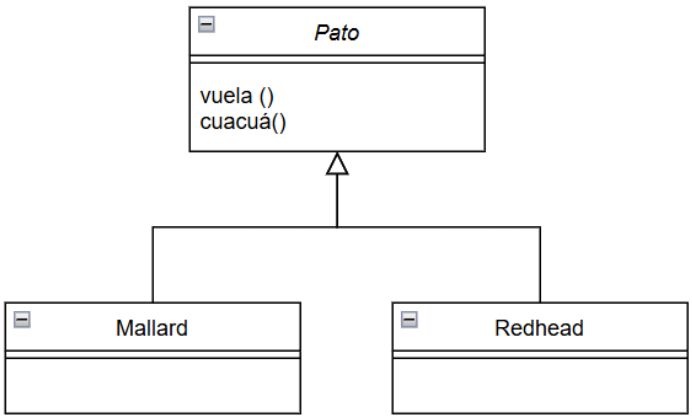
Todo empezó con un pato

Original: *Head First Design Patterns*, editorial **O'Reilly**

El asunto es que en tu empresa van a trabajar con **Patos**. Y el jefe quiere que prepares tu programa para trabajar con Patos. Todos los Patos tienen que poder volar y hacer cuacuá. Y vamos a tener por ahora dos patos diferentes: el Mallard y el Redhead.

Supongo que ya habrás pensado la solución al problema. Al fin y al cabo, es el típico ejercicio de Programación de 1º de la 3ºra evaluación.



Sencillo, ¿verdad? Ponemos los métodos **vuela ()** y **cuacuá ()** en Pato, que es la superclase. Y tenemos dos clases que extienden de Pato: Mallard y Redhead.

Pues no. Esto es más complicado de lo que parece.

En **Programación Orientada a Objetos** hay dos cosas fundamentales: el dato y el comportamiento. Quizás te suene si te lo pongo así explicado:

- Dato: Digamos que describe cómo es esa clase. Son los atributos de la clase. Por ejemplo, nombre del pato.

- Comportamiento: Es lo que hace la clase. Normalmente, los métodos.

Al hacer un programa de acuerdo al Diagrama de Clases de arriba, lo que estamos diciendo es que TODOS los patos tienen el mismo comportamiento. Da igual que sea un Mallard o un Redhead. Esto es porque ambas clases **extienden** de Pato y, por tanto, heredan sus atributos y sus métodos.

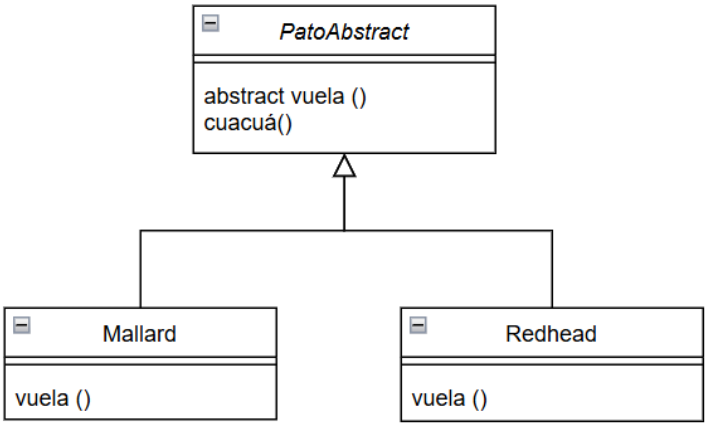
O sea, lo que nos interesa, su comportamiento.

O sea, que todos los patos vuelan igual y dicen cuacuá de la misma forma.

Problema 1

Ahora viene tu jefe y te dice que no, que los Mallard y los Redhead vuelan de forma diferente. Por tanto, no te vale la solución que has propuesto.

De nuevo, la solución es fácil. Como te sabes el mecanismo de herencia a la ferpección, sabes que puedes **redefinir el comportamiento** de un objeto simplemente sobrescribiendo el método que haga falta (@override).



Hago que la clase Pato sea abstracta. Así, me aseguro de que nadie me haga un new Pato (), que no tiene sentido en mi programa. Luego, defino un método abstracto llamado **vuela ()**. Así obligo a Mallard y a Redhead a implementarlo. No hago nada con **cuacuá ()**, dado que todos los patos hacen cuacuá igual, lo dejamos en PatoAbstract y que lo hereden de ahí.

De esta forma, conseguimos que Mallard y Redhead tengan cada uno de ellos un comportamiento propio diferente. Hasta aquí, nada nuevo.

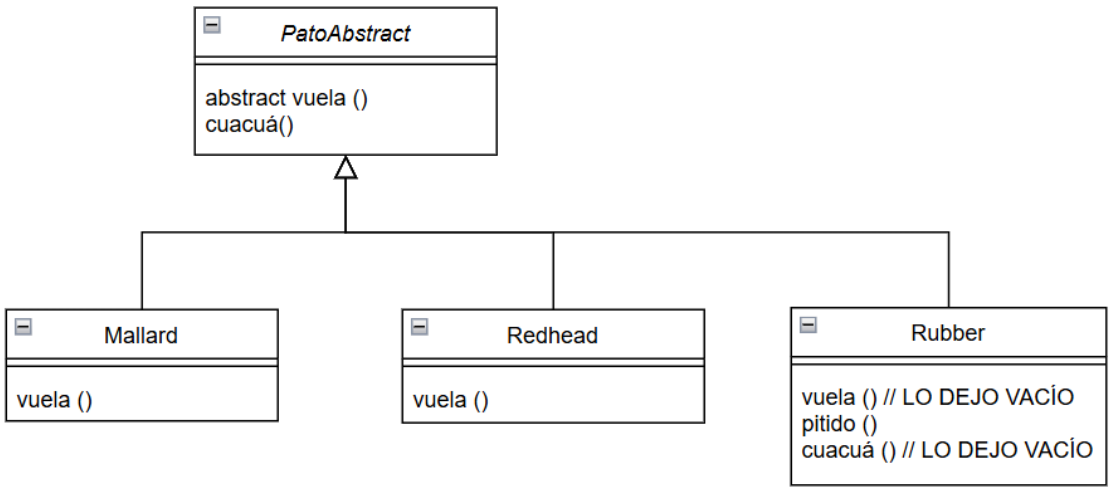
Problema 2

Tu jefe tiene ahora una idea brillante: Patos de Goma. Le preguntas que de dónde se ha sacado eso, pero insiste en que es algo muy importante que el programa necesita. Que lo añadas.

Claro que… un Pato de Goma no vuela… ni hace cuacuá… hace un ruido raro cuando lo estrujas. Un pitido o algo.

Pero vamos, que da igual, que para algo tienes la herencia y sabes de qué va eso del comportamiento, seguro que se te ocurre algo guay.

Como, por ejemplo, lo siguiente…



Mira, hacemos que Rubber extienda de PatoAbstract. Esto me obliga a implementar **volar ()**, pero como los patos de goma no vuelan, dejo el método en blanco. En cuanto a **cuacuá ()**, pues lo heredo también desde PatoAbstract, y como no voy a usar ese código (un pato de goma no hace cuacuá) pues pongo el método para redefinirlo y lo dejo en blanco. Por último, añado el método **pitido ()**, que es un comportamiento exclusivo del pato de goma.

Fácil. Tirado.

… vale, no queda muy limpio tener ahí dos métodos en blanco en Rubber, pero qué se le va a hacer. Sólo es un Pato de tres. Y si pasa algo raro, sólo hay que leerse el código de la clase para entenderlo. Porque usas comentarios y javadoc, ¿verdad? Pues nada oyes, solucionado.

Problema 3

Tu jefe viene y te dice que hay que añadir al programa el Pato de Madera, catorce tipos de patos nuevos, y tres variedades raras de pato que pueden bucear. Claro que, hay que tener en cuenta que no todos nadan, no todos vuelan, y no todos hacen cuacuá. Y que ya de paso aproveches de paso para arreglar esa movida de los métodos en blanco, porque ya no es cosa del Pato de Goma, también del Pato de Madera.

Ya que sabes tanto de herencia, habrás visto la solución: **Interfaces**.

Vamos a hacer un Interfaz vueloInterface, otro cuacuácInterface, y otro nadarInterface. Cada interfaz tiene un método (por ejemplo, volar ()) de forma que, si tienes un Pato que vuela y nada, pero no hace cuacuá, pues ese pato implementa las dos interfaces. Todo por supuesto manteniendo que cada Pato hereda de PatoAbstract. Y retocando un poco las cosas, vamos.

Nada que no se arregle con café y una horita de curro.

Mira en la página siguiente cómo queda el esquema, anda…

Yo lo veo todo claro. Es evidente. Elegante incluso. Lo mejor es que así evitas la recursividad de métodos iterativos y la disyunción de clases abstractas cuando extienden de interfaces no euclidianos en el horizonte de sucesos de un agujero negro los jueves por la noche de los meses impares.

El verdadero problema

Vale, ahora ya sabes que la **herencia**, la **abstracción**, los **interfaces** y el **polimorfismo** que aprendiste en 1º no son la solución a todos los problemas. Son herramientas. Si tienes que clavar un clavo, usas un martillo. Pues esto es lo mismo. Tienes que entender para qué sirve cada uno de ellos y usarlo cuando toque.

Respecto a los patos del Problema 3, es importante que entiendas que, aunque se te haya ocurrido una solución brillante para ese caso en concreto, estás cometiendo errores a largo plazo. Los más evidentes son:

- Difícil mantenimiento: Tienes semejante jaleo de clases, interfaces, etc. que dentro de seis meses cuando te cambien algo, pasarás mucho tiempo toqueteando código. Y no sólo eso, sino que un simple cambio te hará tener que cambiar código en otro lado del programa. Esto genera una cascada de cambios que es difícil de gestionar sin cometer errores.

- Difícil lectura: Aunque tengas el código ahí, para entender el comportamiento del Redhead tendrás que abrir esa clase más tres interfaces diferentes y PatoAbstract. Ahora, imagínate que es un programa de verdad, no uno de patos. ¿Cuánto tiempo vas a dedicar solamente a entender el código?

- Añade una Oca, que es una subclase de Pato que tiene diferente comportamiento. Venga listo, inténtalo.

El asunto es que has intentado **reinventar la rueda**. Hay algo así como miles de programadores que han tenido el mismo problema que tú. Con el tiempo, esta gente ha llegado a refinar el problema hasta el punto en el que existen unas soluciones estándar que pueden arreglar el asunto de forma eficiente. Y la solución son **los patrones de diseño**.

La definición de libro de lo que es un **patrón de diseño** dice que es una solución reutilizable a un problema común en el diseño de software. Se trata de una descripción o plantilla que se puede adaptar y aplicar en diferentes situaciones para resolver problemas recurrentes. Los patrones no son código en sí mismos, sino más bien una forma de cómo resolver el problema.

Lo que tenemos que aprender es a identificar el problema, encontrar el patrón adecuado, **evaluar si es nos es útil o no**, y si eso lo aplicamos. Porque no siempre los patrones son la respuesta.

La respuesta a los patos (con patrón, claro)

Bien. Cuando **diseñas** una aplicación hay que tener en cuenta una serie de principios generales universales. El que nos interesa ahora es este:

"Identifica lo que cambia de tu aplicación y sepáralo de lo que permanece siempre igual"

Lo que viene a significar, que hay que **encapsular** las partes que cambian de tu código, de forma que, si luego hay que modificarlas, no afecten al resto del código. Esto de encapsular quiere decir que ocultes cosas al resto de las clases, de forma que sólo puedan acceder a lo que tú quieras y no sepan exactamente cómo están implementadas.

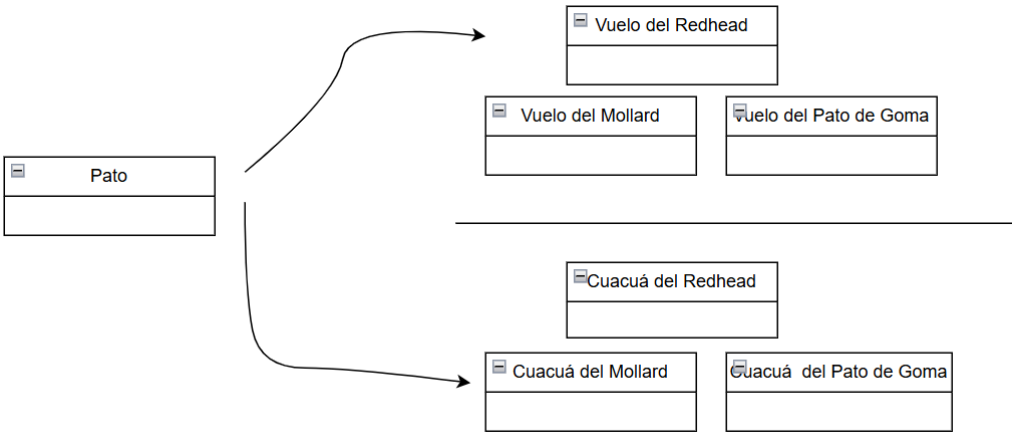
El ejemplo más sencillo posible de encapsulamiento es cuando tú tienes una clase User con los atributos login y pass. No vas a dejar que accedan a los atributos directamente, así que los pones private. Ahora, sólo pueden acceder a login y pass a través de los getters y setters.

Vamos a ver cómo llevamos esto a los dichosos patos.

- Cosas que cambian: **volar ()** y **cuacuá ()**.

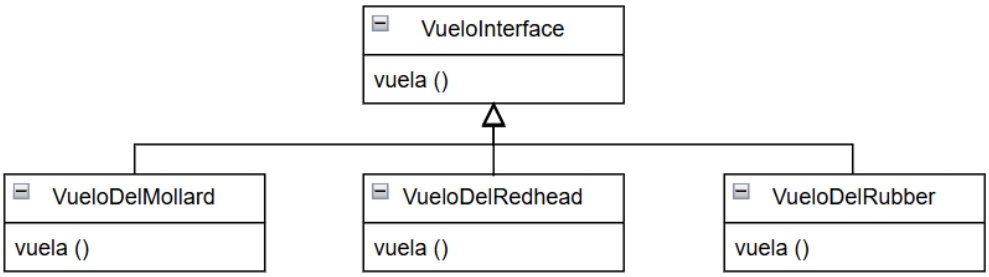
- Cosas que no cambian: otros métodos, que no están mencionados, pero suponemos que existen. Me invento uno para entender mejor la cosa, por ejemplo, **mostrar ()** que nos enseña una foto del pato.

Bien pues, lo que vamos a hacer ahora es **convertir en una serie de clases los comportamientos que cambian**. Tendremos por un lado a la clase Pato y por otro los diferentes vuelos y los diferentes cuacuás.

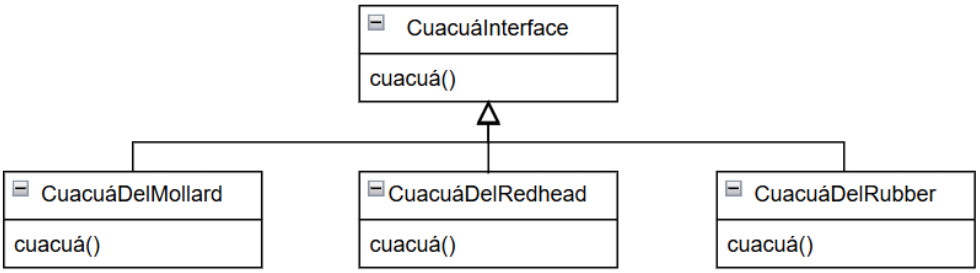


¿Ves cómo lo tenemos separadito? Vale, ahora organicemos y completemos el código. Por cada cosa que cambia, definiremos una **Interfaz** propia para cada comportamiento que cambia. Donde 'Interfaz' en realidad puede ser una **Interfaz** o una **clase Abstracta**. Valen las dos. En muchas ocasiones se dice 'Interfaz' y pueden estar hablando de la una, la otra, o cualquiera de las dos porque da igual. Cosas de programadores.

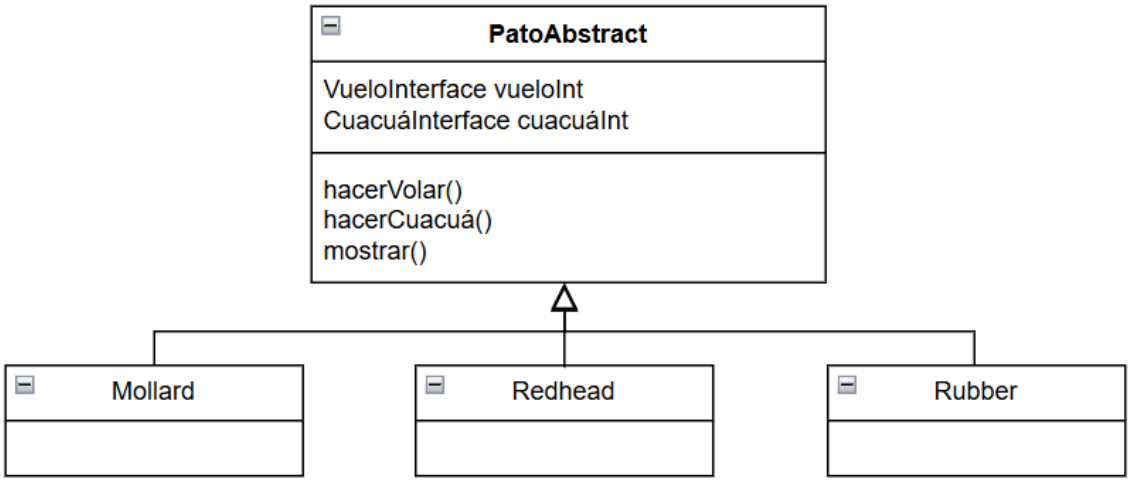
La cosa queda así para el Vuelo:



Y hacemos lo mismo con el cuacuá:



Finalmente, sólo nos queda hacer la clase Pato y sus hijos:



Y ya estarían todos los problemas solucionados. Te lo explico. Ahora, cada vez que tengamos que, a añadir un pato nuevo al programa, tendremos que:

1) Extender PatoAbstract con una nueva clase para el nuevo Pato.

2) Extender VueloInterface con una nueva clase para el vuelo del nuevo Pato. E implementar vuelo ().

3) Extender CuacuácInterface con una nueva clase para el cuacuá del nuevo Pato. E implementar cuacuá ().

Ya sólo nos queda integrarlo todo. Para ello, veamos cómo sería el código de PatoAbstract:

```
public abstract class PatoAbstract {

    VueloInterface  vueloint;
    CuacuácInterface  cuacuaInt;

    public void hacerVolar () {
        vueloInt.volar();
    }

    public void hacerCuacuá () {
        cuacuaInt.cuacuá();
    }
}
```

De esta forma, al llamar a hacerVolar () se estaría llamando al método volar () correspondiente a ese Pato en concreto, por ejemplo, al Mollard. Pero, ¿cómo sería el código de ese Mollard?

Pues así de fácil y sencillo:

```
public class Mollard extends PatoAbstract {

    public Mollard (){
        vueloint = new VueloDelMollard ();
        cuacuaInt = new CuacuáDelMollard ();
    }

    public void mostrar (){
        System.out.prinln ("Foto del Mollard");
    }
}
```

Cada vez que queramos hacer un **new Mollard ()**, se están cargando a la vez los comportamientos del vuelo del Mollard y del cuacuá del Mollard. Nosotros en tiempo de ejecución NO sabemos desde fuera qué se está cargando, ni nos interesa, ni podemos acceder a ello: está **encapsulado**. Pero cada vez que hacemos **mollard.hacerVolar ()**, el dichoso Mollard está volando como se supone que tienen que volar los Mollard.

Por qué esto es buena idea

Cuando tienes varias clases juntas, como ocurre con Mollard y su vueloInt y cuacuáInt, lo que tienes es una **composición**. En lugar de *heredar* un comportamiento, suele ser mejor idea *componer* al Mollard con sus correspondientes vueloInt y cuacuáInt.

El principio dice así:

"Es mejor la composición que la herencia"

Las ventajas son varias. ¿Te has fijado que, si le ponemos un getter & setter a vueloInt y cuacuáInt, podemos cambiar el comportamiento del Mollard cuando nos de la gana? O sea, podemos hacer esto:

```
Mollard mollard = new Mollard();

// El mollard vuela como un Mollard

VueloDelRubber vuelo = new VueloDelRubber();
mollard.setVueloint(vuelo);

// Ahora el mollard vuela como un pato de goma
```

Vale. Pues esta facilidad para añadir patos nuevos, mantener el código, encontrar errores, añadir comportamientos nuevos, y cambiarlos incluso en tiempo de ejecución; son algunas de las ventajas del **Strategy Pattern**.

El Strategy Pattern, como idea, permite que tu código sea flexible a cualquier clase de burrada que se le ocurra a tu jefe. ¿Cambiar el vuelo del Pato de Madera? ¿Hacer que siete patos se comporten como el Redhead? ¿Añadir un comportamiento completamente nuevo?

Pues el mayor tiempo y esfuerzo que harás no va a ser intentar entender esta movida, sino ir directo al cacho de código que tienes que cambiar.

¿A que mola?

Esto que acabas de aprender es un patrón. Hay doce principales y un montón más secundarios. Sí, es una cosa rara. No, la gente no se los sabe de memoria, saben de qué van y para qué se utilizan, y lo consulta. Tú quédate con la idea de lo que es un patrón. Por el momento, es suficiente.

Ejercicio Propuesto

Vamos a programar en serio el ejemplo de los patos utilizando el Strategy Pattern. Crea un programa para tres patos, tres comportamientos de vuelo y tres comportamientos de nadar para cada pato. Los métodos de volar () y nadar () de cada pato solamente tienen que escribir “Estoy volando como un Mollard” o algo parecido para que sepas en todo momento qué comportamiento tiene cada pato.

Añade una Lista en la clase principal que guarde PatoAbstract. A continuación, implementa el siguiente menú:

- Añadir pato a la lista (Mollard, Redhead o Rubber)
- Mostrar vuelos de patos
- Mostrar nadar de patos
- Cambiar vuelo de Mollards, Redheads, o Rubbers por otro diferente
- Cambiar nadar de Mollards, Redheads, o Rubbers por otro diferente

Si cambias el vuelo de los Mollards por el de los Rubbers (por ejemplo) y los muestras por consola, deberías ver el mensaje “Estoy volando como un Rubber” cada vez que se muestre un Mollard.

Finalmente, cuando lo anterior funcione, añade un Wood (Pato de Madera) al programa. Si lo has hecho bien, los cambios en tu código deberían de ser mínimos.