

TRABAJO  
PRÁCTICO  
INTEGRADOR

PROGRAMACIÓN II

**Grupo nº 174**

**PEDIDO - ENVÍO**

**BENJAMIN ATAIDE**

**GONZALO CASTELLINI**

**LAUTARO BUCHEK**

[LINK A VIDEO](#)

## ÍNDICE

<i>Elección de dominio</i> .....	2
<i>Diseño: decisiones clave</i> .....	2
Arquitectura por capas.....	3-4
Persistencia.....	5-6
Validaciones y reglas de negocio.....	6-8
Pruebas realizadas.....	8-10
Conclusión.....	10
Anexo Fuentes e IA.....	3-5

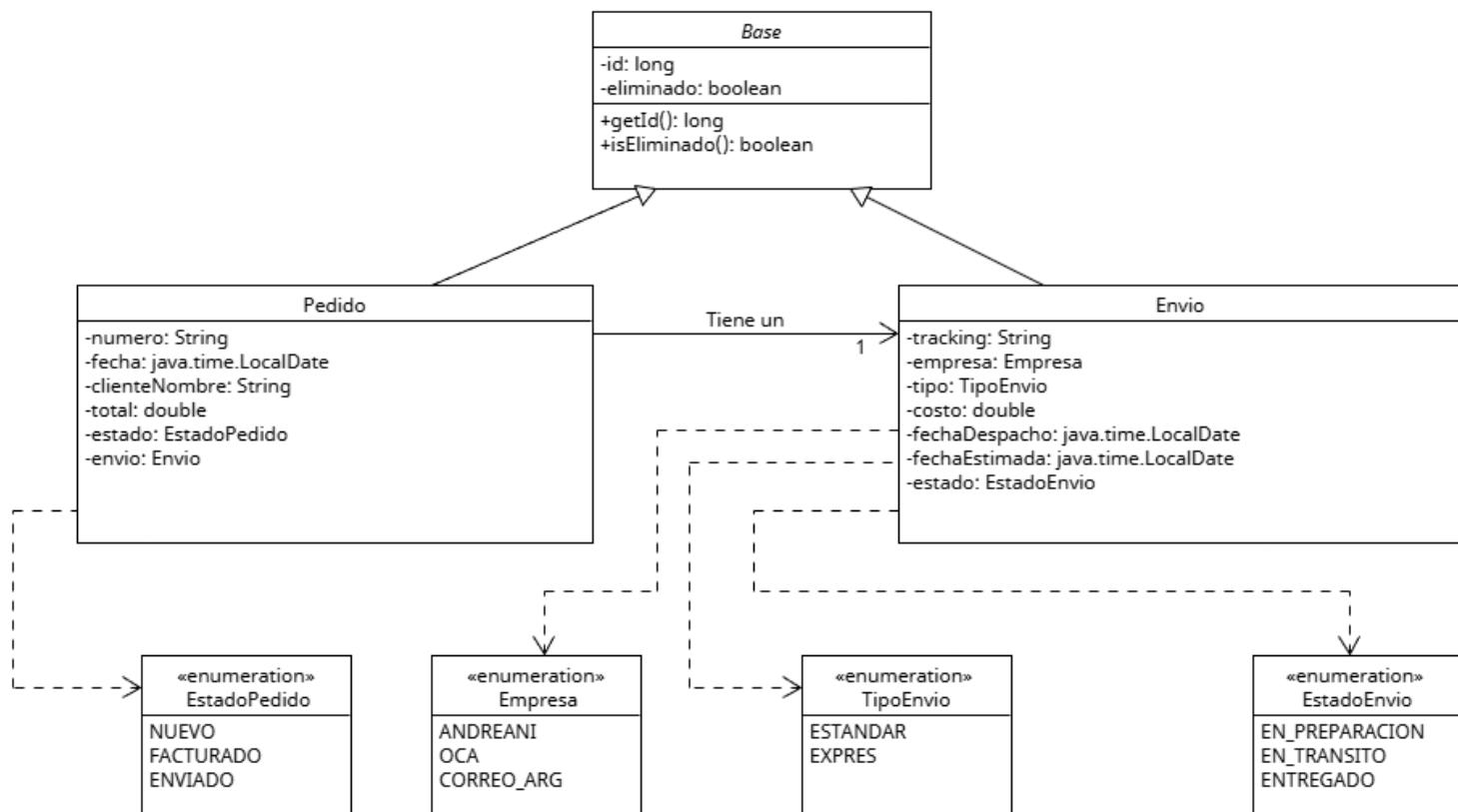
## Elección de dominio

El par elegido para el presente trabajo fue PEDIDO - ENVÍO. Si bien no hubo una razón particular para escoger esta pareja, a todos los integrantes nos interesaba diseñar y crear un proyecto que se asemeje con la implementación que pueden llegar a tener las grandes empresas dedicadas a este rubro en el país.

## Diseño: decisiones clave

Previo a la implementación en código, partimos del diseño base provisto por la cátedra para las clases Pedido y Envío. Ambas respetan tanto campos como tipos de datos. A su vez, se aplicaron también las reglas de negocio para las entidad en la base de datos. Respecto a la relación y cardinalidad, se estableció un vínculo unidireccional 1 → 1, donde una instancia de Pedido referencia a un único Envío. En el caso de la base de datos, la referencia se establece mediante clave foránea con restricción UNIQUE para garantizar el tipo de relación uno a uno.

Debido a que ambas clases presentan dos campos iguales ( id y eliminado), optamos por crear una clase Base con estos atributos, y luego las clases de Pedido - Envío los heredan. De este modo, ahorraremos la duplicación de código y generamos una clase padre que luego sirve como plantilla para el resto de clases del sistema, incluso si decidimos añadir nuevas entidades a futuro.



