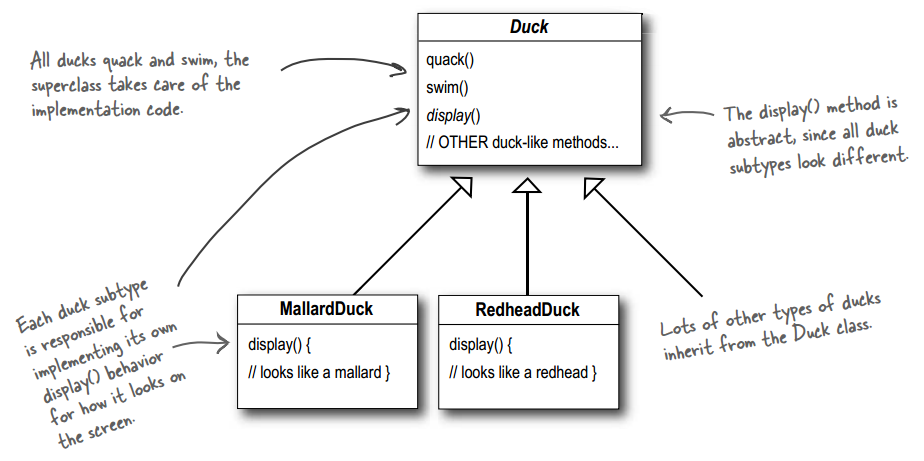
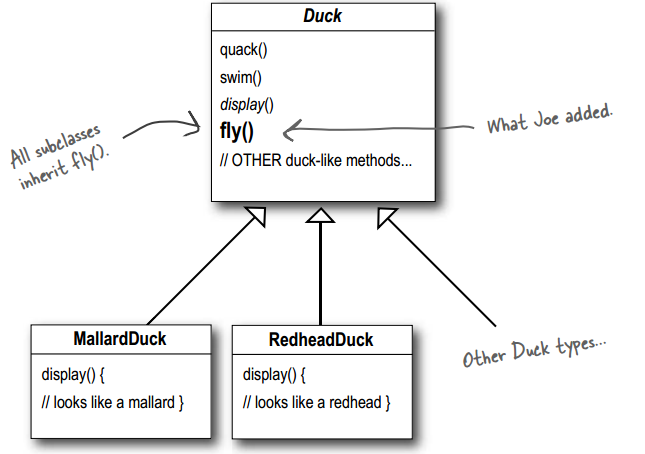
Duck Simulator

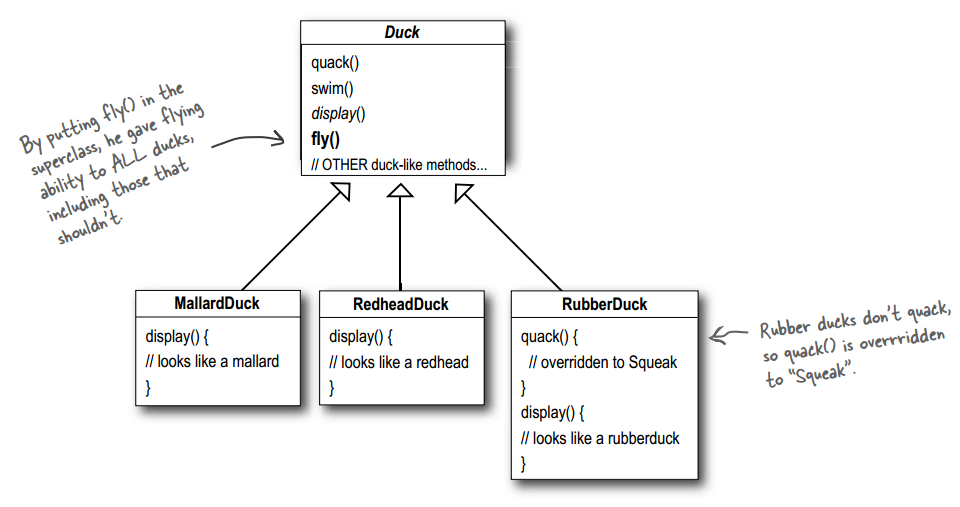
Joe para una compañía que ha creado un juego de simulación muy exitoso. Este juego muestra diversas especies de patos “nadando” y “graznando”. Los diseñadores iniciales del juego utilizaron técnicas OO estándar y crearon el siguiente diseño.



Debido a la competencia, la compañía cree que es tiempo de una gran innovación. Los ejecutivos decidieron que “patos voladores” es lo que necesitan para acabar con sus competidores. Joe piensa que es una solución muy sencilla y realiza el siguiente cambio en el diseño:

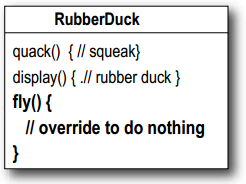


Pero ocurre un gran problema, en una revisión del producto con los stakeholders observaron que los patos de goma empezaron a volar por toda la pantalla. Joe no se dio cuenta que no todos los patos deben volar.

