Ciências ULisboa

Faculdade de Ciências da Universidade de Lisboa

2019/2020

Construção de Sistemas de Software

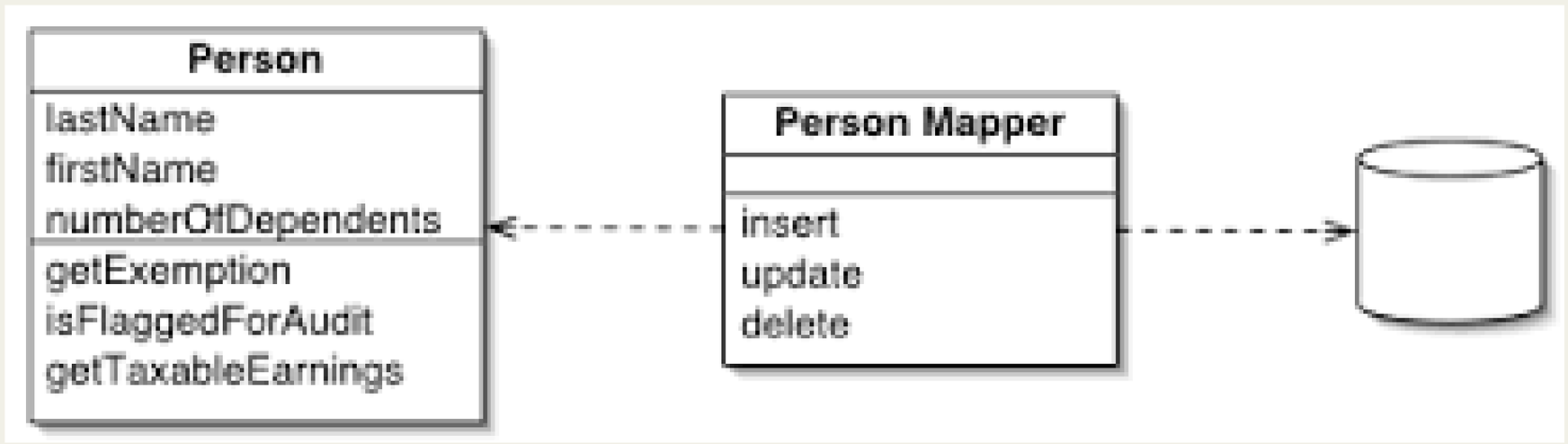
JPA — The Java Persistence API
O/R Structural Patterns

Padrão *Data Mapper*

- Endereça o problema do ***Object-Relational Mapping*** que é especialmente complicado quando a camada de negócio é complexa e foi organizada com o padrão *Domain Model*

  grafo de objetos em memória vs dados guardados em tabelas

- A solução proposta pelo padrão ***Data Mapper*** é ter uma camada com ***mappers*** que movem os dados entre os objetos (em memória) e os registos da base de dados (persistidos), permitindo que cada um dos modelos evolua de forma independente



Padrão *Data Mapper – exemplo*

```
public class SaleMapper {
  // SQL statement: inserts a new sale
  private static final String INSERT_SALE_SQL =
      "INSERT INTO sale (id, date, total, status)
              VALUES (DEFAULT, ?, ?, '" + Sale.OPEN + "')";

  /** Inserts a new sale into the database */
  public static int insert(java.util.Date date) throws … {

      try (PreparedStatement statement =       // get new id
                DataSource.INSTANCE.prepareGetGenKey(INSERT_SALE_SQL)) {
          // set statement arguments
          statement.setDate(1, new java.sql.Date(date.getTime()));
          statement.setDouble(2, 0.0);         // total

          // execute SQL
          statement.executeUpdate();

          // get sale Id generated automatically by the database engine
          try (ResultSet rs = statement.getGeneratedKeys()) {
              rs.next();
              return rs.getInt(1);
          }

  }
```

Padrão *Data Mapper – exemplo*

```
public class CatalogSale {

   /** Creates a new sale, initially its open, and has a total of zero */
   public Sale newSale() throws ApplicationException {

      try {

         // create new entry in the database
         int sale_id = SaleMapper.insert(new Date());

         return SaleMapper.getSaleById(sale_id);

      } catch (PersistenceException e) {
            throw new ApplicationException("Unable to create new sale", e);
      }
   }
```

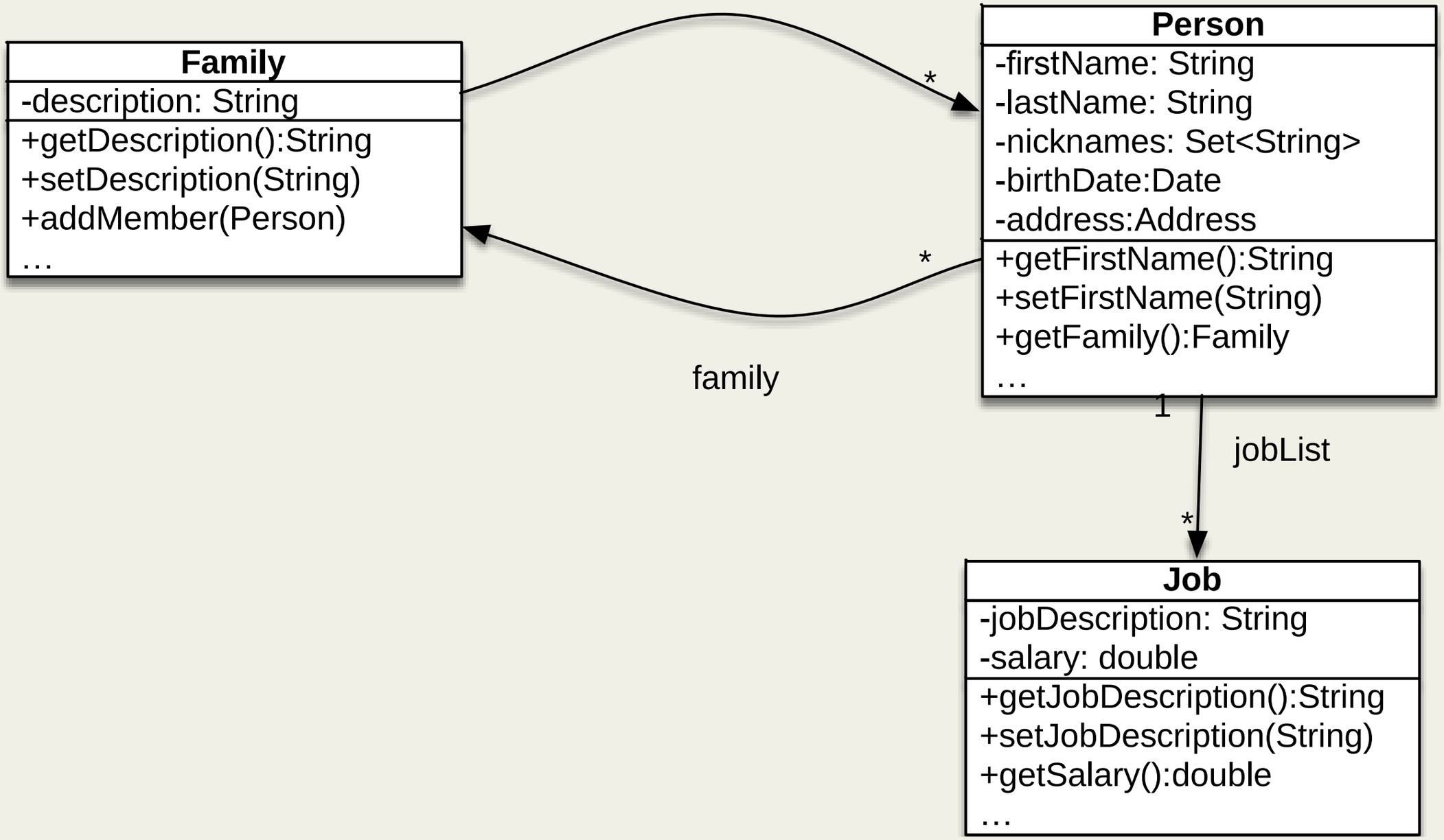
Não existem referências ao `SaleMapper` na classe `Sale`, apenas no catálogo `CatalogSale` que as gere

Java Persistence API

- As implementações da ***Java Persistence API*** (JPA) fornecem uma solução "chave na mão" para o *O/R Mapping*

- Esta API define uma norma para mapear modelos de objetos Java em modelos relacionais
- Principais características:
  - Permite ao programador criar, aceder e modificar as entradas de uma base de dados relacional como se fossem **objetos**
  - O mapeamento entre as classes Java e as tabelas da base de dados é definido através de **metadados** de persistência
  - Estes metadados são usualmente definidos através de **anotações** nas classes Java mas podem também ser definidos em XML
  - Define uma **linguagem de interrogações** tipo SQL — JPQL

Exemplo


Family
-description: String
+getDescription():String
+setDescription(String)
+addMember(Person)
…
*
*
family
Person
-firstName: String
-lastName: String
-nicknames: Set<String>
-birthDate:Date
-address:Address
+getFirstName():String
+setFirstName(String)
+getFamily():Family
…
1
jobList
*
Job
-jobDescription: String
-salary: double
+getJobDescription():String
+setJobDescription(String)
+getSalary():double
…

