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

TP 2 – 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 est la suite du TP1 et permet d'illustrer les notions suivantes :

- le *mapping* des relations d'héritage ;
- les classes insérées ;
- le langage d'interrogation de JPA.

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 réutilisabilité du code.

1 Héritage : personnes, médecins et malades

Les entités sont des objets, et devraient donc être capables d'hériter de l'état et du comportement d'autres entités. Ceci est même essentiel pour le développement d'applications orientées-objet. Nous rappelons qu'il existe différentes manières de représenter l'héritage dans une base de données, celles-ci seront abordées par la suite.

1.1 Les *mapped superclass*

JPA définit un type particulier de classes appelées *mapped superclass*, qui est très utile si l'on souhaite avoir une super-classe (pas une entité) dont vont hériter des entités. Une *mapped superclass* fournit une classe qui permet de stocker des états et des comportements partagés, dont vont hériter des entités. Mais il ne s'agit pas d'une entité, et ces *mapped superclass* ne peuvent donc pas être rendues persistantes. On ne peut donc pas faire de requêtes dessus, et elles ne peuvent pas être les cibles d'une association. Par ailleurs, des annotations comme **@Table** ne sont pas autorisées pour une *mapped superclass*.

Les *mapped superclass* peuvent être comparées aux entités de la même manière que l'on compare des classes abstraites et des classes concrètes : elles peuvent contenir un état et un comportement, mais elles ne peuvent pas être instanciées comme des entités persistantes. D'ailleurs, il est d'usage (et c'est une bonne pratique) que les *mapped superclass* soient des classes abstraites au niveau du modèle objet. On notera cependant qu'on peut faire de l'héritage avec des classes concrètes, auquel cas on utilisera une entité, et qu'une classe abstraite peut aussi être définie comme une entité.

On indique qu'une classe est une *mapped superclass* avec l'annotation **@MappedSuperclass**, comme dans l'exemple ci-dessous :

```
@MappedSuperclass
public abstract class Employee {
    private String nom;
    // ...
}
```

```
@Entity
public class FullTimeEmployee extends Employee {
    private long salary;
    private long pension;
    // ...
}
```

```
@Entity
public class PartTimeEmployee extends Employee {
    @Column(name="H_RATE")
    private float hourlyRate;
    //...
}
```

1.2 Modèles d'héritage avec JPA

JPA propose trois manières différentes de représenter les relations d'héritage dans la base de données. Nous parlons ici de hiérarchie d'héritage où la classe

mère est une entité (et pas une *mapped superclass*). La classe mère doit indiquer la hiérarchie d'héritage en utilisant l'annotation **@Inheritance** au-dessus de la définition de la classe. Cette annotation permet d'indiquer la stratégie qui est utilisée pour le *mapping*, et qui doit être une des trois stratégies définies par la suite.

1.2.1 Stratégie "une seule table"

Une manière performante et répandue pour stocker les états de plusieurs entités qui héritent d'une même entité mère est de définir une seule table dans la base de données. Cette table contient alors l'ensemble de tous les états possibles de chaque entité fille, i.e. que chaque attribut des entités filles aura une colonne correspondante dans la table. Il s'agit de la stratégie "une seule table". En conséquence, pour une ligne donnée de la table, qui représente une instance d'une classe concrète, il peut y avoir un certain nombre de colonnes qui ne sont pas renseignées car elles appartiennent à d'autres classes filles. Par ailleurs, il est nécessaire que toutes les entités dans la hiérarchie d'héritage utilisent des identifiants (pour la clé primaire) de même type.

En général, l'approche "une seule table" est plus coûteuse en terme d'espace de stockage pour les données, mais elle permet d'offrir de meilleures performances pour les requêtes et l'écriture en base de données. En effet, il n'est pas nécessaire de faire de jointures car tout est dans une seule table.

Pour indiquer le choix de cette stratégie, il faut spécifier dans l'annotation **@Inheritance** que l'attribut **strategy** vaut **InheritanceType.SINGLE_TABLE**. Notez que cette stratégie est celle par défaut si rien n'est précisé.

En outre, dans la table générée dans la base de données, il y aura une colonne supplémentaire qui ne correspond à aucun attribut d'aucunes des classes mères ou filles. Il s'agit d'une colonne discriminante qui est nécessaire lorsqu'on utilise la stratégie "une seule table" pour faire de l'héritage. Grâce à cette colonne, il est possible de savoir à quelle entité fille est liée chaque enregistrement dans la base. L'annotation **@DiscriminatorColumn** permet de paramétrer cette colonne, en particulier l'attribut **name** permet de spécifier son nom dans la table, et l'attribut **discriminatorType** permet de spécifier le type de données. Ce dernier attribut peut prendre les valeurs **DiscriminatorType.STRING** (par défaut), **DiscriminatorType.INTEGER** ou **DiscriminatorType.CHAR**.

