

2

(II)

Patrón Strategy

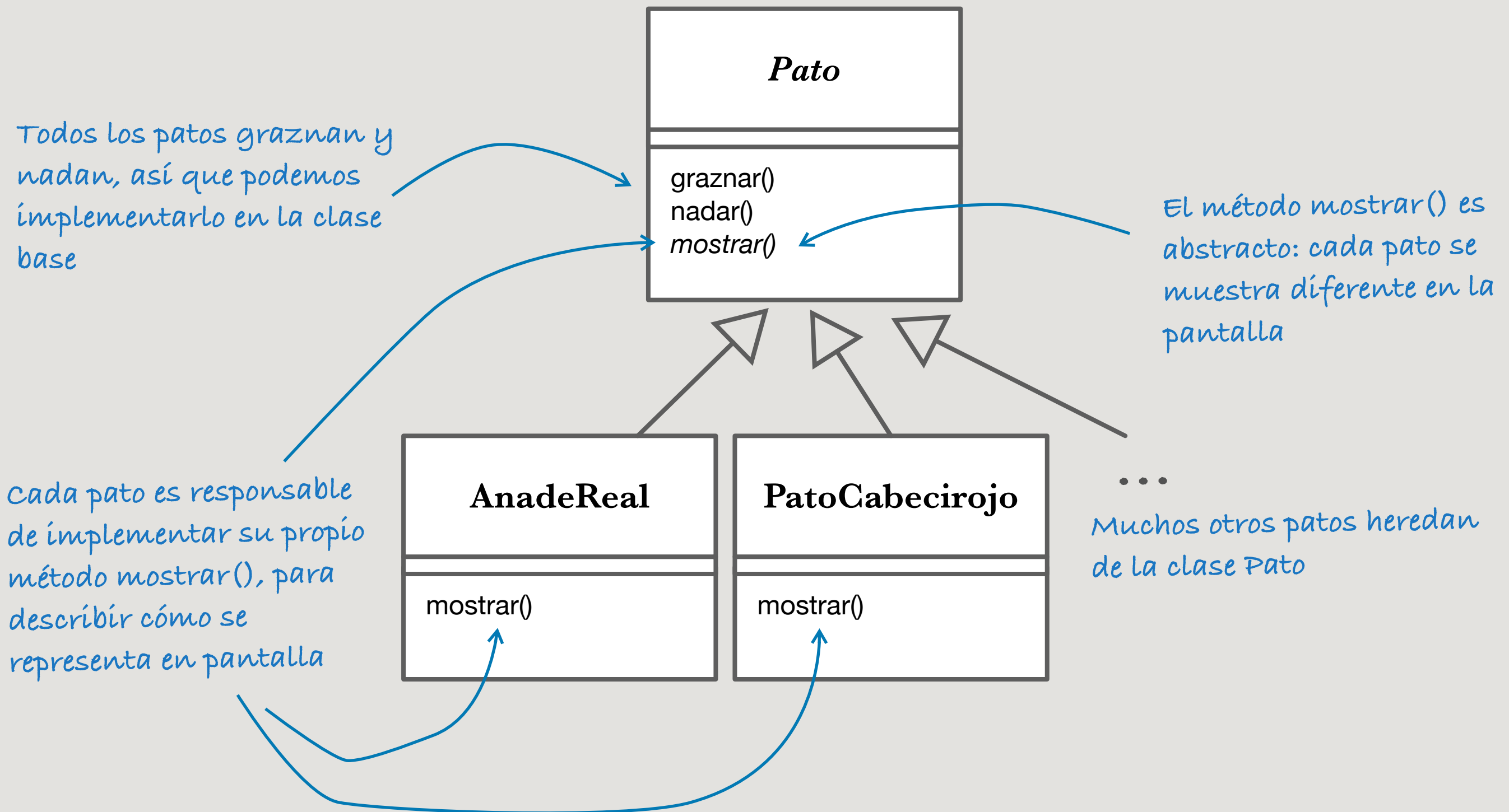
(Patrones de diseño)

