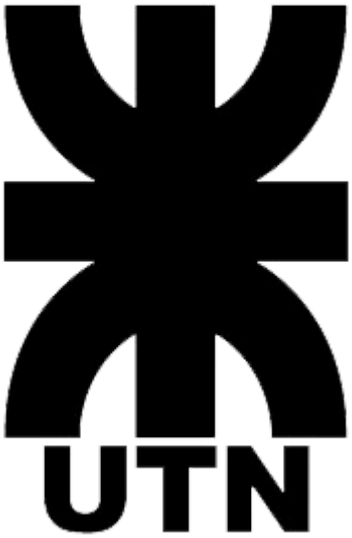
UNIVERSIDAD TECNOLÓGICA NACIONAL

TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN



PROGRAMACIÓN II

Trabajo Final Integrador

Grupo 157:

Vehículo-SeguroVehicular

Estudiantes:

Martín Maine

Juan Martinez

Joaquin Utmazian

Índice:

Introducción:

Este informe técnico documenta el Trabajo Final Integrador (TFI) de Programación 2 para la Tecnicatura Universitaria en Programación. El proyecto aplica Programación Orientada a Objetos, patrones de diseño, persistencia y arquitectura de software por capas. Se desarrolló un sistema de gestión automotriz para administrar vehículos y sus seguros asociados, abordando la gestión de datos relacionales y patrones de diseño esenciales.

El sistema implementa un registro organizado de vehículos y sus seguros, modelado con una relación unidireccional uno a uno (1→1) desde el vehículo al seguro. Permite operaciones CRUD completas sobre las entidades Vehículo y SeguroVehicular, asegurando la integridad referencial mediante transacciones (commit/rollback). Incluye búsquedas específicas (patente, póliza, marca, aseguradora) y utiliza baja lógica para la trazabilidad histórica de los datos.

Objetivos:

- Modelar relación 1→1 unidireccional (Vehículo -> SeguroVehicular) con claves foráneas y unicidad para integridad referencial.
- Crear diagrama UML completo (clases, atributos, métodos, relaciones, paquetes) como guía de implementación.
- Dominar la Programación Orientada a Objetos (POO): Comprender y aplicar encapsulamiento, abstracción (interfaces) y relaciones de objetos en un caso real.
- Implementar Arquitectura por Capas: Aplicar el patrón correctamente, manteniendo la separación de responsabilidades.
- Manejar Transacciones de Base de Datos (ACID): Adquirir experiencia práctica garantizando la integridad de datos.
- Desarrollar Habilidades de Trabajo en Equipo: Coordinar responsabilidades, integrar componentes y utilizar control de versiones.

Desarrollo:

Dominio: Vehículo -> SeguroVehicular

Como grupo optamos por implementar la relación 1→1 unidireccional utilizando el par de entidades Vehículo y SeguroVehicular, seleccionado de las once opciones propuestas en las especificaciones del trabajo práctico. La elección de este dominio se basa en que representa un caso de uso tangible y fácilmente comprensible por todos los miembros del equipo, lo que facilita el diseño de validaciones y la implementación de reglas de negocio coherentes y realistas.

Diseño y UML:

En el contexto de nuestro proyecto, la relación uno a uno es unidireccional, lo que significa que la navegación de la asociación se realiza desde la entidad Vehiculo hacia la entidad SeguroVehicular (Ver anexo 01). La entidad Vehiculo conoce y mantiene una referencia a su SeguroVehicular asociado, pero el SeguroVehicular no mantiene una referencia explícita al Vehiculo. Esta decisión de diseño simplifica el modelo y refleja la realidad de que, desde el punto de vista del negocio, es el vehículo quien posee un seguro, no el seguro quien posee un vehículo.

La implementación de esta relación en la base de datos se logra mediante una clave foránea en la tabla vehiculo que referencia a la clave primaria de la tabla seguro_vehicular, con la restricción UNIQUE aplicada sobre esta clave foránea. Esta restricción garantiza que un mismo seguro no pueda ser asignado a múltiples vehículos, preservando así la cardinalidad uno a uno de la relación.

