# Informe del Proyecto: Mascotas y Microchips

---

## Integrantes y roles

- **Federico Savastano:** Puntos 1 y 2 (Diseño UML y desarrollo de entidades).
- **Fabiana Rossetto:** Puntos 3 y 4 (Base de datos y capa DAO).
- **Gianina Azcurra:** Puntos 5 y 6 (Capa Service y AppMenu).

---

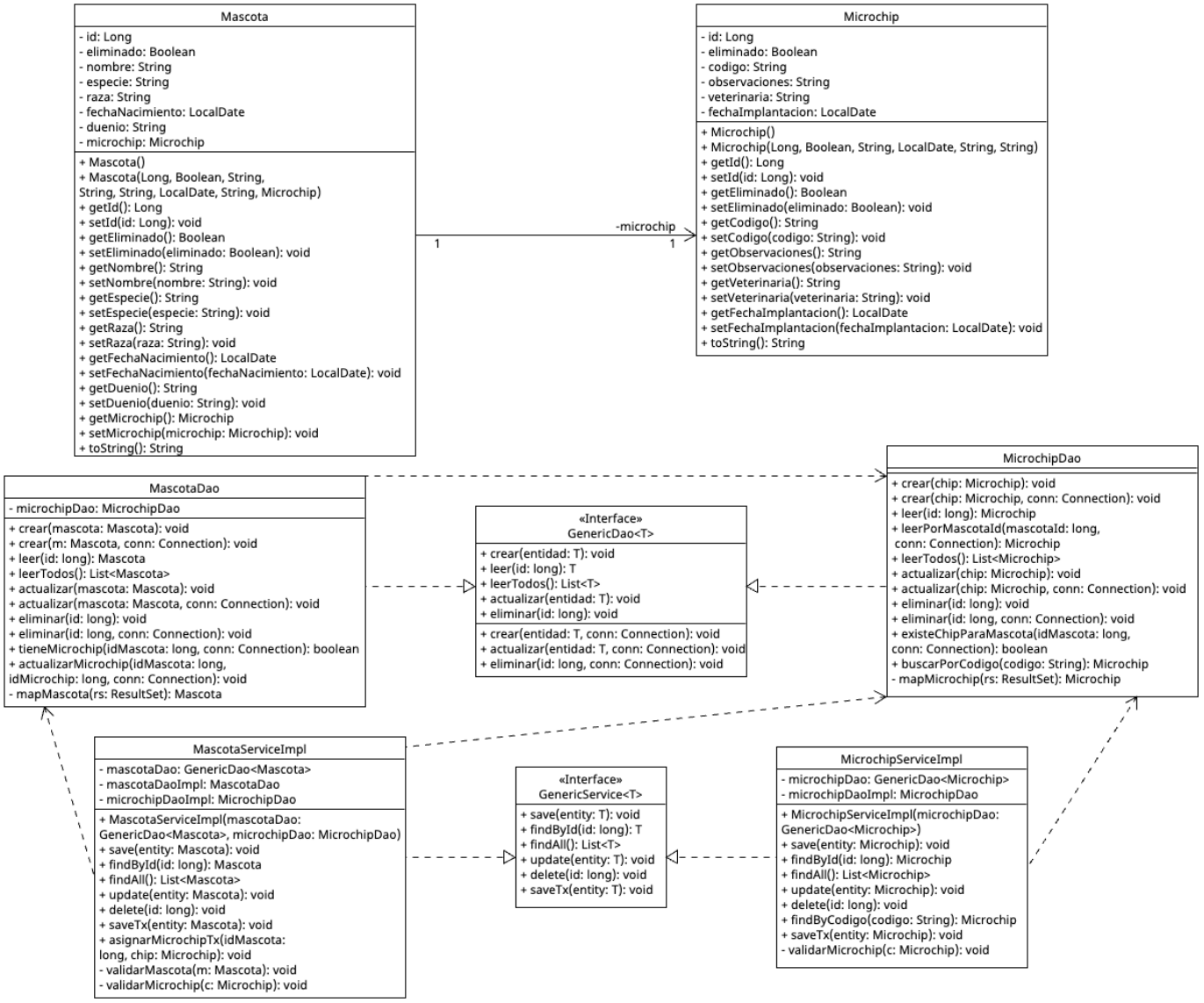
## Elección del dominio y justificación

El dominio elegido es el de **Mascotas y Microchips**, basado en la necesidad de registrar y administrar la información de animales domésticos junto con su correspondiente microchip identificador.
Esta temática permite aplicar una relación **uno a uno (1→1)** entre las entidades, donde cada mascota tiene asociado un único microchip, lo cual es ideal para practicar relaciones de integridad referencial y manejo transaccional.
Además, es un dominio claro, aplicable al mundo real y suficientemente complejo para incluir validaciones, consultas y manejo de errores.

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

La principal decisión de diseño para la relación uno a uno unidireccional (Mascota a Microchip) fue implementar la relación usando una **Clave Foránea (FK) única** en lugar de una Clave Primaria (PK) compartida. Optamos por colocar la FK en la **tabla Mascota (Clase A)**, a través de una columna **Id_microchip**. Para garantizar la estricta relación 1:1, se aplicó una restricción **UNIQUE** sobre esta columna en la base de datos. Esto asegura que un microchip no pueda ser asignado a más de una mascota. A nivel de código Java, la unidireccionalidad se mantiene estrictamente: solo la entidad Mascota posee una referencia al objeto Microchip.

**UML clases:**




## Arquitectura por capas (responsabilidades de cada paquete)

La aplicación sigue una **arquitectura en capas**, dividiendo responsabilidades claramente:

- **config/**
  Contiene la clase DatabaseConnection encargada de manejar la conexión con la base de datos MySQL y centralizar la lectura del archivo db.properties.
  También se incluyen clases de prueba como TestDatabase y TestDao.

- **entities/**
  Incluye las clases de entidades del dominio (Mascota y Microchip), con sus atributos, constructores, métodos getters/setters y toString().

