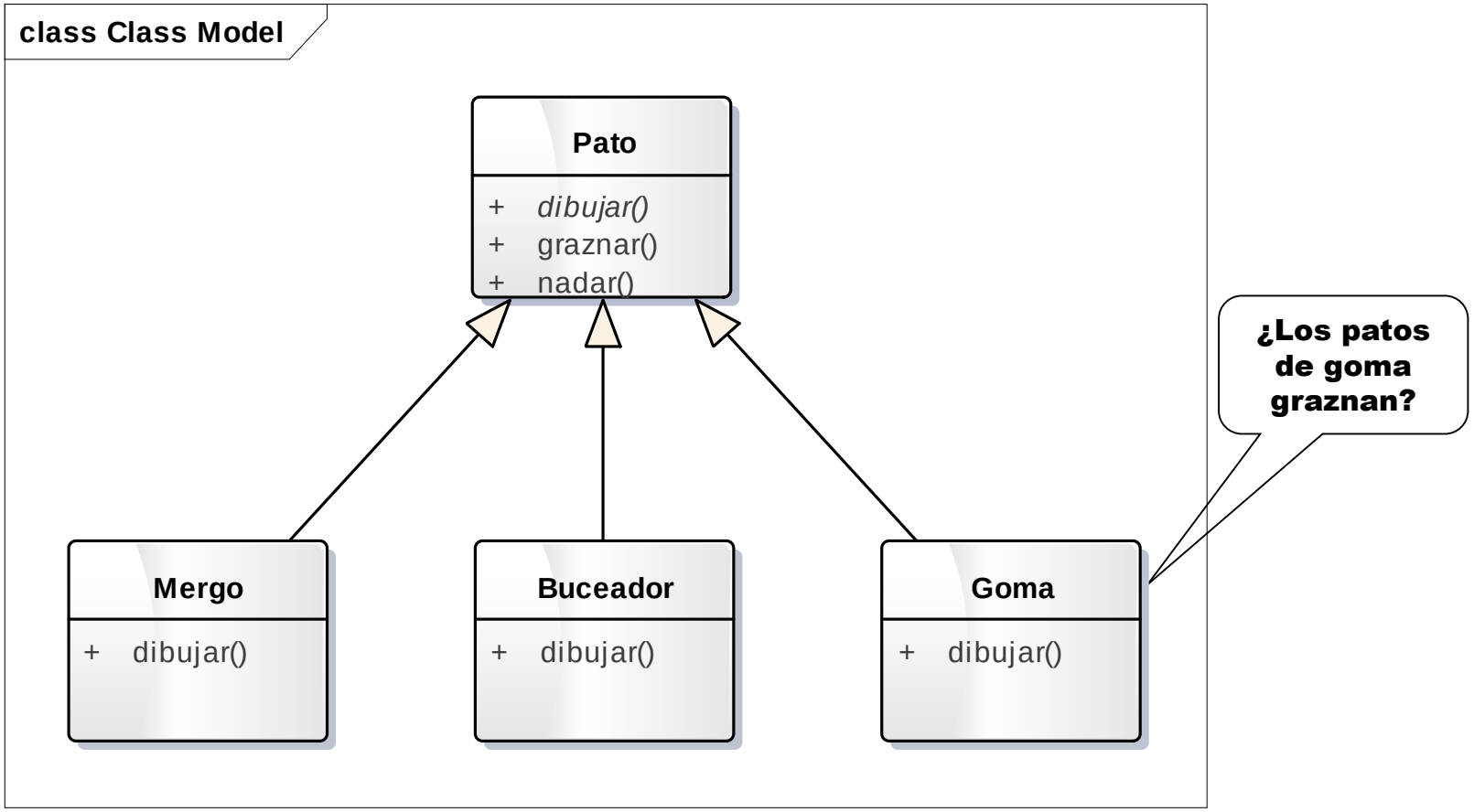
Patrones Software
Tema 1-4: Ejemplo Introducción