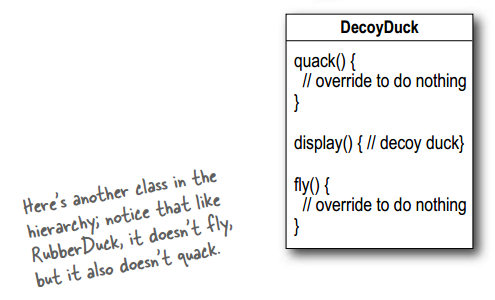


Lo que él pensaba que era un gran uso de la herencia para reutilizar comportamiento, no resultó tan bien a nivel de mantenimiento.

Joe piensa que podría solucionar el problema sobre-escribiendo métodos.



Pero que sucede cuando tiene que agregar un nuevo pato que no deba volar o hacer sonido….



¿Cuáles son las siguientes desventajas de utilizar herencia para diseñar el comportamiento de los patos?

|  |  |
| --- | --- |
| 1. Código duplicado en las subclasses. 2. Cambios del comportamiento en ejecución son difíciles. 3. No podemos lograr que los patos bailen. | 1. Difícil de consolidar cuáles son todos los comportamientos de los patos. 2. Patos no pueden volar y graznar al mismo tiempo. 3. Cambios pueden, no intencionalmente, afectar a otros patos. |

# La Solución

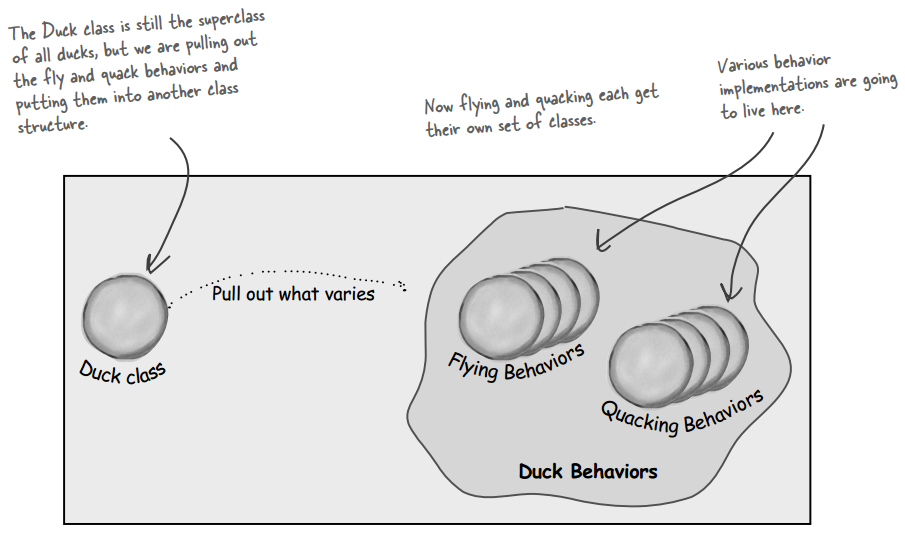
Sabemos que la herencia no está funcionando ya que los comportamientos de los patos se mantienen en cambiando y no es apropiado que todas las subclases tengan esos comportamientos.

**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.

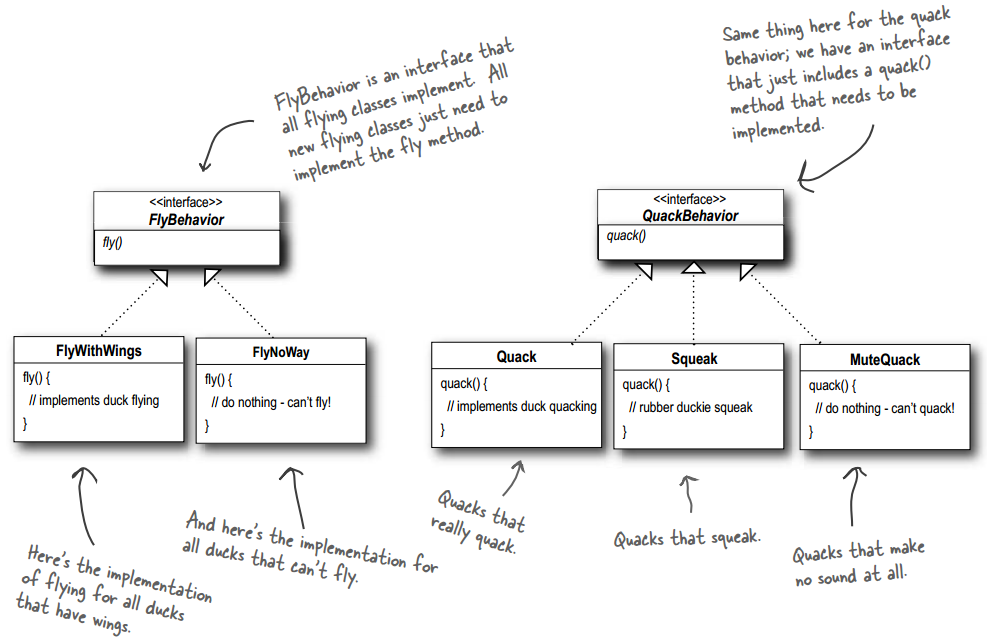
Sacamos los comportamientos “fly” y “quack” fuera de la clase Duck y creamos un nuevo conjunto de clases para representar cada uno de estos comportamientos.

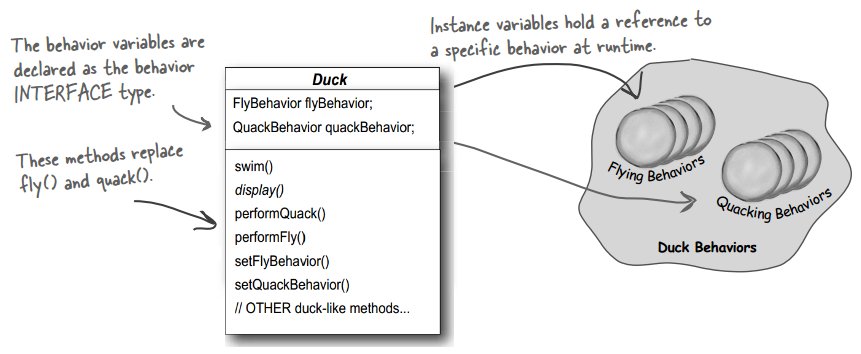


Dibuja un diagrama de clases que represente el nuevo diseño de la aplicación. Debe cumplir lo siguiente:

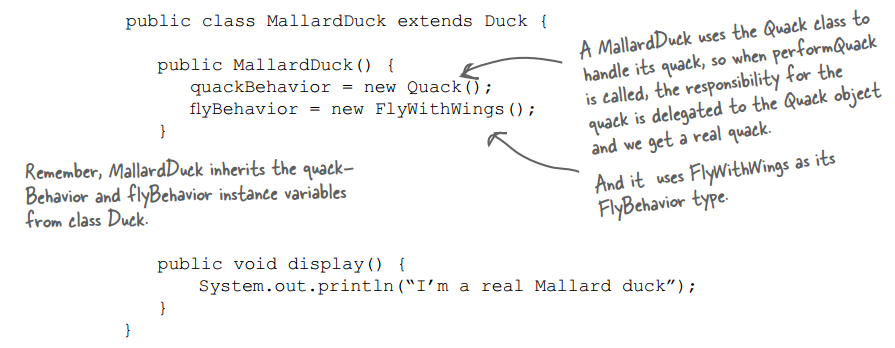
* Utilizar una interfaz para representar cada comportamiento (FlyBehaviour, QuackBehaviour), cada implementación de un comportamiento debe implementar alguna de estar interfaces.
* Por el momento existen 2 comportamientos “fly” y 3 comportamientos “quack”.
* Poder asignar los comportamientos a las subclasses 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





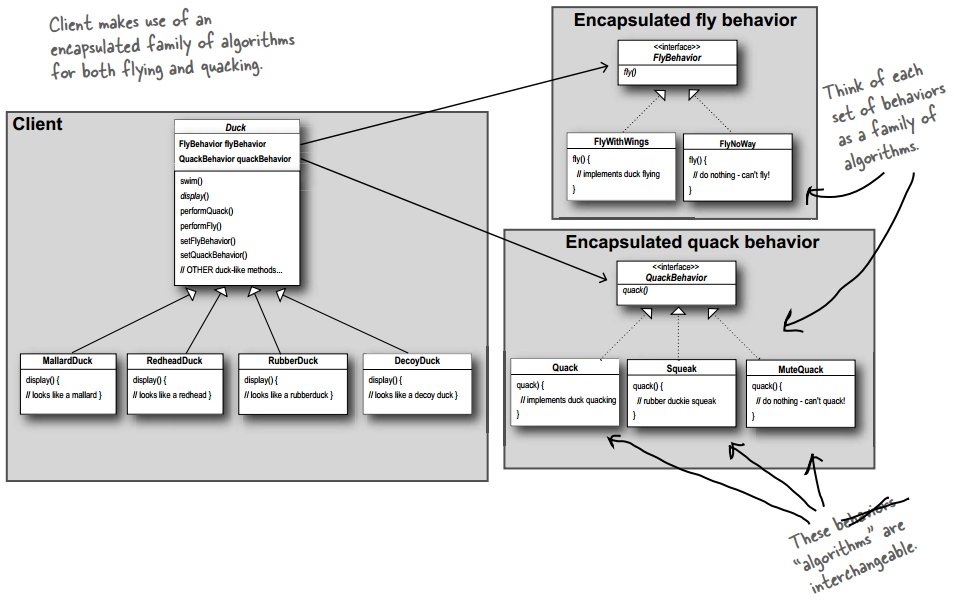
Escribe la implementación de la clase MallardDuck.



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.

# Your First Design Pattern



# Strategy Design Pattern

Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables, el cliente puede elegir que algoritmo usar dinámicamente según sus necesidades.