Diseño del Software

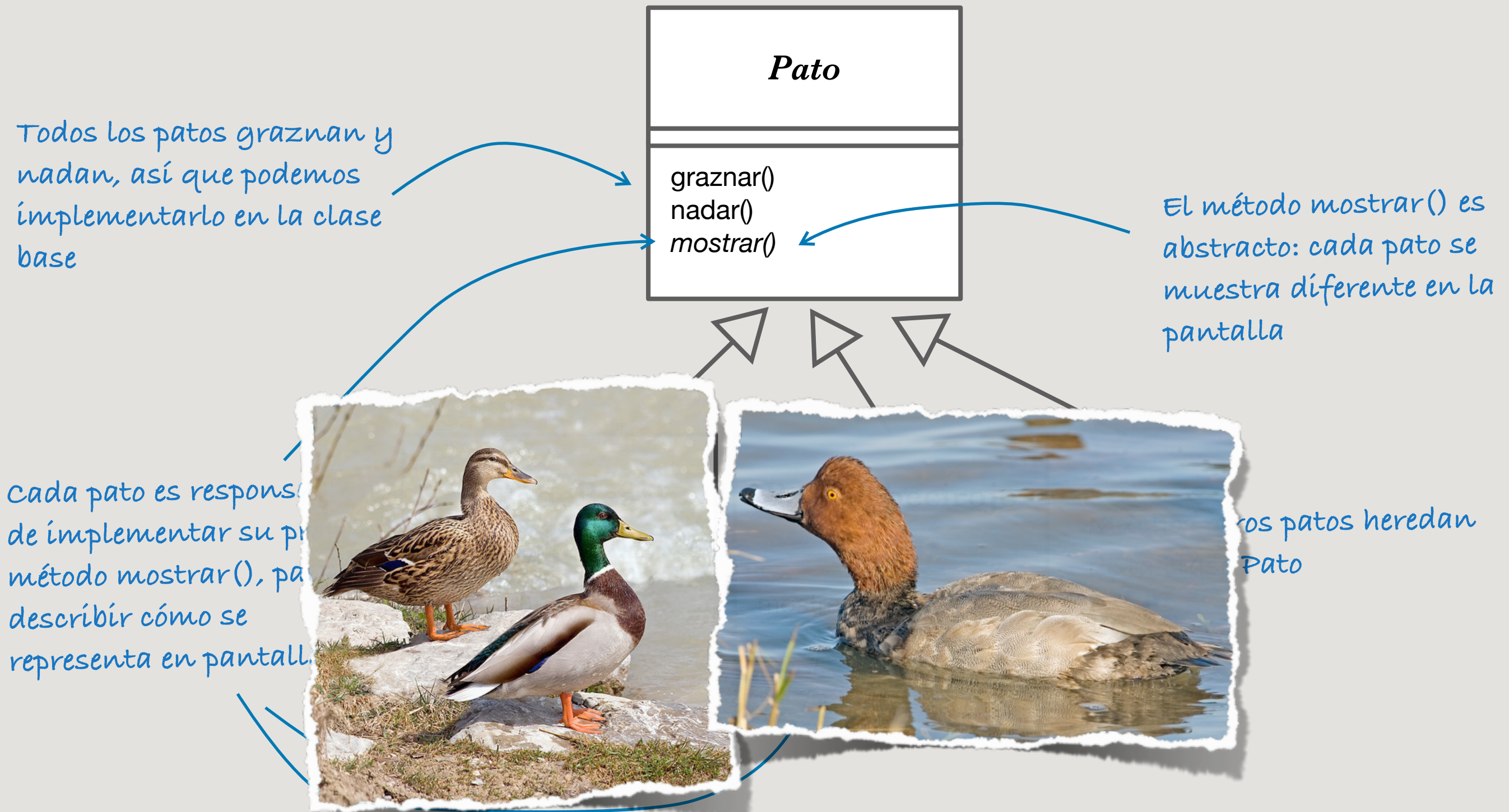
Grado en Ingeniería Informática del Software

Curso 2017-2018

Simulador de patos



Simulador de patos

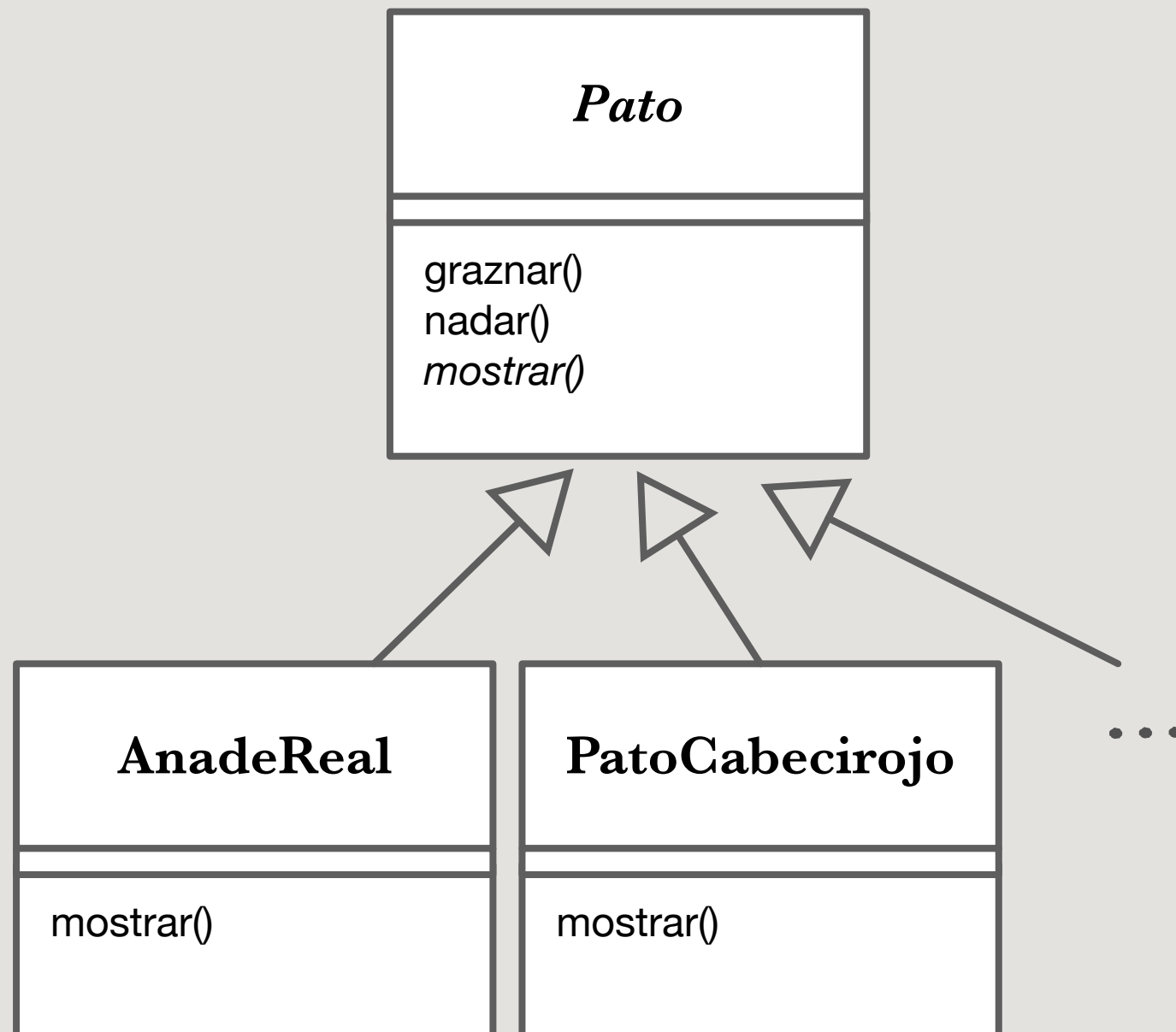


Ahora los patos...