## Arquitectura por capas

### Distribución de funcionalidades por paquetes

#### Config

Contiene las clases encargadas de la configuración y conexión de la Base de Datos. Donde debemos configurar el puerto del servidor MySQL con el cual hayamos iniciado conexión en el SGDB. De igual manera, tenemos que especificar el nombre de la base de datos, así como también, la contraseña de usuario root propia (en caso de haber dispuesto de una).

#### Dao

Este paquete Abstraer y centralizar el acceso a la base de datos.

Este paquete es la única parte del sistema que "sabe" cómo hablar con la base de datos. Su función es aislar la lógica de persistencia (consultas SQL) del resto de la aplicación. Si en el futuro se quisiera cambiar de MySQL a otra base de datos, solo este paquete necesitaría ser modificado.

GenericDAO<T>: Es una interfaz genérica que define el "contrato" para todas las clases DAO. Establece las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) que cualquier entidad debe soportar. Usar una interfaz genérica promueve un diseño consistente y polimórfico.

PedidoDAO.java y EnvioDAO.java: Son las implementaciones concretas de GenericDAO para cada entidad. Aquí es donde residen las sentencias SQL (INSERT, UPDATE, SELECT, etc.), siempre utilizando PreparedStatement para prevenir inyecciones SQL. Se encargan de "traducir" los objetos Java (ej. Pedido) a registros en la base de datos y viceversa.

Responsabilidad Clave: Mapear los datos entre el mundo de los objetos (Java) y el mundo relacional (tablas de la base de datos).

#### Models

Este paquete representa las entidades de negocio del sistema.

Este paquete contiene las clases que modelan los datos con los que trabaja la aplicación. Son "Plain Old Java Objects" (POJOs) que encapsulan los atributos y el estado de las entidades, como un Pedido o un Envío.

Base.java: Es una clase abstracta que sirve como pilar para todas las demás entidades. Su función es evitar la duplicación de código proveyendo campos comunes como id (identificador único) y eliminado (para la eliminación lógica o "soft delete").

