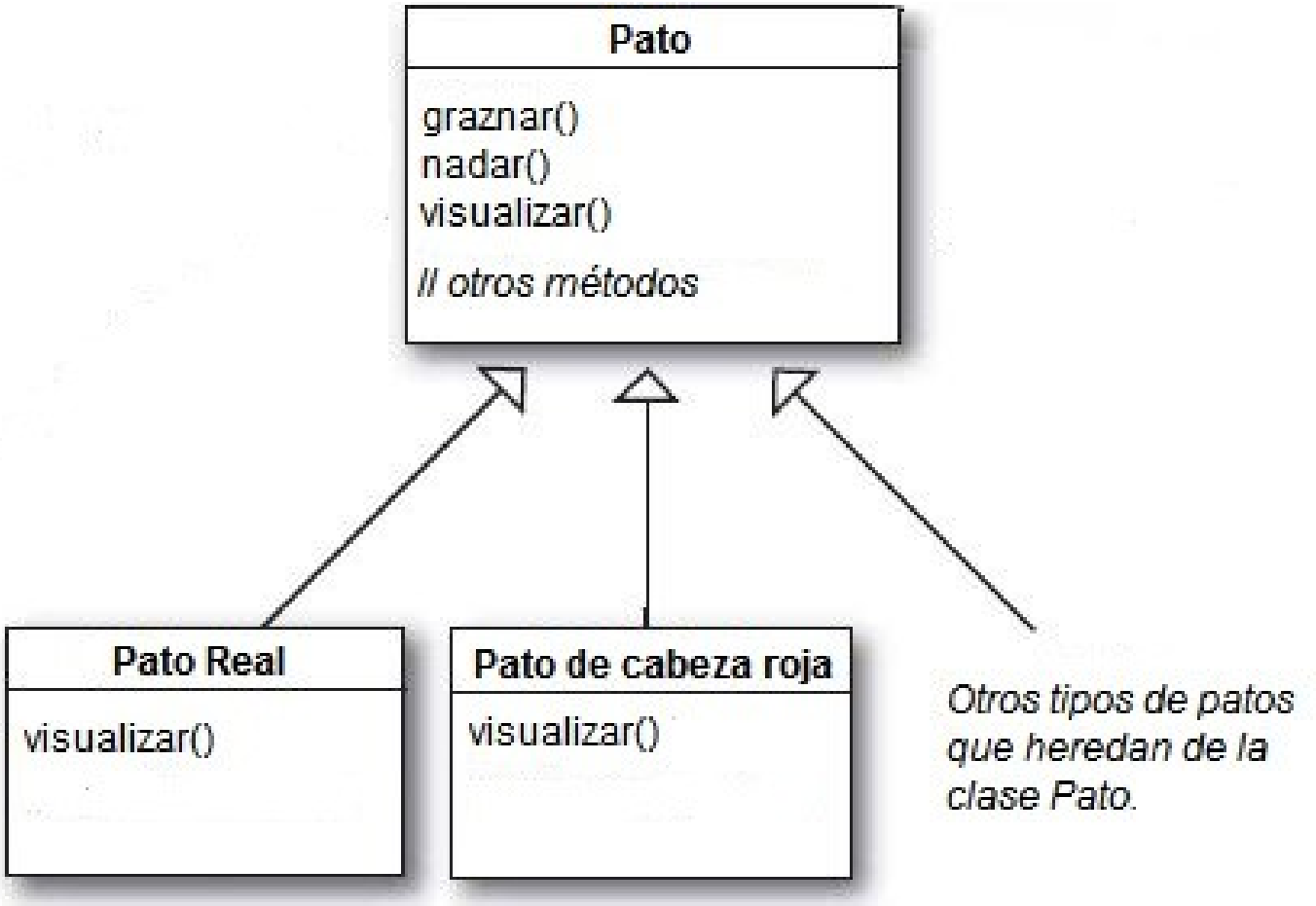
# PATRÓN DE DISEÑO "STRATEGY"

# SIMUDUCK

- **Mario trabaja en una compañía exitosa, realizando un juego de simulación de un estanque de patos. El juego puede mostrar gran variedad de especies de patos nadando y graznando.**

