Programmation Orientée Objet
IG2I, LE4-SI – 2017-2018

TP 1 – JPA : Hôpital

Diego Cattaruzza, Maxime Ogier

Contexte et objectif du TP

Ce TP propose d'utiliser le concept de *mapping* objet-relationnel afin de mettre en place le système d'information des médecins et des malades d'un hôpital. Il s'agit d'utiliser l'API JPA (*Java Persistence API*), grâce à laquelle nous serons capables de créer la base de données (le modèle relationnel) à partir du modèle objet développé en Java.

Ce TP permet d'illustrer les notions suivantes :

- le *mapping* objet-relationnel ;
- l'API JPA (*Java Persistence API*) ;
- le *mapping* des relations d'association.

Bonnes pratiques à adopter

N'oubliez pas de tester votre code au fur et à mesure de votre avancement. Par ailleurs, il vous est très fortement recommandé d'utiliser la Javadoc pour comprendre le fonctionnement des méthodes que vous appelez. Il est également important que votre propre code soit commenté au format Javadoc (commentaire commençant par "`/**`" au-dessus des définitions des attributs et méthodes). Ceci vous permet (1) de spécifier rigoureusement le comportement d'une méthode au moment où vous l'implémentez, (2) de savoir ce que fait une méthode sans avoir besoin de regarder son code (qui d'ailleurs peut ne pas être accessible), et donc (3) de faciliter le partage et la réusabilité du code.

1 Introduction

Vous êtes chargé de mettre en place le système d'informations pour gérer les médecins et les malades d'un hôpital.

Les médecins et les malades sont tous des personnes caractérisées par un identifiant, un nom et un prénom. Chaque malade possède une adresse (rue, numéro de rue et ville). Les médecins sont caractérisés par leur salaire.

Par ailleurs, il existe une hiérarchie interne qui fait qu'un médecin peut gérer un groupe de médecins subalternes, et il est stipulé qu'un médecin a au maximum un seul chef.

L'hôpital est organisé en services. Chaque service est caractérisé par un identifiant, un nom et une localisation. Un service regroupe un ou plusieurs médecins et est dirigé par au plus un médecin. Chaque médecin appartient à exactement un service, mais il est possible qu'un médecin dirige plusieurs services (si il a de très bonnes compétences de management).

Par ailleurs, au niveau médical, les médecins forment des équipes chirurgicales lors des opérations. Chaque équipe est caractérisée par un identifiant et un nom. Un médecin peut participer à plusieurs équipes, avec à chaque participation une seule fonction spécifique (par exemple : responsable, adjoint, interne ou élève).

Le responsable du service informatique de l'hôpital propose le diagramme de classe de la Figure 1 qui reprend l'ensemble de ces éléments.

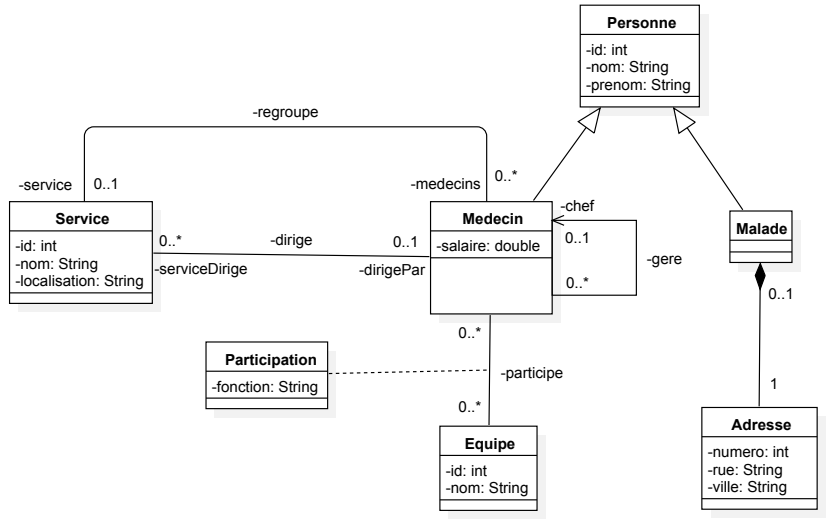

Figure 1: Diagramme de classe pour la gestion du système d'informations de l'hôpital.

Par rapport au contexte général du cours, nous nous intéressons ici à la couche stockage des informations. Nous souhaitons à présent développer la base de données pour la gestion de l'hôpital ainsi que le modèle objet correspondant dans une application Java EE.

Vous avez étudié en cours de POO3 l'API **JDBC** qui permet d'accéder à une base de données depuis une application Java. Cependant, il vous faudrait

concevoir d'une part le modèle relationnel pour la base de données, et d'autre part le modèle objet pour l'application. Les objets dans le modèle relationnel sont représentés sous une forme tabulaire, alors que dans le modèle objet ils sont représentés par un graphe interconnecté d'objets. Lorsque l'on souhaite stocker ou récupérer un objet dans une base de données relationnelle, certaines incompatibilités peuvent survenir pour les raisons suivantes :

- les modèles objets ont plus de granularité que les modèles relationnels ;
- l'héritage n'est pas supporté de base dans les modèles relationnels ;
- les objets ne possèdent pas d'identifiant unique, contrairement au modèle relationnel ;
- la relation entre les entités se traduit de manière différente dans les deux modèles ;
- les modèles relationnels ne sont pas capables de gérer directement les associations à cardinalités multiples.

Par ailleurs, l'utilisation de **JDBC** pour faire des requêtes sur la base oblige aussi à constamment faire la traduction entre modèle objet et modèle relationnel, ce qui est fastidieux !

Ainsi, il est avantageux dans ces cas là d'utiliser ce qu'on appelle le *mapping* objet-relationnel. Il s'agit d'une technique de programmation qui consiste à réaliser la correspondance entre le modèle de données relationnel et le modèle objet de la façon la plus facile possible. Il s'agit donc d'une couche qui va interagir entre l'application et la base de données. Ainsi, lors du développement de l'application, il n'est pas nécessaire de connaître l'ensemble des tables et des champs de la base de données, et il n'est plus obligatoire d'utiliser *SQL* pour faire des requêtes sur la base de données.

Dans ce TP, nous étudions l'utilisation de JPA (*Java Persistance API*) pour faire le *mapping* objet-relationnel dans une application Java. Ce *mapping* permet d'assurer la transformation d'objets vers la base de données et vice versa que cela soit pour des lectures ou des mises à jour (création, modification ou suppression). Cette API peut être mise en œuvre dans des applications Java SE ou Java EE. L'utilisation de JPA ne requiert aucune ligne de code mettant en œuvre l'API JDBC : JPA propose un langage d'interrogation similaire à SQL mais utilisant des objets plutôt que des entités relationnelles de la base de données.

JPA repose sur des entités qui sont de simples POJOs (*Plain Old Java Object*) annotés et sur un gestionnaire de ces entités (*entity manager*) qui propose des fonctionnalités pour les manipuler (ajout, modification, suppression, recherche). Ce gestionnaire est responsable de la gestion de l'état des entités et de leur persistance dans la base de données.

2 Création de la première entité : les services

2.1 Création de la base de données

Question 1. Créez un nouveau projet (de type *Java Application*) avec Netbeans. Créez deux paquetages : un paquetage **modele** qui contiendra les classes pour la modélisation de l'hôpital, et un paquetage **tests** dans lequel nous ajouterons des classes de tests. Créez une nouvelle base de données (qui ne contient rien pour le moment). Indiquez bien un nom d'utilisateur et un mot de passe lors de la création de la base. Vous pouvez vous reporter à la Section 4.1 si vous ne vous rappelez plus comment faire ... Pensez également à importer le pilote JDBC dans le projet, comme indiqué dans la Section 4.2.