Pedido.java y Envio.java: Son las entidades principales del dominio. Heredan de Base y definen los atributos específicos de un pedido (número, fecha, total, etc.) y de un envío (tracking, empresa, costo, etc.). Su responsabilidad es contener y gestionar el estado de un objeto de negocio.

EstadoPedido.java, EstadoEnvio.java, Empresa.java, TipoEnvio.java: Son enumeraciones (enum). Su propósito es crucial para la robustez del código: definen un conjunto fijo y seguro de valores para ciertos atributos (por ejemplo, el estado de un pedido solo puede ser NUEVO, FACTURADO o ENVIADO). Esto previene errores de tipado y hace el código más legible que usar cadenas de texto o números mágicos.

### **Service**

Este paquete Implementa la lógica de negocio y las reglas del sistema.

Esta capa actúa como un intermediario entre la capa de presentación (el menú) y la capa de acceso a datos (DAO). Su función es orquestar las operaciones y aplicar las validaciones que definen cómo debe funcionar el sistema.

GenericService<T>: Al igual que en el DAO, esta interfaz genérica define un contrato común para todos los servicios, asegurando que ofrezcan un conjunto estándar de operaciones.

PedidoServiceImpl.java y EnvioServiceImpl.java: Implementan la lógica de negocio específica para Pedidos y Envíos. Por ejemplo:

Validaciones: Antes de crear o actualizar un pedido, PedidoServiceImpl verifica que el número de pedido no esté duplicado, que el total no sea negativo y que los campos obligatorios estén presentes.

Coordinación: Un servicio puede coordinar operaciones entre múltiples DAOs. Por ejemplo, si crear un pedido implicara registrar algo en otra tabla, el servicio se encargaría de llamar a los DAOs correspondientes.

Diferencia con el DAO: Mientras el DAO solo se preocupa de guardar y leer datos, el Servicio o Service se preocupa de que esos datos sean válidos y consistentes según las reglas del negocio.

### **Main**

Este paquete gestiona la interfaz de usuario (UI) y el ciclo de vida de la aplicación.

Este paquete es el punto de entrada y la cara visible de la aplicación para el usuario. Se encarga de presentar las opciones, capturar la entrada del usuario y delegar las acciones a la capa de servicio. No contiene ninguna lógica de negocio.

Main.java y AppMenu.java: Son los puntos de arranque de la aplicación. AppMenu tiene la responsabilidad principal de:

Inicialización: Crea todas las dependencias necesarias (DAOs, Servicios, etc.) y las "conecta" entre sí (Inyección de Dependencias manual).

Ciclo Principal: Mantiene la aplicación corriendo en un bucle, mostrando el menú y esperando la acción del usuario.

MenuDisplay.java: Su única responsabilidad es mostrar el menú en la consola. Separar esto en su propia clase hace que el código sea más limpio y sigue el Principio de Responsabilidad Única.

MenuHandler.java: Actúa como el "controlador" de la interfaz. Recibe la opción del usuario desde AppMenu, solicita los datos necesarios (ej. "Ingrese el número de pedido"), y llama a los métodos correspondientes en la capa de Service. Traduce las interacciones del usuario en llamadas a la lógica de negocio.

TestConexion.java: Una clase de utilidad muy práctica para verificar rápidamente que la conexión con la base de datos funciona correctamente, sin tener que iniciar toda la aplicación.

## Persistencia

- En nuestro proyecto, la persistencia es fundamental para que los datos clave del negocio (Pedidos y Envíos) no se pierdan al finalizar la ejecución. Para lograrlo, construimos una capa de persistencia sólida, segura y fácil de mantener, conectándonos a una base de datos MySQL mediante JDBC.

El propósito central de nuestro diseño fue separar completamente la lógica de negocio de cómo se guardan los datos. Esta división es crucial para que el sistema sea escalable y más fácil de mantener.

A continuación, las decisiones de diseño y los patrones que implementamos:

### 1. Arquitectura Basada en el Patrón DAO (Data Access Object)

Decidimos usar el patrón DAO para mantener la lógica de negocio lejos de los detalles de la base de datos (SQL).

