## Configuración con anotaciones

Hasta ahora hemos visto cómo configurar el mapeado con ficheros XML (*.hbm.xml), pero existe una alternativa menos tediosa: La configuración de Hibernate con **anotaciones** en las propias **clases Java.**

De esta manera tendremos que mantener **menos ficheros** ya que sólo es necesario el .java.

Las **anotaciones en Java**:

- Proporcionan **información adicional** que puede ser procesadas por un

generador de código fuente, por el **compilador** o por una herramienta de

despliegue

- Las anotaciones en Java comienzan con **'@', por ejemplo, @Entity**

- No cambian la actividad de un programa
- Pueden tener atributos los cuales tienen un valor por defecto. Por ejemplo, en **@Table(name="profesor"),** name es un atributo de **@Table**

- Ayudan a relacionar metadatos (datos) con los componentes del programa, es decir, constructores, estrategias, clases, etc.

## Anotaciones JPA vs anotaciones Hibernate

Las anotaciones relacionadas con la persistencia pertenecen a un paquete en concreto:

- Las anotaciones de JPA pertenecen a **javax.persistence.***
- Las anotaciones de Hibernate pertenecen a **org.hibernate.***

Se recomienda usar las anotaciones del estándar de JPA y evitar en lo posible hacer uso directo de los paquetes de Hibernate.
Permitirá mantener la portabilidad del código y las compatibilidad con versiones futuras que se basen en JPA

Con anotaciones, el archivo **hibernate.cfg.xml** deberá incluir por cada clase Java una etiqueta: <mapping class="paquete.subpaquete.Clase"/>:

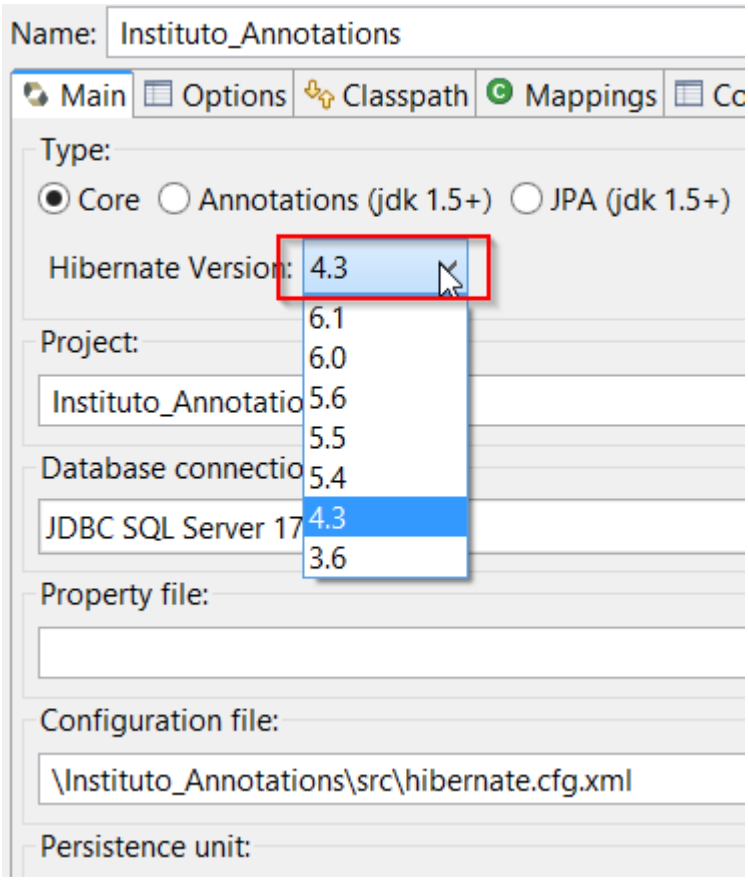
```
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.bytecode.use_reflection_optimizer">false</property>
        <property name="hibernate.connection.driver_class">com.microsoft.sqlserver.jdbc.SQLServe
        <property name="hibernate.connection.password">abc123.</property>
        <property name="hibernate.connection.url">jdbc:sqlserver://localhost:1433;database=insti
        <property name="hibernate.connection.username">user</property>
        <property name="hibernate.default_schema">dbo</property>
        <property name="hibernate.dialect">org.hibernate.dialect.SQLServer2016Dialect</property>
        <property name="hibernate.hbm2ddl.auto">none</property>
        <property name="hibernate.search.autoregister_listeners">true</property>
        <property name="hibernate.validator.apply_to_ddl">false</property>
        <mapping class="modelo.Direccion" />
        <mapping class="modelo.Profesor" />
        <mapping class="modelo.ComunidadAutonoma" />
        <mapping class="modelo.Provincia" />
        <mapping class="modelo.ContactInfo" />
        <mapping class="modelo.Ciclo" />
        <mapping class="modelo.Modulo" />
        <mapping class="modelo.Tiposbasicos" />
    </session-factory>
</hibernate-configuration>
```

