Principi di progetto OO – IV

Stefano Mizzaro

Dipartimento di matematica e informatica
Università di Udine
http://www.dimi.uniud.it/~mizzaro
mizzaro@dimi.uniud.it

PAOO, Lezione 8
11/3/2004

Riassunto (1/3)

- I principi della progettazione OO
  - Livelli di incapsulamento
  - Dipendenze
  - Domini
  - Ingombro
  - Legge di Demeter
  - Coesione
  - Spazio degli stati (ed eredità)
  - Transizioni e comportamento (ed eredità)

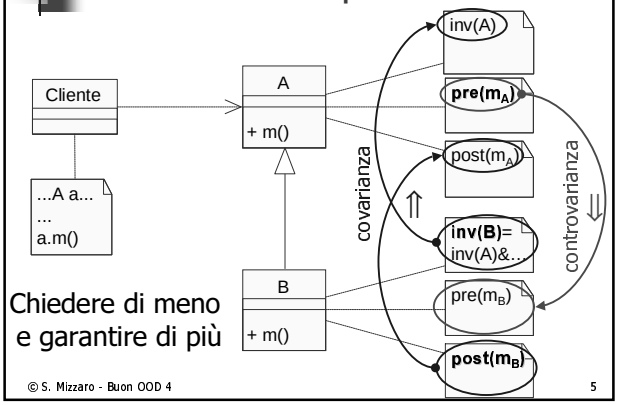
Riassunto (2/3)

- Asserzioni
- Invarianti di classe
- Precondizioni e postcondizioni delle operazioni (metodi)
- Progetto per contratto
- inv, pre, post ed eredità:
  - Principio di sostituibilità (istanza di sottoclasse dove ci si aspetta istanza della sopraclasse)
    - Conformità di tipo
    - Comportamento chiuso

Riassunto (3/3)

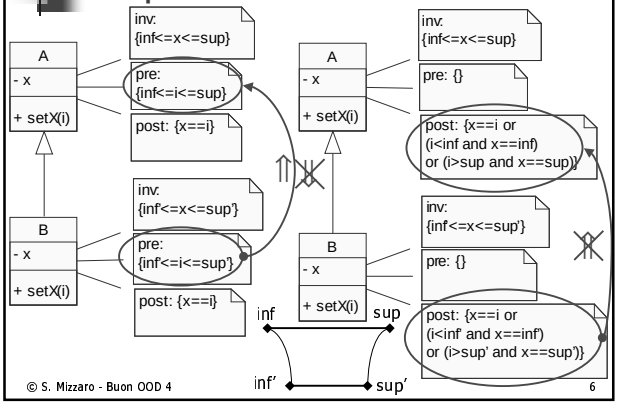
- Per avere la sostituibilità:
- Conformità di tipo
  - inv sottoclasse più forte inv sopraclasse (SdS sottoclasse ha più vincoli sulle dim. della sopraclasse e vincoli nuovi sulle nuove dim.)
  - pre nel metodo della sottoclasse più debole (chiedere di meno, controvarianza)
  - post nel metodo della sottoclasse più forte (garantire di più, covarianza)
- Comportamento chiuso
  - Metodi ereditati dalla sopraclasse devono rispettare l'inv della sottoclasse

Conformità di tipo



Chiedere di meno e garantire di più

Un problema...



Scaletta

- Progetto per contratto in Java
- I pericoli di ereditarietà e polimorfismo
  - Gerarchie errate
  - Gestione del polimorfismo
- Genericità
- Ancora sull'interfaccia di una classe
  - Anelli di operazioni
  - Tipologie di stati e di comportamento
- Ancora sulla coesione, di operazione/metodo

Progetto per contratto in Java

- iContract
  - [iContract su google]
- Filosofia
  - pre, post e inv come tag di Javadoc
  - Sintassi particolare (e abbastanza potente) per esprimere pre, post e inv
  - Tool che compila ed esegue controllando pre, post e inv (e segnalando se non valgono)

Sintassi iContract: invarianti

- **`@invariant <espressione Java>`**
- **`intValue()`** è un metodo della classe **`PositiveInteger`** (ereditato da **`Integer`**)

```
/** A PositiveInteger is an Integer that is
  * guaranteed to be positive.
  *
  * @invariant intValue() > 0
*/
class PositiveInteger extends Integer {
  ...
}
```


