OO (Object Oriented): concetti e termini

Stefano Mizzaro

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

PAOO, Lezione 2
22/1/2004

Riassunto

- Introduzione al corso
  - Obiettivi, motivazioni, collegamenti con altri corsi, lezioni, esami, riferimenti bibliografici,…
  - Argomenti e programma (preliminare)
    - Introduzione (OO, UML)
    - OOD (UML, principi, pattern, refactoring)
    - OOA (UML, principi, pattern di analisi)
    - Approfondimenti e casi di studio (UML, buoni diagrammi, estensioni UML… seminari anche esterni)
  - Non si può essere buoni OOP se non si è buoni OOD!!
- Cosa vuol dire OO?

I vostri concetti

? scambio messaggi
+ classi
+ ereditarieta'
× suddivisione dei compiti
+ polimorfismo
+ metodi
× MVC
+ overidding
+ oggetto
+ istanza
+ costruttore
+ riuso
× CRC
+ interfaccia
+ implementazione
+ astrazione
? (??) interazione
× protocollo
× interfaccia grafica
× sincronizzazione
+ incapsulamento (io)

I concetti di Page-Jones

- Incapsulamento
- Occultamento delle informazioni e dell'implementazione
- Conservazione dello stato
- Identità degli oggetti
- Messaggi
- Classi
- Ereditarietà
- Polimorfismo
- Genericità
- (Contratti)

Scaletta

- I (miei) concetti dell'OO:
  - Tipi di Dato Astratti (TDA)
  - Scambio messaggi
  - Ereditarietà
  - Polimorfismo
- OOD e OOA

Cos'è un TDA

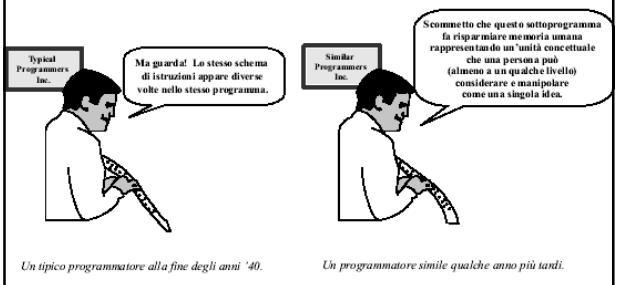
- Nuovo tipo definito dal programmatore
- Tipo = valori + operazioni
  - Tipi predefiniti: ci interessa solo l'esterno!
  - TDA: dovremo definire anche l'interno
- "Astratto": **interfaccia** (l'"esterno") astrae, è separato, da **implementazione** (l'"interno")

Scomposizione funzionale

- Raggruppare istruzioni in una funzione (procedura)
- "Verbi → sottoprogrammi"
- "Estendere il linguaggio con nuove istruzioni"
- Meglio di niente, ma… non funziona!!
  - No riuso
  - Inizializzazioni, input, elaborazione, output
  - Non si adatta ai cambiamenti
  - "Un po' come il DNA" (manca la cellula)

Un approccio "vecchio"

- Il programmatore scompone funzionalmente il problema



Un tipico programmatore alla fine degli anni '40. *Un programmatore simile qualche anno più tardi.*

Scomposizione per TDA

- Raggruppare codice in un nuovo tipo
  - Individuare i tipi di dato che semplificherebbero la scrittura del programma
    - nessun LdP ha tutti i tipi…
  - Definire i nuovi tipi (interfaccia e implementazione)
  - Usare i nuovi tipi
    - (quasi) come se fossero tipi predefiniti
    - astraendo dalla loro implementazione
- "Nomi → tipi"
- "Estendere il linguaggio con nuovi tipi"

TDA: esempi

- Programma di grafica
  - Punti, cerchi, segmenti, …
- Programma per pilota automatico d'aereo
  - Piano di volo, tratta, quota, coordinata, …
- Programma di gestione di un magazzino
  - Magazzino, merce, scaffale, ordine, …
- Programma per robot in un labirinto
  - Robot, griglia, spostamento, traiettoria, …

Il TDA `Punto` in Java

```
class Punto {
  private double x;
  private double y;
  public Punto(double x, double y){
    this.x = x;
    this.y = y;
  }
  public static void setX(Punto p, double x) {p.x=x;}
  public static void setY(Punto p, double y) {p.y=y;}
  public static double getX(Punto p) {return p.x;}
  public static double getY(Punto p) {return p.y;}
}
```


Il TDA `Punto`

- Non è un programma
- È un nuovo tipo di dato ("stampo" per variabili), definito dal programmatore
  - Struttura dati
  - Operazioni (funzioni e procedure)
- Variabili d'istanza (senza `static`)
- Il costruttore
- Il `this`

