



# **PROGRAMACIÓN ORIENTADA A OBJETOS CON ACCESO A BBDD IMPLEMENTACIÓN DE CLASES EN JAVA**

## **MIEMBROS**

Katiane Coutinho Rosa  
Nadia Igual Bravo  
Xavier Melo Pardo  
Daniel Boj

Grupo - Cirice



|  |          |
|--|----------|
| <b>ENLACE GITHUB</b>   | <b>2</b> |
| <b>CREACIÓN DE LA ESTRUCTURA DE DIRECTORIOS DEL PROYECTO</b> | <b>3</b> |
| <b>CLASES GENÉRICAS</b>                                      | <b>4</b> |
| Tipos de colecciones   | 5        |
| <b>MANEJO DE EXCEPCIONES</b>                                 | <b>5</b> |
| <b>IMPLEMENTACIÓN DEL MAIN</b>                               | <b>6</b> |
| <b>CONTROLADOR DE VISTA</b>                                  | <b>7</b> |
| <b>OTROS AÑADIDOS</b>  | <b>7</b> |
| <b>TESTS UNITARIO CON JUNIT</b>                              | <b>7</b> |
| <b>RECURSOS</b>  | <b>8</b> |

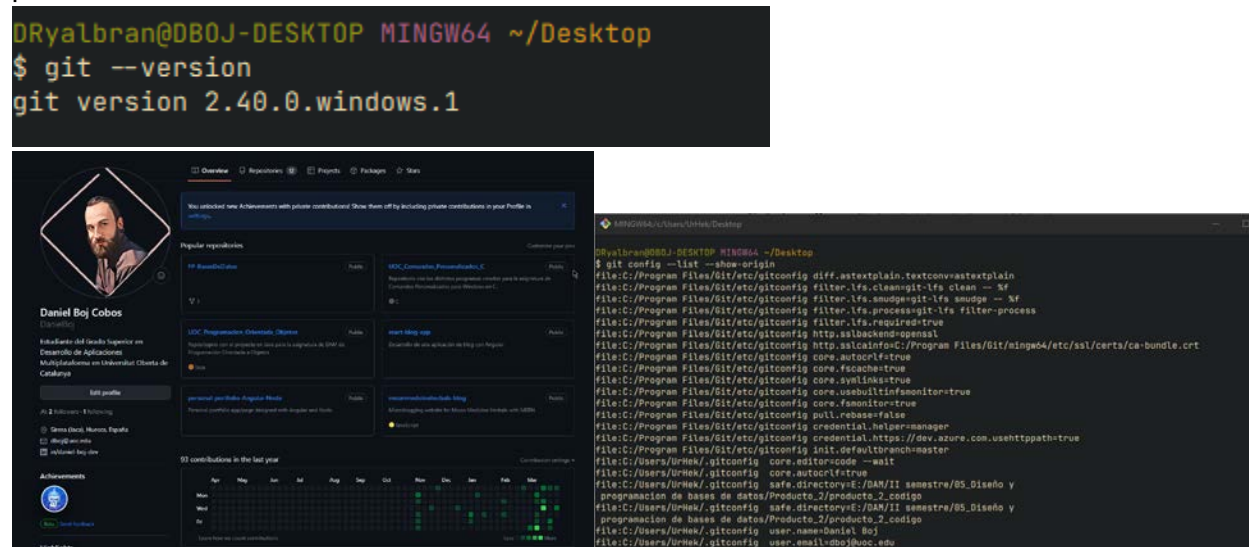
# Producto 2. Implementación de clases en Java

## Enlace GitHub

Para realizar el seguimiento de las tareas que vamos realizando, tanto a nivel individual como grupal, compartimos el enlace a nuestro repositorio de GitHub:

<https://github.com/DanielBoj/UOC-POO-con-acceso-a-BBDD>

Como es requisito de entrega de este producto, explicamos brevemente cómo está organizado el repositorio. Tras la instalación de Git en los equipos y su configuración a través de terminal, hemos creado el repositorio en el GitHub de Daniel ya que ya lo tenía creado y configurado previamente de semestres anteriores:

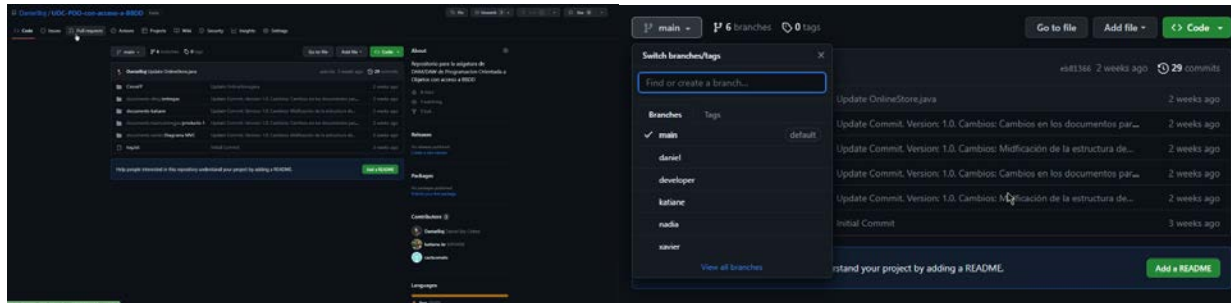


The image shows a terminal window on the left and a GitHub profile page on the right. The terminal window displays the command `$ git --version` and the output `git version 2.40.0.windows.1`. Below this, the command `$ git config --list --show-origin` is executed, showing a list of Git configuration variables and their origins. The GitHub profile page on the right shows the profile of Daniel Boj Cobos, including his bio, repositories, and a contribution graph.

Se ha creado un repositorio identificado con la signatura: **UOC-POO-con-acceso-a-BBDD**

El repositorio es público y consta de 6 ramas:

- Rama *main* → La rama principal del proyecto, en ella realizaremos los merge del proyecto común tras su revisión.
- Rama *developer* → En esta rama realizamos el primer merge del proyecto en común para realizar su revisión antes de fusionarlo a la rama principal.
- Ramas personales → Hay una rama personal para cada uno de los miembros del equipo donde podrá ir subiendo todos los avances en el desarrollo individual del proyecto y desde donde el resto de los miembros podrán ir comparando y comprobando el código.



Para crear el repositorio, realizamos un primer *push* del proyecto para la rama *main*, el miembro encargado creo a posteriori su rama personal y la rama *developer* y el resto de miembros realizaron una clonación del proyecto y crearon sus ramas personales. En cuanto a la gestión del control de versiones, algunos miembros lo estamos realizando mediante *GitBash* directamente con comando, otros mediante *GitKraken* y también estamos experimentando con las opciones de control de versiones que ofrece directamente el IDE de desarrollo IntelliJ.

## Creación de la estructura de directorios del proyecto

El primer paso para la creación del proyecto ha sido generar la estructura de directorios de este, para ello, como comentamos en el primer producto, hemos seguido el patrón de diseño MVC; además, hemos aplicado los principios y buenas prácticas del diseño modular y la creación de un directorio separado para crear los tests unitarios con Junit y aislar nuestro código fuente de nuestro código de pruebas. Igualmente, se ha seguido el modelo de nomenclatura de java para los distintos packages (dominio.módulo), en este caso, el nombre del grupo seguido del nombre del módulo/package.

El directorio *test* se ha generado como un directorio de pruebas, IntelliJ lo muestra en color verde, tiene una carpeta preparada para almacenar recursos para los tests si fueran necesarios, se muestra en color rojo, y los paquetes internos siguen el modelo de nomenclatura comentado y que hace referencia a los módulos con los cuales se relacionen los tests en el directorio fuente, *src*.

Además, dentro del directorio *out*, se ha creado la carpeta *artifacts* donde se generan los ejecutables JAR para nuestro proyecto. También, para seguir aplicando unos buenos principios, se han separado en *packages* aislados las interfaces y el modelo de clase genérica. La idea es mantener estas prácticas a lo largo de todo el desarrollo y aplicarlas también cuando generemos los modelos y controladores para la capa de datos que deba interactuar con la BBDD.



## Clases genéricas

Como comentamos en el anterior producto y en el correo que te mandamos, hemos solucionado el problema de las clases genéricas creando una única **clase genérica**, `Listas<T>`, para manejar toda la implementación de las listas y luego especificando en el controlador el tipo de dato que usaríamos para cada lista, por ejemplo:

`private Listas<Cliente> clientes`. Esto cambia algo las opciones recomendadas por el enunciado, pero pensamos que es una manera más avanzada de realizarlo. Para crear la clase genérica, hemos de realizar una generalización de la clase `ArrayList` mediante el operador diamante:

```
public class Listas<T> implements Iterable<T>
```

La clase genérica prototipa todos los métodos que ejecutan las acciones CRUD y que necesitaremos usar todas las listas que usen esta clase. Hemos de recordar usar el valor genérico `T` siempre que un objeto sea genérico.

