UTN

Programación II

Trabajo Final Integrador

Enlace al video

Emanuel Aaron Brahim Pollini - Comisión 12
Arturo Alonso Kaadú - Comisión 15
Agustin Monardes Casas - Comisión 8
Belén Yarde Buller - Comisión 3

17 de noviembre del 2025

Resumen ejecutivo

El presente trabajo tiene como objetivo desarrollar un sistema administrativo y aplicar conceptos de arquitectura en capas, acceso a datos mediante DAO, validación de reglas de negocio y persistencia con operaciones transaccionales. Se eligió el dominio Paciente → Historia Clínica, basado en una relación 1→1 unidireccional, lo cual permite representar una situación habitual del entorno médico donde cada paciente posee solo una historia clínica, y esta es referenciada de forma independiente para mantener consistencia en accesos y actualizaciones.

Integrantes y roles

Para organizar el desarrollo del equipo, se decidió estructurar el proyecto en cinco capas, a través de los directorios config, entities, dao, service, y main. Esto logra que cada integrante pueda colaborar sin interferir en las responsabilidades de otros.

La división del equipo se realizó en base a los cuatro integrantes del grupo. Arturo Kaadú estuvo a cargo del punto 1, definiendo la arquitectura en capas, la estructura base del proyecto y el diagrama UML que guía la implementación. Belén Yarde Buller desarrolló los puntos 2 y 3, correspondientes a la selección del dominio, justificación del modelo y la creación del esquema relacional junto con los scripts de base de datos. Emanuel Brahim se ocupó de crear el TransactionManager.java dentro del paquete config y del punto 4, implementando la capa DAO completa con JDBC, incluyendo las operaciones CRUD para ambas entidades. Finalmente, Agustin Monardes implementó los puntos 5 y 6, desarrollando la capa de servicios con validaciones de negocio, manejo transaccional y la clase principal que permite la ejecución y prueba del sistema.

