INSTITUTO DE EDUCACIÓN SECUNDARIA

ISIDRA DE GUZMÁN

PRÁCTICA DE EVALUACIÓN CONTINUA (PEC)

Acceso a Datos

Alumno/a: Diego Luengo
Grupo: 2º DAM
Módulo: Acceso a datos
Fecha: 02/02/2026
Curso académico: 2025/2026

1. Descripción del proyecto y objetivos

Contexto y Alcance del Sistema

El proyecto consiste en el desarrollo de una aplicación de consola en Java para la gestión integral de una **Tienda de Productos**. El sistema permite administrar el ciclo de vida de los productos (inventario), la cartera de clientes y el registro de ventas.

El alcance del sistema abarca:

- **Gestión de Inventario**: Control de stock, precios y descripciones de productos.
- **Gestión de Clientes**: Registro de datos personales y saldo disponible (monedero virtual).
- **Gestión de Ventas**: Proceso transaccional que vincula clientes con productos, actualizando stock y saldos en tiempo real.

Reglas de Negocio y Operativa

El sistema implementa las siguientes reglas de negocio para asegurar la integridad de los datos:

1. **Operativa Básica (CRUD)**:
    a. **Altas**: Registro de nuevos productos y clientes.
    b. **Bajas**: Eliminación de registros (con restricciones de integridad referencial).
    c. **Modificaciones**: Actualización de atributos (ej. precio, stock, teléfono).
    d. **Consultas**: Búsquedas por ID y listados generales.
2. **Restricciones de Negocio**:
    a. **Stock No Negativo**: No se pueden realizar ventas si no hay suficiente stock del producto. El stock nunca puede ser menor a 0.
    b. **Saldo Suficiente**: Un cliente no puede realizar una compra si su saldo (dinero) es inferior al total de la venta.
    c. **Integridad de Precios**: Los precios y cantidades deben ser valores positivos.
    d. **Unicidad**: Cada producto y cliente se identifica unívocamente por su ID.

Objetivos Técnicos de la PEC

El objetivo principal de esta práctica es la **refactorización de la capa de acceso a datos**, migrando de un enfoque basado en JDBC (SQL nativo) a un modelo ORM (Object-Relational Mapping) utilizando la especificación **JPA** (Java Persistence API) con **Hibernate** como proveedor.

Los objetivos técnicos específicos incluyen:

1. **Eliminación de SQL Nativo**: Sustituir el uso de `PreparedStatement` y `ResultSet` por la gestión de entidades (`EntityManager`).
2. **Mapeo Objeto-Relacional**: Definir las entidades `Producto`, `Cliente` y `Venta` con anotaciones JPA (`@Entity`, `@Table`, `@ManyToOne`, etc.).

3. **Gestión Transaccional ACID**: Implementar transacciones atómicas para el proceso de venta, asegurando que el descuento de stock, el cobro al cliente y el registro de la venta ocurran como una unidad indivisible (commit) o se deshagan completamente en caso de error (rollback).
4. **Uso de JPQL**: Realizar consultas a la base de datos utilizando el lenguaje de consulta orientado a objetos de Java (JPQL), incluyendo filtros, agregaciones y relaciones.
5. **Validación de Esquema**: Configurar la persistencia para validar estrictamente la coincidencia entre las clases Java y el esquema de base de datos (`hbm2ddl.auto = validate`).

2. Requisitos y entorno

Entorno de Desarrollo

Para la construcción y ejecución del proyecto se han utilizado las siguientes herramientas y versiones:

- **Lenguaje**: Java SE 21 (definido en `maven.compiler.source` y `target`).

- **Gestión de Proyectos**: Maven.
- **Base de Datos**: MySQL Community Server 8.0+.
- **Entorno Integrado de Desarrollo (IDE)**: Visual Studio Code.

Dependencias del Proyecto (pom.xml)

El proyecto gestiona sus librerías externas mediante Maven. Las dependencias principales configuradas son:

```
<dependencies>
    <!-- Conector JDBC para MySQL -->
    <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <version>9.1.0</version>
    </dependency>

    <!-- Hibernate Core (Implementación JPA) -->
    <dependency>
        <groupId>org.hibernate.orm</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>6.4.4.Final</version>
    </dependency>

    <!-- Jakarta Persistence API (Especificación estándar) -->
    <dependency>
        <groupId>jakarta.persistence</groupId>
        <artifactId>jakarta.persistence-api</artifactId>
        <version>3.1.0</version>
    </dependency>
</dependencies>
```

Configuración de Base de Datos y Credenciales

La configuración de la conexión a la base de datos se encuentra centralizada en el archivo estándar de JPA: `src/main/resources/META-INF/persistence.xml`.

```
<persistence-unit name="inventarioPU" transaction-type="RESOURCE_LOCAL">
    <properties>
        <!-- Credenciales y URL de Conexión -->
        <property name="jakarta.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/tienda"/>
        <property name="jakarta.persistence.jdbc.user" value="root"/>
        <property name="jakarta.persistence.jdbc.password" value="****"/> <!--
Oculta por seguridad -->
        <property name="jakarta.persistence.jdbc.driver"
```

```
value="com.mysql.cj.jdbc.Driver"/>

        <!-- Configuración de Hibernate -->
        <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQLDialect"/>
        <property name="hibernate.hbm2ddl.auto" value="validate"/> <!--
Validación estricta -->
    </properties>
</persistence-unit>
```

---

3. Estructura del proyecto (paquetes y clases)

****EL DIAGRAMA ESTA UBICADO EN EL PDF DE ANEXOS****

El proyecto ha sido estructurado siguiendo el patrón de arquitectura **MVC (Modelo-Vista-Controlador)** adaptado a una aplicación de consola, añadiendo capas de **Servicio** y **Repositorio** para desacoplar la lógica de negocio y el acceso a datos.

Diagrama de Árbol del Proyecto

```
src/main/java/com/inventario
├── Main.java                 (Clase Principal)
├── controller                (Capa de Controladores)
│   ├── MainController.java
│   ├── ClienteController.java
│   ├── ProductoController.java
│   └── VentaController.java
├── model                     (Capa de Modelo / Entidades JPA)
│   ├── Cliente.java
│   ├── Producto.java
│   ├── Venta.java
│   └── DetalleVenta.java
├── repository                (Capa de Acceso a Datos / DAO)
│   ├── ClienteRepository.java
│   ├── ProductoRepository.java
│   └── VentaRepository.java
├── service                   (Capa de Servicios / Negocio)
│   ├── ClienteService.java
│   ├── ProductoService.java
│   └── VentaService.java
├── view                      (Capa de Presentación / Consola)
│   ├── MainView.java
│   ├── ClienteView.java
│   ├── ProductoView.java
│   └── VentaView.java
├── util                      (Utilidades Transversales)
│   ├── JPAUtil.java
│   └── Util.java
└── excepciones               (Manejo de Errores)
    └── DatoInvalidoException.java
```

Descripción de Paquetes y Clases

`com.inventario.model` (Entidades)

Contiene las clases POJO (Plain Old Java Objects) mapeadas a las tablas de la base de datos mediante anotaciones JPA.

- **`Cliente`**: Representa a los clientes (ID, nombre, saldo, contacto).
- **`Producto`**: Representa el inventario (ID, precio, stock, descripción).
- **`Venta`**: Cabecera de la factura. Contiene la relación con `Cliente` y la lista de `DetalleVenta`.
- **`DetalleVenta`**: Línea de pedido. Relaciona `Venta` con `Producto` y almacena cantidad y precio histórico.

`com.inventario.repository` (Acceso a Datos)

Clases encargadas de interactuar con el `EntityManager`. Aquí reside el uso de **JPQL** y transacciones.