Adicionalmente, se implementa la estrategia ON DELETE SET NULL, lo que significa que si un seguro es eliminado, la referencia en el vehículo se establece como nula en lugar de eliminar el vehículo. Esta decisión protege la integridad de los datos del vehículo, permitiendo que exista sin seguro asignado.

Arquitectura por capas:

La arquitectura en capas es un patrón de diseño de software que organiza el código en niveles jerárquicos, donde cada capa tiene una responsabilidad específica y se comunica únicamente con las capas adyacentes. Este enfoque promueve la separación de responsabilidades, facilita el mantenimiento, mejora la testabilidad y permite la evolución independiente de cada capa.

En nuestro proyecto, se implementaron las siguientes capas:

- Capa de Entidades (Entities): Contiene las clases de dominio que representan los conceptos del negocio. Estas clases son POJOs (Plain Old Java Objects) que encapsulan los datos y comportamientos básicos de las entidades Vehiculo y SeguroVehicular. No contienen lógica de persistencia ni de negocio compleja, únicamente getters, setters, constructores y métodos auxiliares como toString().
- Capa de Acceso a Datos (DAO - Data Access Object): Esta capa abstrae y encapsula todo el acceso a la base de datos. Implementa el patrón DAO, proporcionando una interfaz limpia para las operaciones CRUD sin exponer los detalles de implementación de JDBC. Cada entidad de dominio tiene su correspondiente clase DAO que maneja su persistencia.
- Capa de Servicio (Service): Actúa como intermediaria entre la capa de presentación y la capa de acceso a datos. Contiene la lógica de negocio, las reglas de validación, y el manejo de transacciones. Es responsable de coordinar operaciones que involucran múltiples DAOs y de garantizar la consistencia de los datos mediante el uso apropiado de transacciones.
- Capa de Presentación (UI/Main): Proporciona la interfaz de usuario, en este caso, un menú interactivo de consola. Se encarga de recibir las entradas del usuario, invocar los servicios correspondientes, y presentar los resultados de manera comprensible.
- Capa de Configuración (Config): Maneja aspectos transversales como la configuración de la conexión a la base de datos, centralizing la gestión de recursos compartidos.

Esta separación permite que cada capa evolucione de forma independiente. Por ejemplo, se podría reemplazar el menú de consola por una interfaz gráfica o una API

Capas de entidades y modelo de dominio

Las entidades representan los conceptos fundamentales del dominio del problema. En la programación orientada a objetos, una entidad es una clase que encapsula tanto los datos como el comportamiento relacionado con un concepto del mundo real.

La clase Vehiculo encapsula los atributos que identifican y describen un vehículo: dominio (matrícula), marca, modelo, año de fabricación, y número de chasis. Además, mantiene una referencia al SeguroVehicular asociado, implementando así la relación uno a uno unidireccional. El dominio funciona como identificador único natural del vehículo, siguiendo el formato de las patentes argentinas.

La clase SeguroVehicular representa un contrato de seguro vehicular con sus atributos característicos: aseguradora (compañía emisora), número de póliza (identificador único del seguro), tipo de cobertura, y fecha de vencimiento. El número de póliza sirve como identificador de negocio único.

Ambas entidades heredan de una clase base abstracta EntidadBase que implementa el patrón de eliminación lógica (soft delete) mediante un atributo booleano "eliminado". Este patrón permite mantener un registro histórico de las entidades sin eliminarlas físicamente de la base de datos, lo cual es fundamental para auditoría y trazabilidad.

El enum Cobertura modela los tipos de cobertura de seguro disponibles: RC (Responsabilidad Civil), TERCEROS (Terceros Completo), y TODO_RIESGO (Cobertura Total). El uso de enumeraciones garantiza type safety y previene valores inválidos en este campo crítico.

Gestión de conexión de bases de datos

La gestión adecuada de conexiones a bases de datos es fundamental para el rendimiento, la escalabilidad y la confiabilidad de una aplicación. Una conexión mal gestionada puede resultar en fugas de recursos, problemas de concurrencia, o agotamiento del pool de conexiones del servidor de base de datos.