- **Contrato General (GenericDAO<T>):** Es una interfaz que define las operaciones básicas de Crear, Leer, Actualizar y Eliminar (CRUD), asegurando que todo se haga de forma consistente.
- **Implementaciones Específicas (PedidoDAO y EnvioDAO):** Son las clases que realmente contienen el código SQL para cada entidad. Esto nos da la libertad de cambiar el motor de base de datos en el futuro sin tanto problema.

### 2. Gestión Centralizada de la Conexión

Manejamos toda la configuración y el uso de las conexiones de forma centralizada en la clase DatabaseConnection.

- **Fábrica de Conexiones:** Es la encargada de entregar conexiones a los DAOs, cargando el driver de MySQL y verificando la configuración al inicio.
- **Manejo de Recursos:** Nos aseguramos de que las operaciones JDBC utilicen bloques try-with-resources. Esto garantiza que las conexiones y statements se cierren solos y de forma segura, evitando olvidarnos de cerrarlos.

### 3. Seguridad: Protección frente a Inyección SQL

Para protegernos contra ataques de inyección SQL, solo construimos las consultas usando PreparedStatement con los *placeholders* (?). Esto garantiza que los valores que nos dé el usuario se asignen de manera segura, eliminando los riesgos.

### 4. Integridad de Datos con Transacciones

Implementamos la clase TransactionManager para asegurar que las operaciones de varios pasos se ejecuten como un todo (atomicidad).

- Nos permite iniciar la transacción (`conn.setAutoCommit(false)`), guardar los cambios (`conn.commit()`) o deshacerlos (`conn.rollback()`).

- Al implementar AutoCloseable, la usamos con try-with-resources para garantizar que si no confirmamos la transacción de forma explícita, se deshaga automáticamente.

## 5. Mapeo Objeto-Relacional (ORM) Manual

La capa DAO se encarga de la "traducción" entre los objetos de Java y las filas de la base de datos.

- **De Objeto a Fila:** Usamos métodos de ayuda (`setParams()`) para pasar los atributos de los objetos a los parámetros de un `PreparedStatement`.
- **De Fila a Objeto:** El método `map()` toma un `ResultSet` y lo usa para crear y llenar el objeto Java correspondiente.

Este enfoque manual proporciona un control total sobre el rendimiento de la persistencia, sin dependencia de frameworks ORM externos.

## Validaciones y reglas de negocio

En nuestro sistema, la integridad y consistencia de los datos son nuestra máxima prioridad. Para asegurar esto, implementamos una estrategia de validación robusta y organizada en múltiples **niveles**, separando claramente las responsabilidades entre la interfaz de usuario (el menú) y la lógica de negocio (los servicios).

El principio clave es que la Capa de Servicio es la guardiana de todas las reglas de negocio. Es la única responsable de garantizar que un objeto (como un Pedido o un Envío) sea completamente válido antes de que toque la base de datos.

### 1. Separación de Responsabilidades: UI vs. Lógica de Negocio

Distinguimos dos tipos de validaciones, asignando cada una a la capa que le corresponde:

Tipo de Validación	Capa Responsable	Objetivo Principal	Ejemplo
<b>Validación de Formato</b>	<b>Capa Main (MenuHandler )</b>	Asegurar que la entrada del usuario tenga el <i>tipo de dato</i> correcto.	Manejo de <code>try-catch</code> para <code>NumberFormatException</code> ( <code>Double.parseDouble</code> ( <code>Double</code> )) o <code>DateTimeParseException</code> ( <code>YYYY-MM-DD</code> ). Si el formato falla, el usuario recibe un mensaje

			inmediato y la capa de servicio ni siquiera se entera.
<b>Validación de Negocio</b>	<b>Capa Service</b>	Aplicar las reglas de dominio del negocio.	Una vez que el dato tiene el tipo correcto (ej., un <b>Double</b> ), verificar que cumpla con las reglas (ej., que un <b>total</b> no sea negativo).

