oggetto e' creato "instanziando" la sua classe.

- *Delegation-based*. Nei linguaggi delegation-based, invece, gli oggetti sono definiti direttamente da altri oggetti aggiungendo nuovi metodi e rimpiazzando metodi gia' presenti.

Il paradigma class-based e' il piu' affermato. L'interesse per il paradigma delegation-based e' principalmente motivato dalla sua semplicita' che lo rende appetibile per particolari applicazioni.

Java e' un linguaggio class-based. Altri linguaggi class-based: C++ e Smalltalk. Un linguaggio delegation based: JavaScript.

I meccanismi di base visti attraverso il linguaggio Java

Prima di introdurre in modo sistematico i vari costrutti del linguaggio Java, illustreremo, attraverso semplici esempi, alcune delle caratteristiche fondamentali della programmazione OO class-based che abbiamo menzionato in precedenza.

Supponiamo di dover scrivere un programma che comporti l'uso **finestre**, in particolare debba poter *dimensionare*, *mostrare*, *nascondere*, *spostare*,... delle finestre (si ricordi la "regola dei nomi e dei verbi"). Possiamo allora pensare di rendere un oggetto responsabile della gestione di una finestra. E in Java e' possibile descrivere gli oggetti di questo tipo definendo una classe.

```
class Window
{ private int x,y,width,height;

  public void setSize (int w, int h)
  { width=w;
    height=h;
  }

  public void show ()
  { <<visualizza sullo schermo la finestra>>
  }

  public void hide ()
  { <<nascondi (rendi invisibile) la finesta>>
  }

  public void move (int dx, int dy)
  { hide();
    x+=dx;
    y+=dy;
    show();
  }
  ...
}
```

Nel codice che definisce la classe `Window` possiamo facilmente riconoscere i meccanismi fondamentali di *aggregazione* (la classe descrive sia dati che metodi) e *incapsulamento* (la keyword `private` specifica che i campi dati non sono accessibili al di fuori della classe).

E' quindi possibile creare ed utilizare oggetti della classe `Window`. Ad esempio la sequenza di istruzioni

```
Window win;              // (1)

win = new Window();      // (2)
win.setSize(200,100);    // (3)
win.show();              // (4)
```

(1) dichiara una variabile `win` che puo' contenere un riferimento (sostanzialmente un "puntatore") ad un oggetto della classe `Window`,
(2) crea un nuovo oggetto della classe `Window` (cioe' alloca uno spazio di memoria per contenere i campi dato specificati nella classe `Window`) e memorizza nella variabile `win` un riferimento a tale oggetto,
(3) invoca il metodo `setSize` sull'oggetto il cui riferimento e' contenuto nella variabile `win`,
(4) mostra l'oggetto riferito dalla variabile `win` sullo schermo.

Si osservi che, durante l'esecuzione di un metodo `m` su di un oggetto `o` di una data classe `C`, ogni invocazione di un metodo definito nella classe, e' da interpretare come una invocazione del metodo sull'oggetto `o` stesso, il cui riferimento e' contenuto nella variabile riservata `this`. Di fatto e' come se, ad esempio, il codice per il metodo `move` fosse il seguente:

```
public void move (int dx, int dy)
{ this.hide();
  this.x+=dx;
  this.y+=dy;
  this.show();
}
```

Anzi, in Java e' possibile scrivere il metodo `move` in questo modo, ma, per comodita', e' anche possibile omettere tutti questi `this`.

Supponiamo ora di aver bisogno di finestre dotate di titolo e di bordo. Invece di metterci a scrivere ex-novo una classe opportuna, potremmo pensare di estendere la classe `Window` modificando alcune funzionalita' e aggiungendone di nuove. Naturalmente vogliamo che la classe `Window` di partenza resti ancora disponibile, perche' non vogliamo influenzare le parti di sistema che ne fanno uso. Il meccanismo di *ereditarieta'* permette esattamente di raggiungere questo obiettivo.

```
class Frame extends Window
{ private String title;
  private int borderWidth;

  ...

  public void setTitle (String t)
  { title=t;
  }

  public void show ()
  { <<visualizza sullo schermo una finestra
      dotata di titolo e di bordo>>
  }
}
```

