Universidad de Jaén
Departamento de Informática

Programación Orientada a Objetos

http://bit.ly/poouja

Curso 2019/2020

José Ramón Balsas Almagro
Víctor Manuel Rivas Santos
Ángel Luis García Fernández
Juan Ruiz de Miras

Práctica 6. Relaciones entre clases (II)

Objetivos

- Continuar trabajando con la creación de clases
- Seguir familiarizándonos con los diagramas de clases UML
- Implementar relaciones de agregación y composición, y con multiplicidad variable

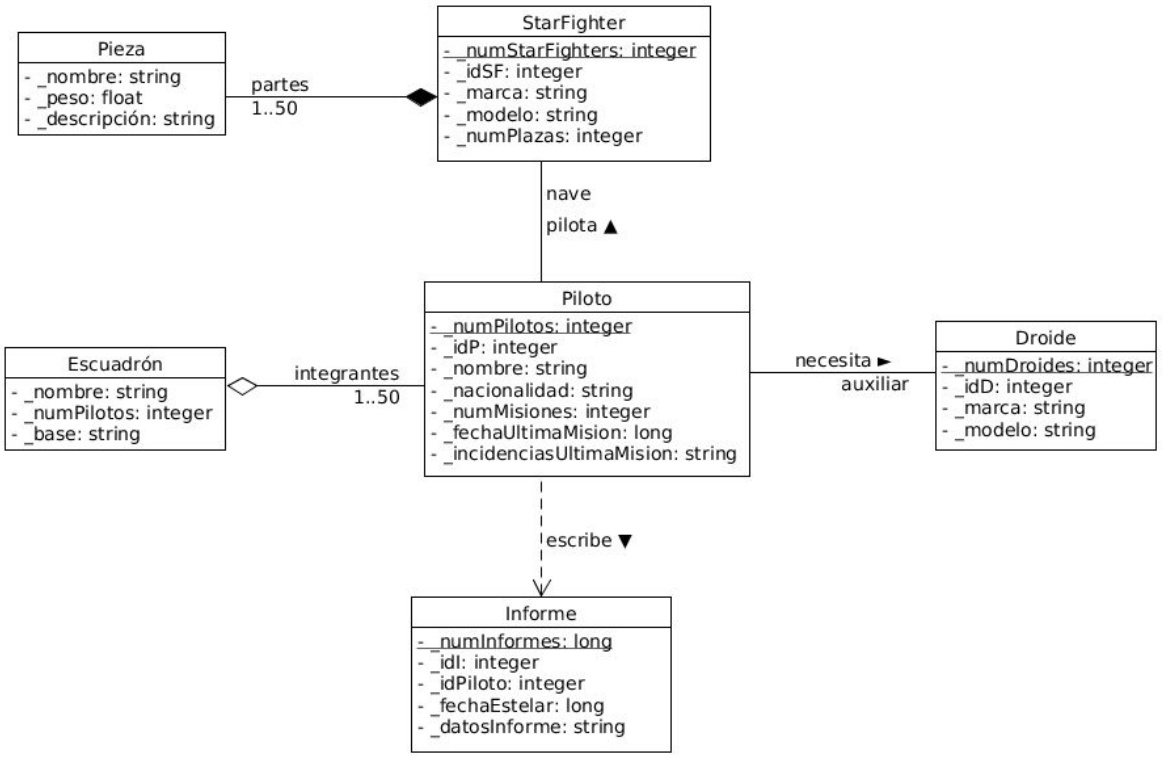
Índice

Contexto de la práctica

Una vez que Casipro Gramo y Yoyapro Gramo, gracias a nuestra ayuda, han conseguido implementar la gestión de naves, pilotos, droides e informes, reciben un nuevo encargo: ampliar la aplicación, incluyendo ahora datos de las piezas que componen las naves, así como organizando los pilotos en escuadrones.

Las piezas que componen las naves se fabrican de forma totalmente adaptada a la nave en cuestión, de forma que no es posible utilizar piezas de una nave en otra distinta. Los pilotos, en cambio, pueden ser asignados a diferentes escuadrones, en función de las bajas que se sufran tras cada batalla.

Dados los nuevos requerimientos, el maestro Yotepro Gramo ha modificado el plano de la aplicación para adaptarlo a la nueva versión:



Una vez más, Casipro Gramo y Yoyapro Gramo se enfrentan al reto de salvar a la flota de la Resistencia del caos organizativo. ¿Podrás ayudarles?