Chaque ligne dans la table aura une valeur dans la colonne discriminante ; cette valeur est appelée valeur discriminante, et elle indique le type de l'entité qui est mémorisée dans la ligne. Chaque entité concrète dans la hiérarchie d'héritage a donc besoin d'une valeur discriminante spécifique par rapport à son type qui permette au fournisseur de persistance d'utiliser le bon type lors de l'enregistrement d'une entité, ou d'une requête. Dans chaque entité concrète, l'annotation **@DiscriminatorValue** permet de spécifier la valeur discriminante pour l'entité en question. Cette valeur est passée sous forme d'une chaîne de caractères dans l'annotation. Cette valeur doit être du même type que celui spécifié par l'annotation **@DiscriminatorColumn** dans l'entité mère. Si l'annotation **@DiscriminatorValue** n'est pas spécifiée, alors c'est le fournisseur de persistance qui attribuera automatiquement une valeur.

Ainsi, on peut définir l'exemple suivant :

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="EMP_TYPE",
                discriminatorType = DiscriminatorType.STRING)
public abstract class Employee { ... }
```

```
@Entity
@DiscriminatorValue("FTEmp")
public class FullTimeEmployee extends Employee { ... }
```

```
@Entity
@DiscriminatorValue("PTEmp")
public class PartTimeEmployee extends Employee { ... }
```

1.2.2 Stratégie "jointure"

Du point de vue d'un développeur Java, un modèle de données qui *mappe* chaque entité avec sa propre table fait sens. Ainsi, il y aura une table différente pour chaque entité, qu'elle soit abstraite ou concrète. On parle de stratégie "jointure". Ceci est la manière la plus efficace de stocker les données partagées par plusieurs entités filles dans une hiérarchie. Ce qui est moins performant, c'est lorsqu'il faut ré-assembler une instance d'une entité fille : il faut utiliser des jointures entre la table d'une entité fille et la table de l'entité mère. L'insertion est aussi plus coûteuse car elle se fait dans plusieurs tables.

Les identifiants (clés primaires) doivent être du même type dans toutes les entités de la hiérarchie. Au niveau de la base de données, la clé primaire dans la table d'une entité fille sera aussi une clé étrangère qui référence la clé primaire de la table de l'entité mère.

Pour utiliser cette stratégie "jointure", il faut spécifier dans l'annotation **@Inheritance** que l'attribut **strategy** vaut **InheritanceType.JOINED**. Les annotations **@DiscrinatorColumn** et **@DiscrimatorValue** peuvent aussi être utilisées ici. Cette fois la colonne discriminante sera présente dans la table correspondant à l'entité mère de la hiérarchie d'héritage.

Ainsi, on peut définir l'exemple suivant :

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="EMP_TYPE",
                discriminatorType = DiscriminatorType.INTEGER)
public abstract class Employee { ... }
```

```
@Entity
@DiscriminatorValue("1")
public class FullTimeEmployee extends Employee { ... }
```

```
@Entity
@DiscriminatorValue("2")
public class PartTimeEmployee extends Employee { ... }
```

1.2.3 Stratégie "une table par classe concrète"

Une troisième approche pour le *mapping* d'une hiérarchie d'héritage est d'utiliser une stratégie où une table est définie pour chaque classe concrète. Ainsi, tous les attributs partagés dans l'entité mère sont redéfinis séparément dans chaque table associée à une entité concrète qui hérite de l'entité mère.

L'aspect négatif de cette stratégie est que les requêtes polymorphes sont alors plus coûteuses qu'avec les autres stratégies : il faut utiliser la commande **UNION** afin de faire la requête sur toutes les tables correspondant aux entités filles, ce qui est coûteux s'il y a beaucoup de données. En revanche, cette stratégie offre de meilleures performances que la stratégie "jointure" si l'on souhaite faire des requêtes sur une classe concrète. Par ailleurs, avec cette stratégie, la colonne discriminante n'a plus d'utilité.

Pour utiliser cette stratégie, il faut spécifier dans l'annotation **@Inheritance** que l'attribut **strategy** vaut **InheritanceType.TABLE_PER_CLASS**. Si une base de données existante est utilisée, il est possible que les colonnes héritées soient nommées différemment dans chacune des tables concrètes. À ce moment là, l'annotation **@AttributeOverride** permet de spécifier le nom qui est associé aux colonnes héritées dans les tables concrètes. Si l'attribut que nous souhaitons renommer correspond à une association, alors il faut utiliser l'annotation **@AssociationOverride**.

Ainsi, on peut définir l'exemple suivant :

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Employee {
    @Id private int id;
    private String name;
    @ManyToOne
    private Department department;
    //...
}
```

