Práctica 1 Desarrollo de Software

Memoria del desarrollo de la práctica.

24 de marzo de 2024

Alonso Doña Martinez

Jose Ramón Plaza Plaza

Jaime Parra Jiménez

Carlos Izquierdo Guzmán

1. Introducción

En este documento se va a realizar la memoria sobre el proceso de realización de los ejercicios correspondientes a la práctica 1, incluyendo el ejercicio opcional. Se ha realizado la práctica utilizando Github para el control de versiones y Latex para la memoria.

2. Ejercicios

2.1. Ejercicio 1 Patrón Factoría Abstracta en Java

Para la realización de este ejercicio primero analizaremos las características/atributos que necesita cada clase a implementar siguiendo los siguientes pasos:

1. Analizamos profundamente los requisitos funcionales de cada clase.
2. Realizamos un diagrama UML que refleje las interacciones entre clases, utilizando el patrón Factoría Abstracta para resolver el problema planteado.

2.1.1. Explicación de la solución propuesta

Una vez estudiado el problema se ha realizado el diagrama UML en función del patrón Factoría Abstracta presentando las siguientes características:

- Una clase interfaz `FactoriaCarrerayBicicleta` donde se van a definir los métodos crearBicleta y crearCarrera las cuales se van a implementar en las Factorías Concretas.
- Dos clases abstractas `FactoriaMontaña` y `FactoriaCarrera` que van a ser las Factorías concretas que implementen los métodos de FactoriaCarrerayBicicleta.
- Otras dos clases abstractas Carrera y Bicicleta que van a servir de base para la implementación de los tipos de Carreras y Bicicletas.
- Dos clases que heredan de Carrera que van a definir los dos tipos concretos de Carrera que encontramos en el problema, CarreraCarretera y CarreraMontaña.
- Dos clases que heredan de Bicicleta que van a definir los dos tipos concretos de Bicicleta que encontramos en el problema, BicicletaMontaña y BicicletaCarretera.

Cabe destacar, como bien se puede observar en el diagrama, el uso en FactoriaCarrerayBicicleta de las clases Carrera y Bicicleta ,en FactoriaMontaña de CarreraMontaña y BicicletaMontaña y en FactoriaCarretera de BicicletaCarretera y CarreraCarretera.