- *¿Cómo harías el diseño?*

- **El primero diseño utiliza OO:**



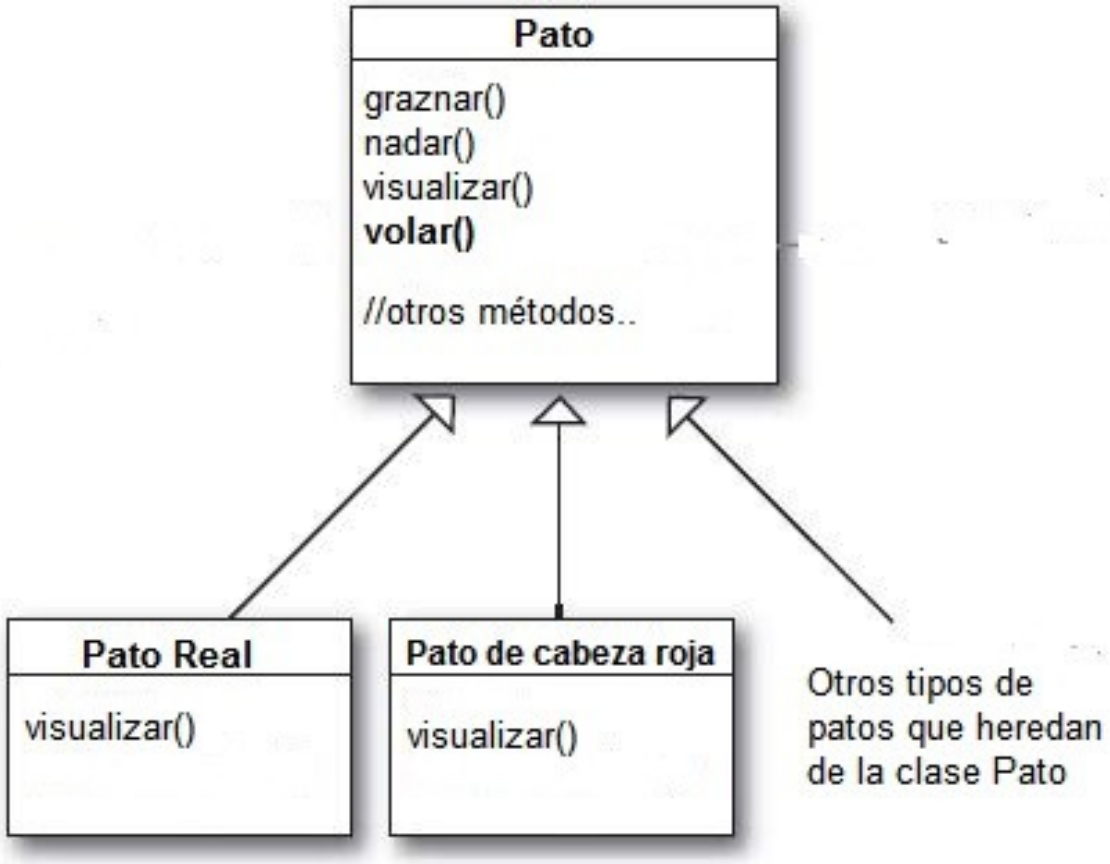
# PERO AHORA NECESITAMOS ALGO INNOVADOR

- **El año pasado la compañía tuvo un gran incremento en la presión de sus competidores, por lo cual decidieron que era momento de pensar en una idea innovadora.**

# PERO AHORA NECESITAMOS QUE LOS PATOS VUELEN…

- **Entonces los ejecutivos acordaron que simular que los *patos vuelan*, era la idea que acabaría con sus competidores.**

- **Mario pensó en agregar un método volar() a la súper clase Pato, por lo cual todas las subclases que heredan de ella, heredarán también este método.**



