DS - Práctica 1

Javier Martínez Godoy
Jesús Pérez León
Ane Anta Iriondo
Ángel Muñoz Cortes
Adrián Romero Vilchez

24 de marzo de 2024

Repositorio Github

UNIVERSIDAD DE GRANADA

Índice

1. Ejercicio 1

1.1. Enunciado

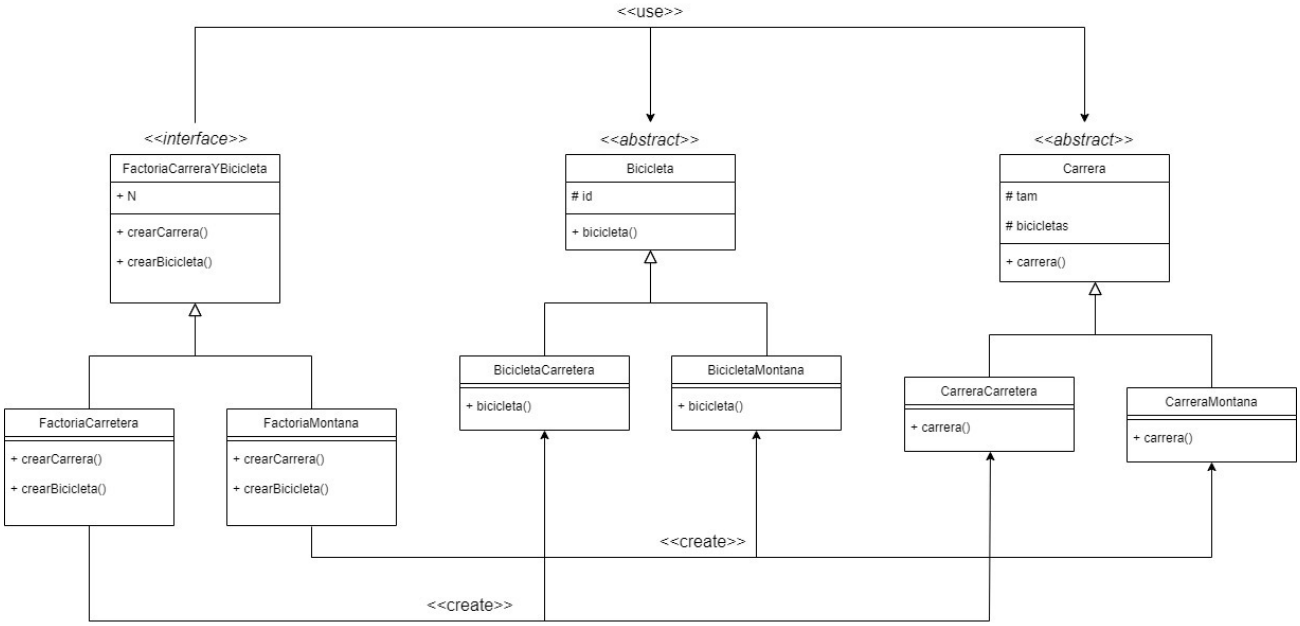
Programa utilizando hebras la simulación de 2 carreras simultáneas con el mismo número inicial (N) de bicicletas. N no se conoce hasta que comienza la carrera. De las carreras de montaña y carretera se retirarán el 20 % y el 10 % de las bicicletas, respectivamente, antes de terminar. Ambas carreras duran exactamente 60 s. y todas las bicicletas se retiran a la misma vez.

Deberán seguirse las siguientes especificaciones:

- Se implementará el patrón de diseño *Factoría Abstracta* junto con el patrón de diseño *Método Factoría*.
- Se usará Java como lenguaje de programación.
- Se utilizarán hebras tanto para las carreras como para las bicicletas.
- Se implementarán las modalidades montaña/carretera como las dos familias/estilos de productos. [1]
- Se definirá la interfaz Java *FactoriaCarreraYBicicleta* para declarar los métodos de fabricación públicos:
  - *crearCarrera* que devuelve un objeto de alguna subclase de la clase abstracta *Carrera* y
  - *crearBicicleta* que devuelve un objeto de alguna subclase de la clase abstracta *Bicicleta*.
- La clase *Carrera* tendrá al menos un atributo *ArrayList<Bicicleta>*, con las bicicletas que participan en la carrera. La clase *Bicicleta* tendrá al menos un identificador único de la bicicleta en una carrera. Las clases factoría específicas heredarán de *FactoriaCarreraYBicicleta* y cada una de ellas se especializará en un tipo de carreras y bicicletas: las carreras y bicicletas de montaña y las carreras y bicicletas de carretera. Por consiguiente, tendremos dos clases factoría específicas: *FactoriaMontana* y *FactoriaCarretera*, que implementarán cada una de ellas los métodos de fabricación *crearCarrera* y *crearBicicleta*.
- Se definirán las clases *Bicicleta* y *Carrera* como clases abstractas que se especializarán en clases concretas para que la factoría de montaña pueda crear productos *BicicletaMontana* y *CarreraMontana* y la factoría de carretera pueda crear productos *BicicletaCarretera* y *CarreraCarretera*.