Sintassi iContract: pre- e post-condizioni

- Tag **`@pre`** e **`@post`**
- Posso usare i nomi dei parametri formali (**`f`**)
- **`return`** indica il valore restituito

```
...
/**
  * @pre f >= 0.0
  * @post Math.abs((return * return) - f) < 0.001
*/
public float sqrt(float f) {
  ...
}
...
```


Sintassi iContract: espressioni

- Espressioni Java, ma non solo
  - Operatore **`@pre`**
  - **`exists`**, **`forall`**, **`implies`**
  - …

```
/** Append an element to a collection.
  *
  * @post c.size() = c@pre.size() + 1
  * @post c.contains(o)
*/
public void append(Collection c, Object o) {
  ...
}
```

```
/**
* @post exists IRoom r in getRooms() | r.isAvailable()
*/
```


Progetto per contratto in Java

- iContract
  - Gestisce anche ereditarietà
  - Attenzione agli effetti collaterali…
- Altri tool [google]
  - jContractor (definisce metodi)
  - JMSAssert (simile)
  - Jass
  - …

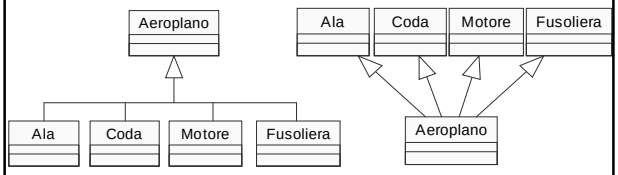
Scaletta

- Progetto per contratto in Java
- I pericoli di ereditarietà e polimorfismo
  - Gerarchie errate
  - Gestione del polimorfismo
- Genericità
- Ancora sull'interfaccia di una classe
  - Anelli di operazioni
  - Tipologie di stati e di comportamento
- Ancora sulla coesione, di operazione/metodo

Ereditarietà

- Molto potente, troppo?
- Molto pericolosa
- Parente del goto
- Non è vero che "per essere OO bisogna usare l'ereditarietà"
- Vediamo alcuni esempi con usi sbagliati

Un aeroplano?



- Aggregati errati

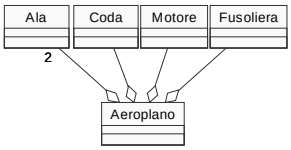
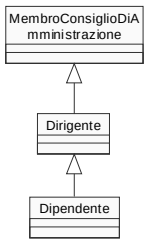
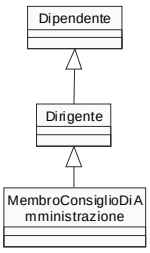


Diagramma organizzativo?

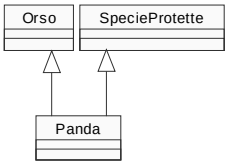


- Gerarchie invertite

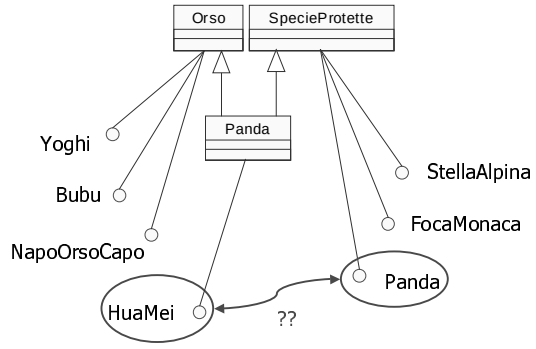


Tutto ok?

- Un panda è un orso
- Il panda è una specie protetta

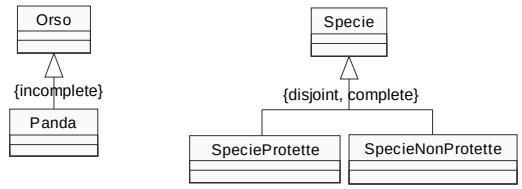


Vediamo le istanze…



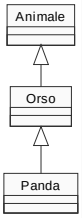
Confusione fra classe e istanza

- Sono 2 gerarchie distinte
  - Animali
  - Specie

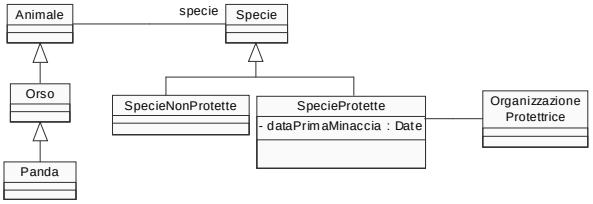


Come metterle in relazione?