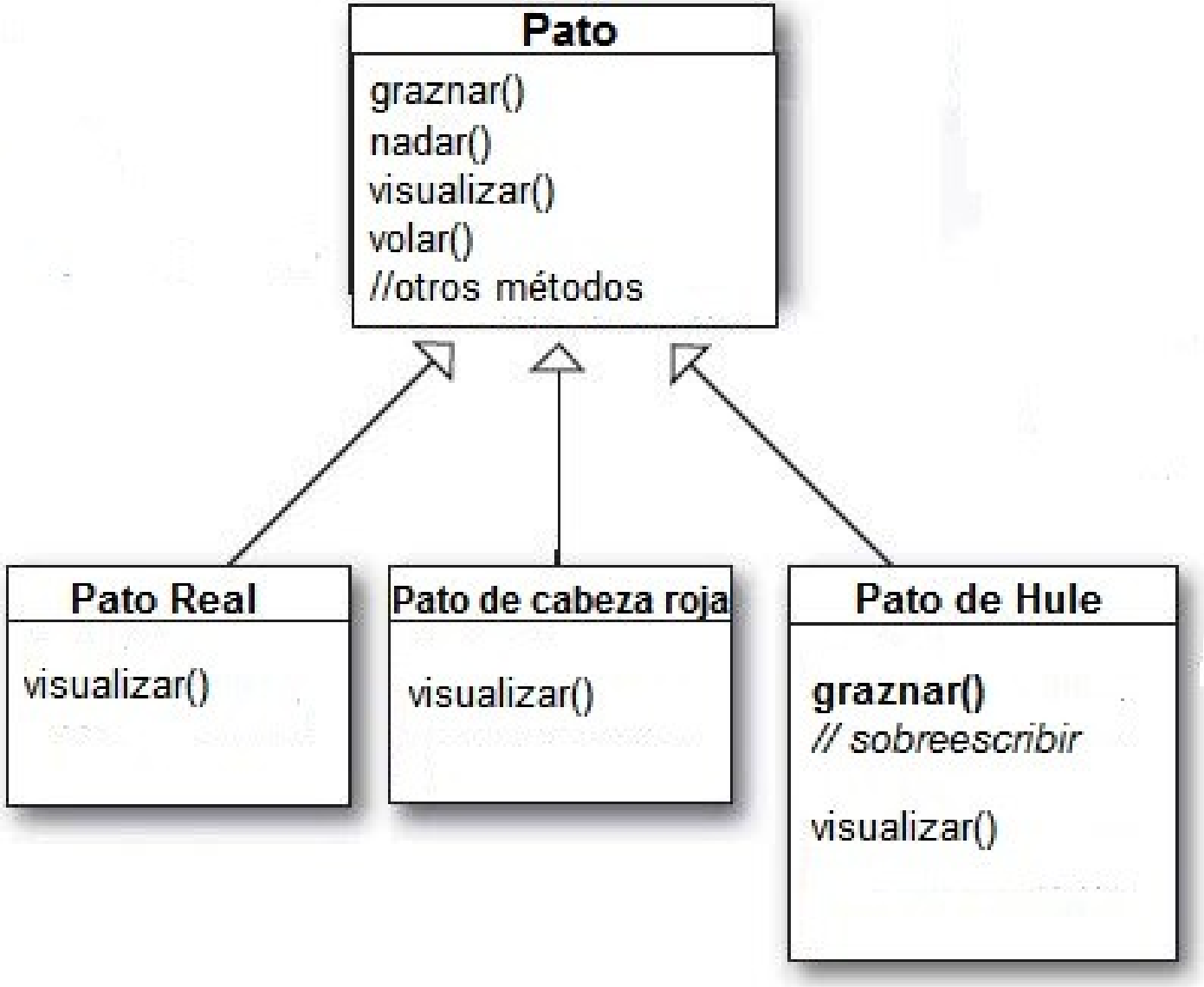

MI TRABAJO AQUÍ
HA TERMINADO

# PERO AHORA EXISTE UN PROBLEMA:

# PERO AHORA EXISTE UN PROBLEMA:

- **Los ejecutivos notaron que a todos los tipos de patos se les agregó un comportamiento que tal vez no sea apropiado para algunos patos.**

# PERO AHORA EXISTE UN PROBLEMA:



- **¿Cuáles podrían ser las desventajas de utilizar herencia para modelar el comportamiento de los patos?**

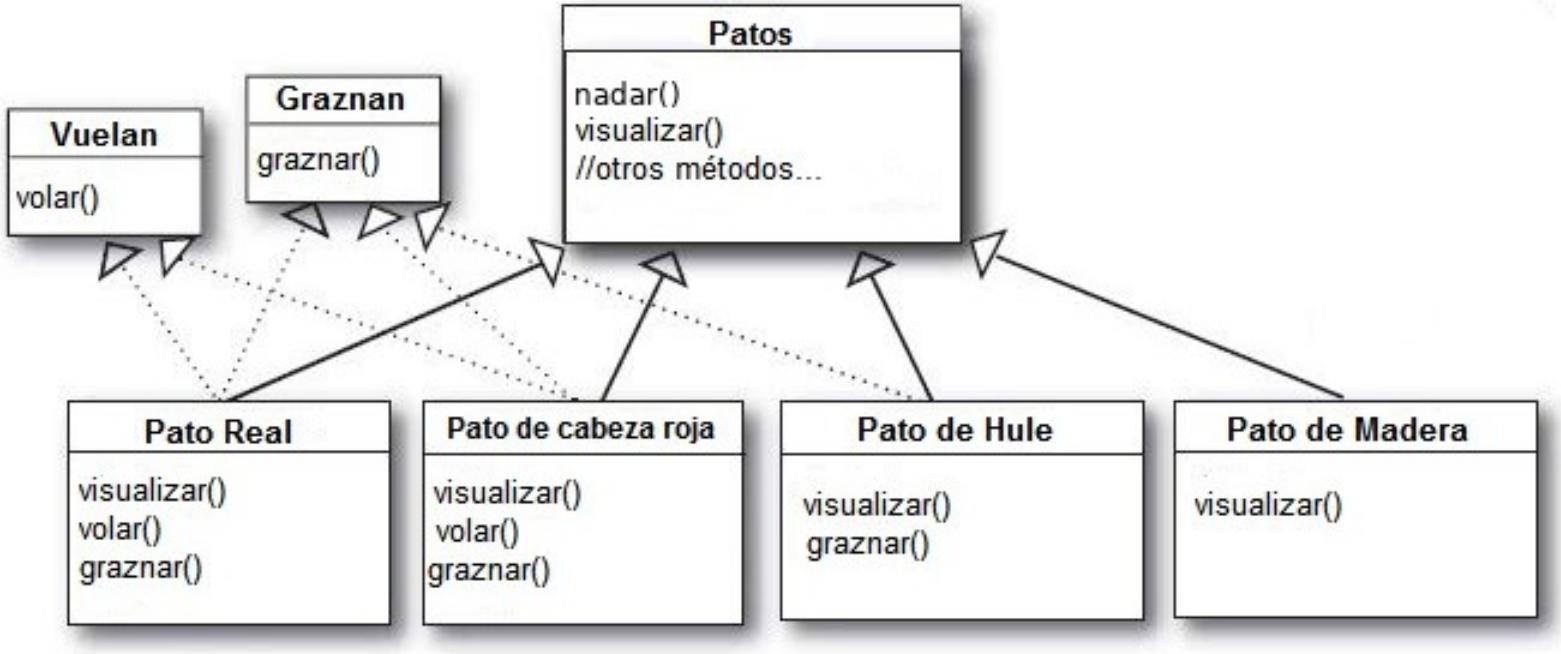
- **¿Cuáles podrían ser las desventajas de utilizar herencia para modelar el comportamiento de los patos?**

