Desarrollo de Software - Práctica 1

Fernando Cuesta Bueno
Carlos Fernández Arrabal
Antonio Manuel García Mesa
22/03/2025

Índice

1. Introducción **3**

2. Ejercicio 1. Patrón Factoría Abstracta en Java **3**
2.1. Clase FactoriaCarrerayBicicleta 3
2.2. Clase FactoriaCarretera 4
2.3. Clase FactoriaMontaña 4
2.4. Clase Bicicleta 4
2.5. Clase BicicletaCarretera 4
2.6. Clase BicicletaMontaña 4
2.7. Clase Carrera 5
2.8. Clase CarreraCarretera 5
2.9. Clase CarreraMontaña 5
2.10. Main 6
2.11. Diagrama UML 6
2.12. Muestras de Ejecución 7

3. Ejercicio 2. Patrón Decorador en Python **8**
3.1. Clase LLM 8
3.2. Clase LLMBasic 8
3.3. Clase LLMDecorator 8
3.4. Clase TranslationDecorator 8
3.5. Clase ExpansionDecorator 8
3.6. Diagrama UML 9
3.7. Muestras de ejecución 9

4. Ejercicio 3. Patrón Strategy para Web Scrapping **10**
4.1. Clase Compositor 10
4.2. Clase SimpleCompositor 10
4.3. Clase TexCompositor 10
4.4. Clase Composition 10
4.5. Main 11
4.6. Diagrama UML 11
4.7. Muestras de Ejecución 12

5. Ejercicio 4. Autenticación de credenciales de usuario **14**
5.1. Clase AuthTarget 14
5.2. Clase Client 14
5.3. Interface Filter 14
5.4. Clase DomainFilter 14
5.5. Clase FilterChain 14
5.6. Clase FilterManager 15
5.7. Clase MailFilter 15
5.8. Clase Message 15
5.9. Clase PasswordLengthFilter 15
5.10. Clase PasswordSpecialCharacterFilter 16

5.11. Clase PasswordUpperCaseFilter 16
5.12. Main 16
5.13. Diagrama UML 16
5.14. Muestras de Ejecución 17

1. Introducción

En esta práctica, se realizarán diferentes programas orientados a objetos, bajo distintos patrones de diseño creacionales y estructurales vistos en la asignatura. Entre los diferentes objetivos que se desean cumplir en esta práctica, encontramos:

- **Familiarizarse con el uso de herramientas.**
- **Aplicar distintos patrones creacionales y estructurales.**
- **Adquirir destreza en la práctica de diseño OO.**
- **Adaptación de los patrones.**

El enlace a nuestro repositorio de Github es el siguiente: `https://github.com/YTTREW/DS_FAC`

2. Ejercicio 1. Patrón Factoría Abstracta en Java

Para realizar el programa se van a utilizar hebras para simular dos carreras de manera simultánea, de montaña y de carretera. Ambas contarán con el mismo número de bicicletas y tendrán una duración de 60 segundos. En cada carrera, habrá una determinada tasa de abandono del total de bicicletas que participan:

- En la carrera de montaña, se retirará el 20 % de las bicicletas antes de que termine la carrera.
- En la carrera de carretera, se retirará el 10 % de las bicicletas antes de que finalice la carrera.
- En ambas carreras, todas las bicicletas se retirarán de manera simultánea.

Para gestionar la fabricación de bicicletas y carreras de cada tipo, se implementará un patrón de diseño Factoría Abstracta junto con el patrón Método Factoría.

A continuación se van a detallar las diferentes interfaces y clases que se han definido en el programa en base a los patrones aplicados y el rol que tiene cada una dentro del ejercicio.

2.1. Clase FactoriaCarrerayBicicleta

La interfaz FactoriaCarreraYBicicleta define el patrón Factoría Abstracta para la creación pública de los dos objetos del ejercicio, las carreras y las bicicletas. Se declaran dos métodos:

- crearCarrera(ArrayList<Bicicleta> bicicletas): Crea y devuelve un objeto de tipo carrera, permitiendo crear los diferentes tipos de carreras (carretera y montaña).
- crearBicicleta(int id): Crea y devuelve un objeto de tipo bicicleta, permitiendo crear los diferentes tipos de bicicletas (carretera y montaña).

2.2. Clase FactoriaCarretera

Esta clase implementa la interfaz mencionada anteriormente. La principal función es crear objetos de tipo Carrera y Bicicleta para la modalidad de carretera. Se declaran dos métodos:

- crearCarrera(ArrayList<Bicicleta> bicicletas): Crea y devuelve un objeto de tipo CarreraCarretera, representando una carrera de tipo carretera con las bicicletas pasadas por parámetro.
- crearBicicleta(int id): Crea y devuelve un objeto de tipo BicicletaCarretera, representando una bicicleta de carretera con un id único asignado.