Entidades

- São a unidade básica de persistência
- Uma entidade é uma classe Java anotada com

  `@javax.persistency.Entity`

  mais um conjunto de **metadados** que definem como as instâncias da classe correspondem às linhas de uma tabela

  – o nome da tabela é por omissão o nome da classe (mas pode ser definido de outra forma com `@Table`)

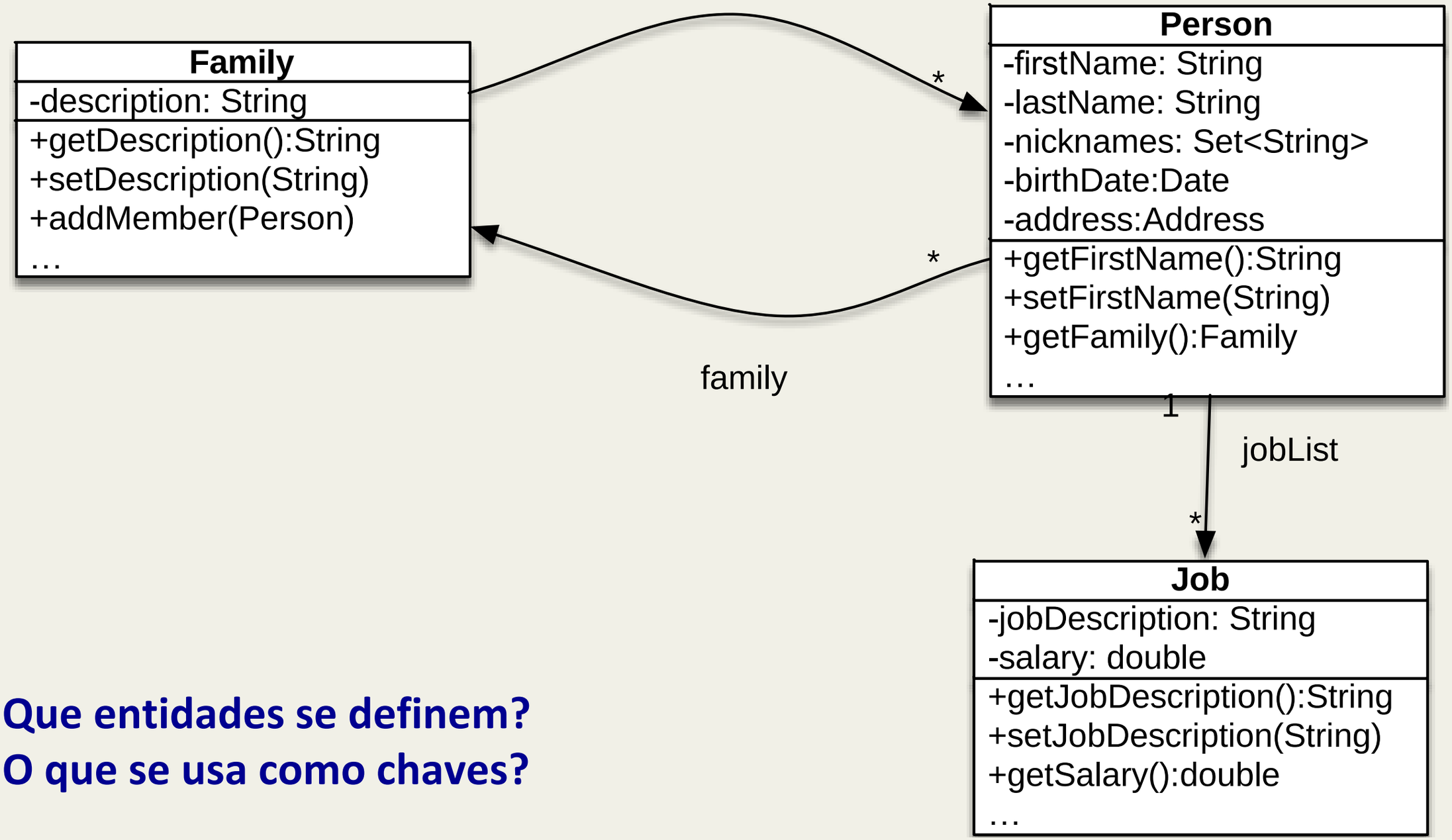
  – tem de ser definido o **atributo identificador**, anotado com `@Id`, que corresponde à coluna que é **chave primária** da tabela

  – podemos dizer que a chave primária é gerada automaticamente com `@GeneratedValue` e, se quisermos, podemos especificar a estratégia de geração

  – todos os atributos são por omissão persistidos (pode-se indicar que não são com `@Transient`)

  – classe tem de ter **construtor sem argumentos** (pode ser *protected*)

Exemplo

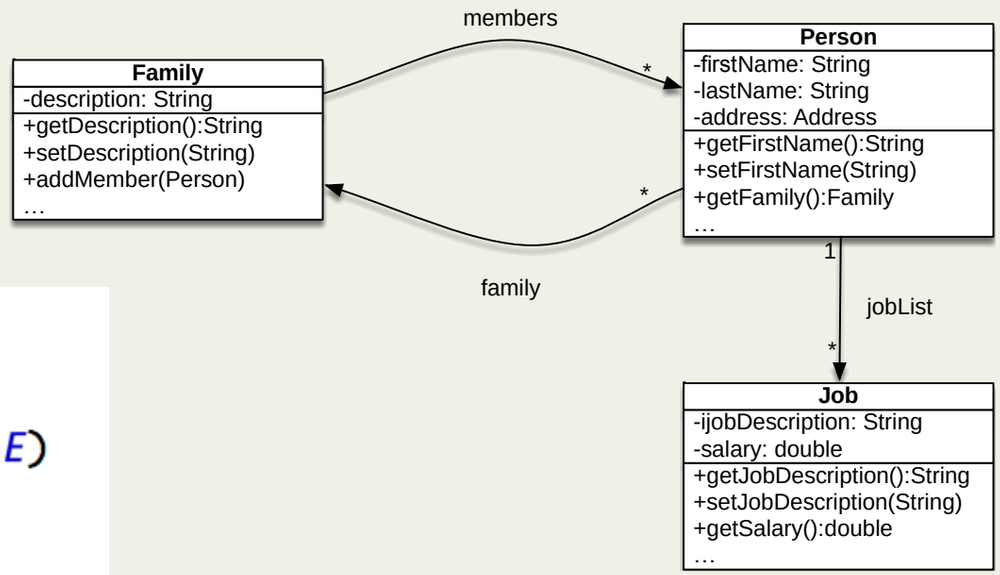


Que entidades se definem?

O que se usa como chaves?

Exemplo

gerada usando uma tabela de geração de chaves




```java
@Entity
public class Family {
    @Id @GeneratedValue(strategy = GenerationType.TABLE)
    private int id;
    private String description;
```

```java
@Entity
public class Person {
    @Id @GeneratedValue(strategy = GenerationType.TABLE)
    private int id;
    private String firstName;
    private String lastName;
```

```java
@Entity
public class Job {
    @Id
    @GeneratedValue(strategy = TABLE)
    private int idJob;
    private double salary;
    private String jobDescr;
```

JPA Details | Console | JUnit | SQL Results | Servers

Attribute '**id**' is mapped as ID.

ID

Column

Name: Default (id)

Table: Default (Job)

Details

Mutable (True)

Type

Converters

Primary Key Gen

Primary key gene

Strategy:

Generator name:

Table Generator

Sequence Generator

Default (Auto)
Auto
Identity
Sequence
✓ Table

JPA De | Consol | JUnit | SQL Re | Server

Type '**Job**' is mapped as entity.

Entity

Table

Name: Default (Job)

Catalog: Default

Schema: Default

Name: Default (Job)

Access: Default (Field)

ID class: <None> Browse...

