Trabajo Práctico Integrador
Aplicación Java con relación 1→1 unidireccional + DAO + MySQL

OBJETIVO: Desarrollar una aplicación en Java que modele dos clases relacionadas mediante una asociación unidireccional 1 a 1 (la clase "A" referencia a la clase "B"), persistiendo datos en una base relacional mediante JDBC y el patrón DAO, con operaciones transaccionales (commit/rollback) y menú de consola para CRUD.

1. *INTEGRANTES Y ROLES*

1.1. *Integrantes*

- Agustín Mario Nicolás Lepka – Comisión 13
- Martín Gómez – Comisión 08
- Matías Ezequiel Gonzalez – Comisión 15
- Franco Esteban Herrera González – Comisión 15

1.2. *Asignación de Roles*

- Agustín Mario Nicolás Lepka: Responsable del diseño e implementación de las entidades del dominio (Pedido, Envio) y de la configuración de la conexión a la base de datos mediante JDBC.
- Martín Gómez: Encargado del desarrollo de la capa de acceso a datos (DAO), incluyendo la implementación de operaciones CRUD, consultas específicas y control transaccional utilizando JDBC.
- Matías Ezequiel Gonzalez: Responsable de la lógica de negocio, desarrollando la capa de servicios (Service) que encapsula validaciones, reglas funcionales y operaciones compuestas entre entidades.
- Franco Esteban Herrera González: Encargado de la construcción de la interfaz de usuario por consola, mediante la clase AppMenu, gestionando la interacción con el usuario, el flujo de navegación y la integración con los servicios

2. *ELECCIÓN DEL DOMINIO*

El dominio elegido es Pedido → Envio. Esta elección se fundamenta en la posibilidad de reutilizar y extender la base de datos "Logisticaenvios", previamente desarrollada en el Trabajo Práctico Integrador (TPI) de la asignatura Base de Datos I. Dicha reutilización permite capitalizar el diseño relacional ya establecido, facilitando la integración con nuevas capas de lógica de negocio, validaciones y control transaccional, en el marco del presente proyecto.

3. *DISEÑO DEL MODELO DE DATOS Y DECISIONES CLAVE*

El sistema desarrollado aborda la gestión de pedidos y envíos en una empresa logística, modelando una relación directa entre ambas entidades. Se optó por una relación **uno a uno (1:1)** entre Pedido y Envio, donde cada pedido puede tener como máximo un envío asociado, y cada envío pertenece exclusivamente a un pedido.

3.1. *Relación 1:1 con clave foránea única*

La relación se implementa mediante una clave foránea única (envio_id) en la tabla Pedido, que referencia a la clave primaria id de la tabla Envio. Esta decisión permite:

- Insertar pedidos sin envío (flexibilidad).
- Asociar envíos posteriormente (flujo progresivo).
- Evitar duplicidad de envíos por pedido (consistencia).

No se utilizó clave primaria compartida porque se buscó mantener independencia entre entidades, permitiendo que Envio tenga su propio ciclo de vida.

3.2. Baja lógica

Ambas entidades incluyen el campo eliminado (boolean) para aplicar baja lógica, evitando la eliminación física de registros. Esto permite:

- Preservar trazabilidad histórica.
- Facilitar auditorías y reportes.
- Reutilizar registros si se desea reactivar pedidos o envíos.

3.3. Generación de identificadores únicos

Los campos numero (Pedido) y tracking (Envio) se generan automáticamente en tiempo de ejecución utilizando System.currentTimeMillis(), lo que garantiza:

- Unicidad sin depender de la base de datos.
- Facilidad de trazabilidad en pruebas y reportes.

4. ARQUITECTURA POR CAPAS

El sistema está organizado en cinco paquetes principales, siguiendo una arquitectura por capas que promueve la separación de responsabilidades, la escalabilidad y el mantenimiento.

