CloudApps

MASTER

2.3 – Persistencia y Análisis de Datos

Tema 1 – Persistencia Relacional

Universidad Rey Juan Carlos

José Manuel Colmenar
josemanuel.colmenar@urjc.es
@jmcolmenarv

code URJC

Índice

- Introducción
- JPA e Hibernate
- Mapeo en Hibernate
- Relaciones entre entidades
- Herencia
- Fetching y eficiencia
- Gestión de concurrencia

Bibliografía

- Martin Fowler (2002). Patterns of Enterprise Application Architecture. Published by Addison-Wesley Professional.
- Merrick Schincariol; Mike Keith; Massimo Nardone (2018). Pro JPA 2 in Java EE 8: An In-Depth Guide to Java Persistence APIs. Published by Apress.
- Vlad Mihalcea (2019). High-Performance Java Persistence. Get the most out of your persistence layer. LeanPub.

Introducción

Objetos vs tablas (*impedance mismatch*)

- Las bases de datos relacionales almacenan los datos en tablas.
- Los datos en un programa son objetos.
  - Contienen estado y comportamiento.
  - Admiten relaciones bidireccionales.
- El esquema de base de datos se gestiona a través de SQL.
- Un modelo orientado a objetos se debe adaptar al esquema de base de datos y a la comunicación SQL.

Introducción

- El **mapeo objeto-relacional** (*ORM*, *Object-Relational Mapping*), ayuda a comunicar ambas representaciones (esquema vs objetos) haciendo que el salto de tecnologías sea transparente.
- **Cada fila de la BD se mapea a un objeto**.
  - En terminología JPA esos objetos son **entidades**.
- ORM traduce transiciones de estado de las entidades a comandos SQL.

Introducción

- **JPA (Java Persistence API)**:
  - Estándar de Java EE para persistencia.
  - Especificación de interfaces a implementar y de metadatos para ORM.
- **Hibernate**:
  - Implementación ORM en Java, nació antes que JPA.
  - Implementa tanto el estándar JPA como su API previa para retro-compatibilidad.
  - Proporciona características nativas no JPA que son relevantes para conseguir eficiencia.
  - http://hibernate.org

Introducción

- **Spring Data**
  - Capa de persistencia de Spring.
  - Se puede usar con cualquier base de datos (relacional o NoSQL).
  - Utiliza Hibernate para las bases de datos relacionales.
  - Proporciona métodos para carga y almacenamiento de datos.
  - Implementa métodos de consulta a partir de los nombres declarados en los repositorios.
    - Los métodos pueden tener varios parámetros que forman expresiones lógicas: Y, O, No…
  - Los métodos se pueden anotar para definir la consulta de forma más precisa con *Java Persistence Query Language* (**JPQL**).
  - http://docs.spring.io/spring-data/jpa/docs/current/reference/html/

JPA e Hibernate

- JPA no se define en términos de comandos SQL sino de **transiciones de estado de las entidades**.
- El *EntityManager* es el interfaz que define las operaciones de persistencia.
  - Se delega en su implementación el trabajo real de persistencia.
- El conjunto de entidades manejadas por el EntityManager se denomina **contexto de persistencia**.
  - El contexto de persistencia detecta los cambios de estado en las entidades y los traduce a comandos SQL en el *flushing*.
- *EntityManager* de JPA y *Session* de Hibernate canalizan las operaciones sobre el contexto de persistencia.
  - Definen todas las transiciones de estado de las entidades.

JPA e Hibernate

- Una entidad puede estar en uno de los siguientes estados:
  - ***New*** (*Transient*): entidad creada pero no mapeada a ninguna fila de la BD.
    - Cuando se convierte en *managed*, el contexto de persistencia genera un comando *insert*.
  - ***Managed*** (*Persistent*): una entidad persistente se corresponde con una fila de la BD, y está gestionada (*managed*) por el contexto de persistencia.
    - Los cambios se transmiten a la BD con comandos *update*.
  - ***Detached***: si el contexto de persistencia se cierra, las entidades se desvinculan de la BD y no se producen actualizaciones.
  - ***Removed***: una entidad sobre la que se ha ejecutado un comando *delete*.