Caching

Queries

Inheritance

Attribute Overrides

```
@Entity
public class Job {
    @Id
    @GeneratedValue(strategy = TABLE)
    private int idJob;
    private double salary;
    private String jobDescr;
```

JPA Structure

Job

idJob

salary

jobDescr

Mapeamento das associações

- As entidades podem ter diferentes tipos de associações com outras entidades no que diz respeito à sua cardinalidade, nomeadamente:

um para um **um para muitos**

muitos para um **muitos para muitos**

- As associações entre as entidades podem ser **unidirecionais** ou **bidirecionais**:
  - a associação é **unidirecional** se apenas uma classe tem uma referência para a outra
  - é bidirecional se têm ambas

Mapeamento das associações

- Define-se a cardinalidade da associação com anotações

  `@OneToOne`

  `@OneToMany`

  `@ManyToOne`

  `@ManyToMany`

  sobre a referência

- Se a entidade X tem uma referência para a entidade Y, a associação mapeia-se numa chave estrangeira da tabela de X
- No caso de relações bidirecionais que não são `@ManyToOne` é preciso definir qual é o lado que é o dono da relação
- A entidade que não é dona da relação tem que indicar o nome do atributo da outra entidade que se mapeia na chave estrangeira com `@mappedBy`

Mapeamento das associações

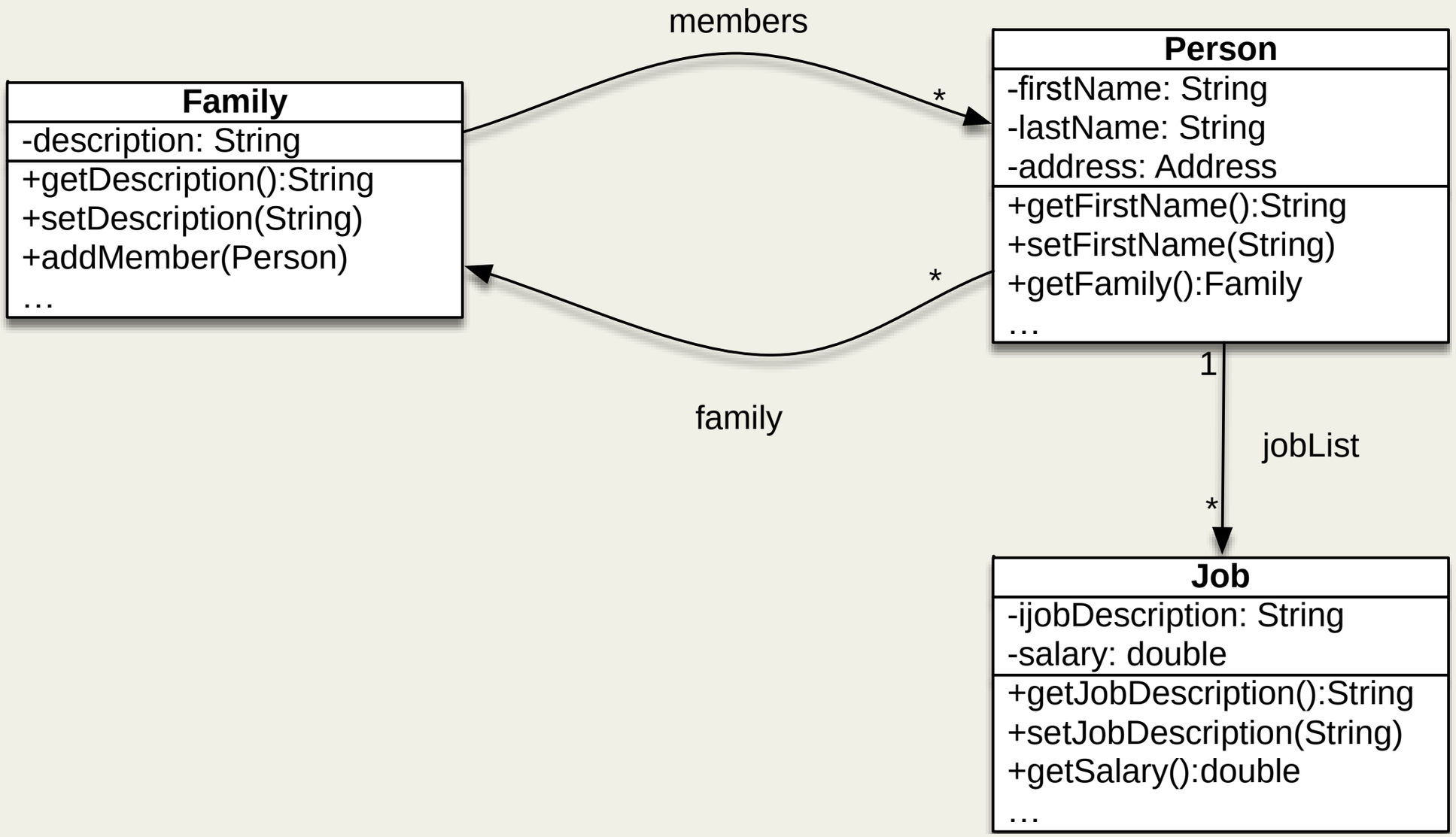
- No caso de associações unidirecionais `@OneToMany` e `@OneToOne` a associação mapeia-se numa tabela separada

- Recorrendo à anotação `@JoinColumn` pode-se especificar uma coluna para juntar este tipo de associação na tabela correspondente à outra classe

```
@JoinColumn(name = "name of foreign key column")
```

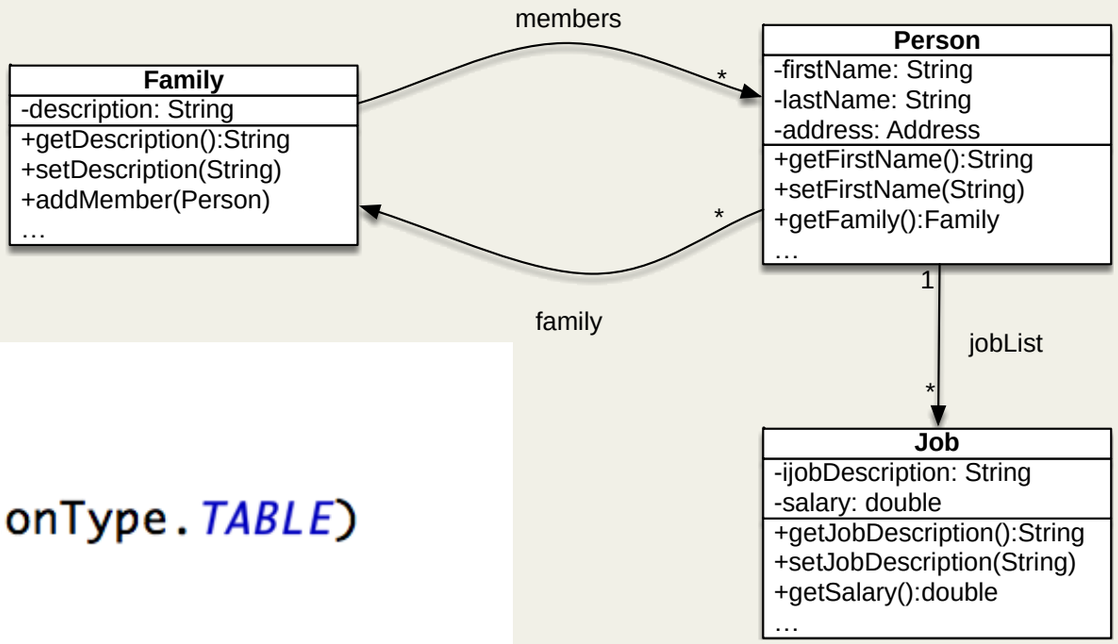
- No caso de associações `@ManyToOne` o dono da relação é sempre a classe que está do lado do *many*

Exemplo



Que associações entre as 3 entidades se devem definir?
Unidirecionais ou bidirecionais? Cardinalidade?

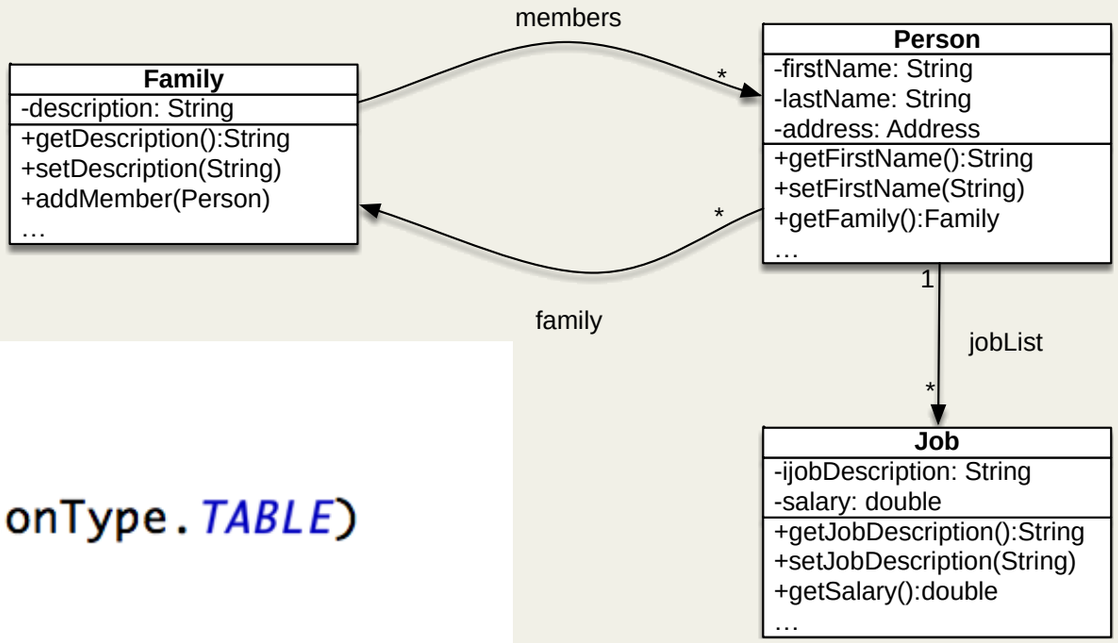
Associação Unidirecional entre *Person* e *Job* (v1)



