

# Trabajo Práctico Integrador

Programación 2 - Entidades Pedido-Envío

❖ **Alumnos:**

- Sussini Guanziroli, Patricio
- Vazquez, Matías Ezequiel

❖ **Materia:** Programación 2

❖ **Tutora:** Florencia Gubiotti

❖ **Profesor:** Ariel Enferrel

❖ **Fecha de Presentación:** 17/11/2025

# INTRODUCCIÓN

El presente informe documenta el desarrollo de un sistema de gestión de pedidos y envíos implementado en Java, que constituye nuestro Trabajo Práctico Integrador para la materia Programación 2. El objetivo principal fue aplicar los conceptos de Programación Orientada a Objetos, persistencia de datos con JDBC y arquitectura en capas en un sistema funcional y profesional.

Decidimos desarrollar un sistema que simula la gestión operativa de una empresa de comercio electrónico, donde cada pedido debe estar asociado obligatoriamente a un envío específico, implementando una relación 1→1 unidireccional estricta. Esta elección nos permitió demostrar el manejo de relaciones entre entidades, transacciones ACID y validaciones de integridad referencial en múltiples niveles.

El sistema fue desarrollado completamente en Java 17, utilizando MySQL 8.0 como motor de base de datos y JDBC como tecnología de persistencia. La aplicación se ejecuta mediante una interfaz de consola interactiva que permite realizar operaciones CRUD completas sobre ambas entidades.

## DECISIONES DE DISEÑO

### **Arquitectura en Capas**

Optamos por implementar una arquitectura en capas claramente diferenciadas, siguiendo el principio de separación de responsabilidades. Esta decisión nos permitió mantener el código organizado, facilitar el mantenimiento y mejorar las posibilidades de testeo y validación sobre el sistema.

### **Capa de Modelos (Models):**

Definimos una clase abstracta Base que contiene los atributos comunes `id` y `eliminado`, aplicando el principio DRY (Don't Repeat Yourself). Las entidades Envío y Pedido heredan de esta clase base, agregando sus atributos específicos. Decidimos incluir enumeraciones dentro de cada entidad para garantizar valores controlados en campos como estado, empresa y tipo de envío.

### **Capa de Acceso a Datos (DAO):**

Implementamos el patrón DAO mediante interfaces genéricas (`GenericDAO<T>`) que definen las operaciones CRUD estándar. Cada entidad tiene su propia implementación concreta (`EnvioDAO`, `PedidoDAO`). Esta capa es responsable exclusivamente de la persistencia, utilizando `PreparedStatement` para prevenir inyección SQL y garantizar la seguridad de las consultas.

### **Capa de Lógica de Negocio (Service):**

Creamos una capa de servicios (`EnvioServiceImpl`, `PedidoServiceImpl`) que encapsula todas las validaciones y reglas de negocio. Decidimos que las validaciones no se realizarán en los DAOs para mantener la separación de responsabilidades, los DAOs solo ejecutan queries, mientras que los Services validan los datos antes de persistirlos.

### **Capa de Presentación (Main):**

Separamos la interfaz de usuario en tres clases especializadas: `AppMenu` (controlador principal), `MenuDisplay` (visualización) y `MenuHandler` (manejo de operaciones). Esta separación nos permitió mantener el código limpio y facilitar futuras modificaciones en la interfaz.

## **Relación 1→1 Unidireccional**

Implementamos una relación 1→1 unidireccional estricta donde `Pedido` conoce a `Envío` pero no viceversa. Decidimos hacer esta relación obligatoria (`envio_id NOT NULL`) porque en el contexto de negocio, todo pedido debe tener un método de envío asociado.

### **Garantías de la relación:**

- Constraint `NOT NULL` en `envio_id`: Garantiza que todo pedido tiene envío
- Constraint `UNIQUE` en `envio_id`: Garantiza que un envío solo puede estar en un pedido
- `ON DELETE RESTRICT`: Impide eliminar envíos con pedidos asociados
- Validaciones en Service: Verifican que el envío exista y no esté ya usado

Elegimos mantener la unidireccionalidad (tabla `envio` no tiene `pedido_id`) para simplificar el modelo y porque el caso de uso no requiere navegar desde `Envío` hacia `Pedido`. Esta decisión también facilita la creación de envíos independientes que posteriormente pueden asignarse a pedidos.

## Soft Delete (Eliminación Lógica)

Optamos por implementar soft delete en lugar de eliminación física para preservar el historial de transacciones. Agregamos el campo eliminado (boolean) en ambas tablas, que se marca como TRUE cuando se "elimina" un registro.

Esta decisión nos permite:

- Mantener integridad referencial histórica
- Realizar auditorías y análisis de datos históricos
- Recuperar registros eliminados accidentalmente
- Cumplir con regulaciones de conservación de datos

Todas las consultas se filtran automáticamente por `eliminado = 0` para mostrar solo registros activos.

