Metodologie di Programmazione

Lezione 29: Design pattern (parte 1)

Lezione 29: Sommario

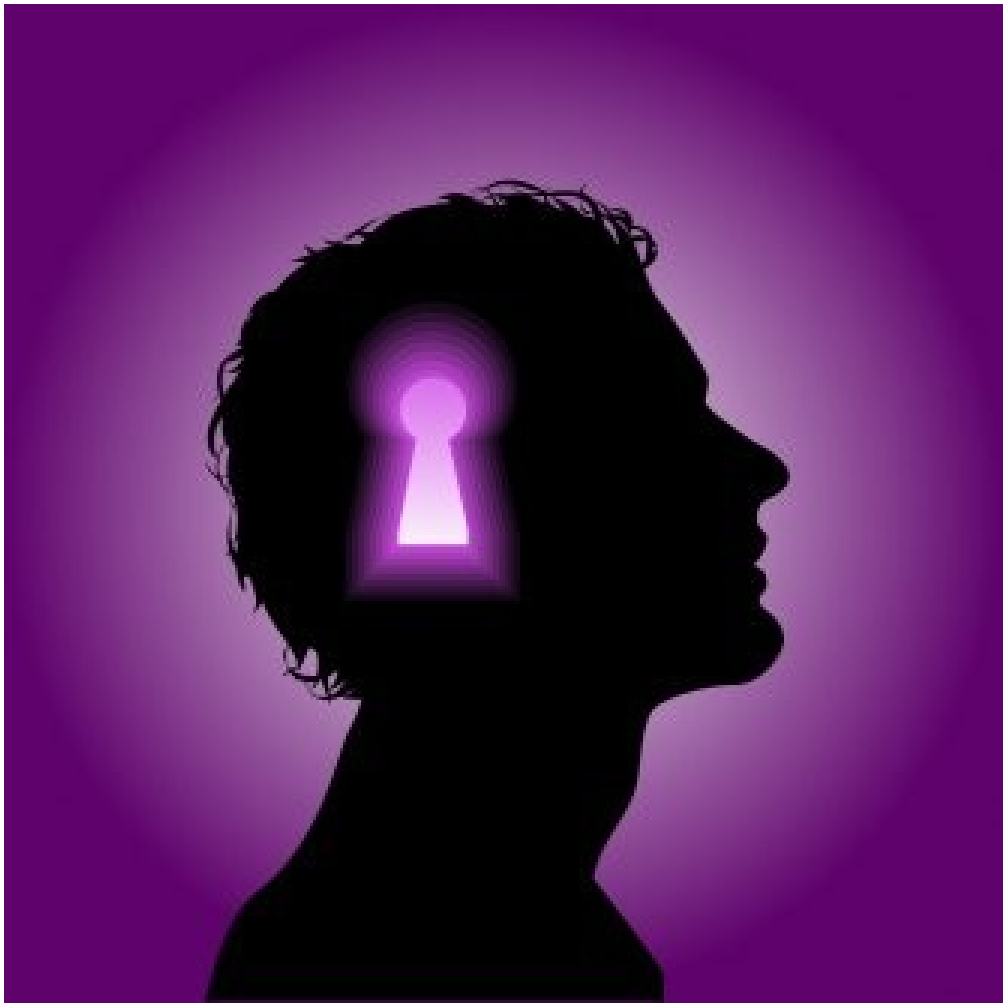
UNITELMA SAPIENZA

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

- Che cos'è un design pattern
- Strategy pattern

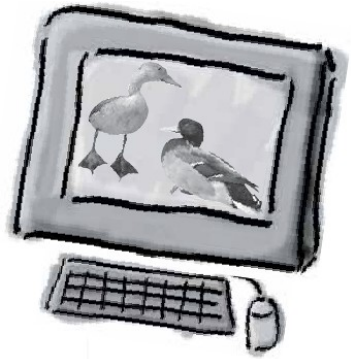
This is all about opening your mind!