2.1.2. Diagramas de Diseño

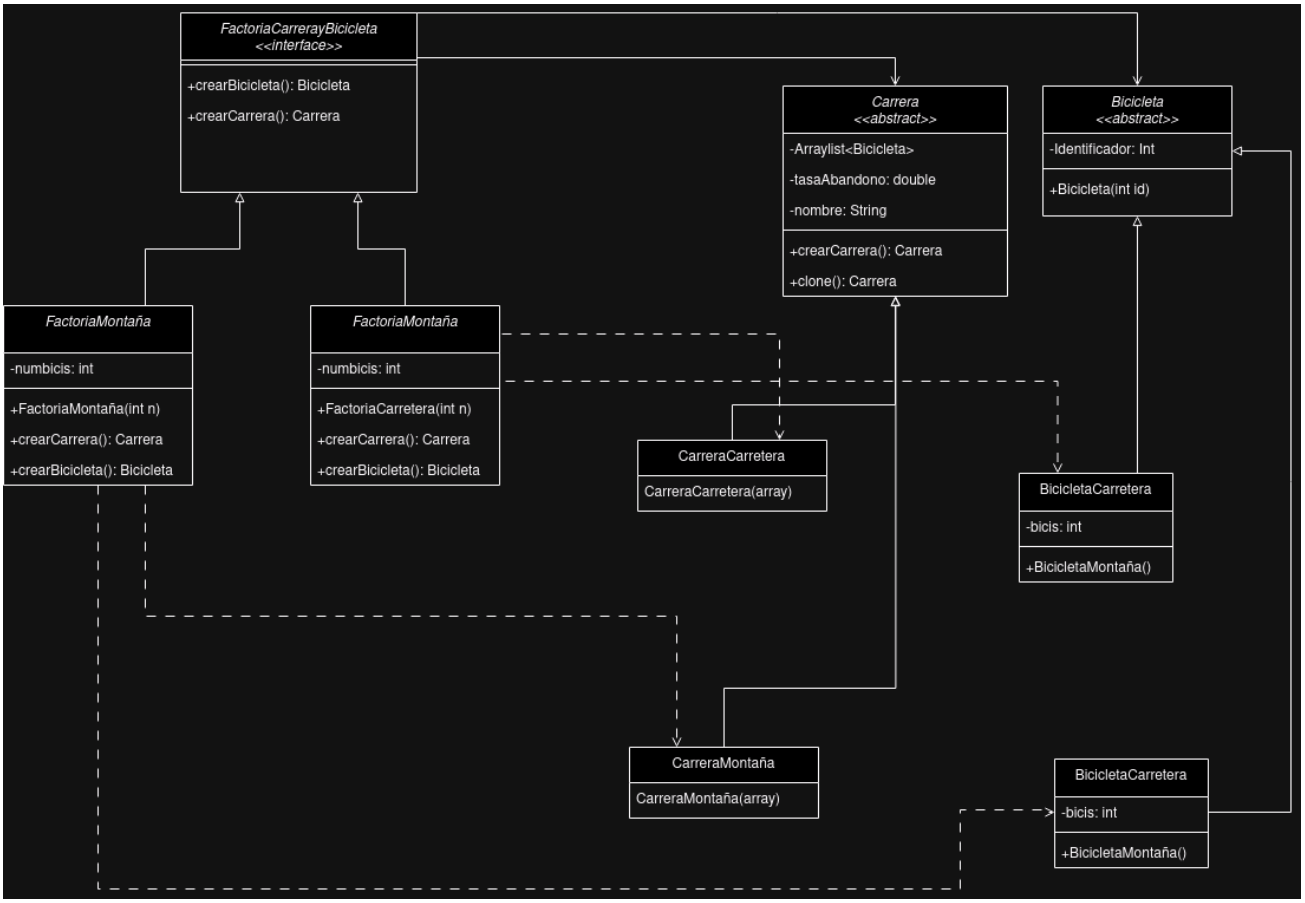

Figura 1: Diagrama Ejercicio 1

2.1.3. Código

Una vez concluido el diseño del diagrama UML, pasamos a la realización del código de cada clase. A continuación se comentarán las partes claves de cada clase definida:

FactoriaCarreterayBicicleta
Creamos la interfaz donde se realizarán las definiciones de las funciones crearBicicleta y crearCarrera.

FactoriaCarretera
En la factoria concreta implementamos su contructor, al que es necesario pasarle el numero de bicis que habrá en la carrera, y las funciones crearCarrera y crearBicicleta donde en CrearCarrera se crean el número de bicis que participarán y se van añadiendo a una lista para pasarsela como parámetro al contructor de CarreraCarretera. CrearBicicleta simplemente devuelve un objeto de BicicletaCarretera

FactoriaMontaña
Sigue el mismo procedimiento que FactoriaCarretera pero en este caso se crean BicicletasMontaña.

Carrera
En ella implementamos el constructor y realizamos los métodos mostrarCarrera que nos servirá para mostrar la información de la carrera creada y carreraTerminada, correrCarrera y run que servirán para el proceso de las hebras.

Bicicleta
Se implementa el constructor, en el se pasará como argumento un identificador.

Además este problema se basa en el uso de hebras para la creación de Carreras para esto la clase Carrera hereda de la clase Thread e implementa un metodo run que es lo que hará la hebra.

class Carrera extends Thread
public void run() En este método nososotros mostramos las carreras con las bicicletas y despues un sleep de 60 segundos.

2.1.4. Problemas Encontrados

En cuanto a los problemas planteados en la resolución de este primer ejercicio, destacan:

1. Problemas para coordinar el proyecto y configurar en cada ordenador de cada integrante la metodología de trabajo a seguir, con Github como herramienta utilizada.

2. Dificultades a la hora de representar en el diagrama UML las relaciones entre cada clase de forma gráfica.

3. Problemas para realizar el esquema y cómo conseguir unir cada factoría concreta con cada clase Carrera y Bicicleta.

Para resolver los problemas surgidos se llevo a cabo el estudio, análisis y observarción de ejemplos en los apuntes y en internet. Para el problema de GitHub, nos basamos en su documentación oficial.

