UTN.BA
DPTO. INGENIERÍA EN SISTEMAS DE INFORMACIÓN
CÁTEDRA DISEÑO DE SISTEMAS

GUÍA DE ORM

El siguiente material está basado en la documentación oficial de JPA y Hibernate

con ejemplos propuestos por el equipo de la Cátedra de Diseño de Sistemas

Versión 1.0

Septiembre 2024

Realizado por:

- **Aylén Sandoval**
- **Juan Ignacio Borda**
- **Ignacio Vilarruel**

Revisado por:

- **Ezequiel Escobar**
- **Lucas Saclier**

Objetivo del apunte y aclaraciones

El objetivo de este apunte es sintetizar las formas que provee, en este caso, Hibernate, para mapear las relaciones existentes entre las entidades persistentes. Cabe aclarar que este apunte es una guía, no tiene la totalidad de casos de mapeos pero sí una introducción a los mismos.

En caso de querer colaborar y ampliar el documento, crear un issue en el repositorio utilizado de referencia [Repositorio].

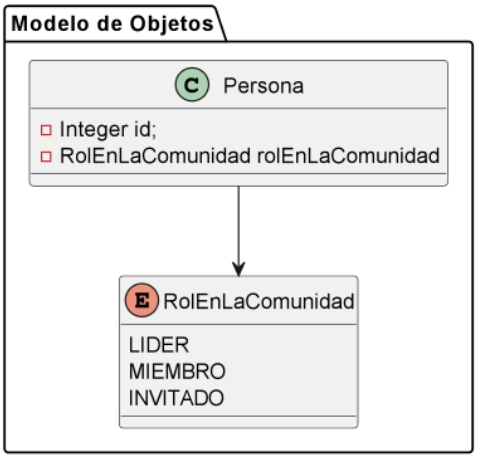
Convenciones utilizadas

- FK = Clave Foránea
- PK = Clave Primaria

Aclaración: No confundir Enumerado con Entidad.

Al usar PlantUml, en el **Modelo de Datos** las entidades se representan con la letra “E”:

En cambio, en el **Modelo de Objetos**, la letra “E” significa enumerado.



ORM - Concepto

ORM (Object Relational Mapping) o Mapeo Objeto-Relacional es una técnica utilizada para convertir los tipos de datos con los que trabaja un lenguaje orientado a objetos a tipos de datos con los que trabaja un sistema de base de datos relacional.

Frameworks ORM

Mientras que ORM es el nombre de la técnica, también se suele conocer como ORMs a los frameworks que la implementan. Algunos ejemplos de estos son Hibernate para Java, Entity Framework para C#, Eloquent para PHP, entre otros.

Estas herramientas introducen una capa de abstracción entre la base de datos y el desarrollador, lo que evita que el desarrollador tenga que escribir consultas “a mano” para recuperar, insertar, actualizar o eliminar datos en la base.

Existen dos tipos de arquitecturas para los framework ORM. Por un lado, se tiene la arquitectura **Active Record** en la cual los ORMs decoran las clases de entidades de dominio agregándoles funcionalidades/responsabilidades relacionadas a las acciones de ABM (alta, baja y modificación) y de búsqueda de las entidades. Por el otro, existe la arquitectura **Data Mapper**, a partir de la cual los ORMs agregan un nuevo componente intermedio entre la Base de Datos y las entidades de dominio denominado “entity manager”, responsable de buscar, agregar, modificar y eliminar objetos de las entidades persistentes.

Relaciones - Modelo Relacional

Recordar que en una base de datos las tablas se asocian entre sí mediante relaciones, estas son:

1. **One To One:** Es una relación entre dos entidades (tablas) A y B, en la cual un registro de la entidad A se corresponde con un único registro de la entidad B. La relación puede ser unidireccional, en cualquier sentido, o bidireccional.
2. **One To Many:** Es una relación entre dos entidades (tablas) A y B en la cual un registro de la entidad A está siendo referenciado por muchos registros de la entidad B.
3. **Many To One:** Es una relación entre dos entidades (tablas) A y B en la cual muchos registros de la entidad A están referenciando al mismo registro de la entidad B.
4. **Many To Many:** Es una relación entre dos entidades (tablas) A y B en la cual un registro de la entidad A puede estar siendo apuntado por muchos registros de la entidad B, y un registro de la entidad B puede estar siendo apuntado por muchos registros de la entidad A. Para poder implementarla, esta relación se debe partir la misma en dos relaciones One To Many. La entidad A debe tener una relación One to Many contra la entidad C, considerada “entidad intermedia”; al igual que la entidad B.

Uso de las anotaciones @Entity y @Table en Hibernate

La anotación **@Entity** en Hibernate se utiliza para marcar una clase de Java como una entidad persistente en la base de datos. Su utilización es obligatoria si se quiere asegurar que el ORM mapee dicha entidad como una tabla en la base de datos relacional. Por otra parte, **@Table** es una anotación opcional utilizada en conjunto con **@Entity** para especificar el nombre de la tabla de la base de datos que corresponde a la entidad.

```
@Entity @Table(name="servicio")
class Servicio {
  @Id @GeneratedValue
  private Integer id;
}
```

Uso de la anotación @Column en Hibernate

La anotación **@Column** en Hibernate se utiliza para mapear un atributo de una entidad a una columna en la base de datos. Permite personalizar aspectos como el nombre de la columna, si permite valores nulos, la longitud de los datos, y otras restricciones.

name: Define el nombre de la columna en la base de datos que se corresponde con el atributo. Si no se especifica, Hibernate usa el nombre del atributo.

```
@Column(name = "nombre_columna")
private String atributo;
```

nullable: Indica si la columna permite valores nulos. Por defecto, nullable = true.

```
@Column(nullable = false)
private String atributoObligatorio;
```

Existen muchos más atributos para Column: name, nullable, unique, length, precision, scale, columnDefinition, insertable, updatable, table, schema, catalog, columnDefinition, generated.

Persistencia de Imágenes:

Si se requiere almacenar imágenes, se debe guardar el PATH de la imagen como un String. De esta forma, se utiliza luego el FileSystem para recuperar la imagen utilizando el Path.