2.3. Clase FactoriaMontaña

Esta clase implementa la interfaz mencionada anteriormente. La principal función es crear objetos de tipo Carrera y Bicicleta para la modalidad de montaña. Se declaran dos métodos:

- crearCarrera(ArrayList<Bicicleta> bicicletas): Crea y devuelve un objeto de tipo CarreraMontaña, representando una carrera de tipo montaña con las bicicletas pasadas por parámetro.
- crearBicicleta(int id): Crea y devuelve un objeto de tipo BicicletaMontaña, representando una bicicleta de montaña con un id único asignado.

2.4. Clase Bicicleta

La clase Bicicleta es una clase abstracta que sirve para representar a una bicicleta, la cual tiene asignada un id único. Se declaran dos métodos y un atributo:

- id (int): Atributo que almacena el identificador de cada bicicleta.
- Bicicleta(int id): Constructor para inicializar un objeto de tipo Bicleta con el id.
- getId() int: Método que devuelve el identificador de la bicicleta.

2.5. Clase BicicletaCarretera

La clase BicicletaCarretera extiende de la clase Bicicleta y sirve para representar una bicicleta de carretera. Se define un método:

- BicicletaCarretera(int id): Constructor que llama al de la clase Bicileta para inicializar la bicicleta de carretera.

2.6. Clase BicicletaMontaña

La clase BicicletaMontaña extiende de la clase Bicicleta y sirve para representar una bicicleta de montaña. Se define un método:

- BicicletaMontaña(int id): Constructor que llama al de la clase Bicicleta para inicializar la bicicleta de montaña.

2.7. Clase Carrera

La clase carrera es una clase abstracta que extiende de Thread, permitiendo que las carreras se ejecuten en paralelo mediante el uso de hebras. Se definen varios métodos y atributos:

- ArrayList<Bicicleta> bicicletas: Atributo que crea una lista con todas las bicicletas que participan en la carrera.
- String tipoCarrera: Atributo que define el tipo de carrera (carretera o montaña) que se va a realizar.
- double tasaRetirada: Atributo que define el porcentaje de bicicletas que se retirarán durante la carrera.
- Carrera(ArrayList<Bicicleta> bicicletas, String tipoCarrera, double tasaRetirada : Constructor encargado de inicilizar los atributos anteriores para crear una carrera.
- retirarBicicletas() void: Método encargado de calcular el número de bicicletas a retirar en función de la tasa de retirada de cada tipo de carrera. Tras calcularlo, las elimina de la lista.
- run() void: Método para definir el comportamiento de las hebras durante la duración de la carrera:
  - Inicia la carrera.
  - Se espera un tiempo aleatorio antes de retirar el porcentaje correspondiente de bicicletas.
  - Retira un número de bicicletas.
  - La carrera continua durante hasta que terminen los 60 segundos.

2.8. Clase CarreraCarretera

La clase CarreraCarretera extiende de la clase abstracta Carrera. Se utiliza para definir el comportamiento de una carrera de carretera. Se define un método:

- CarreraCarretera(ArrayList<Bicicleta> bicicletas): Constructor que llama al de la clase padre para incializar el correspondiente tipo de carrera con sus características (Carretera y 10 % de tasa de abandono).

2.9. Clase CarreraMontaña

La clase CarreraMontaña extiende de la clase abstracta Carrera. Se utiliza para definir el comportamiento de una carrera de montaña. Se define un método:

- CarreraMontaña(ArrayList<Bicicleta> bicicletas): Constructor que llama al de la clase padre para incializar el correspondiente tipo de carrera con sus características (Montaña y 20 % de tasa de abandono).

