1. Integrantes y roles

Integrantes del equipo

Santiago Arroquigaray – Responsable de modelo de dominio, diagrama UML , persistencia y diseño de la base de datos Luciano Cartagena – Responsable de lógica de negocio, servicios (Service), interfaz de usuario por consola, AppMenu y pruebas

Análisis y dominio: identificación de entidades principales (Vehículo, SeguroVehicular, Cobertura) y de las relaciones 1↔1, además de la definición de reglas de negocio (por ejemplo, un seguro no puede estar asociado a más de un vehículo). Diseño de base de datos y persistencia: diseño de tablas SQL, claves primarias y foráneas, y desarrollo de la capa DAO con JDBC (VehiculoDaoImpl, SeguroVehicularDaoImpl). Lógica de negocio: implementación de la capa de servicio (VehiculoService, SeguroVehicularService), incluyendo validaciones y métodos transaccionales. Interfaz de usuario y pruebas: implementación de AppMenu, definición de los flujos de uso, ejecución de pruebas manuales y captura de evidencias (pantallas y consultas SQL).

2. Elección del dominio y justificación

El dominio elegido es la gestión de vehículos y seguros vehiculares.

Es un escenario común, ya que la mayoría de las personas conoce el concepto de patente, vehículo, aseguradora y póliza, pero el principal enfoque era aprender más sobre seguros automovilisticos. Permite incluir una relación 1 a 1 interesante entre vehículo y seguro vehicular, lo que obliga a tomar decisiones de diseño tanto en el modelo de objetos como en la base de datos. Facilita la definición de reglas de negocio claras, como la imposibilidad de asociar un mismo seguro a más de un vehículo o la invalidación de seguros vencidos. El dominio combina una facilidad conceptual con desafíos de modelado y persistencia adecuados a los objetivos del trabajo práctico.

3. Diseño: decisiones clave y UML

3.1. Entidades principales

Base: clase abstracta que concentra los atributos comunes a todas las entidades persistentes: id: Long eliminado: Boolean (para implementar baja lógica). Vehiculo: Atributos: dominio, marca, modelo, anio, nro_chasis, id_seguro y el objeto seguro: SeguroVehicular. Extiende de Base. SeguroVehicular: Atributos: aseguradora, nro_poliza, cobertura: Cobertura, vencimiento. Extiende de Base. Cobertura: Enum con los posibles tipos de cobertura: TERCEROS_COMPLETO, TODO_RIESGO, RESPONSABILIDAD_CIVIL, etc.

3.2. Decisión 1→1: FK única vs PK compartida

Para modelar la relación uno a uno entre Vehiculo y SeguroVehicular se analizaron dos alternativas:

Clave foránea única en la tabla de vehículos (opción implementada) La tabla vehiculos contiene una columna id_seguro que referencia a seguro_vehicular.id. A nivel de dominio, Vehiculo contiene un atributo seguro de

tipo SeguroVehicular. La unicidad de esa relación se controla mediante reglas en la capa de servicio (VehiculoService), que impiden que un mismo id_seguro sea asignado a dos vehículos distintos. Ventajas: Menos acoplamiento entre tablas. Posibilidad de tener vehículos sin seguro (id_seguro nulo). Desventajas: La relación 1↔1 no queda forzada al 100 % por la base de datos, sino por la lógica de negocio. Clave primaria compartida (no implementada) Ambos comparten el mismo id, de modo que Vehiculo.id = SeguroVehicular.id. Esto fuerza estrictamente el 1↔1, pero complica la creación y el manejo de registros (se requiere coordinar las inserciones de ambas tablas y no se pueden tener vehículos sin seguro). Se optó por la opción 1 (FK única en vehiculos) por simplicidad y porque se ajusta mejor al dominio: un vehículo puede existir sin seguro cargado, y un seguro deja de tener sentido sin un vehículo asociado.

3.3. UML

El diagrama UML de clases resume el diseño:

Base es una clase abstracta padre de Vehiculo y SeguroVehicular. Vehiculo tiene una asociación 0..1 con SeguroVehicular (un vehículo puede o no tener un seguro). SeguroVehicular utiliza el enum Cobertura. Existen las capas DAO, Service y la clase AppMenu como fachada de la interfaz de usuario. En el informe, insertar la imagen del diagrama (por ejemplo, Diagram.png).