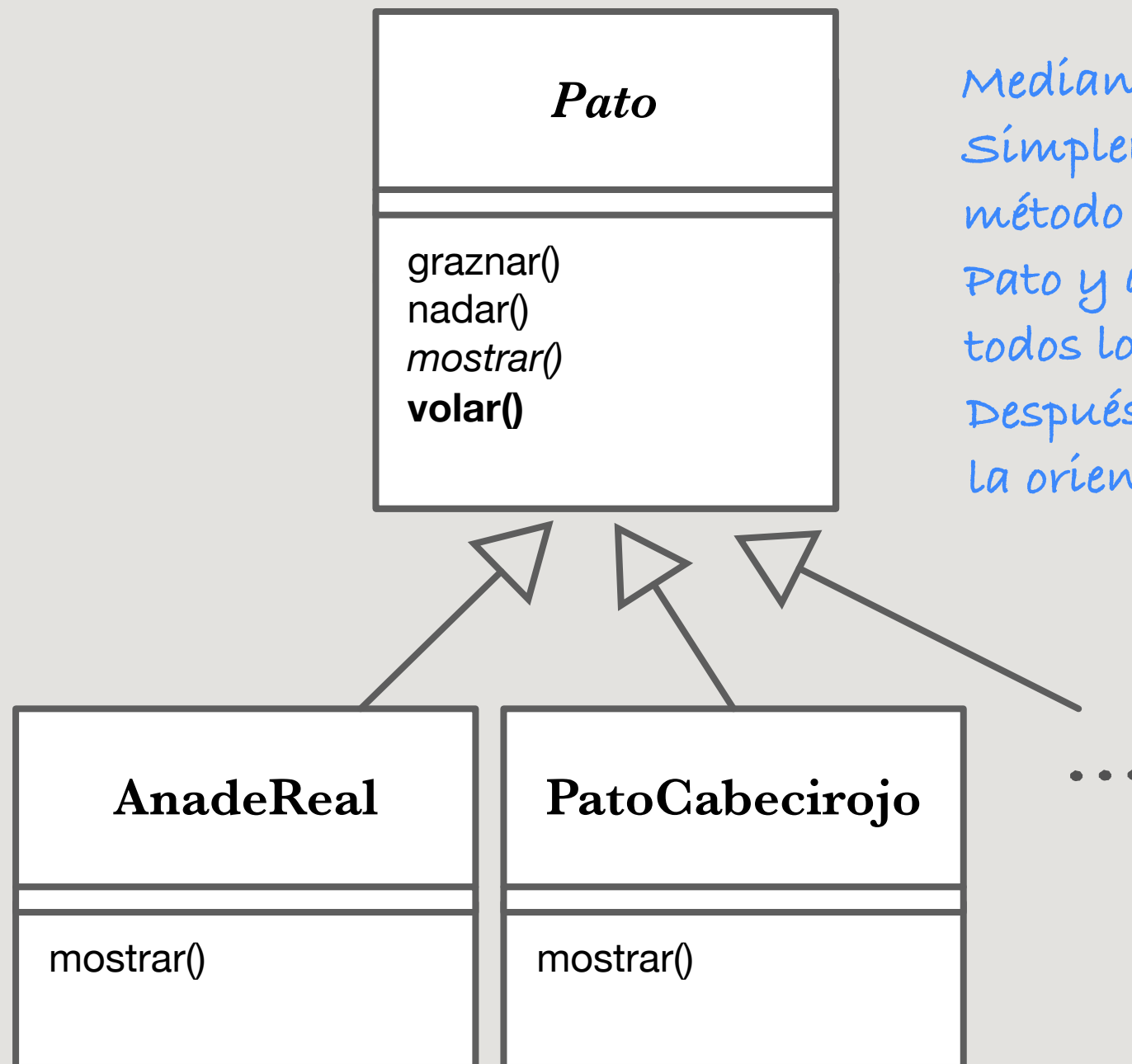
¡vuelan!

¿Cómo lo hacemos?

Ahora los patos... ¡vuelan!



Solución tradicional



Mediante la herencia.
Simplemente añadimos un método «volar» a la clase Pato y automáticamente todos los patos lo heredarán. Después de todo, ¿no era eso la orientación a objetos?