## GESTIÓN DE TRANSACCIONES

### Implementación de TransactionManager

Desarrollamos una clase `TransactionManager` que implementa `AutoCloseable` para gestionar transacciones de forma segura y automática. Esta fue una de las decisiones más importantes del proyecto porque garantiza la atomicidad de operaciones complejas.

Esta implementación nos garantiza que:

- Si olvidamos hacer commit, se ejecuta rollback automáticamente
- Si ocurre una excepción, se revierte toda la transacción
- Los recursos se liberan correctamente mediante try-with-resources

### Operación Transaccional: Crear Pedido con Envío

La funcionalidad más crítica del sistema es la creación de un pedido con su envío asociado en una única transacción atómica. Decidimos implementar un método específico `crearPedidoConEnvio()` que coordina ambas operaciones:

### Flujo de la transacción:

1. **Inicio:** Se obtiene una conexión y se inicia la transacción
2. **Crear Envío:** Se inserta el envío y se obtiene su ID autogenerado
3. **Crear Pedido:** Se inserta el pedido usando el envio\_id obtenido
4. **Commit:** Si ambas operaciones son exitosas, se confirma la transacción
5. **Rollback:** Si cualquier operación falla, se revierten ambas inserciones

### Casos de prueba realizados:

- **Caso exitoso:** Crear pedido y envío con datos válidos → Ambos se crean correctamente
- **Tracking duplicado:** Intentar crear envío con tracking existente → Ninguno se crea (rollback)
- **Número duplicado:** Intentar crear pedido con número existente → Ninguno se crea (rollback)
- **Envío ya usado:** Intentar asociar un envío ya usado por otro pedido → Error de validación

Estas pruebas demostraron que nuestro sistema garantiza la atomicidad: o se crean ambas entidades correctamente, o no se crea ninguna.

## VALIDACIONES IMPLEMENTADAS

Implementamos validaciones en múltiples niveles para garantizar la integridad de los datos:

### Validaciones en Base de Datos

#### Constraints definidos:

- PRIMARY KEY: Identificadores únicos autogenerados
- UNIQUE: Tracking de envío y número de pedido
- NOT NULL: Campos obligatorios (fecha, cliente, total, envio\_id)
- CHECK: Validaciones de rango (costo  $\geq 0$ , total  $\geq 0$ )
- FOREIGN KEY: Integridad referencial con ON DELETE RESTRICT
- ENUM: Valores controlados para estados, empresas y tipos

## Validaciones en Service Layer

Decidimos centralizar las validaciones de negocio en la capa de servicios:

### Envío:

- Tracking único (consultando la base de datos)
- Costo no negativo
- Fechas consistentes (fecha\_estimada >= fecha\_despacho si ambas existen)
- Longitud máxima de tracking (40 caracteres)

### Pedido:

- Número único
- Envío obligatorio y válido
- Envío no usado por otro pedido
- Total no negativo
- Nombre de cliente no vacío (máximo 120 caracteres)
- Número de pedido no vacío (máximo 20 caracteres)

## Validaciones en UI

En la capa de presentación validamos:

- Formato de fechas (dd/mm/aaaa)
- Formato de números decimales
- Campos no vacíos antes de enviar al Service
- Opciones de menú válidas

Esta estrategia de validación en tres niveles garantiza que los datos inválidos se detecten lo antes posible en el flujo de la aplicación.

## FUNCIONALIDADES PRINCIPALES

### CRUD de Envíos

Implementamos las operaciones completas:

- **Crear:** Validación de tracking único, asignación de ID autogenerado
- **Listar:** Consulta con ORDER BY id, filtrando eliminados
- **Buscar por ID:** Búsqueda directa con validación de existencia
- **Buscar por Tracking:** Búsqueda por campo único indexado
- **Actualizar:** Validación de tracking único excluyendo el registro actual
- **Eliminar:** Soft delete con validación de integridad (no eliminar si tiene pedido asociado)

## CRUD de Pedidos

Operaciones implementadas:

- **Crear con Envío (Transacción):** Operación atómica que crea ambas entidades
- **Listar:** Consulta con LEFT JOIN para cargar envío asociado
- **Buscar por ID:** Incluye datos del envío mediante JOIN
- **Buscar por Número:** Búsqueda por campo único con envío asociado
- **Actualizar:** Validación de número único y envío válido
- **Eliminar:** Soft delete del pedido (el envío queda disponible por RESTRICT)

## Características Destacadas

### Uso de LEFT JOIN:

Decidimos usar LEFT JOIN en las consultas de pedidos para cargar el envío asociado en una sola query, evitando el problema N+1. Implementamos alias en las columnas para diferenciar campos con el mismo nombre entre pedido y envío.

### Búsquedas optimizadas:

Aprovechamos los índices UNIQUE en tracking y número para búsquedas rápidas por estos campos clave.

