Classi dinamiche

Classi derivate e relazione IS-A

Si consideri una classe Base e una classe Derived derivata pubblicamente dalla classe Base:

```
class Base {
  /* ... omissis ... */
};

class Derived : public Base {
  /* ... omissis ... */
};
```

Come già sappiamo, la derivazione pubblica consente di effettuare, in maniera implicita, le conversioni di tipo dette up-cast, ovvero la conversione da un puntatore o riferimento per un oggetto Derived verso un puntatore o riferimento per un oggetto Base.

```
Base* base_ptr = new Derived;
```

> Se la derivazione fosse non pubblica, cioè private o protected, tale conversione sarebbe legittima solo se effettuata nel contesto della classe derivata o all'interno di una funzione friend della classe.

Intuitivamente, l'esistenza di questa conversione indica che è possibile utilizzare un oggetto Derived (tipo concreto) come se fosse un oggetto della classe Base (tipo astratto), ignorando eventuali caratteristiche specifiche della classe Derived per concentrarsi sulle caratteristiche che questa classe ha in comune con (eredita da) la classe Base.

Si dice che la classe Derived è in relazione "IS-A" con la classe Base, cioè è una particolare concretizzazione della classe Base, e quindi deve potere essere utilizzato, dall'utente, come se fosse un oggetto di tipo Base.

In altre parole, in questo contesto l'utente vuole lavorare con oggetti di tipo Base, ignorando eventuali differenze tra le varie concretizzazioni possibili.

Torna all'indice

Esempio 1

Dalla classe base Docente possiamo derivare tante classi in relazione IS-A, come Professore_Ordinario, Professore_Associato, Ricercatore, Professore_a_Contratto, ecc. Ognuna di queste classi potrebbe avere caratteristiche (dati o metodi) specifici che la differenziano dalle altre. Un utente che NON sia interessato a queste peculiarità può astrarre da esse, vedendo tutti gli oggetti concreti come istanze di Docente e usando solo l'interfaccia messa a disposizione dalla classe base Docente.

> NOTA: in contesti diversi la derivazione potrebbe essere utilizzata con altri scopi. Per esempio, a volte si usa (secondo alcuni, a sproposito) l'ereditarietà per codificare la relazione "HAS-A": la classe Derived ha un sotto-oggetto di tipo Base, ovvero lo "possiede" e quindi lo può usare.

Per esempio: un Automezzo ha un Motore e, siccome più automezzi di tipo diverso possono usare lo stesso tipo di motore, si potrebbe decidere di usare Motore come classe base comune ai vari automezzi concreti.

La differenza sostanziale, rispetto al caso precedente, è data dal fatto che l'utente di queste classi, probabilmente, è interessato ad usare gli automezzi concreti (e non i motori in essi contenuti, che potrebbero essere visti come dei dettagli implementativi): quindi, l'utente NON è interessato alla possibilità di convertire un automezzo concreto in un Motore, per cui l'uso di ereditarietà pubblica è inappropriato.

Le alternative sono:

1. uso di ereditarietà privata:

```
class Utilitaria : private Motore { /* ... */ };
```

2. uso del contenimento:

```
class Utilitaria { Motore motore; /* ... */ };
```

Tra le due opzioni, dovrebbe essere preferita la seconda, in quanto più intuitiva da usare; la seconda opzione, inoltre, è facilmente estendibile al caso in cui l'automezzo debba contenere più di un solo sotto-oggetto di un determinato tipo (esempio: auto ibride con più motori).

Nel seguito, ci concentreremo sul caso in cui l'utente sia intenzionato a stabilire relazioni di tipo "IS-A", usando quindi l'ereditarietà pubblica e sfruttando le conversioni implicite allo scopo di lavorare con la classe base, astraendo dai dettagli implementativi delle classi derivate.

Torna all'indice

Metodi virtuali e classi dinamiche

> Nota bene: nel seguito sono mostrati spezzoni di codice, incompleti; il loro unico scopo è quello di consentire al lettore di "immaginare" un contesto concreto, ma semplificato al massimo, nel quale applicare le nozioni di cui si sta trattando.