4. Arquitectura por capas

El sistema se estructuró siguiendo una arquitectura por capas:

Capa de entidades (modelo de dominio) – paquete entities Clases: Base, Vehiculo, SeguroVehicular, Cobertura. Responsabilidad: representar los objetos del dominio y sus relaciones, sin lógica de acceso a datos ni de presentación. Capa de acceso a datos (DAO) – paquete dao Interfaces y clases: GenericDao, VehiculoDaoImpl, SeguroVehicularDaoImpl. Responsabilidad: encapsular el acceso a la base de datos mediante JDBC (SQL INSERT, SELECT, UPDATE, DELETE). Se proveen dos variantes de cada método: una que crea su propia conexión y otra que acepta una conexión externa, útil para transacciones. Capa de servicios (lógica de negocio) – paquete service Interfaces y clases: GenericService, VehiculoService, SeguroVehicularService. Responsabilidad: aplicar reglas de negocio y validaciones antes de llamar a los DAO. Ejemplo: validar que la fecha de vencimiento del seguro sea futura, que el dominio no sea nulo ni duplicado, que un seguro no se asigne a más de un vehículo, etc. También se maneja la lógica transaccional (crearVehiculoConSeguro). Capa de presentación / interfaz de usuario – paquete main Clase: AppMenu (y Main como punto de entrada). Responsabilidad: interactuar con el usuario por consola, mostrar menús y mensajes, capturar entradas y coordinar acciones con la capa de servicios. Esta separación facilita la mantenibilidad: futuras modificaciones en BD o en reglas de negocio no impactan directamente en la interfaz de usuario.

5. Persistencia: estructura, orden de operaciones y transacciones

5.1. Estructura de la base de datos

Base de datos: Vehiculos_con_seguro

Tablas principales:

- seguro_vehicular
  - id (PK, BIGINT AUTO_INCREMENT)
  - eliminado (TINYINT(1))
  - aseguradora (VARCHAR)
  - nro_poliza (VARCHAR)
  - cobertura (VARCHAR) – almacena el nombre del enum Cobertura
  - vencimiento (DATE)
- vehiculos
  - id (PK, BIGINT AUTO_INCREMENT)
  - eliminado (TINYINT(1))
  - dominio (VARCHAR, UNIQUE)
  - marca (VARCHAR)
  - modelo (VARCHAR)
  - anio (INT)
  - nro_chasis (VARCHAR)
  - id_seguro (BIGINT, FK a seguro_vehicular.id)

5.2. Orden típico de operaciones

Alta de seguro individual: El usuario carga datos en AppMenu. AppMenu llama a SeguroVehicularService.insertar. El service valida los datos y delega en SeguroVehicularDaoImpl.crear. El DAO ejecuta INSERT y devuelve el seguro con su id generado. Alta de vehículo individual: El usuario carga datos en AppMenu. AppMenu llama a VehiculoService.insertar. Se validan dominio, marca, modelo, y se controla que, si hay id_seguro, ese seguro no esté asociado a otro vehículo. Se delega en VehiculoDaoImpl.crear. Creación transaccional de vehículo + seguro: El usuario selecciona la opción “Crear vehículo con seguro nuevo”. AppMenu.crearVehiculoConSeguroNuevo() pide primero los datos del seguro, luego los del vehículo. AppMenu invoca VehiculoService.crearVehiculoConSeguro(vehiculo, seguro). Dentro del service: Se abre una conexión y se fija setAutoCommit(false). Se valida el seguro y se inserta con SeguroVehicularDaoImpl.crear(seguro, conn). Se actualiza vehiculo.setId_seguro(seguro.getId()). Se valida el vehículo y se inserta con VehiculoDaoImpl.crear(vehiculo, conn). Si todo es correcto, se hace commit(). Ante cualquier excepción de validación o SQL, se hace rollback() y se informa el error al usuario.

5.3. Commit / rollback

Commit: se realiza explícitamente en VehiculoService.crearVehiculoConSeguro cuando ambas inserciones se completan sin error. Rollback: se ejecuta en el bloque catch del método transaccional cuando: La validación de seguro o vehículo falla. Ocurre un SQLException durante alguno de los INSERT. De esta manera se garantiza que nunca se cree un seguro sin su vehículo correspondiente en este flujo, ni viceversa.