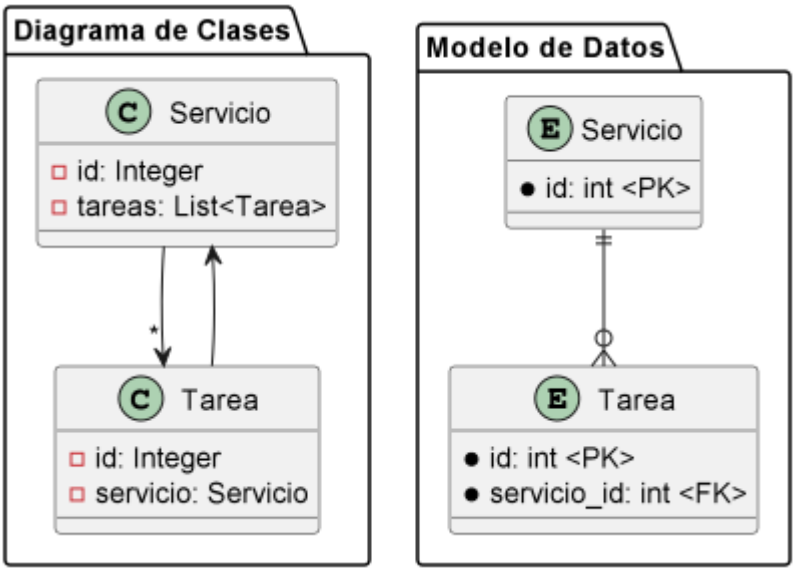
Mapeo Many To One | One To Many

Alternativa 1: Bidireccional a nivel Objetos

```
@Entity @Table(name="servicio")
class Servicio {
  @Id @GeneratedValue
  private Integer id;

  @OneToMany(mappedBy = "servicio")
  private List<Tarea> tareas;
}
```

```
@Entity @Table(name="tarea")
class Tarea {
  @Id @GeneratedValue
  private Integer id;

  @ManyToOne
  @JoinColumn(name="servicio_id",referencedColumnName="id")
  private Servicio servicio
}
```

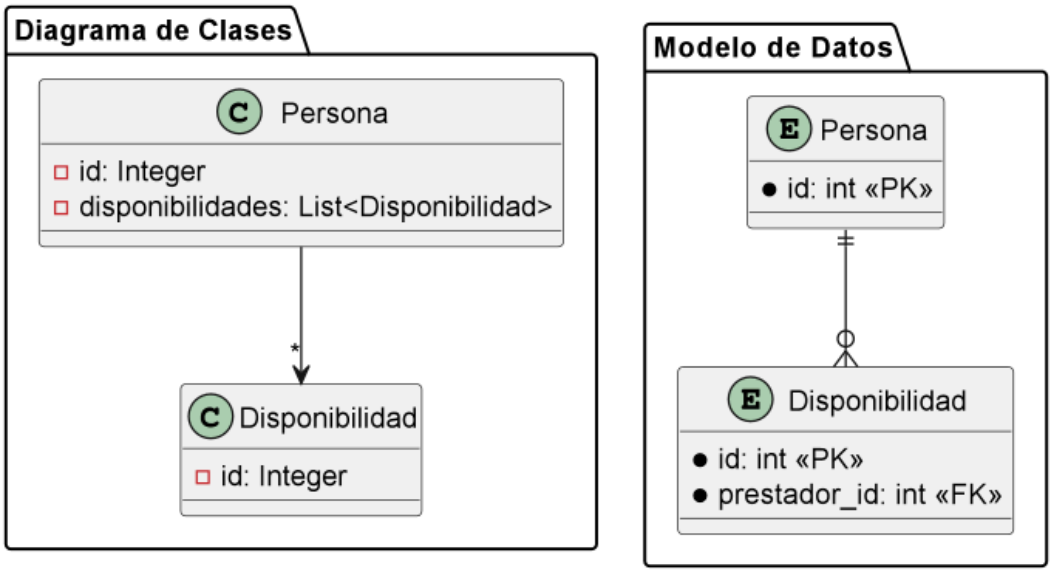
Nota: Si la dirección es bidireccional utilizar la etiqueta *mappedBy.*



Alternativa 2: Unidireccional a nivel Objetos

Si a nivel objetos va a ser Unidireccional en el OneToMany, usamos la *annotation* **@JoinColumn.**

```
@Entity @Table(name="Persona")
class Persona {
  @Id @GeneratedValue
  private Integer id;

  @OneToMany
  @JoinColumn(name="prestador_id", referencedColumnName = "id")
  private List<Disponibilidad> disponibilidades
}
```

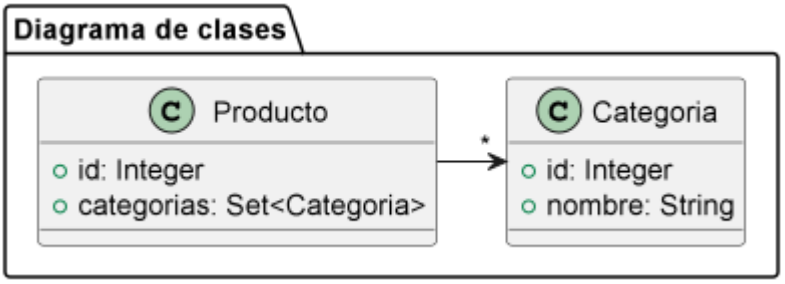


Notas:

- Mediante este mapeo se le define una columna “prestador_id” a la Disponibilidad.
- **ReferencedColumnName** es el atributo id de la disponibilidad.
- Si no se coloca el **@JoinColumn** Hibernate va a hacer una relación many To Many por defecto.

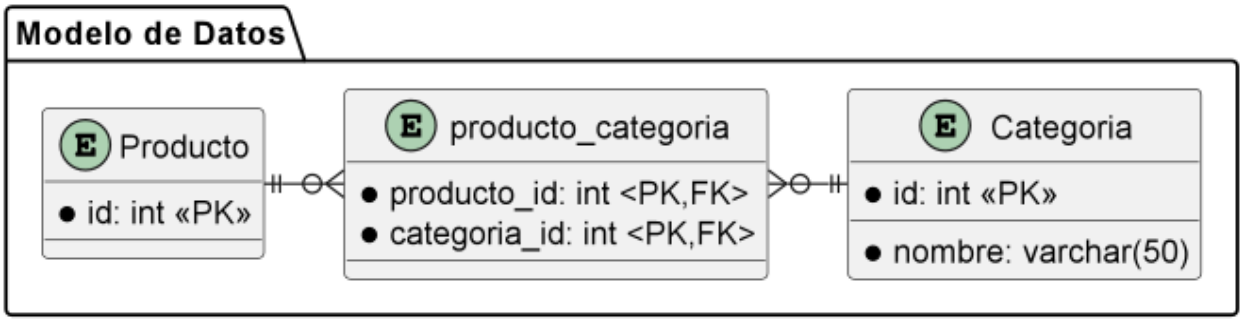
Mapeo: Many To Many

Escenario Simple: (la tabla intermedia contendrá únicamente FK a las entidades)




```
@Entity @Table
public class Producto {
  @Id @GeneratedValue
  private Long id;

  @ManyToMany
  @JoinTable(
    name = "producto_categoria",
    joinColumns = @JoinColumn(name = "producto_id",
    referencedColumnName = "id"),
    inverseJoinColumns = @JoinColumn(name = "categoria_id",
referencedColumnName = "id")
   )
    private Set<Categoria> categorias = new HashSet<>();
}
```

El modelo de datos quedaría de la siguiente manera:



Escenario Complejo

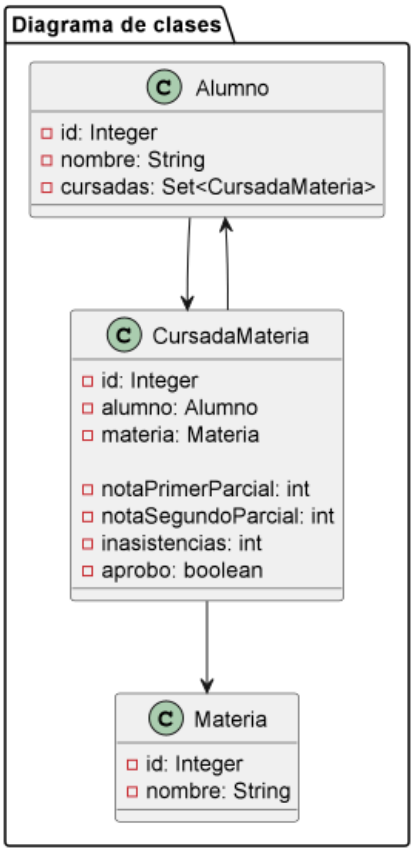
Cuando se presenta una relación many-to-many y necesitamos almacenar datos adicionales en la entidad intermedia, es necesario resolverlo a nivel de objetos creando una clase específica para representar esa relación. Por ejemplo, si se necesita registrar el estado de la cursada de un alumno, una simple relación many-to-many no sería suficiente. Por ello, a nivel de objetos, surge la abstracción CursadaMateria, que permite guardar los datos adicionales necesarios.

```java
@Entity @Table
public class Alumno {
 @Id
 @GeneratedValue
 private Integer id;

 @Column
 private String nombre;

 @OneToMany(mappedBy = "alumno")
 private Set<CursadaMateria> cursadas;
}
```

```java
@Entity @Table
public class Materia {
    @Id
    @GeneratedValue
    Integer id;

    @Column
    String nombre;
}
```

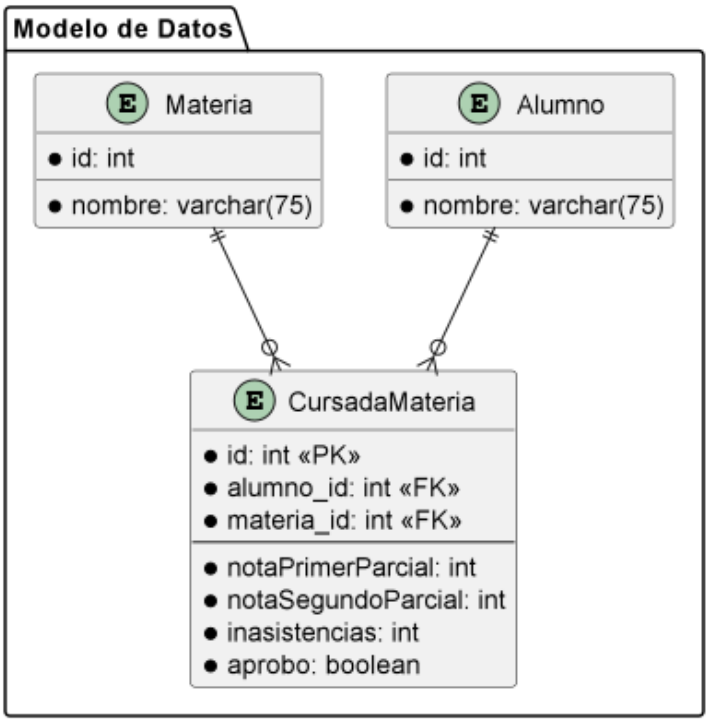
Cursada Materia para guardar atributos del alumno y materia.

```java
@Entity @Table
public class CursadaMateria {
    @Id @GeneratedValue
    private Integer id;

    @ManyToOne
    private Alumno alumno;

    @ManyToOne
    private Materia materia;

    private double notaPrimerParcial;
    private double notaSegundoParcial;
    private int inasistencias;
    private boolean aprobo;
}
```



[1] No se realiza el mapeo de estos 4 atributos por razones de espacio en el documento.

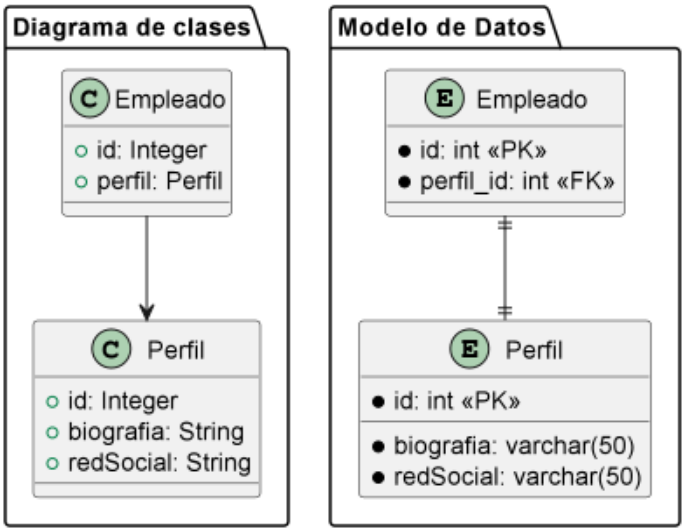
En el ejemplo, el modelo de datos del mapeo many-to-many complejo se podría ver así:



Mapeo Relación OneToOne

```
@Entity @Table(name = "empleado")
public class Empleado {
  @Id @GeneratedValue
  private Integer id;

  @OneToOne
  @JoinColumn(name = "perfil_id", referencedColumnName = "id")
  private Perfil perfil;
}
```

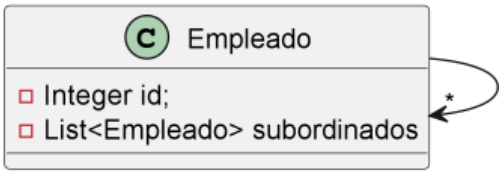
```
@Entity
@Table(name = "perfil")
public class Perfil {
  @Id @GeneratedValue
  private Integer id;

  @Column
  private String biografía;
```



Mapeo de Clases Recursivas

Por ejemplo: en un sistema, un empleado jefe tiene registro de sus subordinados.