Aquí se define la relación **1→1 unidireccional**, donde una Mascota contiene una referencia a Microchip.

- **Dao/**
  Define la interfaz genérica GenericDao<T> con las operaciones CRUD básicas:
  crear(T), leer(long id), leerTodos(), actualizar(T) y eliminar(long id).
- **DaoImplement/**
  contiene las implementaciones concretas del DAO:
    - MascotaDao: maneja las operaciones de persistencia de mascotas.
    - MicrochipDao: maneja las operaciones del microchip
- **default package/**
  Contiene el archivo de configuración db.properties con los parámetros de conexión (URL, usuario, contraseña, driver).
- **SQL/**

  Contiene un script utilizado para crear la base de datos y sus tablas con las restricciones pertinentes(schema), y otro para probarla(Data)

## Persistencia: estructura de la base, orden de operaciones y transacciones

## Estructura de la base de datos

Base de datos: mascotas_chips
Relación principal: **Mascota (A)** → **Microchip (B)** en una relación **1→1 unidireccional**.

Tablas principales:

### Mascota

| Campo | Tipo | Descripción |
|---|---|---|
| Id | INT AUTO_INCREMENT | Identificador único |
| Nombre | VARCHAR(60) | Nombre de la mascota |
| Especie | ENUM('PERRO','GATO','AVE','REPTIL','OTRO') | Tipo de animal |
| Raza | VARCHAR(60) | Raza de la mascota |
| FechaNacimiento | DATE | Fecha de nacimiento |
| Duenio | VARCHAR(120) | Nombre del dueño |
| Eliminado | TINYINT(1) | Eliminación lógica |

#### Microchip

| Campo | Tipo | Descripción |
|---|---|---|
| Id | INT AUTO_INCREMENT | Identificador único |
| Codigo | VARCHAR(25) UNIQUE | Código del microchip |

| | | |
|---|---|---|
| FechaImplantacion | DATE | Fecha de implantación |
| Veterinaria | VARCHAR(120) | Nombre de la veterinaria |
| Observaciones | VARCHAR(255) | Información adicional |
| Id_mascota | INT UNIQUE | Relación 1→1 con Mascota(Id) |
| Eliminado | TINYINT(1) | Eliminación lógica |

La relación 1→1 se garantiza con una **clave foránea única (Id_mascota)** en la tabla Microchip, referenciando a Mascota(Id) con la regla ON DELETE CASCADE.

