Práctica1

Iván Gijón
Alejandro Ramos
Víctor Mojica
Gonzalo Alganza

Marzo 2024

UNIVERSIDAD
DE GRANADA

Índice

1 Ejercicio 1

1.1 Enunciado del problema

En el ejercicio 1 se plantea la resolución de un problema de carreras de bicicletas, utilizando hebras para la simulación de 2 carreras simultáneas con el mismo número de bicicletas. Se distinguen dos tipos de Carreras, de montaña y de carretera, en las de montaña se retiran el 20% de los participantes y en las de carretera el 10%. Ambas durarán exactamente el mismo tiempo, 60 segundos.

1.2 Solución

Para la resolución de dicho problema se usará el patrón Factoría Abstracta, este patrón permite que un sistema pueda ser independiente de como se crean, componen y representan sus productos.
Primero comenzamos implementando los atributos básicos de las clases sin tener en cuenta la implementación de los métodos principales.
Después, implementamos el método principal junto con los métodos necesarios para agregar bicicletas (comprobando que fueran del mismo tipo que de la carrera) y hacer que se retiraran (de manera aleatoria).

1.3 Diagrama UML

A continuación se visualiza el diagrama UML con la solución, y con el patrón factoría abstracta implementado. Además aparecen ya todos los atributos y métodos que usaremos para el correcto funcionamiento.

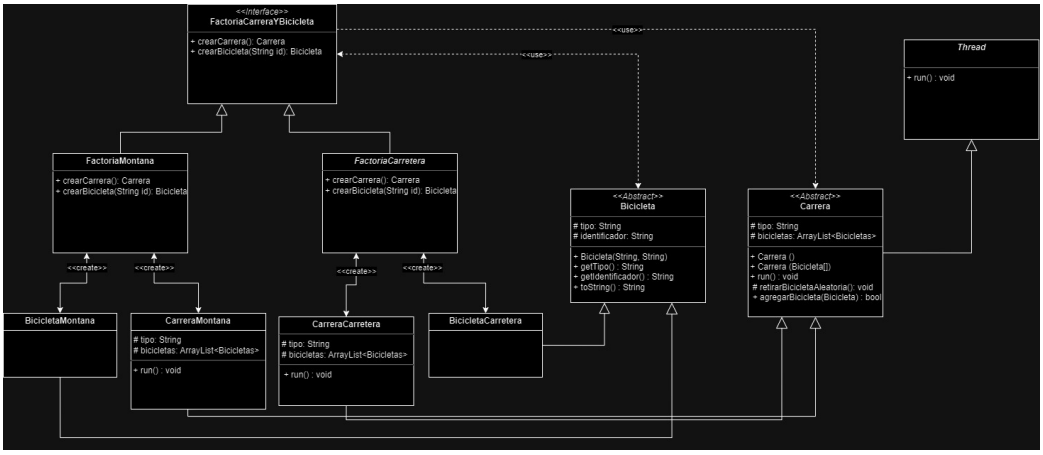

Figure 1: Imagen del diagrama de clases del primer ejercicio en Java

2 Ejercicio 2

2.1 Enunciado del problema

El enunciado del problema es exactamente el mismo que el de el ejercicio 1 a diferencia de que la implementación la hemos realizado con Python y hemos incluido el patrón Prototipo.

2.2 Solución

La solución del ejercicio es la misma que la del primer ejercicio, con la diferencia de que en este ejercicio no hemos usado hebras para las carreras y que hemos implementado el patrón Prototipo para las bicicletas, simulando que se incorporan algunas que abandonan la carrera, pero con otro ciclista (por eso se duplica la bicicleta).

2.3 Diagrama UML

El diagrama de la solución del ejercicio es casi el mismo que el de la solución del primero, a diferencia de que la sintaxis para definir variables y funciones del lenguaje es distinta a la de Java y que en este caso no hemos usado Threads y se ha añadido una función duplicar en las bicicletas para poder duplicar bicicletas para que se reincorporen a la carrera y una función

reincorporar_aleatoria para reincorporar una bicicleta aleatoria. Además, se ha agregado un array de bicicletas que se han retirado a la carrera para saber cuales se han retirado y poder reincorporarlas a la carrera.