Programma che usa il TDA **Punto**

```
class UsaPunto {
  public static void main(String[]args) {
    Punto p;
    p = new Punto(12.0, 34.9);
    Punto q = new Punto(0, 0);
    p = new Punto(2.3, 3.4);
    Punto.setX(p, 2.5);
    double x = Punto.getX(p);
    p = q;  // Alias
  }
}
```


Considerazioni

- Chi deve creare/usare punti ha il compito facilitato
- **new**
- Il **main** non accede a **x** e **y** (né può farlo!)
- **private**: astrazione
- Classe **Cerchio**:
  - Non solo "usa" **Punto**
  - Ogni **Cerchio** contiene un **Punto** (il centro)

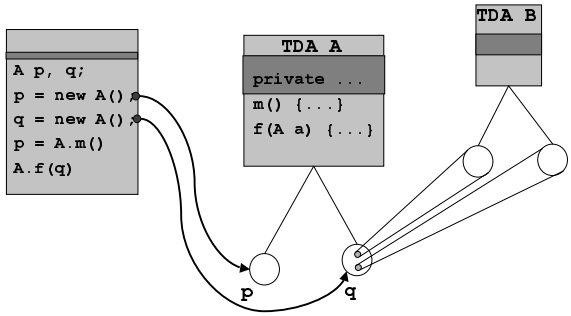
Programma con TDA (1/2)

```
class C {
  ... main ... {
    A p, q;
    p = new A();
    q = new A();
    A.m();
    p = A.f(q);
  }
}
```

```
class A {
  ... private ...
  ... public  ...
  ... void  m() {...}
  ... A f(A a) {...}
}
```

```
class C {
class B {
  ... private ...
  ... public  ...
  ... void  m() {...}
  ... B f(B b) {...}
}
```


Programma con TDA (2/2)



L'incapsulamento

- Occultamento delle informazioni e dell'implementazione (information & implementation hiding)
  - Nascondere l'implementazione, far vedere solo l'interfaccia (funzioni, "manopole" per manipolare i valori del TDA)
  - Gli oggetti del nostro mondo sono fatti così (auto, impianto elettrico, …)
- Information hiding e programmazione strutturata

Vantaggi dell'incapsulamento

- (ovvio: 1 sola variabile invece di tante)
- Scomposizione del problema
- Separazione fra interfaccia e implementazione:
  - Modificabilità (no "effetto domino")
  - Semplicità d'uso
  - Coerenza (consistency): cerchi di raggio = 1…
  - Robustezza
  - Implementazione nascosta ⇒ non documentata…
- Un TDA è "adatto all'ambiente", sopravvive, come una cellula…

Riassunto: TDA

- Il programmatore
  - definisce nuovi tipi (interfaccia e implementazione) nascondendone l'implementazione e poi usa i nuovi tipi
  - No scomposizione funzionale!
- Il Java "ha i TDA", il C no!
  - **`class`**, **`public`**, **`private`**, costruttore, **`new`**, notazione puntata, …
  - Unitarietà:
    - Definiti in un unico punto del programma
    - Gestione unitaria dei valori
  - Inaccessibilità: impossibile accedere all'implementazione
- > modificabilità, estendibilità, riuso, …

Nota storica

- TDA e incapsulamento: anni '60
- OOP: anni '80
- Incapsulamento:
  - fondamentale per OO!
  - non inventato da OO!!

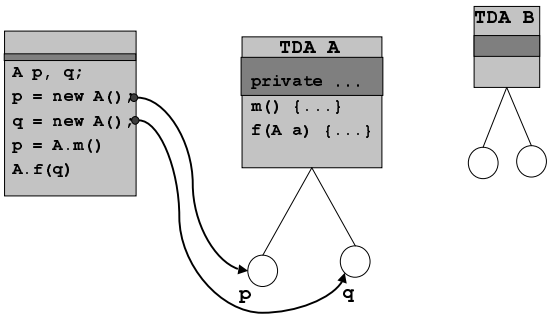
Conservazione dello stato e identità degli oggetti…

- Conservazione dello stato
  - Mah…
  - Ogni variabile mantiene i suoi valori
  - (Quando si invoca un sottoprogramma, no)
- Identità degli oggetti
  - Mah…
  - Ogni oggetto è una zona di memoria sua
  - Può avere un nome
  - Maniglia, riferimento
  - …

