Construction d'une Couche de Persistance

Sébastien NEDJAR et Fabien PESCI

1 Introduction

Dans le premier TP nous avons construit une couche DAO pour simplifier et uniformiser l'accès aux données. L'énorme avantage d'une telle couche est de fournir une couche d'abstraction intermédiaire relativement générique et interchangeable. Ainsi le passage d'une technologie de persistance à une autre peut se faire en remplaçant uniquement cette couche intermédiaire. Nous avons aussi constaté que l'écriture des classes DAO simplistes avec JDBC était une tache longue, fastidieuse et rébarbative. L'objectif de ce TP est d'utiliser JPA pour simplifier l'écriture des DAO et aussi améliorer la fiabilité et la souplesse de notre couche de persistance.

Pour illustrer ce propos, nous utiliserons la base de données « Gestion Pédagogique [1] » que vous avez utilisée lors de vos TP de PL/SQL en début d'année. Le modèle conceptuel des données est rappelé par la figure 1. La figure 2 est une traduction du schéma Entité/Association en un diagramme de classe UML. Dans ce TP nous avons modifié la navigabilité de l'association "Est spécialiste" pour la rendre bi-directionnelle.

Avant de commencer ce TP, je suppose que vous avez compris les concepts du précédent et que vous avez suivi les tutoriels sur Maven et JPA.

2 Gestion de la persistance des données avec JPA

Avant de commencer à travailler, il faut que vous ayez créé grâce à Maven un projet compatible avec JPA2.0 semblable à celui du tutoriel. Vous trouverez à cette adresse http://bit.ly/TP_jpa un squelette de projet pour ce TP. Pour l'utiliser, il vous suffit d'extraire l'archive. Puis à partir d'Eclipse, allez dans le menu `File->Import...`, choisir `Maven->Existing Maven Projects`. Dans le dialogue sélectionnez le dossier TpJPA, cliquez sur le POM du projet et validez.

2.1 Configuration de l'unité de persistance

Comme toujours avec un projet JPA, vous devez commencer par configurer les paramètres d'accès à la base de données. Nous allons comme dans le TP précédent utiliser préférentiellement le serveur MySQL installé localement sur les machines du département. Pour le configurer correctement avant chaque séance lancez la séquence de commandes suivante :

```
mysql --user=root --password=mysql --execute="create database gestionPedaBD"
mysql --user=root --password=mysql --execute="grant all privileges on
    gestionPedaBD.* to monUser@localhost identified by 'monPassword'"
mysql --user=monUser --password=monPassword gestionPedaBD --execute="source
    gestion_peda_mysql.sql"
```

Le fichier `src/main/resources/META-INF/persistence.xml` devra ressembler à ceci :

```
<?xml version="1.0" encoding="UTF-8" ?>

<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd
      "
  version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">
```

1. Script de régénération disponible à l'adresse suivante : http://allegro.iut.univ-aix.fr/~nedjar/gestion_peda_oracle.sql ou http://allegro.iut.univ-aix.fr/~nedjar/gestion_peda_mysql.sql

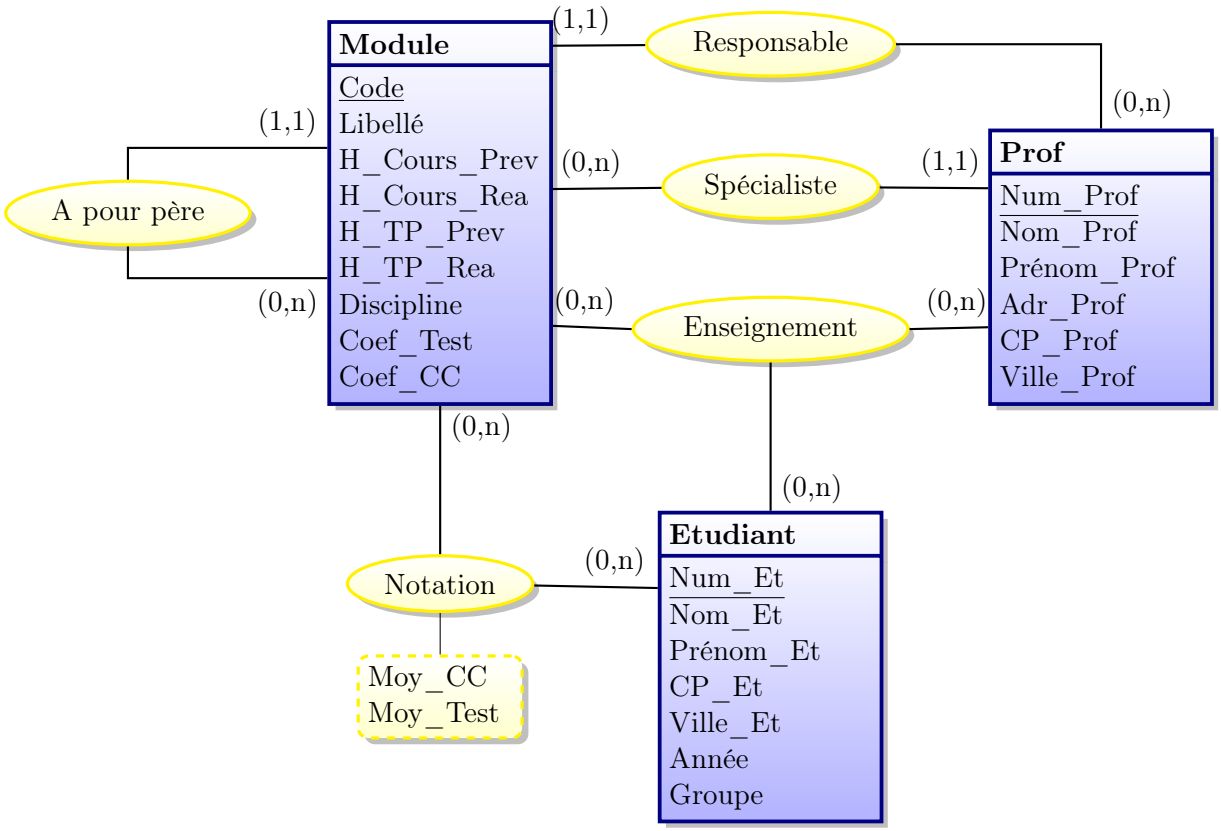

FIGURE 1 – Modèle conceptuel des données de la base « Gestion Pédagogique »

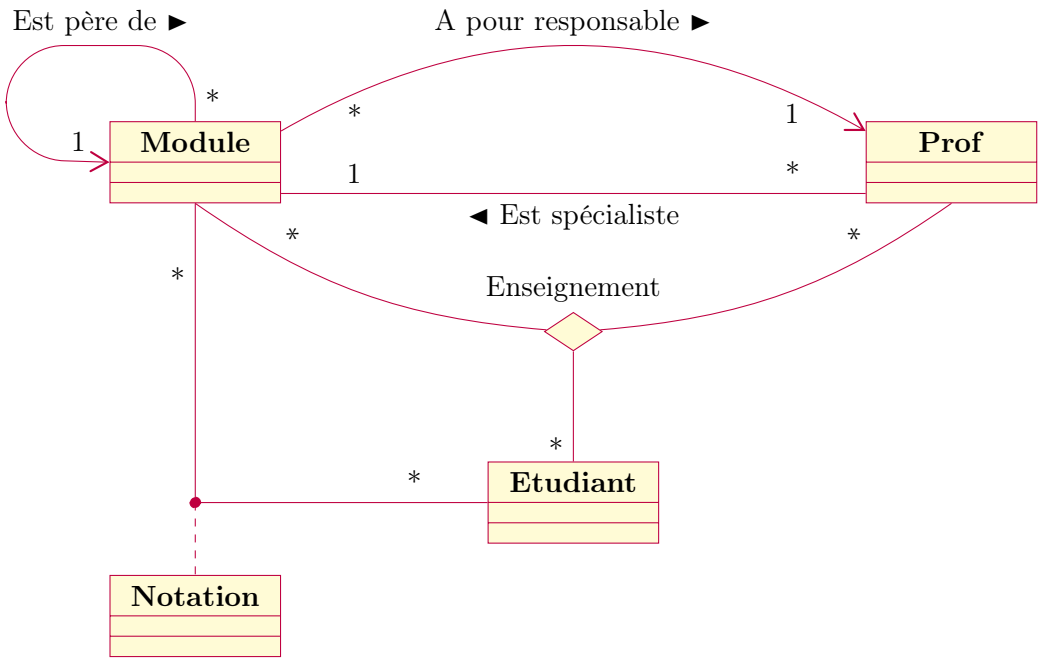

FIGURE 2 – Traduction UML du modèle conceptuel des données de la base « Gestion Pédagogique »

```
  <persistence-unit name="gestionPedaPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>fr.univaix.iut.progbd.beans.Etudiant</class>
    <class>fr.univaix.iut.progbd.beans.Module</class>
    <class>fr.univaix.iut.progbd.beans.Prof</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://
          localhost:3306/gestionPedaBD"/>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.
          Driver"/>
      <property name="javax.persistence.jdbc.user"  value="monUser"/>
      <property name="javax.persistence.jdbc.password"  value="monPassword"/>
      <property name="eclipselink.logging.level" value="ALL"/>
<!--  <property name="eclipselink.ddl-generation"  value="create-tables"/> -->
    </properties>
  </persistence-unit>
</persistence>
```

Remarquez que contrairement au tutoriel, nous ne demandons pas à JPA de nous créer les tables puisqu'elles existent déjà. L'une des principales difficultés de ce TP sera de configurer le mapping des entités pour qu'il corresponde aux relations présentes dans la BD. Comme vous allez le voir, la BD ne satisfaisant pas à la convention par défaut de JPA, il faudra rajouter beaucoup d'annotations.

2.2 Annotations des entités

La configuration du mapping dans JPA peut se faire grâce au mécanisme des annotations Java. Une annotation est une information permettant d'enrichir la sémantique d'une classe sans modifier son code. Dans le tutoriel nous avons déjà rencontré plusieurs d'entre elles :

- `@javax.persistence.Entity` permet à JPA de reconnaître cette classe comme une classe persistante (une entité) et non comme une simple classe Java.
- `@javax.persistence.Id`, quant à elle, définit l'identifiant unique de l'objet. Elle donne à l'entité une identité en mémoire en tant qu'objet, et en base de données via une clé primaire. Les autres attributs seront rendus persistants par JPA en appliquant la convention suivante : le nom de la colonne est identique à celui de l'attribut et le type String est converti en `VARCHAR(255)`.
- `@javax.persistence.GeneratedValue` indique à JPA qu'il doit gérer automatiquement la génération automatique de la clef primaire.
- `@javax.persistence.Column` permet de préciser des informations sur une colonne de la table : changer son nom (qui par défaut porte le même nom que l'attribut), préciser son type, sa taille et si la colonne autorise ou non la valeur null.
- `@javax.persistence.Embedded` précise que la donnée membre devra être intégrée dans l'entité.
- `@javax.persistence.Embeddable` précise que la donnée membre peut être intégrée dans une entité.
- L'annotation `@javax.persistence.OneToOne` indique à JPA que la donnée membre est une association 1 :1.
- L'annotation `@javax.persistence.ManyToOne` indique à JPA que la donnée membre est une association N :1.
- L'annotation `@javax.persistence.OneToMany` indique à JPA que la donnée membre est une association 1 :N.
- L'annotation `@javax.persistence.ManyToMany` indique à JPA que la donnée membre est une association N :M.

Question 1 : Annoter les classes `Etudiant`, `Prof` et `Module` pour qu'elles soient correctement mises en correspondance avec leur relation respective. Dans cette question les associations seront

ignorées pour le moment. Pour pouvoir tester votre code, il faut rajouter l'annotation `@Transient` devant la déclaration de la donnée membre matérialisant l'association.

Question 2 : Copier la classe `App` dans la nouvelle classe `afficheEntite`. Modifier le code de cette classe pour qu'elle récupère l'étudiant dont le `NUM_ET` est 1106 et l'affiche sur la sortie standard.

Question 3 : Copier la classe `App` dans la nouvelle classe `ajouteEntite`. Modifier le code de cette classe pour qu'elle crée un nouveau module, l'affiche, attend 1 minute et le supprime de la base de données. Vérifier en parallèle dans la console SQL que l'ajout et la suppression ont bien lieu.

2.3 Mapping des associations

Dans le cours et dans le TP précédent, nous avons vu comment les associations étaient implémentées en Java. En UML les associations ont une propriété supplémentaire, la navigabilité. En effet, contrairement au relationnel, une association peut n'être accessible qu'à l'une de ses extrémités. De manière générale, une association bidirectionnelle peut être vue comme une paire d'associations unidirectionnelles de sens opposé. Chacune d'elle pouvvent, comme dans le contexte BD, être classées par rapport aux cardinalités en UML de chaque rôle :

- Many-to-one pour les associations hiérarchiques (au sens du MCD) qui à une entité départ associe au plus une seule entité cible et une entité cible peut être associée à plusieurs sources.
- One-to-many pour les associations hiérarchiques (au sens du MCD) qui à une entité départ associe plusieurs entités cibles et une entité cible peut être associée à une source.
- One-to-one pour les associations hiérarchiques (au sens du MCD) qui à une entité départ n'associe qu'une seule entité cible et une entité cible est associé à au plus une source.
- Many-to-many pour les associations qui à une entité départ associe plusieurs entités cibles et une entité cible peut être associée à plusieurs sources.

Du point de vue de la source, les deux premières peuvent être implémentées par une simple donnée membre pointant vers l'entité associée. Elles sont donc appelées associations monovaluées (ou Single-valued associations). Les deux dernières doivent utiliser une collection pour matérialiser tous les liens, on les nomme associations multivaluées (ou Collection-valued associations).

2.3.1 Single-valued associations

Dans le diagramme de classe UML de la figure 2, il y a trois associations monovaluées : *"Est père de"*, *"Est spécialiste"* du point de vue de la classe `Prof` et *A pour responsable* du point de vue de la classe `Module`.

Ce type d'association s'implémente avec l'annotation `@ManyToOne` comme la relation *"à pour département"* du tutoriel. En relationnel, elles s'implémentent par l'ajout d'une clef étrangère du coté de la source de l'association. La convention en JPA pour nommer cet attribut est `<nom de la clef primaire>_<nom de la table d'origine>`. Par exemple la relation `DEPARTEMENT` a une clef appelée `ID` donc la clef étrangère dans `EMPLOYE` s'appelle `ID_DEPARTEMENT`. Dans le cas où la clef n'a pas le nom conventionnel, il faut préciser le nom de l'attribut clef étrangère (aussi appelé attribut de jointure) avec l'annotation `@JoinColumn`. Celle-ci possède un attribut `name` comme `@Column`. Elle doit être placée juste devant la donnée membre matérialisant l'association.

Question 4 : Annoter les entités pour que les associations *"Est père de"*, *A pour responsable* et *"Est spécialiste"* (pour l'instant on considère cette dernière comme unidirectionnelle).

Question 5 : Copier la classe `App` dans la nouvelle classe `afficheResponsable`. Modifier le code de cette classe pour qu'elle récupère la matière BD et affiche l'enseignant qui en est responsable sur la sortie standard. Soyez vigilent que vos méthodes `toString()` n'essaient pas d'afficher tout le graphe des objets.

Question 6 : Copier la classe `App` dans la nouvelle classe `afficheHierarchieModules`. Modifier le code de cette classe pour qu'elle récupère la matière BD et affiche récursivement la hiérarchie des modules de cette matière. Observer la console pour bien comprendre comment EclipseLink charge chacun des modules de cette hiérarchie.

Pour éviter que toutes les associations soient chargées dès la première utilisation, les annotations `OneToOne`, `ManyToOne`, `OneToMany` et `ManyToMany` ont un attribut `fetch` qui permet de demander d'effectuer un chargement à la demande (si `fetch=FetchType.LAZY`) ou un chargement immédiat (si `fetch=FetchType.EAGER`). Cette possibilité évite par exemple que l'ensemble des modules et leur responsable soit chargé lorsque l'on charge une seule matière.

Dans le cas des associations 1 :1 bidirectionnelle, pour paramétrer le second coté (celui qui ne possédera pas nécessairement de clef étrangère dans la BD), il faut rajouter à l'annotation `@OneToOne` l'attribut `mappedBy`. Il indique le nom de la donnée membre utilisée par l'entité liée pour matérialiser l'association dans le sens opposé.

2.3.2 Collection-valued associations

Comme indiqué ci-dessus, les associations 1 :N et M :N doivent associer à une entité plusieurs autres. Dans ce cas, la donnée membre matérialisant l'association devra être une collection. Par exemple dans le tutoriel si nous souhaitons rendre l'association *"à pour département"* navigable dans le sens `Departement` vers `Employe`, nous devons rajouter dans la classe `Departement` une donnée membre et les accesseurs suivants :

```
private Collection<Employe> employes;
public Collection<Employe> getEmployes() {
    return employes;
}
public boolean add(Employe e) {
    return employes.add(e);
}
public boolean remove(Object o) {
        return employes.remove(o);
}
```

Pour paramétrer cette association, il faut lui rajouter l'annotation `@OneToMany` à laquelle on doit préciser le type de l'entité lié et le nom de la donnée membre matérialisant l'association dans l'autre sens. Pour notre exemple, la déclaration de `employes` deviendrait :

```
@OneToMany(targetEntity=Employe.class, mappedBy="departement",
   fetch=FetchType.EAGER)
private Collection<Employe> employes;
```

Notons qu'il y a deux points importants à se souvenir quand on définit une association `one-to-many` bidirectionnelle :

- Le coté many-to-one est le propriétaire de l'association, l'attribut de jointure (`@JoinColumn`) est donc situé de ce coté.
- Le coté one-to-many est le coté inverse donc l'attribut `mappedBy` doit être utilisé.

Si l'on oublie de spécifier l'attribut `mappedBy` dans l'annotation `@OneToMany`, JPA considérera l'association `one-to-many` comme unidirectionnelle et s'attendra à trouver une table de jointure dont le nom est constitué de la concaténation des noms des entités liées.

Question 7 : Modifier et annoter l'entité `Module` pour que l'association *"Est spécialiste"* soit bidirectionnelle.

Question 8 : Copier la classe `App` dans la nouvelle classe `afficheSpécialistes`. Modifier le code de cette classe pour qu'elle récupère la matière BD et affiche tous les spécialistes de cette matière.

Le dernier type d'association qu'il nous reste à étudier sont les associations many-to-many. Pour ces associations en relationnel on utilise des relations dites de jointure. De manière générale, elles sont naturellement bidirectionnelle à cause de cette implémentation relationnelle (même si l'on peut restreindre la navigabilité). Comme pour les associations `one-to-many` bidirectionnelles, l'un des cotés doit porter l'attribut `mappedBy` dans l'annotation `@ManyToMany`. Le coté qui porte cet élément est dit coté inverse alors que le coté qui en est dépourvu sera le propriétaire. Chaque coté de l'association doit être doté d'une collection la matérialisant.

Le paramétrage de la table de jointure utilisée se fait grâce à l'annotation @JoinTable du coté de l'entité propriétaire. Cette annotation possède trois attributs importants :

- `name` qui indique le nom de la table de jointure.
- `joinColumns` qui spécifie l'attribut de la table de jointure qui est une clef étrangère vers l'entité propriétaire. Cet élément prend pour valeur une annotation `@JoinColumn` pour donner tous les paramètres de l'attribut de jointure.
- `inverseJoinColumns` qui spécifie le nom de l'attribut de la table de jointure qui est une clef étrangère vers l'entité située coté inverse. De même que `joinColumns`, cet élément se paramètre avec une annotation `@JoinColumn`.

Question 9 : Modifier et annoter les entités `Module` et `Etudiant` pour implémenter l'association bidirectionnelle *"Notation"* sans les attributs portés.

Question 10 : Copier la classe `App` dans la nouvelle classe `afficheNotation`. Modifier le code de cette classe pour qu'elle récupère la matière BD et affiche tous les étudiants ayant été notés pour cette matière.

La gestion des attributs portés et les associations ternaires avec JPA reposent principalement sur le principe de promotion d'une association en entité. Le travail demandé étant relativement important nous n'avons pas le temps de le faire dans les 4h allouées à ce TP. Pour ceux qui sont intéressés, le mapping complet et la correction seront disponibles sur mon dépôt git à la fin de cette semaine.

2.4 Construction de la couche d'accès aux données

Dans ce paragraphe, nous allons, comme dans le TP précédent, construire une couche dédiée à l'accès aux données qui utilisera le pattern DAO [2] (Data Access Object). Cette couche encapsulera tous les accès à la source de données. Les autres parties de l'application utiliseront uniquement les objets de cette couche pour gérer la persistance.

Chacune d'elles devra contenir des méthodes pour effectuer les 4 opérations de base pour la persistance des données : *créer, récupérer, mettre à jour et supprimer* [3]. Par convention, chacune des classes de DAO devra être nommée par `"DAO"` suivi due nom de la classe métier associée.

- `insert` qui a pour objectif de créer un nouvel étudiant dans la base de données. L'identifiant d'un tuple ne pouvant être connu avant son insertion, cette méthode retourne une copie de l'objet métier passé en paramètre avec un identifiant définitif. L'identité d'un objet dépendant uniquement de l'identifiant, un objet métier créé localement avec le constructeur par défaut (objet temporaire sans identité propre du point de vue de `equals()` et

2. http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html
3. Généralement désigné par l'acronyme anglais CRUD pour *Create, Retrieve, Update et Delete*

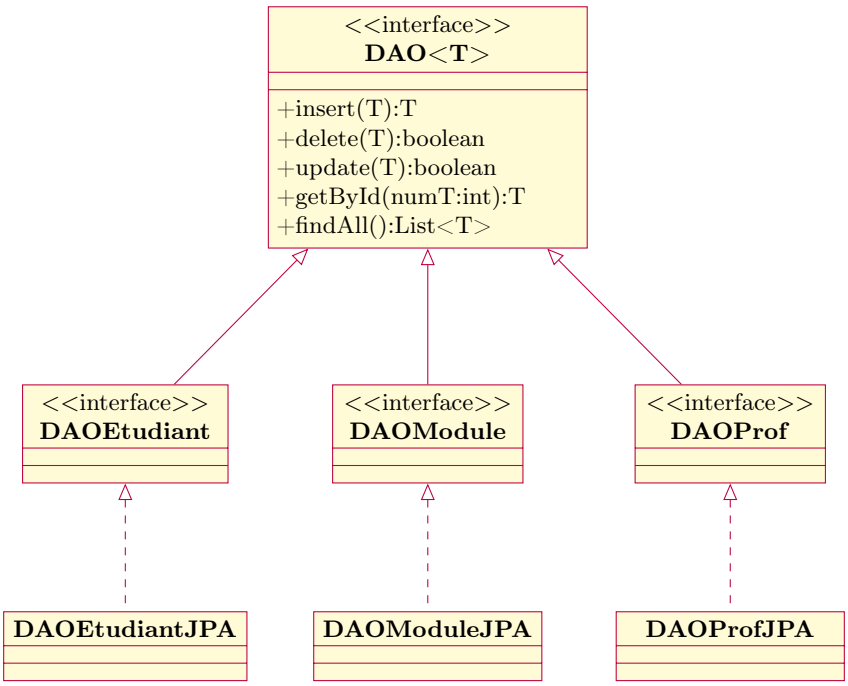

FIGURE 3 – Diagramme de classe de la couche DAO de l'application gestion pédagogique

`hashCode()`) ne devra participer à aucune association avant d'être inséré dans la base avec cette méthode [4].

- `update` qui prend un objet métier en paramètre et essaie faire la mise à jour dans la base de données. La valeur retournée par cette méthode indique si la mise à jour a pu avoir lieu.
- `delete` qui prend un étudiant en paramètre et essaie de le supprimer de la base de données. La valeur retournée par cette méthode indique si la suppression a pu avoir lieu.
- les `get` qui constituent, avec les `find`, les méthodes de récupération des données. Les paramètres passés à ces méthodes permettent de récupérer uniquement les tuples satisfaisants certains critères. La différence entre ces deux familles de méthodes est que les `get` doivent retourner exactement un seul résultat alors que les `find` peuvent en retourner plusieurs.
- les `compute` qui, comme leur nom l'indique, ont pour objectif d'effectuer des calculs sur les étudiants. La plupart du temps (sauf si le calcul demande de ne rapatrier aucune donnée) on préférera, pour des raisons d'efficacité, le faire directement dans le `Sgbd`. Ces méthodes sont donc soit des requêtes SQL agrégatives soit des appels de procédures stockées.

Tous les DAO de notre application ont un certain nombre de méthodes communes. Pour améliorer l'indépendance du code client vis à vis de la couche de persistance, nous ajoutons une interface `DAO` que tous les objets DAO devront implémenter. Les objets métiers dépendront ainsi d'une interface et non d'une implémentation particulière. La figure 3 donne le diagramme de classe de l'ensemble des DAO de l'application gestion pédagogique. Dans sa version complète (voir code présent sur le dépôt git), le pattern présenté utilise des `Abstract Factory` pour améliorer encore la modularité et l'indépendance de la couche de persistance.

Question 11 : Implémenter la classe `DAOEtudiantJPA` devant rendre les mêmes services que celle développé dans le TP précédent.

Question 12 : Implémenter toutes les classes `DAO`.

4. Ces objets sans identité jouent le rôle des objets de transfert de données (*Data Transfer Object*) du pattern DAO original.

Question 13 : Copier la classe `App` dans la nouvelle classe `afficheNotation`. Modifier le code de celle-ci pour que sa boucle principale récupère tous les étudiants de deuxième année, les affiche, puis affiche tous ceux qui ont été notés en « ACSI ».

Question 14 : L'écriture des opérations CRUD des DAO étant toujours identique, écrire la classe abstraite `DAOGeneriqueJPA<T, ID extends Serializable>` qui implémente `DAO<T>`. Modifier les DAO précédemment écrits pour qu'ils dérivent de cette classe. Vérifier que vos programmes fonctionnent toujours