```
@Entity
@AssociationOverride(name="department",
                joinColumns=@JoinColumn(name="DPT"))
public class FullTimeEmployee extends Employee {
    @AttributeOverride(name="name",
                column=@Column(name="FULLNAME"))
    private long salary;
    private long pension;
    // ...
}
```

1.3 Les médecins et les malades sont des personnes

Nous allons à présent modifier le modèle objet, et faire apparaître l'entité **Personne** définie par son identifiant, son nom et son prénom. Comme précédemment, les couples nom/prénom sont uniques. Cette entité pourra être déclarée abstraite car les personnes seront soit des médecins (pour lesquels on définit un salaire), soit des malades. Ainsi, nous souhaitons ici mettre en place le diagramme de classe de la Figure 1.

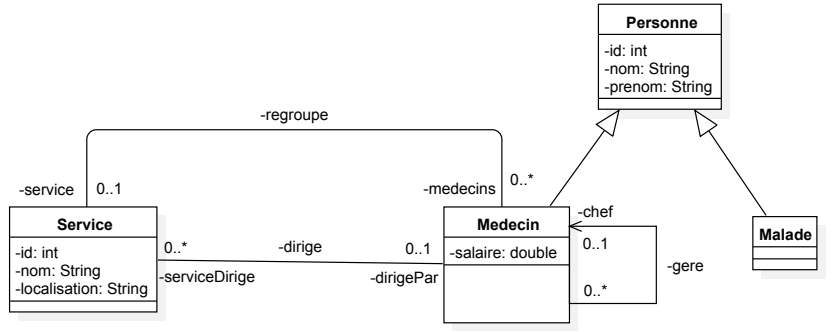

Figure 1: Diagramme de classe pour les entités **Service**, **Personne**, **Medecin** et **Malade**.

Question 1. Ajoutez dans le paquetage **modele** une entité abstraite **Personne**, une entité **Malade** qui hérite de **Personne**, et modifiez l'entité **Medecin** qui hérite aussi de **Personne**. Référez-vous à la Figure 1, et pensez bien à la génération des clés primaires, l'unicité du couple nom/prénom, et l'écriture en majuscule.

Question 2. Complétez le test précédent de la classe **Test3**, en y ajoutant quelques malades qui seront rendus persistants. Vous pouvez vous baser sur le code suivant :

```
Malade mal1 = new Malade("Proviste", "Alain");
Malade mal2 = new Malade("Trahuil", "Phil");
Malade mal3 = new Malade("Ancieux", "Cecile");
Malade mal4 = new Malade("Conda", "Anna");
// ...
et.begin();
// ...
em.persist(mal1);
em.persist(mal2);
em.persist(mal3);
em.persist(mal4);
et.commit();
```

Étudiez ce qui se passe au niveau de la base de données selon que vous utilisez une *mapped superclass* ou une entité ; et pour une entité selon le modèle d'héritage.

2 Classe insérée : la classe adresse

2.1 Définition d'une classe insérée

Une classe insérée est une classe qui dépend d'une autre entité pour son identifiant. Elle n'a pas d'identifiant et fait donc partie de l'état d'une entité distincte. En Java, les classes insérées sont similaires à des associations dans lesquelles la classe (insérée) est la cible d'une association et une entité (celle dans laquelle elle est insérée) est la source. Au niveau de la base de données, la classe insérée est stockée avec le reste de l'état de l'entité dans une ou plusieurs colonnes (une colonne par attribut de la classe insérée). On voit là un exemple du décalage entre les modèles objets et relationnels. Dans le modèle objet, il est tout à fait naturel de considérer une classe distincte pour mettre en évidence l'aspect conceptuel. Dans le modèle relationnel, il est plus simple au niveau du stockage des données de tout stocker dans une seule table.

2.2 *Mapping* d'une classe insérée

Un exemple très classique de classe insérée est l'adresse. Un numéro de rue, un nom de rue, un code postal et une ville forment logiquement le concept d'adresse. Au niveau du modèle objet il est tout à fait naturel d'utiliser une classe spécifique pour représenter une adresse au lieu de dupliquer ses attributs dans tous les objets qui ont une adresse. Par exemple, comme présenté dans la Figure 2, entre un employé et une adresse, on a une relation de composition, i.e. que l'employé est propriétaire de l'adresse et qu'une instance d'une adresse ne peut pas être partagée par un autre objet que l'employé qui la possède.

Avec cette représentation, les informations qui composent une adresse sont donc parfaitement encapsulées dans un objet. De plus, si une autre entité, une entreprise par exemple possède aussi comme information une adresse, alors cette entité peut posséder un attribut qui réfère sur sa propre classe insérée **Address**.

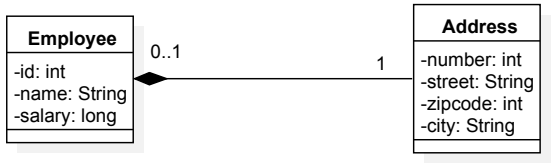

Figure 2: Diagramme de classe : association de composition entre **Employee** et **Address**.

Une classe insérée est marquée comme telle en utilisant l'annotation **@Embeddable** au-dessus de la définition de la classe. Lorsqu'une classe est définie comme pouvant être insérée, alors ses attributs seront rendus persistants en tant qu'une partie d'une autre entité. Les annotations concernant les colonnes (**@Basic**, **@Column**, **@Temporal**, **@Enumerated**) peuvent être ajoutés aux attributs de la classe insérée. Mais la classe insérée ne doit pas contenir d'associations. Par exemple, la classe insérée **Address** est définie de la manière suivante :

