

# Informe Técnico: Gestor de Historial Clínico (TP Programación 2)

**Materia:** Programación 2

**Título:** Gestor de Historial Clínico - Java + MySQL

**\*\*Integrantes (4) y roles**

- **Leonel Jesús Aballay** – Rol: (Capa Services)
  - **Caín Cabrera Bertolazzi** – Rol: (Capa Menu, Config)
  - **Alex Nahuel Austin** – Rol: (Capa Model)
  - **Cristian Gabriel Aguirre** – Rol: (Capa DAO)
- 

## 1. Elección del dominio y justificación

**Dominio elegido:** sistema de gestión de historiales clínicos (pacientes y sus historias médicas).

**Justificación:**

- Es un dominio con necesidad clara de integridad referencial (1→1 paciente–historial) y validaciones (DNI, fecha).
  - Permite ejercitar persistencia relacional, transacciones (crear historial + paciente) y reglas de negocio (datos obligatorios, grupo sanguíneo válido, marcado lógico de eliminado).
  - Es realista y contiene casos de uso típicos: alta/lectura/actualización/baja (CRUD), listados y búsquedas por id.
- 

## 2. Diseño: decisiones clave

### 2.1 Modelo de datos (decisión 1→1)

Se implementó una relación **1→1** entre Paciente y HistorialClinica. En el código observado, la entidad Paciente tiene un atributo HistorialClinica historial y en las sentencias SQL aparece `historial_clinica_id` en la tabla paciente. Esto indica que la relación se implementa con **FK en la tabla paciente** apuntando a la PK id de `historial_clinica`.

**Alternativas y decisión tomada:**

- **PK compartida (historial usa el mismo id que paciente):**
  - Ventaja: garantiza 1↔1 fuerte, elimina la posibilidad de inconsistencia.
  - Desventaja: complica inserciones (hay que insertar una y usar su id para la otra) y no siempre es deseable.
- **FK única en paciente (decisión actual):**
  - Se crea la fila en `historial_clinica` y se referencia desde paciente con un FK único (índice UNIQUE sobre `historial_clinica_id`).
  - Ventaja: implementación más simple y flexible.

### 2.2 Estructura de tablas

```

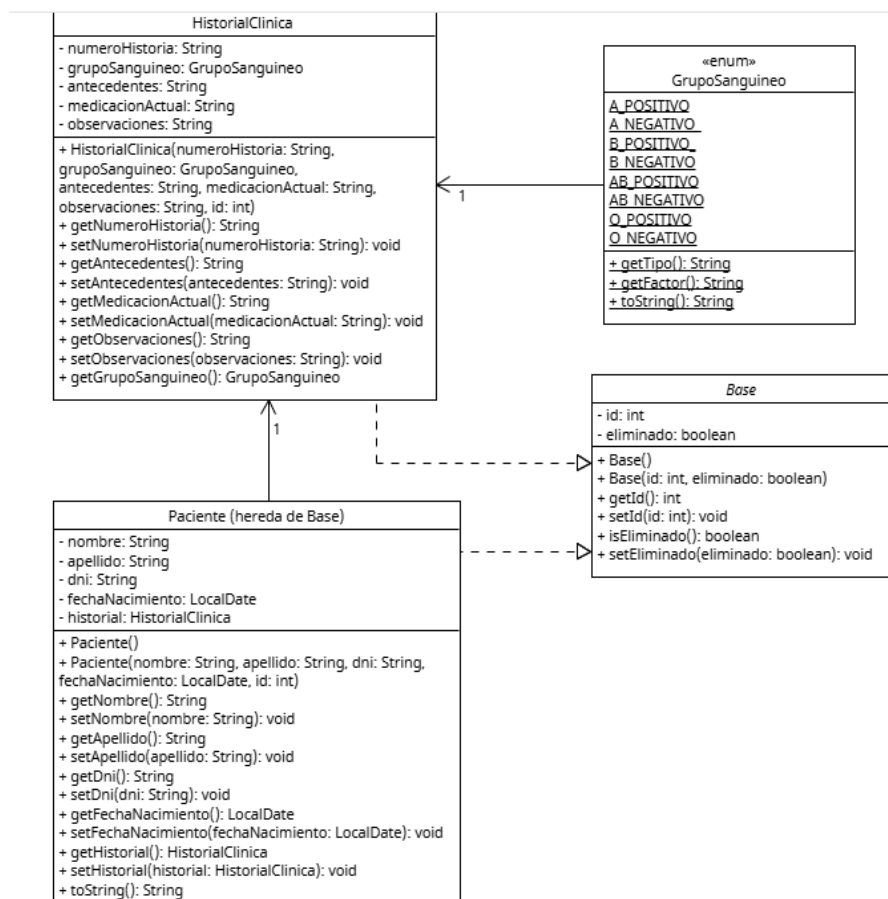
CREATE TABLE historia_clinica (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    eliminado BOOLEAN NOT NULL DEFAULT FALSE,
    nro_historia VARCHAR(20) UNIQUE,
    grupo_sanguineo ENUM('A+', 'A-', 'B+', 'B-', 'AB+', 'AB-', 'O+', 'O-'),
    antecedentes TEXT,
    medicacion_actual TEXT,
    observaciones TEXT,
    CONSTRAINT chk_nro_historia_len CHECK(CHAR_LENGTH(nro_historia) <= 20)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE paciente (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    eliminado BOOLEAN NOT NULL DEFAULT FALSE,
    nombre VARCHAR(80) NOT NULL, apellido VARCHAR(80) NOT NULL,
    dni VARCHAR(15) NOT NULL UNIQUE, fecha_nacimiento DATE,
    historia_clinica_id BIGINT UNIQUE,
    CONSTRAINT chk_nombre_len CHECK(CHAR_LENGTH(nombre) <= 80),
    CONSTRAINT chk_apellido_len CHECK (CHAR_LENGTH(apellido) <= 80),
    CONSTRAINT chk_dni_len CHECK (CHAR_LENGTH(dni) <= 15),
    CONSTRAINT fk_paciente_historia FOREIGN KEY (historia_clinica_id) REFERENCES
    historia_clinica(id) ON DELETE RESTRICT ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

```

### 3. UML

A continuación incluimos un diagrama UML de clases en texto (UMLetino)



## 4. Arquitectura por capas

Se aplicó una arquitectura por capas / paquetes:

- **model:** clases del dominio (Paciente, HistorialClinica, Base, GrupoSanguineo).
  - Responsabilidad: representar datos y sus comportamientos simples (getters/setters).
- **dao:** objetos de acceso a datos (GenericDAO<T>, PacienteDAO, HistorialDAO).
  - Responsabilidad: operaciones CRUD con la base de datos (consultas SQL y manejo de Connection/PreparedStatement).
- **service:** lógica de negocio (GenericService, PacienteServiceImpl, HistorialServiceImpl).
  - Responsabilidad: validaciones, reglas de negocio, coordinación entre DAOs, mensajes de error de alto nivel.
- **main / view:** interfaz de consola (AppMenu, main).
  - Responsabilidad: interacción con el usuario, lectura/escritura por consola, y delegación a los servicios.

**Responsabilidades por paquete:** model = DTOs; dao = persistencia; service = reglas/validaciones; main/view = UI/clase de inicio.

---

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

### 5.1 Orden de operaciones para un alta con historial

Caso: "Crear paciente con historial" (desde AppMenu.crearPacienteConHistorial()):

1. Se crea HistorialClinica con datos ingresados por el usuario.
2. Se invoca `historialService.insertar(historial)` → que llama a `HistorialDAO.insertar(...)` y persiste la fila en `historial_clinica`.
3. Se setea `paciente.setHistorial(historial)` y se invoca `pacienteService.insertar(paciente)` → que llama a `PacienteDAO.insertar(...)` y persiste la fila en `paciente` con `historial_clinica_id`.

### 5.2 Transacciones

- **HistorialDAO.insertar(...):** establece `conn.setAutoCommit(false)`, ejecuta `executeUpdate()` y hace `conn.commit()` si `rowsAffected >= 1`; en catch hace `conn.rollback()` y finalmente `conn.setAutoCommit(true)`. Esto asegura atomicidad dentro de la inserción del historial.
- **PacienteDAO.insertar(...):** establece `conn.setAutoCommit(false)`, ejecuta `executeUpdate()` y hace `conn.commit()` si `rowsAffected >= 1`; en catch hace `conn.rollback()` y finalmente `conn.setAutoCommit(true)`. Esto asegura atomicidad dentro de la inserción del historial.

### 5.3 Dónde se hace commit/rollback

- Commit/rollback parcial: en HistorialDAO (commit/rollback dentro del DAO).
  - PacienteDAO: (commit/rollback dentro del DAO).
-

## 6. Validaciones y reglas de negocio

### Reglas implementadas en PacienteServiceImpl:

- nombre, apellido, dni no nulos ni vacíos.
- insertar imprime mensaje de log.

### Reglas implementadas en HistorialServiceImpl:

- Validación de numeroHistoria no vacío.
  - Validación de grupo sanguineo contra List.of("A+", "A-", ... )
- 

## 7. Pruebas realizadas (evidencias y consultas SQL útiles)

Como la UI es por consola, a continuación se muestran ejemplos de entradas/salidas y consultas SQL que sirven para corroborar resultados.

### 7.1 Ejemplo de ejecución y captura del menú e inserción de paciente(flujo de consola)

```
===== MENU =====
1. Crear paciente con Historial
2. Listar pacientes
3. Listar paciente por id
4. Actualizar Paciente

5. Eliminar Paciente por ID
6. crear Historial
7. Listar Historial
8. Leer Historial por ID
9. Actualizar Historial
10. Eliminar Historial
0. Salir
Ingrese una opcion: 1
Ingrese el nombre: Cristian
Ingrese el apellido:aguirre
Ingrese el DNI: 44123412
Ingrese su fecha en formato yyyy-mm-dd: 2002-12-19
=== CREANDO HISTORIAL ===

Ingrese el numero Historial: 50
1)A+
2)A-
3)B+
4)B-
5)AB+
6)AB-
7)O+
8)O-
Ingrese el Grupo Sanguíneo:
5
Ingrese los antecedentes:
se quebro
Ingrese su medicacion:
pastillas
Observaciones:
descanso 30 dias
Insertando paciente: Cristian
```

### 7.2 Consultas SQL útiles

-- Ver todos los pacientes activos

```
SELECT id, nombre, apellido, dni, fecha_nacimiento, historial_clinica_id FROM paciente WHERE
eliminado = FALSE;
```

```
-- Ver todos los historiales
SELECT id, numero_historia, grupo_sanguineo, antecedentes, medicacion_actual, observaciones FROM
historial_clinica WHERE eliminado = FALSE;
```

```
-- Buscar paciente por id (y su historial)
SELECT p.*, h.numero_historia, h.grupo_sanguineo
FROM paciente p
LEFT JOIN historial_clinica h ON p.historial_clinica_id = h.id
WHERE p.id = 3;
```

---

## 8. Conclusiones y mejoras futuras

Durante el desarrollo del trabajo práctico pudimos aplicar los contenidos principales de Programación 2: arquitectura por capas, acceso a datos con JDBC, validaciones, uso de DAOs, servicios y entidades del dominio. El proyecto nos permitió comprender mejor cómo interactúan la lógica de negocio, la persistencia y la interfaz, y cómo cada capa cumple un rol específico.

El proceso también nos mostró la importancia de probar cada módulo, revisar consultas SQL y mantener consistencia en los modelos. Detectamos varios errores comunes que nos ayudaron a entender la relevancia de una correcta validación y del manejo adecuado de transacciones para evitar datos inconsistentes.

### Mejoras futuras propuestas:

- Unificar el manejo de transacciones en la capa de servicios.
  - Ampliar las validaciones (DNI, fechas, longitudes, etc.).
  - Completar todos los métodos CRUD.
  - Mejorar el manejo de excepciones y mensajes al usuario.
  - Extender el sistema con más funcionalidades, como gestión de turnos o usuarios.
- 

## 9. Fuentes y herramientas utilizadas

- Java - versión 24
  - MySQL - versión 8.x
  - Apache NetBeans - versión 25
  - Driver MySQL Connector - versión 8.4.0
  - HikariCP-6.2.1.jar
  - slf4-api-1.7.36.jar
  - slf4-simple-1.7.36.jar
- 

### Anexos (código relevante y consultas)

- Se adjunta código fuente (DAO, Service, AppMenu, model, config) — incluidos en el repositorio/proyecto.