**En resumen:** La capa de negocio solo recibe datos con el tipo correcto y se enfoca exclusivamente en validar las reglas del dominio.

## 2. Implementación de las Reglas en la Capa de Servicio

Dentro de cada clase de servicio (**PedidoServiceImpl**, **EnvioServiceImpl**), centralizamos todas las validaciones en métodos privados (como **validatePedido**). Estos métodos se ejecutan al inicio de cualquier operación de escritura (insertar, actualizar). Si una validación no pasa, se lanza una **IllegalArgumentException** que detiene la operación antes de que se envíe a la base de datos.

Nuestras reglas de negocio se clasifican en tres categorías:

### a) Reglas de Integridad de Datos (Lo más fundamental)

Son las validaciones que evitan que los datos se corrompan:

- **Campos Obligatorios:** Verificamos que todos los campos definidos como NOT NULL en la base de datos no lleguen como **null** (ej., **Pedido.estado**, **Envio.empresa**).
- **Restricciones de Longitud:** Nos aseguramos de que los textos (**VARCHAR**) no excedan el tamaño definido en la DB (ej., **Pedido.numero** no más de 20 caracteres).

### b) Reglas de Negocio Simples (Reglas de un solo campo)

Definen el dominio de valores válidos para campos específicos:

- **Valores Numéricos:** Garantizamos que montos como **Pedido.total** o **Envio.costo** no sean negativos.
- **Coherencia de Fechas:** En el servicio de Envío, validamos la cronología: la **fechaEstimada** debe ser posterior a la **fechaDespacho** (**fe.isBefore(fd)**).

### c) Reglas de Negocio Complejas (Requieren acceso a datos)

Estas son las validaciones más críticas, ya que dependen del estado actual de la base de datos:

- **Garantía de Unicidad:** Un ejemplo clave es el `Pedido.numero`, que debe ser único en todo el sistema. Para esto, el método `validateNumeroUnique` realiza una consulta a la base de datos (a través del DAO) para verificar duplicados.
- **Manejo Inteligente de Actualizaciones:** Esta validación es crucial. Si estamos *creando* un pedido, el número debe ser nuevo. Si estamos *actualizando* un pedido, permitimos que mantenga su número actual, pero evitamos que se cambie a un número que ya está en uso por **otro** pedido existente (`existente.getId() != pedidold`).

**Conclusión:** Esta estrategia de validación en capas nos permite tener un sistema donde solo los datos válidos y consistentes ingresan a la base de datos. Al centralizar toda la lógica de negocio en la capa de Servicio, nuestro sistema es más seguro, predecible y sobre todo mucho más fácil de mantener.

## Pruebas realizadas

### Test de conexión

```
public class TestConexion {
    public static void main(String[] args) {
        try (Connection conn = DatabaseConnection.getConnection()) {
            if (conn != null) {
                System.out.println("Conexion exitosa a la base de datos");

                DatabaseMetaData metaData = conn.getMetaData();
                System.out.println("Usuario conectado: " + metaData.getUserName());
                System.out.println("Base de datos: " + conn.getCatalog());
                System.out.println("URL: " + metaData.getURL());
                System.out.println("Driver: " + metaData.getDriverName() + " v" + metaData.getDriverVersion());
            } else {
                System.out.println("No se pudo establecer la conexion.");
            }
        } catch (SQLException e) {
            System.err.println("Error al conectar a la base de datos: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Ubicado en la capa Main, al ejecutar el archivo `TestConexion` realizaremos una prueba para verificar que la conexión a la base de datos mediante JDBC sea la correcta.