```
@Embeddable
public class Address {
    private int number;
    private String street;
    private String city;
    @Column(name="ZIP_CODE")
    private String zipcode;
    // ...
}
```

Pour utiliser cette classe insérée dans une entité, l'entité doit posséder un attribut du même type annotée par **@Embedded**. Par ailleurs, en plus de cette annotation, il est possible d'utiliser les annotations **@AttributeOverrides** et **@AttributeOverride** pour renommer les attributs de la classe insérée lors de la persistance dans la base de données. Par exemple, l'entité **Employee** est définie de la manière suivante :

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    @Embedded
    @AttributeOverrides({
            @AttributeOverride(name = "street",
                    column = @Column(name="rue")),
            @AttributeOverride(name="city",
                    column = @Column(name = "ville"))
    })
    private Address address;
    // ...
}
```

La décision d'utiliser des classes insérées ou des entités est fonction de si l'on aura besoin de créer des associations avec ces objets. Les classes insérées ne sont pas destinées à être des entités, et si l'on commence à les concevoir comme tel, alors il est préférable d'utiliser une entité au lieu d'une classe insérée.

2.3 Insertion de la classe Adresse

Nous allons à présent modifier le modèle objet, et faire apparaître la classe insérée **Adresse** définie par un numéro, une rue et une ville. L'entité **Malade** possédera une référence vers cette classe insérée. Ainsi, nous souhaitons ici mettre en place le diagramme de classe de la Figure 3.

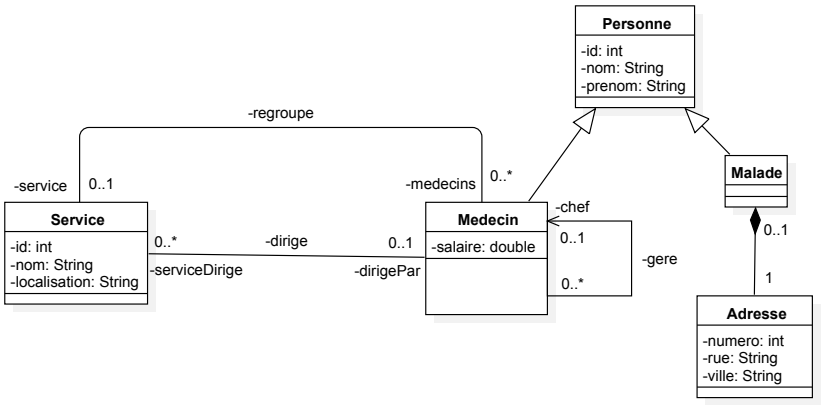

Figure 3: Diagramme de classe avec la classe insérée **Adresse**.

Question 3. Ajoutez dans le paquetage **modele** une classe insérée **Adresse**, et modifiez l'entité **Malade** afin qu'elle possède un attribut de type **Adresse**, comme présenté dans la Figure 3. Les champs rue et ville sont non nuls. N'oubliez pas de modifier le constructeur de la classe **Malade**. Modifiez en conséquence le jeu de test précédent, vérifiez que tout fonctionne correctement et observez la représentation des données dans la base.

3 Langage d'interrogation de JPA

JPA supporte deux types de langages pour rechercher des entités et autres données persistantes depuis la base de données. Le premier langage est *Java Persistence Query Language* (JPQL) qui opère sur le modèle logique des entités et non pas sur le modèle physique de données. Les requêtes peuvent aussi être exprimées en SQL afin de tirer avantage de la base de données sous-jacente.

3.1 Première découverte de JPQL

Un exemple de requête simple en JPQL pour sélectionner toutes les instances d'une classe d'entités est donné par :

```
SELECT e FROM Employee e
```

Cela ressemble fortement à SQL, et c'est normal car JPQL utilise la syntaxe SQL, ce qui permet aux habitués de SQL de facilement s'adapter à JPQL. La différence majeure entre SQL et JPQL pour la requête ci-dessus est qu'au lieu de sélectionner les données dans une table de la base de données, on spécifie une entité du modèle objet. Une autre différence est l'utilisation des l'alias (ici **e** pour **Employee**). Cela permet d'indiquer que le résultat de la requête est de type **Employee**, et ainsi l'exécution de cette requête renverra une liste d'instances de **Employee**. Par ailleurs, avec l'alias il est possible de spécifier un attribut persistant ou même de naviguer à travers les relations entre les entités en utilisant l'opérateur point ('.'). Par exemple, si on souhaite récupérer seulement les noms des employés, on peut utiliser la requête suivante :

```
SELECT e.name FROM Employee e
```

Il est aussi possible de sélectionner une entité qui n'est pas listée dans la clause *FROM*. Par exemple, dans le cas où il y a une association *Many-to-One* entre **Employee** et **Department** (l'attribut est nommé **department**), on peut écrire la requête :

```
SELECT e.department FROM Employee e
```

3.2 Requêtes de sélection avec JPQL

Comme SQL, JPQL supporte la clause *WHERE* pour mettre des conditions sur les données recherchées. De nombreux opérateurs SQL sont aussi disponibles : *IN*, *LIKE*, *BETWEEN*, et des fonctions comme *SUBSTRING* et *LENGTH*. La différence réside surtout dans le fait qu'avec JPQL on utilise des expressions avec les entités et non pas les colonnes des tables. Par exemple, on peut avoir la requête suivante :

```
SELECT e FROM Employee e
WHERE e.department.name = 'SECRETARIAT'
AND e.address.city IN ('Lille','Lens')
```

Si on ne souhaite pas récupérer toutes les données contenues dans les instances d'une classe d'entité, alors il est possible de définir les noms des attributs que l'on souhaite récupérer, comme dans l'exemple suivant :

```
SELECT e.name, e.salary FROM Employee e
```

Le type de résultat d'une requête de sélection ne peut pas être une collection, il faut que ce soit une instance d'entité ou un type d'attribut persistant. Ainsi, si on souhaite naviguer dans une association de type *One-to-Many* par exemple, on ne peut pas utiliser directement la notation pointée avec l'attribut qui est une collection d'objets. Il faudra alors faire une jointure entre les entités.

