UNIVERSIDAD TECNOLOGICA NACIONAL

Trabajo Final Integrador

Aplicación Java con relación 1→1 unidireccional + DAO + MySQL

Alumno

Zupan Lucas

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.

Programacion II

Docente Titular

Ariel Enferrel

Docente Tutor

Diego Vargas

17 de Noviembre de 2025

Objetivo General **3**
Consignas **3**
Dominio (elegir 1 pareja A → B) **3**
Estructura del proyecto (paquetes) **3**
Requerimientos técnicos **3**
1) Diseño (previo al código) 3
2) Entidades (genérico) 4
3) Base de datos (MySQL) 4
4) DAO 4
5) Service (transacciones obligatorias) 4
6) AppMenu (consola) 5
Entregables (obligatorios) **5**
1. Repositorio GitHub público 5
2. Informe (PDF, 6–8 páginas) 5
3. Video (10–15 minutos) 5
Criterios de evaluación **6**
Checklist (sugerido) **6**
Especificación de campos – TP Integrador (1→1 unidireccional) **6**
Mascota → Microchip 7
Notas generales 7
Desarrollo **8**
Elección del dominio y justificación 8
Justificación del dominio 8
Diseño: decisiones clave (1→1, FK única vs PK compartida) + UML 8
1. Decisiones clave del modelo de datos 8
2. Decisiones del diseño orientado a objetos 9
- Uso de clase abstracta Base 9
- DAO Pattern por entidad 9
- Service Layer 9
- Capa de Presentación 9
3. Diagrama UML del sistema 10
4. Justificación del diseño general 10
1. Claridad estructural 11
2. Robustez y consistencia 11
3. Extensibilidad 11
Arquitectura por capas (responsabilidades de cada paquete) 11
1. Capa de Presentación (UI) 11
2. Capa de Negocio (Service Layer) 12
3. Capa de Persistencia (DAO Layer) 13
4. Capa de Modelo (Domain Model) 14
Módulo de Infraestructura 14
Persistencia: estructura de la base, orden de operaciones y transacciones (dónde se hace commit/rollback) 15

Validaciones y reglas de negocio..... 16
Validaciones en la capa de presentación (MenuHandler).....16
Validaciones en la capa de servicio (Service Layer).....17
Mascotas..... 17
Microchips.....17
Validaciones en la base de datos (SQL + Constraints)..... 17
Integración entre capas..... 18
Conclusiones y mejoras futuras.....18
Conclusiones..... 18
Mejoras futuras..... 19
Fuentes y herramientas utilizadas..... 19
Herramientas de desarrollo.....19
Fuentes consultadas.....20
Uso de herramientas de asistencia (IA).....20

Objetivo General

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

Consignas

- Lenguaje: Java (recomendado 21).
- Persistencia: JDBC (sin ORM) con MySQL.
- Patrón de diseño: DAO y capa Service.
- Código limpio, legible, con manejo de excepciones en todas las capas.
- Relación 1→1 unidireccional: sólo la clase “A” contiene el atributo que referencia a “B”.
- Equipos de 4 personas, sin excepción.

Dominio (elegir 1 pareja A → B)

Se debe elegir entre una de estas opciones para resolver el TPI.

- Empleado → Legajo
- Usuario → CredencialAcceso
- Vehiculo → SeguroVehicular
- DispositivoIoT → ConfiguracionRed
- Paciente → HistoriaClinica
- Empresa → DomicilioFiscal
- Libro → FichaBibliografica
- Mascota → Microchip
- Producto → CodigoBarras
- Propiedad → EscrituraNotarial
- Pedido → Envio

Estructura del proyecto (paquetes)

- config/: conexión a la base (lectura de propiedades externas).
- entities/: clases de dominio (A y B) con id y eliminado (baja lógica).
- dao/: interfaces genéricas y DAOs concretos (JDBC + PreparedStatement).
- service/: reglas de negocio, validaciones y orquestación de transacciones
- main/: arranque de la app y AppMenu (consola).