- Variabile d'istanza **`boolean eProtetto`** in **`Orso`**, o in **`Animale`**?
  - Yoghi protetto e Bubu no??
- Variabile di classe (o metodo) **`boolean eProtetto`** in **`Animale`**?
  - Ok, ma a cosa ci serve la gerarchia sulle specie?
  - E se volessimo anche l'attributo **`Date dataPrimaMinaccia`**?
    - Coesione a istanza mista!

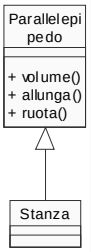


Mettiamoli insieme



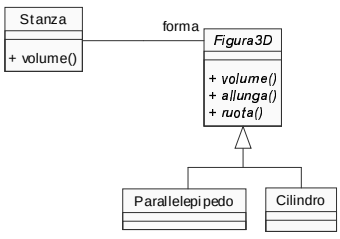
Stanze e parallelepipedi (1/2)

- Una stanza è un parallelepipedo…
- … così eredito **`volume()`** ☺ …
- … ma anche **`allunga()`**, **`ruota()`** ☹
- E se alcune stanze sono cilindriche? ☹☹



Stanze e parallelepipedi (2/2)

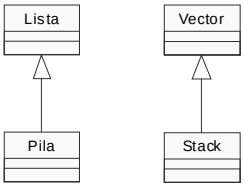
- Una stanza ha un attributo forma che può essere un parallelepipedo



Inoltro di messaggi

- **`volume()`** di **`Stanza`** viene calcolato chiamando **`volume()`** di **`Figura3D`** (o sottoclasse)
- Il messaggio **`volume()`** ad un oggetto di tipo **`Stanza`** viene inoltrato a un oggetto di tipo **`Figura3D`**
- Alternativa di progetto (ad ereditare **`volume()`**) che non garantisce l'accesso a tutti i metodi
  - Pro: posso scegliere
  - Contro: non è automatico, niente polimorfismo

Ereditarietà "sbagliata": un esempio noto…



- Inoltro di messaggi

Scaletta

- Progetto per contratto in Java
- I pericoli di ereditarietà e polimorfismo
  - Gerarchie errate
  - Gestione del polimorfismo
- Genericità
- Ancora sull'interfaccia di una classe
  - Anelli di operazioni
  - Tipologie di stati e di comportamento
- Ancora sulla coesione, di operazione/metodo

Polimorfismo

- Durante l'esecuzione si sceglie automaticamente quale metodo eseguire in corrispondenza a un messaggio
- Polimorfismo delle operazioni
  - Permette la definizione di operazioni con lo stesso nome in classi diverse
- Polimorfismo delle variabili
  - Permette a una variabile di fare riferimento a oggetti di classi diverse

Polimorfismo delle operazioni

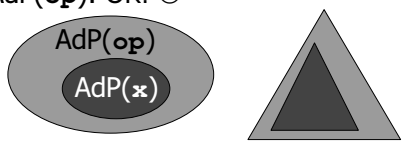
- Def.: Ambito di Polimorfismo (AdP) di un'operazione **`op`** = insieme di classi per cui l'operazione **`op`** è definita
- Di solito l'AdP è un Cono di Polimorfismo: una classe (il Vertice di Polimorfismo) e tutte le sue sottoclassi
- Esempi
  - **`toString()`**, **`area()`** di **`Figura`**, …

Polimorfismo delle variabili

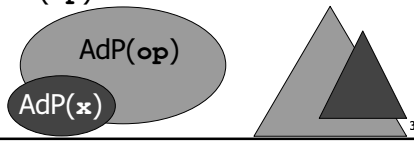
- Def.: Ambito di polimorfismo di una variabile **`x`** = insieme di classi alle quali possono appartenere gli oggetti a cui punta **`x`** durante la sua esistenza
- Anche qui, CdP e VdP
- Esempi
  - **`Figura f; Persona p; Object o;`**