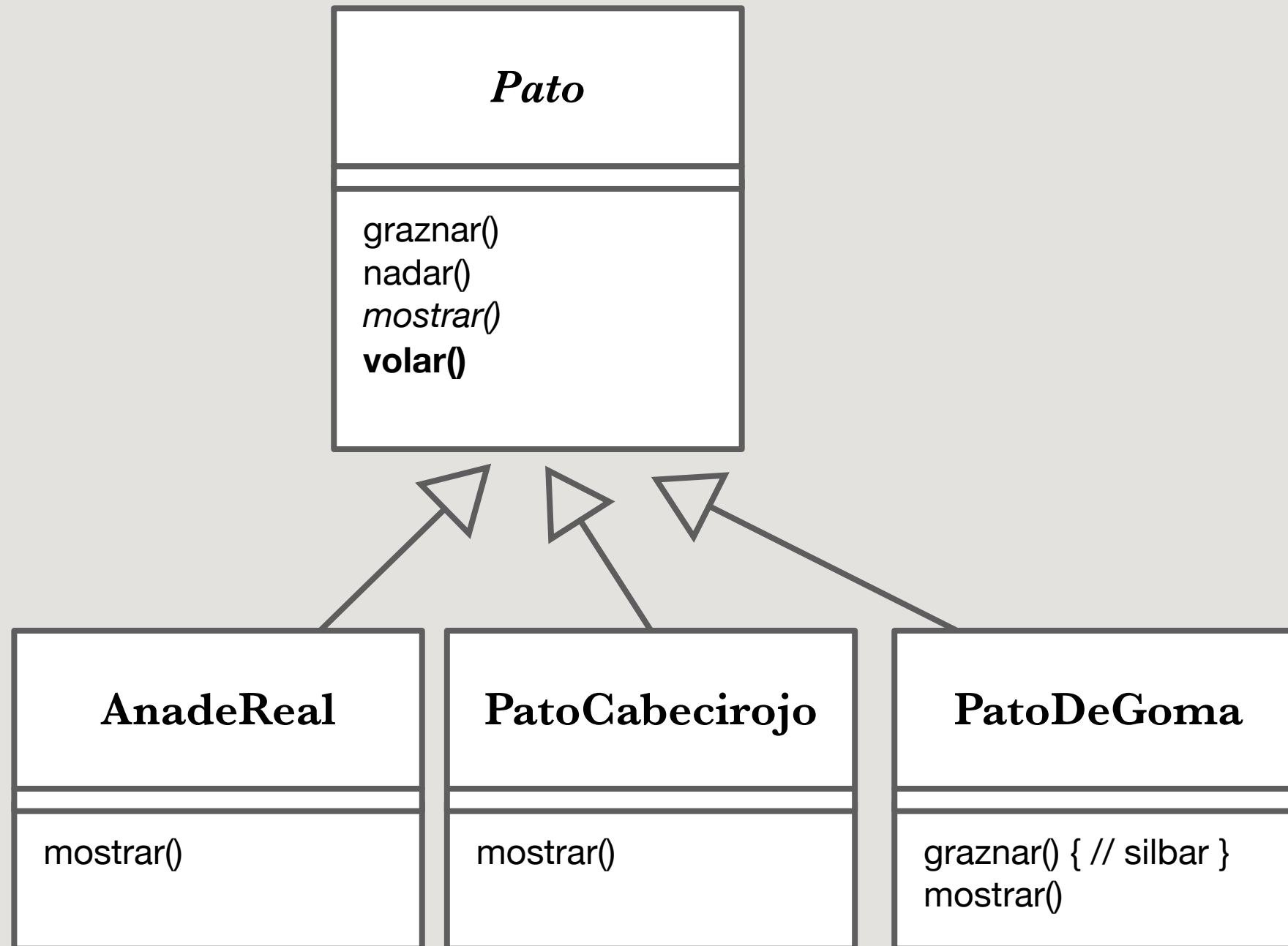
Pero...



Problema

- Ahora tenemos un montón de patos de goma volando por la pantalla
- Al poner volar en la superclase, hemos dado esa habilidad a todos los patos, incluyendo aquellos que no deberían

Patos de goma



Una posibilidad

- Redefinir `volar` en `PatoDeGoma` para que no haga nada

```
volar() {  
    // no hacer nada  
}
```



Problema con la herencia

- ¿Pero qué pasa si ahora tenemos patos de reclamo, de los que se utilizan como cebo para cazar?
 - Ni graznan ni vuelan

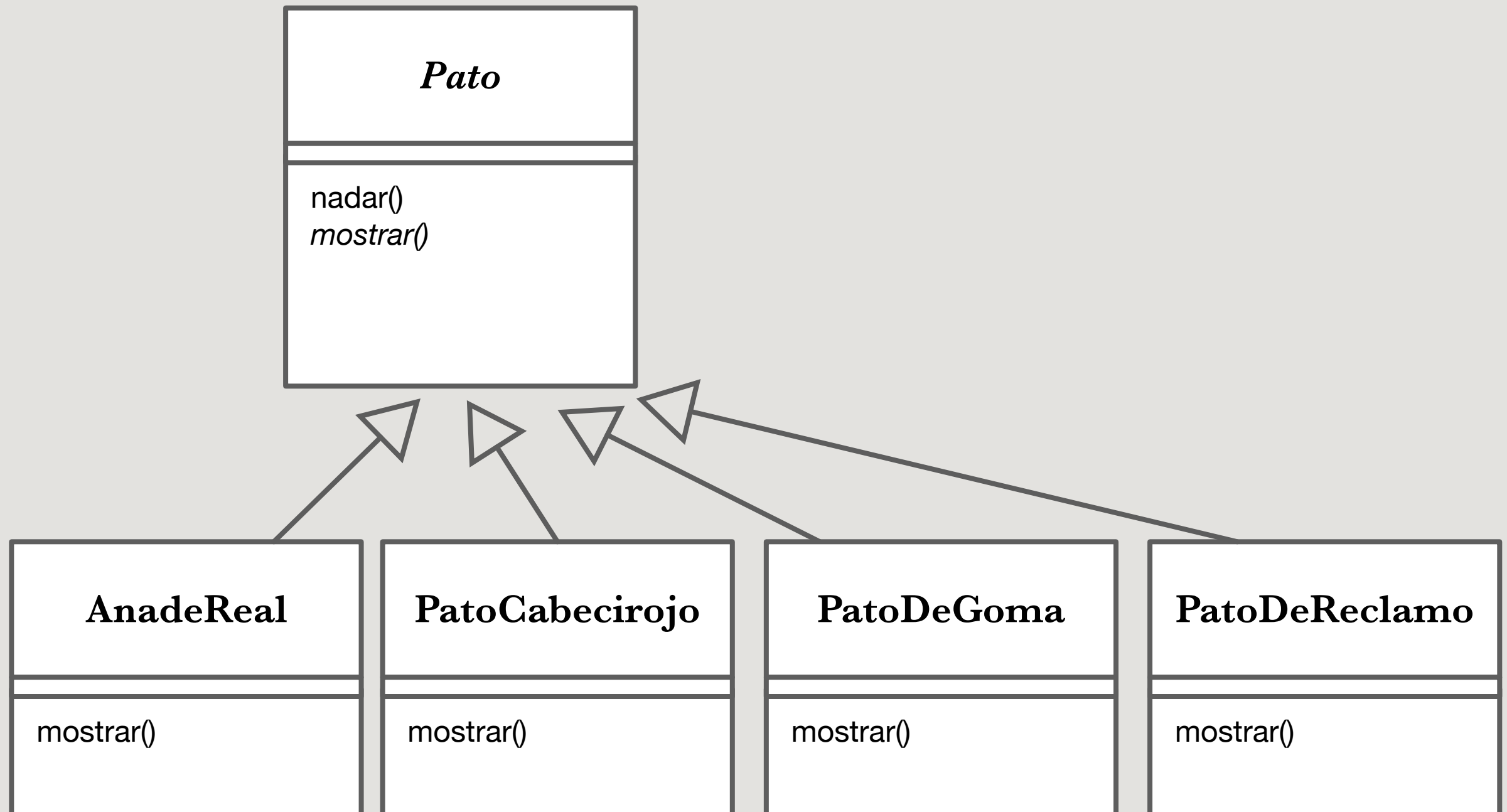
Habría que redefinir a su vez ambos métodos en esta clase, para que no hiciesen nada.