6. Validaciones y reglas de negocio

- Vehículo
  - dominio, marca y modelo no pueden ser nulos ni vacíos.
  - Se intenta que el dominio sea único; la violación de la restricción UNIQUE se captura y se informa.

    - Si se indica id_seguro, se verifica que dicho seguro no esté ya asociado a otro vehículo (verificarSeguroNoAsociado).
- SeguroVehicular
    - aseguradora, nro_poliza y cobertura no pueden ser nulos ni vacíos.
    - La fecha de vencimiento debe ser futura (no se admiten seguros ya vencidos).
    - cobertura debe ser un valor válido del enum Cobertura. Entradas inválidas en el menú generan una IllegalArgumentException.
- Regla de negocio clave 1↔1
    - Solo puede haber un vehículo asociado a un seguro determinado.
    - Al actualizar un vehículo, si se cambia id_seguro, se verifica que el nuevo seguro no esté ligado a otro vehículo existente.
    - Baja lógica
    - Ningún registro se elimina físicamente: se marca el campo eliminado = true.

7. Pruebas realizadas

Las pruebas se centraron en verificar:

Altas simples de vehículos y seguros. Creación transaccional de vehículo con seguro nuevo. Actualizaciones: Cambiar datos de vehículo. Cambiar tipo de cobertura o fecha de vencimiento de un seguro. Reglas de negocio: Intentar asociar un mismo seguro a dos vehículos distintos. Intentar crear un seguro con fecha de vencimiento pasada. Intentar crear vehículo sin dominio. Búsquedas y consultas: Listar vehículos. Buscar vehículo por dominio. Listar seguros y leer por ID. En el informe, es recomendable agregar capturas de pantalla de:

El menú principal. La creación transaccional de vehículo + seguro. Algún mensaje de error por validación (por ejemplo, dominio duplicado). Consultas SQL ejecutadas desde el cliente (por ejemplo SELECT * FROM vehiculos;, SELECT * FROM seguro_vehicular;) mostrando los datos resultantes. Ejemplos de consultas SQL útiles para documentar:

sql Copy SELECT * FROM vehiculos; SELECT * FROM seguro_vehicular; SELECT v.dominio, s.aseguradora, s.nro_poliza FROM vehiculos v LEFT JOIN seguro_vehicular s ON v.id_seguro = s.id;

-- Verificar que cada id_seguro aparezca a lo sumo una vez SELECT id_seguro, COUNT*() as cantidad FROM vehiculos WHERE id_seguro IS NOT NULL GROUP BY id_seguro HAVING COUNT*() > 1;

8. Conclusiones y mejoras futuras

Conclusiones

La arquitectura por capas permitió separar claramente responsabilidades: modelo, persistencia, negocio y presentación. El uso de un enum (Cobertura) hizo más seguro el manejo de tipos de cobertura, evitando errores de tipeo. El manejo transaccional en crearVehiculoConSeguro garantiza consistencia entre vehículo y seguro al crear ambos a la vez. La baja lógica mediante el atributo eliminado preserva el historial de registros y facilita futuras auditorías. Mejoras futuras

Agregar una capa de repositorios genéricos o usar un ORM (por ejemplo JPA/Hibernate) para reducir código repetitivo en la capa DAO. Incorporar una interfaz gráfica (Swing/JavaFX o web) en lugar del menú de consola. Implementar más entidades del dominio: conductores, siniestros, pagos de pólizas, historial de coberturas, etc. Agregar tests automatizados (JUnit) para la capa de servicios. Aplicar validaciones más avanzadas: Formato de dominio (patente) según norma. Formato del número de chasis. Regla de negocio sobre tipos de cobertura según la antigüedad del vehículo.

9. Fuentes y herramientas utilizadas

Lenguaje: Java SE 8+. Base de datos: MySQL / MariaDB. Driver JDBC: mysql-connector-j. IDE: (NetBeans / IntelliJ IDEA / Eclipse – completar con el que usaron). Herramienta de modelado UML: (StarUML / Visual Paradigm / PlantUML – ajustar). Cliente SQL: (MySQL Workbench / consola / phpMyAdmin – ajustar). Uso de herramientas de IA: en el desarrollo del proyecto se utilizó un asistente de IA (Abacus.AI ChatLLM Teams / GPT-5.1) para validación de requerimientos.