Se puede encontrar información detallada sobre las anotaciones y sus atributos en la documentación de Hibernate.

## Generación de anotaciones con el plugin de Hibernate en Eclipse

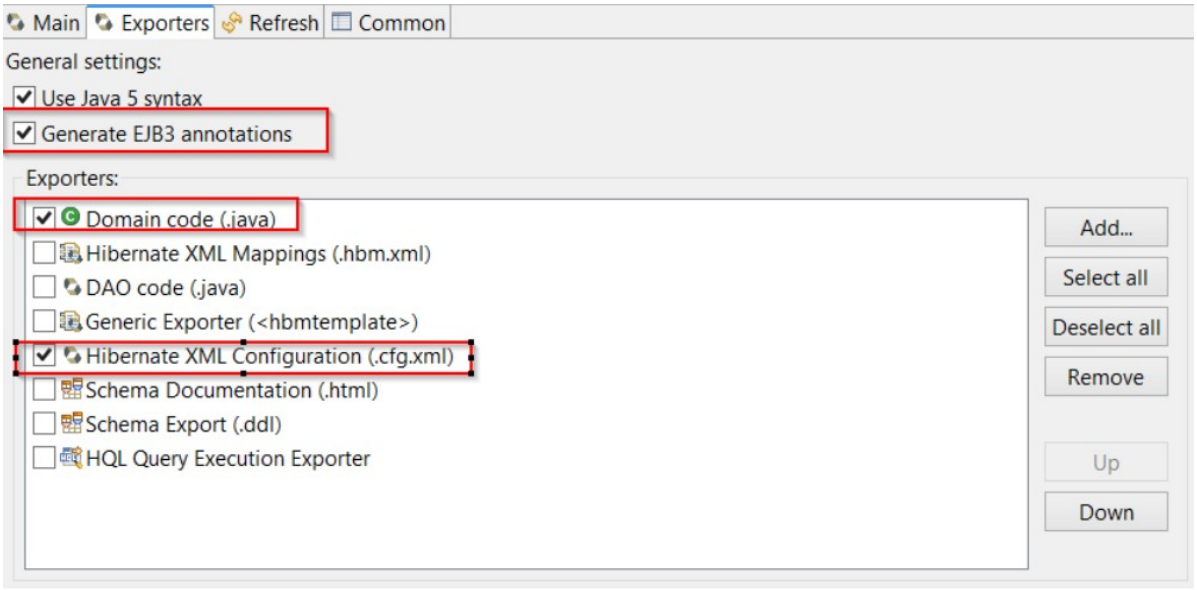
Se seguirán los pasos de la Actividad 3.1 con algunos cambios:

En el paso 13 b. en lugar de seleccionar la versión de Hibernate 5.6 , seleccionaremos la versión 4.3



En el paso 15.f) en la pestaña Exporters seleccionaremos, además de Use Java 5 syntax

- Generate EJB3 annotations
- Domain code (para generar clases Java)
- Hibernate XML Configuration (cfg.xml) (para que modifique




Podemos crear un nuevo proyecto Java y generar sobre la base de datos instituto las clases del dominio para ver el resultado que generaría el plugin.