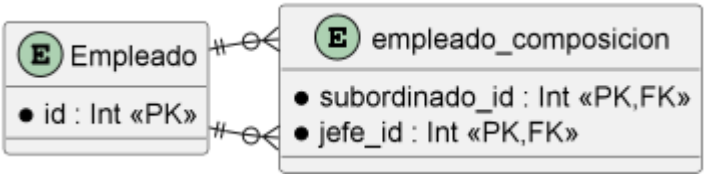
Diagrama de clases a mapear:



Alternativa 1: Crear una tabla intermedia para el mapeo.

```
@Entity
@Table(name = "Empleado")
class Empleado {
  @Id @GeneratedValue
  private Integer id;
  @ManyToMany
  private List<Empleado> subordinados;
}
```

Resulta en el modelo de datos:

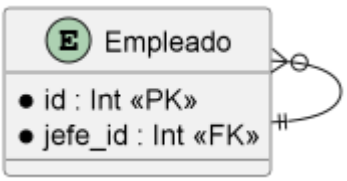


Alternativa 2: Guardar un atributo en la tabla

Empleado se autoreferencia a sí mismo guardando una FK en su misma tabla

```
@Entity
@Table(name = "Empleado")
class Empleado {
  @Id @GeneratedValue
  private Integer id;

  @OneToMany
  @JoinColumn(name="jefe_id", referencedColumnName = "id")
  private List<Empleado> subordinados;
}
```

Resulta en el modelo de datos:



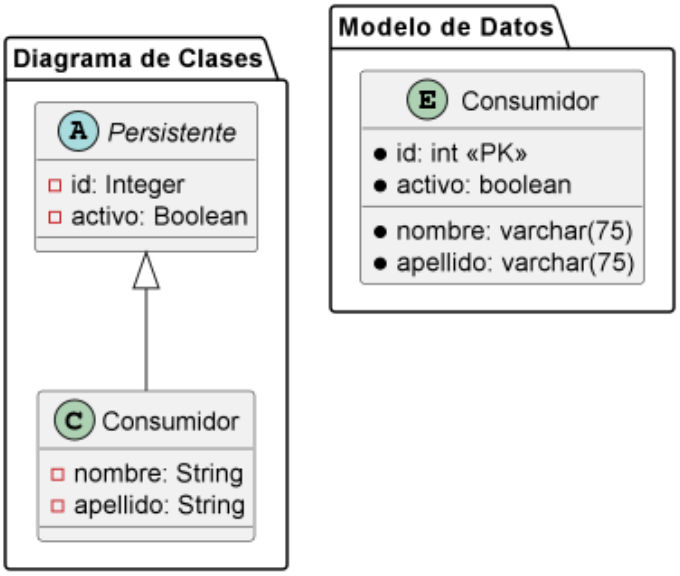
Estrategias para mapear Herencia

Mapeo de Herencia: Mapped Superclass

Se utiliza cuando no se considera a la superclase como una entidad, por lo que los atributos de la superclase son persistidos en las tablas de las clases hijas. Uno de los usos más comunes suele ser cuando todas las clases persistentes tienen una PK subrogada y/o un campo “activo”; pero entre ellas no existe nada más en común.

Es importante aclarar que esta estrategia no permite pedirle al ORM recuperar mediante una consulta a la base de datos la clase padre, pues no es considerada una entidad. En el ejemplo provisto, no se podría consultar por todos las clases que heredan de Persistente.

```
@MappedSuperclass
public abstract class Persistente {
@Id  @GeneratedValue
private Long id;

@Column(name="activo")
private Boolean activo
}
```



Desde la clase hija:

```
@Entity  @Table(name="consumidor")
public class Consumidor extends
Persistente {
  @Column(name="nombre")
  private String nombre

  @Column(name="apellido")
  private String apellido
}
```

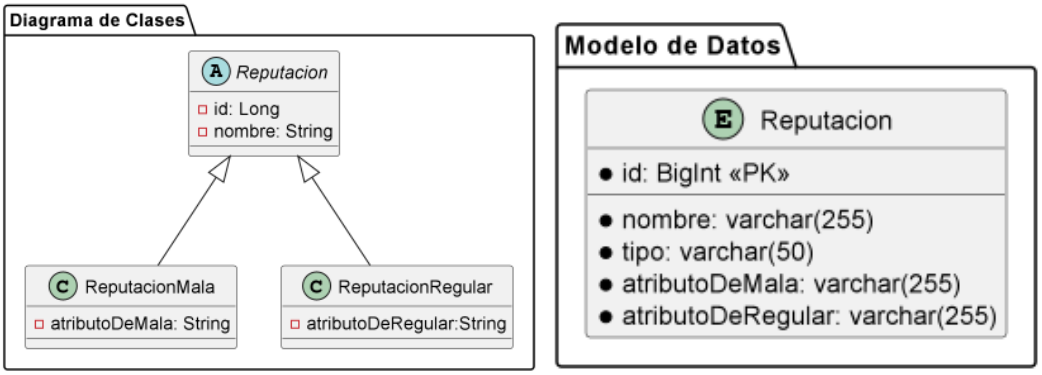
Mapeo de Herencia: Single Table

Suponiendo que existe una única superclase y N clases hijas, Single Table generará una única tabla en la base de datos que contendrá una columna por cada uno de los atributos persistentes de la superclase y una columna por cada atributo persistente de cada una de las clases hijas. Además, tendrá una columna que actuará de "**campo discriminador**", la cual indicará a qué clase pertenece la instancia/fila en cuestión.

```
@Entity @Table(name="reputacion")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="tipo")
public abstract class Reputacion {
  @Id @GeneratedValue
  private Long id;
  @Column(name="nombre")
  private String nombre
}
```

Las clases hijas se distinguen mediante **@DiscriminatorValue**, creando una única tabla con una columna "tipo" que identifica la clase hija según su valor.

```
@Entity  @DiscriminatorValue("mala")
public abstract class ReputacionMala extends Reputacion {
  @Column(name="atributoDeMala")
  private String atributoDeMala
}
@Entity  @DiscriminatorValue("regular")
public abstract class ReputacionRegular extends Reputacion {
}[2]
```



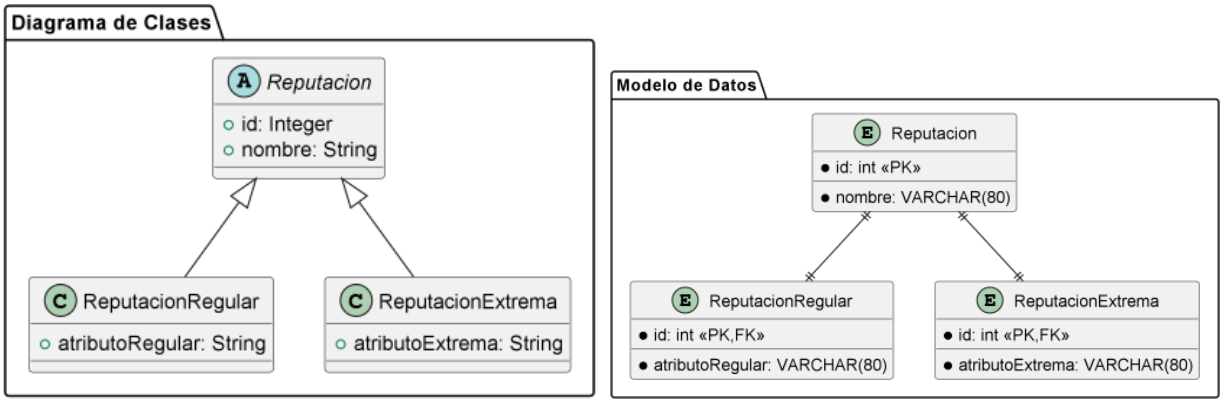
Notas: La estrategia de herencia "Single Table" evita joins al hidratar los objetos, lo que la hace más eficiente en rendimiento. Aunque esto puede generar columnas nulas en la tabla resultante, se gana en performance al consultar solo una tabla.

[2] Se ignoró el mapeo de los atributos por cuestión de espacio.

