UNIVERSIDAD TECNOLÓGICA NACIONAL

TRABAJO PRÁCTICO INTEGRADOR PROGRAMACIÓN II

Tema: desarrollo de 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.

Alumnos:

- Gastón Armando Giorgio - gastongiorgio26@gmail.com

- Matias Ariel Deluca - matiasdeluca2000@gmail.com

- Luciano Demian Contreras - lucianocontrerasestudio04@gmail.com

- Aldo Dario Manfredi - manfredialdo.1979@gmail.com

Materia: Programación II

Profesor: Ariel Enferrel

Fecha de entrega: 17/11/2025

ÍNDICE

I. INTRODUCCIÓN.

El presente informe documenta el desarrollo del **Trabajo Práctico Integrador** de la materia Programación II, enfocado en la implementación de un sistema de gestión para la entidad **Paciente** y su relación uno a uno (1:1) con **Historia Clínica**.

El objetivo central del proyecto fue aplicar y consolidar los principios de la Programación Orientada a Objetos (POO), la **arquitectura en capas** y la **persistencia de datos transaccional** utilizando exclusivamente Java JDBC y MySQL.

A lo largo del desarrollo, hemos priorizado la **integridad de los datos**, implementando la baja lógica (Soft Delete) y asegurando que las operaciones compuestas (creación, actualización y eliminación) se ejecuten bajo un estricto control de commit y rollback.

II. ELECCIÓN DEL DOMINIO Y JUSTIFICACIÓN:

Hemos seleccionado la pareja de entidades **Paciente (A)** y **Historia Clínica (B)** para modelar la relación uno a uno (1:1). Esta elección se justifica por la naturaleza crítica de la **integridad referencial** que exige el dominio: **un paciente debe tener exactamente una historia clínica, y una historia clínica no puede existir sin un paciente asociado.**

Este dominio nos permitió demostrar el manejo de restricciones de unicidad y la orquestación de operaciones transaccionales compuestas.

III. DISEÑO Y ARQUITECTURA

- **Diseño:**

La decisión fundamental de diseño se centró en garantizar la unicidad de la relación 1:1 directamente en la capa de persistencia. Elegimos la opción de **clave foránea única** en la tabla dependiente:

- La tabla **historia_clinica (entidad B)** contiene la columna **paciente_id**.
- Esta columna fue definida como **NOT NULL** (obligatoriedad) y con la restricción **UNIQUE** (unicidad).

Esta aproximación nos asegura que, al intentar asociar una segunda historia clínica al mismo paciente_id, la base de datos (MySQL) detendrá la operación antes de que el service layer la procese, garantizando la integridad de la regla 1:1.

El diagrama UML de clases adjunto detalla la estructura final de nuestro sistema, mostrando la herencia de la clase abstracta Base y la **asociación unidireccional** que fluye desde Paciente hacia HistoriaClinica (Ver ANEXO: diagrama de clases UML).

- **Arquitectura por capas:**

Adoptamos un patrón de **arquitectura en capas** estricto, donde cada paquete tiene una **responsabilidad única** (principio de responsabilidad única):

- **models (modelo):** contiene la representación de las entidades (Paciente, HistoriaClinica) y la clase base abstracta (Base). Su única función es almacenar el estado de los datos.
- **config (infraestructura):** gestiona recursos. Incluye DatabaseConnection (Factory para la conexión JDBC) y TransactionManager (utilidad para el manejo de AutoCloseable).
- **dao/impl (acceso a datos):** contiene las implementaciones específicas de SQL (PreparedStatement) para el CRUD, el mapeo de ResultSet a entidades, y la optimización LEFT JOIN. Los DAOs solo se comunican con la base de datos y no contienen lógica de negocio.
- **service (lógica de negocio):** es el corazón del sistema. Implementa validaciones, reglas de negocio (validar(p)), y orquesta las transacciones complejas.
- **main (presentación):** contiene AppMenu y MenuHandler, y es el único punto de interacción con el usuario (I/O). Delega toda la lógica de negocio a la capa service.

IV. PERSISTENCIA Y TRANSACCIONES

- **Persistencia: estructura de la base, orden de operaciones y transacciones:**

La estructura de la base de datos se definió en el **script 01_create.sql**. Utilizamos BIGINT para las claves primarias y BOOLEAN para el campo eliminado (baja lógica).

El manejo transaccional es la parte más crítica de nuestro diseño y se implementa exclusivamente en la capa de servicio (**PacienteServiceImpl**):

- **Inicio:**

Se obtiene la **Connection** y se llama a **con.setAutoCommit(false)** para iniciar la unidad de trabajo.

- **Coordinación (service-llama-service):**

Los métodos compuestos (como insertar y eliminar) manejan la misma conexión en ambas entidades. **PacienteServiceImpl** llama a los métodos transaccionales del **HistoriaClinicaService** y de su propio **PacienteDao**, asegurando que todas las operaciones compartan el mismo estado transaccional.