Requerimientos técnicos

1) Diseño (previo al código)
    - Diagrama UML de clases con:
        - Atributos (tipo/visibilidad) y métodos (firma/visibilidad).

    - A → B (1..1) unidireccional explícita.
    - Paquetes y dependencias principales.
- El UML guía toda la implementación.

2) Entidades (genérico)

- A: atributos propios + id (INT/BIGINT) + eliminado (BOOLEAN).
- B: atributos propios + id + eliminado.
- A contiene private B detalle; (o nombre equivalente).
- Constructores (vacío y completo), getters/setters y toString() legible.

3) Base de datos (MySQL)

- Clase DatabaseConnection en config/ con método estático que retorne java.sql.Connection.
- Entregar un archivo .sql con las instrucciones para crear la base de datos y sus tablas (CREATE DATABASE, CREATE TABLE, claves primarias, foráneas, índices y restricciones necesarias para 1→1).
- Entregar otro .sql con datos de prueba (INSERTs) para levantar el proyecto desde cero.
- Relación 1→1 recomendada: clave foránea única en la tabla de B (campo a_id con UNIQUE, FOREIGN KEY a A(id), ON DELETE CASCADE). Alternativa válida: clave primaria compartida (la PK de B es también FK a A).

4) DAO

- GenericDao<T> con: crear(T), leer(long id), leerTodos(), actualizar(T), eliminar(long id).
- DAOs concretos para A y B usando PreparedStatement en todas las operaciones.
- Los DAO deben ofrecer métodos que acepten una Connection externa (para participar de la misma transacción).

5) Service (transacciones obligatorias)

- GenericService<T>: insertar, actualizar, eliminar, getById, getAll.
- AService y BService:
    - Abrir transacción: setAutoCommit(false) sobre una conexión compartida.
    - Ejecutar operaciones compuestas (por ej., crear B, asociarla a A y crear A).
    - commit() si todo OK; rollback() ante cualquier error.
    - Restablecer autoCommit(true) y cerrar recursos.
    - Validaciones (campos obligatorios, formatos según dominio) y regla 1→1 (impedir más de un B por A).

6) AppMenu (consola)

- Main invoca AppMenu.
- Convertir entradas a mayúsculas donde aplique.
- Debe permitir CRUD completo de A y B: crear, leer por ID, listar, actualizar y eliminar lógico.
- Incluir al menos una búsqueda por un campo relevante (ej.: DNI, dominio, ISBN, etc., según el dominio elegido).
- Manejo robusto de:
  - Entradas inválidas (parseos numéricos, formatos).
  - IDs inexistentes.
  - Errores de base de datos y violaciones de unicidad.
- Mensajes claros de éxito/error.

Entregables (obligatorios)

1. Repositorio GitHub público

- Código fuente completo.
- README.md con:
  - Descripción del dominio elegido.
  - Requisitos (Java/BD) y pasos para crear la base con el .sql provisto.
  - Cómo compilar y ejecutar (credenciales de prueba y flujo de uso).
  - Enlace al video.
- SQL:
  - Archivo con instrucciones para crear la base y tablas.
  - Archivo con datos de prueba.
- UML (imagen .png/.jpg/.pdf).
- Informe (PDF) dentro del repo.

2. Informe (PDF, 6–8 páginas)

- Integrantes (4) y roles.
- Elección del dominio y justificación.
- Diseño: decisiones clave (1→1, FK única vs PK compartida) + UML.
- Arquitectura por capas (responsabilidades de cada paquete).
- Persistencia: estructura de la base, orden de operaciones y transacciones (dónde se hace commit/rollback).
- Validaciones y reglas de negocio.
- Pruebas realizadas (capturas del menú y consultas SQL útiles).
- Conclusiones y mejoras futuras.
- Citar fuentes y herramientas utilizadas (incluida IA, si la usaron).

3. Video (10–15 minutos)

- Presentación de los 4 integrantes con rostro visible.

- Demostración del flujo CRUD y de la relación 1→1 funcionando.
- Explicación por secciones de código (entities, dao, service, menú).
- Mostrar una operación transaccional y evidenciar el rollback ante un fallo simulado.