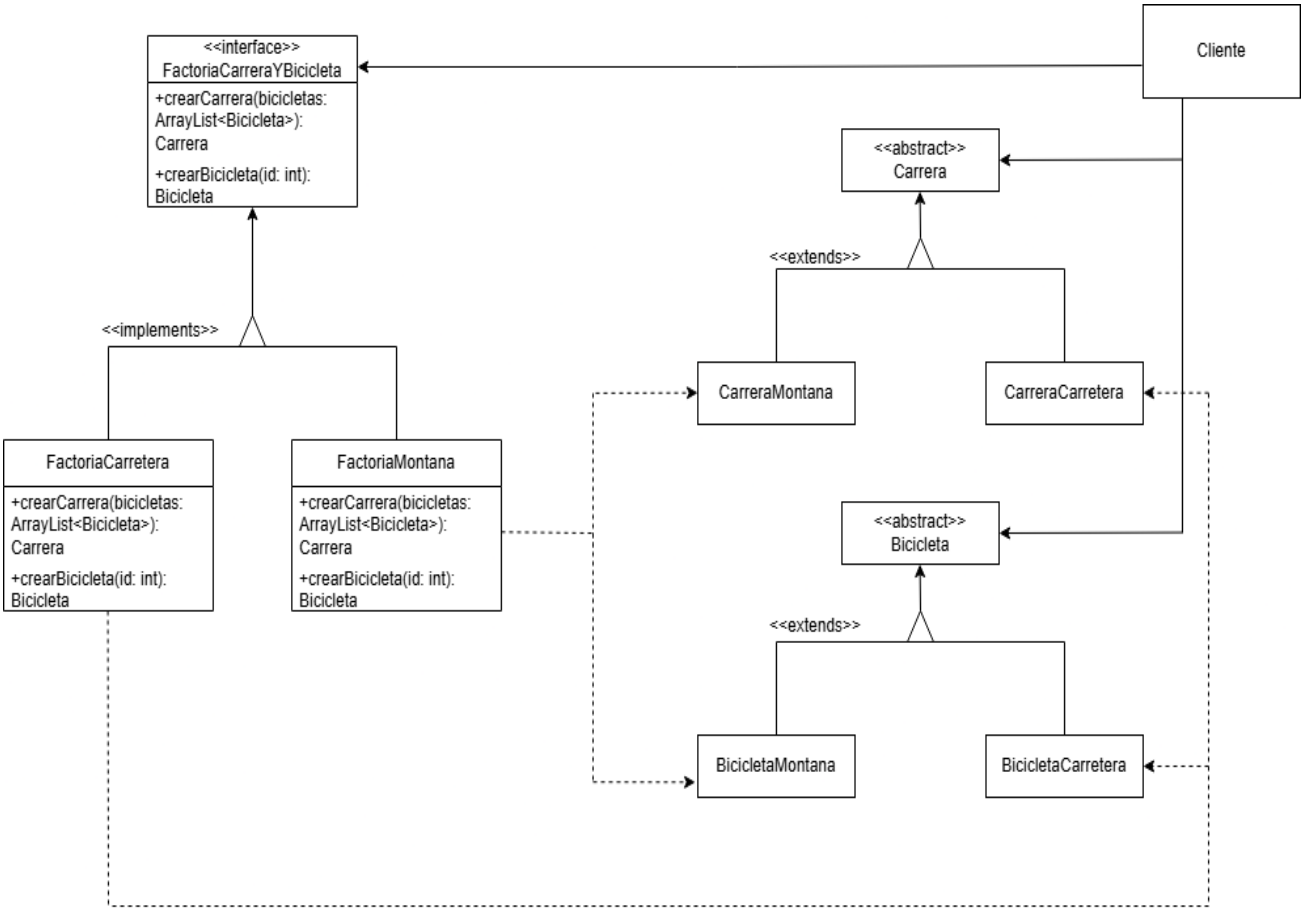
2.10. Main

En el main se van a ejecutar todas las operaciones principales de las carreras. En primer lugar, se solicita el número de bicicletas que se usarán para ambas carreras. A continuación, se crean dos instancias de las factorías, FactoriaCarretera y FactoriaMontana, encargadas de crear las bicicletas y carreras de cada tipo.

Se emplean dos bucles para generar el número de bicicletas especificado para cada tipo de carrera y añadirlas a la lista. También, mediante el uso de las factorías, se crean los dos objetos CarreraCarretera y CarreraMontana.

Una vez creados los objetos necesarios, comienzan las carreras llamando al método 'start' para que se ejecuten en hilos separados. Para mejorar la lógica de salida del programa se incluye un 'while' donde, cada 8 segundos, se muestran cuántas bicicletas quedan en cada carrera mientras estén en ejecución. Finalmente, se espera a que los hilos terminen mediante 'join' y se muestra un mensaje indicando que las carreras han finalizado.

2.11. Diagrama UML



2.12. Muestras de Ejecución

```
PS C:\Users\carlo\Desktop\Universidad\uni\4curso\DS\DS_Repo\DS_FAC> java ejercicio1.Main
Ingrese la cantidad de bicicletas para la carrera: 20
Iniciando la carrera de tipo carretera con 20 bicicletas
Iniciando la carrera de tipo montaña con 20 bicicletas
Número de bicicletas en la carrera de carretera: 20
Número de bicicletas en la carrera de montaña: 20
-------------------------------------------------
Bicicletas retiradas durante la carrera de carretera: 2
La tasa de abandono ha sido del: 10.0%.
-------------------------------------------------
Número de bicicletas en la carrera de carretera: 18
Número de bicicletas en la carrera de montaña: 20
-------------------------------------------------
Número de bicicletas en la carrera de carretera: 18
Número de bicicletas en la carrera de montaña: 20
-------------------------------------------------
Bicicletas retiradas durante la carrera de montaña: 4
La tasa de abandono ha sido del: 20.0%.
-------------------------------------------------
Número de bicicletas en la carrera de carretera: 18
Número de bicicletas en la carrera de montaña: 16
-------------------------------------------------
Número de bicicletas en la carrera de carretera: 18
Número de bicicletas en la carrera de montaña: 16
-------------------------------------------------
Número de bicicletas en la carrera de carretera: 18
Número de bicicletas en la carrera de montaña: 16
-------------------------------------------------
Número de bicicletas en la carrera de carretera: 18
Número de bicicletas en la carrera de montaña: 16
-------------------------------------------------
Número de bicicletas en la carrera de carretera: 18
Número de bicicletas en la carrera de montaña: 16
-------------------------------------------------
Las carreras de montaña y carretera han finalizado.
```