- **`ClienteRepository` / `ProductoRepository`**: Realizan operaciones CRUD básicas y búsquedas filtradas.
- **`VentaRepository`**: Gestiona la lógica transaccional compleja (`realizarVentaCompleta`), asegurando la integridad ACID de la venta.

`com.inventario.service` (Lógica de Negocio)

Actúan como intermediarios entre el controlador y el repositorio.

- **`ClienteService` / `ProductoService` / `VentaService`**: Exponen métodos de alto nivel (ej. `realizarVenta`, `actualizarStock`) ocultando la complejidad del acceso a datos.

`com.inventario.controller` (Controladores)

Orquestan el flujo de la aplicación recibiendo input de la vista y llamando a los servicios .

- **`MainController`**: Menú principal y enrutador.
- **`ClienteController` / `ProductoController` / `VentaController`**: Gestionan los submenús específicos de cada entidad.

`com.inventario.view` (Vistas)

Encapsulan la interacción con la consola (System.in / System.out).

- Se encargan de pedir datos al usuario y mostrar resultados, manteniendo el código limpio de `Scanner` en otras capas.

`com.inventario.util`

- **`JPAUtil`**: Singleton que gestiona el `EntityManagerFactory`. Se inicializa al arrancar y se cierra al salir.
- **`Util`**: Métodos estáticos auxiliares para validación de entradas de usuario.

---

4. Diseño lógico de la base de datos

El diseño de la base de datos sigue un modelo relacional normalizado para evitar redundancias y garantizar la integridad de los datos.

Modelo Relacional

1. Tabla `Cliente`

- **Clave Primaria (PK)**: `id_cliente` (INT, Auto-increment).
- **Atributos**: `nombre` (VARCHAR), `email` (VARCHAR), `telefono` (VARCHAR), `dinero` (DOUBLE).
- **Relaciones**: Relación de 1 a N con `Venta` (Un cliente realiza muchas ventas).

2. Tabla `Producto`

- **Clave Primaria (PK)**: `id_producto` (INT, Auto-increment).
- **Atributos**: `nombre` (VARCHAR), `descripcion` (VARCHAR), `precio` (DOUBLE), `stock` (INT).
- **Restricciones**: Check `stock >= 0`.
- **Relaciones**: Relación de 1 a N con `DetalleVenta`.

3. Tabla `Venta` (Cabecera)

- **Clave Primaria (PK)**: `id_venta` (INT, Auto-increment).
- **Clave Foránea (FK)**: `id_cliente` (Referencia a `Cliente.id_cliente`).
- **Comportamiento**: `ON DELETE CASCADE` (Si se borra el cliente, se borra su historial de ventas).

4. Tabla `DetalleVenta` (Líneas)

- **Clave Primaria (PK)**: `id_detalle` (INT, Auto-increment).
- **Claves Foráneas (FK)**:
  - `id_venta` (Ref. a `Venta.id_venta`).
  - `id_producto` (Ref. a `Producto.id_producto`).
- **Atributos**: `cantidad` (INT), `precio_unitario` (DOUBLE).
- **Función**: Resuelve la relación de muchos a muchos (M:N) entre Ventas y Productos.

Normalización

El esquema cumple con las formas normales principales:

- **1FN**: Todos los atributos contienen valores atómicos indivisibles.
- **2FN**: No existen dependencias parciales; la tabla intermedia `DetalleVenta` tiene su propia clave primaria surrogada (`id_detalle`), y sus atributos dependen de la relación completa.
- **3FN**: No hay dependencias transitivas.

5. Implementación

5.1 Configuración ORM (persistence.xml)

Se utiliza el archivo `persistence.xml` estándar para configurar la unidad de persistencia `inventarioPU`.

- **Proveedor**: Hibernate ORM.
- **Conexión**: JDBC Driver para MySQL 8+.
- **Propiedades**: `hibernate.show_sql` (desactivado para limpieza) y `hibernate.hbm2ddl.auto=validate` para garantizar que la aplicación solo arranque si la base de datos tiene la estructura correcta.