## Orden de operaciones

El orden de inserción es importante para mantener la integridad referencial:

1. Crear primero la **Mascota** (para obtener su Id generado).
2. Luego, crear el **Microchip** asignando su campo Id_mascota con el Id de la mascota creada.

Las operaciones de lectura, actualización y eliminación siguen este mismo orden jerárquico.

## Transacciones

En las clases DAO, las operaciones de escritura (insertar, actualizar, eliminar) utilizan una conexión compartida cuando participan de la misma transacción.
Los commit() y rollback() se manejan en la capa de servicio (que aún no se implementó).
Por ahora, el control transaccional básico se realiza en las clases de prueba (TestDao) mediante conn.setAutoCommit(false) y conn.commit() o conn.rollback() según corresponda.

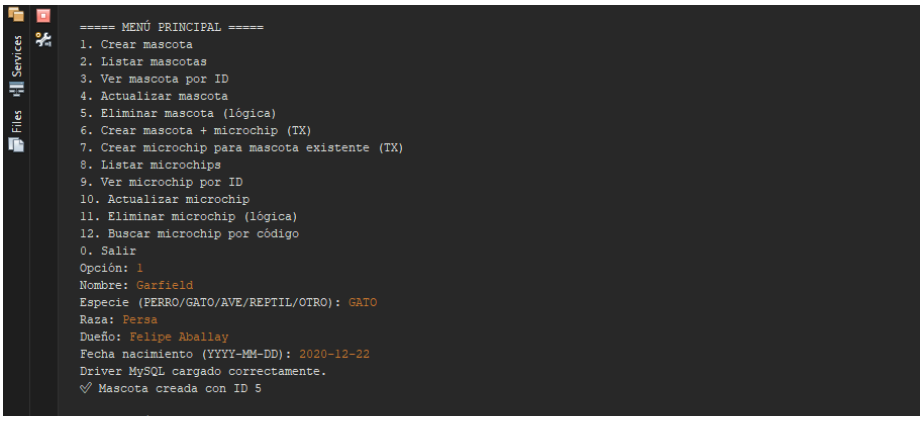
## Validaciones y reglas de negocio

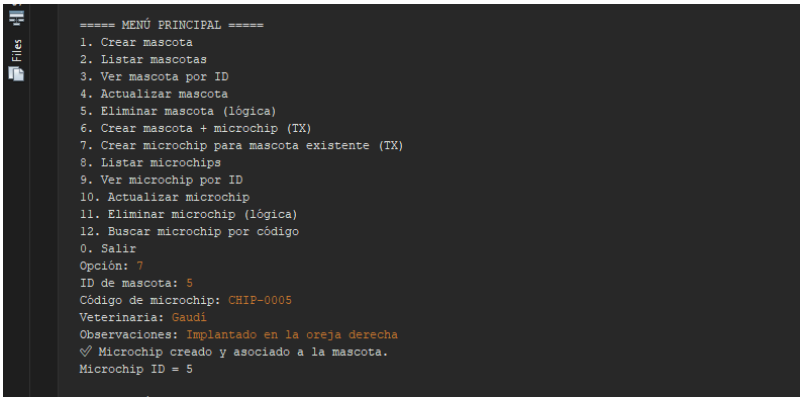
El presente proyecto implementa un sistema de gestión para dos entidades vinculadas: **Mascota** y **Microchip**, las cuales mantienen una relación uno a uno. El objetivo principal es permitir la administración completa de ambas entidades mediante operaciones CRUD, garantizando al mismo tiempo la integridad de los datos a través de transacciones, validaciones y reglas de negocio coherentes. Para lograrlo se utiliza una arquitectura por capas, donde cada capa tiene una responsabilidad bien definida y aislada del resto.

