

4 Strategy

Diseño del Software

Grado en Ingeniería Informática del Software

Curso 2024-2025

Introducción

Simulador de patos

O'REILLY®

Second
Edition

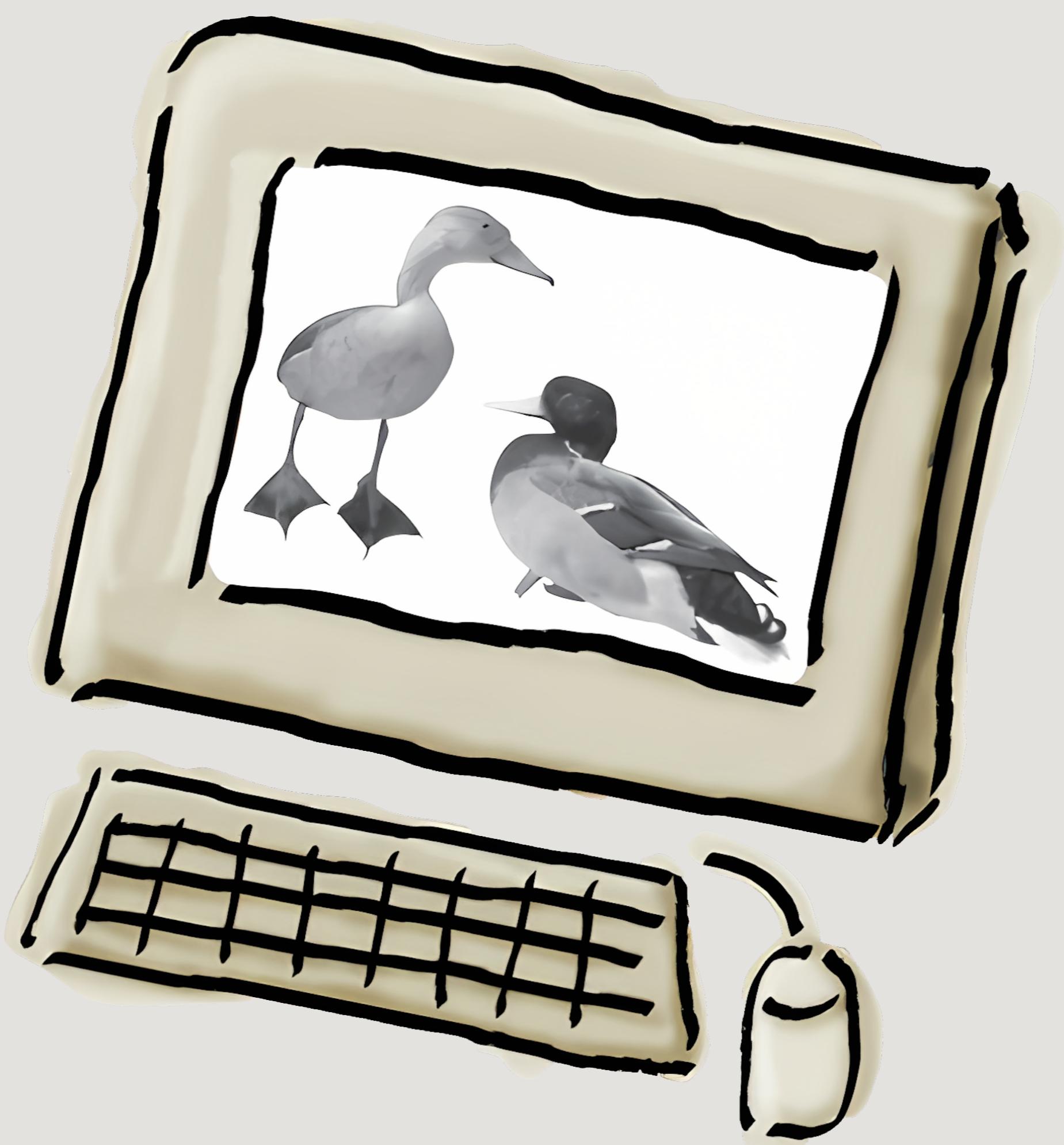
Head First Design Patterns

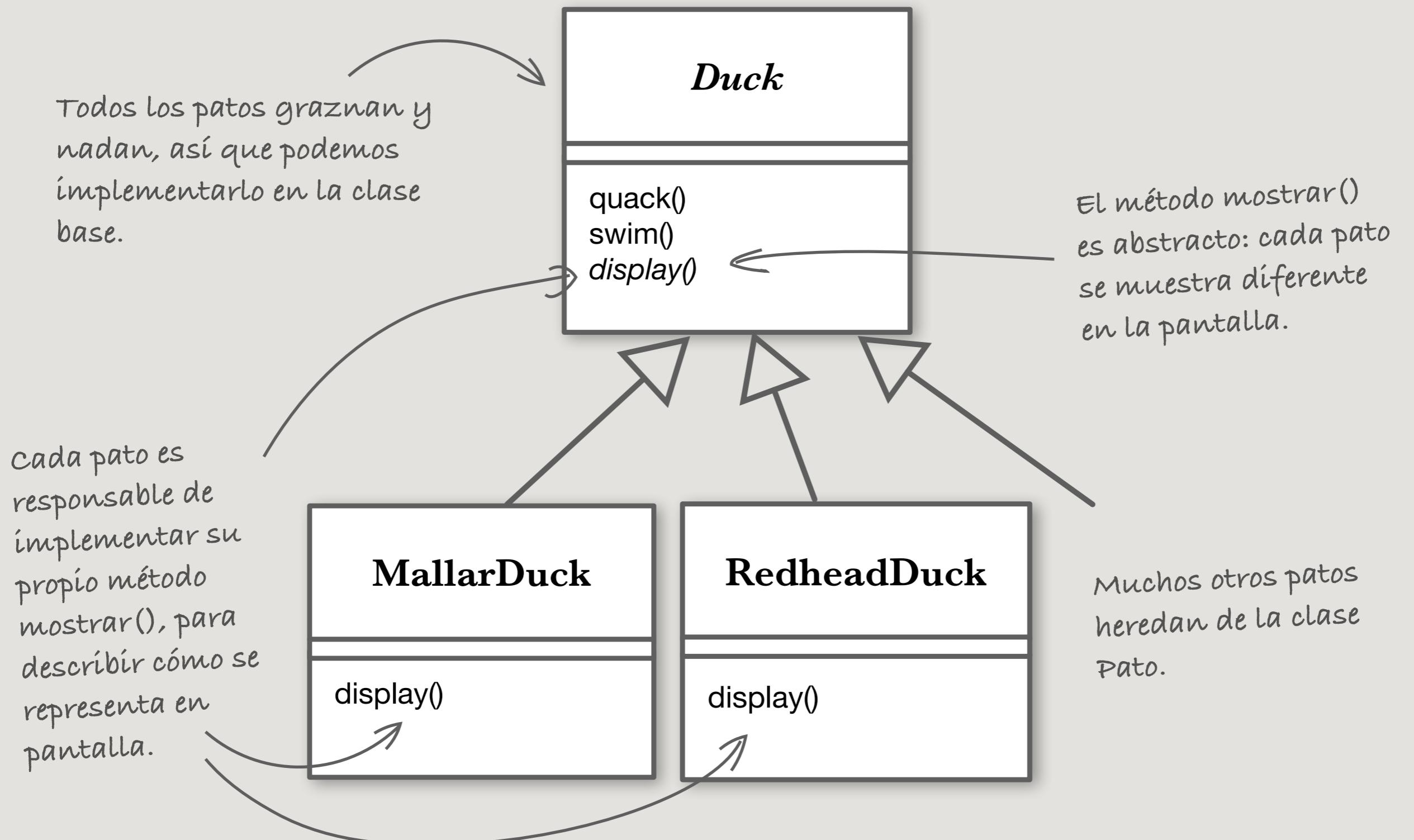
Building Extensible
& Maintainable
Object-Oriented
Software

