Universidad de Granada

Facultad de Ingeniería Informática

Practica 1: Patrones de diseño

DS: Grupo 1.7

Autores :
Emanuel Giraldo Herrera
Thomas Lang
Timur Sorokin
Alejando Iborra Morán

(2023-2024)

1 Ejercicio 1 (Java)

1.1 Análisis

Se trata de realizar una simulación de carreras de bicicletas con las siguientes características:

- **Bicicleta**
  Cada bicicleta tiene un identificador. Existen dos tipos de bicicletas:
  - Carretera
  - Montaña
- **Carrera**
  No se conoce el número de bicicletas antes de que empiece la carrera. Además, para cada tipo de bicicleta existe su tipo de carrera con las siguientes peculiaridades:
  - Carretera
    Antes de finalizar 10% bicicletas abandonan la carrera
  - Montaña
    Antes de finalizar 20% bicicletas abandonan la carrera.
  - Hebras
    Se utilizarán las hebras para la simulación simultanea de varias carreras

  Finalmente, los objetivos que se buscan son aplicar dos patrones de diseño: Factoría Abstracta y Factoría método.

1.2 UML

Antes de realizar la implementación primero hemos de plantear el diagrama UML que relacione las entidades del problema y refleje sus interacciones.

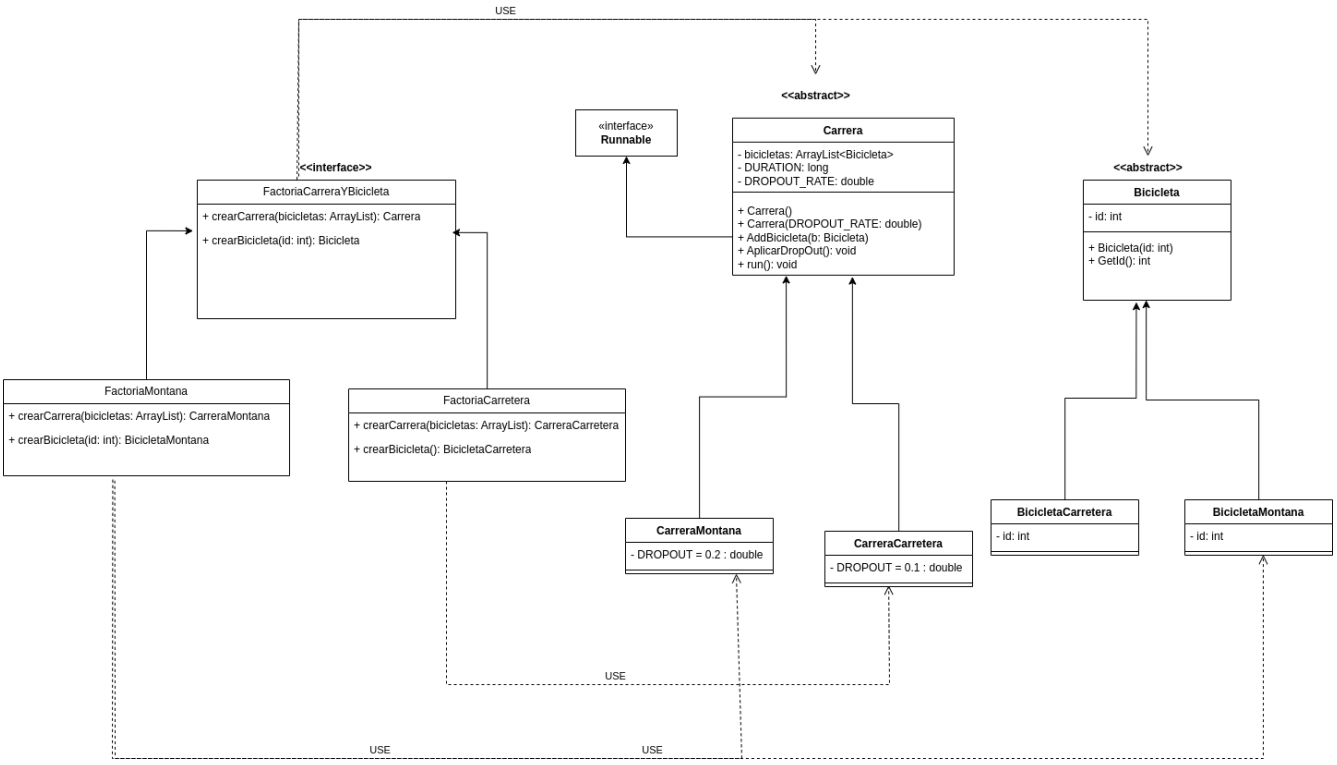

Figure 1: Factoría Abstracta

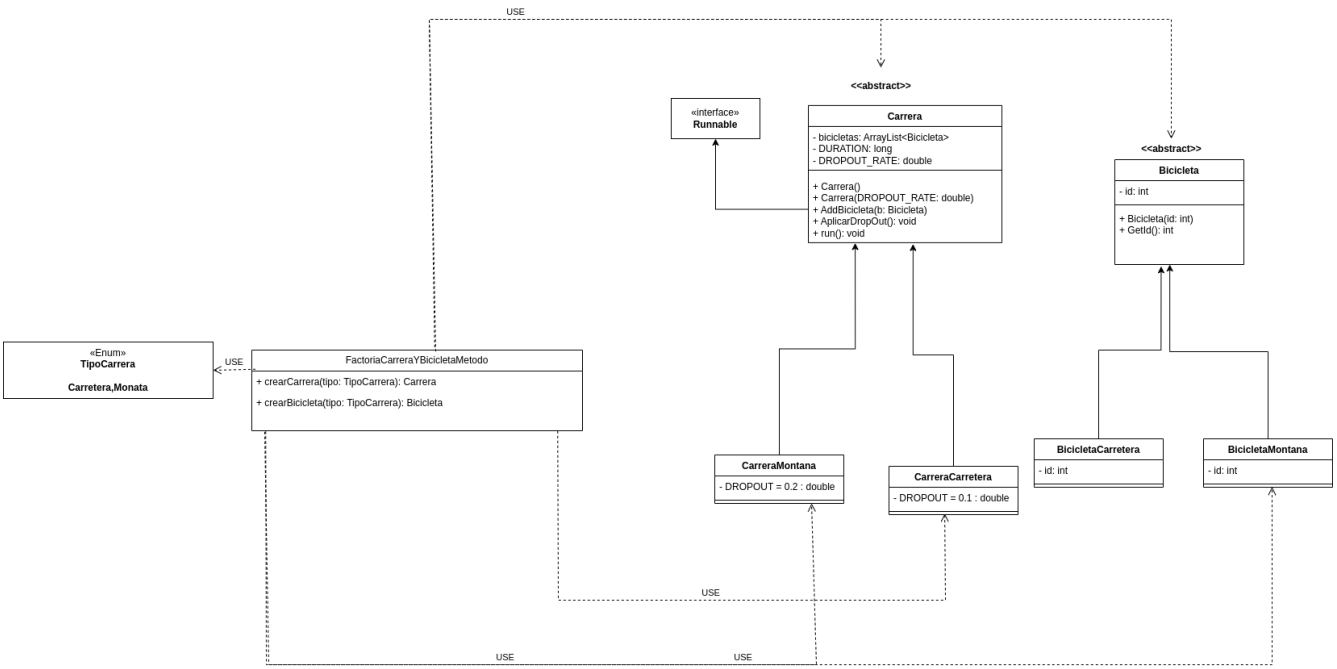

Figure 2: Factoría Método

1.3 Factoría Abstracta

Se trata de un patrón creacional que define una interfaz para crear objetos relacionados sin especificar las clases concretas de esos objetos. Así conseguimos que el código cliente pueda instanciar los objetos sin conocer sus tipos concretos. Esto nos permite desacoplar las implementaciones concretas del código cliente.