JDBC (Java Database Connectivity) es la API estándar de Java para conectarse a bases de datos relacionales. Proporciona un conjunto de interfaces y clases que permiten ejecutar sentencias SQL, procesar resultados, y gestionar transacciones de manera independiente del sistema gestor de base de datos específico.

En nuestro proyecto, la clase DatabaseConnection implementa el patrón de gestión centralizada de conexiones. Esta clase se encarga de:

Cargar el driver JDBC de MySQL mediante reflexión, permitiendo que el DriverManager reconozca y utilice el driver apropiado.

Proporcionar un método estático getConnection() que crea y retorna nuevas conexiones configuradas con los parámetros apropiados, incluyendo la URL de conexión, credenciales de usuario, y configuraciones de zona horaria.

Implementar el método closeConnection() para cerrar conexiones de forma segura, manejando apropiadamente las excepciones que puedan ocurrir durante el cierre.

El uso de try-with-resources en el código que utiliza conexiones garantiza que estas se cierren automáticamente al finalizar su uso, previniendo fugas de recursos. Este patrón es especialmente importante en operaciones de lectura donde se obtiene una conexión, se ejecuta una consulta, y se cierra inmediatamente después de procesar los resultados.

Para operaciones transaccionales, el Service obtiene una conexión, desactiva el autocommit, ejecuta múltiples operaciones, y finalmente realiza commit o rollback según corresponda. Esta conexión se pasa como parámetro a los métodos del DAO, permitiendo que múltiples operaciones participen de la misma transacción.

Patron DATA ACCESS OBJECT (DAO)

El patrón Data Access Object es un patrón estructural que separa la lógica de acceso a datos de la lógica de negocio. Proporciona una interfaz abstracta para interactuar con una fuente de datos, ocultando los detalles de implementación específicos del mecanismo de persistencia.

Los beneficios principales de este patrón incluyen:

Separación de responsabilidades: La lógica de acceso a datos está completamente aislada del resto de la aplicación.

Facilidad de mantenimiento: Los cambios en el esquema de base de datos o en las queries SQL solo afectan a las clases DAO.

Testabilidad: Se pueden crear implementaciones mock de los DAOs para realizar pruebas unitarias sin necesidad de una base de datos real.

Flexibilidad: Es posible cambiar el mecanismo de persistencia (por ejemplo, de JDBC a JPA) sin afectar al resto de la aplicación.

En nuestro proyecto, la interfaz GenericDao define las operaciones CRUD estándar que cualquier DAO debe implementar: crear, leer (por ID y todos), actualizar, y eliminar. Esta interfaz genérica permite que todas las implementaciones de DAO compartan una estructura consistente.

Cada implementación de DAO (SeguroVehicularDao y VehiculoDao) contiene:

Constantes con las consultas SQL como cadenas preparadas, facilitando su mantenimiento y evitando la construcción dinámica de SQL que podría introducir vulnerabilidades.

Métodos públicos que implementan la interfaz GenericDao, cada uno con dos versiones: una que gestiona su propia conexión y otra que recibe una conexión como parámetro para participar en transacciones.

Métodos privados auxiliares como mapResultSetToEntity que transforman los ResultSets de JDBC en objetos del dominio, centralizando esta lógica de transformación.

Métodos de búsqueda específicos del dominio, como buscarPorDominio en VehiculoDao o buscarPorPoliza en SeguroVehicularDao.

El uso consistente de PreparedStatement en lugar de Statement previene ataques de inyección SQL, ya que los parámetros se pasan de forma segura y son tratados como datos, no como código SQL ejecutable.

Capa SERVICE y manejo de transacciones

La capa de servicio constituye el núcleo de la lógica de negocio de la aplicación. Es responsable de orquestar operaciones complejas, aplicar reglas de negocio, realizar validaciones, y gestionar transacciones que abarcan múltiples operaciones de base de datos.