```
prog2int.Main.TestConexion > main >
Output - prog2int (run) #13 x
run:
Conexion exitosa a la base de datos
Usuario conectado: root@localhost
Base de datos: pedido_envio_jdbc
URL: jdbc:mysql://localhost:3307/pedido_envio_jdbc
Driver: MySQL Connector/J vmysql-connector-j-9.5.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

*En caso de que la conexión sea exitosa, se muestra por consola junto con los datos adicionales del usuario conectado, la base de datos y el driver utilizado.*

### Operación de lectura

#### Menu interactivo

```
===== MENU PEDIDOS / ENVIOS =====
1. Crear pedido
2. Listar pedidos
3. Actualizar pedido
4. Eliminar pedido
5. Buscar pedido por numero
6. Crear envio independiente
7. Listar envios
8. Actualizar envio por ID
9. Eliminar envio por ID
0. Salir
Ingrese una opcion:
```

*Listado con las diferentes opciones de operación para interactuar con la base de datos*

#### Listado de envíos (operación de lectura)

```
Ingrese una opcion: 7
ID=2 | tracking=TRK0000002 | empresa=CORREO_ARG | tipo=EXPRESS | estado=EN_PREPARACION | costo=500.0
ID=3 | tracking=TRK0000003 | empresa=OCA | tipo=EXPRESS | estado=EN_TRANSITO | costo=450.0
ID=4 | tracking=TRK00005 | empresa=ANDREANI | tipo=ESTANDAR | estado=EN_PREPARACION | costo=150.0
```

### Operación de escritura (forzando error) - Rollback

Violacion de constraint FK única al crear un pedido y asignarlo a un envío que ya estaba referenciando un pedido existente.

#### Listado de pedidos con sus respectivos envíos asignados ( enviod )

```
Ingrese una opcion: 2
ID=1 | nro=PD0000001 | cliente=Gonzalo | estado=NUEVO | total=2500.0 | envioId=1
ID=2 | nro=PD0000002 | cliente=Benjamin | estado=FACTURADO | total=6500.0 | envioId=2
ID=3 | nro=PD0000003 | cliente=Gonzalo | estado=NUEVO | total=15000.0 | envioId=3
```

Ahora intentaremos crear un nuevo pedido y asignarlo al envío con id : 1 (que como vemos en la captura ya está asociado al primer pedido de la lista( numero PD0000001)

```
Ingrese una opcion: 1
Numero (max 20): PD000007
Fecha (YYYY-MM-DD, Enter para vacio): 2025-11-16
Cliente (nombre completo, Enter para vacio): Lautaro
Total: 439.0
Estado de pedido (NUEVO/FACTURADO/ENVIADO): NUEVO
Desea asociar a un envio nuevo o existente? (n/e): E
Ingrese el ID del envio para asociarlo al pedido creado:
1
Error al hacer insert, transaccion revertida|
Pedido creado. ID: 0
```

El sistema de gestión va a lanzar un error debido a la violación de relación 1-1 de FK con PK (id de envio), el cual estamos detectando de manera correcta en la capa DAO, dentro del método *insertar (Pedido p)*. El manejo de la SQLException realiza un rollback de las operaciones de insert y para garantizarnos del correcto accionar, se notifica por consola con el mensaje: “Error al hacer insert, transacción revertida”.

### Conclusión

Este Proyecto de la materia Programación II fue una oportunidad clave para aplicar y consolidar todo lo aprendido en la materia, enfocándonos en crear una solución que fuera tanto robusta como completamente funcional.

Durante el desarrollo, logramos integrar de forma efectiva los conceptos esenciales de la Programación Orientada a Objetos (POO), como la encapsulación, la herencia y el polimorfismo. Esto fue vital para estructurar el código de manera modular y fácil de mantener. La selección de las tecnologías y metodologías adecuadas simplificó la gestión de datos y la interacción con el usuario, demostrando capacidad para diseñar y llevar a cabo un proyecto de software completo.

En resumen, se convierte en una experiencia sumamente valiosa en la aplicación práctica de la programación para resolver problemas reales. Hemos podido consolidar las habilidades necesarias para afrontar futuros desafíos en el campo del desarrollo de software. Como posible mejora, considero que se podría rediseñar la relación 1 a N, donde muchos pedidos se relacionen con un único envío.

