Patrones de diseño

Práctica 1 - Desarrollo de Software

Autores:

Adam Navarro
Iván Molina
Fernando Lanzarot

UNIVERSIDAD
DE GRANADA

Universidad de Granada
Escuela Técnica Superior de Ingeniería Informática y de Telecomunicación

Grado en Ingeniería Informática

Curso 2023/2024 - Grupo X

Índice

1. Ejercicio 1

1.1. Introducción

En este ejercicio, se ha creado una simulación de carreras de bicicletas utilizando los patrones de diseño **Factoría Abstracta** y **Método Factoría**.
El programa ejecuta dos carreras al mismo tiempo, sin saber cuántas bicicletas habrá hasta que empiece la ejecución. Se han diseñado dos tipos de carreras:

- **Carrera de montaña**: Antes de terminar, se elimina el 20 % de las bicicletas.
- **Carrera de carretera**: Antes de terminar, se elimina el 10 % de las bicicletas.

Ambas carreras duran exactamente 60 segundos, y las bicicletas retiradas salen al mismo tiempo.

1.2. Funcionamiento

Para ejecutar el programa, se debe compilar y ejecutar con el siguiente comando (desde el directorio Ejercicio1):

```
make run ARGS="N N"
```

Donde `N` es el número inicial de bicicletas en cada carrera. El programa crea las carreras y las ejecuta en hebras separadas.

