Construction d'une Couche de Persistance

Sébastien NEDJAR et Fabien PESCI

1 Introduction

Le principal défaut que l'on peut reprocher à JDBC est d'être une API de bas niveau qui conduit à une trop forte imbrication entre le code métier et la base de données. Le code produit est donc trop peu modulaire et trop dépendant du SGBD choisit. Cela implique une moins grande maintenabilité et une plus grande dépendance face à une technologie de manipulation des données. Pour contourner cette difficulté, nous allons construire une couche dédiée à l'accès aux données (souvent appelée couche DAO). La construction d'une telle couche a pour objectif de séparer totalement les accès aux données du code de notre application. Les techniques présentées constituent une première introduction par la pratique aux solutions de persistance Objets/Relationnelle comme Hibernate [1] ou EclipseLink [2].

Pour illustrer ce propos, nous utiliserons la base de données « Gestion Pédagogique [3] » 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.

2 Couche de persistance

L'API JDBC permet de facilement récupérer et manipuler un ensemble de tuples récupéré à partir d'une base de données. Chaque tuple n'est pas simplement une concaténation de valeurs sans rapport les unes avec les autres mais un ensemble de valeurs structuré permettant de modéliser une « entité » de l'univers réel. Lorsqu'un tuple est récupéré à partir de JDBC, il faut donc impérativement conserver ce lien sémantique existant entre les attributs. C'est pour cela que chaque tuple de la base de données devra être associé (mappé) à un objet du langage de programmation. Une fois le mapping établi, l'objet commence son existence autonome comme n'importe quel autre objet de l'application. Son état (ensemble des valeurs de ses propriétés) sera très probablement mis à jour. Afin que ces changements soient visibles pour les autres utilisateurs, il faudra périodiquement synchroniser l'état de l'objet et de la base de données. Une fois les modifications sauvegardées, l'objet pourra être détruit car l'utilisateur peut à tout moment reconstruire un objet semblable à partir de la base de données. Le mapping objet/relationnel permet ainsi de rendre les objets de l'application persistants.

Dans la suite de cette section nous allons montrer une méthode pour créer « proprement » un tel mapping. La solution présentée est principalement pédagogique : elle ne sera en conséquence pas satisfaisante pour une solution déployée à grande échelle, mais sera amplement suffisante pour le développement d'une application mono-utilisateur (comme celle que vous devez réaliser en *Cas 2*).

2.1 Connexion à la base de données

Recréer une connexion pour chaque requête est inutile et coûteux. Pour éviter cela, il faut partager la connexion entre plusieurs traitements. La problématique est de savoir quelles sont les requêtes à exécuter ensemble. La solution classique est d'exécuter au sein d'une même connexion

1. http://www.hibernate.org/
2. http://www.eclipse.org/eclipselink/
3. 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

tous les traitements concourant à la réalisation d'un même objectif. Chacun de ces ensembles de traitements constitue une session. Une même session peut contenir plusieurs transactions (unités de traitement indivisibles) [4]. Tout ceci permet de gérer efficacement et intelligemment les problèmes de concurrence et de reprise après erreur. Réaliser une gestion réaliste des connexions, sessions et transactions demande un travail important que nous n'avons pas les moyens fournir. Notre application étant simple et mono-utilisateur nous utiliserons la méthode dite *« session-per-application »*, c'est à dire qu'il n'y aura qu'une seule connexion active à la fois et tous les objets devront se la partager. Le pattern singleton [5] est mis en œuvre pour que tous les objets de notre application puissent récupérer l'unique instance de la classe `Connection`.

Question 1 : Écrire la classe `ConnexionUnique` dont le diagramme UML vous est donné dans la figure 2. Copier la classe `testJDBC` dans la nouvelle classe `testConnexion`. Modifier le code de cette nouvelle classe pour qu'elle utilise notre objet `ConnexionUnique`.

2.2 Création des classes d'objets métiers

La création d'un mapping entre le « monde objet » et le « monde relationnel » nécessite au préalable la création de modèles de données semblables mais adaptés aux spécificités de chacun de ses mondes. L'objectif est donc de transformer le modèle relationnel de la base « Gestion Pédagogique » (schéma Entité/Association) en un modèle objet satisfaisant (diagramme de classes UML [6]).

La figure 3 est une traduction directe du schéma Entité/Association en un diagramme de classe UML. Chacun des concepts du schéma E/A a été transformé en son équivalent UML [7]. À partir de cette traduction, nous allons montrer les modifications à apporter à ce modèle pour le rendre implémentable.

2.2.1 Modélisation des types d'entités

Pour établir une correspondance entre une entité de notre BD et un objet de notre application, il faut commencer par écrire les classes associées à chacun des types d'entités. Pour des raisons qui apparaitront plus tard, chaque classe métier devra suivre les conventions suivantes :

- La classe doit être « sérialisable » [8] (*i.e.* implémenter l'interface `Serializable`) pour pouvoir sauvegarder et restaurer l'état des instances de cette classe ;
- La classe doit posséder un constructeur sans argument (constructeur par défaut) ;
- Les propriétés privées de la classe (variables d'instances) doivent être accessibles publiquement via des méthodes accesseurs construites avec `get` ou `set` suivi du nom de la propriété avec la première lettre transformée en majuscule (voir le menu `Source` → `Generate Getters and Setters...` dans Eclipse).
- La classe doit surcharger la méthode `toString()` pour pouvoir afficher l'état des instances de cette classe (voir le menu `Source` → `Generate toString()` dans Eclipse).
- La classe doit aussi surcharger les méthodes `equals()` et `hashCode()` héritées de `Object` [9] (voir le menu `Source` → `Generate hashCode() and equals()...` dans Eclipse). Deux objets métiers sont égaux si et seulement si les tuples associés sont égaux dans la base de données (*i.e.* même valeur de clef).

4. http://fr.wikipedia.org/wiki/Transaction_informatique
5. http://fr.wikipedia.org/wiki/Singleton_%28patron_de_conception%29
6. http://uml.free.fr/
7. Le losange est le symbole matérialisant les associations n-aires ($n > 2$). `Notation` est une classe dite d'association. Elle permet de modéliser les attributs portés par une association.
8. http://fr.wikipedia.org/wiki/S%C3%A9rialisation
9. http://download.oracle.com/javase/6/docs/api/java/lang/Object.html

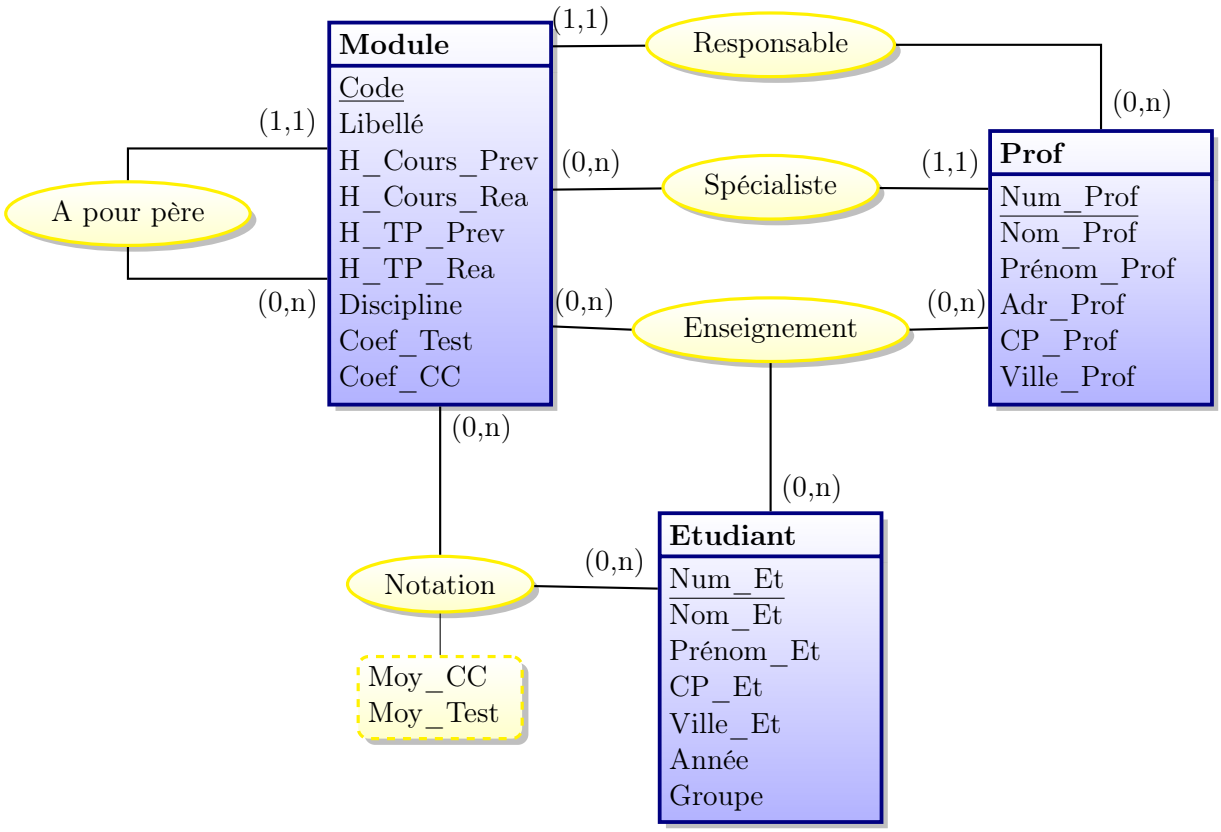

FIGURE 1 – Modèle conceptuel des données de la base « Gestion Pédagogique »

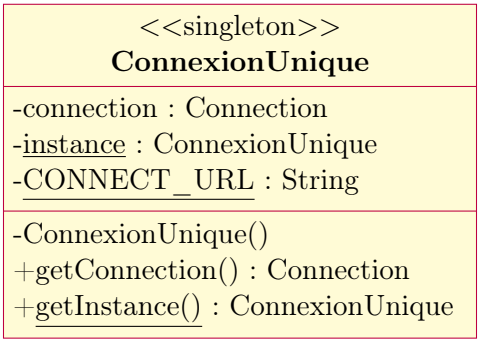

FIGURE 2 – Classe `ConnexionUnique`

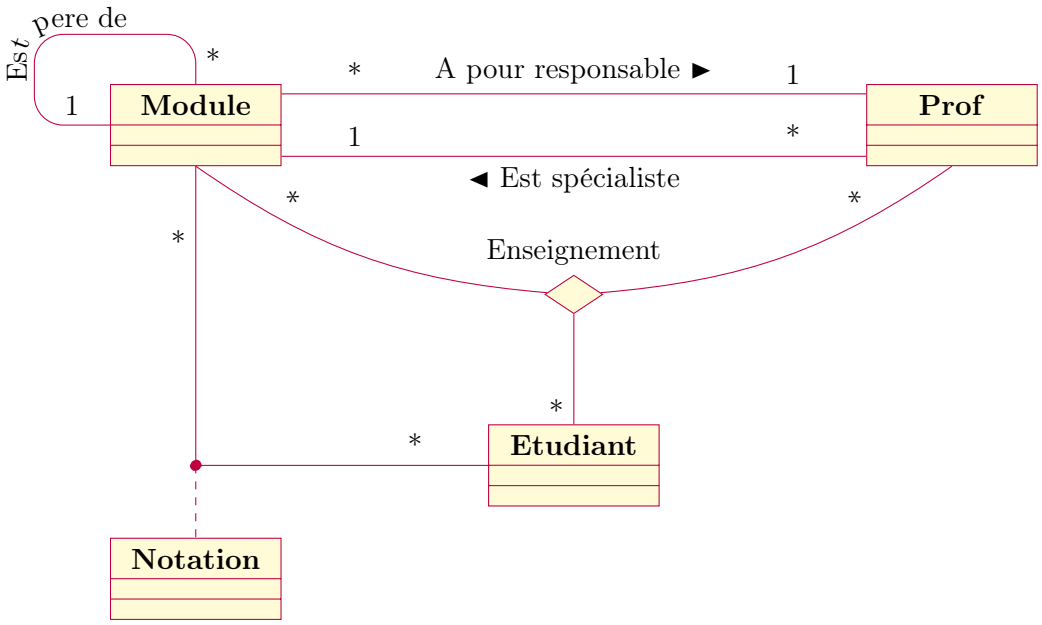

FIGURE 3 – Traduction UML du modèle conceptuel des données de la base « Gestion Pédagogique »

