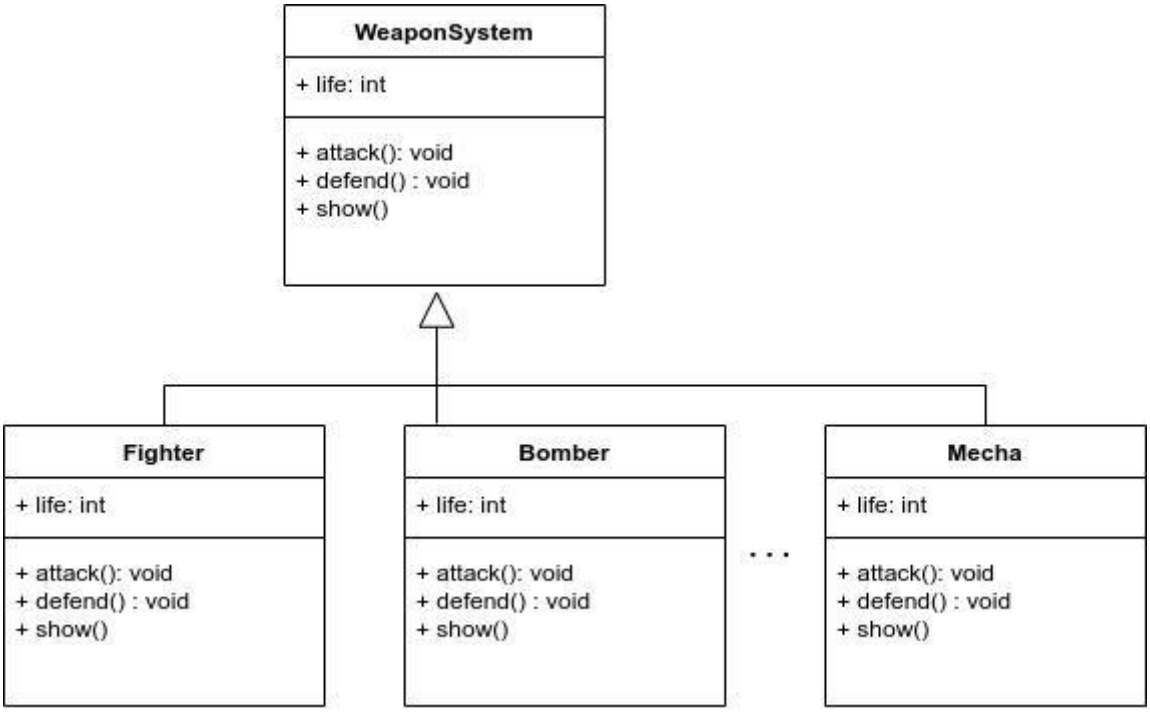
Composición sobre herencia

Usos avanzados de las interfaces

Pensando en un sistema complejo

Supongamos que nos encargan modelar un juego de guerra/estrategia en tiempo real. Entre la multitud de clases que necesitaremos para del juego tendremos diferentes tipos de armas y sus características. Lo primero que podemos hacer es crear una clase SistemaArmas, con las operaciones comunes: atacar, defender y moverse por la pantalla. En el siguiente se muestran solo algunos de las posibles clases:

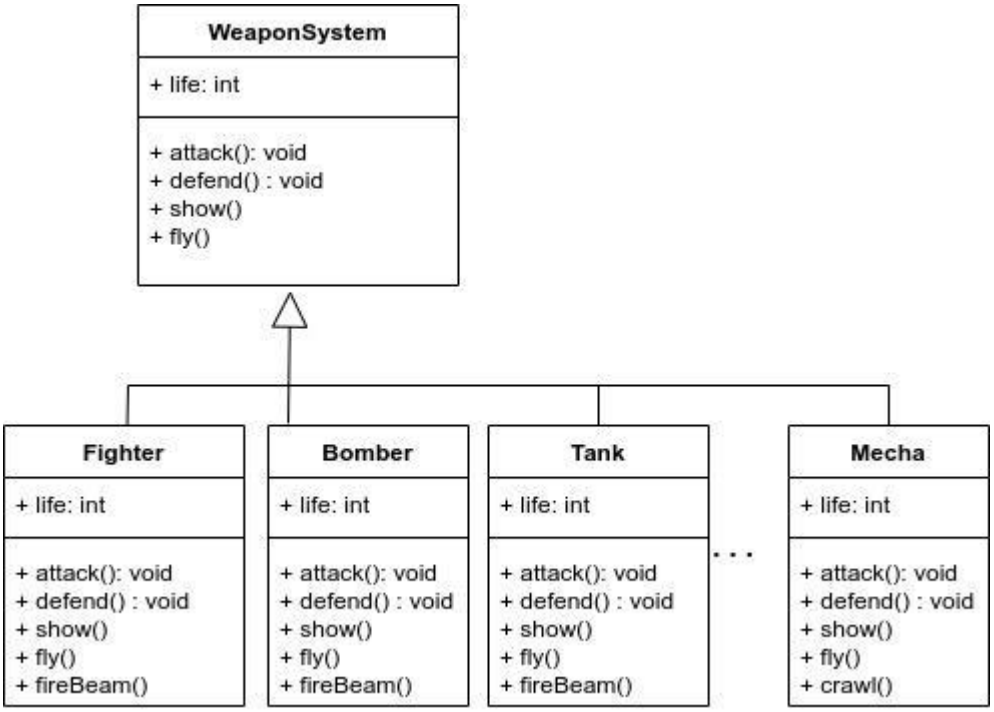


Aquí podemos ver una superclase: WeaponSystem (sistema de armas). y subclases: Fighter (caza), Bomber (bombardero) y Mecha (Robot artillado). A su vez, todos los sistemas de armas pueden atacar una posición (método attack()), pueden defender una posición (método defend()) y deben poder mostrarse en pantalla (show()).

Supongamos que se planea una próxima actualización del juego introducirá nuevos sistemas de armas, helicópteros, aviones, hidrodeslizadores (que un poco se mueven, un poco vuelan… como a 10 centímetros del suelo), también sistemas de armas que pueden realizar ataques con rayos de energía y señuelos que pueden proyectarse en el radar

enemigo. Asimismo, se tiene planeado comenzar con un esquema de micropagos para obtener diferencias en las armas que tengamos a disposición. Teniendo esto en cuenta, es importante que se mantengan las "actualizaciones" de la aplicación al mínimo y que se detenga lo menos posible el sistema. Con este esquema que tenemos, cada modificación requerirá modificar el código, compilar todo, parar el sistema, instalar la actualización y lanzar todo de nuevo. ¡Nuestros clientes no estarán contentos! Entonces, ¿qué podría hacerse con el diseño actual?

Lo primero que podríamos hacer es incluir las nuevas operaciones posibles como el volar (método fly()) y cualquier otra operación posibl en la clase WeaponSystem. De esta forma, podremos soportar cualquier subclase nueva que agreguemos y usarla de manera polimórfica. Podemos pensar en un robot (Mecha) que vuele, ¿pero qué pasa con un tanque?, ¿qué pasaría si nos piden incluir buques? Veamos un ejemplo:



Un sistema de armas puede ser un robot, un bombardero, pero también puede ser un tanque. Entonces, agregar la operación de volar rompe el diseño, porque si agregamos una subclase Tank (Tanque) tendríamos un tanque volador. Sin embargo, es evidente que no todos los vehículos deberían volar. Con la idea de "prepararse para futuras modificaciones" no logramos un buen diseño. Podemos concluir que, en lo que a mantenimiento se refiere, **la herencia no siempre es la mejor opción**.

