PROGRAMACIÓN II

Trabajo Final Integrador: Aplicación Java con relación 1-> 1 unidireccional + DAO + MySQL

Alumno: Mauro Gaspar

Comisión: 4

LINK Video: https://youtu.be/XchXUBEqk8E
LINK GitHub: https://github.com/27mau/UTN-Programacion-2/tree/main/TPI

INTRODUCCIÓN

Tema elegido: *Gestión de Productos y Códigos de Barras (relación 1→1 unidireccional).*

El objetivo general fue desarrollar una aplicación Java de escritorio que implemente una relación 1→1 unidireccional entre dos entidades, aplicando los principios del patrón DAO, la capa de servicios, y operaciones transaccionales mediante JDBC y MySQL**.**

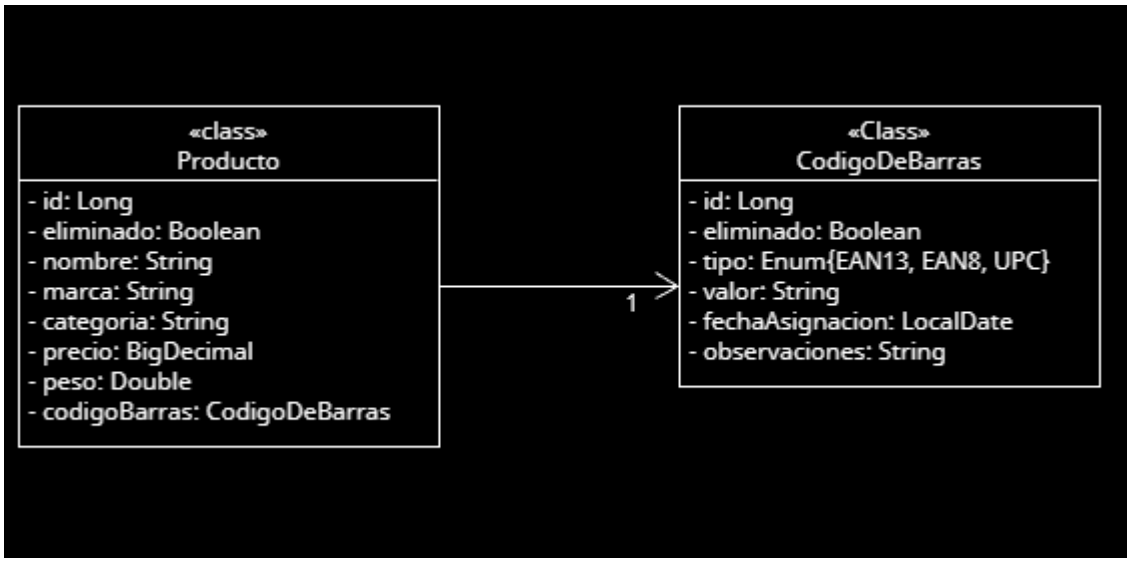
Se seleccionó el dominio **Producto → Código de Barras**, dado que representa un caso de uso común y realista dentro del ámbito comercial y logístico. Cada producto tiene un único código de barras, pero el código puede existir de forma independiente antes de ser asignado.
Esta elección permitió modelar con claridad la relación unidireccional, cumplir las reglas de unicidad y reflejar la integridad referencial entre ambas entidades.

Los objetivos específicos fueron:

- Implementar una base de datos MySQL con claves primarias, foráneas y restricciones adecuadas.
- Desarrollar una arquitectura por capas (config, entities, dao, service, main).
- Asegurar la persistencia mediante JDBC puro y PreparedStatement.
- Implementar operaciones CRUD completas con baja lógica (sin borrado físico).
- Incorporar validaciones de negocio y manejo de errores robusto.
- Controlar transacciones con **commit** y **rollback** desde la capa DAO.
- Proveer una interfaz de consola funcional para interactuar con el sistema.

DISEÑO

1. **Relación 1→1 unidireccional**
    - Se optó por la modalidad **clave foránea única**: la tabla producto contiene el campo codigo_barras_id con restricción UNIQUE.
    - Esto garantiza que cada producto solo pueda tener un código de barras, y que un mismo código no se asigne a más de un producto.
2. **Modelo de clases (UML)**
    - **Producto (A)** contiene una referencia a **CodigoDeBarras (B)**.
    - La relación se representa con una flecha unidireccional.
    - Ambos tienen el atributo eliminado para aplicar bajas lógicas.



3. **Arquitectura por capas**

El sistema se estructuró en **cinco paquetes** según las buenas prácticas del patrón DAO/Service:

Paquete	Responsabilidad principal
config	Conexión a la base de datos (DatabaseConnection).
entities	Clases de dominio con atributos, constructores y toString().
dao	Acceso a la base mediante JDBC, control de transacciones y consultas SQL parametrizadas.
service	Lógica de negocio, validaciones.
main	Interfaz de consola (AppMenuProducto, AppMenuCodigoBarras, Main).