2.2. Ejercicio 2. Patrón Factoría Abstracta + Patrón Prototipo (Python)

Para la realización de este ejercicio primero se pasará el codigo hecho anteriormente en java a python y tras esto seguir los siguientes pasos:

1. Implementar las funciones de copy y deepcopy para su uso posterior, en este caso se han implementado en las distintas carreras y bicicletas.

2. Utilización de las funciones deepcopy de los distintos objetos, en este caso aunque estén implementadas en las carreras y bicicletas solo utilizaremos las copias de bicicletas.

2.2.1. Explicación de la solución propuesta

La solución a este ejercicio es sencillo ya que partimos con que ya lo tenemos hecho en java, por lo que la primera tarea fue traspasar todo del lenguaje java a Python. Tras esto se implementó el metodo copy, donde este hacia copia de los distintos objetos, una vez acabado se implementó el uso de estas en las clases de carreras para copiar las bicis de la carrera. Al hacer esto y cambiar el main para poder comprobar si los metodos implementados estaban bien hechos pero al hacerlo no salió como se esperaba, por lo que al ver que la copia no era lo suficientemente profunda se implementó el metodo deepcopy, tras esto ya funcionaba correctamente la copia.

2.2.2. Diagrama de diseño

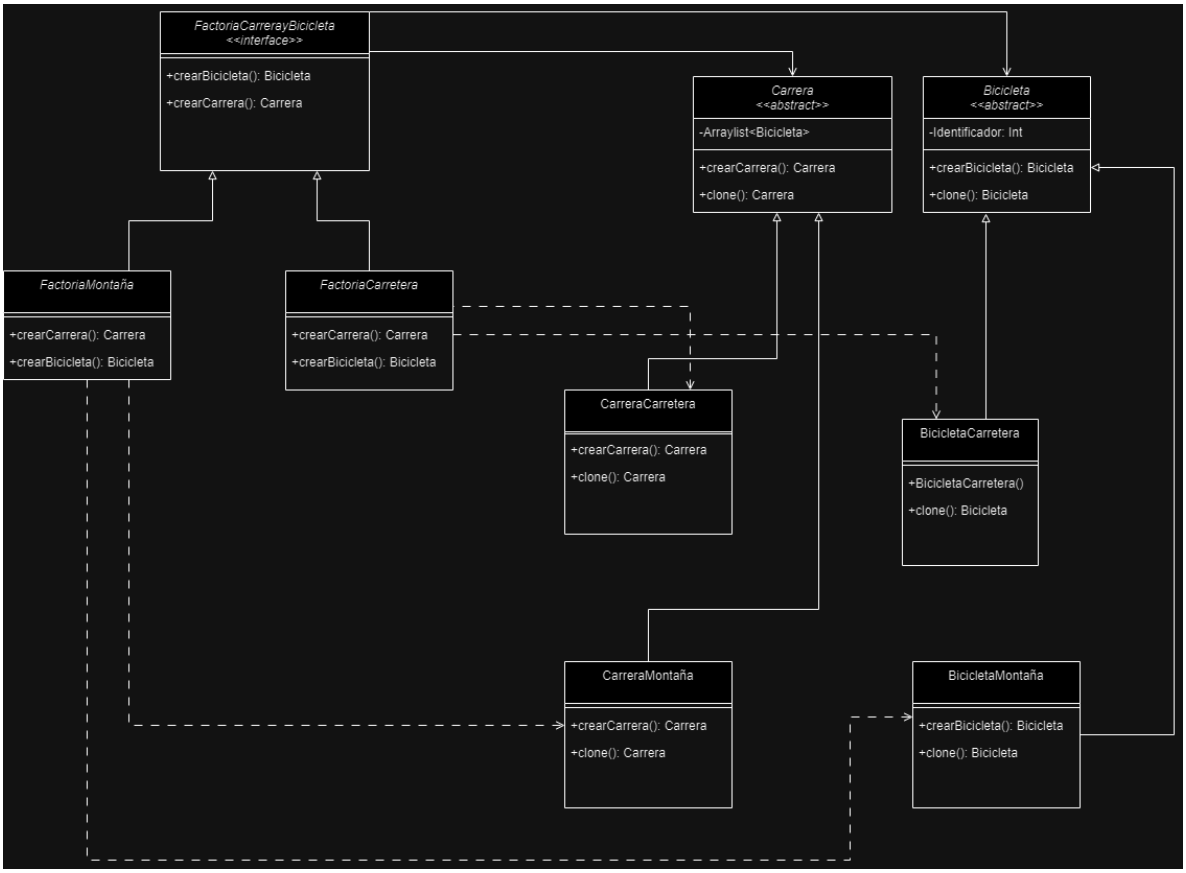

Figura 2: Diagrama UML

2.2.3. Código

```python
class FactoriaMontaña(FactoriaCarrerayBicicleta):
    def __init__(self,n):
        self.num_bicis=n
        FactoriaCarrerayBicicleta.__init__(self)

        if n<0:
            self.num_bicis=0

    def crear_carrera(self):
        bicis = []
        for i in range(self.num_bicis):
            if(len(bicis)==0):
                bicis.append(self.crear_bicicleta())
            else:
                bicis.append(copy.deepcopy(bicis[0]))
        return CarreraMontaña(bicis)

    def crear_bicicleta(self):
        return BicicletaMontaña()
```

Figura 3: Codigo de FactoriaMontaña con copy implementado

```python
class Bicicleta(ABC):

    def __init__(self, identificador):
        self.identificador=identificador

    def __copy__(self):
        new = self.__class__(
            self.identificador
        )
        new.__dict__.update(self.__dict__)
        return new

    def __deepcopy__(self,memo=None):
        if memo is None:
            memo = {}


        new = self.__class__(
            self.identificador
        )
        new.__dict__= copy.deepcopy(self.__dict__,memo)

        return new
```

Figura 4: Codigo de Bicicleta con copy implementado

2.2.4. Problemas encontrados

Al principio la forma de implementar el metodo copy fue un problema ya que no era un metodo propio de la clase, sino una sobrecarga del metodo copy ya implementado en la clase copy, una vez descubierto esto fue sencillo implementarlo, cambiando la llamada de la copia de bicicleta.copy() a copy.copy(bicicleta).

