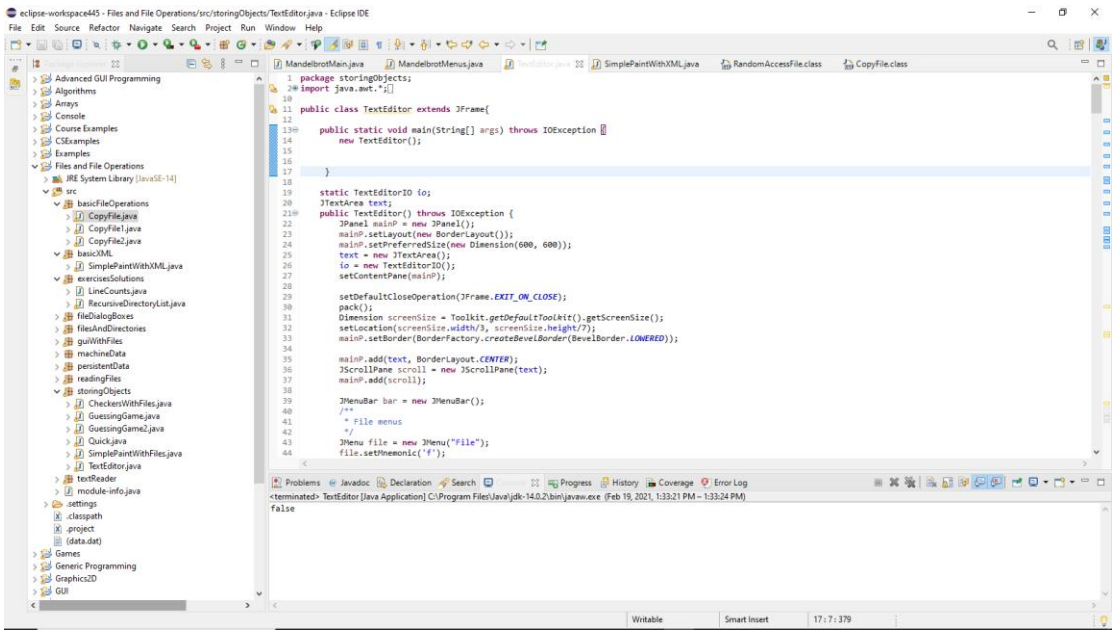
Trabajo Final Integrador (TFI)
Programación II



Profesores: Cinthia Rigoni | Flor Gubiotti | Ariel Enferrel

Alumnos:
Rotolo Martin: Diseño de Arquitectura, Definición de Entidades y Base de Datos | Wolanink Melany Implementación de Servicios, Transacciones y Lógica de Negocio

Tablas utilizadas:

- seguro_vehicular

- vehiculo
  Con relación 1 a 1 opcional entre vehículo y seguro

1) Diseño

Dominio Elegido
Se seleccionó el dominio "Vehiculo → SeguroVehicular". En este modelo:
Clase A: Vehiculo
Clase B: SeguroVehicular

Paquetes y clases
Paquetes y Responsabilidades
Se utilizó una arquitectura en capas para desacoplar responsabilidades:

- config: Contiene DatabaseConnection para la conexión JDBC.
- entities: Contiene los modelos de dominio (Vehiculo, SeguroVehicular).
- dao: Interfaz GenericDAO<T> e implementaciones (VehiculoDAO, SeguroVehicularDAO).
- service: Lógica de negocio y transacciones(GenericService<T>, VehiculoService, SeguroVehicularService)
- main: Punto de entrada y menú (MainApp, AppMenu).

Clases:
Clase Vehiculo (A)
Paquete: entities
Visibilidad de la clase: pública.
Atributos (todos privados):
Long id
String dominio
String marca
String modelo
Integer anio
Boolean eliminado
String nroChasis
SeguroVehicular seguro (esta es la referencia 1→1 a B)

Constructores:
Constructor vacío (sin parámetros).
Constructor con todos los atributos, incluyendo el SeguroVehicular:
Vehiculo(Long id, String dominio, String marca, String modelo, Integer anio, Boolean eliminado, SeguroVehicular seguro)

Métodos:
Getters y setters para todos los atributos.
toString() que devuelva un texto legible, con los datos principales del vehículo y, opcionalmente, un resumen del seguro.