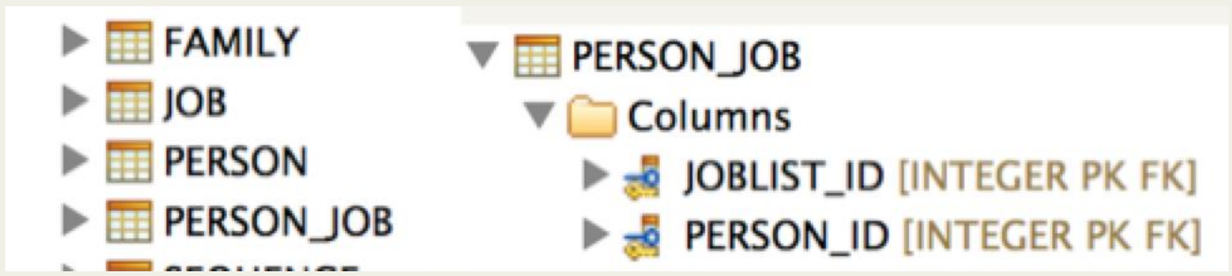

```java
@Entity
public class Person {
    @Id @GeneratedValue(strategy = GenerationType.TABLE)
    private int id;
    private String firstName;
    private String lastName;

    @OneToMany
    private List<Job> jobList = new ArrayList<Job>();
```

```java
@Entity
public class Job {
    @Id
    @GeneratedValue(strategy = TABLE)
    private int idJob;
    private double salary;
    private String jobDescr;
```

Associação Unidirecional entre *Person* e *Job* (v1)

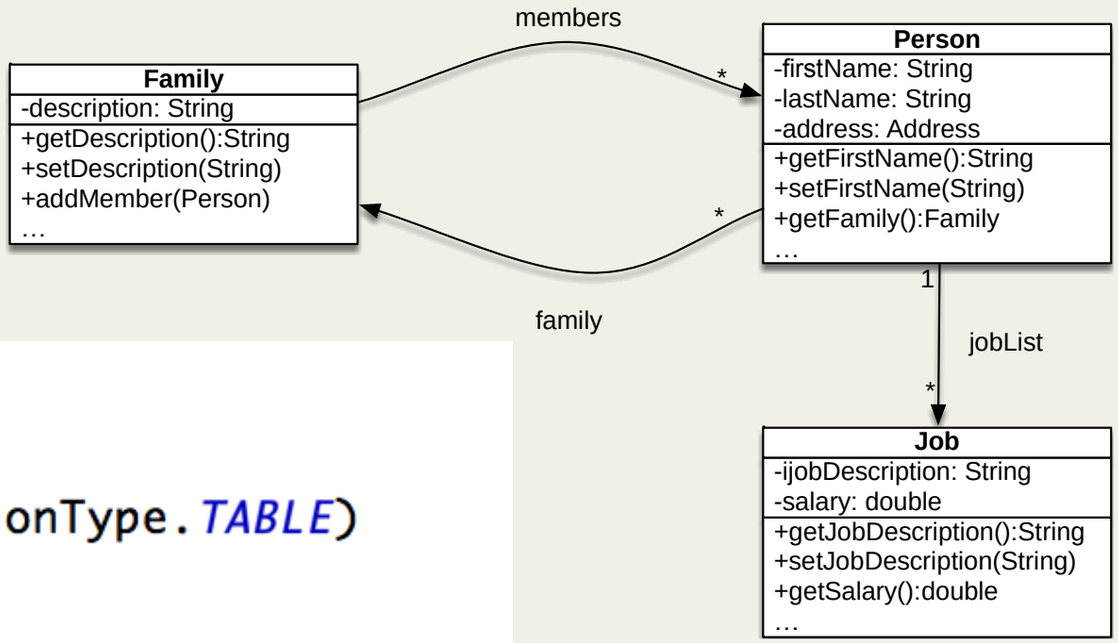



```java
@Entity
public class Person {
    @Id @GeneratedValue(strategy = GenerationType.TABLE)
    private int id;
    private String firstName;
    private String lastName;

    @OneToMany
    private List<Job> jobList = new ArrayList<Job>();
```

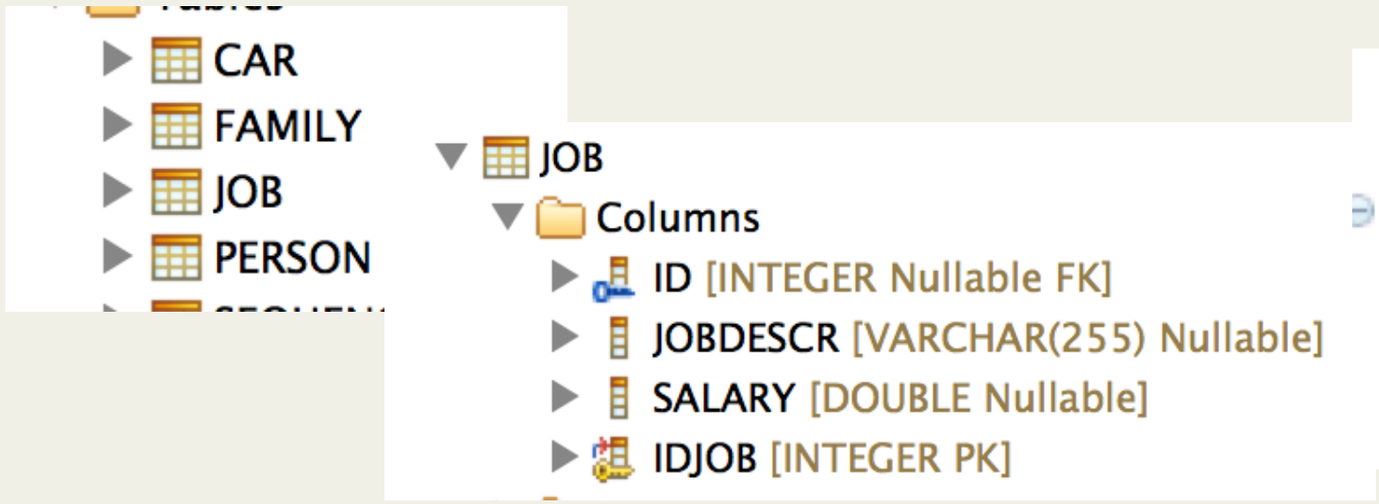



```java
@Entity
public class Job {
    @Id
    @GeneratedValue(strategy = TABLE)
    private int idJob;
    private double salary;
    private String jobDescr;
```

Associação Unidirecional entre *Person* e *Job* (v2)

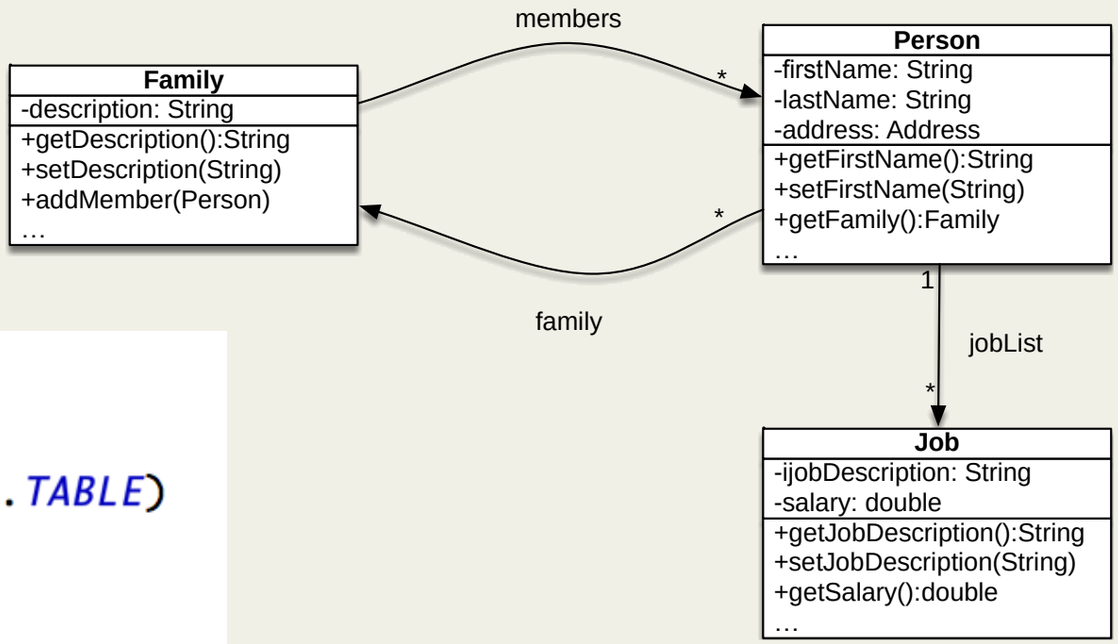



```java
@Entity
public class Person {
    @Id @GeneratedValue(strategy = GenerationType.TABLE)
    private int id;
    private String firstName;
    private String lastName;
    @OneToMany @JoinColumn(name ="ID")
    private List<Job> jobList = new ArrayList<Job>();
```




```java
@Entity
public class Job {
    @Id
    @GeneratedValue(strategy = TABLE)
    private int idJob;
    private double salary;
    private String jobDescr;
```

