Duck Simulator

# La Solución

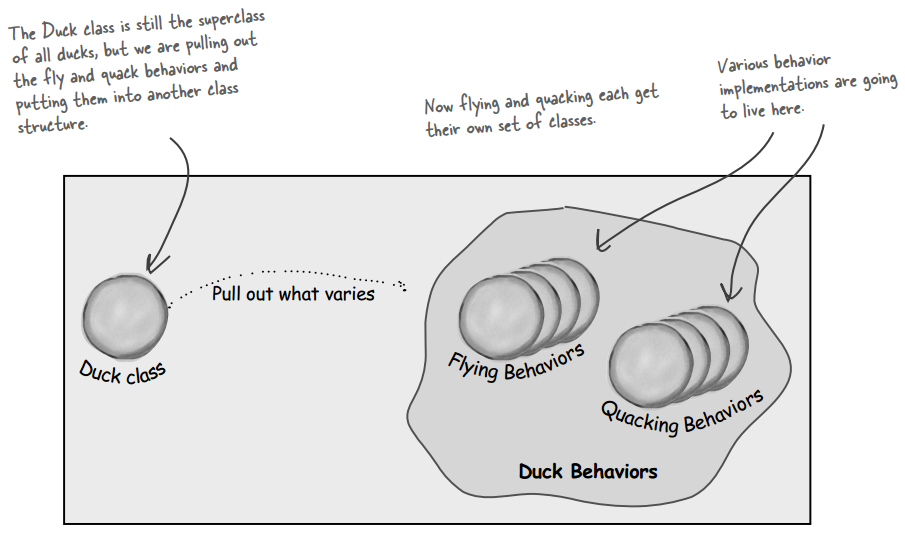
Sabemos que la herencia no está funcionando ya que los comportamientos de los patos se mantienen en cambiando y no es apropiado que todas las subclases tengan esos comportamientos.

**Design Principle “Encapsulate what varies”:** Identificar los aspectos de la aplicación que cambian con nuevos requerimientos y encapsularlos, de la manera que luego podamos alterar o extender estar partes sin afectar el resto del código que se mantiene constante.

¿Qué partes de la aplicación varían o cambian frecuentemente?

Los métodos fly() y quack() de la clase Duck varían entre patos.

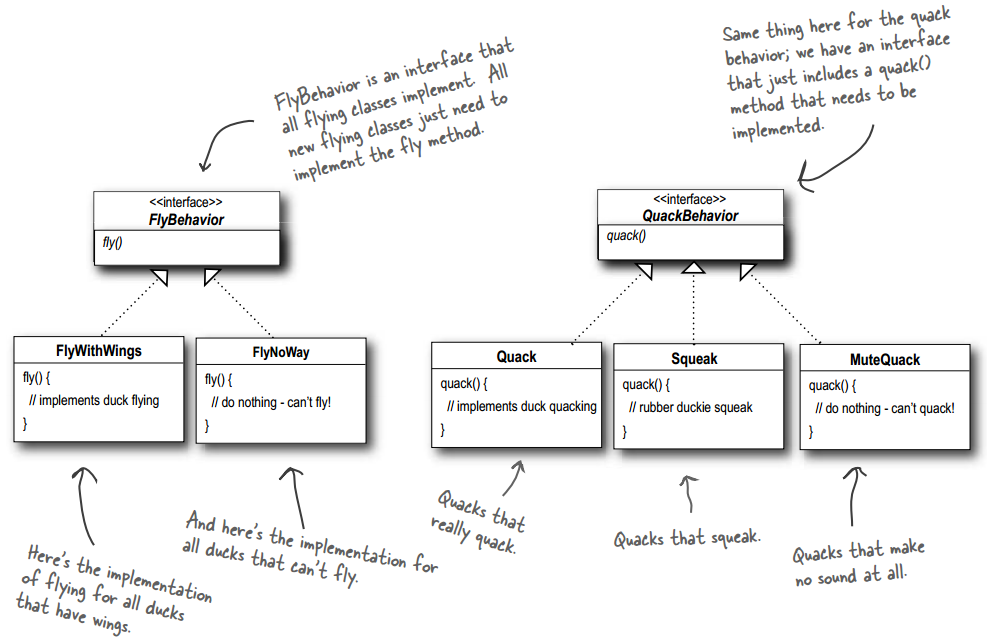
Sacamos los comportamientos “fly” y “quack” fuera de la clase Duck y creamos un nuevo conjunto de clases para representar cada uno de estos comportamientos.