Relación:
La clase Vehiculo tiene un atributo privado de tipo SeguroVehicular. Esa es la asociación unidireccional 1 a 1: desde Vehiculo hacia SeguroVehicular.

Clase SeguroVehicular (B)
Paquete: entities
Visibilidad de la clase: pública.
Atributos (todos privados):
Long id
Boolean eliminado
String aseguradora
String nroPoliza

CoberturaEnum cobertura
java.time.LocalDate vencimiento
B no tiene una referencia al Vehiculo en la clase Java. La relación es unidireccional desde A hacia B.

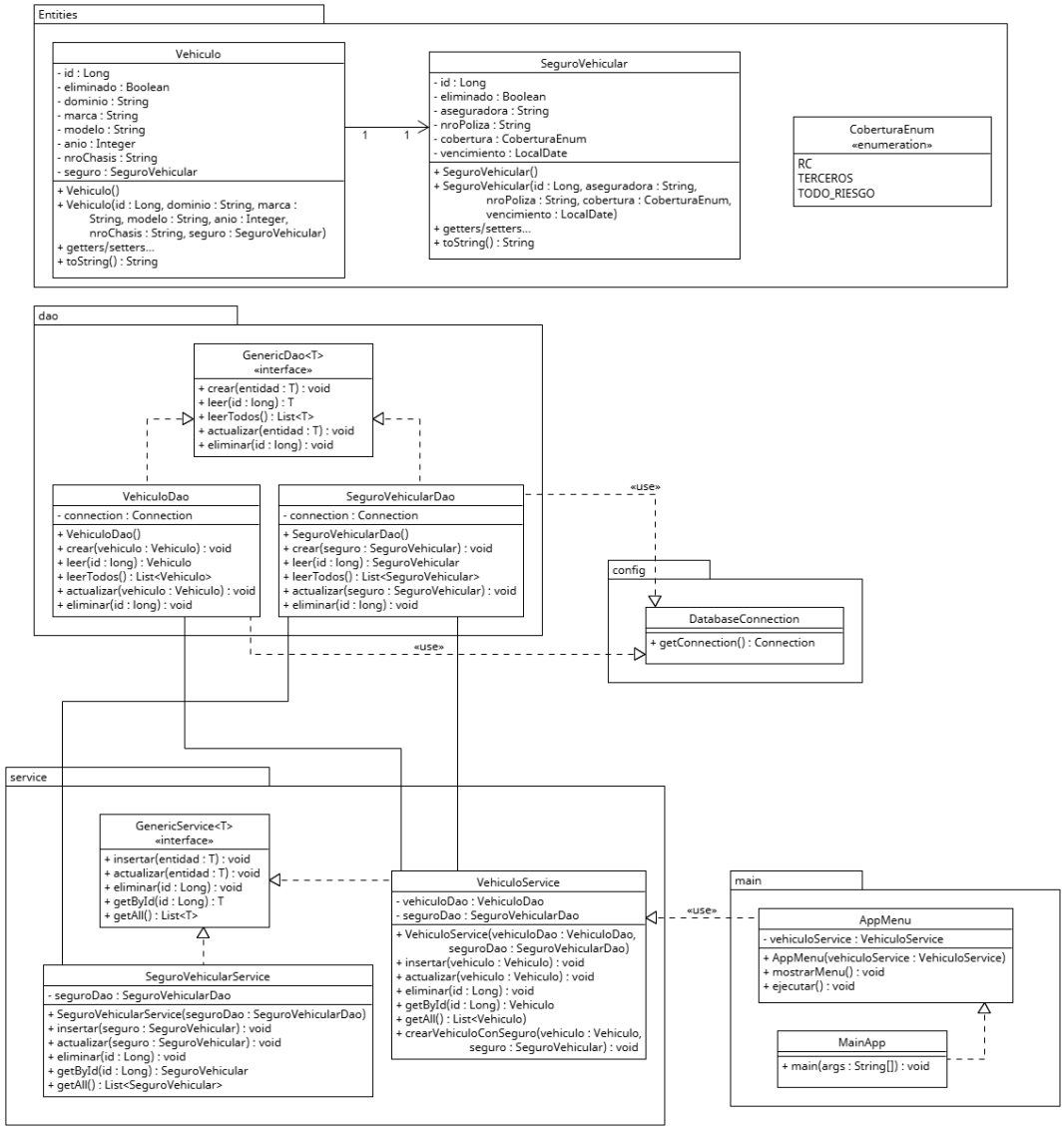
Constructores:
Constructor vacío.
Constructor con todos los atributos:
SeguroVehicular(Long id, Boolean eliminado, String aseguradora, String nroPoliza, CoberturaEnum cobertura, java.time.LocalDate vencimiento)

