IES DR LLUÍS SIMARRO
ENSENYAMENT PÚBLIC
XATIVA

Acceso a datos

# UD04. Patrones de diseño y librerías

## Desarrollo de Aplicaciones Multiplataforma

**Enric Giménez Ribes**

e.gimenezribes@edu.gva.es

# ÍNDICE

# 1. INTRODUCCIÓN

La **arquitectura software** es una descripción de los subsistemas y componentes computacionales de un sistema software, y de las relaciones entre ellos. Es el resultado de la actividad de diseño arquitectónico del software.

La arquitectura de software, tiene que ver con el diseño y la implementación de estructuras de software de alto nivel. Es el resultado de ensamblar un cierto número de elementos arquitectónicos de forma adecuada para satisfacer la mayor funcionalidad y requerimientos de desempeño de un sistema, así como requerimientos no funcionales, como la confiabilidad, escalabilidad, portabilidad, y disponibilidad.

Por ello se suelen usar **patrones de diseño,** cada uno de ellos adecuado para  tipo de problema y unas características específicas.

La **ventaja principal** de este estilo es que el desarrollo se puede llevar a cabo en varios niveles y, en caso de que sobrevenga algún cambio, solamente se ataca al nivel requerido sin tener que revisar entre código mezclado.

Además, permite distribuir el trabajo de creación de una aplicación por niveles; de este modo, cada grupo de trabajo está totalmente abstraído del resto de niveles, de forma que basta con conocer la API que existe entre niveles.

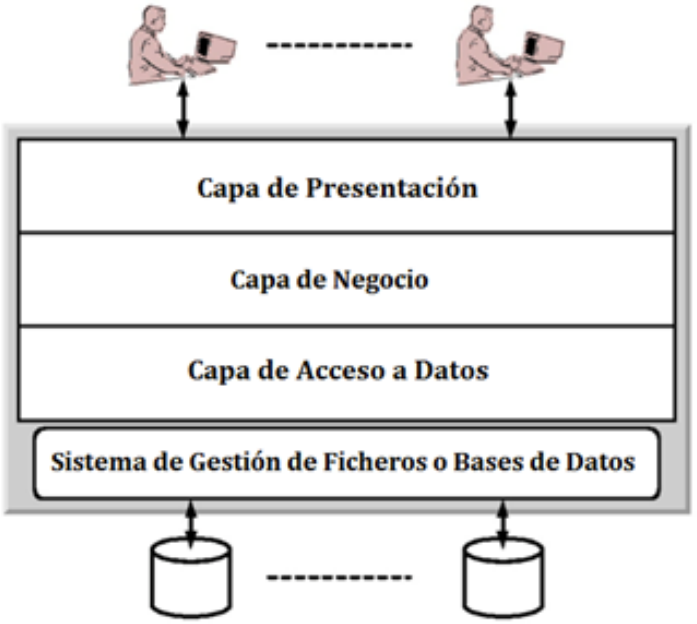
En el diseño de sistemas informáticos actual se suelen usar las arquitecturas multinivel o programación por capas. En dichas arquitecturas a cada nivel se le confía una misión simple, lo que permite el diseño de arquitecturas escalables (que pueden ampliarse con facilidad en caso de que las necesidades aumenten).

El diseño más utilizado actualmente es el diseño en tres niveles (o en tres capas).

Las llamadas (y retorno) en este tipo de diseño ha de cumplir lo siguiente:

- Los componentes (objetos y clases) se agrupan en capas.
- La comunicación (relaciones entre las clases, mensajes entre objetos) solamente se producen entre elementos de la misma capa o de capas contiguas.

La **arquitectura de 3 capas** (en inglés Three-Tier Architecture) sigue el siguiente esquema:




Vamos a describir las 3 capas:

- **Capa de presentación:** es la que ve el usuario (también se la denomina "capa de usuario"), presenta el sistema al usuario, le comunica la información y captura la información del usuario en un mínimo de proceso (realiza un filtrado previo para comprobar que no hay errores de formato). También es conocida como interfaz gráfica y debe tener la característica de ser "amigable" (entendible y fácil de usar) para el usuario. Esta capa se comunica únicamente con la capa de Negocio mediante invocaciones a acciones y consultas, y recibiendo las respuestas y resultados. Básicamente se ocupa de:
  - Entender las peticiones de los usuarios.
  - Ordenar la ejecución de acciones.
  - Comunicar los resultados de las acciones a los usuarios.
  - Tratar ventanas, botones, diálogos, menús y listados.
- **Capa de Negocio:** es donde residen los programas que se ejecutan, se reciben las peticiones del usuario y se envían las respuestas tras el proceso. Se denomina capa de negocio (e incluso de lógica del negocio) porque es aquí donde se establecen todas las reglas que deben cumplirse. Esta capa se comunica con la capa de presentación, para recibir las solicitudes y presentar los resultados, y con la capa de datos, para solicitar al gestor de base

de datos, almacenar o recuperar datos de él. Básicamente, se encarga de satisfacer las peticiones del usuario, pero ignora dónde se guardan los datos y cómo son presentados al usuario. Se ocupa de:

  - Conocer los eventos.
  - Controlar la validez de los datos.
  - Cambiar el estado del dominio.
  - Ejecutar las acciones encomendadas.
  - Conocer las consultas.
  - Comunicar la respuesta.

- **Capa de datos:** es donde residen los datos y es la encargada de acceder a los mismos. Está formada por uno o más gestores de bases de datos (no tiene que ser una base de datos, ya que puede ser, también, un fichero XML, un fichero binario…) que realizan todo el almacenamiento de datos, reciben solicitudes de almacenamiento o recuperación de información desde la capa de negocio (almacenan y recuperan información, pero ignorar cómo tratarla). Se ocupa de:
  - Permitir a la capa de negocio ignorar donde se encuentran los datos.
  - Permitir que determinados objetos del dominio sean persistentes.

Con toda esta estructura se consigue:

- Que un cambio en la representación persistente de los datos (por ejemplo un cambio en el sistema gestor de bases de datos o ficheros), normalmente, solo afecte a la capa de gestión de datos (decimos normalmente que la estructura adecuada para alguna solución con determinados gestores de bases de datos pueden afectar a la capa de negocio, aunque es difícil encontrarse con esto).
- Que un cambio en la interfaz del programa (por ejemplo, un cambio en el sistema de ventanas o en los periféricos usados para la comunicación con el usuario) afecta nada más a la capa de presentación.
- Que la capa de negocio, que encapsula la mayoría de la lógica del programa, sea muy independiente de los cambios de plataforma, sistema operativo, etc.

Una forma de trabajar en Java podría ser la siguiente:

- Vista – Capa de presentación (Presentation Layer).
- BLL – Capa de lógica de negocio (Business Logic Layer).
- DAO – Capa de acceso a datos (Data Access Layer).
- POJO – Capa que mantiene los POJOs (Plain Old Java Object).

Todas estas capas pueden residir en un único ordenador, si bien lo más usual es que haya una multitud de ordenadores en donde reside la capa de presentación (son los clientes de la arquitectura cliente/servidor). Las capas de negocio y de datos pueden residir en el mismo ordenador, y si el crecimiento de las necesidades lo aconseja se pueden separar en dos o más ordenadores. Así, si el tamaño o complejidad de la base de datos aumenta, se puede separar en varios ordenadores, los cuales recibirán las peticiones del ordenador en que resida la capa de negocio.

Si, por el contrario, fuese la complejidad en la capa de negocio lo que obligase a la separación, esta capa de negocio podría residir en uno o más ordenadores que realizan solicitudes a una única base de datos. En sistemas muy complejos se llega a tener una serie de ordenadores sobre los cuales corre la capa de negocio, y otra serie de ordenadores sobre los cuales corre la base de datos.

# 2. PATRONES DE DISEÑO

Los **patrones de diseño** son herramientas para solucionar problemas de diseño enfocados al desarrollo de software. Estos patrones deben ser reusables permitiendo así que sean adaptados a diferentes problemáticas.

Las **principales ventajas** es que permiten tener el código bien organizado, legible y mantenible, además te permite reutilizar código y aumenta la escalabilidad en tu proyecto. En sí, proporcionan una terminología estándar y un conjunto de buenas prácticas en cuanto a la solución en problemas de desarrollo de software.

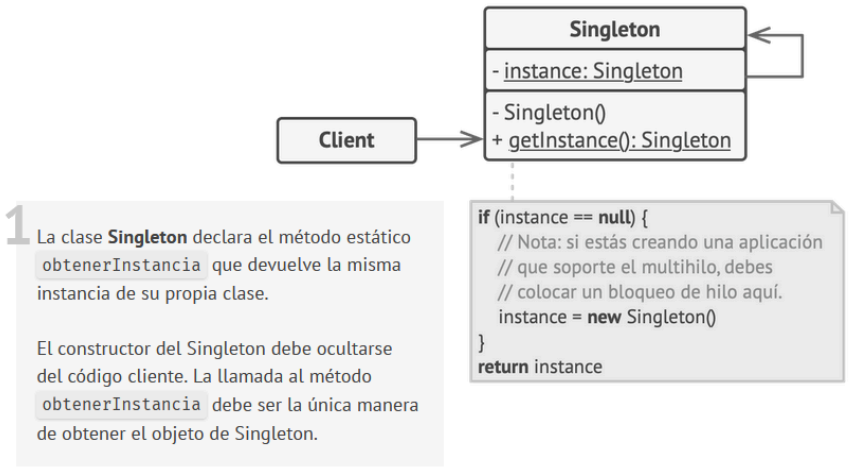
Vamos a describir los patrones más importantes a la hora de diseñar nuestras aplicaciones.

Enlaces relacionados con los patrones de diseño:

- https://devexperto.com/patrones-de-diseno-software/
- https://refactoring.guru/es/design-patterns
- https://www.ecodeup.com/patrones-de-diseno-en-java-mvc-dao-y-dto/

## 2.1. EL PATRÓN SINGLETON

Permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia para toda la aplicación.



El **Patrón Singleton** también se conoce como Instancia única, su objetivo es restringir la creación de objetos pertenecientes a una clase, de modo que solo se tenga una única instancia de la clase para toda la aplicación, garantizando así un punto de acceso global al objeto creado.

Para implementarlo, la clase Singleton debe tener un constructor privado que solo será accedido desde la misma clase. Se crea también una instancia privada de la clase, así como un método estático que permita el acceso a dicha instancia de la forma **Singleton.getInstance()**;

Este patrón es muy útil cuando necesitamos crear una clase común y global para todo el sistema, donde no nos interese crear varias instancias de la misma. Por ejemplo, podríamos pensar en una clase conexión a base de datos que utiliza toda la aplicación, si creamos varias instancias de esta podríamos llenar el sistema con muchos objetos innecesarios que se crean cada vez que instanciamos la clase, por eso restringiendo la creación a un único objeto evitamos problemas de rendimiento y trabajo para el recolector de basura de Java (*garbage collector*).

Otro ejemplo podría ser la creación de un objeto que carga los parámetros de un archivo de propiedades, de esta manera evitaríamos que el sistema lea el archivo cada vez que lo necesite, en vez de eso tan solo llamaríamos al objeto que contiene los parámetros necesarios.

Otro problema que podríamos resolver sería el siguiente: se solicita generar un historial de los eventos ejecutados por los usuarios del sistema, estos eventos se generan cada vez que alguno de los usuarios presiona un botón determinado de un panel de opciones, este historial debe contener la fecha y hora de ejecución del evento, además del usuario y opción presionada. La solución es la siguiente: como nos piden que se genere un historial de los eventos ejecutados por los usuarios utilizaremos el patrón Singleton, ya que este historial es general para todos los usuarios del sistema, así crearemos un punto global para la aplicación que permita ir almacenando cada evento generado independientemente de quien lo ejecute.

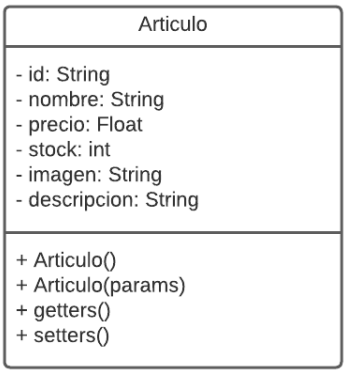
**GestionBD.java**

Vamos a crear una clase nueva llamada **GestionBD.java** para la conexión a la base de datos. Esta clase utilizará el patrón Singleton para crear un único objeto de conexión para toda la aplicación.

**Proyecto: adt4_ejemplo1**

## 2.2. EL PATRON VALUE OBJECT (POJO) / DATA TRANSFER OBJECT (DTO)

Consiste básicamente en la agrupación de datos dentro de un objeto. Estos datos representan los campos de una tabla o entidad de la base de datos y facilitan su mantenimiento y transporte dentro del sistema.




El problema aparece cuando al momento de pasar argumentos de un objeto a un método puede ocasionar que el método reciba gran cantidad de datos pasados como parámetros. Si posteriormente se requiere la modificación de uno de esos argumentos, se obliga a que todos los métodos que reciban estos argumentos sean cambiados, presentándose problemas de acoplamiento y mantenibilidad.

La solución para esto es crear clases que representan las tablas de la BD que utiliza el sistema, de esta manera las propiedades o atributos de la clase serán los campos de la entidad, permitiendo encapsular la información y facilitando la manera en que estos son transportados, así al momento de enviar los parámetros a un método, se envía un solo objeto que los contiene.

A estos objetos se les denomina **POJOs** (**Plain Old Java Object**) y la transferencia de datos como objetos **DTO (Data Transfer Object)**.

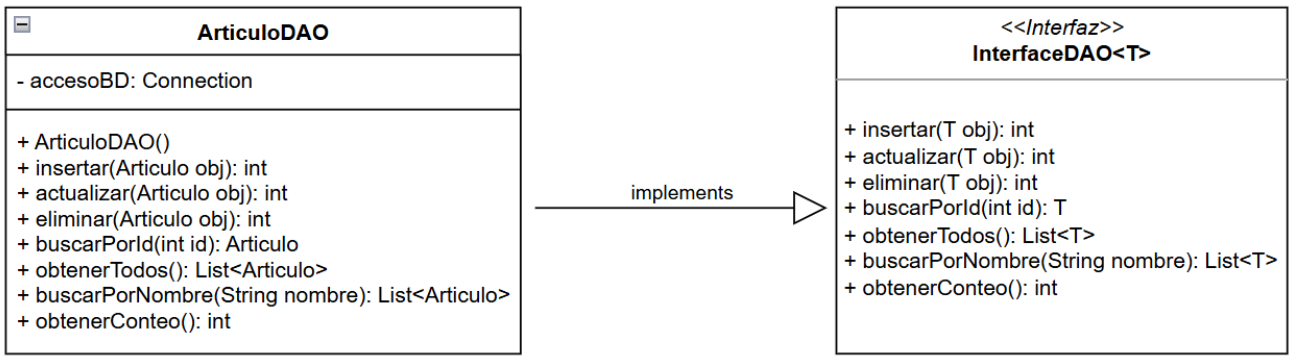
No pondremos ningún ejemplo de esto, ya que más adelante serán utilizados.

## 2.3. EL PATRÓN DATA ACCESS OBJECT (DAO)

Facilita y estructura el acceso a la información de una Base de Datos, separando la persistencia de objetos de la lógica de acceso a datos, brindando mayor flexibilidad, ya que, por ejemplo, al momento de hacer un cambio en la lógica de negocio, esto sería transparente para la lógica de acceso a la información.

El problema aparece cuando se tienen diferentes implementaciones para el acceso a datos, de esta manera al cambiar el proveedor de BD o la forma de conexión puede afectar la manera de acceder a los datos, teniendo que implementar nuevamente el proceso en todos los componentes vinculados.

La solución es utilizar **clases DAO** para encapsular todos los accesos a la fuente de datos, desacoplando de esta manera la lógica de negocio de la lógica de acceso a datos, estableciendo mayor organización y facilitando la mantenibilidad y extensibilidad del código fuente.




En este caso, tenemos una **interfaz común** (**InterfaceDAO.java**) que deben implementar todas las clases DAO que creemos en nuestro programa, así sobreescribimos siempre los mismos métodos.

- **Uso de genéricos (<T>):** Hemos parametrizado la interfaz con un tipo genérico <T>, que representa el tipo de objeto con el que trabajará cada implementación de Dao. Esto permite que cualquier clase que implemente esta interfaz especifique el tipo exacto de entidad, lo que evita el uso de Object y permite el uso de tipado estático.
- **Uso de List en lugar de ArrayList:** Es preferible usar la interfaz List porque proporciona mayor flexibilidad para que las implementaciones devuelvan cualquier tipo de lista.

No vamos a poner un ejemplo de código de este patrón, ya que se va a poner más adelante.

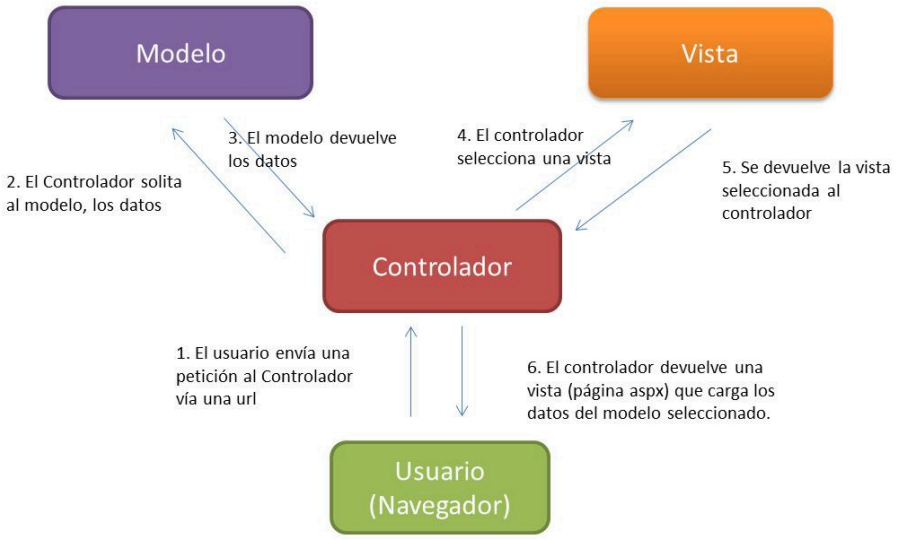
## 2.4. EL PATRÓN MVC

El **patrón MVC** es una arquitectura de diseño software para separar los componentes de aplicación en tres niveles, interfaz de usuario, lógica de control y lógica de negocio.

Es un patrón estructural que permite separar la información y lógica de negocio de la parte visual de la aplicación y la lógica que gestiona las relaciones y eventos del sistema (separar los componentes de aplicación en tres niveles, interfaz de usuario, lógica de control y lógica de negocio).

Viene definido por el siguiente problema: cuando tenemos código que no es fácilmente controlado o mantenible, debido a que tenemos muchas funcionalidades en una sola clase y principalmente cuando queremos dar una representación visual, pero al tener todo centralizado no se puede realizar con eficacia.

**Solución:** Se aplica el MVC donde reestructuramos nuestro código fuente separándolo en 3 partes funcionales con comportamientos definidos que posteriormente son relacionadas entre sí, facilitando la mantenibilidad y flexibilidad del código, estas 3 partes son: el Modelo, la Vista y el Controlador.




- **Modelo:** Representa la información y los datos procesados por el sistema, gestionando la forma como se accede a estos y la lógica de negocio de la aplicación. En la arquitectura de 3 capas corresponde a la capa de lógica de negocio y a la capa de acceso a datos.

- **Controlador:** Representa el puente de interacción entre el Modelo y la Vista, define la forma como se relacionan los componentes de la aplicación. No corresponde a ninguna capa en la arquitectura de 3 capas.
- **Vista:** Es la representación del modelo mediante una interfaz gráfica de usuario, es la forma como el usuario interactúa con el sistema, permitiendo la ejecución de eventos e ingreso de información que serán procesados por el Modelo. En la arquitectura de 3 capas corresponde a la capa de presentación.

Vamos a explicar un ejemplo práctico donde se aplica este modelo mostrando la forma de independizar los componentes de nuestro sistema, además lo combinaremos con los patrones DAO y VO (Value Object) para facilitar el manejo de la información.

El problema a presentar es el siguiente: se solicita desarrollar un sistema de administración de usuarios con un CRUD (Create, Read, Update, Delete) básico. Nos advierten que la aplicación es un prototipo inicial el cual está propenso a cambios, pues aún no se tienen definidas las ventanas con las que se va a trabajar ni la información requerida, por el momento se trabajará con una BD MySQL, pero posiblemente se tenga que migrar a otro Sistema Gestor de Base de Datos (SGBD).

Como solución a dicho problema propondremos aplicar el patrón MVC permitiendo independizar la lógica y la parte visual del sistema usando para eso un controlador que administra los procesos sirviendo como puente entre estos.

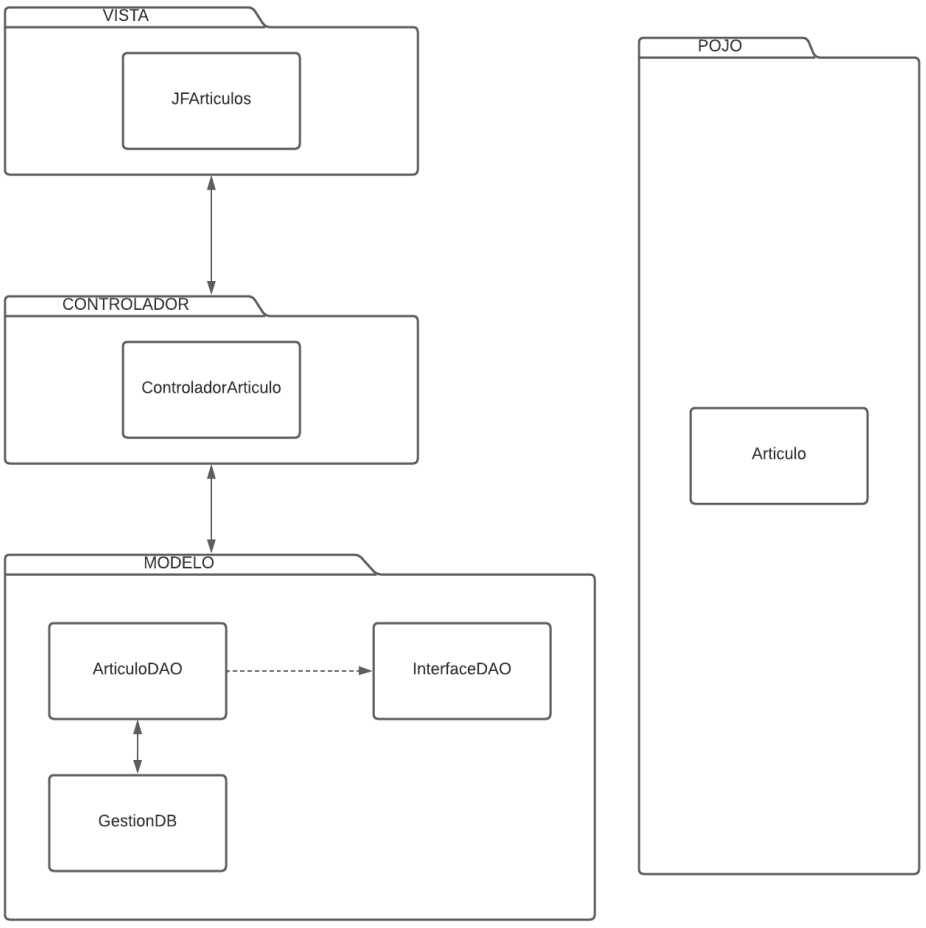
Vamos a crear una base de datos en MySQL utilizando el siguiente Script:

```
CREATE DATABASE adt4_ejemplo2;

USE adt4_ejemplo2;

DROP TABLE IF EXISTS articulos;

CREATE TABLE articulos (
       id VARCHAR(10) NOT NULL COLLATE 'latin1_spanish_ci',
       nombre VARCHAR(50) NOT NULL COLLATE 'latin1_spanish_ci',
       precio DECIMAL(7,2) NOT NULL,
       stock INT(11) NOT NULL,
       imagen VARCHAR(50) COLLATE 'latin1_spanish_ci',
       descripcion VARCHAR(50) COLLATE 'latin1_spanish_ci',
       PRIMARY KEY (id)
)COLLATE='latin1_spanish_ci'ENGINE=InnoDB;
```

```
DELETE FROM articulos;

INSERT INTO articulos (id, nombre, precio, stock, imagen, descripcion) VALUES
      ('i1', 'Reloj 1', 20.00, 07, 'img/reloj1.jpg', 'Desc. del reloj 1'),
      ('i2', 'Reloj 2', 24.00, 05, 'img/reloj2.jpg', 'Desc. del reloj 2'),
      ('i3', 'Reloj 3', 18.00, 84, 'img/reloj3.jpg', 'Desc. del reloj 3'),
      ('i4', 'Reloj 4', 30.00, 15, 'img/reloj4.jpg', 'Desc. del reloj 4'),
      ('i5', 'Reloj 5', 28.00, 07, 'img/reloj5.jpg', 'Desc. del reloj 5'),
      ('i6', 'Camiseta 1', 20.00, 76, 'img/camiseta1.jpg', 'Desc. de la camiseta 1'),
      ('i7', 'Camiseta 2', 30.00, 10, 'img/camiseta2.jpg', 'Desc. de la camiseta 2'),
      ('i8', 'Camiseta 3', 15.00, 55, 'img/camiseta3.jpg', 'Desc. de la camiseta 3'),
      ('i9', 'Camiseta 4', 18.00, 20, 'img/camiseta4.jpg', 'Desc. de la camiseta 4');
```

De esta forma se crea la Base de Datos a la que se va a apuntar, la cual contiene solo una tabla de **Artículos** con la que se realizan todas las operaciones CRUD.

La arquitectura de nuestro sistema será la siguiente:

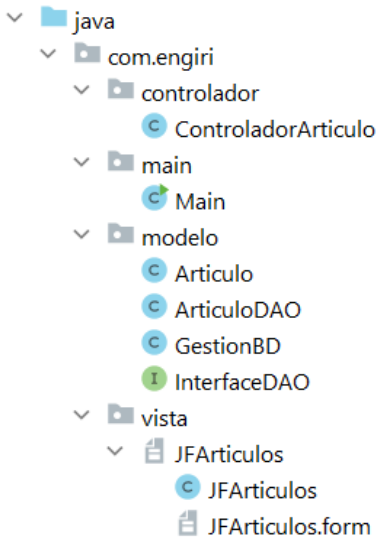


Como vemos tenemos los siguientes paquetes:

- **Vista (View):** contiene la clase *JFArticulo.java* que será la ventana principal donde interactúa el usuario final.
- **Controlador (Controller):** contiene la clase controladora, que se define en el paquete controlador, y la clase se denomina *ControladorArticulo.java*, encargada de establecer todas las relaciones del sistema.
- **Modelo (Model):** en la lógica de negocio aplicaremos los patrones DAO y VO, que será usado también en la parte visual. Se establece el modelo, el cual se compone de las siguientes clases:
    - *Articulo.java:* Contendrá la clase Artículo (POJO), que será la representación de los artículos en la base de datos. Para conseguir esto aplicaremos el patrón VO.
    - *GestiónDB.java:* tendremos un Singleton para la conexión a la base de datos.
    - *InterfaceDAO.java:* desarrollaremos una interfaz común para todos los DAO de nuestro sistema. En este caso, solamente tenemos una clase.
    - *ArticuloDAO.java:* donde realizaremos las operaciones CRUD sobre el SGBD. En nuestro caso, MySQL.

Por último, tendremos la clase Main prepara las instancias iniciales del sistema y posteriormente lo ejecuta (dicha clase se encuentra en otro paquete denominado principal).

La estructura de árbol del proyecto es la siguiente:



### 2.4.1. El modelo

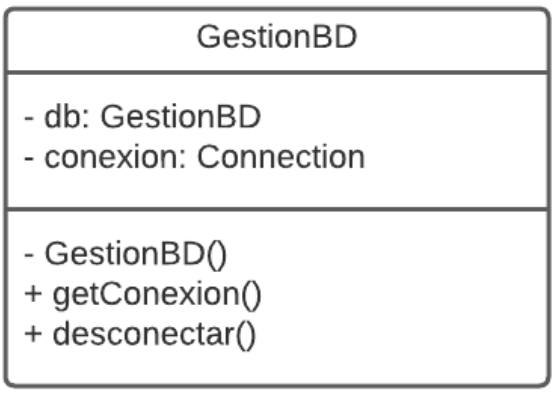
Como se mencionó, en el modelo tenemos la lógica de negocio y el acceso a datos, que serán todas las clases vinculadas con el CRUD a nivel interno.

**Clase GestionBD.java**

En esta clase tenemos la cadena de conexión con la que trabajará nuestra aplicación. En ella se define la base de datos, el usuario, password y driver de conexión, por lo que, si por ejemplo, en un futuro se nos pide conectarnos a una base de datos diferente o establecer un sistema gestor distinto (pero con la misma estructura de tablas y campos), tan solo modificaremos esta clase y dicho cambio será transparente para el resto del sistema.

Utilizaremos el patrón Singleton para implementar dicha clase, de forma que solamente sé instancia una sola vez el objeto que tiene la responsabilidad de realizar la conexión a la base de datos, optimizando de esta forma recursos.

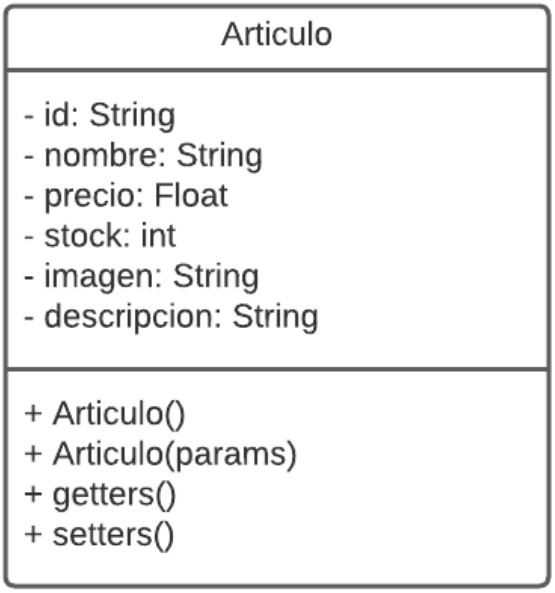
Es posible crear conexiones simples a la base de datos, pero esto hace que varios hilos no pueden usar una misma conexión física con la base de datos simultáneamente, ya que la información enviada o recibida por cada uno de los hilos se entremezcla con la de los otros, haciendo imposible una escritura o lectura coherente en dicha conexión.




Tener en cuenta que los datos de login y password corresponden a un usuario que ya están creados en el SGBD como nombre de usuario “root” y como contraseña “”.

**Clase Articulo.java**

Al utilizar este tipo de clases, aplicamos el patrón Value Object con el que representamos las entidades (Tablas) de la base de datos. La tabla artículos tiene los campos id, nombre, precio, stock, imagen y descripción, por lo que nuestra clase VO tendrá estos mismos atributos y de esta manera podremos transportar a un objeto Artículo con todos estos valores por medio de los métodos setters y getters de cada atributo.

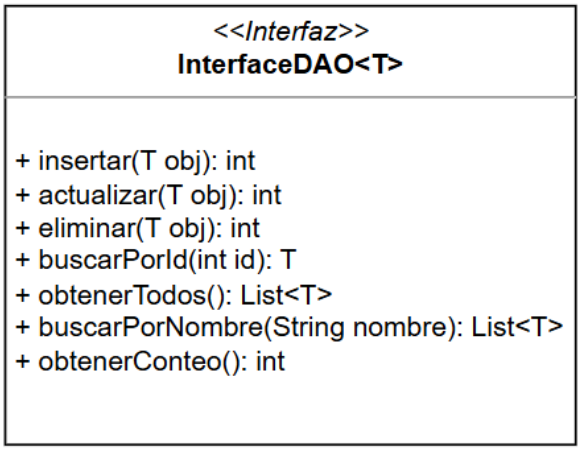



El patrón VO, junto con el patrón Data Transfer Object (DTO) nos facilita enormemente el transporte de la información, evitando que se envíen gran cantidad de parámetros a un método cuando queremos hacer un registro o actualización. También en caso de que se modifique la tabla de la BD, nuestra clase deberá ser modificada, así como los métodos que obtienen la información, más no los métodos que la transportan.

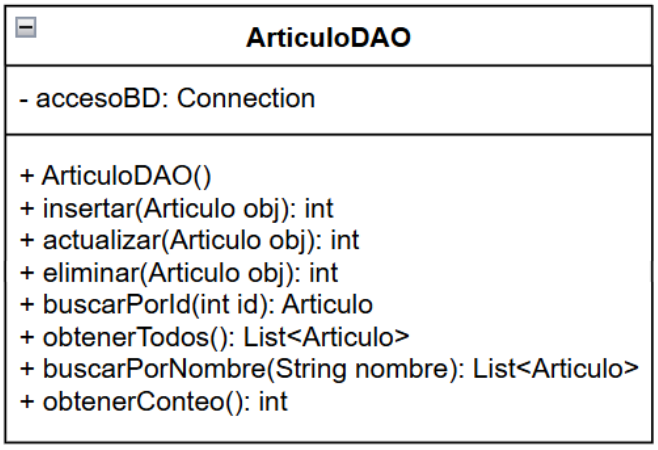
**Clase ArticuloDAO.java**

Cuando utilizamos estas clases, aplicamos el patrón Data Access Object o DAO. Básicamente, este patrón consiste en centralizar los procesos de acceso a la base de datos, evitando inconsistencias y posibles problemáticas cuando esto se realiza a lo largo de la aplicación.

Para que todas las clases sigan el mismo patrón de comportamiento creamos una interfaz DAO con la siguiente estructura:




Nuestras clases DAO implementarán la interfaz, obligando a implementar los métodos anteriores:




Con este patrón independizamos la lógica de negocio de la lógica de acceso a datos, obteniendo mayor organización y flexibilidad en el sistema.

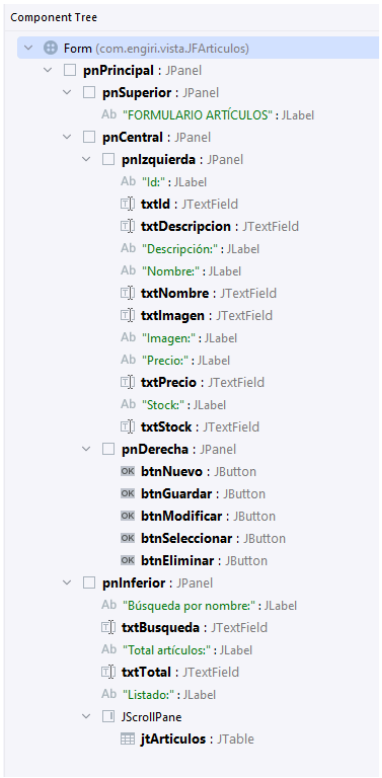
### 2.4.2. La vista

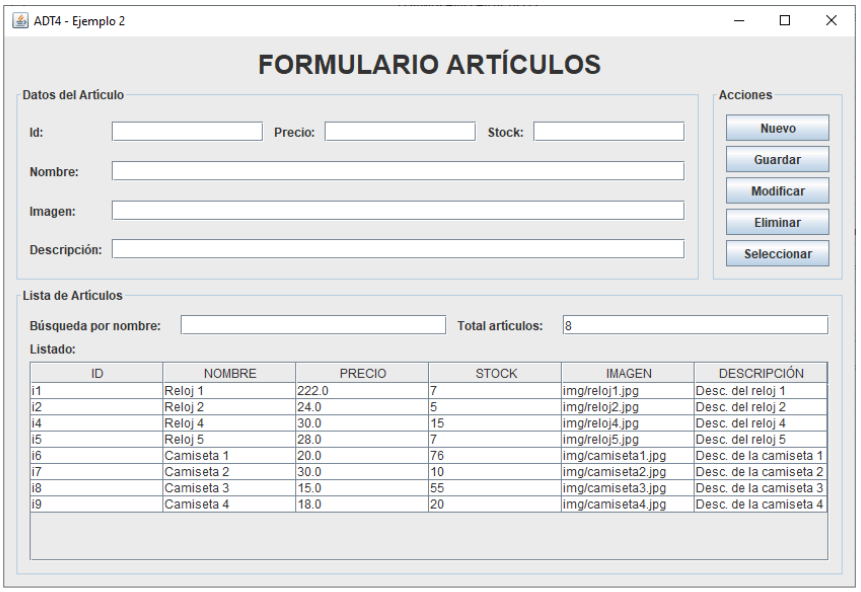
Aquí se define la parte visual del sistema, en la vista estableceremos todas las ventanas o interfaces gráficas de usuario, mediante las cuales representamos todo el modelo, permitiendo la interacción entre la aplicación y el cliente.

**Clase JFArticulos.java**

Esta clase representa la ventana principal. Posee tres paneles diferentes dentro del panel principal:

1. Panel superior: título del formulario.
2. Panel Central:
    a. Datos del artículo: para introducir todos los atributos de un artículo.
    b. Acciones: diferentes botones para realizar las operaciones CRUD.
3. Panel Inferior:
    a. Lista de artículos: muestra la información almacenada en la base de datos.




Se comunica con el modelo mediante la clase controladora y desde esta se cargan las otras ventanas de la aplicación.

### 2.4.3. El controlador

Esta parte del patrón es la que define la lógica de administración del sistema, establece la conexión entre la vista y el modelo. Puede contener instancias locales tanto de clases de la vista como de clases del modelo. Estas instancias tienen sus respectivos métodos setters y getters permitiendo el flujo de llamadas del sistema.

Esta clase controladora será la encargada de interpretar los eventos del usuario y comunicarse con su capa de modelo para realizar las operaciones pertinentes. El modelo le proporcionará la información deseada y el controlador la mostrará por medio de la capa de la vista.

### 2.4.4. Clase principal

Esta clase contiene el método main que ejecuta la aplicación, el método hace un llamado al método iniciar, el cual crea las instancias de las clases JFArticulos (vista) y la clase ArticuloDAO (modelo), estableciendo las relaciones con la clase ControladorArticulo (controlador).

```
public class App {
   public static void main(String[] args) {

       JFArticulos vista = new JFArticulos();
       ArticuloDAO modelo = new ArticuloDAO();

       ControladorArticulo ca = new ControladorArticulo(vista, modelo);

       // Centrar en el centro de la pantalla
       vista.setLocationRelativeTo(null);
       vista.setVisible(true);

   }
}
```

Cada instancia de las clases se las envía a una instancia de la clase Controlador, estableciendo así las relaciones necesarias.

## 2.5. EL PATRÓN INYECCIÓN DE DEPENDENCIAS (DI)

La **inyección de dependencias (DI, Dependency Injection)** es un patrón de diseño y una técnica utilizada en la programación orientada a objetos para gestionar las dependencias entre clases (componentes) de una aplicación. Es decir, cuando una clase necesita otra clase (dependencia), en lugar de crear el objeto dentro de la misma clase, lo debemos suministrar o inyectar. Esto sigue el principio de Hollywood (no nos llames, ya te llamaremos).

La inyección de dependencias se utiliza para lograr varios objetivos:

1. **Desacoplar componentes:** Permite que las clases de una aplicación estén menos acoplados entre sí. Esto facilita la reutilización de componentes, la prueba unitaria y la gestión de cambios en la aplicación.
2. **Facilitar el mantenimiento:** Al reducir el acoplamiento, los cambios en un componente pueden tener un impacto menor en otros componentes.
3. **Mejorar la claridad y la legibilidad:** Hace que el flujo de dependencias entre los componentes sea más evidente y claro.
4. **Facilitar la prueba unitaria:** Permite la sustitución de las dependencias reales por versiones simuladas o de prueba durante las pruebas unitarias.
5. **Promover la inversión de control (IoC):** La inyección de dependencias es un aspecto fundamental de la inversión de control. En lugar de que un componente controle la creación o resolución de sus dependencias, delega esta responsabilidad a un contenedor de IoC.

A continuación, se explica cómo funciona la inyección de dependencias en términos generales:

1. **Identificación de dependencias:** En primer lugar, se identifican las dependencias que un componente necesita para funcionar correctamente. Por ejemplo, un controlador puede depender de un modelo y una vista.
2. **Inyección de dependencias:** Las dependencias se proporcionan al componente desde el exterior. Esto puede hacerse de varias maneras, como pasando objetos como argumentos a un constructor, llamando a métodos de configuración, o configurando propiedades.
3. **Utilización de dependencias:** El componente utiliza las dependencias proporcionadas para realizar sus funciones.

El uso de la inyección de dependencias es especialmente útil en el contexto de frameworks de desarrollo y contenedores de inversión de control (IoC containers), como Spring en Java, que automatizan el proceso de inyección de dependencias y gestionan las dependencias de los componentes de la aplicación.

En resumen, la inyección de dependencias es una técnica que promueve un diseño más modular, flexible y mantenible en aplicaciones orientadas a objetos al permitir la gestión de dependencias de componentes de manera más eficiente y desacoplarlos de sus dependencias directas.

### 2.5.1. Ejemplo

En el controlador de nuestro ejemplo tenemos lo siguiente con inyección de dependencias (DI):

```
public class ControladorArticulo {

   private JFArticulos vista;
   private ArticuloDAO modelo;

   public ControladorArticulo(JFArticulos vistaArticulo, ArticuloDAO modeloArticulo) {
       this.vista = vistaArticulo;
       this.modelo = modeloArticulo;
       ...

   }
   ...
}
```

Sin inyección de dependencias sería:

```
public class ControladorArticulo {

   private JFArticulos vista;
   private ArticuloDAO modelo;

   public ControladorArticulo() {
       this.vista = new JFArticulos();
       this.modelo = new ArticuloDAO();
       ...

   }
   ...
}
```

Ambas opciones tienen sus ventajas y desventajas, y la elección depende de la arquitectura y el flujo de la aplicación que estás diseñando.

**Opción 1: Inyección de dependencias (DI):**

Ventajas:

- Mayor flexibilidad y reutilización de componentes: Puedes usar el mismo modelo o vista en diferentes controladores, lo que es útil si tienes múltiples controladores en tu aplicación.
- Facilita las pruebas unitarias: Puedes crear versiones simuladas o de prueba de los modelos y vistas y pasarlos al controlador durante las pruebas.

Desventajas:

- Puede requerir más configuración inicial.
- Puede aumentar la complejidad de la inyección de dependencias si la aplicación es grande.

**Opción 2: Creación interna de modelos y vistas en el controlador, sin inyección de dependencias:**

Ventajas:

- Más sencillez en la configuración inicial.
- Puede ser apropiado si la relación entre el controlador, el modelo y la vista es fuertemente acoplada y específica para esa parte de la aplicación.

Desventajas:

- Menos flexibilidad y reutilización de componentes.
- Puede ser difícil realizar pruebas unitarias con modelos y vistas reales en lugar de versiones simuladas.

En última instancia, la elección depende de la estructura y los requisitos de tu aplicación. Si estás construyendo una aplicación grande y modular, la inyección de dependencias puede ser una opción más adecuada. Si tu aplicación es más pequeña y la relación entre el controlador, el modelo y la vista es simple y específica, la creación interna en el controlador puede ser más conveniente.

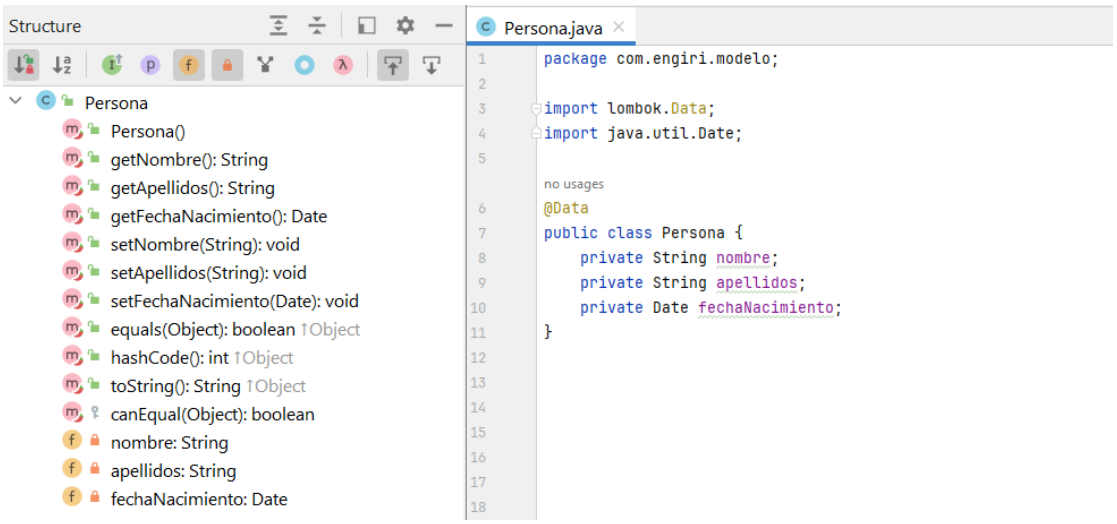
# 3. LIBRERÍA: LOMBOK

**Lombok** es una solución que nos permite evitar tener que escribir una y otra vez ese código repetitivo, que tantas veces tenemos que implementar en nuestras clases POJOS.

Como hemos mencionado, en nuestras clases Java hay mucho código que se repite una y otra vez: constructores, setters y getters. Métodos que quedan definidos una vez que dicha clase ha concretado sus propiedades, y que salvo ajustes menores, serán siempre sota, caballo y rey. Con Lombok se hace eso mismo: definimos las propiedades y nos despreocupamos. Con unas pocas anotaciones cubrimos casi todos los casos que se pueden dar.

## 3.1. Ejemplo

Primero veamos qué puede hacer Lombok por nosotros y luego veamos cómo conseguirlo.



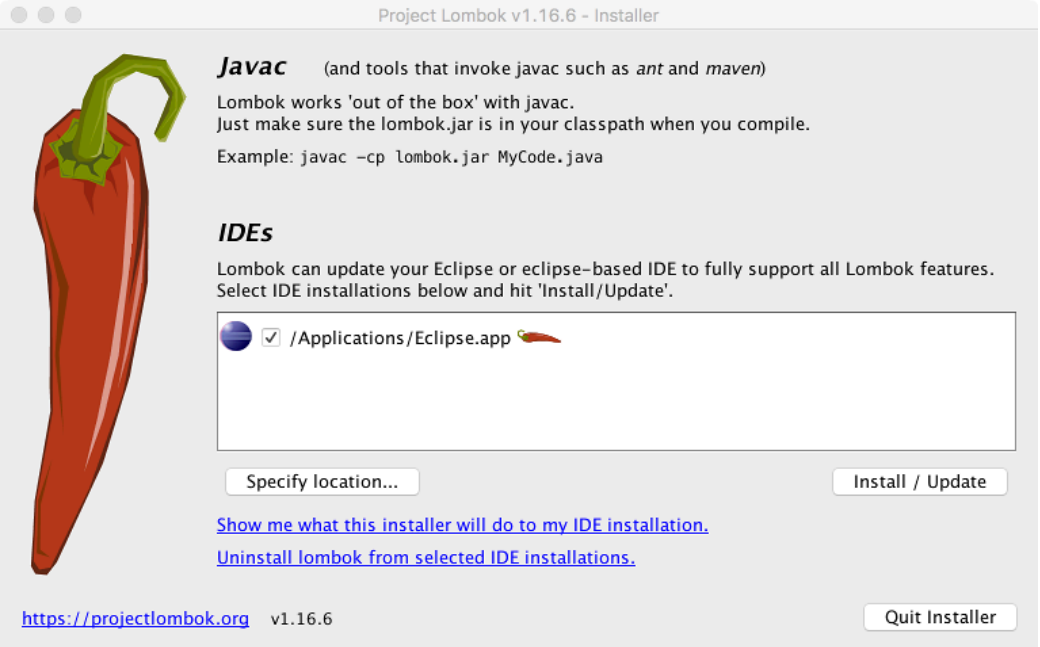

En la imagen anterior se observa como apenas hemos declarado la clase y unas pocas propiedades. Sin embargo, en la ventana Structure de IntelliJ se ven todos los getters, setters, el constructor y algún otro método como equals(), canEqual(), hashCode() y toString(). Y lo único que se ha hecho es anotar la clase con **@Data** y añadir su correspondiente **import**. Está claro que para que esto ocurra, nuestro IDE debe reconocer lombok, y, por otro lado, nuestro proyecto debe importar la librería.

## 3.2. Instalando Lombok

Desde la web del proyecto Lombok se puede descargar un jar, que debería poder instalarse con un simple doble clic. En el caso de que esto no funcionara, habría que recurrir a la línea de comandos y ejecutar:

```
java -jar lombok.jar
```

El instalador intentará descubrir los IDEs que tienes instalados y preguntará en cuáles quieres instalar Lombok. En caso de que no encuentre ninguno de los que soporta, se puede indicar dónde está instalado el IDE en el que queremos añadir lombok. **La instalación en el IDE no es obligatoria**.




Tendremos que añadir obligatoriamente el JAR al proyecto donde lo queramos usar. Lo podemos añadir como dependencias de maven en el pom.xml

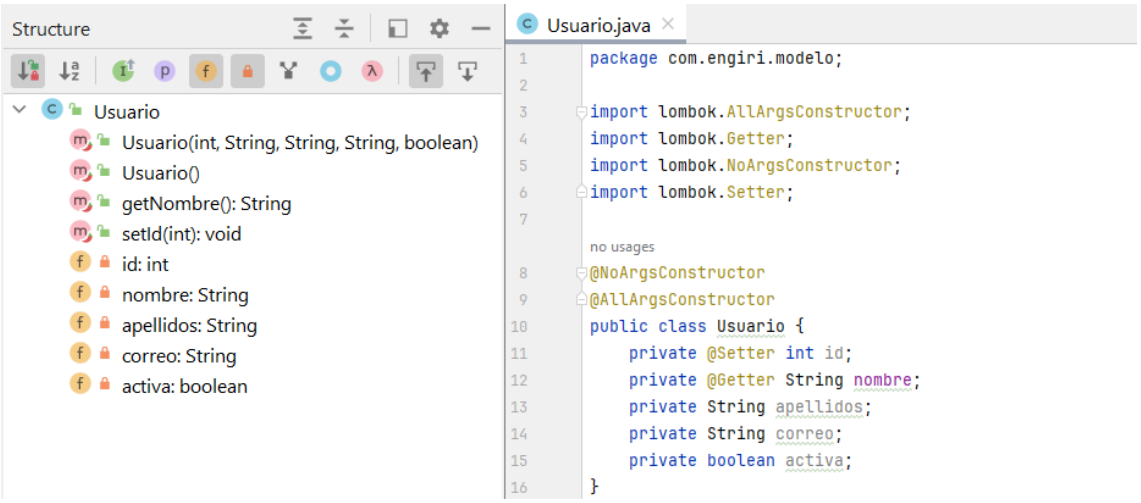
```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.34</version>
    <scope>provided</scope>
</dependency>
```

## 3.3. Algunas características de Lombok

Ya hemos visto lo que puede hacer la anotación @Data por nosotros. Pero hay muchas otras. Algunas de las más reseñables son:

- @Getter
- @Setter
- @NoArgsConstructor
- @AllArgsConstructor
- @Builder
- @Log

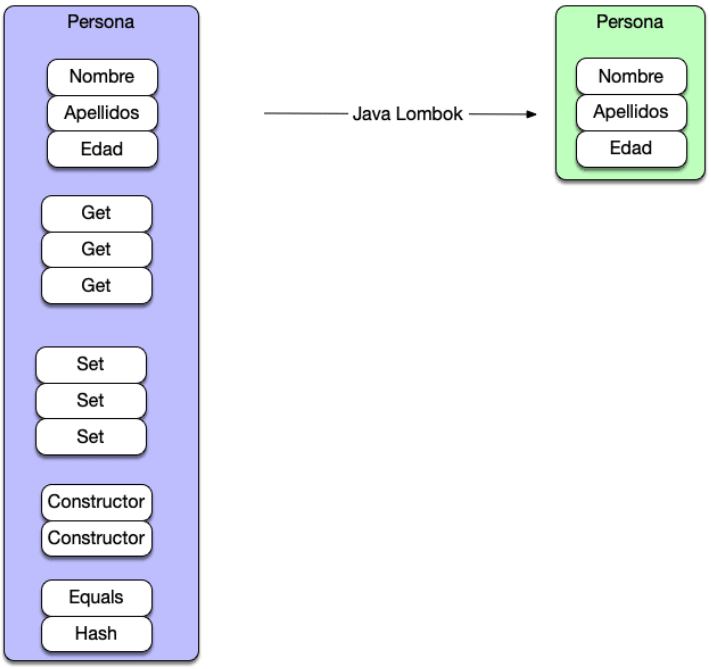
La mayoría son autoexplicativas y no merece la pena profundizar en ellas. Basta con un par de experimentaciones rápidas para darse cuenta de que lo que hacen es lo que dice la anotación. En este sentido, Lombok es muy intuitivo.




En el ejemplo anterior, hemos anotado la clase con @NoArgsConstructor y @AllArgsConstructor, y comprobamos que el resultado en el Navigator, es que tenemos dos constructores: uno sin argumentos y otro con todos los argumentos.

A nivel de propiedades, hemos anotado una con @Setter y otra con @Getter, y automáticamente nos genera dichos métodos para esas propiedades. Estas anotaciones se pueden usar a nivel de clase, y generaría esos métodos para todas las propiedades. Muy fácil.

El resultado es que de forma sencilla y efectiva podemos tener control sobre nuestros POJOs con unas simples anotaciones, permitiéndonos aislarnos de la «morralla» habitual vinculada al tratamiento de este tipo de clases, que en el caso de modelos de datos con numerosos atributos, pueden ser muy tediosos de gestionar, aunque los IDEs actuales nos ayuden a ello.




## 3.4. Referencias

- https://projectlombok.org/
- https://www.adictosaltrabajo.com/2016/02/03/como-reducir-el-codigo-repetitivo-con-lombok/
- https://www.arquitecturajava.com/java-lombok-clases-y-productividad/
- https://openwebinars.net/blog/que-es-lombok/