TRABAJO PRACTICO INTEGRADOR

PROGRAMACION II

TECNICATURA UNIVERSITARIA EN PROGRAMACION

GRUPO 9

- Masseroni Ayelen | *Desarrollo*

https://github.com/M4ss3A/TPI_ProgramacionII

- Nicolas Demiryi | *Desarrollo*

https://github.com/Nicolasdemiryi/TPI_ProgramacionII

- German Dagatti | *Desarrollo*

https://github.com/GabrielTorres25/TPI_ProgramacionII

- Gabriel Torres | *Desarrollo*

https://github.com/GabrielTorres25/TPI_ProgramacionII

LinkVideo: ***https://www.youtube.com/watch?v=ymQMyaYP5bY***

1- Diseño

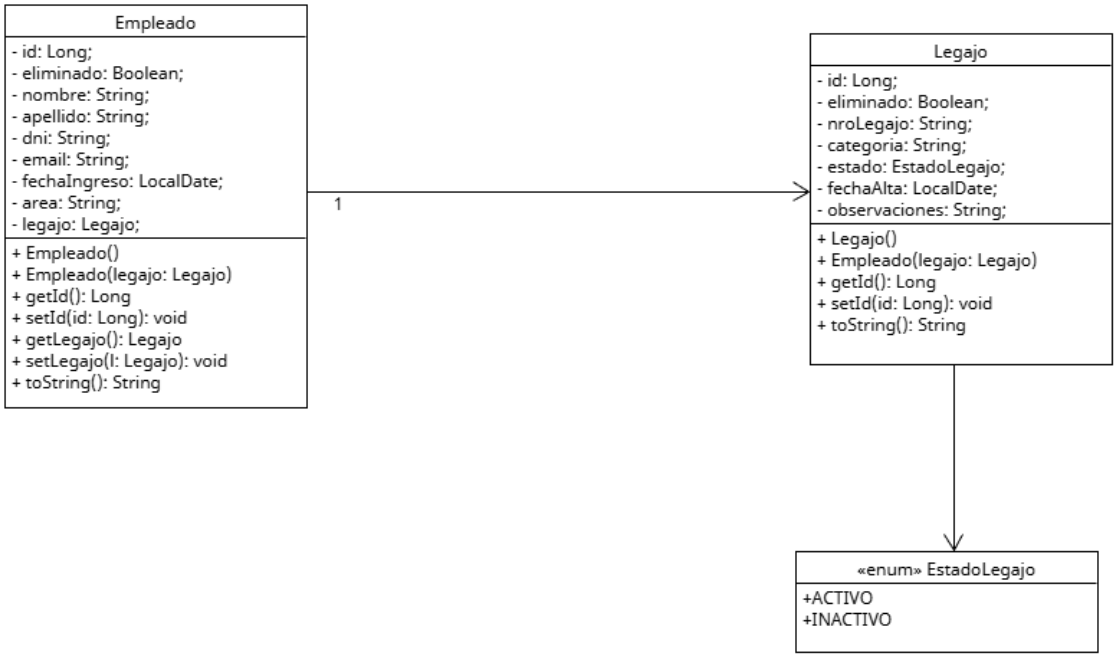
Para el Trabajo Final Integrador se seleccionó el par de entidades Empleado → Legajo. El sistema debe permitir registrar información personal de los empleados y los datos administrativos de su legajo. El modelo del dominio se basa en dos entidades principales y una relación uno a uno, según la consigna.

La relación entre las entidades es 1→1 unidireccional, donde:

- Empleado (A) conoce a Legajo (B) mediante un atributo legajo.
- Legajo (B) no referencia a Empleado.
- Existe exactamente un legajo por cada empleado, y cada legajo pertenece a un único empleado.

Esta elección resulta coherente con el Trabajo Final Integrador de Bases de Datos, donde se utilizó la misma estructura relacional y la misma cardinalidad.

Diagrama UML



2- Entidades

Las entidades del dominio fueron implementadas en el paquete Entities, cumpliendo con la especificación solicitadas. Ambas clases cuentan con un constructor por defecto, constructor completo, getters y setters, y un método toString() descriptivo.