La capa de presentación está compuesta por la clase **AppMenu**, que actúa como interfaz de usuario por consola. Su función es únicamente recibir los datos ingresados por el usuario, mostrar listas, permitir búsquedas y derivar las solicitudes hacia las capas inferiores. Es importante destacar que el menú **no implementa lógica**, no realiza validaciones complejas y nunca accede directamente a la base de datos; su único propósito es invocar a los métodos de la capa de servicios con la información provista por el usuario.

La capa de **servicios** constituye el núcleo del sistema. Allí se concentran todas las **reglas de negocio**, las **validaciones** y especialmente la **gestión de transacciones**. Cada operación compleja, como la creación simultánea de una Mascota y su Microchip, se implementa dentro de métodos transaccionales como saveTx(), los cuales utilizan setAutoCommit(false) para iniciar la transacción y garantizan que todas las acciones se apliquen de manera atómica. Si alguna parte del proceso falla, ya sea por error SQL o por violación de una regla de negocio, se ejecuta un rollback() que revierte todos los cambios, evitando dejar datos inconsistentes. Si todo sale correctamente,

la operación finaliza con un commit() y los cambios quedan confirmados. Los servicios también son responsables de validar los campos obligatorios de cada entidad (por ejemplo, nombre y especie en Mascota o código y fecha de implantación en Microchip), y de aplicar reglas específicas como impedir que una mascota tenga más de un microchip asignado. Se adjuntan a continuación capturas de pantalla correspondientes a la creación de una mascota y la posterior transacción involucrando la creación de un microchip para la misma:




La capa de **acceso a datos (DAO)** se encarga exclusivamente de comunicarse con la base de datos mediante JDBC. Los DAO ejecutan instrucciones SQL para crear, actualizar, listar y eliminar registros (en este proyecto, la eliminación es lógica mediante un campo booleano). También incluyen métodos de consulta específicos, como la búsqueda de microchips por código y la verificación de si una mascota ya posee un microchip asignado. Es importante mencionar que los DAO **no realizan validaciones ni manejan transacciones**; su misión es únicamente ejecutar SQL y mapear los datos entre la base de datos y los objetos Java.

En cuanto al manejo de transacciones, el proyecto implementa dos escenarios principales. El primero es la creación de una **Mascota junto con su Microchip** dentro de una misma operación compuesta. Este proceso se realiza dentro de MascotaServiceImpl.saveTx(), donde ambos objetos se insertan de manera coordinada, asegurando que la mascota no quede creada sin un microchip o viceversa. El segundo escenario es la creación de un microchip para una mascota ya existente, implementado en MicrochipServiceImpl.saveTx(). En este caso, la regla de negocio más importante es la validación de la relación uno a uno, que impide asignar más de un chip a la misma mascota. En ambos casos las transacciones protegen la coherencia de los datos.

Finalmente, el proyecto incorpora una **búsqueda por un campo relevante**, requisito solicitado en la consigna. Para ello se implementó la búsqueda de microchips por su **código único**, permitiendo identificar rápidamente un chip registrado. Esta búsqueda se implementa desde el menú, se procesa en el servicio y se ejecuta mediante un método específico en el DAO.

En conjunto, el sistema cumple con todos los requisitos funcionales y técnicos previstos: CRUD completo para ambas entidades, operaciones transaccionales con manejo adecuado de commit y rollback, validaciones de campos obligatorios, aplicación de reglas de negocio claras, eliminación lógica, arquitectura por capas bien separada y búsqueda por campo relevante. Esto da como resultado una aplicación robusta, organizada y coherente con los principios de diseño profesional.

