Trabajo Final Integrador (TFI)

Aplicación Java con relación 1→1 unidireccional + DAO + MySQL

Objetivo

Desarrollar 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.

Grupo97_Libro--FichaBibliografica

Integrantes del grupo

Sergio Andres Montenegro – Comisión 17
Bruno Morales – Comisión 17
Esteban Daniel Miguel – Comisión 17
Santiago Julian Pace – Comisión 18

Consignas generales

- Lenguaje: **Java** (recomendado 21).
- Persistencia: **JDBC** (sin ORM) con **MySQL**.
- Patrón de diseño: **DAO** y **capa Service**.
- **Código limpio**, legible, con **manejo de excepciones** en todas las capas.
- **Relación 1→1 unidireccional**: sólo la clase "A" contiene el atributo que referencia a "B".
- **Equipos de 4 personas, sin excepción**.

Dominio

- Libro → FichaBibliografica

Estructura del proyecto (paquetes)

- **config/**: conexión a la base (lectura de propiedades externas).
- **entities/**: clases de dominio (A y B) con **id** y **eliminado** (baja lógica).
- **dao/**: interfaces genéricas y DAOs concretos (JDBC + PreparedStatement).
- **service/**: reglas de negocio, validaciones y **orquestación de transacciones**.

- **main/**: arranque de la app y **AppMenu** (consola).

Requerimientos técnicos

1) Diseño (previo al código)

- **Diagrama UML de clases** con:
  - Atributos (tipo/visibilidad) y métodos (firma/visibilidad).
  - **A → B (1..1)** unidireccional explícita.
  - Paquetes y dependencias principales.
- El UML guía toda la implementación.

2) Entidades (genérico)

- **A**: atributos propios + `id` (INT/BIGINT) + `eliminado` (BOOLEAN).
- **B**: atributos propios + `id` + `eliminado`.
- **A** contiene `private B detalle;` (o nombre equivalente).
- **Constructores** (vacío y completo), getters/setters y `toString()` legible.

3) Base de datos (MySQL)

- Clase **DatabaseConnection** en **config/** con método estático que retorne **java.sql.Connection**.
- Entregar un archivo **.sql con las instrucciones para crear la base de datos y sus tablas** (CREATE DATABASE, CREATE TABLE, claves primarias, foráneas, índices y restricciones necesarias para 1→1).
- Entregar **otro .sql con datos de prueba** (INSERTs) para levantar el proyecto desde cero.
- **Relación 1→1 recomendada**: **clave foránea única** en la tabla de **B** (campo `a_id` con `UNIQUE`, `FOREIGN KEY a A(id)`, `ON DELETE CASCADE`). Alternativa válida: **clave primaria compartida** (la PK de B es también FK a A).

4) DAO

- `GenericDao<T>` con: `crear(T)`, `leer(long id)`, `leerTodos()`, `actualizar(T)`, `eliminar(long id)`.
- DAOs concretos para A y B usando **PreparedStatement** en todas las operaciones.
- Los DAO deben ofrecer métodos que **acepten una Connection externa** (para participar de la misma transacción).

5) Service (transacciones obligatorias)

- GenericService<T>: insertar, actualizar, eliminar, getById, getAll.
- AService y BService:
  - **Abrir transacción**: setAutoCommit(false) sobre una conexión compartida.
  - Ejecutar operaciones compuestas (por ej., crear B, asociarla a A y crear A).
  - **commit()** si todo OK; **rollback()** ante cualquier error. o Restablecer autoCommit(true) y cerrar recursos.
  - **Validaciones** (campos obligatorios, formatos según dominio) y **regla 1→1** (impedir más de un B por A).

6) AppMenu (consola)

- Main invoca AppMenu.
- Convertir entradas a **mayúsculas** donde aplique.
- Debe permitir **CRUD completo** de A y B: crear, leer por ID, listar, actualizar y **eliminar lógico**.
- **Incluir al menos una búsqueda por un campo relevante (ej.: DNI, dominio, ISBN, etc., según el dominio elegido).**
- Manejo robusto de:
  - Entradas inválidas (parseos numéricos, formatos).
  - IDs inexistentes.
  - Errores de base de datos y violaciones de unicidad.
- Mensajes claros de éxito/error.

