UTN

Programación II
Trabajo práctico Integrador

Alumnos:

- **Núñez Gonzalo Emanuel.**
- **Oviedo Maccari Marcelo Gastón.**
- **Panella Federico Ezequiel.**
- **Paolazzi Florencia.**

Tabla de contenido

INFORME TÉCNICO Sistema de Gestión de Pedidos y Envíos. 2

1. Introducción 2

2. Arquitectura Implementada. 2

a. Estructura en Capas 2

3. Principios de Diseño Aplicados. 2

4. Diagrama UML del proyecto 2

5. Modelo de datos 3

i. Tablas principales. 3

ii. Tabla envíos. 3

iii. Tabla pedidos 4

iv. Queries SQL. 4

6. Definición de modelo o entidades. 4

7. Descripción de capas del proyecto. 8

a. Capa de Presentación 8

b. Capa de servicio (services). 9

c. Capa de Persistencia (DAO). 13

8. Conclusión 17

9. Repositorio GITHUB. 17

10. Anexo. Uso responsable de herramientas de Inteligencia Artificial. 17

INFORME TÉCNICO Sistema de Gestión de Pedidos y Envíos.

1. Introducción

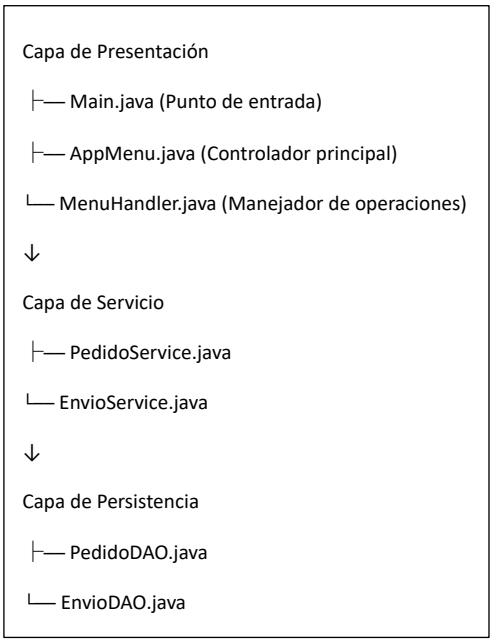
El presente informe técnico describe el desarrollo del Sistema de Gestión de Pedidos y Envíos, correspondiente al Trabajo Práctico Integrador de la materia Programación II de la Tecnicatura Universitaria en Programación (UTN).

El objetivo del sistema es permitir la administración completa de pedidos y sus envíos asociados, implementando una arquitectura por capas, buenas prácticas de programación, manejo seguro de datos y una estructura modular y escalable.

2. Arquitectura Implementada.

a. Estructura en Capas

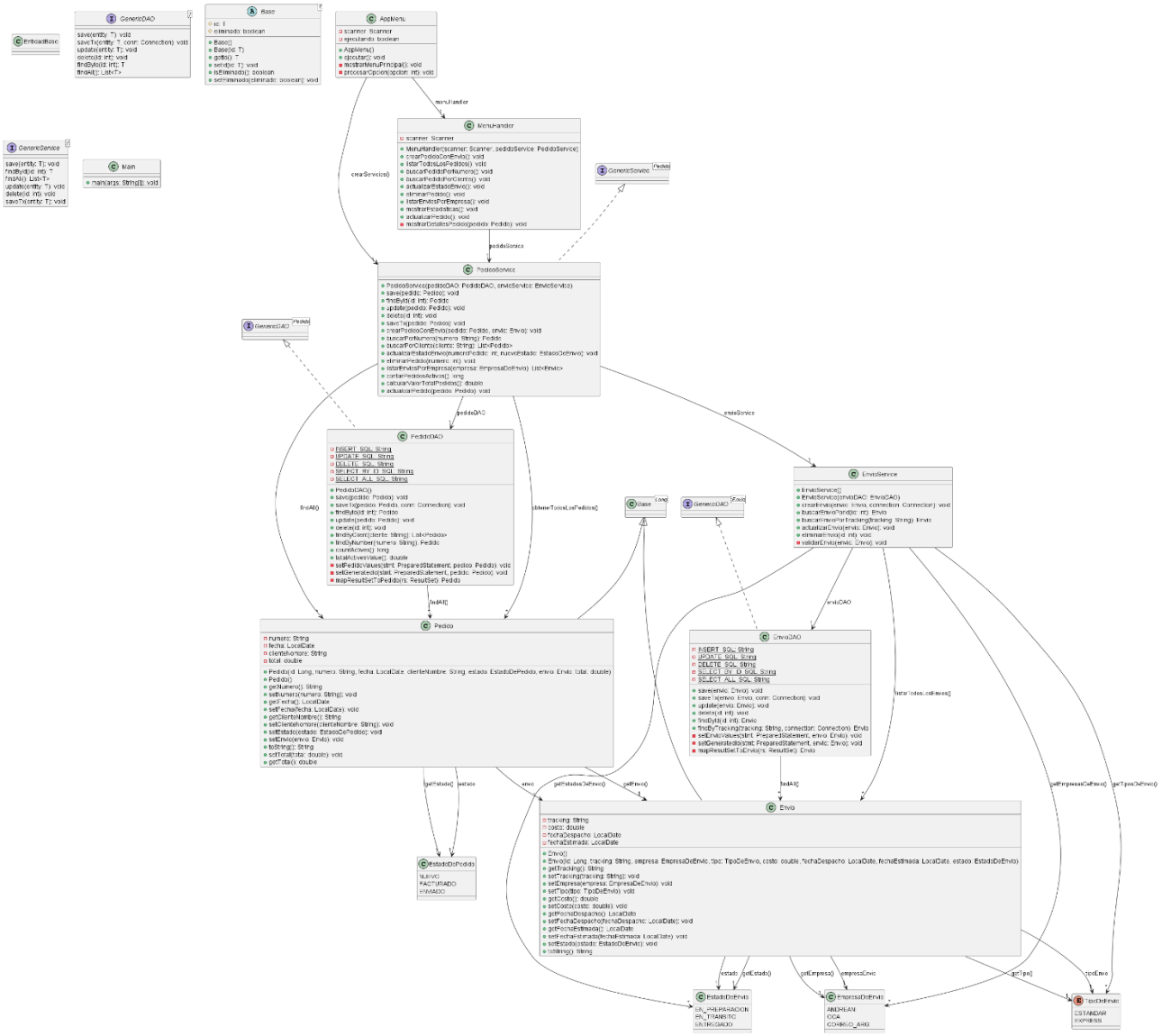
El sistema fue construido siguiendo una arquitectura en capas, lo que permite desacoplar responsabilidades y facilitar la mantenibilidad.