Il existe deux manières de faire des jointures, comme présenté dans les exemples ci-dessous :

```
SELECT p.number FROM Employee e, Phone p
WHERE e = p.employee
AND e.department.name = 'SECRETARIAT'
AND p.type = 'PORTABLE'
```

Ou encore, on peut faire apparaitre explicitement l'opérateur *JOIN* :

```
SELECT p.number FROM Employee e JOIN e.phones p
WHERE e.department.name = 'SECRETARIAT'
AND p.type = 'PORTABLE'
```

La syntaxe pour les requêtes d'agrégation est très similaire à SQL. On retrouve les fonctions d'agrégat *AVG*, *COUNT*, *MIN*, *MAX* et *SUM*, les résultats peuvent être groupés dans la clause *GROUP BY* et filtrés en utilisant la clause *HAVING*. Par exemple, on peut écrire la requête suivante en JPQL :

```
SELECT d, COUNT(e), MAX(e.salary), AVG(e.salary)
FROM Department d JOIN d.employees e
GROUP BY d
HAVING COUNT(e) >= 5
```

3.3 Requêtes paramétrées

JPQL permet de faire des requêtes paramétrées, et supporte deux types de syntaxe pour la liaison des paramètres. Le premier est la liaison par position : les paramètres sont indiqués dans la requête par un point d'interrogation suivi du numéro de paramètre. Lors de l'exécution de la requête, il faut spécifier le numéro de paramètre qui doit être remplacé. Cela donne par exemple la requête paramétrée suivante :

```
SELECT e FROM Employee e
WHERE e.department = ?1
AND e.salary > ?2
```

Avec le second type de syntaxe, des noms sont donnés aux paramètres, et on indique dans la requête qu'il s'agit de paramètre en faisant précéder le nom d'un symbole deux-points (':'). Lors de l'exécution de la requête il faut spécifier le nom du paramètre qui doit être remplacé. Cela donne par exemple la requête paramétrée suivante :

```
SELECT e FROM Employee e
WHERE e.department = :dept
AND e.salary > :base
```

3.4 Définir une requête

Pour configurer et exécuter une requête avec JPA, on utilise l'interface **Query**. Une implémentation de cette interface est obtenue en appelant les méthodes **createQuery()**, **createNamedQuery()** ou **createNativeQuery()** de l'interface **EntityManager**. Il y a deux approches pour définir une requête : une requête peut être spécifiée dynamiquement à l'exécution, ou configurée dans les métadonnées de l'unité de persistance et référencée par son nom. Les requêtes dynamiques ne sont rien d'autre que des chaînes de caractères et peuvent être définies à la volée. Les requêtes nommées sont statiques et non modifiables mais elles sont plus efficaces à l'exécution car le fournisseur de persistance fait une seule fois la traduction de la requête de JPQL vers SQL au démarrage de l'application.

3.4.1 Requêtes dynamiques

Pour exécuter dynamiquement une requête, il suffit d'écrire la requête sous forme d'une chaîne de caractères, puis de la passer en paramètre de la méthode **createQuery()** appelée sur l'instance de **EntityManager**. On obtient alors une instance de type **Query**.

Si la requête est paramétrée, on utilise la méthode **setParameter(int position, Object value)** ou **setParameter(String name, Object value)** de

l'interface **Query**. Le choix dépend de la manière dont ont été définis les paramètres.

Si l'on souhaite exécuter la requête et récupérer un seul résultat, on fait appel à la méthode **getSingleResult()** de l'interface **Query**. Cette méthode renvoie un objet de type **Object** qu'on pourra caster au besoin.

Si l'on souhaite exécuter la requête et récupérer une liste de résultats, on fait appel à la méthode **getResultList()** de l'interface **Query**. Cette méthode renvoie un objet de type **List**.

Par exemple, pour un employé donné par son nom et le nom de son département, on peut utiliser le code suivant :

```
final String strQuery = "SELECT e.salary FROM Employee e "
        + "WHERE e.department.name = :deptName "
        +"AND e.name = :empName ";
Query query = em.createQuery(strQuery);
query.setParameter("deptName", deptName);
query.setParameter("empName", empName);
long salary = (long) query.getSingleResult();
```

Ou si l'on souhaite récupérer la liste des employés d'un département :

```
final String strQuery = "SELECT e FROM Employee e "
        + "WHERE e.department.name = :deptName ";
Query query = em.createQuery(strQuery);
query.setParameter("deptName", deptName);
List<Employee> employees = query.getResultList();
```

3.4.2 Requêtes nommées

Les requêtes nommées sont un outil puissant pour organiser la définition des requêtes et améliorer les performances d'une application. Une requête nommée est définie en utilisant l'annotation **@NamedQuery** qui doit être placée dans la définition de la classe d'une entité. Les requêtes nommées sont en général définies dans la classe d'entité qui correspond le plus directement au résultat de la requête. L'annotation définie le nom de la requête et le texte de la requête.