3. Ejercicio 2. Patrón Decorador en Python

Para este ejercicio empleamos el patrón Decorador para trabajar con LLMs y la API de Hugging Face.

3.1. Clase LLM

Esta clase será la interfaz que indicará qué métodos deben ser implementados en las siguientes clases. En este caso presenta un único método:

- generate_summary(text, input_lang, output_lang, model): será implementado por las clases LLMBasic y la clase LLMDecorator.

3.2. Clase LLMBasic

Esta clase será la encargada de realizar un resumen a partir de un texto y un modelo de entrada. Además, se le debe indicar el token empleado en Hugging Face para poder realizar la consulta. Presenta los siguientes métodos:

- __init__: constructor de la clase al que se le indica el token.
- generate_summary(text, input_lang, output_lang, model): función que devuelve el resumen del texto.

3.3. Clase LLMDecorator

Esta se trata de una clase abstracta, la cual permitirá implementar las distintas clases decoradoras. Sus métodos son:

- __init__: constructor de la clase al que se le proporciona una instancia de la clase LLM.
- generate_summary(text, input_lang, output_lang, model): función que devuelve el resumen del texto.

3.4. Clase TranslationDecorator

Decorador el cual se encargará de añadir la funcionalidad de traducir un texto tras una llamada al modelo proporcionado.

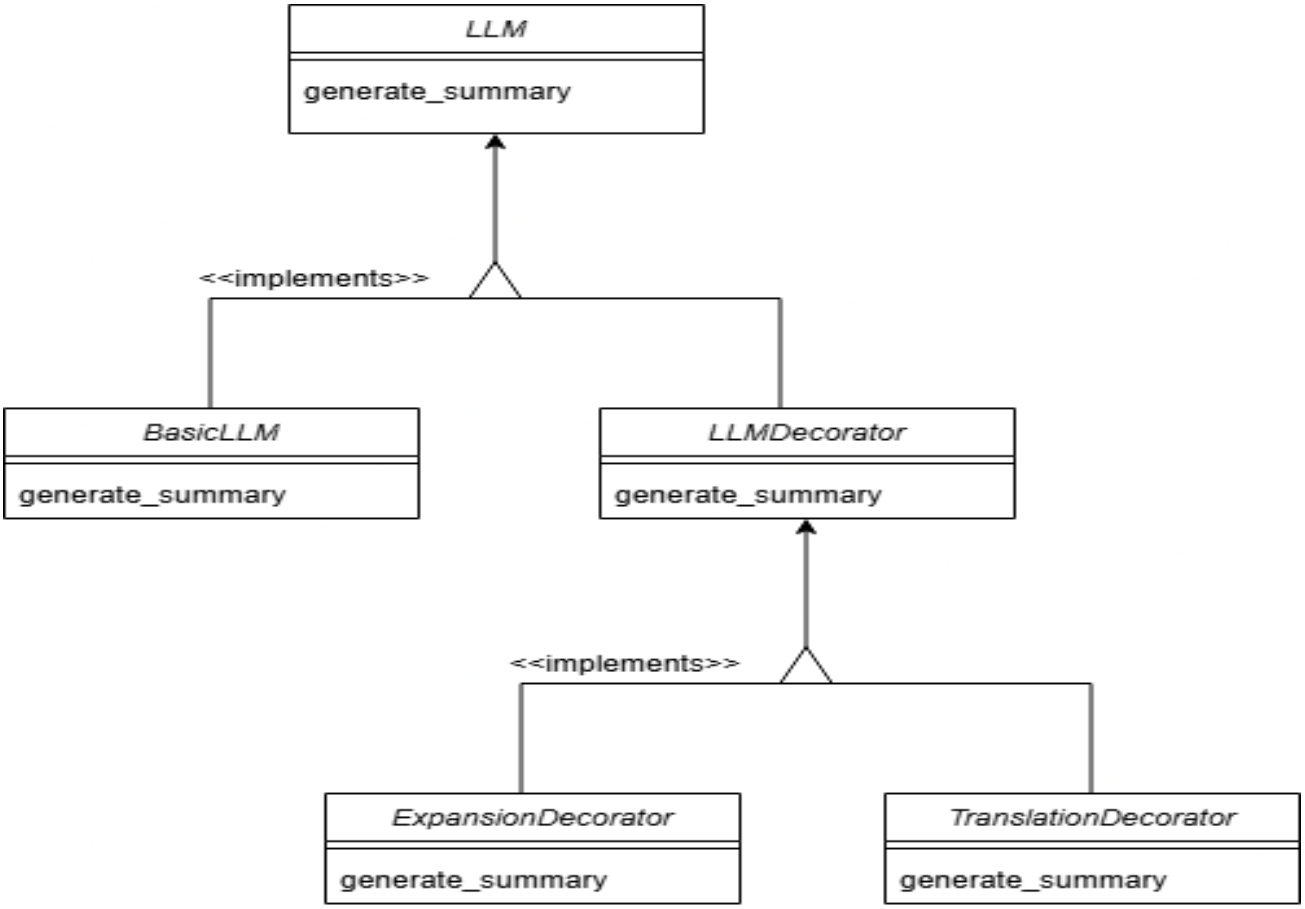
- generate_summary(text, input_lang, output_lang, model): función que devuelve el texto traducido.