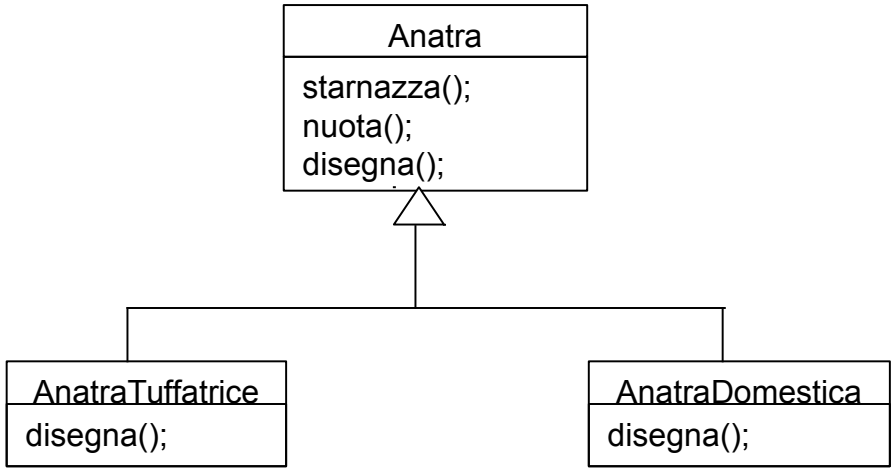
UNITELMA SAPIENZA
SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

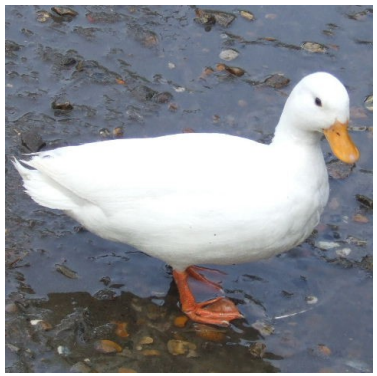


Iniziamo con un caso concreto: un Simulatore di Anatre

- Hai appena sviluppato un fantastico simulatore di anatre: IDuck
- Hai strutturato le classi in questo modo:

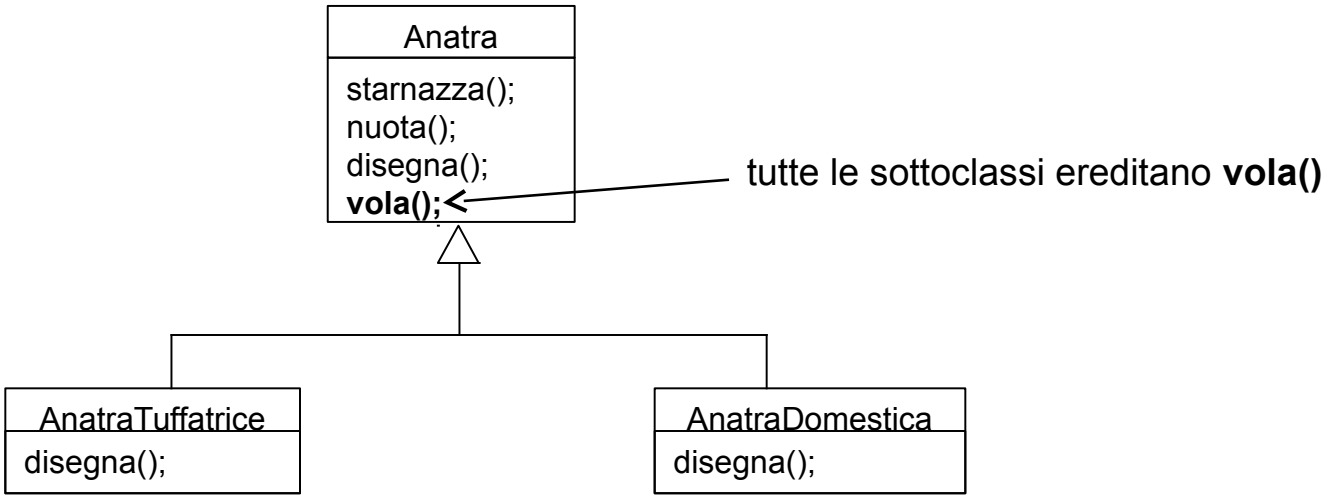






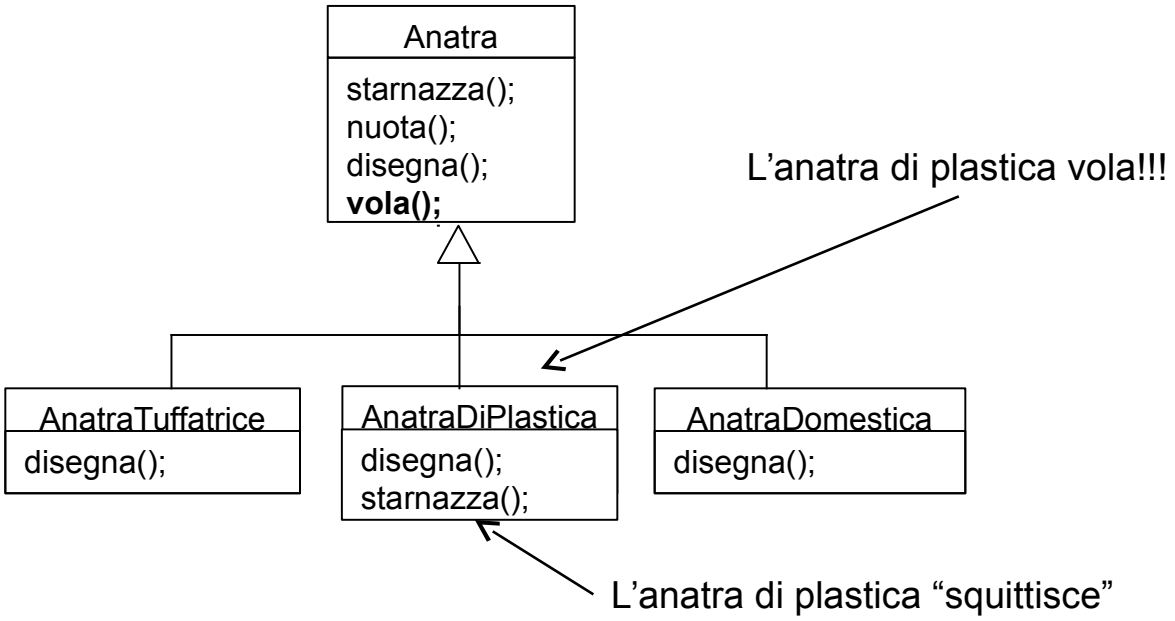
Ma ora vogliamo aggiungere il volo!

- Il cliente decide che il simulatore deve permettere alle anatre di volare
- Stiamo programmando a oggetti: quanto può essere difficile?



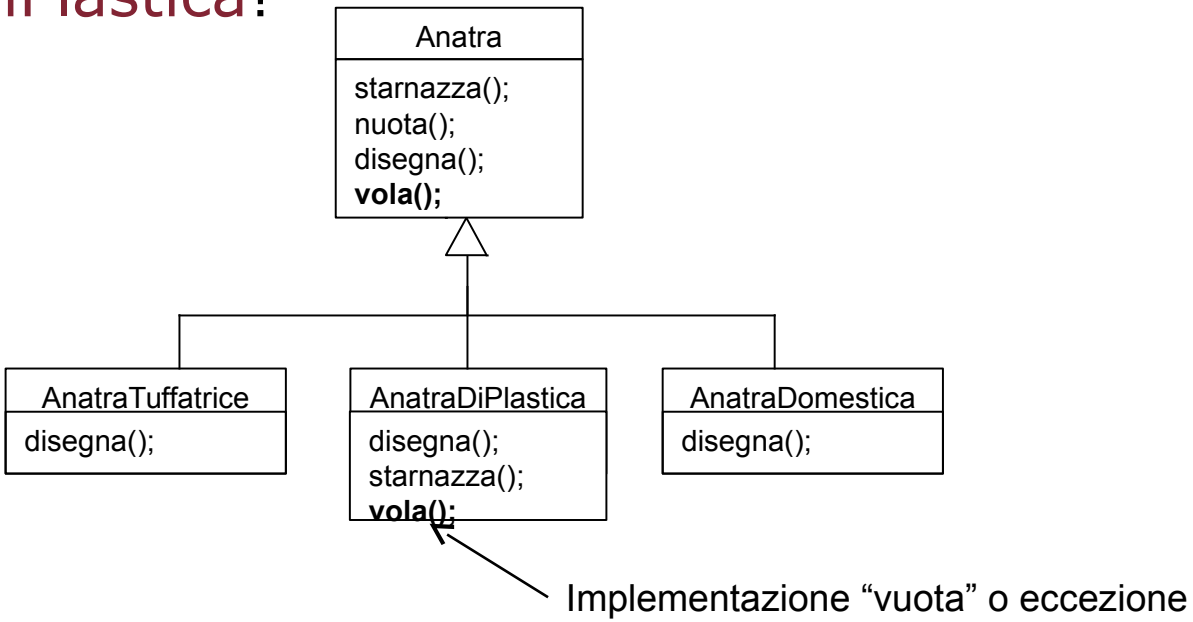
Ma qualcosa può andare male!

