Corso di Linguaggi di Programmazione

Lezione 18

Alberto Ceselli
alberto.ceselli@unimi.it

Dipartimento di Informatica
Università degli Studi di Milano

07 Maggio 2013

Features richieste al linguaggio:

- Supporto per ADT (lezione 15)
- Ereditarietà
- Binding dinamico di chiamate a metodi (e possibilità di *polimorfismo*)

Ripasso: il concetto di sottotipo

- un tipo A è un insieme di elementi, un sottotipo B del tipo A è un *sottoinsieme* degli elementi di A
- per ogni elemento $b \in B$ vale $b \in A$: vale la relazione **è-un** rispetto ad A.
- più concretamente: in un programma B può apparire ovunque appare A.
- N.B un sottotipo è *compatibile* con il tipo da cui deriva, ma in generale **non equivalente**!

Ripasso: classi e sottoclassi

- fra classi e sottoclassi c'è una relazione *gerarchica*
- una sottoclasse è un sottotipo se assicuro che:
  - una sottoclasse possa solo *aggiungere* proprietà e metodi
  - oppure *ridefinire* metodi della superclasse in modo *compatibile* (senza causare errori di tipo)
- *ereditarietà*: è la proprietà di una sottoclasse di possedere caratteristiche della superclasse
- Una sottoclasse può *ridefinire* un metodo delle sue superclassi (*overriding*).
- l'ereditarietà può essere *singola* o *multipla*.

Ereditarietà singola e multipla

- Il linguaggio permette ad una sottoclasse di ereditare da più superclassi?
- `Arredamento_mult1.cpp`

1 Problema dell'ambiguità

  - Es. A deriva sia da B che da C; sia B che C hanno un metodo `display` che può essere ereditato.
  - C++ permette eredità multipla, e risolve ambiguità utilizzando `::`, ovvero A non eredita `display`, ma `B::display` e `C::display`

Ereditarietà singola e multipla

- Il linguaggio permette ad una sottoclasse di ereditare da più superclassi?
- `Arredamento_mult2.cpp`

2 Problema degli antenati comuni ("diamond" o "shared" inheritance)
   - Es. Sia B che C derivano da D; A deriva sia da B che da C: A deve ereditare due copie di tutti i metodi di D? Oppure solo una copia? Da quale genitore?
   - C++ permette eredità multipla, e risolve ambiguità utilizzando ::

Implementazione di ereditarietà singola e multipla

- A compile time viene costruito per ogni classe (e sottoclasse) un *class instance record* (CIR), in cui sono memorizzate proprietà e puntatori ai metodi.
- Ereditarietà multipla in sintesi: molta flessibilità, marginale problema di efficienza, sostanziale problema di struttura e manutenzione del codice.

Ereditarietà e scope

- Modificatori di scope:
  - `public`: visibile ai metodi della classe, metodi delle sottoclassi e codice client
  - `private`: visibile solo ai metodi della classe
  - `protected`: visibile ai metodi della classe e delle sottoclassi (a volte a *tutte le sottoclassi nel package*, es. Java), ma non a codice client
  - alcuni linguaggi hanno modificatori appositi per il package (es. `internal` in C#)
  - C++ utilizza anche `friend`
  - In C++ i modificatori di scope sono utilizzati anche durante l'estensione di una classe `class Sedia :` **`public`** `Mobile` ...

Ereditarietà e scope

- N.B. Ricordando che una sottoclasse è un sottotipo se assicuro che:
  - una sottoclasse possa solo *aggiungere* proprietà e metodi
  - oppure *ridefinire* metodi della superclasse in modo *compatibile* (senza causare errori di tipo)
- utilizzare `private` nella derivazione di una classe C++ potrebbe impedire alle sottoclassi di essere sottotipi della superclasse.

Binding dinamico

- Variabili **polimorfiche:** una variabile di tipo superclasse può fare riferimento anche ad oggetti *di una sua qualsiasi sottoclasse*.
- Le sottoclassi potrebbero aver sovrascritto alcuni metodi della superclasse.
- **binding dinamico:** la porzione di codice da attivare (metodo sottoclasse / metodo superclasse) viene deciso a run-time
- `Arredamento_v2.cpp`

Esempio di applicazione: lista eterogenea

`Arredamento_lista.cpp`

Type checking e polimorfismo

- Molti linguaggi hanno strong typing come obiettivo.
- Binding statico permette static type checking (vedi lezioni precedenti)
- E il binding dinamico? Alternative

Type checking e polimorfismo

- Molti linguaggi hanno strong typing come obiettivo.
- Binding statico permette static type checking (vedi lezioni precedenti)
- E il binding dinamico? Alternative
  1. richiedere che tipi di parametri formali ed attuali corrispondano perfettamente (equivalenza), e che il tipo del valore restituito dal metodo corrisponda con il tipo atteso nella chiamata. (il prototipo del metodo che sovrascrivere deve essere identico a quello del metodo sovrascritto)
  2. rilassamento: richiedere *compatibilità* anziché *equivalenza*
  3. differire il controllo sui tipi a run-time (molto più dispendioso)

Overloading

- un sottoprogramma *generico* è un sottoprogramma la cui computazione può essere effettuata su dati di tipo diverso durante attivazioni diverse
- un sottoprogramma si dice **overloaded** se ha lo stesso nome di un altro sottoprogramma nello stesso scope
- `Arredamento_overload.cpp`
- domanda: possono esserci metodi con *stessi parametri formali*, ma che restituiscono valori di *tipo diverso*?

```
class Stack {
...
  double get();
  int get();
...
}
```

Overloading

- risposta: in generale non è ammesso. Come potrei fare a scegliere quale dei due deve essere chiamato basandomi sul tipo dei parametri attuali?
- inferenza di tipo?