Planteamiento del problema

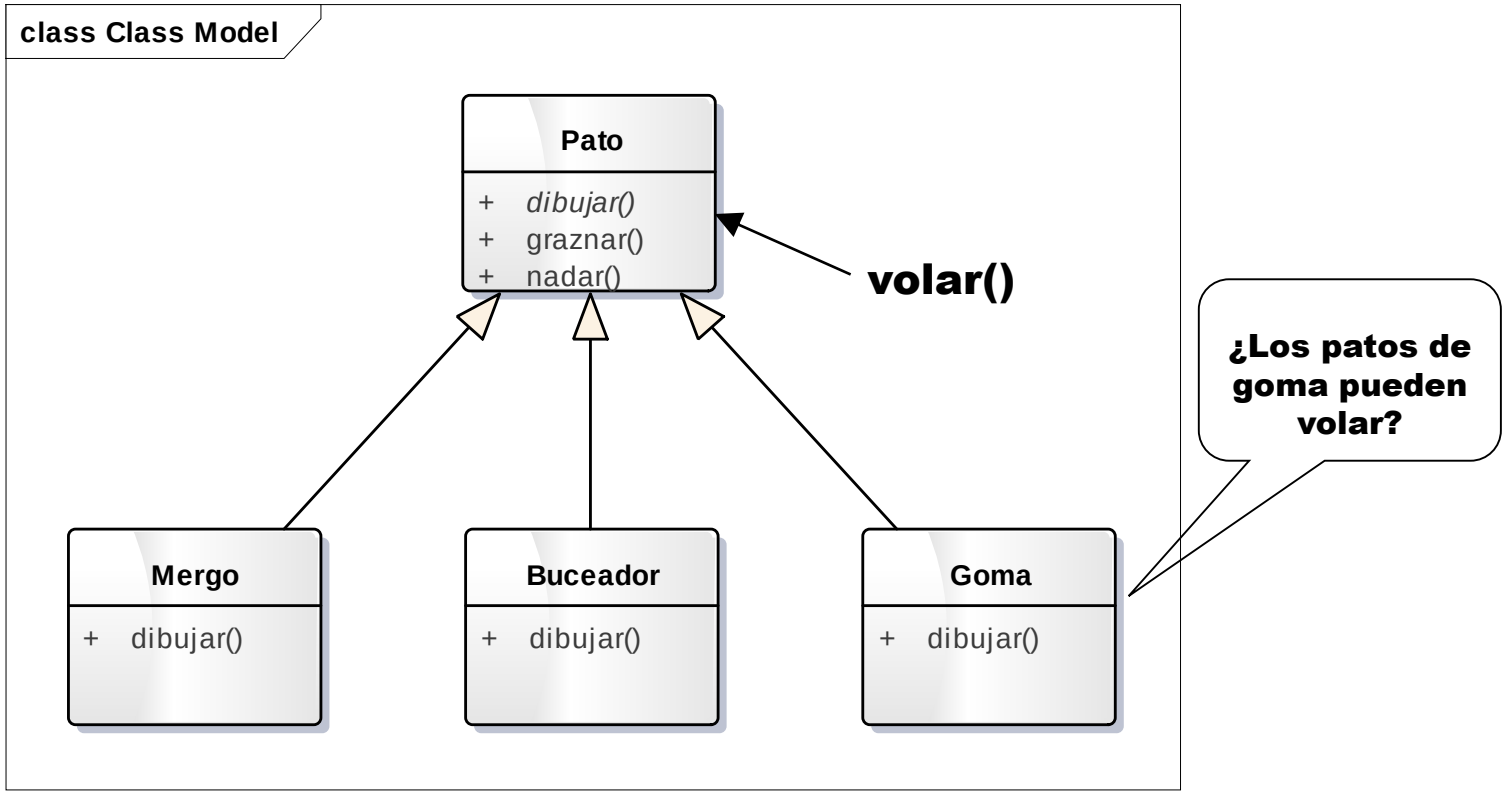
- Se pretende hacer un juego que represente un simulador de patos.
- El juego permite visualizar una gran variedad de especies nadando en un lago y graznando.
- El diseño inicial del sistema usa técnicas estándar de POO y crea una jerarquía de clases.