Par exemple, si on souhaite définir une requête nommée pour trouver le salaire d'un employé dont on donne en paramètres le nom et le nom de département, on pourra écrire dans l'entité **Employee** :

```
@NamedQuery(name="findSalaryForNameAndDepartment",
        query="SELECT e.salary FROM Employee e "
                + "WHERE e.department.name = :deptName "
                + "AND e.name = :empName")
@Entity
public class Employee { ... }
```

Le nom de la requête nommée est à la portée de l'unité de persistance et doit donc être unique. Ceci est important à garder à l'esprit, car si on utilise des noms génériques comme "findAll" dans plusieurs entités cela ne fonctionnera pas correctement. Une bonne pratique consiste donc à préfixer le nom de la requête nommée par l'entité dans laquelle elle est définie. Par exemple, la requête "findAll" dans l'entité **Employee** devrait être nommée "Employee.findAll".

Si l'on souhaite définir plusieurs requêtes nommées concernant l'entité **Employee** on pourra écrire :

```
@NamedQueries({
        @NamedQuery(name="Employee.findAll",
                query="SELECT e FROM Employee e"),
        @NamedQuery(name="Employee.findByPrimaryKey",
                query="SELECT e FROM Employee e "
                        + "WHERE e.id = :id"),
        @NamedQuery(name="Employee.findByName",
                query="SELECT e FROM Employee e "
                        + "WHERE e.name = :name")
})
@Entity
public class Employee { ... }
```

Pour exécuter une requête nommée, on appellera la méthode **createNamedQuery()** de l'interface **EntityManager**. Par exemple, pour exécuter la requête "Employee.findByName", on écrira :

```
Query query = em.createNamedQuery("Employee.findByName");
query.setParameter("name", name);
Employee emp = (Employee) query.getSingleResult();
```

3.5 Requêtes de sélection sur la base de données de l'hôpital

Question 4. Modifiez l'unité de persistance (fichier **persistance.xml**) afin de ne pas écraser les tables existantes et leur contenu. Pour cela, choisissez **none** pour la stratégie de génération des tables, et choisissez **none** pour la propriété **eclipselink.dll-generation**. Ajoutez dans le paquetage **tests** une classe **TestJPQL1** dans laquelle vous proposerez une méthode qui prend en paramètre un nom de service, et qui affiche sur la console tous les médecins qui travaillent dans ce service (il ne devra pas y avoir de sensibilité à la casse). Testez le bon fonctionnement de cette méthode.

Question 5. Ajoutez dans le paquetage **tests** une classe **TestJPQL2** dans laquelle vous proposerez une méthode qui prend en paramètre un nom de service, et qui affiche sur la console tous les noms et les salaires des médecins qui travaillent dans ce service. Il ne faudra pas que la requête récupère les médecins,

mais seulement les informations nécessaires. Testez le bon fonctionnement de cette méthode.

Question 6. Ajoutez dans l'entité **Medecin** une requête nommée qui sélectionne tous les noms des médecins qui ont un chef dont le nom et le prénom sont passés en paramètres. Ajoutez dans le paquetage **tests** une classe **TestJPQL3** dans laquelle vous proposerez une méthode qui prend en paramètre un nom et un prénom, et qui affiche sur la console tous les noms des médecins récupérés avec la requête nommée créée précédemment. Testez le bon fonctionnement de cette méthode.

3.6 Mise à jour des entités

3.6.1 Modification des entités dans une transaction

Il est possible de mettre à jour des entités (en modifiant leurs attributs) dans une transaction. Pour ce faire, on peut d'abord récupérer les entités que l'on souhaite modifier avec une requête de sélection. Puis, dans une transaction, on fait appel aux méthodes de la classe pour mettre à jour certains attributs. Lors de la validation de la transaction, les données seront alors mises à jour dans la base.

Par exemple, si on souhaite ajouter 10 euros au salaire de l'employé dont le nom est "DUPONT", on peut écrire le code suivant :

```
Query query = em.createNamedQuery("Employee.findByName");
query.setParameter("name", "DUPONT");
Employee emp = (Employee) query.getSingleResult();
final EntityTransaction et = em.getTransaction();
try {
    et.begin();
    emp.setSalary(emp.getSalary() + 10);
    et.commit();
} catch (Exception ex) {
    et.rollback();
}
```

3.6.2 Mise à jour en volume

Il est aussi possible de faire des mises à jour en volume (*bulk update*). Ce type de mise à jour est réalisé avec une requête de type *UPDATE*. La requête doit concerner un seul type d'entité et on peut modifier un ou plusieurs attributs des entités (un attribut simple ou une association *One-to-One* ou *Many-to-One*). Il est possible d'utiliser la clause *WHERE* pour sépcifier les entités concernées par la mise à jour. La syntaxe est semblable à celle de SQL mais on utilise les expressions sur les entités au lieu des tables et des colonnes. Par ailleurs, pour effectuer cette mise à jour en volume, il faut appeler la méthode **executeUpdate()** de l'interface **Query**. Cette méthode doit être appelée à l'intérieur d'une transaction, et lors de la validation les données sont mises à jour dans la base.

Par exemple, si l'on souhaite augmenter de 10 euros le salaire des employés du département "SECRETARIAT", on peut écrire le code suivant :

```
final String strQuery = "UPDATE Employee e "
            + "SET e.salary = e.salary + 10 "
            + "WHERE e.department.name = :deptName";
Query query = em.createQuery(strQuery);
query.setParameter("deptName", "SECRETARIAT");
final EntityTransaction et = em.getTransaction();
try {
    et.begin();
    query.executeUpdate();
    et.commit();
} catch (Exception ex) {
    et.rollback();
}
```

Il faut cependant faire attention avec ces mises à jour en volume car le contexte de persistance n'est pas mis à jour afin de refléter le résultat de la mise à jour. Les opérations en volume sont exécutées comme des requêtes SQL directement sur la base de données, contournant l'état des entités dans le contexte de persistance. Ainsi, dans l'exemple précédent, si au début de la transaction des entités on exécute une requête de sélection, les entités récupérées par la requêtes seront dans le contexte de persistance, et on ne verra donc pas la mise à jour du salaire pour ces entités (même si la mise à jour à bien été effectuée en base de données). Si par ailleurs on opère des modifications sur les entités dans le contexte de persistance, alors il est possible de se retrouver dans un état inconsistant.

Ainsi, soit il faut faire la mise à jour en volume au tout début de la transaction, soit il faut faire appel à la méthode **refresh()** de l'interface **EntityManager**. Cette méthode prend en paramètre une entité qui est dans le contexte de persistance. Si l'on suspecte que des changements ont eu lieu dans la base de données mais ne sont pas répercutées sur une entité gérée dans le contexte de persistance, la méthode **refresh()** permet de mettre à jour l'état de l'entité avec les données présentes dans la base. Voici un exemple d'utilisation de **refresh()** pour mettre à jour le contexte de persistance :

```
et.begin();
// Requête de sélection et récupération des entités
// Ces entités font alors partie du contexte de persistance
List<Employee> employes = querySelect.getResultList();
// Mise à jour en volume qui concerne certaines entités
// du contexte de persistance (qui ne sont donc plus à jour)
queryUpdate.executeUpdate();
// On met à jour le contexte de persistance
for(Employee emp : employes) {
    em.refresh(emp);
}
et.commit();
```

3.7 Mise à jour des salaires des médecins

Question 7. Ajoutez dans le paquetage **tests** une classe **TestJPQL4** dans laquelle vous proposerez une méthode qui prend en paramètre un nom de service, et qui renvoie la liste des médecins de ce service. Dans la méthode principale, après avoir récupéré les médecins d'un service, vous mettrez à jour les entités afin d'augmenter leur salaire de 5% (on ne fera pas de mise à jour en volume). Testez que l'augmentation fonctionne bien et qu'elle a bien été prise en compte dans la base.

Question 8. Ajoutez dans le paquetage **tests** une classe **TestJPQL5**. Ce test doit permettre de mettre en évidence les problèmes liés à la mise à jour en volume. Dans une transaction, commencez par effectuer une requête de sélection afin de récupérer des médecins qui feront donc partie du contexte de persistance. Effectuez ensuite une mise à jour en volume afin de fixer les salaires de tous les médecins à 2000 euros. Afficher les médecins récupérés par la requête de sélection. Leur salaire a-t-il été mis à jour ? Mettez à jour le contexte de persistance avec la méthode **refresh()**, et vérifiez maintenant si les salaires des médecins récupérés par la requête de sélection sont à jour.

4 Participation des médecins dans les équipes

Nous allons à présent ajouter dans le modèle objet la classe **Equipe** définie par un identifiant et un nom. Une équipe est composée de médecins, et un médecin peut intervenir dans plusieurs équipes, il y a donc une association bidirectionnelle de type *Many-to-Many* entre les entités **Medecin** et **Equipe**. Pour cette association, on désire connaître la fonction du médecin dans l'équipe. Pour cela, on utilise la classe d'association **Participation**. Ainsi, nous souhaitons ici mettre en place le diagramme de classe de la Figure 4.