Criterios de evaluación

- Correctitud funcional (CRUD, 1→1 unidireccional real en código y base).
- Diseño/arquitectura (DAO/Service, responsabilidades claras, validaciones).
- Calidad de código (legibilidad, nombres, excepciones, PreparedStatement).
- Persistencia (integridad referencial, unicidad para 1→1, scripts SQL reproducibles).
- Documentación (README claro, informe sólido, UML coherente).
- Presentación (video dentro del tiempo, explicación técnica, participación equitativa).
- Entrega (repo público completo, proyecto compilable/ejecutable desde cero).

Checklist (sugerido)

- UML actualizado y consistente con el código.
- A → B 1→1 garantizado por FK única o PK compartida.
- CRUD de A y B con baja lógica.
- Service orquesta transacciones con commit/rollback.
- DAO acepta Connection externa.
- README.md con pasos para crear base y ejecutar, más enlace al video.
- Archivos SQL (creación + datos) incluidos y probados.
- Informe y UML dentro del repo.
- Menú de consola usable y con manejo de errores.

Especificación de campos – TP Integrador (1→1 unidireccional)

Se listan los campos obligatorios de cada par de clases (A → B) disponible para el TP Integrador. En todos los casos:

- La relación es unidireccional 1→1 desde A hacia B (A contiene una referencia a B).
- Ambas clases incluyen id (clave primaria) y eliminado (baja lógica).

Tip: Los nombres y longitudes son sugeridos; podés ajustarlos manteniendo la semántica, unicidad y la relación 1→1.

Mascota → Microchip

Clase Mascota (A)

Campo	Tipo (Java)	Reglas / Notas
Id	Long	PK
eliminado	Boolean	Baja lógica
nombre	String	NOT NULL, máx. 60
especie	String	NOT NULL, máx. 30
raza	String	máx. 60
fechaNacimiento	java.time.LocalDate	
duenio	String	NOT NULL, máx. 120
microchip	Microchip	Referencia 1→1 a B

Clase Microchip (B)

Campo	Tipo (Java)	Reglas / Notas
Id	Long	PK
eliminado	Boolean	Baja lógica
codigo	String	NOT NULL, UNIQUE, máx. 25
fechaImplantacion	java.time.LocalDate	NOT NULL, máx. 30
veterinaria	String	máx. 120
observaciones	String	máx. 255

Notas generales

- eliminado: usarlo para bajas lógicas (ocultar en listados, no borrar físicamente).
- Validar en Service los campos obligatorios y formatos (email, DNI, ISBN, dominio, CUIT, etc.).

Desarrollo

Elección del dominio y justificación

El dominio elegido para este proyecto es la gestión de mascotas y sus microchips, un escenario muy utilizado en veterinarias, refugios y registros municipales.

Justificación del dominio

- Permite modelar de forma clara una **relación 1→1** real entre entidades (Mascota–Microchip).
- Es un contexto simple pero con suficiente complejidad para aplicar:
    - Restricciones de unicidad
    - Soft delete
    - Validaciones de negocio
    - Operaciones CRUD completas
    - Búsquedas con filtros
    - Relaciones referenciales
    - Transacciones para mantener consistencia
- Es lo suficientemente flexible como para agregar futuras ampliaciones (dueños, vacunas, veterinarias, historial clínico, etc.)

El dominio se adapta perfectamente a los requisitos del TPI y permite demostrar el uso correcto del patrón DAO, manejo de JDBC, validaciones, modelo de capas, arquitectura limpia y manejo de transacciones.

Diseño: decisiones clave (1→1, FK única vs PK compartida) + UML.

1. Decisiones clave del modelo de datos

Durante el análisis del dominio “Mascota – Microchip” se identificó que la relación entre ambas entidades es uno a uno (1 → 1), con las siguientes características:

- Una mascota puede poseer cero o un microchip.
- Un microchip solamente puede pertenecer a una mascota.
- Un microchip no puede reasignarse a otra mascota una vez implantado (restricción real del dominio y reforzada mediante un TRIGGER).