Eric Freeman &
Elisabeth Robson
with Kathy Sierra & Bert Bates



A Brain-Friendly Guide



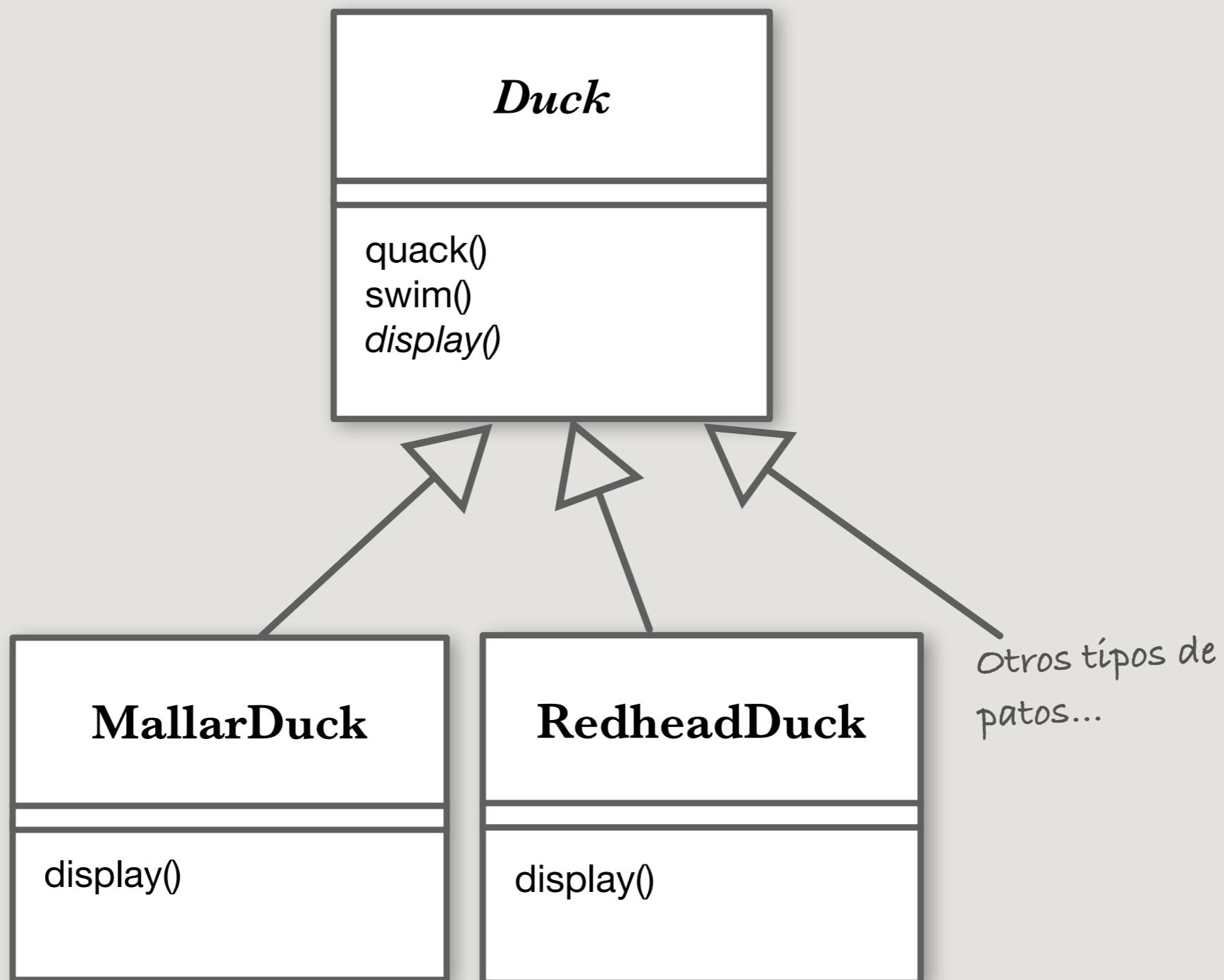


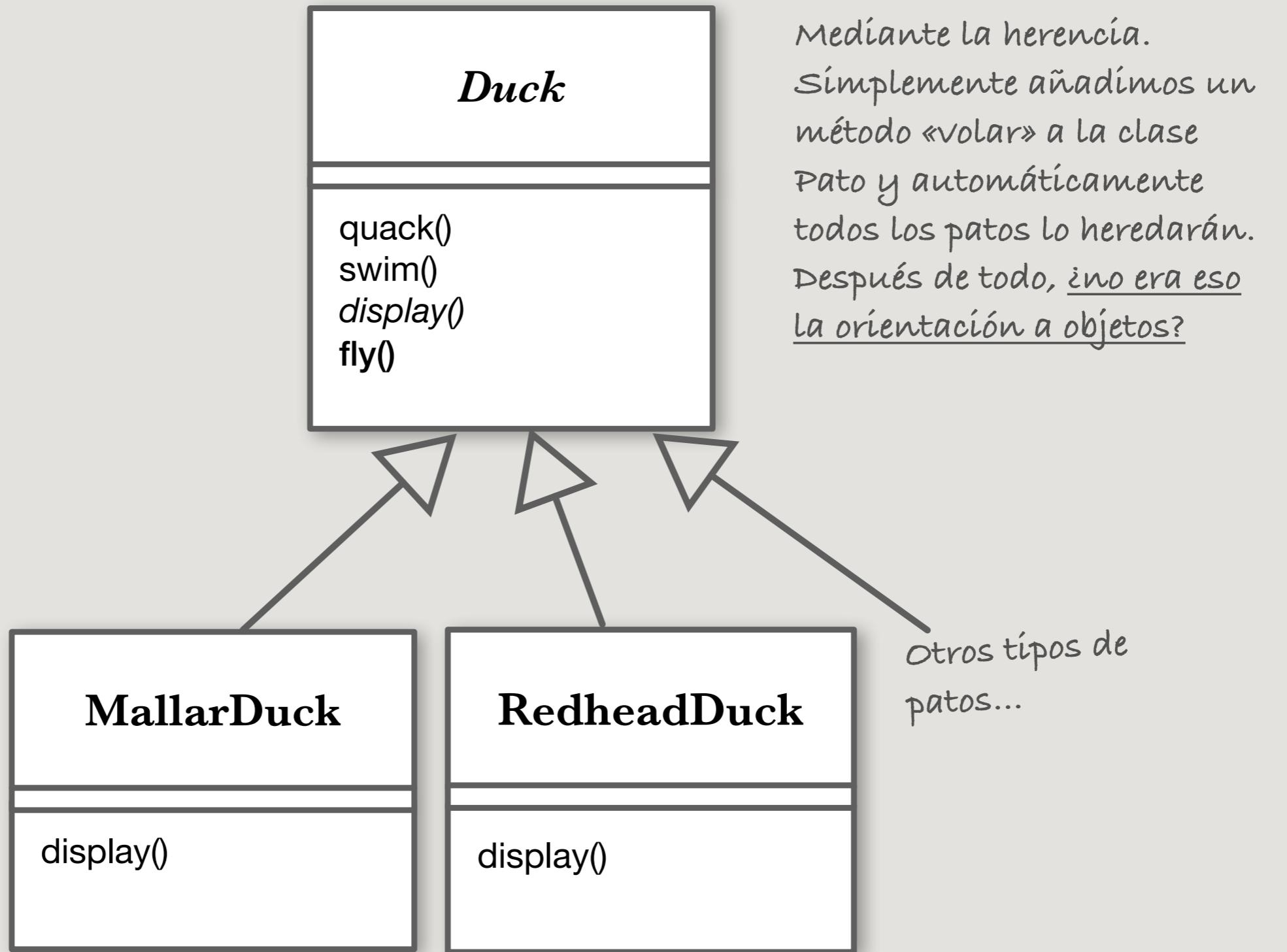
Ahora los patos...

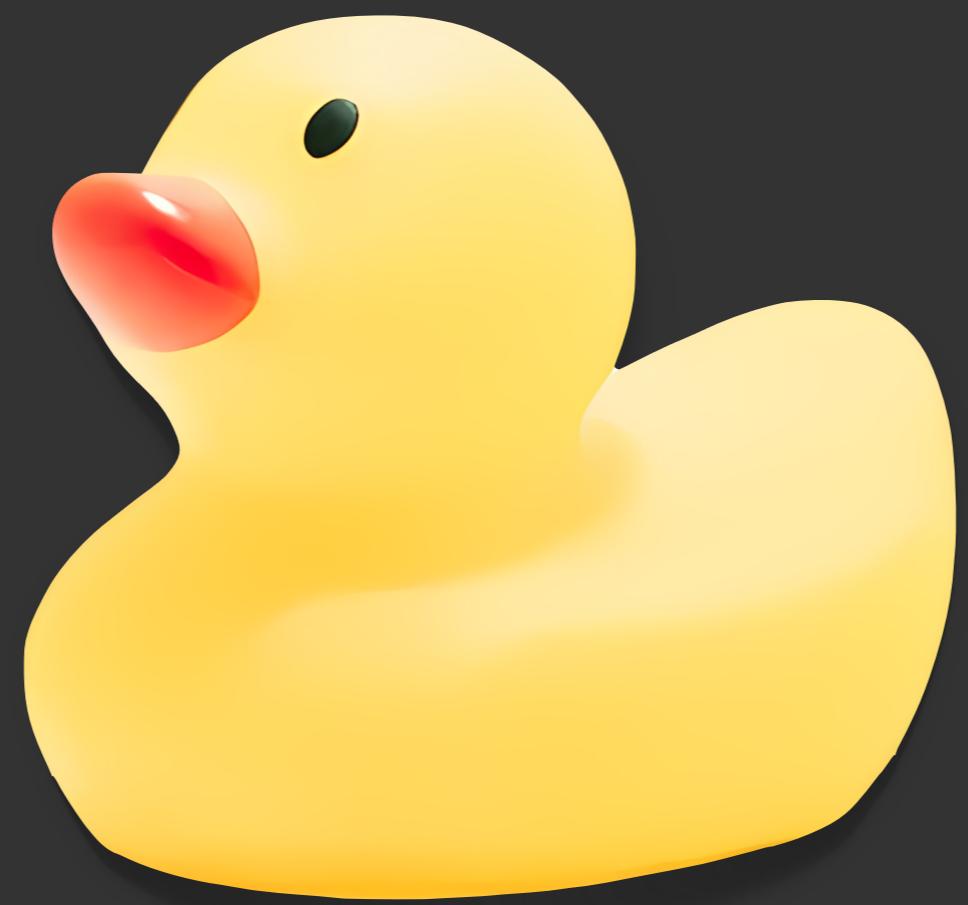
¡Vuelan!

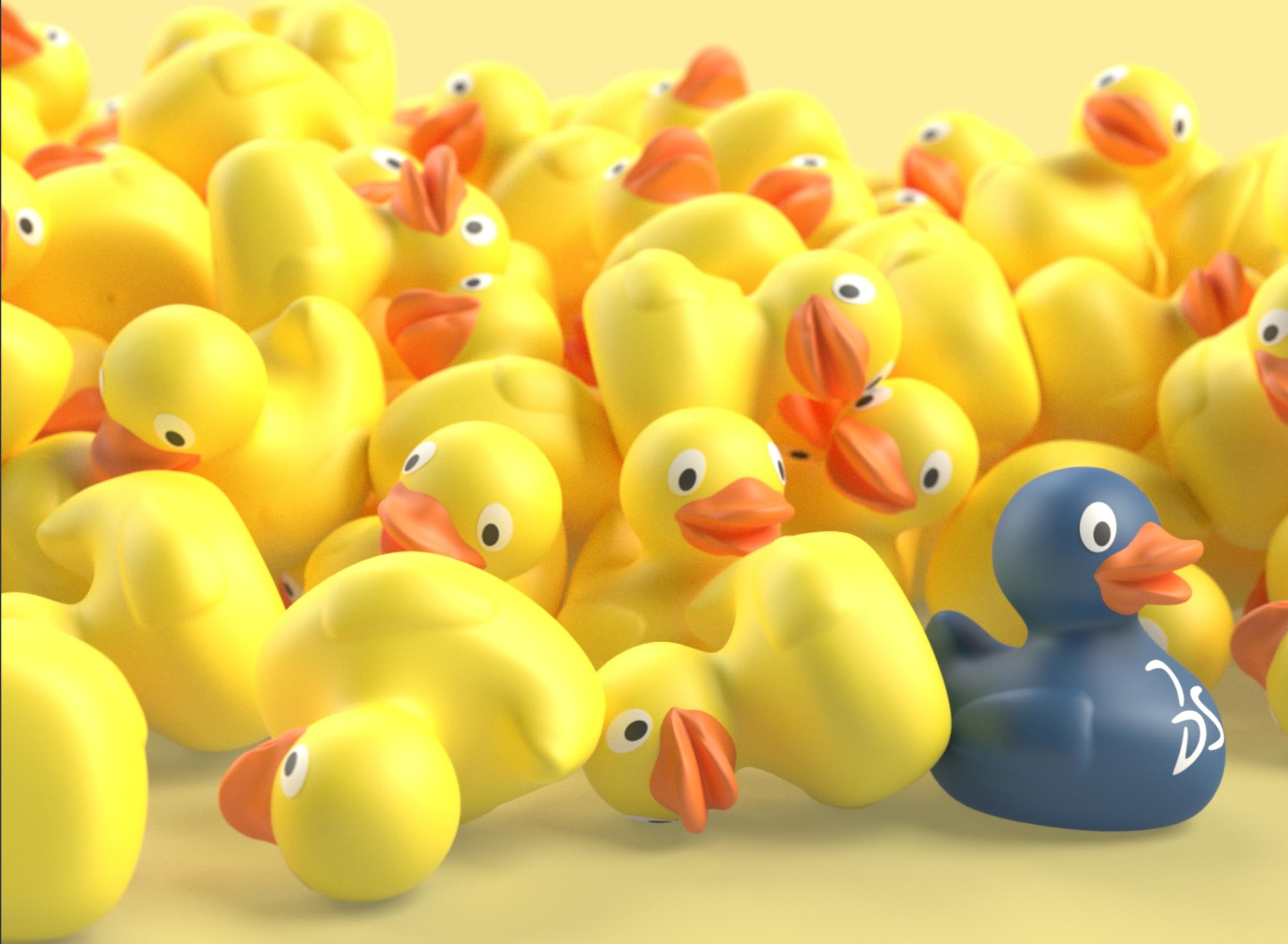
¿Cómo lo hacemos?











Ahora tenemos un montón de patos de goma volando por la pantalla.

Al implementar volar en la superclase, hemos dado esa habilidad a todos los patos, incluyendo aquellos que no deberían.*



Aquí debemos entender «no deberían» como que la implementación de volar es distinta, y que no volar no es más que otra forma de volar (del mismo modo que lo habría sido, por ejemplo, volar montado en un cohete).

Cuidado

Es preciso matizar algo importante con respecto a la explicación del ejemplo en el libro.

El problema que pretende resolver el patrón no tiene nada que ver con que unos patos vuelen y otros no.

El libro «Head First Design Patterns» del que está tomado este ejemplo comete unos cuantos errores en su explicación.

Aquí, supondremos que «no volar» no es más que una variante válida de la operación volar, y que tiene sentido llamarla sobre cualquier tipo de pato (aunque no haga nada).

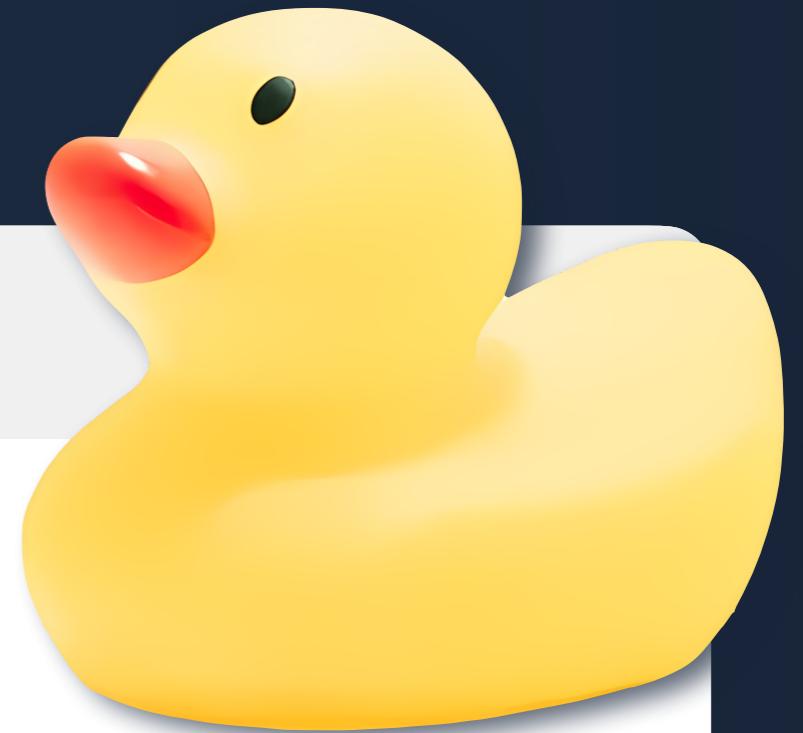
En otras palabras, consideraremos «no volar» como **otra forma de volar** (un comportamiento diferente del método «volar»), algo que sí representa fielmente el problema que pretende resolver este patrón.

Si no fuera así—es decir, si no se cumple el Principio de Sustitución de Liskov—, entonces el patrón no sería aplicable.



RubberDuck.java

```
class RubberDuck extends Duck {  
  
    public void fly() {  
        // do nothing  
    }  
}
```



Después de todo, si suponemos que es un comportamiento válido, redefinir volar para que no haga nada parece una decisión perfectamente razonable (tal como haríamos con cualquier otra forma de volar).

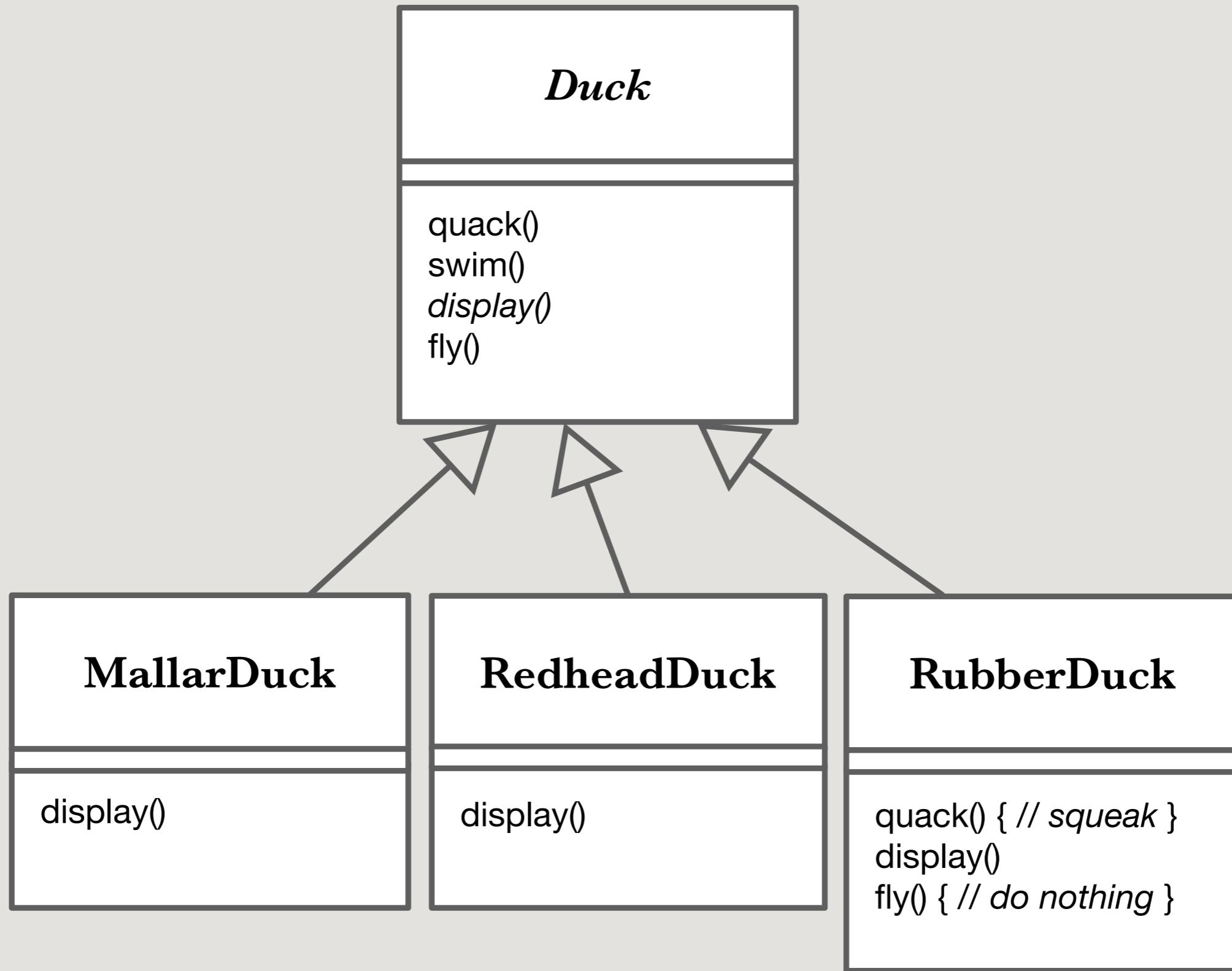
O como ocurre, por ejemplo, con el método quack (graznar): los patos de goma no graznan, sino que emiten un chifrido característico cuando se les aprieta.

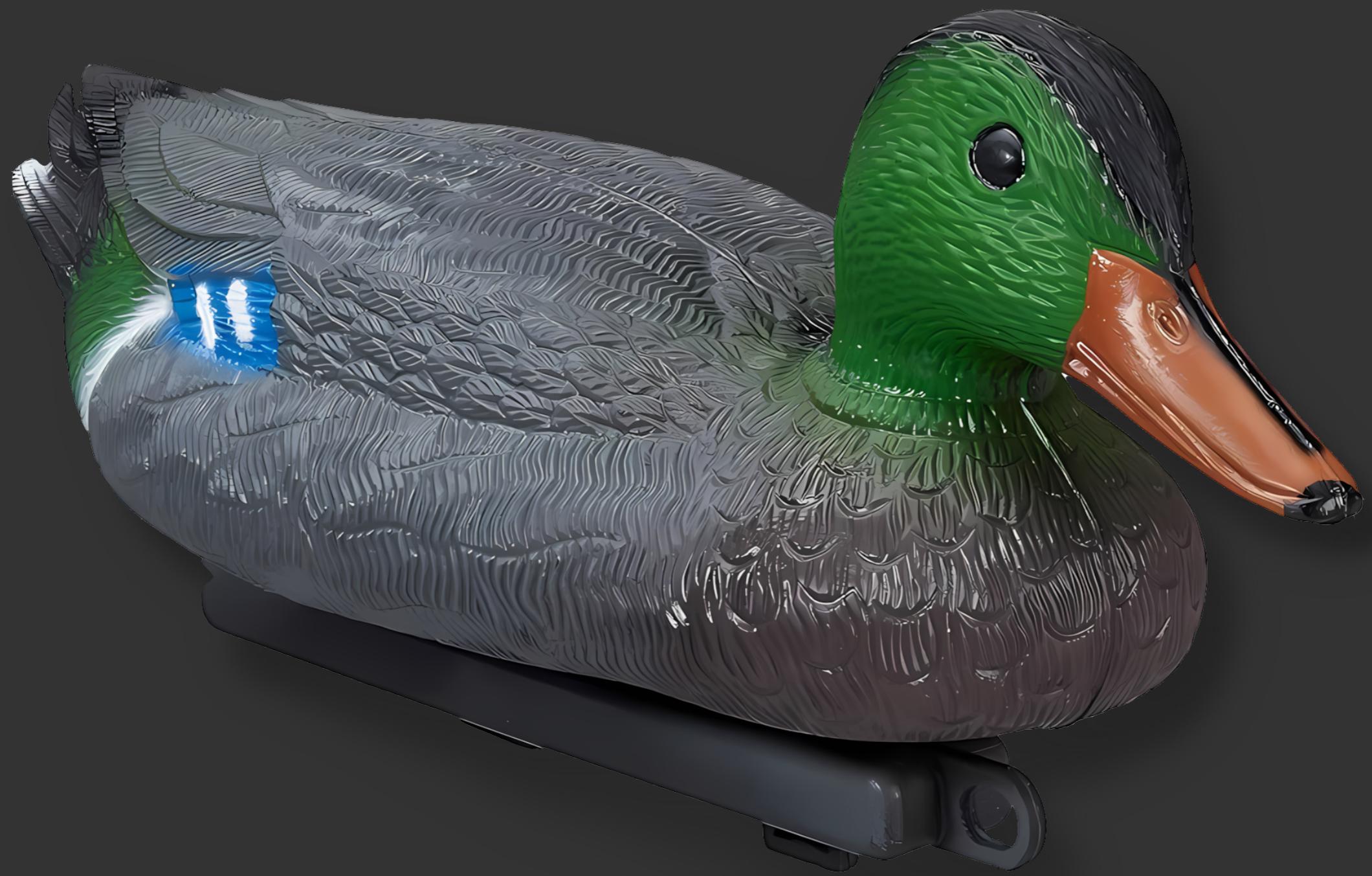


RubberDuck.java

```
class RubberDuck extends Duck {  
  
    public void quack() {  
        squeak();  
    }  
  
    public void fly() {  
        // do nothing  
    }  
}
```