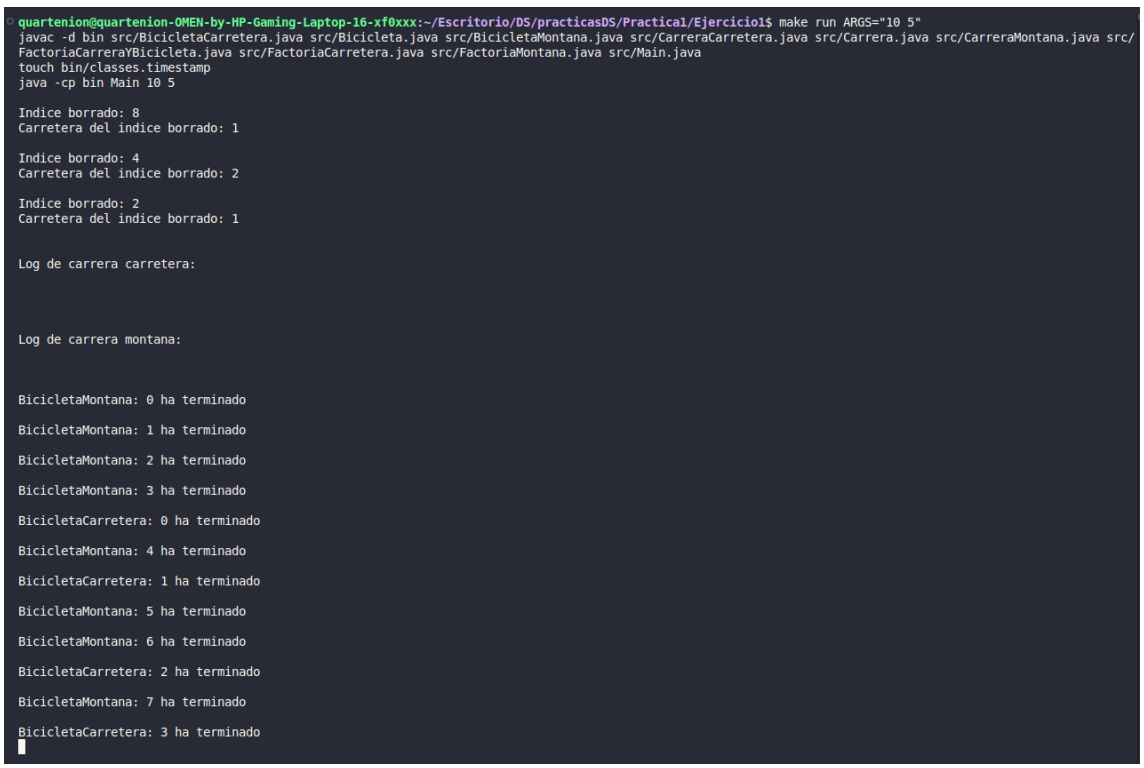


Figura 1: Ejemplo funcionamiento

1.3. Implementación

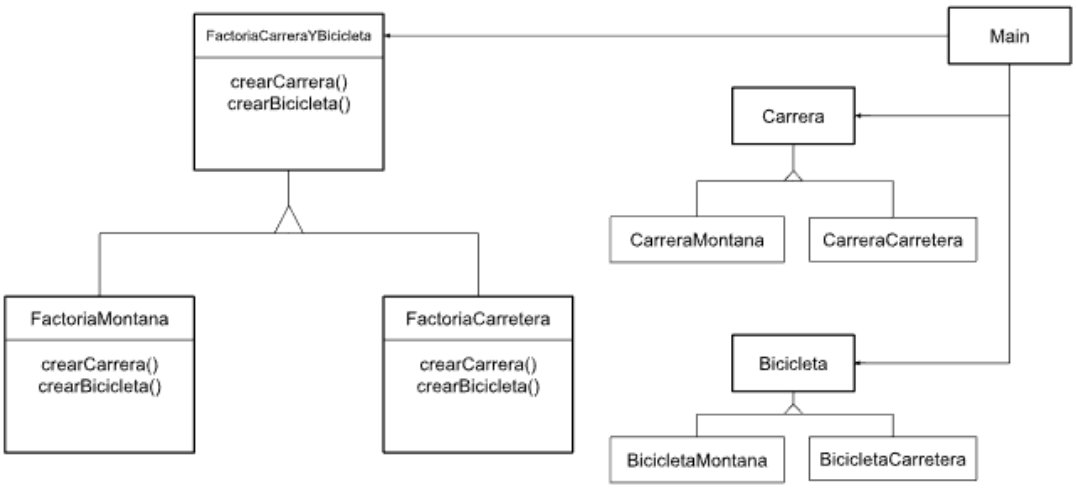


Figura 2: Diagrama UML

Para que el programa siga el patrón de Factoría, el código se ha dividido en varias partes:

- **Interfaz FactoriaCarreraYBicicleta**: Se encarga de definir los métodos para crear carreras y bicicletas.
- **Carreras**: Clases que implementan la lógica de duración de la carrera y eliminación de bicicletas en la carrera. Estas se dividen en CarreraCarretera y CarreraMontana.
- **Bicicletas**: Tipos de bicicleta según el tipo de carrera. Estas se dividen en BicicletaCarretera y BicicletaMontana.
- **Factorías**: Clases que se encargan de crear bicicletas y carreras según el tipo. Estas se dividen en FactoriaCarretera y FactoriaMontana
- **Clase principal (Main)**: Se encarga de la ejecución del programa y de lanzar las carreras en hebras separadas.

1.3.1. Implementación del Main

El método `main` se encarga de leer los parámetros de entrada, crear las carreras y asignar bicicletas. Para esto:

- Se crean dos factorías (`FactoriaCarretera` y `FactoriaMontana`) para generar carreras y bicicletas.
- Se instancian las carreras y se les añaden las bicicletas correspondientes usando las factorías.
- Para ejecutar ambas carreras al mismo tiempo, se utiliza un `ExecutorService` con dos hebras, para ejecutar las carreras en paralelo.

1.3.2. Ejecución en hebras

Para que ambas carreras se ejecuten al mismo tiempo, se ha usado la interfaz `Runnable`. Cada carrera corre en una hebra separada, permitiendo la ejecución en paralelo.

2. Ejercicio 2

Ejercicio sobre el patrón de diseño decorador

2.1. Introducción

Este ejercicio sirve para aplicar el patrón de diseño **decorador** al mismo tiempo que se interactura y aprende un poco el uso de la API de **Hugging Face** para interactuar con los LLM. Se utiliza el patrón decorador ya que se aplicará una transformación base a un texto (BasicLLM) y luego se tendrán unas clases decoradoras que utilizarán modelos LLM para aplicar una determinada transformación, que en este caso serán de traducción y de expansión.
Con este patrón se puede tener una clase con un texto determinado que no será modificado, serán las otras clases las que tendrán acceso al objeto base del que extraerán el texto, lo transformarán y devolverán cuando se les pida.

2.2. Funcionamiento

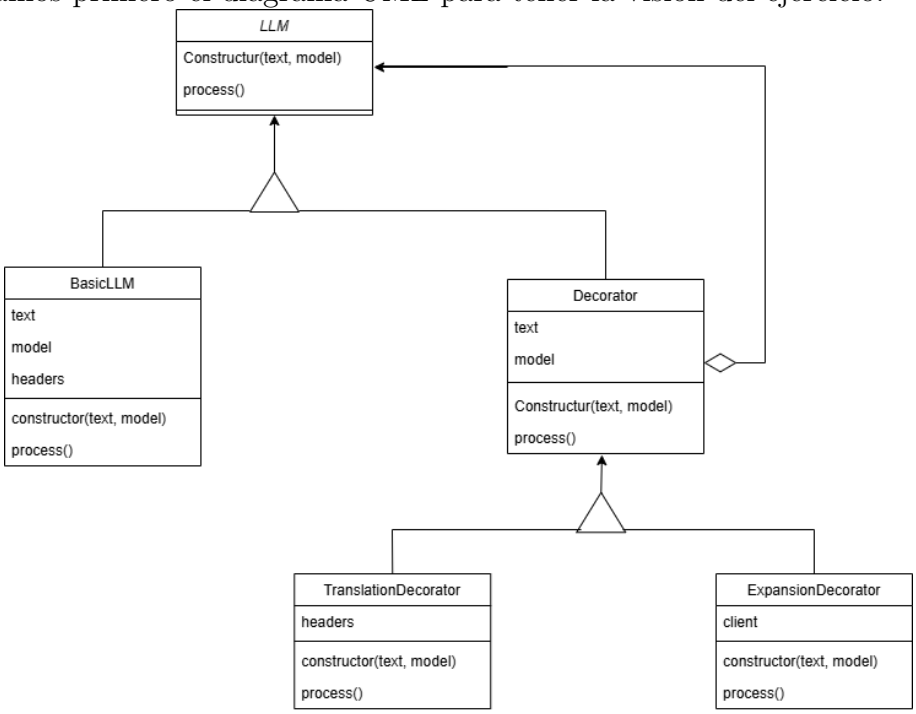
Para ejecutar el código necesitaremos instalar python y en nuestro entorno de python las dos bibliotecas siguientes:

```
pip install requests
pip install huggingface-hub
```

