

GRADO EN INGENIERÍA INFORMÁTICA
DESARROLLO DE SOFTWARE



**UNIVERSIDAD
DE GRANADA**

**P1 : Uso de patrones de diseño
creacionales y estructurales en OO**

Juan Miguel Acosta Ortega (*acostaojuanmi@correo.ugr.es*)

Jesús Pereira Sánchez (*jesuspereira@correo.ugr.es*)

David Serrano Domínguez (*davidserrano07@correo.ugr.es*)

Raúl Florentino Serra (*raulfloren@correo.ugr.es*)

21 de marzo de 2024

Índice

1	Ejercicio 1. Patrón Factoría Abstracta y Método Factoría en Java	4
1.1	Diseño de la primera versión del código (sin hebras)	4
1.2	Diseño de la versión final con hebras	5
2	Ejercicio 2. Patrón Factoría Abstracta + Patrón Prototipo (Python)	9
3	Ejercicio 3. Patrón Builder y Patrón Observer (Python)	13
3.1	Planteamiento del problema	13
3.2	Patrón Builder	14
3.3	Patrón Observer	21
3.4	Solución final: Patrón Builder y Patrón Observer	23
3.4.1	El programa main	23
4	Ejercicio 4. Patrón Filtro Intercepción en Python	25
4.1	Diseño UML del problema	25
4.2	Implementación de los Filtros	26
4.3	Implementación de Gestor Filtros y Cadena Filtros	28
4.4	Implementación del Salpicadero	30
4.5	Implementación cliente	30
4.6	Implementación Main	30
5	Ejercicio Opcional : Web Scrapping (Patrón Estrategia)	32
5.1	Modularizando el código	34

Objetivos

Se realizarán programas Orientados a Objetos (OO) bajo distintos patrones de diseño creaciones y estructurales.

1. Familiarizarse con el uso de herramientas que integren las fases de diseño e implementación de código en un marco de Orientación a Objetos (OO)
2. Aprender a aplicar distintos patrones creacionales y estructurales a problemas diversos
3. Adquirir destreza en la práctica de diseño OO

4. Aprender a adaptar los patrones a las especificidades de distintos lenguajes OO

1. Ejercicio 1. Patrón Factoría Abstracta y Método Factoría en Java

En este primer ejercicio se han de usar los patrones de diseño Abstract Factory y Factory Method en Java para conseguir una solución que se adecue al enunciado.

El primer paso y tras entender que Abstract Factory es un patrón de diseño creacional que nos permite producir familias de objetos relacionados sin especificar sus clases concretas, y que el Factory Method es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán, será hacer una primera revisión de diagrama UML en el que representar la arquitectura del software.

1.1. Diseño de la primera versión del código (sin hebras)

Se han añadido las clases necesarias para representar de manera simple una solución sin hebras. En nuestro caso también hemos añadido una agregación dando a entender que las carreras se componen de varias bicicletas independientes a estas.

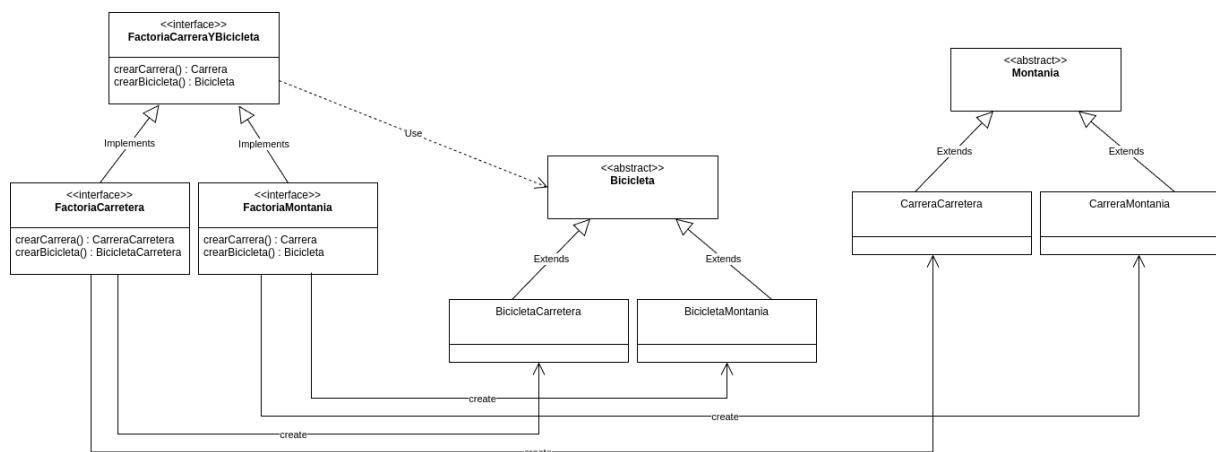


Figura 1: Primera revisión de diagrama sin implementación de hebras.

En esta primera versión del código funciona todo lo respectivo a los patrones requeridos y sólo faltan los requisitos referentes las hebras.

```

FactoriaCarreraYBicicleta fac1 = new FactoriaCarretera();
FactoriaCarreraYBicicleta fac2 = new FactoriaMontania();

Carrera car1 = fac1.crearCarrera();
Carrera car2 = fac2.crearCarrera();

Bicicleta bici1 = fac1.crearBicicleta();
Bicicleta bici2 = fac2.crearBicicleta();

car1.mostrarTipo();
car2.mostrarTipo();

bici1.mostrarTipo();
bici2.mostrarTipo();

```

Figura 2: Instanciación de las diferentes bicicletas y carreteras.

1.2. Diseño de la versión final con hebras

Uno de los requisitos de la solución es que las carreras tuvieran la posibilidad de empezar y acabar a la vez, para ello debían ejecutarse de forma simultánea y sincronizada, es decir, concurrentemente. En este caso al utilizar Java para este ejercicio decidimos hacer uso del interface Java Runnable pues es uno de los interfaces más utilizados en el lenguaje para ejecutar tareas de forma concurrente. Esto cambia el diseño inicial en que las clases Carrera y Bicicleta implementan esta interfaz.

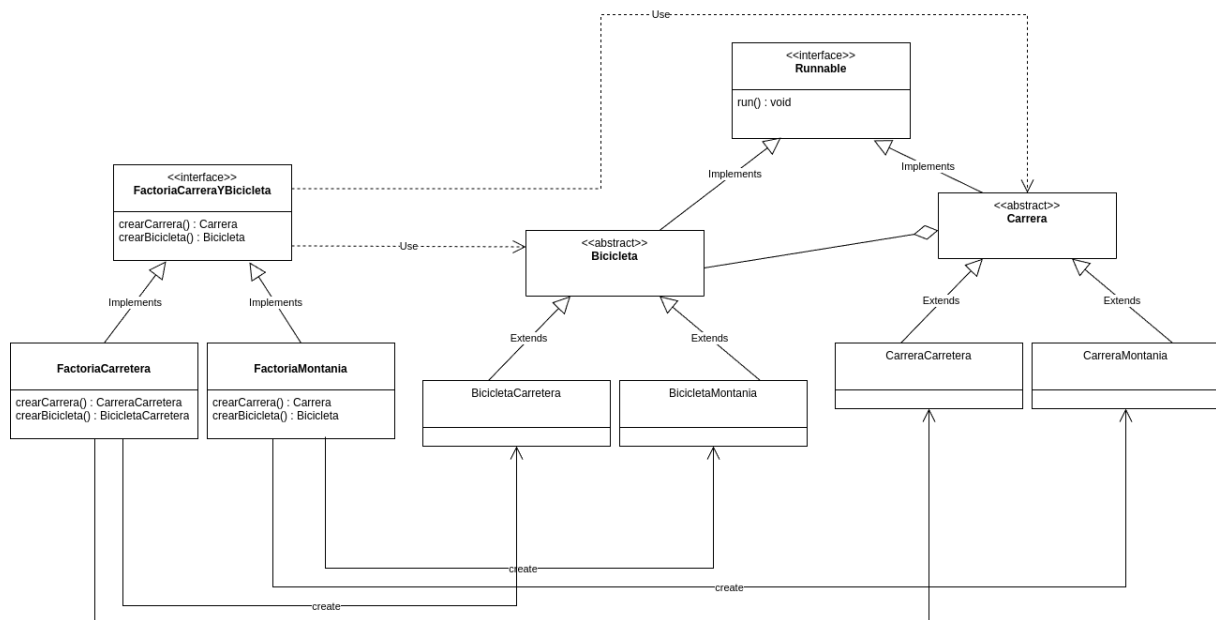


Figura 3: Versión completa del diagrama UML para el ejercicio.