1. *Código duplicado a través de las subclases.*
2. *Difícil conocer información sobre el comportamiento de todos los patos.*
3. *No podemos hacer que los patos bailen.*
4. *Los patos no pueden volar y graznar al mismo tiempo.*
5. *Los cambios pueden afectar involuntariamente a otros patos.*

# EN ESTE CASO NO ES BUENA OPCIÓN LA HERENCIA…

- **Los ejecutivos ahora quieren actualizar el sistema cada seis meses, así que Mario pensó en utilizar una forma más limpia de hacer que no todos los tipos de patos vuelen o graznen.**

# EL NUEVO DISEÑO: Uso de interfaces




*¿Qué opinas?*

# La única constante en el desarrollo de software

De acuerdo, ¿cuál es la única cosa con la que siempre puedes contar en el desarrollo de software? **El cambio**

No importa dónde trabaje, qué esté construyendo o en qué lenguaje está programando.

No importa qué tan bien diseñe una aplicación, con el tiempo una aplicación debe crecer y cambiar o morirá.

## NECESITAMOS QUE AL MODIFICAR ALGÚN COMPORTAMIENTO, NO SEA NECESARIO REALIZAR DEMASIADOS CAMBIOS EN LAS SUBCLASE…

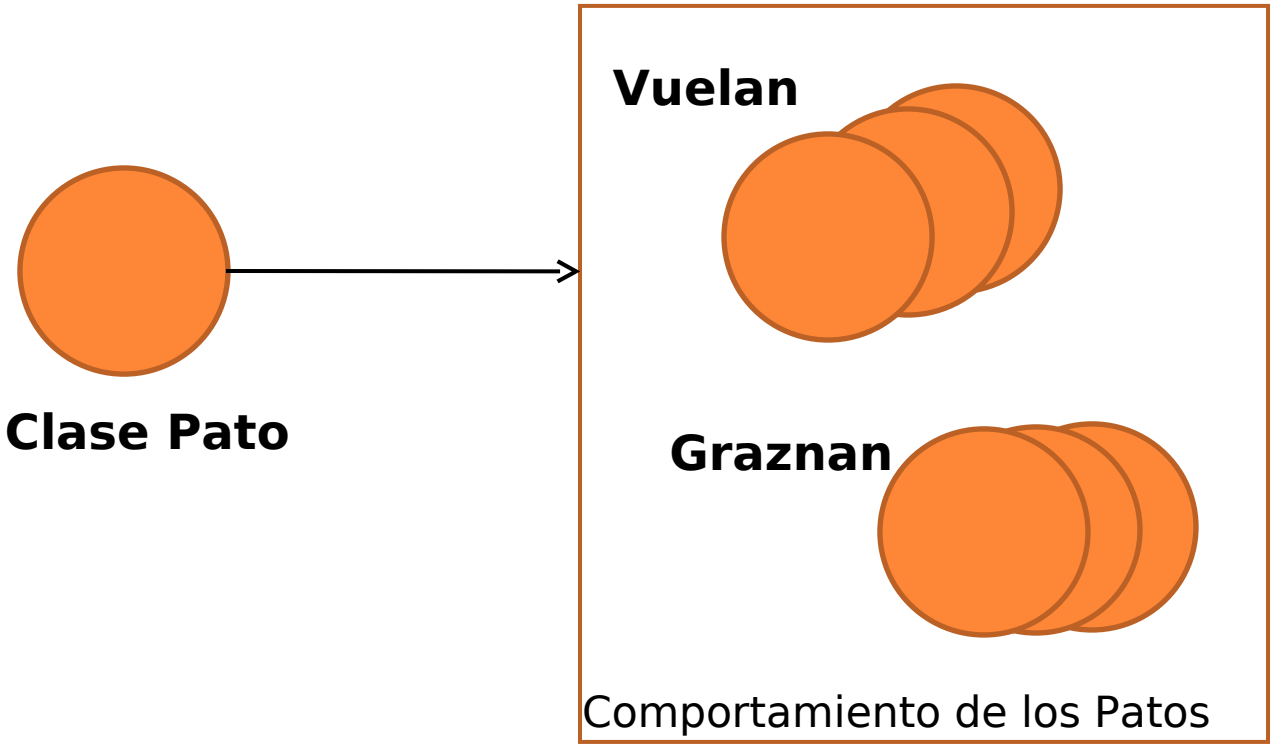
***¿Cómo lo obtenemos?***

- **Principio de Diseño:** *Identificar los aspectos de que varían encapsulandolos y separarlos de los que no cambian.*

