Universidad de Granada

Facultad de Ingeniería Informática

Practica 1: Patrones de diseño

UNIVERSIDAD DE GRANADA

DS: Grupo 1.7

Emanuel Giraldo Herrera
Thomas Lang
Timur Sorokin
Alejando Iborra Morán

(2023-2024)

1 Ejercicio 1 (Java)

1.1 Análisis

Se trata de realizar una simulación de carreras de bicicletas con las siguientes características:

- **Bicicleta**
  Cada bicicleta tiene un identificador. Existen dos tipos de bicicletas:
  - Carretera
  - Montaña
- **Carrera**
  No se conoce el número de bicicletas antes de que empiece la carrera. Además, para cada tipo de bicicleta existe su tipo de carrera con las siguientes peculiaridades:
  - Carretera
    Antes de finalizar 10% bicicletas abandonan la carrera
  - Montaña
    Antes de finalizar 20% bicicletas abandonan la carrera.
  - Hebras
    Se utilizarán las hebras para la simulación simultanea de varias carreras

Finalmente, los objetivos que se buscan son aplicar dos patrones de diseño: Factoría Abstracta y Factoría método.

1.2 UML

Antes de realizar la implementación primero hemos de plantear el diagrama UML que relacione las entidades del problema y refleje sus interacciones.