Para que dos carreras se ejecuten de manera simultánea declaramos dos hebras que a su vez serán dos carreras diferentes. Al tener implementado el método run en la clase Carrera podemos sincronizarlas y controlar su comportamiento.

```

FactoriaCarreraYBicicleta fac1 = new FactoriaCarretera();
FactoriaCarreraYBicicleta fac2 = new FactoriaMontania();

Carrera car1 = fac1.crearCarrera();
Carrera car2 = fac2.crearCarrera();

Thread t_car1 = new Thread(car1);
Thread t_car2 = new Thread(car2);

t_car1.start();
t_car2.start();

```

Figura 4: Instanciación de carreras y sus respectivas hebras.

Aparte de crear hebras para que las carreras se ejecuten de manera simultánea, también vamos a declarar una hebra para cada uno de las bicis de ambas carreras, de esta manera todas las bicis empiezan las carreras al mismo tiempo. Vamos a almacenar los hilos de bicis de cada carrera en un vector de tipo Thread que se encontrara en la clase Carrera y cada carrera tendrá los hilos de las bicicletas que participan en ella.

La cantidad de bicicletas que vamos a tener es un número aleatorio entre 10 y 20, además será la misma cantidad para ambas carreras.

```

Bicicleta bici1;
Random random = new Random();
int nBicicletas = random.nextInt( bound: 11) + 10;
System.out.println("En cada carrera vamos a tener: "+nBicicletas+" inscritas");
System.out.println("\nInscripciones de Carrera Carretera:");

for(int i=0;i<nBicicletas;i++){
    bici1= fac1.crearBicicleta( id: i+1);
    Thread hilo = new Thread(bici1);
    car1.addHilosBicis(hilo);
    car1.aniadirBici(bici1);
    System.out.println("Bicicleta con ID: "+bici1.id);
}

System.out.println("\nInscripciones de Carrera Montaña:");
for(int i=0;i<nBicicletas;i++) {
    bici1 = fac2.crearBicicleta( id: i+1);
    Thread hilo = new Thread(bici1);
    car2.addHilosBicis(hilo);
    car2.aniadirBici(bici1);
    System.out.println("Bicicleta con ID: " + bici1.id);
}

```

Figura 5: Instanciación de bicicletas para cada carrera y sus respectivas hebras.

Destacar que en el método run de las Clases Bicicletas vamos a tener un output que nos va a mostrar el ID de la bici y que carrera ha empezado a correr dicha bici.

```
@Override
public void run() { System.out.println("Empezo a correr la bici de Carretera con ID : " + id); }
```

Figura 6: Implementación de método run de la Bicicleta de Carretera

El hilo de cada bicicleta va a empezar en el método run de la carrera a la que pertenece, ya que vamos a tener un bucle for que va a recorrer el array de hilos y para cada uno va a llamar a su método hilo.start() como se muestra en la foto que viene a continuación. Además en el método run de cada carrera vamos a tener un Thread.sleep() que va a controlar el tiempo que va a durar la carrera.

```
@Override
public void run() {
    try {
        for(int i=0;i<getHilosBicicletas().size();i++){
            getHilosBicicletas().get(i).start();
        }
        Thread.sleep( millis: this.duracion*1000);
        System.out.println("\nTermino la Carrera de carretera");
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

Figura 7: Implementación de método run de la Carrera de Carretera

Ahora vamos a pasar a hablar de como gestionamos las bicicletas que se retiran de la carrera. Para ello lo primero que hemos hecho es que tras hacer el hilo1.start() e hilo2.start (damos inicio a ambas carreras) vamos a meter un Thread.sleep(10000) del hilo del main para así evitar que las carreras se retiren nada más al principio y lo hagan pasado 10 segundos. Tras pasar dicho tiempo vamos a llamar una función que pertenece a la Clase Carrera llamada getRetirados() y que le pasamos como parámetro el número de bicis que queremos que se retiren, en nuestro caso como no es un número fijo y se nos decía 10 por ciento para la carrera Carretera y 20 por ciento para carrera Montaña haremos dicho porcentaje con el número de bicis totales que tenemos en las carreras como mostramos a continuación.

El método getRetirados(n) lo que hará será coger n y generara una cantidad de bicis aleatorias igual a n y posteriormente meterlos en un vector que llamaremos retirados y devolverlo.

```

System.out.println("\nEmpiezan ambas carreras a la vez \n");
hilo1.start();
hilo2.start();

Thread.sleep(10000);
ArrayList<Bicicleta> retirados1 = car1.getRetirados((int) Math.ceil(0.1*nBicicletas));
ArrayList<Bicicleta> retirados2 = car2.getRetirados((int) Math.ceil(0.2*nBicicletas));
System.out.println("\nSe han retirado "+retirados1.size()+ " bicicletas de la Carrera carretera: ");
for(int i=0;i<retirados1.size();i++){
    System.out.println("Bicicleta con ID: " + retirados1.get(i).id + " y tipo " + retirados1.get(i).tipo);
}

System.out.println("\nSe han retirado "+retirados2.size()+ " bicicletas de la Carrera de Montaña: ");
for(int i=0;i<retirados2.size();i++){
    System.out.println("Bicicleta con ID: " + retirados2.get(i).id + " y tipo " + retirados2.get(i).tipo);
}

```

Figura 8: Manejo de bicicletas retiradas de ambas carreras

```

public ArrayList<Bicicleta> getRetirados(int n){
    Random ran = new Random();
    ArrayList<Bicicleta> retirados = new ArrayList<>();
    int retirado=0;
    Bicicleta actual;
    while(retirados.size()<n){
        for (int i=0;i<n;i++){
            retirado = (int) (Math.random() * 10);
            actual=getParticipantes().get(retirado);
            if(!retirados.contains(actual)) retirados.add(getParticipantes().get(retirado));
        }
    }
    return retirados;
}

```

Figura 9: Implementación de método getRetirados de la Clase Carrera

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

Este ejercicio sigue la línea del anterior con dos cambios sustanciales. El primero es el cambio de lenguaje de Java a Python, el cuál supuso un pequeño reto para la mayoría del grupo ya que hemos trabajado poco con él, y el otro es el de añadir el patrón de diseño Prototipo el cuál permite utilizar como prototipos un grupo de objetos prefabricados, configurados de maneras diferentes.

Estos cambios afectan poco a la arquitectura, en concreto eliminamos la interfaz Runnable, ya que en Python la creación de hebras se puede realizar sin depender de ninguna clase abstracta (se importa la librería threading), y añadimos la interfaz Prototipo, la cuál implementa Bicicleta para poseer el método clone(). Esto nos permite hacer clones de las bicicletas en lugar de instanciar nuevas copiando varias veces sus atributos.

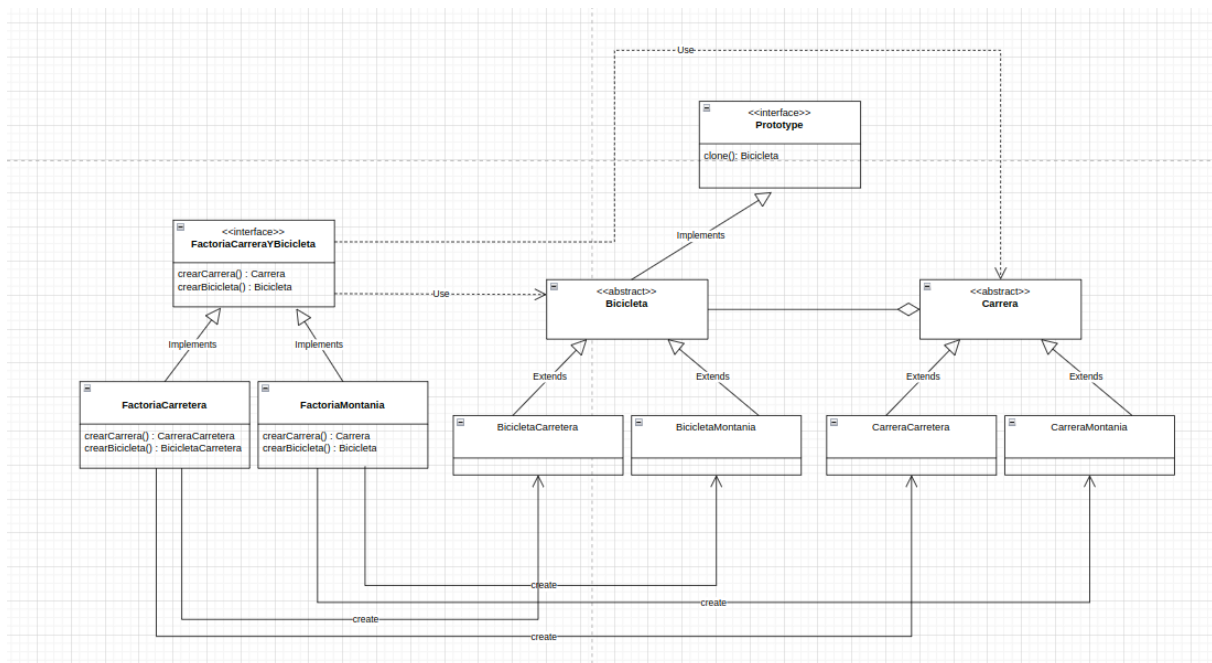


Figura 10: Diagrama UML Ejercicio 2

El ejercicio en sí mismo consta de una primera parte en la que se transcribe el código en Java a Python prescindiendo de lo innecesario, y una segunda parte de adaptación.

En primer lugar y como se describe anteriormente se usan la librería threading para el tema de las hebras (en este caso únicamente las carreras serán hebras), la librería time para simular el tiempo de las carreras, y demás librerías para operaciones matemáticas y números aleatorios ...

El main de este ejercicio se ocupa de instanciar mediante factorías las dos carreras y dos bicis que más adelante servirán para clonarse en el método run() de las carreras. Salvo eso es similar al main del ejercicio anterior:

```
1 usage  👤 JuanmiAcosta +1
def main():
    fac1 = FactoriaCarretera()
    fac2 = FactoriaMontania()

    car1 = fac1.crearCarrera()
    car2 = fac2.crearCarrera()

    bici1 = fac1.crearBicicleta()
    bici2 = fac2.crearBicicleta()

    t1 = threading.Thread(target=car1.run, args=[bici1])
    t2 = threading.Thread(target=car2.run, args=[bici2])

    t1.start()
    t2.start()

    t1.join()
    t2.join()
```

Figura 11: Main del ejercicio 2

Es en el método `run()` de sendas Carreras (distinguimos entre `CarreraMontania` y `CarreraCarretera` para poner diferentes output) dónde se hacen los `time.sleep()`, se clonan las bicicletas para inscribirlas a las carreras, y se retiran el respectivo porcentaje de ellas de las carreras que le correspondan. Para obtener un output en el que siempre se retiren bicicletas generamos un numero aleatorio entre 10 y 20, y redondeamos el porcentaje hacia arriba, es decir si son 11 bicicletas se retiraran 2.

Hacemos más de un `time.sleep()` para sincronizar los output.

```
2 usages (2 dynamic)  JuanmiAcosta
def run(self, bici):
    print("Comienza la carrera de montaña\n")

    num_bicis = randint(a: 10, b: 20)

    for i in range(num_bicis):
        clone = bici.clone()
        self.aniadirBici(clone)

    print("Estan listas las ", num_bicis, " para la carrera de montaña\n")

    time.sleep(4)

    veinte_por = math.floor(num_bicis * 0.2)
    retiradas = []

    for i in range(veinte_por):
        bici_aleatoria = randint(a: 0, num_bicis-i-1)
        self.getParticipantes().pop(bici_aleatoria)
        retiradas.append(bici_aleatoria)

    print("Se retiraron las bicis de montaña numero")
    for i in range(veinte_por):
        print("\t", retiradas[i])

    time.sleep(6)
    print("\nCarrera de montaña terminada\n")
```

Figura 12: Ejemplo del método run()

Para terminar mostramos un oputput de ejemplo, el número de bicis en cada carrera como el enunciado indica no se sabe hasta que estas dan comienzo:

```
Comienza la carrera de carretera

Estan listas las 10 para la carrera de carretera
Comienza la carrera de montania

Estan listas las 19 para la carrera de montania


Se retiraron las bicis de montania numero
    12
    7
    11
Se retiraron las bicis de carretera numero
    3
    7


Carrera de carretera terminada


Carrera de montania terminada
```

Figura 13: Output de ejemplo ejercicio 2.

3. Ejercicio 3. Patrón Builder y Patrón Observer (Python)

3.1. Planteamiento del problema

Tu vecino Paco, el cual regenta la típica hamburguesería de barrio, quiere informatizar su negocio. Para ello, ha confiado en tí, un estudiante de informática el cual arrastra algunas asignaturas de 1ero, pero sabe programar perfectamente, y más importante: **sabe como aplicar los patrones Builder y Observer**, ya que ha elegido **Desarrollo del Software** y sabrá estructurar perfectamente el problema.

El local dispone de 3 tipos de hamburguesas con su tiempo de preparación según los ingredientes:

1. *Hamburguesa normal*: Pan, lechuga, tomate, queso de cabra, cebolla, pepinillos, bacon, y carne.
Tiempo de preparación: 1.9s. Precio: 5€
2. *Hamburguesa sin gluten*: Pan sin gluten, resto ingredientes de la hamburguesa normal.
Tiempo de preparación: 2s. Precio: 6€
3. *Hamburguesa vegana*: No lleva queso de cabra, la carne y el bacon son veganos, el resto de ingredientes son los mismos que la hamburguesa normal.
Tiempo de preparación: 2.1s Precio: 5.5€

Queremos organizar la creación de las hamburguesas para respetar tanto los ingredientes como el orden en el que se ponen estos mismos.

Por otro lado, el cocinero que se encarga de recoger y cocinar, quiere hacer que los pedidos que estén listos sean mostrados según su id en una pantalla, para que los clientes hagan cola fuera del local y no dentro, que es un local pequeño. Además, para que Hacienda no le crujía a final de mes, Paco necesita tomar un historial de los pedidos realizados en cada día, en el que se ponga el precio de cada hamburguesa individual y el precio total de pedido.

Para la primera parte del problema, usaremos el patrón builder, en el que el **cocinero** será el director, luego tendremos los builder de cada una de las hamburguesas y la propia hamburguesa que nos servirá de base.

En la segunda parte del problema, usaremos un patrón observer en el que tendremos la **pantalla** como observador, pendiente de los pedidos que se terminen (las actualizaciones), las cuales serán notificadas por el observado, el mismo **cocinero**, el cual hará la función de un mostrador.

La conexión de estas dos partes segmentadas es el **cocinero-mostrador**, el cual hace la función de crear las hamburguesas (**builder**) y la de notificar a la pantalla cuando hay un pedido terminado (**subject**). Planteamos los problemas por separado, y luego los juntamos.

3.2. Patrón Builder

Planteamos el siguiente diagrama UML, basado en el **Patrón Builder**

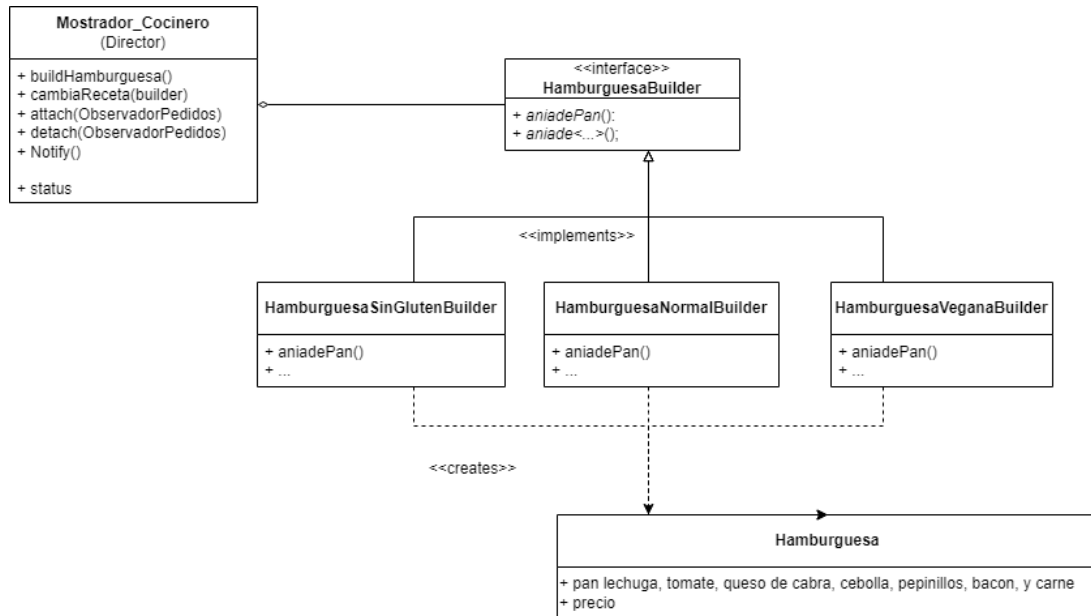


Figura 14: UML Patrón Builder.