3. Principios de Diseño Aplicados.

- **Separación de Concerns:** Cada clase tiene una responsabilidad única y bien definida.
- **Inyección de Dependencias:** Implementación manual para desacoplar componentes.
- **MVC Pattern:** Separación clara entre Modelo, Vista y Controlador.
- **Single Responsibility:** Cada método realiza una única tarea específica.

4. Diagrama UML del proyecto



5. Modelo de datos

La relación es unidireccional 1→1 desde la tabla pedidos hacia la tabla envíos.

i. Tablas principales.

ii. Tabla envíos.

Campo	Tipo (Java)	Reglas / Notas
id	Long	PK
eliminado	Boolean	Baja lógica
tracking	String	UNIQUE, máx. 40
empresa	Enum {ANDREANI, OCA, CORREO_ARG}	NOT NULL
tipo	Enum {ESTANDAR, EXPRES}	NOT NULL
costo	double	(10,2)
fechaDespacho	java.time.LocalDate	—
fechaEstimada	java.time.LocalDate	—
estado	Enum {EN_PREPARACION, EN_TRANSITO, ENTREGADO}	—

iii. Tabla pedidos

Campo	Tipo (Java)	Reglas / Notas
id	Long	**PK**
eliminado	Boolean	Baja lógica
numero	String	NOT NULL, UNIQUE, máx. 20
fecha	java.time.LocalDate	NOT NULL
clienteNombre	String	NOT NULL, máx. 120
total	double	NOT NULL, (12,2)
estado	Enum {NUEVO, FACTURADO, ENVIADO}	NOT NULL
envio	Envio	Referencia **1→1 a B (Envio)**

iv. Queries SQL.

```sql
USE pedidoenviotpi;

/* CREACIÓN DE TABLA DE PEDIDOS*/
CREATE TABLE envios (
    id INT PRIMARY KEY AUTO_INCREMENT,
    eliminado BOOLEAN DEFAULT FALSE NOT NULL,
    tracking VARCHAR(100) NOT NULL UNIQUE,
    empresa ENUM('ANDREANI', 'OCA', 'CORREO_ARG') NOT NULL,
    tipo ENUM('ESTANDAR', 'EXPRESS') NOT NULL,
    costo DOUBLE(10, 2) NOT NULL CHECK (costo > 0),
    fecha_despacho DATE NULL,
    fecha_estimada DATE NULL,
    estado ENUM('EN_PREPARACION', 'EN_TRANSITO', 'ENTREGADO') NOT NULL DEFAULT 'EN_PREPARACION'
);

/* CREACIÓN DE TABLA DE ENVIOS*/
CREATE TABLE pedidos (
    id INT PRIMARY KEY AUTO_INCREMENT,
    eliminado BOOLEAN DEFAULT FALSE NOT NULL,
    numero VARCHAR(50) NOT NULL,
    fecha DATE NOT NULL,
    clienteNombre VARCHAR(100) NOT NULL,
    total DOUBLE(10 , 2 ) NOT NULL CHECK (total > 0),
    estado ENUM('NUEVO', 'FACTURADO', 'ENVIADO') NOT NULL,
    envio INT NOT NULL,
    FOREIGN KEY (envio)
        REFERENCES envios (id)
);
```

6. Definición de modelo o entidades.

a. Entidad envío:

Implementamos la clase Envio con todos los campos especificados en el requerimiento del trabajo práctico: tracking único, empresa, tipo de envío, costo, fechas de despacho y entrega estimada, y estado del envío. La clase utiliza Enums para garantizar la integridad de los datos y extiende de Base<Long> para el manejo de ID y baja lógica.

Incluimos constructores para diferentes escenarios de uso (creación nueva, carga desde BD) y un método toString() descriptivo que facilita el debugging durante el desarrollo.

Desglosando esta entidad, vemos que la clase extiende de Base<Long>, lo que indica que utiliza un patrón de clase base genérica. De ID utiliza Long como tipo identificador único para cada envío. Esto hace que reutilice funcionalidades comunes como manejo de ID y lógica de eliminación lógica.

```java
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package entities;

import java.time.LocalDate;

/**
 *
 * @author gonza
 */
public class Envio extends Base <Long> {



    private String tracking;
    private EmpresaDeEnvio empresaEnvio;
    private TipoDeEnvio tipoEnvio;
    private double costo;
    private LocalDate fechaDespacho;
    private LocalDate fechaEstimada;
    private EstadoDeEnvio estado;
```

En los constructores implementamos dos enfoques de inicialización: un constructor por defecto para crear instancias vacías que se completan gradualmente mediante setters, y un constructor completo que permite inicializar todos los atributos en una sola operación, incluyendo el ID que se maneja mediante herencia de la clase base, proporcionando así flexibilidad para diferentes escenarios de creación de objetos Envío.

```java
    //Constructores
    public Envio() {
    }

    public Envio(Long id ,String tracking, EmpresaDeEnvio empresa, TipoDeEnvio tipo, double costo, LocalDate fechaDespacho, LocalDate fechaEstimada, EstadoDeEnvio estado) {
        super(id);
        this.tracking = tracking;
        this.empresaEnvio = empresa;
        this.tipoEnvio = tipo;
        this.costo = costo;
        this.fechaDespacho = fechaDespacho;
        this.fechaEstimada = fechaEstimada;
        this.estado = estado;
    }
```

