Tipi di dati astratti

Lezione 15 di Programmazione 2

Docente: Giuseppe Scollo

Università di Catania, sede di Comiso (RG)
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Studi in Informatica applicata, AA 2007-8

Indice

classi e oggetti: origini, cenni storici

la **programmazione orientata agli oggetti** estende la programmazione procedurale con i concetti di

- **classe: incapsulamento** di dati e operazioni per l'accesso ad essi
- **oggetto: istanza dinamica** di una classe, ha **stato** (variabili d'istanza) e **identità** (riferimento)

nasce nell'ambito dei linguaggi di simulazione: Simula 67

ben si presta alla rappresentazione della **concorrenza**

si sviluppa negli anni '80 come **paradigma di progettazione del software**

linguaggi più diffusi: Smalltalk, C++, Eiffel, Java

anni '90: diventa il paradigma dominante nella **produzione industriale di software**

standardizzazione:

- **OMG:** Object Management Group, http://www.omg.org
- **CORBA:** Common Object Request Broker Architecture
- **UML:** Unified Modeling Language

tipi di dati astratti, interfacce, visibilità

il concetto di **tipo di dato astratto** (ingl. abstract data type : **ADT**) nasce negli anni '70, come costrutto per la definizione, da parte del programmatore, di **costruttori di tipi di dati**

- nei linguaggi OO: concetti di **interfaccia, classe, classe astratta** (v. prossima lezione)
- matematicamente: classe di **algebre similari**, specificata da **assiomi**

ADT nei linguaggi OO:

- **Java**, ha un costrutto specifico per la dichiarazione di interfacce, ma limitato al profilo sintattico delle operazioni
- Eiffel offre la Business Object Notation (BON) per la specifica assiomatica di precondizioni e postcondizioni sulle operazioni, secondo il principio del **Design by Contract** (Meyer)
- **C++** non ha un costrutto specifico per le interfacce: definite nella **pragmatica** del linguaggio

visibilità dei membri (variabili e operazioni) di una classe: **public**, **protected**, **private**

- **N.B.** significati diversi in linguaggi OO diversi

in C++: si conviene che l'interfaccia di una classe consista dei profili delle sue **operazioni pubbliche**

- **ADT**: tipo di dato accessibile solo attraverso un'interfaccia

esempio: metrica cartesiana rivisitata

riformuliamo nella sintassi delle classi la definizione del tipo di dato dei punti nello spazio bidimensionale, con una funzione per la distanza da un punto dato (v. lezione 7)

Punto2D.h: interfaccia + struttura dati

```
class Punto2D
{ public:
    Punto2D();                        // costruttore
    float distanza2D(Punto2D) const; // operazione "query"
  private:
    float x, y; }
```

implementazione **Punto2D.c++** (con specifica di ambito mediante "**::**") :

```
#include <cstdlib>
#include <cmath>
#include "Punto2D.h"
Punto2D::Punto2D()
{ x = 1.0*rand()/RAND_MAX; y = 1.0*rand()/RAND_MAX; }
float Punto2D::distanza2D(Punto2D a) const
{ float dx = x-a.x, dy = y-a.y; return sqrt(dx*dx + dy*dy); }
```

funzioni e classi friend

l'esempio precedente mostra una soluzione di **compromesso** fra

> l'obiettivo di separare l'interfaccia dall'implementazione, e
> la necessità di conoscere a tempo di compilazione (del client dell'ADT, che può invocarne il costruttore) i requisiti di memoria delle istanze dell'ADT

un altro compromesso, stavolta rispetto al principio di incapsulamento, è fornito dalla possibilità di dichiarare come **friend**, in un classe, funzioni e classi che **non** ne sono membri

> tali funzioni e classi hanno accesso alla parte privata della classe

ad esempio, la funzione **`float distanza2D(Punto2D a, Punto2D b)`** potrebbe venir definita al di fuori della classe **`Punto2D`**, con il codice visto nella lezione 7, pur di includere nella definizione di questa classe la dichiarazione

```
friend float distanza2D(Punto2D, Punto2D);
```

ridefinizione di operatori

con le classi si possono definire nuovi tipi di dati: per poterli usare in modo simile ai tipi di dati predefiniti sono necessari alcuni accorgimenti, fra i quali l'**overloading** di operatori di uso comune

eccone due esempi, per il tipo di dato **Punto2d** visto sopra

ridefiniamo l'eguaglianza di due punti, per farla valere se la loro distanza è inferiore a una data soglia:

```
friend int operator==(Punto2D a, Punto2D b)
{ return(distanza2D(a, b) < .001; }
```

supponiamo ora che il suddetto tipo di dato sia dotato di due funzioni membro pubbliche per leggerne le coordinate:

```
float X() const { return x; }
float Y() const { return y; }
```

in tal caso possiamo ridefinire l'operatore << della classe standard **ostream**, per specificare l'output di dati di tipo **Punto2d** in un formato conveniente:

```
ostream& operator<<(ostream& t, Punto2d p)
{ t << "(" << p.X() << "," << p.Y() << ")"; return t; }
```

esercizi

1. modificare l'implementazione dell'ADT **Punto2D** in modo da rappresentare punti in coordinate polari
2. rivisitare la prima parte dell'esercizio 3 della lezione 7 nella sintassi delle classi: definire interfaccia, struttura dati e implementazione della classe dei punti nello spazio cartesiano a **N** dimensioni, con una funzione che calcoli la distanza dell'oggetto punto da un punto dato come argomento, dove **N** sia un parametro del costruttore dell'ADT