Consideriamo il seguente esempio

```
class Printer {
public:
 void print(const Doc& doc);
};

class FilePrinter : public Printer {
public:
 void print(const Doc& doc);
};

class NetworkPrinter : public Printer {
public:
 void print(const Doc& doc);
};
```

Supponiamo che il codice utente debba stampare alcuni documenti utilizzando una stampante e, non essendo interessato ai dettagli implementativi, utilizzi l'astrazione `Printer` nel modo seguente:

```
void stampa_tutti(const std::vector& docs, Printer* printer) {
 for (const auto& doc : docs)
  printer->print(doc);
}
```

Il chiamante invocherà la funzione `stampa_tutti` passando un puntatore ad una stampante concreta (una specifica istanza di `FilePrinter` o `NetworkPrinter`), sfruttando l'up-cast consentito dalla relazione "IS-A". Quando esamina la chiamata al metodo `print`, il compilatore si troverà a fare la risoluzione dell'overloading conoscendo solo il tipo statico di `printer` (puntatore alla classe base `Printer`), senza avere conoscenza di quello che è il vero tipo dinamico (puntatore ad una delle specifiche classi derivate dalla classe base): di conseguenza, effettuerà la ricerca delle candidate nella classe `Printer` e troverà solo il metodo `Printer::print`, che verrà scelto come migliore funzione utilizzabile.

In realtà, però, l'utente vorrebbe che fosse invocato il metodo specifico della stampante concreta passata alla funzione, che potrebbe dovere fare operazioni diverse a seconda della classe di appartenenza (per esempio, una `NetworkPrinter` potrebbe tenere traccia del numero di pagine stampate dai vari utenti). Intuitivamente, ogni classe concreta "ridefinisce" il metodo `print` per fargli fare la cosa corretta per il contesto specifico: questa ridefinizione del metodo dovrebbe prevalere (override) rispetto a quella della classe. Serve quindi un meccanismo tecnico che consenta di interrogare (a tempo di esecuzione) il puntatore, allo scopo di capire quel è il suo tipo dinamico e quindi "ridirezionare" la chiamata del metodo `print` alla classe concreta corretta: questo meccanismo tecnico effettua la cosiddetta "risoluzione dell'overriding" e, nel caso del C++, viene attivato solo quando i metodi della classe base sono stati dichiarati essere "metodi virtuali" (cioè, ridefinibili nelle classi derivate).

```
class Printer {
public:
 virtual void print(const Doc& doc);
};
```

Una classe che contenga almeno un metodo virtuale viene detta classe dinamica, in quanto per gli oggetti di questa classe vengono messe a disposizione le funzionalità che consentono di implementare la risoluzione dell'overriding e, più in generale, la *RTTI* (Run-Time Type Identification).

A livello implementativo, ad ogni oggetto che è istanza di una classe dinamica viene associato un puntatore (non accessibile direttamente da parte dell'utente) usando il quale il RTS (Run-Time Support) del linguaggio può raggiungere le informazioni di tipo della classe. L'esistenza di questo puntatore si può notare se si confrontano, usando l'operatore `sizeof`, oggetti di classi dinamiche e statiche (cioè non dinamiche):

```
#include

class Statica {
 ~Statica() {}
};

class Dinamica {
 virtual ~Dinamica() {}
};

int main() {
 std::cout << "sizeof(Statica) = " << sizeof(Statica) << std::endl;
 std::cout << "sizeof(Dinamica) = " << sizeof(Dinamica) << std::endl;
}
```

Avendo dichiarato virtuale il metodo

```
  void Printer::print(const Doc&);
```

una classe derivata da `Printer` che lo ridefinisca (usando lo stesso nome e lo stesso numero e tipo degli argomenti) ne fa l'overriding; si noti che, se il metodo non fosse stato dichiarato `virtual` nella classe base, NON si avrebbe overriding, ma si avrebbe invece hiding. Nella classe derivata non è necessario (ma è consentito) ripetere la parola chiave virtual. Se si uso lo standard C++11 o superiore è anche consigliato usare la parola chiave `override`, da usarsi alla fine della dichiarazione, in questo modo:

```
class NetworkPrinter : public Printer {
public:
 void print(const Doc& doc) override;
};
```

L'uso di "override" è utile perché causa un errore nel caso in cui nella classe base `Printer` non esista un metodo virtuale corrispondente. L'errore è meno frequente di quanto si possa immaginare, perché:

1. potremmo esserci dimenticati di usare la parola chiave `virtual` nella classe base;
2. potremmo avere modificato leggermente il tipo del metodo, cambiando il numero o il tipo dei parametri.