- **Fin (commit/rollback):**

  - **COMMIT:** Si todas las operaciones se ejecutan sin excepciones (ej. pacienteDao.create y hcService.insertar finalizan), se llama a **con.commit()** para hacer permanentes los cambios.

  - **ROLLBACK:** Si se captura cualquier **Exception** durante el proceso transaccional, se llama a **con.rollback()** dentro del bloque catch para revertir todos los cambios y dejar la base de datos en un estado consistente.

- **<u>Orden de operaciones críticas</u>:**

1) Inserción (insertar): se inserta primero el Paciente (A) para obtener su ID autogenerado, y luego se inserta la Historia Clínica (B) utilizando ese paciente_id como clave foránea única.

2) Eliminación (eliminar): se ejecuta la baja lógica primero sobre la Historia Clínica (B) y luego sobre el Paciente (A) dentro de la misma transacción. Esto previene que, si la baja del Paciente falla, la Historia Clínica quede marcada como eliminada sin un paciente. La baja es siempre UPDATE tabla SET eliminado = TRUE.

V. REGLAS DE NEGOCIO:

- **<u>Validaciones y reglas de negocio</u>:**

Implementamos validaciones esenciales en la capa service:

- <u>Regla 1</u>: campos obligatorios:

Verificación estricta de que los **atributos críticos** de la entidad (nombre, apellido, dni) y de la entidad dependiente (nroHistoria) **no sean nulos ni contengan solo espacios en blanco**.

La validación se ejecuta en el método privado validar(T t) de las implementaciones del service, utilizando el chequeo null o .isBlank().

- Regla 2: integridad 1:1 forzada:

Asegura que la regla de negocio de la relación uno a uno se cumpla: **un paciente debe crearse siempre con una historia clínica asociada**.

Se valida explícitamente en el PacienteServiceImpl.validar(p) que el atributo anidado (HistoriaClinica) no sea nulo antes de permitir el inicio de la transacción de inserción.

- Regla 3: conversión y validación de formatos:

Gestiona la conversión segura de tipos de datos. Esto incluye la **validación de fechas** (ej. LocalDate) y la **conversión segura de valores predefinidos** desde la consola a tipos Enum.

Se utiliza la función LocalDate.parse() con manejo de excepciones y el método estático Enum.fromDb() en la entidad HistoriaClinica para la conversión segura del GrupoSanguineo.

- Regla 4: unicidad multi-nivel:

El **DNI debe ser único** y esta validación debe ser verificada en dos niveles para prevenir condiciones de carrera: 1) En la **capa Service** antes de la persistencia, y 2) Mediante un **UNIQUE constraint** en la tabla paciente de la base de datos.

La validación se ejecuta en PacienteServiceImpl.validar() y la BD provee la validación final.

- Regla 5: baja lógica de entidad compuesta:

La eliminación debe ser lógica y atómica para la relación 1:1. No se permite la eliminación física (DELETE). Al dar de baja al Paciente, su Historia Clínica asociada debe ser marcada como inactiva en la misma transacción.

Se implementa en PacienteServiceImpl.eliminar(id) mediante un UPDATE transaccional que asegura que tanto paciente.eliminado como historia_clinica.eliminado pasen a ser TRUE.

- <u>Regla 6</u>: preservación de datos en actualización:

En las operaciones de actualización, **los campos de entrada que el usuario deja vacíos (presionando Enter) deben preservar su valor original en la base de datos.** Solo se deben modificar los datos que han sido explícitamente reingresados.

Se aplica en MenuHandler con la lógica if (!input.isBlank()) { entity.setField(input); }, asegurando que el objeto solo modifique los atributos que el usuario pretendía cambiar.

VI. CALIDAD DE CÓDIGO Y OPTIMIZACIÓN:

Cumplimos con el uso de nombres significativos, formato consistente y control de excepciones. Además, implementamos dos prácticas de alto valor:

1) Optimización N+1 (LEFT JOIN):

Para las operaciones de lectura (getById, getAll), se modificó el PacienteDaoImpl para utilizar un **LEFT JOIN**. Esto reduce las consultas de N + 1 a **1 sola consulta** a la base de datos para cargar el Paciente junto con su Historia Clínica, garantizando una alta eficiencia.

2) Baja Lógica (Soft Delete):

En lugar de borrar físicamente los registros, todas las operaciones de delete utilizan **UPDATE tabla SET eliminado = TRUE** y todas las consultas de lectura filtran **WHERE eliminado = FALSE**.

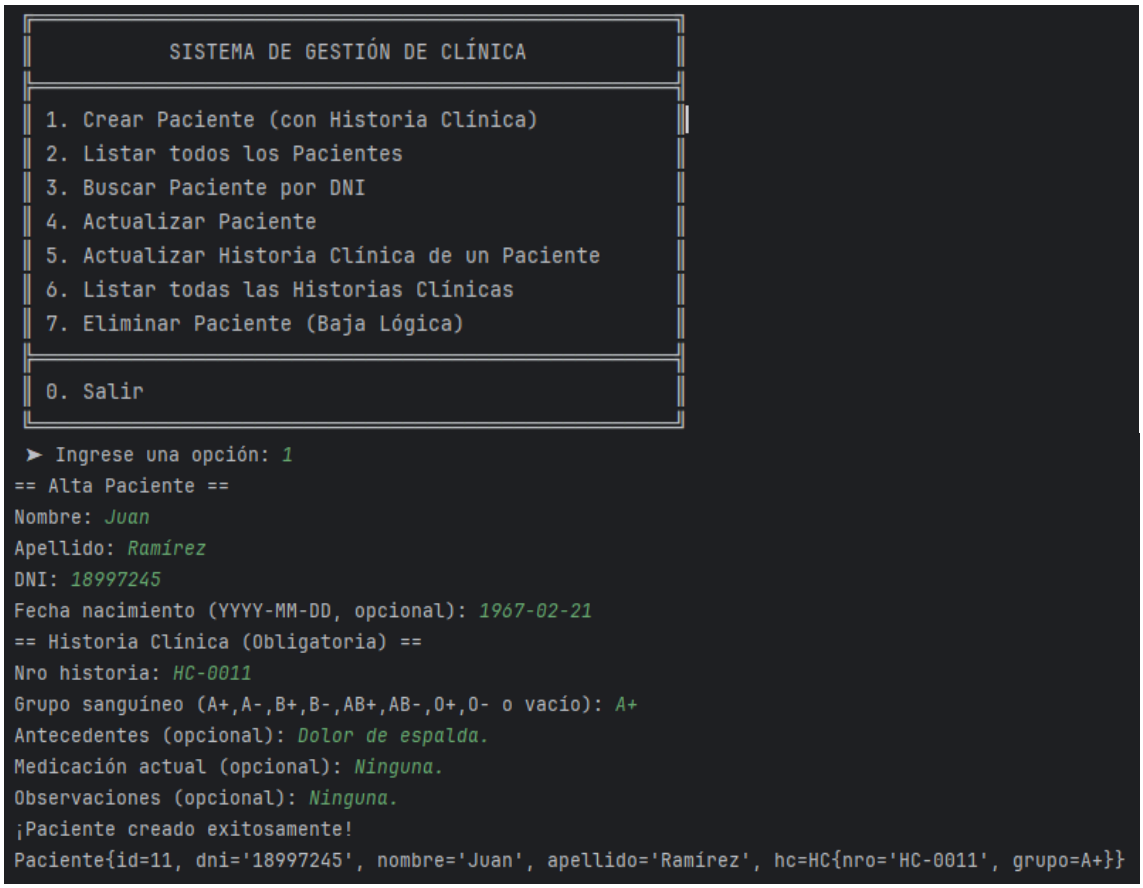
VII. PRUEBAS REALIZADAS:

Realizamos pruebas exhaustivas en el entorno de desarrollo (IntelliJ + Workbench) para verificar la atomicidad, validación de unicidad y eliminacion lógica.

A. PRUEBAS DE INSERCIÓN Y UNICIDAD (OPCIONES 1, 3, 5 Y 6)

1) Prueba de inserción atómica (A + B):

El usuario crea un paciente nuevo con su historia clínica.



- Verificación en workbench:

```
SELECT * FROM tpi_prog2.paciente;
```

id	eliminado	nombre	apellido	dni	fecha_nacimiento
1	0	Carlos	Perez	30123456	1985-05-15
2	0	Rodrigo	Mendez	31987654	1986-11-20
3	0	Ana	Gomez	32543210	1990-03-25
4	0	Martin	Lopez	33876543	1982-12-10
5	0	Sofia	Ruiz	34112233	1995-07-01
6	0	Javier	Diaz	35776655	1975-01-20
7	0	Lucia	Torres	36334455	2001-09-08
8	0	Diego	Castro	37009988	1988-04-12
9	0	Elena	Vega	38555444	1999-11-30
10	0	Hugo	Flores	39666777	1970-06-18
11	0	Juan	Ramírez	18997245	1967-02-21

```
SELECT * FROM tpi_prog2.historia_clinica;
```

	id	elimi	nro_historia	grupc	antecedentes	medicacion_actua	observaciones	paci	fecha_apertura
▶	1	0	HC-0001	A+	Alergia al polen.	Loratadina 10mg.	Seguimiento ...	1	NULL
	2	0	HC-0002	O-	Cirugía de apéndic...	Ninguna.	Sin seguimien...	2	NULL
	3	0	HC-0003	B+	Asma infantil.	Salbutamol.	Control pulm...	3	NULL
	4	0	HC-0004	AB-	Fractura de tibia (...	Ninguna.	Rehabilitació...	4	NULL
	5	0	HC-0005	O+	Alergia a la penicili...	Ninguna.	Anotar la aler...	5	NULL
	6	0	HC-0006	A-	Hipertensión leve.	Enalapril 5mg.	Monitoreo de...	6	NULL
	7	0	HC-0007	B-	Vacunación comple...	Ninguna.	Paciente pedi...	7	NULL
	8	0	HC-0008	AB+	Amigdalectomía.	Ninguna.	Se recomiend...	8	NULL
	9	0	HC-0009	A+	Intolerancia a la la...	Suplementos Vit.	Revisar nivel...	9	NULL
	10	0	HC-0010	O-	Apendicitis, obesid...	Metformina.	Derivación a ...	10	NULL
	11	0	HC-0011	A+	Dolor de espalda.	Ninguna.	Ninguna.	11	2025-11-15

Figura 1. Prueba de creación atómica. Estas capturas verifican el flujo exitoso de la Opción 1. Se demuestra la **Regla 2: Integridad 1:1** y el éxito de la transacción de inserción (commit), resultando en un nuevo paciente y su historia clínica asociados correctamente.

2) Prueba de fallo de unicidad (DNI duplicado):

El usuario inserta el DNI de un paciente activo (Ej: 30123456).

```
  ➤ Ingrese una opción: 1
== Alta Paciente ==
Nombre: José
Apellido: Giménez
DNI: 30123456
Fecha nacimiento (YYYY-MM-DD, opcional): 1980-09-04
== Historia Clínica (Obligatoria) ==
Nro historia: HC-0012
Grupo sanguíneo (A+,A-,B+,B-,AB+,AB-,O+,O- o vacío): A+
Antecedentes (opcional): Ninguno.
Medicación actual (opcional): Ninguna.
Observaciones (opcional): Ninguna.

Error al crear paciente: Error transaccional al insertar: Duplicate entry '30123456' for key 'dni'
```

Figura 2. Demostración de ROLLBACK y validación de unicidad. Se valida el manejo de excepciones de la capa service al intentar violar la **Regla 4: Unicidad Multi-Nivel (DNI)**. El sistema intercepta el error, muestra un mensaje limpio y ejecuta con.rollback(), asegurando que no se inserte ningún registro parcial.

3) Prueba de Fallo de validación (campo obligatorio):

El usuario deja el campo “Nombre” en blanco.

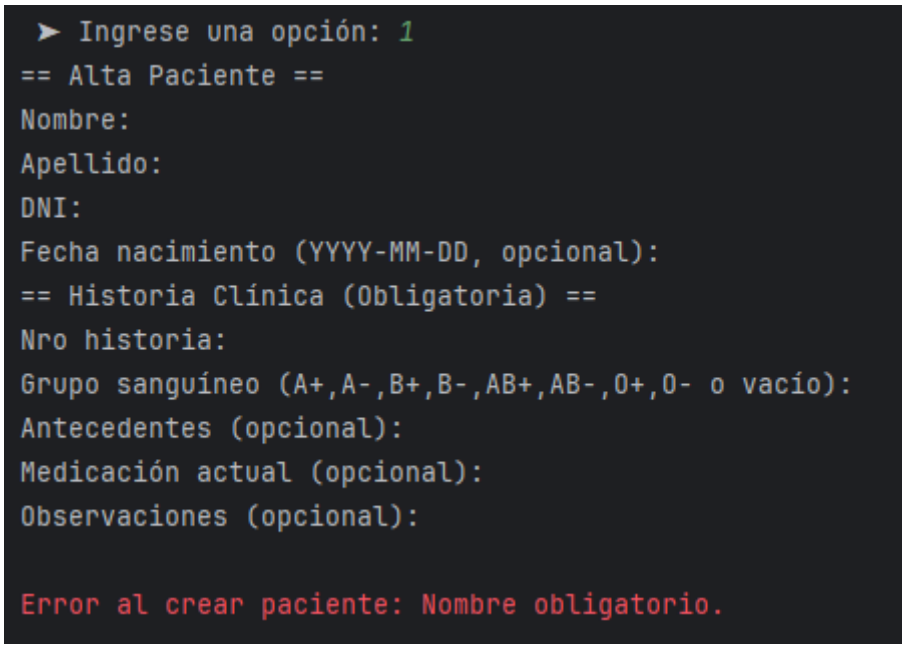


Figura 3. Fallo controlado por campo obligatorio. Se verifica el cumplimiento de la **Regla 1: campos obligatorios**. La capa service detecta el valor null o isBlank() en un campo crítico (ej., Nombre) y detiene la ejecución antes de llegar a la base de datos.

B. PRUEBAS DE ACTUALIZACIÓN

4) Actualización parcial de paciente:

El usuario intenta modificar solo el apellido del paciente de “Pérez” a “Rodriguez”:

- Comprobación con la opción 3 (búsqueda de paciente por DNI):

```
 ➤ Ingrese una opción: 3
Ingrese DNI a buscar: 30123456

═══════════ FICHA DEL PACIENTE ═══════════
 Nombre Completo     : Carlos Rodríguez
 DNI                 : 30123456
 Fecha Nacimiento    : 1985-05-15
──────────────────────────────────────────

 DATOS CLÍNICOS
 Nro. Historia       : HC-0001
 Grupo Sanguíneo     : A+
 Observaciones       : Seguimiento anual con especialista.
══════════════════════════════════════════
```

Figura 4. Preservación de datos en actualización. Se valida la **Regla 6: Preservación de datos**. Se demuestra que solo el campo modificado (Ej., Apellido) fue alterado, mientras que los campos omitidos (Enter) se mantuvieron inalterados, confirmando la lógica de if (!input.isBlank()).

5) Actualización atómica de historia clínica (fallo por ID Inexistente):

```
 ➤ Ingrese una opción: 5
Ingrese el ID del Paciente cuya Historia Clínica desea actualizar: 999
No se encontró un Paciente con ese ID.
```

Figura 5. Manejo robusto de ID inexistente. Se valida el control de flujo al intentar la Opción 5 con un ID no válido (Ej., 999). La aplicación captura el error en la capa service y notifica al usuario sin exponer la traza (SQLException).

6) Prueba de atomicidad de actualización (modificación de grupo sanguíneo):

El usuario desea actualizar el grupo sanguíneo del paciente en su historia clínica.

```
 ➤ Ingrese una opción: 5
Ingrese el ID del Paciente cuya Historia Clínica desea actualizar: 1
Editando Historia Clínica: HC{nro='HC-0001', grupo=A+}
Nuevo grupo (A+,A-,etc. o vacío para dejar): B+
Nuevas observaciones (o vacío para dejar):
¡Historia Clínica actualizada con éxito!
```

- Comprobación con la opción 6 (listar historias clínicas):

```
➤ Ingrese una opción: 6
```

```
ID: 1    | Nro HC: HC-0001    | Grupo: B+    | Fecha: -

• Antecedentes:
  Alergia al polen.

• Medicación Actual:
  Loratadina 10mg.

• Observaciones:
  Seguimiento anual con especialista.
```

Figura 6. Evidencia de transacción atómica de UPDATE. Esta prueba verifica el cumplimiento de la **Regla 14: La operación de actualización debe ser atómica**. Se demuestra que el cambio realizado al grupo sanguíneo del paciente mediante la Opción 5 activó una transacción que, al finalizar con éxito, actualizó los datos en la tabla historia_clinica y se ejecutó con.commit().

C. PRUEBAS DE ELIMINACIÓN (OPCIÓN 7):

7) Baja lógica atómica (app UI):

El usuario intenta eliminar un paciente por su ID.

```
➤ Ingrese una opción: 7
Ingrese el ID del Paciente a eliminar (Baja Lógica): 1
Paciente (ID: 1) eliminado con éxito (baja lógica).
```

Figura 7. Ejecución de la transacción de baja lógica. Se muestra la confirmación de la Opción 7, la cual activa la transacción compuesta. Este paso verifica la **Regla 5: baja lógica atómica de entidad compuesta** a nivel de la interfaz de usuario.

- Verificación de la baja lógica en Workbench:

```
SELECT * FROM tpi_prog2.paciente;
```

id	eliminado	nombre	apellido	dni	fecha_nacimiento
1	1	Carlos	Rodríguez	30123456	1985-05-15

```
SELECT * FROM tpi_prog2.historia_clinica;
```

id	eliminado	nro_historia	grupo_sanguineo	antecedentes	medicacion_actual	observaciones	paciente_id	fecha_apertura
1	1	HC-0001	B+	Alergia al polen.	Loratadina 10mg.	Seguimiento ...	1	NULL

Figura 8. Verificación de atomicidad de baja lógica en BD. Estas **capturas de Workbench** (después de ejecutar SELECT * FROM tpi_prog2.paciente; y SELECT * FROM tpi_prog2.historia_clinica;) es la evidencia final. Se verifica que la columna eliminado paso de tener un valor de 0 (FALSE) a **1 (TRUE)** en ambas tablas (paciente y historia_clinica), demostrando el cumplimiento de la **Regla 5**.

8) Prueba de ID inválido:

El usuario ingresa un ID negativo (`-5`) en la Opción 7 (Eliminar).

```
 ➤ Ingrese una opción: 7
Ingrese el ID del Paciente a eliminar (Baja Lógica): -5

Error al eliminar paciente: El ID de Paciente es inválido.
```

Figura 9: Verificación de ID Inválido. Esta prueba tiene como objetivo validar que el sistema intercepta y rechaza un identificador numérico que, aunque válido sintácticamente (es un número), es **inválido lógicamente** (es menor o igual a cero).

La lógica aquí verifica que la **capa de servicio** aplica la validación lógica antes de comunicarse con la base de datos.

1. **Activación de la regla:** El usuario ingresa un ID negativo (ej., -5) en la Opción 7 (Eliminar).
2. **Validación:** El service (PacienteServiceImpl) detecta que el ID viola la regla de ser positivo (if (id <= 0)), lo que causa un IllegalArgumentException.

3. **Resultado en consola:** El MenuHandler intercepta la excepción del service y muestra un mensaje de error limpio: **"El ID de paciente es inválido"**.

Esto demuestra que no se necesita enviar una consulta innecesaria al servidor de base de datos para saber que un ID negativo es imposible.

VIII. CONCLUSIONES:

La realización de este Trabajo Práctico Integrador nos permitió consolidar la comprensión de los pilares fundamentales de Programación II. El proyecto demostró la habilidad del equipo para diseñar e implementar una **arquitectura de software limpia y robusta**, adhiriéndose estrictamente al patrón de Capas (Service, DAO, Model).

Logramos resolver desafíos técnicos complejos, siendo los más significativos la **gestión transaccional** de las operaciones que involucran la relación 1:1 entre Paciente e Historia Clínica, y la **optimización del rendimiento** de las consultas de lectura.

La implementación exitosa de las transacciones atómicas (commit/rollback) en operaciones compuestas (crear, actualizar y eliminar) y la aplicación del LEFT JOIN para mitigar el problema N+1 confirman nuestro dominio de la persistencia de datos y las buenas prácticas de eficiencia en el desarrollo backend.

XI. MEJORAS FUTURAS:

Algunos puntos de mejora podrían ser:

1- La implementación de un Pool de Conexiones (Connection Pooling).

Actualmente, nuestra capa DAO abre y cierra una nueva conexión JDBC (java.sql.Connection) por cada operación de persistencia (CRUD). Este proceso de establecer, autenticar y liberar recursos por cada solicitud genera una **alta latencia** y consume recursos significativos del servidor de bases de datos.

La implementación de un Pool de Conexiones (utilizando librerías estándar como **HikariCP** o **c3p0**) permitiría:

- **Reutilización:** mantener un conjunto limitado de conexiones activas y listas para su uso.
- **Reducción de latencia:** eliminar el costoso tiempo de inicialización de la conexión.

- **Gestión de recursos:** limitar el número máximo de conexiones concurrentes a la base de datos, previniendo la sobrecarga del servidor MySQL.

Esto optimizaría drásticamente el rendimiento y la escalabilidad del sistema, especialmente bajo cargas de trabajo concurrentes.

2- La implementación de una Interfaz Gráfica de Usuario (GUI).

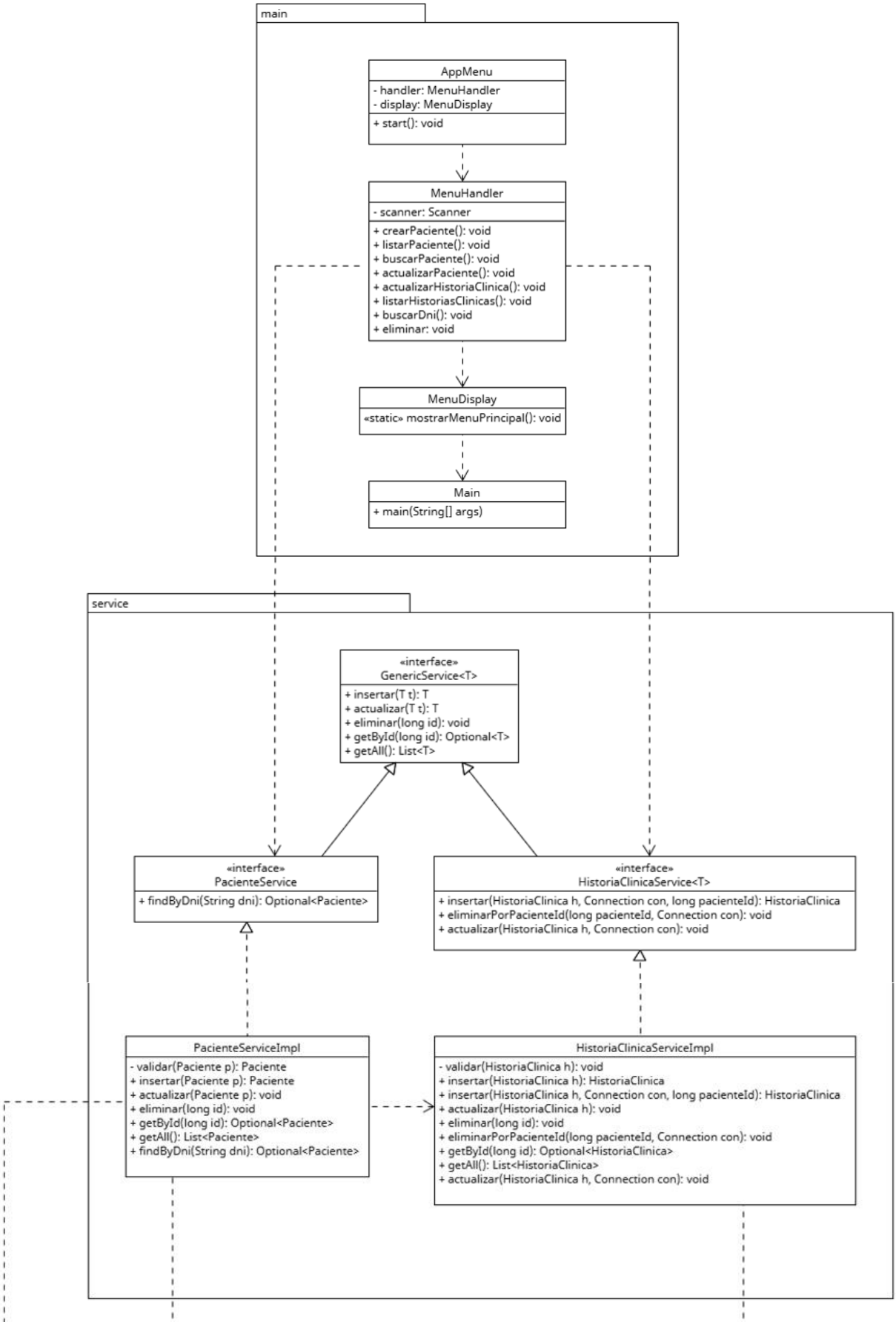
Actualmente, nuestra aplicación utiliza una Interfaz de Línea de Comandos (CLI), que, aunque funcional y robusta, es restrictiva. Migrar a una GUI (utilizando frameworks como **Swing** o, preferiblemente, **JavaFX**) permitiría:

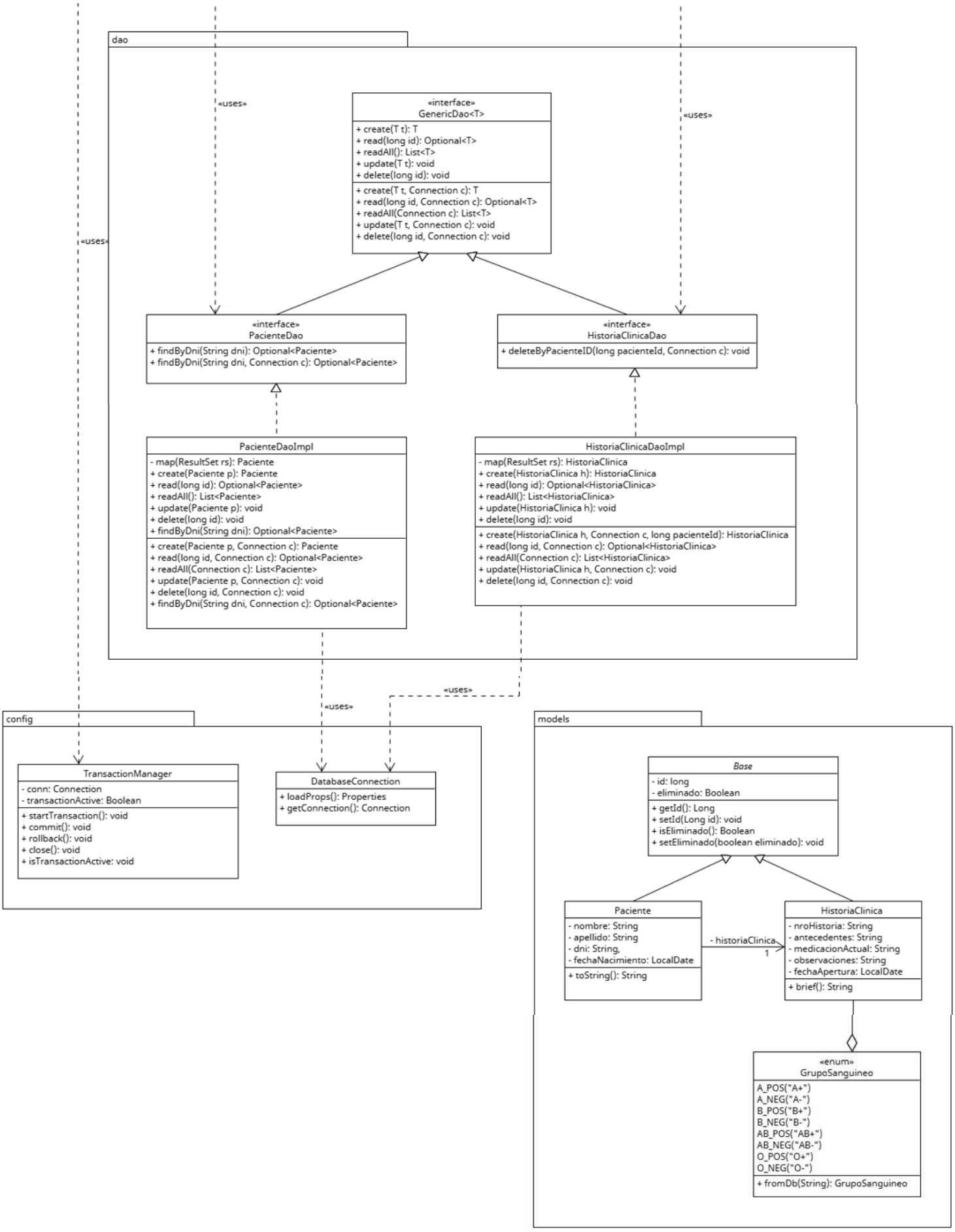
- **Mejorar la experiencia de usuario (UX):** ofrecería un entorno visual intuitivo con botones y formularios dinámicos para la gestión de pacientes.
- **Reducir errores de entrada:** se podrían utilizar listas desplegables y selectores de calendario para la captura de datos (como el GrupoSanguineo y las fechas), minimizando los errores de formato que actualmente debe gestionar la capa de presentación (MenuHandler).
- **Presentación profesional:** proveería una presentación más moderna y acorde a los estándares industriales para el manejo y visualización de los datos tabulares.

X. FUENTES Y HERRAMIENTAS UTILIZADAS:

- **Lenguajes y plataformas:** Java 17, Gradle 8.12, IntelliJ IDEA, MySQL 8.0, XAMPP y MySQL Workbench.
- **Librerías:** Connector/J (JDBC Driver).
- **Herramienta de soporte (IA):** se utilizó la herramienta de Inteligencia Artificial (Gemini Pro) como asistente de documentación, validación arquitectónica y debugging avanzado.

XI. ANEXO: diagrama de clases UML




dao
«uses»
«uses»
«uses»
«interface»
GenericDao<T>
+ create(T t): T
+ read(long id): Optional<T>
+ readAll(): List<T>
+ update(T t): void
+ delete(long id): void
+ create(T t, Connection c): T
+ read(long id, Connection c): Optional<T>
+ readAll(Connection c): List<T>
+ update(T t, Connection c): void
+ delete(long id, Connection c): void
«interface»
PacienteDao
+ findByDni(String dni): Optional<Paciente>
+ findByDni(String dni, Connection c): Optional<Paciente>
«interface»
HistoriaClinicaDao
+ deleteByPacienteID(long pacienteId, Connection c): void
PacienteDaoImpl
- map(ResultSet rs): Paciente
+ create(Paciente p): Paciente
+ read(long id): Optional<Paciente>
+ readAll(): List<Paciente>
+ update(Paciente p): void
+ delete(long id): void
+ findByDni(String dni): Optional<Paciente>
+ create(Paciente p, Connection c): Paciente
+ read(long id, Connection c): Optional<Paciente>
+ readAll(Connection c): List<Paciente>
+ update(Paciente p, Connection c): void
+ delete(long id, Connection c): void
+ findByDni(String dni, Connection c): Optional<Paciente>
HistoriaClinicaDaoImpl
- map(ResultSet rs): HistoriaClinica
+ create(HistoriaClinica h): HistoriaClinica
+ read(long id): Optional<HistoriaClinica>
+ readAll(): List<HistoriaClinica>
+ update(HistoriaClinica h): void
+ delete(long id): void
+ create(HistoriaClinica h, Connection c, long pacienteId): HistoriaClinica
+ read(long id, Connection c): Optional<HistoriaClinica>
+ readAll(Connection c): List<HistoriaClinica>
+ update(HistoriaClinica h, Connection c): void
+ delete(long id, Connection c): void
«uses»
«uses»
config
TransactionManager
- conn: Connection
- transactionActive: Boolean
+ startTransaction(): void
+ commit(): void
+ rollback(): void
+ close(): void
+ isTransactionActive: void
DatabaseConnection
+ loadProps(): Properties
+ getConnection(): Connection
models
Base
- id: long
- eliminado: Boolean
+ getId(): Long
+ setId(Long id): void
+ isEliminado(): Boolean
+ setEliminado(boolean eliminado): void
Paciente
- nombre: String
- apellido: String
- dni: String,
- fechaNacimiento: LocalDate
+ toString(): String
- historiaClinica
1
HistoriaClinica
- nroHistoria: String
- antecedentes: String
- medicacionActual: String
- observaciones: String
- fechaApertura: LocalDate
+ brief(): String
«enum»
GrupoSanguineo
A_POS("A+")
A_NEG("A-")
B_POS("B+")
B_NEG("B-")
AB_POS("AB+")
AB_NEG("AB-")
O_POS("O+")
O_NEG("O-")
+ fromDb(String): GrupoSanguineo