Al comprobar si la copia es efectiva nos encontramos tambien que la copia no era del todo correcta, ya que si se hacia algo en una clase como quitar bicicletas de una carrera, en la copia tambien desaparecen, por lo que tuvimos que implementar el deepcopy para que esto no ocurriera, una vez implementado ya funciona bien la copia.

2.3. Ejercicio 3: Patrón libre

2.3.1. Problema Planteado

Para la realización del ejercicio 3 hemos decidido plantear la siguiente situación: Queremos crear un sistema que nos permita crear diferentes personajes para un videojuego, encontramos dos tipos de personajes diferentes:

- Guerrero
- Mago

Ambos personajes comparten los mismos atributos:

1. **Arma**: contiene la variable que almacena el arma usada por el personaje.
2. **Habilidad**: contiene la variable que almacena la habilidad usada por el personaje.
3. **Armadura**: contiene un objeto de la clase armadura que almacena la armadura del personaje.

Tanto arma como habilidad serán unicos para cada personaje y se asignarán de forma predefinida es decir, solo hay un tipo de arma y habilidad para cada tipo de personaje.

En cuanto a la armadura podemos encontrar dos tipos diferentes:

- Armadura de fuego
- Armadura de planta

Es necesario para el desarrollo del videojuego que la armadura pueda ir variando en tiempo de ejecución.

Finalmente se requiere que el programa pueda mostrar por pantalla cada uno de los personajes desarrollados y que se pueda modificar su armadura en su momento.

2.3.2. Explicación de la solución propuesta

Una vez analizado el problema hemos decidido resolverlo de la siguiente forma:
Para implementar el patrón builder hemos definido las siguientes clases: PersonajeBuilder, una clase abstracta donde hemos definido las operaciones a realizar, su propio constructor y createnewpersonaje donde se inicializa al propio personaje, de esa clase abstractan heredan tanto GuerreroBuilder como MagoBuilder que se encargan de realizar todas las funciones definidas en PersonajeBuilder de forma propia, añadiendo cada componente al objeto, por ejemplo cada uno configura: su propia habilidad, arma como elemento al objeto e implementan una operación extra set-armadura que devuelve el atributo armadura. Ambas dos clases permiten la construcción del producto final en este caso "Personaje". Para la creación de la armadura hemos usado el patrón Decorador definiendo por tanto una interfaz Armadura donde es definida la función dar-apariencia la cual es implementada tanto por armadura simple (que sería la que viene por defecto) y por FuegoDecorador y PlantaDecorador que nos permitirán modificar esta armadura en tiempo de ejecución.