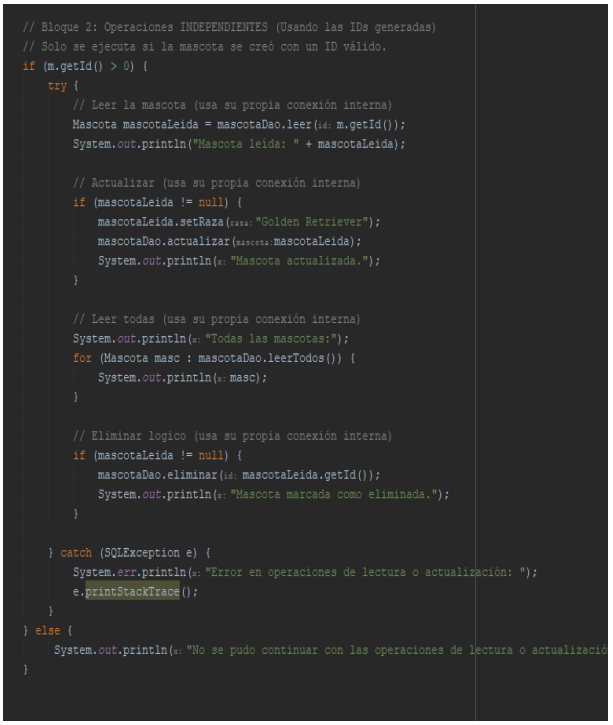
## Pruebas realizadas (capturas del menú y consultas SQL útiles)

Durante las pruebas en consola (desde `TestDao`), se realizaron las siguientes verificaciones:

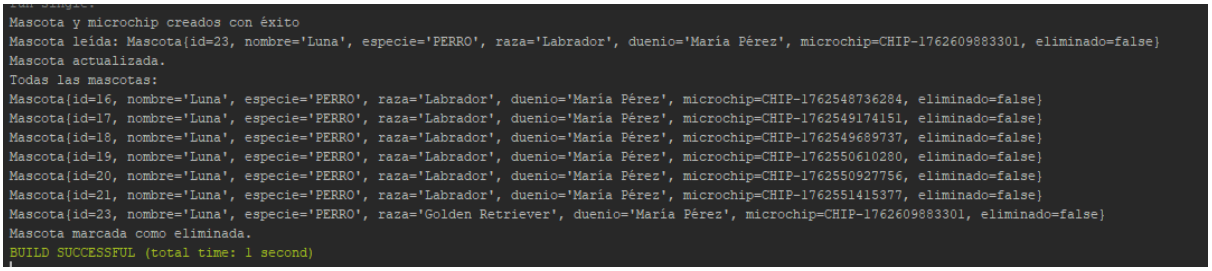
- Conexión exitosa a la base de datos (DatabaseConnection).
- Inserción correcta de una mascota con su microchip asociado en una misma transacción.
- Lectura de mascota por ID incluyendo su microchip relacionado.
- Actualización de datos de la mascota.
- Eliminación lógica (cambio del campo Eliminado a true).
- Verificación de restricciones UNIQUE (código de microchip y Id_mascota).