[1] Considérese que el concepto de producto en patrones de diseño, tiene una definición muy abierta, para que pueda ajustarse a cada problema concreto, pero que en todo caso los productos deberán implementarse como objetos que pueden ser muy distintos entre sí, es decir, sin relación de herencia entre ellos.

1.2. Diagrama UML



1.3. Resolución

Empezamos creando la *interfaz FactoriaCarreraYBicicleta*, que define un contrato para una factoría abstracta que crea objetos relacionados con carreras y bicicletas. Los métodos que implementa son los siguientes:

- *crearCarrera*: Mëtodo para crear un carrera. Devuelve una instancia de la clase Carrera.
- *crearBicicleta*: Método para crear una bicicleta. Devuelve una instancia de la clase Bicicleta

La clase abstracta *Bicicleta* proporciona una estructura base para representar bicicletas. Contiene un atributo **id**, tipo entero que es el identificador único de cada bicicleta. Tiene un método abstracto *bicicleta()* que muestra la bicicleta, y será implementado por las clases concretas.

La clase abstracta *Carrera* proporciona una estructura base para representar carreras. Contiene un atributo **tam**, tipo entero que representa el número máximo de bicicletas que participan en la carrera, además de un ArrayList<Bicicleta> **Bicicletas**, que almacena las bicicletas que participan en la carrera. Tiene un método abstracto *carrera()* que muestra la carrera, y será implementado por las clases concretas. También tiene definido el constructor *Carrera(tam)*, que recibe el tamaño de de la carrera.

La clase *BicicletaCarretera* representa una bicicleta diseñada específicamente para carreras de carretera. Implementa el método *bicicleta()*, heredado de *Bicicleta*, que muestra una bicicleta de carretera, e imprime un mensaje indicando que es una bici de carretera y su respectivo identificador.

La clase *BicicletaMontana* representa una bicicleta diseñada específicamente para carreras de montaña. Implementa el método *bicicleta()*, heredado de *Bicicleta*, que muestra una bicicleta de montaña, e imprime

un mensaje indicando que es una bici de montaña y su respectivo identificador.

La clase *CarreraCarretera* representa un carrera específica que se lleva a cabo en carreteras. Implementa el método *carrera()*, heredado de *Carrera*, que muestra un carrera de carretera, e imprime un mensaje indicando que es una carrera de carretera. También implementa el constructor *CarreraCarretera(tam)*, que recibe el tamaño de la carrera.

La clase *CarreraMontana* representa un carrera específica que se lleva a cabo en montañas. Implementa el método *carrera()*, heredado de *Carrera*, que muestra un carrera de montaña, e imprime un mensaje indicando que es una carrera de montaña. También implementa el constructor *CarreraMontana(tam)*, que recibe el tamaño de la carrera.

La clase *FactoriaCarretera* implementa la intezfaz *FactoriaCarreraYBicicleta* y proporciona métodos para crear carreras y bicicletas específicas para carreteras. Implementa los siguientes métodos:

- *crearCarrera()*: Método para crear una carrera de carretera. Devuelve una instancia de la clase *Carrera* que representa una carrera de carretera. El método crea una instancia de CarreraCarretera con el tamaño N. Agrega bicicletas a la carrera. Calcula la cantidad de bicicletas a eliminar (10 %). Genera una lista de índices aleatorios para bicicletas a eliminar. Calcula la nueva longitud de la lista de bicicletas después de eliminar. Crea una nueva carrera con la nueva longitud calculada. Copia las bicicletas de la carrera original a la nueva carrera, omitiendo las eliminadas. Actualiza la carrera con la nueva carrera.
- *crearBicicleta()*: Método para crear una bicicleta de carretera. Devuelve un instancia de la clase *Bicicleta*.

La clase *FactoriaMontana* implementa la intezfaz *FactoriaCarreraYBicicleta* y proporciona métodos para crear carreras y bicicletas específicas para montañas. Implementa los siguientes métodos:

- *crearCarrera()*: Método para crear una carrera de montaña. Devuelve una instancia de la clase *Carrera* que representa una carrera de montaña. El método crea una instancia de CarreraMontana con el tamaño N. Agrega bicicletas a la carrera. Calcula la cantidad de bicicletas a eliminar (20 %). Genera una lista de índices aleatorios para bicicletas a eliminar. Calcula la nueva longitud de la lista de bicicletas después de eliminar. Crea una nueva carrera con la nueva longitud calculada. Copia las bicicletas de la carrera original a la nueva carrera, omitiendo las eliminadas. Actualiza la carrera con la nueva carrera.
- *crearBicicleta()*: Método para crear una bicicleta de montaña. Devuelve un instancia de la clase *Bicicleta*.