Primer análisis



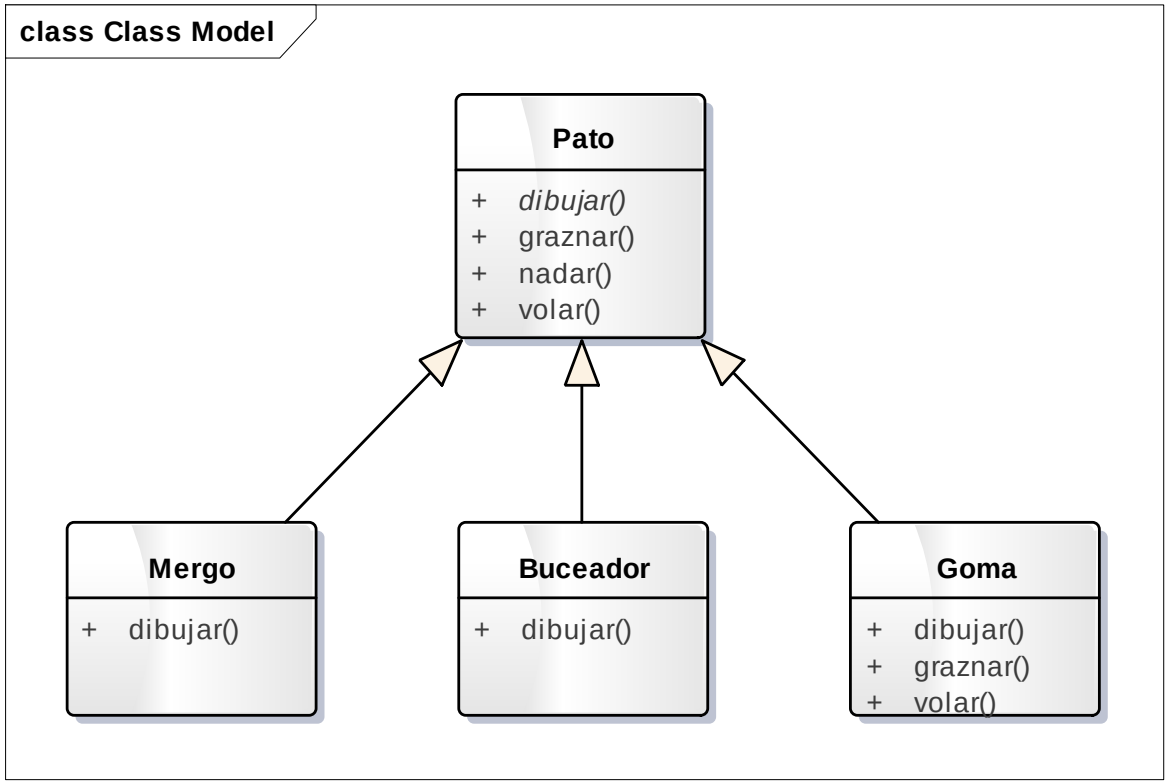
Cambios en la especificación 1

- Se desea añadir al simulador la posibilidad de que los patos vuelen.