Necesitaremos también crear una cuenta en la página de **Hugging Face** y crearnos un token, el cual guardaremos y usando el comando *huggingface-cli login* en la consola de nuestro entorno e introduciremos nuestro token, de esta forma el código tendrá acceso al token para poder conectarse a la API.

2.3. Realización

Veamos primero el diagrama UML para tener la visión del ejercicio:



Viendo esto el ejercicio consta de 5 archivos en total, teniendo:

```
main.py
llm_interface.py -> Clase interfaz de la que heredarán el resto
llm_basic.py     -> Clase que hereda de la interfaz y que es la
                    base de la que partir
llm_decorators.py-> Contiene la clase decorator y sus subclases,
                    que tomarán un objeto del basic para transformarlo
config.json      -> Contiene la información sobre el texto de entrada y los modelos
```

En esta práctica se ha usado la API de inferencia, es decir, que no se necesitan tener los modelos de manera local descargados, se hacen peticiones como modelo cliente/servidor. Para la clase basic (modelo resumen) y el decorador de traducción se ha usado requests que hace la operación POST para hacer una solicitud, y luego para la expansión se ha usado un modelo que pide usar la biblioteca huggingface-hub, definiendo un çlienteçon InferenceClient que recibe la key, y es este objeto client el que hace la llamada con "text generation". A ejemplos prácticos ambas formas funcionan igual, solo que usan bibliotecas distintas.

2.4. main

En el main vamos a tener algo muy simple, solo se cargan los datos del json y se va a crear los modelos de cada clase para luego utilizar process y mostrar el resultado por pantalla:

```python
if __name__ == '__main__':
    nombre_archivo = 'config.json'
    json = cargar_json(nombre_archivo)

    input_text = json['text']
    summ_model_name = json["model_llm"]
    trans_model_name = json["model_translation"]
    expan_model_name = json["model_expansion"]
    input_lang = json["input_lang"]
    output_lang = json["output_lang"]


    summary_model = BasicLLM(input_text, summ_model_name)
    translation_model = TranslationDecorator(summary_model, trans_model_name, input_lang, output_lang)
    expansion_model1 = ExpansionDecorator(summary_model, expan_model_name)
    expansion_model2 = ExpansionDecorator(translation_model, expan_model_name)

    texto_resumido = (summary_model.process())
    print(f"\n\nEl texto resumido es el siguiente: \n {texto_resumido}")


    texto_traducido = (translation_model.process())
    print(f"\n\nEl texto traducido es el siguiente: \n {texto_traducido}")


    texto_expandido = (expansion_model1.process())
    print(f"\n\nEl texto expandido es el siguiente: \n {texto_expandido}")


    texto_traducido_expandido = (expansion_model2.process())
    print(f"\n\nEl texto traducido y luego expandido es el siguiente: \n {texto_traducido_expandido}")
```

Se puede ver como se crea el básico a partir del texto inicial y luego los decoradores a partir del modelo básico.

2.5. Interfaz y BasicLLM

La interfaz va a definir tanto el constructor que se le introduce texto y modelo como el método process. Este último tiene un nombre bastante genérico, pero tiene sentido porque va a realizar muchas posibles tareas (resumen, traducción, expansión, ...)

```python
# Interfaz para modelos llm
class LLM(ABC):  4 usages  Adam Navarro Megías *
    @abstractmethod  Adam Navarro Megías
    def __init__(self, text, model):
        pass

    @abstractmethod  2 usages (2 dynamic)  Adam Navarro Megías
    def process(self):
        pass
```

La clase basic tiene el constructor con text y model que guarda en variables de la clase, luego crea headers con el token que tenemos en nuestro entorno (se podría poner en código pero es una violación de seguridad absoluta). Luego el método process utiliza requests.post que recibe el modelo (enlace al mismo que viene del json), headers para conexión y el texto en json. [0]["summary-text"] es para acceder al texto, porque se devuelve un listado de json.

```python
# Clase base que va a usar el modelo para generar un resumen del texto de entrada
class BasicLLM(LLM):  1 usage  Adam Navarro Megías *
    def __init__(self, text, model):  Adam Navarro Megías
        self.text = text
        self.headers = {"Authorization": f"Bearer {get_token()}"}
        self.model = model

    def process(self):  3 usages (2 dynamic)  Adam Navarro Megías
        return ((requests.post(self.model, headers=self.headers,
                               json={"inputs": self.text}) ).json())[0]['summary_text']
```

2.6. Decoradores

En los decoradores tenemos la clase abstracta Decorator que implementa la interfaz y que define el constructor creando y asignando las variables de texto y modelo. La variable de texto en este caso es un objeto del BasicLLM, que cuando se vaya a hacer la llamada a la API para procesar se llamará antes al process del objeto basic.