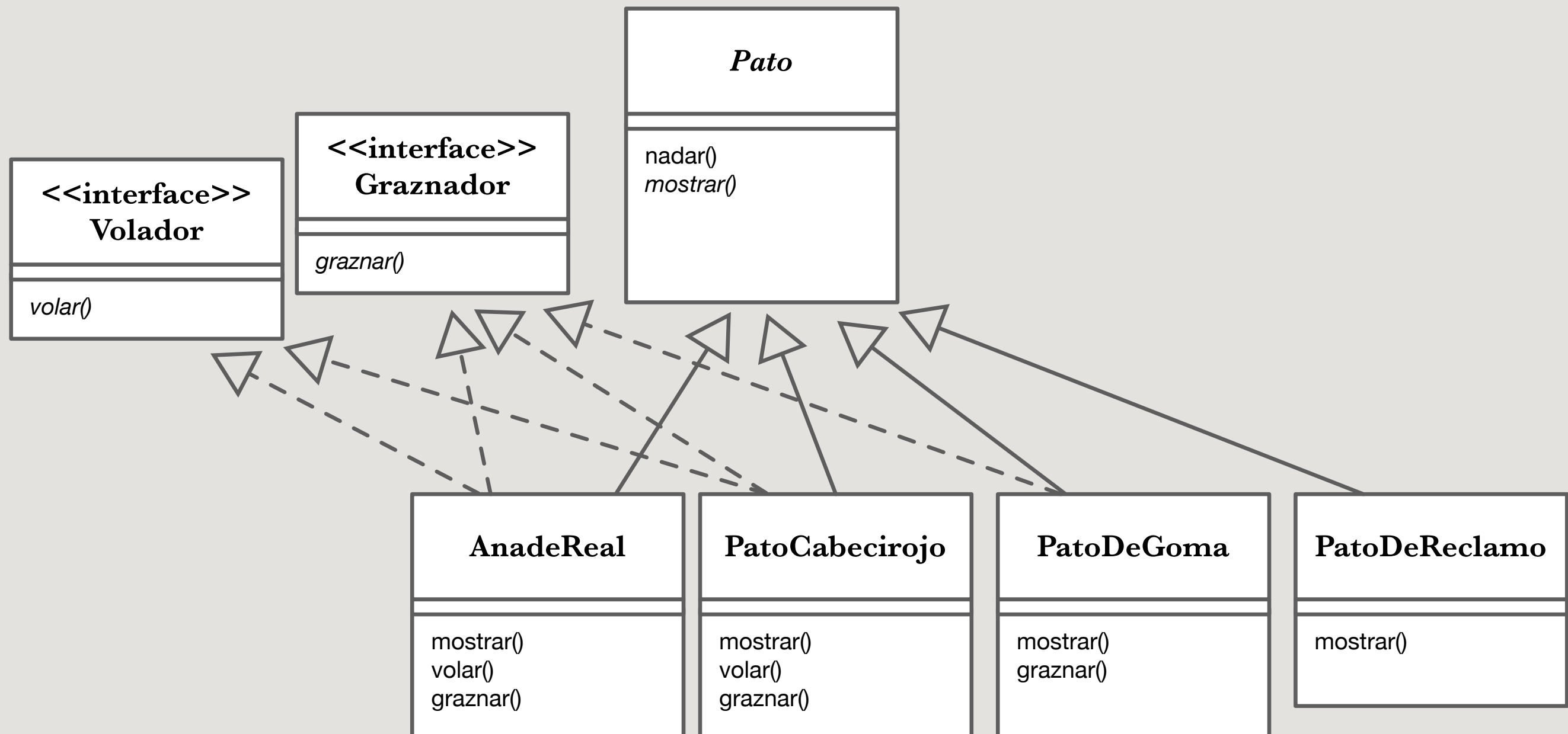
*Se empieza a
complicar...*

¿Alguna solución mejor?

¿Y con interfaces de Java?



¿Y con interfaces de Java?



Pero perdemos algo...

Solución con interfaces

- **Resuelve parte del problema**
 - Ya no hay patos con comportamientos inapropiados
- **Pero introduce otro nuevo**
 - Desaparece la reutilización de código
 - O, lo que es lo mismo, introducimos duplicación de código