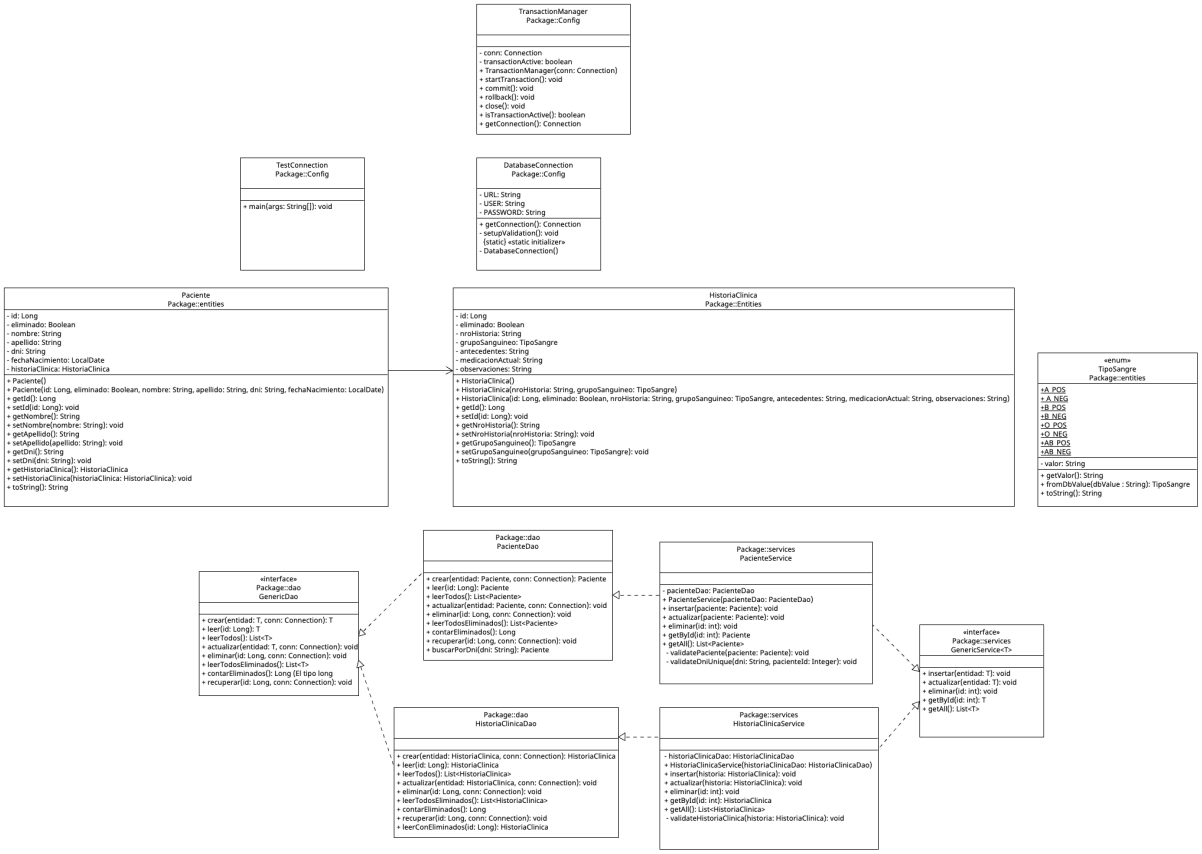
1. Diseño de estructura y UML

La primera fase del trabajo consistió en el desarrollo del diseño y la estructura base. Para representar la relación Paciente ↔ Historia Clínica se evaluaron dos estrategias de implementación:

1. Utilizar la misma clave primaria para ambas entidades (PK compartida)
2. Utilizar una clave primaria independiente y una FK única con restricción 1→1.

Se eligió la segunda opción (FK única) porque ofrece mayor flexibilidad en la persistencia, facilita la inserción y actualización independiente de las entidades, y también porque permite que la historia clínica sea opcional en el momento de creación del paciente (estado “sin historia médica registrada”). La unicidad de la relación se garantiza mediante una FK con restricción UNIQUE, lo que evita la duplicación o inconsistencias lógicas.

El diseño UML adjunto refleja esta decisión estructural y actúa como guía directa para las capas DAO y Service. El diseño UML que se presenta a continuación incluye los atributos, métodos públicos y dependencias que servirán como guía explícita para toda la implementación. Desde esta base, se definieron las decisiones arquitectónicas de conexión.



2) Entidades

Las entidades Paciente e HistoriaClinica se modelaron siguiendo el esquema de base de datos y el diagrama UML, y representan una relación unidireccional 1 a 1 donde un Paciente puede tener una única HistoriaClinica asociada. Con el objetivo de centralizar los atributos comunes, se creó la clase abstracta Base, que contiene los campos id y eliminado; aunque en el UML estos aparecían dentro de cada entidad, su ubicación en una superclase mantiene el mismo modelo conceptual y ofrece una mejora en la implementación al evitar código duplicado.

En cuanto a la particularidad de cada una de las entidades, la clase Paciente cuenta con atributos propios como nombre, apellido, dni y fechaNacimiento, junto con la referencia al atributo privado historiaClinica, materializando de esta manera la asociación unidireccional 1 a 1 con la Entidad B. HistoriaClinica, por su parte, contiene los atributos nroHistoria, grupoSanguineo, antecedentes, medicacionActual y observaciones. Ambas clases implementan constructores completos y vacíos, así como getters, setters y un método toString() legible, lo que facilita su uso en el patrón DAO y la reconstrucción de objetos desde la base de datos mediante JDBC. Adicionalmente, se incorporó el enum TipoSangre para representar los grupos sanguíneos válidos de forma tipada y mantener una correspondencia exacta con los valores definidos en la base de datos.

3) Base de datos (MySQL)

Luego de crear las entidades genéricas del proyecto, se definieron las clases necesarias para realizar la conexión a la base de datos desde el paquete config. Antes de ejecutar el script, se levantó el proyecto SQL desde cero mediante la ejecución de los archivos de creación de tablas (ver 01_esquema.sql), y datos de prueba (ver 02_catalogos.sql y 03_carga_masiva.sql). En la definición del esquema de la base de datos, se incorporó la propiedad AUTO_INCREMENT en el campo id de las tablas principales para que MySQL pueda asignar automáticamente un identificador único a cada registro insertado, simplificando las operaciones de alta desde el código Java y garantizando la integridad referencial entre las entidades.

A su vez, se optó por la creación de un usuario para desarrolladores (dev) con su contraseña (Grupo54Dev) para llevar a cabo las conexiones de manera segura y controlada. El siguiente SQL configura dichas credenciales y permisos sobre la base tfi_bd1:

```
CREATE USER IF NOT EXISTS 'dev'@'localhost' IDENTIFIED BY 'Grupo54Dev';
GRANT ALL PRIVILEGES ON tfi_bd1.* TO 'dev'@'localhost';
FLUSH PRIVILEGES;
```

Con la base creada y cargada, y el usuario de desarrollo configurado, se validó la conectividad utilizando la clase DatabaseConnection, que contiene un constructor privado para impedir que pueda ser instanciada, y cuyo objetivo es proveer la conexión a la base de datos mediante su método estático getConnection(). En su implementación, se definen las constantes URL, USER y PASSWORD, que almacenan la información de conexión. Estas variables pueden sobrescribirse dinámicamente utilizando system properties (-Ddb.url, -Ddb.user, -Ddb.password), lo que permite modificar la configuración sin recompilar el código.