*Toma lo que varía y "encapsularlo" para que no afecte al resto de tu código.*

*¿El resultado? ¡Menos consecuencias involuntarias de los cambios de código y más flexibilidad en sus sistemas!*

- Los métodos volar() y graznar() son comportamientos de la clase Pato que varían según el tipo pato.



# Diseñando los comportamientos del pato

Nos gustaría mantener las cosas flexibles; después de todo, fue la inflexibilidad en los comportamientos de los patos lo que nos metió en problemas en primer lugar. Y sabemos que queremos asignar comportamientos a las instancias de Pato. Por ejemplo, es posible que deseemos instanciar una nueva instancia de PatoMallard e inicializarla con un tipo específico de comportamiento de vuelo. Y mientras estamos allí, ¿por qué no nos aseguramos de que podamos **cambiar el comportamiento** de un pato **dinámicamente**? En otras palabras, deberíamos incluir métodos de establecimiento de comportamiento en las clases de Pato para que podamos cambiar el comportamiento de vuelo de PatoMallard en **tiempo de ejecución**.

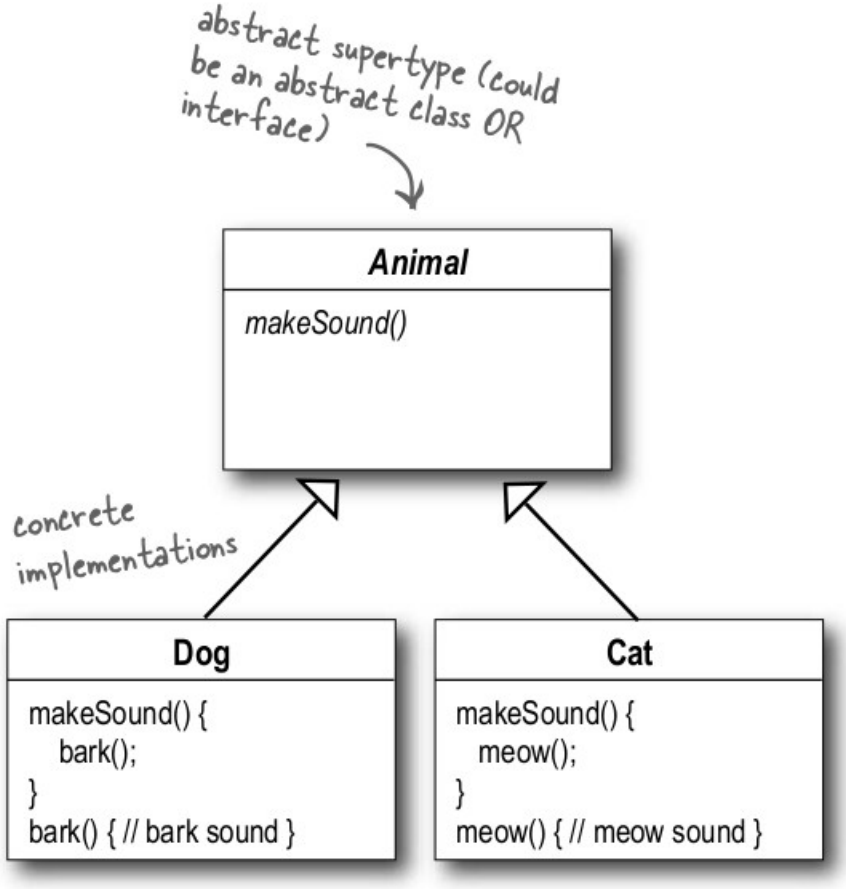
# Principio de diseño

Programa a una interfaz, no a una implementación.