La relación unidireccional se modeló desde Empleado hacia Legajo, garantizando que cada Empleado tenga exactamente un Legajo asociado.

La clase Empleado (A) : La entidad Empleado representa a una persona que trabaja en la organización. Contiene datos personales, laborales y un enlace hacia su legajo.

ATRIBUTOS

id : Long → Identificador único.

eliminado : boolean → Indica baja lógica.

nombre : String → Nombre del empleado.

apellido : String → Apellido del empleado.

dni : String → Documento nacional de identidad.

email : String → Correo electrónico.

fechaIngreso : LocalDate → Fecha en que ingresó a la empresa.

area : String → Área o sector donde trabaja.

detalle : Legajo → Referencia al legajo asociado (relación 1 → 1).

```java
public class Empleado {

    private Long id;
    private boolean eliminado;
    private String nombre;
    private String apellido;
    private String dni;
    private String email;
    private LocalDate fechaIngreso;
    private String area;
    private Legajo detalle;      // referencia 1→1 a Legajo

    public Empleado() {
    }

    public Empleado(Long id, boolean eliminado, String nombre, String apellido,
                    String dni, String email, LocalDate fechaIngreso,
                    String area, Legajo detalle) {
        this.id = id;
        this.eliminado = eliminado;
        this.nombre = nombre;
        this.apellido = apellido;
        this.dni = dni;
        this.email = email;
        this.fechaIngreso = fechaIngreso;
        this.area = area;
        this.detalle = detalle;
    }

    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public boolean isEliminado() { return eliminado; }
    public void setEliminado(boolean eliminado) { this.eliminado = eliminado; }
```

```java
    public String getDni() { return dni; }
    public void setDni(String dni) { this.dni = dni; }

    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }

    public LocalDate getFechaIngreso() { return fechaIngreso; }
    public void setFechaIngreso(LocalDate fechaIngreso) { this.fechaIngreso = fechaIngreso; }

    public String getArea() { return area; }
    public void setArea(String area) { this.area = area; }

    public Legajo getDetalle() { return detalle; }
    public void setDetalle(Legajo detalle) { this.detalle = detalle; }

    @Override
    public String toString() {
        return "Empleado{" +
                "id=" + id +
                ", eliminado=" + eliminado +
                ", nombre='" + nombre + '\'' +
                ", apellido='" + apellido + '\'' +
                ", dni='" + dni + '\'' +
                ", email='" + email + '\'' +
                ", fechaIngreso=" + fechaIngreso +
                ", area='" + area + '\'' +
                ", detalle=" + (detalle != null ? detalle.getId() : null) +
                '}';
    }
```

La clase Legajo (B): La entidad Legajo representa la documentación administrativa del empleado.

ATRIBUTOS

id : Long → Identificador único.

eliminado : boolean → Indica baja lógica.

nroLegajo : String → Número de legajo asignado.

categoria : String → Categoría laboral.

estado : EstadoLegajo → Estado del legajo (ACTIVO/INACTIVO).

fechaAlta : LocalDate → Fecha en que se generó el legajo.

observaciones : String → Información adicional o notas internas.

```java
public class Legajo {

    private Long id;
    private boolean eliminado;
    private String nroLegajo;
    private String categoria;
    private EstadoLegajo estado;
    private LocalDate fechaAlta;
    private String observaciones;

    public Legajo() {
    }

    public Legajo(Long id, boolean eliminado, String nroLegajo,
                  String categoria, EstadoLegajo estado,
                  LocalDate fechaAlta, String observaciones) {
        this.id = id;
        this.eliminado = eliminado;
        this.nroLegajo = nroLegajo;
        this.categoria = categoria;
        this.estado = estado;
        this.fechaAlta = fechaAlta;
        this.observaciones = observaciones;
    }

    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public boolean isEliminado() { return eliminado; }
    public void setEliminado(boolean eliminado) { this.eliminado = eliminado; }

    public String getNroLegajo() { return nroLegajo; }
    public void setNroLegajo(String nroLegajo) { this.nroLegajo = nroLegajo; }

    public String getCategoria() { return categoria; }
    public void setCategoria(String categoria) { this.categoria = categoria; }

    public EstadoLegajo getEstado() { return estado; }
    public void setEstado(EstadoLegajo estado) { this.estado = estado; }

    public LocalDate getFechaAlta() { return fechaAlta; }
    public void setFechaAlta(LocalDate fechaAlta) { this.fechaAlta = fechaAlta; }

    public String getObservaciones() { return observaciones; }
    public void setObservaciones(String observaciones) { this.observaciones = observaciones;

    @Override
    public String toString() {
        return "Legajo{" +
                "id=" + id +
                ", eliminado=" + eliminado +
                ", nroLegajo='" + nroLegajo + '\'' +
                ", categoria='" + categoria + '\'' +
                ", estado=" + estado +
                ", fechaAlta=" + fechaAlta +
                ", observaciones='" + observaciones + '\'' +
                '}';
    }
}
```

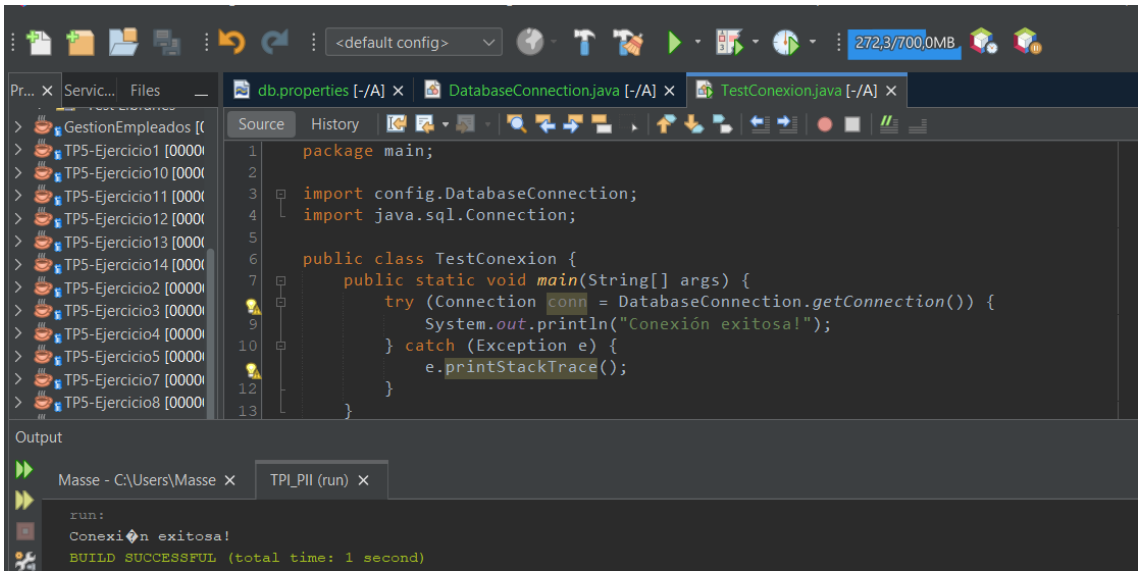
3- Base de datos (MySQL)

Para la persistencia se reutiliza la base de datos diseñada en el Trabajo Final Integrador de Bases de Datos, ya que el dominio es el mismo (Empleado → Legajo) y mantiene la relación 1→1 unidireccional requerida.

Para que el proyecto Java pueda interactuar con dicha base, se implementó un módulo de conexión basado en JDBC.

Está compuesta por:

- Script de creación – estructura.sql - Incluye las sentencias SQL necesarias para crear las tablas del sistema, especialmente la tabla empleado, cuyo campo primario es legajo.
- Script de carga inicial – datos.sql: Contiene registros de ejemplo para verificar el correcto funcionamiento de las operaciones de lectura y escritura.
- Archivo de configuración – db.properties: Se creó un archivo dentro del paquete config que permite parametrizar la conexión
- Clase DatabaseConnection- Dentro del paquete config se desarrolló la clase encargada de: cargar el archivo db.properties, registrar el driver JDBC, crear la conexión usando DriverManager, aplicar patrón Singleton (reutiliza la misma conexión).
- Incorporación del driver JDBC- Para que Java pueda comunicarse con
- Prueba de la conexión – TestConexion- Se implementó una clase de verificación que intenta obtener una conexión y muestra el resultado: Conexión exitosa!



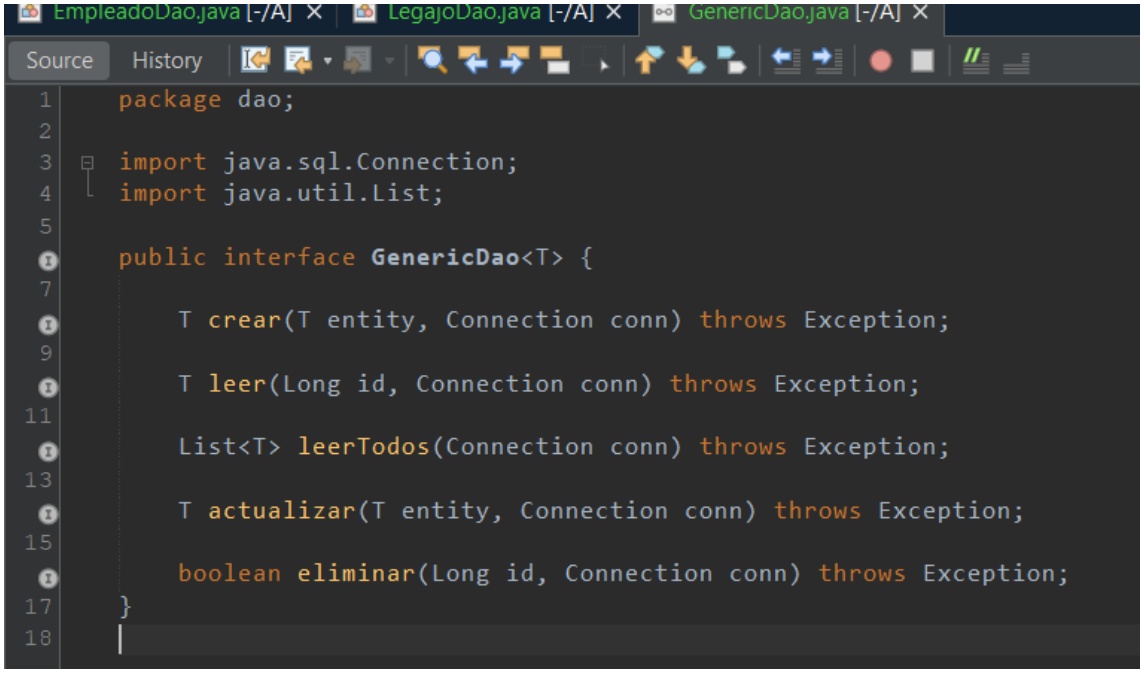
De esta manera se verificó el correcto funcionamiento del módulo.

4- DAO

En esta sección se implementó la capa DAO con el objetivo de desacoplar la lógica de acceso a datos del resto de la aplicación, siguiendo el patrón DAO.

Se desarrolló la interfaz genérica **GenericDao<T>,** que define las operaciones CRUD fundamentales que deben respetar todos los repositorios del sistema. Esta interfaz incluye los métodos:

- crear(T entity, Connection conn)
- leer(Long id, Connection conn)
- leerTodos(Connection conn)
- actualizar(T entity, Connection conn)
- eliminar(Long id, Connection conn)



DAO concreto para la Entidad B – Legajo

Se creó la clase LegajoDao, que implementa GenericDao<Legajo> y encapsula todas las operaciones de persistencia sobre la tabla legajo.

Cada método utiliza PreparedStatement para:

- *insertar nuevos legajos*
- *consultar legajos por ID*
- *obtener todos los registros no eliminados*
- *actualizar datos existentes*

- *realizar baja lógica mediante el campo eliminado*

La clase cumple con la responsabilidad única de administrar exclusivamente la entidad Legajo.

DAO concreto para la Entidad A – Empleado

Se desarrolló la clase EmpleadoDao, que implementa GenericDao<Empleado> y gestiona todos los accesos a la tabla empleado.

Además de los atributos propios del empleado, este DAO persiste la relación 1→1 con Legajo mediante la columna legajo_id.

Cada operación utiliza PreparedStatement:

- para insertar un nuevo empleado junto con su referencia al legajo correspondiente
- consultar empleados por ID
- listar todos los empleados activos (no eliminados)
- actualizar datos de empleado
- realizar baja lógica estableciendo eliminado = true

Este diseño asegura independencia entre los DAOs y permite que la capa de servicio combine ambas operaciones dentro de una misma transacción.

5- Service

La capa Service constituye el nivel de lógica de negocio del sistema y se encarga de orquestar las operaciones que involucran a las entidades del dominio. Su función principal es asegurar que las reglas del negocio se cumplan correctamente, centralizando validaciones, coordinando interacciones entre múltiples DAO y administrando las transacciones necesarias para preservar la consistencia de los datos.

En esta capa se implementan decisiones clave del modelo, tales como:

- aplicación de reglas de validación previa,
- control de la relación 1→1 entre Empleado y Legajo,
- administración explícita de transacciones mediante JDBC, incluyendo commit y rollback.

Con el fin de cumplir con los requerimientos del trabajo integrador, se desarrollaron tres servicios:

GenericService

Define la interfaz común que deben implementar los servicios del dominio.
Establece la estructura básica de operaciones de negocio mediante los métodos: crear, leer, leerTodos, actualizar y eliminar.
Este enfoque promueve abstracción, reutilización y uniformidad en la implementación de servicios.

LegajoService

Implementa las reglas de negocio específicas de la entidad *Legajo* (entidad B).
Se encarga de validar los campos propios del legajo (número, categoría, estado, etc.) y de ejecutar operaciones CRUD mediante *LegajoDao*.
Este servicio utiliza una conexión independiente para cada operación simple, dado que la manipulación del Legajo en forma aislada no requiere transacciones compuestas.

EmpleadoService

Es el servicio más complejo, ya que gestiona la entidad *Empleado* (entidad A) y coordina la creación, actualización y asociación de un *Legajo*.
Además de implementar los métodos CRUD definidos por GenericService<Empleado>, incorpora lógica de negocio adicional:

- Validación completa del empleado (nombre, apellido, DNI, email, área, etc.).
- Asociación entre Empleado y Legajo garantizando la regla de unicidad 1→1.
- Ejecución de una operación transaccional compuesta: **crear el Legajo, asignarlo al Empleado y finalmente persistir el Empleado** utilizando una única conexión JDBC.

Dentro de este servicio, el método transaccional controla explícitamente el ciclo setAutoCommit(false) → commit() / rollback() → setAutoCommit(true), asegurando que los cambios sean atómicos.
Si ocurre un error en cualquiera de las operaciones, se revierte toda la transacción para evitar inconsistencias como empleados sin legajo o legajos huérfanos.

Validaciones - Empleado

```
    // ===== VALIDACIONES =====
    private void validarEmpleado(Empleado empleado) throws Exception {
        if (empleado == null)
            throw new Exception("El empleado no puede ser null.");

        if (empleado.getNombre() == null || empleado.getNombre().trim().isEmpty())
            throw new Exception("El nombre del empleado es obligatorio.");

        if (empleado.getApellido() == null || empleado.getApellido().trim().isEmpty())
            throw new Exception("El apellido del empleado es obligatorio.");

        if (empleado.getDni() == null || empleado.getDni().trim().isEmpty())
            throw new Exception("El DNI es obligatorio.");

        if (!empleado.getDni().matches("\\d+"))
            throw new Exception("El DNI debe ser numérico.");

        if (empleado.getEmail() != null && !empleado.getEmail().isEmpty() &&
                !empleado.getEmail().matches("^[\\w.-]+@[\\w.-]+\\.[A-Za-z]{2,6}$"))
            throw new Exception("El email no tiene un formato válido.");

        if (empleado.getArea() != null && empleado.getArea().length() > 50)
            throw new Exception("El área no puede superar los 50 caracteres.");
    }
    private void validarLegajo(Legajo legajo) throws Exception {
        if (legajo == null)
            throw new Exception("El legajo no puede ser null.");

        if (legajo.getNroLegajo() == null || legajo.getNroLegajo().trim().isEmpty())
            throw new Exception("El número de legajo es obligatorio.");

        if (legajo.getNroLegajo().length() > 20)
            throw new Exception("El número de legajo no puede superar los 20 caracteres.");

        if (legajo.getCategoria() != null && legajo.getCategoria().length() > 30)
            throw new Exception("La categoría no puede superar los 30 caracteres.");

        if (legajo.getEstado() == null)
            throw new Exception("El estado del legajo es obligatorio.");
```

Validaciones – Legajo

```
    private void validarLegajo(Legajo legajo) throws Exception {
        if (legajo == null)
            throw new Exception("El legajo no puede ser null.");

        if (legajo.getNroLegajo() == null || legajo.getNroLegajo().trim().isEmpty())
            throw new Exception("El número de legajo es obligatorio.");

        if (legajo.getNroLegajo().length() > 20)
            throw new Exception("El número de legajo no puede superar los 20 caracteres.");

        if (legajo.getCategoria() != null && legajo.getCategoria().length() > 30)
            throw new Exception("La categoría no puede superar los 30 caracteres.");

        if (legajo.getEstado() == null)
            throw new Exception("El estado del legajo es obligatorio.");
    }
```

Test Service

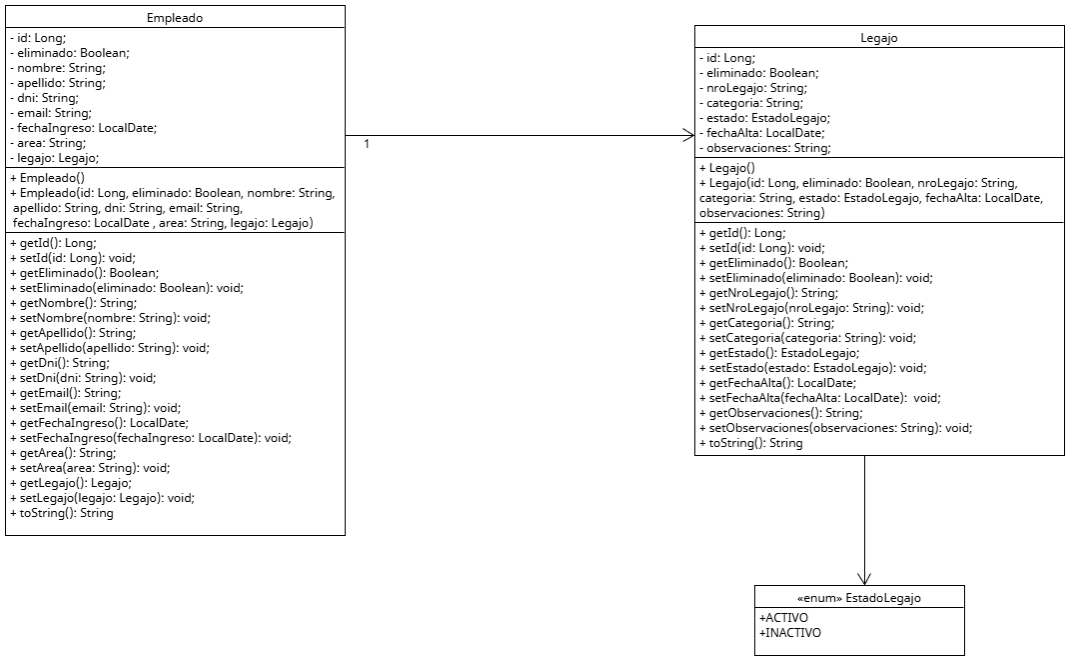
```
package main;

import Entities.Empleado;
import Entities.EstadoLegajo;
import Entities.Legajo;
import Service.EmpleadoService;

import java.time.LocalDate;

public class TestService {

    public static void main(String[] args) {

        try {
            EmpleadoService service = new EmpleadoService();

            // ========================
            // PRUEBA 1: CASO EXITOSO
            // ========================
            Legajo legajoOk = new Legajo(
                    100L,
                    false,
                    "LEG-100",
                    "Junior",
                    EstadoLegajo.ACTIVO,
                    LocalDate.of(2024, 1, 1),
                    "Alta normal"
            );

            Empleado empleadoOk = new Empleado(
                    100L,
                    false,
                    "Ayelen",
                    "Masseroni",
                    "40123456",
                    "ayelen@example.com",
                    LocalDate.of(2024, 1, 10),
                    "Sistemas",
                    null
```

```
            System.out.println("====== PRUEBA 1: CASO EXITOSO ======");
            service.crearEmpleadoConLegajo(empleadoOk, legajoOk);
            System.out.println(">> Fin de PRUEBA 1\n");

            // ========================
            // PRUEBA 2: CASO CON ERROR
            // ========================
            Legajo legajoError = new Legajo(
                    101L,
                    false,
                    "", // nroLegajo vacío -> error
                    "Senior",
                    EstadoLegajo.ACTIVO,
                    LocalDate.of(2024, 2, 1),
                    "Prueba con error"
            );

            Empleado empleadoError = new Empleado(
                    101L,
                    false,
                    "",             // nombre vacío -> error
                    "Gomez",
                    "40999888",
                    "correo_malformado", // email inválido
                    LocalDate.of(2024, 2, 10),
                    "IT",
                    null
            );

            System.out.println("====== PRUEBA 2: CASO CON ERROR (VALIDACIÓN + ROLLBACK) ======");
            service.crearEmpleadoConLegajo(empleadoError, legajoError);
            System.out.println(">> Fin de PRUEBA 2\n");

        } catch (Exception e) {
            System.out.println("Ocurrió una excepción en TestService:");
            System.out.println(e.getMessage());
            e.printStackTrace();
        }
```

Resultado

```
Output - TPI_PII (run)
run:
====== PRUEBA 1: CASO EXITOSO ======
Error en la transacci□n. Se realiz□ rollback.
Detalle del error: Duplicate entry '100' for key 'PRIMARY'
Ocurri□ una excepci□n en TestService:
Duplicate entry '100' for key 'PRIMARY'
java.sql.SQLIntegrityConstraintViolationException: Duplicate entry '100' for key 'PRIMARY'
        at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java:109)
        at com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException(SQLExceptionsMapping.java:114)
        at com.mysql.cj.jdbc.ClientPreparedStatement.executeInternal(ClientPreparedStatement.java:988)
        at com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdateInternal(ClientPreparedStatement.java:1166)
        at com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdateInternal(ClientPreparedStatement.java:1101)
        at com.mysql.cj.jdbc.ClientPreparedStatement.executeLargeUpdate(ClientPreparedStatement.java:1448)
        at com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdate(ClientPreparedStatement.java:1084)
        at Dao.LegajoDao.crear(LegajoDao.java:33)
        at Service.EmpleadoService.crearEmpleadoConLegajo(EmpleadoService.java:118)
        at main.TestService.main(TestService.java:43)
BUILD SUCCESSFUL (total time: 2 seconds)
```

Diagrama UML final



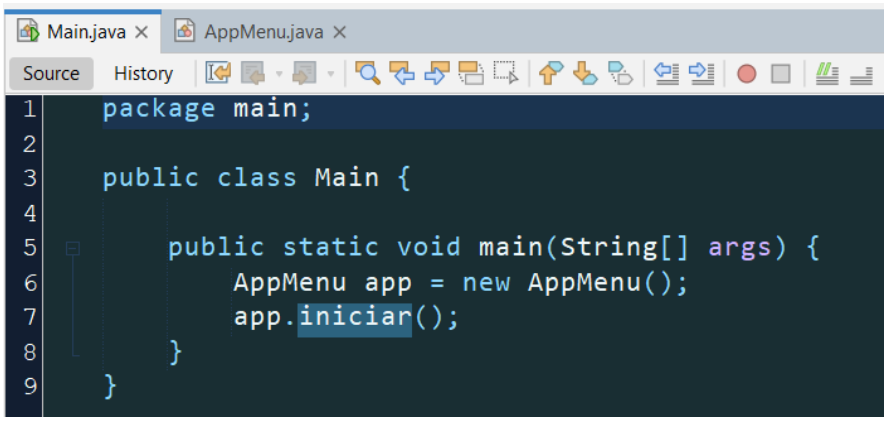
6- Main

Nos centraremos en dos archivos clave, Main.java, punto de entrada de la aplicación, y AppMenu.java, que implementa la interfaz de usuario por consola.

Main.java

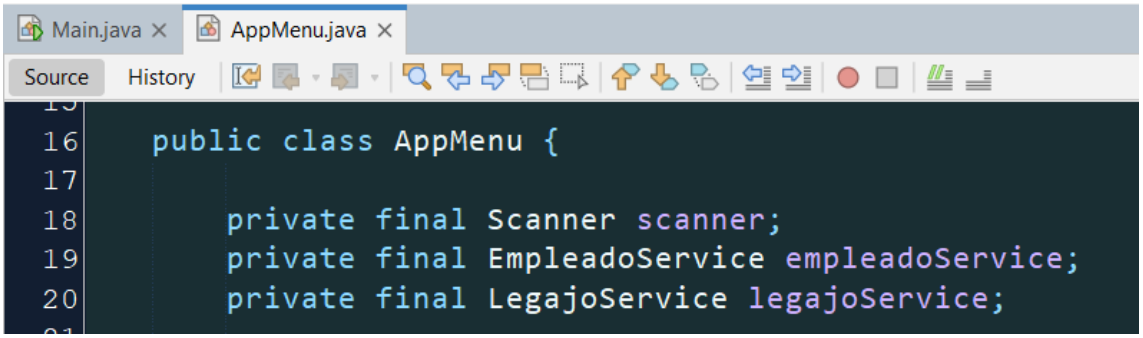
Este archivo actúa como clase de entrada del programa. Contiene únicamente el método main, este diseño respalda el principio de responsabilidad única,