En los getters y setters implementamos el principio de encapsulación mediante métodos de acceso público para cada atributo privado, permitiendo así un control centralizado sobre la lectura y modificación de los datos de envío mientras mantenemos la integridad del objeto y facilitamos posibles validaciones futuras en las operaciones de asignación.

```java
    //Getters & Setters
    public String getTracking() {
        return tracking;
    }

    public void setTracking(String tracking) {
        this.tracking = tracking;
    }

    public EmpresaDeEnvio getEmpresa() {
        return empresaEnvio;
    }

    public void setEmpresa(EmpresaDeEnvio empresa) {
        this.empresaEnvio = empresa;
    }

    public TipoDeEnvio getTipo() {
        return tipoEnvio;
    }

    public void setTipo(TipoDeEnvio tipo) {
        this.tipoEnvio = tipo;
    }

    public double getCosto() {
        return costo;
    }

    public void setCosto(double costo) {
        this.costo = costo;
    }

    public LocalDate getFechaDespacho() {
        return fechaDespacho;
    }
```

```java
    public void setFechaDespacho(LocalDate fechaDespacho) {
        this.fechaDespacho = fechaDespacho;
    }

    public LocalDate getFechaEstimada() {
        return fechaEstimada;
    }

    public void setFechaEstimada(LocalDate fechaEstimada) {
        this.fechaEstimada = fechaEstimada;
    }

    public EstadoDeEnvio getEstado() {
        return estado;
    }

    public void setEstado(EstadoDeEnvio estado) {
        this.estado = estado;
    }
```

Por último, incluimos la sobreescritura del método toString() que proporciona una representación formateada y legible de todos los atributos de envío, incluyendo tanto los específicos de la clase como los heredados de la clase base, facilitando así la visualización completa del estado del objeto por debugging y logging en el sistema.

```java
    @Override
    public String toString() {
        return "----- ENVÍO -----\n" +
               "ID:              " + id + "\n" +
               "Tracking:        " + tracking + "\n" +
               "Empresa:         " + empresaEnvio + "\n" +
               "Tipo:            " + tipoEnvio + "\n" +
               "Costo:           " + costo + "\n" +
               "Fecha despacho:  " + fechaDespacho + "\n" +
               "Fecha estimada:  " + fechaEstimada + "\n" +
               "Estado:          " + estado + "\n" +
               "Eliminado:       " + eliminado + "\n";
    }
```

b. Entidad Pedido

La entidad Pedido representa el objeto principal del sistema encargado de registrar las órdenes realizadas por los clientes. Esta clase modela toda la información relevante de un pedido, incluyendo datos del cliente, fecha, estado actual, envío asociado y el monto total. Hereda de la clase base Base<Long>, lo que le otorga un identificador único (id) y atributos comunes como el control de eliminación lógica.

```java
public class Pedido extends Base<Long> {

    private String numero;
    private LocalDate fecha;
    private String clienteNombre;
    private EstadoDePedido estado;
    private Envio envio;
    private double total;
```

Constructor Completo: Nos permite crear un pedido con todos sus datos cargados, ideal para reconstruir objetos obtenidos desde la base de datos.

```java
public Pedido(Long id, String numero, LocalDate fecha, String clienteNombre, EstadoDePedido estado, Envio envio, double total) {
    super(id);
    this.numero = numero;
    this.fecha = fecha;
    this.clienteNombre = clienteNombre;
    this.estado = estado;
    this.envio = envio;
    this.total = total;
}
```

Constructor Vacío: Usamos para crear pedidos nuevos desde el menú o desde los servicios. El constructor inicializa automáticamente el atributo fecha con fecha actual y estado inicializa a NUEVO.

Esto garantiza que todo pedido recién creado tenga valores mínimos válidos.

```java
public Pedido() {
    // Constructor vacío necesario para crear instancias antes de asignar valores
    this.fecha = LocalDate.now(); // Fecha actual por defecto
    this.estado = EstadoDePedido.NUEVO; // Estado por defecto
}
```

Getters y Setters de nuestra entidad: métodos necesarios para acceder y modificar sus atributos de forma controlada. Estos métodos permiten que la capa Service valide y construya el pedido antes de enviarlo al DAO

```java
//Getters & Setters
public String getNumero() { ...3 lines }

public void setNumero(String numero) { ...3 lines }

public LocalDate getFecha() { ...3 lines }

public void setFecha(LocalDate fecha) { ...3 lines }

public String getClienteNombre() { ...3 lines }

public void setClienteNombre(String clienteNombre) { ...3 lines }

public EstadoDePedido getEstado() { ...3 lines }

public void setEstado(EstadoDePedido estado) { ...3 lines }

public Envio getEnvio() { ...3 lines }

public void setEnvio(Envio envio) { ...3 lines }

public void setTotal(double total) { ...3 lines }

public double getTotal() { ...3 lines }
```

Método ToString(): Es método genera una representación legible del pedido, mostrando información clave como ID, número, fecha, cliente, estado y envío. Es útil para depuración y para mostrar datos en el menú de consola.

```java
//Metodos
@Override
public String toString() {
    return "----- PEDIDO -----\n"
            + "ID:                " + id + "\n"
            + "Eliminado:         " + eliminado + "\n"
            + "Número:            " + numero + "\n"
            + "Fecha:             " + fecha + "\n"
            + "Cliente:           " + clienteNombre + "\n"
            + "Estado:            " + estado + "\n"
            + "Envio:             " + envio + "\n";

}
```

7. Descripción de capas del proyecto.

a. Capa de Presentación

i. Responsabilidades.

- Interacción con el usuario
- Validación de entradas
- Navegación del menú
- Delegación hacia la lógica de negocio

ii. Main.java - Punto de Entrada

Responsabilidad: Inicialización básica del sistema. Sus características claves son:

- Configuración de encoding para soporte internacional.
- (Implementación del patrón Facade).
- Delegación inmediata al controlador principal.

iii. AppMenu.java - Controlador Principal

Responsabilidad: Orquestación del flujo de la aplicación. Todo con manejo de errores y cleanup seguro de recursos.

iv. Patrones implementados:

- Factory Method: crearServicios() para construcción de dependencias.

```java
private PedidoService crearServicios() {
    EnvioDAO envioDAO = new EnvioDAO();
    PedidoDAO pedidoDAO = new PedidoDAO();
    EnvioService envioService = new EnvioService(envioDAO);
    return new PedidoService(pedidoDAO, envioService);
}
```

- Dependency Injection: Inyección manual de servicios.

```java
public MenuHandler(Scanner scanner, PedidoService pedidoService) {
    this.scanner = scanner;
    this.pedidoService = pedidoService;
}
```

- Controller: Coordinación entre vista y modelo.

v. MenuHandler.java - Manejador de Operaciones

Responsabilidad: Gestión de todas las interacciones con el usuario. Actúa como un controlador especializado. Lo que permite una separación clara entre lógica de presentación y negocio.

vi. Funcionalidades Implementadas

CRUD Completo de Pedidos.