Mapeo de Herencia : Joined

Suponiendo que existe una única superclase y N clases hijas, el resultado de mapear la herencia con la estrategia Joined generará, en la base de datos, una tabla por la superclase y N tablas más, una por cada clase hija.

```
@Entity @Table(name="reputacion")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Reputacion {
  @Id @GeneratedValue
  private Integer id
  @Column(name="nombre")
  private String nombre
}
@Entity @Table(name="reputacion_regular")
public abstract class ReputacionRegular extends Reputacion {
  @Id @GeneratedValue
  private Integer id
  @Column(name="nombre")
  private String nombre
}
```



Notas:

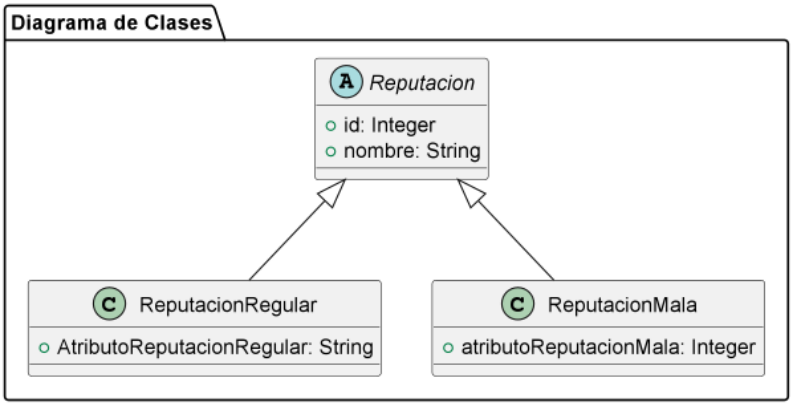
- La tabla que representa a la superclase contendrá, únicamente, tantas columnas como atributos persistentes existan en dicha clase.
- Cada una de las tablas que representan a las clases hijas contendrán tantas columnas como atributos persistentes existan en dichas clases.
- Cada una de las PKs de las tablas que representan a las clases hijas, a su vez serán una FK a la tabla que representa a la superclase.
- Para recuperar un objeto con todos sus atributos (los propios + los que están en la superclase), el ORM debe realizar *joins* entre las tablas.

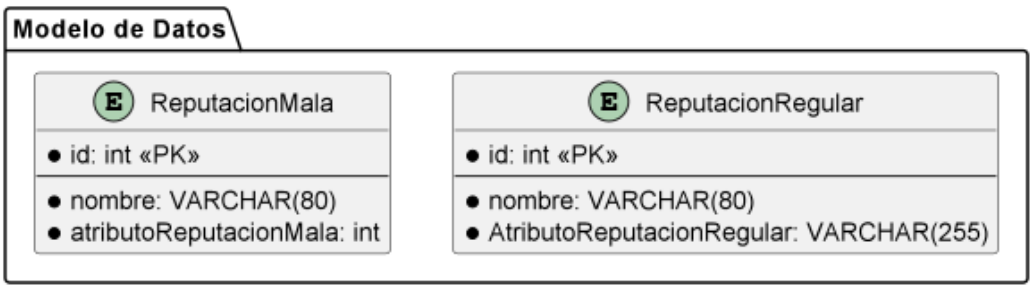
Mapeo de Herencia: Table Per Class

- Suponiendo que existe una única superclase y N clases hijas concretas, el resultado de mapear la herencia con la estrategia Table per Class generará, en la base de datos, N tablas: una por cada clase hija.
- Todos los atributos persistentes de la superclase serán persistidos en cada una de las tablas que mapean contra las clases hijas.
- Para recuperar polimórficamente todos los objetos, el ORM debe realizar *unions* entre las tablas que mapean contra las clases hijas.

```
@Entity  @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Reputacion {
  @Column(name="nombre")
  private Long id;
  @Column(name="nombre")
  private String nombre;
}
```

```
@Entity  @Table(name="reputacion_regular")
public abstract class ReputacionRegular extends Reputacion {
@Column
private String algunAtributo;
}
```

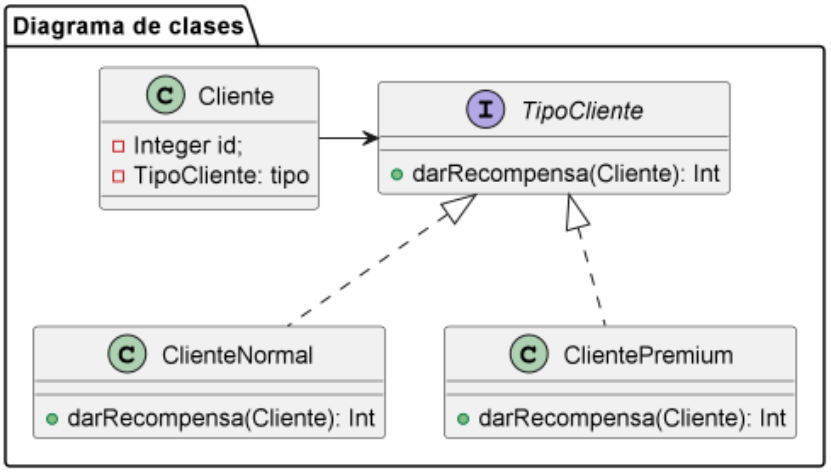




Mapeo de Interfaces:

Interfaz con clases concretas stateless:

Cuando una interfaz es *stateless*, es decir, las clases que la implementan no contienen atributos, es recomendable mapear la interfaz utilizando un *converter*. Esto asegura que no se pierda la referencia a la interfaz desde otras clases.




```java
public interface TipoCliente {
    public void darRecompensa(Cliente cliente);
}

public class ClienteNormal implements TipoCliente {
    @Override
    public void darRecompensa(Cliente cliente) {
        // Todo: implementar
    }
}

public class ClientePremium implements TipoCliente {
    @Override
    public void darRecompensa(Cliente cliente) {
        // Todo: implementar
    }
}
```

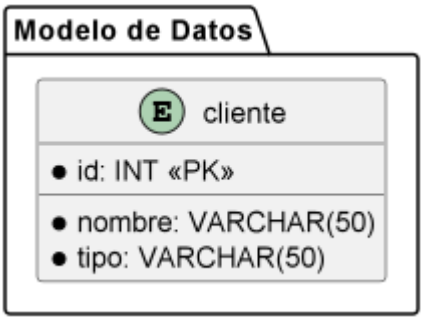
El converter para el caso de ejemplo quedaría de esta forma:

```
@Converter(autoApply = true)
public class TipoClienteConverter implements
AttributeConverter<TipoCliente, String> {

    @Override
    public String convertToDatabaseColumn(TipoCliente tipo) {
        if (tipo instanceof ClienteNormal ) {
            return "normal";
        } else if (tipo instanceof ClientePremium ) {
            return "premium";
        }
        return null;
    }

    @Override
    public TipoCliente convertToEntityAttribute(String dbData) {
        if ("normal".equals(dbData)) {
            return new ClienteNormal();
        } else if ("premium".equals(dbData)) {
            return new ClientePremium();
        }
        return null;
    }
}
```

Para cualquier definición de un converter se deben definir los métodos tanto para el proceso de mapeo (*convertToDatabaseColumn*) contra la base de datos como para el proceso de hidratación (*convertToEntityAttribute*).