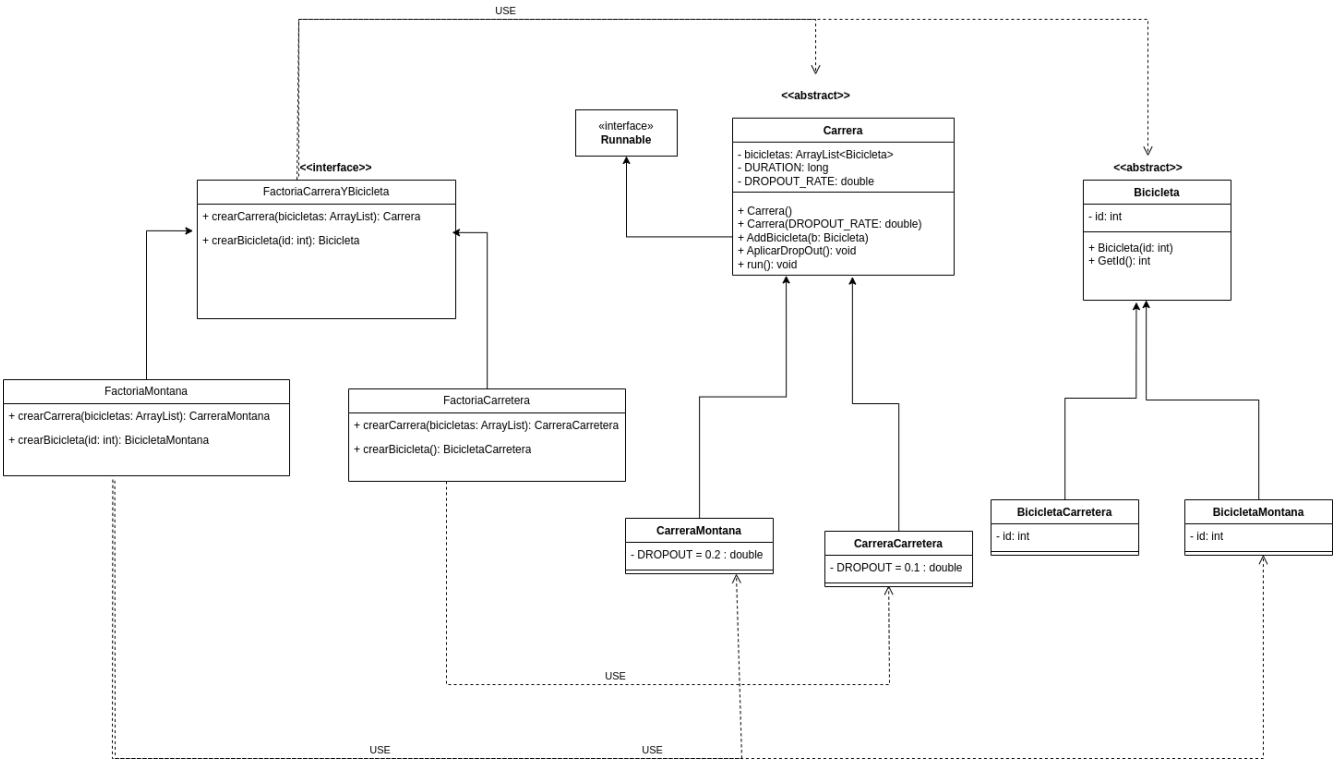

Figure 1: Factoría Abstracta

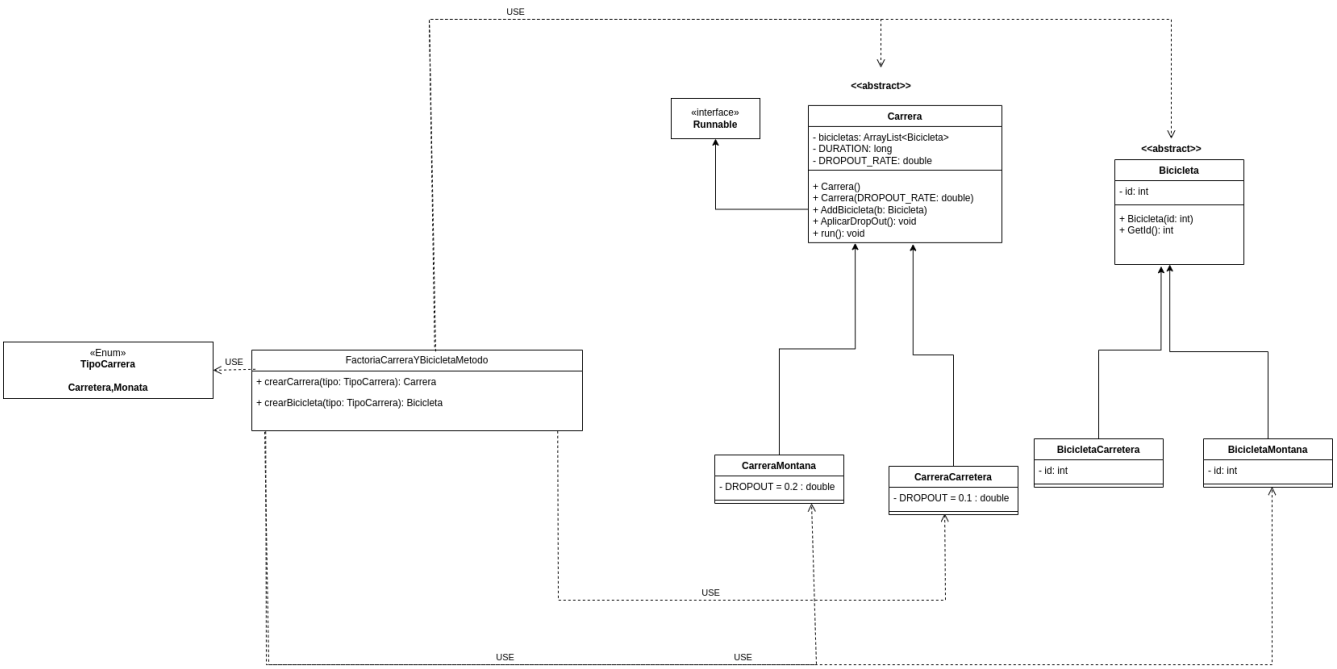

Figure 2: Factoría Método

1.3 Factoría Abstracta

Se trata de un patrón creacional que define una interfaz para crear objetos relacionados sin especificar las clases concretas de esos objetos. Así conseguimos que el código cliente pueda instanciar los objetos sin conocer sus tipos concretos. Esto nos permite desacoplar las implementaciones concretas del código cliente.

En nuestro problema tenemos dos entidades básicas que son *Bicicleta* y *Carrera*. Hemos visto durante el análisis que existe una relación entre estas dos clases, es decir, en una *Carrera* compiten las *Bicicletas*. También vimos que ambas entidades pueden ser de dos tipos: *carretera* y *montaña*. Cada uno de ellos con sus características *concretas*. Entonces, tenemos que en una *Carrera de montaña* participan *Bicicletas de montaña*. Y en una *Carrera de carretera* participan *bicicletas de carretera*. Por tanto distinguimos los siguientes elementos:

- ***Bicicleta***
  Es una entidad que representa una bicicleta abstracta y actúa como una base para los distintos tipos de bicicletas que pueden existir.
  - **BicicletaCarretera**
    Es una bicicleta concreta con características añadidas que son propias de carretera.
  - **BicicletaMontana**
    Es una bicicleta concreta con características añadidas que son propias de montaña.
- ***Carrera***
  Al igual que en el caso de *Bicicleta* se trata de una clase abstracta pues engloba ciertas características comunes a todos los tipos de carreras que pueden existir.
  - **CarreraCarretera**
    Es una carrera concreta en la que participan bicicletas de carretera.
  - **CarreraMontana**
    Es una carrera concreta en la que participan bicicletas de montaña.