Se optó por un diseño FK única (mascotas.microchip_id) en lugar de PK compartida o una tabla intermedia de vinculación.

La FK única fue elegida porque:

1. Refleja de manera natural la regla: *“una mascota puede o no tener microchip”* → FK **nullable**.
2. Permite mantener a “microchips” como entidad independiente, con vida propia, útil para pre-carga o stock.
3. Evita complejidad innecesaria de una tabla puente, ya que no existe multiplicidad.
4. Facilita operaciones CRUD simples sin sobrecargar el lado del microchip.

Restricción anti-reasignación

Se implementó un **trigger BEFORE UPDATE** en la tabla *microchips* que impide modificar el atributo `mascota_id` cuando ya tiene un valor asignado.

Esto refuerza la lógica del dominio: *Un microchip implantado no puede reasignarse*

2. Decisiones del diseño orientado a objetos

El sistema se organizó aplicando principios de diseño OO:

- Uso de clase abstracta `Base`

- Provee atributos y comportamiento común: `id` y `eliminado`.
- Evita duplicación y centraliza responsabilidad.

- DAO Pattern por entidad

- `MascotaDAO` y `MicrochipDAO`.
- Uso sistemático de PreparedStatements.
- Manejo estructurado de JOIN, INSERT, UPDATE, DELETE lógico.

- Service Layer

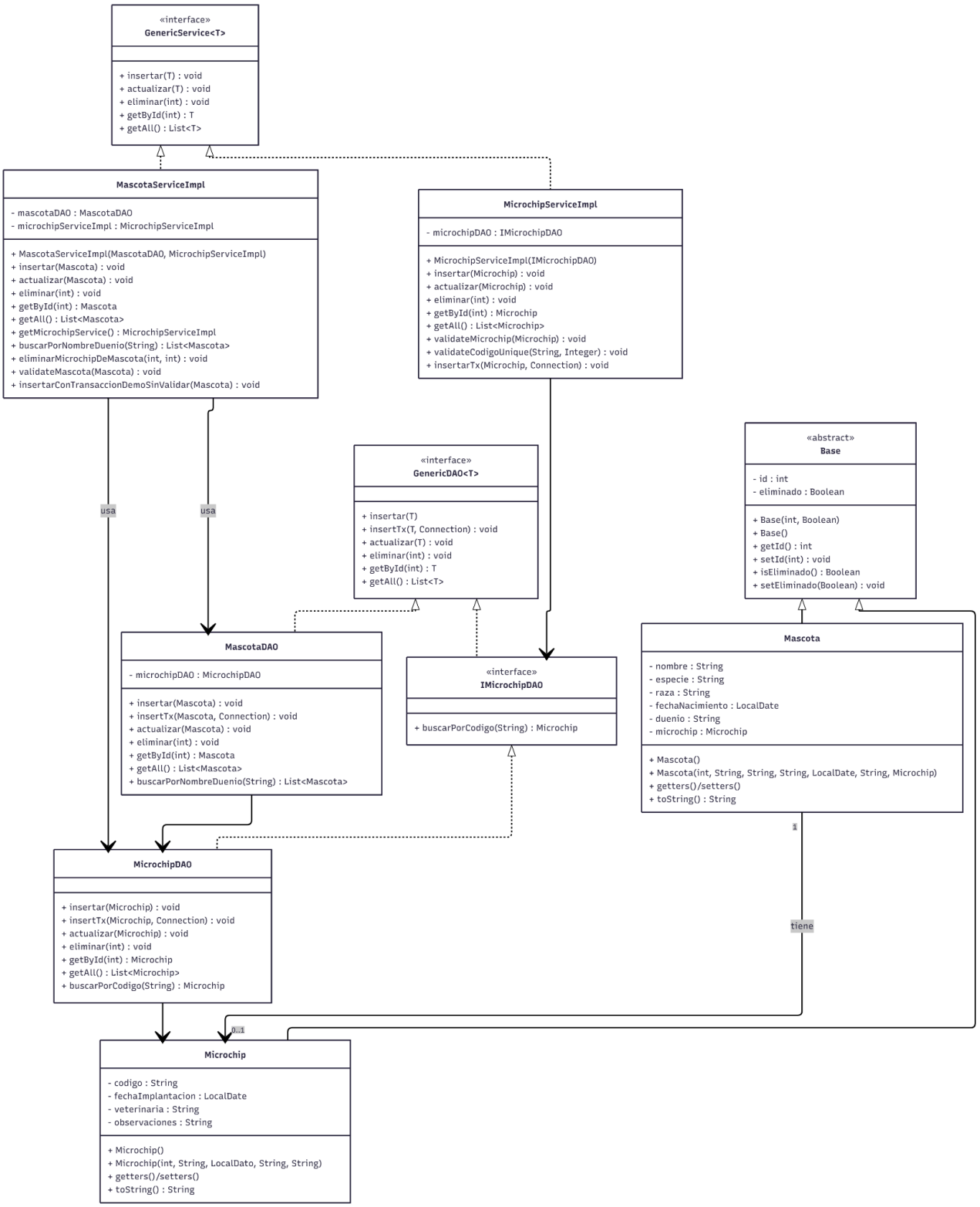
- Aplica reglas de negocio.
- Orquesta acciones entre DAOs.
- Implementa operaciones transaccionales seguras.