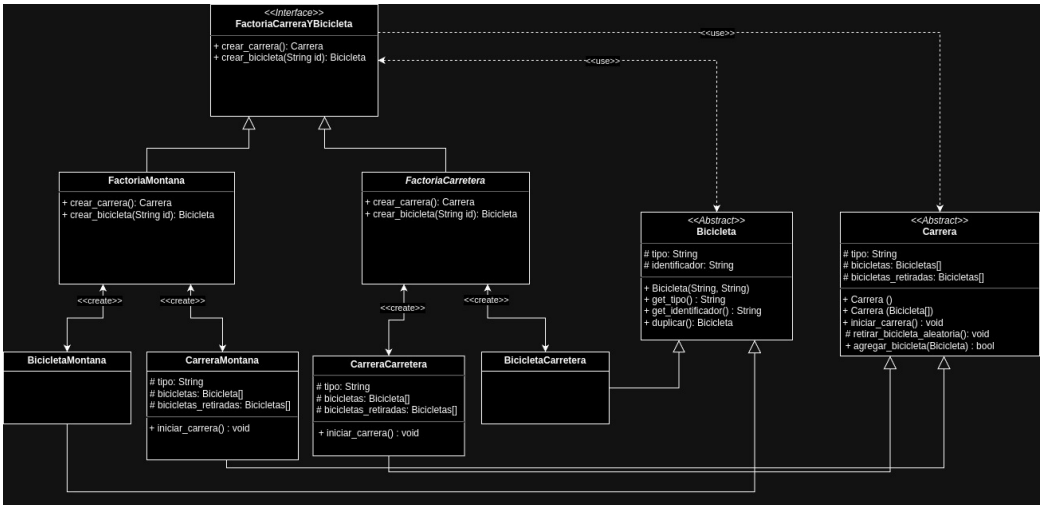

Figure 2: Imagen del diagrama de clases del segundo ejercicio en Python

3 Ejercicio 3

3.1 Enunciado del problema

En este ejercicio se pide la implantación libre de un problema usando algún patrón que no haya sido usado antes.En nuestro caso hemos decidido elegir el patrón Builder. El problema que hemos planteado es el siguiente:
Nuestro problema se basa en la creación de diferentes tipos de objetos:

- **Casa:** Existen diferentes tipos de casas que puedes crear:
  - Casa de Campo
  - Chalet
  - Apartamento

  Cada casa tiene **una cocina, una baño , una sala de estar y un dormitorio**, nosotros hemos decidido que se puede personalizar la cocina y el baño por lo que tendremos diferentes tipos que podremos crear de dichos lugares.

- **Cocina:** Una cocina contendrá varios elementos, **un horno,una vitrocerámica, un lavavajillas, y una encimera** que personalizaremos en función del tipo de cocina que se haya elegido. Tipos de cocina:
  - Cocina de lujo: Tendrá instalaciones de muy buena calidad.
  - Cocina estándar: Tendrá instalaciones asequibles en relación calidad/precio.
  - Cocina económica: Tendrá instalaciones que serán básicas y económicas.

- **Baño:** Un baño contendrá varios elementos, **una ducha,un bidé, un lavabo, y un espejo** que personalizaremos en función del tipo de baño que se haya elegido. Tipos de cocina:
  - Baño de lujo: Tendrá instalaciones de muy buena calidad.
  - Baño estándar: Tendrá instalaciones asequibles en relación calidad/precio.

  - Baño económica: Tendrá instalaciones que serán básicas y económicas.

- **Sala de Estar:** No se podrá personalizar, solo se sabe si pertenece a una casa de campo, a un apartamento o a un chalet.

- **Dormitorio:** Tampoco se podrá personalizar, solo se sabe si pertenece a una casa de campo, a un apartamento o a un chalet.