```

1  from HamburguesaBuilder import HamburguesaBuilder
2  from Subject import Subject
3  from Pedido import Pedido
4  class Cocinero(Subject):
5      def __init__(self, builder):
6          self._builder = builder
7
8          self.status = "Tomando pedido"
9          self.observers = []
10         self.pedidoActual = Pedido()
11
12         def build_hamburguesa(self):
13             self.status = "Cocinando"
14
15             self._builder.create_new_hamburguesa()
16             self._builder.aniadePan()
17             self._builder.aniadeLechuga()
18             self._builder.aniadeTomate()
19             if hasattr(self._builder, 'aniadeQuesoCabra'):
20                 self._builder.aniadeQuesoCabra()
21             self._builder.aniadeCebolla()
22             self._builder.aniadePepinillos()
23             self._builder.aniadeBacon()
24             self._builder.aniadeCarne()
25             self._builder.aniadePrecio()
26
27             self.pedidoActual.aniadeHamburguesa(self._builder.hamburguesa)
28             self.status = "Tomando pedido"
29
30         def cambiaReceta(self, builder):
31             self._builder = builder
32
33         def attach(self, observer):
34             self.observers.append(observer)
35
36         def detach(self, observer):
37             self.observers.remove(observer)
38
39         def notify(self):
40             for notified in self.observers:
41                 notified.update(self.pedidoActual)
42             self.pedidoActual = Pedido()

```

Figura 15: Class Cocinero.

```
1  from abc import ABC, abstractmethod
2  from Hamburguesa import Hamburguesa
3
4  class HamburguesaBuilder(ABC):
5      def __init__(self):
6          self.hamburguesa = None
7
8      def create_new_hamburguesa(self):
9          self.hamburguesa = Hamburguesa()
10
11     @abstractmethod
12     def aniadePan(self):
13         pass
14
15     @abstractmethod
16     def aniadeLechuga(self):
17         pass
18
19     @abstractmethod
20     def aniadeTomate(self):
21         pass
22
23     def aniadeQuesoCabra(self): # Este método no tiene por qué ser implementado
24         pass
25
26     @abstractmethod
27     def aniadeCebolla(self):
28         pass
29
30     @abstractmethod
31     def aniadePepinillos(self):
32         pass
33
34     @abstractmethod
35     def aniadeBacon(self):
36         pass
37
38     @abstractmethod
39     def aniadeCarne(self):
40         pass
41
42     @abstractmethod
43     def aniadePrecio(self):
44         pass
```

Figura 16: Class Hamburguesa Builder.


```
1  from HamburguesaBuilder import HamburguesaBuilder
2  import time
3  class HamburguesaNormalBuilder(HamburguesaBuilder):
4      def aniadePan(self):
5          self.hamburguesa.pan = "Pan normal"
6          time.sleep(0.1)
7
8      def aniadeLechuga(self):
9          self.hamburguesa.lechuga = "Lechuga fresca"
10         time.sleep(0.1)
11
12     def aniadeTomate(self):
13         self.hamburguesa.tomate = "Tomate pera"
14         time.sleep(0.1)
15
16     def aniadeQuesoCabra(self):
17         self.hamburguesa.quesoCabra = "Queso de cabra recién cortado"
18         time.sleep(0.1)
19
20
21     def aniadeCebolla(self):
22         self.hamburguesa.cebolla = "Cebolla llorosa"
23         time.sleep(0.1)
24
25
26     def aniadePepinillos(self):
27         self.hamburguesa.pepinillos = "Pepinillos"
28         time.sleep(0.1)
29
30
31     def aniadeBacon(self):
32         self.hamburguesa.bacon = "Bacon grasiento"
33         time.sleep(0.5)
34
35
36     def aniadeCarne(self):
37         self.hamburguesa.carne = "Carne Wagyu"
38         time.sleep(0.8)
39
40     def aniadePrecio(self):
41         self.hamburguesa.precio = 5
42
```

Figura 17: Class Hamburguesa Normal Builder.

```
1  from HamburguesaBuilder import HamburguesaBuilder
2  import time
3  class HamburguesaSinGlutenBuilder(HamburguesaBuilder):
4  @↑    def aniadePan(self):
5        self.hamburguesa.pan = "Pan sin gluten"
6        time.sleep(0.2)
7
8  @↑    def aniadeLechuga(self):
9        self.hamburguesa.lechuga = "Lechuga verde"
10       time.sleep(0.1)
11
12  @↑    def aniadeTomate(self):
13        self.hamburguesa.tomate = "Tomate pera"
14        time.sleep(0.1)
15
16  @↑    def aniadeQuesoCabra(self):
17        self.hamburguesa.quesoCabra = "Queso de cabra (sin trazas)"
18        time.sleep(0.1)
19
20  @↑    def aniadeCebolla(self):
21        self.hamburguesa.cebolla = "Cebolla llorosa"
22        time.sleep(0.1)
23
24  @↑    def aniadePepinillos(self):
25        self.hamburguesa.pepinillos = "Pepinillos"
26        time.sleep(0.1)
27
28  @↑    def aniadeBacon(self):
29        self.hamburguesa.bacon = "Bacon grasiento"
30        time.sleep(0.5)
31
32  @↑    def aniadeCarne(self):
33        self.hamburguesa.carne = "Carne Wagyu (sin trazas)"
34        time.sleep(0.8)
35
36  @↑    def aniadePrecio(self):
37        self.hamburguesa.precio = 6
```

Figura 18: Class Hamburguesa Sin Gluten Builder.

```
1  from HamburguesaBuilder import HamburguesaBuilder
2  import time
3  class HamburguesaVeganaBuilder(HamburguesaBuilder):
4
5      def aniadePan(self):
6          self.hamburguesa.pan = "Pan normal"
7          time.sleep(0.1)
8
9      def aniadeLechuga(self):
10         self.hamburguesa.lechuga = "Lechuga fresca"
11         time.sleep(0.1)
12
13     def aniadeTomate(self):
14         self.hamburguesa.tomate = "Tomate pera"
15         time.sleep(0.1)
16
17     def aniadeCebolla(self):
18         self.hamburguesa.cebolla = "Cebolla llorosa"
19         time.sleep(0.1)
20
21     def aniadePepinillos(self):
22         self.hamburguesa.pepinillos = "Pepinillos"
23         time.sleep(0.1)
24
25     def aniadeBacon(self):
26         self.hamburguesa.bacon = "Bacon Vegano reseco"
27         time.sleep(0.6)
28
29     def aniadeCarne(self):
30         self.hamburguesa.carne = "Carne Vegana de dudosa procedencia"
31         time.sleep(1)
32
33     def aniadePrecio(self):
34         self.hamburguesa.precio = 5.5
```

Figura 19: Class Hamburguesa Vegana Builder.

