OOP
Parte 4

Interfacce e polimorfismo

INTERFACCE

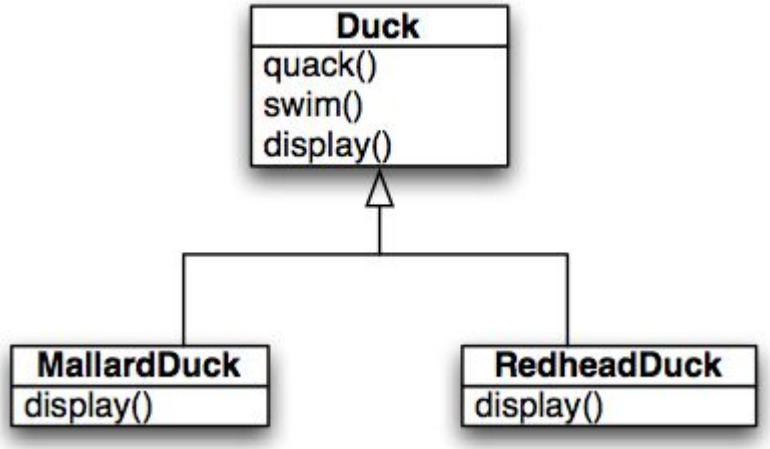
SIMULATORE DI PAPERE

SIMULATORE DI PAPERE

Ci danno il compito di scrivere un simulatore di papere che nuotano e fanno quack.

Questo è come modelliamo la situazione.

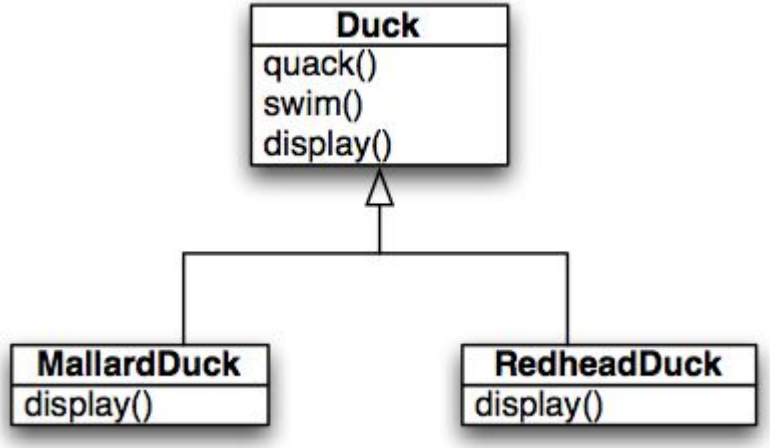
Sviluppiamo un software che utilizzi questo modello.



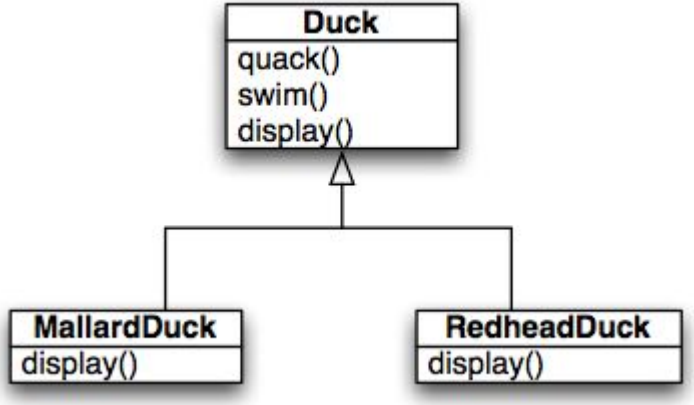
SIMULATORE DI PAPERE

Ora ci chiedono di considerare il fatto che tutte le papere volano.

Come possiamo risolvere?