2.2 Généralités sur la persistance des objets avec JPA

2.2.1 Notion d'entité

Une entité est un objet qui peut être rendu persistant (l'état de l'objet peut être stocké dans une base de données). Typiquement, une entité représente une table dans le modèle relationnel, et chaque instance d'entité correspond à une ligne dans cette table. L'état persistant d'une entité est représenté à travers des champs ou propriétés persistants. Ces champs ou propriétés utilisent des annotations afin de réaliser le *mapping* objet-relationnel.

En fait, une entité est une classe qui doit respecter quelques spécifications :

- elle doit être annotée avec les annotations du paquetage **javax.persistence.Entity** ;
- elle doit posséder un constructeur par défaut (avec une visibilité **public** ou **protected**) ;
- elle ne doit pas être déclarée **final**, et aucune de ses méthodes ou attributs persistants ne doivent être déclarés **final** ;
- elle doit posséder un attribut qui représente la clé primaire dans la base de données.

De manière simple, il suffit de rajouter l'annotation **@Entity** au-dessus de la définition d'une classe, et l'annotation **@Id** au-dessus de l'attribut qui représente la clé primaire dans la base pour qu'elle devienne une entité. Par exemple :

```
@Entity
public class Employee {
    @Id
    private int id;
    ...
}
```

2.2.2 Gestionnaire d'entités

Il s'agit d'objets de type **EntityManager** définis dans le paquetage **javax.persistence**. Le gestionnaire d'entités est l'interlocuteur principal pour le développeur. Il fournit les méthodes pour gérer les entités : les rendre persistantes, supprimer leur valeur de la base de données, les retrouver dans la base de données.

La méthode **persist(objet)** de la classe **EntityManager** permet de rendre persistant un objet (en fait une entité). L'objet est alors géré par le gestionnaire d'entités : toute modification apportée à l'objet sera enregistrée dans la base de données. L'ensemble des entités gérées par un gestionnaire d'entités s'appelle un contexte de persistance.

2.2.3 Unité de persistance

Il s'agit d'une configuration nommée qui contient les informations nécessaires à l'utilisation d'une base de données. Elle est définie dans un fichier xml appelé **persistence.xml**. Ce fichier comporte en particulier le nom de l'unité de persistance, un ensemble de propriétés qui permettent de spécifier la connexion à la base de données, ainsi que les noms des classes entités qui sont gérées par cette unité de persistance.

Comme pour JDBC, l'utilisation de JPA nécessite un fournisseur de persistance qui implémente les classes et les méthodes de l'API. EclipseLink et Hibernate sont les fournisseurs de persistance les plus connus. Ce fournisseur de persistance est aussi mentionné dans l'unité de persistance.

2.3 Création d'une unité de persistance

Question 2. En suivant les indications de la Section 4.3, créez une unité de persistance dans votre projet.

2.4 Quelques précisions sur les entités

Cette section présente quelques détails sur les entités, et en particulier sur les annotations (du paquetage **javax.persistence**) qui peuvent être utilisées.

2.4.1 Accès par champs ou propriétés

L'état d'une entité doit être accessible au fournisseur de persistance afin de pouvoir stocker les informations dans la base de données. De même, lorsque l'état d'une entité est chargé depuis la base de données, le fournisseur doit pouvoir insérer les informations dans une nouvelle instance de l'entité. La manière d'accéder à l'état d'une entité est appelée le mode d'accès, et il existe deux modes : accès par les champs, ou accès par les propriétés.

L'accès par les champs signifie que le fournisseur de persistance passera directement par les attributs de la classe entité pour récupérer ou modifier l'état d'une entité. Dans ce cas là, les annotations se font au-dessus des attributs de la classe entité. Les attributs doivent alors être déclarés avec une visibilité **private** (de préférence), **protected** ou **package**. La visibilité **public** n'est pas autorisée car l'état de l'entité pourrait alors être accédé et modifié par n'importe quelle classe de la machine virtuelle. L'avantage de ce type d'accès est qu'il n'est pas nécessaire d'avoir les accesseurs et mutateurs pour chacun des attributs, et on respecte ainsi bien le principe d'encapsulation (si vous ne savez pas ce que c'est, allez lire le polycopié d'introduction du cours de POO2).

L'accès par propriétés est lié à la définition des *JavaBeans*. Un *JavaBean* est simplement une classe Java qui respecte certaines conventions :

- elle doit implémenter l'interface **Serializable** ;
- elle doit posséder un constructeur par défaut ;
- ses attributs ont une visibilité **private**, mais il doivent être accessibles publiquement par des accesseurs et mutateurs : on parle de propriété ;
- si une propriété est nommée **prop**, alors l'accesseur doit se nommer **getProp()** (ou **isProp()** dans le cas d'un booléen) et le mutateur **setProp()** ;
- la classe ne doit pas être déclarée **final**.

Dans le cas de l'accès par propriétés, le fournisseur de persistance passera par les accesseurs et mutateurs pour récupérer ou modifier l'état d'une entité. Dans ce cas là, les annotations se font au-dessus de la définition de l'accesseur. Et bien évidemment, il faut bien respecter les conventions de nommage ainsi que le typage : le type de retour de l'accesseur doit être identique au type du paramètre du mutateur. Il faut faire attention avec l'utilisation de ce type d'accès car il va à l'encontre du principe d'encapsulation ; il n'est donc pas recommandé de l'utiliser de manière systématique !

Dans la suite, on ne fera pas nécessairement la distinction entre champs et propriétés, et on parlera plus généralement d'attribut.

2.4.2 *Mapping* vers une table

Pour définir une entité, il faut utiliser, au-dessus de la définition de la classe, l'annotation **@Entity**. Cette annotation peut avoir un attribut **name** qui donne le nom de l'entité. Par défaut, le nom de l'entité est le nom terminal (sans le nom du paquetage) de la classe.

Par ailleurs, lors du *mapping*, dans les cas simples, une table correspond à une classe, et le nom de la table est le nom de la classe. Pour donner à la table un autre nom que le nom de la classe, il faut ajouter une annotation **@Table** en dessous de l'annotation **@Entity**. Cette annotation a un attribut **name** qui donne le nom de la table qui correspond à l'entité. Il est aussi possible à ce niveau de définir des contraintes d'unicité sur plusieurs colonnes de la tables, comme présenté dans l'exemple ci-après.

De plus, il est possible de définir de manière explicite le mode d'accès par défaut aux attributs de l'entité (par champ ou par propriété). Pour cela, on utilise l'annotation **@Access** avec en attribut **AccessType.FIELD** ou **AccessType.PROPERTY**. Cette annotation est elle aussi insérée en-dessous de l'annotation **@Entity**.

Ainsi, on peut avoir l'exemple suivant :

```
@Entity(name="emplEntity")
@Table(name="EMP",
        uniqueConstraints={
            @UniqueConstraint(
                columnNames ={"FIRSTNAME", "LASTNAME"})
        }
)
@Access(AccessType.FIELD)
public class Employee {
    @Id
    private int id;
    private String firstName;
    private String lastName;
    ...
}
```

2.4.3 *Mapping* pour les types basiques

Tous les attributs d'une entité qui sont d'un type "basique" de Java peuvent être rendus persistants, et le sont par défaut, sans annotations. Derrière ces types "basiques", on trouve les types primitifs de Java, les **String**, les types temporels de Java et de JDBC, les types énumérés et les types d'objets sérialisables. Notez que le fournisseur de persistance se chargera de faire la conversion entre le type Java et le type JDBC approprié. Il est possible qu'une exception soit levée si la conversion ne peut pas être effectuée de manière automatique, ou si le type n'est pas sérialisable.