La clase ***main***, que es el método principal del programa. Este método crea las factorías y carreras para las carreras de montaña y de carretera, crea e inicia las hebras para ejecutar las carreras, y luego espera a que las hebras terminen. Finalmente, imprime el número de bicicletas que completaron la carrera.
También se crea la clase *CarreraRunnable*, que implementa la interfaz Runnable y representa una hebra que ejecuta una carrera. Tiene definidos e implementados dos métodos:

- *CarreraRunnable(Carrera carrera)*: Es el constructor de la clase, que recibe la carrera que la hebra va a ejecutar.
- *run()*: Método que se ejecuta cuando se inicia la hebra. Este método hace que la hebra dure 60 segundos e imprima el identificador de cada bicicleta que completó la carrera.

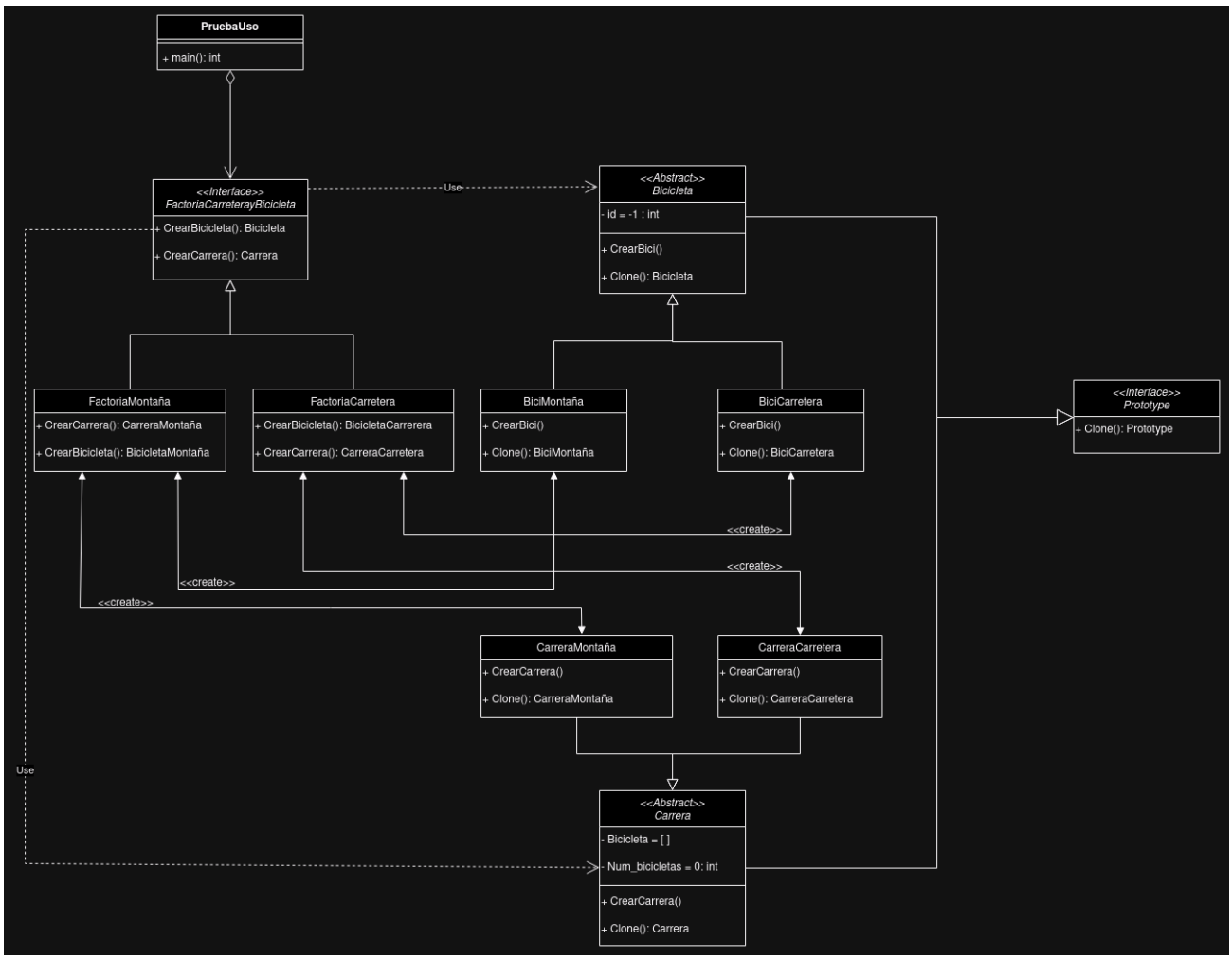
2. Ejercicio 2

Enunciado

Diseña e implementa una aplicación con la misma funcionalidad que la del ejercicio anterior, pero que aplique el patrón *Prototipo* junto con el patrón *Factoría Abstracta*.