- ***FactoriaCarreraYBicicleta***
  Es una interfaz que define dos operaciones: crear bicicleta y crear carrera. No obstante, no especifica el tipo concreto del objeto que se va a crear.
  - **FactoriaCarretera**
    Es una factoría especializada en crear las carreras y bicicletas de carretera
  - **FactoriaMontana**
    Es una factoría especializada en crear las carreras y bicicletas de montaña

1.4 Consideraciones sobre Factoría Abstracta

- La clase *Carrera* implementa interfaz Runnable para representar código que puede ser ejecutado por un hilo independiente para conseguir la concurrencia de las carreras.
- La clase *Carrera* es abstracta pero no tiene ningún método abstracto puesto que, según el enunciado dado, todas las carreras tienen el mismo comportamiento que se resumen en añadir bicicleta, aplicar dropout y simular la competición. Entonces parece más coherente definir un comportamiento predeterminado en la clase base/abstracta que repetir el mismo código en diferentes archivos. Además, se mantiene como abstracta puesto que preferimos restringir la instanciación de esta clase.
- Se ha aplicado el mismo razonamiento para la *Bicicleta*. Aunque los dos tipos de bicicletas que hay podrían ser suprimidos debido a que no aportan nada al problema pues solo se diferencian en el dato miembro *id*. Quizá en una simulación más compleja en la que, por ejemplo, la velocidad máxima este definida por el tipo de bicicleta tendría más significado la separación de los tipos.

1.5 Factoría Método

Es un patrón creacional que permite crear objetos sin especificar la clase. En vez de crear los objetos de forma directa lo que haremos es usar un método para crear y devolverlos.

Para ello se ha optado por definir una nueva entidad *FactoriaCarreraYBicicletaMetodo* en la que se definen dos operaciones: *crearBicicleta(TipoCarrera)* y *crearCarretera(TipoCarrera)*. En ambas mediante un switch basado en *TipoCarrera* se decide el tipo de objeto se debe crear siendo *TipoCarrera* un enum trivial.

1.6 Solución

En la carpeta src/EJ1 ofrecemos una solución al problema planteado. En esta carpeta se puede consultar los archivos correspondientes a cada entidad del digrama UML.

En el archivo *main.java* se han definido dos funciones *FinalFactoriaAbtracta* y *FinalFactoriaMetodo* en las que se muestra el uso de los dos patrones implementados.

1.7 Ejecución

En la ejecución del programa podemos ver como se simulan las carreras simultáneamente utilizando hebras, primero con el patrón *Factoría Abstracta*, y posteriormente, haciendo uso del patrón *Factoría de Método*, tal y como hemos comentado.