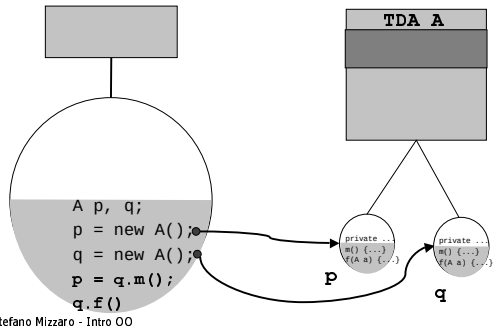
Secondo ingrediente dell'OO: lo scambio messaggi

- DaTDA e funzioni/procedure definite nel TDA a oggetti attivi che si scambiano messaggi
- Da "esegui il metodo **`getX()`** della classe **`Punto`** con argomento **`q`**":
  - **`double a = Punto.getX(q);`**
- a "manda il messaggio **`getX`** all'oggetto **`q`**":
  - **`double a = q.getX();`**
- "Approccio TDA": variabili contengono stato e basta
- "Approccio OO": oggetti contengono stato e metodi

Programma con TDA (di nuovo)



Programma con scambio messaggi



La classe `Punto` in Java

```
class Punto {
  private double x;
  private double y;
  public Punto(double x, double y){
    this.x = x;
    this.y = y;
  }
  public void setX(double x) {this.x = x;}
  public void setY(double y) {this.y = y;}
  public double getX() {return x;}
  public double getY() {return y;}
}
```


Programma che usa `Punto`

```
class UsaPunto {
  public static void main(String[]args) {
    Punto p;
    p = new Punto(12.0, 34.9);
    Punto q = new Punto(0, 0);
    p = new Punto(2.3, 3.4);
    p.setX(2.5);
    double x = p.getX();
    p = q;  // Alias
  }
}
```


Confronto: TDA vs. OO

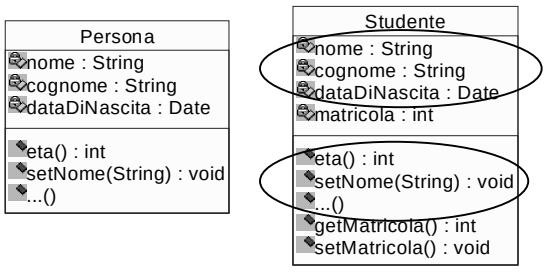
- Filosoficamente:
  - variabili passive vs. oggetti attivi
- In pratica: un parametro in meno
  - **`public static void setX(Punto p, double x)`** vs. **`public void setX(double x)`**
  - **`Punto.getX(q)`** vs. **`q.getX()`**
- **`static`**
- Terminologia:
  - Attributi
  - Metodi

Cosa fa il main

- Inizia l'esecuzione
- Crea oggetti
- Manda messaggi a oggetti (invoca metodi d'istanza)
- Invoca direttamente metodi **`static`**

Terzo ingrediente: ereditarietà

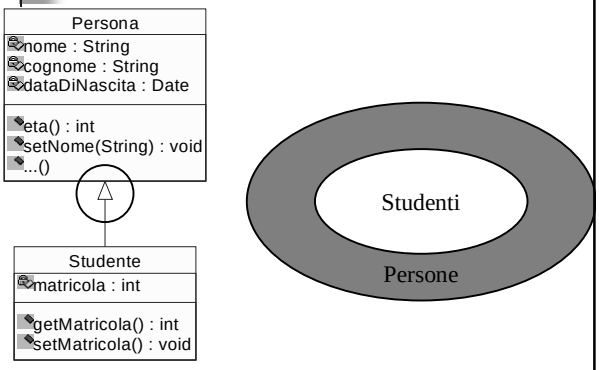
- La classe **`Persona`**…
- … e la classe **`Studente`**



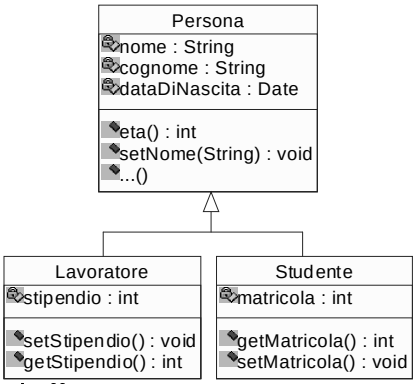
Un problema

- Duplicazione di codice
- Fatica inutile
- Pericolo di incoerenze
- Soluzione: ereditare!
  - **`Studente`** eredita da **`Persona`** attributi e metodi (e ne aggiunge, sovrascrive, …)

