

17-11-2025

## TRABAJO PRACTICO INTEGRADOR

BASES DE DATOS

### Alumnos:

- Marcos Farid
- Francisco
- Hugo Tach
- Ezequiel Alejandro Ventura

**Coordinador:** Oscar Londero

### Link del video:

<https://www.youtube.com/watch?v=mmo64bDnNtE>

# 1 Introducción

El presente trabajo práctico integrador tiene como objetivo aplicar los conocimientos adquiridos a lo largo de la materia **Programación II**, desarrollando un **sistema de gestión de pedidos y envíos** que simula el funcionamiento de una pequeña empresa de distribución.

El proyecto fue implementado en **Java**, utilizando una **arquitectura por capas** que separa las responsabilidades en distintos módulos:

- **Modelo (Model):** contiene las clases que representan las entidades del dominio, como **Pedido**, **Envio**, **Cliente**, y los distintos tipos y estados asociados.
- **Acceso a Datos (DAO):** gestiona la conexión con la base de datos, implementando el patrón **Data Access Object (DAO)** para realizar operaciones CRUD (crear, leer, actualizar y eliminar) de manera desacoplada y reutilizable.
- **Servicios (Service):** encapsula la lógica de negocio, interactuando con la capa DAO para procesar las operaciones solicitadas por el usuario o la interfaz.
- **Aplicación principal (Main):** contiene los menús y puntos de entrada del programa, facilitando la interacción con el sistema a través de consola.

Cada integrante del grupo fue responsable de una capa específica, trabajando de forma colaborativa mediante **GitHub**, utilizando técnicas de **fork**, **branch** y **pull request** para integrar el código de manera ordenada y controlada.

El sistema permite **registrar pedidos, asociarlos con envíos, consultar registros existentes, realizar modificaciones y aplicar bajas lógicas**, garantizando la persistencia y coherencia de los datos.

A través de este proyecto, se logró integrar conceptos fundamentales de la programación orientada a objetos, manejo de excepciones, conexión a bases de datos mediante JDBC y control de versiones con Git.

En síntesis, el trabajo refleja la aplicación práctica de los principios de **modularidad, reutilización y buenas prácticas de desarrollo**, consolidando las competencias necesarias para el desarrollo de software profesional en Java.

## 2 Herramientas y Metodología

Se emplearon diversas **herramientas de software** y **metodologías de trabajo colaborativo**, con el fin de garantizar una organización eficiente, un control adecuado de versiones y la aplicación de buenas prácticas de programación.

### Lenguaje y Entorno de Desarrollo

- **Lenguaje:** Java (versión 17)  
Se utilizó Java por ser un lenguaje orientado a objetos robusto, multiplataforma y

ampliamente utilizado en entornos empresariales.

- **Entorno de desarrollo (IDE):** NetBeans  
Este entorno permitió compilar, ejecutar y depurar el código de forma sencilla, facilitando la gestión del proyecto a través de su estructura modular.
- **Base de datos:** MySQL  
Se utilizó como sistema de gestión de base de datos relacional (RDBMS) para almacenar la información persistente de pedidos, envíos y clientes.
- **Conexión a la base de datos:** JDBC (Java Database Connectivity)  
Se implementó la conexión directa entre Java y MySQL mediante `Connection`, `PreparedStatement` y `ResultSet`, respetando las buenas prácticas del manejo de recursos y excepciones.

## Control de Versiones y Trabajo Colaborativo

- **Git y GitHub:**  
El equipo utilizó **GitHub** como plataforma principal para la gestión del código fuente y el control de versiones.  
Cada integrante realizó un **fork** del repositorio principal, trabajó en su propia **rama (branch)**, y luego integró sus cambios mediante **pull requests**, asegurando una correcta trazabilidad y revisión de código antes del merge final.

## 3 Diseño y Arquitectura

El sistema fue diseñado siguiendo una **arquitectura multicapa (por capas)**, con el propósito de lograr una estructura modular, escalable y de fácil mantenimiento.

Esta arquitectura permite separar claramente las responsabilidades de cada componente, facilitando la lectura, el testeo y la futura expansión del sistema.