2.3.3. Diagrama de Diseño

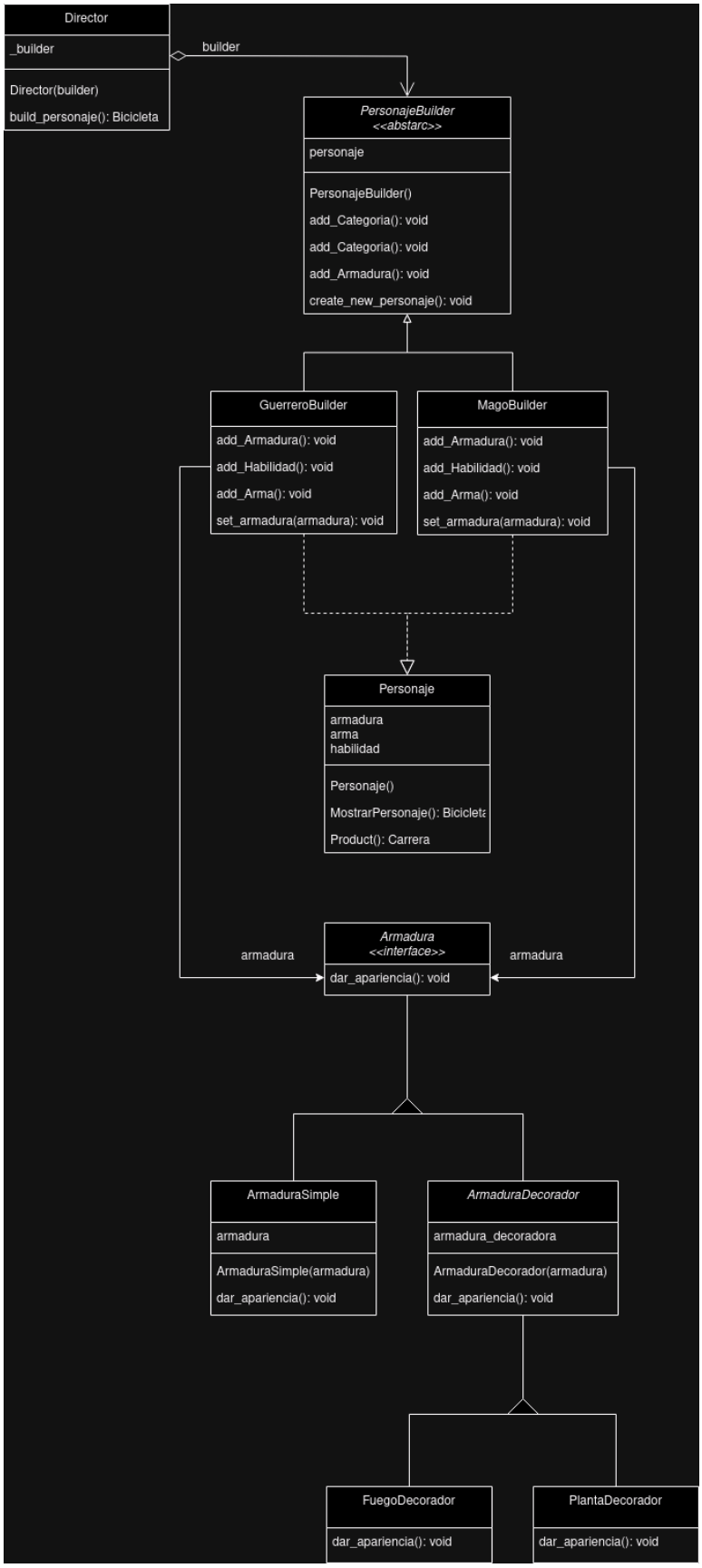

Figura 5: Diagrama UML.

2.3.4. Código

- Para la construcción de personajes, hemos decidido usar el patrón builder. Hemos definido las siguientes clases: La clase director que se encarga de crear los personajes a traves del builder. La clase abstracta PersonajeBuilder que define los métodos para la creación de personajes. Las clases GuerreroBuilder Y MagoBuilder que implementan los métodos de PersonajeBuilder permiten construir personajes específicos, cada una de estos builder conoce cómo construir un tipo específico de personaje. La clase Personaje representa el producto final que se crea utilizando el patrón, contiene un constructor por defecto y un método para mostrar información.