Une annotation optionnelle **@Basic** peut être placée au-dessus d'un attribut pour marquer de manière explicite qu'il est persistant. Cette annotation sert surtout pour la documentation et n'a rien d'obligatoire.

En revanche, si on souhaite qu'un attribut ne soit pas persistant, on peut utiliser l'annotation **@Transient** au-dessus de l'attribut. S'il s'agit d'un champ, il est aussi possible d'utiliser le mot clé **transient** lors de la définition : **private transient int var;**

Si l'on souhaite définir de manière plus précise les caractéristiques de la colonne (dans la base de données) associée à un attribut, alors il faut utiliser l'annotation **@Column** au-dessus de la définition de l'attribut. Cette annotation comporte les attributs suivants :

- **name** : précise le nom de la colonne liée (le nom de l'attribut est utilisé par défaut) ;
- **unique** : précise s'il y a contrainte d'unicité sur la colonne (**true**) ou non **false**) ;
- **nullable** : précise si la colonne accepte les valeurs nulles ou non ;
- **insertable** : précise si la valeur doit être incluse lors de l'exécution de la requête SQL INSERT (**true** par défaut) ;
- **updatable** : précise si la valeur doit être mise à jour lors de l'exécution de la requête SQL UPDATE (**true** par défaut) ;
- **columnDefinition** : précise le code SQL pour la définition de la colonne dans la base de données (utile en cas de problème de conversion automatique) ;
- **length** : précise la longueur maximale pour un champ texte (255 par défaut) ;
- **precision** : précise le nombre maximum de chiffres que la colonne peut contenir pour un champ numérique ;
- **scale** : précise le nombre de chiffres après le séparateur décimal, pour un champ numérique flottant.

Ainsi, on peut avoir l'exemple suivant :

```
@Entity
public class Employee {
    @Id
    private int id;
    @Column(name="FNAME",
            length = 45,
            unique = true,
            nullable = false)
    private String firstName;
    @Column(precision = 5,
            scale = 2)
    private double height;
    ...
}
```

Par ailleurs, il est possible que l'on sache à l'avance que l'accès à certains attributs d'une entité seront très rares. Il est alors possible d'optimiser les performances lors de la récupération de l'entité en allant chercher uniquement

les données dont nous pensons qu'elles sont fréquemment consultées. Ainsi, il est intéressant de pouvoir déclarer que certaines données ne seront pas chargés lors de la lecture initiale de l'entité dans la base, mais seront rapportées seulement lorsque c'est nécessaire. On parle de chargement différé (*lazzy fetching*).

Par défaut, tous les attribut sont chargés initialement (*eagearly*) Mais l'annotation **@Basic** possède un attribut **fetch** qui peut valoir **FetchType.LAZY** ou **FetchType.EAGER** selon que l'on souhaite que le chargement de l'attribut se fasse de manière différée ou immédiate.

Pour des attributs dont le type est une énumération (qui peut avoir été définie par le programmeur), par défaut la colonne associée sera de type entier, et contiendra la valeur ordinale assignée à l'élément de l'énumération (0 pour le premier élément, 1 pour le deuxième, $n-1$ pour le $n^{\text{ième}}$). Ceci n'est pas toujours très pertinent, notamment par rapport aux potentielles évolutions du code, et on peut souhaiter stoker dans la base les chaînes de caractères qui définissent les constantes de l'énumération.

Pour ce faire, il existe une annotation **@Enumerated** qui se place au-dessus de la définition de l'attribut dont le type est un type énuméré. On peut ainsi définir le type de données stockées : **@Enumerated(EnumType.ORDINAL)** stocke les valeurs ordinales avec un type entier, et **@Enumerated(EnumType.STRING)** stocke les valeurs des constantes avec un type chaîne de caractères.

Les types temporels qui sont considérés comme des types basiques sont : **Date**, **Time** et **Timestamp** du paquetage **java.sql**, ainsi que **Date** et **Calendar** du paquetage **java.util**. Les types de **java.sql** ne posent pas de soucis. Mais pour les types de **java.util**, il faut des méta-données supplémentaires pour indiquer quel type de données utiliser lors de la conversion vers un type SQL. Pour ce faire, on utilise l'annotation **@Temporal** au-dessus de la définition de l'attribut de type temporel. Et on spécifie la conversion à utiliser : **TemporalType.DATE**, **TemporalType.TIME** and **TemporalType.TIMESTAMP**.

2.4.4 *Mapping* pour les clés primaires

Chaque entité qui est mappée avec une table dans la base de données relationnelle doit posséder un mappage vers la clé primaire dans la table. Ceci est réalisé par l'annotation **@Id** qui indique l'attribut qui identifie de manière unique l'entité. Les règles vues précédemment pour le *mapping* des types "basiques" s'appliquent aussi pour la colonne qui représente l'identifiant de l'entité. Par contre, les clés primaires sont supposées être insérables, non nulles et sans possibilité de mise à jour. Ainsi, si on utilise l'annotation **@Column** sur la clé primaire, il faut veiller à ne pas modifier les attributs **insertable**, **nullable** et **updatable**.

Plusieurs types de données sont autorisés pour les clés primaires, dont les entiers, les flottants, les caractères, **String**, **Date**. On évitera tout de même les types flottants à cause des problèmes de précision.

Si la clé est de type entier, l'annotation **@GeneratedValue** peut être utilisée pour indiquer que la clé sera générée automatiquement par le système de gestion de bases de données. Cette annotation possède un attribut **strategy** qui indique comment la clé sera générée. Cet attribut peut prendre les valeurs :

- **GenerationType.AUTO** : le type de génération est choisi par le fournisseur de persistance, selon le SGBD (c'est la valeur par défaut) ; si on connait le SGBD utilisé et que l'on sait quelle valeur choisir, il est préférable de ne pas utiliser **AUTO** ;
- **GenerationType.SEQUENCE** : la génération de la clé est réalisée en utilisant une séquence (c'est le cas pour une base Oracle ou PostGreSQL par exemple) ; l'annotation **@SequenceGenerator** peut être utilisée pour préciser comment est réalisée la génération de la séquence ;
- **GenerationType.IDENTITY** : une colonne de type **IDENTITY** est utilisée (c'est le cas pour une base Derby ou MySQL par exemple) ;
- **GenerationType.TABLE** : une table qui contient la prochaine valeur de l'identifiant est utilisée ; il est possible de spécifier la table utilisée avec l'annotation **@TableGenerator**.

En résumé, faites votre choix en fonction des possibilités offertes par le SGBD utilisé.

Ainsi, si l'on travaille avec une base Derby, on peut avoir l'exemple suivant :

```
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "NEMP")
    private int id;
    ...
}
```

2.5 Création de l'entité Service

Question 3. En suivant les indications de la Section 4.4, ajoutez dans le paquetage **modele** une entité **Service** qui représente un service de l'hôpital avec un identifiant, un nom et une localisation (voir la Figure 1). La clé primaire sera générée automatiquement et dans la base de donnée la colonne devra s'appeler **SERVNO**. Les noms doivent être uniques, et le nom et la localisation doivent avoir une taille maximale et être non nuls. Ils doivent aussi être écrits en majuscules pour éviter les problèmes liés à la casse. Par ailleurs, ajoutez un constructeur par défaut et un constructeur par données. N'oubliez pas de redéfinir les

méthodes **equals** et **hashCode** (il faut éviter de se baser sur l'identifiant généré automatiquement pour l'égalité car l'identifiant n'est généré que lorsque l'entité est rendue persistante dans la base). Redéfinissez aussi la méthode **toString**.

