UNIVERSIDAD DE GRANADA

Escuela Técnica Superior de Ingenierías Informática y Telecomunicación

Desarrollo de Software

Práctica 1: Uso de patrones de diseño creacionales y estructurales en OO

Autores:
Álvaro Ruiz Luzón
Luis Miguel Guirado Bautista
Miguel Bravo Campos
Miguel Ángel Serrano Villena
Ginés Torres Almagro

24 de marzo de 2024

Índice

1. Patrón Factoría Abstracta en Java

1.1. Análisis y extracción de requisitos

1.1.1. Descripción

Programar la simulación de 2 carreras simultáneas con el mismo número inicial (N) de bicicletas. Se usarán hebras. 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. Todas las bicicletas se retiran a la misma vez.

Se deberán de seguir las siguientes especificaciones:

- Se implementará el patrón de diseño Factoría Abstracta junto con el patrón de diseño Método Factoría[1].
- Se usará Java como lenguaje de programación.
- Se utilizarán hebras para que las carreras se inicien de forma simultánea.
- Se implementarán las modalidades montaña/carretera como las dos familias/estilos de productos.
- 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, 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 heredarán de FactoriaCarreraYBicicleta. 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] Dado que Java no permite que un método devuelva varias cosas, no ha sido posible implementar el Método Factoría junto con Factoría Abstracta. En el ejercicio 2 sí ha sido posible debido a la flexibilidad que otorga el lenguaje de programación de alto nivel Python a causa de ser dinámicamente tipado

1.1.2. Requisitos funcionales

- El programa tiene que realizar dos carreras de bicicletas.
- Cada carrera tiene un número **N** de bicicletas participando, el cual se computa aleatoriamente en cada ejecución del programa justo antes de que den comienzo.
- En la carrera de montaña se retira el 20 % de las bicicletas antes de que llegue a su fin.
- En la carrera de carretera se retira el 10 % de las bicicletas antes de que llegue a su fin.

1.1.3. Requisitos no funcionales

- El programa tiene que realizar las dos carreras simultáneamente (es decir, hay que utilizar la concurrencia, manejando dos hebras).
- Ambas carreras deben durar exactamente 60 segundos.
- Tras obtener un ganador en la carrera, las bicicletas de esta carrera en concreto se retirarán a la vez.
- Se tienen que implementar las carreras como hebras.

1.1.4. Requisitos de información o similares

- El programa mostrará por pantalla los inicios y finales de ambas carreras, junto a las identificaciones de las bicicletas participantes. Es decir, aparecerá un trazado de los eventos que se van dando a lo largo de las carreras, con el objeto de simularlas de forma textual.
- Al final, se anuncia el ganador de cada carrera.

1.2. Diseño del programa

El programa se ha diseñado utilizando el patrón de diseño Factoría Abstracta[2]. Para ello, se han creado tres paquetes para aislar cada parte relevante.

El primero de ellos es "factoria", el cual alberga la interfaz "FactoriaCarreraYBicicleta", que se encargará de declarar los métodos que llevarán a cabo la creación de los objetos necesarios para desarrollar las carreras de bicicletas. También hay dos factorías concretas que implementarán la interfaz anterior, que se encargarán de crear los objetos necesarios para cada tipo de carrera ("FactoriaMontana" para las carreras y bicicletas de montaña y "FactoriaCarretera", para las carreras y bicicletas de carretera).

El segundo paquete es "carrera", el cual alberga lo relativo a las carreras. Aparece una clase abstracta "Carrera", la cual representará a las carreras, con

[2]Ver el diagrama UML asociado en la página siguiente.

una serie de métodos comunes y varios abstractos, como es el caso de "iniciarCarrera" y "finalizarCarrera", que serán implementados por las clases hijas "CarreraMontana" y "CarreraCarretera", que representan los tipos específicos de carreras. Es importante mencionar también que Carrera implementa la interfaz Runnable de Java, lo que permitirá la ejecución posterior de N carreras a la vez, pasando las clases hijas mencionadas a darle implementación al método "run" para tal efecto; y, por último, el paquete "bicicleta", el cual alberga lo relativo a las bicicletas.

Aparece una clase abstracta "Bicicleta", la cual representará a las bicicletas, con una serie de métodos comunes y otro abstracto, como es el caso de "avanzar", que será implementado por las clases hijas "BicicletaMontana" y "BicicletaCarretera", que representan los tipos específicos de bicicletas.

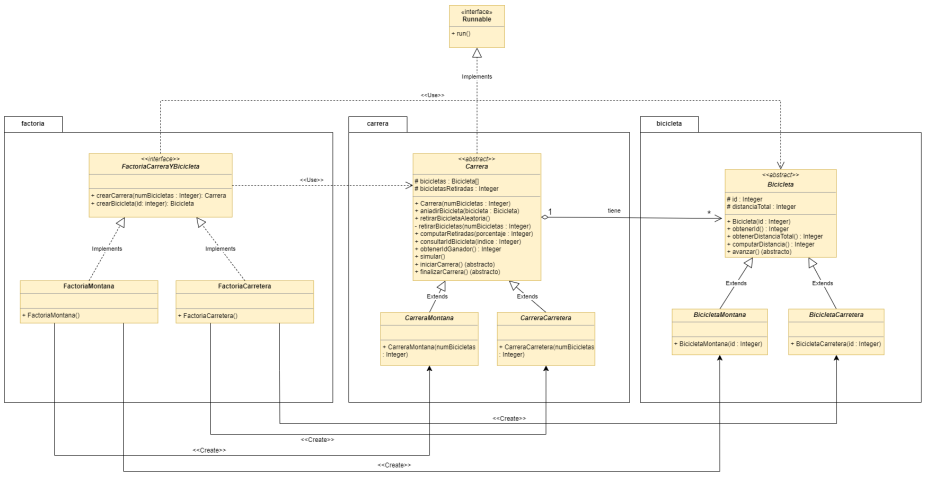


Figura 1: Diagrama UML Ejercicio 1

1.3. Implementación

Tras haber descrito y mostrado el esquema general de todo el diseño, se pasa a describir el funcionamiento general de este programa con el objeto de ilustrar la idea que persigue un patrón como el de Factoría Abstracta:

- Se computa el número de bicicletas que participarán en las carreras de forma aleatoria.
- Se crean las factorías concretas para manejar cada tipo de carrera.
- Se crean las carreras específicas con su factoría asociada.
- Se añaden las bicicletas a cada carrera con su factoría asociada.
- Se crea un pool de dos hilos (uno para cada carrera).

- Se lanzan las dos carreras.
- Se termina el programa.

Se puede concluir que las factorías ayudan a abstraerse de la creación de los objetos específicos de cada tipo de carrera, evitando en muchos casos, además de posibles errores por el manejo de muchos objetos agrupados en familias.