Métodos:
Getters y setters para todos los atributos.
toString() legible.

Clase DatabaseConnection
Paquete: config
Responsabilidad: centralizar la conexión JDBC a MySQL.
Atributos (privados y estáticos, opcional):
Ruta al archivo de propiedades (por ejemplo db.properties) donde se guardan url, user, password.

Métodos:
public static Connection getConnection() throws SQLException
Lee las propiedades (driver, url, usuario, contraseña).
Crea y retorna un objeto java.sql.Connection.

Paquetes y dependencias principales
Main depende de service y de entities (para mostrar datos por consola).
Service depende de dao, entities y config (porque usa las conexiones).
Dao depende de entities y config.
Entities no depende de los otros paquetes (solo de tipos básicos y clases estándar de Java).
Config solo usa clases estándar de Java (Properties, Connection, etc.).
Esto cumple con la separación por capas: presentación (consola) → service → DAO → base de datos.



2) Entidades (genérico)

Las clases del paquete entities/ se implementaron como POJOs (Plain Old Java Objects) cumpliendo los siguientes requisitos:

Ambas clases (Vehiculo y SeguroVehicular) incluyen los campos id (Long) y eliminado (Boolean) para la baja lógica.

Se proveen constructores (vacío y completo), métodos getters y setters para todos los atributos, y un método toString() legible para facilitar la depuración y la visualización en consola.

La relación se materializa con el atributo private SeguroVehicular seguro; dentro de la clase Vehiculo.

3) Base de datos (MySQL)

La persistencia se realiza en una base de datos MySQL, diseñada para ser consistente con las entidades.

Conexión a la Base de Datos
La conexión se centraliza en la clase config.DatabaseConnection, que utiliza un método estático getConnection(). Los datos de acceso (URL, usuario, contraseña) se leen desde un archivo externo db.properties para evitar hardcodear credenciales en el código fuente.

Estrategia de Relación 1-a-1
Para garantizar la cardinalidad 1-a-1 a nivel de base de datos, se utilizó la estrategia de Clave Foránea Única (UNIQUE FOREIGN KEY), una de las opciones recomendadas.

La tabla vehiculo (A) actúa como tabla principal.
La tabla seguro_vehicular (B) contiene una columna vehiculo_id que es una Clave Foránea (FOREIGN KEY) que referencia a vehiculo(id).
Adicionalmente, se aplica una restricción UNIQUE KEY sobre esta misma columna vehiculo_id.

Esto asegura que un id de vehículo solo pueda aparecer una vez en la tabla seguro_vehicular, implementando la relación 1-a-1 de forma robusta.

Archivos SQL Entregados
Se proporcionan los dos scripts SQL requeridos para la reproducibilidad del proyecto:

01_schema_tfi_seguros.sql: Contiene las sentencias CREATE DATABASE, USE y CREATE TABLE para ambas tablas, incluyendo claves primarias (PK), la clave foránea (FK), las restricciones UNIQUE (para dominio, nro_chasis, nro_poliza y vehiculo_id) y la configuración ON DELETE CASCADE.

02_data_tfi_seguros.sql: Contiene las sentencias INSERT necesarias para poblar la base de datos con datos de ejemplo que permiten probar la aplicación inmediatamente (incluyendo casos de vehículos con y sin seguro).

En el punto 4 se implementó el acceso a datos mediante el patrón **DAO**, siguiendo la arquitectura propuesta y utilizando JDBC con PreparedStatement para garantizar seguridad y eficiencia en las operaciones contra la base de datos.