3.5. Clase ExpansionDecorator

Decorador el cual se encargará de añadir la funcionalidad de ampliar el contenido de un texto tras una llamada al modelo proporcionado.

- generate_summary(text, input_lang, output_lang, model): función que devuelve el texto una vez se ha ampliado.

3.6. Diagrama UML



3.7. Muestras de ejecución

4. Ejercicio 3. Patrón Strategy para Web Scrapping

Para realizar el programa se van a utilizar dos estrategias de scrapeo distintas. Una va a ser BeautifulSoup y la otra selenium.

4.1. Clase Compositor

Esta clase va a ser nuestra clase base para formar las dos formas de scrapeo. Para ello, esta clase es una clase abstracta que está compuesta por el siguiente método:

- compose(self, url): Que se implementará en las clases hijas TexCompositor y SimpleCompositor.

4.2. Clase SimpleCompositor

Esta clase va a representar el scrapeo usando la estrategia BeautifulSoup. Para ello usamos la librería requests de python para poder conectarnos a la página web y extraer los datos. Implementa los siguientes métodos:

- compose(self, url): Extrae mediante una petición HTTP el HTML de la página y lo convierte en un objeto para poder trabajar con él. Finalmente, extrae los datos.
- extraer datos(self, soup): Recorre el objeto HTML y extrae las citas, autores y etiquetas de la página para almacenarlas en un diccionario.

4.3. Clase TexCompositor

Esta clase va a representar el scrapeo usando la estrategia selenium. Para ello usamos la librería webdriver.Chrome para iniciar el navegador Chrome y poder trabajar con la página:

- compose(self, url): Carga la página y extrae los datos.
- extraer datos(self): Encuentra el div "quotez extrae todos los datos de la página para almacenarlo en un diccionario.
- close(self): Cierra el navegador.