Problemas de
mantenimiento

*si hay una constante en el
desarrollo de software ésta es...*

el CAMBIO

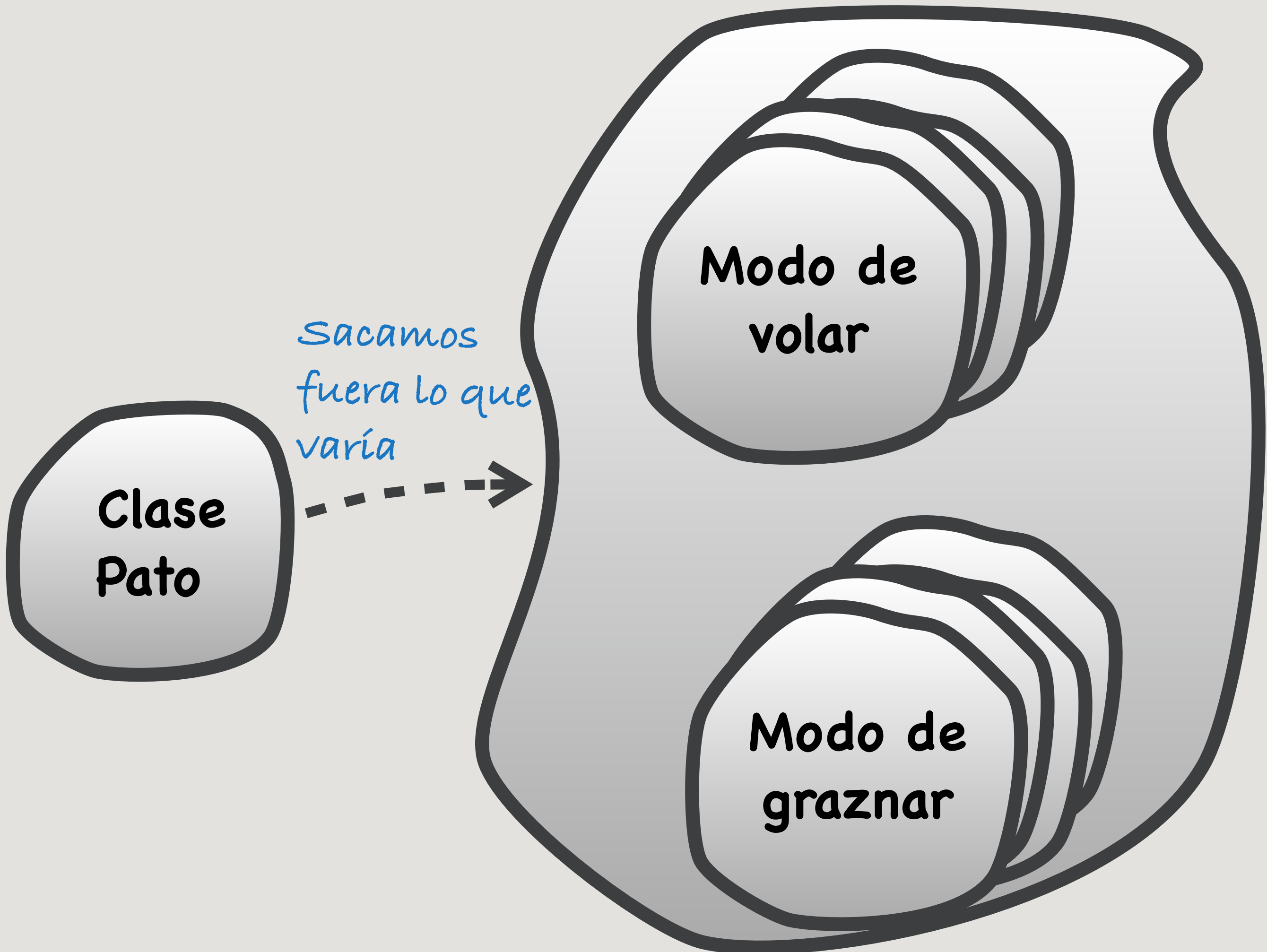
Se va a producir.

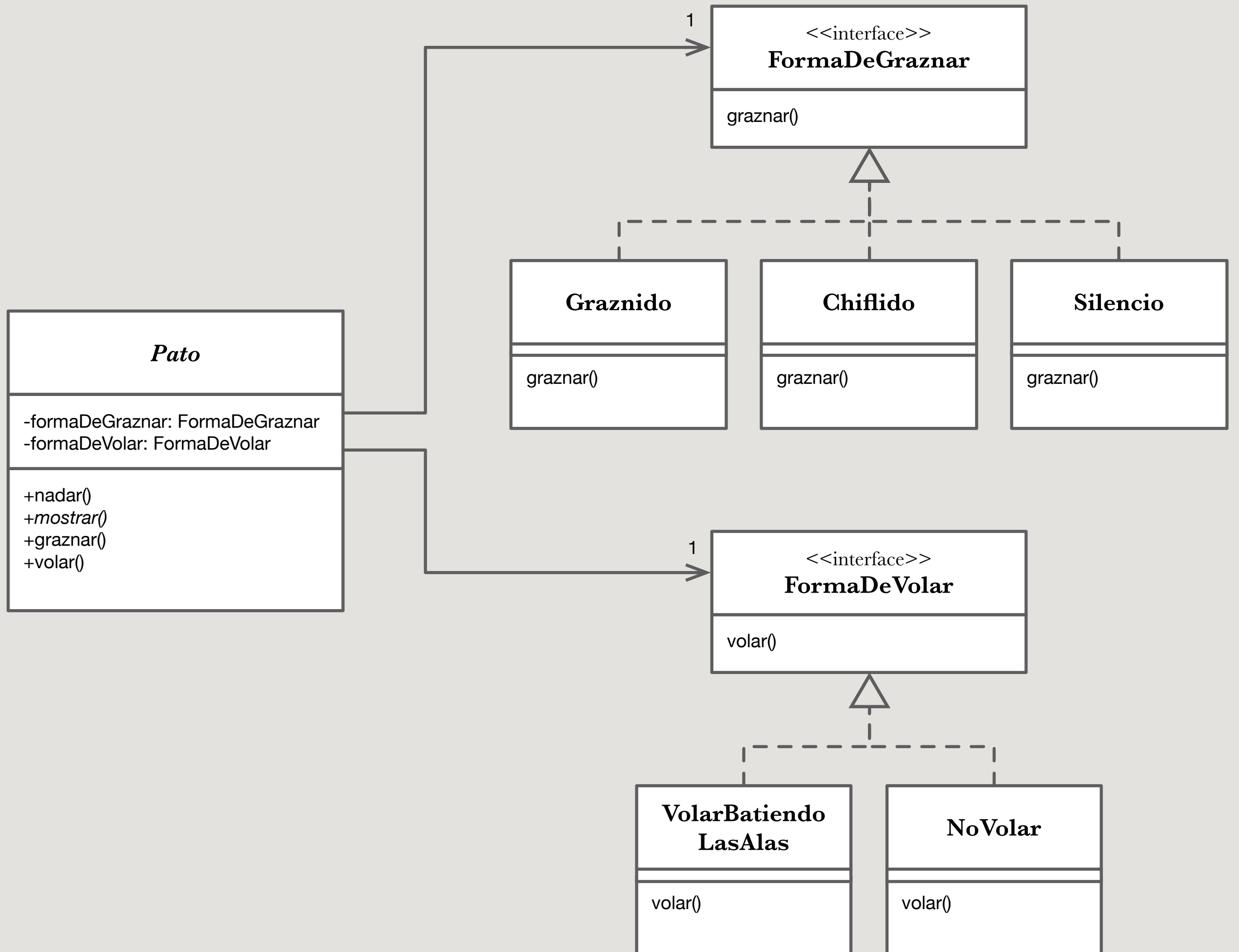
Siempre.