Polimorfismo che funziona

- **`x.op()`**
- AdP(**`x`**) ⊂ AdP(**`op`**): OK! ☺



- AdP(**`x`**) ⊄ AdP(**`op`**): KO! ☹



Esempi

- **`dispositivoDiFabbrica.accendi()`**
  - Di tipo **`DispositivoAccendibile`** (**`Luce`**, **`Motore`**, **`Sirena`**, …): OK ☺
  - Di tipo **`Dispositivo`** (**`Portello`**, **`Tubo`**, **`Serbatoio`**, …), o magari **`Object`**: KO ☹

Scaletta

- Progetto per contratto in Java
- I pericoli di ereditarietà e polimorfismo
  - Gerarchie errate
  - Gestione del polimorfismo
- Genericità
- Ancora sull'interfaccia di una classe
  - Anelli di operazioni
  - Tipologie di stati e di comportamento
- Ancora sulla coesione, di operazione/metodo

Genericità

- Classe parametrica (o "generica", o "template" in C++):
  - Classe che ha come parametro un nome di classe ad ogni creazione di istanza
  - Es.: **`Pila`** di **`int`** vs. **`Pila`** generica

```
class Pila<T> {
  <T>[] elementi;
  int numElementi;
  public Pila(int dim) {
    elementi = new <T>[dim];
    ...
  }
}
```

```
Pila<Float> p;
p=new Pila<Float>(5);
p.push(new Float(0.4));
...
Pila<Integer> p1;
p1=new Pila<Integer>(5);
p1.push(new Integer(4));
```


Genericità in Java

- Dalla J2SE 1.5 (in beta!) [google]
- Generics vs. casting: più sicuro!

```
ArrayList list = new ArrayList();
list.add(0, new Integer(42));
int total = ((Integer)list.get(0)).intValue();
```

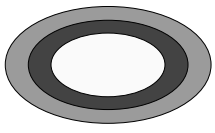
```
ArrayList<Integer> list =  new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = list.get(0).intValue();
```


Scaletta

- Progetto per contratto in Java
- I pericoli di ereditarietà e polimorfismo
  - Gerarchie errate
  - Gestione del polimorfismo
- Genericità
- Ancora sull'interfaccia di una classe
  - Anelli di operazioni
  - Tipologie di stati e di comportamento
- Ancora sulla coesione, di operazione/metodo

Anelli di operazioni

- Operazioni più interne e più esterne
- Operazioni definite in termini di altre operazioni



```
!this.vuota()
```

```
class Pila {
  public void pop(int x){
    if (this.numElem!=0) {
      ...
    }
  }
  public boolean vuota (){
    ...
  }
}
```

Anelli di operazioni

- N.B. Tutte operazioni pubbliche
  - Esterno ≠ pubblico e interno ≠ privato!
  - Tutte possono accedere agli attributi
- Esempio classico: costruttori
  - Un costruttore con tutti gli argomenti che fa tutto
  - Altri costruttori con meno argomenti che chiamano il precedente con argomenti attuali opportuni
- (Attenzione: non supporre l'invariante all'interno dei costruttori…
  - …e attenzione a invocare altri metodi…)

Anelli di operazioni: pro e contro

- Pro
  - Evita duplicazioni di codice
  - Limita le dipendenze
    - Dall'implementazione
    - Dalle sopraclassi
  - Maggiore incapsulamento (modifiche all'implem.)
- Contro
  - Maggiore inefficienza (poca)
  - Serve maggiore attenzione, disciplina

Ancora sull'interfaccia di una classe

- Un oggetto si muove nello spazio degli stati in seguito ai messaggi ricevuti
- In quali stati?
  - Tipologie di stati dell'interfaccia di una classe
- Con quali transizioni?
  - Tipologie di comportamento dell'interfaccia di una classe

Tipologie di stati

- Stati illegali: L'interfaccia dà modo di raggiungere stati che non rispettano l'inv.
- Stati incompleti: L'interfaccia non consente di raggiungere tutti gli stati legali
- Stati non pertinenti: L'interfaccia consente di raggiungere stati che non c'entrano (es.: Pila con `getPenultimo()`)
- Stati ideali: ok

Tipologie di comportamento

- 7 tipologie di comportamento permesse dall'interfaccia di una classe. Comportamenti:
  - Illegali
  - Pericolosi
  - Non pertinenti
  - Incompleti
  - Scomodi
  - Replicati
  - Ideali