Finalmente, cabe comentar que, en un primer momento, se optó por hacer que las bicicletas fueran también hebras, para dar más dinamismo a las carreras, pero esto lo haremos en el ejercicio 2 como aspecto diferenciador entre ejercicios similares. El diagrama de flujo asociado a lo anteriormente comentado es el que sigue:

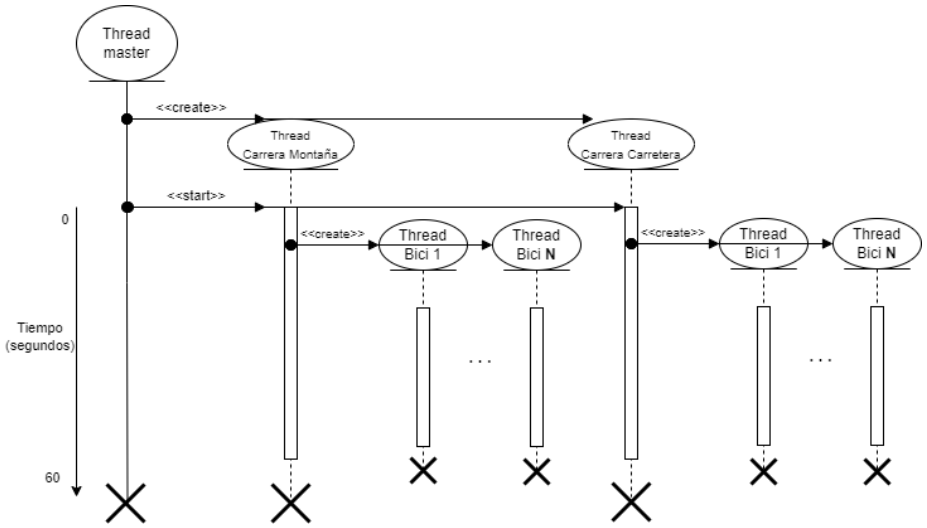


Figura 2: Diagrama de flujo Ejercicio 1

2. Patrón Factoría Abstracta + Patrón Prototipo (Python)

2.1. Análisis y extracción de requisitos

La descripción del problema y su análisis es muy similar respecto al anterior. Las diferencias son las siguientes:

- Como requisito no funcional se ha implementado como hebras tanto las bicicletas como las carreras.
- Como requisitos de información cabe destacar que se imprimirá por pantalla tanto el inicio y fin de las carreras, junto con el avance de cada bicicleta de cada carrera junto con su identificador.
- Se informará cuando se vayan a retirar las bicicletas, así como cuáles han sido retiradas junto con su id y la carrera a la que pertenecían.
- Finalmente se mostrará al ganador en ambas carreras junto con su id

2.2. Diseño del programa

El programa se ha diseñado con la unión de tres patrones: Factoría Abstracta, Prototipo y Método Abstracto. Vamos a ir explicando cómo hemos implementado el código para seguir con estos diseños[3].

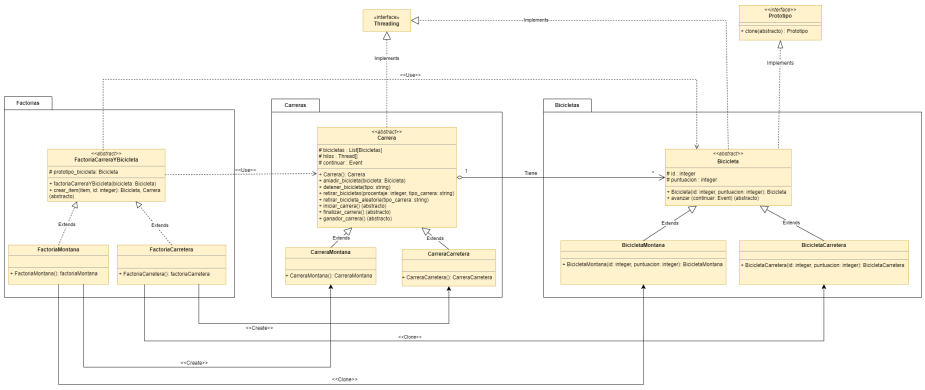


Figura 3: Diagrama UML Ejercicio 2

En primer lugar nos centramos en el patrón **factoría abstracta**, para ello, utilizamos la clase FactoríaBicicletasYCarreras cuyo objetivo es crear tanto carreras como bicicletas. Como hay dos tipos de carreras y bicicletas (montaña y carretera) se requiere de dos factorías hijas que heredan de la clase padre. Disponemos FactoríaCarretera y FactoríaMontana, donde se crean bicicletas y

[3]Ver el diagrama UML asociado.

carreras de Carretera, y de Montaña respectivamente. De esta forma conseguiremos separar la creación de la composición y representación de los objetos de carreras y bicicletas.

En segundo lugar explicaremos cómo hemos implementado el patrón **prototipo**. Hemos creado una interfaz llamada prototipo que contiene el método clonar de la que hereda la clase abstracta Bicicleta, donde es implementada.

Solamente hace falta que la herede Bicicleta, ya que se necesita un objeto prototipo, creado previamente, para clonar y así crear otros objetos. Como solo habrá una carrera de cada tipo, pero más de una bicicleta en cada carrera, tiene sentido que únicamente herede Bicicleta esa función. Esta función es usada en las clases del paquete factorías cuando van a crear una bicicleta.

Las Factorías tienen un objeto prototipo de Bicicleta creado previamente, luego al crear las bicicletas simplemente tiene que hacer que el prototipo llame a la función clonar (heredada de la interfaz Prototipo) para obtener una nueva Bicicleta y proceder a actualizar las instancias de la nueva Bicicleta.

En último lugar desarrollaremos el uso del patrón **método abstracto**. En principio no estaba detallado en el ejercicio dos; sin embargo se decidió aplicarlo ya que era compatible con los dos patrones mencionados anteriormente.

Para llevar a cabo este patrón se "unió" los dos métodos que tenían FactoríaMontana y FactoríaCarretera, crearBicicleta y crearCarretera en uno solo llamado crear_item. En este método abstracto heredado de FactoríaBicicletasYCarreras realiza la misma función que los otros dos. La única distinción es que se crea un objeto u otro en función de un parámetro en forma de cadena texto, puede tener dos valores 'bicicleta' o 'carretera'.

2.3. Implementación

Explicaremos brevemente cómo ha sido implementado, centrándonos en el uso de los hilos y en la gestión de las carreras.

En primer lugar, se crearon las carreteras y bicicletas que eran añadidas a las listas de Bicicletas de las carreras. Al mismo tiempo, se creaban los hilos correspondientes a esas bicicletas y se guardaban en el array de Hilos. La función a ejecutar por estos hilos sería 'avanzar' en la clase Bicicleta, que permite a las bicicletas avanzar esperando un segundo entre cada avance.

Al crear los hilos de las bicicletas, se les pasaría como argumento el atributo 'continuar', que pertenece a la clase Carrera y es de tipo evento, con valor true. Este valor permitirá avanzar a los hilos de bicicleta cuando la carrera esté activa. Cuando se termina, se actualiza el valor a false, simulando el final de la carrera; entonces, los hilos de las bicicletas terminarían su ejecución, es decir, su avance.

Para conseguir que las carreras actúen como hilos, se utilizó el siguiente código:

```
hilo_x = Thread(target=carrera_x.iniciar_carrera)
```

x representa tanto montaña como carretera.

Tras crear estos dos hilos, se inicia su ejecución ejecutando la función iniciar_carrera. En ella, se ajusta un temporizador de 60 segundos.

```
temporizador=threading.Timer(60.0,lambda:self.detener\
    ↪ _bicicletas(x))
```

x representa tanto montaña como carretera.

Después, las carreras van poniendo en ejecución a sus hilos de bicicleta para que avancen. Tras pasar 60 segundos, se ejecuta la función detener_bicicletas, donde se actualiza el atributo continuar a false, luego se bloquea el avance de las bicicletas (termina la ejecución de los hilos de las bicicletas), ya que pasaron 60 segundos y terminó la carrera. Tras esto, se retiran un porcentaje de bicicletas en función del tipo de carrera con la función detener_bicicletas. Finalmente, se terminan los hilos de las carreras. Anuncian que ha finalizado la carrera y muestran al ganador, que será quien haya conseguido más puntos.

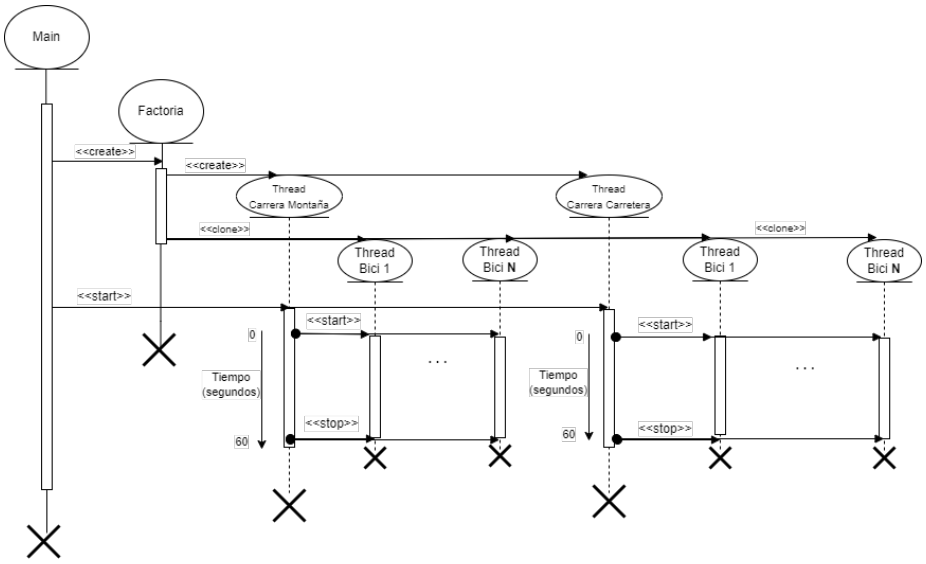


Figura 4: Diagrama de flujo Ejercicio 2

3. Patrón libre estudiado en teoría en Python/-Java/Ruby (a elegir)

3.1. Diseño del programa

El programa consiste en una fábrica de bicicletas y se ha diseñado con la unión de dos patrones de diseño[4]: *Builder*, el cual es un patrón creacional, que permite construir objetos complejos paso a paso; y *Decorator*, el cual es un patrón estructural, que permite añadir extras a objetos colocando estos dentro de objetos encapsuladores.

Se ha separado el diseño en cuatro paquetes, los cuales contienen toda la funcionalidad necesaria para combinar los mencionados patrones de diseño:

- **Paquete "bicicleta"**: contiene el producto que se quiere construir. En este caso particular, se trata de bicicletas. Hay una superclase abstracta "Bicicleta" que representa a una bicicleta genérica, de las cuales derivan tres, aunque solo dos de ellas están dentro de este paquete, como se podrá entender más adelante. Estas son "BicicletaMontana" y "BicicletaCarretera", las cuales representan a una bicicleta de montaña y una bicicleta de carretera, respectivamente.

- **Paquete "constructor"**: contiene la funcionalidad necesaria para aplicar el patrón Builder. Cuenta con una superclase abstracta "Constructor", la cual cuenta con una serie de métodos necesarios para realizar la construcción de las bicicletas genéricas. Para las bicicletas específicas que se han descrito anteriormente, existen otras clases constructoras que derivan de la anterior, como es el caso de "ConstructorBicicletaMontana" y "ConstructorBicicletaCarretera". Además, aparece otra clase llamada "ConstructorBicicletaDecorada", que se encargará de agregar decoración a las bicicletas de los tipos anteriores (este constructor se relaciona ya con el otro patrón que se utiliza en este ejercicio, como es el patrón *Decorator*).

- **Paquete "director"**: contiene una clase interesante para dotar al patrón Builder de la máxima abstracción en lo referente a la construcción de las bicicletas. Esta clase se llama "Director", y su razón de ser está en encapsular los pasos del ensamblado para construir los diferentes tipos de bicicletas.

- **Paquete "decorador"**: contiene una superclase abstracta "DecoradorBicicleta", que cuenta con unas características muy peculiares, las cuales son clave a la hora de integrar la funcionalidad del patrón *Decorator*. Estas características son que deriva de "Bicicleta" y tiene una instancia de "Bicicleta" a su vez como atributo. Esto permitirá agregar capas sobre la bicicleta de base que tendrá almacenada como atributo protegido. Además,

[4] Ver el diagrama UML asociado en la página siguiente.

aparecen dos clases más, “DecoradorBicicletaConEstampado” y “DecoradorBicicletaConFunda”, que sirven para aplicar decoraciones específicas a las bicicletas que contengan (estampado y funda, respectivamente).

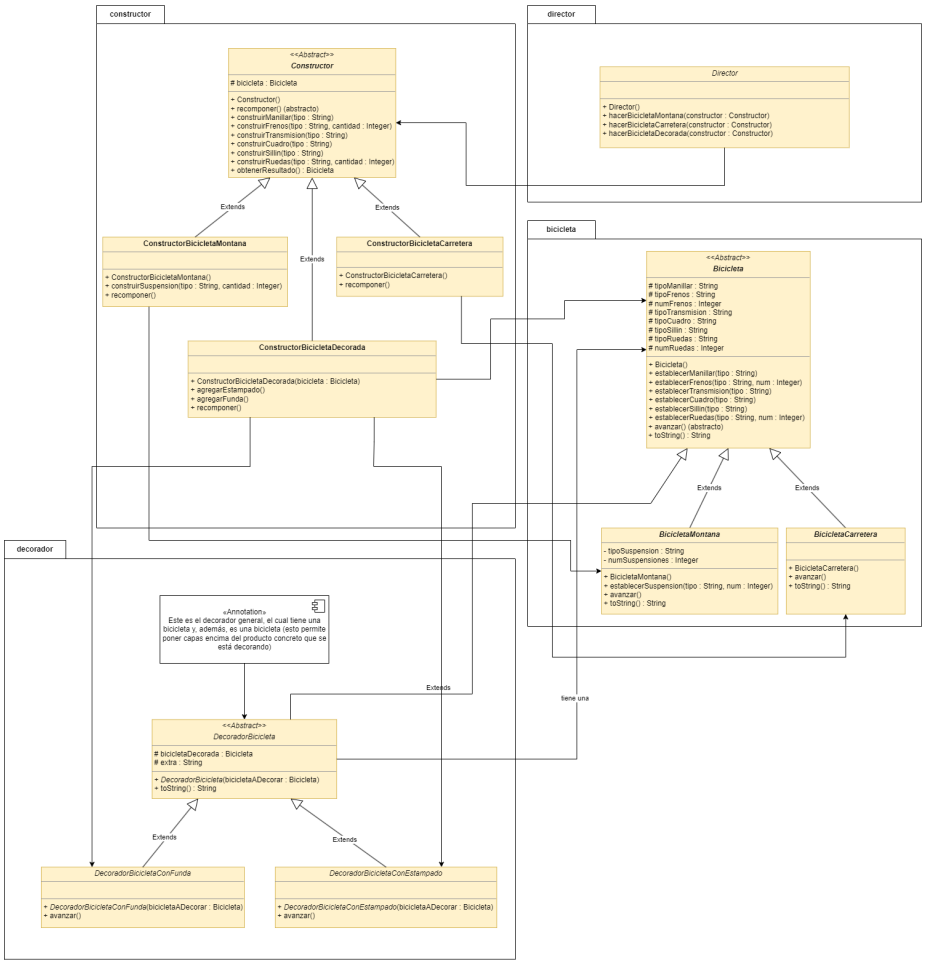


Figura 5: Diagrama UML Ejercicio 3

3.2. Implementación del programa

La idea del programa principal es ilustrar esa mezcla de patrones de una manera lo más natural posible. La descripción de este es bastante ilustrativa y consigue justificar la existencia de todo lo descrito en el subapartado anterior de diseño, por lo que pasamos a detallarla a continuación.

En primer lugar, se crea una instancia del director, de manera que todas las construcciones que se realicen se hagan a través de él. Tras esto, se procede a construir una bicicleta de montaña sin más, para lo cual se crea un constructor de ese tipo de bicicleta, para, acto seguido, llamar al método del director específico para el ensamblado de este tipo de bicicleta, pasándole su constructor como parámetro. Tras la ejecución completa del método anterior, el resultado final, es decir, la bicicleta comentada ya terminada, estará almacenada en el atributo heredado del constructor base. Para obtenerla, basta con llamar al método de su constructor para tal efecto, almacenándola en una variable.

Una vez contamos con la bicicleta de montaña, queremos decorarla con un estampado y cubrirla con una funda para que no se dañe. Para ello, basta con crear un constructor para tal efecto (el que hay para decorar bicicletas), pasándole como parámetro la bicicleta de base sobre la que se quiere trabajar (la bicicleta de montaña anterior) y, acto seguido, utilizar de nuevo el director para llamar al método correspondiente para decorar bicicletas, pasándole el constructor que se acaba de crear para realizar la decoración. Tras la ejecución completa del método anterior, el resultado final, es decir, la bicicleta comentada ya terminada, estará almacenada en el atributo heredado del constructor base. Para obtenerla, basta con llamar al método de su constructor para tal efecto, almacenándola en una variable.

Finalmente, también se construye una bicicleta de carretera para ilustrar que no hay diferencias en la forma de construirla, pues se ha conseguido abstraer todo el proceso de ensamblado independientemente del tipo de bicicleta de la que se trate, así como el tipo de decoración o decoraciones que se quieran añadir. Es por ello que la bicicleta de carretera se construye siguiendo los mismos pasos descritos anteriormente para la bicicleta de montaña.

Para más detalle adjuntamos este siguiente diagrama de secuencia como apoyo a esta explicación anterior:

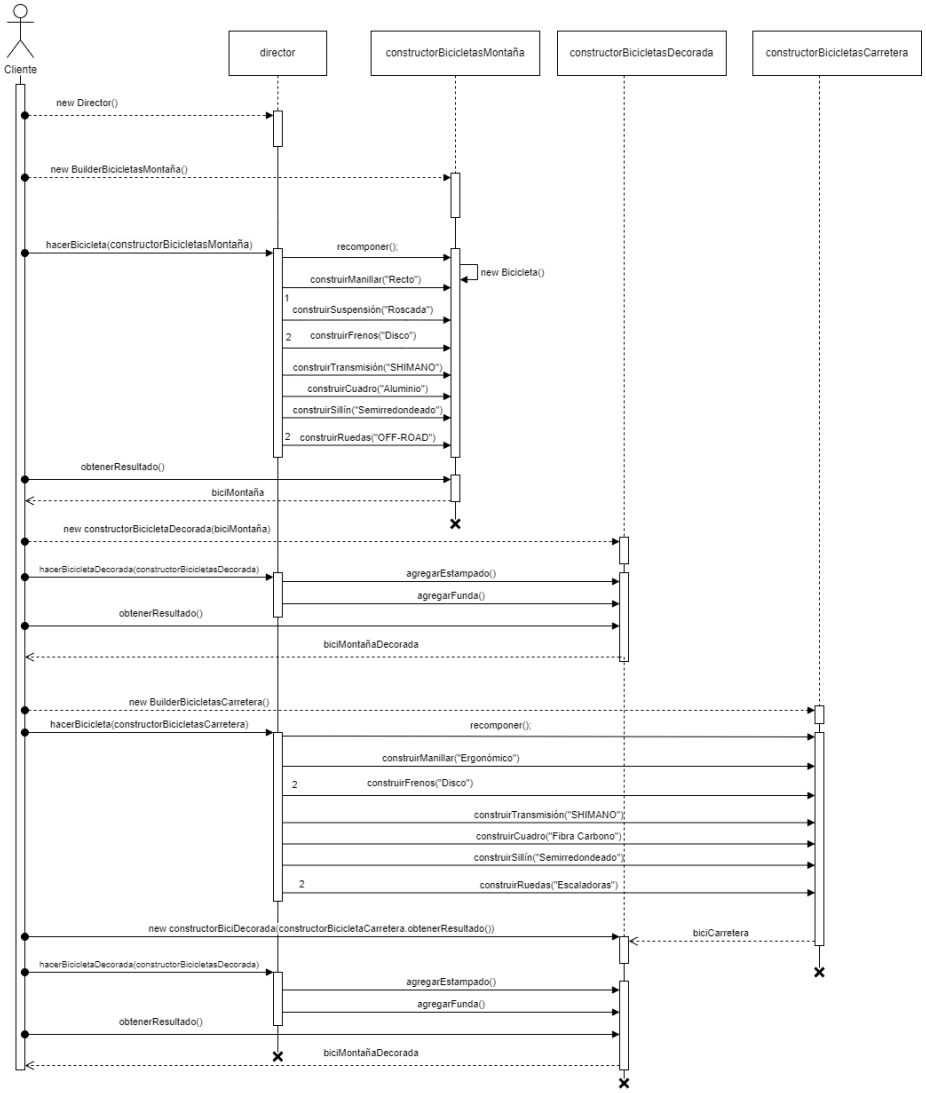


Figura 6: Diagrama Secuencia Ejercicio 3

3.3. Aspectos peculiares de este diseño

En el diseño descrito para este ejercicio aparecen algunos aspectos que pueden generar dudas. De cara a despejar esas dudas, pasamos a explicar dichos aspectos:

- **"ConstructorBicicletaDecorada"** sabe de la existencia de los decoradores específicos: esto es así porque los métodos que alberga necesitan utilizar internamente instancias de estos decoradores, puesto que, no olvidemos, son bicicletas también.
- **"ConstructorBicicletaDecorada"** recibe en su constructor de clase una "Bicicleta" como parámetro, mientras que los otros dos constructores no la reciben: esto es así porque otorga una mayor flexibilidad, por permitir que cualquier tipo de bicicleta pueda ser decorada, sin discriminar por el tipo específico, como sí ocurre en las otras dos clases constructoras específicas.
- Los constructores se han de pasar a los métodos del director que encapsulan el ensamblado de las bicicletas, no devolviendo estos el resultado, sino el propio constructor: esto es así porque forma parte de las características del patrón Builder, ya que la idea es que los constructores se encargue de todo el proceso de construcción, en el que también se encuentra devolver la bicicleta resultante.
- **"DecoradorBicicleta"** es una Bicicleta y tiene una Bicicleta: esto es así para permitir añadir capas a una bicicleta de base, de manera que los estados anteriores de cada bicicleta queden sin modificar. Esto puede permitir incluso que se puedan deshacer esas capas para volver a estados previos en las fases de construcción.

3.4. Notas finales

Cabe mencionar que los métodos del director pasan por defecto los parámetros necesarios para crear los tipos de bicicletas contemplados. Esto es así simplemente por una cuestión de comodidad, pues basta para ilustrar el funcionamiento del programa. Una opción más elaborada si se quisiera mejorar este programa, sería pasarle a estos métodos como parámetros todas las especificaciones de los componentes de las bicicletas según el tipo. Además, por supuesto, se pueden añadir más tipos de bicicletas, de decoradores y de combinaciones posibles de decoraciones en forma de métodos de la clase "Director".

4. Patrón (estilo) arquitectónico Filtros de intercepción en Python. Simulación del movimiento de un vehículo con cambio automático

4.1. Introducción

Este ejercicio consiste en diseñar e implementar un simulador de un vehículo con una interfaz gráfica de usuario.

- Tendrá un panel que representa el salpicadero con un velocímetro en dos magnitudes: RPM (revoluciones por minuto) y los KM/H; y un cuenta-kilómetros total y reciente.
- Tendrá un panel con los mandos (acelerador, freno y botón de encendido) y el estado del vehículo. El usuario podrá interactuar con el simulador a través de los mandos.
- Al interactuar con los mandos, podremos alterar el estado del vehículo, así como su velocidad y kilómetros recorridos.
- Emplearemos Python como lenguaje de programación.
- Emplearemos **ttkbootstrap** [5]

4.2. Modelo

4.2.1. Requisitos funcionales

- El usuario podrá alternar entre encender y apagar el vehículo.
- El usuario podrá alterar la velocidad del vehículo acelerando y frenando.
- El usuario no podrá acelerar ni frenar si el vehículo está apagado.
- El usuario podrá ver la velocidad angular y lineal del vehículo actuales.
- El usuario podrá ver los kilómetros recorridos totales y recientes (entre encendidos) actuales.
- El usuario podrá ver el estado del motor.

[5] Instalación: python -m pip install ttkbootstrap o con python -m pip install git+https://github.com/israel-dryer/ttkbootstrap .

También necesario instalar tkinter: apt-get install python-tk .

4.2.2. Requisitos no funcionales

- El usuario podrá interactuar con la aplicación mediante una interfaz de usuario.
- El usuario podrá alterar el estado del vehículo mediante el uso de solicitudes.
- El usuario no podrá ir a más de 5000 RPM.
- El vehículo tendrá unas ruedas con un radio de 15 centímetros.

4.2.3. Patron software empleado: Intercepting Filters

Para el modelo utilizaremos el patrón de diseño software Filtros de Intercepción. Este patrón consiste esencialmente en transformar o procesar una solicitud o entrada del usuario mediante una serie de filtros de manera secuencial. Emplea un estilo arquitectónico de flujo de datos de cauces o tuberías. Se pueden implementar fácilmente nuevos filtros y se puede modificar la secuencia de filtros de manera muy sencilla. Normalmente se suele utilizar en aplicaciones de mensajería para la censura de palabras soeces o en solicitudes HTTP.

En este caso, realizaremos "solicitudes" al vehículo que contiene todos los datos para modificar su velocidad y estado del motor.

Figura 7: Patrón de Filtros de Intercepción entre el Usuario y el vehículo.

4.2.4. Diagrama de clases

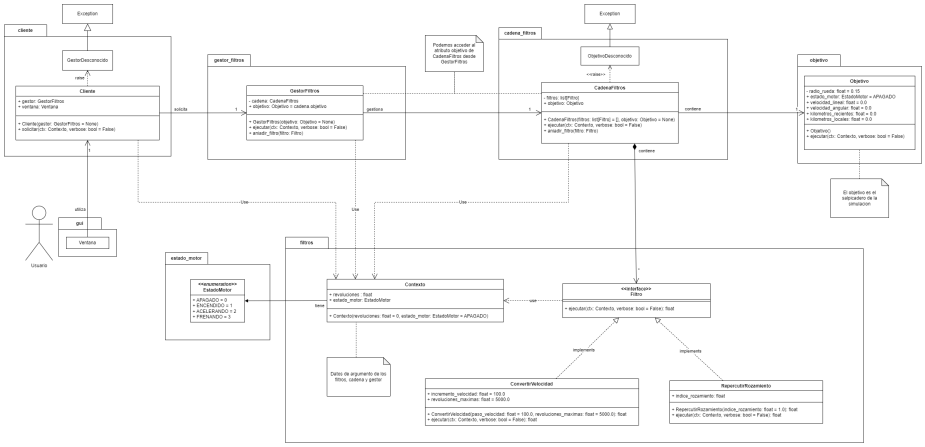


Figura 8: Diagrama UML del programa de simulación.

4.2.5. Diagrama de secuencia al realizar una solicitud

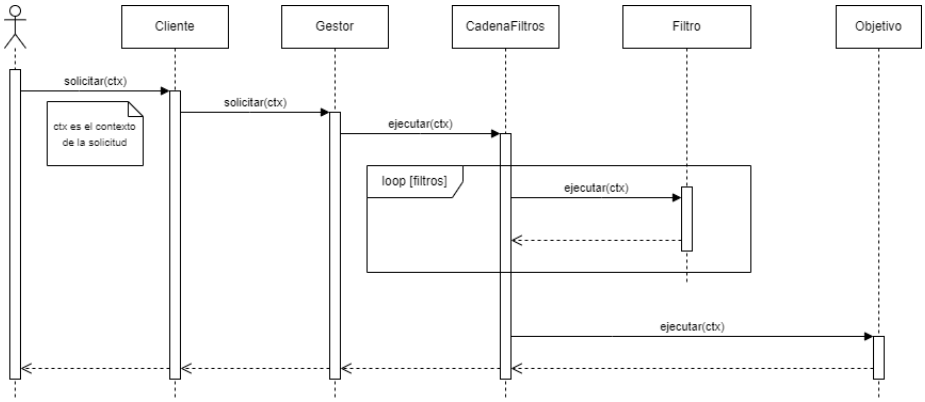


Figura 9: Diagrama de secuencia de solicitud.

El usuario realiza una solicitud como cliente con un contexto determinado, que se delega al gestor. El gestor le pasa la solicitud proporcionada por el usuario a la cadena de filtros. Una vez que la cadena ha recibido la solicitud, la procesa por cada uno de los filtros que contiene. Por último se envía la solicitud ya modificada al objetivo para que esté realice las tareas necesarias: actualizar su estado con los datos de la solicitud.

4.2.6. Descripción breve de las clases

- **Contexto**
  Estructura de datos utilizada para asignar los datos necesarios a la solicitud Está compuesta de un valor decimal que indica las revoluciones por minuto del vehículo y el estado del motor.

  - EstadoMotor:
    Numeración utilizada para indicar los posibles 4 estados del vehículo: apagado, encendido, acelerando y frenando.

- **Filtro**
  Interfaz implementada por las subclases CalcularVelocidad y RepercutirRozamiento. Todas las subclases que implementan esta interfaz tendrán que tener obligatoriamente la función 'ejecutar'.

  - CalcularVelocidad:
    Filtro que altera la velocidad del vehículo.
    Si el vehículo está acelerando, aumenta la velocidad en un valor determinado hasta el límite de 5000 RPM.
    Si el vehículo está frenando, reducirá la velocidad en el mismo valor determinado.
  - RepercutirRozamiento:
    Filtro que reduce muy ligeramente la velocidad del vehículo si este se encuentra en movimiento.

- **Cadena Filtros**
  Clase con una lista de filtros y una referencia al objetivo.
  Esta clase se encarga de que la secuencia pase por todos los filtros que contenga y, finalmente al objetivo.
  También posee una función 'ejecutar' pero no implementa la interfaz Filtro, ya que no se considera uno, hace que los filtros y el objetivo realicen sus ejecuciones.
  Se pueden añadir más filtros a la cadena con la función aniadir_filtro. La agregación se hace de manera secuencial, de manera que el filtro que se añadió el primero, será el primero en tratar con la solicitud.

- **Objetivo**
  Clase que almacena el estado del vehículo, este se compone de:

  - El radio de las ruedas
  - El estado del motor
  - La velocidad linear
  - La velocidad angular
  - Los kilómetros recorridos totales
  - Los kilómetros recorridos recientes

También posee una función 'ejecutar' que coge los valores del contexto de la solicitud y los utiliza para actualizar su estado interno. Esta función también realiza la conversión de RPM a KM/H, conoce el retardo entre solicitudes para el cálculo de los kilómetros recorridos (véase apartado 3.2) y se encarga de reiniciar los kilómetros recientes a cero si se cumple que el motor está apagado.

- **GestorFiltros**
  Clase utilizada para gestionar la cadena de filtros. Contiene una cadena de filtros. Se pueden añadir filtros a la cadena desde esta clase y realizar solicitudes. También conoce el objetivo y se puede asignar otro si el programador lo desea.

- **Cliente**
  Clase utilizada por el usuario. Contiene un gestor de filtros. Solamente se podrán realizar solicitudes desde esta clase.

4.3. Vista

4.3.1. Primer enfoque

El simulador contendrá los siguientes elementos, ordenados de manera jerárquica:

- Salpicadero
  - Velocímetro
    - Revoluciones por minuto
    - Kilómetros por hora
  - Cuentakilómetros
    - Total
    - Reciente (se reinicia cuando se apaga el motor)
- Mandos
  - Texto con el estado del vehículo
  - Acelerador (botón para mantener presionado)
  - Freno (botón para mantener presionado)
  - Mecanismo de encendido (botón "palanca")

Un mockup inicial de la interfaz de usuario, antes de que se tomara la decisión de mover el cuentarrevoluciones al velocímetro:

Figura 10: Diseño de la interfaz del simulador.

4.3.2. Consideraciones

Al ser una simulación, las solicitudes del cliente tendrán que realizarse de manera constante en tiempo de ejecución. Para no sobrecargar mucho los recursos del usuario, habrá que establecer un retardo entre solicitudes.

Emplearemos un retardo de 100 milisegundos.

Podemos ver una solicitud como si fuera un "tick" de un videojuego, una unidad de tiempo dentro de la simulación.

La función de los botones será la de modificar el contexto de la solicitud. Es importante notar que el único dato que realmente cambiaremos interactuando con los botones será el estado, la velocidad será modificada por los filtros.

El botón de encendido podrá encender el vehículo solamente si está apagado y solamente podrá apagarlo en el resto de casos

El botón para acelerar podrá acelerar el vehículo solo si se mantiene pulsado. Al dejar de pulsar, volverá al estado inicial anterior a pulsar el botón. Lo mismo se aplica para el botón de frenado.

Los botones tendrán acceso a la interfaz de usuario para que puedan reflejar los cambios en la misma.

4.3.3. Componentes de la interfaz

- **Ventana**
  La interfaz de usuario en sí. Contendrá dos componentes gráficos: el salpicadero y los mandos; así como un cliente para hacerla funcional.
- **Frames (general)**
  Los frames son componentes que pueden contener otros componentes. Todos tienen en común un pequeño título en la parte superior.
  - Primario: destinado al componente del salpicadero y de los mandos.
  - Secundario: Destinado al componente del velocímetro, el cuentakilómetros y el frame para los mandos.
  - Terciario: destinado a los visores. Los visores son frases que muestran un valor numérico.

- **Salpicadero**
  Frame primario compuesto de un velocímetro y un cuentakilómetros.
  - Velocímetro
    Frame secundario
    - RPM: Visor que muestra las revoluciones por minuto del vehículo.
    - KM/H: Visor que muestra los kilómetros por hora del vehículo.
  - Cuentakilómetros
    Frame secundario
    - Total: Visor que muestra los kilómetros recorridos en total.
    - Reciente: Visor que muestra los kilómetros recorridos desde el último encendido del motor.
- **Mandos**
  - Etiqueta para el estado.
  - Botones
    Frame secundario
    - **Botón de encendido**: botón que alterna entre estados al pulsarlo, podremos apagar y encender el vehículo usando este botón.
    - **Botón de acelerar**: botón que, al mantenerlo presionado, acelerara el vehículo.
    - **Botón de frenar**: idéntico al botón de acelerar, solo que realizará la acción contraria (frenar).

4.4. Problemas durante el desarrollo

Al realizar la siguiente cuestión: Donde deberá realizarse la conversión de la velocidad angular a velocidad lineal? En un principio pensamos que debía realizarse en el filtro cambio de la velocidad, pero luego surgió el problema de que:

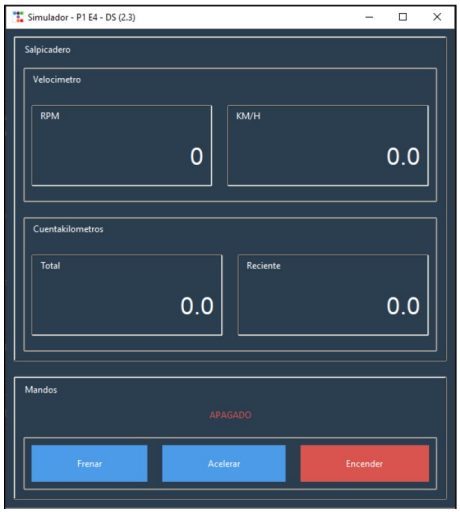
- El filtro del rozamiento se aplicaba sobre la velocidad ya calculada a linear **(km/h)**.
- El objetivo no podía saber la velocidad angular ya que había sido modificada por los filtros.

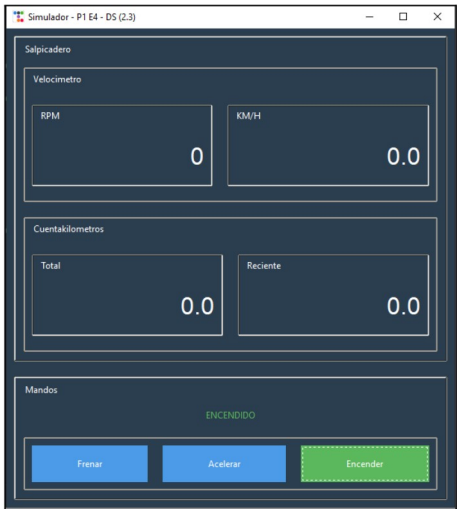
Resolvimos el problema después de releer el enunciado del ejercicio: si queremos que el objetivo sepa tanto la velocidad angular como la lineal, la conversión de la velocidad debía realizarse en el objetivo, no en los filtros. La suposición anterior, la cual fue errónea, pudo deberse a cómo debía llamarse el filtro dentro del código: CalcularVelocidad, lo que pudo haber causado un malentendido.

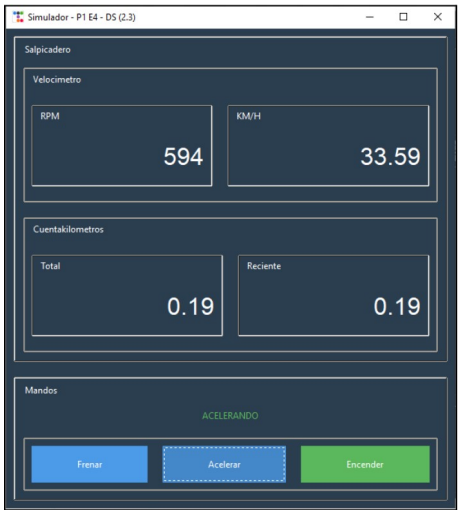
A partir de terminar el modelo, el desarrollo empezó a complicarse desde la implementación de la interfaz:

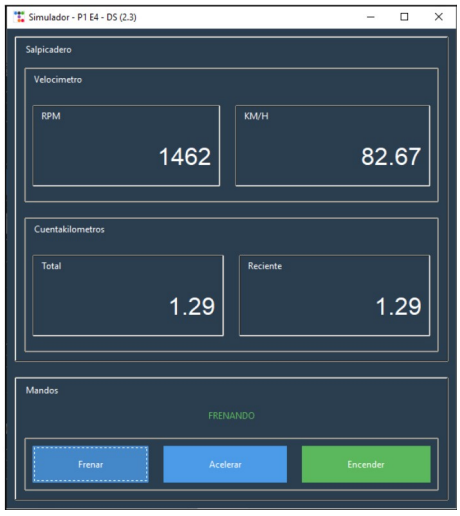
- Primero el código estaba en un solo fichero, pero luego hubo que separarlo todo mejor en un módulo con varios ficheros.
- El planteamiento sobre cómo debía comunicarse la vista con el modelo llevó más tiempo del deseado.

4.5. Capturas de la aplicación final









5. 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

5.1. Análisis y extracción de requisitos

5.1.1. Descripción

El proyecto consiste en el diseño e implementación de una aplicación de Web Scraping en Python que utiliza el Patrón Strategy para extraer información en tiempo real sobre acciones específicas de la página web de Yahoo Finanzas (https://finance.yahoo.com/quote/). Este sitio web incluye información detallada sobre acciones individuales, como Tesla (TSLA), y se actualiza dinámicamente con los últimos datos de mercado. El objetivo principal es recolectar datos sobre el precio de cierre anterior, el precio de apertura, el volumen y la capitalización de mercado de una acción específica.

Para manejar la extracción de datos de forma flexible y poder cambiar la técnica de scraping según las necesidades o restricciones del entorno, se implementará el Patrón Strategy con dos estrategias concretas: BeautifulSoup y Selenium. Esto se debe a que la página de Yahoo Finanzas utiliza JavaScript para controlar el comportamiento y la presentación de los contenidos, lo que significa que algunos datos solo están disponibles después de la ejecución de scripts en el cliente. Mientras que BeautifulSoup es eficaz para analizar el HTML estático inicial recibido del servidor, Selenium es capaz de interactuar con la página ya cargada, ejecutar JavaScript y acceder así al contenido dinámico.

La aplicación deberá ser capaz de almacenar la información scrapeada en un archivo .json, sobrescribiendo los datos anteriores con cada ejecución para reflejar la información más actual.

5.1.2. Requisitos Funcionales

- La aplicación debe permitir seleccionar entre el uso de BeautifulSoup y Selenium como estrategias para el scraping.
- Debe recuperar específicamente el precio de cierre anterior, precio de apertura, volumen y capitalización de mercado de la acción deseada.
- La información extraída se almacenará en un archivo .json, actualizándose (borrando el contenido anterior) con cada ejecución del script.
- Aunque el foco está en la funcionalidad backend, se valora una simple interfaz de usuario o un método claro para ingresar la acción deseada y seleccionar la estrategia de scraping.

5.1.3. Requisitos no funcionales

- El diseño debe permitir la adición fácil de nuevas estrategias de scraping.
- El código debe ser legible, bien organizado y fácil de mantener.
- La solución debe ser eficiente en tiempo de ejecución, especialmente importante para la estrategia que involucre Selenium, conocida por ser más lenta.
- El sistema debe manejar y reportar adecuadamente errores o cambios en la estructura del sitio web que podrían afectar la extracción de datos.

5.1.4. Requisitos de Información o similares

- La estructura del archivo .json debe reflejar claramente los datos scrapeados, siendo fácilmente interpretable.

5.1.5. Consideraciones Adicionales

Dada la dinámica de la página web de Yahoo Finanzas, que emplea JavaScript para actualizar su contenido, resulta crucial elegir la herramienta adecuada para cada situación. Como se ha mencionado, BeautifulSoup solo podrá interactuar con el contenido HTML inicial, mientras que Selenium ofrece la capacidad de ejecutar JavaScript y acceder a los datos actualizados. Esta distinción subraya la importancia de implementar el Patrón Strategy, permitiendo la selección flexible de la técnica de scraping más adecuada según el contexto.

5.2. Diseño del programa

El diseño del programa se basa en el **patrón estrategia**. Para ello, se crea una clase donde se define una familia de técnicas a usar para scrapear datos, llamada Scrape_Strategy. Es decir, permite que el cliente pueda utilizar un algoritmo u otro, en este caso, Selenium o BeautifulSoap, en función de lo que necesite. Esta elección se realiza en tiempo de ejecución mediante una interacción con el usuario. Este diseño permite que se pueda ampliar la lista de técnicas en el futuro, simplemente creando una nueva clase hija de Scrape_Strategy.

En el núcleo del diseño se encuentra la clase Contexto que tiene un algoritmo de la interfaz Scrape_Strategy. Esta interfaz tiene una función abstracta **scrape** implementadas por las clases derivadas, Beautiful_Soup_Strategy y Selenium_Strategy. Luego, promueve la mantenibilidad al permitir la modificación de las estrategias existentes sin afectar a las clases que las utilizan, ni a otros algoritmos. Ambas hacen uso del fichero de configuración que permite una fácil integración y manipulación de los datos a scrapear.

Por último, se genera un fichero .json donde se almacenan los datos scrapeados.

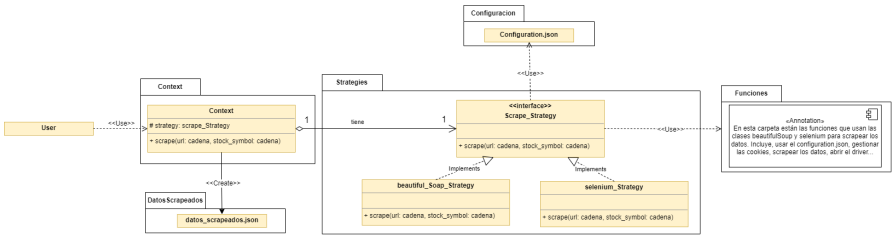


Figura 11: Diagrama UML Ejercicio 5

5.3. Implementación

En primer lugar hablamos de la clase **Contexto**, clase cuyo cometido es darle un capa de abstracción al cliente. Esta clase tiene un atributo de la interfaz **Scrape_Strategy**, y el método scrape. En este método es donde se realiza el scrapeo. Para ello, usa su atributo y llama a la función scrape. En función de que técnica haya elegido el cliente, se scrapeará con un algoritmo u otro.

Seguimos con la interfaz **Scrape_Strategy**. En ella se declara el método abstracto scrape, que será implementado independientemente por las clases derivadas.

A continuación explicaremos cada uno de los algoritmos/técnicas:

5.3.1. Beautiful Soup

BeautifulSoupStrategy: En primer lugar se realiza una carga del fichero configuration.json donde se encuentra los patrones los cuales se realizarán las búsquedas en la url especificada. Tras ello se realiza una solicitud HTTP GET a la url proporcionada y se devuelve el contenido de la página:

```
response = requests.get(url)
```

Después se crea un objeto BeautifulSoup:

```
soup = BeautifulSoup(html\_content, 'html.parser')
```

El html.parser es un analizador incluido en la biblioteca estándar de Python que BeautifulSoup utiliza para interpretar o descomponer el contenido HTML. Cuando se crea una instancia de BeautifulSoup y se le pasa html_content junto con html.parser, esta línea de código indica que se analice el contenido HTML utilizando el analizador de HTML integrado en Python.

Este analizador transforma el texto HTML crudo en un árbol de objetos Python que representa la estructura del documento HTML. De esta manera, puedes navegar por el árbol del DOM (Modelo de Objeto de Documento) y extraer la información necesaria, como elementos, atributos y texto.

A continuación se empieza a extraer los datos. Para ello se itera sobre los selectores configurados en configuration.json. En cada iteración se intenta encontrar un elemento en el HTML:

```
found\_element = soup.find(bs\_selector['tag'], bs\
    ↪ _selector['attributes'])
```

La función find sirve para buscar en el árbol de elementos HTML, y para ello toma dos argumentos. El primero que representa la etiqueta HTML que queremos buscar en este caso será td. Y el segundo contiene los atributo adicionales que permiten afinar la búsqueda, su valor es:

```
"data-test": "TD_VOLUME-value"
```

El data test es un atributo personalizado que se utiliza comúnmente para identificar elementos de la interfaz de usuario en pruebas de software automatizadas.Y el segundo es el valor asignado a este atributo personalizado.

En definitiva, la línea de código completa (found_element = soup.find(bs_selector['tag'], bs_selector['attributes'])) busca el primer elemento dentro del documento HTML procesado por soup que coincida con la etiqueta y los atributos dados. El resultado, el primer elemento encontrado que coincide con estos criterios, se asigna a la variable found_element.

En caso de que no se encuentre el elemento se aplica un selector alternativo:

```
span_element = soup.find(fallback_selector['tag'],
    ↪ text=fallback_selector['text'])
```

Se usa de nuevo la función find pero en este caso el primer argumento es una etiqueta, span, que contenga un texto, por ejemplo Volumen. Si se ha encontrado el elemento se aplica esta función:

```
sibling = span_element.find_next_sibling(
    ↪ fallback_selector['sibling'])
```

En este método busca el siguiente elemento hermano que tenga la misma estructura especificada por el argumento, en este caso, td. El término hermano se refiere a dos o más elementos que están el mismo nivel del árbol del DOM, es decir, que tengan el mismo elemento padre.

Si finalmente no se ha encontrado el elemento que buscamos se le asigna un mensaje de error.

Sino ha habido algún error generado por no encontrar el contenido de la página tras la petición url, se envían los datos al fichero **datos_scrapear.json**, y sino no se escribe nada en el fichero.

5.3.2. Selenium

SeleniumStrategy: En primer lugar al igual que en la anterior técnica se carga el fichero de configuración .json. Tras esto se crea el driver. Este es un componente que actúa como enlace entre el código de programación y el navegador permitiendo que el primero controle al segundo. En primer lugar se le debe configurar las opciones del navegador:

```
options = Options()
options.add_argument('headless')
options.add_argument("--disable-logs")
options.add_argument("--log-level=3")
```

El primer argumento significa que se ejecutará en modo sin interfaz gráfica, ya que no es necesario mostrarla. El segundo para deshabilitar los logs de Chrome puede producir durante su ejecución para mantener el output limpio. Y el último tiene el mismo propósito que el anterior, ya que solo deja mostrar mensajes de error por pantalla. Los niveles de registro suelen variar de 0 a 3, donde 0 es VERBOSE y 3 significa que solo se muestran errores.

```
driver = webdriver.Chrome(options=options)
driver.get(url)
```

Se crea una nueva instancia pasando las opciones configuradas. Obtener el driver, que es quien puede interactuar con el navegador web. Con la función get() se navega a la URL que le pasa como argumento cargando la página web en el navegador controlado por Selenium.

El siguiente paso es manejar las cookies. Se le da la opción al usuario a rechazar o aceptar las cookies mediante la interacción con la terminal. En función de su respuesta se elige una opción en la configuration.json:

```
if decision == "Y" or decision == "Yes":
        cookies_selector = config['cookies']['aceptar'
            ↪ ]
    else:
        cookies_selector = config['cookies']['rechazar
            ↪ ']
```

A continuación se aceptan o rechazan las cookies en el navegador:

```
cookies_button = WebDriverWait(driver, 10).until(EC.
    ↪ element_to_be_clickable((By.XPATH,
    ↪ cookies_selector)))
```

Estas lineas usan la clase **WebDriverWait** junto con el método until para esperar 10 segundos como máximo para que el elemento se encuentre en un estado que permita hacer click sobre él (element_to_be_clickable que hereda de ExpectedConditions(EC)), sino lanza una excepción de timeout.

Para localizar el elemento se usa By.XPATH, cookies_selector. El XPATH es una sintaxis para navegar en el árbol HTML de una página permitiendo seleccionar nodos o elementos en función del segundo parámetro, en este caso, cookies_selector. Sus valores pueden ser:

```
//button[contains(@class, 'reject-all') and @name='
    ↪ reject']

"//button[contains(@class,␣'accept-all')␣and␣@name='
    ↪ agree']"
```

Lo que hace es seleccionar todos los elementos button en todo el documento (//button). Después filtra los elementos button seleccionando aquellos cuyo atributo class contiene el texto reject-all/accept-all y su atributo name tenga valor reject/agree.

Para terminar esta explicación, una vez que el botón de cookies se ha identificado y confirmado que sea clickeable, se realiza un click sobre ese botón.

```
cookies_button.click()
```

Tras gestionar las cookies nos disponemos a buscar los datos en la página web. La idea de escrapear los datos es exactamente igual que en BeautifulSoup solo que adaptado con las funciones de Selenium. Se itera sobre cada valor a buscar con el uso del fichero de **configuration.json** y se usan estas funciones:

```
element = WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.XPATH,
                ↪ selenium_selector)) )

element = WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.XPATH,
                ↪ fallback_selector)))
```

Aparecen dos búsquedas ya que si falla la primera se intenta con la segunda. Y sino se encuentra el valor definitivamente se asigna un texto con mensaje de fallo. Expliquemos como funciona estas funciones.

Tienen una estructura similar a la ya explicada anteriormente. El único cambio es la función de condición presence_of_element_located. La condición en este caso es que esperá hasta que el elemento especificado este presente en el

DOM de la página sin necesidad de que sea visible Ya hemos explicado para que sirve XPATH, luego explicaremos lo que hay dentro de selenium_selector y fallback_selector respectivamente con un ejemplo:

```
1) "//td[@data-test='TD_VOLUME-value']"

2) "//span[contains(text(),␣'Volumen')]/following-
   ↪ sibling::td[1]"
```

Como vemos los valores tienen una semejanza con las ya explicadas con **BeautifulSoup**, ya que estamos buscando lo mismo en la misma página web. Su explicación es la misma que ya explicada en la otra técnica. Es definitiva:

En el primero busca en todo el texto (//) la etiqueta td cuyo atributo data-test sea igual a TD_VOLUMEN-value.

Y en el segundo, se busca en todo el documento (//) el elemento hermano, con la estructura td[1] (primer elemento que sea un td), de la etiqueta span cuyo texto contenga la palabra Volumen. Es decir, busca el primer td que esté al mismo nivel y que siga inmediatamente después del span encontrado.

Después de intentar extraer todos los valores deseados de la página web, si surge algún problema al acceder a la URL especificada, se generará un mensaje de error y no se realizará ninguna escritura en el archivo **datos_scrapear.json**. Por el contrario, si la operación de extracción se realiza con éxito, los datos obtenidos se almacenarán adecuadamente en dicho archivo.

Pero antes de eso se debe de cerrar las ventanas del navegador abiertas y terminar la sesión del WebDriver liberando los recursos reservados.

5.3.3. configuration.json

Comentamos que el fichero de configuración **configuration.json** se ha construido en función de la estructura HTML de la página web Yahoo Finanzas. Contiene los patrones para las cookies y los valores escrapeados distinguiendo por técnicas (Selenium y BeautifulSoup).

5.3.4. ej5opt.json

El archivo **ej5opt.json** es el punto de interacción para el usuario, donde se seleccionan la técnica de scraping que desea aplicar. A través de este archivo, se crean los objetos necesarios, y se invoca la función scrape perteneciente a la clase **Contexto**. Además, informa al usuario sobre el éxito o fracaso de la actualización de datos. En caso de éxito, los datos extraídos se almacenan en **datosScrapeados.json**; si se encuentra con algún problema, el usuario será notificado adecuadamente, y no actualiza el fichero.

5.3.5. Consideraciones

Explicaremos porque hemos decidido elegir usar **XPATH** para localizar elementos en la página web.

Escogimos ese método por su flexibilidad, permitiendo seleccionar elementos complejos y específicos dentro de una página web. Con **XPath** se puede navegar por el DOM para encontrar elementos basados en atributos, texto, y relaciones entre elementos, combina múltiples criterios, incluyendo posición y relación con otros elementos, lo que es ideal para páginas dinámicas o con estructuras cambiantes, permite búsquedas tanto absolutas como relativas, facilitando la localización de elementos en contextos variados y aprovecha patrones específicos y relaciones en el DOM, útil cuando otros identificadores son dinámicos o insuficientes.

Pensamos también en usar **selector CSS** pero por sus limitaciones dejamos de lado usarlo. Tiene capacidades limitadas para navegar por relaciones entre elementos más allá de selecciones simples de hijos o hermanos adyacentes, aunque es eficiente para encontrar elementos por clase, id o atributos, es menos versátil para navegaciones complejas o basadas en texto y no tiene la funcionalidad de localizar elementos directamente por el texto que contienen. Sin embargo, ofrece una sintaxis más sencilla y directa para casos de uso más simple, y en estos casos es más eficiente y fácil de usar.