5.2 Gestión del contexto de persistencia

Para manejar el ciclo de vida de la persistencia se ha implementado la clase utilitaria `JPAUtil` (Singleton Pattern).

- **`EntityManagerFactory`**: Es un objeto "pesado" y costoso de crear. Se inicializa **una sola vez** (static) al inicio de la aplicación y se comparte globalmente. (Justificación: Thread-safe, alto coste de instanciación).
- **`EntityManager`**: Es el gestor de entidades "ligero". Se crea **uno nuevo para cada operación o transacción** y se cierra inmediatamente después (`try-with-resources` o bloque `finally`). (Justificación: No es thread-safe, representa una sesión de trabajo unitaria).

5.3 Entidades y mapeo

Las clases del modelo utilizan anotaciones JPA estándar:

- `@Entity`: Marca la clase como persistente.
- `@Table(name = "...")`: Asigna el nombre exacto de la tabla.
- `@Id`, `@GeneratedValue`: Define la clave primaria y su estrategia de autoincremento.
- `@ManyToOne` / `@OneToMany`: Definen las relaciones.
    - Ejemplo: `Venta` tiene `@OneToMany(mappedBy="venta", cascade=CascadeType.ALL)` hacia `DetalleVenta`, lo que permite que al guardar una venta se guarden automáticamente sus detalles.

5.4 Operaciones CRUD y Gestión de Transacciones Básica

Todas las operaciones de modificación en JPA (Create, Update, Delete) requieren una transacción activa. El patrón implementado en los repositorios es:

1. Obtener `EntityManager`.
2. Obtener `EntityTransaction` (`em.getTransaction()`).
3. Iniciar transacción (`tx.begin()`).
4. Ejecutar operación (`persist`, `merge`, `remove`).
5. Confirmar cambios (`tx.commit()`).
6. Capturar excepciones: Si hay error, hacer `tx.rollback()`.
7. Bloque `finally`: Cerrar siempre el `EntityManager`.

5.5 Consultas JPQL (Java Persistence Query Language)

Se utilizan consultas orientadas a objetos para operaciones complejas.

- **Listado con relación**: `SELECT v FROM Venta v JOIN FETCH v.cliente ORDER BY v.id DESC` (Optimización para evitar N+1 selects).
- **Agregación**: `SELECT SUM(d.cantidad * d.precioUnitario) FROM DetalleVenta d WHERE d.venta.id = :idVenta`.
- **Filtrado dinámico**: `SELECT p FROM Producto p WHERE p.campo = :valor`.

5.6 Gestión de transacciones (Justificación y Caso Complejo)

Para el caso de uso **"Realizar Venta"**, se requiere una consistencia fuerte (ACID). No basta con guardar la venta; hay que descontar stock y restar dinero al cliente simultáneamente.

En `VentaRepository.realizarVentaCompleta()`:

1. Se abre una única transacción.
2. Se recuperan las entidades `Cliente` y `Producto` (que quedan gestionadas/managed).
3. Se verifican reglas de negocio (tiene stock? tiene saldo?). Si falla algo, se lanza excepción y **rollback a todo**.
4. Se modifican las entidades (`producto.setStock(...)`, `cliente.setDinero(...)`). Al estar gestionadas, JPA detecta los cambios automáticamente (Dirty Checking).
5. Se persiste la nueva `Venta`.
6. Se hace `commit` final.

6. Dificultades encontradas

He encontrado problemas sobre todo en las “etiquetas” de los atributos de las clases, pero he podido encontrar una solucion gracias a los documentos del aula virutal y preguntando a la IA como funciona

7. Bibliografía

Maven Repository: jakarta.persistence

Introduction to Jakarta Persistence 3.2 | Baeldung

Fuentes de IA

Aula Virtual de acceso a datos