- Se instancia la clase AppMenu, que encapsula toda la lógica de interacción con el usuario.
- Se invoca al método iniciar(), que inicia el bucle principal de la aplicación.



AppMenu.java

Es una clase que gestiona la interacción con el usuario a través de consola. Utiliza java.util.Scanner para capturar entradas y delega la lógica de negocio a los servicios EmpleadoService y LegajoService. Scanner, para leer entradas del usuario. EmpleadoService y LegajoService, clases de servicio que abstraen el acceso a datos y la lógica de negocio.



Método iniciar()

Este método contiene el bucle principal de la aplicación. Se ejecuta hasta que el usuario elige salir.

Utiliza try-with-resources para garantizar el cierre automático del Scanner. El menú es textual, con navegación por consola. Se capturan excepciones genéricas para evitar caídas del programa.

```
------ MENÚ EMPLEADOS ------
1) Crear Empleado + Legajo (transacción)
2) Ver Empleado por ID
3) Listar Empleados
4) Actualizar Empleado
5) Eliminar lógico Empleado
0) Volver
---------------------------
Elija una opción: 0

========== MENÚ PRINCIPAL ==========
1) Gestión de Empleados
2) Gestión de Legajos
3) Buscar empleado por DNI
0) Salir
====================================
Elija una opción: 0
Saliendo de la aplicación...
```

Transaccionalidad

Un punto destacado es el método crearEmpleadoConLegajo(), que ilustra el uso de una transacción a nivel de servicio. Este método garantiza que ambas entidades se persistan de forma atómica. Si ocurre un error, se deshacen los cambios.

Conclusión

El diseño de esta aplicación refleja buenas prácticas como:

- Separación de responsabilidades.
- Manejo centralizado de entradas de usuario.
- Uso de servicios para encapsular la lógica de negocio.
- Validación exhaustiva de entradas.