4.4. Clase Composition

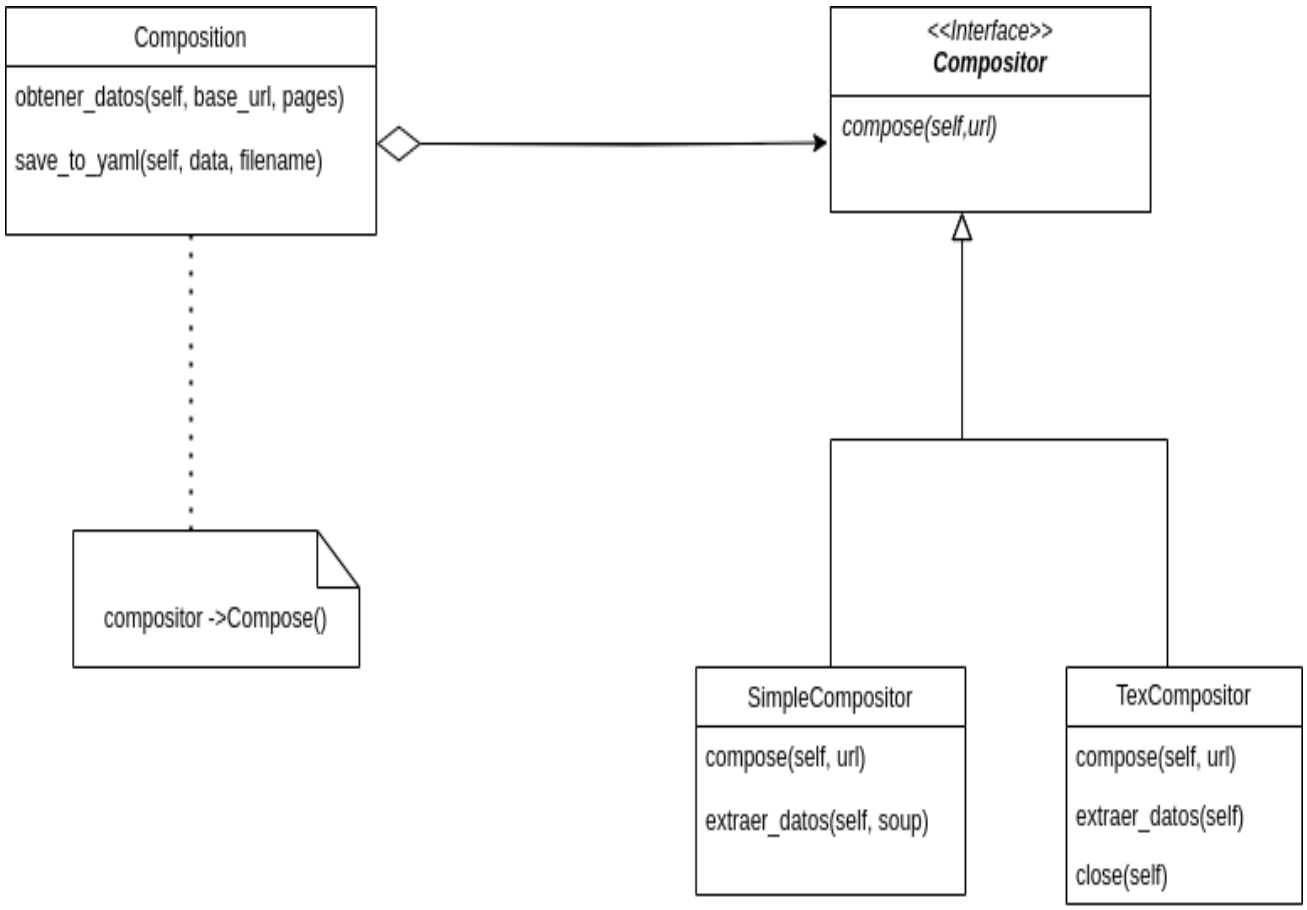
Esta clase es la principal del patrón de diseño Strategy. Es la que nos permie trabajar con las distintas estrategias de scrapeo de nuestra práctica. Está compuesta por los siguientes métodos:

- obtener datos(self, base url, pages): Obtiene todas las citas de las primeras cinco páginas.
- save to yaml(self, data, filename): Abre un yaml en modo escritura y almacena todos los datos en él. Permitiendo caracteres especiales.

4.5. Main

El main es muy sencillo. Se encarga de crear un Composition usando BeautifulSoup y otro con selenium. Una vez tiene los dos creados, obtiene los datos de la pagina web y almacena los resultados de cada estrategia en un fichero yml distinto para cada uno.

4.6. Diagrama UML



4.7. Muestras de Ejecución

Ejecución con Selenium:


```yaml
- author: Albert Einstein
  tags:
  - change
  - deep-thoughts
  - thinking
  - world
  text: “The world as we have created it is a process of our thinking. It cannot be
    changed without changing our thinking.”
- author: J.K. Rowling
  tags:
  - abilities
  - choices
  text: “It is our choices, Harry, that show what we truly are, far more than our
    abilities.”
- author: Albert Einstein
  tags:
  - inspirational
  - life
  - live
  - miracle
  - miracles
  text: “There are only two ways to live your life. One is as though nothing is a
    miracle. The other is as though everything is a miracle.”
- author: Jane Austen
  tags:
  - aliteracy
  - books
  - classic
  - humor
  text: “The person, be it gentleman or lady, who has not pleasure in a good novel,
    must be intolerably stupid.”
- author: Marilyn Monroe
  tags:
  - be-yourself
  - inspirational
  text: “Imperfection is beauty, madness is genius and it's better to be absolutely
    ridiculous than absolutely boring.”
- author: Albert Einstein
  tags:
  - adulthood
  - success
  - value
  text: “Try not to become a man of success. Rather become a man of value.”
- author: André Gide
```