Un oggetto conosce se stesso

- Come per gli ADT, i metodi di una classe hanno accesso agli altri metodi ed alle proprietà della classe
- ... e anche all'oggetto stesso
- Es. `this` in C++
- Spesso ha accesso anche alla superclasse (Es. `super` in Java o `base` in C#) o alle superclassi (`superclasse_A::` in C++)

Classi e tipi base:

- in OOP puro, tutto è un oggetto, *compresi i tipi base* (int, char ...) e tutta la computazione avviene tramite scambio di messaggi tra questi oggetti
- pulito, ma in pratica l'overhead è troppo elevato
- **wrapper:** classi che incapsulano i tipi base
- **boxed:** un tipo base incapsulato in un wrapper
- A volte gestiti automaticamente (Es. Java)

Boxing e unboxing

- Boxing in Java fino a v. 1.4:

```
// Assigning primitive type to wrapper type
    Integer iWrapper = new Integer(10);
```

- Autoboxing in Java dalla v. 1.5:

```
// Assigning primitive type to wrapper type
    Integer iWrapper = 10;
```

- Unboxing in Java fino a v. 1.4:

```
// Assigning object to primitive.
public void intMethod(Integer iWrapper){
   int iPrimitive = iWrapper.intValue();
}
```

- Autounboxing in Java dalla v. 1.5:

```
// Assigning object to primitive.
public void intMethod(Integer iWrapper){
   int iPrimitive = iWrapper;
}
```

Up e down - cast

- Domanda: assegnamenti fra superclassi e sottoclassi sono validi?

```
class Sedia: public Mobile {
...
}
...
 Mobile m;
 Sedia s;
...
 m = s; // Es 1
...
 s = m; // Es 2
...
```

Up e down - cast

- Es 1: **upcast** (OK, compatiblità di tipi)

```
class Sedia: public Mobile {
...
}
...
 Mobile m;
 Sedia s;
...
 m = s; // Es 1
...
 s = m; // Es 2
...
```

- Es 2: **downcast** (deve essere gestito come cast esplicito!)

```
 s = (Sedia) m; // Es 2
```

Membri di una classe e membri di un'istanza

- Concettualmente: fino ad ora abbiamo parlato di proprietà e metodi di *istanze* di una classe
- potrebbero esserci membri della classe stessa?
- C++, Java, C# utilizzano la parola chiave `static` per descrivere membri della classe
- Esempio:

```
class Mobile {
...
static const double PesoLegno = 10.0;
static double calcolaPeso(double volume);
...
}
...
  public static int main(...) {
  ...
    System.out.println( Mobile.PesoLegno );
    Mobile.calcolaPeso(4.2);
  ...
  }
```

Interface ed implementation

- Alcuni linguaggi non supportano ereditarietà multipla.
- Java prevede la definizione di classi puramente astratte (`interface` ed `implements`) (Es1 `Container.java` e `List.java`)
- Ogni classe può implementare più interfacce (Es2 `Container`, `Comparable` e `Printable` .java)

C++

- `struct` e `class` gestite allo stesso modo (i membri di una `struct` sono per default `public`, quelli di `class` sono `private`)
- Binding per default *statico*, a meno che non venga specificato `virtual`

C++

- Possibilità di definire *interfacce* a metodi:

```
public class shape {
  public:
    virtual void draw() = 0;
    ...
}
```

- `draw()` è *pure virtual*, una classe con almeno un metodo *pure virtual* è detta *astratta* e non può essere istanziata
- Valutazione: molto potente, molti controlli di tipo a compile time, preserva l'obiettivo efficienza del C, ma articolato e difficile da gestire.

Java

- Utilizza parola chiave `extends` per implementare ereditarietà singola
- Simula ereditarietà multipla tramite `implements` di più interfacce.
- Le classi sono istanziate dinamicamente (`new`)
- Hanno un costruttore (metodo con nome della classe) e metodo `finalize` chiamato dal garbage collector
- (Es3 : `List.java` e `DList.java`)

Java

- Ogni metodo può essere "overridden", tranne quelli marcati `final` (ESEMPIO su List.java)
- Il binding è *dinamico* per tutti i metodi, tranne quelli `final`
- Tutte le classi discendono da `Object`
- Valutazione: supporto consistente (ed esclusivo) di OOP, gestione semplice di scoping ed ereditarietà

C sharp

- `extends`, `implements` ed `interface` analogo a Java
- Ereditarietà singola, ma possibilità di implementare più interfaces.
- Valutazione: stile intermedio tra C++ e Java.
- binding statico per default, ma possibilità di definire metodi `virtual` e corrispondenti `override`
- pure virtual di C++ $\rightarrow$ parola chiave `abstract`.
- ogni classe che contiene (anche per ereditarietà) un metodo `abstract` deve essere marcata come `abstract`.

Ada 95

- Ha un meccanismo di classi e sottoclassi tramite tipi `tagged`:
  type `Person` **is tagged private**
  ...
  type `Student` **is new** `Person` **with**
- gestito anche meccanismo di coercion (a variabile di tipo Person posso assegnare valori tipo Student)
- solo ereditarietà singola, ma meccanismo simile a `friend` in C++ (Child library packages)
- nessun meccanismo per restringere l'ereditarietà (sempre sottotipi)