## Anotaciones a nivel de clase

- **@Entity:** Cada clase POJO persistente debe anotarse con @Entity. Para las clases POJO rigen las mismas reglas que vimos en la sección Clases POJO del archivo PDF: UD3 Herramientas de mapeamento objeto-relacional (ORM)_V3.pdf
- **@Table**: permite especificar los detalles de la tabla que va a ser usada para persistir la entidad en la base de datos. La anotación **@Table** tiene los siguientes atributos:
    - **name**: nombre de la tabla. Si no se usa, se usa el nombre de la clase sin el paquete

- **catalog**: nombre de base de datos
- **schema**: nombre de un conjunto al que pertenece la tabla (por defecto en SQL Server es dbo)
- **uniqueConstraints**: Indican restricciones de unicidad en la tabla

**En Ciclo.java:**

```
14 @Entity
15 @Table(name = "cicloformativo", catalog = "instituto")
16 public class Ciclo implements java.io.Serializable {
17
18     private Integer idCiclo;
19     private String nombre;
20     private Integer horas;
21
```

**En ContactInfo.java:**

```
18 @Entity
19 @Table(name = "contactInfo", catalog = "instituto",
20 uniqueConstraints = { @UniqueConstraint(columnNames = "email"),
21         @UniqueConstraint(columnNames = "profesorId") })
22 public class ContactInfo implements java.io.Serializable {
23
24     private Integer id;
25     private Profesor profesor;
26     private String email;
27     private String tlfMovil;
28
```

## Named queries

Se pueden añadir queries con nombre a nivel de clase:

```
22     /
23   @Entity
24   @Table(name = "profesor", catalog = "instituto")
25   @NamedQuery(name = "findAllProfesores",
26   query = "select p from Profesor p ")
27   public class Profesor implements java.io.Serializable {
28
```

## Anotaciones a nivel de método getter o atributo

Estas anotaciones pueden  utilizar dos tipos de acceso:

- Acceso de atributo (field access): Se sitúan justo encima de los atributos
- Acceso de propiedad (property access): Se sitúan sobre los métodos get de una atributo. Es el tipo de acceso que genera el plugin de Hibernate para Eclipse.

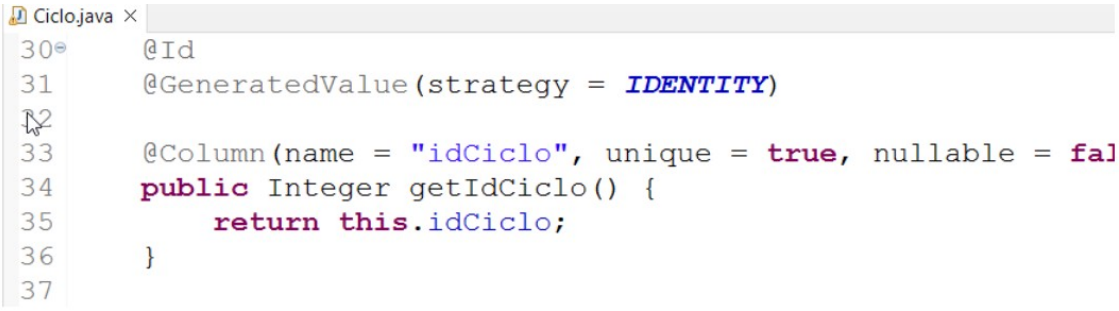
No se recomienda mezclar ambos tipos.

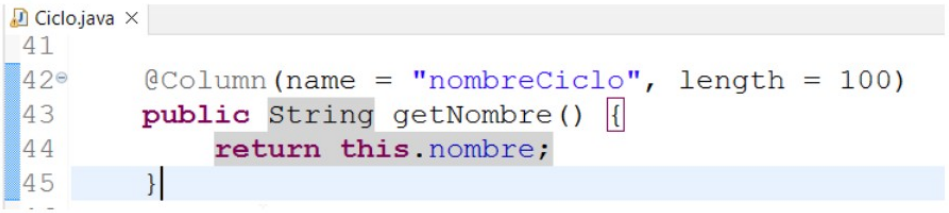
- **@Id**: Indica la propiedad Java que se identifica con **la clave primaria** de la tabla (puede haber una combinación de diferentes campos).  En caso de que sea necesario se puede determinar cuál va a ser la forma para generar la PRIMARY KEY con la anotación @GeneratedValue.
- **@GeneratedValue** indica la estrategia de generación de valores de PK y tienes dos parámetros:
  - **strategy**: Especifica la estrategia de generación de identificadores numéricos. Es de tipo GenerationType (enumerado). Entre otros, es posible especificar los siguientes valores
    - *SEQUENCE*: usar una secuencia: En estas BBDD es posible: (1) generar el identificador (mediante una consulta especial) y (2) a continuación insertar la fila con el identificador generado. Ejemplos: Oracle, SQL Server y PostgreSQL
    - *IDENTITY*: mapear la clave a una columna contador (como estamos usando en SQL Server) Otros SGBD que usan este tipo son: MySQL y  PostgreSQL
    - *AUTO*: Hibernate decide la estrategia en función de la BD usada
  - **generator**: Especifica el nombre del generador que se usará en el caso de la estrategia SEQUENCE
- **@Column:** indica el nombre de la columna asociada al atributo o propiedad Java con el atributo **name**. Si no se usa, por defecto, se usa el nombre del atributo Java. **@Column** puede tener otros atributos como:
  - **nullable**: Indica con true/false si puede contener NULL en BD (default true).
  - **unique**: Genera o nouna restricción UNIQUE con true/false: (default false).
  - **length**: Indica la longitud de caracteres para un string (por defecto 255)