4. **Regla de unicidad**
   La integridad se asegura desde SQL con:

```
codigo_barras_id BIGINT UNIQUE,
FOREIGN KEY (codigo_barras_id) REFERENCES codigo_barras(id)
```

   Además, desde el Service, se valida que un mismo código no se reasigne.

5. **Criterio de persistencia**

- Uso de PreparedStatement para todas las operaciones CRUD.
- Métodos DAO que aceptan Connection externa para transacciones compartidas.
- Mapeo manual de ResultSet a objetos (sin ORM).

PERSISTENCIA Y TRANSACCIONES

La persistencia del sistema se implementó siguiendo el patrón **DAO (Data Access Object)**, de manera que cada entidad del modelo (Producto y CodigoDeBarras) posee una clase dedicada para realizar las operaciones CRUD mediante JDBC puro. Esta capa es responsable de interactuar directamente con la base de datos MySQL, ejecutando sentencias SQL parametrizadas mediante PreparedStatement.

Control transaccional

Todo el manejo de transacciones (commit y rollback) debe realizarse en la capa de persistencia (DAO)**.** Esto asegura que la capa que interactúa directamente con la base sea también la responsable de garantizar la atomicidad, consistencia y durabilidad de las operaciones.

Por este motivo, el sistema implementa métodos transaccionales *compuestos* dentro del DAO, especialmente para las operaciones donde intervienen múltiples tablas. El caso más significativo es la creación de un **Producto** junto con su **Código de Barras**, que debe ejecutarse en una **única transacción** para mantener la integridad del modelo relacional 1→1.

Inicio, confirmación y reversión de transacciones

Para manejar transacciones manuales, se desactiva el auto–commit sobre la conexión:

```
conn.setAutoCommit(false);
```

Esto permite ejecutar varias operaciones SQL dentro de la misma transacción.
El flujo es:

1. **Iniciar transacción** (auto–commit = false).
2. Ejecutar las operaciones SQL dependientes:
    - Insertar el código de barras
    - Insertar el producto asociado
3. **Si todas las operaciones son exitosas → commit**
4. **Si ocurre cualquier error → rollback**, revirtiendo completamente los cambios.

Ejemplo simplificado (DAO):

```
public Producto crearConCodigo(Producto p, CodigoDeBarras c) throws
Exception {
    Connection conn = null;
    try {
        conn = DatabaseConnection.getConnection();
```

```
            conn.setAutoCommit(false);

            CodigoDeBarras creado = crearCodigo(c, conn);
            crearProducto(p, creado.getId(), conn);

            conn.commit();
            return p;

        } catch (Exception e) {
            if (conn != null) conn.rollback(); // revertir todo
            throw e;
        } finally {
            if (conn != null) conn.close();
        }
    }
```

Justificación del manejo transaccional en el DAO

El uso de transacciones en la capa DAO permite:

- Garantizar **atomicidad** en operaciones multi–tabla.
- Mantener la **integridad referencial** del modelo 1→1.
- Evitar que un código o producto quede “a medio crear”.

La capa **Service** queda encargada solo de:

- reglas de negocio,
- validaciones,
- coordinación de operaciones, pero **sin interactuar directamente con commits o rollbacks**.

Simulación de fallos y demostración del rollback

El TFI requiere demostrar un caso práctico donde una operación compuesta falle y el sistema ejecute un rollback completo. Esto se implementó mediante: lanzamiento de excepciones controladas, o generación de errores SQL intencionales (ej. valor duplicado del código), lo que permite observar que, el primer INSERT se ejecuta y la segunda falla, entonces el DAO activa rollback y ningún registro queda persistido.

Esta demostración permite verificar que el manejo transaccional funciona de manera correcta y consistente.

VALIDACIONES Y REGLAS DE NEGOCIO

Las validaciones y reglas de negocio se aplican en la **capa de servicio**, antes de interactuar con la base de datos.

El objetivo es evitar errores lógicos o inconsistencias antes de ejecutar las operaciones SQL.

Ejemplo de validación en capa de servicio

```
if (producto.getNombre() == null || producto.getNombre().isBlank()) {
    throw new IllegalArgumentException("El nombre del producto es
obligatorio.");
}
if (producto.getPrecio() == null || producto.getPrecio().doubleValue()
< 0) {
    throw new IllegalArgumentException("El precio no puede ser
negativo.");
}
```

Estas validaciones previas permiten **interceptar errores antes de ejecutar SQL**, evitando fallos de integridad y mostrando mensajes claros al usuario.

Manejo de errores y retroalimentación

- Las excepciones capturadas se transforman en **mensajes amigables** mostrados en consola desde AppMenuProducto o AppMenuCodigoBarras.
- Si ocurre una falla grave, la transacción se revierte (rollback) y el sistema continúa funcionando.
- Se diferencian errores de **validación de negocio** (ej. nombre vacío) y **errores técnicos** (SQL o conexión).

Beneficio general

Gracias a las validaciones y a las transacciones:

- Se mantiene la **integridad de la base de datos**.
- Se cumplen las **restricciones del modelo relacional**.
- El usuario final recibe mensajes comprensibles.
- El sistema es **robusto y confiable** ante errores.

PRUEBAS Y RESULTADOS

Con el objetivo de validar el correcto funcionamiento del sistema y demostrar el cumplimiento de los requisitos del Trabajo Final Integrador, se realizaron diferentes pruebas sobre cada una de las funcionalidades del sistema: operaciones CRUD, validaciones, consultas y manejo transaccional (commit/rollback).
Las pruebas se ejecutaron desde la interfaz de consola (Main → AppMenuProducto / AppMenuCodigoBarras) utilizando la base de datos MySQL definida en el esquema del proyecto.

1. Pruebas de creación (CREATE)

1.1 Crear código de barra

Se ingresó un código válido con los siguientes datos:

- Tipo: EAN13
- Valor: 1234567890123
- Fecha: 2025-11-17
- Observaciones: Prueba

Resultado:
El sistema guardó el registro correctamente y devolvió un ID autogenerado.
El valor no duplicado fue validado correctamente por la base de datos.

```
--- Crear Código de Barras ---
Tipo (EAN8, EAN13, UPC): EAN13
Valor: 1234567890123
Fecha asignación (YYYY-MM-DD o vacío): 2025-11-17
Observaciones (opcional): Prueba
Código creado con éxito. ID generado: 393217
```

```
Ingrese ID del código: 393217

Código encontrado:
CodigoDeBarras[id=393217, tipo=EAN13, valor=1234567890123, fecha=2025-11-17]
```

1.2 Crear producto sin código

Datos ingresados:

- Nombre: “Yerba Mate Playadito”
- Marca: “Playadito”
- Categoría: “Alimentos”
- Precio: 2800
- Peso: 0.500

Resultado:
Producto creado correctamente. Se comprobó en MySQL que:

```
eliminado = false
codigo_barras_id = null
```

Consola:

```
Ingrese ID del producto: 393218

Producto encontrado:
Producto[id=393218, nombre=Yerba Mate Playadito, marca=Playadito, cat=Alimentos, precio=2800.00, codigo=N/A]
```

1.3 Crear producto con código (prueba transaccional)

Se ingresa en Producto:

- Nombre: “Aceite Natura”
- Marca: “Natura”
- Precio: 1300

Código:

- Tipo: EAN13
- Valor: 9312345612345

Resultado:

- Código insertado correctamente
- Producto insertado correctamente
- Asociados dentro de 1 única transacción
- Commit exitoso
- Ambas tablas actualizadas de forma coherente

```
Ingrese ID del producto: 393219

Producto encontrado:
Producto[id=393219, nombre=Aceite Natura, marca=Natura, cat=, precio=4300.00, codigo=9312345612345]
```

2. Pruebas de lectura (READ)

2.1 Buscar por ID

Al consultar un ID existente, el sistema imprimió el objeto con todos sus atributos, incluyendo: datos del producto, código asociado (si existía).

Al consultar un ID inexistente →

Resultado esperado: mensaje claro:

```
Error: No existe un producto con ese ID.
```

2.2 Listar todos

Se ejecutó “Listar Productos” y “Listar Códigos” luego de varias operaciones.

Resultado:
La salida mostró todos los registros no eliminados (eliminado = false), permitiendo verificar el estado actualizado del sistema en cada prueba.

3. Pruebas de actualización (UPDATE)

Se modificó un producto existente, cambiando:

- precio
- peso
- categoría

Resultado:
El sistema validó correctamente los valores:

- No permitió peso negativo
- No permitió campos vacíos obligatorios

En caso de error, mostró:

```
Error: El precio no puede ser negativo.
```

Cuando los datos fueron correctos:

```
Producto actualizado.
```

Mis pruebas mostraron que el producto se actualizó en MySQL con:

```
UPDATE producto SET eliminado WHERE id = 393219;
```

Consola:

```
Producto actual:
Producto[id=393219, nombre=Aceite Natura, marca=Natura, cat=, precio=4300.00, codigo=9312345612345]
```

```
Producto encontrado:
Producto[id=393219, nombre=Aceite Natura, marca=Natura, cat=Alimentos, precio=5000.00, codigo=9312345612345]
```

4. Pruebas de eliminación (DELETE – baja lógica)

Al eliminar un producto o código, el sistema no borró físicamente el registro.

Resultado en base:

```
eliminado = true
```

El registro dejó de aparecer en:

- Listar productos
- Listar códigos

Consola:

```
Ingrese ID del producto: 38

Producto encontrado:
Producto[id=38, nombre=Ítem 64 - Decoración 227, marca=Marca_M1, cat=Hogar, precio=222.19, codigo=0010000000064]
```

```
Ingrese ID del producto: 38
Error: No existe un producto con ese ID.
```

5. Prueba clave del TFI: Manejo transaccional y Rollback

Esta es la prueba fundamental del Trabajo Final, diseñada para demostrar la integridad transaccional del sistema: se debe iniciar una transacción, forzar un error y ejecutar un rollback correcto para asegurar la no inserción de datos inconsistentes.

Caso probado: Error forzado al crear un producto con código

El método transaccional del DAO contiene un código opcional:

```
if ("ERROR".equalsIgnoreCase(producto.getNombre())) {
    throw new RuntimeException("Error simulado para demostrar
rollback.");
}
```

Procedimiento de prueba

1. En el menú “Crear producto con código”, se ingresó:
    - Nombre: ERROR
    - Precio: valor válido
    - Tipo de código válido
2. Al ejecutarse el método transaccional:
    - Insertó primero el código
    - Lanzó error simulado
    - El DAO capturó la excepción
    - Ejecutó:

```
conn.rollback();
```

Resultado esperado:

- **No se insertó ningún producto**
- **No quedó insertado el código de barras**
- La base quedó exactamente igual que antes de la operación
- La consola mostró:

```
Error: Error simulado para demostrar rollback.
```

Verificación en MySQL

Se ejecutó:

```
SELECT * FROM codigo_barras ORDER BY id DESC;
SELECT * FROM producto ORDER BY id DESC;
```

Consola:

```
--- Crear Producto con Código (Transacción) ---
Nombre: ERROR
Marca (opcional): Ayudin
Categoria (opcional): Hogar
Precio: 2000
Peso (opcional): 1

--- Código de Barras ---
Tipo (EAN8, EAN13, UPC): EAN13
Valor: 1234567894321
Fecha asignación (YYYY-MM-DD o vacío): 2025-11-15
Observaciones (opcional):
Error: Error simulado para demostrar rollback.
```

El código insertado inicialmente nunca apareció → rollback exitoso.

CONCLUSIONES

El desarrollo del sistema de **Gestión de Productos y Códigos de Barras** permitió aplicar de manera práctica los contenidos teóricos de la asignatura Programación II**,** integrando conocimientos de POO, JDBC, validaciones y arquitectura multicapa**.**

A lo largo del proyecto, se lograron aprendizajes clave como la comprensión e implementación del patrón DAO en Java, la aplicación de transacciones **(commit/rollback)** para garantizar la consistencia de datos, y el uso de validaciones de

negocio y bajas lógicas para un control seguro. El sistema fue diseñado de forma modular y extensible, separando las capas de presentación, servicio y persistencia.

En síntesis, el sistema cumple con todos los requisitos funcionales y técnicos, demostrando el dominio de los principios de la programación orientada a objetos y la persistencia en Java. Es importante destacar que el uso de herramientas de inteligencia artificial y entornos colaborativos mejoró la productividad, la claridad del código y la calidad del informe, ejemplificando cómo este enfoque híbrido complementa el proceso educativo al ser utilizado con criterio y responsabilidad.

FUENTES, HERRAMIENTAS Y CONSIDERACIONES ÉTICAS

Herramientas utilizadas

Durante el desarrollo del proyecto se emplearon diversas herramientas de software libre y entornos académicos, elegidos por su estabilidad y compatibilidad con el ecosistema Java:

Herramienta	Descripción / Uso principal
Java SE 21	Lenguaje de programación principal y versión recomendada para compatibilidad con JDBC.
MySQL 8.0	Sistema de gestión de base de datos relacional (RDBMS) utilizado para la persistencia.
MySQL Workbench	Diseño, ejecución de scripts SQL y verificación de integridad referencial.
IntelliJ IDEA / NetBeans	Entornos de desarrollo integrados (IDE) usados para codificación, depuración y empaquetado.
Git & GitHub	Control de versiones, colaboración y seguimiento de cambios del proyecto.
ChatGPT (OpenAI)	Asistencia en redacción, documentación y generación de ejemplos didácticos, bajo supervisión y revisión humana.

Fuentes de apoyo teórico:

1. Materiales de cátedra: *Programación II – (pdf, videos, **Microteaching**).*