*Otro de los principios básicos
del diseño OO*

*Identificar aquellos aspectos
que varían y separarlos de lo
que tiende a permanecer igual.*

Es decir... encapsular el concepto que varía





Nota

- **Vamos a suponer que esas implementaciones «vacías» de NoVolar y Silencio son correctas**
 - Que lo sean o no (desde un punto de vista de un buen diseño OO) dependerá de cada caso concreto
 - El problema aquí es que el ejercicio pretende ser tan simple que en el fondo se complica, al no tener claros los requisitos
 - Realmente eso representa un problema general de la OO con la herencia y los lenguajes con comprobación estática de tipos, como Java o C++
 - Centrémonos aquí en lo que se consigue con este nuevo diseño

¿Qué hemos hecho?

- **Simplemente aplicar tres principios básicos de la OO:**

- Encapsular el concepto que varía
 - ▶ En este caso, los comportamientos de volar y de graznar, y los sacamos fuera de la clase Pato
- Favorecer la composición sobre la herencia
- Programar para una interfaz, no para una implementación
 - ▶ La clase Pato sólo trata con FormaDeVolar y FormaDeGraznar, pero no sabe cuáles son

*Pues así, sin saberlo, hemos
llegado a la implementación de
un patrón de diseño...*

Patrón Strategy

Intención (propósito)

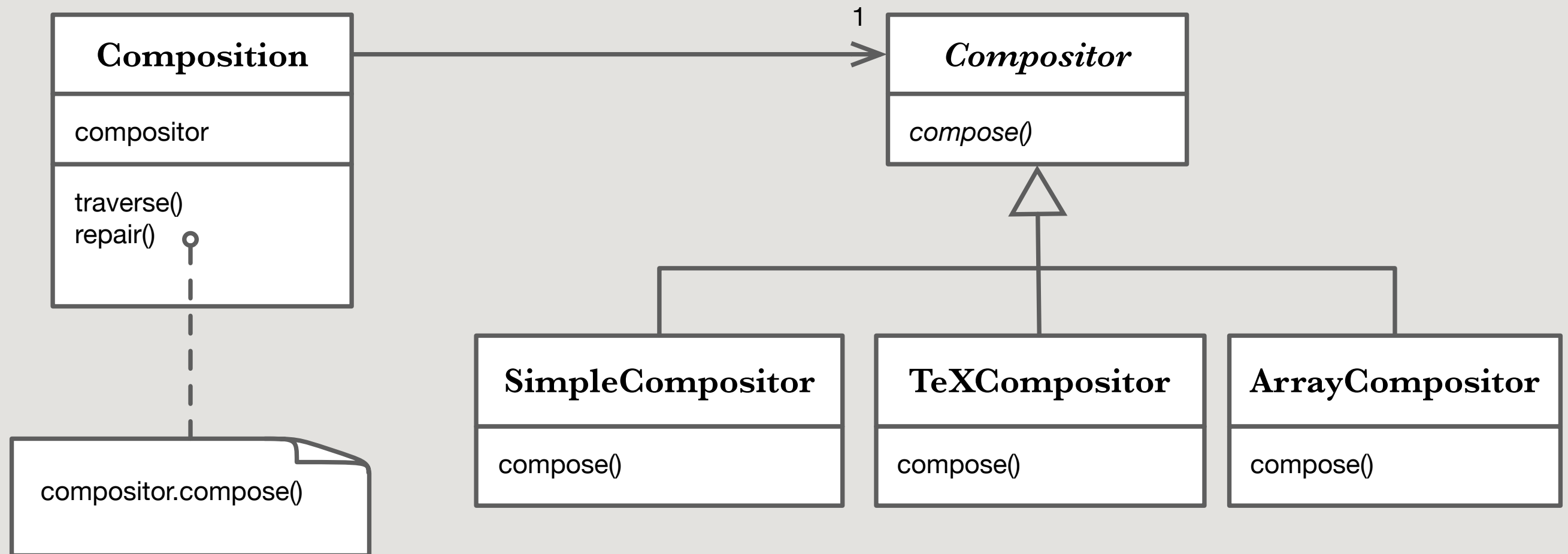
Define una familia de algoritmos, encapsula cada uno y los hace intercambiables. Permite que el algoritmo varíe de forma independiente a los clientes que lo usan.