Se elegirá el lenguaje de programación Python o Ruby. Para simplificar: no es necesario el uso de hebras.

2.1. Diagrama UML



2.2. Resolución del ejercicio

La resolución de este ejercicio es muy parecida a la anterior, donde hemos creado una factoria *interface* de carreras y bicicletas, de la que heredan las factorias de Montaña y de Carretera.

Estas últimas, implementan varios métodos heredados de la clase *interface*, llamados *crearCarrera()* y *crearBicicleta()*, donde en cada Factoria se creará un objeto diferente, bien sea de montaña o de carretera.
Por otro lado, tenemos dos clases ≪*abstract*≫: *Bicicleta* y *Carrera*, que definen un método para crear un objeto de la clase, que será implementado en las clases hijas, *BiciMontaña*, *BiciCarretera*, *CarreraMontaña* o *CarreraCarretera*, creando así el objeto respectivo de montaña o de carretera.

Para crear los objetos no es necesario que se haga uno a uno, sino que podemos crear únicamente el primero y a partir de ese, clonar el resto. De este modo, hemos creado una clase *interface Prototype* donde definimos un método *clone()* que heredarán *Bicicleta* y *Carrera*, que a la vez heredarán e implmentarán sus hijas.

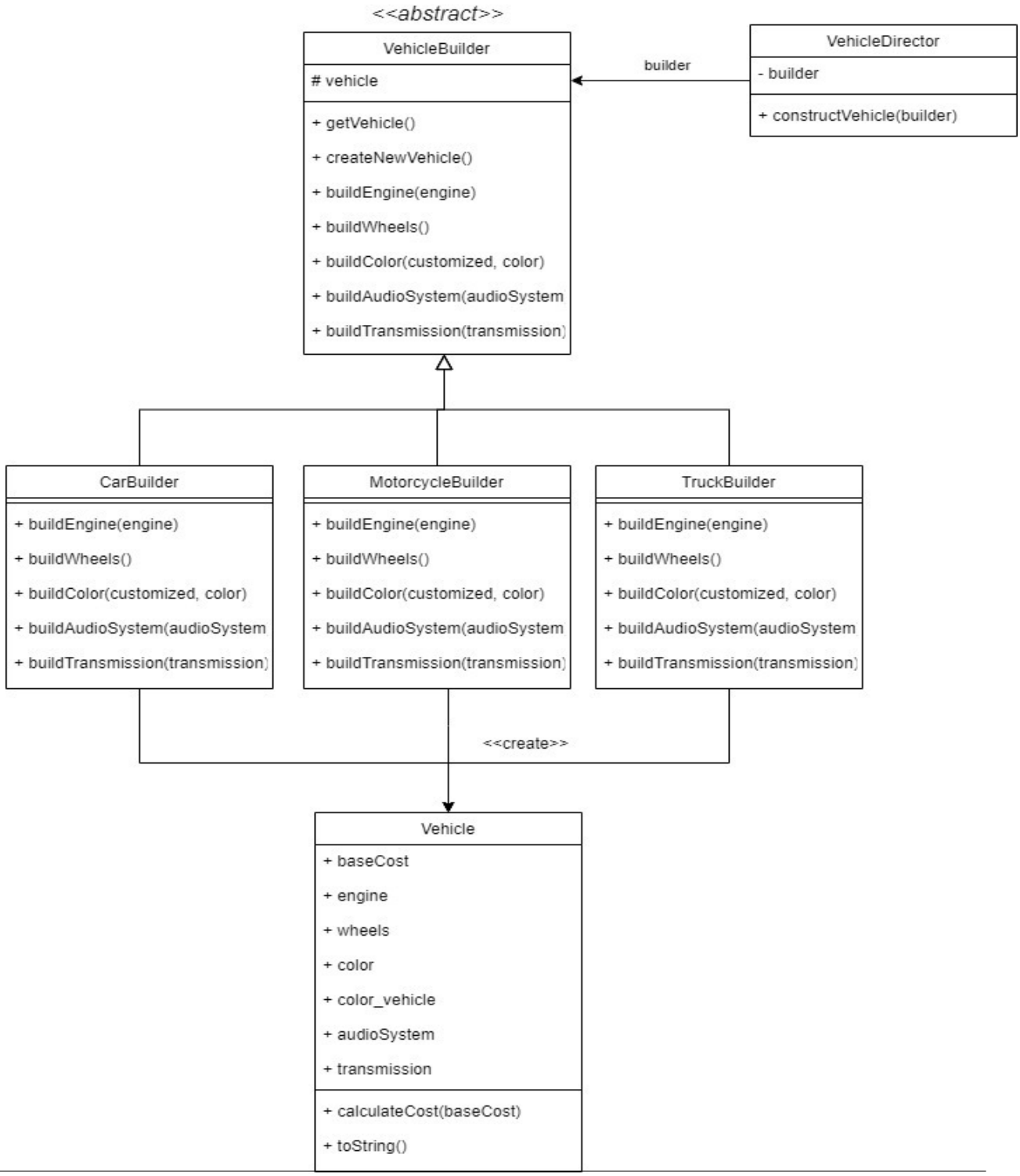
3. Ejercicio 3

3.1. Enunciado

Diseño e implementación de una aplicación que utilice un patrón de diseño libre. Hemos diseñado una aplicación que crea vehículos, desde sus ruedas y motor hasta la transmisión y el color. Para ello, hemos decidido usar el patrón de diseño *Builder*.