A partir de ahora, los comportamientos de Pato vivirán en una clase separada, una clase que implementa una interfaz de comportamiento particular.

De esta forma, las clases de Pato no necesitarán conocer ninguno de los detalles de implementación para sus propios comportamientos.

"Programa a una interfaz" realmente significa "Programar a un supertipo".




**Programming to an implementation** would be:

```
Dog d = new Dog();
d.bark();
```

Declaring the variable "d" as type Dog (a concrete implementation of Animal) forces us to code to a concrete implementation.

But **programming to an interface/supertype** would be:

```
Animal animal = new Dog();
animal.makeSound();
```

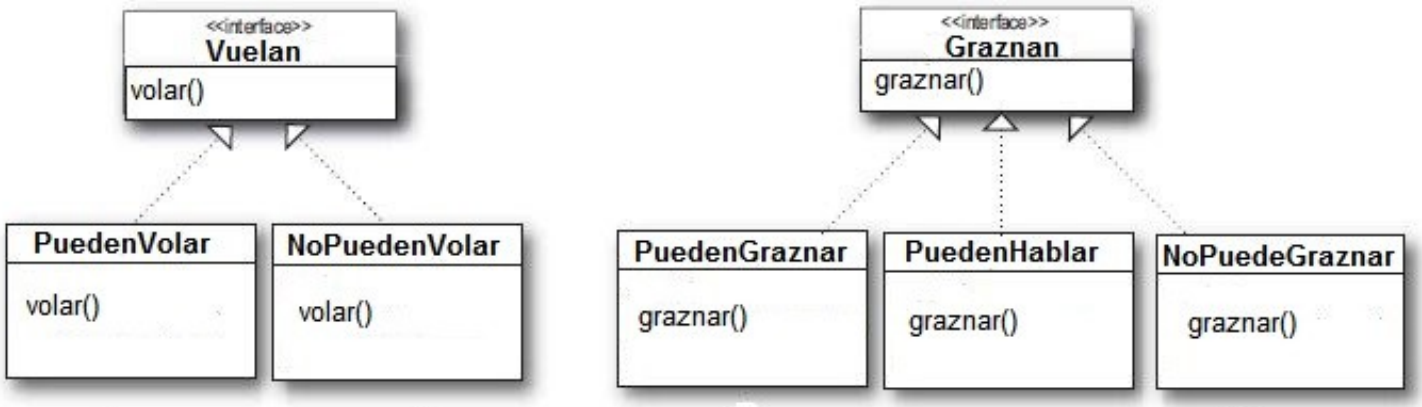
We know it's a Dog, but we can now use the animal reference polymorphically.

Even better, rather than hard-coding the instantiation of the subtype (like new Dog()) into the code, **assign the concrete implementation object at runtime:**

```
a = getAnimal();
a.makeSound();
```

We don't know WHAT the actual animal subtype is... all we care about is that it knows how to respond to makeSound().

# ¿CÓMO HARÍAS EL DISEÑO DEL CONJUNTO DE CLASES QUE IMPLEMENTEN EL COMPORTAMIENTO VOLAR Y GRAZNAR?



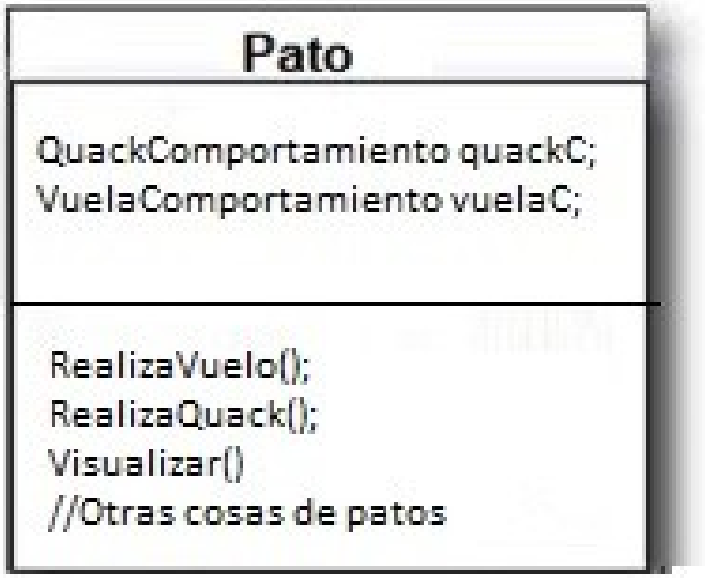

**Con este diseño:**

- *Es posible reutilizar código.*
- *Podemos agregar nuevos comportamientos sin modificar los existentes.*