Associação Bidirecional entre *Person* e *Family*



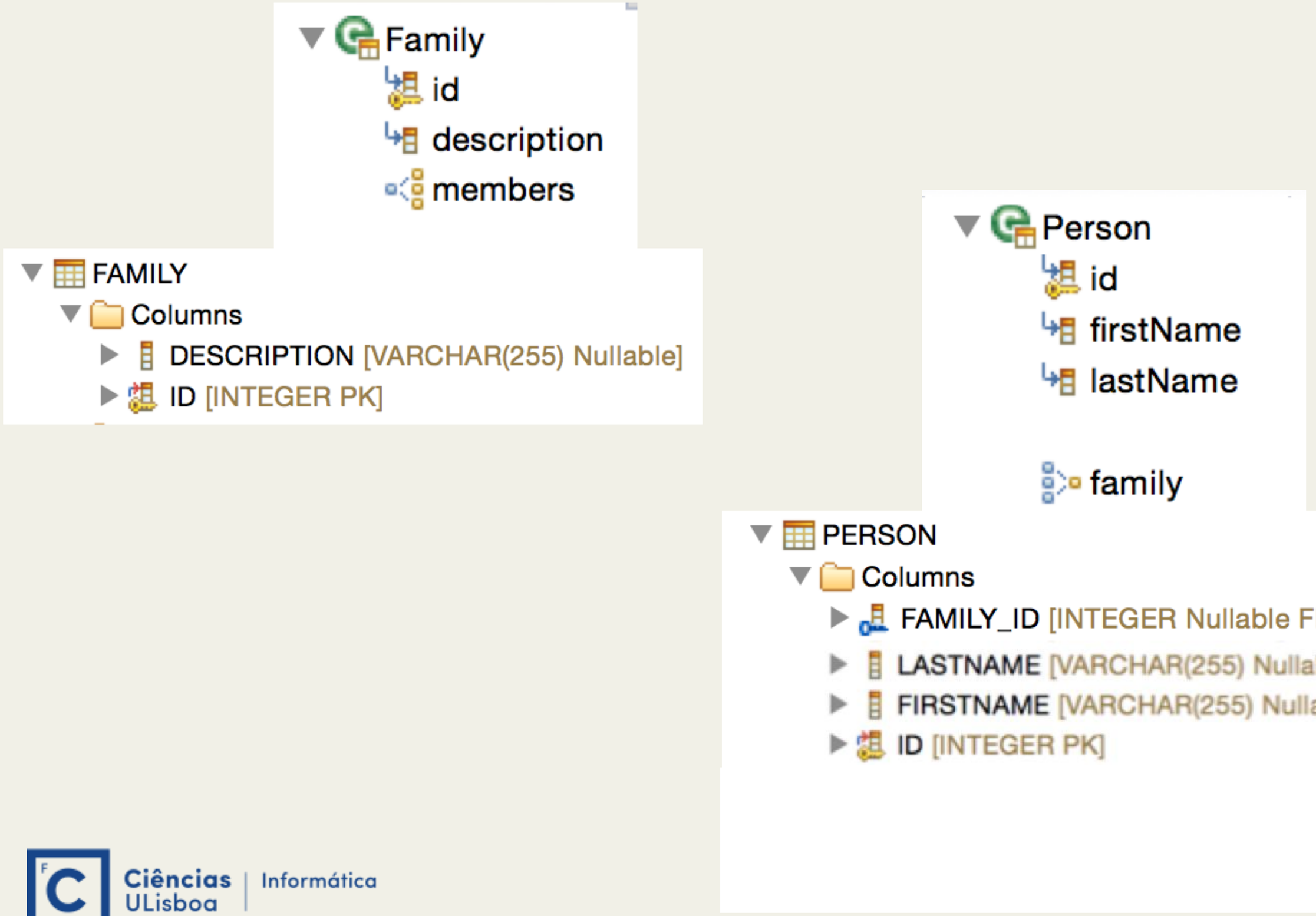

```java
@Entity
public class Family {
    @Id @GeneratedValue(strategy = GenerationType.TABLE)
    private int id;
    private String description;
```

```java
    @OneToMany(mappedBy = "family")
    private List<Person> members = new ArrayList<Person>();
```

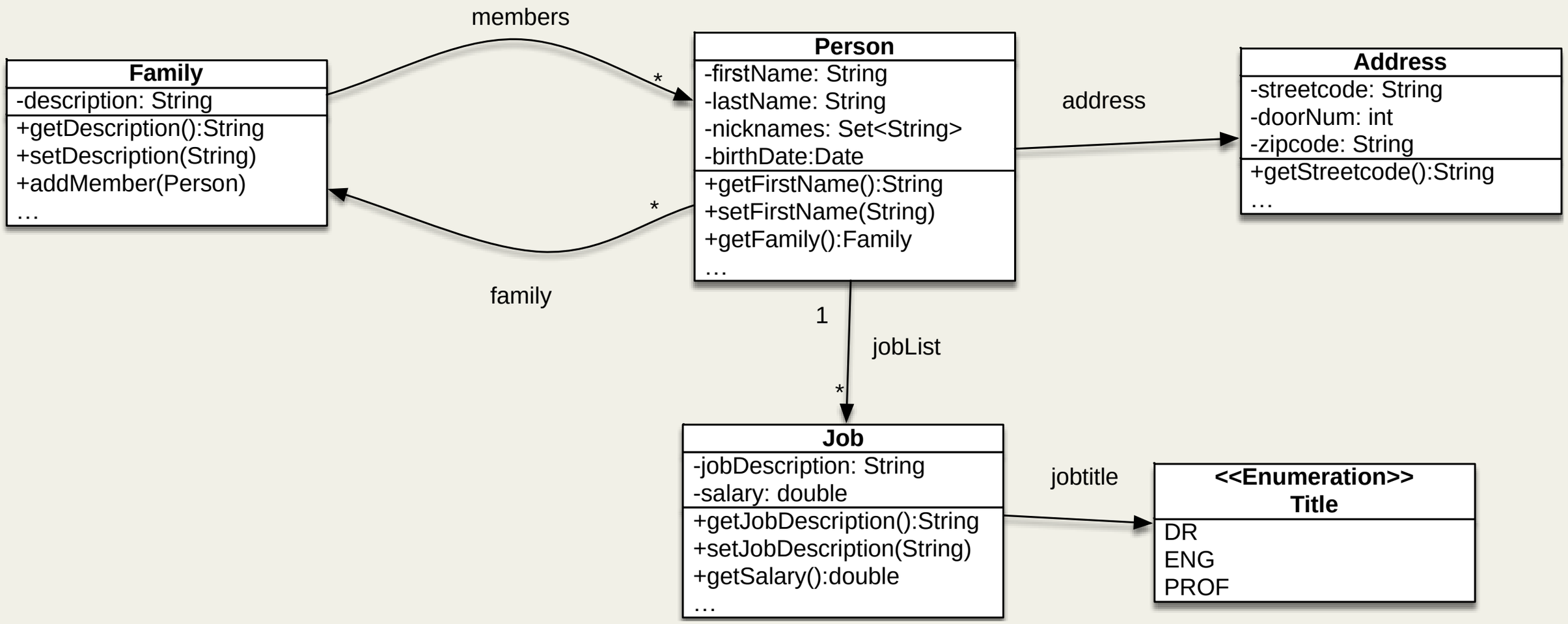
```java
@Entity
public class Person {
    @Id @GeneratedValue(strategy = GenerationType.TABLE)
    private int id;
    private String firstName;
    private String lastName;
```

```java
    @ManyToOne
    private Family family;
```

Associação Bidirecional entre *Person* e *Family*



Exemplo (versão do enunciado)



E o que fazer com atributo *address* do tipo *Address* em *Person ?*

E com o atributo *nicknames ?*

Mapeamento de atributos “compostos”

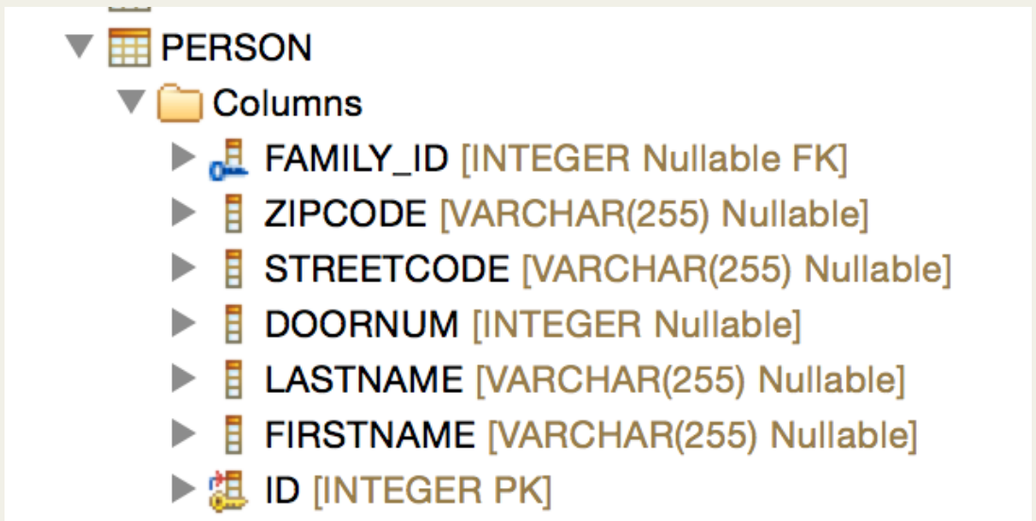
- Mapeamento de um atributo composto `t` da classe `T`
  - Usar `@Embeddable` para anotar a classe `T`
  - Usar `@Embedded` para anotar atributo `t`
  - A classe `T` é mapeada na tabela da classe dona do atributo `t`
  - Os objetos do tipo `T` não têm identidade própria, ou seja:
    - nunca serão partilhados entre vários objetos entidade
    - podem apenas ser procurados usando a chave da entidade dona

- Mapeamento de um atributo `t` do tipo `Collection<T>` onde `T` é básico ou *embeddable*
  - Usar `@ElementCollection` para anotar o atributo `t`
  - Associação mapeada numa tabela separada

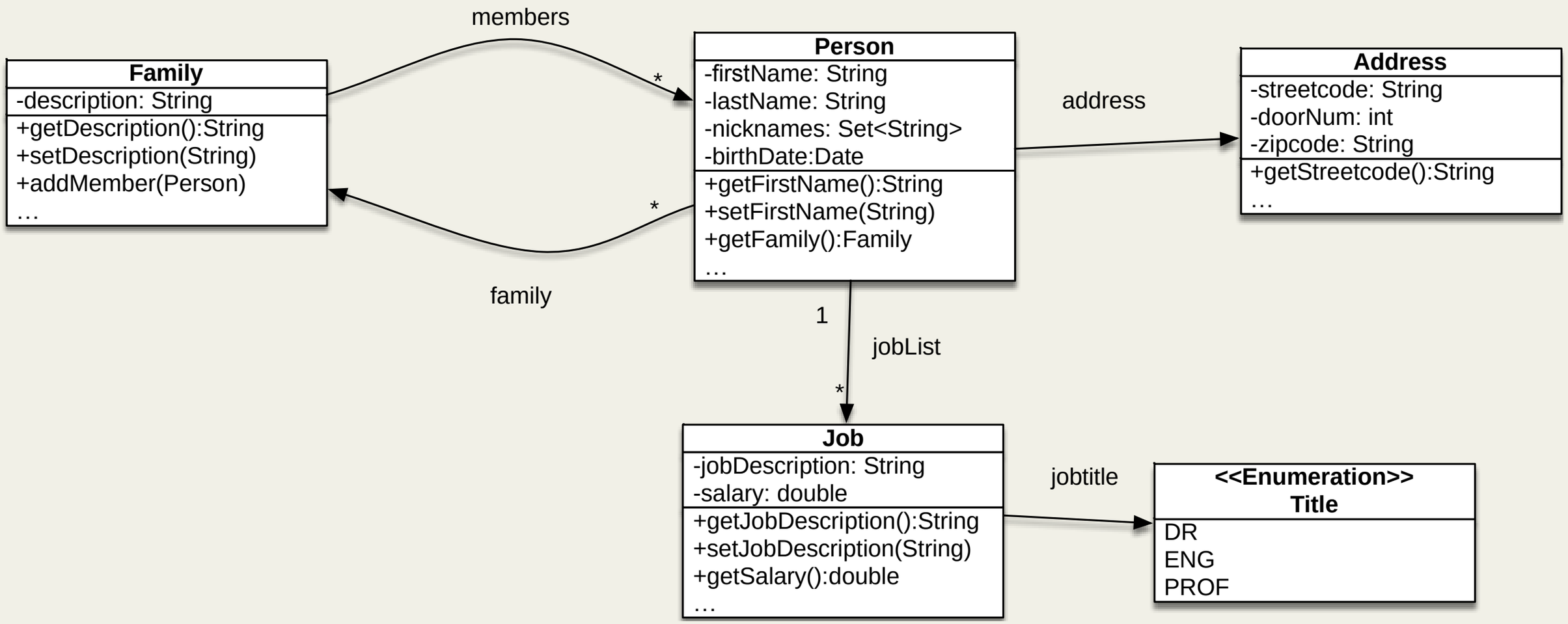
Atributos compostos de *Person*

```java
@Entity
public class Person {

    @Id @GeneratedValue(strategy = GenerationType.TABLE)
    private int id;

    private String firstName;
    private String lastName;

    @Embedded
    private  Address address;
    @ElementCollection
    private Set<String> nicknames = new HashSet<String>();

    @ManyToOne
    private Family family;
```

```java
@Embeddable
public class Address {

    private String streetCode;
    private int doorNum;
    private String zipcode;
```



Exemplo (versão do enunciado)

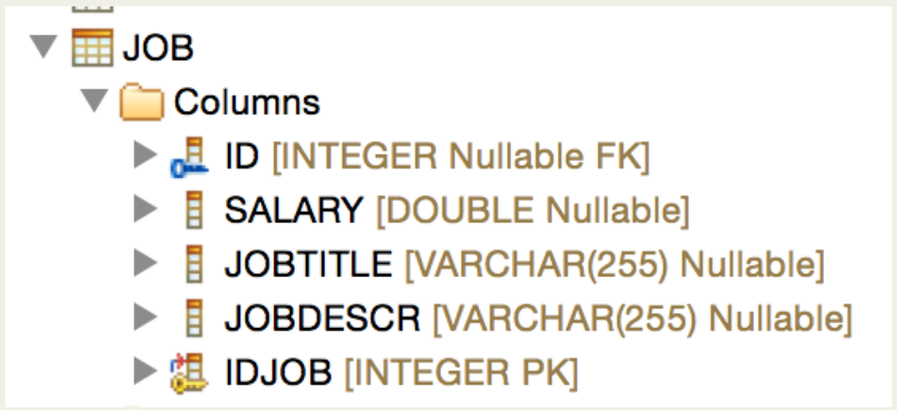


E o que fazer com atributo *jobTitle* do tipo *Title* em *Job?*

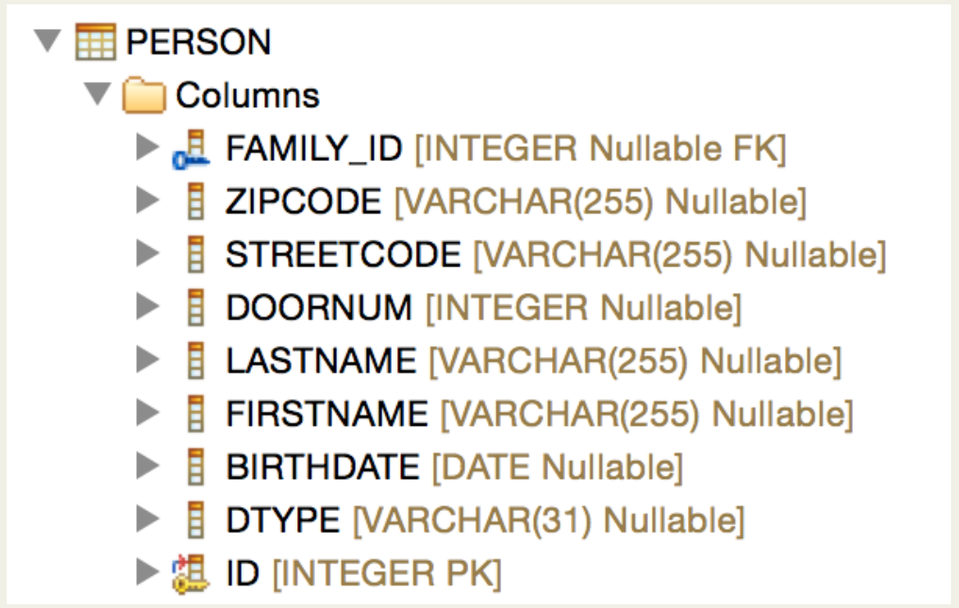
E com o atributo *birthDate* em *Person?*

Atributos com 1) tipo enumerado e 2) Date

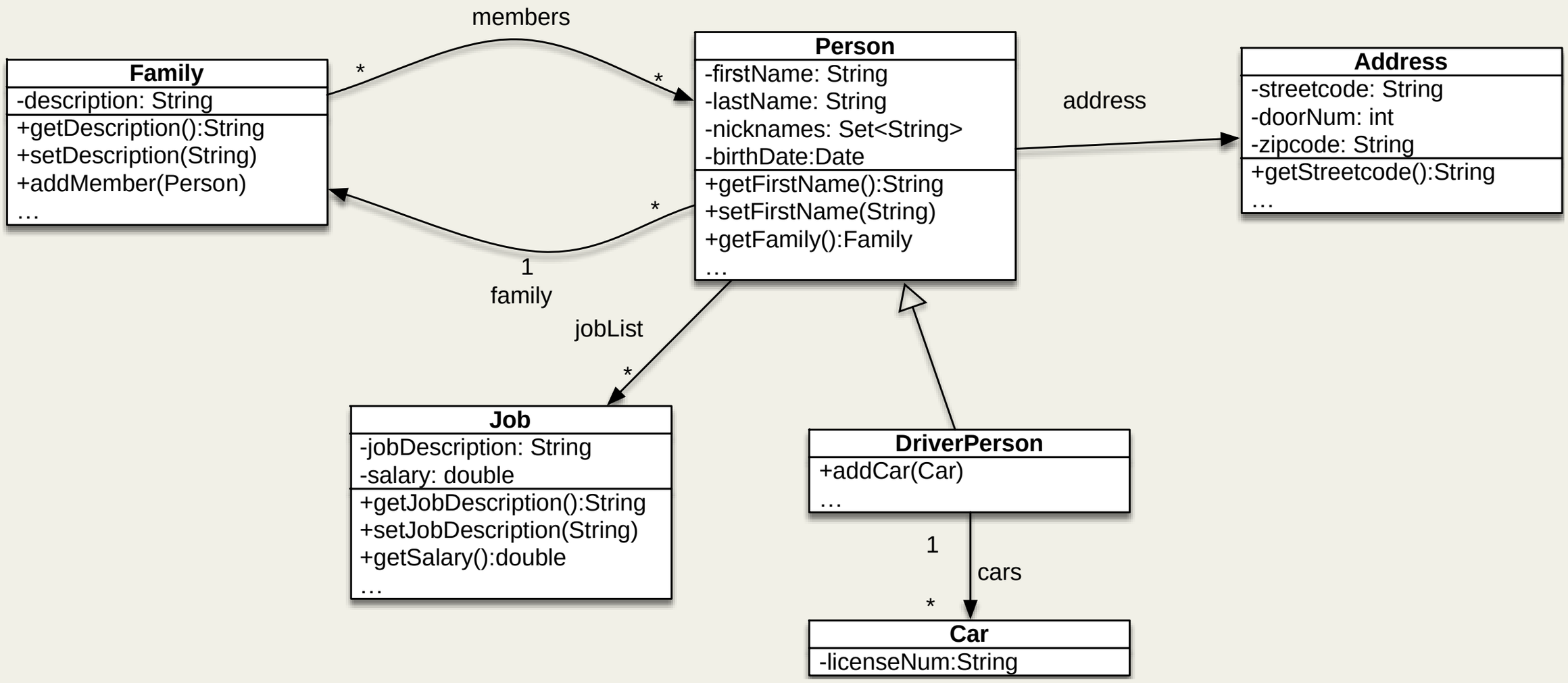
```java
@Entity
public class Job {
    @Id
    @GeneratedValue(strategy = TABLE)
    private int idJob;

    private double salary;

    private String jobDescr;

    @Enumerated(EnumType.STRING)
    private Title jobTitle;
```




```java
@Entity
@Inheritance(strategy=SINGLE_TABLE)
public class Person {
    @Id @GeneratedValue(strategy = GenerationType.TABLE)
    private int id;
    private String firstName;
    private String lastName;

    @Embedded
    private  Address adress;

    @ManyToOne
    private Family family;

    @Temporal(TemporalType.DATE)
    private Date birthDate;
```



Exemplo (estendido)



Que entidades adicionais se definem?

Como se codifica a herança?

Mapeamento da Herança

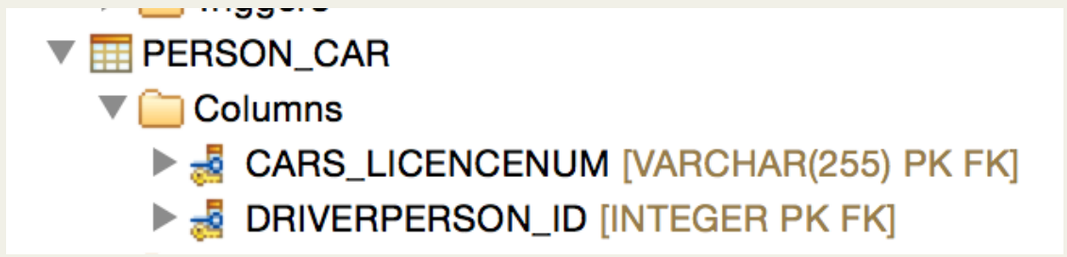
- A forma como o JPA mapeia as entidades herdadas é definido pela anotação `@Inheritance` na classe raiz da hierarquia

- Há as seguintes estratégias:
  - **SINGLE_TABLE**: Uma única tabela por hierarquia de classes
  - **TABLE_PER_CLASS**: Uma tabela por entidade concreta
  - **JOINED**: Atributos que são específicos a uma subclasse são mapeados numa tabela diferente daquela em que são mapeados os atributos da classe pai.
    - A superclasse é uma entidade, mas classes intermédias podem não ser representadas se as anotarmos com `@MappedSuperClass`, passando os atributos para as subclasses
- A estratégia deve ser escolhida tendo em conta as características da hieraquia (influencia o desempenho)

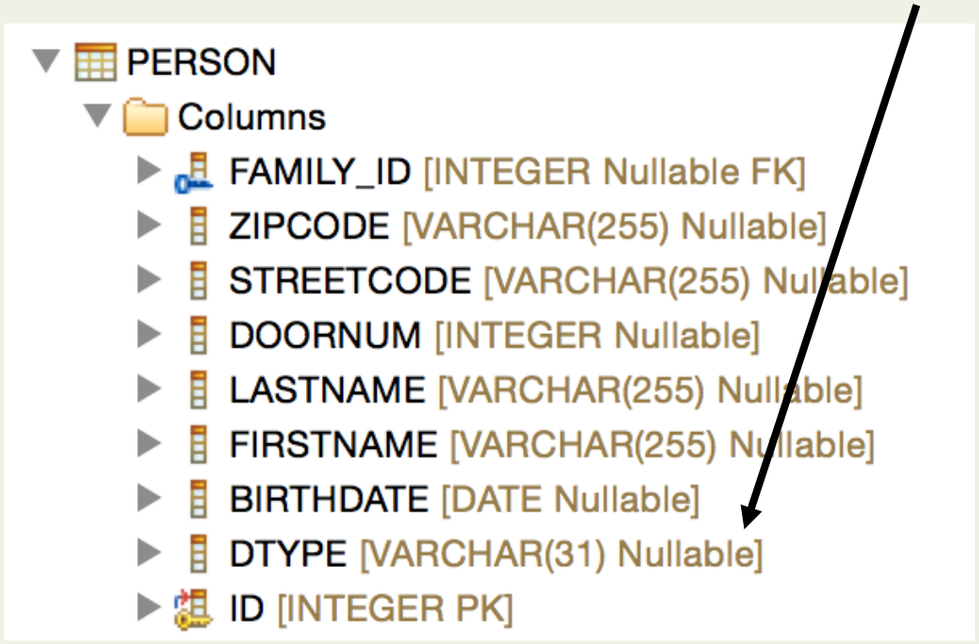
Herança de *Person*

```java
@Entity
@Inheritance(strategy=SINGLE_TABLE)
public class Person {
    @Id @GeneratedValue(strategy = GenerationType.TABLE)
    private int id;
    private String firstName;
    private String lastName;
    @Embedded
    private  Address adress;

    @ManyToOne
    private Family family;

    @ElementCollection
    private Set<String> nicknames;
```

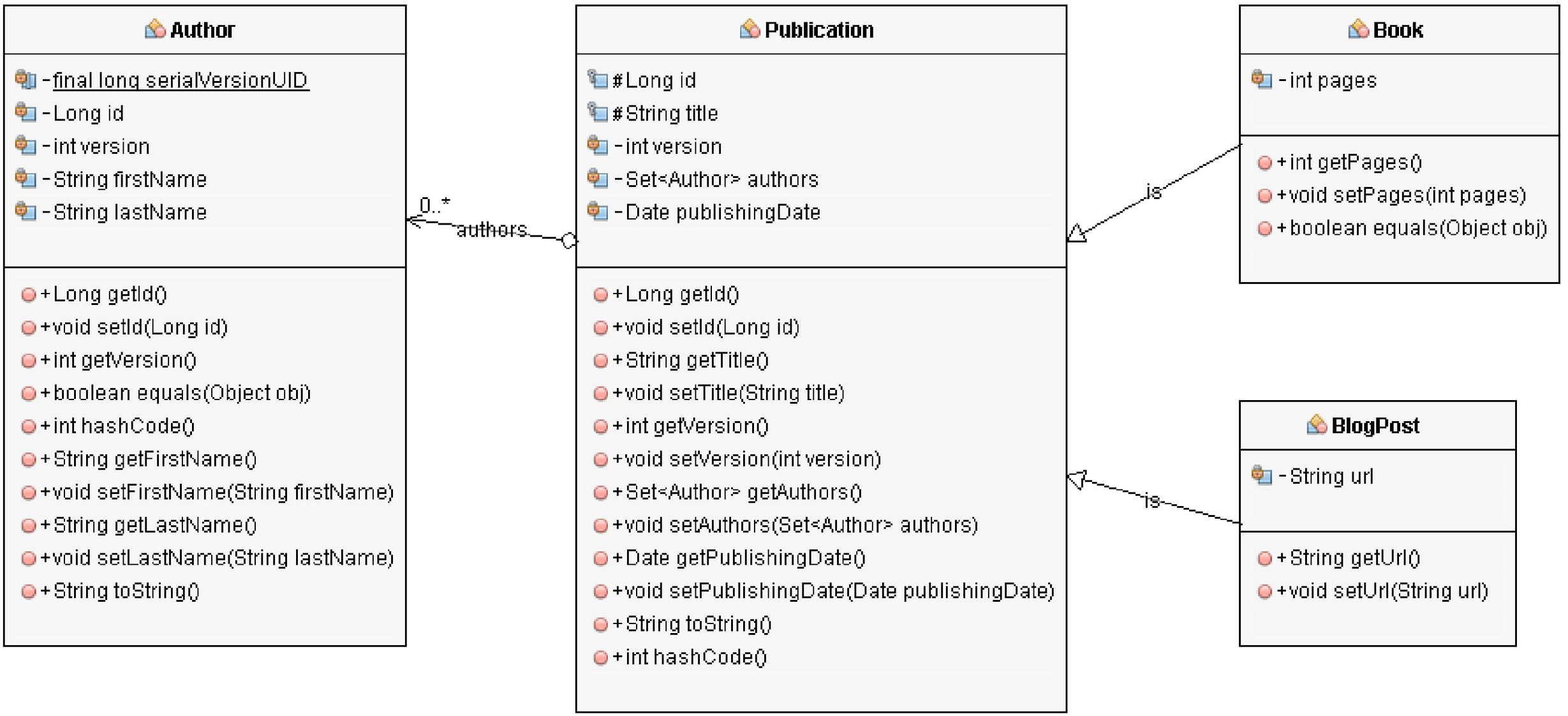
```java
@Entity
public class DriverPerson extends Person {

    @OneToMany
    private Set<Car> cars;

    public DriverPerson(){
```