### 3.1. Elección del Dominio

Se ha seleccionado el dominio Pedido - Envío (A - B), de acuerdo con las opciones provistas en la consigna.

### 3.2. Modelo de Entidades y Paquetes

Se definieron las clases de dominio (POJOs) Pedido (Clase A) y Envío (Clase B) en el paquete `utn.tfi.programacion2.entities`, siguiendo la estructura de paquetes requerida. Ambas clases incluyen los atributos `id` (Long) y `eliminado` (Boolean) para cumplir con los requisitos de persistencia y baja lógica.

La relación 1-a-1 unidireccional ( $\rightarrow$ ) se implementó declarando un atributo privado de tipo Envío dentro de la clase Pedido (`private Envío envio;`), siguiendo la consigna.

### 3.3. Diseño de Base de Datos (MySQL)

Se optó por un esquema relacional en MySQL para persistir el modelo de objetos. Se crearon las tablas `pedido` y `envio`.

Decisión de Diseño (Relación 1-a-1):

Para implementar la relación 1-a-1 (Pedido  $\rightarrow$  Envío) en la base de datos, se tomó la decisión de añadir una clave foránea (FK) en la tabla pedido (A) que referencia al id de la tabla envío (B).

Para garantizar la integridad y la restricción 1-a-1 real (exigida por la rúbrica<sup>555</sup>), se añadió una restricción UNIQUE sobre dicho campo (envio\_id). Esto impide que un envío sea asignado a más de un pedido. El campo envio\_id es NULLABLE para permitir que un Pedido exista antes de que se le asigne un Envío.

Se proveen los scripts crear\_bd.sql y datos\_prueba.sql para garantizar la reproducibilidad total del proyecto.

### 3.4. Configuración de Conexión

Se implementó una clase DatabaseConnection en el paquete utn.tfi.programacion2.config<sup>7</sup>. Esta clase centraliza los datos de conexión (URL, usuario, contraseña) y provee un método estático getConnection() que retorna una java.sql.Connection<sup>8</sup>. Esta conexión será gestionada (apertura y cierre, commit/rollback) por la capa de Servicios, tal como lo pide la consigna.

### 3.5. Diagrama de Clases (UML)

El siguiente diagrama UML ilustra la estructura del dominio, los atributos principales con visibilidad privada (-) y la relación unidireccional 1  $\rightarrow$  0..1 entre las entidades.

## 4 Implementación de Capas (Las partes de tus compañeros, DAO, Service, etc.).

### 4.1. CAPA DAO

#### GenericDao:

El archivo GenericDao define una **interfaz genérica** que sirve como contrato para todas las clases DAO del proyecto.

Su objetivo es **estandarizar las operaciones básicas de acceso a datos** para cualquier entidad del sistema (por ejemplo, Pedido, Envío, etc.), garantizando que todas sigan la misma estructura y principios de programación genérica.

Los métodos definidos son:

- save(T entity): Inserta una nueva entidad en la base de datos.
- findById(long id): Busca una entidad por su identificador.
- findAll(): Devuelve una lista con todas las entidades almacenadas.
- update(T entity): Actualiza los datos de una entidad existente.

- `delete(long id)`: Realiza una baja lógica o física, según el caso.
- `saveTx(T entity, Connection conn)`: Permite ejecutar inserciones dentro de una **transacción activa**, optimizando la gestión de commits y rollbacks.

Esta interfaz actúa como **base abstracta** para todos los DAO concretos del sistema, fomentando el **principio de reutilización y desacoplamiento** entre la lógica de negocio y el acceso a datos.

## EnvioDao:

La interfaz `EnvioDao` extiende de `GenericDao<Envio>`, especializándola para trabajar con objetos de tipo `Envio`.

Permite agregar métodos propios y consultas específicas relacionadas con la entidad `Envio`, como por ejemplo buscar por estado o empresa de envío.

De esta forma, `EnvioDao` funciona como una interfaz especializada que mantiene la estructura genérica pero deja abierta la posibilidad de ampliar comportamientos según las necesidades del dominio.