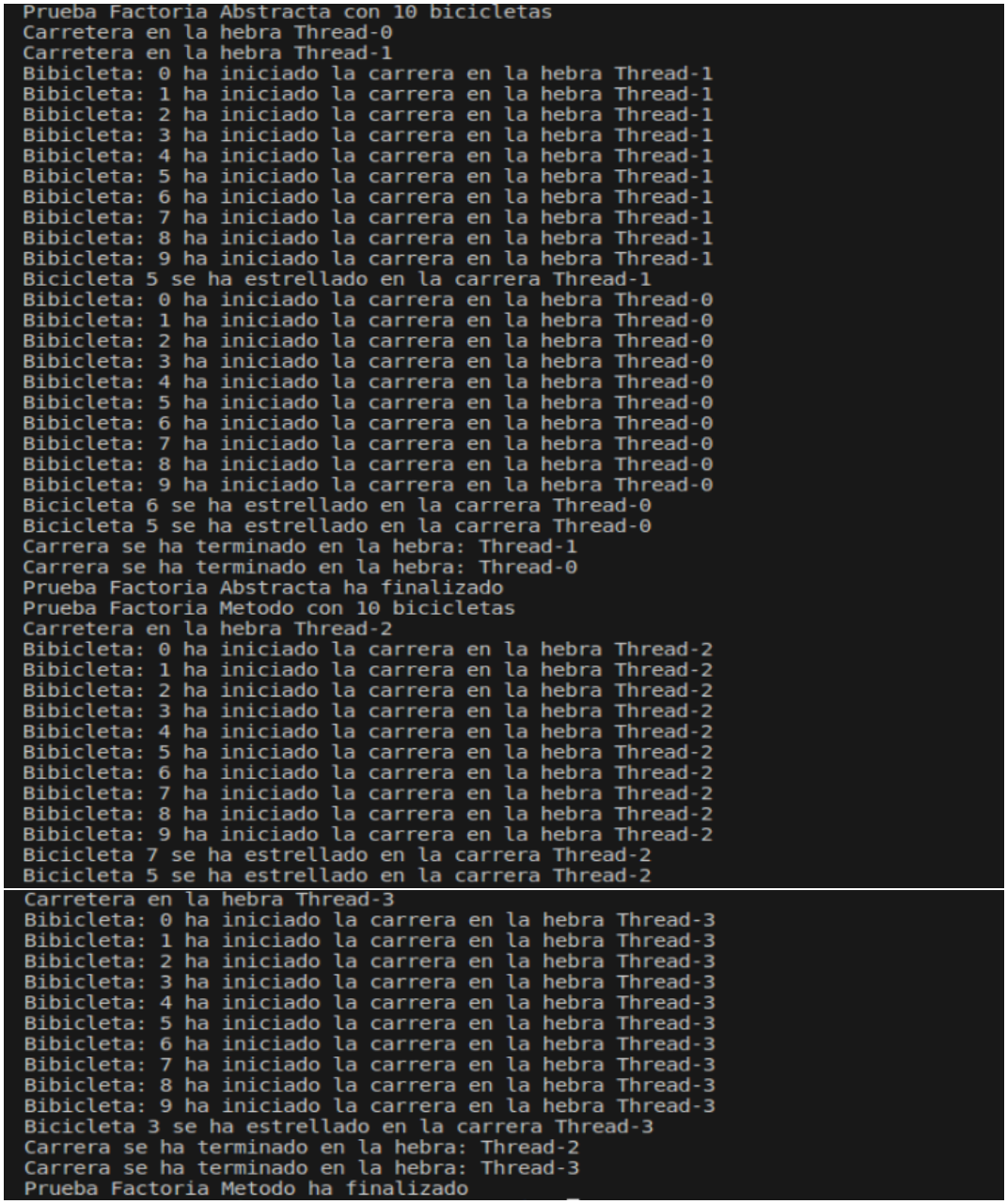

Figure 3: Ejecucion del ejercicio 1

2 Ejercicio 2

2.1 Análisis

En este problema se pide realizar la implementación de Factoria Abstracta del ejercicio 1 en el lenguaje Python sin utilizar las hebras y además realizar la implementación del patron *Prototipo*.

2.2 Factoria Abstracta

Utilizamos el modulo *abc* para poder definir clases abstractas. Las clases abstractas son clases que no pueden ser instanciadas directamente, sino que se utilizan como base para otras clases que las heredan. El planteamiento no cambia en absolutamente nada en comparación con el ejercicio 1 siendo la única diferencia el lenguaje de programación y su sintaxis subyacente.

2.3 Prototipo

Este patrón consiste en crear nuevos objetos clonando otro existente. En nuestro caso vamos a definir un método para la entidad *Bicicleta* que nos permita realizar una copia del objeto existente. Puesto que hablamos de copiar los objetos es necesario recordar los conceptos *shallow copy y deep copy*.

- *Shadow Copy* copia solo la estructura del objeto pero comparte los datos miembros entre el original y la copia, lo que hace que los cambios afecten a ambos. Es más rápida y consume menos memoria.
- *Deep Copy* crea copias independientes evitando la compartición de memoria por tanto los cambios en una copia no afectan al objeto original ni viceversa. Es más lenta y consume más memoria.

Para implementar metodo *clone()* podemos recurir al *constructor de copia* en el que realizaremos una copia profunda del objeto recibido como argumento. O alternativamente podemos usar modulo *copy* que nos dá interfaces para duplicar los objetos. Nosotros hemos optado por este último.

2.4 Ejecución

```
==========TEST FACTORIA CARRETERA==========
Carrera de Carretera con 10 bicicletas y dropout 0.2 ha iniciado
Bicicleta 1 ha estrellado
Bicicleta 0 ha estrellado
Carrera Terminada con 8 bicicletas
==========FIN==========
==========TEST FACTORIA MONTAÑA==========
Carrera de Montana con 10 bicicletas y dropout 0.1 ha iniciado
Bicicleta 0 ha estrellado
Carrera Terminada con 9 bicicletas
==========FIN==========
==========TEST PROTOTIPO==========
Carrera de Montana con 10 bicicletas y dropout 0.1 ha iniciado
Bicicleta 1 ha estrellado
Carrera Terminada con 9 bicicletas
Carrera de Carretera con 10 bicicletas y dropout 0.2 ha iniciado
Bicicleta 2 ha estrellado
Bicicleta 3 ha estrellado
Carrera Terminada con 8 bicicletas
==========FIN==========
```

Figure 4: Ejecucion del ejercicio 2

3 Ejercicio 3

3.1 Análisis

En este ejercicio se propone realizar diseño e implementación de una aplicación que utilice un patrón de diseño libre. Para realización de este ejercicios proponemos el siguiente enunciado:

La compañía de un prestigioso juego de rol nos ha encargado crear un programa sencillo que permita a los usuarios crear y visualizar los atributos de los personajes iniciales que pueden crear.

Cada personaje tendrá los siguientes atributos:

- **Primarios:** Cada atributo primario es un valor numerico positivo, y su suma sera igual a 56. Los valores de cada atributo primario cambiaran dependiendo de la raza elegida para el personaje
  - Fuerza
  - Destreza
  - Resistencia
  - Inteligencia
  - Sabiduria
  - Carisma
  - Percepcion
- **Secundarios:** Cada atributo secundario se calculara segun las operaciones anteriores, las cuales vendran modificadas por la clase del personaje
  - Vida (Resistencia + fuerza)
  - Stamina (Destreza + Resistencia)
  - Mana (Inteligencia + Sabiduria)
  - Persuasion (Carisma + Sabiduria)
  - Agilidad (Destreza + Inteligencia)
  - Intimidacion (Fuerza + Carisma)
  - Critico (Percepcion + Inteligencia)
  - Punteria (Destreza + Percepcion)
- **Nombre**
- **Raza**
  - Humano
  - Orco
  - Elfo
  - Enano
- **Clase**
  - Caballero
  - Ladron
  - Mago
  - Ranger

Se pide que aquellas clases del programa que solo se usen una vez sean únicas y que se evite instanciar de mas. También se pide que las funciones para preguntar al usuario y las funciones de creación de personajes estén separadas.