- **Patrón de comportamiento, de objetos**
- **También conocido como**
 - Policy

Motivación

- **Un procesador de textos puede incorporar distintos algoritmos de separación de un texto en líneas:**
 - Una estrategia línea a línea, simple pero horrible desde el punto de vista tipográfico, que hace inviable la justificación
 - ▶ Ejemplo: Microsoft Word, cualquier navegador actual
 - El algoritmo de TeX
 - ▶ Lo hace con cada párrafo: minimiza los «ríos» en el texto
 - Otra que distribuya un número fijo de elementos en cada línea
 - Etcétera

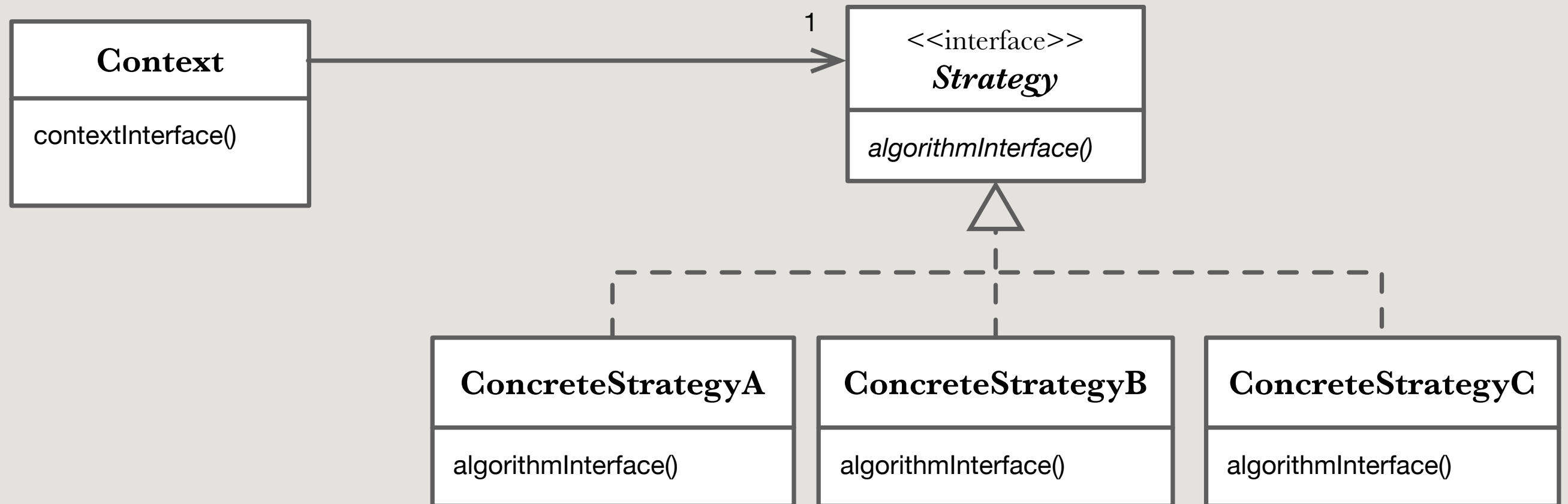
Motivación



Aplicabilidad

- **Permite configurar una clase con un comportamiento determinado de entre varios**
- **Se necesitan distintas variantes de un algoritmo**
- **Los distintos comportamientos de una clase aparecen como múltiples sentencias condicionales**
 - El patrón Strategy permite mover cada rama de esos condicionales anidados a su propia clase

Estructura



Participantes

- **Strategy (Compositor)**
 - Declara una interfaz común
- **ConcreteStrategy (SimpleCompositor, TeXCompositor, ArrayCompositor)**
 - Implementa los distintos algoritmos
- **Context (Composition)**
 - Se configura con una estrategia concreta
 - Mantiene una referencia a dicho objeto Strategy
 - Puede definir operaciones para permitir a la estrategia acceder a los datos que necesite

Colaboraciones

- **La Estrategia y el Contexto colaboran para implementar el algoritmo escogido**
 - El contexto puede pasar todos los datos que necesita al llamar a la estrategia concreta
 - O se puede pasar a sí mismo como referencia para que aquélla llame a los métodos que necesite
- **Los clientes colaboran con el contexto**
 - Pueden pasarle la estrategia concreta

Consecuencias

- **Define familias de algoritmos relacionados**
- **Es una alternativa a la herencia**
 - Hace que el contexto sea más fácil de entender, modificar y mantener
 - Evita la duplicación de código
 - Evita la explosión de subclases
 - ▶ Cuando, como en el caso de los patos, tenemos dos o más conceptos que varían
 - Se puede cambiar dinámicamente


Consecuencias

- **Elimina las múltiples sentencias condicionales**
 - Muchas veces son el indicador de que necesitamos aplicar un patrón (éste u otro, según el caso)

```
void Composition::Repair () {  
    switch (_breakingStrategy) {  
    case SimpleStrategy:  
        ComposeWithSimpleCompositor();  
        break;  
    case TeXStrategy:  
        ComposeWithTeXCompositor();  
        break;  
    // ...  
    }  
    // merge results with existing composition, if necessary  
}
```

Consecuencias

- **Elimina las múltiples sentencias condicionales**
 - Muchas veces son el indicador de que necesitamos aplicar un patrón (éste u otro, según el caso)

```
void Composition::Repair () {  
    _compositor->Compose();  delegación  
    // merge results with existing composition, if necessary  
}
```

Tras aplicar el patrón Strategy

Consecuencias

- El cliente puede elegir entre varias implementaciones

¡Incluso dinámicamente!

- Los clientes deben conocer las distintas estrategias (según el GoF)

No tiene por qué ser así: el patrón también tiene sentido aunque no se cambie dinámicamente y sólo lo haga internamente el contexto, para tener un diseño flexible que nos permitiese luego acomodar futuros cambios fácilmente (o, incluso haciéndolo en tiempo de ejecución, el contexto podría tomar la estrategia concreta a partir de un fichero de configuración, sin recibirla de los clientes).

- Puede complicarse la comunicación entre el contexto y las estrategias
- Crece el número de objetos

Otros posibles usos

- **Validadores de campos de formularios**
- **Distintas modalidades de juego**
 - En un juego de póquer, dominó, etcétera
 - Niveles de dificultad en el ajedrez o un simulador de coches