### Manejo de recursos:

Utilizamos try-with-resources consistentemente para garantizar el cierre automático de Connection, PreparedStatement y ResultSet, evitando fugas de memoria.

## HERRAMIENTAS Y RECURSOS UTILIZADOS

### Tecnologías

- **Java 17:** Lenguaje de programación principal
- **MySQL 8.0:** Sistema de gestión de base de datos
- **JDBC (MySQL Connector/J 8.0.33):** Driver de conexión a base de datos
- **NetBeans IDE:** Entorno de desarrollo integrado

## Herramientas de Desarrollo

- **Git:** Control de versiones
- **GitHub:** Repositorio remoto y colaboración
- **MySQL Workbench:** Diseño y administración de base de datos
- **Draw.io:** Creación de diagramas UML

## Asistencia de Inteligencia Artificial

Durante el desarrollo del proyecto utilizamos asistentes de IA (ChatGPT y Claude) como herramientas de consulta y apoyo en los siguientes aspectos:

- Revisión de sintaxis y mejores prácticas de Java
- Sugerencias para estructurar el patrón DAO y Service Layer
- Resolución de errores específicos de JDBC y transacciones
- Optimización de consultas SQL con JOINS
- Generación de documentación Javadoc
- Revisión de validaciones y casos borde

Es importante aclarar que toda la arquitectura, diseño de base de datos, lógica de negocio e implementación fue desarrollada por nuestro equipo. Las herramientas de IA fueron utilizadas como consultoras técnicas, similar a consultar documentación oficial o Stack Overflow, pero todas las decisiones de diseño y el código final son producto de nuestro trabajo y comprensión de los conceptos de la materia.

## CONCLUSIONES

El desarrollo de este Trabajo Práctico Integrador nos permitió aplicar de manera integral los conceptos aprendidos en Programación 2, consolidando nuestro conocimiento en:

**Aspectos técnicos:**

- Arquitectura en capas y separación de responsabilidades
- Patrones de diseño (DAO, Service Layer, Factory)
- Gestión de transacciones ACID con commit y rollback
- Manejo de relaciones 1→1 en base de datos y código
- Validaciones multi-nivel para integridad de datos
- Uso correcto de recursos JDBC con try-with-resources

### Aprendizajes clave:

1. **Importancia de las transacciones:** Comprendimos que en operaciones que afectan múltiples tablas, las transacciones son fundamentales para mantener la consistencia de datos. El rollback automático previene estados inconsistentes en la base de datos.
2. **Validación en capas:** Aprendimos que las validaciones deben estar distribuidas estratégicamente: constraints en BD para integridad física, validaciones en Service para reglas de negocio, y validaciones en UI para experiencia de usuario.
3. **Soft Delete vs Hard Delete:** Implementar eliminación lógica nos enseñó la importancia de preservar datos históricos y mantener integridad referencial en sistemas reales.
4. **Separación de responsabilidades:** La arquitectura en capas facilitó enormemente el mantenimiento del código y la detección de errores, cada clase tiene una responsabilidad clara y única.

### Desafíos enfrentados:

- **Gestión de transacciones:** Inicialmente tuvimos problemas con rollbacks duplicados, que resolvimos implementando el patrón AutoCloseable en TransactionManager.
- **Mapeo de ResultSet con JOIN:** Enfrentamos conflictos de nombres de columnas en los JOINS, que solucionamos usando alias (p.id, e.id).
- **Commits no guardados en MySQL Workbench:** Durante las pruebas iniciales, descubrimos que los datos insertados manualmente en MySQL Workbench no eran visibles desde la aplicación Java. Investigando, identificamos como desactivada la opción de AutoCommit en MySQL Workbench al crear la base de datos y cargar los datos de prueba. Por lo que nuestros INSERTs estaban en una transacción pendiente sin confirmar. Al ejecutar COMMIT manualmente en Workbench, los datos persistieron correctamente y se volvieron visibles para la aplicación. Este desafío

nos enseñó sobre niveles de aislamiento transaccional y la importancia de confirmar las transacciones explícitamente cuando AutoCommit está desactivado.

## REFERENCIAS

- Oracle Java Documentation. (2024). *JDBC API Documentation*.  
<https://docs.oracle.com/en/java/javase/17/docs/api/java.sql/>
- MySQL Documentation. (2024). *MySQL 8.0 Reference Manual*.  
<https://dev.mysql.com/doc/refman/8.0/en/>
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.
- Material de la cátedra: Programación 2, Universidad Tecnológica Nacional.
- OpenAI ChatGPT y Anthropic Claude: Consultas técnicas sobre sintaxis Java, JDBC y mejores prácticas (2025).

## ANEXO

- [https://github.com/MatiasEzequielVazquez/app\\_pedido\\_envio](https://github.com/MatiasEzequielVazquez/app_pedido_envio)