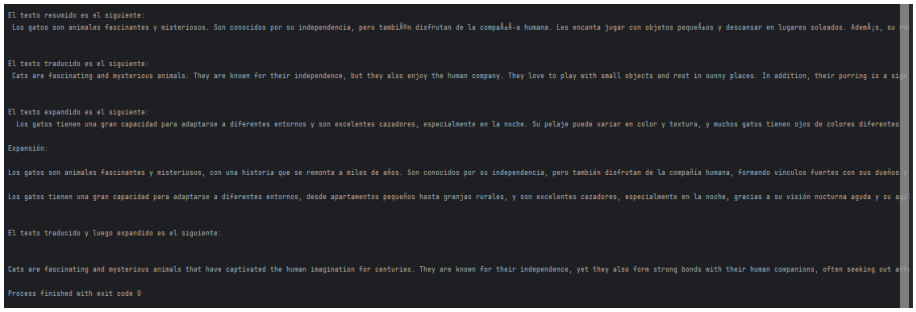
```python
# Clase Decoradora abstractaque será padre de los decoradores específicos
class Decorator(LLM):  2 usages  Adam Navarro Megías *
    def __init__(self, text, model):  Adam Navarro Megías
        self.text = text
        self.model = model

    @abstractmethod  2 usages (2 dynamic)  Adam Navarro Megías
    def process(self):
        pass
```

Los decoradores específicos en este caso son dos, la traducción que la hace un modelo Helsinki usando los idiomas especificados en el json y luego una versión de un modelo Qwen para la expansión, que se le introduce un prompt para que lo haga.

```python
# Decorador específico para la traducción de un lenguaje a otro
# Tener en cuenta que "text" aquí es el objeto no es un texto plano
class TranslationDecorator(Decorator):  1 usage  Adam Navarro Megías *
    def __init__(self, text, model, input_lang, output_lang):  Adam Navarro Megías *
        model_changed = model.replace("input", f"{input_lang}").replace("output", f"{output_lang}")
        super().__init__(text, model_changed)
        self.headers = {"Authorization": f"Bearer {get_token()}"}

    def process(self):  3 usages (2 dynamic)  Adam Navarro Megías *
        return ((requests.post(self.model, headers=self.headers,
                               json={"inputs": self.text.process()}) ).json())[0]['translation_text']



# Decorador específico para extender un texto con más información y datos
class ExpansionDecorator(Decorator):  2 usages  Adam Navarro Megías *
    def __init__(self, text, model):  Adam Navarro Megías *
        super().__init__(text, model)
        self.client = InferenceClient(api_key=get_token())

    def process(self):  4 usages (2 dynamic)  Adam Navarro Megías *
        prompt = f"Expande el siguiente texto con más detalles, en el mismo idioma del texto y solo devuelveme el texto: \n\n{self.text.process()}"
        return self.client.text_generation(prompt, model=self.model, max_new_tokens=300)
```

2.7. Salida y conclusión

La salida del programa es muy simple, se muestra el texto resumido, luego el traducido, luego el expandido y por último el expandido del traducido. Hay que tener en cuenta que a veces puede tardar más de lo normal, especialmente en primeras ejecuciones y también que puede dar fallos de decodificación de json (no conozco la razón), pero si se ejecuta varias veces se debería de ver funcionando sin problema.



Para concluir, este ejercicio implementa el patrón decorador de una manera simple para ilustrar el funcionamiento, a parte de utilizar una pincelada de la API de hugging face para ver las posibles opciones que tiene

3. Ejercicio 3

3.1. Introducción

En este ejercicio, se ha desarrollado un programa en Python que usa el patrón *Estrategia* para extraer información de la web `https://quotes.toscrape.com/`. El objetivo es obtener y procesar las citas de las primeras cinco páginas usando dos métodos: **Selenium** y **Beautiful Soup**.
Este patrón permite cambiar de método de extracción sin modificar el código principal.

3.2. Funcionamiento

Antes de ejecutar el programa, es necesario instalar las dependencias con:

```
pip install beautifulsoup4 pyyaml selenium requests
```

Para ejecutar el programa:

```
python3 main.py
```

El programa solicitará una entrada: **1** para Selenium o **2** para Beautiful Soup.

3.2.1. Ejecución con Selenium

Si se elige **1**, el programa usará **Selenium** para extraer datos.

```
(ejer3) quartenion@quartenion-OMEN-by-HP-Gaming-Laptop-16-xf0xxx:~/Escritorio/DS/practicasDS/Practica1/Ejercicio3$ python3 main.py
Elige la estrategia:
1)SELENIUM
2)BEAUTIFUL SOUP
1
Datos guardados en "quotes_data_selenium.yaml"
(ejer3) quartenion@quartenion-OMEN-by-HP-Gaming-Laptop-16-xf0xxx:~/Escritorio/DS/practicasDS/Practica1/Ejercicio3$
```

Figura 3: Inicio de la ejecución con Selenium

Selenium interactúa con la página web y extrae las citas:

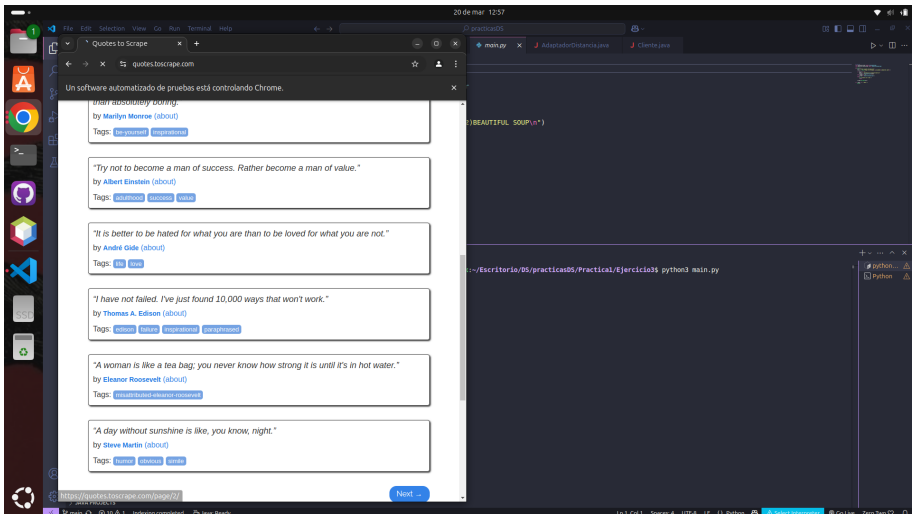


Figura 4: Extracción de datos con Selenium en ejecución

Los datos se guardan en un archivo YAML con el siguiente formato:

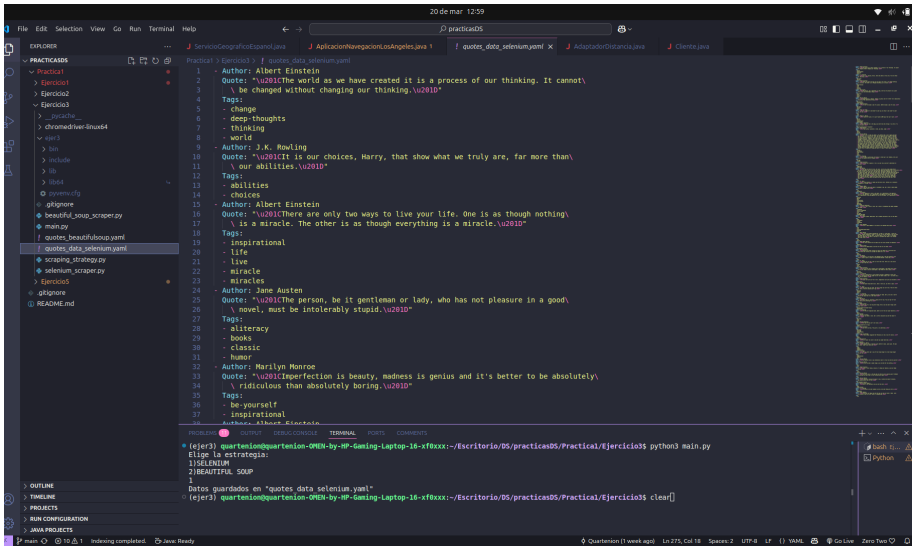


Figura 5: Archivo YAML generado con Selenium

`quotes_data_selenium.yaml` almacena la información extraída.

3.2.2. Ejecución con Beautiful Soup

Si se elige **2**, el programa usará **Beautiful Soup** para extraer datos.

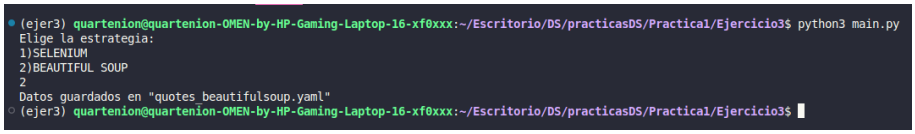


Figura 6: Ejecución del programa con Beautiful Soup

Los datos se almacenan en un archivo YAML con este formato:

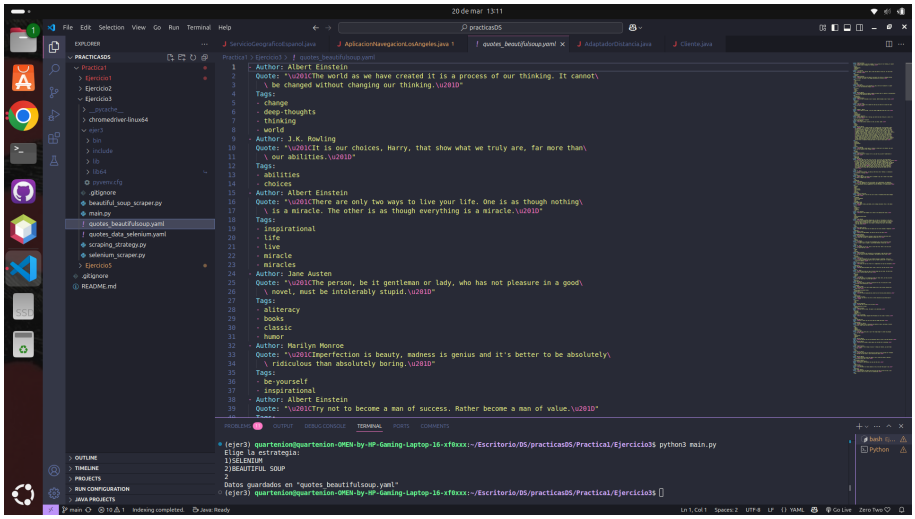


Figura 7: Archivo YAML generado con Beautiful Soup

`quotes_data_bs4.yaml` almacena los datos extraídos.