# Pregunta

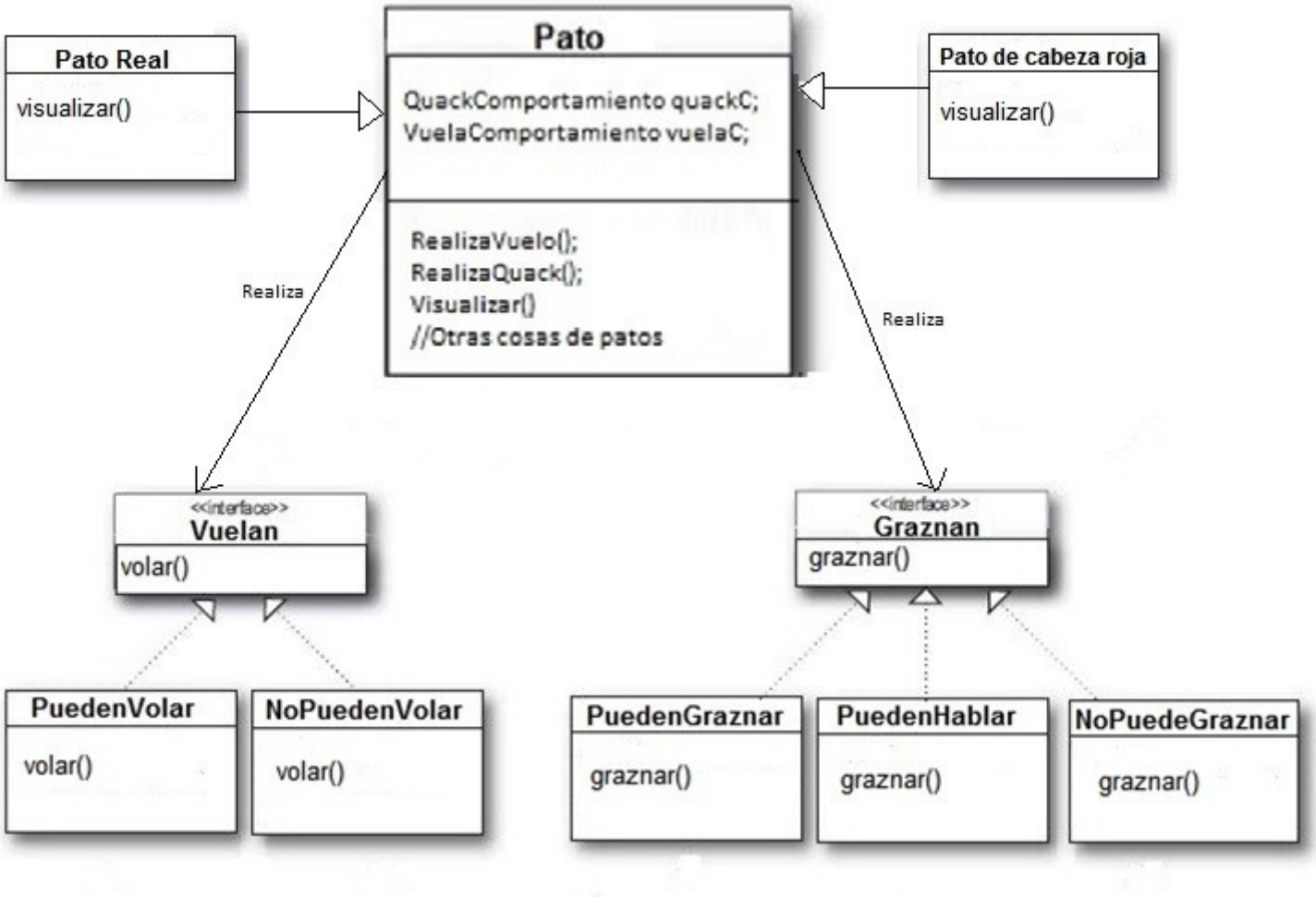
¿Siempre tengo que implementar mi aplicación primero, ver dónde están cambiando las cosas, y luego volver y separar y encapsular esas cosas?