Más atributos en

https://docs.jboss.org/hibernate/stable/annotations/reference/en/html_single/#entity-mapping-property-column

En **Ciclo.java** se utiliza el acceso de propiedad (sobre getters):




## Anotaciones para propiedades de fecha y hora

@Temporal: Debe usarse para especificar la precisión de fechas/horas con el enumerado TemporalType.*:

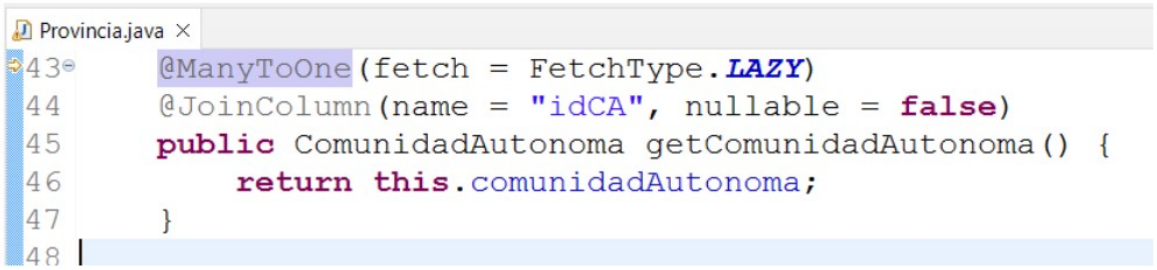
Tiposbasicos.java

```
136         this.stri = stri;
137     }
138
139     @Temporal(TemporalType.DATE)
140     @Column(name = "dateDate", length = 10)
141     public Date getDateDate() {
142         return this.dateDate;
143     }
144
145     public void setDateDate(Date dateDate) {
146         this.dateDate = dateDate;
147     }
148
149     @Temporal(TemporalType.TIME)
150     @Column(name = "timeTime", length = 8)
151     public Date getTimeTime() {
152         return this.timeTime;
153     }
154
155     public void setTimeTime(Date timeTime) {
156         this.timeTime = timeTime;
157     }
158
159     @Temporal(TemporalType.TIMESTAMP)
160     @Column(name = "dateTime2", length = 19)
161     public Date getDateTime2() {
162         return this.dateTime2;
163     }
164
```

## 1:M con anotaciones

- Se utiliza @OneToMany (en lado Uno) o @ManyToOne (en lado Muchos) sobre los atributos/propiedades que definen la relación
  Si la relación es unidireccional, solo se anota el lado que permite navegar hacia el otro
- Hibernate por defecto utiliza una estrategia LAZY para relaciones, excepto en los casos Muchos-a-Uno y Uno-a-Uno (que utiliza una estrategia EAGER). El plugin ha utilizado aquí la estrategia lazy para recuperar la Comunidad Autónoma relacionada con una provincia.
- Se utiliza **@JoinColumn** sobre el atributo/propiedad que mantiene la relación, para especificar la columna que actúa como clave foránea. Se indica que la la columna idCA no puede tener valores NULL

En las relaciones bidireccionales, el lado inverso tiene que usar el elemento **mappedBy** en @OneToOne, @OneToMany y @ManyToMany (No se puede usar en @ManyToOne porque en una relación bidireccional el lado Muchos siempre es el lado propietario)




**mappedBy** especifica el nombre del atributo/propiedad del otro lado (lado propietario) de la relación


## M:N con anotaciones

- Se utiliza @ManyToMany sobre los campos/propiedades que definen la relación
- Si la relación es unidireccional, el lado propietario es el que permite navegar hacia el otro
- Si la relación es bidireccional, se puede elegir cualquier lado como propietario (y el otro será el lado inverso)
- **El lado propietario** debe incluir **@JoinTable** que tiene los siguientes atributos:
  - **name**: nombre de la tabla en la que se mapea la relación
  - **joinColumns**:

claves foráneas (normalmente una) de la tabla en la que se mapea la relación que referencian las claves primarias

de la tabla en la que se mapea la entidad del lado propietario.