3.3. Implementación

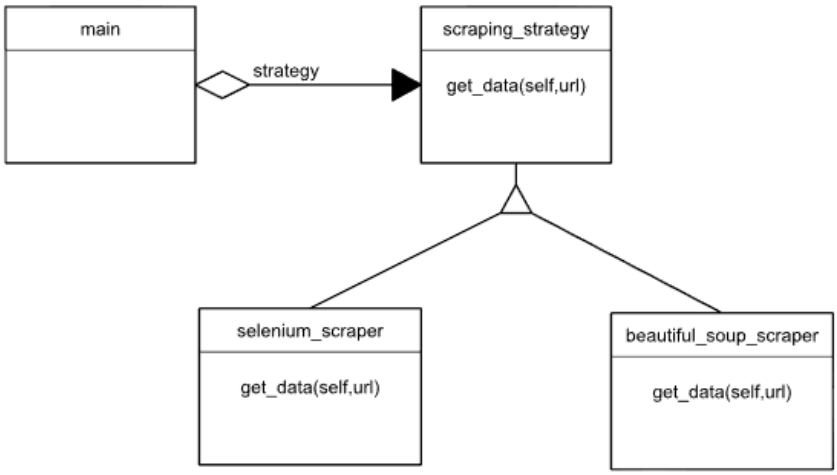


Figura 8: Diagrama UML

Para implementar el patrón de estrategia, los datos se organizan en cuatro archivos diferentes:

- **scraping_strategy.py**: Contiene una clase abstracta que define un método a ser implementado por las clases que hereden de ella.
- **selenium_scraper.py**: Implementa Selenium para extraer citas de la web de forma automatizada.
- **beautifulsoup_scraper.py**: Implementa Beautiful Soup para extraer las citas.
- **main.py**: Ejecuta el programa y permite elegir entre los dos métodos de extracción.

3.3.1. Explicación de Selenium

En `selenium_scraper.py`, se ha configurado un driver de Chrome para navegar por la web. Se ha utilizado un XPath para localizar los elementos que contienen citas, autores y etiquetas. Para obtener estos XPath, se ha inspeccionado la página con las herramientas de desarrollo del navegador, identificando los elementos clave y copiando sus rutas.
Un XPath sigue esta estructura:

```
//etiqueta[@atributo="valor"]
```

Por ejemplo, el código usa `.//span[@class="text"]` para encontrar el texto de una cita dentro de su contenedor.

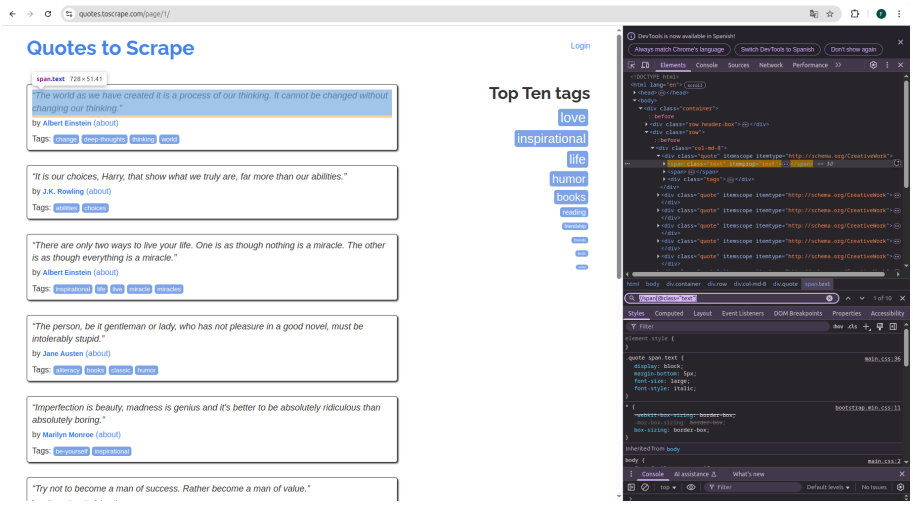


Figura 9: Ejemplo de búsqueda de XPath

Los datos obtenidos se han almacenanado en una lista de diccionarios, donde cada diccionario representa una cita con sus datos (texto, autor y etiquetas). El script avanza por varias páginas pulsando el botón "Siguiente"hasta completar la cantidad deseada. Finalmente, la información se guarda en un archivo YAML.