- Anche le anatre di plastica volano!
- Nell'aggiungere il metodo vola non abbiamo pensato al fatto che non tutte le anatre volano
- L'ereditarietà è utile per il riuso, ma meno per la manutenzione!



Come risolvere il problema?

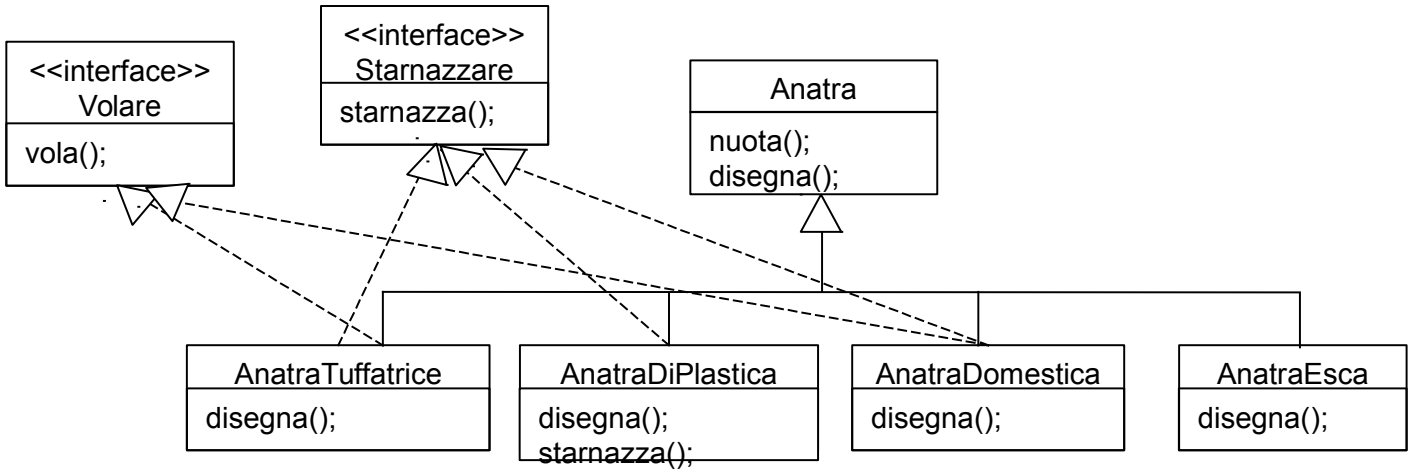
- **Idea 1:** Sovrascriviamo il metodo vola nell'AnatraDiPlastica!



- Ma se aggiungiamo un altro tipo di anatra, l'AnatraEsca?
- Le anatre esca sono esche: non volano e non starnazzano...

E se usassimo le interfacce?

- **Idea 2:** implementiamo un'interfaccia per ogni comportamento dell'anatra
- L'ereditarietà non è sufficiente, proviamo con le interfacce…



- E' una buona scelta?

