18 - Classi parzialmente definite: Classi Astratte e Interfacce

Programmazione e analisi di dati
Modulo A: Programmazione in Java

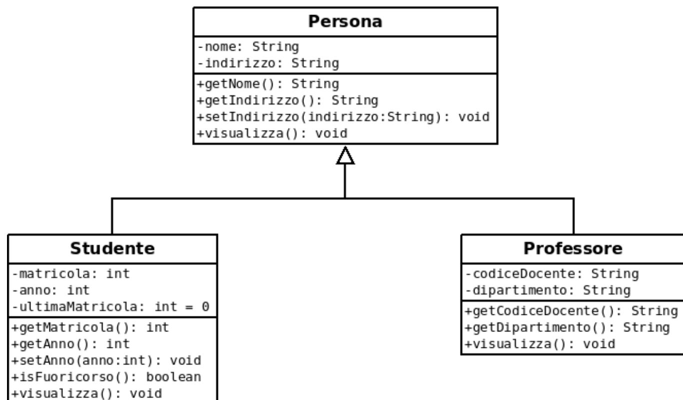
Paolo Milazzo

Dipartimento di Informatica, Università di Pisa
http://www.di.unipi.it/∼milazzo
milazzo@di.unipi.it

Corso di Laurea Magistrale in Informatica Umanistica
A.A. 2013/2014

Ereditarietà come meccanismo di astrazione (1)

L'esempio delle classi `Persona-Studente-Professore` mostra come usando l'ereditarietà si possono raggruppare informazioni comuni a più classi.



Ereditarietà come meccanismo di astrazione (2)

Ogni tanto, però, questo meccanismo di raggruppamento delle informazioni comuni a più classi porta ad ottenere classi che non sono ben definite...

Supponiamo di avere le seguenti classi `Sfera` e `Cubo`

```
public class Sfera {

    private double raggio;
    private double pesoSpecifico;

    public Sfera(double raggio, double ps) {
        this.raggio = raggio;
        pesoSpecifico = ps;
    }

    public double volume() {
      return 4/3 * Math.PI * Math.pow(raggio,3);
    }

    public double superficie() {
      return 4 * Math.PI * raggio * raggio;
    }

    public double peso() {
      return pesoSpecifico * volume();
    }
}
```

Ereditarietà come meccanismo di astrazione (3)

```
public class Cubo {

    private double lato;
    private double pesoSpecifico;

    public Cubo(double lato, double ps) {
        this.lato = lato;
        pesoSpecifico = ps;
    }

    public double volume() {
      return Math.pow(lato,3);
    }

    public double superficie() {
      return 6*lato*lato;
    }

    public double peso() {
      return pesoSpecifico * volume();
    }
}
```

Ereditarietà come meccanismo di astrazione (4)

Le classi `Sfera` e `Cubo` hanno diverse cose in comune

Potremmo quindi pensare di creare una classe `Solido` che raggruppa i membri comuni di `Sfera` e `Cubo`

La classe `Solido` dovrebbe contenere:

- la variabile `pesoSpecifico`
- il metodo `peso()` identico nelle due classi (e che quindi potrebbe essere ereditato senza overriding)
- i metodi `volume()` e `superficie()`, per due motivi:
  - tutti i solidi hanno un volume e una superficie
  - il metodo `peso()` invoca `volume()`

Ma come si calcolano superficie e volume di un solido generico?

- non si può fare... ogni solido ha le sue formule...

Classi Astratte (1)

Soluzione: Definire i `volume()` e `superficie()` come metodi astratti

- Un metodo astratto è un metodo che prevede solo una intestazione, ma che non è implementato
- Una classe che contiene (almeno) un metodo astratto è una classe astratta

Classi Astratte (2)

Ecco la classe astratta Solido:

```
public abstract class Solido { // classe astratta

    private double pesoSpecifico;

    public Solido(double ps){ pesoSpecifico = ps; }

    public double peso (){
      return volume() * pesoSpecifico;
    }

    public abstract double volume(); // metodo astratto

    public abstract double superficie(); // metodo astratto
}
```

Classi Astratte (3)

Nella classe astratta `Solido`

- Viene definita tramite il modificatore `abstract` (prima di `class`)
- Il modificatore `abstract` viene usato anche nei metodi astratti
- I metodi astratti consistono della sola intestazione seguita da `;`
- Nel metodo `peso()` si può richiamare `volume()` anche se è astratto

Una classe astratta

- non può essere usata per creare oggetti (la classe non è completa...)

```
Solido x = new Solido()  // ERRORE
```

- può solo essere estesa da un'altra classe che ne definisce i metodi astratti (tramite overriding)
- può prevedere costruttori che saranno richiamati dalle sottoclassi (tramite `super`)

Classi Astratte (4)

Ridefinamo le classi `Sfera` e `Cubo`

- Aggiungiamo anche un metodo `toString()`

```
public class Sfera extends Solido {

    private double raggio;

    // pesoSpecifico e'  definito nella superclasse

    public Sfera(double raggio, double ps) {
        super(ps);
        this.raggio = raggio;
    }

    public double volume() {
      return 4/3 * Math.PI * Math.pow(raggio,3);
    }

    public double superficie() {
      return 4 * Math.PI * raggio * raggio;
    }

    // peso() e' definito nella superclasse

    public String toString(){
      return "Sfera ("+ raggio +")";
    }
}
```

Classi Astratte (5)

```
public class Cubo extends Solido {

    private double lato;

    // pesoSpecifico e'  definito nella superclasse

    public Cubo(double lato, double ps) {
        super(ps);
        this.lato = lato;
    }

    public double volume() {
      return Math.pow(lato,3);
    }

    public double superficie() {
      return 6*lato*lato;
    }

    // peso() e' definito nella superclasse

    public String toString(){
      return "Cubo ["+ lato +"]";
    }
}
```