3.3.2. Explicación de Beautiful Soup

En `beautifulsoup_scraper.py`, el programa descarga el código HTML de la página y extrae las citas sin necesidad de abrir un navegador.

Busca los datos dentro del código usando clases CSS:

- El texto de la cita está en un `<span>` con clase `"text"`.
- El autor está en un `<small>` con clase `.ªuthor"`.

A diferencia de Selenium, que hace clic en "Siguiente", este método cambia la URL manualmente para avanzar de página.
Los datos se guardan en un archivo YAML con el mismo formato que Selenium.
Beautiful Soup es más rápido, pero no funciona bien si la página usa JavaScript para cargar contenido.

4. Ejercicio 4

Ejercicio sobre el patrón de filtros de intercepción

4.1. Introducción

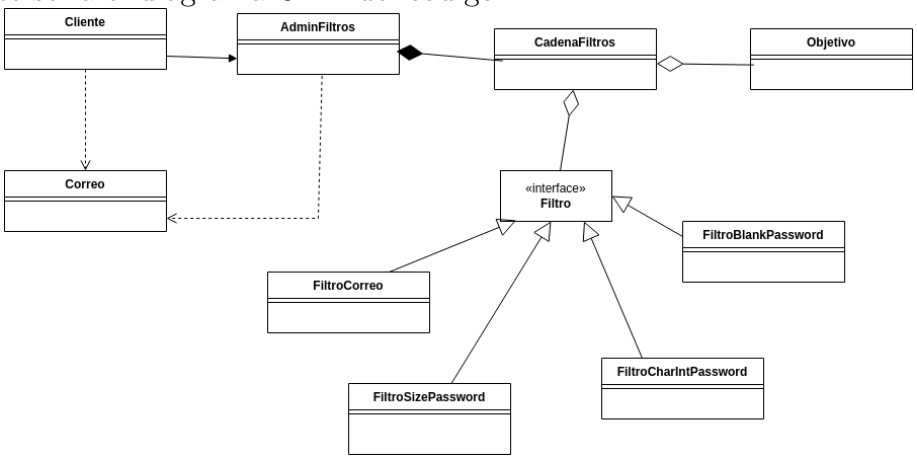
Este ejercicio trata de aplicar el patrón de diseño de filtros de intercepción. Se utiliza este patron ya que incorpora servicios de manera transparente mediante intercepciones. Mejora la reutilizacion de código al permitir la combinación de filtros.
La idea es que en base a un correo y una contraseña al enviarse se pueda comprobar sin variar el codigo si con una serie de reglas se cumplen los filtros al objetivo final.

4.2. Funcionamiento

Para ejecutar el codigo es muy sencillo, simplemente se abre el proyecto en netbeans y en el main se añaden los correos y contraseñas a los que se quiere ver si cumplen los filtros descritos con anterioridad

4.3. Realización

ESte sería el diagrama UML del codigo:



```
main.java

Cliente.java -> Clase desde donde se enviar los correos

Correo.java -> Clase que tiene como atributos el email y contraseña que pasaran por

Objetivo.java -> Clase que recibe el correo e imprime si paso los filtros

AdminFiltros.java -> Esta clase se encarga de p

CadenaFiltros.java -> Esta clase se encarga de almacenar los filtros, de saber cual

Filtro.java -> Interfaz de la que heredan todos los filtros

FiltroCorreo.java -> Observa si el email cumple el filtro y devuelve verdadero o fa

FiltroSizePassword.java -> Observa si el tamaño de la contraseña es mayor de 8

FiltroChatIntPassword.java -> Observa si hay al menos una minuscula, una
mayuscula, y un entero

FiltroBlankPassword.java -> Se asegura que no hay ningun espacio en blanco
```

En esta práctica se creo un objeto Objetivo, al cual se le asigna un admin filtro. Ha ese admin filtro se le cargan (Añaden) los filtros que se le quiera aplicar, despúes se crea el objeto CLiente (El que envia el correo) y se envian, al hacer eso se mostrará por el output si el email y la constraseña se aceptan según los filtros, en este caso el correo debe tener una letra antes del @, y despues gmail/hotmail .com, para la contraseña tienen que ser minimo de 8 de longitud, contener minuscula, mayuscula y un numero como minimo.

4.4. Main y output

```java
package practicalejercicio4;

public class PracticalEjercicio4 {

    public static void main(String[] args) {
        Objetivo correoFiltro = new Objetivo();

        AdminFiltros admin = new AdminFiltros(correoFiltro);

        //Que tenga algo antes de @ algo y luego de @ gmail.com o hotmail.com
        admin.addFiltro(new FiltroCorreo());

        //Que el tamaño de la contraseña sea minimo 8 caracteres
        admin.addFiltro(new FiltroSizePassword());

        //Que en la contraseña minimo haya una minuscula, una mayuscula y un numero
        admin.addFiltro(new FiltroCharIntPassword());

        //No haya espacion en blanco
        admin.addFiltro(new FiltroBlankPassword());

        Cliente correo = new Cliente(admin);

        correo.enviarCorreo("ivanMol@gmail.com", "123442W@");
        correo.enviarCorreo("ivanMol@hotmail.com", "Hjkhuh234.");
        correo.enviarCorreo("ivanMol@correo.com", "aa33aA44aafrt");
        correo.enviarCorreo("ivanMol", "12345678iuytrew");
    }

}
```