Identifica a que classe pertence

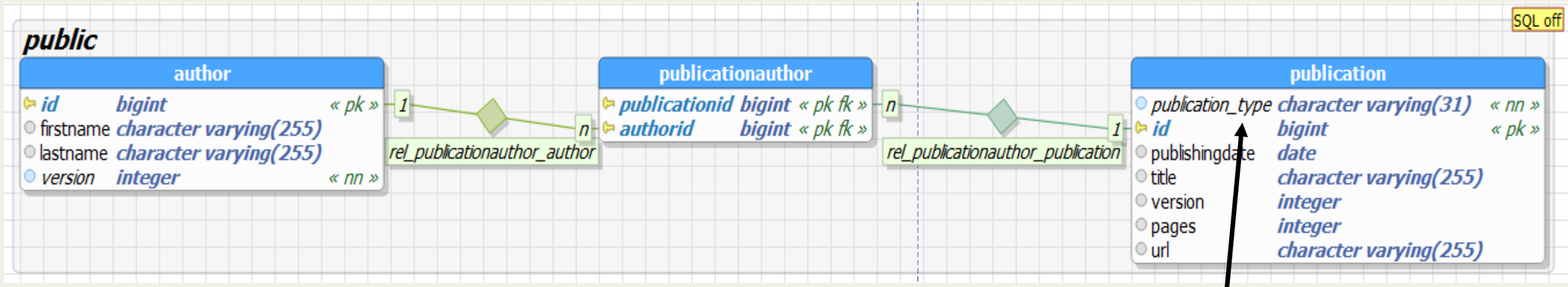


Exemplo


Author
-final long serialVersionUID
-Long id
-int version
-String firstName
-String lastName
+Long getId()
+void setId(Long id)
+int getVersion()
+boolean equals(Object obj)
+int hashCode()
+String getFirstName()
+void setFirstName(String firstName)
+String getLastName()
+void setLastName(String lastName)
+String toString()
0..*
authors
Publication
#Long id
#String title
-int version
-Set<Author> authors
-Date publishingDate
+Long getId()
+void setId(Long id)
+String getTitle()
+void setTitle(String title)
+int getVersion()
+void setVersion(int version)
+Set<Author> getAuthors()
+void setAuthors(Set<Author> authors)
+Date getPublishingDate()
+void setPublishingDate(Date publishingDate)
+String toString()
+int hashCode()
is
Book
-int pages
+int getPages()
+void setPages(int pages)
+boolean equals(Object obj)
is
BlogPost
-String url
+String getUrl()
+void setUrl(String url)

Estratégia Single Table

- Todos os atributos de todas as classes da hierarquia residem na mesma tabela.
- Facilidade de procura, mas gasto de memória em campos vazios

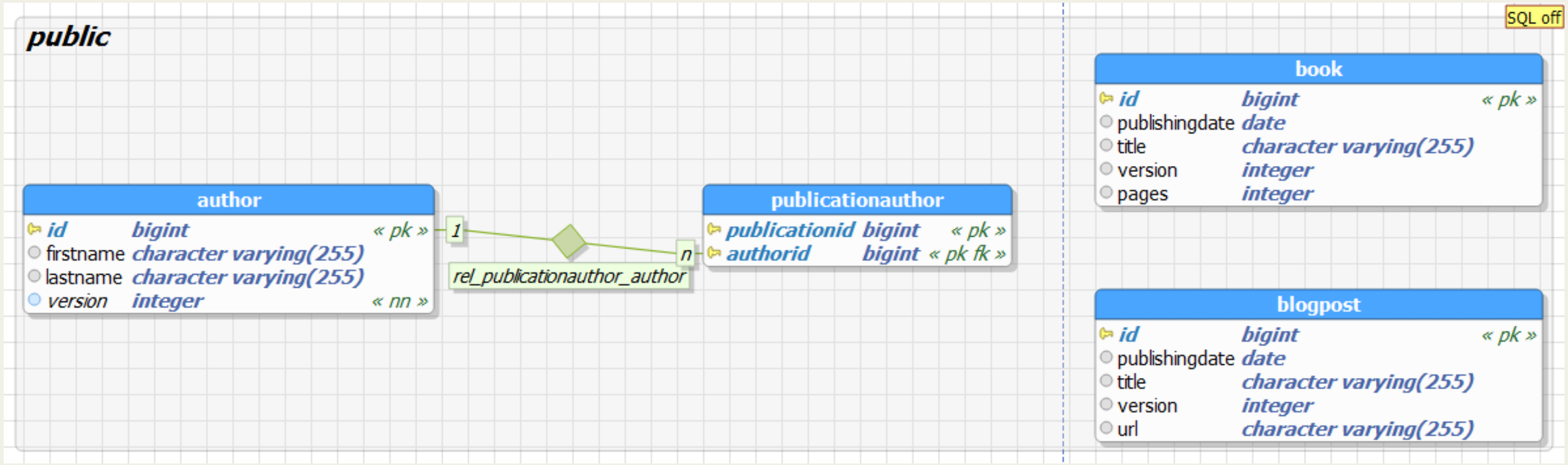



```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "publication_type")
public abstract class Publication { …
```

Identifica a que classe pertence um registo da tabela
Nome default DTYPE

Estratégia Table per Class

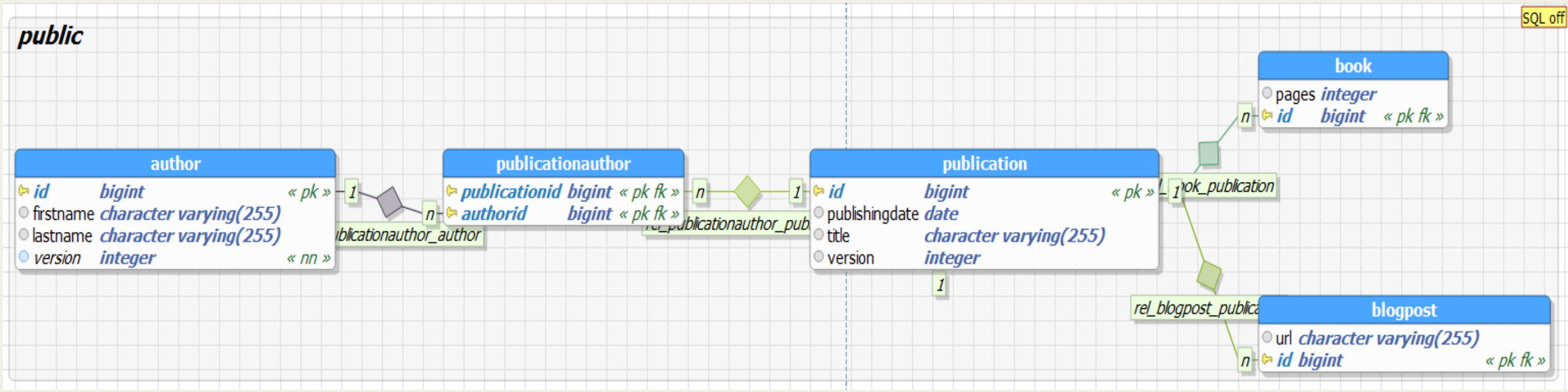
- Cada classe concreta é uma tabela
- Mais económico em termos de memória
- Procuras mais complexas




```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Publication { …
```

Estratégia Joined

- Similar à tabela por classe (concreta)
- A superclasse também é incluída.
- Uma *query* requer um *join* de duas tabelas para recolher todos os dados
- Melhor para queries polimórficas (que retornam dados também das subclasses)




```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Publication { …
```