3.2 UML

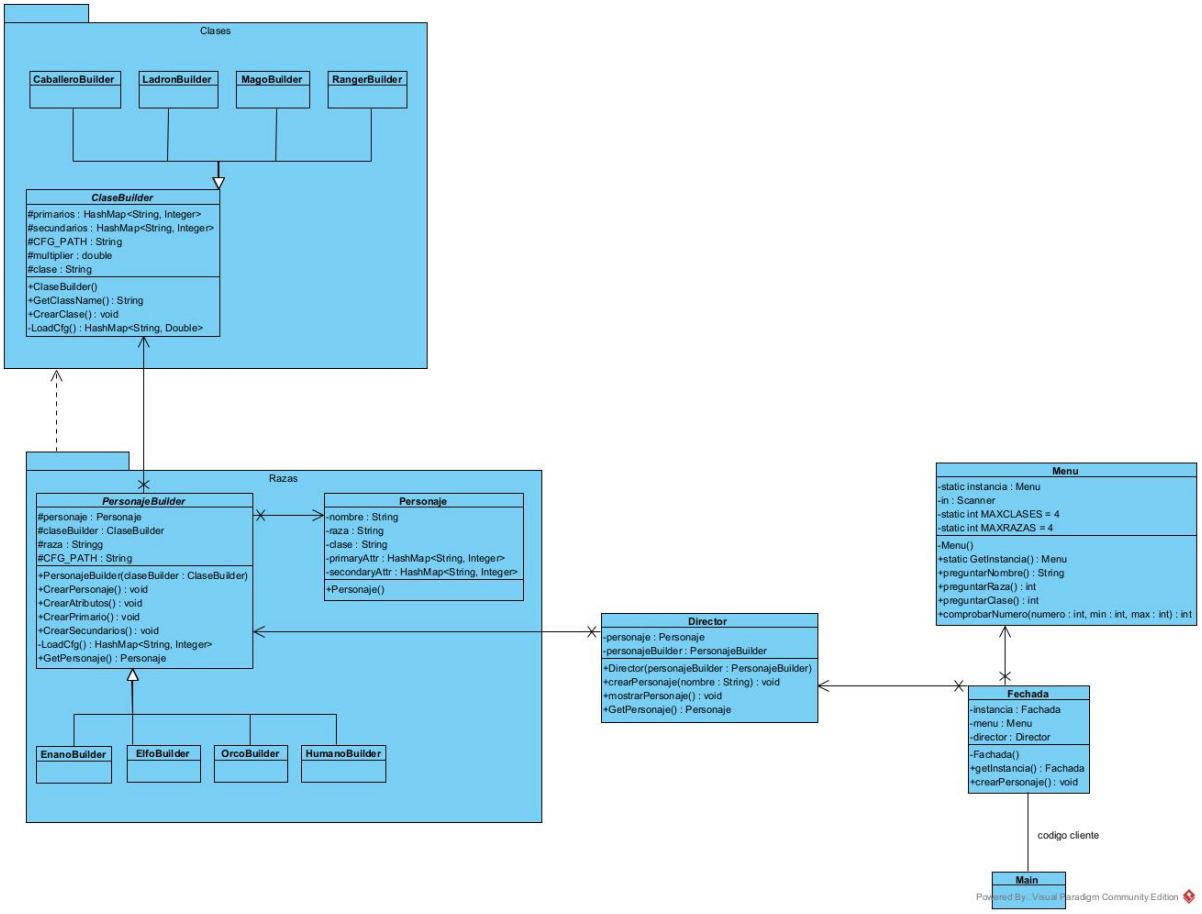

Figure 5: ejercicio 3

3.3 Consideraciones

- La clase *ClaseBuilder* es una clase abstracta que actúa como base para las clases hijas. De modo que hemos optado por dar una funcionalidad predeterminada a los métodos para encapsular un comportamiento común y evitar repetir el mismo código en diferentes archivos. De ser necesario una modificación consideramos que es adecuado que cada clase hija mediante override especialice su comportamiento.

- Mismo razonamiento se ha aplicado a *PersonajeBuilder*

- Utilizamos clase *Director* para abstraer el proceso de creación del personaje, es decir, esta clase se encarga de orquestar los pasos necesarior para construir el objeto de tipo *Personaje*.

- Utilizamos *Menu* para recoger el input del usuario. Se trata de un *Singleton* pues pensamos que al interactuar a traves de la terminal no tiene sentido que existan muchos objetos de *Menu* instaciados.