- Capa de Presentación

- Consola por menú (`AppMenu` + `MenuHandler`).
- Input validado.
- Lógica de negocio completamente aislada.

3. Diagrama UML del sistema

A continuación se presenta el diagrama UML completo utilizado como base del desarrollo:



4. Justificación del diseño general

El diseño final fue guiado por tres objetivos:

1. Claridad estructural

- Entidades bien separadas.
- Relaciones claramente definidas en OOP y en la BD.
- Código mantenible gracias a capas estrictas.

2. Robustez y consistencia

- Soft delete en todas las entidades para preservar histórico.
- Validaciones en la capa Service.
- Integridad referencial con FK única + trigger anti-reasignación.

3. Extensibilidad

La arquitectura permite ampliar fácilmente el proyecto, por ejemplo:

- agregar historial veterinario,
- múltiples dueños,
- más reglas de negocio,
- interfaz gráfica futura.

Arquitectura por capas (responsabilidades de cada paquete)

El sistema fue desarrollado siguiendo una arquitectura en cuatro capas, separando completamente la lógica de presentación, negocio, persistencia y modelo. Este enfoque mejora la mantenibilidad, escalabilidad y claridad del proyecto.

A continuación se describen las responsabilidades de cada capa y de cada paquete del proyecto.

1. Capa de Presentación (UI)

Paquete: `Main`

Clases principales:

- `AppMenu`
- `MenuHandler`

Responsabilidad

Esta capa contiene toda la interacción con el usuario a través de la consola.
Sus tareas son:

A. Mostrar el menú y pedir datos al usuario.
B. Validar que el input sea correcto en cuanto a tipo (enteros, fechas, strings no vacíos).
C. Delegar toda la lógica de negocio a los servicios.
D. Mostrar mensajes de éxito, error o información.
E. No contiene ninguna lógica de negocio ni lógica de base de datos.

Notas

- `AppMenu` inicializa todas las dependencias (DAOs → Services → MenuHandler).
- `MenuHandler` contiene el flujo de cada operación del menú:

2. Capa de Negocio (Service Layer)

Paquete: `Service`

Clases principales

- `MascotaServiceImpl`
- `MicrochipServiceImpl`
- `GenericService<T>` (interfaz)

Responsabilidad

La capa de servicios implementa todas las reglas de negocio del sistema. Su función principal es orquestar la lógica entre la UI y los DAOs.

Funciones clave:

- Validar datos de entrada antes de enviarlos al DAO.

- Aplicar reglas del dominio:
  - nombre, especie y dueño obligatorios,
  - código de microchip único,
  - microchip no reasignable,
  - evitar inconsistencias.

- Coordinar inserciones entre entidades relacionadas:
  - insertar primero microchip,
  - luego mascota con FK correcta.

- Implementar transacciones:
    - rollback en caso de fallos,
    - commit solo si todo es correcto.