Una transacción es una unidad lógica de trabajo que agrupa una o más operaciones de base de datos que deben ejecutarse de forma atómica. Las transacciones deben cumplir con las propiedades ACID:

- Atomicidad: Todas las operaciones de la transacción se completan exitosamente, o ninguna se aplica. No existen estados intermedios.
- Consistencia: La transacción lleva la base de datos de un estado válido a otro estado válido, manteniendo todas las restricciones de integridad.
- Aislamiento: Las operaciones de transacciones concurrentes no interfieren entre sí. Cada transacción se ejecuta como si fuera la única en el sistema.
- Durabilidad: Una vez que una transacción se confirma (commit), los cambios persisten incluso ante fallos del sistema.

El manejo de transacciones JDBC se basa en setAutoCommit(false), commit() y rollback(). Por defecto, autocommit es true. Para transacciones, se desactiva, se ejecutan las operaciones y se confirma (commit) o revierte (rollback) si hay fallos.

Las clases VehiculoService y SeguroVehicularService aplican este patrón: inician con setAutoCommit(false), validan el negocio primero, usan commit() si todo es exitoso o rollback() ante excepciones, y un bloque finally asegura el cierre de la conexión y la restauración del autocommit.

El método insertarVehiculoConSeguro es un ejemplo clave de transacción compleja (crea seguro, asigna, crea vehículo) que garantiza la consistencia.

Las validaciones en la capa de servicio incluyen: campos obligatorios, formatos (ej. dominio), unicidad (dominio, chasis, póliza), rangos (año), y disponibilidad de seguros.

Interface de usuario - Menú de consola

La interfaz de usuario es el punto de interacción entre el sistema y el usuario final. Aunque en este proyecto se implementa mediante un menú de consola en lugar de una interfaz gráfica, los principios de diseño de UI siguen siendo aplicables: claridad, consistencia, retroalimentación al usuario, y manejo de errores.

La clase AppMenu implementa un sistema de menús jerárquico con navegación intuitiva. El diseño utiliza el patrón de menú en cascada donde cada opción puede llevar a un submenú o ejecutar una acción específica.

El menú principal divide las operaciones en tres categorías lógicas: gestión de vehículos (CRUD completo), gestión de seguros (CRUD completo), y operaciones especiales que demuestran transacciones complejas.

Cada submenú mantiene un bucle while que solo se interrumpe cuando el usuario selecciona la opción de volver al menú anterior, implementando así una navegación por profundidad.

El manejo de entrada del usuario se realiza mediante métodos auxiliares especializados: leerTexto para cadenas obligatorias, leerTextoOpcional para cadenas que pueden estar vacías, leerEntero para números, y leerFecha para fechas con formato específico. Estos métodos incluyen validación y manejo de errores, como la captura de InputMismatchException cuando el usuario ingresa texto donde se espera un número.

La retroalimentación al usuario es constante y clara: mensajes de éxito con el símbolo ✓, mensajes de error con ✗, y advertencias con ⚠. Cada operación confirma su resultado, ya sea exitoso o fallido, proporcionando información relevante como los IDs generados para nuevos registros.

El manejo de excepciones a nivel de UI captura errores de la capa de servicio y los presenta de forma comprensible para el usuario, ocultando detalles técnicos innecesarios pero proporcionando información suficiente para entender qué salió mal.

Validaciones y reglas de negocio.

Las validaciones de datos son clave para la integridad del sistema y se dividen en técnicas y de negocio. Las validaciones técnicas revisan aspectos formales (tipo, longitud, formato, obligatoriedad). En nuestro proyecto, se implementan en la capa de servicio. Ejemplos incluyen el formato de patente argentina (ABC123 o AB123CD) y el rango del año del vehículo (1900 al año actual + 1), además de la verificación de longitud de campos como aseguradora, marca y modelo.

Las validaciones de negocio verifican reglas específicas del dominio que van más allá de la corrección formal de los datos. Estas incluyen:

- Unicidad de identificadores de negocio: Se verifica que no existan duplicados de dominio, número de chasis, o número de póliza antes de insertar o actualizar. Esta

validación protege contra errores de entrada y garantiza la unicidad de estos campos críticos.