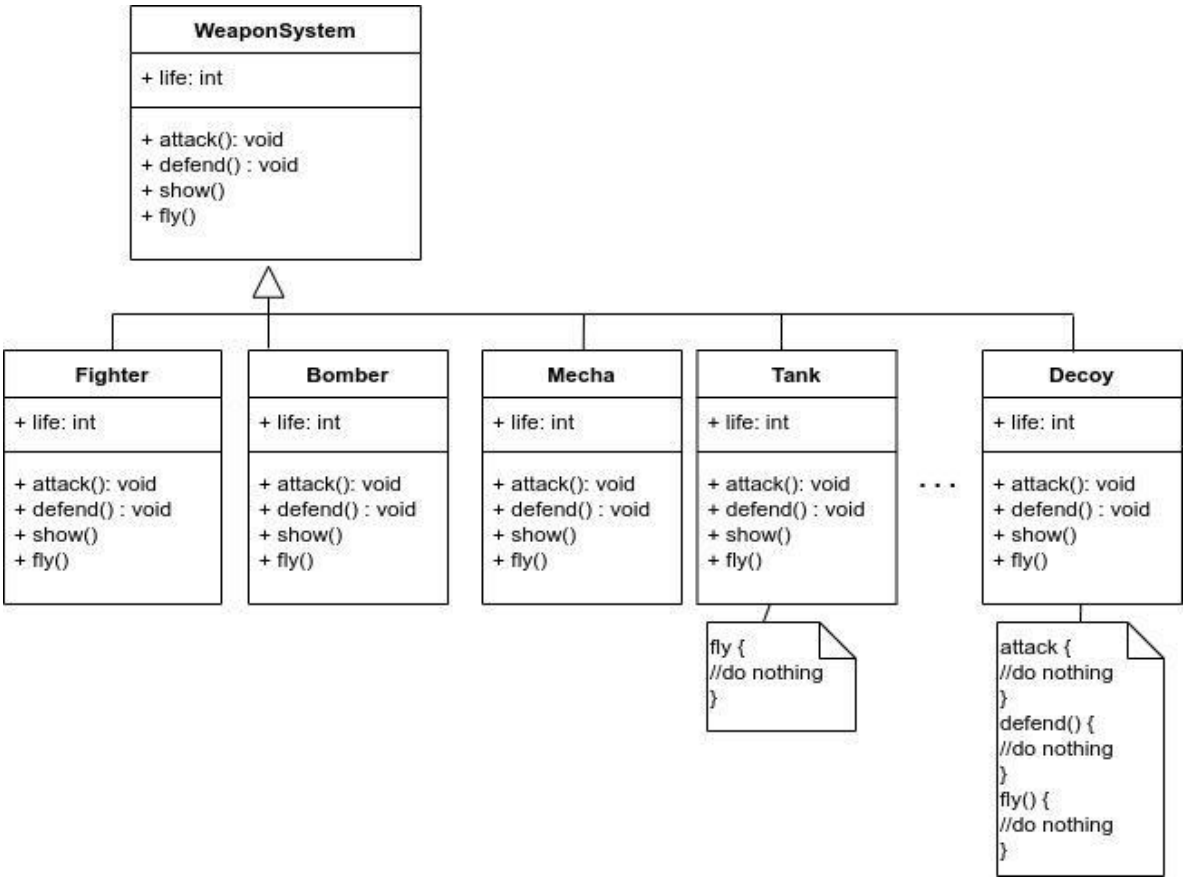
Alternativas

Seamos sinceros, ¿por qué perder la posibilidad de utilizar WeaponSystem de forma polimórfica por una tontería como que un tanque no pueda volar? ¿Qué tal si sobrescribimos la operación fly() en el tanque y la dejamos "vacía" (sin implementación) y hacemos lo mismo en todas aquellas clases "que no deban volar" (tanque, submarino, etc.)?

```
public class Tank extends WeaponSystem {
  //implement methods
  public void fly() {
    //do nothing
  }
}
```

O podríamos lanzar una excepción no chequeada, por ejemplo, "NoPuedeVolarException", o algo por el estilo. En uno u otro caso, si sobrescribimos el fly() para dejarle una implementación vacía, ¿qué pasaría si agregáramos un nuevo tipo de sistema de armas, por ejemplo, un portaaviones? No debería volar ni arrastrarse: otra vez, tengo que dejar una implementación vacía. Para empeorar las cosas, dijimos que íbamos a contar con señuelos. Estos tampoco deberían volar, ni atacar, ni nada. De pronto, dejar implementaciones vacías empieza a perder su atractivo. En realidad, si una clase va a tener un método que "no hace nada" ni siquiera debería tenerlo.

Empezamos a detectar un problema con las implementaciones vacías. No se puede empezar a dejar implementaciones vacías por todos lados. En esencia, estamos repitiendo código: implementamos múltiples veces el fly() sin que haga nada, dejando un software susceptible a errores. Es decir, nada impide que un programador se equivoque y termine poniendo algo en el bloque de código del método fly() de un submarino. Arribaríamos, entonces, a algo así:

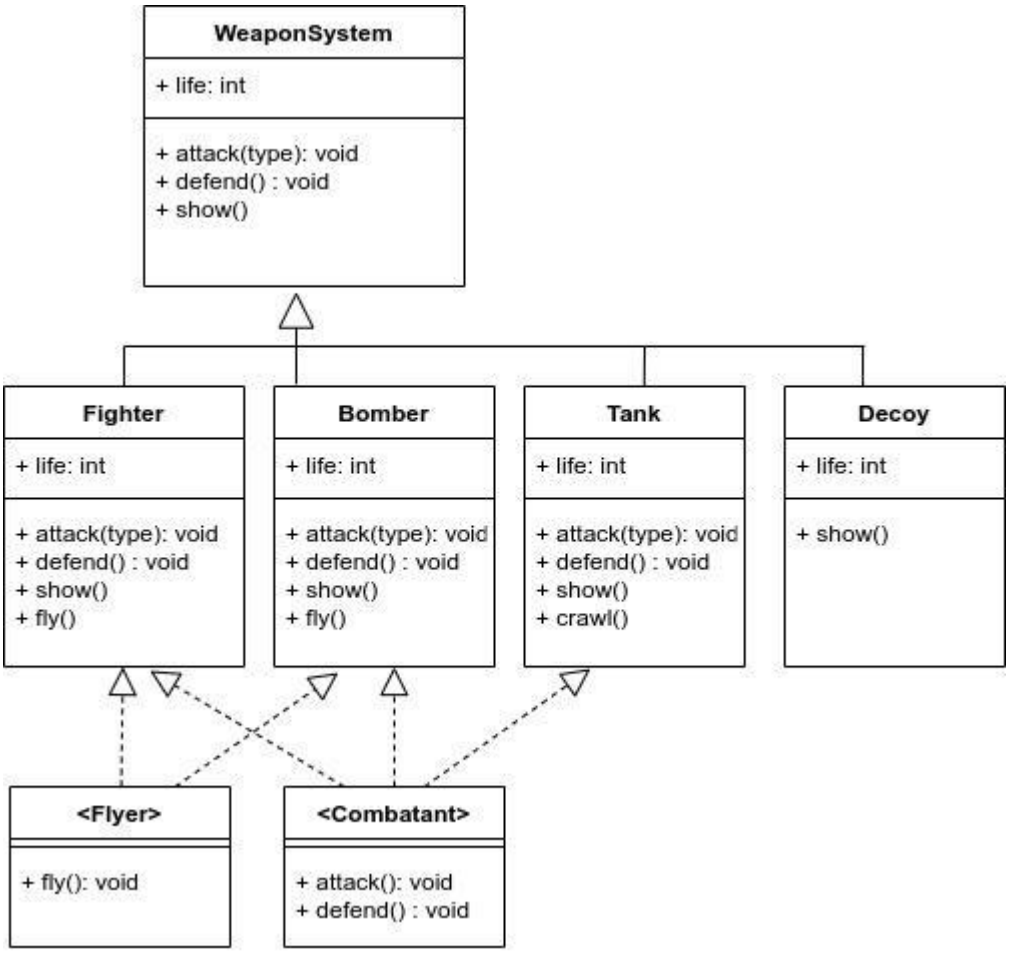


Para no repetir tanto código (porque por más que los métodos no hagan nada están ahí, escritos) podríamos hacer de WeaponSystem una clase abstracta o agregar algunas clases en medio de la jerarquía, en las cuales tengamos implantaciones vacías por defecto. Sin embargo, de esa manera no solucionamos el problema: sigue habiendo vehículos que tienen disponibles operaciones que ni siquiera deberían poder intentar. No importa que un tanque "no haga nada" al intentar volar, ¡nunca debería tener esa operación disponible!

Intentando solucionar el problema

Con lo visto anteriormente, podemos ver las **desventajas que puede tener la herencia** para establecer el comportamiento de un sistema de armas. Así como lo planteamos, una solución con una herencia simple duplica código en las hijas (o en su defecto, nos lo podríamos ahorrar en una clase abstracta). Además, un cambio puede ser sencillo (agregar o quitar un método), pero podría afectar todo el modelo dado que ese método se hereda en las subclases. Asimismo, cambiar el comportamiento de los vehículos mientras el programa corre es casi imposible. Justamente, lo que queremos evitar es detener todo el sistema para hacer cambios.

La primera solución puede venir del lado de las interfaces. Sabemos que nos pueden ayudar mucho, ya que nos permiten agregar comportamiento sin depender de una jerarquía. Veamos cómo quedaría el modelo anterior con interfaces (se sacaron algunas clases por brevedad).



¡Bien! Implementamos interfaces, solucionamos parte del problema. En nuestra pantalla, puede haber ítems combatientes (Combatants), ítems que vuelan que son objetos de tipo Flyer, por ejemplo. Incluso tenemos al Señuelo (Decoy) que no hace nada de eso.

Recordemos que **una de las premisas impuestas era que se pueda cambiar la forma en que realiza sus acciones** cada sistema de armas según algún esquema de pagos.

Por ejemplo, si el usuario nos paga 5 dólares tiene la posibilidad de hacer vuelo hipersónico con sus ítems voladores durante 1 semana. Entonces, cualquier ítem volador, en vez de moverse los casilleros en la pantalla que corresponde, lo multiplica por 3. ¿Cómo podríamos, durante la ejecución, es decir, sin detener el sistema, hacer que los

Flyers de un usuario particular se movieran más casilleros de lo normal? ¿Qué pasaría si hubiera 10 o 20 sistemas de armas diferentes, a lo cuales con un pago adicional les pudiésemos cambiar sus comportamientos? Definitivamente no podemos darle una "versión mejorada" del juego al usuario que pagó 5 dólares y que, al término de una semana, se le baje otra versión "normal".