3.2 Solución

Para la solución se ha usado un Patrón Builder para Casa,otro para Cocina y otro para el Baño.Este patrón es muy útil para la construcción de objetos complejos y separa la construcción de dicho objeto de su representación. Como ya se ha dicho antes gracias a este patrón podremos crear un tipo de casa y asignarle el tipo de baño y cocina que queramos.Tendremos un Director principal que será el director de la casa en el que llamaremos a los otros Directores y construirán el tipo de baño y cocina que se pida.
Para todos los tipos de casas tendremos diferentes Builders, **ApartamentoBuilder,ChaletBuilder,CasaDeCampoBuilder** con lo que el director creara las casas. De igual manera con Baño y Cocina

3.3 Diagrama UML

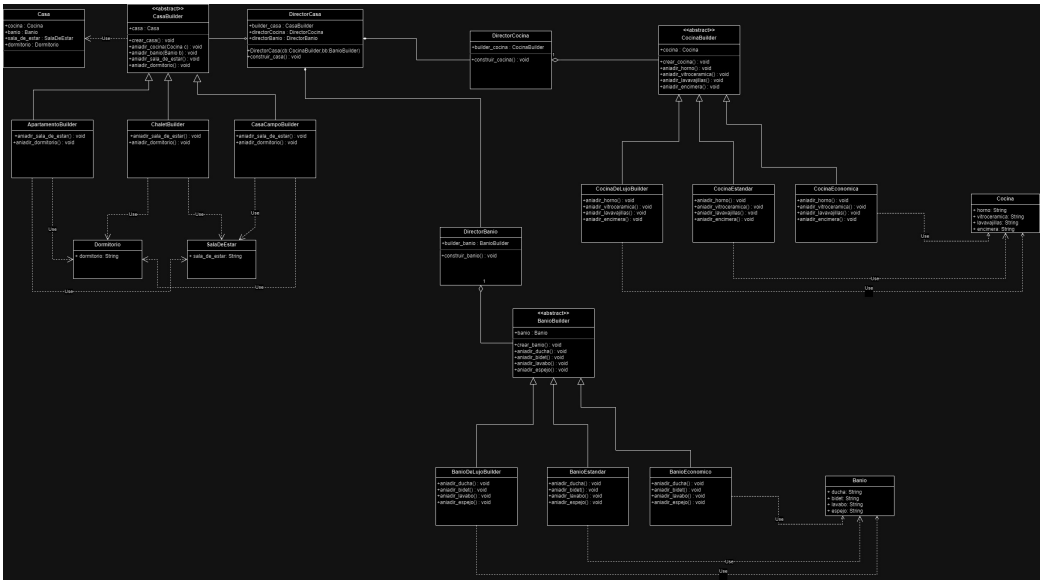

Figure 3: Imagen del diagrama de clases del tercer ejercicio en Python

4 Ejercicio 4

4.1 Enunciado del problema

Para el ejercicio 4 de la práctica se nos pide que simulemos el funcionamiento de un coche con una interfaz donde tendremos unos controles del coche y un salpicadero que nos mostrará velocidad, revoluciones por minuto y un cuentakilómetros.
Se nos pide que hagamos uso del patrón de Filtros, implementando dos filtros, uno para el cálculo de la velocidad dependiendo del estado del motor y otro para repercutir el rozamiento de las ruedas con el medio en el que está el coche. Primero se calculará la velocidad y después se le restará a la velocidad el rozamiento.

4.2 Solución

En nuestro caso hemos decidido seguir utilizando Python, pues nos dará una interfaz más bonita para nuestra aplicación y hemos aprovechado para aprender un poco cómo funciona la creación de una interfaz en python.

Para la creación de la interfaz hemos utilizado el paquete customtkinter que es un fork de el típico paquete para creación de interfaces tkinter, el cual funciona de la misma manera que tkinter solo que tiene unos estilos ya predefinidos para que nuestra aplicación parezca más "moderna".

Para la solución hemos separado la implementación en 3 tipos distintos de clases:

- **Clases:** Contiene las clases Coche (clase principal que contiene todos los elementos de la aplicación) y Motor (un enum con los estados posibles del motor del coche).
- **Filtros:** En esta carpeta hemos incluido los dos filtros distintos junto con el gestor de filtros (se encarga de gestionar los filtros y ejecutar una petición con todos los filtros) que contiene una cadena de filtros, para la cual hemos hecho otra clase en la que se comprueba si hay algún problema con un filtro a la hora de agregarlo.

- **UI:** Contiene todas las clases que hemos creado para gestionar la interfaz de usuario, cada una extiende una clase distinta de customtkinter (como puede ser un frame o un toplevel, que se usa para vincular dos ventanas distintas). Como principales están el Salpicadero y los Controles. En el salpicadero tenemos:
  - Velocímetro
  - Cuentarrevoluciones
  - Cuentakilómetros

El salpicadero actualiza las revoluciones por minuto del motor cada 500ms, pues este se actualiza cada 500ms llamando a su función *update*(), al igual que la ventana de controles.
El cuentakilómetros actualiza los kilómetros recorridos cada segundo.

4.3 Diagrama UML

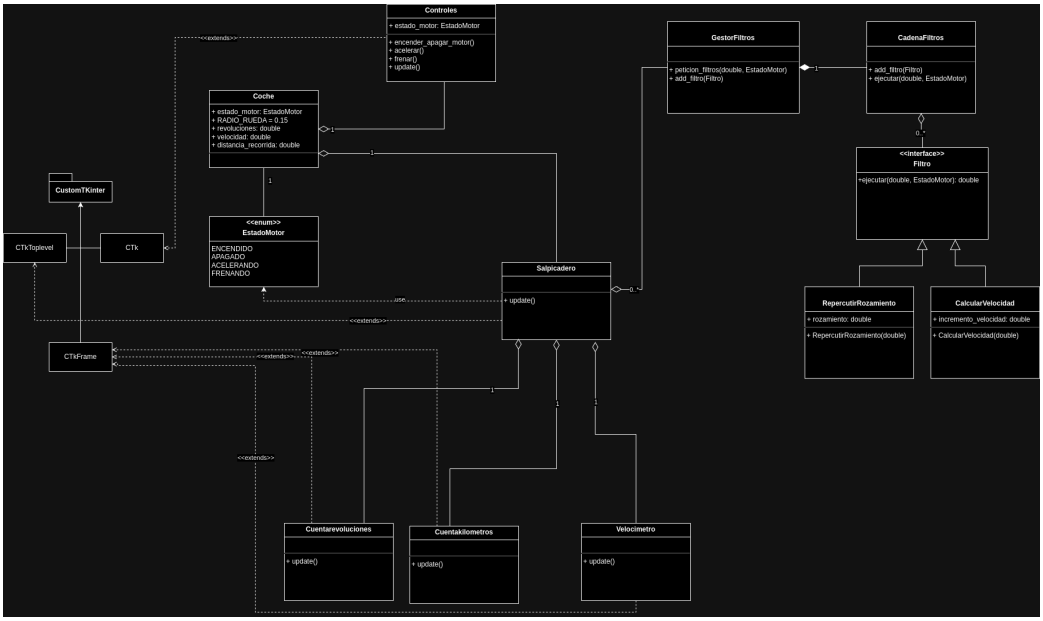

Figure 4: Imagen del diagrama de clases del tercer ejercicio en Python

5 Ejercicio 5

5.1 Enunciado del problema

En este ejercicio se pide Scrapear diferente información de la página web `https://finance.yahoo.com/` utilizando el Patrón Strategy. Vamos a utilizar dos estrategias para extraer la información de la página web.

- BeatifulSoup
- Selenium

5.2 Solución

El patrón Estrategia permite resolver un problema utilizando varios algoritmos distintos proporcionando el mismo resultado. En nuestro caso tenemos una interfaz "scrapestrategy" de la cual se implementan los dos algoritmos que vamos usar.
También tenemos el contexto, donde indicamos la estrategia que vamos a usar.

- **BeautifulSoup:** Aquí hay un resumen de lo que se ha hecho en el código:

    1. Se importan las bibliotecas necesarias, incluidas `requests` para realizar solicitudes HTTP, `BeautifulSoup` para analizar el contenido HTML, `json` para trabajar con archivos JSON y `os` para operaciones relacionadas con el sistema operativo.
    2. Se definen selectores específicos para encontrar elementos HTML en la página web que contienen los datos que queremos extraer.
    3. Se define la clase `BeautifulSoupStrategy` que implementa el método `scrape` para implementar el método de scrapear.
    4. Se buscan elementos específicos en el HTML utilizando los selectores definidos anteriormente. Si se encuentran, se extraen los datos y se almacenan en un diccionario.
    5. Los datos extraídos se escriben en un archivo JSON llamado "data-beautiful.json".