```
3 class Hamburguesa:
4     def __init__(self):
5         self.pan = None
6         self.lechuga = None
7         self.tomate = None
8         self.quesoCabra = None
9         self.cebolla = None
10        self.pepinillos = None
11        self.bacon = None
12        self.carne = None
13        self.precio = 0
14
15    def __str__(self):
16
17        ingredientes = ''
18
19        if (self.pan != None) :
20            ingredientes=ingredientes+f"{self.pan}"
21        if (self.lechuga != None) :
22            ingredientes=ingredientes+f" {self.lechuga},"
23        if (self.tomate != None) :
24            ingredientes=ingredientes+f" {self.tomate},"
25        if (self.quesoCabra != None) :
26            ingredientes=ingredientes+f" {self.quesoCabra},"
27        if (self.cebolla != None) :
28            ingredientes=ingredientes+f" {self.cebolla},"
29        if (self.pepinillos != None) :
30            ingredientes=ingredientes+f" {self.pepinillos},"
31        if (self.bacon != None) :
32            ingredientes=ingredientes+f" {self.bacon},"
33        if (self.carne != None) :
34            ingredientes=ingredientes+f" {self.carne}."
35
36        return ingredientes
37
```

Figura 20: Class Hamburguesa.

3.3. Patrón Observer

Planteamos el siguiente diagrama UML, basado en el **Patrón Builder**

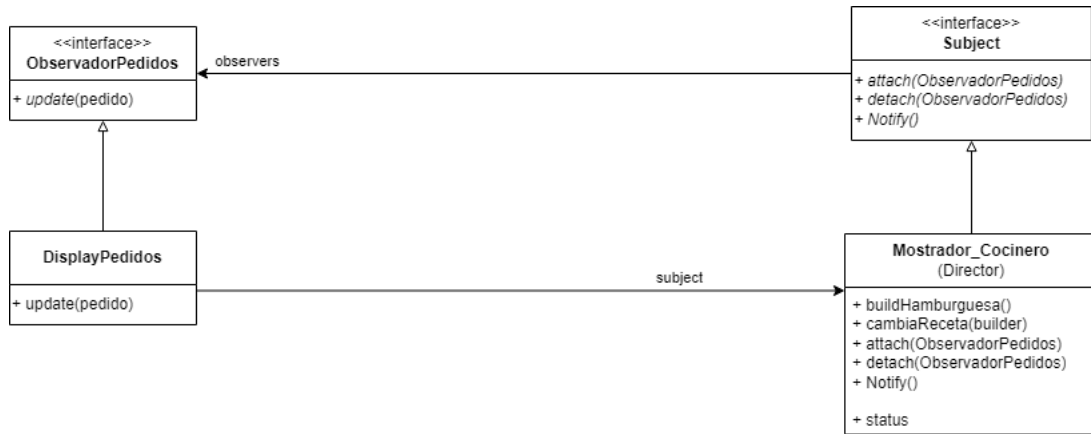


Figura 21: UML Patrón Observer.

```

1  from abc import ABC, abstractmethod
2
3  @↓ class Subject(ABC):
4      @abstractmethod
5      @↓ def attach(self, observer):
6          pass
7
8      @abstractmethod
9      @↓ def detach(self, observer):
10         pass
11
12         @abstractmethod
13         @↓ def notify(self):
14             pass
  
```

Figura 22: Class Subject.

```
1  from abc import ABC, abstractmethod
2
3  @↓ class ObservadorPedido(ABC):
4      @abstractmethod
5  @↓ def update(self, pedido):
6      pass
```

Figura 23: Class Observador Pedido.

```
1  from ObservadorPedido import ObservadorPedido
2  import os
3  class DisplayPedidos(ObservadorPedido):
4      def __init__(self):
5          self.historial = []
6          if os.path.exists("historial.txt"):
7              os.remove("historial.txt")
8          self.f = open("historial.txt", "x")
9          self.f.close()
10
11  @↑ def update(self, pedido):
12      self.historial.append(pedido)
13      print(f"\nEl pedido {pedido.idPedido} está listo!")
14      self.saveToTxt(pedido)
15
16  def saveToTxt(self, pedido):
17      self.f = open("historial.txt", "a")
18      self.f.write(pedido.__str__())
19      self.f.close()
```

Figura 24: Class Display Pedidos.

Además se implementa una clase Pedidos encargada de contener varias hamburguesas en conjunto y mostrar el precio final en su conjunto.

```

1 import datetime
2 import time
3 class Pedido:
4     def __init__(self):
5         self.idPedido = time.strftime( format: '%H%M%S', time.localtime())
6         self.hamburguesas = []
7         self.precio = 0
8
9     def anadeHamburguesa(self, hamburguesa):
10         self.hamburguesas.append(hamburguesa)
11         self.precio = self.precio + hamburguesa.precio
12
13     def __str__(self):
14         cadena = f"\nPedido {self.idPedido}:\n"
15         index = 1
16         for hamburguesa in self.hamburguesas:
17             if index==1:
18                 cadena= cadena + f"\tHamburguesa {index}: {hamburguesa.__str__()} Precio {hamburguesa.precio}"
19             else:
20                 cadena= cadena+f"\n\tHamburguesa {index}: {hamburguesa.__str__()} Precio {hamburguesa.precio}"
21             index+=1
22         cadena = cadena + f"\n\tTotal {self.precio}€"
23         return cadena

```

Figura 25: Class Pedidos.

3.4. Solución final: Patrón Builder y Patrón Observer

Combinando los dos patrones, nos quedaría lo siguiente:

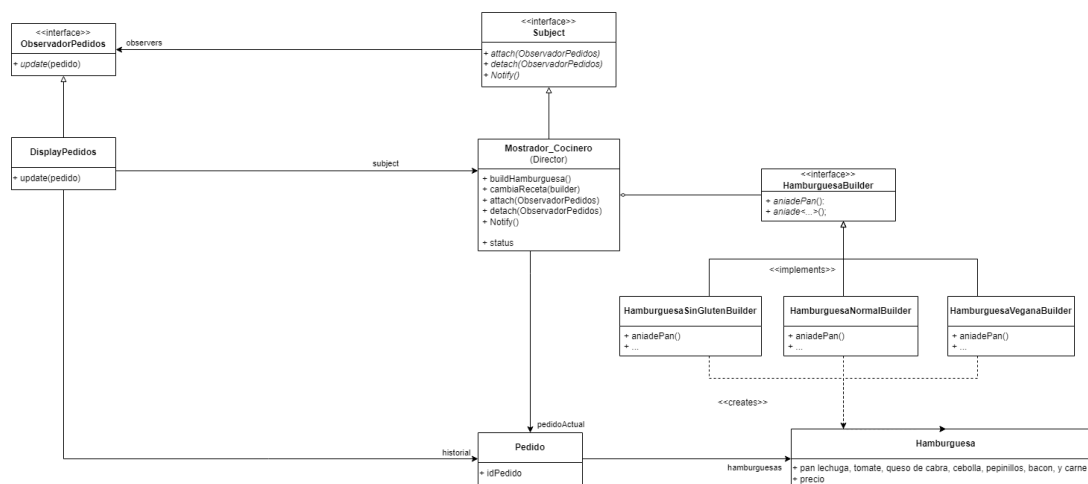


Figura 26: UML Ejercicio 3.

3.4.1. El programa main

```

1 # Creamos la pantalla, la cual notifica de los pedidos listos
2 pantalla = DisplayPedidos()
3
4 # Creamos la "receta" de las hamburguesas sin gluten y le decimos al cocinero que la cocine
5 sinGluten = HamburguesaSinGlutenBuilder()

```

```
6 cocinero = Cocinero(sinGluten)
7 cocinero.attach(pantalla)
8 cocinero.build_hamburguesa()
9 # print("Hamburguesa sin gluten: ")
10 # sinGluten.hamburguesa.__str__()
11 cocinero.notify() # Fin del pedido 1
12
13 # Creamos la "receta" de las hamburguesas normales y que se cocinen
14 normal = HamburguesaNormalBuilder()
15 cocinero.cambiaReceta(normal)
16 cocinero.build_hamburguesa()
17 # print("\nHamburguesa normal: ")
18 # normal.hamburguesa.__str__()
19
20 # Creamos la "receta" de las hamburguesas veganas y que se cocinen
21 vegana = HamburguesaVeganaBuilder()
22 cocinero.cambiaReceta(vegana)
23 cocinero.build_hamburguesa()
24 # print("\nHamburguesa vegana: ")
25 # vegana.hamburguesa.__str__()
26 cocinero.notify() # Fin del pedido 2
```

Listing 1: main.py

De este programa obtenemos por consola la salida:

```
1
2 El pedido 211642 esta listo!
3 El pedido 211644 esta listo!
4 Process finished with exit code 0
```

Listing 2: Salida de la pantalla

Podemos ver como la pantalla lista los pedidos realizados. La funcionalidad del historial que lleva a cabo la pantalla también guarda en un documento de texto el historial de pedidos, con cada una de las hamburguesas con sus precios respectivos y el precio total. Podemos comprobarlo en el fichero generado:

Pedido 211642:

Hamburguesa 1: Pan sin gluten Lechuga verde, Tomate pera,
Queso de cabra (sin trazas), Cebolla llorosa, Pepinillos,
Bacon grasiento, Carne Wagyu (sin trazas). Precio 6
Total 6€

Pedido 211644:

Hamburguesa 1: Pan normal Lechuga fresca, Tomate pera,
Queso de cabra recién cortado, Cebolla llorosa, Pepinillos,
Bacon grasiento, Carne Wagyu. Precio 5
Hamburguesa 2: Pan normal Lechuga fresca, Tomate pera,
Cebolla llorosa, Pepinillos, Bacon Vegano resaco,
Carne Vegana de dudosa procedencia. Precio 5.5
Total 10.5€

4. Ejercicio 4. Patrón Filtro Intercepción en Python

En este ejercicio vamos a implementar el patrón filtro intercepción mediante la representación del salpicadero de un coche que nos mostrara: Velocidad lineal en Km/h, distancia recorrida en Km y velocidad angular(revoluciones del motor)

El primer paso es entender que Filtro Intercepción es un patrón de diseño conducta que nos permite incorporar servicios de manera transparente y mediante intercepciones automáticas, utilizando una estructura de Tubería y Filtro para procesar datos secuencialmente.

4.1. Diseño UML del problema

Este sería el diseño UML añadiendo las clases que se nos piden en el ejercicio para poder representar de manera correcta la simulación de un vehiculo con cambio automático, aplicandole el patrón filtro intercepción.

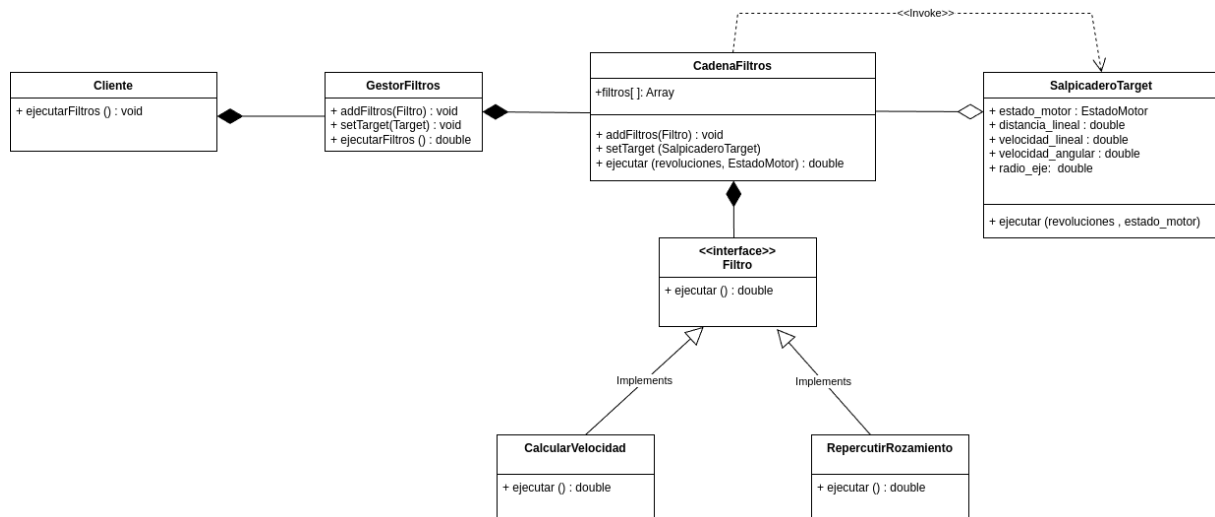


Figura 27: Versión del diagrama UML para el ejercicio.

Nos encontramos con un diseño que nos va a permitir implementar el problema que se nos esta planteando mediante la utilización del patrón arquitectónico Filtro Intercepción

4.2. Implementación de los Filtros

Como se ha mostrado en el diagrama UML anterior vamos a tener un interfaz Filtro del cual van a heredar dos clases que van a ser filtros específicos: CalcularVelocidad y RepercutirRozamiento.

```
from abc import ABC, abstractmethod

4 usages  ⚡ JuanmiAcosta

class Filtro(ABC):

    3 usages (3 dynamic)  ⚡ JuanmiAcosta

    @abstractmethod
    def ejecutar(self, revoluciones, estado_motor):
        pass
```

Figura 28: Implementación interface Filtro.

```
class CalcularVelocidad(Filtro):

    ⚡ David Serrano

    def __init__(self):
        self.incremento_velocidad = 0

    3 usages (3 dynamic)  ⚡ JuanmiAcosta +1 *

    def ejecutar(self, revoluciones, estado_motor):

        if (estado_motor == EstadoMotor.EstadoMotor.APAGADO or estado_motor == EstadoMotor.EstadoMotor.ENCENDIDO):
            self.incrementoVelocidad = 0
        elif estado_motor == EstadoMotor.EstadoMotor.ACCELERANDO:
            self.incrementoVelocidad = 100
        elif estado_motor == EstadoMotor.EstadoMotor.FRENANDO:
            self.incrementoVelocidad = -100

        if revoluciones + self.incrementoVelocidad > 5000:
            self.incrementoVelocidad = 5000 - revoluciones

        rev_devolver = revoluciones + self.incrementoVelocidad

        if rev_devolver <= 0:
            return 0
        else:
            return rev_devolver
```

Figura 29: Implementación clase CalcularVelocidad.

```
from Filtro import Filtro

2 usages  👤 JuanmiAcosta
class RepercutirRozamiento(Filtro):
    3 usages (3 dynamic)  👤 JuanmiAcosta
    def ejecutar(self, revoluciones, estado_motor):
        return revoluciones - (revoluciones*0.005)
```

Figura 30: Implementación clase RepercutirRozamiento.

Como podemos ver tenemos el Filtro que es un interface y luego de el heredan CalcularVelocidad y RepercutirRozamiento, implementando cada uno el método ejecutar aplicando el filtro que le corresponde a cada uno. Además como se puede observar en la clase CalcularVelocidad controlamos que las revoluciones nunca superen las 5000 como se nos indicaba en el ejercicio.

4.3. Implementación de Gestor Filtros y Cadena Filtros

El patrón filtro intercepción contiene una clase gestor filtros que se encarga de gestionar la petición del cliente de los filtros y que contiene la cadena de filtros que vamos a tener que ejecutar. Por otro lado vamos a tener la cadena de filtros que contiene todos los filtros que vamos a ejecutar antes de hacer la ejecución de en el salpicadero para obtener los valores finales del mismo tras aplicar todos los filtros correspondientes.

```
class CadenaFiltros():  
    ~~~~~  
  
    ~ JuanmiAcosta +1  
    def __init__(self):  
        self.filtros = []  
        self.salpicadero_target = None  
  
    2 usages ~ JuanmiAcosta  
    def addFiltro(self, filtro):  
        self.filtros.append(filtro)  
  
    1 usage (1 dynamic) ~ David Serrano  
    def setTarget(self, target):  
        self.salpicadero_target = target  
  
    3 usages (3 dynamic) ~ JuanmiAcosta +1  
    def ejecutar(self, revoluciones, estado_motor):  
        for filtro in self.filtros:  
            revoluciones = filtro.ejecutar(revoluciones, estado_motor)  
  
            if self.salpicadero_target is not None:  
                self.salpicadero_target.ejecutar(revoluciones, estado_motor)  
  
        return revoluciones  
~~~~~
```

Figura 31: Implementación clase CadenaFiltros.

Como podemos observar en la captura anterior CadenaFiltros lo que hace es ejecutar cada uno de los filtros para así aplicar las restricciones que forman dicha cadena e ir guardando las revoluciones que obtenemos de cada filtro en una variable revoluciones que posteriormente se la pasaremos a la función ejecutar del salpicadero target junto con el estado del motor, siempre que dicho salpicadero no sea nulo. Por último vamos a devolverle las revoluciones al gestor de filtros para que este a su vez se la pueda pasar al cliente.

```
class GestorFiltros():  
  
    David Serrano +1  
    def __init__(self):  
        self.cadena_filtros = None  
  
    1 usage David Serrano +1  
    def addCadenaFiltro(self, cadenaFiltro):  
        self.cadena_filtros = cadenaFiltro  
  
    2 usages (1 dynamic) David Serrano +1  
    def setTarget(self, salpicaderoTarget):  
        self.cadena_filtros.setTarget(salpicaderoTarget)  
  
    1 usage JuanmiAcosta  
    def setCadenaFiltros(self, cadena):  
        self.cadena_filtros=cadena  
  
    1 usage (1 dynamic) JuanmiAcosta +1  
    def ejecutarFiltros(self):  
        revoluciones = self.cadena_filtros.salpicadero_target.getVelocidadAngular()  
        estado = self.cadena_filtros.salpicadero_target.getEstadoMotor()  
  
        revoluciones = self.cadena_filtros.ejecutar(revoluciones, estado)  
  
        return revoluciones
```

Figura 32: Implementación clase GestorFiltros.