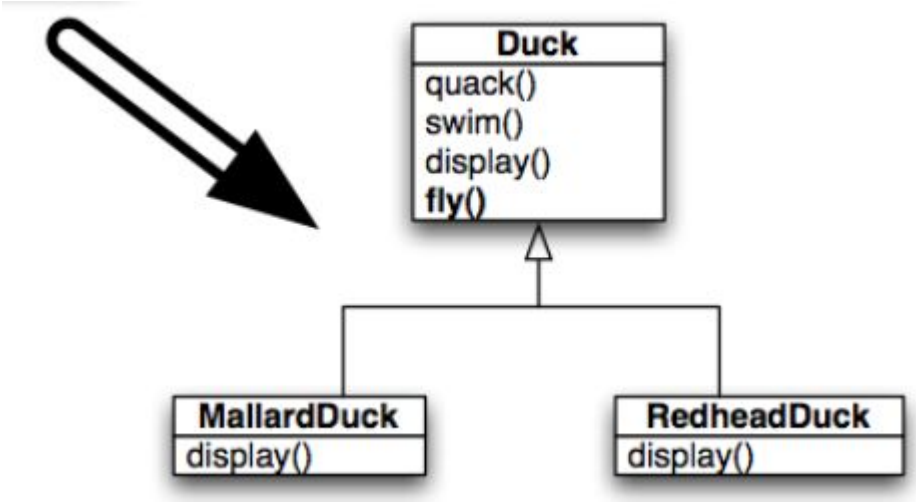


SIMULATORE DI PAPERE



Facile!!! Sfruttiamo l'ereditarietà.
Aggiungiamo il metodo **fly()** nella classe base.

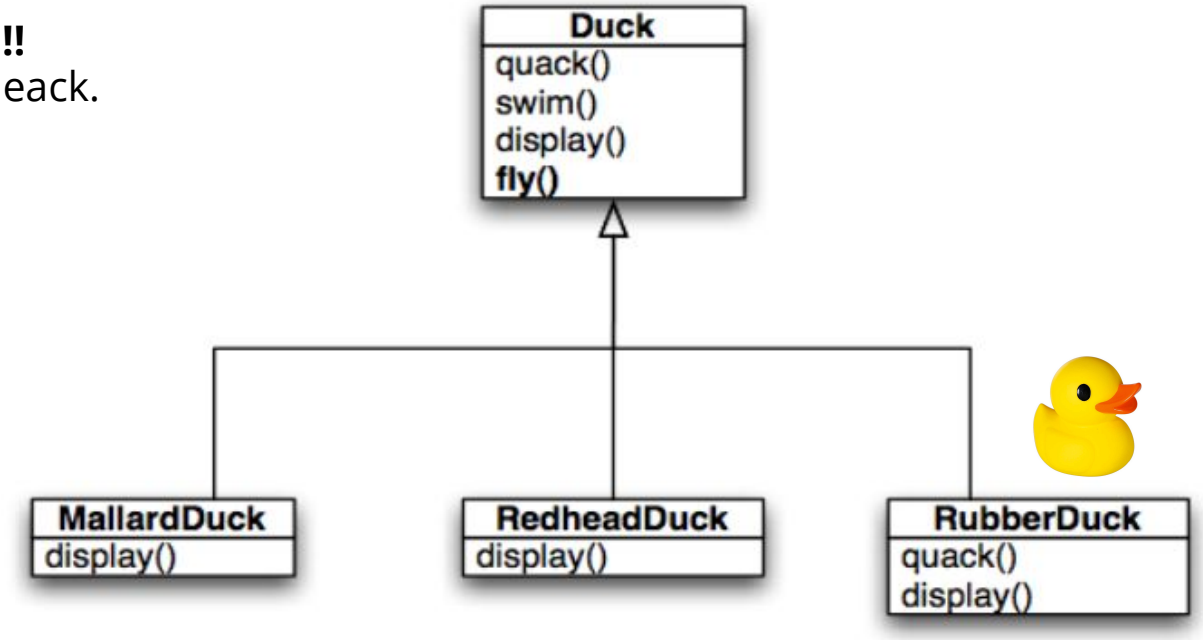
In questo modo tutte le papere posseggono la capacità di volare.



SIMULATORE DI PAPERE

Nel frattempo si aggiunge alle papere anche la papera di gomma (**RubberDuck**).

