DESARROLLO DEL SOFTWARE

Práctica 1 - Uso de patrones de diseño creacionales y estructurales en OO

Jose Francisco López Rubio
Jorge Navarro Molina
Carmen Quiles Ramírez
Óscar Picado Cariño

09/03/2024

UNIVERSIDAD DE GRANADA

Tabla de contenidos

1. Introducción

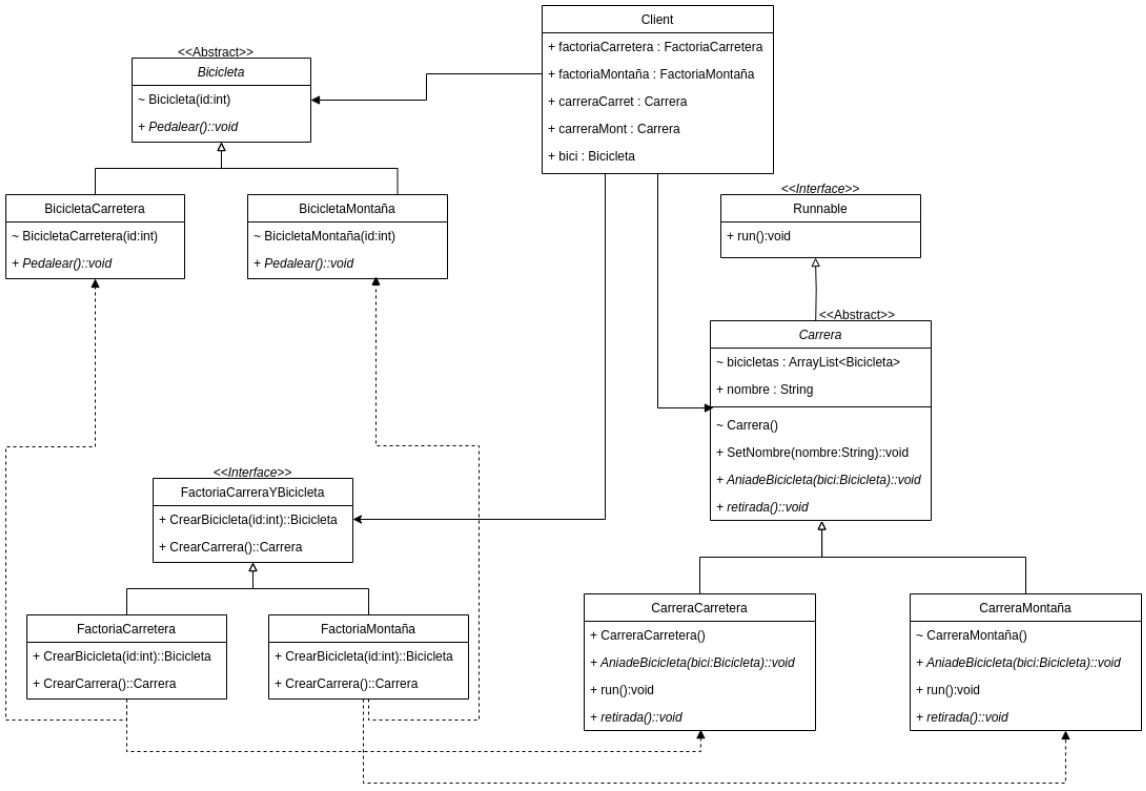
En esta primera práctica, hemos hecho uso de nuestro conocimiento en determinados patrones de diseño creacionales y estructurales y lo hemos aplicado a los diferentes ejercicios propuestos.

2. Desarrollo de la práctica

Para seguir el trabajo de forma más transparente y eficiente, se ha hecho uso de un repositorio de GitHub: https://github.com/jorgenavmol/DS-Practicas. Cada uno hemos clonado ese repositorio en los ordenadores locales y así, seguir el trabajo de otros fácilmente.

2.1. Ejercicio 1: Patrón Factoría Abstracta (Java)

Para este ejercicio hemos tenido que usar el patrón *Factoría Abstracta* junto con el patrón de diseño *Metodo Factoría*, aplicandolo a un ejemplo de carreras de bicis de dos tipos: de carretera y de montaña.



Para realizar la simulación de ambas carreras hemos creado 2 carreras (hebras) de N bicicletas (N se pasa por teclado a gusto del usuario) a partir de las clases creadoras concretas *FactoriaMontana*

y *FactoriaCarretera*. Una vez creadas, a ambas les añadimos las bicicletas que vamos creando sobre la marcha a partir de las clases *Factorias* segun el tipo de carrera que vayan a correr.

Una vez todo creado, ahora sí crearemos las hebras que representan cada una de las carreras que se van a realizar. Esto lo haremos a partir de la interfaz *Runnable*, la cual proporciona la funcionalidad de las hebras y de la cual hereda la clase *Carrera* para poder simularla correctamente. Las comenzamos a la vez, tanto la de montaña como la de carretera y esperamos 60 segundos antes de interrumpirlas.

2.1.1. Funcionamiento

Todo el ejercicio parte a partir de la interfaz *FactoriaCarreraYBicicleta*, la cual declara las dos funciones principales para crear biciletas y carreras. Las dos clases abstractas que implementan los metodos de dicha interfaz son *FactoriaCarretera* y *FactoriaMontana*. Estas dos, para implementarlas, utilizan los constructores de las clases relacionadas respectivas, es decir, por ejemplo *FactoriaCarretera* utiliza los constructores de *CarreraCarretera* y *BicicletaCarretera*.