En nuestro problema tenemos dos entidades básicas que son *Bicicleta* y *Carrera*. Hemos visto durante el análisis que existe una relación entre estas dos clases, es decir, en una *Carrera* compiten las *Bicicletas*. También vimos que ambas entidades pueden ser de dos tipos: *carretera* y *montaña*. Cada uno de ellos con sus características *concretas*. Entonces, tenemos que en una *Carrera de montaña* participan *Bicicletas de montaña*. Y en una *Carrera de carretera* participan *bicicletas de carretera*. Por tanto distinguimos los siguientes elementos:

- ***Bicicleta***
  Es una entidad que representa una bicicleta abstracta y actúa como una base para los distintos tipos de bicicletas que pueden existir.
  - **BicicletaCarretera**
    Es una bicicleta concreta con características añadidas que son propias de carretera.
  - **BicicletaMontana**
    Es una bicicleta concreta con características añadidas que son propias de montaña.
- ***Carrera***
  Al igual que en el caso de *Bicicleta* se trata de una clase abstracta pues engloba ciertas características comunes a todos los tipos de carreras que pueden existir.
  - **CarreraCarretera**
    Es una carrera concreta en la que participan bicicletas de carretera.
  - **CarreraMontana**
    Es una carrera concreta en la que participan bicicletas de montaña.

- ***FactoriaCarreraYBicicleta***
  Es una interfaz que define dos operaciones: crear bicicleta y crear carrera. No obstante, no especifica el tipo concreto del objeto que se va a crear.
  - **FactoriaCarretera**
    Es una factoría especializada en crear las carreras y bicicletas de carretera
  - **FactoriaMontana**
    Es una factoría especializada en crear las carreras y bicicletas de montaña

1.4 Consideraciones sobre Factoría Abstracta

- La clase *Carrera* implementa interfaz Runnable para representar código que puede ser ejecutado por un hilo independiente para conseguir la concurrencia de las carreras.
- La clase *Carrera* es abstracta pero no tiene ningún método abstracto puesto que, según el enunciado dado, todas las carreras tienen el mismo comportamiento que se resumen en añadir bicicleta, aplicar dropout y simular la competición. Entonces parece más coherente definir un comportamiento predeterminado en la clase base/abstracta que repetir el mismo código en diferentes archivos. Además, se mantiene como abstracta puesto que preferimos restringir la instanciación de esta clase.
- Se ha aplicado el mismo razonamiento para la *Bicicleta*. Aunque los dos tipos de bicicletas que hay podrían ser suprimidos debido a que no aportan nada al problema pues solo se diferencian en el dato miembro *id*. Quizá en una simulación más compleja en la que, por ejemplo, la velocidad máxima este definida por el tipo de bicicleta tendría más significado la separación de los tipos.

1.5 Factoría Método

Es un patrón creacional que permite crear objetos sin especificar la clase. En vez de crear los objetos de forma directa lo que haremos es usar un método para crear y devolverlos.

Para ello se ha optado por definir una nueva entidad *FactoriaCarreraYBicicletaMetodo* en la que se definen dos operaciones: *crearBicicleta(TipoCarrera)* y *crearCarretera(TipoCarrera)*. En ambas mediante un switch basado en *TipoCarrera* se decide el tipo de objeto se debe crear siendo *TipoCarrera* un enum trivial.

1.6 Solución

En la carpeta src/EJ1 ofrecemos una solución al problema planteado. En esta carpeta se puede consultar los archivos correspondientes a cada entidad del digrama UML.

En el archivo *main.java* se han definido dos funciones *FinalFactoriaAbtracta* y *FinalFactoriaMetodo* en las que se muestra el uso de los dos patrones implementados.

2 Ejercicio 2

2.1 Análisis

En este problema se pide realizar la implementación de Factoria Abstracta del ejercicio 1 en el lenguaje Python sin utilizar las hebras y además realizar la implementación del patron *Prototipo*.

2.2 Factoria Abstracta

Utilizamos el modulo *abc* para poder definir clases abstractas. Las clases abstractas son clases que no pueden ser instanciadas directamente, sino que se utilizan como base para otras clases que las heredan. El planteamiento no cambia en absolutamente nada en comparación con el ejercicio 1 siendo la única diferencia el lenguaje de programación y su sintaxis subyacente.

2.3 Prototipo

Este patrón consiste en crear nuevos objetos clonando otro existente. En nuestro caso vamos a definir un método para la entidad *Bicicleta* que nos permita realizar una copia del objeto existente. Puesto que hablamos de copiar los objetos es necesario recordar los conceptos *shallow copy y deep copy*.

- *Shadow Copy* copia solo la estructura del objeto pero comparte los datos miembros entre el original y la copia, lo que hace que los cambios afecten a ambos. Es más rápida y consume menos memoria.
- *Deep Copy* crea copias independientes evitando la compartición de memoria por tanto los cambios en una copia no afectan al objeto original ni viceversa. Es más lenta y consume más memoria.

Para implementar metodo *clone()* podemos recurir al *constructor de copia* en el que realizaremos una copia profunda del objeto recibido como argumento. O alternativamente podemos usar modulo *copy* que nos dá interfaces para duplicar los objetos. Nosotros hemos optado por este último.

bñlñabladfgadfgkjkadrogjadfgh