Entonces, **hasta aquí las interfaces solo resolvieron una parte del problema**, es decir, ya no habrá tanques voladores (o señuelos que ataquen y defiendan). Tenemos múltiples interfaces que nos ayudan, pero **no podemos tener un solo tipo de objetos para usar polimorfismo**. Adicionalmente, de querer cambiar el comportamiento de los objetos, de agregar nuevas formas de volar o de atacar requeriría de cambios en el código. Es decir, mejoramos, pero todavía falta. **No siempre la herencia y las interfaces solucionan todos los problemas.**

Cambios en el sistema

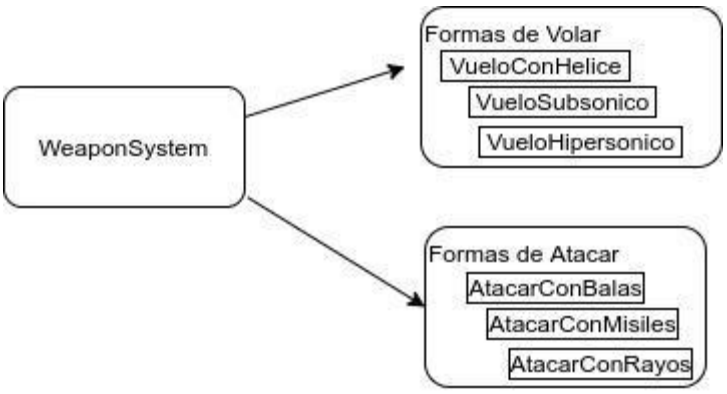
Según lo que vimos, debemos tener la posibilidad de incorporar cambios continuamente (por ejemplo, en época de navidad podríamos hacer que un bombardero lance gorros de Santa Claus, por USD 2.99, solo por diversión). Terminada la época de Navidad, deberemos sacar esa opción. Es decir, siempre hay nuevos cambios, nuevas necesidades, nuevas reglas, etc. Lo mejor, para ahorrarse problemas a futuro, es "encerrar" aquellos aspectos que cambien, para que al cambiar no afecten el resto del sistema. Conviene **encapsular** todo lo que cambie y aislarlo de lo que es fijo a lo largo del tiempo en nuestro sistema. Así, se logrará que los cambios no afecten de forma inesperada a otras partes del sistema y se logrará mayor flexibilidad.

En este sentido, es importante identificar los aspectos de nuestra aplicación que sean cambiantes y separarlos de aquello que quedan siempre fijos. Debemos agarrar las partes de nuestro sistema que varían regularmente y encapsularlas para que luego se puedan extender o cambiar sin afectar a las partes que no varían.

Volviendo a nuestro juego, sabemos que los comportamientos attack() y fly() y podrían cambiar de un WeaponSystem a otro. Por su lado, el show() o el defend() son siempre los mismos: uno muestra los datos en pantalla y el otro se interpone entre un enemigo y su objetivo.

Una alternativa, sería aislar estos dos comportamientos y englobarlos en pequeñas "familias" de comportamientos similares. Es decir, deberíamos tener algo que agrupe las diferentes formas de realizar el ataque con attack(), y las diferentes formas de volar con el fly().

Por cuestiones de simplicidad, diremos que las operaciones de show() y defend() son siempre iguales. Entonces, nuestra clase WeaponSystem conservará el atributo “life” y las operaciones show() y defend() como están. Solo quitaremos y aislaremos las operaciones de fly() y attack().