- Por último, utilizamos clase *Fachada* ofrecemos una interfaz unificada para la interacción con el sistema. De esta forma simplificamos el uso de las clases por el código cliente.

- Como conslución, hemos obtenido un sistema con bajo nivel de acoplamiento. Cada módulo es responsable de una tarea especifica con su lógica adecuada

3.4 Ejecución

La aplicación le preguntará al usuario por el nombre del personaje, y le permitirá escoger qué raza y clase quiere ser. Tras obtener respuesta, el programa deberá crear el personaje según los parametros elegidos, y mostrar por pantalla el valor el nombre del personaje, su raza y su clase, al igual que todos sus atributos con su valor.

```
Escriba el nombre del personaje: Luke Skywalker
Eliga la raza del personaje:
1: Humano
2: Elfo
3: Enano
4: Orco
3
Eliga la clase del personaje:
1: Caballero
2: Ranger
3: Mago
4: Ladron
3
Inicializando Clase...
Inicializando Clase...
Creando personaje...
Creando personaje: enano
Creando clase: mago

Personaje: Luke Skywalker, mago enano

-- Atributos primarios:
----- Fuerza: 10
----- Destreza: 6
----- Resistencia: 9
----- Inteligencia: 6
----- Sabiduria: 9
----- Carisma: 6
----- Percepcion 10

-- Atributos secundarios:
----- Vida: 9
----- Estamina: 11
----- Mana: 22
----- Persuasion: 15
----- Agilidad: 12
----- Intimidacion: 16
----- Critico: 20
----- Punteria: 16
```

Figure 6: Ejecucion del ejercicio 3

4 Ejercicio 4

4.1 Analyse

In this exercise we had to program a simulator of an engine based on the revolutions per minute (RPM). The imposed model to use for this execise is the Intercepting filter model (filtros de intercepción) which will be further explained in the UML Flowchart chapter.

The program should have an Graphical interface (GUI):

One interactive window with the possibility to

- start/stop the engine
- start/stop accelerating
- start/stop braking;

One window which updates in real time

- RPM
- total kilometres driven
- kilometers driven since the last engine start
- current speed in km/h

4.2 UML and Flowchart

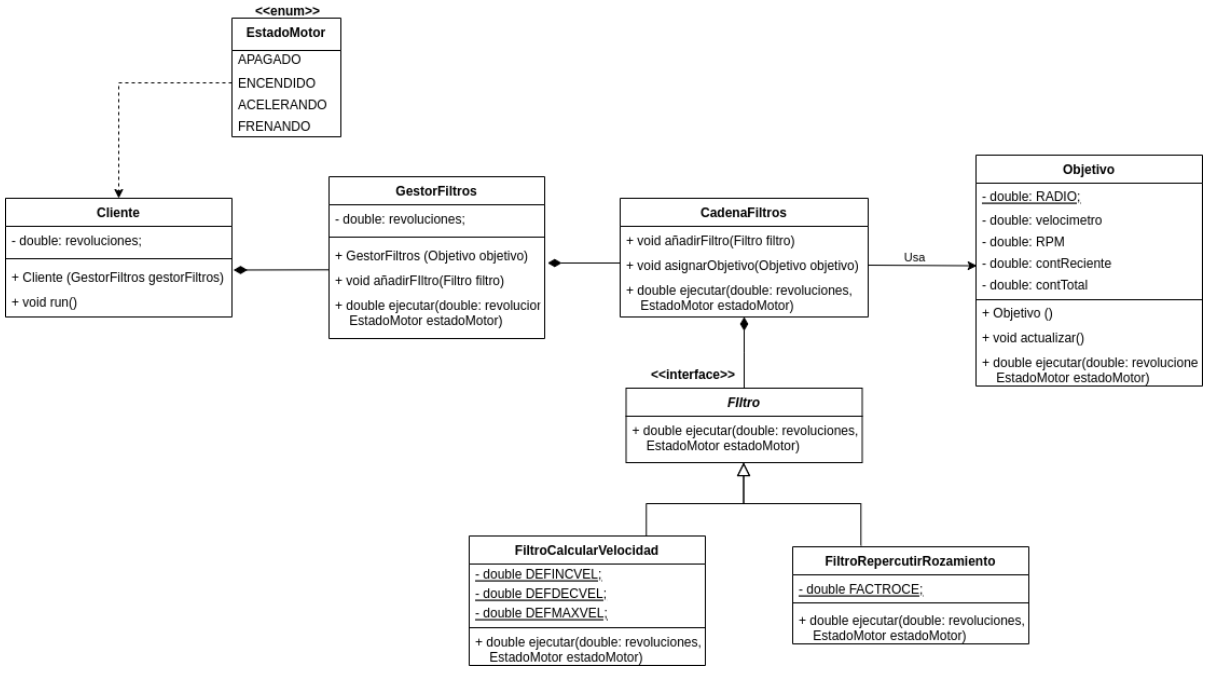

Figure 7: ejercicio 4

The intercepting filter model is best explained using the flowchart (Figure 8). The heart brain of the program is the "Gestor Filtros". It gets called by the client which in our case is the GUI with the buttons Start/stop, accelerate and break. The GestorFiltro in our case simulates the engine but also creates the object of the class CadenaFiltros and adds the needed Filters to the array in the CadenaFiltros object. The filters are xused to give constraints to the

command we want to execute. In our case we have two filters one adding/decreasing rpm and one that calculates the friction. The filters return their values to the object CadenaFiltro and after all filters are applied the values go back to GestorFiltro which sends it to the Objetivo which is in our case the Salpicadero where our computed value can be printed on the window.

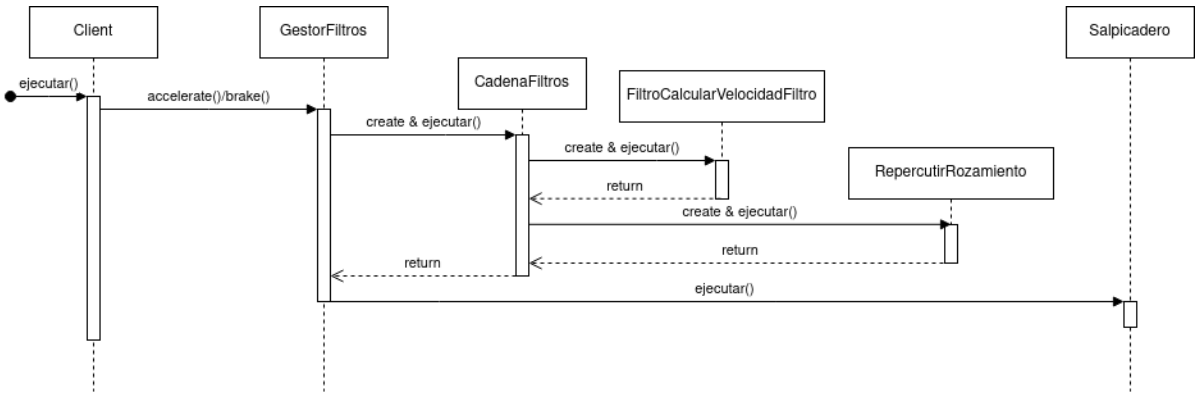

Figure 8: ejercicio 4 Flowchart

4.3 Considerations

There were 3 main considerations made during the conception of the program

- 1. The engine
  The question was where to put the code for the engine. We decided putting it in the GestorFiltros and it get's logically called when the Encender button is activated
- 2. Update frequency
  We decided updating the rpms (Increasing or decreasing the rpm) every second.
- 3. Problem running engine and user interface simultaneously
  We encountered the problem in the devolopement that if we run the GUI and the engine on the same process they block each other so the engine isn't operable (buttons can't be pushed) when the engine is running

4.4 Execution

When the program is started two windows open:

- An interactive window where you choose your action
- A non interactive window where the actual informations are shown