- **Selenium:** Aquí está un resumen de lo que se ha hecho en el código:

    1. Se define la clase `SeleniumStrategy`, que extiende de la clase `ScrapeStrategy`, e implementa el método `scrape` para realizar el scrapeo de datos.
    2. Se inicializa un navegador web (en este caso, Firefox) y se carga la URL proporcionada.
    3. Aceptamos las cookies, si es necesario, esperando la presencia del botón de aceptación y haciendo clic en él si está presente.
    4. Se espera a que los elementos de interés estén presentes en la página y luego se obtienen sus valores de texto utilizando los selectores CSS definidos anteriormente.
    5. Se manejan las excepciones en caso de que los elementos no se encuentren o haya un tiempo de espera excedido.
    6. Los datos extraídos se almacenan en un diccionario llamado `data` y se escriben en un archivo JSON llamado "data-selenium.json".
    7. Finalmente, se cierra el navegador web y se devuelve un mensaje indicando el éxito del proceso o cualquier error encontrado durante el raspado de datos.

5.2.1 Consideraciones y diferencias entre paquetes

Cabe destacar una gran diferencia entre BeautifulSoup y Selenium.
Selenium abre un navegador en modo bot y actua exactamente igual que un usuario, pudiendo hacer click sobre elementos o introducir información en campos de texto.
Sin embargo, BeautifulSoup lo único que hace es hacer una petición al servidor que aloja la página web y procesar la información del archivo HTML, es por eso que los valores que muestran ambas estrategias son distintos
Se puede ver en este caso puesto que la página de yahoo finance actualiza los valores de las acciones en directo haciendo uso de javascript, por lo que BeautifulSoup no tomará los valores en directo de las acciones, es por eso que puede ser que al mirar el archivo json generado por Selenium no esté actualizado a los últimos valores.

Sin embargo, como he comentado antes, Selenium ejecuta todo como un si lo abrieramos nosotros en el navegador, por lo que los datos son totalmente en directo.

5.3 Diagrama UML

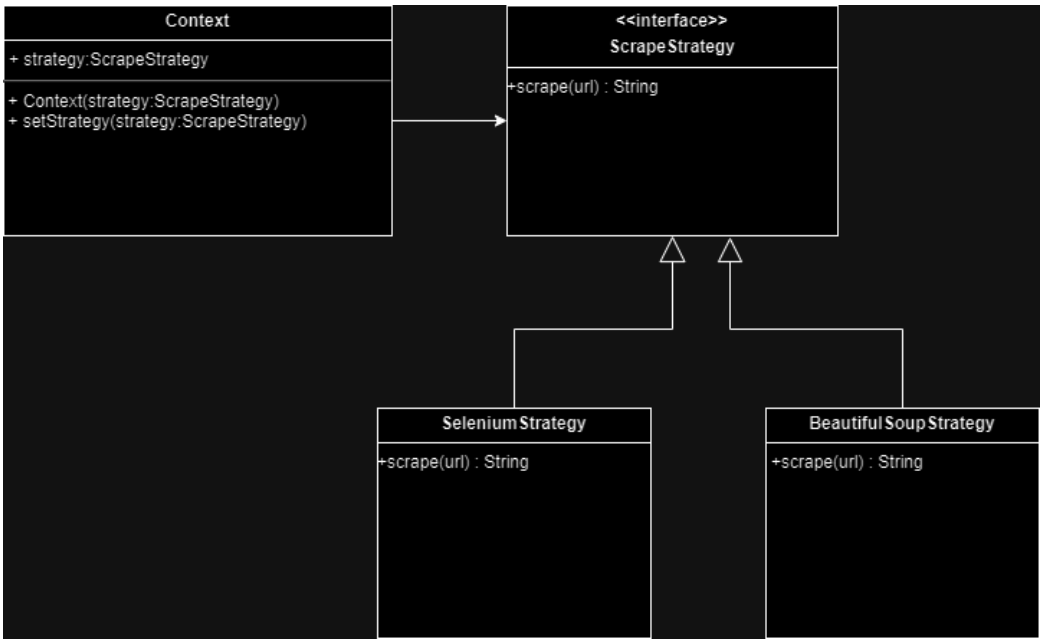

Figure 5: Imagen del diagrama de clases del quinto ejercicio