En la implementación de estos métodos hemos intentado tener en cuenta:

- Búsqueda de uso de funciones puras.
- Intentar mantener la lista original intacta e inalterada si no es necesario actualizarla.
- Usar el polimorfismo e implementar las interfaces nativas de ArrayList que podemos necesitar en nuestros métodos.

## Tipos de colecciones

Como tipo de colección hemos escogido los ArrayList ya que no hay ningún caso particular que necesite de otro tipo de estructura de datos más compleja: No necesitamos diccionarios, así que no es necesario implementar tables de Hash, tampoco tenemos que aplicar patros FIFO o LIFO, ni se requieren trabajar con grafos o árboles ya que no hemos de recorrer nodos mediante jerarquías especiales.

De este modo, los ArrayList nos ofrecen las ventajas de acceso mediante índices de los Array, unidas a las de las Listas: Mejor BigO de escritura, tamaño dinámico y métodos de clase para lectura y escritura.

## Manejo de excepciones

Todos los métodos que hemos implementado gestionan el manejo de excepciones mediante los bloques Try... Catch. Siempre que lo hemos tenido claro, hemos usado un tipo de excepción concreto, si ha sido necesario, implementando más de un tipo mediante un operador or (|). Cuando no hemos tenido claro qué excepción concreta usar, hemos optado por llamar al tipo de objeto más general, Exception.

Por otro lado, hemos manejado internamente los casos en los que el retorno de un método puede generar un error, básicamente, los retornos que pueden devolver un *null* y, de igual modo, hemos controlado los parámetros para que no se pasen argumentos *null*. Además, muchos métodos implementan controles internos de valores de variables y argumentos para evitar errores.

El manejo de excepciones permite que el programa no se detenga de forma abrupta, lo que hemos hecho ha sido capturar el mensaje de la excepción y lanzarlo por consola para poder realizar un seguimiento si se produce una excepción y aplicar el código necesario para que el programa pueda seguir ejecutándose sin que se produzcan errores.

Para evitar recibir argumentos *null* usamos la librería **org.jetbrains.annotations**.

### Ejemplo:

Controlamos que no se pasen argumentos *null* y manejamos la posible excepción en la ejecución. Si el método falla, retorna un objeto *null*, este escenario lo manejamos en el método que realiza la llamada mediante un bloque condicional.

```
8 usages Daniel Boj
public Artículo createArtículo(@NotNull Artículo artículo) {
    // Intentamos añadir el artículo a la lista
    try {
        return this.articulos.add(artículo)? artículo : null;
    } catch (NullPointerException e) {
        System.out.println(e.getMessage());
        return null;
    }
}
```

A continuación, vemos un ejemplo de bloque condicional que maneja posibles valores *null*.

```
if (artículoSrc != null) {
    // Comparamos los dos objetos
    if (artículo.compareTo(artículoSrc) == 0) {
        System.out.println(MessageFormat.format( pattern: "El artículo {0} ya existe", artículo.getDescripcion()));
        return null;
    } else {
        // Manejamos la colisión de códigos
        artículo.setCodArtículo(artículo.generateCodigo( key: artículo.getDescripcion() + "$!*"));
    }
}
```

En muchos casos, cuando nos enfrentamos al diseño de un bloque condicional, si pensamos primero qué hacer en caso de la negación de la condición a analizar, nos podremos ahorrar líneas de código.

## Implementación del main

Nuestra implementación de la clase *main* es algo distinta a la sugerida en el enunciado de la actividad y sigue el patrón MVC que diseñamos en el producto 1.

Para empezar, la clase se encuentra en el package *controlador* y tiene como propiedades una instancia de cada uno de los controladores del programa; el controlador principal y los dos controladores internos. Estos controladores se instancian en el método *main* y se asignan a la clase *main*, *OnlineStore*, para realizar todo el control de flujo de ejecución a través de una instancia de la clase *main* que llamamos *prg* (programa). Además, para mantener la asociación bidireccional entre el Controlador y cada uno de los módulos principales, Vista y Modelo, estos reciben como argumento la instancia del Controlador principal que hemos creado. Tras la instanciación, el método *main* llama al método *init* donde podemos seguir implementando la lógica de flujo de ejecución de nuestro programa.