Punto de partida

El punto de partida de esta práctica son las clases ya escritas por Yoyapro y Casipro Gramo al alimón. El proyecto NetBeans (sí, el Clan todavía usa NetBeans ;-)) está disponible en el siguiente enlace: http://bit.ly/poouja1617pr6

Ejercicios

1. Modifica la clase *StarFighter* para que se incluya la relación entre un star fighter y sus piezas. Añade métodos para añadir y eliminar piezas del star fighter, teniendo en cuenta lo que se ha dicho antes: una pieza sólo puede ser utilizada en un único starfighter, y al retirarla de la nave, hay que destruirla

2. Implementa la clase *Escuadrón*, teniendo en cuenta la relación existente entre un escuadrón y los pilotos que lo forman

3. Añade a la clase *StarFighter* un método *calculaPeso()*, que devuelve la suma de los pesos de las piezas que componen el starfighter

4. Añade a la clase *Escuadrón* un método *promedioMisiones()*, que devuelve el número medio de misiones de los pilotos integrantes del escuadrón

5. Completa la función *main*:

- Crea un array de 5 pilotos.
- Utilizando el método *Piloto::fromCSV* carga en el array de pilotos los datos proporcionados en el array *datosPilotos[]*.
- Crea 2 escuadrones. El primero estará formado por los dos primeros pilotos, y el segundo por los otros tres
- Muestra por consola el nombre y el promedio de misiones de cada escuadrón
- Crea 2 starfighters, y añade 3 piezas a cada uno
- Muestra por consola el peso de cada starfighter
- Libera los recursos utilizados

6. (**Opcional**) En algunos puntos en el código, Casipro y Yoyapro Gramo dejaron comentarios Doxygen indicando tareas pendientes (con la etiqueta **@todo**), principalmente relacionadas con excepciones. Intenta terminarlas

Contenidos

Agregación y composición

En esta práctica completamos la primera parte del Tema 3, dedicado a las relaciones entre clases. A la dependencia y la asociación ya estudiadas en la práctica anterior, añadimos ahora la agregación y la composición.

- La **agregación** implica que los objetos de una clase A almacenan información relativa a un grupo de objetos de clase B, con un cierto orden jerárquico: un equipo de fútbol es una agregación de futbolistas; un ejército es una agregación de soldados; un ramo de flores es una agregación de flores… Los objetos agregados pueden existir fuera de la agregación.
- La **composición** representa un nivel más de *intensidad semántica* en la relación; los objetos de clase A están hechos de objetos de clase B, de forma que si un objeto de clase A se destruye, también hay que destruir los objetos de clase B que lo forman: un pedido está hecho de líneas de pedido; un microprocesador está compuesto por, entre otras cosas, la memoria caché y los registros; un blog está compuesto por las entradas que su autor escribe… En general, los componentes no pueden existir fuera del compuesto.

A la hora de la implementación, hay que tener en cuenta los siguientes aspectos, ya comentados en las clases de teoría:

- En general, la **agregación** se suele implementar mediante punteros o referencias: los objetos de clase A hacen referencia así a los objetos de clase B que agregan. Serán necesarios métodos en la clase A para añadir y quitar objetos de clase B, pero la responsabilidad de destruir los objetos que se retiren de la agregación **no** recae en la clase A, ya que esos objetos se pueden reutilizar.
- La **composición** también se suele implementar mediante punteros desde los objetos

compuestos a los objetos componentes. En este caso, **es responsabilidad del objeto compuesto** el crear y destruir los objetos que lo componen. Este aspecto se reflejará tanto en constructores y destructor como en los métodos para añadir y eliminar componentes.

En los ejemplos de código del Tema 3 tenéis ejemplos de implementación de agregación (http://bit.ly/poouja1516t32) y composición (http://bit.ly/poouja1516t33).

Relaciones con multiplicidad variable

Aunque se pueden dar situaciones en que la cardinalidad de las relaciones es fija (por ejemplo: un equipo de baloncesto está formado por 12 jugadores), lo habitual es que la cardinalidad de las relaciones sea variable (por ejemplo: una familia es una agregación de personas, pero no hay un número fijo de personas en todas las familias). En estos casos, hay que recurrir a distintos tipos de **contenedores** para almacenar los punteros o referencias a través de los cuales se establecen las relaciones.

El estudio de este tipo de contenedores no es el objetivo de esta asignatura, así que vamos a limitar su uso a una estructura ya conocida: los arrays (vectores, si se trata de una sola dimensión) de tamaño fijo.

En general, representaremos una relación con multiplicidad variable mediante un array de un tamaño máximo que elijamos (lo aconsejable es utilizar una constante para fijar este tamaño), junto con un número entero que almacene cuántas de las posiciones del array están realmente en uso. Un ejemplo del mundo real de esta estrategia es un hotel: en el momento de su construcción, no se sabe cuántas personas se van a alojar en él, sino que se proyecta la construcción de un número de habitaciones, y cuando ya está terminado, es necesario llevar un control de qué habitaciones están ocupadas y por quién; si el hotel está lleno, no se admiten más clientes.

En los ejemplos de código del Tema 3 tenéis un ejemplo de implementación de composición con multiplicidad variable (http://bit.ly/poouja1516t34).