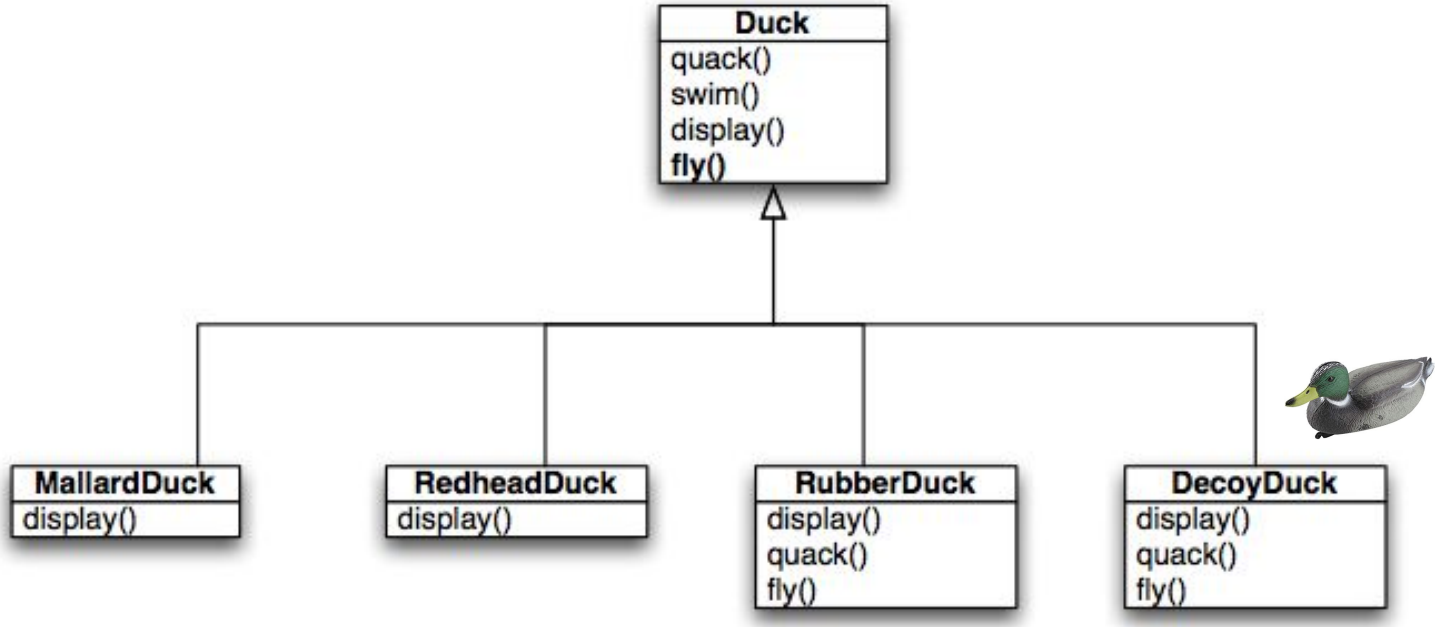
Ora c'è un problema!
Le **RubberDuck NON volano!!!**
E poi non fanno quack ma squeack.



SIMULATORE DI PAPERE

Altro problema: si aggiunge la **DecoyDuck** che è una papera finta, per decorazione. Ovviamente questa papera non vola e non fa versi.

Risolviamo facendo **override** del metodo **fly()** nelle papere che non volano in modo da non fare far niente a quelle sottoclassi e facciamo **override** anche di **quack()** così da cambiare il comportamento delle papere che fanno un verso diverso dalle altre.