- Usar Patrón Builder o algún Patrón que NO sea creacional para no repetir.
- Se valorará complejidad del problema y solución acordada en grupo.
- Extra: Combinación de patrones.

3.2. Diagrama UML



3.3. Resolución

Empezamos creando la clase abstracta *VehicleBuilder*, que define un contrado para construir vehículos, y nos permitirá contruir objetos complejos paso a paso. Contiene un atributo **vehicle** de tipo Vehicle, que representa el vehículo que se está construyendo. Los métodos que implementa son los siguientes:

- *getVehicle()*: Método para obtener el vehículo construido. Devuelve el vehículo construido.
- *createNewVehicle()*: Método para crear un nuevo vehículo. Crea una nueva instancia de Vehicle.
- *buildENgine(engine)*: Método abstracto para construir el motor del vehículo. El parámetro *engine* indica el tipo de motor.
- *buildWheels()*: Método abstracto para construir las ruedas del vehículo.
- *buildColor(customized, color)*: Método abstracto para contruir el color del vehículo. El parámetro *customized* indica si el color es personalizado o no. El parámetro *color* indica el color del vehículo.
- *buildAudioSystem(audioSystem)*: Método abstracto para construir el sistema del vehículo. El parámetro audioSystem indica si el vehículo tiene sistema de audio o no.
- *buildTransmission(transmission)*: Método abstracto para construir la transmisión del vehículo. El parámetro transmission indica si el vehículo tiene transmisión o no.

La clase *VehicleDirector* actúa como el director en el patrón *Builder*. Coordina el proceso de contrucción de un vehículo utilizando un **Vehicle Builder** específico. Contiene el atributo *builder*, de tipo VehicleBuilder, que se utilizará para construir el vehículo. Los métodos que implementan son:

- *VehicleDirector(builder)*: Constructor de la clase. EL parámetro *builder* es el constructor de vehículos a utilizar.
- *constructVehicle()*: Método para construir el vehículo. Coordina el proceso de construcción de un vehículo utilizando el constructor de vehículos especificado.

La clase *CarBuilder* es una implementación concreta de *VehicleBuilder* que se utiliza para contruir vehículos de tipo automóvil. Los métodos que implementa son:

- *CarBuilder()*: Constructor de la clase. Crea una nueva instancia de *CarBuilder* y llama al método *createNewVehicle()* para inicializar el nuevo vehículo.
- *buildEngine(engine)*: Método para contruir el motor del automóvil. Devuelve el tipo de motor.
- *buildWheels()*: Método para construir las ruedas del automóvil.
- *buildColor(customized, color)*: Método para construir el color del automóvil. El parámetro customized indica si el color es o no personalizado. El parámetro colo indica el color del automóvil.
- *buildAudioSystem(audioSystem)*: Método para construir el sistema de audio del automóvil. El parámetro audioSystem indica si el automóvil tiene un sistema de audio o no.
- *buildTransmission(transmisión)*: Método para construir la transmisión del automóvil. El parámetro transmission indica si el automóvil tiene o no transmisión.

La clase *MotorcycleBuilder* es una implementación concreta de *VehicleBuilder* que se utiliza para contruir vehículos de tipo motocicleta. Los métodos que implementa son:

- *MotorcycleBuilder()*: Constructor de la clase. Crea una nueva instancia de *MotorcycleBuilder* y llama al método *createNewVehicle()* para inicializar el nuevo vehículo.
- *buildEngine(engine)*: Método para contruir el motor de la motocicleta. Devuelve el tipo de motor.
- *buildWheels()*: Método para construir las ruedas de la motocicleta.
- *buildColor(customized, color)*: Método para construir el color de la motocicleta. El parámetro customized indica si el color es o no personalizado. El parámetro colo indica el color de la motocicleta.
- *buildAudioSystem(audioSystem)*: Método para construir el sistema de audio de la motocicleta. El parámetro audioSystem indica si la motocicleta tiene un sistema de audio o no.
- *buildTransmission(transmisión)*: Método para construir la transmisión de la motocicleta. El parámetro transmission indica si la motocicleta tiene o no transmisión.