Codificare i comportamenti mediante interfacce

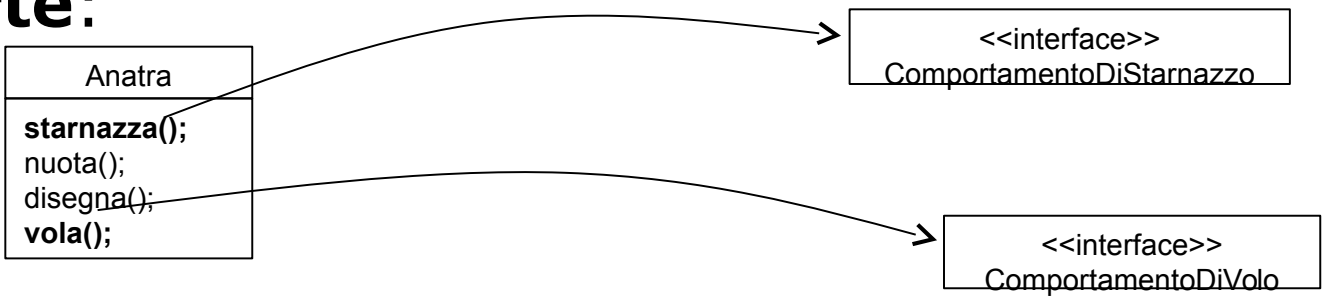
- Una buona scelta: permette a ciascuna sottoclasse di implementare i comportamenti che effettivamente essa deve modellare

- Una cattiva scelta: distrugge ogni possibile **RIUSO** del codice, perché dobbiamo reimplementare le interfacce per ogni sottoclasse

Abbiamo bisogno di un Design Pattern!

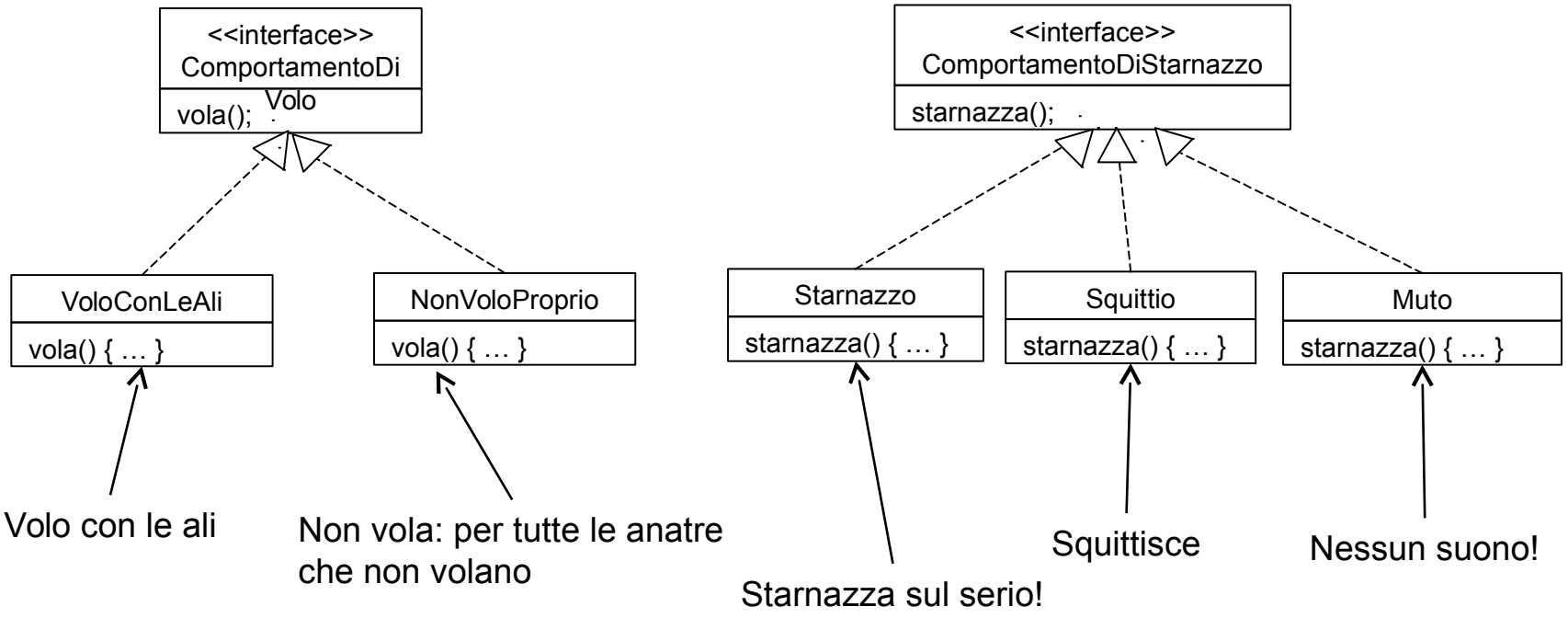
Principio di design: identifica gli aspetti della tua applicazione che variano e separali da quelli che rimangono uguali