Ejecución con BeautifulSoup:

```
- author: Albert Einstein
  tags:
  - change
  - deep-thoughts
  - thinking
  - world
  text: “The world as we have created it is a process of our thinking. It cannot be
    changed without changing our thinking.”
- author: J.K. Rowling
  tags:
  - abilities
  - choices
  text: “It is our choices, Harry, that show what we truly are, far more than our
    abilities.”
- author: Albert Einstein
  tags:
  - inspirational
  - life
  - live
  - miracle
  - miracles
  text: “There are only two ways to live your life. One is as though nothing is a
    miracle. The other is as though everything is a miracle.”
- author: Jane Austen
  tags:
  - aliteracy
  - books
  - classic
  - humor
  text: “The person, be it gentleman or lady, who has not pleasure in a good novel,
    must be intolerably stupid.”
- author: Marilyn Monroe
  tags:
  - be-yourself
  - inspirational
  text: “Imperfection is beauty, madness is genius and it's better to be absolutely
    ridiculous than absolutely boring.”
- author: Albert Einstein
  tags:
  - adulthood
  - success
  - value
  text: “Try not to become a man of success. Rather become a man of value.”
- author: André Gide
```

5. Ejercicio 4. Autenticación de credenciales de usuario

En este ejercicio se va a realizar un sistema de autenticación de credenciales de un usuario, con el correo y la contraseña, aplicando el patrón de filtros de intercepción. Para ello, se han implementado las siguientes clases.

5.1. Clase AuthTarget

Esta clase se encarga de mostrar si se ha autenticado correctamente el correo. Se declara el siguiente método:

- authenticate(Message message) void: Notifica por pantalla que se ha autenticado correctamente el correo.

5.2. Clase Client

Esta clase se encarga de enviar las credenciales (correo y contraseña) al FilterManager para que sean procesadas. Se declaran los siguientes métodos:

- Client(FilterManager filterManager): Constructor que asigna el FilterManager correspondiente.
- sendCredentials(String correo, String contraseña) void: Encapsula y envia las credenciales al FilterManager.

5.3. Interface Filter

Esta interfaz define el esquema a seguir por el resto de filtros a implementar. Se declara el siguiente método:

- execute(Message message) void: Método que cada clase que implemente Filter deberá sobreescribir para implementar el filtro.

5.4. Clase DomainFilter

Esta clase se encarga de implementar uno de los filtros solicitados. En concreto, verifica si el correo contiene alguno de los dominios válidos: @gmail.com o @hotmail.com. Se declara el siguiente método:

- void execute(Message message): Este método de la clase Filter se sobrescribe para verificar si el correo proporcionado termina o no con los dos dominios válidos.

5.5. Clase FilterChain

Esta clase se encarga de gestionar y ejecutar todos los filtros sobre las credenciales. Se declaran los siguientes métodos:

- void addFilter(Filter filter): Añade un filtro a la cadena.
- void setTarget(AuthTarget target): Añade un target a la clase
- void execute(Message message): Ejecuta todos los filtros almacenados el la lista y si todos los filtros pasan correctamente, se envia el mensaje al target.

5.6. Clase FilterManager

Esta clase se encarga de gestionar la cadena de filtros y ejecutar el proceso de autenticación. Se declaran los siguientes métodos:

- FilterManager(AuthTarget target): Marca el target a la cadena de filtros.
- void addFilter(Filter filter): Añade a la lista de filtros un nuevo filtro.
- void authenticate(Message message): Ejecuta el proceso de autenticación para las credenciales.

5.7. Clase MailFilter

Esta clase se encarga de verificar si el correo tiene un formato correcto. Se declara el siguiente método:

- void execute(Message message): Este método de la clase Filter se sobrescribe para verificar que el correo tenga el formato correcto. En concreto, que tenga texto antes del caracter '@'.

5.8. Clase Message

Esta clase se encarga de representar las credenciales de un usuario (email y contraseña). Se declaran los siguientes métodos:

- Message(String email, String password): Constructor de la clase message.
- String getEmail(): Devuelve el atributo email.
- String getPassword(): Devuelve el atributo contraseña.
- void setEmail(String email): Inicializa el atributo email.
- void setPassword(String password): Inicializa el atributo contraseña.

5.9. Clase PasswordLengthFilter

Esta clase se encarga de verificar si la contraseña tiene una longitud concreta. Se declara el siguiente método:

- void execute(Message message): Este método de la clase Filter se sobrescribe para verificar si la contraseña tiene la longitud correcta. En concreto, una logitud entre 5 y 50 caracteres.

5.10. Clase PasswordSpecialCharacterFilter

Esta clase se encarga de verificar si la contraseña tiene al menos un caracter especial. Se declara el siguiente método:

- void execute(Message message): Este método de la clase Filter se sobrescribe para verificar la contraseña si tiene al menos un carácter especial.