## EnvioDaoImpl

La clase `EnvioDaoImpl` implementa la interfaz `EnvioDao` y contiene las operaciones CRUD concretas para la tabla `envio`.

Se comunica directamente con la base de datos utilizando JDBC mediante una clase auxiliar `DatabaseConnection`.

Principales características:

- **Inserción (save):**  
Inserta un nuevo envío y recupera su ID autogenerado usando `Statement.RETURN_GENERATED_KEYS`.
- **Consulta (findById):**  
Devuelve un objeto `Envio` si encuentra una fila coincidente con el ID recibido, mapeando los campos mediante el método privado `mapResultSetToEnvio`.
- **Listado (findAll):**  
Retorna todos los envíos no eliminados lógicamente.

- Eliminación lógica (`delete`):  
Marca un envío como eliminado sin borrarlo físicamente de la base, manteniendo la integridad referencial.
- Mapeo (`mapResultSetToEnvio`):  
Convierte los datos obtenidos del `ResultSet` en un objeto `Envio`, transformando tipos SQL (por ejemplo `Date`) a tipos Java (`LocalDate`) y enums (`EstadoEnvio`, `TipoEnvio`, `Empresa`).

Esta implementación refleja el uso de baja lógica, encapsulamiento de consultas y reutilización de código mediante métodos privados.

## **PedidoDao**

La interfaz `PedidoDao` extiende `GenericDao<Pedido>` y representa el contrato de acceso a datos para la entidad `Pedido`.

Permite mantener un diseño modular y coherente con el resto del sistema, además de ofrecer la posibilidad de agregar consultas específicas (por ejemplo, `findByEstado` o `findByCliente`).

## **PedidoDaoImpl**

`PedidoDaoImpl` es la clase que implementa los métodos definidos en `PedidoDao`.

Gestiona las operaciones sobre la tabla `pedido` aplicando buenas prácticas de programación y el patrón DAO.

Principales operaciones:

- Inserción (`save`):  
Inserta un nuevo pedido, incluyendo los datos del cliente, el total y el estado.  
Si el pedido tiene un envío asociado, también guarda su ID como clave foránea (`envio_id`).
- Consulta por ID (`findById`):  
Busca un pedido específico y lo reconstruye mediante el método `mapResultSetToPedido`.
- Listado (`findAll`):  
Recupera todos los pedidos registrados en la base.
- Actualización (`update`):  
Permite modificar campos específicos del pedido (como nombre del cliente, total y estado) sin alterar el resto.

- Eliminación lógica (delete):  
Marca el pedido como eliminado mediante un campo booleano, evitando la pérdida de datos.
- Mapeo (mapResultSetToPedido):  
Convierte cada fila del ResultSet en un objeto Pedido, incluyendo la relación con Envio si existe.
- Transacciones (saveTx):  
El método está preparado para una futura implementación dentro de una transacción JDBC, lo que permitirá ejecutar operaciones combinadas (por ejemplo, guardar un pedido y su envío en un mismo commit).

## Conclusión general

La capa DAO implementada cumple el rol de intermediaria entre la base de datos y la lógica de negocio, separando responsabilidades y evitando dependencias directas del resto del sistema sobre JDBC.

Gracias a esta estructura:

- Se mejora la mantenibilidad del código.
- Se facilita la reutilización y prueba de componentes.
- Se garantiza una arquitectura modular y escalable, con soporte para futuras integraciones (por ejemplo, una capa de servicios REST o una migración a JPA/Hibernate).

## 4.2 Capa Service

La capa Service constituye el núcleo de la lógica de negocio del proyecto.

Su propósito principal es coordinar las operaciones entre la capa DAO y la capa de presentación (menú de consola), garantizando la validez de los datos y la consistencia transaccional en la base de datos.

Está compuesta por las siguientes clases e interfaces:

### Interfaz GenericService

Es una interfaz genérica que define las operaciones básicas del CRUD (crear, leer, listar, actualizar y eliminar) para cualquier entidad del sistema.

Sirve como modelo común para los distintos servicios (PedidoService y EnvioService), garantizando coherencia y reutilización de código.