Esta conexión puede verificarse mediante la clase de prueba TestConnection, que abre la conexión y, de ser exitosa, muestra metadatos (usuario, base de datos, URL y driver) a través de la consola:

```
Conexion exitosa a la base de datos
Usuario conectado: dev@localhost
Base de datos: tfi_bd1
URL: jdbc:mysql://localhost:3306/tfi_bd1
Driver: MySQL Connector/J vmysql-connector-j-8.4.0 (Revision: 1c3f5c149e0bfe31c7fbeb24e2d260cd890972c4)
BUILD SUCCESSFUL (total time: 0 seconds)
```

4) Capa de Acceso a Datos (DAO)

La Capa de Acceso a Datos implementa el patrón Data Access Object (DAO) con el objetivo de desacoplar la lógica de negocio de los detalles de persistencia. Esta arquitectura favorece la mantenibilidad, la reutilización y el control preciso de las transacciones, requisito fundamental para garantizar las propiedades ACID en operaciones compuestas.

4.1) TransactionManager.java (Gestor del Ámbito Transaccional)

Ubicada en el paquete config, esta clase implementa un gestor transaccional basado en una única Connection, suministrada por el pool de conexiones. Su diseño como objeto instanciable y su implementación de AutoCloseable permiten que el Service lo utilice en un bloque try-with-resources, asegurando que la conexión se cierre correctamente incluso si ocurre una excepción.

Responsabilidades principales:

- startTransaction(): desactiva el auto-commit e inicia explícitamente la transacción.
- commit()/rollback(): confirma o revierte los cambios y delega el cierre seguro de la conexión.
- close(): restablece el auto-commit, garantiza que no queden transacciones sin finalizar y devuelve la conexión al pool.

Toda la capa Service obtiene la conexión activa a través de getConnection(), lo que obliga a que todos los DAOs utilicen la misma Connection durante una transacción compuesta.

4.2) GenericDao.java (Contrato General de Persistencia) En el paquete dao

La interfaz GenericDao<T extends Base> define un contrato uniforme para la persistencia de entidades, permitiendo que las implementaciones concreten el acceso a la base sin duplicar lógica.

Aspectos destacables:

- Los métodos que modifican el estado (crear, actualizar, eliminar, recuperar) reciben una Connection obligatoria, permitiendo delegar el control transaccional a la capa Service.
- Los métodos de consulta (leer, leerTodos, contarEliminados) administran su propia conexión del pool, optimizando el rendimiento en operaciones que no requieren transacción.
- Se integra soporte para borrado lógico, incluyendo lectura de eliminados y recuperación.

Este diseño asegura independencia entre lógica de negocio, transacciones y acceso a datos.

4.3) Implementaciones Concretas

PacienteDao.java y HistoriaClinicaDao.java, ambos en el paquete dao, implementan el acceso a la base mediante consultas SQL parametrizadas (PreparedStatement) y el mapeo de resultados hacia las entidades Java.

Características comunes:

- Inyección de Connection en los métodos de modificación para garantizar atomicidad.
- Borrado lógico mediante UPDATE ... SET eliminado = TRUE.
- Obtención de ID autogenerado con RETURN_GENERATED_KEYS, asignando inmediatamente el ID al objeto entidad.
- Mapeo completo de entidades, incluyendo el tipo enumerado TipoSangre y valores LocalDate.

Particularidades:

- PacienteDao: incluye LEFT JOIN hacia HistoriaClinica para reconstruir la relación 1→1 si la historia existe, respetando que un Paciente puede no tener Historia Clínica.
- HistoriaClinicaDao: maneja exclusivamente la persistencia de la historia clínica, incluyendo la conversión del tipo enumerado hacia su almacenamiento en la base.

5) Servicios (Service)

La capa de Services implementa la lógica de negocio, aplica validaciones y coordina las transacciones entre los DAOs. Está diseñada para que cada operación de negocio sea atómica cuando corresponde, delegando la persistencia a las implementaciones DAO y utilizando TransactionManager para asegurar commit/rollback según el resultado de la operación.