Classi Astratte (6)

E ora un main...

```
import java.util.Scanner;

public class UsaSolido {
  public static void main(String args[]) {
  Scanner input = new Scanner(System.in);

  System.out.println("Vuoi creare una sfera o un cubo (s/c)?");
  char scelta = input.nextLine().charAt(0);

  if (scelta=='s' || scelta=='c') {
        System.out.print("Peso Specifico? ");
        double ps = input.nextDouble();

        Solido sol; // variabile di tipo Solido

        if (scelta == 's') {
            System.out.print("Raggio? ");
            double raggio = input.nextDouble();
            sol = new Sfera(raggio, ps);
        } else {
            System.out.print("Lato? ");
            double lato = input.nextDouble();
            sol = new Cubo(lato, ps);
        }
        System.out.println( "Ho creato un solido " + sol  +
                            " con volume " + sol.volume() +
                            " e peso " + sol.peso() );
    }
  }
```

Dalle Classi Astratte alle Interfacce (1)

Abbiamo visto che le classi astratte includono metodi astratti

- una classe astratta è definita solo parzialmente

Esasperando l'uso dei metodi astratti potremmo avere una classe astratta dotata solo di metodi astratti

```
// descrive prodotti commerciabili
public abstract class ProdottoPrezzato {

  // restituisce una descrizione del prodotto
  public abstract double getDescrizione();

  // restituisce il prezzo del prodotto
  public abstract double getPrezzo();
}
```

Dalle Classi Astratte alle Interfacce (2)

Una classe che consiste di soli metodi astratti (o anche costanti) può essere meglio descritta tramite una interfaccia

```
// non e' una classe... e' una interfaccia
public interface ProdottoPrezzato {

  // restituisce una descrizione del prodotto
  public String getDescrizione();

  // restituisce il prezzo del prodotto
  public double getPrezzo();
}
```

Interfacce (1)

Un'interfaccia si definisce in maniera simile a una classe astratta, ma:

- è costituita da soli metodi astratti (è completamente non definita)
- non usa il modificatore `abstract`
- usa la parola chiave `interface` al posto di `class`

Inoltre, una classe che implementa i metodi dell'interfaccia deve usare la parola chiave `implements` invece che `extends`

```
public class DVD extends ProdottoPrezzato {  // NO!!!
```

```
public class DVD implements ProdottoPrezzato {  // OK!!!
```

Interfacce (2)

Un esempio di uso:

```
public class DVD implements ProdottoPrezzato {

  private String descrizione;
  private static double PREZZO = 19.90;

  public DVD(String descrizione) {
    this.descrizione = descrizione;
  }

  public String getDescrizione() {
    return descrizione;
  }

  public double getPrezzo() {
    return PREZZO;
  }

}
```

Interfacce (3)

Un altro esempio:

```
public class Farina implements ProdottoPrezzato {

  private double prezzoAlKg;
  private double peso = 0.0

  public Farina(double prezzoAlKg) {
    this.prezzoAlKg = prezzoAlKg;
  }

  public String getDescrizione() {
    return "Farina";
  }

  public void setPeso(double peso) {
    if (peso>0) this.peso = peso;
  }

  public double getPrezzo() {
    return prezzoAlKg*peso;
  }
}
```

Interfacce (4)

Ma... che vantaggio offrono le interfacce rispetto alle classi astratte?

- Una classe può estendere una sola classe (astratta o meno)
- Una classe può implementare tante interfacce

Ad esempio, consideriamo un'altra interfaccia:

```
public interface ProdottoPesabile {

  public void setPeso(double peso);

}
```

Interfacce (5)

Possiamo modificare `Farina` indicando anche l'interfaccia `ProdottoPesabile` dopo `implements`

```
public class Farina implements ProdottoPrezzato, ProdottoPesabile {

  private double prezzoAlKg;
  private double peso = 0.0

  public Farina(double prezzoAlKg) {
    this.prezzoAlKg = prezzoAlKg;
  }

  public String getDescrizione() {
    return "Farina";
  }

  public void setPeso(double peso) {
    if (peso>0) this.peso = peso;
  }

  public double getPrezzo() {
    return prezzoAlKg*peso;
  }
}
```

Interfacce (6)

Come si usano gli oggetti di classi che implementano determinate interfacce?

Esempio: La classe `Scontrino` usa oggetti che implementano `ProdottoPrezzato`

```
public class Scontrino {

  ... // altri membri

  public void aggiungiProdotto(ProdottoPrezzato x) {
    ...
    this.totScontrino += x.getPrezzo();
    ...
  }

}
```

Interfacce (7)

Altro esempio: La classe `Bilancia` usa oggetti che implementano `ProdottoPesabile`

```
public class Bilancia {

  ... // altri membri

  public void pesa(ProdottoPesabile x) {
    ...
    double peso = leggipeso(); // metodo ausiliario
    x.setPeso(peso);
    ...
  }

}
```

Interfacce (8)

Grazie al fatto che una classe può implementare più interfacce, nell'esempio abbiamo che

- gli oggetti della classe `Farina` possono essere usati sia da `Scontrino` che da `Bilancia`

Gli oggetti della classe `DVD` invece possono essere usati solo da `Scontrino`

Per concludere, i meccanismi dell'ereditarietà (`extends`) e delle interfacce (`implements`) possono essere usati insieme

```
public class Salame extends Affettato
                   implements ProdottoPesabile, ProdottoPrezzato {

  ......

}
```