- Come separarli? **Incapsulandoli**
- In questo modo tali parti del sistema varieranno in modo indipendente dalle altre parti
- Eliminiamo i metodi corrispondenti ai comportamenti che vogliamo **modellare a parte**:



Modellare i comportamenti di una classe

- Progettiamo un'interfaccia per ogni comportamento
- Per ogni possibile tipo di comportamento implementiamo l'interfaccia



Integrare i comportamenti nella classe

- La classe Anatra ora **delega** i suoi comportamenti di volo e starnazzo, invece di implementarli direttamente

PRIMA

Anatra

starnazza();
nuota();
disegna();
vola();

DOPO

Anatra
ComportamentoDiVolo compVolo; **ComportamentoDiStarnazzo**

nuota();
disegna();
effettuaVolo() { compVolo.vola(); }
effettuaStarnazzo()
{ compStarnazzo.starnazza(); }

- Dove impostare il comportamento di volo specifico per ciascuna sottoclasse di Anatra?
- Nel costruttore di ciascuna sottoclasse:

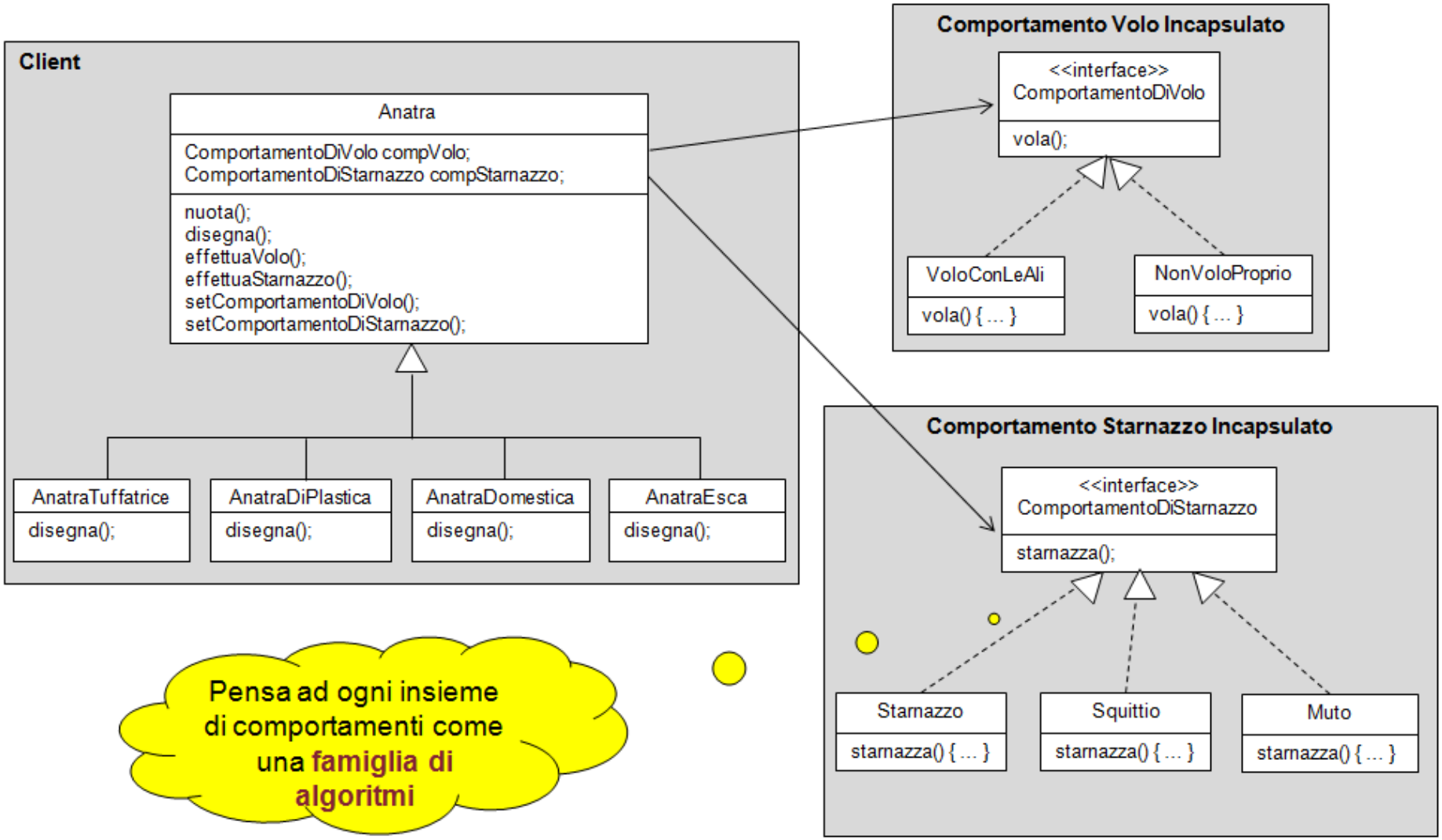
```
public class AnatraDomestica extends Anatra
{
  public AnatraDomestica()  { compVolo = new VoloConAli();  compStarnazzo = new Starnazzo(); }
  }
```