- Para la modificación de la armadura en tiempo de ejecución, hemos decidido utilizar el patrón decorador donde cada personaje empiza con una armadura simple y haciendo uso de este patrón podemos cambiar el tipo de la armadura.

2.3.5. Problemas Encontrados

Durante la realización del ejercicio nos hemos encontrado con la complejidad de enlazar los dos patrones de forma correcta, además como complejidad extra debíamos evitar la creación de excesivas clases ya que podrían acabar produciendo un código demasiado complejo. Para solucionarlo nos hemos documentado con información contenida en Internet y en los apuntes.

2.4. Ejercicio 4 Patrón filtros de intercepción en Java

2.4.1. Problema Planteado

Este ejercicio consiste en realizar una interfaz gráfica donde poder implimentar un salpicadero de un vehículo, para su desarrollo es necesario aplicar el patrón Filtros de Intercepción. Hemos decidido hacerlo en Java.

2.4.2. Explicación de la solución propuesta

En primer lugar hemos creado una clase objetivo a la que hemos llamado salpicadero.En esta clase hemos implimentado una interfaz de usuario para poder interactuar con él, dandonos la posibilidad de poder ver la velocidad, revoluciones y distancia, además de tener la opción de interactuar con el mediante unos botones con los que poder encender el motor, acelerar o frenar.

2.4.3. Código

Filtro
Creamos la interfaz Filtro donde declaramos el método ejecutar que será implementado por sus clases hijas.

FiltroCalcularVelocidad
Hija de la clase Filtro donde implementamos el método ejecutar el cual se encarga de actualizar las revoluciones dependiendo de si se está frenando o acelerando.

FiltroRepeecutirRozamiento
Hija de la clase Filtro donde implementamos el método ejecutar el cual se encarga de aplicar un rozamiento a las revoluciones.

Cliente
La clase Cliente se encarga de mandar la petición de Salpicadero a travez del Gestor de Filtros.

GestorFiltros
Se encarga de añadir y crear los filtros que más tarde serán usados.

CadenaFiltro
La clase CadenaFiltros se encarga de crear la lista de filtros y aplicarlos.

2.4.4. Diagrama de Diseño

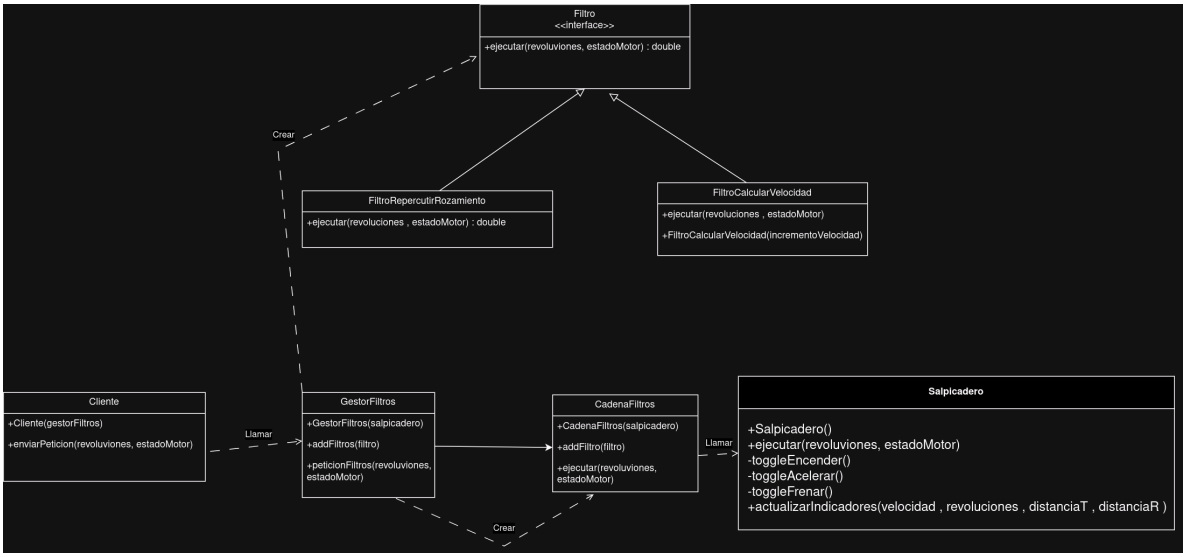

Figura 6: Diagrama UML.