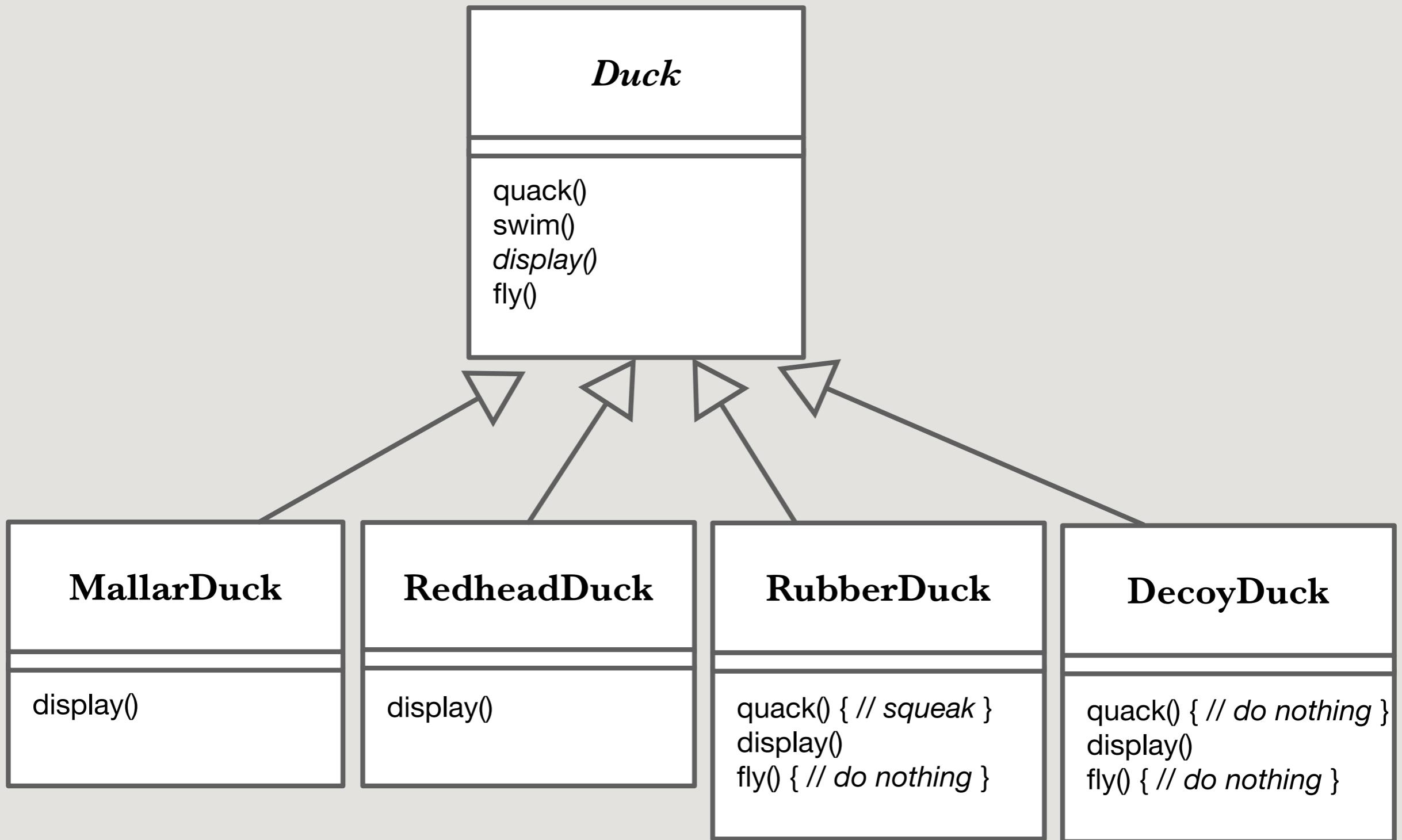






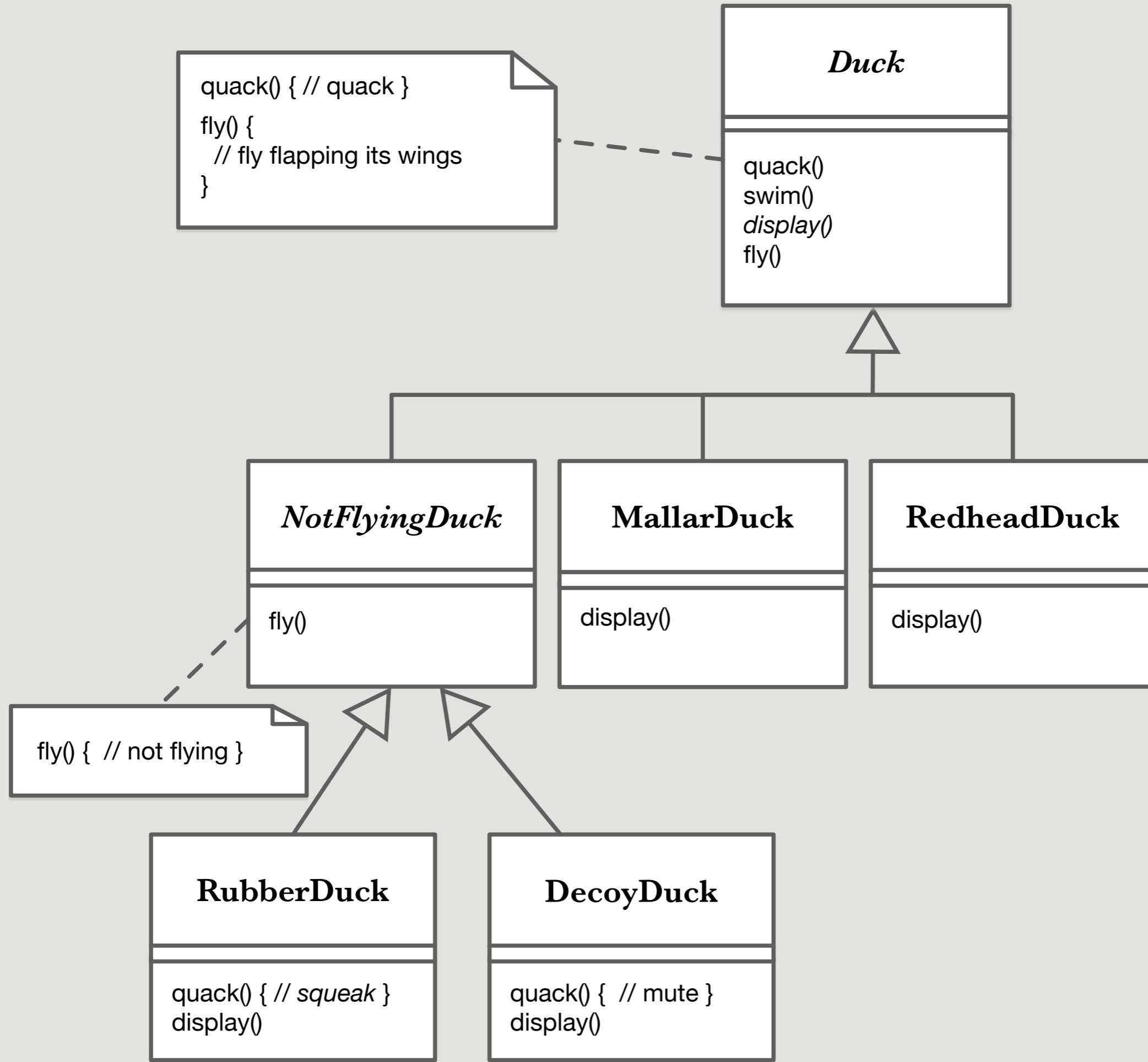
Los patos de reclamo se usan
como cebo para cazar.

Ni graznan ni vuelan.

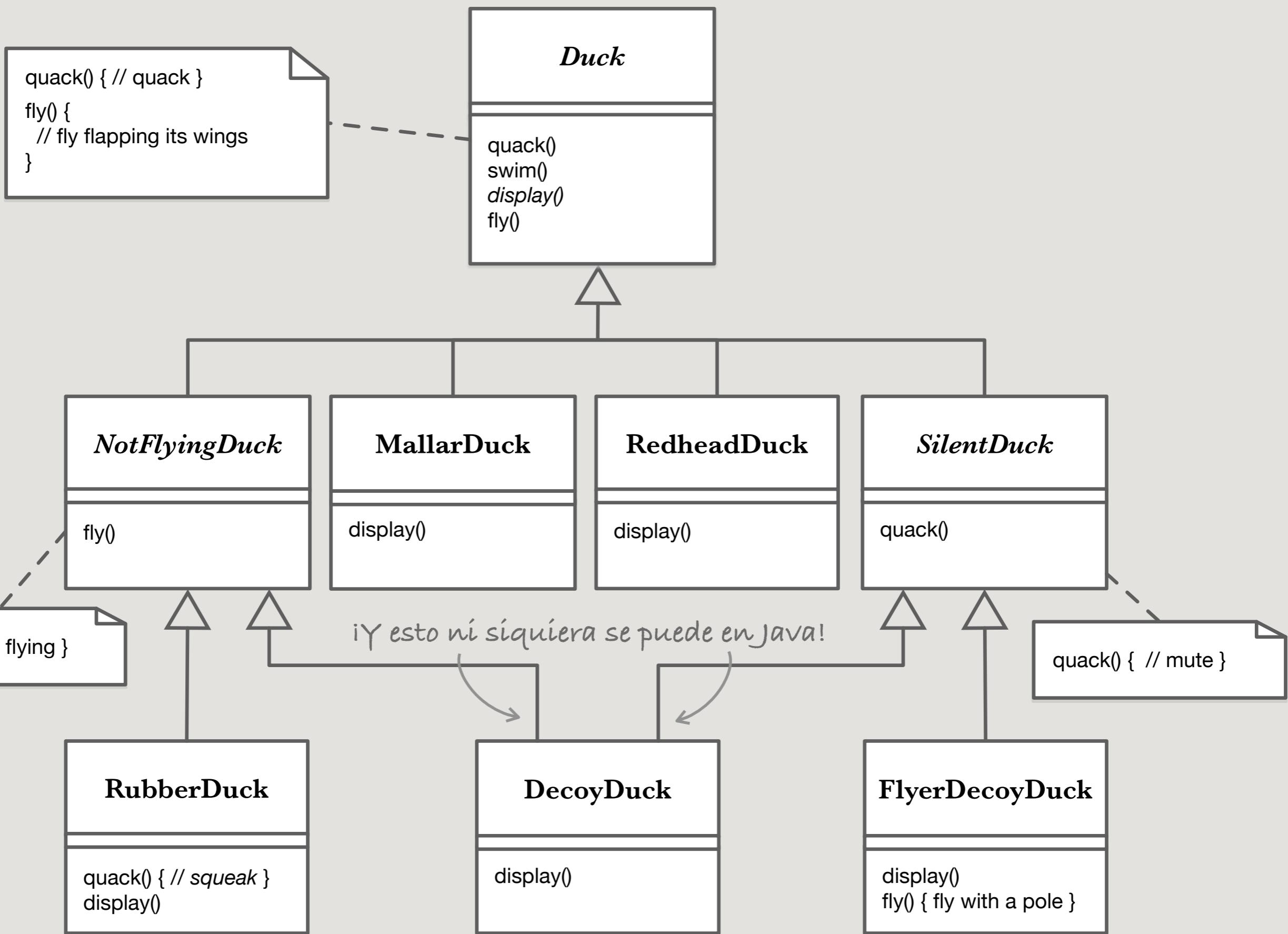


Lógica duplicada

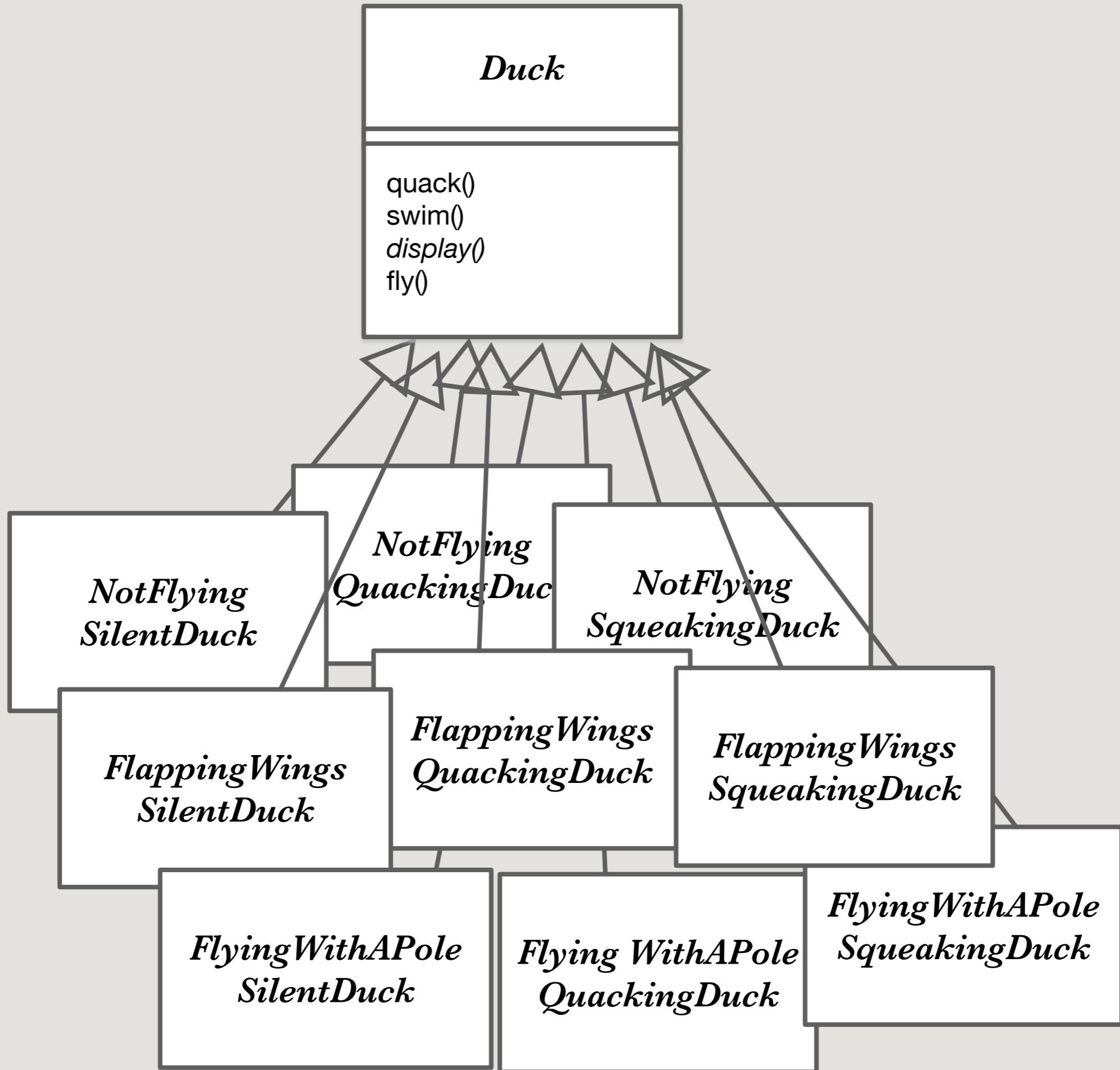
Ahora tenemos dos tipos de patos que no vuelan.







*¡Se empieza
a complicar!*



*¡Explosión
de subclases!*

¿Podemos ver un patrón aquí?

El comportamiento de volar y graznar ni es común a todas las clases ni tampoco específico de cada una.

Por el contrario, da la impresión de que tenemos diferentes comportamientos de cada una de esas operaciones compartidos por distintas clases.

¿Qué pasaría ahora si algunos patos volasen montados en un cohete?

si hay una constante en el desarrollo de software, esta es...

¡El cambio!

Se va a producir.

si hay una constante en el desarrollo de software, esta es...

¡El cambio!

siempre.

*uno de los principios
básicos del diseño OO*

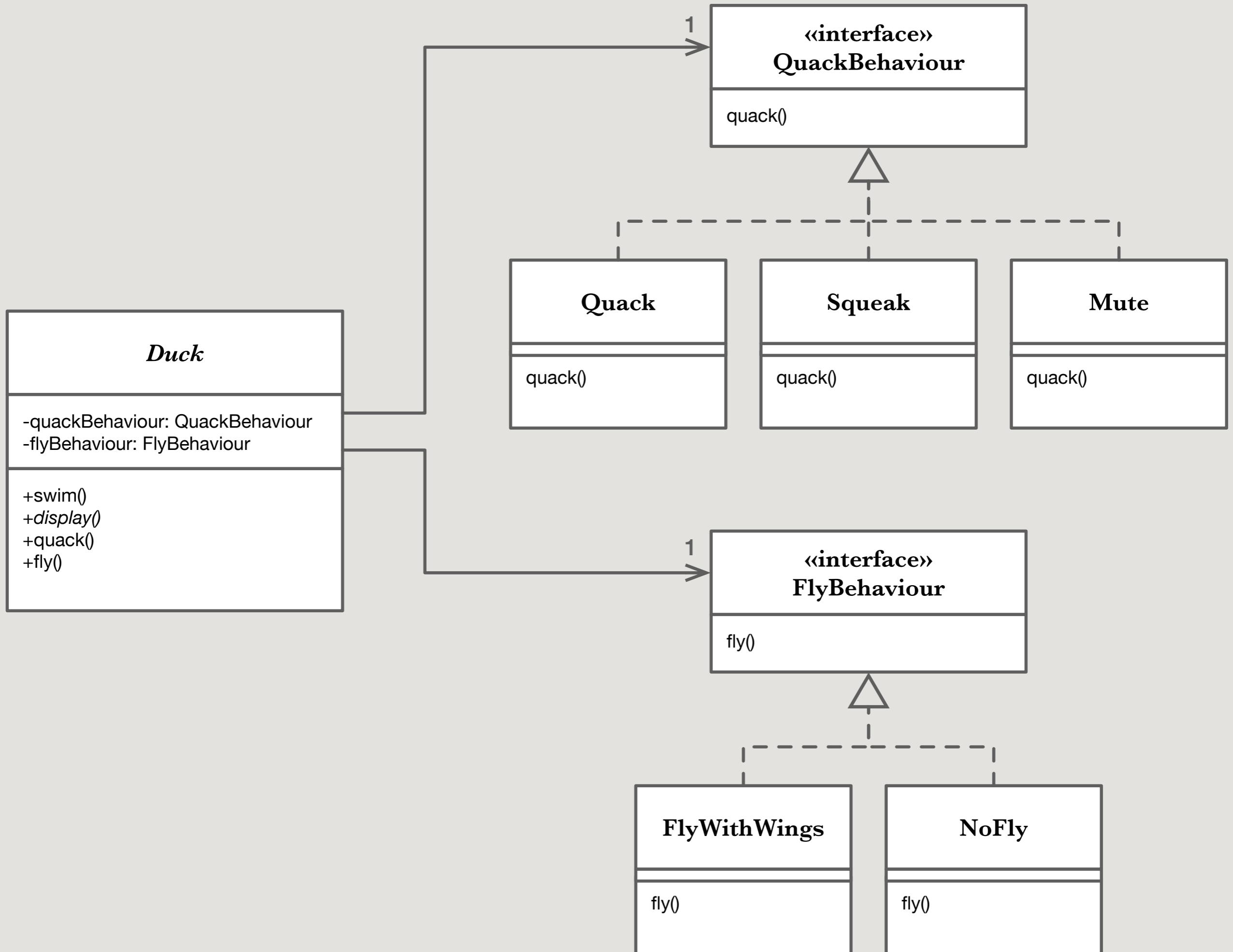
*Identificar aquellos aspectos
que varían y separarlos de
lo que tiende a permanecer
igual.*

O sea... encapsular el concepto que varía.



*sacamos
fuera lo
que varía*





¿Qué hemos hecho?

Simplemente, aplicar tres principios básicos del diseño orientado a objetos.

Encapsular el concepto que varía.

En este caso, las formas de volar y graznar, sacándolas fuera de la clase pato.

*Favorecer la composición
frente a la herencia.*

*Programar para una
interfaz, no para una
implementación.*

*La clase Pato sólo trata con FormaDeVolar y
FormaDeGraznar, pero no sabe quiénes son.*

Patrón Strategy

**Patrón de comportamiento,
de objetos.**

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.

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.

Microsoft Word, Apple Pages, Amazon Kindle, navegadores web...

Una estrategia línea a línea, simple pero horrible desde el punto de vista tipográfico, que hace inviable la justificación.

Microsoft Word, Apple Pages, Amazon Kindle, navegadores web...

El algoritmo de TeX. Lo hace con cada párrafo: minimiza los «ríos» en el texto.

Una estrategia línea a línea, simple pero horrible desde el punto de vista tipográfico, que hace inviable la justificación.

Microsoft Word, Apple Pages, Amazon Kindle, navegadores web...

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.

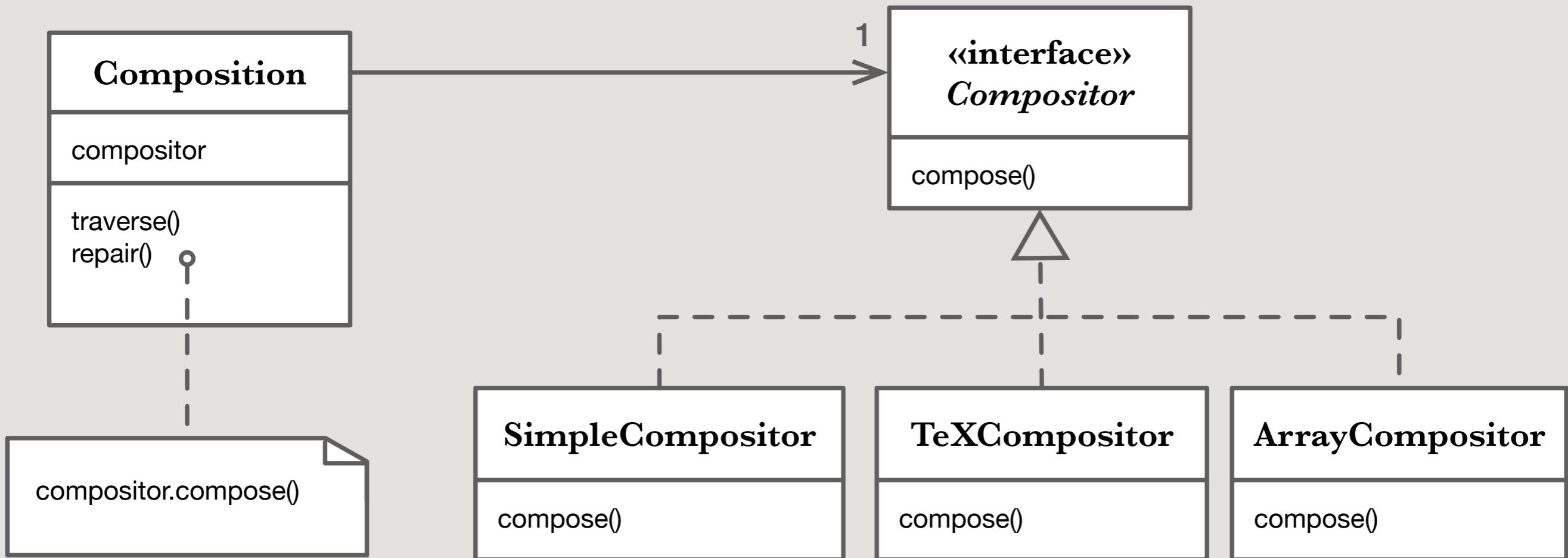
Una estrategia línea a línea, simple pero horrible desde el punto de vista tipográfico, que hace inviable la justificación.

Microsoft Word, Apple Pages, Amazon Kindle, navegadores web...

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.



Aplicabilidad

Úsese el patrón Strategy cuando

**Muchas clases relacionadas
solo se diferencian en su
comportamiento.**

Las estrategias permiten configurar una sola clase con un comportamiento determinado de entre varios.

**Necesitamos distintas versiones
de un algoritmo.**

**Un algoritmo usa datos que los
-clientes no deberían conocer.**

el contexto

El patrón permite evitar exponer complejas estructuras de datos específicas de cada algoritmo concreto.

**Los distintos comportamientos
de una clase aparecen como
múltiples sentencias
condicionales.**

El patrón Strategy permite mover cada rama
condicional a su propia clase.

Aunque no se mencionan explícitamente en el libro, otras causas de aplicabilidad del patrón son:

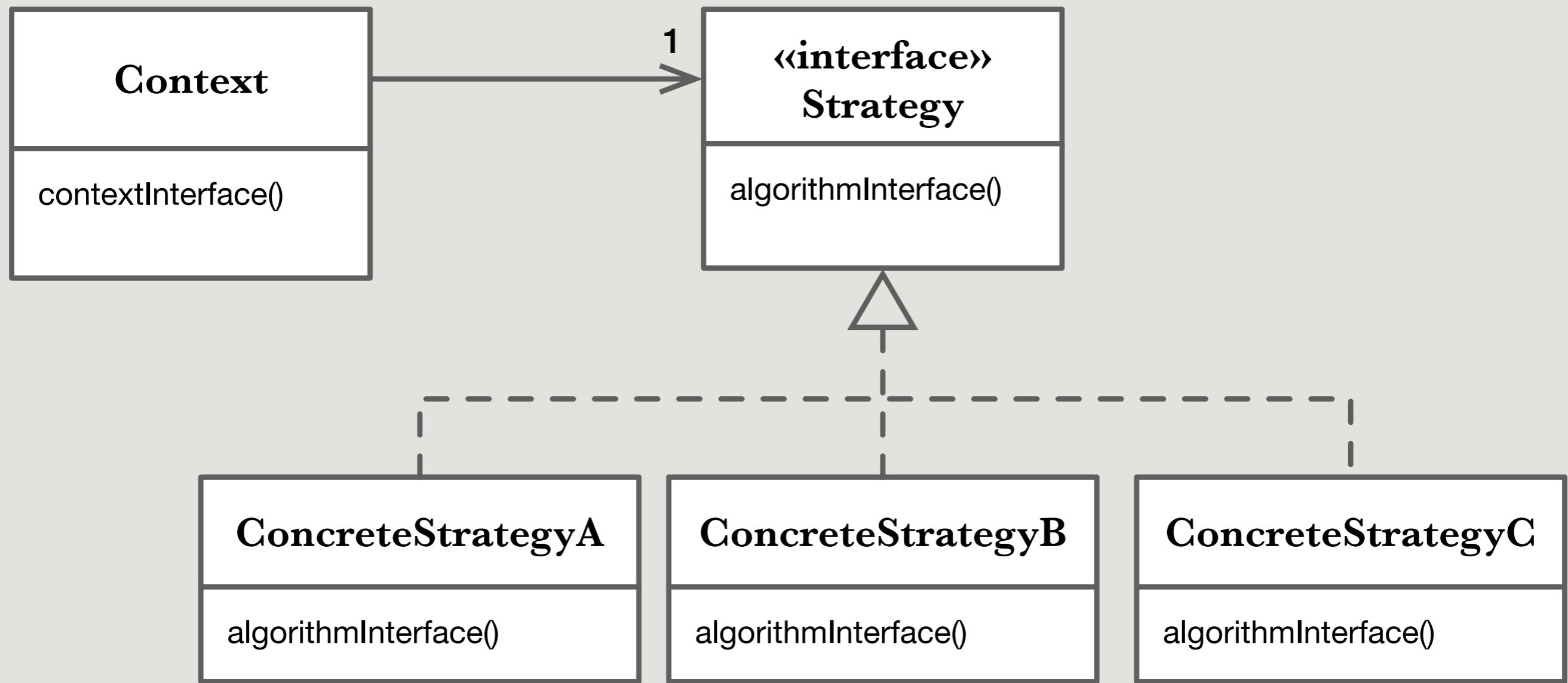
Una clase tiene dos o más aspectos que varían.

Daría lugar a una explosión combinatoria si lo intentásemos modelar mediante la herencia.

Es lo que se conoce como **explosión de subclases**.

**Queremos poder cambiar el
comportamiento de una operación
en tiempo de ejecución.**

Estructura



Participantes

Strategy

(Compositor)

Declara una interfaz común para todos los algoritmos.

El contexto usa dicha interfaz para llamar al algoritmo definido por una estrategia concreta.

ConcreteStrategy

(SimpleCompositor, TeXCompositor, ArrayCompositor)

Implementa los distintos algoritmos por medio de la interfaz Strategy.

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 aquella llame a los métodos que necesite.

El contexto reenvía las peticiones de los clientes a su estrategia.

Los clientes normalmente crearán y pasarán una estrategia concreta al contexto.

A partir de ese momento los clientes interactúan únicamente con el contexto.

Normalmente hay una familia de estrategias concretas que el cliente puede escoger.

Consecuencias

(Ventajas e inconvenientes)

Familias de algoritmos relacionados

La jerarquía de estrategias define una familia de algoritmos o comportamientos para que los contextos los reutilicen.

Alternativa a la herencia

La herencia clásica también permite implementar distintos algoritmos o comportamientos.

Podríamos, en efecto, heredar del contexto directamente para implementar los distintos comportamientos. Pero, al hacerlo, los estamos ligando indefectiblemente a la clase, lo que da lugar a consecuencias normalmente no deseables.

Por el contrario, al aplicar el patrón obtenemos los siguientes beneficios.

Alternativa a la herencia

Hace que el contexto sea más fácil de entender, modificar y mantener.

Con la solución basada en la herencia, ambas responsabilidades siguen estando en la misma clase.

El patrón permite que tanto el contexto como las estrategias se centren en una única tarea, cumpliendo así el Principio de Responsabilidad Única, lo que los hace más cohesivos y, por tanto, más fáciles de entender y modificar.

Alternativa a la herencia

Evita la duplicación de código.

Cuando hay más de un aspecto que varía y el lenguaje no admite herencia múltiple.

Alternativa a la herencia

Evita la explosión de subclases.

Cuando una clase puede variar por más de un motivo.

Alternativa a la herencia

Se puede cambiar dinámicamente.

El patrón permite configurar el contexto con una estrategia u otra, e incluso cambiar de estrategia en tiempo de ejecución.

Elimina las múltiples sentencias condicionales

Cuando se combinan diferentes comportamientos en una única clase, resulta difícil evitar las sentencias condicionales para seleccionar el comportamiento adecuado.

Al encapsular cada uno en una estrategia separada se elimina la necesidad de dichas sentencias condicionales.



Composition.cpp

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



Composition.cpp

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

Elección de implementaciones

Las estrategias pueden proporcionar diferentes implementaciones del mismo comportamiento.

El cliente puede elegir entre varias implementaciones. ¡Incluso en tiempo de ejecución!

Los clientes deben conocer las distintas estrategias

Por lo tanto, deberíamos utilizar el patrón solo cuando dicha variación en el comportamiento sea relevante para los clientes.

Esto puede ser un inconveniente potencial del patrón si no se elige correctamente.

Se complica la comunicación entre el contexto y las estrategias

Diferentes estrategias pueden requerir diferentes datos del contexto.

Esto significa que puede haber ocasiones en las que el contexto cree e inicialice parámetros que nunca se utilicen. Si esto es un problema, entonces será necesario un acoplamiento más estrecho entre la estrategia y el contexto.

Crece el número de objetos

Al tener cada estrategia como clases separadas.

A veces se puede reducir esta sobrecarga implementando estrategias como objetos sin estado que los contextos pueden compartir. Cualquier estado residual es mantenido por el contexto, que lo pasa en cada llamada a la estrategia.

Sin embargo, esto rara vez es un problema.

Una consecuencia adicional (y crucial) del patrón, que el libro no indica explícitamente, es que

Se pueden añadir nuevas estrategias

Sin que el contexto se entere.

Cumpliendo así el Principio de Abierto-Cerrado.

Implementación

(Cuestiones a tener en cuenta)

Definir las interfaces de la estrategia y el contexto

Deben permitir a cualquier estrategia concreta acceder a cualquier dato que necesite del contexto, y viceversa.

Una opción es que el contexto pase los datos como parámetros a las operaciones de la estrategia.

De este modo la estrategia y el contexto se mantienen desacoplados.

A cambio, el contexto puede que le pase datos a la estrategia que no necesita.

Y una tercera consecuencia de este enfoque que el libro no menciona es que debemos asegurarnos de que los datos transmitidos son suficientes no solo para las estrategias actuales, sino para las nuevas que puedan surgir en el futuro.

Otra técnica es que el contexto se pase a sí mismo como parámetro.

Y que la estrategia le pida los datos al contexto.

La estrategia puede almacenar una referencia a su contexto cuando se crea, eliminando la necesidad de pasar nada en absoluto.

En cualquier caso, la estrategia puede pedir exactamente lo que necesita. Pero ahora el contexto debe definir una interfaz más elaborada para sus datos, con mayor acoplamiento entre ambos.

Y tal vez comprometiendo la ocultación de la información de la clase.

Hacer las estrategias opcionales

Según el libro, el contexto puede simplificarse si no es obligatorio tener una estrategia.

Esto es lo que se menciona en el libro:

El contexto comprueba si tiene un objeto estrategia, antes de acceder a él.

Si lo hay, lo usa normalmente. En caso contrario, lleva a cabo su comportamiento predefinido.

La ventaja es que los clientes no tienen que tratar con las estrategias a no ser que no les sirva dicho comportamiento predeterminado.

Sin embargo, en estos casos suele ser posible (y mejor) contar con una estrategia predeterminada.

Que puede ser creada por el propio contexto.



Composition.java

```
class Composition {  
  
    private Compositor compositor;  
  
    public Composition() {  
        this(new SimpleCompositor());  
    }  
  
    public Composition(Compositor compositor) {  
        this.compositor = compositor;  
    }  
    // ...  
}
```

Usos conocidos

Patrones relacionados

Decorator

Aunque tienen propósitos diferentes (cambiar la funcionalidad existente en vez de añadir nueva) y los consiguen de formas diferentes (mediante composición en un caso, envolviendo un objeto en otro, en el otro), en algunos casos muy concretos podrían incluso considerarse alternativas.