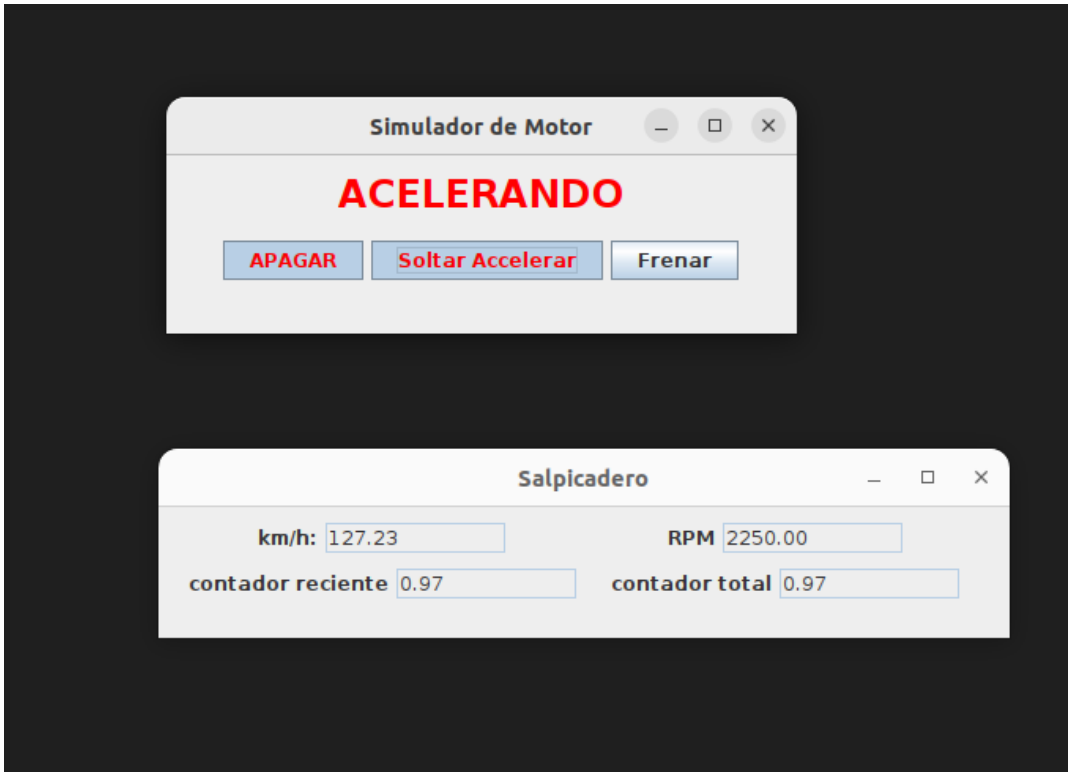

Figure 9: Ejecucion del ejercicio 4

5 Ejercicio 5

5.1 Análisis

En este problema se pide diseñar y realizar una aplicación de WebScraping en Python donde se utilice el Patrón *Strategy* para scrapear información en vivo de acciones. Para ello se seguirán dos estrategias cada una usando una biblioteca distinta de Python: BeautifulSoup y Selenium.

5.2 Strategy

El patrón *Strategy* o *Estrategia* define una familia de algoritmos, encapsula cada uno y los hace intercambiables en tiempo de ejecución. Para ello definimos una clase abstracta llamada *ScraperStrategy* y las estategias de scrapping correspondientes *BeautifulSoupScraper* y *SeleniumScraper*. Además, es necesario crear un clase que sirva para intercambiar las estrategias definidas en cada ejecución, sirviendo para esto la clase *Context*.

5.3 Solución

Horadando en el código de cada estrategia de scrapping, *SeleniumScraper* ha supuesto un ejercicio de mayor complejidad. Al intentar llevar a cabo este algoritmo, en primera instancia, obteníamos errores debido a la ventana emergente que informa de las cookies nada más entrar en la web. Para sortearla, es necesario pulsar sobre cualquiera de los botones de *Aceptar* o *Rechazar*, sin embargo, estos botones son visibles únicamente si la pantalla cumple unas propiedades específicas de tamaño, si no, es necesario realizar un scroll vertical para alcanzar dichos botones. La solución planteada a esta tesitura ha sido primero maximizar la ventana, garantizando así la visibilidad de los botones, para luego pulsar sobre el botón de *Rechazar* y finalmente acceder a los datos que queremos obtener. Utilizando *BeautifulSoup* ha sido necesario hacer únicamente esta útlima parte.

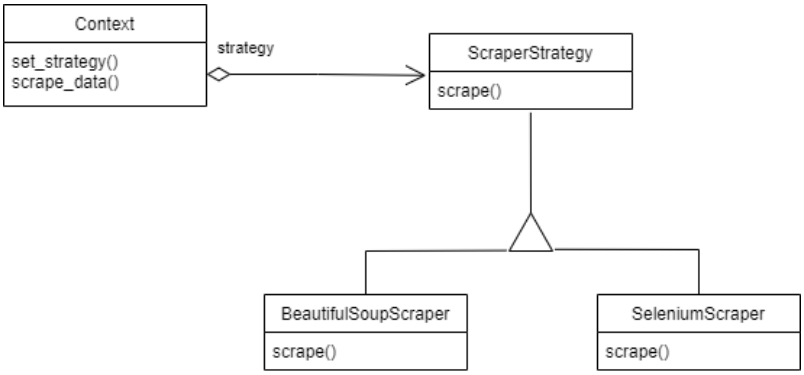

Figure 10: Strategy

5.4 Ejecución

Durante la ejecución debemos ingresar el símbolo de la acción, y luego seleccionar un método de scrapping: 1 para *BeautifulSoup* y 2 para *Selenium*. Finalemente, esto nos permitirá guardar los datos que queríamos alamcenar en el documento url_info.json.

```
Ingrese el símbolo de la acción (por ejemplo, TSLA para Tesla): TSLA
Seleccione el método de scraping (1 para BeautifulSoup, 2 para Selenium): 2

Datos guardados exitosamente en url_info.json
```

Figure 11: Ejecucion del ejercicio 5

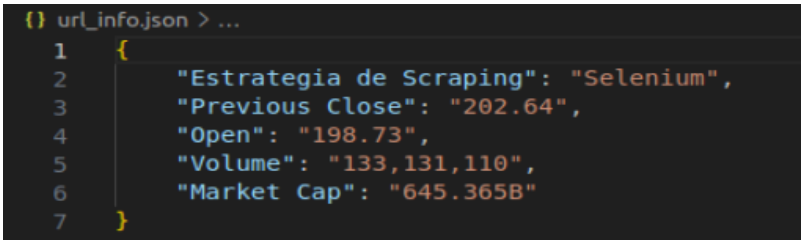

Figure 12: Archivo resultante de la ejecución del ejercicio 5