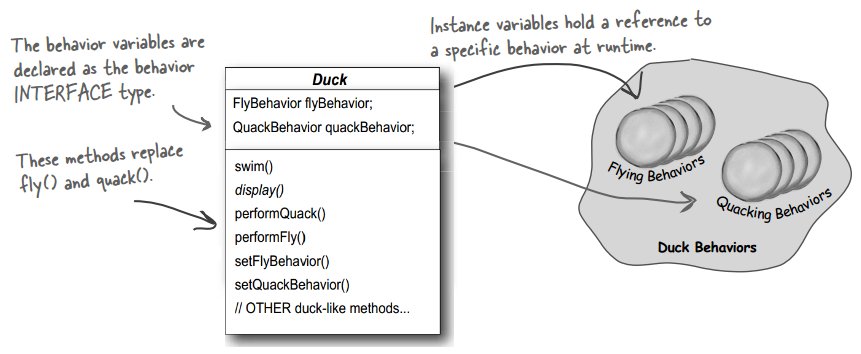


Dibuja un diagrama de clases que represente el nuevo diseño de la aplicación. Debe cumplir lo siguiente:

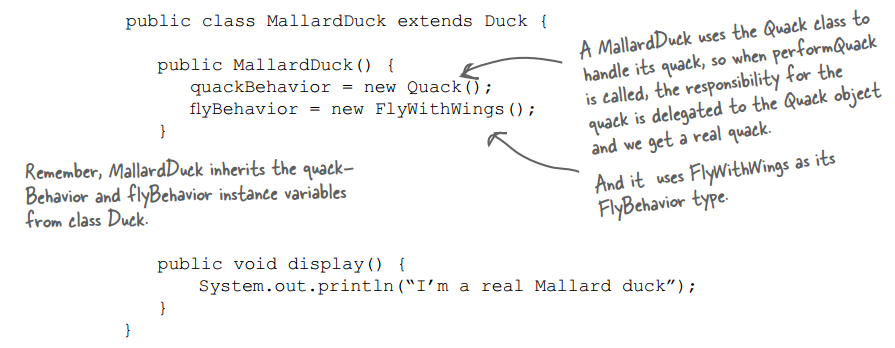
* Utilizar una interfaz para representar cada comportamiento (FlyBehaviour, QuackBehaviour), cada implementación de un comportamiento debe implementar alguna de estar interfaces.
* Por el momento existen 2 comportamientos “fly” y 3 comportamientos “quack”.
* Poder asignar los comportamientos a las subclasses de Duck. Ejemplo, instanciar un MallardDuck e inicializarlo con un tipo específico de FlyBehaviour.
* Las subclases de Duck delegan sus comportamientos “fly” y “quack” a las instancias de FlyBehaviour y QuackBehaviour.
* Cambiar dinámicamente los comportamientos “fly” y “quack” en las subclases.

# Class Diagram





Escribe la implementación de la clase MallardDuck.



Qué beneficios tiene este nuevo diseño.

* Otros objetos pueden reutilizar los comportamientos “fly” y “quack”. Tenemos los beneficios de la reutilización sin los problemas de la herencia.
* Agregar nuevos comportamientos sin modificar o tocar ninguna de las clases ya existentes.
* Cambiar los comportamientos en tiempo de ejecución.