Una vez declaradas las clases creadoras, pasamos a las creadas. La clase abstracta *Bicicleta* tiene un atributo *id*, el cual sirve para poder identificar y diferenciarla de las demás. Tiene un constructor basico y un metodo *pedalear()*, tan solo declarado para que su implementación se diferencia segun si es de carretera o de montaña.

La clase abstracta *Carrera* tiene una implementación bastante más desarrollada que la de *Bicicleta*. Sus atributos son un vector de bicicletas, junto con el nombre de la propia carrera, y un entero ya inicializado que define el instante en el que se retirarán las bicicletas durante la realización de la carrera. El constructor simplemente inicializa el vector de bicicletas, y el unico método no abstracto que implementa es *SetNombre()*, que como su propio nombre indica, establece el nombre de la carrera. La implementación de los otros dos métodos abstractos se delega a las concretas, la cual cambiara según la clase correspondiente.

```
public abstract class Carrera implements Runnable{
    protected ArrayList<Bicicleta> bicicletas;
    public String nombre;
    int instanteRetirada = 4;

    Carrera(){
        bicicletas = new ArrayList<>();
    }
    public void SetNombre(String nombre){
        this.nombre = nombre;
    }
    public abstract void aniadeBicicleta(Bicicleta bici);
    public abstract void retirada();
}
```

Cada una de las clases de bicicletas que heredan de las anteriores abstractas mencionadas tienen un constructor, el cual simplemente utiliza el constructor de la superclase, y un método *pedalear()* que imprime un mensaje junto con su identificador cada vez que se llama.

```
public class BicicletaCarretera extends Bicicleta {
    BicicletaCarretera(int id){
        super(id);
    }

    @Override
    public void pedalear(){
        System.out.println("Corre una bicicleta de carretera de id: " + super.id);
    }
}
```

Las clases que heredan *Carrera* implementan 3 métodos declarados en clases superiores: un método *aniadeBicicleta*, un método *run*, declarado en la clase Runnable, y el cual se encarga de lanzar la hebra y que corran todas las bicicletas del vector de la clase; y finalmente un metodo *retirada*, el cual se encarga de retirar las bicicletas, cuya cantidad depende del tipo de carrera que se trata. El valor del instante de retirada es meramente arbitrario, de forma que pase un tiempo desde que comenzó la carrera antes de ocurrir la retirada.

```
@Override
public void run() {
    int contador = 0;
    try {
        while (!Thread.interrupted()) {
            contador++;
            if(contador == instanteRetirada){
                this.retirada();
            }
            for (Bicicleta bicicleta : bicicletas) {
                ((BicicletaCarretera)bicicleta).pedalear();
            }
            Thread.sleep(1000); // Simular una iteración de la carrera cada segundo
        }
    } catch (InterruptedException e) {
        System.out.println(nombre + " finalizada.");
    }
}
```

```
@Override
public void retirada(){
    int N = bicicletas.size();
    double retiradas = N*0.1;
    Random random = new Random();

    for(int i = 0; i < retiradas; i++){
        int rand = random.nextInt(bicicletas.size());
        //Para que se retine un mínimo de 1 bicicleta, N debe ser 10 o más
        System.out.println("Se retira la bicicleta de carretera: " + bicicletas.get(rand).id);
        bicicletas.remove(rand);
    }
}
```

Un dato a añadir es que para que se de la retirada de al menos una bicicleta, en una carrera de carretera es necesario que haya como mínimo 10 bicicletas corriendo (se retira el 10 porciento de bicicletas), y en una carrera de montaña es necesaria la cantidad mínima de 5 bicicletas corriendo (se retira el 20 porciento de bicicletas).

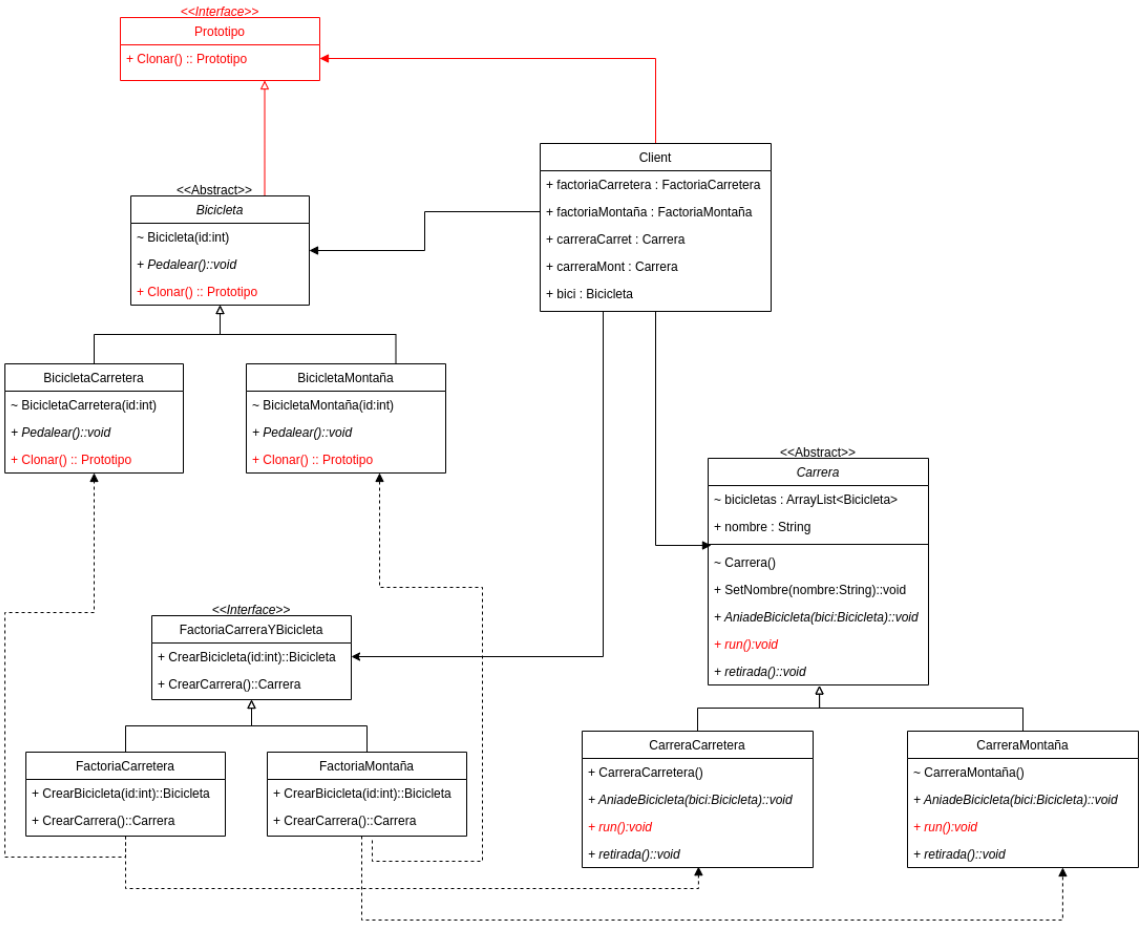
2.1.2. Ejemplo de ejecución

Para compilar, una vez situados en la **carpeta P1**, se usará el siguiente comando: **javac E1/*.java**
Una vez compilado ejecutaremos **Main**, para ello hacemos: **java E1.Main**

```
Corre una bicicleta de carretera de id: 7
Corre una bicicleta de carretera de id: 8
Corre una bicicleta de carretera de id: 9
Corre una bicicleta de montaña de id: 16
Corre una bicicleta de montaña de id: 17
Corre una bicicleta de montaña de id: 18
Corre una bicicleta de montaña de id: 19
Se retira la bicicleta de montaña: 16
Se retira la bicicleta de carretera: 6
Corre una bicicleta de carretera de id: 0
Corre una bicicleta de carretera de id: 1
Se retira la bicicleta de montaña: 15
Corre una bicicleta de montaña de id: 10
Corre una bicicleta de carretera de id: 2
Corre una bicicleta de carretera de id: 3
Corre una bicicleta de carretera de id: 4
Corre una bicicleta de carretera de id: 5
```

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

En el siguiente ejercicio, hemos aplicado el patrón *Prototipo* junto con el patrón *Factoría Abstracta* en el lenguaje de Python a la implementación del ejercicio anterior.



Sin embargo, esta versión del ejercicio varía en varios ámbitos:

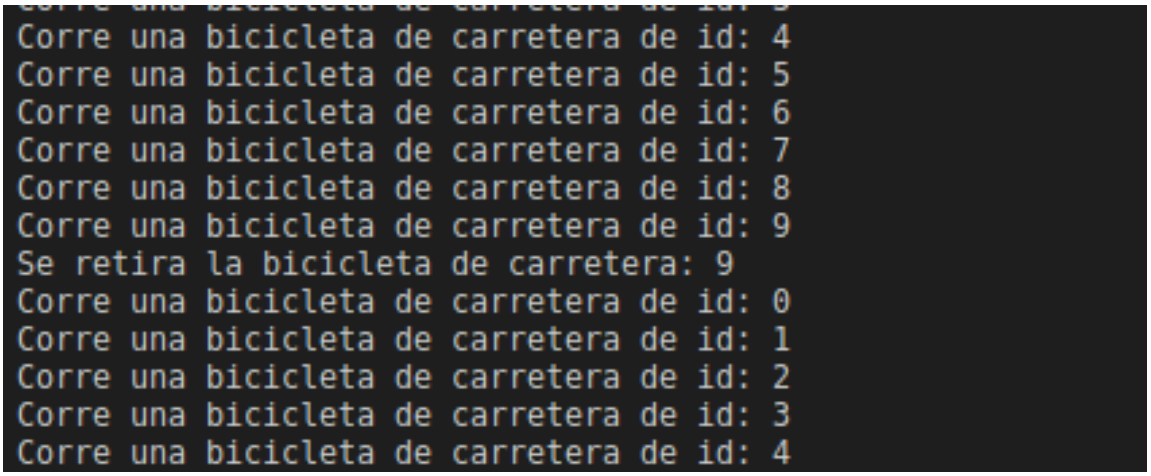
Primero, para simplificar más las cosas, no hemos hecho uso de hebras para la creación de las carreras, ya que en lenguaje Python la manipulación de estas es mucho mas compleja que en Java. Seguimos utilizando una función *run()*, pero esta ya no hereda de la clase *Runnable*.

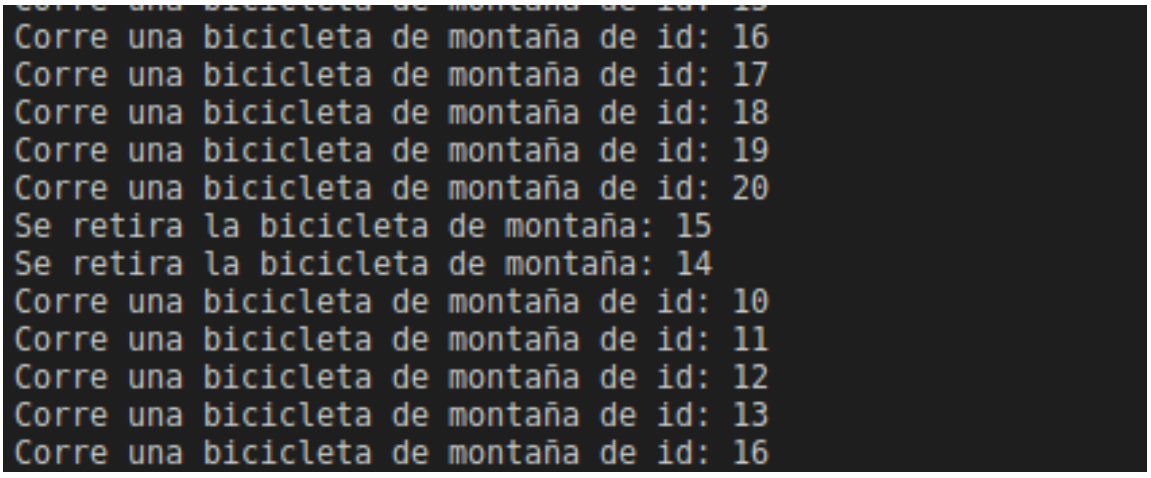
También hacemos adicionalmente uso del patrón *Prototype*, el cual sirve para clonar objetos de clases, reduciendo el trabajo de la propia creación de nuevos objetos. Este patrón lo hemos aplicado en concreto para la creación de bicicletas para cada carrera, donde creamos una bicicleta de cada tipo, y a partir de cada una vamos haciendo clones, modificando sus identificadores con el método añadido *setId()* de la clase Bicicleta, ya que al clonar se clonan también todos los atributos.

```
bici = factoria_carretera.crearBicicleta(0)
carrera_carretera.aniadeBicicleta(bici)

for i in range(N-1):
    bici = bici.clonar()
    bici.setId(i+1)
    carrera_carretera.aniadeBicicleta(bici)
```

2.2.1. Ejemplo de ejecución

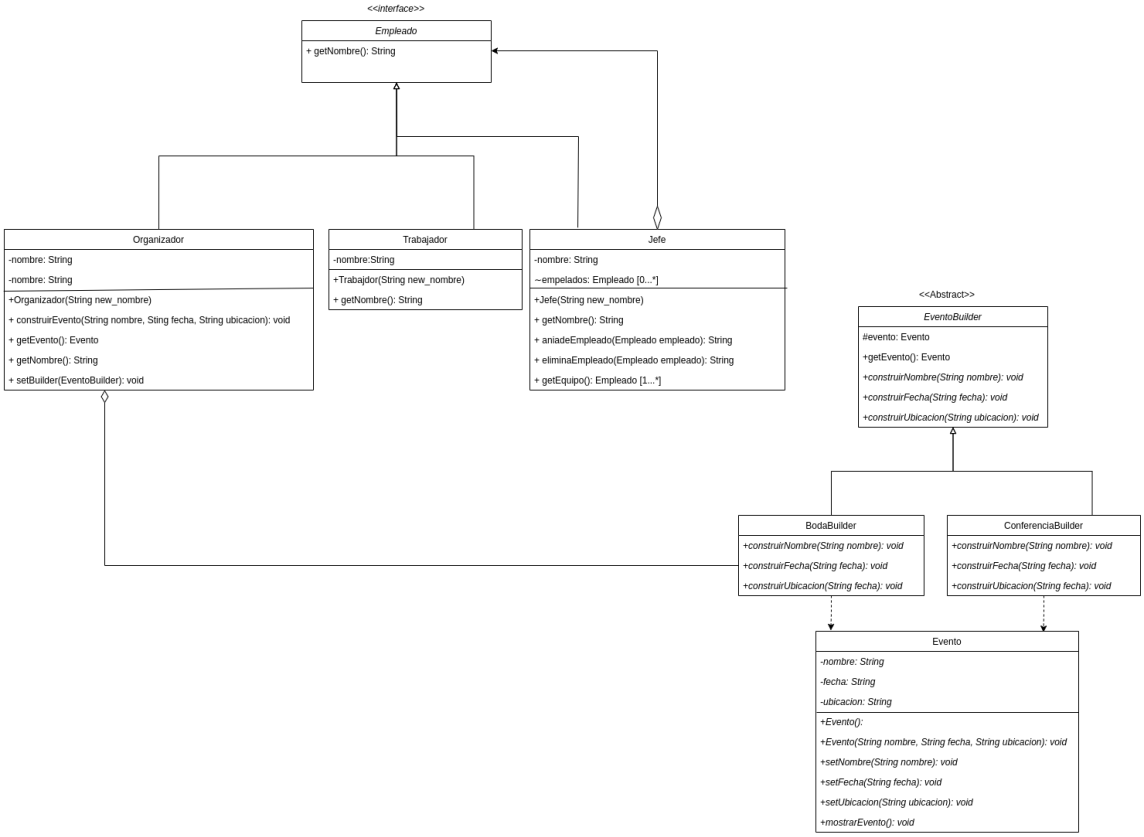
Una vez situados en la **carpeta P1**/Para ejecutar **Main**, utilizamos el siguiente comando: **python3 Main.py 10**





2.3. Ejercicio 3: Patrón Libre (Java)

En este ejercicio, se propuso realizar una implementación con patrones de diseño libres. Nuestra propuesta es la siguiente:



2.3.1. Exposición del problema

Se trata de un comité de organización de eventos con una jerarquía de trabajo del estilo Jefe-Empleado, que simula la jerarquía existente en empresas reales representado por el *Patrón Composite*: un *Jefe* tiene varios *Empleados* a su cargo, que pueden ser *Trabajadores* u otros Jefes de menor cargo. Sin embargo, surgió el siguiente problema: ¿quien toma el rol de "Clase Directora" y maneja la creación de eventos?

Así, surgió el empleo de *Organizador*, limitando a que sólo empleados de este tipo pudieran crear eventos y simplificando la implementación. Este comité se encarga de la gestión y organización de varios eventos, como lo es una Boda o una Conferencia, haciendo uso del *Patrón Builder*. El director, por ende, será el *Organizador*, pues es el que posee los conocimientos para poder generar eventos de este tipo.

```
public interface Empleado{
    public String getNombre(); //Equivalente al método mostrar() de Composite
}

public class Jefe implements Empleado{
    private String nombre;
    ArrayList<Empleado> empleados = new ArrayList<Empleado>(); //Empleados a su cargo
    {...}
}

public class Trabajador implements Empleado{
    private String nombre;
    {...}
}

public class Organizador implements Empleado{
    private EventoBuilder builder;
    private String nombreOrg;
    public void construirEvento(String nombre, String fecha, String ubicacion)
    {...}
}
```

Las clases *Organizador*, *Trabajador* y *Jefe* heredan de una interfaz, *Empleado*, pues, por encima de su especialización en el área, todos son en esencia empleados de un Jefe. Además, el *Organizador* es el que posee el objeto constructor para poder generar un Evento mediante *construirEvento()*, ocultando la creación al resto de clases.

2.3.2. Funcionamiento

El programa principal esta hecho mediante un menú interactivo, en el que el usuario (normalmente será un administrador) escribe manualmente los datos que se indican. Su funcionamiento es el siguiente:

- Primero, se pide por pantalla los *Jefes* al cargo del proyecto, sus Empleados asociados y su rol en el proyecto. Conforme se van introduciendo, se van creando dependiendo del tipo de *Empleado* que sean y añadiéndolo a la lista del jefe correspondiente.

  ```
  Scanner scanner = new Scanner(System.in);
  ArrayList<Jefe> jefes = new ArrayList<Jefe>();
  ArrayList<Organizador> organizadores = new ArrayList<Organizador>();

  System.out.print("Numero de jefes deseados: ");
  int numJefes = scanner.nextInt(); {...}
  ```

```
for(int i=0; i<numJefes; i++){
    //Se introduce el nombre del jefe (i) a crear
    jefes.add(new Jefe(nombreJefe));

    //Se introduce el numero de empleados a cargo de dicho jefe
    for(int j=0; j<numEmpleados; j++){
        //Se introduce el nombre y tipo del Empleado
        switch (tipoEmpleado) {
            case 1:
                //Se crea el organizador
                organizadores.add(organizador);
                jefes.get(i).aniadeEmpleado(organizador);
                break;

            case 2:
                //Se crea el jefe
                jefes.get(i).aniadeEmpleado(jefe);
                break;

            case 3:
                //Se crea el trabajador
                jefes.get(i).aniadeEmpleado(trabajador);
                break;

            default:
                //Mensaje de error
        }
    }
}
```

- En segundo lugar, abordamos una situación .ºespecial". Para la construcción del evento, se necesita como mínimo un *Organizador*, y en esta sección se aborda este problema. Si no se ha creado ninguno, se da la opción de crearlo y añadirlo a un jefe existente. Posteriormente, mostramos todos los organizadores existentes.

```
if (organizadores.size() == 0){
    //Se crea el organizador con el nombre y se muestran los jefes existentes para asociarlo
    for (int i=1; i<=jefes.size(); i++){
        System.out.print(i + "- " + jefes.get(i-1).getNombre() + "\n");
    }
```

```
    int indice = scanner.nextInt() - 1;
    jefes.get(indice).aniadeEmpleado(organizador);
    organizadores.add(organizador);
}

//Lista los organizadores y se elige uno de ellos
for (int i=1; i<=organizadores.size(); i++){
    System.out.print(i + "- " + organizadores.get(i-1).getNombre() + "\n");
}
int organizadorElegido = scanner.nextInt()-1; {...}
```

- Por último, hacemos que el organizador seleccionado cree el evento, usando los parámetros introducidos manualmente. Acabamos mostrando los detalles del evento y la jerarquía del comité.

```
EventoBuilder builder;
Organizador directorOrg;
String nombreEvento, fechaEvento, ubicacionEvento;

//Se introduce el tipo de evento que queremos y creamos un constructor según éste.
if (tipoEvento ==  1) {
    builder = new ConferenciaBuilder();
    directorOrg = organizadores.get(organizadorElegido);
    directorOrg.setBuilder(builder);
    {...}
    //Se introducen los parámetros del evento y lo construimos
    directorOrg.construirEvento(nombreEvento, fechaEvento, ubicacionEvento);
    {...} //Mostramos los detalles del evento
}

//Se siguen los mismos pasos para la boda
else if (tipoEvento == 2) {...}

//Mostramos los jefes y sus empleados.
for(int i=0; i<jefes.size(); i++){
    System.out.println("Empleados a cargo de " + jefes.get(i).getNombre() + ":");
    ArrayList<Empleado> cargo = jefes.get(i).getEquipo();
    for(int j=0; j<cargo.size(); j++){
        System.out.println(cargo.get(j).getNombre());
    }
}
```

Hemos optado por esta implementación y modificado un poco el diseño de la clase Directora ya que queríamos representar que, aunque solo un objeto Organizador cree un evento, fuera el rol el que fuera capaz de ser *Builder* (es decir, asociar la clase Directora a un rol, más que a una persona). Por ello, no construimos el objeto director con el Builder, sino que creamos el objeto, mostramos todos los organizadores existentes, y luego le asociamos el objeto constructor al organizador elegido, pudiendo crear y establecer la jerarquía y posteriormente asociar el objeto constructor.

```
public Organizador(String nombreOrg) {
    this.nombreOrg = nombreOrg;
}

public void setBuilder(EventoBuilder builder) {
    this.builder = builder;
}
```

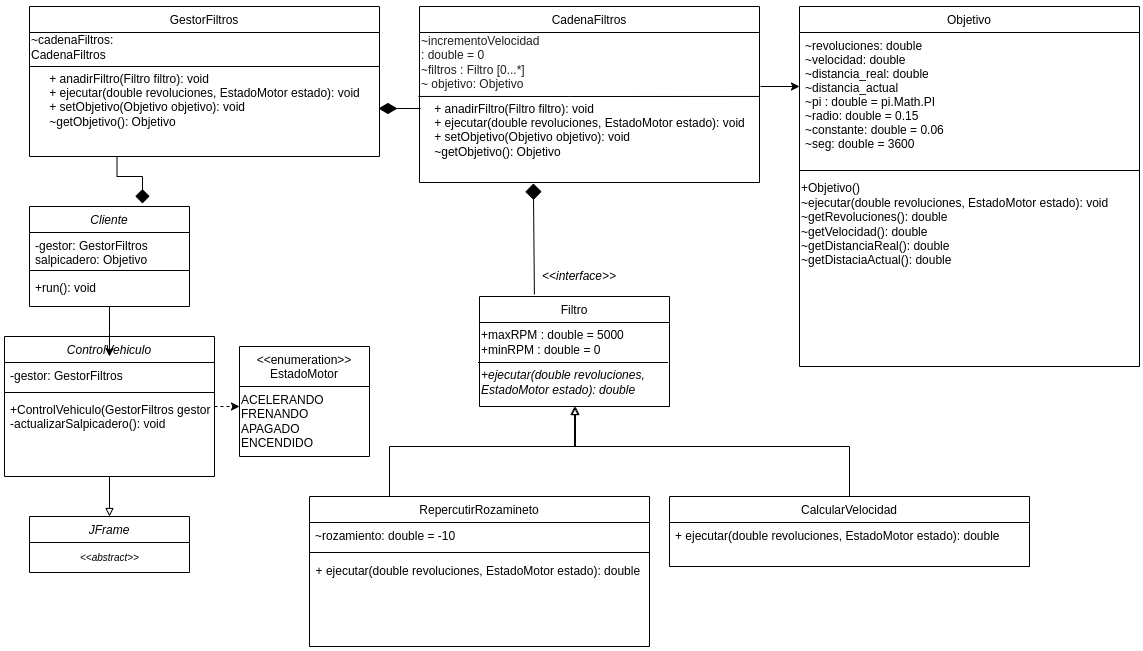
2.3.3. Ejemplo de ejecución

Para compilar, una vez situados en la **carpeta P1**, se usará el siguiente comando: **javac E3/*.java** Una vez compilado ejecutaremos **Main**, para ello hacemos: **java E3.Main**

```
Numero de jefes deseados: 2
Nombre del jefe numero 1: Jose
Numero de empleados a cargo de Jose: 1
Nombre del empleado numero 1: Oscar
¿Qué tipo de empleado es Oscar? (Organizador=1, Jefe=2, Trabajador=3): 1
Nombre del jefe numero 2: Carmen
Numero de empleados a cargo de Carmen: 1
Nombre del empleado numero 1: Jorge
¿Qué tipo de empleado es Jorge? (Organizador=1, Jefe=2, Trabajador=3): 1
¿Cual de los siguientes empleados desea que organice el evento?:
1- Organizador - Oscar
2- Organizador - Jorge
2
¿Qué tipo de evento desea crear? (Conferencia=1, Boda=2): 2
Nombre del evento: Boda Granada
Fecha del evento: 10/05/2024
Ubicación del evento: Armilla

----------------------------------------
Organizador - Jorge
Nombre del evento: Boda Boda Granada
Fecha del evento: 10/05/2024
Ubicacion del evento: Armilla
----------------------------------------

Empleados a cargo de Jefe - Jose:
Organizador - Oscar
Empleados a cargo de Jefe - Carmen:
Organizador - Jorge
```

2.4. Ejercicio 4: Patrón Filtros de Intercepción (Java)

En este ejercicio implementaremos el funcionamiento del salpicadero de un coche usando el patrón (estilo) arquitectónico filtros de interacción. Además, será necesario la realización de una interfaz gráfica.



2.4.1. Exposición del problema y funcionamiento

Se quiere representar en el salpicadero de un vehículo los parámetros más relevantes del movimiento del mismo, estos son: velocidad (calculada a partir de las revolcuiones del motor), RPM y contadores de kilómetros (tanto actual como real). Estas revoluciones serán primero filtradas mediante software independiente al sistema capaz de calcular el cambio de las revoluciones como consecuencia del estado del motor (*EstadoMotor*) y del rozamiento. Se crearán dos filtros *CalcularVelocidad* y *RepercutirRozamiento* que implementan la interfaz *Filtro* para calcular las revoluciones y modificar las mismas en base al rozamiento.
Entidades de modelado:

- Objetivo: Se necesitan las variables revoluciones, velocidad, distancia real y distancia actual para almacenar el valor de estos parámetros y poder ir actualizándolos usando el método ejecutar. Este se encarga de actualizar las revoluciones, calcular la velocidad haciendo uso de algunas constantes y aumentar la distancia recorrida según el estado del motor. Además esta clase cuenta con varios métodos get para acceder a sus argumentos y mostrarlos desde otras clases.

```
public class Objetivo{
    double revoluciones, velocidad, distancia_real, distancia_actual;
    const double PI, radio, seg, constante;
    {...}

    void ejecutar(double revoluciones, EstadoMotor estado){
        this.revoluciones = revoluciones;
        this.velocidad = 2 * pi * radio * revoluciones * constante;
        if (estado == EstadoMotor.APAGADO){
            this.distancia_actual = 0;
        }
        else if(estado == EstadoMotor.ACELERANDO){
            distancia_actual += velocidad/seg;
            distancia_real += velocidad/seg;
        }
        else if (estado == EstadoMotor.FRENANDO && velocidad > 0){
            distancia_actual += velocidad/seg;
            distancia_real += velocidad/seg;
        }
    }
    {...}
}
```

- Filtro: Es una interfaz implementada por las clases *CalcularVelocidad* y *RepercutirRozamiento*. Las constantes declaradas en la interfaz sirven para asignar un máximo y un mínimo en las revoluciones del motor. La clase CalcularVelocidad asigna un nuevo valor a las revoluciones del motor haciendo uso de las revoluciones iniciales (pasadas como parámetro) y del estado en el que se encuentra el motor. La clase RepercutirRozamiento simula el rozamiento que afecta al vehículo reduciendo sus revoluciones en un número constante, en caso de que se alcancen el máximo o el mínimo de revoluciones se devuelven estas mismas (máxima o mínima). Así, quedarían aplicados ambos filtros a los parámetros del objetivo.

```
interface Filtro {
    double maxRPM = 5000;
    double minRPM = 0;

    abstract double ejecutar(double revoluciones, EstadoMotor estado);
}

public class CalcularVelocidad implements Filtro{
    public double ejecutar(double revoluciones, EstadoMotor estado){
        double incrementoVelocidad = 0;
```

```
            switch(estado) {
                case ACELERANDO:
                    incrementoVelocidad = 30.0;
                    break;
                case FRENANDO:
                    incrementoVelocidad = -30;
                    break;
                case APAGADO:
                    incrementoVelocidad = 0; revoluciones = 0;
                    break;
                case ENCENDIDO:
                    incrementoVelocidad = 0; revoluciones = 500;
                    break;
            }
            return revoluciones + incrementoVelocidad;
        }
    }

    public class RepercutirRozamiento implements Filtro{
        double rozamiento = -10;

        public double ejecutar(double revoluciones, EstadoMotor estado){
            if(revoluciones + rozamiento >= maxRPM) return maxRPM;
            else if (revoluciones + rozamiento <= minRPM) return minRPM;
            else return revoluciones + rozamiento;
        }
    }
```

- ControlVehiculo: En nuestro caso el cliente (main), además se aprovecha esta clase para la creación de la interfaz gráfica. En el main se crea un Objetivo y un GestorFiltros asociado a este obejtivo y se usa ControlVehiculo para el funcionamiento del mismo. Como puntos a destacar de la clase ControlVehiculo, en el constructor se manejan los atributos y carácterísticas de la interfaz, además de que cuando es necesario se llama a *gestor.cambio(gestor.cadenaFiltros.objetivo.revoluciones, EstadoMotor);*, función con la que se activa el funcionamiento de los filtros y los cambios necesarios en los atributos del Objetivo.

```
    public class ControlVehiculo extends JFrame {
        {atributos de la interfaz gráfica}

        private GestorFiltros gestor;
        private Timer acelerarTimer;
        private Timer frenarTimer;
```

```
    public ControlVehiculo(GestorFiltros gestor) {
        this.gestor = gestor;

        gestor.anadirFiltro(new CalcularVelocidad());
        gestor.anadirFiltro(new RepercutirRozamiento());

        {...}

        // Cambiar texto del botón Encender cuando está seleccionado
        encenderButton.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                if (encenderButton.isSelected()) {
                    {...}
                    gestor.cambio(gestor.cadenaFiltros.objetivo.revoluciones, EstadoMotor.ENCEN
                    actualizarSalpicadero();
                } else {
                    {...}
                    gestor.cambio(gestor.cadenaFiltros.objetivo.revoluciones, EstadoMotor.APAGA
                    actualizarSalpicadero();
                }
            }
        });

        {...}

// Crear el Timer para acelerar
        acelerarTimer = new Timer(100, new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                gestor.cambio(gestor.cadenaFiltros.objetivo.revoluciones, EstadoMotor.ACELERAND
                actualizarSalpicadero();
            }
        });

        // Comportamiento al pulsar y soltar el botón de acelerar
        acelerarButton.addActionListener(new ActionListener() {
            {...}
        });

        // Crear el Timer para frenar
        frenarTimer = new Timer(100, new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                gestor.cambio(gestor.cadenaFiltros.objetivo.revoluciones, EstadoMotor.FRENANDO)
```

```
                actualizarSalpicadero();
            }
        });

        // Comportamiento al pulsar y soltar el botón de frenar
        frenarButton.addActionListener(new ActionListener() {
            {...}
        });

        {...}
    }

    // Método para actualizar el salpicadero con los valores del GestorFiltros
    private void actualizarSalpicadero() {
        {...} //Cambios en las variables de la interfaz
    }

    public static void main(String[] args) {
        Objetivo salpicadero = new Objetivo();
        GestorFiltros gestor = new GestorFiltros(salpicadero);
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                ControlVehiculo controlVehiculo = new ControlVehiculo(gestor);
                controlVehiculo.setVisible(true);
            }
        });
    }
}
```

- CadenaFiltros: Se usa un ArrayList para almacenar y gestionar los filtros, además hay relación con el Objetivo para actualizar los valores una vez pasados todos los filtros (*this.objetivo.ejecutar(suma, estado);*).

```
public class CadenaFiltros {
    double incrementoVelocidad = 0;
    ArrayList<Filtro> filtros = new ArrayList<>();
    Objetivo objetivo;

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

```
    public void ejecutar(double revoluciones, EstadoMotor estado){
        double suma = revoluciones;
        for (Filtro filtro : filtros){
            suma = filtro.ejecutar(suma, estado); //Se pasan todos los filtros
        }
        this.objetivo.ejecutar(suma, estado); //Actualización del objetivo
    }
    {...}
}
```

- GestorFiltros: El gestor tiene asociada una cadena de filtros que mandará a ejecutar (método cambio), además de tener un Objetivo asociado y asignado a través del constructor. Este gestor puede añadir nuevos filtros además de ejecutarlos.

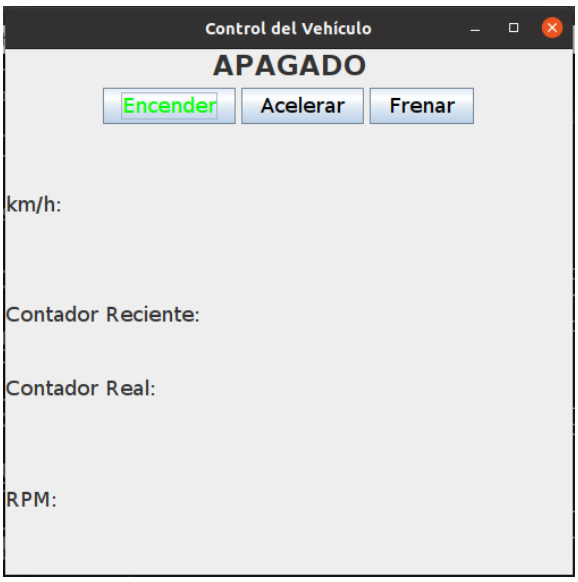
```
public class GestorFiltros {
    CadenaFiltros cadenaFiltros;

    public GestorFiltros(Objetivo objetivo){
        cadenaFiltros = new CadenaFiltros();
        cadenaFiltros.setObjetivo(objetivo);
    }

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

    public void cambio (double revoluciones, EstadoMotor estado){
        cadenaFiltros.ejecutar(revoluciones, estado);
    }
    {...}
}
```

- EstadoMotor: Enumerado utilizado para controlar el estado del motor en cada momento.

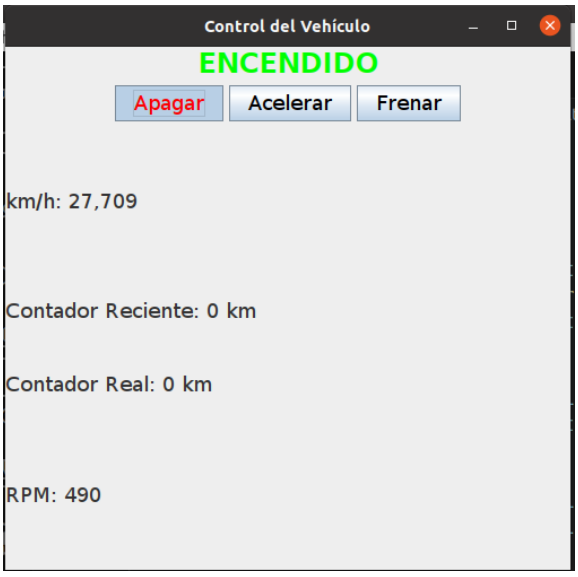
```
public enum EstadoMotor {
    ACELERANDO,
    FRENANDO,
    APAGADO,
    ENCENDIDO;
}
```

2.4.2. Ejemplo de ejecución

Para compilar, una vez situados en la **carpeta P1**, se usará el siguiente comando: **javac E4/*.java**
Una vez compilado ejecutaremos **ControlVehiculo**, para ello hacemos: **java E4.ControlVehiculo**



Vehículo apagado



Vehículo encendido y parado

Vehículo acelerando hasta que se deje de pulsar el acelerador o se pulse el freno

Vehículo frenando hasta que se deje de pulsar el freno o se pulse el acelerador

Vehículo apagado pero habiendo sido 'usado'

3. Conclusión

Durante la realización de esta práctica nos hemos enfrentado a diferentes problemas con los que hemos aprendido a implementar diferentes patrones de diseño como el Metodo Abstracto, Factoria Abstracta, el Patron Composite junto al builder y el Patron de Filtros de Implementación. También nos ha ayudado a recordar algunos conceptos de Java y hemos tenido una primera toma de contacto con python y LaTeX en la realización de la memoria. Por otro lado ha sido nuestra primera vez usando GitHub, lo que nos ha venido muy bien para poder repartirnos el trabajo, ir haciendo actualizaciones en el repositorio para ver el progreso y que todos podamos tener el proyecto en nuestros ordenadores.