Por otro lado el ejecutar de gestor de filtros lo que hace es obtener la velocidad angular actual y el estado actual del motor, para posteriormente pasárselo a las cadena de filtros que y ejecutar dicha cadena, obteniendo de vuelta las revoluciones y devolviendoselas al cliente. Por otro lado vemos que el gestor de filtro recibiera mediante setTarget el salpicaderoTarget para posteriormente pasarselo a cadena filtros usando un método que tiene dicha clase con el mismo nombre, ya que cadena será la encargada de llamar al método ejecutar de la clase salpicadero.

4.4. Implementación del Salpicadero

Ahora vamos a explicar como hemos implementado el salpicadero que seria nuestra clase objetivo si hablamos del patrón filtro interceptación, en esta clase vamos a tener los atributos que deseamos mostrar finalmente, es decir, estado del motor, velocidad lineal, velocidad angular(RPM) y distancia recorrida.

Como funciones vamos a tener el ejecutar que básicamente calcula la velocidad lineal y actualiza el estado del motor, aparte tendremos varios print que nos mostraran estado del motor y el salpicadero que se nos pide en el ejercicio y los setters y getters de los atributos descritos anteriormente.

```
class SalpicaderoTarget():
    def __init__(self, velocidad_lineal, velocidad angular, distancia, estado_motor):
        self.radio_eje = 0.15
        self.estado_motor_actual = estado_motor
        self.velocidad_lineal = velocidad_lineal
        self.velocidad angular = velocidad angular
        self.distancia = distancia

    def ejecutar(self, revoluciones, estado_motor):
        self.estado_motor_actual = estado_motor
        self.velocidad_lineal = 2 * math.pi * self.radio_eje * revoluciones * (60 / 1000)

        if self.velocidad_lineal <= 0:
            self.setVelocidadLineal(0)
```

Figura 33: Implementación clase SalpicaderoTarget.

En la imagen anterior mostramos las dos funciones principales de esta clase que es su constructor y el método ejecutar, omitimos los prints, getters y setters ya que no creemos necesario mostrar como hemos implementado dichas funciones.

4.5. Implementación cliente

La implementación de esta clase es muy sencilla ya que simplemente tenemos el constructor y una función llamada peticiónFiltros() que lo que hace es una vez pasado el gestor de filtros en el constructor llamar a su método ejecutarFiltros() que como ya hemos explicado anteriormente lo que hacía era ejecutar las distintas cadenas de filtros para obtener las revoluciones y posteriormente dichas revoluciones junto con el estado del motor pasarselas al ejecutar SalpicaderoTarget.

4.6. Implementación Main

El main se encarga de crear el target(salpicadero), el gestor de filtros, la cadena de filtros , los filtros respectivos y de crear también el cliente.

A la cadena de filtros se le añaden los diferentes filtros, esta se le añade al gestor de filtros, y este último tiene el cliente (El cliente es el que realmente se encarga de empezar la operación de ejecutar filtros).

El target es invocado como se refleja en el diagrama UML la cadena de filtros.

Una vez comprobamos que todo funciona y que con diferentes datos en los atributos del salpicadero target nos disponemos a crear la interfaz gráfica.

Esta consta de una ventana principal (pedalera) que consta de tres botones (Acelerar / soltar acelerador , encender y apagar), esta pantalla tiene una hija que es la que muestra los valores del salpicadero que nos interesan, y actualizándose cada segundo según parámetros y estado del motor.

A su vez invocamos un hilo que será el que realmente se encargue de ir actualizando los parámetros que pasamos a la interfaz gráfica.

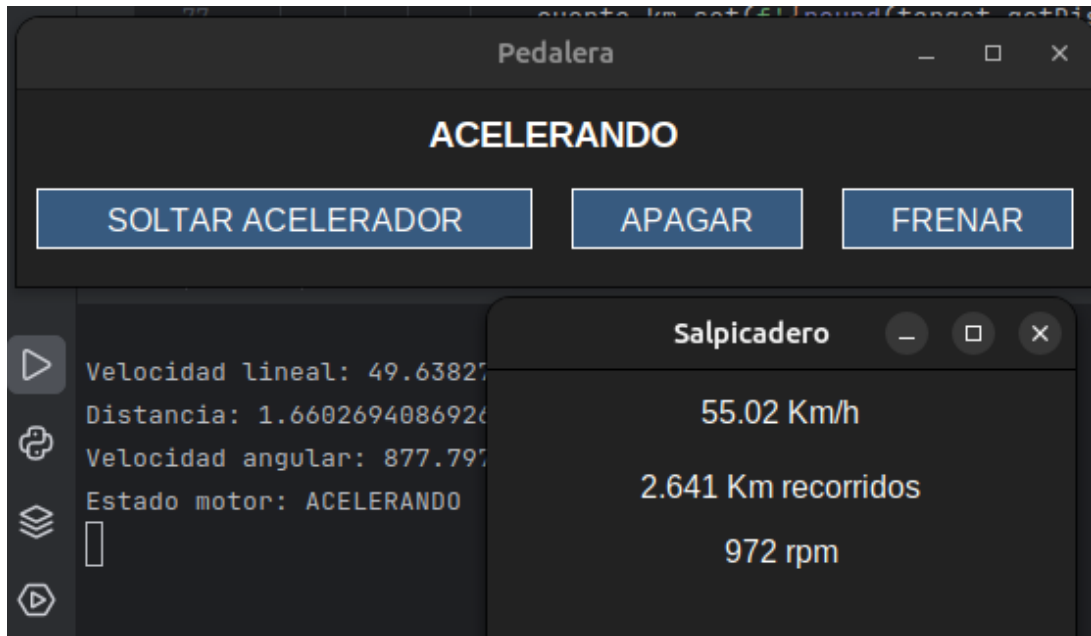


Figura 34: Interfaz gráfica con Tkinter.

5. Ejercicio Opcional : Web Scrapping (Patrón Estrategia)

El propósito de esta práctica es hacer uso del patrón de diseño conductual Estrategia a la par que aprendemos un poco más sobre las posibilidades de python en diferentes ámbitos.

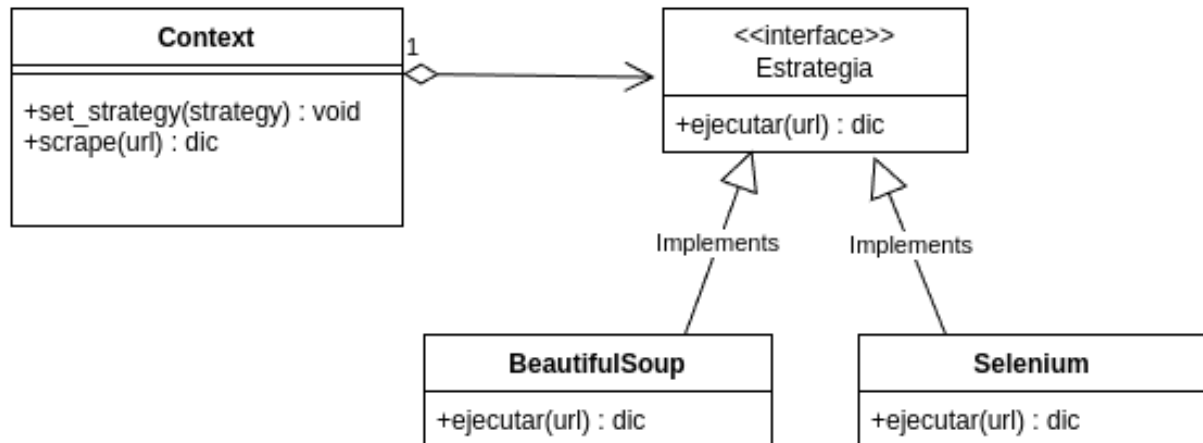


Figura 35: Diagrama UML ejercicio 5

En este caso las dos estrategias a valorar son usar BeautifulSoup o Selenium para conseguir el mismo objetivo este es el de conseguir los siguientes valores sobre TESLA en Yahoo Finance :

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

Además esta información la debemos guardar en un fichero .json que se actualizará con cada ejecución del programa con los nuevos valores recogidos.

EL grosor de la práctica, además de entender para qué y cómo se usa este patrón, está en conocer y aprender la manera de trabajar de estas dos herramientas para hacer web scrapping.

Selenium se desenvuelve mejor en páginas relativamente más complejas y dinámicas, aunque a costa de un mayor consumo de recursos computacionales. Esto se puede ver en la numerosa cantidad de funcionalidades disponibles para incluso simular el movimiento humano. BeautifulSoup sin embargo es más fácil de empezar a utilizar y, aunque más limitado en los sitios web que puede raspar, es ideal para proyectos más pequeños donde las páginas fuente están bien estructuradas.