Primero se definió la interfaz genérica GenericDao<T>, la cual establece las operaciones CRUD básicas que cualquier entidad del sistema debe poseer: crear, leer, listar, actualizar y eliminar (baja lógica en este caso).
Luego se desarrollaron dos implementaciones concretas:

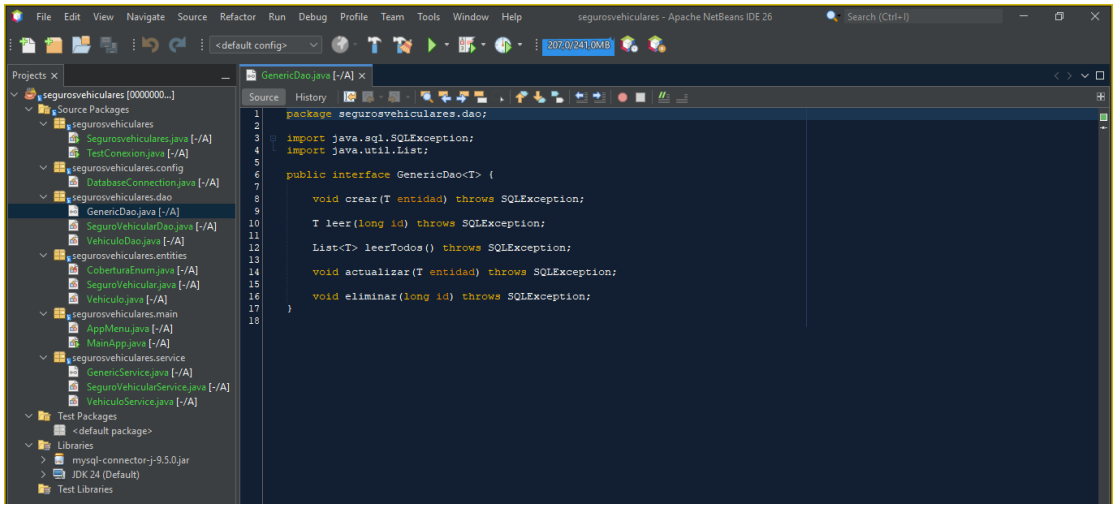
- **SeguroVehicularDao**
- **VehiculoDao**

Ambos DAO encapsulan completamente la lógica de acceso a la base de datos, evitando que las capas superiores conozcan detalles de SQL.
VehiculoDao incluye además el método buscarPorDominio(), requerido por el trabajo práctico para realizar búsquedas por campos relevantes.
Los DAOs están preparados para operar con una conexión externa (Connection

recibida por constructor), lo que permite ejecutar transacciones desde la capa de servicios.

Este diseño promueve la separación de responsabilidades y mantiene el sistema escalable y mantenible.

4. Implementación de los DAO (Data Access Object)



En esta etapa desarrollamos la capa DAO con el objetivo de aislar completamente el acceso a la base de datos del resto de la lógica de negocio. De esta forma, conseguimos un diseño más modular, mantenible y alineado con el patrón DAO solicitado en la consigna del TFI.

4.1. Definición de la interfaz genérica GenericDao<T>

Como primer paso, definimos la interfaz genérica GenericDao<T>, que actúa como contrato común para todos los DAOs del proyecto. En esta interfaz incluimos los métodos necesarios para realizar las operaciones CRUD básicas:

- crear(T entidad)
- leer(long id)
- leerTodos()
- actualizar(T entidad)
- eliminar(long id)

La decisión de crear una interfaz genérica nos permitió:

- Unificar la forma de trabajar con todas las entidades.
- Reducir duplicación de código entre los DAOs concretos.

- Facilitar cambios futuros, ya que si es necesario agregar un nuevo método genérico, se puede hacer desde una única interfaz.

4.2. DAOs concretos: VehiculoDao y SeguroVehicularDao

A partir de GenericDao<T>, implementamos los DAOs concretos para cada entidad del dominio elegido:

- VehiculoDao
- SeguroVehicularDao

En ambos casos, utilizamos JDBC con PreparedStatement para todas las operaciones, cumpliendo con la consigna de no usar ORM.
Para cada método:

- Armamos las sentencias SQL correspondientes (INSERT, SELECT, UPDATE, “baja lógica” con UPDATE).
- Asignamos los parámetros de los PreparedStatement respetando el tipo de dato de cada campo.
- Ejecutamos las consultas y, cuando fue necesario, recorremos el ResultSet para reconstruir las entidades Java.

En la lectura de datos:

- En VehiculoDao, mapeamos cada fila a un objeto Vehiculo.
- En SeguroVehicularDao, mapeamos cada fila a un objeto SeguroVehicular.

Además, implementamos versiones de los métodos DAO que aceptan una Connection externa, lo cual es clave para que varias operaciones participen de una misma transacción (por ejemplo, crear un seguro y el vehículo asociado en una única unidad de trabajo).

4.3. Manejo de la baja lógica y filtrado de registros

En lugar de eliminar registros físicamente, implementamos la baja lógica usando el campo eliminado en ambas tablas:

- El método eliminar(long id) actualiza el campo eliminado a true.
- Los métodos leerTodos() y las consultas de listado filtran registros con eliminado = false, de modo que los elementos dados de baja no aparezcan en el menú de la aplicación.

Esta estrategia nos permitió:

- Mantener un histórico de datos en la base.
- Evitar problemas de integridad referencial al borrar registros que puedan estar relacionados.

4.4. Relación 1→1 en la capa DAO

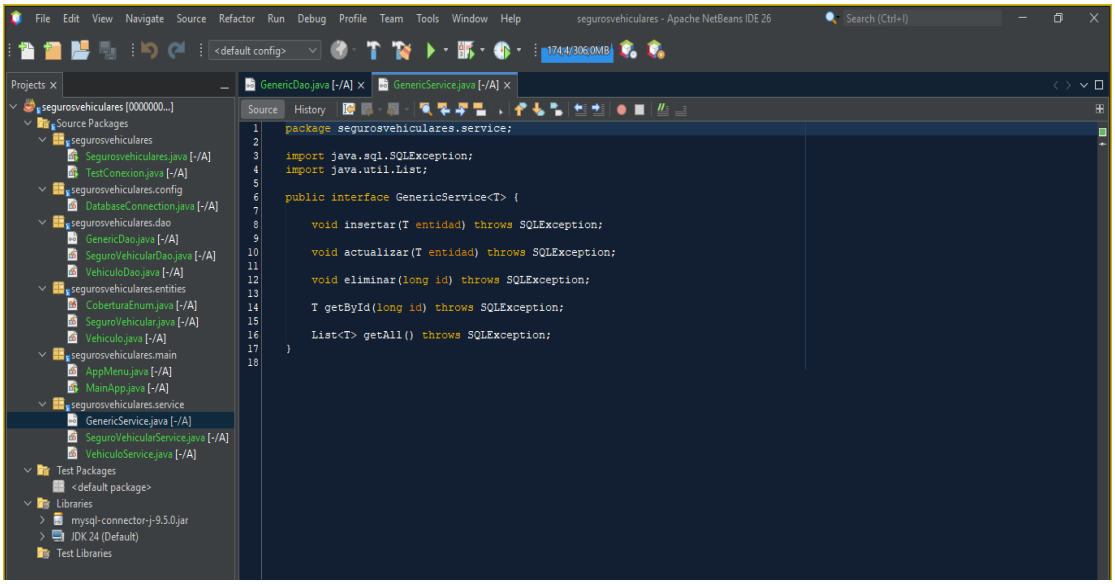
Dado que el dominio elegido fue Vehiculo → SeguroVehicular, la relación 1→1 debía reflejarse también en la capa DAO.

Para ello:

- En VehiculoDao, incluimos lógica para recuperar el seguro asociado a cada vehículo cuando correspondía.
- En SeguroVehicularDao, implementamos métodos específicos para buscar un seguro por el vehículo asociado, teniendo en cuenta que solo puede haber uno por vehículo.

De esta manera, garantizamos desde la capa de acceso a datos que la relación 1→1 se comporte correctamente y que no se generen inconsistencias al momento de crear, actualizar o eliminar registros.

5. Implementación de la capa Service (transacciones y lógica de negocio)



Una vez construida la capa DAO, desarrollamos la capa Service, que se encarga de:

- Aplicar las reglas de negocio.
- Realizar validaciones sobre los datos ingresados.

- Orquestar operaciones transaccionales usando commit y rollback.

5.1. Creación de GenericService<T> y servicios concretos

Siguiendo el mismo criterio de la capa DAO, definimos la interfaz genérica GenericService<T> con métodos como:

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

Luego, implementamos los servicios concretos:

- VehiculoService
- SeguroVehicularService

Estos servicios utilizan internamente los DAOs y se apoyan en la clase de conexión a la base de datos (DatabaseConnection) para manejar las transacciones.

5.2. Manejo de transacciones (commit/rollback)

Para las operaciones que involucran más de una tabla, decidimos trabajar con transacciones explícitas.
En particular:

- Obtenemos una conexión y seteamos setAutoCommit(false) para iniciar una transacción manual.
- Ejecutamos las operaciones en un bloque try (por ejemplo, crear primero el seguro y luego el vehículo).
- Si todas las operaciones se realizan correctamente, hacemos commit().
- Si ocurre alguna excepción (por ejemplo, violación de unicidad o error de validación), realizamos rollback() para deshacer todos los cambios.

Este enfoque nos permitió garantizar la atomicidad de las operaciones: o se guarda todo correctamente, o no se guarda nada.

5.3. Operaciones compuestas: creación de Vehículo + Seguro

Un caso clave que implementamos en la capa Service es la creación conjunta de un vehículo y su seguro.
El flujo que seguimos fue:

1. Validamos los datos del seguro y del vehículo (campos obligatorios, rangos, formatos).

2. Creamos primero el SeguroVehicular utilizando el DAO correspondiente dentro de la misma conexión.

3. Asociamos el seguro creado al vehículo en el objeto Java.

4. Insertamos el vehículo en la tabla vehiculo, asegurando que la relación 1→1 quede correctamente establecida.

5. Confirmamos la operación con commit() si todo fue exitoso.

6. Si ocurre algún problema en cualquiera de los pasos, ejecutamos rollback() y se informa el error al usuario.

5.4. Reglas de negocio y validaciones

Durante el desarrollo de la capa Service definimos y aplicamos varias reglas de negocio, entre ellas:

- Validar que campos obligatorios (como dominio, aseguradora, tipo de cobertura, vencimiento) no lleguen nulos o vacíos.

- Controlar que el dominio del vehículo no se repita (unicidad).

- Verificar que no exista más de un seguro asociado a un mismo vehículo.

- Validar fechas de vencimiento coherentes (por ejemplo, que el vencimiento no sea una fecha demasiado antigua).

- Manejar de forma amigable los casos en los que se intenta actualizar o eliminar un registro que no existe.

Estas validaciones evitan que datos inconsistentes o incompletos lleguen a la base de datos y mejoran la robustez de la aplicación.

5.5. Limpieza y cierre de recursos

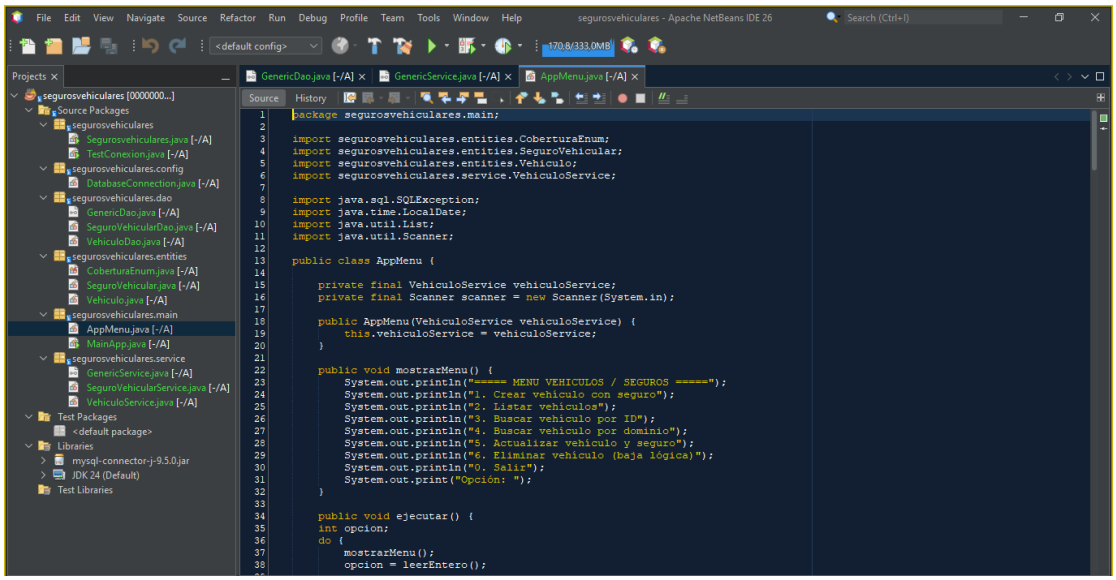
Finalmente, nos aseguramos de que, tanto en casos de éxito como de error:

- La conexión vuelva a setAutoCommit(true).

- Se cierren correctamente los recursos utilizados (conexiones, statements, etc.).

De esta manera, prevenimos fugas de recursos y aseguramos un uso adecuado de la conexión a la base.

6. Desarrollo del menú de consola (AppMenu)



Para completar la aplicación, implementamos un menú de consola que permite al usuario final interactuar con el sistema de manera sencilla, cumpliendo el requisito de contar con un AppMenu con operaciones CRUD completas y una búsqueda por campo relevante.

6.1. Estructura general del menú

En la clase principal (main) inicializamos el menú de la aplicación y creamos un bucle principal que muestra las opciones disponibles al usuario.
Entre las opciones generales incluimos:

- Alta de vehículos y seguros.
- Consulta de vehículos por ID.
- Listado de todos los vehículos.
- Actualización de datos.
- Baja lógica (marcar como eliminado).
- Búsqueda por campo relevante (por ejemplo, dominio).

Para la captura de datos utilizamos Scanner y organizamos el menú con estructuras do-while y switch para manejar las distintas opciones.

6.2. Manejo de errores y entradas inválidas

Durante la implementación del AppMenu, prestamos especial atención al manejo de errores, dado que el usuario puede ingresar valores no válidos.

Entre las decisiones que tomamos se encuentran:

- Validar y parsear cuidadosamente los datos numéricos (long, int, etc.), atrapando NumberFormatException.

- Mostrar mensajes claros cuando el usuario introduce un ID inexistente.

- Gestionar las excepciones provenientes de la base de datos (por ejemplo, errores de conexión o violaciones de claves únicas) y mostrar mensajes entendibles, sin exponer detalles técnicos innecesarios.

- En algunos casos, normalizamos entradas (por ejemplo, convertir a mayúsculas) para facilitar la comparación de opciones del menú o valores ingresados.

6.3. Búsqueda por campo relevante

Tal como indica la consigna, implementamos al menos una búsqueda por un campo relevante del dominio.
En este trabajo, la búsqueda se basó en el dominio del vehículo, ya que es un dato característico y único.

El flujo de esta funcionalidad es el siguiente:

1. El usuario ingresa el dominio a buscar.

2. El menú envía la solicitud al Service, que a su vez usa el DAO para consultar la base.

3. Si se encuentra el vehículo, se muestran sus datos y, en caso de existir, también los del seguro asociado.

4. Si no se encuentra, se informa claramente que no existe ningún vehículo con ese dominio o que el mismo pudo estar dado de baja lógica.

6.4. Mensajes al usuario y experiencia de uso

A lo largo del desarrollo del AppMenu, cuidamos la redacción de los mensajes, de modo que el usuario pueda:

- Entender qué dato está ingresando.

- Saber si la operación se realizó correctamente.

- Recibir una explicación clara cuando algo falla (ID inexistente, formato inválido, etc.).

De esta forma, el menú de consola no solo cumple con los requisitos técnicos del TFI, sino que también ofrece una experiencia de uso razonable y ordenada, permitiendo probar fácilmente las operaciones CRUD y la relación 1→1 entre Vehiculo y SeguroVehicular.

LINK DE YOUTUBE: https://youtu.be/8WCfGglJohk?si=HZSbEWpH32uJt0KD