## ANEXO IA

### 💡 Tema: Transacciones, rollback y arquitectura DAO/Service

---

#### 👤 CodeCraft:

Estamos terminando la capa Service y tenemos dudas sobre cómo manejar correctamente las transacciones, especialmente en operaciones donde intervienen múltiples DAOs.

Entendemos el `setAutoCommit(false)`, pero no estamos seguros de:

1. Cuándo usar rollback,
2. Cómo evitar commits parciales,
3. Si el Service debería manejar todo o si conviene delegar algo al DAO.

---

¿Nos podés explicar la forma correcta de estructurar esto en un sistema con arquitectura en capas?

---

#### 🤖 ChatGPT:

Claro. En una arquitectura por capas como la de su proyecto, las transacciones deben manejarse exclusivamente en la capa Service, nunca en los DAOs.

Esto se debe a que el DAO solo conoce sentencias SQL individuales, pero no entiende el “contexto” de la operación completa, mientras que el Service sí.

La estructura correcta es:

- ① El Service inicia la transacción

```
conn.setAutoCommit(false);
```

- ② El Service llama a los DAOs

Ejemplo: insertar un Pedido y luego su Envío asociado.

- ③ Si alguna parte falla

Capturan la excepción y realizan:

```
conn.rollback();
```

- ④ Si todo sale bien

```
conn.commit();
```

- ⑤ El Service siempre devuelve el control limpio y consistente.

Con esto garantizan atomicidad:

O se hace todo, o no se hace nada.

El DAO NO debe hacer commit ni rollback.

Su única responsabilidad es ejecutar SQL y mapear datos.

---

### CodeCraft:

Perfecto. Otra duda: ¿Está bien que cada DAO abra su propia conexión, o todos deben usar la misma conexión para que la transacción sea consistente?

---

### ChatGPT:

Para operaciones transaccionales, todos los DAOs involucrados deben compartir la misma

Si cada DAO abre su propia conexión, no existe garantía de atomicidad, porque MySQL tratará cada operación como transacciones separadas.

La forma correcta es:

- El Service crea o solicita una única conexión desde DatabaseConnection.
- Esa conexión se pasa a los DAOs y ellos la usan para todas las operaciones involucradas.

---

Su diseño de **TransactionManager** y la inyección manual que están usando va bien encaminado.

---

 **CodeCraft:**

Gracias. Otra cosa que nos complica: el mapeo manual entre objetos Java y filas SQL. ¿Cómo sabemos si estamos respetando las buenas prácticas? No usamos Hibernate, así que todo es manual.

---

 **ChatGPT:**

Lo que están haciendo se llama ORM manual.

Las buenas prácticas para implementarlo correctamente son:

- Crear un método **map(ResultSet rs)** por cada DAO.
- Crear un método **setParams(PreparedStatement ps, Entidad e)** para los INSERT y UPDATE.
- No mezclar lógica de mapeo con lógica de negocio.
- No construir SQL con concatenaciones: siempre PreparedStatement.

- Respetar los tipos equivalentes Java ↔ SQL  
(`LocalDate` ↔ `DATE`, `ENUM` ↔ `VARCHAR`, `boolean` ↔ `tinyint`).

Esto les dará código limpio, mantenible y sin acoplamiento innecesario.

---

 **CodeCraft:**

Última duda: ¿Dónde deben ir las validaciones de negocio?

Estamos repartiendo cosas entre el menú y el Service, pero no estamos seguros.

---

 **ChatGPT:**

La regla es simple:

- El menú valida formato  
(que el usuario no ponga letras donde debe ir un número).
- El Service valida reglas de negocio  
(que un costo sea positivo, que una fecha estimada no sea menor que la fecha de despacho, que un estado sea válido, etc.).

Si el Service recibe un dato inválido, debe lanzar una excepción.

Así evitan corrupción de datos antes de llegar al DAO.

Esta separación es fundamental y está bien implementada en su proyecto.