DAO

El patrón **DAO (Data Access Object)** es un patrón de diseño estructural que proporciona una capa de abstracción entre la lógica de negocio y la fuente de datos, como bases de datos o servicios externos. Este patrón ayuda a separar la lógica de acceso a datos del resto de la aplicación, lo que hace el código más modular, mantenible y fácilmente extensible.

1. ¿Qué es el patrón DAO?

El patrón DAO se utiliza para gestionar el acceso a los datos de forma independiente de la tecnología de persistencia utilizada (como bases de datos relacionales, archivos, servicios web, etc.). En lugar de interactuar directamente con las bases de datos, las clases DAO ofrecen una interfaz a través de la cual la aplicación puede realizar operaciones de **crear**, **leer**, **actualizar** y **eliminar** (CRUD) en los objetos.

2. Componentes del patrón DAO

El patrón DAO generalmente consta de los siguientes componentes:

- **Entidad o Modelo**: Representa la estructura de los datos que se almacenan o se recuperan de la base de datos.
- **DAO Interface**: Define los métodos de acceso a los datos (por ejemplo, `create`, `read`, `update`, `delete`).
- **DAO Implementation**: Implementa los métodos definidos en la interfaz para interactuar con la base de datos.
- **Connection Manager**: Gestión de la conexión a la base de datos.

3. Ventajas de utilizar DAO

- **Desacoplamiento**: La lógica de negocio no está directamente ligada al acceso a datos. Esto facilita el mantenimiento y las pruebas.
- **Flexibilidad**: Se puede cambiar la tecnología de persistencia (como cambiar de MySQL a PostgreSQL) sin afectar a la lógica de negocio.
- **Mantenibilidad**: La implementación de la persistencia se encuentra aislada en una capa específica, lo que facilita el mantenimiento y la evolución del sistema.

4. Implementación del patrón DAO en Java

4.1. Modelo o Entidad

Primero definimos una clase de modelo que representa la entidad que vamos a almacenar en la base de datos.

```
public class Alumno {
    private int id;
    private String nombre;
    private int edad;      // Constructor
    public Alumno(int id, String nombre, int edad) {
```

```
    this.id = id;
    this.nombre = nombre;
    this.edad = edad;
}
// Getters y setters
public int getId() { return id; }
public void setId(int id) { this.id = id; }
public String getNombre() { return nombre; }
public void setNombre(String nombre) { this.nombre = nombre; }     public int
getEdad() { return edad; }     public void setEdad(int edad) { this.edad =
edad; } }
```

4.2. DAO Interface

La interfaz `AlumnoDAO` define los métodos que se van a implementar para interactuar con la base de datos.

```
import java.util.List;  public interface AlumnoDAO {
    void crear(Alumno alumno);
    Alumno leer(int id);
    List<Alumno> leerTodos();
    void actualizar(Alumno alumno);
    void eliminar(int id);
}
```

4.3. DAO Implementation

La clase `AlumnoDAOImpl` implementa la interfaz `AlumnoDAO` y proporciona la lógica para realizar las operaciones en la base de datos usando JDBC.

```
import java.sql.*;
import java.util.ArrayList;
import java.util.List;
public class AlumnoDAOImpl implements AlumnoDAO {
    private Connection conn;     // Constructor de la conexión a la base de
datos
    public AlumnoDAOImpl() {
        try { this.conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mi_base_de_datos",
"usuario", "contrasena");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

```
    @Override
    public void crear(Alumno alumno) {
        String sql = "INSERT INTO alumnos (nombre, edad) VALUES (?, ?)";
        try (PreparedStatement pstmt = conn.prepareStatement(sql)) {
            pstmt.setString(1, alumno.getNombre());
            pstmt.setInt(2, alumno.getEdad());
            pstmt.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    @Override
    public Alumno leer(int id) {
        String sql = "SELECT * FROM alumnos WHERE id = ?";
        try (PreparedStatement pstmt = conn.prepareStatement(sql)) {
            pstmt.setInt(1, id);
            ResultSet rs = pstmt.executeQuery();
            if (rs.next()) {
                return new Alumno(rs.getInt("id"), rs.getString("nombre"),
                rs.getInt("edad"));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return null;
    }
    @Override
    public List<Alumno> leerTodos() {
        String sql = "SELECT * FROM alumnos";
        List<Alumno> alumnos = new ArrayList<>();
        try (Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery(sql)) {
            while (rs.next()) {
                alumnos.add(new Alumno(rs.getInt("id"),
rs.getString("nombre"),
                rs.getInt("edad")));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return alumnos;
    }
    @Override
```

```
    public void actualizar(Alumno alumno) {
        String sql = "UPDATE alumnos SET nombre = ?, edad = ? WHERE id = ?";
        try (PreparedStatement pstmt = conn.prepareStatement(sql)) {
            pstmt.setString(1, alumno.getNombre());
            pstmt.setInt(2, alumno.getEdad());
            pstmt.setInt(3, alumno.getId());
            pstmt.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void eliminar(int id) {
        String sql = "DELETE FROM alumnos WHERE id = ?";
        try (PreparedStatement pstmt = conn.prepareStatement(sql)) {
            pstmt.setInt(1, id);
            pstmt.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

4.4. Uso del DAO

En la clase principal o en el servicio, puedes utilizar el DAO para realizar las operaciones CRUD sin tener que preocuparte directamente por los detalles de la base de datos.

```
public class Main {
    public static void main(String[] args) {
        AlumnoDAO alumnoDAO = new AlumnoDAOImpl();   // Crear un nuevo alumno
        Alumno alumno1 = new Alumno(0, "Juan", 25);
        alumnoDAO.crear(alumno1);                    // Leer todos los
alumnos
        List<Alumno> alumnos = alumnoDAO.leerTodos();
        for (Alumno alumno : alumnos) {
            System.out.println(alumno.getId() + ": " + alumno.getNombre() + "
- "
            + alumno.getEdad());
        }          // Actualizar un alumno
        Alumno alumno2 = new Alumno(1, "Carlos", 26);
        alumnoDAO.actualizar(alumno2);               // Eliminar un alumno
        alumnoDAO.eliminar(1);
```

```
    }
}
```

5. Consideraciones adicionales

- **Transacciones**: Cuando trabajas con múltiples operaciones (como en una actualización y eliminación), es recomendable manejar las transacciones para asegurarse de que todas las operaciones se completen correctamente o se reviertan en caso de error.
- **Manejo de excepciones**: Asegúrate de manejar correctamente las excepciones, especialmente `SQLException` para poder tomar decisiones adecuadas en caso de error.
- **Cierre de recursos**: Siempre utiliza la sentencia `try-with-resources` para asegurar que los recursos como `Connection`, `PreparedStatement`, y `ResultSet` se cierren automáticamente.

6. Ventajas del patrón DAO

- **Desacoplamiento**: Separación de la lógica de acceso a datos de la lógica de negocio.
- **Mantenibilidad**: La capa de acceso a datos es independiente y puede modificarse sin afectar al resto de la aplicación.
- **Facilita las pruebas**: La implementación del acceso a datos se puede simular o mockear para facilitar las pruebas unitarias.

7. Mejores prácticas para implementar DAO

- **Uso de Interfaces**: Siempre define una interfaz DAO para que la implementación pueda cambiarse fácilmente (por ejemplo, usar otra base de datos o incluso un sistema no relacional) sin modificar la lógica de negocio.
- **Inyección de Dependencias**: Evita crear la conexión dentro del DAO; mejor pásala por constructor o usa un pool de conexiones para mejorar la gestión y pruebas.
- **Manejo de transacciones**: En aplicaciones más complejas, no manejes la transacción dentro del DAO, sino en una capa superior (servicios), para controlar mejor las operaciones atómicas.
- **Uso de PreparedStatement**: Siempre usa `PreparedStatement` para evitar inyección SQL y mejorar rendimiento en consultas repetidas.

---

8. Ejemplo de inyección de la conexión

En lugar de crear la conexión en el DAO, se pasa desde fuera:

```
public class AlumnoDAOImpl implements AlumnoDAO {
    private Connection conn;      // Constructor recibe la conexión
    public AlumnoDAOImpl(Connection conn) {
        this.conn = conn;
    }
```

```
    // ... métodos CRUD ...
}
```

Y en el código cliente:

```
try (Connection conn = DriverManager.getConnection(...)) {
    AlumnoDAO alumnoDAO = new AlumnoDAOImpl(conn);     // usar alumnoDAO...
}
```

---

9. Ejemplo básico de manejo de transacciones (en capa servicio)

```
public class AlumnoService {
    private Connection conn;
    private AlumnoDAO alumnoDAO;
    public AlumnoService(Connection conn) {
        this.conn = conn;
        this.alumnoDAO = new AlumnoDAOImpl(conn);
    }

    public void actualizarEliminar(Alumno alumno, int idEliminar){
        try {
            conn.setAutoCommit(false);
            alumnoDAO.actualizar(alumno);
            alumnoDAO.eliminar(idEliminar);
            conn.commit();
        } catch (SQLException e) {
            conn.rollback();
            throw e;
        } finally {
            conn.setAutoCommit(true);
        }
    }
}
```