- Exponer métodos para el menú (búsquedas, cambios, eliminaciones seguras).

Notas

- La UI nunca llama directamente a los DAOs.
- Toda excepción técnica se convierte en un mensaje entendible para la UI.

3. Capa de Persistencia (DAO Layer)

Paquete: `Dao`

Clases principales

- `MascotaDAO`
- `MicrochipDAO`
- `GenericDAO<T>` (interfaz)
- `IMicrochipDAO` (interfaz)

Responsabilidad

Los DAOs encapsulan todo el acceso a la base de datos mediante JDBC:

Funciones clave:

- Ejecutar consultas SQL usando `PreparedStatement`.

- Mapear filas del `ResultSet` a objetos Java (Model).

- Insertar, actualizar y eliminar lógicamente registros.

- Implementar búsquedas:
    - por ID,
    - por nombre o dueño,
    - por código de microchip.

- Implementar `LEFT JOIN` para cargar relaciones 1 -> 1.

- Manejar operaciones dentro o fuera de una transacción.

Notas

- DAO no valida reglas de negocio; solo ejecuta SQL.
- El menú no accede al DAO directamente, sólo a través del Service.

4. Capa de Modelo (Domain Model)

Paquete: `Models`

Clases principales

- `Mascota`
- `Microchip`
- `Base` (abstracta)

Responsabilidad

Representa las entidades del dominio, sus atributos y su identidad.

Funciones clave:

- Encapsular el estado (id, eliminado, propiedades).
- Proveer métodos de acceso (get/set).
- Compartir atributos comunes mediante la clase base:
    - `id`,
    - `eliminado`.

Notas

- No contiene SQL ni lógica de negocio.
- Solo estructura y representación de datos reales del dominio.

Modulo de Infraestructura

Paquete: `Config`

Clases principales

- `DatabaseConnection`
- `TransactionManager`

Responsabilidad

Brinda soporte técnico a las capas superiores:

DatabaseConnection

- Gestiona parámetros de conexión.
- Provee conexiones JDBC mediante un método estático.
- Valida configuración desde el arranque.

TransactionManager

- Controla transacciones manuales:
    - `startTransaction()`,
    - `commit()`,
    - `rollback()`,
    - `close()` con rollback implícito.
- Garantiza atomicidad, consistencia y reversión completa.

Persistencia: estructura de la base, orden de operaciones y transacciones (dónde se hace commit/rollback).

La persistencia del sistema se implementa con MySQL, utilizando dos tablas: `mascotas` y `microchips`. La estructura está normalizada y refleja directamente la relación del dominio: una mascota puede tener cero o un microchip, mientras que cada microchip solo puede pertenecer a una mascota. Para asegurar esto, la clave foránea `microchip_id` en la tabla `mascotas` está acompañada de una restricción UNIQUE, lo que garantiza una relación 1→1 real.

La tabla `microchips` almacena información básica como el código del chip (único y obligatorio), la fecha de implantación, la veterinaria y observaciones. La tabla `mascotas` contiene nombre, especie, raza, fecha de nacimiento y dueño, además del `microchip_id`. Ambas incluyen el campo `eliminado`, que se utiliza para implementar un soft delete: en lugar de borrar físicamente un registro, se lo marca como eliminado. Todas las consultas de lectura consideran únicamente registros con `eliminado = FALSE`, lo cual preserva coherencia histórica y evita problemas con claves foráneas.

Orden de operaciones

Para mantener la integridad referencial, la inserción de una mascota que incluye microchip sigue un orden estricto. Primero se valida y guarda el microchip (si es nuevo), obteniendo su ID generado automáticamente por la base. Recién después se inserta la mascota utilizando ese ID

como clave foránea. Si el microchip ya existe, se actualiza antes de continuar. Este flujo asegura que nunca se intente guardar una mascota con una referencia inválida.

Uso de transacciones

El proyecto incluye manejo explícito de transacciones a través del componente `TransactionManager`. Este módulo controla cuándo se inicia (`setAutoCommit(false)`), cuándo se confirma (`commit()`), y cuándo se revierte (`rollback()`). La transacción se utiliza principalmente en la funcionalidad de demostración de rollback, cuyo objetivo es mostrar cómo el sistema puede revertir automáticamente cambios incompletos.