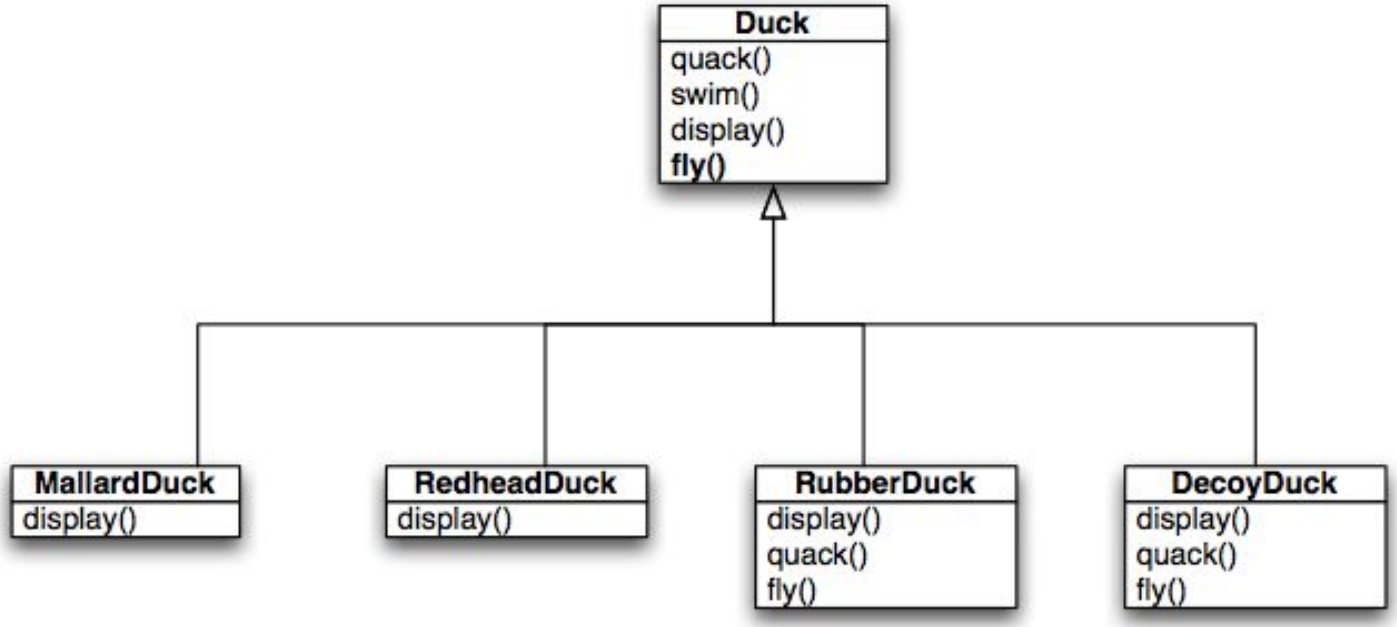


SIMULATORE DI PAPERE

La soluzione funziona ma è pessima, sia dal punto di vista del design che della manutenzione futura.

In questo modo chiamare il metodo RubberDuck.fly() non produrrà nessuna azione, stessa cosa per la DecoyDuck.

Che brutto quando una classe offre un metodo ma poi questo metodo non fa niente!!!