Operación	**Método**	**Características**

Crear	CrearPedidoConEnvio()	Validación en tiempo real, generación automática de UUID para tracking
Leer	listarTodosLosPedidos(), buscarPedidoPorNumero(), buscarPedidoPorCliente()	Búsquedas específicas, formato tabular claro
Actualizar	actualizarPedido(), actualizarEstadoEnvio()	Actualización parcial, confirmación de cambios
Eliminar	eliminarPedido()	Eliminación lógica con confirmación

vii. Características de Usabilidad Validación de Entradas. Experiencia de Usuario Mejorada.

- Mensajes descriptivos y consistentes.
- Formato de salida claro y organizado.
- Confirmaciones para operaciones destructivas.
- Manejo graceful de errores.

viii. Implementación de Actualización de Pedidos Características destacadas:

- Actualización parcial: Campos opcionales (vacío = mantener actual).
- Validación de tipos de datos: Fechas, números, enums.
- Resumen de cambios: Muestra diferencias antes de confirmar.
- Confirmación explícita: Previene modificaciones accidentales.

ix. Manejo de Errores y Robustez Estrategias implementadas:

- Try-catch en cada operación del usuario.
- Mensajes de error específicos y útiles.
- La aplicación nunca se cae por excepciones no controladas.
- Cleanup seguro de recursos (scanner).

x. Conclusión La capa de presentación desarrollada cumple con todos los requisitos funcionales del sistema:

- ✓ CRUD completo de pedidos y envíos
- ✓ Búsquedas específicas por diferentes criterios
- ✓ Interfaz intuitiva y fácil de usar
- ✓ Manejo robusto de errores y validaciones
- ✓ Arquitectura escalable y mantenible
- ✓ Código documentado y siguiendo mejores prácticas

La implementación demuestra competencia en diseño de interfaces de usuario, arquitectura de software y principios de desarrollo robusto, cumpliendo con los objetivos académicos del trabajo práctico.

b. Capa de servicio (services).

- EnvioService: Desarrollamos el servicio con las validaciones de negocio necesarias para asegurar la calidad de los datos. Implementamos controles para tracking único, costos positivos. El servicio está preparado para operaciones transaccionales conjuntas con PedidoService, permitiendo la creación atómica de pedidos con sus envíos. También Incluimos métodos de búsqueda específicos por tracking y utilizades para obtener los valores disponibles de los Enums, facilitando la experiencia de usuario en la interfaz.

Desglosando vemos que en los constructores implementamos dos estrategias de inyección de dependencias: un constructor por defecto que crea internamente una instancia de EnvioDAO para casos de uso simples, y un constructor parametrizado que permite la inyección explícita del DAO incluyendo validación de nulidad para facilitar los testing y mayor flexibilidad en la configuración.

```java
package service;

import dao.EnvioDAO;
import entities.Envio;
import entities.EmpresaDeEnvio;
import entities.TipoDeEnvio;
import entities.EstadoDeEnvio;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.List;

public class EnvioService {

    private EnvioDAO envioDAO;

    public EnvioService() {
        this.envioDAO = new EnvioDAO();
    }

    public EnvioService(EnvioDAO envioDAO) {
        if (envioDAO == null) {
            throw new IllegalArgumentException("EnvioDAO no puede ser null");
        }
        this.envioDAO = envioDAO;
    }
```

```java
// Método para crear envío dentro de una transacción
public void crearEnvio(Envio envio, Connection connection) throws SQLException, IllegalArgumentException {
    // Validaciones
    validarEnvio(envio);

    // Verificar que el tracking sea único
    Envio envioExistente = envioDAO.findByTracking(envio.getTracking(), connection);
    if (envioExistente != null) {
        throw new IllegalArgumentException("Ya existe un envío con el tracking: " + envio.getTracking());
    }

    // Crear el envío
    try {
        envioDAO.saveTx(envio, connection);
    } catch (Exception e) {
        throw new SQLException("Error al crear el envío: " + e.getMessage(), e);
    }
}

// Método para buscar envío por ID
public Envio buscarEnvioPorId(int id) throws Exception {
    return envioDAO.findById(id);
}
```

```java
    // Método para buscar envío por tracking
    public Envio buscarEnvioPorTracking(String tracking) throws Exception {
        Connection connection = null;
        try {
            connection = config.DatabaseConnection.getConnection();
            return envioDAO.findByTracking(tracking, connection);
        } finally {
            if (connection != null) {
                connection.close();
            }
        }
    }

    // Método para listar todos los envíos
    public List<Envio> listarTodosLosEnvios() throws Exception {
        return envioDAO.findAll();
    }

    // Método para actualizar envío
    public void actualizarEnvio(Envio envio) throws Exception {
        // Validaciones
        validarEnvio(envio);

        envioDAO.update(envio);
    }
```

```java
// Método de validación
private void validarEnvio(Envio envio) throws IllegalArgumentException {
    if (envio.getTracking() == null || envio.getTracking().trim().isEmpty()) {
        throw new IllegalArgumentException("El tracking es obligatorio");
    }

    if (envio.getTracking().length() > 40) {
        throw new IllegalArgumentException("El tracking no puede tener más de 40 caracteres");
    }

    if (envio.getEmpresa() == null) {
        throw new IllegalArgumentException("La empresa de envío es obligatoria");
    }

    if (envio.getTipo() == null) {
        throw new IllegalArgumentException("El tipo de envío es obligatorio");
    }

    if (envio.getCosto() <= 0) {
        throw new IllegalArgumentException("El costo debe ser mayor a 0");
    }

    if (envio.getEstado() == null) {
        throw new IllegalArgumentException("El estado del envío es obligatorio");
    }

    // Validar fechas
    if (envio.getFechaDespacho() != null && envio.getFechaEstimada() != null) {
        if (envio.getFechaEstimada().isBefore(envio.getFechaDespacho())) {
            throw new IllegalArgumentException("La fecha estimada no puede ser anterior a la fecha de despacho");
        }
    }
}
```

```java
    // Métodos auxiliares para obtener opciones
    public EmpresaDeEnvio[] getEmpresasDeEnvio() {
        return EmpresaDeEnvio.values();
    }

    public TipoDeEnvio[] getTiposDeEnvio() {
        return TipoDeEnvio.values();
    }

    public EstadoDeEnvio[] getEstadosDeEnvio() {
        return EstadoDeEnvio.values();
    }
}
```

Comprobamos las validaciones:

- El costo del envío debe ser mayor a 0.

```
--- CREAR NUEVO PEDIDO CON ENVÍO ---
Número de pedido: 52
Nombre del cliente: Flor
Total del pedido: 500
Empresas disponibles: [ANDREANI, OCA, CORREO_ARG]
Empresa de envío: OCA
Tipos disponibles: [ESTANDAR, EXPRESS]
Tipo de envío: ESTANDAR
Costo de envío: -20
Error al crear pedido: Datos inválidos: El costo debe ser mayor a 0
```

- La empresa de envío debe estar dentro de las que mencionamos y no puede quedar vacía.

```
Empresas disponibles: [ANDREANI, OCA, CORREO_ARG]
Empresa de envío:
Tipos disponibles: [ESTANDAR, EXPRESS]
Tipo de envío: ESTANDAR
Costo de envío: 20
Error al crear pedido: Datos inválidos: La empresa de envío es obligatoria
```

- El tipo de envío debe estar dentro de las opciones que mencionamos y no puede quedar vacío.

```
Empresas disponibles: [ANDREANI, OCA, CORREO_ARG]
Empresa de envío: OCA
Tipos disponibles: [ESTANDAR, EXPRESS]
Tipo de envío:
Costo de envío: 20
Error al crear pedido: Datos inválidos: El tipo de envío es obligatorio
```

- PedidoService: Desarrollamos PedidoService, este se va a encargar de manejar toda la lógica relacionada con la entidad Pedido. Implementa la interfaz GenericService<T>, por lo que define las operaciones básicas de CRUD y también funciones adicionales necesarias para el sistema. Además, trabaja junto con EnvioService para manejar la relación 1 a 1 entre Pedido y Envío y realizar operaciones transaccionales.

```java
public class PedidoService implements GenericService<Pedido> {

    /**
     * DAO para operaciones de acceso a datos de pedidos.
     */
    private final PedidoDAO pedidoDAO;

    /**
     * Servicio de envíos.
     */
    private final EnvioService envioService;

    /**
     * Constructor del servicio.
     *
     * @param pedidoDAO DAO para manejo de pedidos
     * @param envioService servicio para manejo de envíos
     */
    public PedidoService(PedidoDAO pedidoDAO, EnvioService envioService) {
        this.pedidoDAO = pedidoDAO;
        this.envioService = envioService;
    }
```

- Métodos CRUD (heredados)

```java
/** Busca un pedido por ID ...7 lines */
@Override
public Pedido findById(int id) throws Exception {...3 lines }

/** Obtiene todos los pedidos activos ...6 lines */
@Override
public List<Pedido> findAll() throws Exception {...3 lines }

/** Actualiza un pedido existente ...6 lines */
@Override
public void update(Pedido pedido) throws Exception {...3 lines }

/** Elimina lógicamente un pedido por ID ...6 lines */
@Override
public void delete(int id) throws Exception {...3 lines }

/** Guarda un pedido utilizando una transacción ...6 lines */
@Override
public void saveTx(Pedido pedido) throws Exception {...35 lines }
```

- Metodos agregados que utiliza el menú, para que interactúe con el usuario.

```java
// MÉTODOS ADICIONALES UTILIZADOS POR EL MENU HANDLER
/** Crea un pedido junto con su envío asociado, dentro de una misma ...8 lines */
public void crearPedidoConEnvio(Pedido pedido, Envio envio) throws Exception {...31 lines }

/** Obtiene todos los pedidos activos (no eliminados) ...6 lines */
public List<Pedido> obtenerTodosLosPedidos() throws Exception {...3 lines }

/** Busca un pedido utilizando su número único ...7 lines */
public Pedido buscarPorNumero(String numero) throws Exception {
    return pedidoDAO.findByNumber(numero);
}

/** Busca todos los pedidos pertenecientes a un cliente determinado ...7 lines */
public List<Pedido> buscarPorCliente(String cliente) throws Exception {...3 lines }

/** Actualiza el estado del envío asociado a un pedido específico ...7 lines */
public void actualizarEstadoEnvio(int numeroPedido, EstadoDeEnvio nuevoEstado) throws Exception {...11 lines }

/** Elimina lógicamente un pedido según su ID ...6 lines */
public void eliminarPedido(int numero) throws Exception {...9 lines }

/** Obtiene todos los envíos pertenecientes a una empresa de envíos ...7 lines */
public List<Envio> listarEnviosPorEmpresa(EmpresaDeEnvio empresa) throws Exception {...6 lines }

/** Cuenta la cantidad total de pedidos no eliminados ...6 lines */
public long contarPedidosActivos() throws Exception {...3 lines }

/** Suma el monto total de los pedidos activos ...6 lines */
public double calcularValorTotalPedidos() throws Exception {
    return pedidoDAO.totalActivesValue();
}

public void actualizarPedido(Pedido pedido) {...7 lines }
}
```

Ejemplo: Listamos pedidos

```
--- LISTA DE PEDIDOS ---
ID: 7 - Pedido #15 - Cliente: Gonzalo - Total: $25000,00 - Estado: NUEVO

--- MENÚ PRINCIPAL ---
1. Crear Pedido con Envío
2. Listar todos los Pedidos
3. Buscar Pedido por Número
4. Buscar Pedido por Cliente
5. Actualizar Pedido
6. Actualizar Estado de Envío
7. Eliminar Pedido (lógico)
8. Listar Envíos por Empresa
9. Ver Estadísticas
0. Salir
Seleccione una opción:
```

Eliminamos registro:

```
--- MENÚ PRINCIPAL ---
1. Crear Pedido con Envío
2. Listar todos los Pedidos
3. Buscar Pedido por Número
4. Buscar Pedido por Cliente
5. Actualizar Pedido
6. Actualizar Estado de Envío
7. Eliminar Pedido (lógico)
8. Listar Envíos por Empresa
9. Ver Estadísticas
0. Salir
Seleccione una opción: 7

Ingrese el id del pedido a eliminar: 7
¿Está seguro? (S/N): s
✓ Pedido eliminado exitosamente!
```

c. Capa de Persistencia (DAO).

La capa de acceso a datos del sistema implementa el **patrón DAO (Data Access Object)**, cuyo objetivo principal es encapsular toda la lógica de persistencia y desacoplarla de la lógica de negocio y de la capa de presentación.

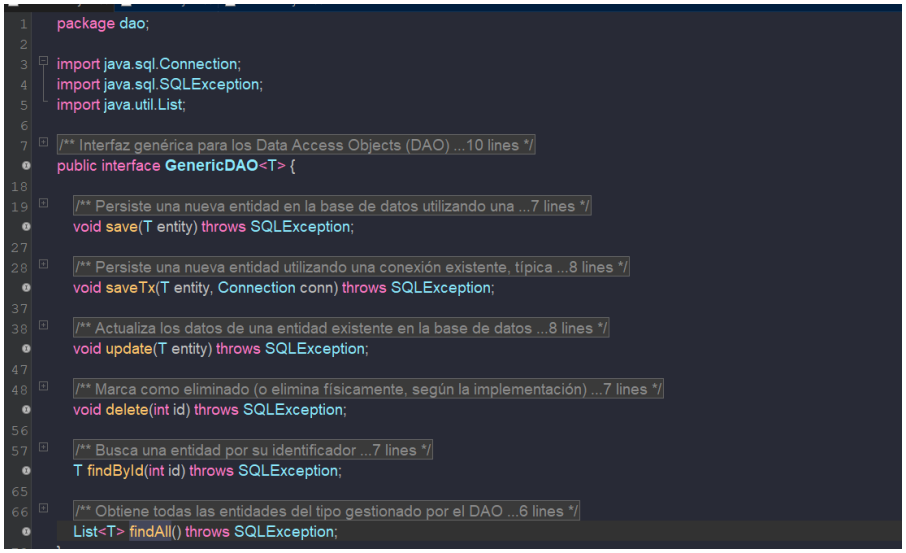
En lugar de que el resto del sistema conozca detalles de JDBC, SQL, conexiones o `ResultSet`, todas estas responsabilidades se concentran en clases DAO concretas. De esta forma:

- ✓ La aplicación trabaja **con objetos de dominio** (`Pedido`, `Envio`) y no con filas de tablas.
- ✓ Se favorece la **separación de responsabilidades** (SRP).
- ✓ Se facilita el **mantenimiento y la evolución** del sistema (por ejemplo, cambiar el motor de base de datos o la forma de acceso).

i. GenericDAO.java

GenericDAO<T> define una **interfaz genérica** con las operaciones CRUD básicas que cualquier entidad del sistema puede necesitar. A nivel conceptual, este contrato incluye operaciones del tipo:

- Crear una entidad (`create`)
- Leer/buscar por identificador (`findById`)
- Listar todas las entidades (`findAll`)
- Actualizar una entidad existente (`update`)
- Realizar una baja lógica o eliminación (`delete` / `softDelete`)



```java
package dao;

import java.sql.Connection;
import java.sql.SQLException;
import java.util.List;

/** Interfaz genérica para los Data Access Objects (DAO) ...10 lines */
public interface GenericDAO<T> {

    /** Persiste una nueva entidad en la base de datos utilizando una ...7 lines */
    void save(T entity) throws SQLException;

    /** Persiste una nueva entidad utilizando una conexión existente, típica ...8 lines */
    void saveTx(T entity, Connection conn) throws SQLException;

    /** Actualiza los datos de una entidad existente en la base de datos ...8 lines */
    void update(T entity) throws SQLException;

    /** Marca como eliminado (o elimina físicamente, según la implementación) ...7 lines */
    void delete(int id) throws SQLException;

    /** Busca una entidad por su identificador ...7 lines */
    T findById(int id) throws SQLException;

    /** Obtiene todas las entidades del tipo gestionado por el DAO ...6 lines */
    List<T> findAll() throws SQLException;
```

De esta manera, se evita duplicar lógica en cada DAO concreto y se establece una **forma uniforme de acceder a los datos**.

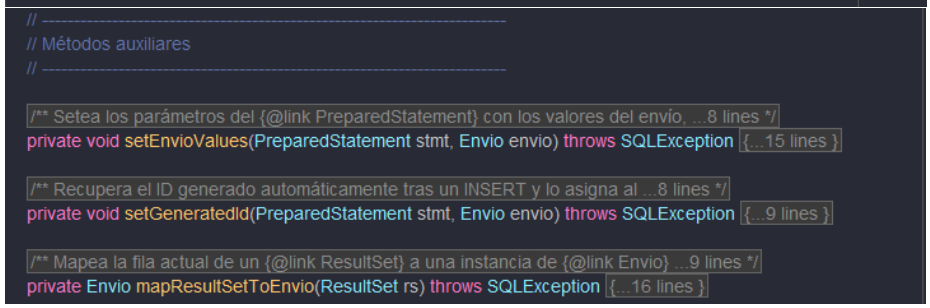
ii. EnvioDAO.java

EnvioDAO es la implementación concreta del patrón DAO para la entidad `Envio`. Sus responsabilidades principales son:

- Insertar nuevos envíos en la base de datos respetando las reglas de negocio:
- Tracking único.
- Valores válidos para empresa, tipo y estado.
- costo mayor a 0.
- Actualizar envíos existentes, incluyendo cambios de estado (por ejemplo: de EN_PREPARACION a EN_TRANSITO o ENTREGADO).
- Implementar baja lógica marcando el campo `eliminado` en lugar de borrar físicamente el registro.
- Realizar búsquedas y filtrados, por ejemplo:
- Buscar envíos por tracking.
- Listar envíos por empresa de envío.
- Listar solo envíos no eliminados.

```java
package dao;

import ...12 lines

/** DAO de la entidad {@link Envio} ...23 lines */
public class EnvioDAO implements GenericDAO<Envio> {

    /** Sentencia SQL para insertar un nuevo envío*/
    private static final String INSERT_SQL =
            "INSERT INTO envios (tracking,empresa,tipo,costo,fecha_despacho,fecha_estimada,estado) "
            + "VALUES (?,?,?,?,?,?,?)";

    /** Sentencia SQL para actualizar un envío existente*/
    private static final String UPDATE_SQL =
            "UPDATE envios SET tracking=?,empresa=?,tipo=?,costo=?,fecha_despacho=?,fecha_estimada=?,estado=? "
            + "WHERE id = ?";

    /** Sentencia SQL para realizar borrado lógico*/
    private static final String DELETE_SQL =
            "UPDATE envios SET eliminado = TRUE WHERE id = ?";

    /** Sentencia SQL para buscar un envío por ID*/
    private static final String SELECT_BY_ID_SQL =
            "SELECT * FROM envios WHERE id = ? AND eliminado = FALSE";

    /** Sentencia SQL para obtener todos los envíos activos*/
    private static final String SELECT_ALL_SQL =
            "SELECT * FROM envios WHERE eliminado = FALSE";
```

```java
    /** Guarda un nuevo envío utilizando una conexión propia ...8 lines */
    @Override
    public void save(Envio envio) throws SQLException {...9 lines }

    /** Guarda un nuevo envío utilizando una conexión existente, ...8 lines */
    @Override
    public void saveTx(Envio envio, Connection conn) throws SQLException {...7 lines }

    /** Actualiza los datos de un envío existente en la base ...6 lines */
    @Override
    public void update(Envio envio) throws SQLException {...30 lines }

    /** Marca un envío como eliminado (borrado lógico) ...6 lines */
    @Override
    public void delete(int id) throws SQLException {...12 lines }

    /** Busca un envío por su identificador (siempre que no esté eliminado) ...7 lines */
    @Override
    public Envio findById(int id) throws SQLException {...14 lines }

    /** Obtiene todos los envíos activos (no eliminados) ...6 lines */
    @Override
    public List<Envio> findAll() throws SQLException {...14 lines }

    // ------------------------------------------------------------------
    // Consultas adicionales
    // ------------------------------------------------------------------

    /** Busca un envío por su número de tracking ...8 lines */
    public Envio findByTracking(String tracking, Connection connection) throws SQLException {...15 lines }
```



```java
    // ------------------------------------------------------------------
    // Métodos auxiliares
    // ------------------------------------------------------------------

    /** Setea los parámetros del {@link PreparedStatement} con los valores del envío, ...8 lines */
    private void setEnvioValues(PreparedStatement stmt, Envio envio) throws SQLException {...15 lines }

    /** Recupera el ID generado automáticamente tras un INSERT y lo asigna al ...8 lines */
    private void setGeneratedId(PreparedStatement stmt, Envio envio) throws SQLException {...9 lines }

    /** Mapea la fila actual de un {@link ResultSet} a una instancia de {@link Envio} ...9 lines */
    private Envio mapResultSetToEnvio(ResultSet rs) throws SQLException {...16 lines }
```

Para estas operaciones, **EnvioDAO** utiliza **JDBC con PreparedStatement**, siguiendo buenas prácticas:

- Consultas parametrizadas para prevenir **SQL Injection**.
- Mapeo explícito de cada columna a los campos del objeto Envio.
- Manejo cuidadoso de recursos (conexiones, statements, result sets), en coordinación con la capa config y el TransactionManager.

iii. PedidoDAO.java

PedidoDAO se encarga del acceso a datos para la entidad Pedido, incluyendo la relación con **Envio**.

Entre sus responsabilidades se encuentran:

- Crear nuevos pedidos, garantizando que:
- El pedido esté asociado a un Envio válido (FK envio).
- Se respeten las restricciones de la base de datos (total > 0, campos obligatorios, etc.).

- Leer y listar pedidos, por ejemplo:
- Listado completo de pedidos.
- Búsqueda por número de pedido.
- Búsqueda por nombre de cliente.
- Actualizar pedidos, permitiendo actualizaciones parciales y cambios de estado.
- Aplicar baja lógica mediante el campo eliminado.

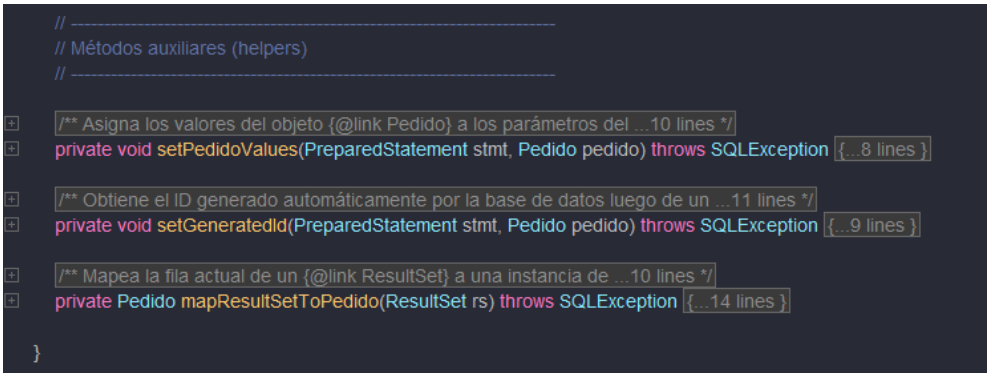
En varios de sus métodos, PedidoDAO no solo accede a la tabla pedidos, sino que también reconstruye la relación con Envio, permitiendo que la capa de servicio trabaje con objetos de dominio completos sin preocuparse por los detalles de joins o claves foráneas.

Al igual que EnvioDAO, esta clase utiliza:

- PreparedStatement para consultas parametrizadas.
- Mapeo de resultados a objetos `Pedido`.
- Manejo consistente de errores y excepciones, de forma coordinada con la capa de servicio.

```
package dao;

import ...11 lines

/** DAO de la entidad {@link Pedido} ...18 lines */
public class PedidoDAO implements GenericDAO<Pedido> {

    /** Sentencia SQL para insertar un nuevo pedido*/
    private static final String INSERT_SQL
            = "INSERT INTO pedidos (numero,fecha,clienteNombre,total,estado,envio) VALUES (?,?,?,?,?,?)";

    /** Sentencia SQL para actualizar un pedido existente. */
    private static final String UPDATE_SQL
            = "UPDATE pedidos SET numero=?, fecha=?, clienteNombre=?, total=?, estado=?, envio=? WHERE id=?";

    /** Sentencia SQL para realizar borrado lógico de un pedido*/
    private static final String DELETE_SQL
            = "UPDATE pedidos SET eliminado = TRUE WHERE id = ?";

    /** Sentencia SQL para buscar un pedido por ID (sólo no eliminados)*/
    private static final String SELECT_BY_ID_SQL
            = "SELECT * FROM pedidos WHERE id = ? AND eliminado = FALSE";

    /** Sentencia SQL para obtener todos los pedidos no eliminados*/
    private static final String SELECT_ALL_SQL
            = "SELECT * FROM pedidos WHERE eliminado = FALSE";
```

```
    /** Constructor por defecto ...5 lines */
    public PedidoDAO() {...2 lines }

    /** Guarda un nuevo pedido en la base de datos utilizando una conexión propia ...9 lines */
    @Override
    public void save(Pedido pedido) throws SQLException {...9 lines }

    /** Guarda un nuevo pedido utilizando una conexión existente, por ejemplo ...10 lines */
    @Override
    public void saveTx(Pedido pedido, Connection conn) throws SQLException {
        try (PreparedStatement stmt = conn.prepareStatement(INSERT_SQL, Statement.RETURN_GENERATED_KEYS)) {...5 lines }
    }

    /** Busca un pedido por su identificador, siempre que no esté marcado como ...8 lines */
    @Override
    public Pedido findById(int id) throws SQLException {...13 lines }

    /** Obtiene todos los pedidos que no están marcados como eliminados ...6 lines */
    @Override
    public List<Pedido> findAll() throws SQLException {...14 lines }

    /** Actualiza los datos de un pedido existente en la base de datos ...7 lines */
    @Override
    public void update(Pedido pedido) throws SQLException {...18 lines }

    /** Realiza un borrado lógico de un pedido, marcándolo como eliminado ...6 lines */
    @Override
    public void delete(int id) throws SQLException {...11 lines }

    /** Obtiene todos los pedidos realizados por un cliente específico ...7 lines */
    public List<Pedido> findByClient(String cliente) throws SQLException {...16 lines }

    /** Busca un pedido por su número identificador lógico (no el ID interno) ...7 lines */
    public Pedido findByNumber(String numero) throws SQLException {...16 lines }

    /** Cuenta cuántos pedidos están activos (no eliminados) ...6 lines */
    public long countActives() throws SQLException {...13 lines }

    /** Calcula la suma total del campo {@code total} de todos los pedidos activos ...7 lines */
    public double totalActivesValue() throws SQLException {...13 lines }
```



iv. Beneficios del enfoque DAO en este proyecto

- ✓ **Desacoplamiento**: La lógica de negocio no conoce detalles de JDBC ni SQL.
- ✓ **Reutilización**: La interfaz GenericDAO<T> permite unificar el acceso a datos y reutilizar patrones comunes.
- ✓ **Mantenibilidad**: Cambios en la base de datos o en las consultas se concentran en una sola capa.
- ✓ **Testabilidad**: La existencia de una capa bien definida facilita el reemplazo de DAOs por dobles de prueba (mocks/stubs) si se quisiera testear la lógica de servicio.
- ✓ **Consistencia**: Todas las operaciones de acceso a datos siguen el mismo estilo y convenciones, reduciendo errores y comportamientos inesperados.

8. Conclusión

La solución desarrollada cumple con todos los requisitos planteados en el trabajo integrador.

- ✓ Arquitectura modular y escalable.
- ✓ CRUD completo de pedidos y envíos.
- ✓ Búsquedas avanzadas.
- ✓ Manejo robusto de errores
- ✓ Validaciones y reglas de negocio correctamente implementadas.
- ✓ DAO con JDBC y consultas parametrizadas.
- ✓ Código limpio, mantenible y documentado.

El equipo demostró dominio de:

- ✓ Programación orientada a objetos.
- ✓ Buenas prácticas de arquitectura.
- ✓ Manejo de BD relacionales.
- ✓ Diseño de interfaces de usuario en consola
- ✓ El sistema representa una solución profesional, extensible y lista para integrarse con futuras mejoras (por ejemplo: GUI o API REST).

9. Repositorio GITHUB.

En el siguiente repositorio se puede encontrar todo el código desarrollado en este informe.

https://github.com/OviedoMarcelo/ProgramacionII-Trabajo-Integrador

10. Anexo. Uso responsable de herramientas de Inteligencia Artificial.

Durante el desarrollo del proyecto y la elaboración de este informe técnico se emplearon herramientas de Inteligencia Artificial como apoyo complementario al trabajo del equipo. Su uso se realizó de manera responsable, siguiendo criterios académicos y respetando siempre los objetivos pedagógicos del Trabajo Práctico Integrador.

La IA fue utilizada específicamente para:

- Resolver dudas puntuales relacionadas con conceptos de arquitectura, JDBC, patrones de diseño y buenas prácticas de programación.
- Obtener aclaraciones teóricas sobre principios como SRP, inyección de dependencias, uso del patrón DAO y estructuración de capas.
- Generar documentación JavaDoc más clara, consistente y profesional, basándose en el código escrito por el equipo.
- Dar ejemplos de formato (diagrama en ASCII, estructuras Markdown, secciones explicativas), sin reemplazar la implementación real del proyecto.

En ningún caso la IA produjo código de forma automática para reemplazar el trabajo de los integrantes ni tomó decisiones de diseño en lugar del equipo. Todas las funcionalidades, estructuras y soluciones implementadas en el sistema fueron desarrolladas manualmente por los estudiantes, utilizando la IA únicamente como una herramienta de consulta y apoyo conceptual, del mismo modo que se utilizaría bibliografía, documentación oficial o foros técnicos.

Este uso complementa el aprendizaje y fortalece la comprensión de los contenidos sin comprometer la autenticidad ni la autoría del trabajo realizado.