5.11. Clase PasswordUpperCaseFilter

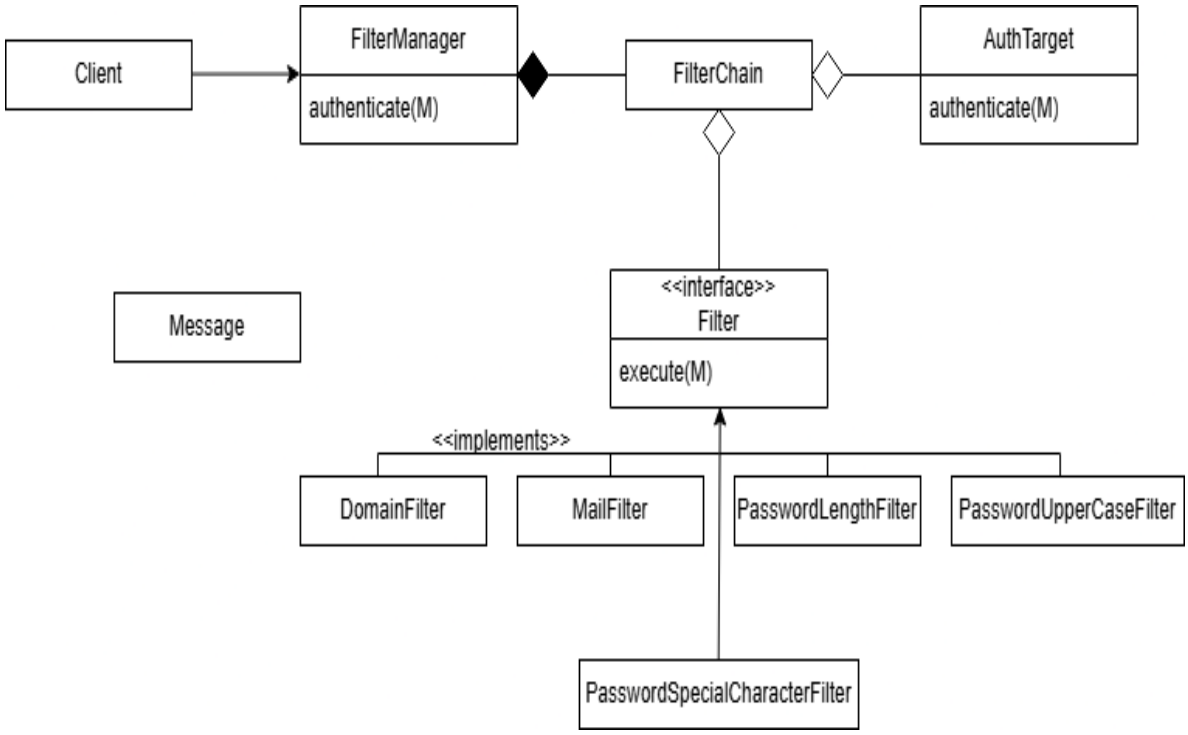
Esta clase se encarga de verificar si tiene al menos una mayúscula la contraseña. Se declara el siguiente método:

- void execute(Message message): Este método de la clase Filter se sobrescribe para verificar si la contraseña tiene al menos una mayúscula.

5.12. Main

El main de este ejercicio lo que hace es agregar al filterManager todos los filtros que se le aplican al correo y contraseña mencionados anteriormente, pide por pantalla el correo y la contraseña y aplica los filtros a las credenciales.

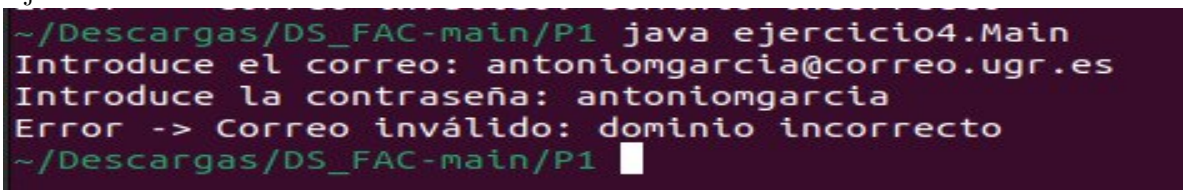
5.13. Diagrama UML



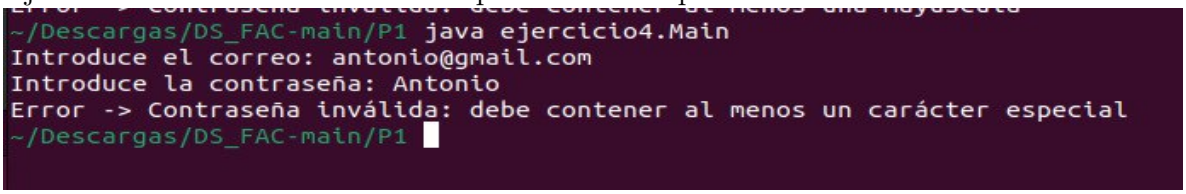
5.14. Muestras de Ejecución

Ejecución con correo válido:

Ejecución con dominio inválido:



Ejecución con contraseña inválida por caracter especial:



Ejecución con contraseña inválida por asuencia de mayúscula: