Buenos días Don Daniel. ¡Vargas Cipollo!

Tema 3.1: Acceso a los Datos (Bases de datos relacionales y O/R Mapping)

1. Bases de datos relacionales y SQL

- **SQL (Structured Query Language)**
  - Lenguaje estándar para manipular y consultar datos en BD relacionales.
  - DML: *SELECT*, *INSERT*, *UPDATE*, *DELETE*.
  - DDL: *CREATE TABLE*, *ALTER TABLE*, *DROP TABLE*.
- **Inconvenientes de incrustar SQL en código**
  - Mezcla lógica de negocio y acceso a datos → difícil de mantener.
  - Riesgo de inyección SQL si no se usan mecanismos seguros.
  - Cambiar estructura BD fuerza cambios en todo el código.
- **Patrones de acceso a datos**
  1. **Active Record**
     - Cada objeto representa directamente una fila de tabla.
     - Encapsula campos de la fila y lógica de negocio asociada.
     - Métodos para *save()*, *delete()*, *find()* dentro de la propia clase.
     - Útil si la lógica de dominio es sencilla.
  2. **Gateways**
     - Clases auxiliares que ofrecen operaciones CRUD (Create, Retrieve, Update, Delete).
     - Tipos:
       - *Row Data Gateway*: un objeto por cada fila de tabla (atributo por columna). Sin lógica de negocio, sólo mapea datos.
       - *Table Data Gateway*: un objeto por tabla, encapsula todo el SQL para acceder a la tabla entera. Interfaz simple (métodos para obtener, insertar, borrar, actualizar).
- **Seguridad: Inyección de código SQL**
  - Construir cadenas SQL concatenando parámetros de entrada es peligroso.
  - Ej.:

```
String sql = "SELECT * FROM usuarios WHERE username='" + user +
             "' AND password='" + pass + "'";
```

→ Entrada maliciosa: `user = admin' OR '1'='1`.

- **Solución**: usar **PreparedStatement** (consultas parametrizadas).

```
PreparedStatement ps = conn.prepareStatement(
   "SELECT * FROM usuarios WHERE username = ? AND password = ?");
ps.setString(1, user);
ps.setString(2, pass);
ResultSet rs = ps.executeQuery();
```

- **Conexión mediante JDBC (Java DataBase Connectivity)**

1. **Cargar driver JDBC**

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

2. **Obtener conexión**

```
Connection conn = DriverManager.getConnection(
  "jdbc:oracle:thin:@localhost:1521:SID", "usuario", "password");
```

3. **Ejecutar consultas**

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM clientes");
while (rs.next()) {
  String nombre = rs.getString("name");
  Date fecha = rs.getDate("birthdate");
  // …
}
```

4. **Tipos SQL–Java (mapeo)**

   - CHAR/VARCHAR ↔ String (`getString()`)
   - DECIMAL/NUMERIC ↔ BigDecimal (`getBigDecimal()`)
   - FLOAT/DOUBLE ↔ double (`getDouble()`)
   - INTEGER ↔ int (`getInt()`)
   - DATE ↔ java.sql.Date (`getDate()`)
   - BINARY ↔ byte[] (`getBytes()`)
   - BLOB ↔ InputStream/Blob (`getBinaryStream()`, `getBlob()`)

---

2. Object/Relational Mapping (O/R Mapping)

- **Objetivo**: separar modelo de objetos de la representación en BD.

- Objetos no conocen estructura de tablas ni SQL.
- Permite cambiar BD sin tocar modelo de dominio y viceversa.
- Facilita mantenimiento, pruebas y evolución.

- **Patrones estructurales de O/R Mapping**

  1. **Identity Field**

     - Cada objeto lleva un atributo *ID* que coincide con la clave primaria de la tabla.
     - Clave simple: usarla directamente.
     - Clave compuesta: más complejo, hay que decidir cómo mapear varios campos.

  2. **Foreign Key Mapping**

     - Para relaciones 1→N o N→1.
     - En BD: clave externa en la tabla "muchos".
     - En objetos: atributo que referencia al objeto "uno".
     - Ejemplo: `Order.customerId` (BD) ↔ `Order.getCustomer()` (objeto).

  3. **Association Table Mapping**

     - Para relaciones N↔N.
     - Tabla intermedia con dos claves externas.
     - Proceso de consulta:
       1. Buscar filas de la tabla de asociación con clave externa A.
       2. Para cada fila, usar la otra clave para cargar el objeto B.
     - En objetos: listas o colecciones de objetos relacionados.

  4. **Herencia (Inheritance Mapping)**

     - **Single Table Inheritance**
       - Una sola tabla para toda la jerarquía.
       - Columna "discriminator" para indicar subclase.
       - Ventaja: consultas sencillas; desventaja: muchas columnas nulas.
     - **Concrete Table Inheritance**
       - Una tabla por cada clase concreta.
       - Cada tabla contiene sólo sus campos (no hereda columnas de padre).
       - Ventaja: no hay nulos; desventaja: duplicación de claves y datos.
     - **Class Table Inheritance**
       - Una tabla por cada clase (padre y subclases).
       - Tablas ligadas por clave primaria compartida.
       - Ventaja: normalización; desventaja: joins requeridos para cargar objetos.

- **Patrones de comportamiento de O/R Mapping**

1. **Identity Map**
    - Mantener un mapa (cache) por cada tabla: clave → objeto.
    - Cuando se pide un objeto, primero se busca en el mapa; si existe, se reutiliza.
    - Evita duplicar instancias de la misma fila.
2. **Lazy Load**
    - Retrasar la carga de datos relacionados hasta que se acceda a ellos.
    - Formas de implementación:
        - **Inicialización perezosa**: campos null hasta primer acceso.
        - **Proxy**: objeto ligero que carga objeto real al primer uso.
        - **Value Holder**: método `getValue()` que obtiene datos bajo demanda.
        - **Ghost**: objeto "vacío" que se rellena completamente cuando se usa.

- **Patrones con metadatos**
    1. **Metadata Mapping**
        - Información de mapeo (clase ↔ tabla, atributo ↔ columna) almacenada en metadatos (XML, anotaciones).
        - Minimiza esfuerzo de codificación manual de correspondencias.
    2. **Query Objects**
        - Clases que representan consultas SQL como estructuras de objetos.
        - Permiten construir consultas usando objetos y atributos, y luego traducir a SQL.
        - Ejemplo:

```
QueryBuilder qb = new QueryBuilder();
qb.from(Author.class).where("lastName").equalsIgnoreCase(lastName)
;
List<Author> lista = qb.list();
```

    3. **Repository**
        - Interfaz que aísla el dominio de la capa de persistencia.
        - Clientes crean objetos "criterio" (detalles de búsqueda) y piden al repositorio objetos que cumplan criterios.
        - Repositorio ejecuta consultas, devuelve conjuntos de objetos.
        - Mejora legibilidad y claridad de código (no hay SQL explícito en dominio).

---

3. Tecnologías para acceso a datos

3.1 Acceso SQL directo

- **ODBC (Open DataBase Connectivity)**: API de Microsoft para BD relacionales.
- **JDBC (Java DataBase Connectivity)**: API Java (ejemplos en sección 1).
- **ADO.NET**: API .NET para BD.
- **DB-API (Python)**: estándar de Python para BD (ej. sqlite3, psycopg2).

3.2 Uso de JDBC (resumen rápido)

1. **Carga de driver**: `Class.forName("driverClassName")`.
2. **Conexión**: `DriverManager.getConnection(url, user, pass)`.
3. **Statement / PreparedStatement** para ejecutar consultas.
4. **ResultSet** → iterar filas; métodos `getXxx(columnName)`.
5. **Cerrar** `ResultSet`, `Statement`, `Connection` siempre en bloque finally.

3.3 Mecanismos de seguridad

- **PreparedStatement**: evita concatenar variables en SQL.
- **Transacciones**: `conn.setAutoCommit(false)`, `conn.commit()`, `conn.rollback()`.
- **Manejo de excepciones**: capturar `SQLException` y revertir transacción si falla.

---

4. Tecnologías de O/R Mapping

4.1 JPA (Java Persistence API)

- Especificación estándar Java para persistencia de objetos.
- **Anotaciones principales**:
  - `@Entity`: marca clase como entidad persistente.
  - `@Id`: atributo que representa clave primaria.
  - `@ManyToOne`, `@OneToMany`, `@ManyToMany`: relaciones entre entidades.
  - `@Table(name="...")`, `@Column(name="...")`: personalizar nombres de tabla/columna.
- **JPQL (Java Persistence Query Language)**
  - Lenguaje orientado a objetos para consultas (similar a SQL, pero sobre entidades).
  - Ejemplo:

```
String q = "SELECT a FROM Author a WHERE LOWER(a.lastName) =
LOWER(:ln)";
Query query = em.createQuery(q);
query.setParameter("ln", lastName);
List<Author> autores = query.getResultList();
```

4.2 Hibernate / NHibernate (ORM populares)

- Implementación de JPA en Java: **Hibernate**

    - Configuración mediante anotaciones o archivos XML (`hibernate.cfg.xml`, `*.hbm.xml`).
    - Mapeo de clases a tablas (identidad, relaciones, herencia).
    - Soporta **Lazy Loading**, **Caching** (1er y 2º nivel), transacciones, etc.

- Ejemplo de mapeo XML en Hibernate:

```
<hibernate-mapping>
  <class name="Employee" table="EMPLOYEE">
    <id name="id" column="id" type="int">
      <generator class="native"/>
    </id>
    <property name="firstName" column="first_name" type="string"/>
    <property name="lastName" column="last_name" type="string"/>
    <property name="level" column="level" type="int"/>
  </class>
</hibernate-mapping>
```

- **Operación READ con Hibernate**

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
List<Employee> empleados = session.createQuery("FROM Employee").list();
tx.commit();
session.close();
```

4.3 Otros frameworks ORM

- **.NET**: Entity Framework Core, NHibernate.
- **Python**: SQLAlchemy (ORM completo), Django ORM (framework MVC).
- **PHP**: Eloquent (Laravel), Doctrine ORM.

Posibles ejercicios de examen

1. **Definiciones clave**
    - Explica qué es un **Active Record** y cuándo conviene usarlo.
    - Describe las diferencias entre **Row Data Gateway** y **Table Data Gateway**.
2. **Inyección SQL**
    - ¿Por qué es peligrosa la construcción de sentencias SQL mediante concatenación de cadenas?
    - Muestra cómo mitigar la inyección usando **PreparedStatement** en Java.
3. **Patrones de O/R Mapping**
    - Para cada uno de los patrones: *Foreign Key Mapping*, *Association Table Mapping*, *Identity Map*, *Lazy Load*, da una breve descripción y un ejemplo práctico.

- Dibuja cómo mapear una relación N a M entre `Student` y `Course` usando **Association Table Mapping**.

4. **Herencia en ORM**
   - Explica las tres estrategias de mapeo de herencia: **Single Table**, **Concrete Table**, **Class Table**. Indica ventajas e inconvenientes de cada una.
   - Propón una jerarquía de clases (por ej., `Vehicle` → `Car`, `Motorcycle`) y describe cómo se vería la(s) tabla(s) en cada estrategia.
5. **JDBC y operaciones**
   - Escribe un fragmento de código Java usando JDBC para:
     1. Conectar a una BD MySQL.
     2. Insertar un nuevo registro en una tabla `employees`.
     3. Recuperar y mostrar todos los registros de `employees`.
   - Explica el ciclo de vida de una transacción en JDBC (auto-commit, commit, rollback).
6. **JPA y JPQL**
   - Dada la entidad:

     ```
     @Entity
     class Author {
       @Id private Integer id;
       private String firstName;
       private String lastName;
       @ManyToMany private List<Book> books;
     }
     ```

     Escribe una consulta JPQL para obtener todos los autores cuyo apellido empiece por “Garc”.
7. **Frameworks ORM**
   - Compara brevemente **Hibernate** y **SQLAlchemy** en cuanto a características principales (configuración, lazy loading, caching).
   - Razona cuándo elegirías un ORM en lugar de SQL puro (ventajas y desventajas).

Ejercicios adicionales para repaso rápido

1. **Relacionar conceptos**
   - Une con flechas:
     - Active Record → encapsula tupla y lógica en la misma clase.
     - Table Data Gateway → objeto único con métodos CRUD para toda la tabla.
     - Identity Map → evita duplicar instancias, usa un mapa de objetos.
     - Lazy Load → retrasa carga de datos hasta que se accede.

2. **Ventajas e inconvenientes**
    - **Active Record**
        - Ventajas: simple, directo para proyectos pequeños; todo en una sola clase.
        - Inconvenientes: acopla lógica de negocio y persistencia; difícil escalar.
    - **Table Data Gateway**
        - Ventajas: centraliza acceso a tabla; independiza consultas SQL del dominio.
        - Inconvenientes: no gestiona relaciones; puede crecer mucho si hay muchas tablas.
    - **ORM (genérico)**
        - Ventajas: separa dominio y persistencia; portable entre BD; facilita pruebas.
        - Inconvenientes: curva de aprendizaje; posible sobrecarga de rendimiento; complejidad de configuración.
3. **Breves definiciones**
    1. *PreparedStatement*: sentencia SQL precompilada con parámetros, evita inyección.
    2. *EntityManager* (JPA): interfaz principal para operar con entidades (CRUD, consultas).
    3. *Session* (Hibernate): contexto de persistencia, administra transacciones y caché de primer nivel.
    4. *Cache de primer nivel*: almacén de objetos por sesión para evitar recargar entidades ya consultadas.
4. **Verdadero/Falso**
    1. "En JDBC, `autoCommit` está habilitado por defecto" → **V**.
    2. "Un `Row Data Gateway` contiene lógica de negocio sobre los datos de la fila" → **F** (sólo acceso).
    3. "En *Single Table Inheritance*, todas las subclases comparten una misma tabla con un campo `discriminator`" → **V**.
    4. "`Lazy Load` siempre carga todos los datos de una entidad al instanciarla" → **F** (carga bajo demanda).

Recomendaciones para el estudio rápido

- Memorizar diferencias entre **Active Record**, **Gateways** y **ORM**.
- Saber cuándo usar **PreparedStatement** y cómo funciona la inyección SQL.
- Practicar diagramas de **mapeo de relaciones** (1→N, N→M) y **herencia**.
- Recordar pasos básicos de **JDBC**: carga driver, conexión, statement, result set, cierre recursos.
- En **JPA/Hibernate**, entender anotaciones clave (`@Entity`, `@Id`, relaciones) y cómo se traduce a tablas.
- Repasar ejemplos prácticos de **Consulta JPQL** y sentencias **SQL** clásicas.
- Ensayar la redacción rápida de fragmentos de código JDBC y JPQL bajo condiciones de examen.

¡Ánimo con la preparación y éxito en el examen!