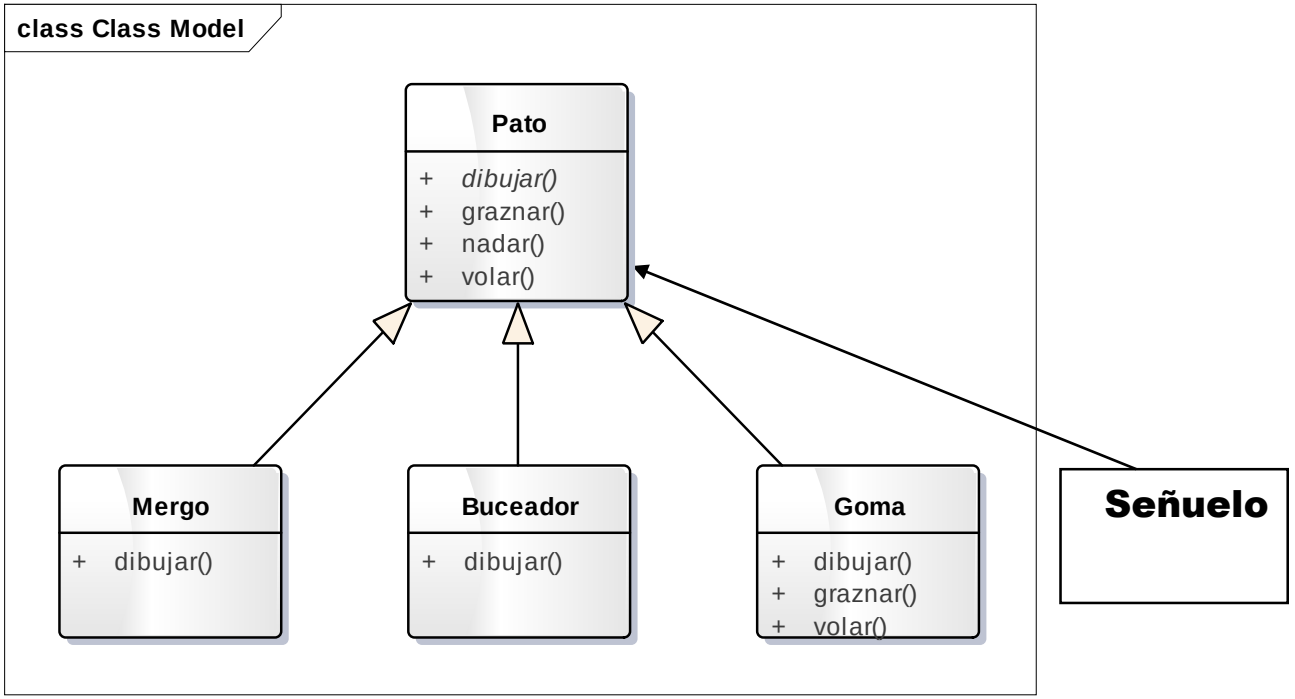
Cambios en la especificación 1. Solución

- La clase Pato de Goma redefine los métodos graznar() y volar() dejándolos vacíos.



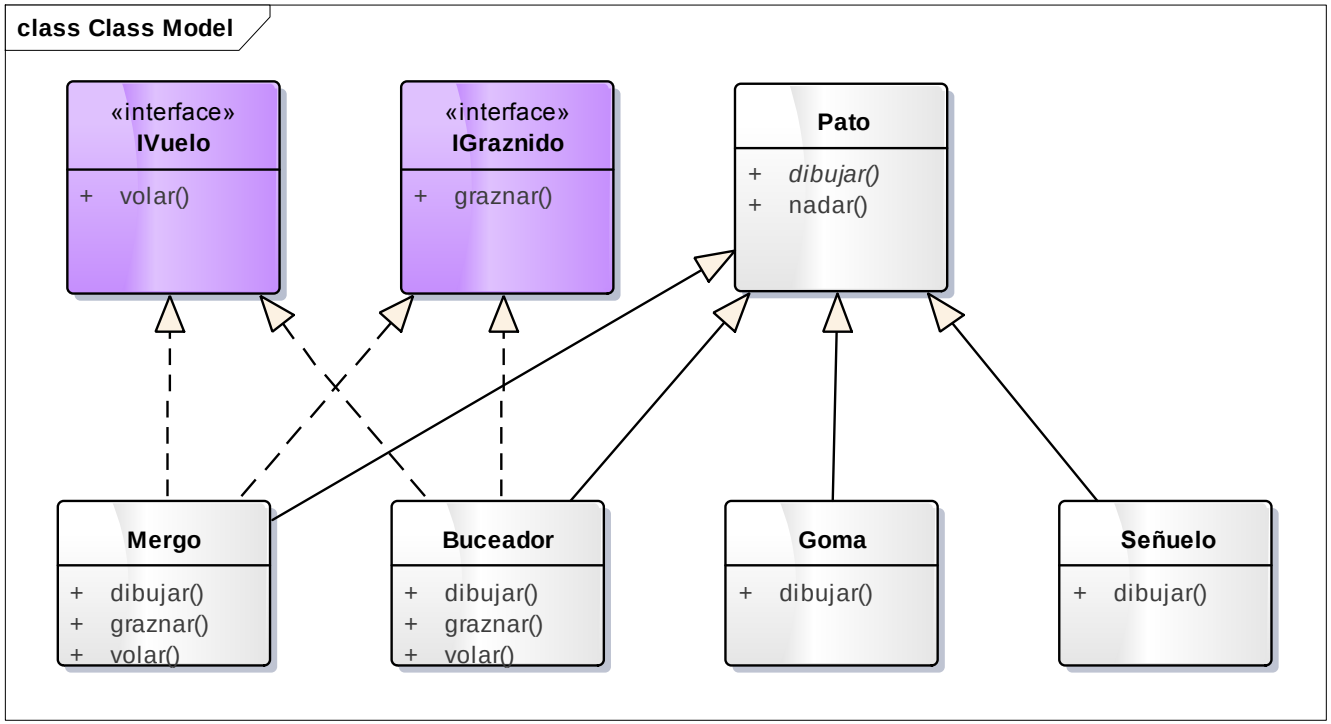
Cambios en la especificación 2

- Pero ¿que pasa si añadimos la clase Pato de Señuelo? ¿Redefinimos los métodos graznar() y volar() dejándolos vacíos?.



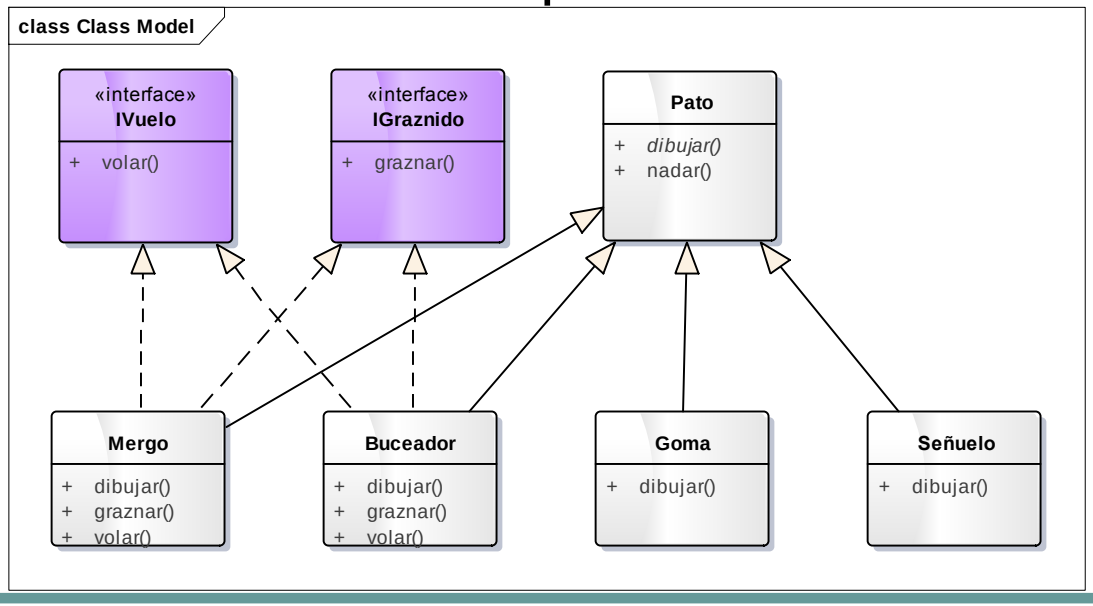
Cambios en la especificación 2. Solución

- Mejor creamos interfaces con los métodos volar y graznar y aquellas clases que los necesiten los implementan.



Cambios en la especificación 2. Solución

- ¿Pero que pasa si tenemos un montón de clases de patos que necesitan volar o graznar? Todos deben implementar los métodos. No reutilizamos el código de estos comportamientos. Por lo tanto el mantenimiento se hace imposible.



Principio de Diseño

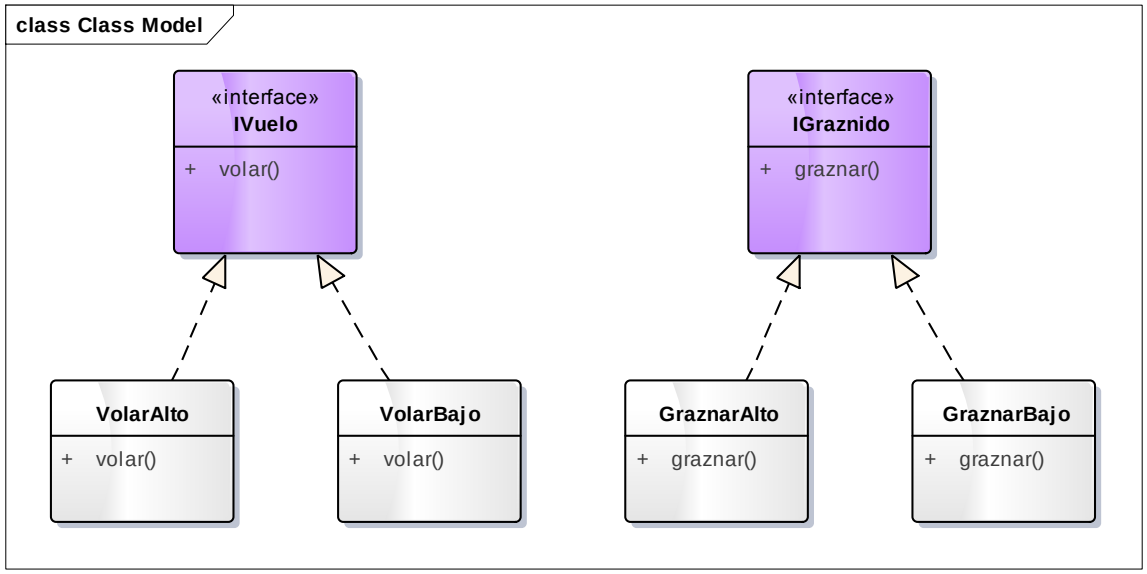
- La herencia no ha resuelto los problemas de cambio de comportamiento en las subclases, ya que no todas las subclases tienen el mismo comportamiento.
- Las interfaces están bien en principio, ya que solo las clases que necesitan un determinado comportamiento las implementan. Pero cuando tenemos que modificar un comportamiento debemos ir al código de todas las clases que han implementado el interface.
- En Java los métodos de las interfaces no tienen código por lo tanto no hay reutilización.

Principio de Diseño

- Principio de diseño:
  - Identifica aquellos aspectos de tu aplicación que varían y sepáralos de aquellos que no varían.
- Es decir, si tienes código que cambia con cada nuevo requerimiento, debes separarlo del resto que permanece invariable.
- Otra forma de verlo: Si separas las partes que varían y las encapsulas, más tarde podrás cambiarlas sin que afecten a otras partes.
- Los patrones precisamente proporcionan formas de mantener independientes ciertas partes del sistema.