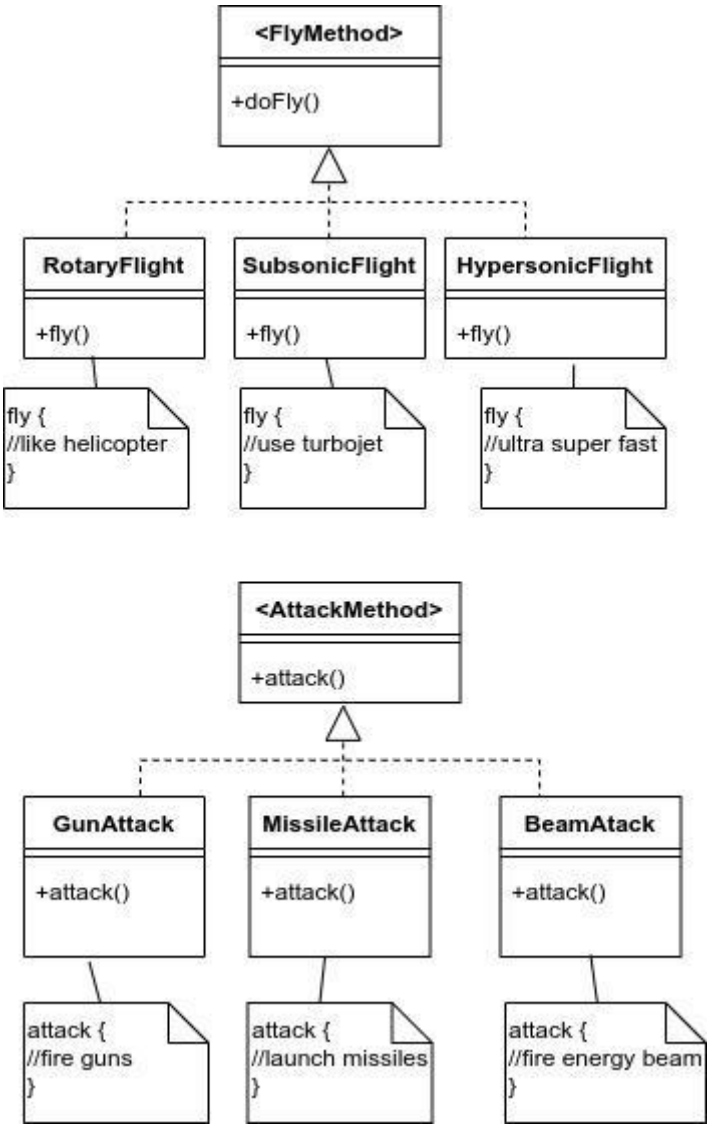
Familias de comportamientos

Según lo que vimos, podemos tener una forma de volar o de atacar por defecto. Si el usuario paga un extra, deberíamos poder “asignar” una forma de atacar o de volar en diferentes momentos durante la ejecución del juego, sin parar el sistema, sin cambiar el código, sin liberar una nueva versión, yo quiero poder cambiar el comportamiento. Si el usuario pagó un extra, le puedo cambiar la forma de volar de sus aviones.

En ese sentido, cada forma de atacar o volar ya no vendrá dictada por el sistema de armas del cual se trate. Sino que el sistema de armas tiene una forma de atacar o volar. Hasta aquí el comportamiento de atacar y el código asociado (su implementación) estaba fijo. Ahora podemos establecerlo en tiempo de ejecución. Sin embargo, hay que tener cuidado, porque de poner la forma concreta de atacar o volar, estaríamos acoplando una “forma de” con un sistema de armas en particular. ¿Recuerdan lo que ocurría con las interfaces en el caso de las Colecciones? Teníamos un tipo de colección, pero podíamos alterar las implementaciones y todo seguía funcionando. Sería interesante aplicar aquí el mismo concepto. Necesitamos introducir una interfaz que dictamine de la forma de volar y atacar. De esta forma, cuando a un WeaponSystem se le ordene atacar o volar (se le ejecute el método attack() o fly()) este lo hará directamente sin saber los detalles de implementación de cada comportamiento. Esto se debe a que hemos encapsulado las formas de volar y atacar.

Antes, lo que hacíamos era o heredar directamente el comportamiento desde de la clase SistemaDeArmas o especificarlo en alguna de sus hijas. En ambos casos, dependíamos de una implementación. De esta manera, **no se podía alterar el comportamiento sin alterar el código**.