Tipologie di comportamento

- Illegali: consente transizioni illegali
  - Es.: `Pila` con operazione `estrai(i)` (estrae l'elemento i-esimo; ottengo sempre una pila…)
  - Sono le transizioni a essere illegali, non gli stati!
- Pericolosi: consente di raggiungere stati ∉ SdS
  - Es.: `Rettangolo` con `spostaVertice()`
  - Qui sono gli stati raggiunti a essere illegali

Tipologie di comportamento

- Non pertinenti: consente comportamenti che non hanno nulla a che fare con la classe
  - Es.: `Cliente` con `calcolaDifferenzaDate()`
  - Fat interface…
- Incompleti: non consente tutti i comportamenti leciti
  - Es.: Ordine nello stato "approvato" che non può tornare in "non approvato". Quando il cliente fallisce…
  - Magari tutti gli stati leciti vengono raggiunti, ma non in ogni momento: mancano transizioni

Tipologie di comportamento

- Scomodi: consente tutti i comportamenti legali, ma per alcuni servono più messaggi
  - Es.: `Alfiere` con `solleva()`, `sposta()` e `posa()`, invece di `muovi()`
- Replicati: lo stesso comportamento può essere ottenuto in più modi
  - Es.: `Robot` con `giraADx()` e `giraVersoDx(gradi)`
  - Non sempre è negativo!

Tipologie di comportamento

- Ideali:
  - Legali e non pericolosi
    - Si possono effettuare solo transizioni legali
    - (Si passa da stati legali a stati legali)
  - Completi
    - Si possono effettuare tutte le transizioni legali
    - (Si possono raggiungere tutti gli stati legali)
  - Pertinenti
    - Non ci sono operazioni non pertinenti
  - Non scomodi
  - Non replicati

Scaletta

- Progetto per contratto in Java
- I pericoli di ereditarietà e polimorfismo
  - Gerarchie errate
  - Gestione del polimorfismo
- Genericità
- Ancora sull'interfaccia di una classe
  - Anelli di operazioni
  - Tipologie di stati e di comportamento
- Ancora sulla coesione, di operazione/metodo

Ancora sulla coesione (1/2)

- Delle operazioni (avevamo visto delle classi!)
- Coesione alternativa ☹
  - Più comportamenti alternativi in un'operazione
  - Es.: In `Rettangolo`, `scalaORuota(fattore,angolo,flag)` (se `flag` è `true` scala, altrimenti ruota)
  - Oppure `scalaORuota(quantita,flag)`
- Coesione multipla ☹
  - Più comportamenti in un'operazione
  - `scalaERuota(fattore,angolo)`

Ancora sulla coesione (2/2)

- Coesione funzionale (o ideale) ☺
  - Un unico comportamento nell'operazione, che fa un'unica cosa
- I nomi sono importanti!
  - "e": coesione multipla
  - "o": coesione alternativa

Conclusione?

- Nell'OOD è raro che ci sia un'"Unica Risposta Giusta e Perfetta"
- Bisogna trovare compromessi
  - A volte la bassa qualità di una classe porta a una qualità maggiore di tutto il resto del progetto
- Abbiamo visto criteri che servono per giudicare, discutere pro e contro di una soluzione

Riassunto

- Progetto per contratto in Java
- I pericoli di ereditarietà e polimorfismo
  - Gerarchie errate
  - Gestione del polimorfismo
- Genericità
- Ancora sull'interfaccia di una classe
  - Anelli di operazioni
  - Tipologie di stati e di comportamento
- Ancora sulla coesione, di operazione/metodo

Bibliografia

- Meilir Page-Jones, *Progettazione a oggetti con UML*, Apogeo, 2002, capp. 12 – 14 (non tutti)
- Google

Criteri di qualità delle classi

- Dipendenze limitate e che non attraversano i domini
- Ingombro di una classe adeguato al dominio a cui ∈
- Coesione di classe ideale
- Conformità di tipo
- Comportamento chiuso
- Interfaccia che supporta stati e comportamento ideali
- Operazioni con coesione funzionale

A che punto siamo

- 1/3 del corso
- Introduzione
- Criteri per buon OOD
- Design pattern
- Refactoring
- OOA (use case, pattern di analisi)
- "Varie" (agenti, casi di studio, …)