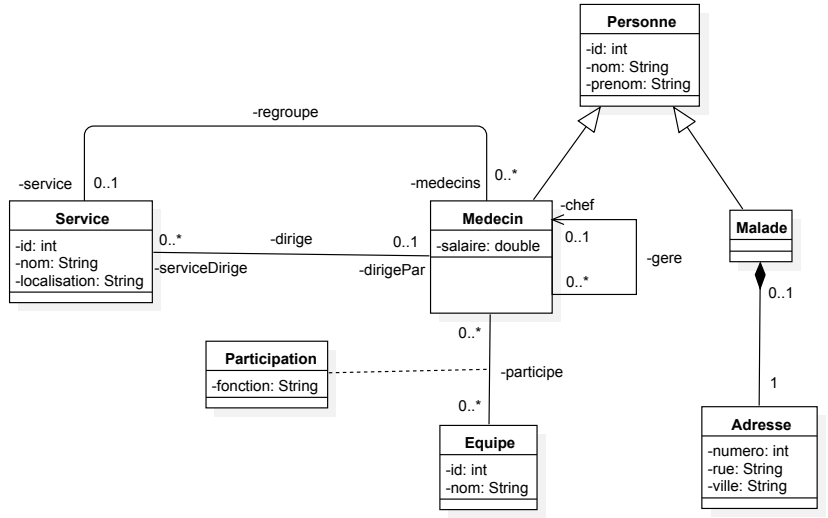

Figure 4: Diagramme de classe avec la participation des médecins aux équipes.

4.1 Modélisation

Si la classe d'association n'était pas présente, nous aurions une relation *Many-to-Many* entre les entités **Medecin** et **Equipe**.

Avec JPA, ce type d'association est défini en utilisant l'annotation **@ManyToMany** dans les deux entités aux extrémités de l'association. Une des entités indiquera avec l'attribut **mappedBy** dans l'annotation **@ManyToMany** le nom de l'attribut correspondant à l'association dans l'autre entité. Ensuite, au niveau de la base, les deux tables correspondant aux entités ne posséderont pas de clé étrangère. Dans le cas d'une association **ManyToMany**, on ajoute une table de jointure qui contiendra deux colonnes, chaque colonne étant une clé étrangère sur une des extrémités de l'association.

Dans le cas où il y a une classe d'association sur l'association de type *Many-to-Many*, au niveau de la base de données, la table liée à la classe d'association jouera le rôle de table de jointure. Ainsi, dans notre exemple, tout se passe comme si nous avions le diagramme UML de la Figure 5.

Dans ce cas, il n'y a plus explicitement d'associations *Many-to-Many*, mais des associations *One-to-Many* et *Many-to-One*.

4.2 Ajout des entités Equipe et Participation

Question 9. Ajoutez dans le paquetage **modele** une entité **Equipe**, et une

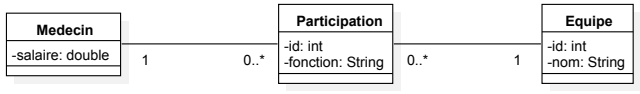

Figure 5: Une autre modélisation de la participation des médecins dans les équipes.

entité **Participation**, ainsi que les associations entre les entités **Medecin**, **Participation** et **Equipe**, comme présenté dans la Figure 5. N'oubliez pas de définir les constructeurs nécessaires. En particulier, pour **Participation** proposez un constructeur qui prend en paramètres un médecin, une équipe et une fonction ; et qui s'assure que les associations sont correctement réalisée pour modéliser la participation du médecin dans l'équipe. N'oubliez pas de redéfinir aussi les méthodes **equals()**, **hashCode()** et **toString()**.

Question 10. Dans l'unité de persistance, remettez-vous en mode "drop and create". Pour cela, choisissez **drop-and-create** pour la stratégie de génération des tables, et choisissez **drop-and-create-tables** pour la propriété **eclipselink.dll-generation**. Dans le paquetage **tests**, ajoutez une classe **Test4** dans laquelle vous ferez un test de persistance d'entités **Participation** et **Equipe**. On pourra reprendre ce qui a été fait dans la classe **Test3** et ajouter le code suivant après avoir déclaré les médecins :

```
Equipe eq1 = new Equipe("equipe 1");
Equipe eq2 = new Equipe("equipe 2");
Participation part1 = new Participation(med1, eq1, "chirurgien");
Participation part2 = new Participation(med1, eq2, "assistant chirurgien");
Participation part3 = new Participation(med2, eq1, "anesthésiste");
Participation part4 = new Participation(med3, eq2, "cardiologue");
Participation part5 = new Participation(med4, eq1, "interne");
Participation part6 = new Participation(med4, eq2, "interne");
```

Vérifiez que tout fonctionne correctement.

4.3 Quelques requêtes supplémentaires

Question 11. Désactivez le "drop and create" dans l'unité de persistance. Dans le paquetage **tests**, ajoutez une classe **Test5** dans laquelle vous proposerez trois méthodes pour :

- afficher la liste de toutes les personnes (malades et médecins) ;
- afficher la liste des médecins qui participent à au moins une équipe, avec pour chaque médecin les équipes dans lesquelles ils participent ;
- afficher la liste des médecins qui ne participent à aucune équipe.

Vous utiliserez pour cela des requêtes nommées. Pour information, avec JPQL, on peut savoir si une collection est vide avec l'opérateur *IS EMPTY*. Il est possible de visualiser les requêtes SQL effectuées sur la base de données. Pour cela, dans l'unité de persistance, activez la propriété **eclipselink.logging.level** avec la valeur **FINE**. Vérifiez que tout fonctionne correctement, et observez les requêtes SQL qui sont effectuées sur la base de données.

References

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