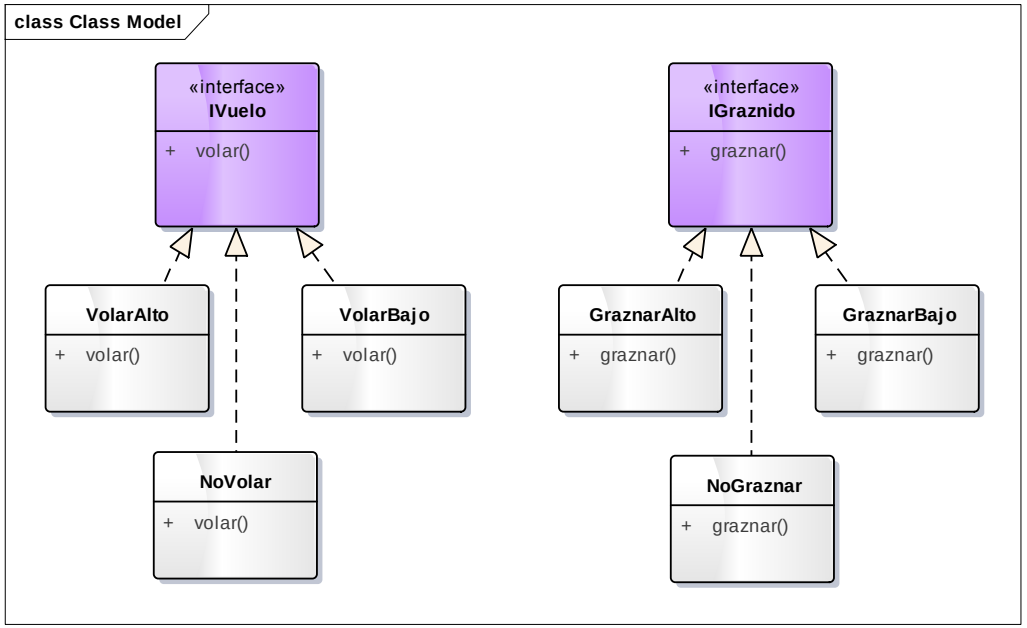
Separando comportamientos

- Para separar los comportamientos vamos a crear dos conjuntos de clases uno para volar y otro para graznar.
- Queremos tener un comportamiento predefinido de volar y graznar pero que incluso podamos cambiar en tiempo de ejecución.



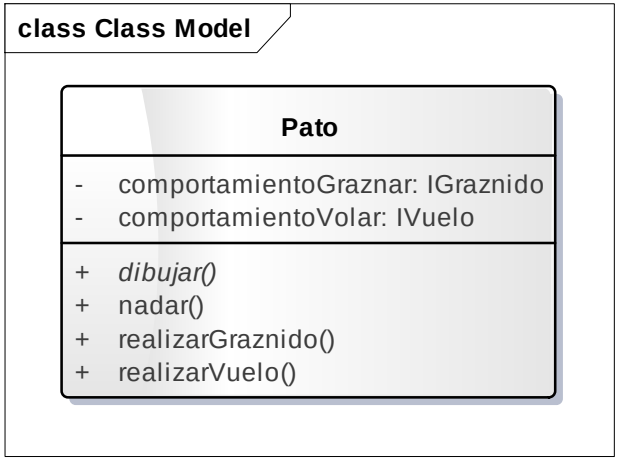
Separando comportamientos

- Con este diseño otros tipos de objetos pueden reutilizar los comportamientos de volar y graznar ya que no están escondidos en la clase Pato.
- Podemos añadir nuevos comportamientos sin modificar ningún comportamiento anterior ni las clases que heredan de Pato.



Integrando comportamientos

- Ahora la clase Pato delega su comportamiento, en vez de definirlo dentro.
- Estamos utilizando el patrón de comportamiento ***estrategia*** que define una familia de algoritmos, los encapsula y los hace intercambiables, estos pueden cambiar de forma independiente al uso que hace el cliente de ellos.



Estructura final