5.1) Estructura general

- Paquete: services.
- Interfaz genérica: GenericService<T> define las operaciones básicas esperadas (insertar, actualizar, eliminar, getById, getAll, getAllDeleted, countDeleted, recuperar).
- Implementaciones concretas: PacienteService y HistoriaClinicaService, cada una encargada de las reglas de negocio de su dominio.

5.2) PacienteService

Responsabilidad principal: orquestar operaciones sobre Paciente y, cuando corresponda, coordinar la creación/actualización de la HistoriaClinica asociada manteniendo atomicidad.

Métodos y comportamiento clave:

- insertar(Paciente paciente): valida campos obligatorios, verifica unicidad de DNI y crea el paciente dentro de una transacción (TransactionManager).
- actualizar(Paciente paciente): valida datos, verifica existencia y unicidad de DNI (excluyendo al propio registro) y actualiza dentro de transacción.
- eliminar(Long id): realiza baja lógica (eliminado = TRUE) en transacción.

- recuperar(Long id): restablece un paciente eliminado lógicamente en transacción.
- crearPacienteConHistoriaOpcional(Paciente paciente, HistoriaClinica hc): flujo atómico que crea el Paciente y, si se proporciona, crea la HistoriaClinica vinculada usando la misma Connection. Si cualquier paso falla se ejecuta rollback() y no se persiste nada.

Validaciones implementadas (ejemplos):

- Nombre, apellido y DNI obligatorios.
- Fecha de nacimiento válida (no futura, no anterior a 1900).
- Unicidad del DNI.

5.3) HistoriaClinicaService

Responsabilidad principal: gestionar la persistencia y validaciones de HistoriaClinica, tanto en transacciones propias como en transacciones coordinadas desde PacienteService.

Métodos y comportamiento clave:

- crearHistoriaClinicaConPaciente(HistoriaClinica historia, long pacienteId): crea una historia para un paciente existente en transacción propia.
- crearHistoriaClinica(TransactionManager tm, HistoriaClinica historia, long pacienteId): versión diseñada para usarse dentro de una transacción ya abierta por el Service (por ejemplo en crearPacienteConHistoriaOpcional).
- actualizar, eliminar, recuperar, getById, getAll etc., con validaciones previas (nroHistoria obligatorio, grupo sanguíneo obligatorio).

Manejo de errores esperados:

- Si se intenta crear una segunda historia para el mismo pacienteId la base de datos lanza una violación de UNIQUE sobre paciente_id. El servicio captura esta excepción y la puede traducir a un mensaje de negocio amigable (p. ej. “El paciente ya tiene una Historia Clínica asignada”), manteniendo el rollback.

5.4) Transaccionalidad en los Services

- Todos los métodos que modifican datos relevantes se ejecutan dentro de un try-with-resources (TransactionManager tm = new TransactionManager(...)) y siguen el patrón: tm.startTransaction(); try { ... tm.commit(); } catch { tm.rollback(); throw; }.
- TransactionManager garantiza que autoCommit se maneje correctamente y que la conexión se cierre/restaure al terminar, evitando fugas de recursos.
- Este diseño asegura las propiedades ACID para operaciones compuestas (ej.: crear paciente + historia).

6) Main y AppMenu

6.1) Main

Clase: main.Main

Responsabilidad: punto de entrada de la aplicación. Se limita a inicializar y arrancar la interfaz de consola (AppMenu). En caso de error crítico en el arranque imprime la traza y espera interacción del usuario antes de finalizar para facilitar la inspección.

Flujo:

1. public static void main(String[] args) imprime mensaje de inicio.
2. Instancia AppMenu y llama appMenu.iniciar().
3. Manejo de excepciones global para errores de inicialización.

6.2) AppMenu (Interfaz de consola)

Clase: main.AppMenu

Responsabilidad: gestión interactiva por consola de las operaciones del sistema. Actúa como capa UI ligera que invoca a los Services.