2.6 Utilisation du gestionnaire d'entités

2.6.1 Présentation du concept de gestionnaire d'entités

Pour qu'une entité puisse être rendue persistante dans la base de données, et que l'on puisse manipuler les entités persistantes, il est nécessaire de faire appel aux méthodes d'un gestionnaire d'entités. Ce gestionnaire d'entités est défini en Java par l'interface **EntityManager**. C'est ce gestionnaire d'entités auquel on délègue le réel travail de persistance. Jusqu'à ce qu'un gestionnaire d'entités soit utilisé pour effectivement créer, lire ou écrire une entité, cette entité n'est rien d'autre qu'un objet Java classique (non persistant).

Lorsqu'un gestionnaire d'entités obtient une référence sur une entité, qu'on lui a passé ou qu'il a lue dans la base, cet objet (l'entité) est dit géré par le gestionnaire d'entités. L'ensemble des instances d'entités gérées par un gestionnaire d'entités à un instant donné est appelé un contexte de persistance. Un contexte de persistance vérifie la propriété suivante : à chaque instant, il ne peut pas exister deux entités différentes qui représentent des données identiques dans la base. Par exemple, s'il existe une entité **Employee** avec un identifiant (clé primaire dans la base) de 158 dans le contexte de persistance, alors aucun autre objet de type **Employee** avec un identifiant valant 158 ne peut exister dans ce même contexte de persistance.

Les gestionnaires d'entités sont configurés pour être capables de rendre persistant et de gérer certains types d'objets, de lire et d'écrire dans une base de données. Les gestionnaires d'entités sont implémentés par un fournisseur de persistance (*EclipseLink* ou *Hibernate* par exemple).

2.6.2 Obtenir un gestionnaire d'entités

Tous les gestionnaires d'entités sont créés par un objet de type **EntityManagerFactory** (on parle de "fabrique"). L'unité de persistance indique les paramètres et les classes d'entités utilisés par tous les gestionnaires d'entités (**EntityManager**) obtenus depuis une instance unique de **EntityManagerFactory** qui est liée directement à l'unité de persistance. Il y a donc une relation $1-1$ entre une unité de persistance et son **EntityManagerFactory**. Dans l'environnement Java SE, nous pouvons utiliser la classe appelée **Persistence**, dont la méthode statique **createEntityManagerFactory(String persistenceUnitName)** retourne l'instance unique de **EntityManagerFactory** liée à l'unité de persistance nommée **persistenceUnitName**. L'exemple suivant montre comment créer un **EntityManagerFactory** pour l'unité de persistance nommée **EmployeePU** :

```
EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("EmployeePU");
```

À présent que nous avons une fabrique (**EntityManagerFactory**), nous pouvons facilement en obtenir un gestionnaire d'entités. L'exemple suivant montre comment obtenir un gestionnaire d'entités à partir de la fabrique obtenue dans l'exemple précédent :

```
EntityManager em = emf.createEntityManager();
```

Avec ce gestionnaire d'entités, nous pouvons maintenant commencer à travailler avec des entités persistantes.

2.6.3 Rendre une entité persistante

Rendre une entité persistante est l'opération qui consiste à prendre une entité qui n'a pas de représentation persistante dans la base de données (ou qui a été modifiée hors de la base), et à sauvegarder son état afin qu'il puisse être récupéré plus tard. Pour ce faire, l'interface **EntityManager** dispose de la méthode **persist()**. Le code suivant présente un exemple pour rendre persistant une instance de la classe **Employee** :

```
Employee emp = new Employee(158, "Jean", "Peuplu", 1650);
em.persist(emp);
```

La première ligne de cet exemple est simplement la création d'une instance **emp** de **Employee**. La seconde ligne utilise le gestionnaire d'entités créé précédemment afin de rendre persistante l'instance **emp**. L'employé **emp** sera alors sauvegardé dans la base de données. Et à la fin de l'appel à **persist()**, **emp** sera une entité gérée dans le contexte de persistance du gestionnaire d'entité.

2.6.4 Rechercher une entité

Lorsqu'une entité est dans la base de données, alors ce que l'on souhaite souvent faire c'est de la récupérer à nouveau (pour la lire ou la modifier). Ceci s'effectue avec la méthode **find()** du gestionnaire d'entité. Par exemple, pour récupérer l'instance de **Employee** dont la clé primaire (identifiant) est 158, on écrira :

```
Employee emp = em.find(Employee.class, 158);
```

On passe en paramètres de la méthode **find()** la classe de l'entité recherchée et l'identifiant ou la clé primaire qui identifie l'entité particulière recherchée. Lorsque l'appel à la méthode **find()** est terminé, l'objet renvoyé par la méthode sera une entité gérée par le gestionnaire d'entités, i.e. qu'elle fera partie du contexte de persistance associé au gestionnaire d'entité.

Par ailleurs, vous pouvez remarquer qu'il n'est pas nécessaire de caster l'objet renvoyé par la méthode **find()**. En effet, le type de retour de la méthode **find** est le même que la classe qui est passée en paramètre, i.e. que formellement la signature de la méthode est : **<T> T find(class<T> classeEntite, Object clePrimaire)**.

Si l'objet recherché a été supprimé de la base de données, ou que la clé primaire n'existe pas, alors la méthode retourne **null**. Lors de l'utilisation de la variable qui doit contenir l'entité recherchée, il faut donc vérifier que sa valeur n'est pas **null**.

2.6.5 Supprimer une entité

La suppression d'une entité revient à la supprimer de la base de données, comme si l'on effectuait une requête de type *DELETE*. Afin de supprimer une entité, cette entité doit être gérée, i.e. qu'elle est présente dans le contexte de persistance. L'application doit donc charger l'entité (la rendre persistance ou la rechercher dans la base) avant de la supprimer. La méthode pour supprimer une entité est la méthode **remove()**. Par exemple, pour supprimer une entité que l'on vient de rechercher dans la base, on écrira :

```
Employee emp = em.find(Employee.class, 158);
if (emp != null) {
    em.remove(emp);
}
```

Dans cet exemple, on fait d'abord le chargement de l'entité depuis la base de données, puis on la supprime par un appel à la méthode **remove()** sur le gestionnaire d'entités. On vérifie avant de supprimer une entité que cette dernière n'est pas **null** afin que la méthode **remove()** ne lève pas une exception de type **IllegalArgumentException**.

2.6.6 Mise à jour d'une entité

Une entité peut être mise à jour de différentes manières. Pour le moment, nous présentons le cas le plus fréquent et le plus simple. C'est le cas où une entité est gérée et nous souhaitons y apporter des modifications. Il est toujours possible de faire appel à la méthode **persist()** ou **find()** sur le gestionnaire d'entités afin de récupérer une référence sur une entité gérée. L'exemple ci-dessous propose d'augmenter de 1000 le salaire de l'employé dont l'identifiant est 158 :

```
Employee emp = em.find(Employee.class, 158);
emp.setSalary(emp.getSalary() + 1000);
```

Notez bien que dans ce cas là nous ne faisons pas appel au gestionnaire d'entités pour modifier l'objet, on modifie directement l'objet. Pour cette raison il est important que l'entité soit gérée, sinon le fournisseur de persistance n'a aucun moyen de détecter le changement, et aucun changement ne sera alors effectué sur la représentation persistante de l'employé.