Como observamos en el main se crea un objeto objetivo, se crea un adminFiltros con el objetivo pasado como parametro, y se le añaden todos los filtros que se quieran utilizar, por último se crea el CLiente y se le añade el adminFiltro y ya es enviar los correos a los que se quiere aplicar los filtros

```
Correo rechazado: ivanMol@gmail.com
Correo aceptado: ivanMol@hotmail.com
El correo es ivanMol@hotmail.com, Hjkhuh234.
Correo rechazado: ivanMol@correo.com
Correo rechazado: ivanMol
```

Se puede observar como el primero es rechazado aunque el email este bien, ya que la contraseña esta mal, faltaría una minuscula. El segundo si es aceptado y muestra tanto el email como la contraseña al hacerlo. El tercero esta mal el correo ya que no es gmail ni hotmail, por lo que ni siquiera comprueba la contraseña y el cuarto igual que el tercero.

4.5. AdminFiltros y CadenasFiltros

```java
package practicalejercicio4;

public class AdminFiltros {
    private CadenaFiltros cadenaFiltros;

    public AdminFiltros(Objetivo correoFiltro){
        cadenaFiltros = new CadenaFiltros();
        cadenaFiltros.setObjetivo(correoFiltro);
    }

    public void addFiltro(Filtro filtro){
        cadenaFiltros.addFiltro(filtro);
    }

    public void postCorreo(Correo correo){
        cadenaFiltros.ejecutar(correo);
    }
}
```

A la clase AdminFiltros se le encarga la tarea de añadir filtros lo cual se delega en la clase CadenaFiltros donde se encuentra el array de todos los filtros añadidos. Tambien se encuentra el metodo postCorreo que es el que se encarga de ejecutar el Objetivo.

```java
package practica1ejercicio4;

import java.util.ArrayList;
import java.util.List;

public class CadenaFiltros implements Filtro {
    private List<Filtro> filtros = new ArrayList<>();
    private Objetivo correoFiltro;

    public void addFiltro(Filtro filtro) {
        filtros.add(filtro);
    }

    public void setObjetivo(Objetivo correoFiltro) {
        this.correoFiltro = correoFiltro;
    }

    @Override
    public boolean ejecutar(Correo correo) {
        for (Filtro filtro : filtros) {
            if (!filtro.ejecutar(correo)) {
                System.out.println("Correo rechazado: " + correo.getEmail());
                return false;
            }
        }

        System.out.println("Correo aceptado: " + correo.getEmail());
        correoFiltro.string(correo);
        return true;
    }
}
```

A la clase CadenaFiltros se le asigna un Objetivo desde AdminFiltros y se le van añadiendo uno a uno. También tiene el método ejecutar que recorre todos los filtros y en caso de que no pase los filtros rechaza el correo, y si los pasa todos lo acepta e imprime el email y la contraseña del mismo.

4.6. Filtros

Como podemos ver en la siguiente captura el **filtro del correo** simplemente de definir la expresion regular en la que se puede poner cualquier simbolo antes de @, y despues del mismo un gmail.com o hotmal.com. Por ultimo devuelve true o false si se cumple.

```java
package practica1ejercicio4;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class FiltroCorreo implements Filtro{

    @Override
    public boolean ejecutar(Correo correo) {
        String regex = "^[^@]+@(gmail|hotmail)\\.com$";
        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(correo.email);
        return matcher.matches();
    }

}
```

El siguiente filtro revisa si hay espacios en blanco en la contraseña ya que no se puede, por lo que comprueba si contiene . Y devuelve true o false dependiendo de si encontro o no

```java
package practica1ejercicio4;

public class FiltroBlankPassword implements Filtro {
    @Override
    public boolean ejecutar(Correo correo) {
        return !correo.getPassword().contains(" ");
    }
}
```

El filtro de size comprueba si la contraseña es de un tamaño igual o superior a 8 y dependiendo de ello devuelve true o false.

```java
package practica1ejercicio4;

public class FiltroSizePassword implements Filtro {
    @Override
    public boolean ejecutar(Correo correo) {
        return correo.getPassword().length() >= 8;
    }
}
```

El ultimo filtro añadido trata de con la expresion regular controlar que haya minimo una minuscula, una mayuscula y un numero, en caso de que se cumpla estos minimos devolvera true, en otro caso false.

```java
package practicalejercicio4;

public class FiltroCharIntPassword implements Filtro {
    @Override
    public boolean ejecutar(Correo correo) {
        String regex = "^(?=.*[a-z])(?=.*[A-Z])(?=.*\\d).+$";
        return correo.getPassword().matches(regex);
    }
}
```

Todos estos filtros son estrictos, es decir con uno que no se cumpla ya no sigue revisando ya que sería incorrecto. Se aplican uno a uno y si todos devuelven true significará que esta todo bien. EStan modularizados permitiendo que sea mucho mas sencillo poner mas filtros ya que solo habria que crearlo y añadirlo a AdminFiltro.