- Relación uno a uno: Antes de asignar un seguro a un vehículo, se verifica que ese seguro no esté ya asignado a otro vehículo. Esta validación es fundamental para mantener la integridad de la relación uno a uno.
- Consistencia temporal: Se emiten advertencias cuando las fechas de vencimiento de seguros son muy antiguas, aunque no se impide su creación. Esta validación flexible permite manejar casos especiales manteniendo al usuario informado.
- Validaciones en cascada: Al actualizar un vehículo, si se cambia el dominio, se verifica que el nuevo dominio no esté en uso. Si se mantiene el mismo dominio, esta validación se omite.

El principio de "fail fast" se aplica consistentemente: las validaciones se ejecutan lo más temprano posible en el flujo de procesamiento, antes de iniciar transacciones o ejecutar operaciones costosas de base de datos. Esto mejora el rendimiento y proporciona retroalimentación inmediata al usuario.

PREPAREDSTATEMENT y seguridad

La seguridad en el acceso a bases de datos es clave, siendo la inyección SQL una vulnerabilidad peligrosa que permite a un atacante manipular consultas para acceder o modificar datos. PreparedStatement de JDBC ayuda a prevenir esto al usar consultas precompiladas con parámetros, separando la estructura del SQL de los datos, a diferencia de Statement.

Los beneficios de usar PreparedStatement incluyen:

- Prevención de inyección SQL: Los parámetros se tratan como datos puros, no como código SQL ejecutable. El driver de JDBC se encarga de escapar apropiadamente cualquier carácter especial.
- Mejor rendimiento: La consulta se precompila una vez y puede ejecutarse múltiples veces con diferentes parámetros. El motor de base de datos puede optimizar y cachear el plan de ejecución.
- Código más limpio y mantenible: La separación entre la estructura de la query y los datos hace el código más legible.
- Type safety: Los métodos setString(), setInt(), setDate(), etc., garantizan que los parámetros tienen el tipo correcto.

En nuestro proyecto, todas las consultas SQL utilizan PreparedStatement. Por ejemplo, en lugar de construir un SQL como "SELECT * FROM vehiculo WHERE dominio = '" + dominio + "'", se usa "SELECT * FROM vehiculo WHERE dominio = ?" y luego se establece el parámetro con stmt.setString(1, dominio). El uso de constantes para las consultas SQL (declaradas como static final String) centraliza las queries, facilita su mantenimiento, y previene errores de escritura al evitar la construcción dinámica de SQL.

Eliminación lógica (SOFT DELETE)

La eliminación lógica, también conocida como soft delete, es un patrón de diseño donde los registros no se eliminan físicamente de la base de datos sino que se marcan como eliminados mediante un campo booleano. Este enfoque contrasta con la eliminación física (hard delete) donde los registros se eliminan permanentemente.

Las ventajas de la eliminación lógica incluyen:

- Auditoría y trazabilidad: Se mantiene un registro completo de todos los datos históricos, fundamental para auditorías, análisis histórico, y cumplimiento regulatorio.
- Recuperación de datos: Los registros eliminados accidentalmente pueden recuperarse simplemente cambiando el flag de eliminado.
- Integridad referencial: Se evitan problemas con referencias a registros eliminados. Aunque un registro esté marcado como eliminado, las claves foráneas que lo referencian siguen siendo válidas.
- Análisis histórico: Se pueden realizar análisis que incluyan datos históricos que ya no están activos en el sistema.

Nuestro proyecto aplica el patrón de eliminación lógica: la clase EntidadBase incluye el atributo booleano eliminado. Las consultas listan automáticamente solo las entidades activas (eliminado = FALSE), pero las búsquedas por ID permiten acceder a registros eliminados si es necesario. Los métodos DAO de eliminación realizan un UPDATE que establece eliminado = TRUE en lugar de un DELETE, manteniendo la información intacta y permitiendo la reversión.

Normalización y diseño de base de datos

El diseño de la base de datos sigue los principios de normalización, un proceso que organiza las tablas y columnas de una base de datos relacional para reducir la redundancia y mejorar la integridad de los datos.