4.1. Paquetes y responsabilidades:

- **Entities**: define las entidades del dominio (Pedido, Envio, Base) con sus atributos, constructores, getters/setters y métodos auxiliares como getInfoPedido() y getInfoEnvio().
- **Dao**: gestiona el acceso a datos mediante JDBC. Las clases PedidoDao y EnvioDao implementan GenericDAO, que define operaciones CRUD genéricas. Se incluyen métodos transaccionales como insertTx() y consultas específicas como getByTracking() o tieneEnvio().
- **Service**: encapsula la lógica de negocio. PedidoService y EnvioService implementan GenericService y delegan en los DAOs. Se incluye el método compuesto crearPedidoConEnvio() que coordina una transacción entre ambas entidades.
- **Main**: contiene la interfaz de usuario por consola. AppMenu presenta un menú interactivo con opciones CRUD, búsqueda, asociación de envío, y testeo automático. Main delega directamente en AppMenu.
- **Config**: centraliza la configuración de infraestructura. DatabaseConnection gestiona la conexión JDBC a MySQL, encapsulando URL, usuario, contraseña y carga del driver.

4.2. Flujo de ejecución

1. El usuario interactúa con el menú (AppMenu).
2. Se invoca un método del Service correspondiente.
3. El Service delega en el Dao.
4. El Dao accede a la base de datos usando DatabaseConnection.

Este flujo garantiza modularidad, testeo independiente por capa y facilidad de mantenimiento.

5. *PERSISTENCIA Y TRANSACCIONES*

La persistencia de datos en el sistema “Logística de Envíos” se implementa mediante una base de datos relacional MySQL, accedida a través de JDBC desde la capa DAO. Esta sección detalla la estructura de la base de datos, el flujo de operaciones de escritura y lectura, y el manejo de transacciones para garantizar la integridad y consistencia de los datos.

5.1. *Estructura de la base de datos*

El modelo relacional está compuesto por dos tablas principales:

- Pedido: representa una orden de compra realizada por un cliente. Contiene campos como id, numero, fecha, clienteNombre, total, estado, envio_id (clave foránea), y eliminado (baja lógica).
- Envio: representa el envío asociado a un pedido. Incluye id, tracking, empresa, tipo, costo, fechaDespacho, fechaEstimada, estado, y eliminado.

La relación entre ambas tablas es de tipo uno a uno (1:1), implementada mediante una clave foránea única: el campo envio_id en Pedido referencia a id en Envio. Esta clave foránea está definida como UNIQUE, lo que garantiza que un pedido no pueda tener más de un envío y que un envío no pueda ser compartido por múltiples pedidos.
Además, ambos modelos implementan un mecanismo de baja lógica mediante el campo eliminado, lo que permite conservar los registros en la base de datos sin que aparezcan en las consultas activas (SELECT * FROM ... WHERE eliminado = FALSE), facilitando la trazabilidad y el mantenimiento histórico.

5.2. *Acceso a datos y operaciones persistentes*

El acceso a la base de datos se realiza a través de las clases PedidoDao y EnvioDao, que implementan la interfaz GenericDAO. Estas clases encapsulan las operaciones CRUD básicas:

- insertar(T entidad)
- actualizar(T entidad)
- eliminar(long id) (baja lógica)
- getById(long id)
- getAll()

Además, se implementan métodos específicos como:

- PedidoDao.getByNumero(String numero)
- PedidoDao.tieneEnvio(long pedidoId, Connection conn)
- EnvioDao.getByTracking(String tracking)
- insertTx(...): métodos sobrecargados para realizar inserciones dentro de una transacción activa.

Estas operaciones utilizan conexiones JDBC obtenidas desde la clase DatabaseConnection, que centraliza la configuración de acceso a la base de datos.

5.3. *Control de transacciones*

El sistema implementa control transaccional explícito para garantizar la atomicidad de operaciones críticas, especialmente aquellas que involucran múltiples entidades. Un

ejemplo clave es el método crearPedidoConEnvio(Pedido pedido, Envio envio) en la clase PedidoService.
Este método realiza las siguientes acciones:

1. Obtiene una conexión a la base de datos y desactiva el autocommit (conn.setAutoCommit(false)).
2. Valida los datos del pedido y del envío (campos obligatorios, valores positivos, fechas coherentes).
3. Verifica si el pedido ya tiene un envío asociado mediante pedidoDao.tieneEnvio(...).
4. Inserta el envío utilizando envioDao.insertTx(envio, conn).
5. Asocia el envío al pedido (pedido.setEnvio(envio)) y lo inserta con pedidoDao.insertTx(pedido, conn).
6. Si todo es exitoso, ejecuta conn.commit() para confirmar los cambios.
7. En caso de error, se captura la excepción, se ejecuta conn.rollback() y se lanza una nueva excepción con el mensaje de error.
8. Finalmente, se restablece el autocommit y se cierra la conexión.
Este enfoque garantiza que ambas operaciones (inserción del envío y del pedido) se realicen como una unidad atómica, cumpliendo con las propiedades ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad):

- Atomicidad: ambas operaciones se completan o ninguna se aplica.
- Consistencia: se respetan las reglas de integridad referencial (por ejemplo, no se permite asociar un pedido a un envío inexistente).
- Aislamiento: al usar una única conexión y desactivar el autocommit, se evita que otras transacciones interfieran durante el proceso.
- Durabilidad: una vez confirmado el commit, los cambios persisten incluso ante fallos del sistema (gracias al uso de logs de transacción de MySQL).

5.4. Consideraciones adicionales

- El uso de métodos insertTx(...) permite reutilizar la lógica de inserción dentro de un contexto transaccional, evitando duplicación de código.
- La validación previa a la transacción evita errores comunes como claves duplicadas, datos nulos o inconsistencias de negocio.
- El diseño permite escalar a operaciones más complejas (por ejemplo, múltiples envíos por pedido) con ajustes mínimos en la lógica transaccional.

6. VALIDACIONES Y REGLAS DE NEGOCIO

La robustez del sistema “Logística de Envíos” se sustenta en un conjunto de validaciones y reglas de negocio que garantizan la integridad de los datos, el cumplimiento de las restricciones funcionales, y la coherencia entre las entidades. Estas validaciones se aplican tanto en la capa de presentación (AppMenu) como en la capa de servicios (PedidoService, EnvioService), asegurando que los datos ingresados por el usuario sean consistentes antes de interactuar con la base de datos.

6.1. Validaciones de entrada

Las validaciones se implementan de forma preventiva, evitando errores comunes como campos vacíos, valores inválidos o inconsistencias lógicas. A continuación se detallan las

principales validaciones aplicadas:
Pedido

- **Número de pedido (numero)**: se genera automáticamente con un prefijo "PED" seguido del timestamp actual (System.currentTimeMillis()), asegurando unicidad sin intervención del usuario.
- **Nombre del cliente (clienteNombre)**: debe ser ingresado por el usuario y no puede estar vacío. Se convierte a mayúsculas para mantener uniformidad.
- **Total del pedido (total)**: debe ser un valor numérico mayor a cero. Se valida con Double.parseDouble(...) y se captura NumberFormatException en caso de error.
- **Estado del pedido (estado)**: debe ser uno de los valores permitidos: NUEVO, FACTURADO, ENVIADO. Se valida mediante comparación explícita en consola, evitando estados inválidos o mal escritos.
- **Envio asociado (envio)**: puede ser null al momento de crear el pedido, pero si se asocia posteriormente, debe cumplir con las validaciones correspondientes.

Envío

- **Tracking (tracking)**: se genera automáticamente con el prefijo "TRK" seguido del timestamp actual. No se permite que esté vacío.
- **Empresa (empresa)**: debe ser una de las opciones válidas: CORREO_ARG, OCA, ANDREANI. Se valida mediante comparación en consola.
- **Tipo de envío (tipo)**: debe ser ESTANDAR o EXPRESS. Se valida de forma similar a la empresa.
- **Costo (costo)**: debe ser un valor numérico positivo. Se valida con Double.parseDouble(...) y control de rango.
- **Fechas (fechaDespacho, fechaEstimada)**: se valida que la fecha de despacho no sea posterior a la fecha estimada, evitando inconsistencias temporales.
- **Estado del envío (estado)**: se asigna automáticamente como EN_PREPARACION al momento de creación.

Estas validaciones se aplican tanto en el flujo interactivo del menú como en el método crearPedidoConEnvio(...), donde se encapsulan dentro de la transacción para evitar que datos inválidos ingresen a la base.

6.2. Reglas de negocio

Además de las validaciones técnicas, el sistema implementa reglas de negocio que reflejan las restricciones funcionales del dominio logístico. Estas reglas aseguran que las operaciones respeten la lógica del negocio y evitan estados inconsistentes.
Reglas aplicadas

- **Un pedido no puede tener más de un envío**: antes de asociar un envío, se verifica mediante pedidoDao.tieneEnvio(...) si el pedido ya tiene uno. Si es así, se lanza una excepción y se cancela la operación.
- **No se permite asociar un envío a un pedido eliminado**: se valida que el campo eliminado sea FALSE antes de realizar actualizaciones.
- **Solo se actualizan pedidos activos**: las operaciones de modificación (actualizar, eliminar) verifican que el pedido esté activo.
- **Estados válidos de pedido**: se restringe el estado a tres valores definidos (NUEVO, FACTURADO, ENVIADO), evitando estados intermedios o inválidos.
- **Control de fechas en envíos**: se impide que la fecha de despacho sea posterior a la

estimada, lo que refleja una regla lógica del negocio de entregas.
- **Baja lógica**: la eliminación de pedidos y envíos no borra físicamente los registros, sino que marca el campo eliminado = TRUE, preservando la trazabilidad.

Estas reglas están codificadas en la capa Service, lo que permite mantener la lógica de negocio separada del acceso a datos y de la interfaz de usuario. Esta separación facilita el mantenimiento, la reutilización y la escalabilidad del sistema.

6.3. Validaciones en la interfaz (AppMenu)

La clase AppMenu complementa las validaciones técnicas con controles interactivos que guían al usuario durante el ingreso de datos:
- Mensajes claros ante errores de formato (NumberFormatException, campos vacíos).
- Confirmaciones antes de eliminar registros.
- Reintentos en búsquedas fallidas (buscarPedidoInteractivo()).
- Validación de opciones mediante bucles while y banderas de control.

Este enfoque mejora la experiencia del usuario, reduce errores operativos y refuerza la integridad del sistema desde el punto de entrada.

7. PRUEBAS REALIZADAS

La validación funcional del sistema “Logística de Envíos” se llevó a cabo mediante pruebas manuales e interactivas, ejecutadas desde la interfaz de usuario por consola (AppMenu). Estas pruebas permitieron verificar el correcto funcionamiento de las operaciones CRUD, la integridad de las transacciones, el cumplimiento de las reglas de negocio, y la respuesta ante errores o entradas inválidas.

7.1. Pruebas de menú interactivo

El sistema presenta un menú principal con 11 opciones funcionales, cada una vinculada a una operación específica. Se realizaron pruebas exhaustivas sobre cada una de ellas:
Opciones de gestión de pedidos
- **Crear pedido sin envío**: se validó la generación automática del número de pedido, la asignación de fecha actual, la validación del total, y la persistencia en base de datos.
- **Listar pedidos**: se verificó que solo se muestren pedidos activos (eliminado = FALSE) y que se presenten con formato legible.
- **Buscar pedido por ID y por número**: se testearon búsquedas exitosas y fallidas, incluyendo reintentos y mensajes de error claros.
- **Actualizar estado de pedido**: se validó la restricción de estados (NUEVO, FACTURADO, ENVIADO) y la persistencia del cambio.
- **Eliminar pedido (baja lógica)**: se confirmó que el pedido no se elimina físicamente, sino que se marca como eliminado, y que no aparece en listados posteriores.

Opciones de gestión de envíos
- **Asociar envío a pedido**: se verificó la validación de empresa, tipo, costo y fechas, así como la restricción de que un pedido no tenga más de un envío.
- **Listar envíos**: se confirmó que se muestran envíos activos con sus datos principales.
- **Buscar envío por ID y por tracking**: se testearon búsquedas exitosas y fallidas, con validación de formato y mensajes adecuados.

Otras funcionalidades
- **Crear pedido con envío (transacción compuesta)**: se validó la inserción simultánea

de ambas entidades, el control de errores, y la confirmación de éxito.
- **Testeo automático**: se ejecutó el método ejecutarTesteoAutomatico() para simular un flujo completo de inserción, recuperación, actualización, listado y eliminación, sin intervención del usuario.

7.2. Testeo automático

El método ejecutarTesteoAutomatico() implementa una secuencia de operaciones que permite verificar el funcionamiento integral del sistema:

1. **Inserción de envío**: se crea un objeto Envio con datos válidos y se persiste en la base.
2. **Inserción de pedido**: se crea un objeto Pedido asociado al envío anterior y se guarda.
3. **Recuperación por ID**: se obtiene el pedido recién insertado y se muestra su información.
4. **Actualización de estado**: se modifica el estado del pedido a FACTURADO y se persiste el cambio.
5. **Listado de pedidos**: se muestra el conjunto de pedidos activos, incluyendo el recién creado.
6. **Eliminación (baja lógica)**: se marca el pedido como eliminado y se verifica que no aparezca en listados posteriores.

Este método permite validar la interacción entre capas (Main → Service → Dao → DB) y garantiza que las operaciones fundamentales se ejecuten correctamente.

7.3. Pruebas de validación y errores

Se realizaron pruebas específicas para verificar el comportamiento ante entradas inválidas o inconsistentes:

- Ingreso de texto en campos numéricos (total, costo) → se captura NumberFormatException y se muestra mensaje de error.
- Ingreso de estados inválidos → se rechaza la entrada y se solicita corrección.
- Asociación de envío a pedido ya vinculado → se lanza excepción y se cancela la operación.
- Fechas de despacho posteriores a la estimada → se rechaza la operación por violación de regla de negocio.
- Confirmaciones de eliminación → se solicita validación explícita del usuario antes de aplicar baja lógica.

Estas pruebas demuestran que el sistema está preparado para manejar errores de forma controlada, evitando caídas y preservando la integridad de los datos.

8. CONCLUSIONES Y MEJORAS FUTURAS

8.1. Conclusiones

El desarrollo del sistema “Logística de Envíos” permitió aplicar de forma integrada conceptos clave de programación orientada a objetos, arquitectura por capas, persistencia con JDBC, validaciones de negocio y control transaccional. A lo largo del proyecto se logró construir una solución funcional, modular y extensible, capaz de gestionar pedidos y envíos con integridad y trazabilidad.

Entre los principales logros se destacan:

- **Diseño relacional sólido**: la relación uno a uno entre Pedido y Envio se implementó correctamente mediante clave foránea única, permitiendo flexibilidad y consistencia.

- **Separación de responsabilidades**: la arquitectura por capas (Entities, Dao, Service, Main, Config) permitió organizar el sistema de forma clara, facilitando el mantenimiento y la escalabilidad.
- **Persistencia segura**: se implementaron operaciones CRUD completas, baja lógica, y control transaccional explícito para garantizar la integridad de los datos.
- **Validaciones robustas**: se aplicaron controles en múltiples capas para evitar errores de entrada, estados inválidos y violaciones de reglas de negocio.
- **Interfaz interactiva funcional**: el menú por consola permite al usuario realizar todas las operaciones de forma guiada, con mensajes claros y confirmaciones.
- **Testeo automatizado**: el método ejecutarTesteoAutomatico() simula un flujo completo de uso, validando la interacción entre capas y la persistencia de datos.

El sistema está preparado para crecer, con interfaces genéricas (GenericDAO, GenericService) que permiten extender la lógica a nuevas entidades sin modificar la estructura base. Además, el uso de clases como Base y métodos reutilizables como insertTx() demuestran un enfoque orientado a la reutilización y la calidad del código.

8.2. Mejoras Futuras

Si bien el sistema cumple con los objetivos planteados, existen oportunidades claras de mejora y expansión que podrían potenciar su funcionalidad y adaptabilidad:

a) Interfaz gráfica o web

Migrar la interfaz de consola a una GUI (Swing, JavaFX) o a una aplicación web (Spring Boot, JSP, REST API) permitiría mejorar la experiencia del usuario, facilitar la navegación, y abrir el sistema a entornos productivos.

b) Paginación y filtros en listados

Agregar paginación, ordenamiento y filtros en los métodos getAll() permitiría manejar grandes volúmenes de datos de forma más eficiente, especialmente en entornos reales con cientos o miles de registros.

c) Pool de conexiones

El uso de un pool de conexiones (como HikariCP o Apache DBCP) permitiría mejorar el rendimiento y la escalabilidad del sistema, especialmente en escenarios concurrentes o multiusuario.

d) Expansión del modelo

El sistema podría extenderse para incluir nuevas entidades como Producto, DetallePedido, Cliente, o Factura, lo que permitiría modelar procesos logísticos más completos y realistas.

9. FUENTES UTILIZADAS

- ***Campus Virtual UTN - https://tup.sied.utn.edu.ar/course/view.php?id=31***
- ***MySQL - https://dev.mysql.com/doc/connector-j/en/connector-j-installing.html***