En esta demostración, se ingresan primero los datos del microchip y este se inserta dentro de una transacción sin ser confirmado todavía. Luego se intenta insertar la mascota. Si alguno de los datos de la mascota es inválido (por ejemplo, un dueño vacío que viola un CHECK de la base de datos), ocurre una excepción. Debido a que ambos inserts se realizan dentro de la misma transacción, el `TransactionManager` detecta el error al cerrarse y ejecuta el rollback correspondiente. Como resultado, ni la mascota ni el microchip quedan registrados en la base.

Esta operación permite visualizar claramente el concepto de atomicidad: o se realizan todas las operaciones correctamente, o ninguna. Es un buen ejemplo práctico para mostrar en el video final.

Validaciones y reglas de negocio.

El sistema aplica un conjunto de validaciones distribuidas entre la interfaz de usuario (capa de presentación), la capa de servicios y la propia base de datos. Este enfoque asegura que los datos ingresados sean consistentes, válidos y coherentes con el dominio, evitando errores de persistencia y garantizando integridad semántica.

Validaciones en la capa de presentación (MenuHandler)

La consola realiza verificaciones básicas antes de enviar los datos al servicio. Su objetivo principal es mejorar la experiencia del usuario y evitar errores evidentes.

- Los campos obligatorios como **nombre**, **especie** y **dueño** solicitan reingreso si se presionó Enter vacío.
- Los campos opcionales, como la raza o la fecha de nacimiento, permiten ingresar vacío para dejar el valor en `null`.
- Las fechas se validan y, si el formato es incorrecto, se ignora el nuevo valor y se conserva el anterior.

Esta capa evita que el usuario pueda continuar con datos incompletos o mal formateados, mejorando la robustez del proceso de carga.

Validaciones en la capa de servicio (Service Layer)

La capa de servicios implementa las reglas de negocio reales del sistema. Aquí no se validan cuestiones estéticas sino condiciones estrictas que garantizan coherencia y validez del dominio.

Principales reglas:

Mascotas

- `nombre`, `especie` y `dueño` deben existir y no estar vacíos.
- El ID debe ser mayor a 0 para operaciones de actualización o eliminación.
- Una mascota solo puede tener **0 o 1 microchip** asignado.
- Para eliminar un microchip, primero debe actualizarse la mascota para eliminar la referencia (eliminación segura).

Microchips

- El **código** es obligatorio y debe ser único en todo el sistema.
- En una actualización, si el código coincide con otro microchip diferente, la operación se rechaza.
- El ID debe ser mayor a 0 para actualizaciones y eliminaciones.

La capa de servicios también coordina operaciones complejas, como crear una mascota con microchip o eliminar microchips sin dejar claves foráneas huérfanas.

Validaciones en la base de datos (SQL + Constraints)

La base de datos refuerza las reglas más críticas mediante restricciones:

- `chk_duenio_no_vacio`, `chk_nombre_no_vacio`, `chk_especie_no_vacio`: aseguran que campos obligatorios no queden vacíos.
- `UNIQUE (microchip_id)` en mascotas garantiza la relación **1→1** con microchips.
- `UNIQUE (codigo)` en microchips evita duplicados.
- Las claves foráneas con `ON DELETE RESTRICT` impiden eliminar microchips usados.
- Valores booleanos para `eliminado` son validados por `CHECK (eliminado IN (0,1))`.

Además, el sistema utiliza **soft delete**, por lo cual nunca se pierden datos físicamente, sino que se marcan como inactivos.

Integración entre capas

Las validaciones están distribuidas de forma complementaria:

- **Presentación**: evita errores previsibles y mejora experiencia.
- **Servicio**: aplica reglas de negocio estrictas y coordinación entre entidades.
- **Base de datos**: garantiza integridad referencial y consistencia estructural.