2.6.7 Transactions

Pour le moment, nous n'avons pas évoqué la notion de transaction lorsque l'on manipule des entités. Normalement, les entités sont créées, mises à jour, et supprimées dans une transaction, et une transaction est nécessaire lorsque des changements sont réalisés sur la base de données. Les changements réalisés sur la base de données réussissent ou échouent de manière atomique, c'est pourquoi la vue persistante d'une entité doit être transactionnelle. On évite ainsi des états inconsistants.

En fait, dans tous les exemples précédents, sauf pour l'appel de la méthode **find()**, nous aurions dû entourer les appels aux méthodes par une transaction. La méthode **find()** n'est pas une opération mutante, ainsi elle ne nécessite pas une transaction.

Dans un environnement Java EE, *Java Transaction API* (*JTA*) est utilisé. Pour le moment, nous nous intéressons à l'environnement Java SE, et les transactions sont gérées à l'aide d'objets de type **EntityTransaction**. Il est nécessaire de commencer et de valider (*commit*) une transaction avant et après l'appel aux méthodes de manipulation des entités. Pour récupérer un **EntityTransaction**, il faut appeler la méthode **getTransaction()** sur le gestionnaire d'entités (**EntityManager**). Ensuite, la méthode **begin()** de **EntityTransaction** permet de commencer une transaction, et la méthode **commit()** permet de valider la transaction. Si on ne souhaite pas valider la transaction (si une exception a été levée par exemple), alors on peut appeler la méthode **rollback()** de **EntityTransaction** afin d'annuler la transaction.

Par exemple, la persistance d'une nouvelle entité peut se faire de la manière suivante :

```
em.getTransaction().begin();
Employee emp = new Employee(158, "Jean", "Peuplu", 1650);
em.persist(emp);
em.getTransaction().commit();
```

2.6.8 Synthèse

Voici un exemple complet d'utilisation du gestionnaire d'entités :

```
public static void main(String[] args) {
    final EntityManagerFactory emf =
                    Persistence.createEntityManagerFactory("EmployeePU");
    final EntityManager em = emf.createEntityManager();
    try{
        final EntityTransaction et = em.getTransaction();
        try{
            et.begin();
            // creation d'une entite persistante
            Employee emp = new Employee(158, "Jean", "Peuplu", 1650);
            em.persist(emp);
            et.commit();
        } catch (Exception ex) {
            et.rollback();
        }
    } finally {
        if(em != null && em.isOpen()){
            em.close();
        }
        if(emf != null && emf.isOpen()){
            emf.close();
        }
    }
}
```

Les objets de type **EntityManagerFactory**, **EntityManager** et **EntityTransaction** peuvent être déclarés **final** car ils ne doivent pas être modifiés. Par ailleurs, en cas de levée d'une exception, il faut bien veiller à faire un **rollback()** sur la transaction pour l'annuler. Et dans tous les cas (bloc **finally**), on ferme le gestionnaire d'entités et la fabrique de gestionnaire d'entités afin de libérer les ressources qui leur sont associées, avec la méthode **close()**.

2.7 Test persistance d'entités Service

Question 4. Dans le paquetage **tests**, ajoutez une classe **Test1** dans laquelle vous ferez un test de persistances d'entités **Service**. Vous pouvez vous baser sur le code suivant :

```
et.begin();
Service serv1 = new Service("Cardiologie", "Bat A, 1er étage");
Service serv2 = new Service("Pneumologie", "Bat B, 1er étage");
Service serv3 = new Service("Urgence", "Bat C, 1er étage");

em.persist(serv1);
em.persist(serv2);
serv1.setLocalisation("Bat D, 2ème étage");
et.commit();
```

Quel est le contenu de la base de données après l'exécution de la méthode de test ? Vérifiez que vous savez expliquer pourquoi la base contient ces données.

Question 5. Dans le fichier *persistance.xml* qui décrit l'unité de persistance, modifiez la stratégie de génération des tables : choisissez *None* à la place de *Drop and Create*. Que se passe-t-il quand vous ré-exécutez le test ? Dans le fichier *persistance.xml*, vous pouvez définir des propriétés afin de récupérer les scripts SQL de création et suppression des tables. Pour ce faire, activez la propriété **eclipselink.ddl-generation** avec la valeur **drop-and-create-tables** ; et activez également la propriété **eclipselink.ddl-generation.output-mode** avec la valeur **both**. Après l'exécution du test, vous devriez trouver dans le répertoire du projet les scripts nommés **createDDL.jdbc** et **dropDDL.jdbc**.

3 Associations : médecins et services

3.1 *Mapping* des associations avec JPA

Si les entités étaient seulement composées d'attributs simples, alors le *mapping* objet-relationnel serait trivial. Mais la plupart des entités doivent être capables de référencer ou d'être en relation avec d'autres entités.

Pour rappel, les associations bidirectionnelles peuvent être vues comme une paire d'associations unidirectionnelles. Ainsi, en fonction des cardinalités de la source et de la cible de la relation, on définit 4 types de relations unidirectionnelles :

- association *Many-to-One* (N − 1) ;
- association *One-to-One* (1 − 1) ;
- association *One-to-Many* (1 − N) ;
- association *Many-to-Many* (M − N).

Ces noms de type de relation sont aussi ceux utilisés dans les annotations qui sont utilisées pour indiquer le type de relation d'un attribut pour le *mapping*. Les associations sont donc déclarées dans les entités à l'aide d'annotations. Comme pour le *mapping* des types basiques, le *mapping* pour les relations peut être appliqué soit sur les champs, soit sur les propriétés.

3.1.1 *Mapping* Many-to-One

En UML, la classe source possède un attribut implicite du type de la classe cible si l'association est navigable. Considérons l'exemple de la Figure 2 avec une association *Many-to-One* entre **Employee** et **Department**.

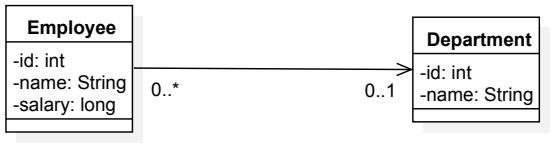

Figure 2: Exemple d'association *Many-to-One* en UML.

Dans le modèle objet, la classe **Employee** possédera un attribut de type **Department** qui permettra de référencer un seul département.

Un *mapping Many-to-One* est défini en annotant l'attribut dans l'entité source (l'attribut qui réfère sur l'entité cible) avec l'annotation **@ManyToOne**. En poursuivant notre exemple, cela donne :

```
@Entity
public class Employee {
    // ...
    @ManyToOne
    private Department department;
    // ...
}
```

Dans la base de données, le *mapping* d'une association signifie que l'on aura une clé étrangère : une colonne d'une table réfère la clé (en principe la clé primaire) d'une autre table. Avec JPA, on parle de colonne de jointure (*join column*), et l'annotation **@JoinColumn** est utilisée pour configurer ce type de colonne.

Dans presque toutes les associations, indépendamment de la source et de la cible, un des deux côtés de l'association aura la colonne de jointure dans sa table. Ce côté est appelé le propriétaire de l'association. Ceci est important pour le *mapping* car les annotations qui définissent le *mapping* sur les colonnes dans la base (par exemple **@JoinColumn**) sont toujours définies du côté du propriétaire de l'association.

Dans le cas d'une association **Many-to-One**, le côté propriétaire de l'association est la source, qui contient aussi la référence vers la cible. C'est donc là aussi que devra se trouver l'annotation **@JoinColumn**.