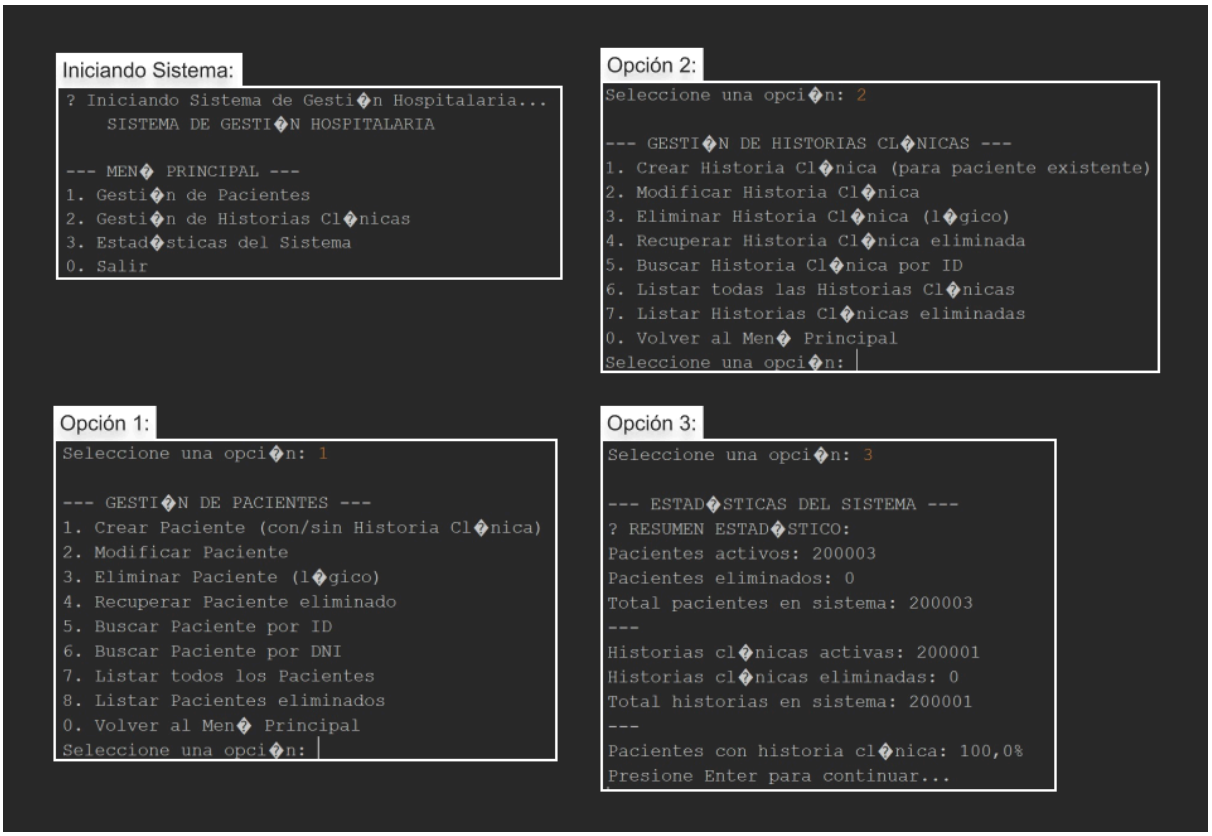
Características principales:

- Menú principal con opciones:

    1. Gestión de Pacientes
    2. Gestión de Historias Clínicas
    3. Estadísticas del Sistema
    4. Salir

- Submenús para Pacientes y Historias: crear (con/sin historia), modificar, eliminar (lógico), recuperar, buscar por ID, buscar por DNI, listar activos y listar eliminados.
- Captura de datos por consola (Scanner) con validaciones básicas (formato fecha, entradas vacías, confirmaciones).
- En la creación de pacientes pregunta si se desea crear además una Historia Clínica; si la respuesta es afirmativa, captura los campos de la HC y llama a pacienteService.crearPacienteConHistoriaOpcional(...) (operación transaccional).
- Otras operaciones llaman a los Services correspondientes (PacienteService, HistoriaClinicaService) y muestran mensajes claros de éxito/error.

A continuación, se muestra la ejecución de Main y las respuestas correspondientes a cada una de las opciones que componen el menú:



En la siguiente captura, se muestra un ejemplo de creación de un nuevo paciente de forma correcta, sin historia clínica, ejecutando el commit para guardarlo en la base de datos:

```
--- CREAR NUEVO PACIENTE ---
Nombre: Pepe
Apellido: Juare
DNI: 34256987
Fecha de nacimiento (YYYY-MM-DD) [opcional]: 1997-05-15
�Desea crear una Historia Cl�nica para este paciente? (S/N): n
[DEBUG] Transacci�n iniciada.
[DEBUG] Commit realizado correctamente.
[DEBUG] Conexi�n cerrada y liberada.
? Transacci�n completada - Paciente creado
? Paciente creado exitosamente sin Historia Cl�nica
Presione Enter para continuar...
```