2.5. Ejercicio 5 Scrape Strategy

2.5.1. Problema Planteado

En la realización del ejercicio 5 se plantea realizar el "scrapping"de las páginas de yahoo de la bolsa realizando un programa a partir del patrón Estrategia, se pide desarrollar dos tipos de estrategias distintas:

- Estrategia Beautiful Soup
- Estrategia Selenium

Ambas estrategias buscan obtener el mismo resultado scrapear el contenido de una página web para obtener los siguientes valores:

1. **Precio de cierre anterior**
2. **precio de apertura**
3. **Volumen**
4. **Capitalización de mercado**

Finalmente se requiere que el programa almacene la información en un fichero json actualizandose cada vez que se ejecute el programa.

2.5.2. Explicación de la solución propuesta

Una vez analizado el problema hemos decidido resolverlo de la siguiente forma: En primer lugar hemos definido las clases a realizar para seguir el patrón Estrategia, hemos definido una interfaz ScrapeStrategy donde se definirán las operaciones a realizar en las clases que implementan las diferentes estrategias a seguir, en nuestro caso BeautifulSoupStrategy y SeleniumStrategy, finalmente hemos definido la clase Contexto que servirá de intermediario entre las estrategias y el usuario.

2.5.3. Diagrama de Diseño

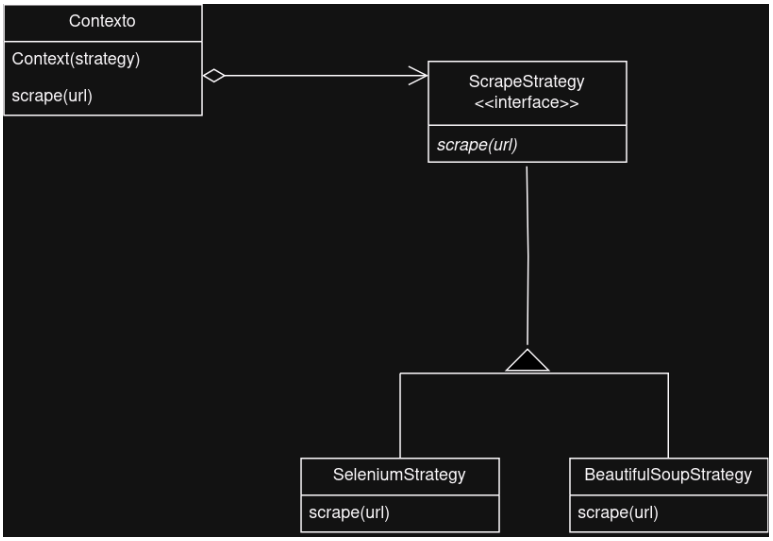

Figura 7: Diagrama UML.

2.5.4. Pseudocódigo

Para la realización del ejercicio se ha trabajado con Beautiful Soup, una biblioteca de Python, y con Selenium, una herramienta de automatización web. Se va a proceder a comentar como se ha desarrollado cada una.