Par défaut une colonne de jointure sera nommée dans la base de données avec le nom de l'attribut qui référence la cible dans l'entité source, un *underscore* et le nom de la clé primaire dans l'entité cible. Avec l'annotation **@JoinColumn** il est possible de définir le nom de la colonne de jointure avec l'attribut **name**, comme présenté dans l'exemple suivant :

```
@Entity
public class Employee {
    // ...
    @ManyToOne
    @JoinColumn(name="DEPT_ID")
    private Department department;
    // ...
}
```

3.1.2 *Mapping* One-to-One

Dans le cas d'un *mapping One-to-One*, on utilisera l'annotation **@OneToOne**. Et on pourra naturellement utiliser l'annotation **@JoinColumn** afin de nommer la colonne de jointure.

Dans le cas d'un *mapping* **One-to-One** bidirectionnel, il faudra utiliser deux associations unidirectionnelles. Mais au niveau de la base de données, une seule entité pourra être propriétaire de l'association, et dans le cas de l'association *One-to-One*, cela peut être n'importe lequel des côtés de l'association. L'annotation **@JoinColumn** sera donc utilisée uniquement du côté du propriétaire. En revanche, de l'autre côté, il faut spécifier quel est le côté propriétaire en définissant l'attribut **mappedBy** de l'annotation **@OneToOne**. La valeur affectée à **mappedBy** doit être le nom de l'attribut dans l'entité propriétaire qui réfère sur l'entité non propriétaire de l'association.

Voici un exemple d'association *One-to-One* bidirectionnelle entre un employé et une place de parking :

```
@Entity
public class Employee {
    // ...
    @OneToOne
    @JoinColumn(name="PSPACE_ID")
    private ParkingSpace parkingSpace;
    // ...
}
```

```
@Entity
public class ParkingSpace {
    // ...
    @OneToOne(mappedBy="parkingSpace")
    private Employee employee;
    // ...
}
```

3.1.3 *Mapping* One-to-Many

Lorsque l'entité source référence une ou plusieurs instances d'entités cibles, une collection d'associations est utilisée. Lorsqu'une entité est associée avec une collection d'autres entités, c'est le plus souvent sous la forme d'un *mapping One-to-Many*.

Par exemple, si l'on prend le cas d'un département constitué de plusieurs employés (un employé ne pouvant faire partie que d'un seul département), alors il s'agit cette fois d'une association *Many-to-One*. Mais cette fois, l'association est a priori bidirectionnelle par nature. En effet, lorsque l'entité source possède un nombre arbitraire d'entités cibles dans une collection, il n'est pas possible au niveau de la base de données de stocker un nombre indéfini de clés étrangères dans la table de l'entité source. Ainsi une solution souvent mise en place est de laisser la table de l'entité cible stocker une clé étrangère qui réfère l'entité source de l'association. C'est pourquoi les associations *One-to-Many* sont bidirectionnelles et que le propriétaire de l'association est la cible de l'association.

Ainsi, pour l'association entre les départements et les employés, le propriétaire de l'association étant l'entité **Employee**, l'entité sera celle définie précédemment dans le *mapping Many-to-One*. Du côté non propriétaire de l'association (l'entité **Department**), il faut faire le *mapping* de la collection de **Employee** comme une association *One-to-Many* en utilisant l'annotation **@OneToMany**. Comme il ne s'agit pas du côté propriétaire de l'association, il faut alors définir l'attribut **mappedBy** de l'annotation **@OneToMany** comme présenté pour le *mapping One-to-One* bidirectionnel. Ainsi, pour la définition de l'entité **Department**, nous obtenons :

```
@Entity
public class Department {
    // ...
    @OneToMany(mappedBy="department")
    private Collection<Employee> employees;
    // ...
}
```

Plusieurs types de collections peuvent être utilisés pour sauvegarder les entités multiples d'une association. Selon les besoins, il est possible d'utiliser les interfaces génériques **Collection<E>**, **Set<E>**, **List<E>** et **Map<K,E>**. Il y a juste quelques règles à suivre pour leur utilisation. La première règle est de définir la collection avec l'une des interfaces mentionnées : les classes qui implémentent ces interfaces ne doivent pas être utilisées dans la définition des attributs mais peuvent être utilisées pour l'initialisation des attributs. La seconde règle est que la classe d'implémentation de la collection peut être invoquée tant que l'entité n'est pas gérée par le gestionnaire d'entités. Une fois que l'entité est gérée, il faut toujours utiliser l'interface de la collection pour manipuler cette dernière. Ceci est dû au fait que lorsque l'entité est gérée, le fournisseur de persistance peut remplacer le type concret de la collection par une autre implémentation de son choix.

Si on utilise une collection de type **List**, alors il est possible d'utiliser l'annotation **@OrderBy** au-dessus de la définition de la liste afin de spécifier l'ordre dans lequel on veux récupérer les éléments dans la base de données. Cependant, l'ordre n'est pas préservé lorsque les données sont écrites dans la base. Par défaut, l'ordre est donné par la clé primaire, mais on peut aussi spécifier en paramètre de l'annotation les attributs (séparés par des virgules) qui définissent l'ordre, et si l'ordre est croissant (**ASC**) ou décroissant (**DESC**).

Ainsi, si l'entité **Employee** possède un attribut **status** et un attribut **name**, alors on peut écrire dans l'entité **Department** :

```
@OneToMany(mappedBy="department")
@OrderBy("status DESC, name ASC")
private List<Employee> employees;
```

Si on utilise une collection de type **Map**, et que la clé est un attribut de l'entité cible de l'association, alors il faut utiliser l'annotation **@MapKey** au-dessus de la définition de la table associative, et spécifier avec l'attribut **name** quel est le nom de l'attribut qui va servir de clé.

Ainsi, si dans **Department**, on souhaite utiliser une table associative pour stocker les employés, et dont la clé est le nom de l'employé, alors on a :

```
@OneToMany(mappedBy="department")
@MapKey(name="name")
private Map<String, Employee> employees;
```

3.1.4 Chargement différé

Nous avons vu dans la Section 2.4.3 qu'il est possible de définir que le chargement d'un attribut sera différé. Au niveau des attributs basiques, ce chargement différé ne permet pas de faire des gains très importants sur les temps de chargement des données depuis la base. Mais lorsque cela concerne des associations, le chargement différé peut énormément améliorer les performances. Cela permet de réduire le nombre de requêtes SQL exécutées, d'améliorer le temps d'exécution des requêtes, et le temps de chargement des données.

Pour des associations de type *Many-to-One* ou *One-to-One*, les attributs correspondants sont chargés initialement par défaut. Pour les associations de type *One-to-Many* (et *Many-to-Many*), les attributs correspondants sont chargés en différé par défaut. Il faut tout de même noter que le type de chargement est une indication pour le fournisseur de persistance, mais ce dernier n'est pas forcé de respecter cette indication. Pour les associations bidirectionnelles, le chargement peut être différé d'un côté et être initial de l'autre côté.

Il est toujours possible de définir explicitement le mode de chargement en utilisant l'attribut **fetch** des annotations liées aux relations (**@ManyToOne**,

@OneToOne, **@OneToMany**). Cet attribut peut prendre les valeurs : **FetchType.LAZY** ou **FetchType.EAGER**.

3.2 Opérations en cascade

Par défaut, chaque méthode du gestionnaire d'entités s'applique uniquement à l'entité passée en argument de la méthode. La méthode ne va pas être appliquée en cascade sur les autres entités qui sont liées par une association à l'entité sur laquelle s'opère la méthode. Pour des méthodes comme **remove()**, c'est en général le comportement souhaité. Mais pour **persist()**, il est fortement probable que si une entité est en relation avec une autre entité, on souhaite que les deux entités soient rendues persistantes en même temps.