Nuestro esquema cumple con las tres primeras formas normales:

- Primera Forma Normal (1NF): Todos los atributos contienen valores atómicos (indivisibles). No existen grupos repetitivos ni arrays dentro de las columnas.
- Segunda Forma Normal (2NF): Todos los atributos no clave dependen completamente de la clave primaria. No existen dependencias parciales.
- Tercera Forma Normal (3NF): No existen dependencias transitivas. Todos los atributos no clave dependen directamente de la clave primaria y no de otros atributos no clave.

La tabla seguro_vehicular solo contiene atributos del seguro (aseguradora, póliza, cobertura, vencimiento) sin redundancia de datos del vehículo. La tabla vehiculo incluye solo sus atributos (dominio, marca, modelo, año, chasis) y se relaciona con el seguro mediante clave foránea, no duplicación de datos.

El uso de claves primarias autoincrementales (BIGINT AUTO_INCREMENT) proporciona identificadores únicos estables. Las restricciones de unicidad sobre dominio, número de chasis y número de póliza garantizan la no duplicación. Los índices en campos clave (marca, modelo, aseguradora, vencimiento) optimizan el rendimiento de las consultas.

Conclusión:

El desarrollo de este sistema de gestión de vehículos y seguros ha permitido aplicar de manera integral los conceptos fundamentales de programación orientada a objetos, arquitectura de software, y persistencia de datos mediante JDBC.

La implementación de la relación uno a uno unidireccional demuestra cómo modelar restricciones de negocio complejas tanto a nivel de código como de base de datos, garantizando la integridad de los datos mediante múltiples mecanismos: restricciones de base de datos, validaciones en la capa de servicio, y manejo apropiado de transacciones.

La arquitectura en capas ha resultado en un sistema mantenible y extensible, donde cada capa tiene responsabilidades claras y acoplamiento mínimo con las demás. Esta separación facilita tanto el testing como la evolución futura del sistema.

El uso del patrón DAO ha abstracto completamente los detalles de acceso a datos, permitiendo que el resto de la aplicación trabaje con objetos del dominio sin conocimiento de SQL o JDBC. Esta abstracción facilita posibles migraciones a otros mecanismos de persistencia en el futuro.

El manejo de transacciones garantiza la consistencia de los datos incluso ante fallos parciales, demostrando cómo operaciones complejas que involucran múltiples entidades pueden ejecutarse de forma atómica.

Las validaciones implementadas a múltiples niveles garantizan la integridad de los datos, previniendo estados inconsistentes y proporcionando retroalimentación clara al usuario sobre errores en los datos ingresados.

El proyecto demuestra que es posible desarrollar aplicaciones robustas y bien diseñadas sin depender de frameworks complejos, utilizando únicamente las APIs estándar de Java y aplicando principios sólidos de diseño de software.

Bibliografía

Cimino, C. (s. f.). OBJETOS (POO) desde -10 en Java [Lista de reproducción]. YouTube. Recuperado el 14 de febrero de 2025, de https://www.youtube.com/watch?v=voMOPqtnJto&list=PLOw7b-NX043aSC7ZNtEuVfY8xZoNzVqdJ

KeepCoding. (s. f). *¿Qué es ACID en bases de datos?* https://keepcoding.io/blog/que-es-acid-bases-datos/

Microsoft. (s. f). *ACID properties*. Microsoft Learn. https://learn.microsoft.com/es-es/windows/win32/cossdk/acid-properties

Oracle. (s. f). *JDBC basics*. Oracle Java Documentation. https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html

Oracle. (s. f.). *The JDBC API*. Oracle Java Tutorials. https://docs.oracle.com/javase/tutorial/jdbc/

ReactiveProgramming. (s. f). *Data Access Object*. https://reactiveprogramming.io/blog/es/patrones-arquitectonicos/dao

Stack Overflow. (2011, diciembre 1). *How can prepared statements protect from SQL injection attacks?* https://stackoverflow.com/questions/8263371/how-can-prepared-statements-protect-from-sql-injection-attacks

Universidad Tecnológica Nacional. (s, f). *Lenguaje SQL* [Apuntes de cátedra].

Anexos

Anexo 01: Diagrama UML realizado en draw.io