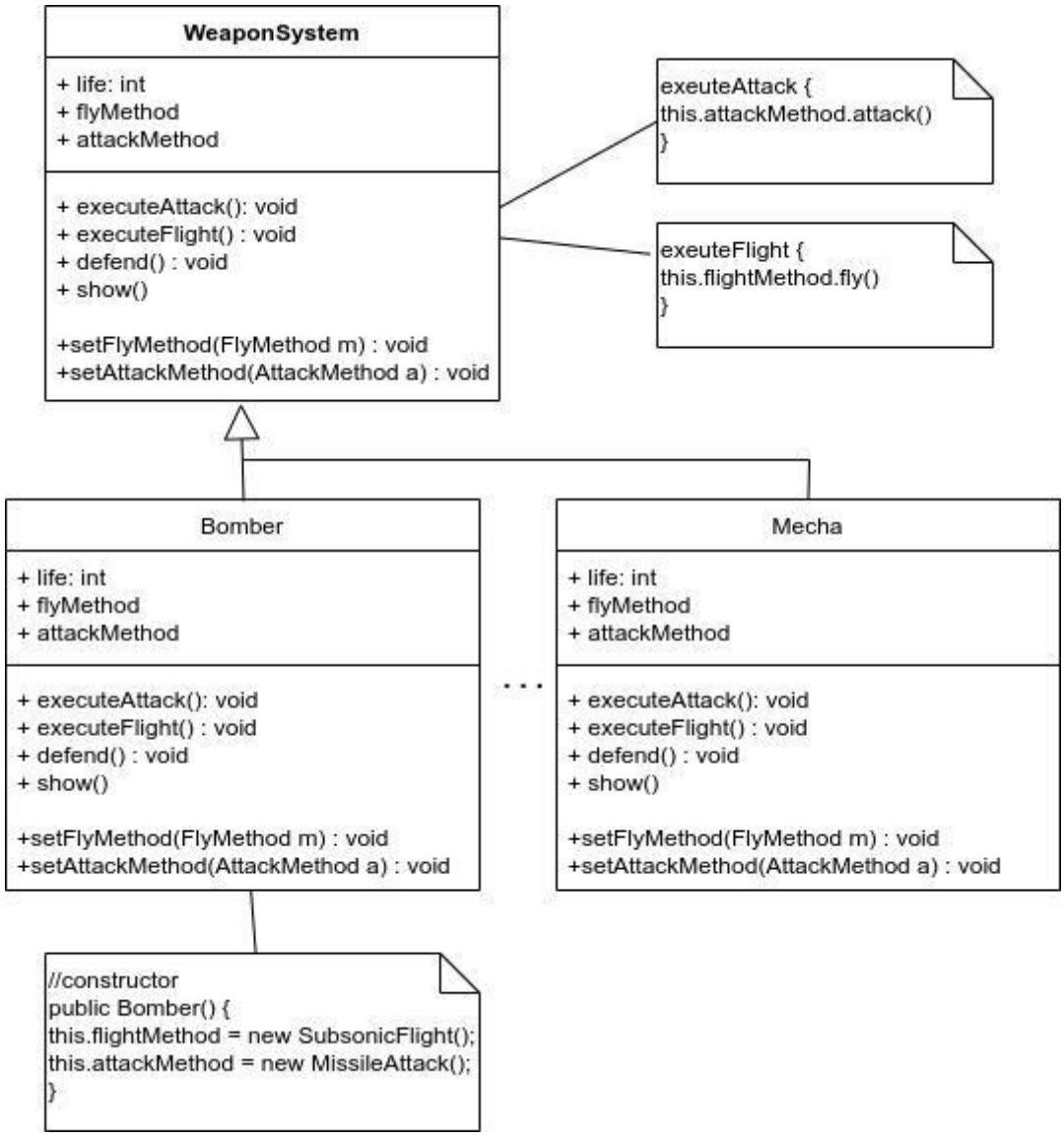
Ahora, de lo que disponemos es una **interfaz que define un comportamiento**. Las sucesivas y diferentes implementaciones de estas dictarán cómo se llevará a cabo. ¡A partir de ahora, podemos alterar el comportamiento sin tocar código!



Un efecto secundario de este approach es el hecho de que las clases que implementan los diferentes comportamientos ahora pueden ser reutilizadas en otros escenarios. Por ejemplo, si decidiéramos actualizar nuestro juego y que las batallas sucedan también en el

espacio, algunas “formas de atacar” podríamos utilizarlas nuevamente. Sin mencionar que se pueden agregar nuevas formas sin afectar el diseño ni el código existente.

Incorporando las formas de volar y atacar, nuestra aplicación original quedaría de la siguiente manera (se omiten algunas clases para mayor brevedad):



Las formas de atacar y volar son ahora referencias a clases que tendrán la responsabilidad de atacar y volar según corresponda. En vez de que el SistemaDeArmas sea quien maneje el ataque y el vuelo, estos comportamientos se delegan al objeto referenciado por el atributo “flightMethod” o “attackMethod”. Hasta aquí, no importa qué método de ataque o de vuelo se trate, solo importa que ese objeto sabe cómo volar (tiene implementado el fly()) de la forma correspondiente. Pasa lo mismo con el ataque.