Especificación de campos – TP Integrador (1→1 unidireccional)

Se listan los **campos obligatorios** de cada par de clases (A → B) disponible para el TP Integrador. En todos los casos:

- La relación es **unidireccional 1→1** desde **A** hacia **B** (A contiene una referencia a B).
- Ambas clases incluyen **id** (clave primaria) y **eliminado** (baja lógica).

Tip: Los nombres y longitudes son sugeridos; podés ajustarlos manteniendo la semántica, unicidad y la relación 1→1.

1) Libro → FichaBibliografica

Clase Libro (A)

Campo	Tipo(Java)	Reglas/Notas
id	Long	PK
eliminado	Boolean	Baja lógica
titulo	String	NOT NULL, máx. 150
autor	String	NOT NULL, máx. 120
editorial	String	máx. 100
anioEdicion	Integer	
fichaBibliografica	FichaBibliografica	Referencia 1→1

Clase FichaBibliografica (B)

Campo	Tipo(Java)	Reglas/Notas
id	Long	PK
eliminado	Boolean	Baja lógica
isbn	String	**UNIQUE**, máx. 17
clasificacionDewey	String	máx. 20
estanteria	String	máx. 20
idioma	String	máx. 30

Informe Técnico del Proyecto Integrador

1. Objetivo del proyecto

Desarrollar una aplicación de gestión bibliográfica con operaciones CRUD completas sobre entidades Libro y FichaBibliográfica, integrando validaciones, persistencia y control transaccional.

2. Arquitectura y capas

La arquitectura por capas es un enfoque estructurado que permite organizar un sistema en módulos con responsabilidades bien definidas, facilitando el mantenimiento, la escalabilidad y la reutilización del código. Es ampliamente utilizada en aplicaciones empresariales y educativas por su claridad y modularidad.
Consiste en dividir una aplicación en secciones horizontales llamadas *capas*, donde cada una tiene una función específica y se comunica solo con la capa inmediatamente inferior. Este modelo aplica el principio de *Separación de preocupaciones (SoC)*, promoviendo que cada capa se enfoque en una responsabilidad concreta.

1. **Capa de presentación (UI)**
   - Interactúa con el usuario.
   - Recoge entradas y muestra resultados.
   - **En Java, suele estar representada por clases como AppMenu o MenuHandler. En nuestro proyecto utlizamos las dos clases, se encuentran en el paquete Main.**
2. **Capa de lógica de negocio (Service)**
   - Contiene las reglas del negocio.
   - Coordina operaciones entre entidades y persistencia.
   - **Ejemplo: LibroService, FichaBibliograficaService.**
3. **Capa de acceso a datos (DAO)**
   - Se encarga de la comunicación con la base de datos.
   - Realiza operaciones CRUD.
   - **Ejemplo: LibroDAO, FichaBibliograficaDAO.**
4. **Capa de entidades (Model)**
   - Define las estructuras de datos.
   - **Representa objetos del dominio como Libro o FichaBibliografica.**
5. **Capa de infraestructura (Config)**
   - Maneja aspectos técnicos como conexiones, transacciones, seguridad.
   - Ejemplo: **DatabaseConnection, TransactionManager**
   - TransactionManager encapsula la gestión de transacciones JDBC, asegurando rollback automático y cierre seguro.