  - **inverseJoinColumns**:

claves foráneas (normalmente una) de la tabla en la que se mapea la relación que referencian las claves primarias

de la tabla en la que se mapea la entidad del lado inverso.

El plugin añade tanto en **joinColumns** como en **inverseColumns** además información para indicar que no son columnas que formarán parte de una sentencia UPDATE y que no pueden contener NULL

**El lado inverso** debe incluir **@ManyToMany(mappedBy="propiedad_Java_del_otro_extremo_que_permite_navegar")** indicando la propiedad/atributo Java del otro extremo que permite navegar en la relación.

En la relación M:N existente entre Profesor y Módulo el plugin genera las siguientes anotaciones:

Profesor.java

```
111
112     @ManyToMany(fetch = FetchType.LAZY)
113     @JoinTable(name = "profesormodulo", catalog = "instituto",
114     joinColumns = {
115             @JoinColumn(name = "idProfesor", nullable = false, updatable = false) },
116     inverseJoinColumns = {
117                     @JoinColumn(name = "idModulo", nullable = false, updatable = false) })
118     public Set<Modulo> getModulos() {
119         return this.modulos;
120     }
```

Modulo.java

```
60
61     @ManyToMany(fetch = FetchType.LAZY)
62     @JoinTable(name = "profesormodulo", catalog = "instituto",
63     joinColumns = {
64             @JoinColumn(name = "idModulo", nullable = false, updatable = false) },
65     inverseJoinColumns = {
66                     @JoinColumn(name = "idProfesor", nullable = false, updatable = false) })
67     public Set<Profesor> getProfesors() {
68         return this.profesors;
69     }
70
```

Vemos que ambos extremos se han escogido como propietarios, por lo que habrá que cambiar el código generado.

Si escogemos Profesor como lado propietario, habrá que cambiar en **Modulo.java** para obtener lo siguiente:

Modulo.java

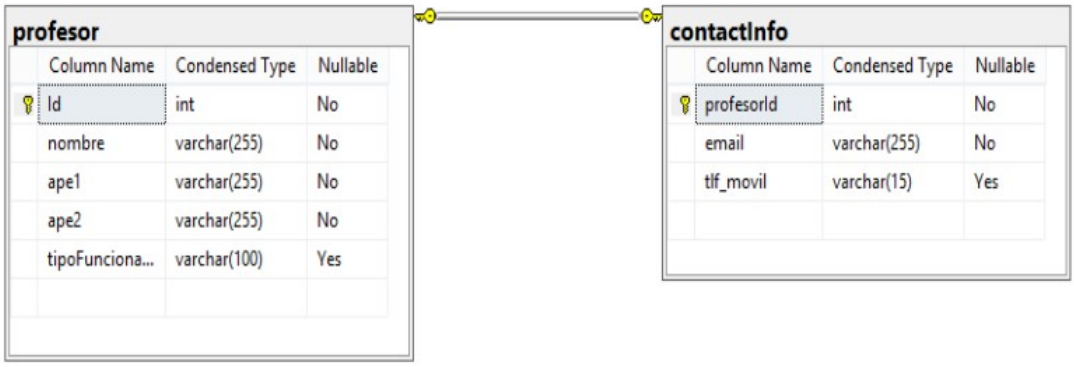
```
61    @ManyToMany(mappedBy = "modulos")
62    public Set<Profesor> getProfesors() {
63        return this.profesors;
64    }
65
66    public void setProfesors(Set<Profesor> profesors) {
67        this.profesors = profesors;
68    }
```

## Relaciones 1:1

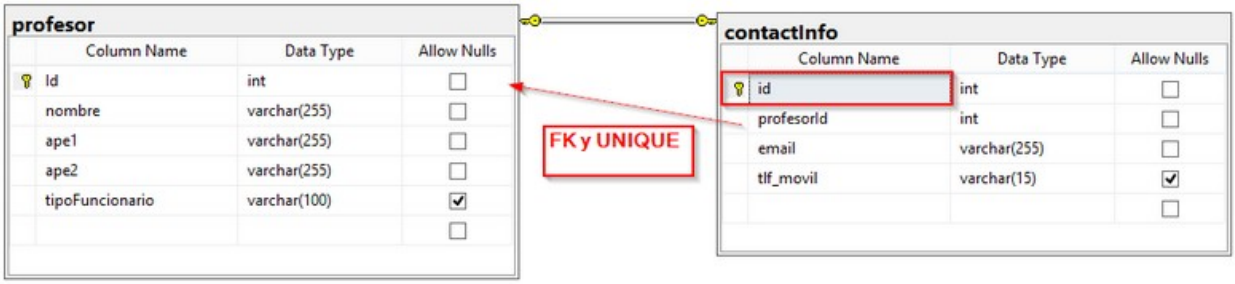
En las relaciones 1:1 puede haber varias posibilidades de implementación en la BD. Aquí veremos 2:

**a) Asociaciones de clave primaria (PK):** La PK de contactInfo tiene el mismo valor que la PK que en profesor

**b)**

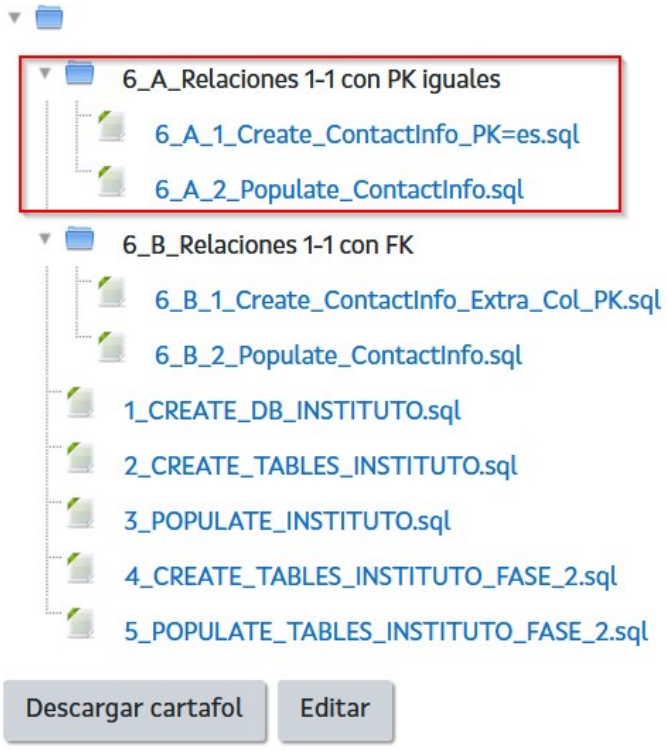



**Asociaciones con clave foránea y esta clave foránea debe ser única:**




## 1:1 con misma PK y anotaciones

Para que el plugin realice esta opción, hay que cambiar la BD con los scripts disponibles en el la carpeta **6_A_Relaciones 1-1 con PK iguales**




El plugin generará esta relación @OneToOne:

Profesor.java

```
100     @OneToOne(fetch = FetchType.LAZY, mappedBy = "profesor")
101     public ContactInfo getContactInfo() {
102         return this.contactInfo;
103     }
104
```

Como **mappedBy** aparece en Profesor, indica que el plugin ha indentificado **ContactInfo** como lado propietario. Esto tiene sentido porque **ContactInfo** es la entidad Java cuya tabla tiene la FK que mantiene la relación, además la FK es la clave primaria.

ContactInfo.java

```
55      @OneToOne(fetch = FetchType.LAZY)
56      @PrimaryKeyJoinColumn
57      /*
58       * The @PrimaryKeyJoinColumn annotation does say that the primary key of the
59       * entity is used as the foreign key value to the associated entity.
60       */
61      public Profesor getProfesor() {
62          return this.profesor;
63      }
```

**@PrimaryKeyJoinColumn** indica que la clave primaria de ContactInfo es además la FK que mantiene la relación con profesor.

El getter de profesorId se anota además:

```
42      @GenericGenerator(name = "genr", strategy = "foreign",
43              parameters = @Parameter(name = "property", value = "profesor
44      @Id
45      @GeneratedValue(generator = "genr")
46      @Column(name = "profesorId", unique = true, nullable = false)
47      public int getProfesorId() {
48          return this.profesorId;
49      }
50
```

Usa un @GenericGenerator con un nombre a escoger (genr en la imagen),

strategy="foreign": La clave primaria es generada por una clave foránea de la entidad indicada por el atributo profesor.