Una vez con esa diferencia clara empezamos con BeautifulSoup, y es que al haber un valor de ejemplo ya recogido, los demás no son difíciles de encontrar. Para estos use el valor ya utilizado en el anterior , 'data-test', que se encuentra dentro de un td:

Y es que además con la herramienta de selección de elementos del DOM en la herramienta de inspección es muy fácil encontrar estos datos:


```

<tr class="C($primaryColor) W(51%)> == $0
  <span>Previous Close</span>
</td>
<td class="Ta(end) Fw(600) Lh(14px)" data-test="PREV_CLOSE-value">178.65</td>
</tr>

```

Figura 36: Encontrando el valor de cierre

52 Week Range	td.Ta(end).Fw(600).Lh(14px) 147.75 × 36	
Volume	85,118,590	Ex-Dividend Date
Avg. Volume	108,031,931	1y Target Est

Figura 37: Encontrando el valor de volumen

Una vez terminamos de recoger los datos estos los guardamos en un diccionario en python, el cual más tarde pasamos a JSON , este JSON es el que guardamos en el archivo más adelante.

Selenium tiene una forma parecida de trabajar aunque salvando las distancias. Como se menciona anteriormente posee muchas más funcionalidades y peso de cálculo ya que realmente simula la interacción con el navegador, hasta el punto que si quisiéramos aceptar las cookies podríamos aunque no fuera necesario:

```

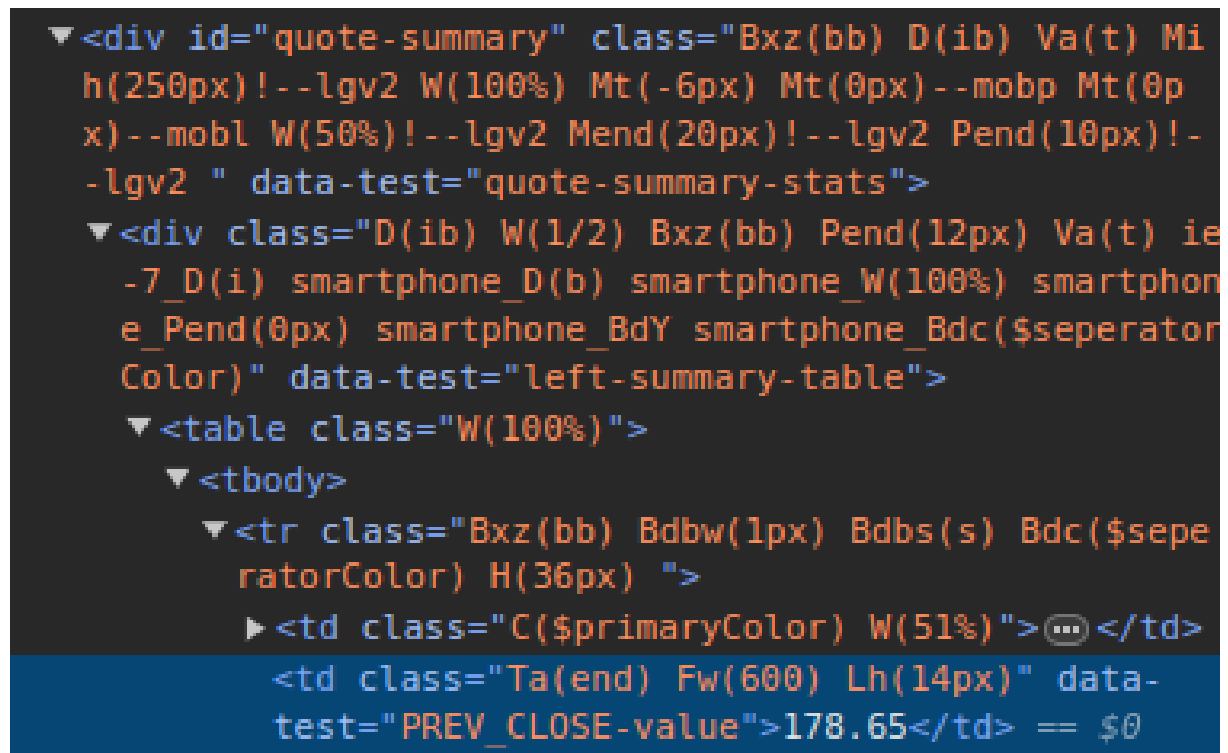
<button type="submit" class="btn secondary accept-all " name="agree" value="agree">Aceptar todo</button> == $0

```

Figura 38: Click al botón de las cookies

Los elementos en esta herramienta también se pueden detectar de diferentes maneras, por tag, por name por clases, por xpath ...

En nuestro caso hemos decidido usar el By.XPATH, ya que comprendiendo la sintaxis es fácil llegar hasta el elemento que queremos, por ejemplo este es uno de los path para llegar al valor de apertura : '//*[@id="quote-summary"]/div[1]/table/tbody/tr[2]/td[2]'.



```

▼ <div id="quote-summary" class="Bxz(bb) D(ib) Va(t) Mi
h(250px)!--lgv2 W(100%) Mt(-6px) Mt(0px)--mobp Mt(0p
x)--mobl W(50%)!--lgv2 Mend(20px)!--lgv2 Pend(10px)!-
-lgv2 " data-test="quote-summary-stats">
▼ <div class="D(ib) W(1/2) Bxz(bb) Pend(12px) Va(t) ie
-7_D(i) smartphone_D(b) smartphone_W(100%) smartphon
e_Pend(0px) smartphone_BdY smartphone_Bdc($separator
Color)" data-test="left-summary-table">
▼ <table class="W(100%)">
  ▼ <tbody>
    ▼ <tr class="Bxz(bb) Bdbw(1px) Bdbb(s) Bdc($sepe
ratorColor) H(36px) ">
      ▶ <td class="C($primaryColor) W(51%)>...</td>
        <td class="Ta(end) Fw(600) Lh(14px)" data-
test="PREV_CLOSE-value">178.65</td> == $0

```

Figura 39: Elementos HTML anidados hasta llegar al elemento

Con esto ya hemos terminado con las dos estrategias y con el requisito de generar un JSON con los datos.

5.1. Modularizando el código

Al separar el código en sus respectivos ficheros nos queda esta organización:

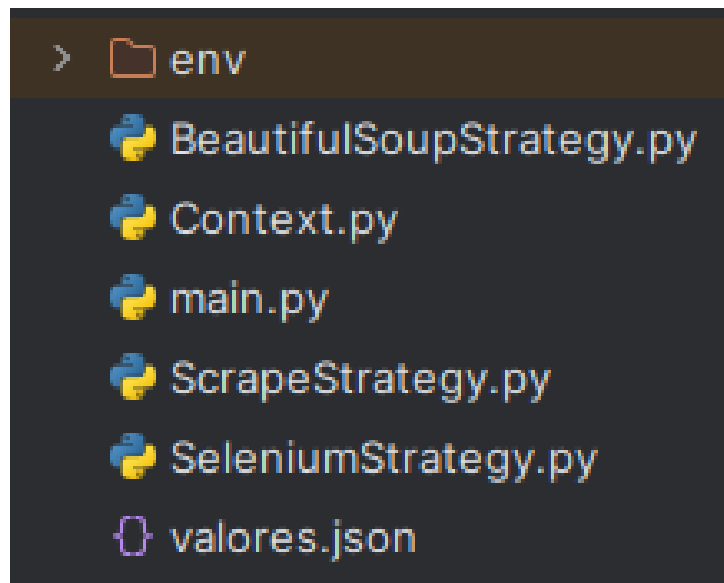


Figura 40: Organización de ficheros

Además para estructurar más el tema de variables como el tipo de componente HTML del DOM a buscar, tag, y valores hemos almacenado estos en atributos de sus respectivas clases, en concreto los valores a buscar se han almacenado en un diccionario con los valores o path-nombre como clave-valor. Haciendo esto el método de scrapear en BeautifulSoupStrategy se nos queda de la siguiente manera:

```
1 usage (1 dynamic)  JuanmiAcosta
def scrape(self, url):

    response = requests.get(url)

    if response.status_code == 200:

        soup = BeautifulSoup(response.content, 'html.parser')

        for i in range(len(self.valoresABuscar)):
            self.buscarYObtenerValores(soup, i)

        return self.valores

    else:

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

Figura 41: Scrape en BeautifulSoup