Torna all'indice

Esempio 2

```
class Printer {
public:
 virtual std::string name() const;
 void print(const Doc& doc);
};

class NetworkPrinter : public Printer {
public:
 // Errore: in Printer::name il parametro implicito this è qualificato const.
 std::string name() override;
 // Errore: il metodo Printer::print non è dichiarato virtual.
 void print(const Doc& doc) override;
};
```

Torna all'indice

Metodi virtuali puri e classi astratte

Quando si definisce una classe base come `Printer`, spesso non si ha la possibilità di fornire una implementazione sensata per i metodi virtuali. Intuitivamente, la classe `Printer` fornisce "solo" l'interfaccia del concetto astratto di stampante e di conseguenza non può stampare davvero un documento: l'unico suo scopo è quello di ridirezionare la chiamata ad una delle classi concrete. Invece di fornire una implementazione fittizia (per esempio, una che lanci una eccezione) è preferibile indicare che il metodo virtuale è "puro", usando la sintassi "`= 0`" al termine della sua dichiarazione.

```
class Printer {
public:
  virtual std::string name() const = 0;
  virtual void print(const Doc& doc) = 0;
};
```

Una classe che contenga metodo virtuali puri è detta classe astratta (in senso tecnico; spesso si usa dire che una classe è astratta anche in senso "metodologico", non tecnico, creando un po' di confusione). Il fatto che un metodo virtuale sia puro significa che ogni classe concreta che eredita dalla classe astratta `Printer` è tenuta a fare l'overriding del metodo; se NON fa l'overriding, il metodo rimane puro e quindi la classe derivata è anche essa una classe astratta (in senso tecnico). Si noti che NON è possibile definire un oggetto che abbia come tipo una classe astratta: questi possono solo essere usate come classi base per derivate altre classi (astratte o concrete).

Esempio 3

```
  Printer p; // errore: Printer è astratta
  NetworkPrinter np; // ok, se NetworkPrinter ha effettuato l'overriding
                     // di tutti i metodi virtuali puri di Printer
```

Torna all'indice

I distruttori delle classi astratte

I distruttori delle classi astratte dovrebbero essere sempre dichiarati virtual e non dovrebbero mai essere metodi puri (ovvero, occorre fornirne l'implementazione). Ovvero, l'interfaccia di una classe dinamica astratta dovrebbe tipicamente avere la struttura seguente:

```
class Astratta {
public:
 // metodi virtuali puri
 virtual tipo_ritorno1 metodo1(parametri1) = 0;
 virtual tipo_ritorno2 metodo2(parametri2) = 0;
 virtual tipo_ritorno3 metodo3(parametri3) = 0;
 // ...

 // distruttore virtuale NON puro (definito, non fa nulla)
 virtual ~Astratta() {}
};
```

La ragione per questo modo di definire il distruttore è legata alla necessità di consentire una corretta distruzione degli oggetti delle classi concrete derivate dalla classe astratta. Infatti:

1. il distruttore della classe concreta invoca (implicitamente) il distruttore delle sue classi base, che quindi deve essere definito (cioè non può essere un metodo puro);
2. se il distruttore della classe astratta NON fosse virtuale, si avrebbero dei memory leaks.

Torna all'indice

Esempio 4

```
class Astratta {
public:
 virtual void print() const = 0;
 ~Astratta() {} // distruttore errato: non è virtuale.
};

class Concreta : public Astratta {
 std::vector vs;
public:
 Concreta() : vs(20, "stringa") {}
 void print() const override {
  for (const auto& s : vs)
    std::cout << s << std::endl;
 }

 // Nota: il distruttore di default sarebbe OK; lo ridefiniamo solo
 // per fargli stampare qualcosa, così che sia evidente il fatto che
 // non è stato invocato.
 ~Concreta() { std::cout << "Distruttore Concreta" << std::endl; }
};

int main() {
 Astratta* a = new Concreta;
 a->print();
 // memory leak: non viene distrutto il vector nella classe concreta.
 delete a; // invoca il distruttore di Astratta (che non è virtual)
}
```

Risoluzione overriding

Viene effettuata a tempo di esecuzione dal RTS (supporto a tempo di esecuzione). Si noti che, in ogni caso, a tempo di compilazione viene fatta la risoluzione dell'overloading nel solito modo.

Affinché si attivi l'overriding occorre che:

1. il metodo invocato sia un metodo virtuale (esplicitamente o implicitamente, se ereditato da una classe base);
2. il metodo viene invocato tramite puntatore o riferimento (altrimenti non vi può essere distinzione tra il tipo statico e il tipo dinamico dell'oggetto e quindi si invoca il metodo della classe base);
3. almeno una delle classi lungo la catena di derivazione che porta dal tipo statico al tipo dinamico ha effettuato l'overriding (in assenza di overriding, si invoca il metodo della classe base);
4. il metodo NON deve essere invocato mediante qualificazione esplicita (la qualificazione esplicita causa l'invocazione del metodo come definito nella classe usata per la qualificazione).

Torna all'indice