Por último, se evidencia la respuesta del programa al intentar agregar a un paciente de forma incorrecta, sin historia clínica, ejecutando el rollback para no guardarlo en la base de datos:

```
--- CREAR NUEVO PACIENTE ---
Nombre: Raul
Apellido:
DNI: 33264787
Fecha de nacimiento (YYYY-MM-DD) [opcional]:
�Desea crear una Historia Cl�nica para este paciente? (S/N): n
[DEBUG] Transacci�n iniciada.
[DEBUG] Rollback realizado correctamente.
[DEBUG] Conexi�n cerrada y liberada.
? Transacci�n revertida - Rollback realizado
? Error: El apellido del paciente es obligatorio
Presione Enter para continuar...
```

La confirmación de que el rollback funcionó se detalla a continuación:

```
--- BUSCAR PACIENTE POR DNI ---
DNI del paciente: 33264787
? No se encontr� paciente con DNI: 33264787
Presione Enter para continuar...
```

7) Conclusiones

Se logró implementar un sistema CRUD completo para el dominio Paciente → Historia Clínica, aplicando arquitectura en capas, DAO, Services y transacciones. El sistema logra una gestión eficiente de pacientes mediante un registro completo con datos esenciales (DNI único, datos personales), flexibilidad para crear pacientes con o sin historia clínica inicial, una búsqueda rápida por DNI para acceso inmediato en emergencias y la eliminación lógica que preserva datos históricos sin perder información.

La relación 1→1 con FK única garantiza consistencia y flexibilidad en la persistencia de datos, esto asegura que cada paciente tenga un máximo de una historia clínica, que la información médica esté asociada al paciente correcto y que no haya historias clínicas 'huérfanas' sin paciente asociado.

Las validaciones de negocio y el manejo transaccional aseguran que los datos críticos (como pacientes e historias) se mantengan correctos y coherentes: si falla la creación de historia clínica, el paciente tampoco se registra. Se evitan los datos médicos inválidos y el rollback automático previene estados inconsistentes.

La separación de responsabilidades (DAO / Service / Main) facilita la mantenibilidad y escalabilidad del sistema, las entidades representan los datos con un mínimo de lógica, los DAOs no validan las reglas de los datos en SQL. Las dependencias fluyen en una sola dirección: AppMenu no conoce SQL, Services no conoce la UI, los DAOs no conocen las reglas de negocio. Los cambios se aíslan en capas específicas, lo que hace que la escalabilidad futura sea más fácil de implementar.

En este campo, como posibles implementaciones futuras podría lograrse:

- Una interfaz gráfica (GUI) para mejorar la experiencia de usuario.
- Incorporar búsquedas avanzadas y filtros en la UI.
- Integrar pruebas automatizadas (JUnit) para cubrir la lógica de negocio y DAOs.
- Extender el modelo para manejar múltiples historias por paciente si cambia el requerimiento.
- Encriptamiento de datos sensibles y cumplimiento de normas de privacidad.

Herramientas utilizadas

Para el desarrollo del Trabajo Práctico Integrador se utilizaron diversas herramientas que permitieron implementar la arquitectura en capas, el diseño UML y la documentación completa del proyecto. A continuación se describen las principales:

- NetBeans IDE 17

  Uso: Desarrollo del código fuente, organización del proyecto, ejecución y depuración. Enlace: https://netbeans.apache.org/

- UMLETino (UMLet Web Editor)

  Uso: Creación del diagrama UML de clases utilizado como guía estructural para toda la implementación del sistema (entidades, interfaces, DAO, servicios y relaciones). Enlace: https://www.umlet.com/umletino/

- MySQL 8.x

  Uso: Creación de la base de datos, tablas y restricciones; ejecución de queries SQL de prueba; validación de claves foráneas y unicidad. Enlace: https://www.mysql.com/

- MySQL Workbench / Cliente SQL

  Uso: Verificación de estructura, ejecución de consultas, pruebas de integridad referencial. Enlace: https://www.mysql.com/products/workbench/

- Git & GitHub

  Uso: Control de versiones del proyecto, coordinación en equipo, pushes y commits individuales, y publicación del repositorio final. Enlace: https://github.com/

- ChatGPT (OpenAI)

  Uso: Asistencia para la construcción de textos, revisión de diseño arquitectónico, explicación de conceptos técnicos y mejora de documentación. Enlace: https://chat.openai.com