No implementa lógica pero se encarga de establecer el contrato que las clases concretas deben cumplir. Por otra parte, mejora la mantenibilidad al permitir aplicar el mismo patrón a diferentes entidades.

### Interfaz EnvioService

Extiende GenericService<Envio> e incorpora métodos específicos para la entidad Envio, como la búsqueda por tracking.

Responsabilidades:

Define operaciones adicionales necesarias para el manejo del envío (por ejemplo, findByTracking(String tracking)). Además, permite centralizar las reglas de negocio relacionadas con los envíos.

## **Clase EnvioService**

Implementa la lógica de negocio de EnvioService, utilizando el DAO correspondiente (EnvioDaoImpl) para acceder a la base de datos.

Además de los métodos CRUD, realiza validaciones previas sobre los datos antes de persistirlos.

Responsabilidades:

- Validar campos obligatorios: tracking, empresa, tipo, costo > 0.
- Delegar operaciones CRUD al DAO (envioDao).
- Manejar excepciones de forma controlada, devolviendo mensajes claros al usuario.
- Garantizar que los registros marcados como eliminados (eliminado = true) no se procesen como activos.

Antes de guardar un envío, el método save(Envio envio) verifica que el tracking no este vacío y que el costo sea positivo. Solo si pasa las validaciones se ejecuta envioDao.save(envio).

## **Interfaz PedidoService**

Extiende GenericService<Pedido> e incorpora métodos específicos del dominio, en especial el método transaccional crearPedidoCompleto, que inserta simultáneamente un pedido y su envío asociado.

Responsabilidades:

Declara el método crearPedidoCompleto(Pedido pedido) que orquesta la creación del pedido y su envío en una misma transacción. También incluye la búsqueda por número Incluye la búsqueda por número (findByNumero).

## **Clase PedidoServiceImpl**

Es la clase más relevante de la capa Service ya que implementa la lógica de negocio del pedido y maneja operaciones transaccionales con control de commit/rollback, asegurando la integridad de los datos entre las tablas pedido y envío.

### Validaciones de negocio

Antes de persistir un pedido, se valida:

- numero no vacío ni duplicado.
- clienteNombre obligatorio.
- total > 0.
- que el estado no sea nulo.

- Si existe envío, validar su tracking.

### Manejo de transacciones

El método crearPedidoCompleto realiza:

- Apertura de una conexión a la base de datos.
- Desactivación de autoCommit (conn.setAutoCommit(false)).
- Inserción del Envío con envioDao.saveTx(envio, conn).
- Inserción del Pedido asociado con pedidoDao.saveTx(pedido, conn).
- Ejecución de commit() si todo sale bien.
- Ejecución de rollback() en caso de excepción.
- Cierre y restablecimiento del autoCommit.

### Integridad de la relación 1→1

Al usar la misma conexión en ambas inserciones, se garantiza que el Pedido y su Envío solo se creen si ambos se insertan correctamente, cumpliendo el requerimiento de asociación unidireccional y evitando inconsistencias.

## 6 Conclusiones.

La realización de este trabajo nos permitió consolidar la arquitectura por capas (DAO/Service) y la persistencia de datos con JDBC2. El desafío principal fue la gestión de la relación 1-a-1 (Pedido → Envío) y la implementación de operaciones transaccionales (commit/rollback), logrando una aplicación funcional que asegura la integridad de los datos.

## 7 Herramientas utilizadas.

Para la realización de esta primera etapa del trabajo (Diseño, Entidades y Configuración), se emplearon las siguientes herramientas y fuentes:

### Herramientas

- **IDE (Entorno de Desarrollo):** Apache NetBeans.
- **Gestión de Base de Datos:** MySQL Workbench.
- **Gestión de Dependencias y Build:** Apache Maven (integrado en NetBeans).
- **Control de Versiones:** Git (integrado en NetBeans).
- **Alojamiento de Repositorio:** GitHub.
- **Modelado UML:** [diagrams.net](https://diagrams.net) (anteriormente [draw.io](https://draw.io)).
- **Asistencia de IA:** Se utilizó Gemini (Google) como asistente para la guía paso a paso, la estructuración del código y la resolución de dudas.