Para este producto, *init* realiza una carga previa de datos, inicializa la vista principal y retorna un valor de retorno para la finalización del programa, en este caso, si se produce una excepción en la ejecución principal, retornará 1 y si el programa se ejecuta con éxito, 0.

## Controlador de Vista

Siguiendo nuestro modelo de clases desarrollado para el producto 1, nuestro contralador interno para el *package* Vista se llama MenuPrincipal y no GestionOs. El controlador maneja también el menú contextual principal y se encarga de llamar a la vista que se relacione con la opción escogida. Además, implementa toda la conexión con el Controlador principal de la App.

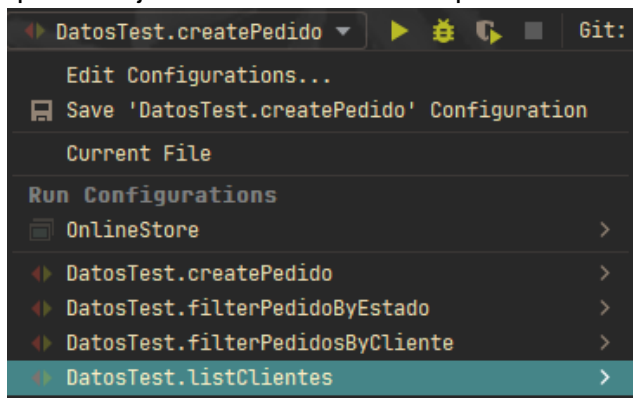
## Otros añadidos

Hemos creado interfaces para controlar mejor los métodos que deben implementar los modelos y los métodos generales para las vistas. Además, hemos creado un interfaz HashCode que implementan las clases que deben generar códigos únicos. Por otro lado, hemos implementado el interfaz de Java Comparable en todos los casos que hemos creído necesarios.

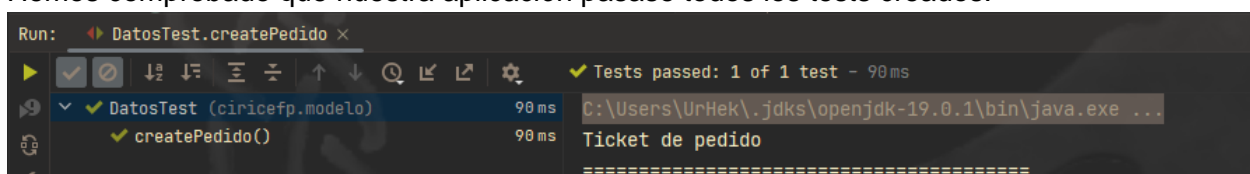
## Tests Unitario con JUnit

Como comentábamos en la sección sobre la creación de los directorios, hemos creado una directorio independiente para diferenciar la implementación de los tests y la de la app y permitir su ejecución de forma aislada.

Hemos credo los distintos tests de prueba sobre la clase Data ya que es el controlador interno que maneja todos los métodos a aplicar sobre los modelos de la aplicación.



Hemos comprobado que nuestra aplicación pasase todos los tests creados.





## Recursos

*Run tests* / *IntelliJ IDEA*. (s. f.). IntelliJ IDEA Help.

<https://www.jetbrains.com/help/idea/2023.1/performing-tests.html>

Programando en JAVA. (2022, 9 junio). *PRUEBAS UNITARIAS en JAVA (JUnit 5) - Tutorial*

*Completo Fácil*. YouTube. <https://www.youtube.com/watch?v=74sCIDEYSQ4>

GeeksforGeeks. (2019, 22 mayo). *Iterable forEach method in Java with Examples*.

<https://www.geeksforgeeks.org/iterable-foreach-method-in-java-with-examples/>

Shakmuria. (2018, 24 febrero). *151.- Interface Comparable en Java (Ordenar edades de objetos*

*de tipo Empleado*). YouTube. <https://www.youtube.com/watch?v=vNC3-vG2-xk>

*Java - Interfaces*. (s. f.). [https://www.tutorialspoint.com/java/java\\_interfaces.htm](https://www.tutorialspoint.com/java/java_interfaces.htm)

Coding with John. (2022, 18 enero). *Java Unit Testing with JUnit - Tutorial - How to Create And*

*Use Unit Tests*. YouTube. <https://www.youtube.com/watch?v=vZm0lHciFsQ>

HolaMundo. (2022, 18 febrero). *Aprende GIT ahora! curso completo GRATIS desde cero*.

YouTube. <https://www.youtube.com/watch?v=VdGzPZ31ts8>