MAURO EZEQUIEL PONCE

Trabajo Final Integrador - Programación II

Sistema de Gestión de Pedidos y Envíos

1. Introducción

Para este Trabajo Final Integrador de Programación II, me propuse un desafío práctico: desarrollar un sistema simple de gestión de Pedidos y Envíos. El objetivo principal era aplicar e integrar los conceptos clave que vimos en la materia.

Para lograrlo, el proyecto combina:

- **Programación en Java** como lenguaje principal.
- **SQL Server** como motor de base de datos relacional.
- **JDBC** (el driver oficial de Microsoft) para realizar la conexión.
- Un **diseño en capas** y un manejo explícito de **transacciones**.

El resultado es una aplicación de consola organizada bajo una arquitectura clara (entidades, DAO, servicios y presentación). Esta separación de responsabilidades no solo facilita la mantenibilidad, sino que también prepara el proyecto para futuras extensiones.

2. Dominio elegido: Pedido → Envío

El núcleo del sistema gira en torno a la relación "uno a uno" entre dos entidades principales:

- **Pedido:** Representa la orden que realiza un cliente.
- **Envío:** Es el despacho físico asociado directamente a ese pedido.

Conceptualmente, cada pedido puede tener, como máximo, un envío, y cada envío pertenece a un único pedido. Esta relación se modela tanto en UML como en la base de datos, donde la tabla Pedido guarda una referencia (clave foránea) al ID del Envio.

Elegir este dominio me permitió aplicar conceptos clave de integridad referencial, claves foráneas y restricciones de unicidad. Además, fue el escenario ideal para demostrar cómo una operación de negocio (como crear un pedido junto con su envío) debe ejecutarse de forma atómica.

3. Diseño de la base de datos

La base de datos, llamada TPI_Prog2_PedidoEnvio, se construye a partir del script database.sql. Las tablas centrales son:

- **Tabla Envio:** Almacena todos los datos logísticos: código de seguimiento (tracking), empresa de correo, tipo de envío, costo, fechas de despacho y entrega, estado, etc.

- **Tabla Pedido:** Contiene la información de la orden: número de pedido, fecha, nombre del cliente, total y estado.

Ambas tablas incluyen un campo eliminado de tipo BIT para implementar el **borrado lógico**, evitando así la pérdida física de información.

Para implementar la **relación 1 a 1**, la tabla Pedido incluye la columna id_envio. Esta columna no solo es una clave foránea que referencia a Envio(id), sino que además tiene una restricción UNIQUE. Esto garantiza que un mismo envío no pueda ser asignado a más de un pedido, cumpliendo con la cardinalidad requerida.

Adicionalmente, preparé otros scripts SQL para crear un usuario específico (tpi_user), cargar datos de prueba y realizar verificaciones de *rollback* durante el desarrollo.

4. Arquitectura de la aplicación Java

La aplicación sigue una arquitectura en capas bien definida, donde cada componente tiene una responsabilidad única. El código fuente en src está organizado en los siguientes paquetes:

4.1. Capa de configuración (config)

Aquí, la clase DatabaseConnection encapsula todos los detalles de la conexión: la URL de JDBC, el usuario y la contraseña.

Provee un método estático getConnection() que devuelve un objeto Connection listo para ser usado. De esta forma, el resto de la aplicación no necesita conocer los detalles del driver ni las credenciales.

4.2. Capa de modelo (entities)

Este paquete contiene los POJOs (Plain Old Java Objects): Pedido y Envio.

Estas clases son representaciones en Java de las tablas de la base de datos. La clase Pedido mantiene una referencia directa a su objeto Envio, replicando la relación 1 a 1 en el modelo de objetos.

4.3. Capa de acceso a datos (dao y dao.impl)

En dao definí las interfaces PedidoDAO y EnvioDAO, que establecen el "contrato" de las operaciones CRUD (crear, buscar, listar, actualizar y borrado lógico).

Las implementaciones en dao.impl tienen responsabilidades claras:

- Usan PreparedStatement para ejecutar las sentencias SQL, previniendo SQL Injection y facilitando el paso de parámetros.
- Mapean las filas del ResultSet a las instancias de los objetos Pedido y Envio.
- **Importante:** Los DAO no manejan la transacción (no hacen commit ni rollback). Simplemente reciben una Connection por parámetro y la utilizan, delegando el control transaccional a la capa superior.

4.4. Capa de servicio (service y service.impl)

Esta es la capa donde reside la lógica de negocio. La interfaz PedidoService define operaciones de alto nivel, como "crear un pedido completo con su envío".

La implementación PedidoServiceImpl es el **núcleo transaccional** del sistema. El método para crear un pedido sigue estos pasos:

1. Obtiene una conexión de DatabaseConnection.
2. Desactiva el autocommit (setAutoCommit(false)).
3. Invoca al EnvioDAO para insertar el envío y recupera el ID generado.
4. Asocia ese ID al objeto Pedido.
5. Invoca al PedidoDAO para insertar el pedido.
6. Si ambos pasos fueron exitosos, ejecuta commit() para confirmar la transacción.
7. Si ocurre *cualquier* excepción (por ejemplo, una restricción de base de datos), ejecuta rollback() y lanza un error controlado.

Este diseño aplica los principios **ACID** (Atomicidad, Consistencia, Aislamiento y Durabilidad), garantizando que la creación de pedido y envío se comporte como una unidad indivisible: o se completan ambos, o no se persiste ninguno.

4.5. Capa de presentación (main)

Finalmente, en el paquete main, la clase AppMenu ofrece una interfaz de texto simple que utiliza Scanner para leer la entrada del usuario.

Esta capa se limita a mostrar opciones, validar datos básicos (fechas, números) e invocar al PedidoService para ejecutar las operaciones. Está totalmente desacoplada del resto del sistema, por lo que podría reemplazarse fácilmente por una interfaz gráfica o una API REST en el futuro.

5. Manejo de transacciones y teoría aplicada

El manejo de transacciones con JDBC es uno de los pilares del trabajo. Así es como apliqué los principios ACID en la práctica:

- **Atomicidad:** La creación del pedido y su envío es una operación única. Si una de las dos inserciones falla, el rollback() asegura que *ninguno* de los registros se guarde.
- **Consistencia:** Las restricciones definidas en la BBDD (claves primarias, foráneas y UNIQUE) garantizan que cualquier cambio aceptado (con commit) deje a la base en un estado válido.
- **Aislamiento:** Aunque en este proyecto el foco no está en la concurrencia, el uso de transacciones proporciona el marco adecuado para trabajar con múltiples operaciones simultáneas en un escenario real.
- **Durabilidad:** Una vez que se ejecuta commit(), los cambios quedan persistidos en la base de datos, incluso ante fallos posteriores de la aplicación.

La clave de esta implementación fue centralizar la lógica transaccional en la capa de servicio, que actúa como coordinadora de los diferentes DAO implicados.

6. Consideraciones de seguridad y buenas prácticas

Para asegurar la calidad y robustez del código, apliqué varias buenas prácticas relevantes:

- **Uso de PreparedStatement:** En lugar de concatenar cadenas de texto para armar el SQL, lo que reduce el riesgo de inyecciones SQL.
- **Usuario de BBDD específico:** Se creó el usuario tpi_user con los permisos mínimos necesarios, en lugar de utilizar el usuario administrador (sa).
- **Borrado lógico:** El uso del campo eliminado permite "borrar" registros sin perder la información física, facilitando auditorías futuras.
- **Arquitectura en capas:** Fomenta la cohesión interna y el bajo acoplamiento entre componentes, haciendo el sistema más mantenible.

7. Conclusión

Este trabajo me permitió conectar la teoría de Programación II con un caso de uso práctico y completo. Logré construir un sistema funcional que va desde el diseño de la base de datos hasta una aplicación Java robusta que maneja transacciones de forma explícita.

El resultado es un proyecto que demuestra la capacidad de:

- Modelar un dominio de negocio (Pedidos y Envíos) de manera clara.
- Garantizar la integridad de los datos mediante restricciones en la BBDD y lógica de servicio.
- Aplicar la teoría de transacciones ACID en un escenario real.
- Mantener un código organizado, modular y fácilmente extensible.

Todo el material (código fuente, scripts SQL, diagramas y documentación) se encuentra disponible en el repositorio de GitHub del trabajo para su revisión y ejecución.