Finalmente, se mapea la interfaz aplicando el converter:

```
@Entity @Table
public class Cliente {
    @Convert(converter = TipoCliente.class)
    @Column(name = "tipo")
    private TipoCliente tipo;
}[3]
```

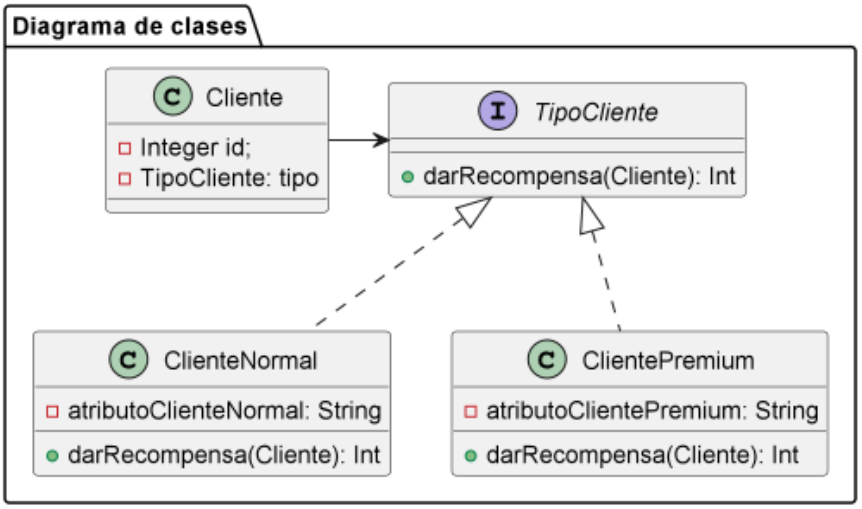


[3] No olvidar mapear el id del cliente.

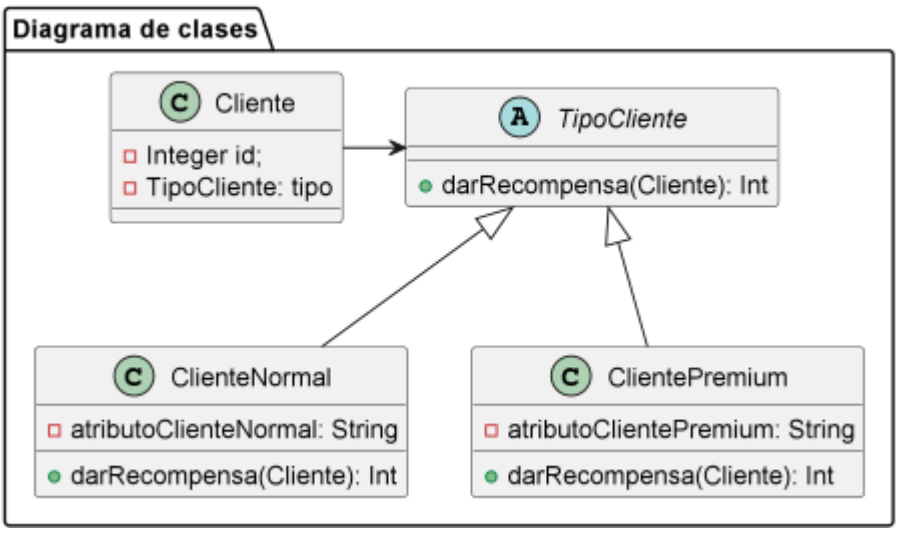
Interfaz con clases concretas stateful

Cuando una interfaz es *stateful*, es decir, las clases que la implementan contienen atributos, se puede intentar realizar inyección de dependencias. En caso de que resulte complejo hacerlo, se debe convertir la interfaz en clase abstracta y mapearla como se ha indicado anteriormente.

Estado Inicial



Luego de convertir la interfaz a clase abstracta



Persistir como lo visto en: Estrategia de herencia

Mapeo de enumerados

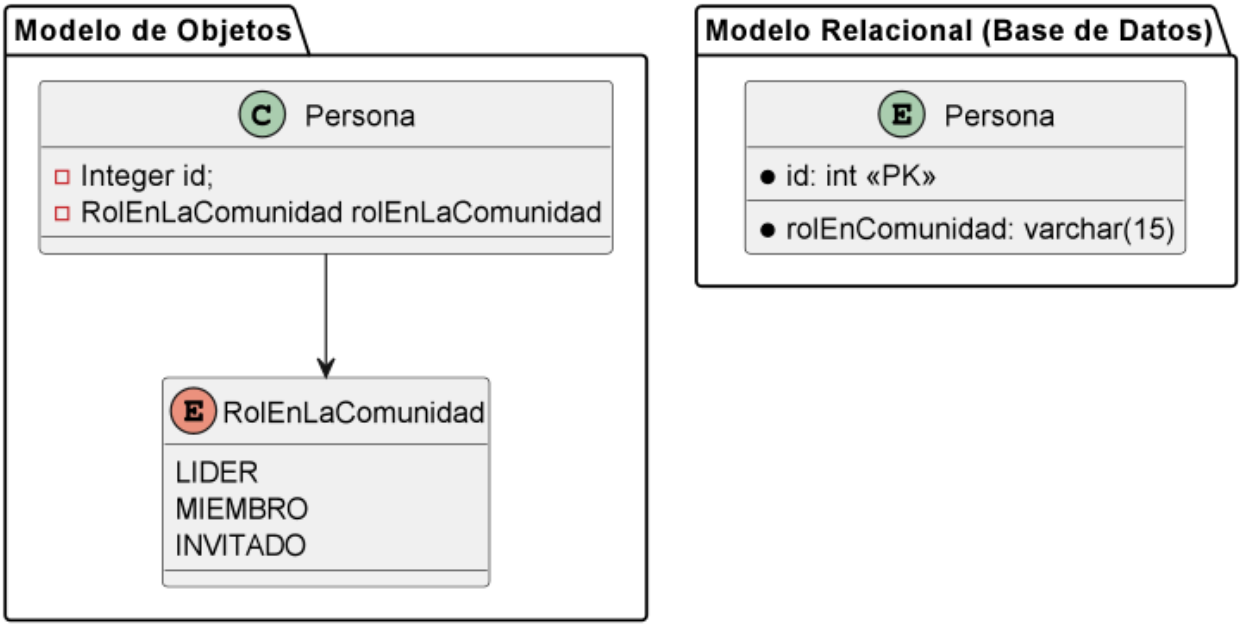
Los enumerados no deberían ser persistidos como tablas, sino que se puede optar por estas dos alternativas provistas por las *annotations* de Hibernate:

Alternativa 1: Enumerado persistido con su valor

```
@Enumerated (EnumType.STRING)
@Column(name="rolEnComunidad",nullable = false)
private RolEnLaComunidad rolEnLaComunidad;
```

Asi quedaria la consulta *Select * From Persona* en caso de persistir como un EnumType.String

id	rolEnComunidad
1	LIDER
2	MIEMBRO
3	INVITADO
NULL	NULL



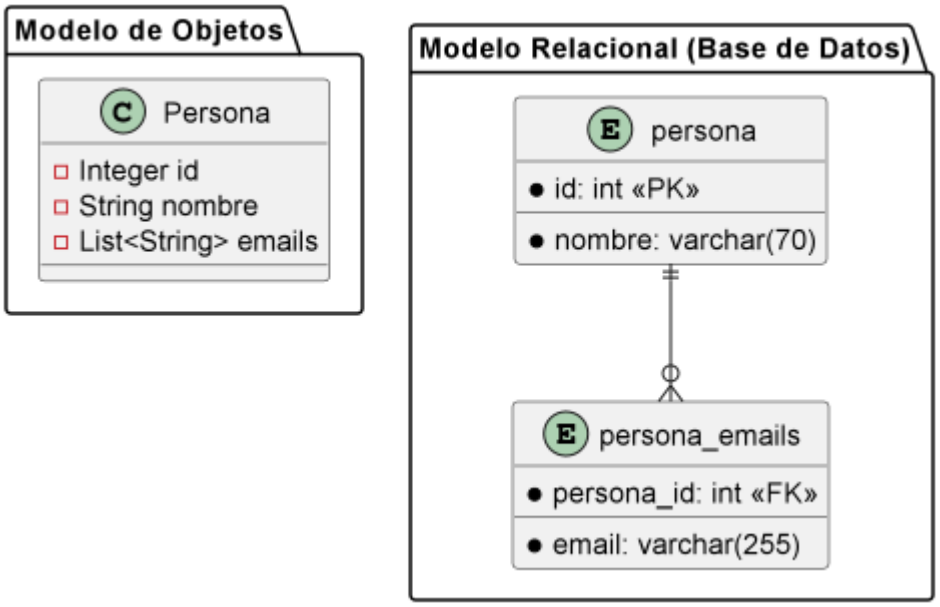
Alternativa 2: Enumerado persistido como Ordinal

```
@Enumerated (EnumType.ORDINAL)
@Column(name="rolEnComunidad",nullable = false)
private RolEnLaComunidad rolEnLaComunidad;
```

Mapeo de listas

Mapeo de lista con tipos de datos primitivos / Wrappers:

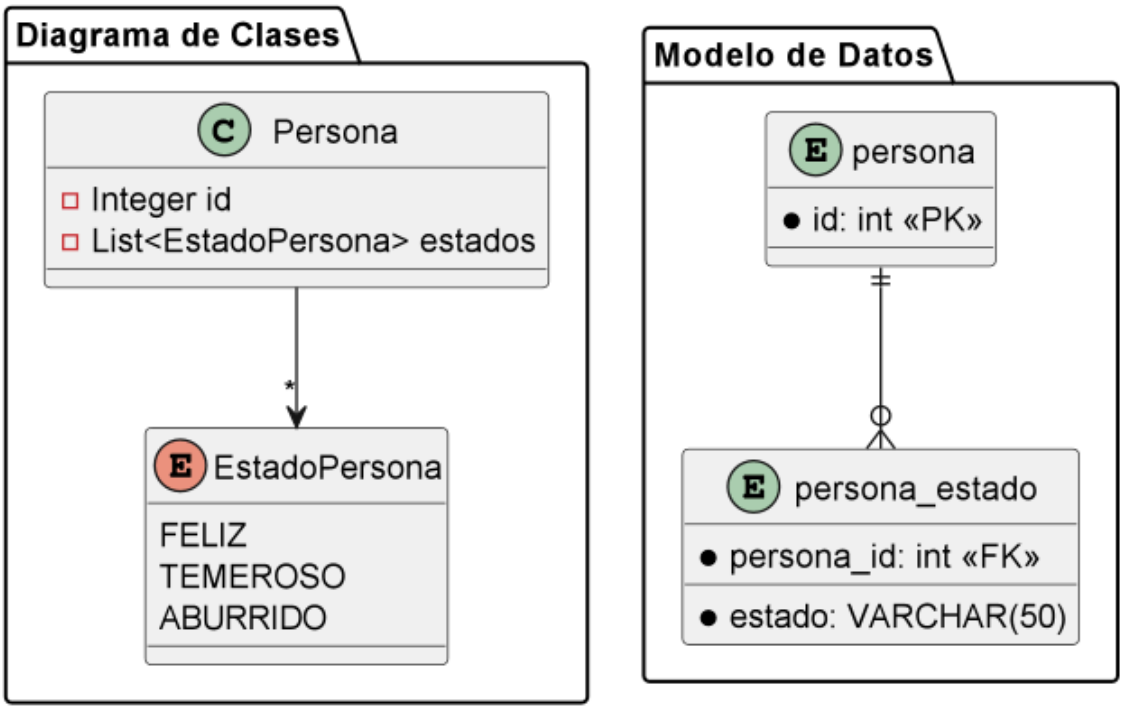
```
@ElementCollection
@CollectionTable(name = "persona_emails", joinColumns = @JoinColumn(name =
"persona_id"))
@Column(name = "email")
private List<String> emails;
```



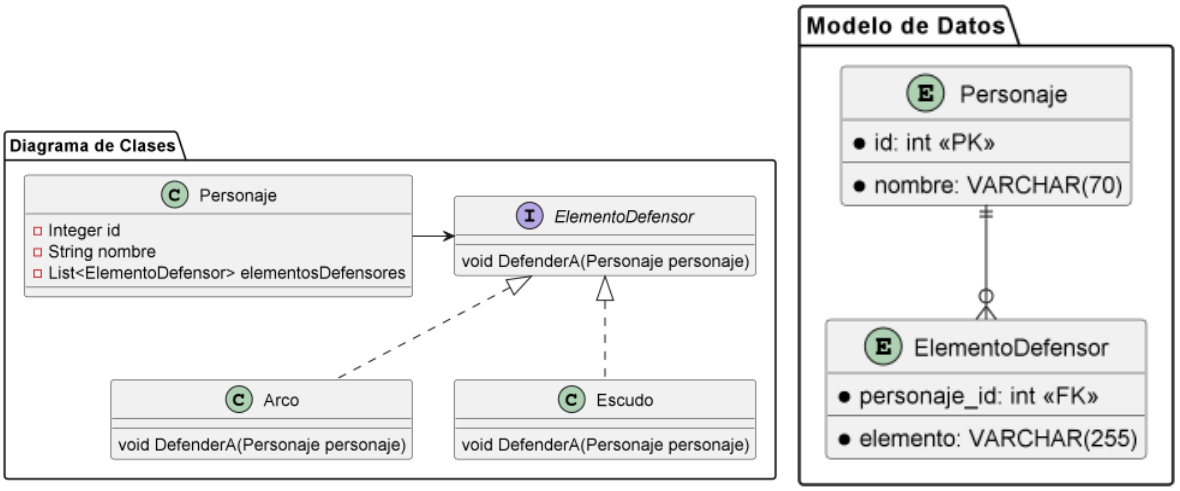
Mapeo de lista de enumerados

```
@Entity
@Table(name = "persona")
public class Persona {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Enumerated(EnumType.STRING)
    @ElementCollection()
    @CollectionTable(name = "persona_estado",
    joinColumns = @JoinColumn(name =
"persona_id",referencedColumnName="id")
    )
    @Column(name = "estado")
    private List<EstadoPersona> estados;
}
```

```
public enum EstadoPersona { FELIZ, TEMEROSO , ABURRIDO}
```