Siccome la classe `Frame` estende la classe `Window`, un oggetto della classe `Frame` possiedera' tutti i campi dati specificati nella sopraclasse `Window`, oltre ai nuovi campi specificati nella sottoclasse `Frame`. Inoltre `Frame` specifica dei nuovi metodi (come ad es. `setTitle`), ridefinisce il metodo `show` (perche' il visualizare una finestra con bordo e titolo comporta operazioni diverse dal visualizare una finestra "nuda"), e eredita i metodi che non ridefinisce (come ad es. `move`, perche' spostare un oggetto

della classe `Frame` comporta gli stessi passi dello spostare un oggetto della classe `Window`). Si noti tuttavia che, nell'eseguire il metodo `move` su di un oggetto `Frame` si eseguira' il metodo `show` specificato nella classe `Frame`. Ad esempio, nelle istruzioni seguenti:

```
Window win= new Window ();
Frame frm = new Frame();
...
win.move(20,10);
frm.move(20,10);
```

la prima chiamata di `move` comporta l'esecuzione del metodo `show` della classe `Window` sull'oggetto riferito da `win`, mentre la seconda chiamata di `move` comporta l'esecuzione del metodo `show` della classe `Frame` sull'oggetto riferito da `frm`. Infatti la variabile riservata `this` conterra' il riferimento in `win` nel primo caso, e il riferimento in `frm` nel secondo.

Questo fenomeno e' un aspetto del meccanismo di *binding dinamico* associato al meccanismo di *subtyping/sostitutivita'*: in generale una variabile `v` di tipo `C` puo' contenere un riferimento ad un oggetto appartenente ad una qualunque sottoclasse (sottotipo) `C'` di `C`, e quando viene invocato un metodo `m` attraverso un'istruzione:

```
v.m(...)
```

ad essere eseguito sara' il metodo `m` specificato nella classe `C'` dell'oggetto `o'` il cui riferimento e' memorizzato in `v`.

La combinazione di subtyping e binding dinamico permette quindi un maggiore riutilizzo del codice: se Java usasse il *binding statico* (in base al quale il codice del metodo `m` che deve essere eseguito in un'istruzione `v.m` sarebbe determinato staticamente, in fase di compilazione, e non dinamicamente in base alla classe

dell'oggetto riferito da `v`) la classe `Frame` potrebbe in ogni caso ereditare senza problemi il metodo `setSize` della classe `Window` (che si limita a usare le variabili di istanza `x` e `y`, ma (*senza* il binding dinamico) non potrebbe ereditare il metodo `move` che avrebbe dovuto essere duplicato.

Vediamo qualche altro esempio di applicazione del meccanismo di subtyping e del binding dinamico. Il frammento di codice

```
Window win;
Frame frm = new Frame();
win=frm;
frm.setSize(100,50);
win.show();
```

e' legale: siccome `Frame` e' sottotipo di `Window` la variabile `win` puo' contenere un riferimento ad un oggetto di tipo `Frame`. Dopo l'esecuzione dell'assegnamento `win=frm`, entrambe le variabili `frm` e `win` riferiranno lo stesso oggetto (creato dalla seconda riga di codice), e l'invocazione di metodo `win.show()` comportera' l'esecuzione del metodo `show` specificato nella classe `Frame` sul frame (di dimensioni 100×200) riferito da `win`.

Supponiamo di avere una classe

```
class C {
   ...
   void m (Window w) {
   ...
   w.show();
   ...
   }
}
```

il seguente frammento codice e' legale:

```
C v = new C();
Frame frm = new Frame();
...
v.m(frm);
```

e, in virtu' del meccanismo di binding dinamico, durante l'esecuzione del metodo `m` l'istuzione `w.show()` causera' l'esecuzione del metodo `show` definito nella classe `Frame`.

In questo modo e' anche possibile, ad esempio, memorizzare in un vettore riferimenti ad oggetti di classi diverse (ma appartenenti ad una superclasse comune, come ad es. `Window`) e scrivere un ciclo che invochi automaticamente la versione opportuna del metodo `show` per ogni elemento del vettore.

```
Window winArr = new Window[5];
winArr[0] = new Window();
winArr[1] = new Frame();
...
for (int i=0; i<5; i++) winArr[i].show();
```