Persona e Studente



... e Lavoratore



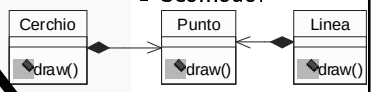
Ereditarietà in Java

- Parola riservata **`extends`**
- **`class Studente extends Persona {`**
- ... e **`Studente`** ha metodi e attributi di **`Persona`**
- Sovrascrittura: nella sottoclasse posso specializzare il comportamento
  - N.B. Sovrascrittura (*overriding*) ≠ sovraccarico (*overloading*)
    - Sovraccarico: nome =, firme ≠
    - Sovrascrittura: nome =, firme =

Polimorfismo (1/2)

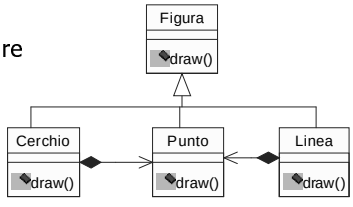
```
Punto[] punti;
Cerchio[] cerchi;
...
if (x instanceof Punto)
  punti[i] = x;
else if (x instanceof Cerchio)
  cerchi[i] = x;
else if ...
for (int i = ...){
  punti[i].draw();
  cerchi[i].draw();
  ...
}
```

- Programma di grafica
- Struttura dati per tutte le figure
- Tanti array...
- Scomodo!



Polimorfismo (2/2)

- È più comodo parlare alla classe base!
- Si può fare!

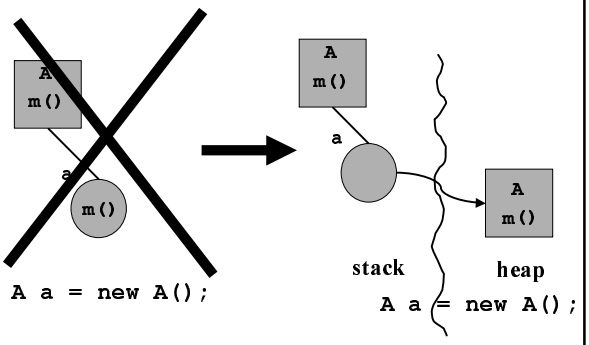



```
Figura[] figure = new Figura[100];
figure[i] = new Punto();
figure[j] = new Cerchio();
for (int k = ...)
  figure[k].draw();
```

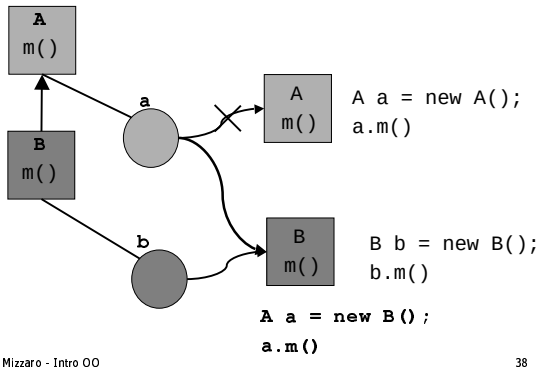

Perché il polimorfismo funziona

- N.B. Polimorfismo va combinato con:
  - eredità (**`Figura`** è sopraclasse) e
  - sovrascrittura (**`draw()`** è sovrascritto)
- Maniglie
  - Le variabili non contengono gli oggetti
  - Le variabili contengono il riferimento (la maniglia) agli oggetti

Le maniglie



Maniglie ed eredità



Terminologia

- Late binding, run-time binding
  - (durante l'esecuzione)
- Non early binding, compile-time binding
  - (durante la compilazione)
- Dynamic method lookup
- Sovrascrittura, sovraccarico
- Eckel:
  "If it isn't late binding, it isn't poliymorphism"

Riassunto: cosa è la OOP?

- TDA
  - Incapsulamento, interfaccia, implementazione
  - Composizione, uso
- Scambio messaggi
  - Oggetti attivi che si "parlano"
  - Metodi/attributi d'istanza e di classe
- Ereditarietà
  - Estensione, sovrascrittura, sottotipo, …
- Polimorfismo
  - Maniglie, late binding, "talk to the base class"

OOD e OOA

- Confine labile
  - Coplien: per fare una buona analisi bisogna pensare agli strumenti esistenti
- Forse ancora più sfuocato in OO
  - Pretesa di rendere "isomorfi" lo spazio del problema e lo spazio delle soluzioni
  - Processo di sviluppo iterativo, A, D e P ripetuti…
- Ha senso didatticamente
- Ci torneremo

Riassunto

- I (miei) concetti dell'OO:
  - Tipi di Dato Astratti (TDA)
  - Scambio messaggi
  - Ereditarietà
  - Polimorfismo
- OOD e OOA