Mapeo de listas de interfaces stateless



Se puede ver que el Personaje tiene una **List<ElementoDefensor>** y elemento defensor es una **interfaz**, por lo tanto se tiene una lista de clases que no tienen atributos.
Se inicia creando el **ElementoDefensorConverter** y se mapea de la siguiente manera:

```
En la clase Personaje:
  @ElementCollection
  @CollectionTable(name = "ElementoDefensor")
  @Convert(converter = ElementoDefensorConverter.class)
  @Column(name = "elemento")
  private List<ElementoDefensor> elementosDefensores;
```

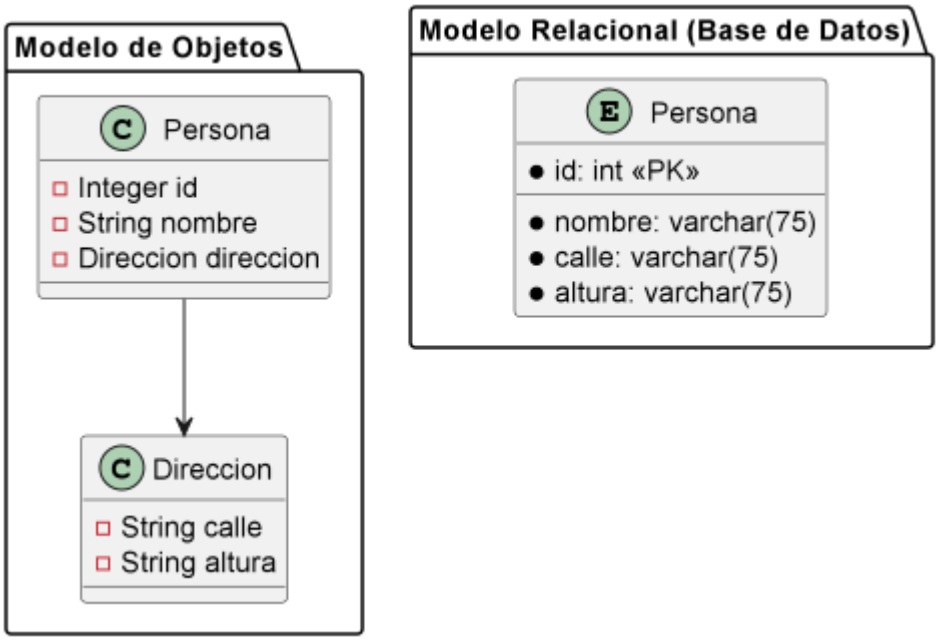
```
@Converter(autoApply = true)
public class ElementoDefensorConverter implements
AttributeConverter<ElementoDefensor, String> {

	public String convertToDatabaseColumn(ElementoDefensor elementoDefensor) {
		return elementoDefensor.getClass().getSimpleName();
	}

	public ElementoDefensor convertToEntityAttribute(String dbData) {
		switch (dbData) {
		case "Arco":
			return new Arco();
		case "Escudo":
			return new Escudo();
		default:
			throw new IllegalArgumentException("Unknown" + dbData);
		}
	}}
```

Embeber clases

Embeber una clase es incluirla dentro de otra, evitando que se represente como una entidad separada en la base de datos. En JPA/Hibernate, se usa ***@Embedded*** en la clase contenedora y ***@Embeddable*** en la clase embebida. Esto permite que los atributos de la clase embebida se almacenen en la misma tabla que la clase contenedora, simplificando la estructura de la base de datos.

```java
@Entity @Table(name"persona")
class Persona {
  @Id @GeneratedValue()
  private Long id;
  @Embedded
  private Direccion direccion;
}
```

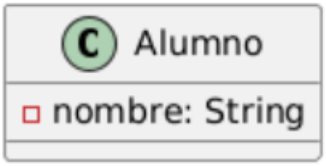
```java
@Embeddable
public class Direccion {
  @Column
  private String calle;
  @Column
  private String altura;
}
```



Query Builder

Query Builder es una herramienta que permite construir consultas SQL mediante el uso de objetos, en vez de escribirlas directamente como sentencias SQL puras. Puede resultar útil en casos donde las consultas son dinámicas o complejas ya que se pueden agregar filtros de manera flexible.

Ejemplo:



Considerando que existe una clase Alumno que tiene un atributo “nombre” se desea generar una consulta que obtenga todas las instancias de Alumnos cuyo nombre empiece con “A”. Para lograr esto se utilizan las clases de JPA (Java Persistence API) CriteriaQuery y CriteriaBuilder. [4]

```
public List<Alumno> filtrarAlumnosDeInicialA() {
 EntityManager entityManager = entityManagerFactory.createEntityManager();
 List<Alumno> alumnosConA = null;

 try {
   CriteriaBuilder cb = entityManager.getCriteriaBuilder();
   CriteriaQuery<Alumno> criteriaQuery = cb.createQuery(Alumno.class);
   Root<Alumno> root = criteriaQuery.from(Alumno.class);

   // Filtro para que el nombre comience con la letra "A"
   criteriaQuery.select(root).where(cb.like(root.get("nombre"), "A%"));

   alumnosConA = entityManager.createQuery(criteriaQuery).getResultList();
 } finally {
   entityManager.close();
 }

 return alumnosConA; // Devuelve la lista de alumnos
}
```

[4] Es un ejercicio a modo de ejemplo, también se pueden realizar Joins, ordenar los resultados, etc.
Para más información: JPA Criteria Queries | Baeldung

Referencias

- Documentación oficial de Hibernate - En línea [Hibernate]
- Diseño de Datos: Modelo Relacional – Zaffaroni Juan – En línea [Modelo relacional]
- Diseño de Datos: Mapeo Objetos Relacional – Dodino Fernando, Bulgarelli Franco – En línea [Apunte Mapeo Objetos/Relacional]
- Guía de persistencia de JPA – Bulgarelli Franco, Prieto Gastón – En línea [Guía de anotaciones de JPA]
- Introducción a los Sistemas de Base de Datos, C.J Date, Edit: Pearson, 2001.
- Overview of JPA/Hibernate Cascade Types – Baeldung -En línea [Overview of JPA/Hibernate Cascade Types. | Baeldung]
- Query Builder / Criteria Queries JPA Criteria Queries | Baeldung
- Query Builder / Criteria Queries Using the Criteria API to Create Queries :: Jakarta EE Documentation