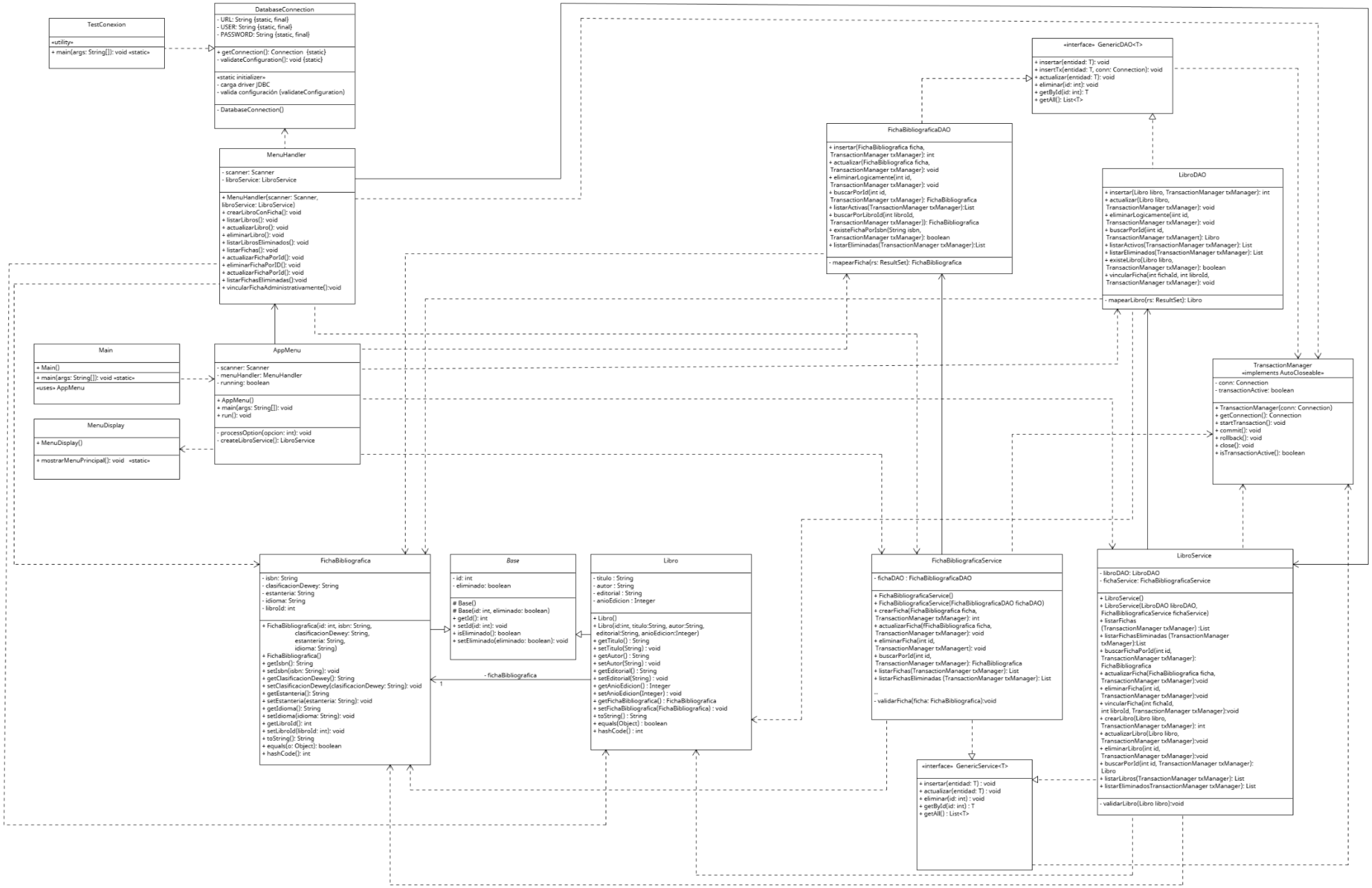
Ventajas de esta arquitectura

- **Modularidad**: cada capa puede evolucionar independientemente.
- **Reutilización**: servicios y DAOs pueden ser reutilizados en múltiples contextos.
- **Escalabilidad**: facilita la incorporación de nuevas funcionalidades.
- **Mantenibilidad**: el código está organizado y es más fácil de depurar.
- **Testabilidad**: permite pruebas unitarias por capa.

Fuentes:

Arquitectura en capas: ¿Qué es y cómo funciona cada capa?
Arquitectura en Capas – Análisis completo + Tradicional vs Modernas, DDD, DIP (Cap 5) – RJ Code Advance
Arquitectura en Capas

2. UML


TestConexion
«utility»
+ main(args: String[]): void «static»
DatabaseConnection
- URL: String {static, final}
- USER: String {static, final}
- PASSWORD: String {static, final}
+ getConnection(): Connection {static}
- validateConfiguration(): void {static}
«static initializer»
- carga driver JDBC
- valida configuración (validateConfiguration)
- DatabaseConnection()
«interface» GenericDAO<T>
+ insertar(entidad: T): void
+ insertTx(entidad: T, conn: Connection): void
+ actualizar(entidad: T): void
+ eliminar(id: int): void
+ getById(id: int): T
+ getAll(): List<T>
MenuHandler
- scanner: Scanner
- libroService: LibroService
+ MenuHandler(scanner: Scanner, libroService: LibroService)
+ crearLibroConFicha(): void
+ listarLibros(): void
+ actualizarLibro(): void
+ eliminarLibro(): void
+ listarLibrosEliminados(): void
+ listarFichas(): void
+ actualizarFichaPorId(): void
+ eliminarFichaPorID(): void
+ actualizarFichaPorId(): void
+ listarFichasEliminadas():void
+ vincularFichaAdministrativamente():void
FichaBibliograficaDAO
+ insertar(FichaBibliografica ficha, TransactionManager txManager): int
+ actualizar(FichaBibliografica ficha, TransactionManager txManager): void
+ eliminarLogicamente(int id, TransactionManager txManager): void
+ buscarPorId(int id, TransactionManager txManager): FichaBibliografica
+ listarActivas(TransactionManager txManager):List
+ buscarPorLibroId(int libroId, TransactionManager txManager)): FichaBibliografica
+ existeFichaPorIsbn(String isbn, TransactionManager txManager): boolean
+ listarEliminadas(TransactionManager txManager):List
- mapearFicha(rs: ResultSet): FichaBibliografica
LibroDAO
+ insertar(Libro libro, TransactionManager txManager): int
+ actualizar(Libro libro, TransactionManager txManager): void
+ eliminarLogicamente(iint id, TransactionManager txManager): void
+ buscarPorId(iint id, TransactionManager txManager): Libro
+ listarActivos(TransactionManager txManager): List
+ listarEliminados(TransactionManager txManager): List
+ existeLibro(Libro libro, TransactionManager txManager): boolean
+ vincularFicha(int fichaId, int libroId, TransactionManager txManager): void
- mapearLibro(rs: ResultSet): Libro
Main
+ Main()
+ main(args: String[]): void «static»
«uses» AppMenu
AppMenu
- scanner: Scanner
- menuHandler: MenuHandler
- running: boolean
+ AppMenu()
+ main(args: String[]): void
+ run(): void
- processOption(opcion: int): void
- createLibroService(): LibroService
MenuDisplay
+ MenuDisplay()
+ mostrarMenuPrincipal(): void «static»
TransactionManager
«implements AutoCloseable»
- conn: Connection
- transactionActive: boolean
+ TransactionManager(conn: Connection)
+ getConnection(): Connection
+ startTransaction(): void
+ commit(): void
+ rollback(): void
+ close(): void
+ isTransactionActive(): boolean
FichaBibliografica
- isbn: String
- clasificacionDewey: String
- estanteria: String
- idioma: String
- libroId: int
+ FichaBibliografica(id: int, isbn: String, clasificacionDewey: String, estanteria: String, idioma: String)
+ FichaBibliografica()
+ getIsbn(): String
+ setIsbn(isbn: String): void
+ getClasificacionDewey(): String
+ setClasificacionDewey(clasificacionDewey: String): void
+ getEstanteria(): String
+ setEstanteria(estanteria: String): void
+ getIdioma(): String
+ setIdioma(idioma: String): void
+ getLibroId(): int
+ setLibroId(libroId: int): void
+ toString(): String
+ equals(o: Object): boolean
+ hashCode(): int
Base
- id: int
- eliminado: boolean
Base()
Base(id: int, eliminado: boolean)
+ getId(): int
+ setId(id: int): void
+ isEliminado(): boolean
+ setEliminado(eliminado: boolean): void
Libro
- titulo : String
- autor : String
- editorial : String
- anioEdicion : Integer
+ Libro()
+ Libro(id:int, titulo:String, autor:String, editorial:String, anioEdicion:Integer)
+ getTitulo() : String
+ setTitulo(String) : void
+ getAutor() : String
+ setAutor(String) : void
+ getEditorial() : String
+ setEditorial(String) : void
+ getAnioEdicion() : Integer
+ setAnioEdicion(Integer) : void
+ getFichaBibliografica() : FichaBibliografica
+ setFichaBibliografica(FichaBibliografica) : void
+ toString() : String
+ equals(Object) : boolean
+ hashCode() : int
- fichaBibliografica
1
FichaBibliograficaService
- fichaDAO : FichaBibliograficaDAO
+ FichaBibliograficaService()
+ FichaBibliograficaService(FichaBibliograficaDAO fichaDAO)
+ crearFicha(FichaBibliografica ficha, TransactionManager txManager): int
+ actualizarFicha(FichaBibliografica ficha, TransactionManager txManager): void
+ eliminarFicha(int id, TransactionManager txManagert): void
+ buscarPorId(int id, TransactionManager txManager): FichaBibliografica
+ listarFichas(TransactionManager txManager): List
+ listarFichasEliminadas (TransactionManager txManager): List
...
- validarFicha(ficha: FichaBibliografica):void
LibroService
- libroDAO: LibroDAO
- fichaService: FichaBibliograficaService
+ LibroService()
+ LibroService(LibroDAO libroDAO, FichaBibliograficaService fichaService)
+ listarFichas (TransactionManager txManager) :List
+ listarFichasEliminadas (TransactionManager txManager):List
+ buscarFichaPorId(int id, TransactionManager txManager): FichaBibliografica
+ actualizarFicha(FichaBibliografica ficha, TransactionManager txManager):void
+ eliminarFicha(int id, TransactionManager txManager):void
+ vincularFicha(int fichaId, int libroId, TransactionManager txManager):void
+ crearLibro(Libro libro, TransactionManager txManager): int
+ actualizarLibro(Libro libro, TransactionManager txManager):void
+ eliminarLibro(int id, TransactionManager txManager):void
+ buscarPorId(int id, TransactionManager txManager): Libro
+ listarLibros(TransactionManager txManager): List
+ listarEliminadosTransactionManager txManager): List
- validarLibro(Libro libro):void
«interface» GenericService<T>
+ insertar(entidad: T) : void
+ actualizar(entidad: T) : void
+ eliminar(id: int) : void
+ getById(id: int) : T
+ getAll() : List<T>

3. Transacciones y consistencia

- Se utiliza TransactionManager para controlar transacciones con autocommit = false, lo que permite agrupar múltiples operaciones en una única unidad de trabajo.
- Las operaciones que afectan a Libro y su FichaBibliográfica se ejecutan de forma atómica, compartiendo la misma conexión y garantizando que ambas entidades se creen o actualicen de forma sincronizada.
- En caso de error, se ejecuta rollback() automáticamente al cerrar el bloque try-with-resources, lo que evita inconsistencias en la base de datos.
- Se aplica el principio de consistencia transaccional: el sistema asegura que los datos permanezcan válidos y coherentes antes y después de cada operación, incluso ante fallos.
- El diseño respeta el modelo ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad), fundamental en sistemas que requieren integridad de datos:
    - Atomicidad: todas las operaciones dentro de una transacción se completan o ninguna lo hace.
    - Consistencia: se preservan las reglas de negocio y restricciones de integridad.
    - Aislamiento: cada transacción se ejecuta como si fuera la única en el sistema.
    - Durabilidad: los cambios confirmados persisten incluso ante caídas del sistema.
- El uso de TransactionManager desacopla la lógica de negocio del manejo de conexiones, permitiendo una arquitectura más limpia y testeable.
- La implementación favorece la trazabilidad y el control de errores, ya que cada operación crítica está encapsulada y validada antes de ser confirmada

Ejemplo:

```
public void crearLibroConFicha() {
    try (TransactionManager txManager = new TransactionManager(DatabaseConnection.getConnection())) {
        txManager.startTransaction();
```

4. Correctitud funcional

- CRUD completo implementado para ambas entidades, con operaciones de alta, baja lógica, modificación y consulta.
- Relación 1→1 efectiva entre Libro y FichaBibliográfica, , garantizada mediante claves foráneas y validaciones en la capa de servicio.
- Búsquedas por ID, por atributos y listados diferenciados (activos/eliminados), lo que permite segmentar la información según estado lógico.
- Validaciones previas en capa de servicio para evitar duplicados y entradas inválidas, aplicando reglas de negocio sobre longitud, unicidad y formato.

- Control de errores y excepciones con mensajes descriptivos, que permiten identificar rápidamente inconsistencias o fallos en la entrada de datos.
- Integridad referencial asegurada en operaciones compuestas, como la creación conjunta de Libro y FichaBibliográfica, evitando registros huérfanos.
- Separación de responsabilidades entre capas, lo que permite que cada operación se ejecute en el contexto adecuado (DAO para persistencia, Service para lógica).
- Pruebas manuales de flujo completo validadas desde la interfaz de consola, confirmando que las operaciones reflejan correctamente los cambios en la base de datos.
- Consistencia entre estados lógicos y físicos, ya que el sistema distingue entre eliminación lógica (eliminado = true) y eliminación física, preservando trazabilidad.
- Extensibilidad prevista para nuevas búsquedas o filtros, gracias al diseño modular de los métodos listarActivos, listarEliminados, buscarPorId, etc.

5. Repositorio y ejecución

- El proyecto se encuentra en GitHub con README detallado.
- Incluye scripts SQL para creación de tablas y carga de datos de prueba.
- Puede ejecutarse desde cero siguiendo los pasos del README.

6. Interfaz y experiencia

- **Menú por consola con opciones claras y numeradas**, lo que facilita la navegación y reduce errores de entrada.
- **Validaciones de entrada robustas**, incluyendo manejo de casos borde como IDs inexistentes, duplicados, campos vacíos o mal formateados.
- **Mensajes informativos y de error bien diferenciados**, con íconos y texto descriptivo que orientan al usuario en cada paso.
- **Flujo de interacción secuencial y guiado**, que permite al usuario completar operaciones sin ambigüedad ni sobrecarga cognitiva.
- **Retroalimentación inmediata tras cada acción**, confirmando el éxito o el motivo del fallo, lo que mejora la confianza en el sistema.
- **Diseño extensible para futuras interfaces gráficas o web**, gracias a la separación entre lógica de negocio y presentación.
- **Documentación técnica mediante JavaDocs en todos los métodos públicos**, lo que mejora la experiencia del desarrollador y facilita el mantenimiento:
    - Cada método incluye descripción, parámetros, excepciones y comportamiento esperado.
    - Se explicitan las reglas de negocio y los efectos transaccionales.
- **Consistencia en la nomenclatura y estructura de métodos**, lo que permite una curva de aprendizaje más rápida para nuevos integrantes del equipo.

- **Mensajes de consola alineados con el estado lógico del sistema**, como diferenciación entre entidades activas y eliminadas.

7. Trabajo en equipo y control de versiones

- Uso de Git con commits distribuidos y mensajes descriptivos.
- Integración colaborativa mediante ramas y revisiones.
- Cada integrante aportó paquetes funcionales y documentación.

8. Diagramas UML

- Diagrama de clases que refleja las relaciones entre entidades, servicios y DAO.
- Consistencia entre el modelo UML y la implementación final.

9. Decisiones destacadas

- Separación estricta de responsabilidades entre capas.
- Uso de TransactionManager para evitar errores de conexión y garantizar atomicidad.
- Validaciones en Service para mantener la lógica fuera del DAO.
- Eliminación lógica para preservar trazabilidad.

Video explicativo del proyecto:

https://youtu.be/_0qjnT92jyY