Consultas SQL útiles utilizadas:

```
MascotaDao mascotaDao = new MascotaDao();
MicrochipDao microchipDao = new MicrochipDao();

// declaré las entidades fuera de cualquier bloque try-catch.
Microchip chip = null;
Mascota m = new Mascota();

// Bloque 1: Transacción para la CREACIÓN (usa conexión transaccional)
try (Connection conn = DatabaseConnection.getConnection()) {
    conn.setAutoCommit(bln:false);

    // Crear un microchip
    chip = new Microchip(
            id: null, eliminado:false,
            "CHIP-" + System.currentTimeMillis(),
            fechaImplantacion: LocalDate.of(year:2025, month: 1, dayOfMonth: 1),
            veterinaria:"VetPlus",
            observaciones: "Implantado correctamente"
    );

    microchipDao.crear(chip, conn);

    // Crear y setear la mascota 'm'
    m.setNombre(nombre: "Luna");
    m.setEspecie(especie:"PERRO");
    m.setRaza(raza:"Labrador");
    m.setFechaNacimiento(fechaNacimiento:LocalDate.of(year:2020, month: 5, dayOfMonth: 15));
    m.setDuenio(duenio: "María Pérez");
    m.setMicrochip(microchip: chip);
    m.setEliminado(eliminado: false);

    // Esto llama a m.setId() con el ID generado, y 'm' es accesible afuera
    mascotaDao.crear(m, conn);

    conn.commit();

    System.out.println(x: "Mascota y microchip creados con éxito");
    } catch (SQLException e) {
    System.err.println(x: "Error en la transacción de creación: ");
    e.printStackTrace();
    return; // Salir si la creación falla.
}
```



```
// Bloque 2: Operaciones INDEPENDIENTES (Usando las IDs generadas)
// Solo se ejecuta si la mascota se creó con un ID válido.
if (m.getId() > 0) {
    try {
        // Leer la mascota (usa su propia conexión interna)
        Mascota mascotaLeida = mascotaDao.leer(id: m.getId());
        System.out.println("Mascota leida: " + mascotaLeida);

        // Actualizar (usa su propia conexión interna)
        if (mascotaLeida != null) {
            mascotaLeida.setRaza(raza:"Golden Retriever");
            mascotaDao.actualizar(mascota:mascotaLeida);
            System.out.println(x: "Mascota actualizada.");
        }

        // Leer todas (usa su propia conexión interna)
        System.out.println(x: "Todas las mascotas:");
        for (Mascota masc : mascotaDao.leerTodos()) {
            System.out.println(x: masc);
        }

        // Eliminar logico (usa su propia conexión interna)
        if (mascotaLeida != null) {
            mascotaDao.eliminar(id: mascotaLeida.getId());
            System.out.println(x: "Mascota marcada como eliminada.");
        }

    } catch (SQLException e) {
        System.err.println(x: "Error en operaciones de lectura o actualización: ");
        e.printStackTrace();
    }
} else {
    System.out.println(x: "No se pudo continuar con las operaciones de lectura o actualizació
}
```




## Conclusiones

El proyecto logró implementar correctamente un modelo 1:1 entre Mascota y Microchip, con una arquitectura basada en capas DAO–Service y validaciones que garantizan la integridad del dominio.
Se desarrolló una conexión a base de datos mediante JDBC, junto con operaciones CRUD completas, consultas específicas, manejo de excepciones y una implementación transaccional en el servicio de microchips que asegura el cumplimiento de la regla de negocio principal: una mascota solo puede tener un microchip.

Además, se integraron pruebas básicas de conexión y manipulación de datos que permiten verificar el correcto funcionamiento del sistema.
El diseño final es claro, modular y fácilmente extensible para futuras funcionalidades.

## Mejoras Futuras

Aunque el proyecto cumple con los requisitos planteados, existen oportunidades de mejora que podrían optimizar rendimiento, escalabilidad y experiencia de uso:

**1. Implementación de un Pool de Conexiones (HikariCP)**

Actualmente la conexión se maneja directamente con JDBC. Integrar **HikariCP** permitiría:

- Reducir el tiempo de apertura de conexiones.
- Evitar cuellos de botella en escenarios de mayor carga.
- Gestionar conexiones de forma más eficiente y segura.

**2. Separación de Interfaces y DAOs Específicos**

Ampliar la estructura para que los DAOs específicos no dependan directamente del DAO genérico y puedan manejar consultas personalizadas sin realizar castings.

**3. Pruebas Unitarias y de Integración**

Agregar tests con JUnit para:

- Validaciones de servicios.
- Comprobación automática del cumplimiento de la regla 1:1.
- Verificación de transacciones (commit/rollback).

**4. Manejo Centralizado de Errores**

Crear una clase de utilidades o un manejador de excepciones para reducir repetición de mensajes y mejorar la consistencia de la capa Service.

**5. Interfaz de Usuario Mejorada**

Agregar una interfaz gráfica o migrar a una API REST en el futuro permitiría utilizar el sistema de forma más intuitiva.

---

## Citar fuentes y herramientas utilizadas

- **Lenguaje:** Java 17
- **IDE:** Apache NetBeans
- **Base de datos:** MySQL Workbench
- **Control de versiones:** GitHub
- **Herramientas de diseño:** Draw.io (para UML)
- **Asistencia de desarrollo:** ChatGPT (OpenAI) / Gemini