Heureusement, JPA fournit un mécanisme pour définir quand est-ce que les méthodes comme **persist()** doivent être appliquées en cascade sur les relations. Pour cela, on utilise l'attribut **cascade** des annotations d'association (**@OneToOne**, **@OneToMany**, **@ManyToOne**). On y définit la liste des opérations du gestionnaire d'entités à appliquer en cascade pour l'association considérée. Les valeurs peuvent être : **CascadeType.PERSIST**, **CascadeType.REFRESH**, **CascadeType.REMOVE**, **CascadeType.MERGE** et **CascadeType.ALL** qui correspond à toutes ces opérations.

L'attribut **cascade** est unidirectionnel. Si on souhaite avoir le même comportement des deux côtés d'une association, il faut donc l'écrire de manière explicite des deux côtés de l'association.

Par exemple, si l'on souhaite que lorsqu'un employé est rendu persistant ou supprimé, sa place de parking le soit aussi, on peut écrire :

```
@Entity
public class Employee {
    // ...
    @OneToOne(cascade={
                    CascadeType.PERSIST,
                    CascadeType.REMOVE })
    @JoinColumn(name="PSPACE_ID")
    private ParkingSpace parkingSpace;
    // ...
}
```

3.3 Associations entre les médecins et les services

On souhaite à présent ajouter l'entité **Medecin** ainsi que les relations entre les entités **Service** et **Medecin** et la relation réflexive sur **Medecin**. Pour le moment, on ne s'occupe pas de la relation d'héritage : **Medecin** hérite de **Personne**, et tous les attributs de l'entité **Personne** sont déportés dans l'entité **Medecin**. Ainsi, nous souhaitons ici mettre en place le diagramme de classe de la Figure 3.

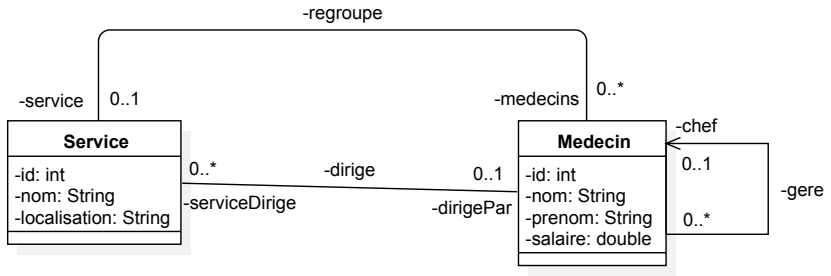

Figure 3: Diagramme de classe pour les entités **Medecin** et **Service**.

Question 6. Ajoutez dans le paquetage **modele** une entité **Medecin** qui représente un médecin de l'hôpital avec un identifiant, un nom, un prénom et un salaire (voir la Figure 3). La clé primaire sera générée automatiquement. Les couples nom et prénom doivent être uniques et non nuls. Ils doivent aussi être écrits en majuscules pour éviter les problèmes liés à la casse. Le salaire contient deux chiffres après la virgule. Par ailleurs, ajoutez un constructeur par défaut et un constructeur par données. Redéfinissez aussi les méthodes **equals**, **hashCode** et **toString**.

Question 7. Dans le paquetage **tests**, ajoutez une classe **Test2** dans laquelle vous ferez un test de persistance d'entités **Medecin**. Vous pouvez vous baser sur le code suivant :

```
et.begin();
Medecin med1 = new Medecin("Trancen", "Jean", 2135.23);
Medecin med2 = new Medecin("Gator", "Coralie", 3156.00);
Medecin med3 = new Medecin("Gator", "Magalie", 2545.3723);
em.persist(med1);
em.persist(med2);
em.persist(med3);
et.commit();
```

Vérifiez que les médecins sont bien enregistrés en base, et que le salaire contient au maximum 2 chiffres de précision. Vous pouvez aussi tester ce qu'il se passe si vous essayez de rendre persistants deux médecins avec le même nom et prénom.

Question 8. Modifiez les entités **Service** et **Medecin** afin d'ajouter les associations bidirectionnelles **regroupe** et **dirige** entre ces entités (voir la Figure 3). Ajoutez dans la classe **Service** une méthode **public boolean addMedecin(Medecin m)** qui ajoute le médecin **m** au service, et renvoie un booléen indiquant si l'ajout a pu être réalisé ou non. Si l'ajout est réalisé, il faut aussi mettre à jour le service du médecin. Si le médecin était déjà dans un autre service, il faut l'en retirer, pour garder les données consistantes. De la

même manière, ajoutez dans la classe **Medecin** une méthode **public boolean addServiceDirige(Service s)** qui ajoute le service **s** aux services dirigés par le médecin, et renvoie un booléen indiquant si l'ajout a pu être réalisé ou non. Si l'ajout est réalisé, il faut aussi mettre à jour le médecin qui dirige le service. Si le service était dirigé par un autre médecin, il faut retirer ce service des services dirigés par ce médecin, pour garder les données consistantes.

Question 9. Dans le paquetage **tests**, ajoutez une classe **Test3** dans laquelle vous ferez un test de persistance d'entités **Medecin** et **Service**. Vous pouvez vous baser sur le code suivant :

```
Service serv1 = new Service("Cardiologie", "Bat A, 1er étage");
Service serv2 = new Service("Pneumologie", "Bat B, 1er étage");
Service serv3 = new Service("Urgence", "Bat C, 1er étage");
Medecin med1 = new Medecin("Trancen", "Jean", 2135.23);
Medecin med2 = new Medecin("Gator", "Coralie", 3156.00);
Medecin med3 = new Medecin("Gator", "Magalie", 2545.37);
Medecin med4 = new Medecin("Hitmieu", "Helmer", 1873.30);
Medecin med5 = new Medecin("Cambronne", "Maude", 3765.20);
Medecin med6 = new Medecin("Haybon", "Sylvain", 2980.00);
serv1.addMedecin(med1);
serv1.addMedecin(med2);
serv1.addMedecin(med3);
serv2.addMedecin(med4);
serv3.addMedecin(med5);
serv3.addMedecin(med6);
med4.addServiceDirige(serv2);
med5.addServiceDirige(serv1);
med5.addServiceDirige(serv3);
```

Pour la persistance, vous pourrez par exemple utiliser la persistance en cascade, et utiliser le code suivant :

```
et.begin();
em.persist(serv1);
em.persist(serv2);
em.persist(serv3);
et.commit();
```

Vérifiez que les enregistrements en base sont effectués correctement, et regardez aussi la structure des tables avec les clés étrangères. Par ailleurs, n'hésitez pas à faire des tests différents afin de bien comprendre la notion de persistance en cascade.

Question 10. Modifiez l'entité **Medecin** afin d'ajouter l'association réflexive **gere** (voir la Figure 3). Ajoutez dans la classe **Medecin** une méthode **public boolean setChef (Medecin m)** qui défini le chef du médecin, et renvoie un booléen indiquant si l'ajout a pu être réalisé ou non (typiquement un médecin ne peut pas être le chef de lui-même ...). Modifiez le jeu de test précédent (**Test3**) en ajoutant une hiérarchie dans les médecins, et vérifiez que tout fonctionne correctement.

4 Quelques notions utiles

4.1 Travailler avec des bases de données sous Netbeans

Netbeans est un IDE très complet, et il propose en particulier une interface d'interaction avec des serveurs de base de données. Nous allons voir ici comment configurer une connexion à une base de données Java DB. Java DB est un système de gestion des bases de données relationnelles, basé sur le langage de programmation Java et SQL. Il s'agit de la version Oracle du projet Apache Derby, projet open source de base de données relationnelles. Java DB est inclus dans le Java Development Kit (JDK).

4.1.1 Création d'une base de données Java DB

Après avoir ouvert Netbeans, suivez les étapes suivantes afin de créer votre base de données :

1. ouvrez la fenêtre *Services* (*Ctrl* + *5*) ;
2. développez le nœud *Databases* ;
3. clic-droit sur *Java DB* puis cliquez sur *Start server* ;
4. clic-droit sur *Java DB* puis cliquez sur *Create Database ...* ;
5. dans la fenêtre qui s'est ouverte, rentrez les informations sur le nom de la base, le nom de l'utilisateur et le mot de passe (ne mettez pas de chaînes de caractères vides, et retenez le mot de passe ...); il est aussi possible de choisir l'emplacement de la base ;
6. cliquez sur le bouton *OK* ;
7. à présent vous devez voir que dans l'onglet *Databases* une connexion à votre nouvelle base de données a été ajoutée.

4.1.2 Connexion à la base de données

Pour vous connecter à une base de donnée Java DB, il vous suffit d'ouvrir la fenêtre *Services*, et dans le nœud *Databases* vous trouvez la liste des connections aux bases précédemment créées. Choisissez la connexion qui vous intéresse, faire un clic-droit dessus, et choisissez *Connect...*.

En faisant un clic-droit sur la connexion, et en choisissant *Properties* il est possible de retrouver des informations utiles comme l'URL de la base, la classe du Driver ou le nom de l'utilisateur. Vous pouvez également choisir *Rename* afin de donner un nom plus explicite à la connexion.

4.1.3 Création d'une table

Lorsque vous avez effectué la connexion à la base de données, vous pouvez développer le nœud relatif à la connexion. Vous y trouverez la liste des schémas, et nous nous intéressons ici au schéma qui a été généré par défaut. Si vous développez le nœud correspondant au schéma, vous trouverez trois dossiers correspondants aux tables, vues et procédures relatives à ce schéma.

Il y a deux manières de créer les tables de votre base de données :

- faites un clic-droit sur le nœud *Tables* et choisissez *Create table...*, puis donnez un nom à votre table, ajoutez les colonnes nécessaires puis cliquez sur *OK*;
- faites un clic-droit sur le nœud *Tables* et choisissez *Execute command...*; une page blanche de l'éditeur SQL s'ouvre alors dans la fenêtre principale; il vous suffit d'y rentrer votre script de requêtes SQL pour la création de vos tables, puis d'exécuter ce script (*Ctrl* + *Shift* + *E*).

Il vous est alors possible de visualiser les tables dans l'onglet *Tables*, et en développant l'onglet d'une table de visualiser les champs de cette table.

Un clic-droit sur une table puis *View Data...* vous permet de visualiser les données contenues dans la table, et aussi si nécessaire de rajouter ou supprimer à la main certaines données.

4.1.4 Requêtes SQL sur la base

Une fois que les tables ont été créées il est également possible de faire des requêtes SQL sur ces tables. Pour cela, faites un clic droit sur le nœud relatif à la connexion, ou bien sur le nœud relatif aux tables, et choisissez *Execute command...* puis entrez votre requête SQL dans l'éditeur SQL qui s'est ouvert. Il suffit ensuite d'exécuter cette requête (*Ctrl*+*Shift*+*E*) afin de visualiser le résultat de la requête.

4.1.5 Connexion à une base de donnée externe

Il est aussi possible de vous connecter via Netbeans à une base de données créée avec un autre système de gestion des bases de données que Java DB. Pour ce faire, dans la fenêtre *Services*, il faut faire un clic-droit sur *Databases*, choisir *New Connection...*. Dans la fenêtre de configuration de la connexion, il faut dans un premier temps choisir le driver approprié en fonction de votre SGBD, puis remplir les informations nécessaires à la connexion.

4.2 Importer un pilote JDBC dans un projet sous Netbeans

Si on utilise Derby comme SGBD, alors le pilote peut être importé en ajoutant la librairie *Java DB Driver*. Pour cela, dans la fenêtre *Projects*, dans le projet concerné, faites un clic-droit sur *Librairies* et choisissez *Add Library...* puis sélectionnez *Java DB Driver*.

Pour l'utilisation d'un autre SGBD, il faut en général télécharger sur le site de l'éditeur le fichier jar contenant le pilote. Il suffit ensuite de l'ajouter aux librairies du projet. Pour cela, dans la fenêtre *Projects*, dans le projet concerné, faites un clic-droit sur *Librairies* et choisissez *Add JAR/Folder...* puis sélectionnez le jar.

4.3 Création de l'unité de persistance

Après avoir créé une base de données et réalisé la connexion à cette base sous Netbeans, suivez les étapes suivantes afin de créer une unité de persistance pour l'utilisation de JPA :

1. ouvrez la fenêtre *Projects* (*Ctrl* + *1*) ;
2. clic-droit sur le nom du projet dans lequel vous souhaitez créer l'unité de persistance puis cliquez sur *New* et sélectionnez *Other ...* ;
3. dans la catégorie *Persistence*, choisissez un fichier de type *Persistence Unit* puis cliquez sur *Next*;
4. dans la fenêtre qui s'est ouverte, rentrez les informations sur le nom de votre unité de persistance, le fournisseur de persistance, la connexion à la base de données et la stratégie de génération des tables (en phase de test, il est préférable de choisir *Drop and Create* ;
5. cliquez sur le bouton *Finish* ;
6. à présent vous devez voir qu'un fichier **persistence.xml** a été ajouté au projet (sous un répertoire **META-INF**) ; vous avez accès à toutes les propriétés de ce fichier en mode *Design* ou en mode *Source* sous format xml.

4.4 Création d'une entité

Après avoir créé une unité de persistance, il est possible d'utiliser les assistants Netbeans pour créer des entités. Pour cela, il suffit de suivre les étapes suivantes :

1. ouvrez la fenêtre *Projects* (*Ctrl* + *1*) ;
2. clic-droit sur le nom du paquetage dans lequel vous souhaitez créer l'entité, puis cliquez sur *New* et sélectionnez *Other ...* ;

3. dans la catégorie *Persistence*, choisissez un fichier de type *Entity Class* puis cliquez sur *Next*;

4. dans la fenêtre qui s'est ouverte, rentrez les informations sur le nom de l'entité, le nom du paquetage, et le type de clé primaire (**Long** par défaut) ;

5. cliquez sur le bouton *Finish*.

Vous pouvez alors regarder le code généré par Netbeans, et vous rendre compte que la classe a été déclarée sérialisable (avec un identifiant **serialVersionUID**), que les annotations pour l'entité et la clé primaire sont présentes, que les accesseurs et mutateurs de la clé primaire sont présents, et que les méthodes **equals**, **hashCode** et **toString** ont été redéfinies. Les méthodes **equals()** et **hashCode()** sont définies de telle sorte qu'elle se basent sur la clé primaire, ce qui est cohérent.

Il faut maintenant compléter le code avec les annotations nécessaires et la définition des autres attributs de l'entité.

Par ailleurs, vous pouvez vérifier que l'entité a bien été ajoutée dans l'unité de persistance.

References

[1] Keith, M. and Schincariol, M., *Pro JPA 2: Second Edition - A definitive guide to mastering the Java Persistence API, APress, 2013*