JPA e Hibernate

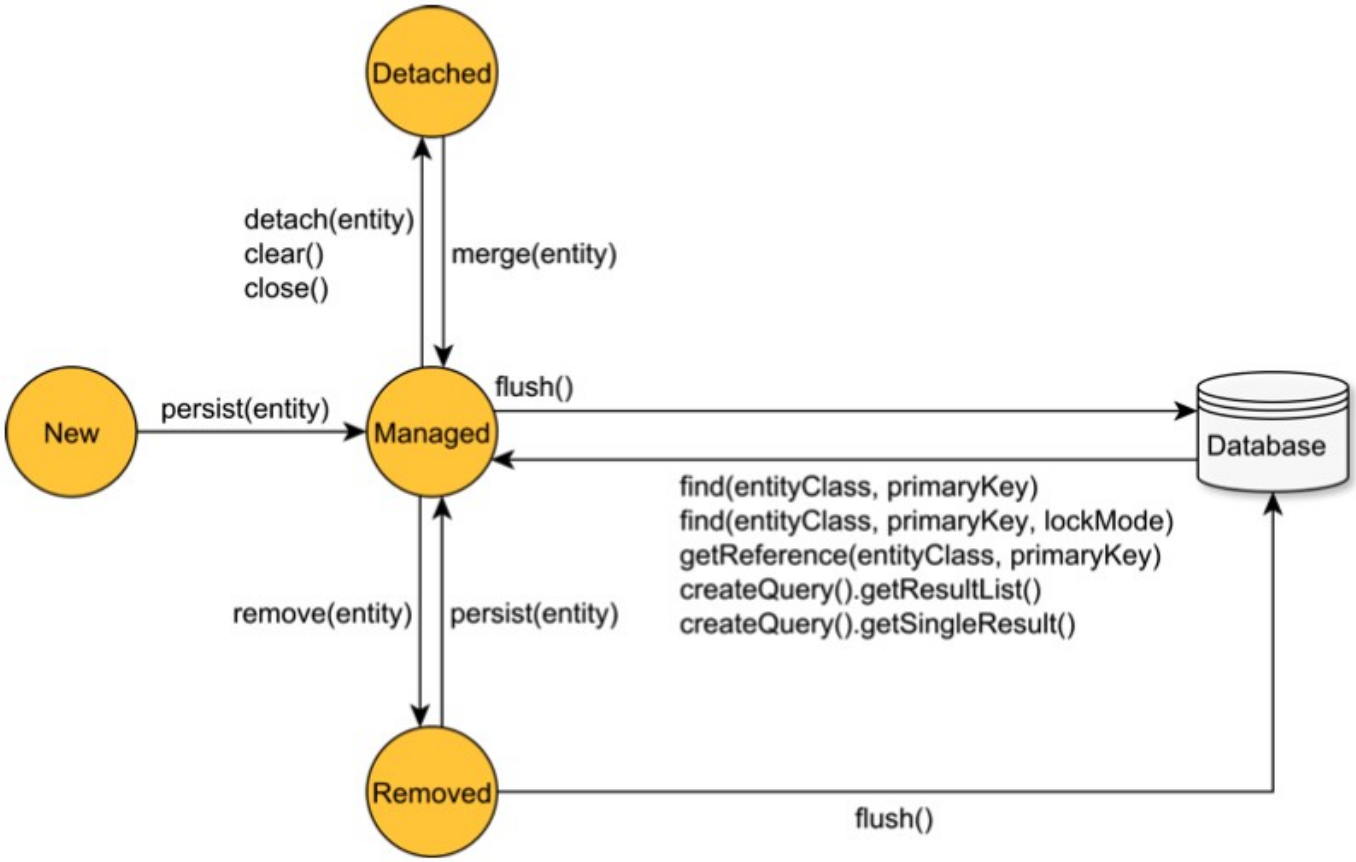


Figure 7.4: JPA entity state transitions

Fuente: Vlad Mihalcea (2019). High-Performance Java Persistence.

JPA e Hibernate

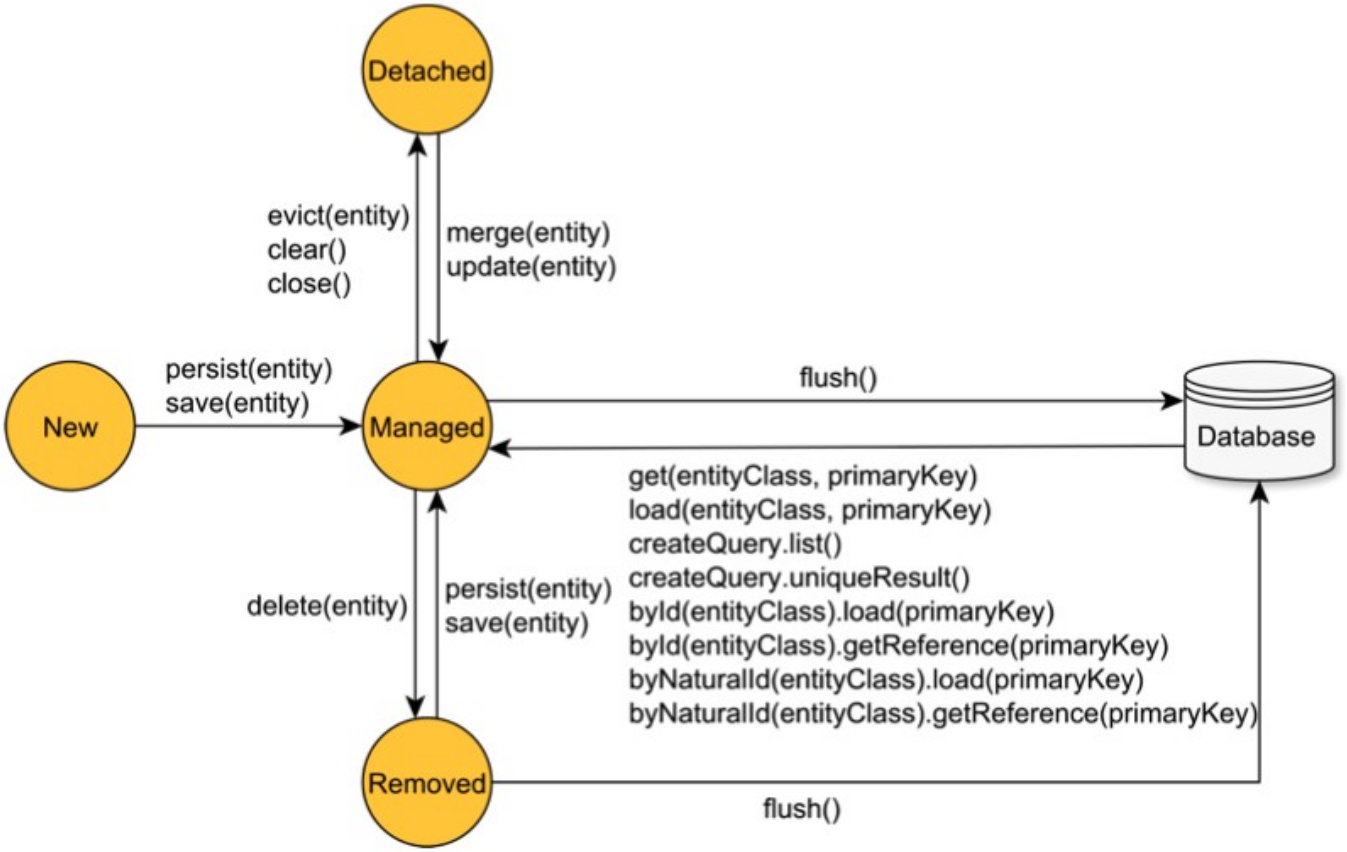


Figure 7.5: Hibernate entity state transitions

Fuente: Vlad Mihalcea (2019). High-Performance Java Persistence.

Mapeo en Hibernate

- Una entidad que se compone de atributos simples se transforma en una fila de una tabla.
- JPA maneja tres tipos de mapeo: ***type***, ***embedded*** y ***entity***.
- **A nivel de tipo (*type*)**, Hibernate realiza el mapeo entre los tipos JDBC (aplicados en el *driver* de la BD) y los tipos de Java.
  - Los tipos primitivos (*int*, *char*, ...) solamente se mapean a columnas NOT NULL. Para columnas que admiten NULL se deben utilizar los *wrappers* (*Integer*, *Char*, ...)
  - Las fechas son complicadas: horarios de verano e invierno, decalajes (*leap seconds*), ...
    - Se suelen guardar los datos en UTC y transformar a la zona horaria en la capa de datos.

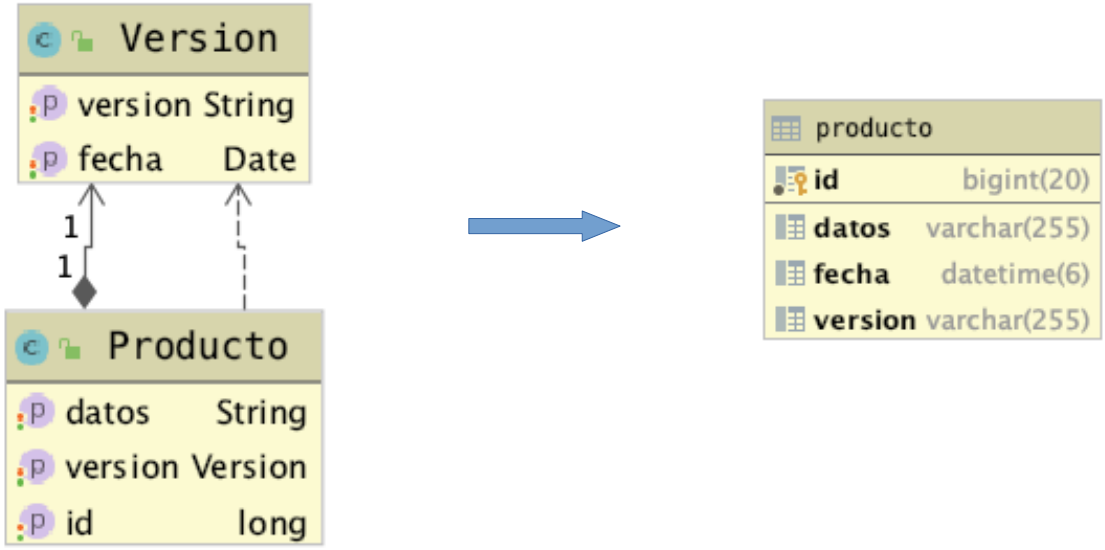
Mapeo en Hibernate

- El mapeo ***embedded*** se corresponde con la composición de objetos:
  - Si una entidad incluye un objeto *embeddable*, entonces sus atributos se añaden como columnas a la tabla que representa la entidad.
- Los objetos incrustados:
  - Se anotan con la etiqueta ***@Embeddable***.
  - No pueden tener identificador (clave), porque provocaría que cada entidad tuviese varios identificadores.
  - Tiene estado, pero se controla por la entidad "padre", no por el contexto de persistencia.

Mapeo en Hibernate

Ejemplo:

- Producto es la entidad padre.
- Version es un objeto incrustado.



Mapeo en Hibernate

Ejemplo:

- Entidad padre

```
@Entity
public class Producto {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    // Por ejemplo, marca y modelo
    private String datos;

    private Version version;

    public Producto() { }

   ...
}
```

Mapeo en Hibernate

Ejemplo:

- Objeto incrustado

```
@Embeddable
public class Version {

    private String version;
    private Date fecha;

    public Version() {}

    public Version(String version, Date fecha) {
        this.version = version;
        this.fecha = fecha;
    }

    // Getters y setters
    ...

}
```

Código en GitHub:

https://github.com/MasterCloudApps/2.3.Persistencia-y-analisis-de-datos/tree/master/tema1/ejemplo1_springdata_jpa_embedded

Relaciones entre entidades

- El mapeo ***entity*** se corresponde con la asociación de entidades.
- En el modelo relacional hay tres tipos de relaciones:
  - Uno a uno (1:1)
  - Uno a muchos (1:N)
  - Muchos a muchos (N:M)
- Desde el punto de vista del esquema, las entidades se traducen a tablas y las relaciones a claves ajenas.
- Estas relaciones se mapean en Hibernate de dos maneras:
  - Unidireccional: solamente una entidad (principal) tiene referencia a la otra.
  - Bidireccional: desde ambas entidades se accede a la asociada.
  - Se utilizan las anotaciones `@OneToOne`, `@OneToMany`, `@ManyToOne` y `@ManyToMany`
- Se puede utilizar *cascade* para no tener que guardar las dos entidades de la relación explícitamente a través de los repositorios. Basta con guardar la principal.

Relaciones entre entidades

- Las relaciones entre entidades también sirven para recuperar datos.
- Al recuperar datos de la BD existen dos opciones:
  - *Eager fetch*: carga "ansiosa", recupera también las entidades asociadas en una relación.
    - Puede afectar al rendimiento si hay muchas entidades relacionadas y tienen gran anchura.

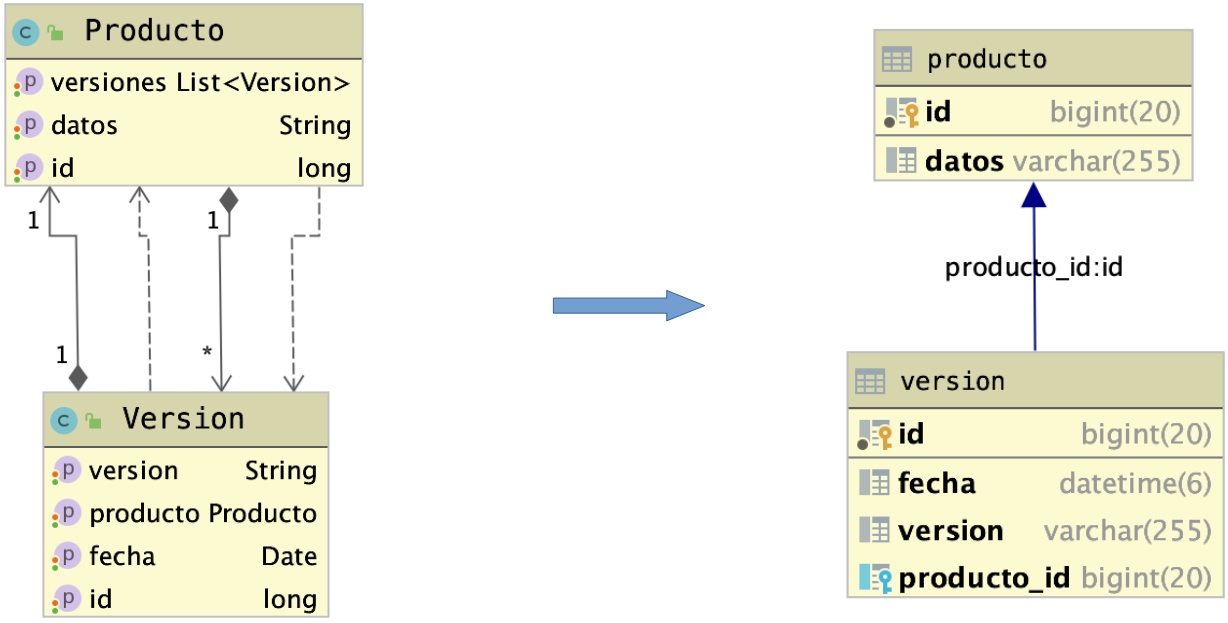
```
@OneToMany(fetch = FetchType.EAGER)
```

  - *Lazy fetch*: carga perezosa, no recupera entidades asociadas.
- Por defecto, desde JPA 2.0 el *fetch type* es LAZY.
  - Mejor para rendimiento.

Relaciones entre entidades

Ejemplo: `OneToMany`

- Un Producto puede tener varias Versiones (1:N)



Código en GitHub:

https://github.com/MasterCloudApps/2.3.Persistencia-y-analisis-de-datos/tree/master/tema1/ejemplo2_springdata_one_to_many

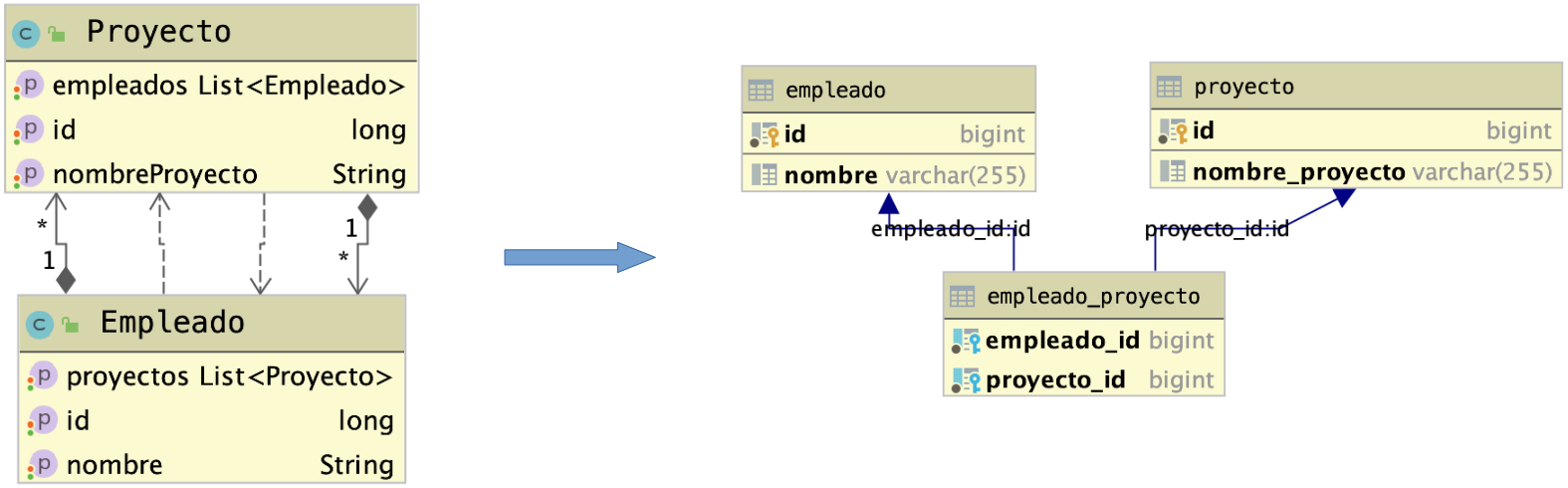
Relaciones entre entidades

- Las **relaciones N:M** (muchos a muchos) son las más complicadas de gestionar.
  - Una entidad A se relaciona con varias B y viceversa.
  - Implica tener una nueva tabla para almacenar las parejas de relaciones.
- Hibernate infiere la creación de una nueva tabla a partir de las anotaciones `@ManyToMany`.
- Existen algunos inconvenientes de rendimiento.

Relaciones entre entidades

Ejemplo: `ManyToMany`

- Un Empleado trabaja en varios Proyectos y un Proyecto tiene varios Empleados.



Código en GitHub:
https://github.com/MasterCloudApps/2.3.Persistencia-y-analisis-de-datos/tree/master/tema1/ejemplo3_springdata_many_to_many

Relaciones entre entidades

Ejemplo: `ManyToMany`

- El código SQL que se genera al añadir elementos es el que se espera: instrucciones *insert* para todas las tablas.
- Sin embargo, al eliminar el vínculo entre dos elementos, el comportamiento es diferente: borra todos los vínculos y después restaura.

```
// Guardando datos ...
Empleado e1 = new Empleado("Juan");
Proyecto p1 = new Proyecto("Proy 1");
Proyecto p2 = new Proyecto("Proy 2");
Proyecto p3 = new Proyecto("Proy 3");

List<Proyecto> proyectos = new ArrayList<>();
proyectos.add(p1);
proyectos.add(p2);
proyectos.add(p3);
e1.setProyectos(proyectos)
empleadoRepository.save(e1);

// Borrado de datos
e1.getProyectos().remove(p2);
empleadoRepository.save(e1);
```

Relaciones entre entidades

Ejemplo: `ManyToMany`

- El código SQL que se genera al añadir elementos es el que se espera: instrucciones *insert* para todas las tablas.
- Sin embargo, al eliminar el vínculo entre dos elementos, el comportamiento es diferente: borra todos los vínculos y después restaura.

```
// Inserción
insert into empleado (nombre, id) values (?, ?)
insert into proyecto (nombre_proyecto, id) values (?, ?)
insert into proyecto (nombre_proyecto, id) values (?, ?)
insert into proyecto (nombre_proyecto, id) values (?, ?)
insert into empleado_proyecto (empleado_id, proyecto_id) values (?, ?)
insert into empleado_proyecto (empleado_id, proyecto_id) values (?, ?)
insert into empleado_proyecto (empleado_id, proyecto_id) values (?, ?)

// Borrado:
select empleado0_.id as id1_0_1_, empleado0_.nombre as nombre2_0_1_,
proyectos1_.empleado_id as empleado1_1_3_, proyecto2_.id as proyecto2_1_3_,
proyecto2_.id as id1_2_0_, proyecto2_.nombre_proyecto as nombre_p2_2_0_ from
empleado empleado0_ left outer join empleado_proyecto proyectos1_ on
empleado0_.id=proyectos1_.empleado_id left outer join proyecto proyecto2_ on
proyectos1_.proyecto_id=proyecto2_.id where empleado0_.id=?

delete from empleado_proyecto where empleado_id=?
insert into empleado_proyecto (empleado_id, proyecto_id) values (?, ?)
insert into empleado_proyecto (empleado_id, proyecto_id) values (?, ?)
```

Relaciones entre entidades

Ejemplo: `ManyToMany`

- Si se invierte el orden de los proyectos asociados al empleado pasa lo mismo!!

```
// Inversión de los datos
e1.getProyectos().sort(Collections.reverseOrder(Comparator.comparing(Proyecto::getId)));
empleadoRepository.save(e1);
```

```
select empleado0_.id as id1_0_1_, empleado0_.nombre as nombre2_0_1_,
proyectos1_.empleado_id as empleado1_1_3_, proyecto2_.id as proyecto2_1_3_,
proyecto2_.id as id1_2_0_, proyecto2_.nombre_proyecto as nombre_p2_2_0_ from
empleado empleado0_ left outer join empleado_proyecto proyectos1_ on
empleado0_.id=proyectos1_.empleado_id left outer join proyecto proyecto2_ on
proyectos1_.proyecto_id=proyecto2_.id where empleado0_.id=?
select proyecto0_.id as id1_2_0_, proyecto0_.nombre_proyecto as nombre_p2_2_0_
from proyecto proyecto0_ where proyecto0_.id=?
delete from empleado_proyecto where empleado_id=?
insert into empleado_proyecto (empleado_id, proyecto_id) values (?, ?)
insert into empleado_proyecto (empleado_id, proyecto_id) values (?, ?)
```

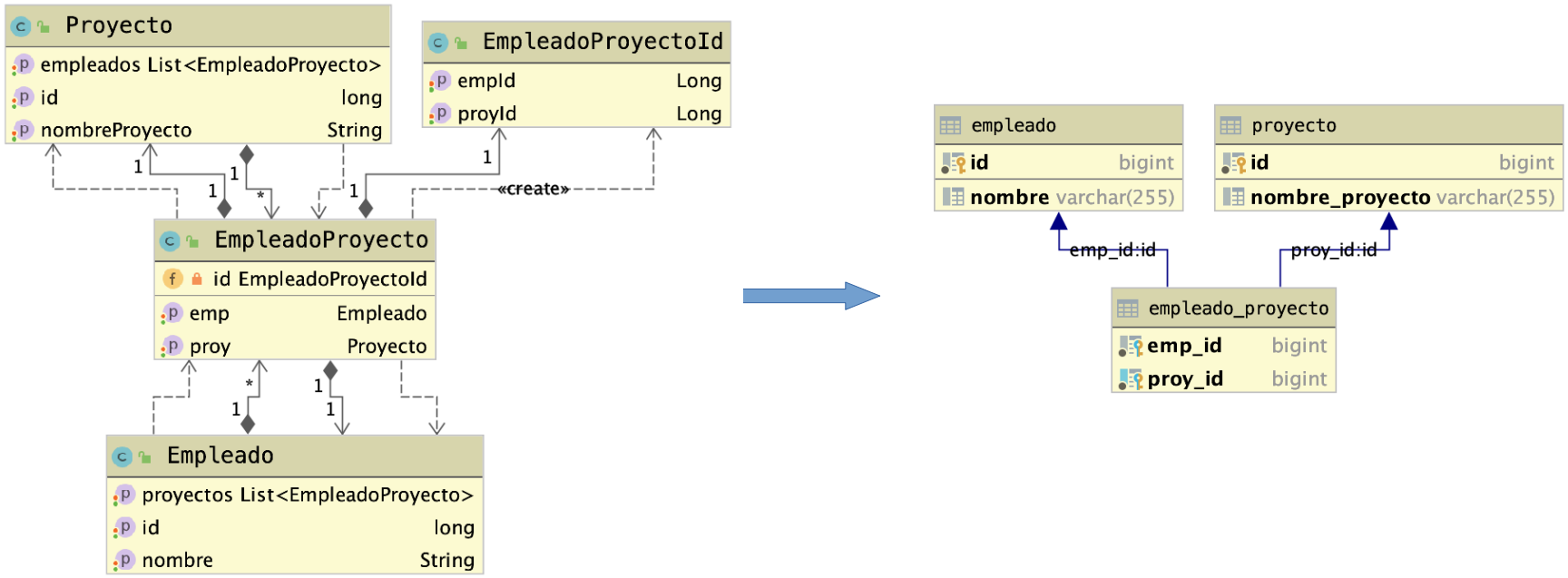
Relaciones entre entidades

- Para mejorar el rendimiento se puede optar por una aproximación tipo 1:N.
  - Una relación N:M se compone de dos relaciones 1:N a nivel de tablas.
- Se necesita una clave compuesta por los IDs de las entidades a relacionar.
  - Es necesario un `@Embeddable` (que debe implementar el interfaz *Serializable*).
- Se utiliza el lado del hijo para realizar las operaciones con peor rendimiento “por defecto”.

Relaciones entre entidades

Ejemplo: `ManyToMany` optimizado.

- Un Empleado trabaja en varios Proyectos y un Proyecto tiene varios Empleados.



Código en GitHub:
https://github.com/MasterCloudApps/2.3.Persistencia-y-analisis-de-datos/tree/master/tema1/ejemplo4_springdata_many_to_many_optimized

Relaciones entre entidades

Ejemplo: `ManyToMany` optimizado.

```
// Guardando datos ...
Empleado e1 = new Empleado("Juan");
Proyecto p1 = new Proyecto("Proy 1");
Proyecto p2 = new Proyecto("Proy 2");
Proyecto p3 = new Proyecto("Proy 3");

List<EmpleadoProyecto> rels = new ArrayList<>();
EmpleadoProyecto e1p1 = new EmpleadoProyecto(e1,p1);
EmpleadoProyecto e1p2 = new EmpleadoProyecto(e1,p2);
EmpleadoProyecto e1p3 = new EmpleadoProyecto(e1,p3);
rels.add(e1p1); rels.add(e1p2); rels.add(e1p3);

e1.setProyectos(rels);
empleadoRepository.save(e1);

// Se puede borrar el dato directamente tras desvincularlo
e1p2.getEmp().getProyectos().remove(e1p2);
e1p2.getProy().getEmpleados().remove(e1p2);
e1p2.setEmp(null);
e1p2.setProy(null);
empleadoProyectoRepository.delete(e1p2);
```

```
delete from empleado_proyecto where emp_id=? and proy_id=?
```

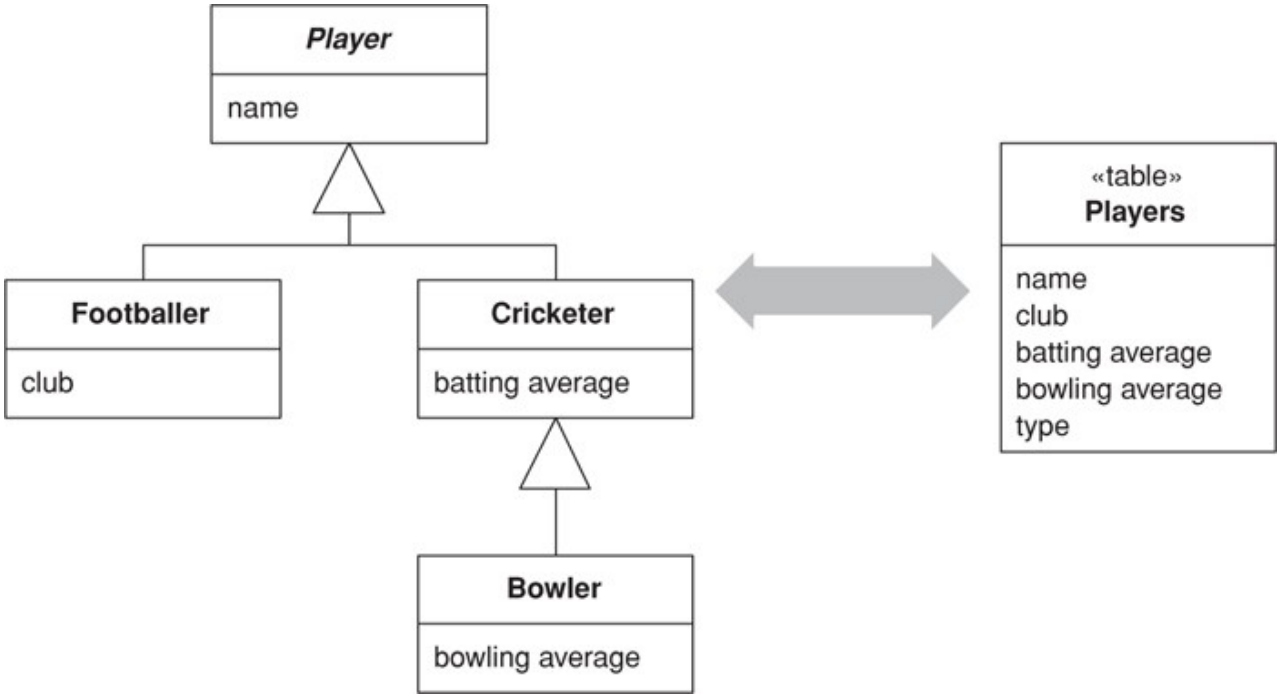
Herencia

- La herencia es una característica fundamental de la POO.
- Si el ORM ya es un problema en sí, incorporar la herencia lo complica aún más.
- Aunque el estándar SQL99 incorpora herencia de tipos, esta característica rara vez se implementa en SGBDs.
  - Si no hay soporte del SGBD se deben buscar alternativas.
- La herencia se implementa a través de relaciones entre tablas (claves ajenas).
- Martin Fowler define tres modos de implementación de herencia en BBDD relacionales en su libro *Patterns of Enterprise Application Architecture*:
  - *Single Table Inheritance*
  - *Class Table Inheritance*
  - *Concrete Table Inheritance*
- No es necesario aplicar el mismo patrón de herencia a todas las clases.

Herencia

***Single Table Inheritance*:**

- Utiliza una sola tabla para mapear todas las clases en una jerarquía.



Fuente: Martin Fowler (2002). Patterns of Enterprise Application Architecture.

Herencia

***Single Table Inheritance*:**

- En JPA es la estrategia por defecto, pero se puede indicar explícitamente como `InheritanceType.SINGLE_TABLE`.
- Hay un campo en la tabla que indica la clase (type) del dato almacenado en la fila.
- Los campos no relevantes quedan a *null*.
- Al leer los datos hay que saber qué clase cargar.
- La escritura la hace la superclase de la jerarquía.

Herencia

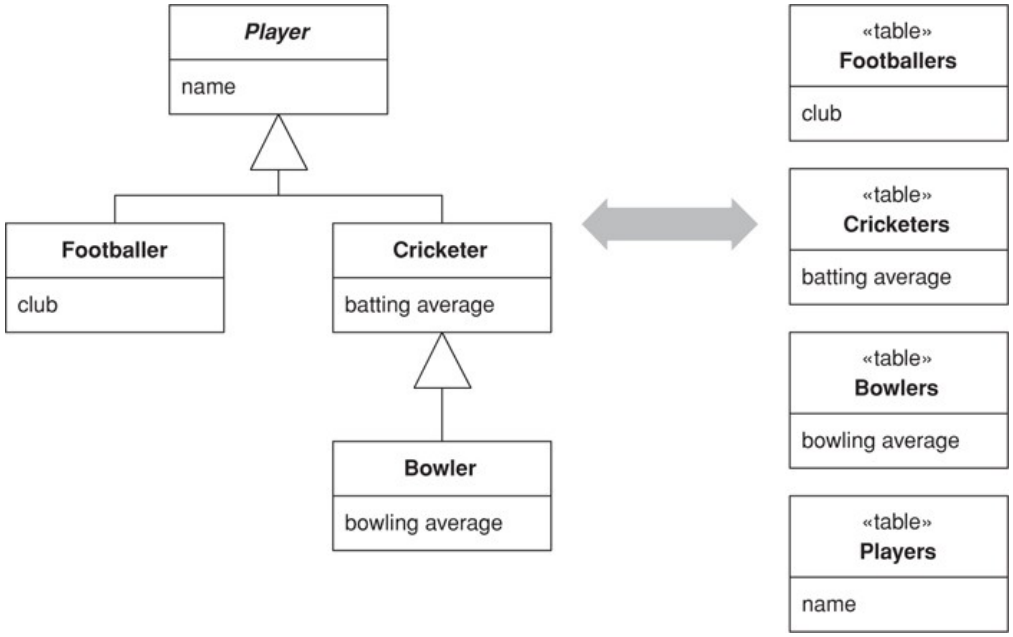
***Single Table Inheritance*:**

- Ventajas:
  - Solamente hay una tabla. No habrá *joins* entre miembros de la jerarquía.
  - Permite consultas polimórficas.
  - Cualquier refactorización de la jerarquía no altera la BD.
- Inconvenientes:
  - Pérdida de semántica de la jerarquía en la BD.
  - Puede generar una tabla muy ancha y con campos vacíos.
  - Si hay campos con nombres similares en diferentes subclases puede introducir confusión. Se recomienda usar como prefijo del nombre del campo el nombre de la subclase.

Herencia

***Class Table Inheritance*:**

- Mapea cada clase a una tabla y gestiona la herencia con asociaciones entre tablas (claves ajenas).



Fuente: Martin Fowler (2002). Patterns of Enterprise Application Architecture.

Herencia

***Class Table Inheritance*:**

- En JPA se indica con `@Inheritance(strategy = InheritanceType.JOINED)` en la clase madre.
- Una posible implementación es utilizar el mismo *id* en las filas de las clases hija y padre.
  - Además la clave primaria de la tabla hija es también clave ajena hacia la madre.
  - No se producen conflictos.
  - La alternativa es que las clases hijas tengan su propia *id*.
- Rendimiento: al guardar un objeto se realizan dos operaciones *insert*: una en la tabla madre y otra en la hija.
  - Si la herencia es más profunda, implica más *insert*.

Herencia

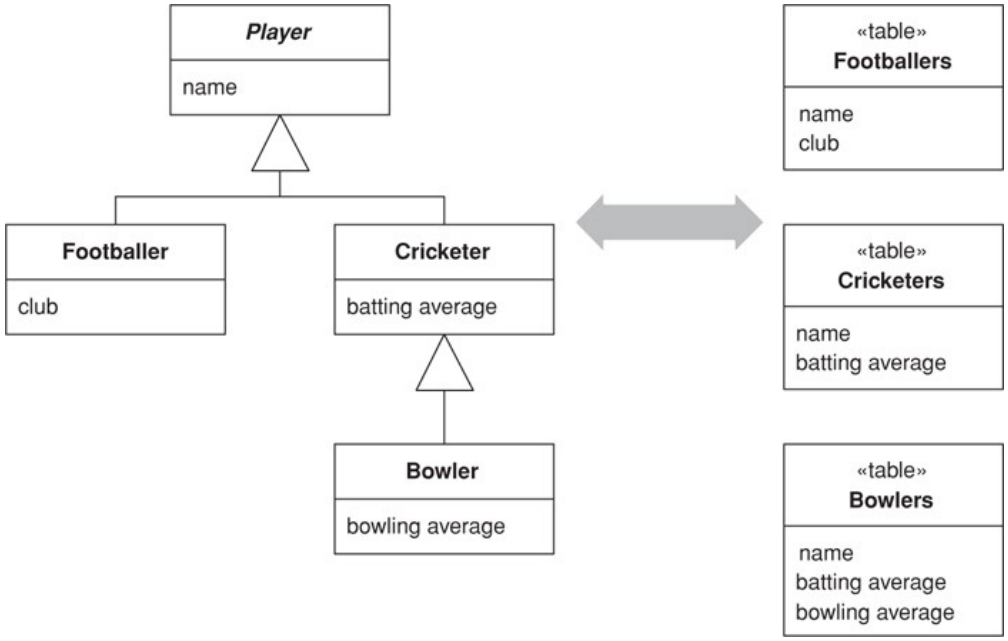
***Class Table Inheritance*:**

- Ventajas:
  - Todas las columnas son relevantes en las tablas, sin desperdicio de espacio.
  - No hay pérdida de semántica.
- Inconvenientes:
  - La recuperación de datos siempre implica un *join*.
  - La refactorización de campos hacia arriba o abajo de la jerarquía siempre implica cambios en la BD.
  - Las tablas madre son cuellos de botella por los accesos que reciben.
  - Si hay gran normalización, las consultas pueden ser complejas.

Herencia

***Concrete Table Inheritance*:**

- Mapea cada clase hija a una tabla, propagando hacia abajo los campos de la parte superior de la jerarquía.



Fuente: Martin Fowler (2002). Patterns of Enterprise Application Architecture.

Herencia

***Concrete Table Inheritance*:**

- En JPA se indica con `@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)` en la clase madre.
- Es importante que los *id* de las filas en la herencia sean diferentes a lo largo de la jerarquía.
  - Hibernate lo consigue con `hibernate_sequence`.
  - Si no se puede conseguir, se recomienda utilizar claves compuestas que incluyan el nombre de la tabla (clase).
- Las escrituras son más rápidas que en *class table inheritance*, pero las lecturas solamente son eficientes si se realizan sobre las subclases.
  - Las lecturas sobre la clase "padre" aplican *UNION ALL* entre todas las tablas de la jerarquía.

Herencia

***Concrete Table Inheritance*:**

- Ventajas:
  - Cada tabla es auto contenida y no tiene campos irrelevantes.
    - Puede ser útil para otras aplicaciones que no consideran objetos ni/o herencia.
  - No hay *joins* en lectura de datos de una subclase.
  - El acceso a las subclases individualmente reparte la carga y reduce cuellos de botella.

Herencia

***Concrete Table Inheritance*:**

- Inconvenientes:
  - Las claves primarias pueden ser difíciles de manejar.
  - No se pueden utilizar relaciones de base de datos en clases abstractas.
  - En caso de refactorización de campos arriba o abajo de la jerarquía, se tiene que modificar las tablas, aunque menos que en *class table inheritance*.
  - Si se modifica un campo de superclase hay que cambiar cada tabla que tenga este campo porque los campos de superclase están duplicados en las tablas.
  - Una consulta sobre la superclase obliga a comprobar todas las tablas, lo que lleva a múltiples accesos a la base de datos.

Herencia

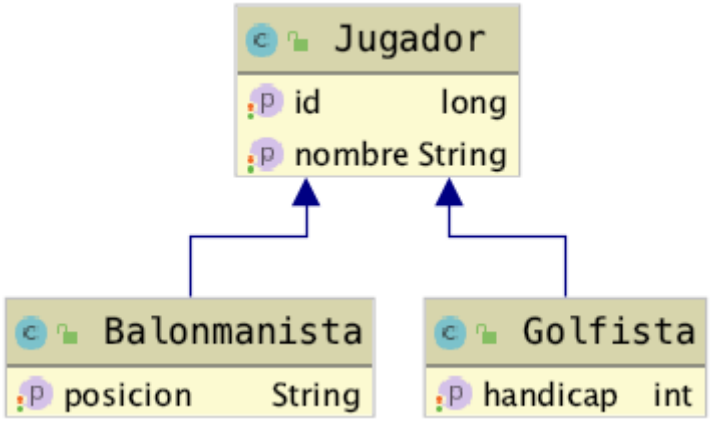
***Mapped Superclass*:**

- En la estrategia *concrete table inheritance* puede que sea conveniente eliminar la jerarquía de la base de datos porque no hay claves ajenas implicadas.
  - Se cambia la anotación `@Entity` por `@MappedSuperclass` en la superclase.
  - Se hace que la superclase sea **abstracta**.
- No se pueden hacer consultas polimórficas.
- Esta variante mejora la eficiencia:
  - Escrituras en una sola tabla.
  - Lecturas sin *joins*.

Herencia

Ejemplo:

- Jerarquía de jugadores de diferentes deportes.
- En cada paquete del ejemplo se utiliza una estrategia de implementación diferente.



Código en GitHub:

https://github.com/MasterCloudApps/2.3.Persistencia-y-analisis-de-datos/tree/master/tema1/ejemplo5_springdata_herencia

Herencia

Buenas prácticas sobre herencia:

- Implementar herencia en BD es complicado. Se recomienda hacerlo solamente si los beneficios son mayores que los inconvenientes.
- *Single table inheritance* es recomendable si existen pocas clases en la jerarquía.
  - Sus operaciones tienen buen rendimiento.
  - Si hay soporte para disparadores se pueden controlar restricciones sobre datos.
- *Class table inheritance* es recomendable si hay muchas clases en la jerarquía y no son tan necesarias las consultas polimórficas.
  - Las consultas polimórficas no son eficientes.
- *Concrete table inheritance* es el método menos efectivo para consultas polimórficas.
  - Se puede aumentar la eficiencia con *mapped superclasses*.
- Las estrategias de implementación de herencia se pueden combinar.