Metodi "personalizzabili"

- Grazie all'incapsulamento separato dei comportamenti, abbiamo in realtà dei metodi "personalizzabili"
- Ad esempio, potremmo aggiungere alla classe Anatra i metodi:

```
public void setComportamentoDiVolo(ComportamentoDiVolo c)
 {
    compVolo = c;
 }
public void setComportamentoDiStarnazzo(ComportamentoDiStarnazzo c)
 {
    compStarnazzo = c;
 }
```

Strategy Pattern: The Big Picture



Che principio abbiamo utilizzato?

Principio di design:
preferisci la composizione all'ereditarietà

- Invece di ereditare il comportamento, le anatre ottengono il loro comportamento mediante una composizione di oggetti di comportamento
- Implementa il principio della delega (delegation), spostando la responsabilità sulla classe delegata

Your first Design Pattern: the Strategy Pattern!

Il mio primo Design Pattern

Esercizio: Camminare

UNITELMA SAPIENZA

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

```
abstract public class EssereVivente
{
    abstract public void cammina();
}

public class Animale extends EssereVivente
{
    @Override
    public void cammina()
    {
        System.out.println("Sto camminando a 4 zampe...");
    }
}


public class Millepiedi extends Animale
{
    @Override
    public void cammina()
    {
        System.out.println("Sto camminando a 62 zampe...");
    }
}
```

```
public class Pianta extends EssereVivente
{
    @Override
    public void cammina()
    {
        System.out.println("Affonda le radici...");
    }
}


public class Uomo extends EssereVivente
{
    private int eta;

    @Override
    public void cammina()
    {
        if (eta >= 80) System.out.println("cammina con bastone");
        else if (eta >= 3) System.out.println("cammina su 2 arti");
        else System.out.println("cammina su 4 arti");
    }

    public int getEta() { return eta; }
    public void invecchia() { eta++; }
}
```

Esercizio: Array comparabile in modo flessibile

- Si progetti una classe generica che modella un array di elementi
- La classe, mediante lo strategy pattern, permette di modificare il modo di comparare gli array:
  - In ordine lessicografico (dal minore al maggiore)
  - In ordine lessicografico inverso (dal maggiore al minore)
  - Per dimensione degli array

Cos'è un Design Pattern

- E' un modo di **PENSARE** il codice e la struttura a oggetti
- I pattern sono il "succo" dei principi di progettazione orientata agli oggetti
  - I Design Pattern non vanno direttamente nel codice
- Vanno **prima** nella tua mente: