DigitalHouse>
Coding School

Certified Tech Developer
The Ultimate Degree

Programación Orientada a Objetos

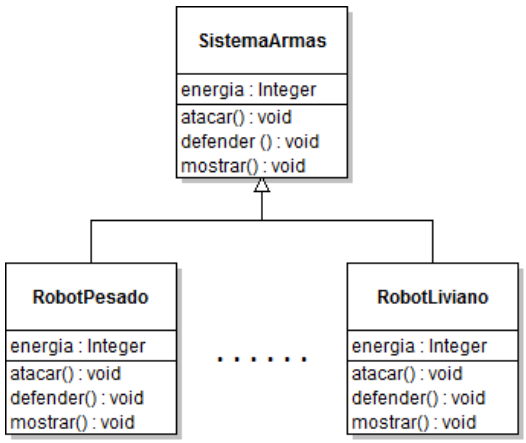
Interfaces

Objetivo

Identificar y modelar las clases e interfaces involucradas en el enunciado especificando sus atributos, responsabilidades y relaciones entre las mismas.

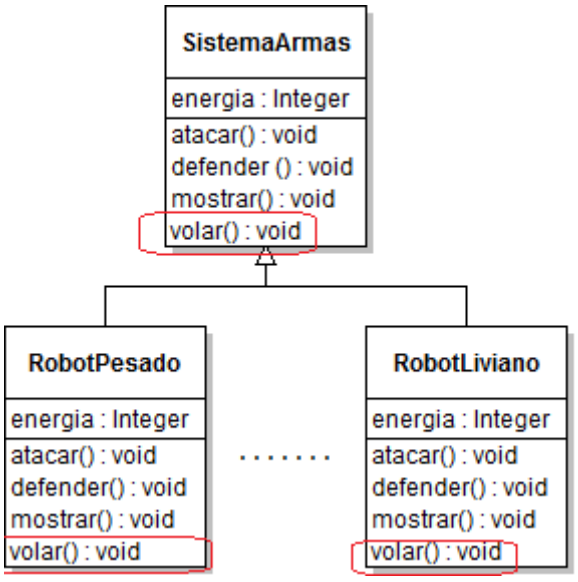
Presentación del caso "Batalla del futuro"

Supongamos que hay que modelar un juego mobile de guerra/estrategia en tiempo real: "Batalla del futuro". Entre las clases del juego, tendremos los diferentes tipos y sus características. El primer paso, es definir un robot con sus operaciones básicas:



Lo primero que podemos hacer es crear una clase SistemaArmas, con las operaciones comunes: atacar, defender y mostrarse (en pantalla, con sus datos). **Supongamos que una próxima actualización del juego introducirá sistemas de armas voladores**. Sería importante que se mantengan las "actualizaciones" de la *app* al mínimo y que se detenga lo menos posible el sistema. Entonces, ¿qué podría hacerse con el diseño actual?

Normalmente se incluiría la operación volar() en la clase SistemaArmas. Pero ¿qué pasa si entre los sistemas de armas que quiero incluir hay un tanque? ¡Los tanques no vuelan! Veamos un ejemplo:

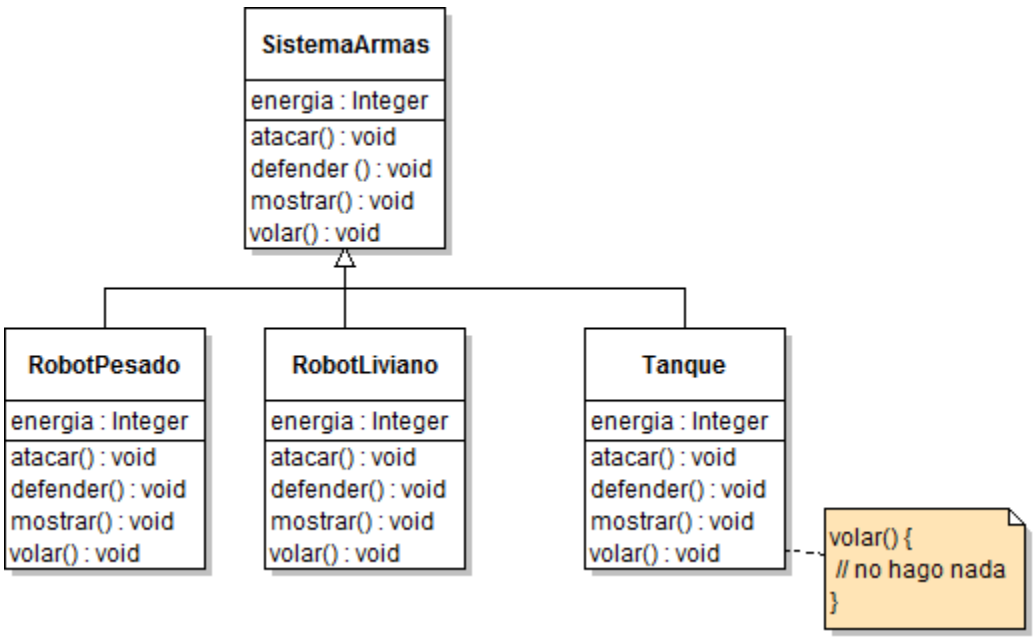


A simple vista, un sistema de armas, puede ser un robot, un tanque, un bombardero… entonces, agregando el método volar() rompe el diseño, porque de existir una clase hija Tanque, tendríamos un tanque volador. Entonces, ¡No todos los vehículos deberían volar! No es un buen diseño. Ni siquiera el diseño original era bueno, dado que un cambio local a una clase generó un efecto colateral generalizado.

Entonces, en lo que a *mantenimiento* respecta, **la herencia no siempre es la mejor opción**.

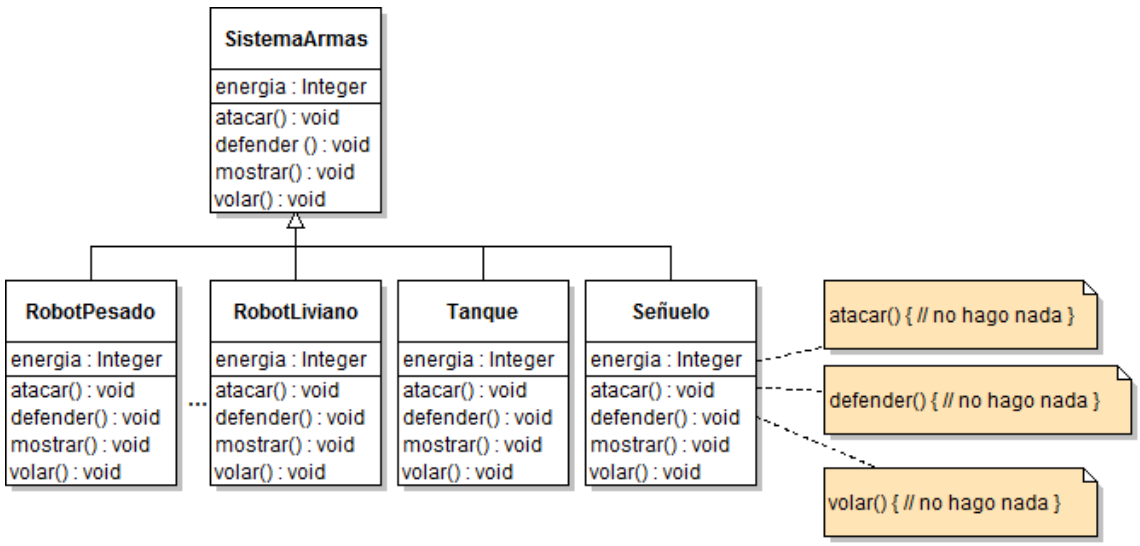
Planteo de soluciones

Qué tal si sobrescribimos la operación volar() y la dejamos "vacía" (sin implementación) en todas aquellas clases que "no deban volar" (tanque, submarino, etc.).



Si sobrescribimos el volar para dejarle una implementación vacía, ¿qué pasa si agrego un nuevo tipo de vehículo, por ejemplo, un portaaviones? Un barco no debería volar: otra vez, tengo que dejar una implementación vacía.

Para empeorar las cosas, ¿qué pasa si agrego un VehículoSeñuelo? Ese tampoco debería "volar" y no queremos seguir agregando implementaciones vacías. Atención con este caso en particular, porque tampoco debería atacar, ni defenderse. De ser así, no se puede empezar a dejar implementaciones vacías por todos lados —ya vimos que esta no es la solución correcta, por varias razones—. Además, estaríamos duplicando el código por todos lados y dejamos un software susceptible a errores.



Podríamos hacer clases abstractas, en las cuales tengamos implantaciones vacías por defecto. Aunque, si necesitamos una clase que tenga diferentes combinaciones de atacar(), defender(), volar(), sumergirse() etcétera, perderíamos mucho en lo que a polimorfismo respecta.

Los problemas de las soluciones planteadas

Con estos aspectos a mano, podemos ver qué **desventajas** tiene la **herencia** para establecer el comportamiento de un sistema de armas.

- Se duplica código en las hijas.
- Un cambio sencillo podría afectar todo el modelo.
- Cambiar el comportamiento de los vehículos en runtime es casi imposible.

¿Qué otra construcción vimos que podría solucionar este problema?

¡Hasta la próxima!