La clase *TruckBuilder* es una implementación concreta de *VehicleBuilder* que se utiliza para contruir vehículos de tipo camión. Los métodos que implementa son:

- *TruckBuilder()*: Constructor de la clase. Crea una nueva instancia de *TruckBuilder* y llama al método *createNewVehicle()* para inicializar el nuevo vehículo.
- *buildEngine(engine)*: Método para contruir el motor del camión. Devuelve el tipo de motor.
- *buildWheels()*: Método para construir las ruedas del camión.
- *buildColor(customized, color)*: Método para construir el color del camión. El parámetro customized indica si el color es o no personalizado. El parámetro colo indica el color de la motocicleta.
- *buildAudioSystem(audioSystem)*: Método para construir el sistema de audio del camión. El parámetro audioSystem indica si el camión tiene un sistema de audio o no.
- *buildTransmission(transmisión)*: Método para construir la transmisión del camión. El parámetro transmission indica si el camión tiene o no transmisión.

La clase *Vehicle* representa un vehículo y proporciona métodos para calcular su costo y representarlo como una cadena de texto. Los atributos que posee son:

- **baseCost**(double): Indica el costo base del vehículo.
- **engine**(string): Indica el tipo de motor del vehículo.
- **wheels**(int): Indica el número de ruefas del vehículo.
- **color**(boolean): Indica si el color del vehículo es personalizado (true) o no (false);
- **color_vehicle**(string): Indica el color de vehículo.
- **audioSystem**(boolean): Indica si el vehículo tiene sistema de audio (true) o no (false);
- **transmisión**(boolean): Indica si el vehículo tiene transmisión personalizada (true) o no (false);

Los métodos que implementa son:

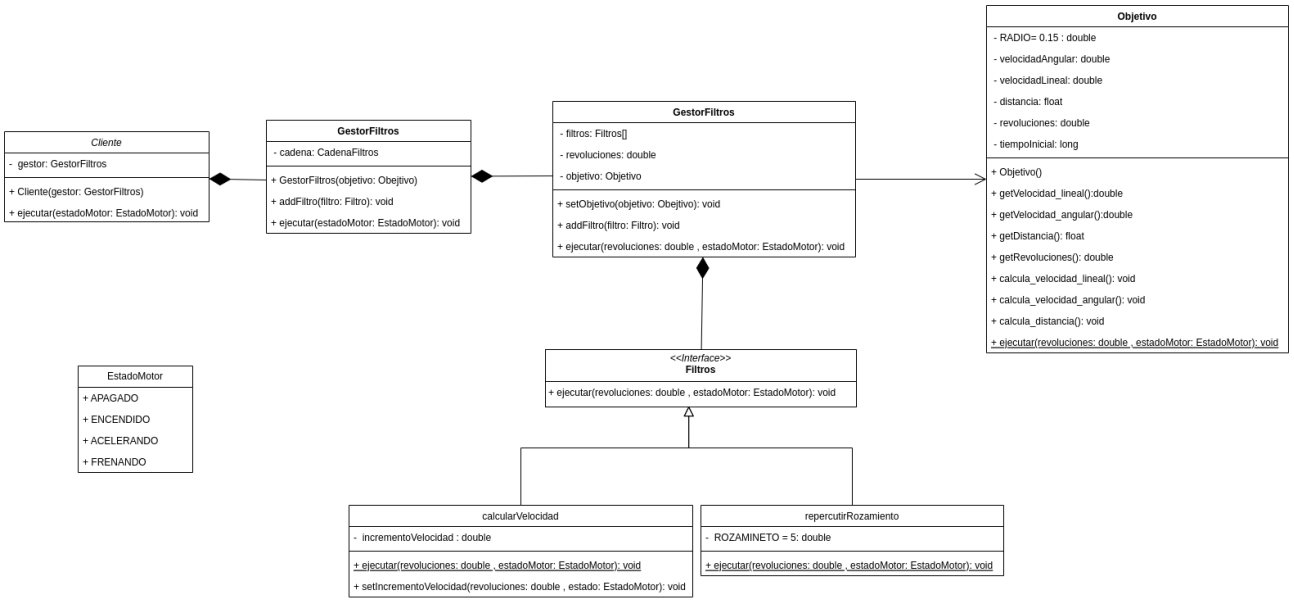
- *calculateCost(baseCost)*: Método para calcular el costo total del vehículo. El parámetro **baseCost** indica el costo base del vehículo. Devuelve el costo total del vehículo, incluido los costos adicionales por características personalizadas.
- *toString()*: Método para representar el vehículo como una cadena de texto. Devuelve una cadena de texto que representa al vehículo.

La clase ***main*** es la clase principal del programa, que interactúa con el usuario para configurar y construir vehículos personalizados.

El método pedirá al usuario que elija el tipo de vehículo. A continuación, solicitará información específica del vehículo, como el tipo de motor, el color, la transmisión y el sistema de audio. Finalmente, construye el vehículo, muestra la información dle vehículo y calcula su costo total.

4. Ejercicio 4

4.1. Diagrama UML



4.2. Resolución

Empezaremos creando la clase *Cliente*, que representará un cliente que utiliza un **GestorFiltros** para establecer las revoluciones del motor apartir de un estado que se le pase. La clase tiene el atributo **gestor**, que indica el gestor de filtros utilizado por el cliente. Los métodos de la clase son:

- *Cliente(GestorFiltros g)*: Constructor de la clase que recibe un gestor de filtros como parámetroy lo asigna al atributo **gestor**.

- *ejecutar(EstadoMotor estadoMotor)*: Método que ejecuta los filtros en el motor utilizando el gestor de filtros asignado. El parámetro que recibe es el estado actual del motor.

La clase *GestorFiltros* gestiona una cadena de filtros y proporciona métodos para agregar filtros y ejecutarlor en un motor. Tiene un atributo **cadena** que indica la cadena de filtros suministrada por el gestor. Define los siguiente métodos:

- *GestorFiltros(Objetivo o)*: Constructor de la clase que recibe un objetivo como parámetro sobre el que se aplicarán los filtros y que se pasará a **CadenaFiltros**.
- *addFiltro(Filtro f)*: Agrega un filtro a la cadena de filtros suministrada por el gestor.
- *ejecutar(EstadoMotor estadoMotor)*: Ejecuta la cadena de filtros en el motor. El parámetro *estadoMotor* indica el estado actual del motor.

La clase *CadenaFiltros* gestiona una lista de filtros y proporciona métodos para agregar filtros, establecer un objetivo y ejecutar los filtros en un motor. Los atributos de la clase son:

- **filtros**: Una lista de filtros.
- **objetivo**: El objetivo al que se aplicarán los filtros.
- **revoluciones**: Las revoluciones actuales del motor.

Los métodos que implementa son:

- *addFiltro(Filtro f)*: Agrega un filtro a la lista de filtros.
- *setObjetivo(Objetivo o)*: Establece el objetivo al que se aplicarán los filtros.
- *ejecutar(EStadoMotor estadoMotor)*: Ejecuta la lista de filtros en el motor y, opcionalmente, ejecuta el objetivo.

La clase interfaz *Filtro* define el método *ejecutar(double revoluciones, EstadoMotor estadoMotor)* para ejecutar filtros en un motor, donde **revoluciones** son las revoluciones actuales del motor y **estadoMotor** el estado actual del motor, y que devuelve el resultado de aplicar el filtro (double);

La clase *FiltroCalcularVelocidad* implementa la interfaz *Filtro* y proporcionar métodos Para calcular la velocidad de un motor. Los atributos de la clase son:

- **incrementoVelocidad**: Parámetro que indica el cambio de velocidad.
- **MAX_REVOLUCIONES**: Representa el máximo de revoluciones permitidas antes de ajustar la velocidad.

Los métodos que implementa son:

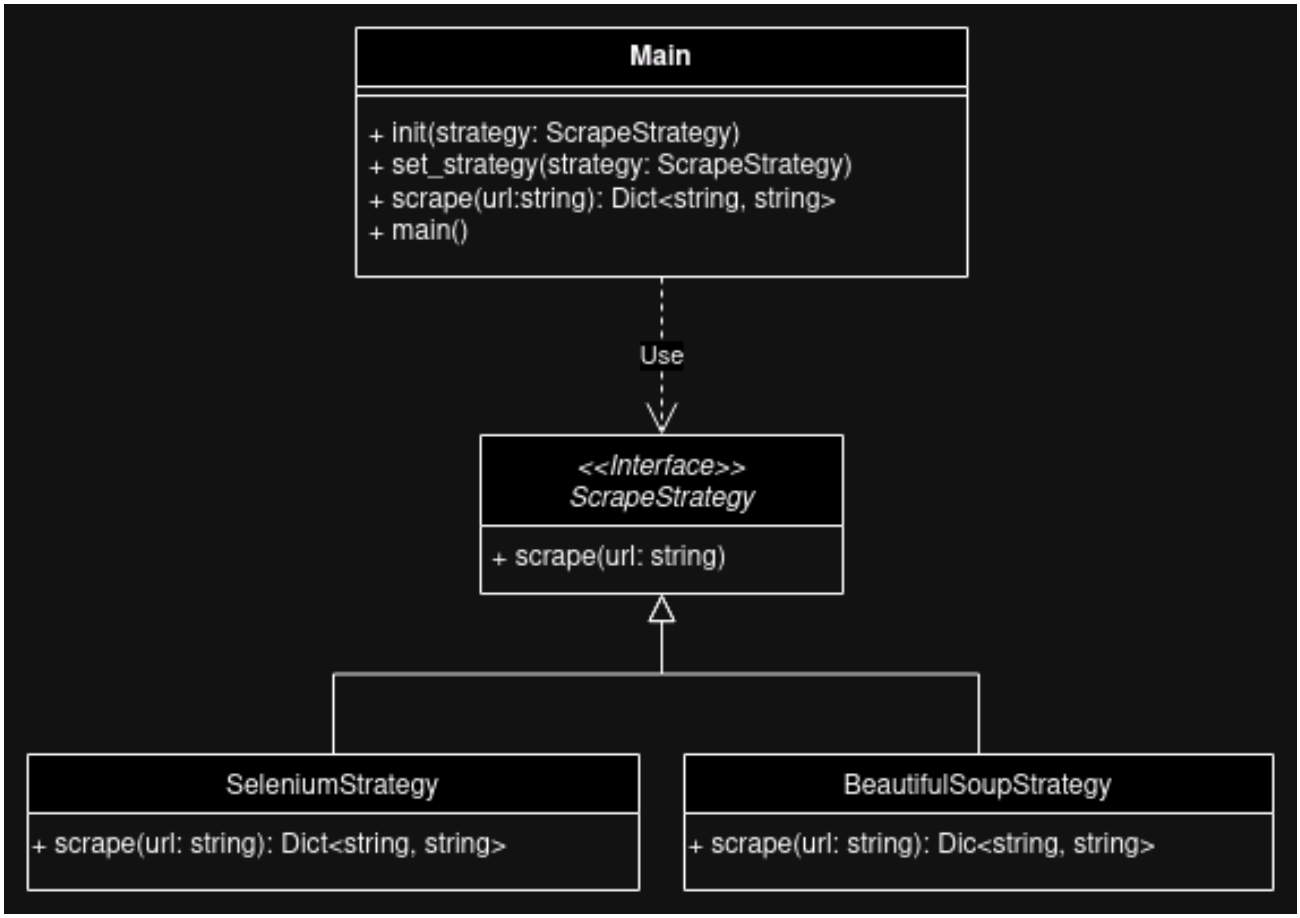
- *setIncrementoVelocidad(double revoluciones, EstadoMotor estado)*: Establece el incremento de velocidad según las revoluciones y el estado del motor, en caso de que las revoluciones sean mayores que **MAX_REVOLUCIONES** hace que el **incrementoVelocidad** sea 0.
- *ejecutar(double revoluciones, EstadoMotor estadoMotor)*: Calcula la velocidad del motor en función de las revoluciones y el estado del motor, y devuelve nuevo valor para las revoluciones de motor. En caso de ser menor que 0, devuelve **revoluciones** = 0.

5. Ejercicio 5

5.1. Enunciado

Diseño de una aplicación de WebScraping en Python donde se utilice el Patrón Strategy para scrapear información en vivo de acciones en la página web ***https://finance.yahoo.com/quote/***, donde se puede añadir el sufijo de la acción para acceder a información en vivo de ésta.
Ejemplo con TESLA: ***https://finance.yahoo.com/quote/TSLA***.

5.2. Diagrama UML



5.3. Resolución

Empezamos con la clase interfaz *Scraper*, que contendrá la definición de la función *scrape(self, url)*, que estará implementada en las clases hijas *Selenium* y *BeautifulSoup*.

La clase *BeautifulSoup* implementa el método *scrape(self, url)*, que se encarga de acceder a la url que le pasamos como parámetro. Realizamos un GET a la url. Verificamos si tuvimos exito en la operación si el código de estado es 200 (OK). Si tuvimos exito, procedemos a estraer los datos de la página web, de lo contrario,

devuelve un mensaje de error diciendo que no pudimos obtener la página. Si la solicitud fue exitosa, creamos un objeto BeautifulSoup y procedemos a encontrar los elementos buscados con el método de búsqueda *find()*.
Los datos serán almacenados en un diccionario, que será devuelto como resultado de la función.

La clase *Selenium* implementa el método *scrape*, que se configura para utilizar el navegador Chrome en modo ***sin cabeza*** (headless), para que el navegador se ejecute en segundo plano sin mostrar interfaz gráfica, mejorando la velocidad en el proceso de scrapeo.
Luego el navegador accede a la página de las acciones de Tesla en Yahoo Finance utilizando la url y espera hasta que el botón de aceptar cookies sea detectable para hacer el click. Después de esto, utilizando la clase *WebDriverWait*, se establece un periodo de espera explícito para cada elemento que queremos leer mediante su **XPATH** específico. De esta manera se asegura que el script haga una espera adecuada para que los elementos estén presentes en la página antes de intentar acceder a sus datos.
Una vez extraidos los datos se cierra el navegador limpiamente para asegurar que no queden procesos ejecutándose en segundo plano.

En la clase ***main*** crearemos un dato llamado **url**, que contendrá la url de la página web que queremos scrapear. Llamaremos al constructor de la clase, que recibe una estrategia. Luego, llamaremos al método *scrape(url)*, que llamará al método *scrape(url)* del objeto estrategia, y almacenará el resultado en la variable **data**, que contendrá el diccionario de los elementos buscados.
Por último, imprimiremos los valores encontrados y los añadiremos a un archivo json llamado ***data.json***.