**Ahora nuestra clase pato queda así:**




**Y los comportamientos se pueden modificar sin alterar al pato.**

# Nuestro diagrama completo sería algo como esto:



```
public abstract class Duck {
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    public Duck() {
    }

    public void setFlyBehavior(FlyBehavior fb) {
        flyBehavior = fb;
    }

    public void setQuackBehavior(QuackBehavior qb) {
        quackBehavior = qb;
    }

    abstract void display();

    public void performFly() {
        flyBehavior.fly();
    }

    public void performQuack() {
        quackBehavior.quack();
    }

    public void swim() {
        System.out.println("All ducks float, even decoys!");
    }
}
```

Los comportamientos se manejan de la siguiente manera:

```java
public interface QuackBehavior {

    public void quack();
}
```

```java
public class Squeak implements QuackBehavior {

    public void quack() {
        System.out.println("Squeak");
    }
}

public class MuteQuack implements QuackBehavior {

    public void quack() {
        System.out.println("<< Silence >>");
    }
}

public class FakeQuack implements QuackBehavior {

    public void quack() {
        System.out.println("Qwak");
    }
}
```

Análogamente se hace el comportamiento de Volar

```
public class DecoyDuck extends Duck {

    public DecoyDuck() {
        setFlyBehavior(new FlyNoWay());
        setQuackBehavior(new MuteQuack());
    }

    public void display() {
        System.out.println("I'm a duck Decoy");
    }
}

public class MallardDuck extends Duck {

    public MallardDuck() {
        setFlyBehavior(new FlyWithWings());
        setQuackBehavior(new Quack()));
    }

    public void display() {
        System.out.println("I'm a real Mallard duck");
    }
}
```

# Principio de diseño

Favorecer la composición sobre la herencia.

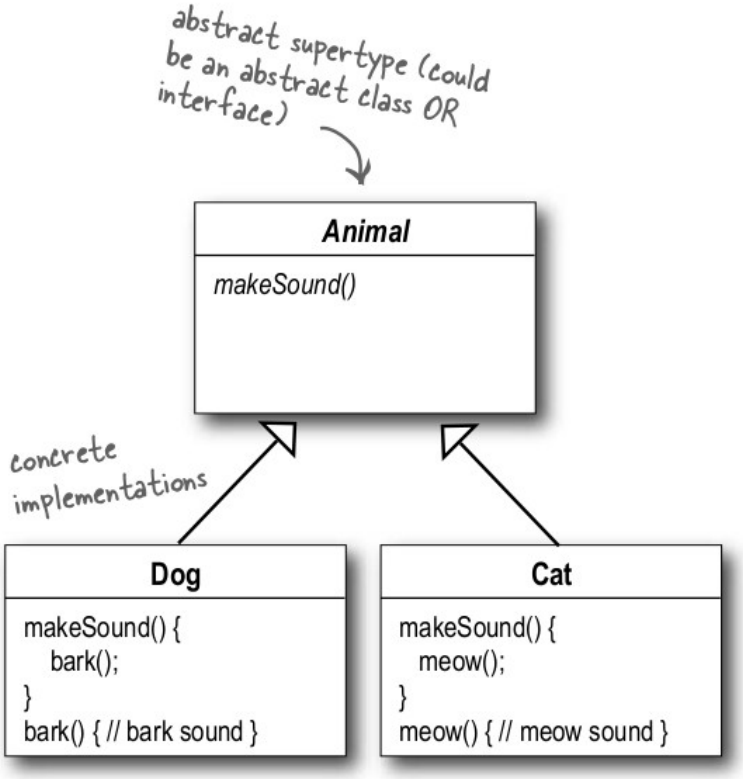
Crear sistemas con composición le da mucha más flexibilidad. No solo le permite encapsular una familia de algoritmos en su propio conjunto de clases, sino que también le permite cambiar el comportamiento en tiempo de ejecución siempre que el objeto con el que está redactando implemente la interfaz de comportamiento correcta

# Strategy

El patrón de estrategia define una familia de algoritmos, encapsula cada uno y los hace intercambiables. La estrategia permite que el algoritmo varíe independientemente de los clientes que lo usan.

## ¡¡¡CONSEJO IMPORTANTE!!!

- **Principio de Diseño:** *Programa para las interfaces (o clases abstractas), no para la implementación.*




**Programming to an implementation** would be:

```
Dog d = new Dog();
d.bark();
```

Declaring the variable "d" as type Dog (a concrete implementation of Animal) forces us to code to a concrete implementation.

But **programming to an interface/supertype** would be:

```
Animal animal = new Dog();
animal.makeSound();
```

We know it's a Dog, but we can now use the animal reference polymorphically.

Even better, rather than hard-coding the instantiation of the subtype (like new Dog()) into the code, **assign the concrete implementation object at runtime:**

```
a = getAnimal();
a.makeSound();
```

We don't know WHAT the actual animal subtype is... all we care about is that it knows how to respond to makeSound().