Entonces, tiempo de ejecución, cuando sea época de Navidad y el usuario pague unos dólares para divertirse, podrá lanzar gorros de Santa sobre sus enemigos. Al momento del pago, a todos los bombarderos del usuario le haremos:

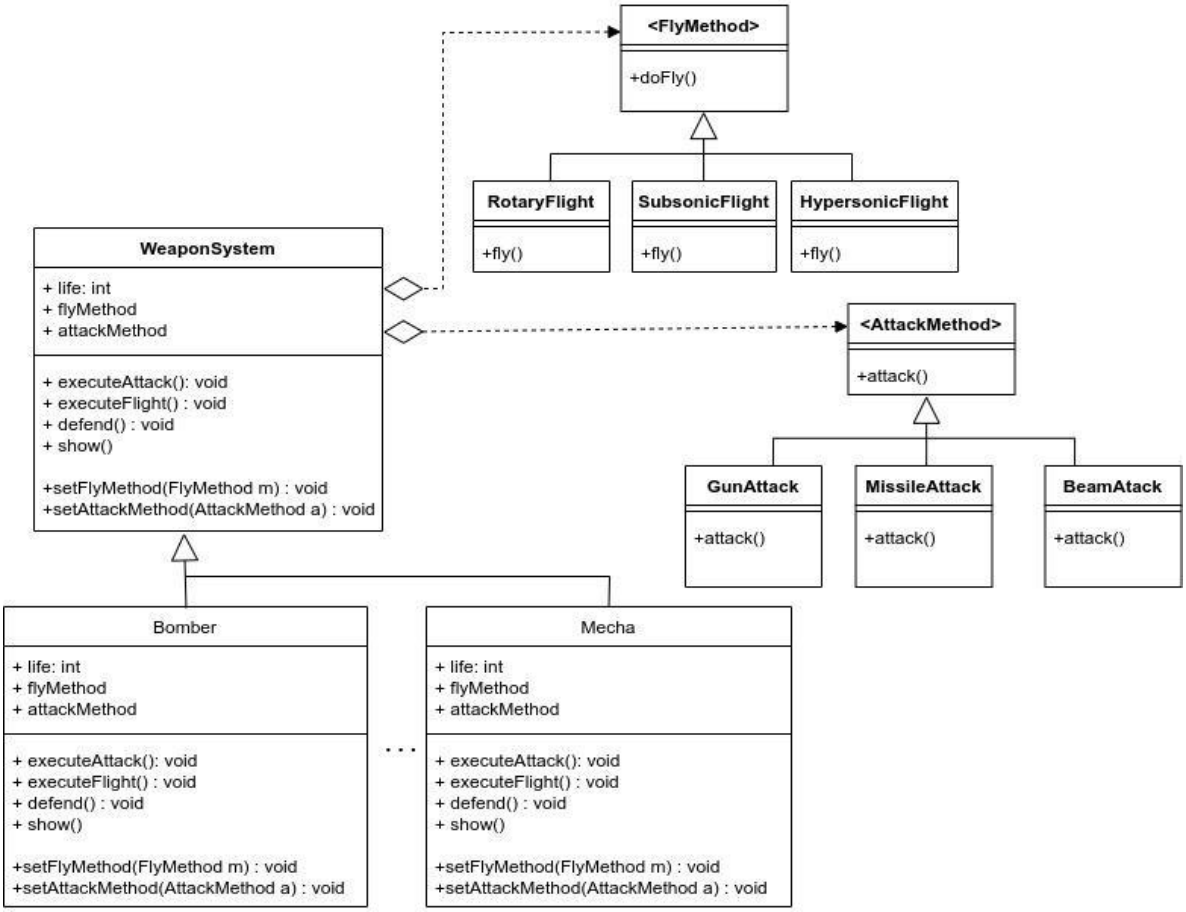
```
setAttackMethod(new SantaBombs());
```

Luego, sistema de armas (un bombardero, en este caso), cada vez que ejecute el método heredado executeAttack(), lanzará simpáticos gorritos destruyendo todo a su paso. Terminada la semana, volveremos a hacer a cada bombardero:

```
setAttackMethod(new MissileAttack());
```

Favorecer la composición por sobre la herencia

Si vemos el panorama completo, veríamos algo así (se quitaron algunas clases para mayor brevedad):



Al ver el problema de forma general, podemos considerar cada “forma de” como diferentes “algoritmos” o “familias de algoritmos”. Es decir, cada “algoritmo” representa algo que un sistema de armas puede hacer.

Ahora sabemos que componer nos da mayor flexibilidad que heredar. Heredar nos “encierra” en comportamientos fijos, las herencias son muy rígidas y requieren de modificaciones al código para poder alterar el comportamiento (sobrescribir un método, por ejemplo). Componer nos permite encapsular los comportamientos e intercambiarlos sin afectar otras partes del sistema y más aún: nos permite cambiar los comportamientos en **RUNTIME** (siempre que se respeten las interfaces), mediante simples métodos setter.

Por lo tanto, resulta muy importante **favorecer la composición por sobre la herencia.**

¿Te animas a revisitar el problema anterior del zoo virtual y aplicar composición? ¿Crees que es posible? ¿En qué punto de la jerarquía de clases lo aplicarías? ¡Inténtalo!