Interfacce

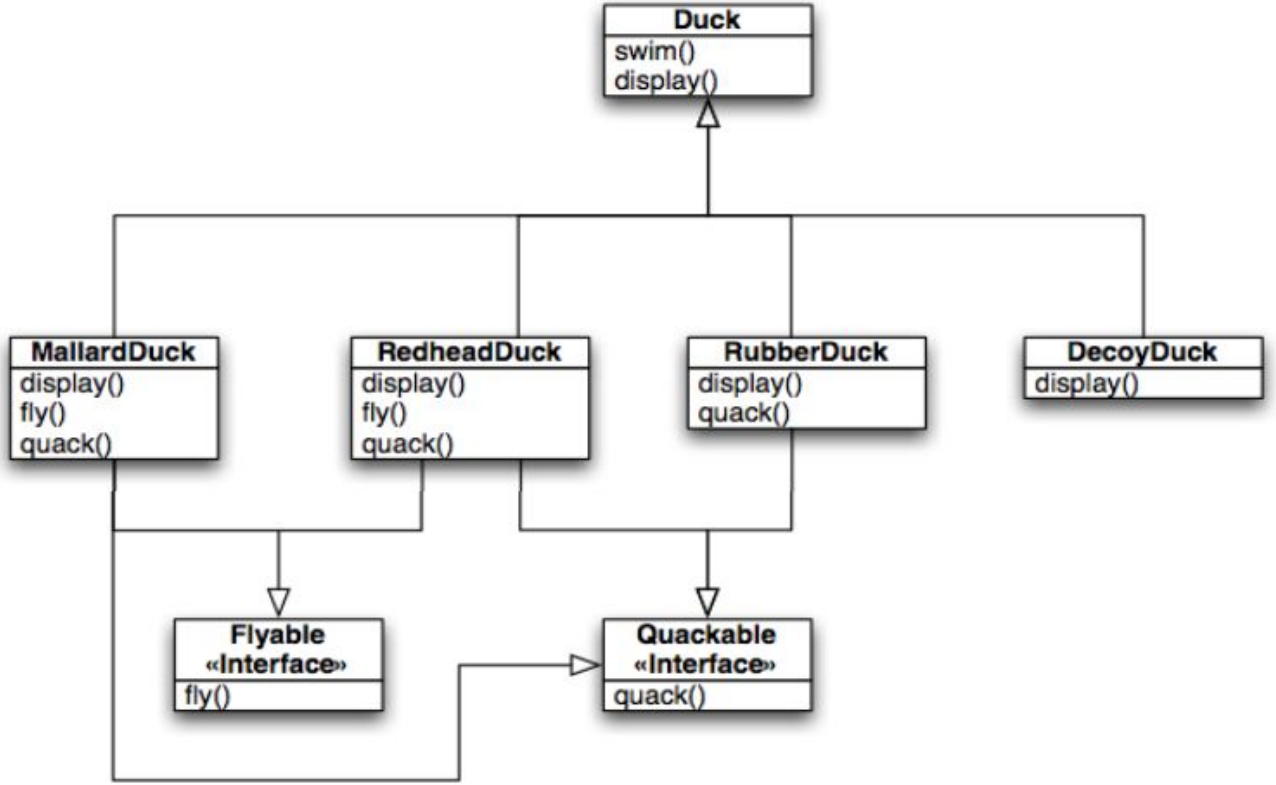
espongono i servizi offerti da una classe ai suoi utilizzatori

devono essere **rispettate** in modo obbligatorio dalla classe

aggiungono capacità alla classe che le implementa

L'interfaccia è un contratto che deve essere rispettato dalle classi che decidono di implementarlo.

SIMULATORE DI PAPERE - SOLUZIONE



La soluzione è quella di creare l'interfaccia **Flyable** e **Quackable** e così possiamo evitare di aggiungere il metodo fly e quack nella classe base ma invece, ad esempio, far implementare l'interfaccia Flyable solo a quelle classi che realmente devono possedere la capacità di volare.

Interfacce in C#

```
public interface IFlyable
{
    int Fly();
}

public interface IQuackable
{
    string Quack();
}
```

POLIMORFISMO

Polimorfismo

Letteralmente significa "avere molte forme" ed è quella tecnica che permette di **sostituire** una **classe concreta** con la sua **classe base** o la sua **interfaccia**, forte del fatto che la classe concreta deve obbligatoriamente implementare anche la classe base o l'interfaccia.

Ognuna delle persone in figura ha una propria identità, caratteristiche e capacità.
È indispensabile che il cassiere sia a conoscenza della reale identità di ogni cliente? No.
Per il cassiere è sufficiente sapere che il cliente vuole comprare degli articoli ed è in grado di pagarli.

Esempio di polimorfismo

```
List<Duck> laghetto = new List<Duck>();
MallardDuck mallard = new MallardDuck();
RubberDuck rubber = new RubberDuck();

laghetto.Add(mallard);
laghetto.Add(rubber);

foreach (Duck d in laghetto)
{
    string duckName = d.Display();
    int swimMeters = d.Swim();
    Console.WriteLine("{0} ha nuotato per {1} metri.", duckName, swimMeters);

    IFlyable fd = d as IFlyable;
    if(fd != null)
    {
        int flyMeters = fd.Fly();
        Console.WriteLine("{0} ha volato per {1} metri.", duckName, flyMeters);
    }
    else
    {
        Console.WriteLine("{0} non vola.", duckName);
    }
}
```