Beautiful Soup

Para la realizacion de la estrategia con BeautifulSoup se ha implementado la función scrape, definida previamente en la interfaz, cuyo funcionamiento es el siguiente: En primer lugar se produce un request para obtener el contenido de la pagina en cuestión la cual es pasada mediante el parámetro url, si la petición es correcta se continua con el procedimiento , a continuación se crea el objeto de la clase BeautifulSoup importada al comienzo de la clase y se produce los respectivos find de cada valor, para conocer el nombre exacto de las etiquetas es necesario acceder al modo desarrollador para inspeccionar los elementos, finalmente se devuelven los valores en un diccionario.

```python
import json
import requests
from bs4 import BeautifulSoup
from ScrapeStrategy import ScrapeStrategy

class BeautifulSoupStrategy(ScrapeStrategy):
    def scrape(self, url):
        response = requests.get(url)
        if response.status_code == 200:
            soup = BeautifulSoup(response.content, 'html.parser')

            open_value_td = soup.find('td', {'data-test': 'OPEN-value'}).text
            if open_value_td == None:
                open_value_td = 'Open Value not found'

            precio_cierre_anterior = soup.find('td',{'data-test': 'PREV_CLOSE-value'}).text
            if precio_cierre_anterior == None:
                precio_cierre_anterior = 'Precio cierre anterior not found'

            volumen = soup.find('td',{'data-test': 'TD_VOLUME-value'}).text
            if volumen == None:
                volumen = 'Volumen not found'

            capitalizacion_mercado = soup.find('td',{'data-test': 'MARKET_CAP-value'}).text
            if capitalizacion_mercado == None:
                capitalizacion_mercado = 'Capitalizacion Mercado Value not found'

            # Crear un diccionario con la información
            return {
                'Open Value': open_value_td,
                'Previous Close': precio_cierre_anterior,
                'Volume': volumen,
                'Market Cap': capitalizacion_mercado
            }

        else:
            return f'Failed to retrieve the webpage, status code: {response.status_code}'

```

Figura 8: Código Beautiful Soup.

Selenium

Para la realización de la estrategia con Selenium se ha implementado la función scrape, definida previamente en la interfaz, cuyo funcionamiento es el siguiente: En primer lugar se produce un webdriver.chrome para obtener el navegador que se prefiera, en nuestro caso hemos usado Chrome, posteriormente accedemos a la url establecida en el main y llevamos a cabo una espera para asegurarnos que los elementos estén cargados para proceder a recopilarlos, es decir no se pasa a recogerlos hasta que se aseguran que están cargados, posteriormente con la función find-element obtenemos el elemento requerido a partir del XPath dentro de la página el cual se obtiene a partir del modo desarrollador inspeccionando la página, finalmente cerramos el driver y se devuelven los valores en un diccionario.

Para la realización de la clase Contexto simplemente definimos su constructor donde vamos a introducir la estrategia utilizada y la función scrape que va a devolver el resultado de la función scrape de la estrategia pasada como parámetro al constructor.

```python
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from ScrapeStrategy import ScrapeStrategy
import json

class SeleniumStrategy(ScrapeStrategy):
    def scrape(self, url):

        # Inicializamos el driver de Selenium
        driver = webdriver.Chrome()  # Aquí se puede utilizar el driver del navegador que prefieras

        # Accedemos a la página
        driver.get(url)

        # Espera a que los elementos carguen
        element_present = EC.presence_of_element_located((By.XPATH, "/html/body/div[1]/div/div/div[1]/div/div[3]/div[1]
        WebDriverWait(driver, 10).until(element_present)

        # Extraemos la información relevante
        previous_close = driver.find_element(By.XPATH,'//*[@id="quote-summary"]/div[1]/table/tbody/tr[1]/td[2]').text

        open_value = driver.find_element(By.XPATH,'//*[@id="quote-summary"]/div[1]/table/tbody/tr[2]/td[2]').text


        volume = driver.find_element(By.XPATH, '//*[@id="quote-summary"]/div[1]/table/tbody/tr[7]/td[2]').text

        market_cap = driver.find_element(By.XPATH, '//*[@id="quote-summary"]/div[2]/table/tbody/tr[1]/td[2]').text

        # Cerramos el driver
        driver.quit()

        # Devolvemos los datos en un diccionario

        return {
            'Open Value': open_value,
            'Previous Close': previous_close,
            'Volume': volume,
            'Market Cap': market_cap
        }
```

Figura 9: Código Selenium.

2.5.5. Problemas Encontrados

Durante la realización del ejercicio hemos encontrado bastantes problemas en establecer bien la ruta XPath en el caso de Selenium ya que había que ser bastante específico para poder obtener el valor, además hemos tenido que entender bien el flujo de trabajo con estas dos herramientas. Para solucionarlo hemos consultado documentación en Internet.

2.6. Modos de compilación

Instrucciones de compilación:
Ejercicio 1 : javac Ejercicio1/Main.java; java Ejercicio1.main
Ejercicio 2 : python Main.py
Ejercicio 3 : python Main.py
Ejercicio 4 : javac Ejercicio4/Cliente.java; java Ejercicio4.Cliente
Ejercicio 5 : python Main.py