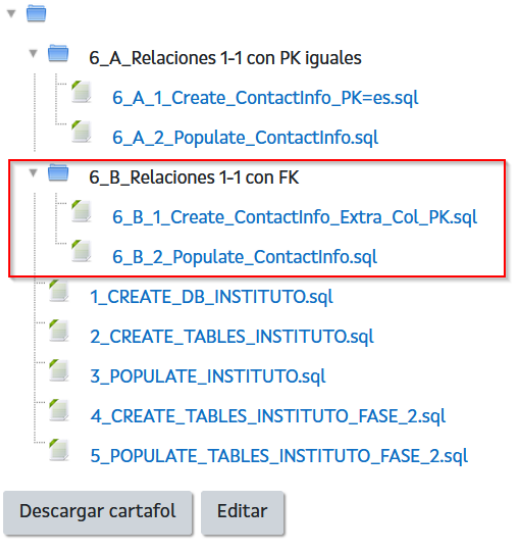
@GenericGenerator se puede sustituir por lo siguiente con @MapIds, que es la opción recomendada:

```java
    // @GenericGenerator(name = "genr", strategy = "foreign",
    // parameters = @Parameter(name = "property", value = "profesor"))
    // @Id
//  @GeneratedValue(generator = "genr")
    @Id
    @Column(name = "profesorId", unique = true, nullable = false)
    public int getProfesorId() {
        return this.profesorId;
    }

    public void setProfesorId(int profesorId) {
        this.profesorId = profesorId;
    }

    @MapsId
    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "profesorId")
    // @PrimaryKeyJoinColumn
    public Profesor getProfesor() {
        return this.profesor;
    }
```

En la documentación, existe una 3ª opción con una tabla intermedia. Más información en la URL: https://docs.jboss.org/hibernate/stable/annotations/reference/en/html_single/#d0e1382

## Relaciones 1:1 con FK y anotaciones

Tendremos que ejecutar los scripts SQL en la subcarpeta 6_Relaciones 1-1 con FK sql de la carpeta: https://www.edu.xunta.gal/fpadistancia/mod/folder/view.php?id=517029




La generación automática del plugin genera la implementación de relaciones 1:1 como si fuese una relación 1:N con restricción de unicidad, así que las anotaciones generadas son como en el caso de 1:N

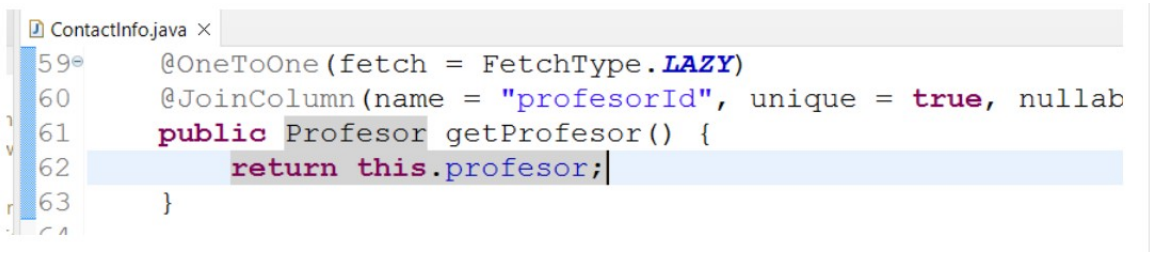
Profesor sería el lado inverso, de ahí el uso de **mappedBy**

El uso de anotaciones nos permite usar @OneToOne en ambos extremos, por lo que tendremos que modificar la anotaciones generadas automáticamente por el plugin:

**En ContactInfo.java:**

- Cambiamos la anotación de @ManyToOne a @OneToOne

- He añadido updatable=false para indicar que la columna profesorId no puede formar parte de los UPDATE pues la información de contacto no modificará a quién pertenece sino datos como el email o el teléfono




**En Profesor.java:**

- Modificaremos el atributo Set<ContactInfo> para que sea un único objeto de ContactInfo.

- Actualizaremos los getters y setters de contactInfo para que no haga referencia a Set<ContactInfo> sino a un solo ContactInfo

- Cambiamos la anotación de @OneToMany a @OneToOne