Fetching y eficiencia

- JDBC permite tomar el control sobre las sentencias SQL utilizadas.
  - Permite trabajar en comandos eficientes.
  - Requiere gran conocimiento de SQL.
  - Obliga a una implementación completa de la capa de persistencia.
- La utilización de ORMs ayuda a tratar con la “impedancia” de tablas vs objetos.
- Los ORM como Hibernate generan comandos SQL de manera automática.
- Permiten recopilar un grafo completo de entidades a través de un comando sencillo.
  - Pueden llevar a rendimiento poco eficiente si se desconoce su funcionamiento interno.
    - ***Too-much-fetching***

Fetching y eficiencia

- Es conveniente distinguir entre **entidades y otros resultados de consultas**.
- Una entidad es un dato que se hace persistente en la BD.
- Es habitual que haya consultas que producen resultados que no se corresponden exactamente con entidades. Fundamentalmente:
  - Subconjuntos de datos.
  - Consultas agregadas.
- Para trabajar con este tipo de situaciones se utilizan **DTOs (Data Transfer Objects)**.
  - Son proyecciones que permiten aprovecharse del mecanismo de persistencia automático.
  - No son entidades.

Fetching y eficiencia

- Un *workaround* típico cuando se trabaja con proyecciones de la BD es cargar un grafo de entidades y después filtrar en memoria.
- Las consultas sobre entidades no son la solución universal:
  - Si se utiliza un subconjunto de los datos, existe un desperdicio de recursos: memoria, ancho de banda, tiempo (recolector de basura), ...
  - Los resultados de las consultas sobre entidades son más difíciles de paginar, especialmente si tienen entidades asociadas.
  - Se aplican mecanismos de bloqueo y comprobación de cambios sobre los datos, que típicamente no se modifican en este tipo de consultas.
- Las consultas sobre entidades son útiles si es necesario modificar las entidades cargadas.
- Cargar más datos de los necesarios incrementa el tiempo de respuesta y desperdicia recursos.

Fetching y eficiencia

- Las proyecciones DTO son ideales al trabajar con subconjuntos de datos, consultas de solo lectura o agregaciones.
- Las proyecciones DTO son *type-safe*.
- Los repositorios permiten la utilización de proyecciones DTO a través de JPQL y la utilización de constructores.

- Implementación en Hibernate con JPQL:
  - Se crea la clase DTO (no es una entidad).
  - Se crea una consulta *ad hoc* que devuelva campos que coincidan con la definición de la clase.
  - La consulta debe “invocar” al constructor del DTO.

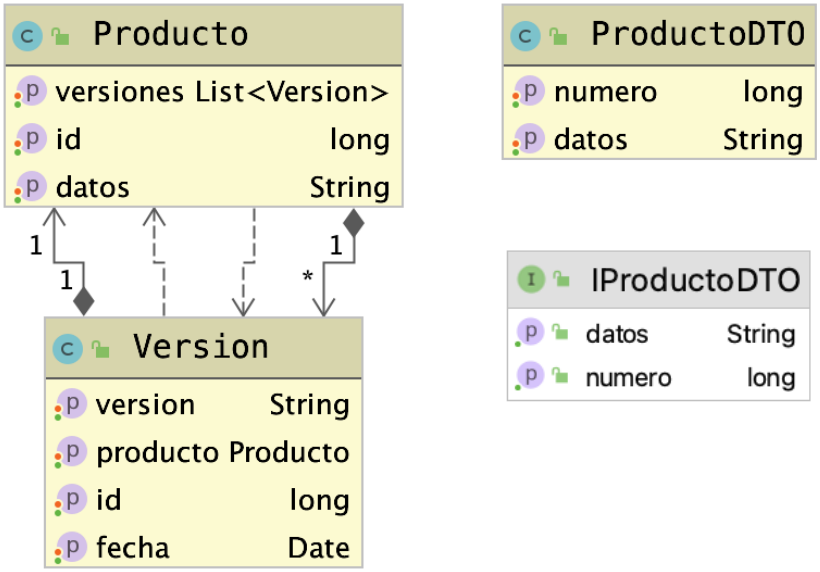
Fetching y eficiencia

- Implementación en Hibernate con *queries* nativas.
  - El DTO debe ser un interfaz que solamente incluya los *getters*.
    - Se definen tantos elementos como campos sean necesarios.
  - La *query* nativa devolverá los atributos usando los nombres indicados en los *getters* del interfaz.
  - No es necesario utilizar todos los elementos declarados en el interfaz.

Fetching y eficiencia

Ejemplo:

- Producto y versión con proyecciones DTO.



Código en GitHub:

https://github.com/MasterCloudApps/2.3.Persistencia-y-analisis-de-datos/tree/master/tema1/ejemplo6_springdata_dto

Gestión de concurrencia

- El acceso concurrente a bases de datos es muy común, y requiere una gestión eficiente.
- Las lecturas concurrentes no provocan reducción del rendimiento.
  - Salvo funciones de agregación bloqueantes.
- Las escrituras sí pueden provocar reducción de rendimiento.
- Transacciones: una transacción es un conjunto de operaciones que se ejecuta de manera atómica. Verifica las propiedades ACID
  - *Atomicity* (atomicidad): se ejecutan completamente o se deshacen completamente.
  - *Consistency* (consistencia): transición entre estados consistentes.
  - *Isolation* (aislamiento): no se afecta a otras transacciones.
  - *Durability* (persistencia): los cambios perduran.

Gestión de concurrencia

Grado de aislamiento (Isolation level): ¡afecta a las lecturas!

- READ_UNCOMMITED: Lectura de valores escritos por otras transacciones que todavía no han finalizado (y podrían abortarse finalmente) (*Dirty reads*)
- READ_COMMITED: Si durante la transacción se lee la misma fila dos veces y esa fila es modificada por otra transacción entre medias se leen valores diferentes (*Non-repeatable reads*)
- REPEATABLE_READ: Si se hace una consulta y se obtienen unas filas y otra transacción añade nuevas filas y se vuelve a hacer la misma consulta, aparecen filas nuevas (*Phantom reads*)
- SERIALIZABLE: Las transacciones están aisladas entre sí. Su comportamiento es como si se ejecutaran secuencialmente.

Gestión de concurrencia

Spring Data

- Anotación @Transactional hace que un método se ejecute como una sola transacción.
  - Solamente si el método anotado se invoca desde otro *bean*.
    - https://codete.com/blog/5-common-spring-transactional-pitfalls/
- Tanto *save* como *saveAll* están anotados como @Transactional
  - Afecta a rendimiento tanto del propio almacenamiento de datos como de la interacción con otros accesos concurrentes.
- Conviene repartir el trabajo en varias hebras.

Gestión de concurrencia

Ejemplo:

- Inserción de 10.000 productos
- Bucle *save* vs *saveAll*
- Concurrencia 1: 4 hebras escribiendo a la vez
  - Bucle *save* vs *saveAll* y conteo concurrentes.
- Concurrencia 2: 4 hebras escribiendo a la vez
  - Mismo código en dos métodos, uno anotado como @Transactional.
  - Se usa un *bean* diferente para que @Transactional funcione.

Código en GitHub:

https://github.com/MasterCloudApps/2.3.Persistencia-y-analisis-de-datos/tree/master/tema1/ejemplo7_springdata_concurrencia