Cuando alguna validación falla, la operación se interrumpe en la capa correspondiente, impidiendo que datos incorrectos lleguen a una capa inferior. En transacciones, una violación en la base desencadena el mecanismo de rollback asegurando atomicidad.

Conclusiones y mejoras futuras.

Conclusiones

El desarrollo del sistema “Mascotas & Microchips” permitió aplicar de forma integrada todos los conceptos trabajados en la materia: modelado relacional, diseño orientado a objetos, consultas SQL, uso de JDBC, arquitectura por capas y manejo de transacciones. El resultado es una aplicación robusta que:

- Implementa una **relación 1 a 1** correctamente diseñada y sustentada por constraints (`UNIQUE` y `FK`).
- Distribuye responsabilidades de manera clara entre capas (Model–DAO–Service–Controller).
- Garantiza la **consistencia de los datos** mediante validaciones en tres niveles: presentación, servicio y base de datos.
- Utiliza correctamente el **soft delete**, preservando historial e integridad.
- Demuestra el uso de **transacciones**, asegurando atomicidad y rollback ante fallos.
- Permite una interacción clara y simple mediante un menú de consola bien estructurado.

En conjunto, el proyecto constituye un ejemplo realista de cómo una aplicación pequeña puede mantener una arquitectura limpia, extensible y coherente, incluso sin frameworks externos.

Mejoras futuras

Aunque el sistema cumple con todos los requisitos del trabajo práctico, existen diversas mejoras que podrían implementarse en versiones posteriores:

1. Integración con interfaz gráfica (GUI)

Reemplazar el menú por una interfaz gráfica (JavaFX, Swing o web) permitiría una interacción más amigable y profesional.

2. Manejo centralizado de errores

Crear un módulo dedicado para gestionar mensajes de error y logs, evitando que la consola mezcle mensajes del sistema con los del menú.

3. Validador de negocio más completo

Agregar reglas adicionales, tales como:

- Validar formatos de código del microchip (ej: “MC-XXXX”).
- Opcionalmente restringir fechas futuras o edades imposibles en mascotas.
- Validar longitud máxima de campos antes de enviarlos a la base.

4. Sistema de reportes

Agregar funcionalidades como:

- Listado de mascotas por especie.
- Mascotas sin microchip.
- Microchips disponibles.

Esto permitiría trabajar consultas más ricas para administración.

Fuentes y herramientas utilizadas.

Herramientas de desarrollo

Para llevar adelante el proyecto se utilizaron las siguientes herramientas:

- **Apache NetBeans 21**
- **Java SE 17**
- **MySQL Community Server 8.x + MySQL Workbench**
- **Git y GitHub**
- **Mermaid.js**

Fuentes consultadas

Durante el desarrollo se consultaron distintos materiales para reforzar conceptos y resolver dudas técnicas:

- Documentación oficial de **Java SE 17**: https://docs.oracle.com/en/java/javase/17/
- Documentación oficial de **MySQL 8** (Foreign Keys, Constraints, DATE, UNIQUE y diseño de schemas): https://dev.mysql.com/doc/
- Guía de patrones de diseño y organización en capas (DAO–Service–Controller)
- Material teórico de la cátedra, videos explicativos y ejemplos de trabajos previos.
- Foros de referencia general (StackOverflow, GeeksForGeeks) para dudas puntuales sobre:
    - PreparedStatement
    - JDBC y manejo de transacciones
    - Soft delete y modelado de relaciones 1 -> 1
    - Validaciones en capa DAO vs. Service

Uso de herramientas de asistencia (IA)

Durante el desarrollo se utilizó **ChatGPT** como herramienta de apoyo para:

- Revisar y depurar SQL (especialmente restricciones 1->1).
- Ayudar en la generación de código repetitivo (especialmente en DAOs).
- Resolver dudas conceptuales de arquitectura.
- Alinear el diseño UML con el modelo real.
- Redactar documentación y mejorar claridad del informe.

Todas las decisiones técnicas y correcciones finales fueron validadas manualmente y probadas directamente sobre el código y la base de datos.

Link al repositorio:

https://github.com/LucasZupan/TPI-Prog2-BD