Module
-code : String
-libelle : String
-hCoursPrev : int
-hCoursRea : int
-hTpPrev : int
-hTpRea : int
-discipline : String
-coefTest : int
-coefCc : int

Prof
-numProf : int
-nomProf : String
-prenomProf : String
-adrProf : String
-cpProf : String
-villeProf : String

Etudiant
-numEt : int
-nomEt : String
-prenomEt : String
-cpEt : String
-villeEt : String
-annee : int
-groupe : int

FIGURE 4 – Classe `Etudiant`, `Module` et `Prof`

Question 2 : Implémenter (en respectant les conventions ci-dessus) les classes `Etudiant`, `Module` et `Prof` dont le diagramme UML incomplet vous est donné dans la figure 4. Copier la classe `testJDBC` dans la nouvelle classe `testEntite`. Modifier le code de cette classe pour que sa boucle principale remplisse un `ArrayList` [10] d'objets `Etudiant` et qu'elle affiche le contenu de cette liste en utilisant la méthode `toString()`.

2.2.2 Modélisation des types d'association hiérarchiques

Dans la figure 3 tous les types d'association binaires du MCD ont été représentés par leurs équivalents en UML (Booch et al., 1999; Rumbaugh et al., 1999). Ces associations UML sont symbolisées par un trait liant deux classes. Les multiplicités (les nombres situés aux extrémités de l'association) correspondent aux cardinalités du MCD mis à part qu'elles sont placées à l'inverse. Par exemple, pour indiquer qu'une classe `A` peut participer 0 ou 1 fois à une association avec la classe `B`, on placerait la multiplicité `0..1` du coté de `B`. UML permet d'écrire certaines multiplicités de manière simplifiée : `0..*` devient `*` et `1..1` devient `1`.

Par défaut les associations sont bidirectionnelles, cela signifie qu'une instance à l'une des extrémités peut savoir avec quelles autres instances elle est liée par cette association. Dans la

10. http://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html

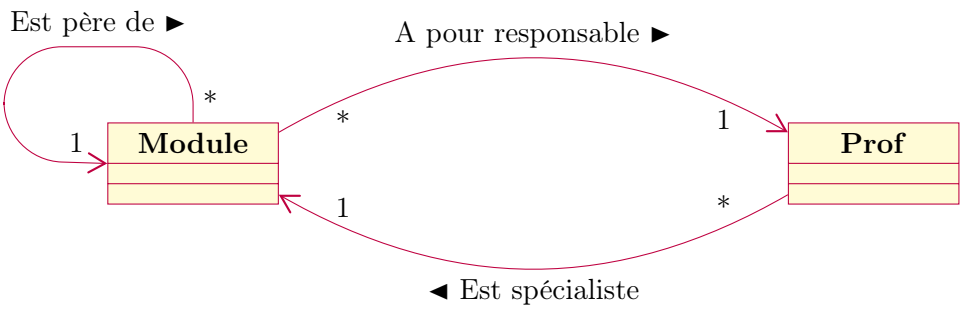

FIGURE 5 – Association à navigabilité restreinte

pratique, ce double lien peut être coûteux à maintenir, c'est pourquoi UML permet de privilégier un seul sens en interdisant l'accès dans l'autre. C'est ce que l'on appelle la restriction de la navigabilité d'une association. Elle est symbolisée par une flèche indiquant le sens de navigation permis.

La figure 5 montre le sens de navigation des trois types d'association hiérarchiques de la base « Gestion Pédagogique ». Les sens de navigation choisis imposent que :

- pour chaque instance de la classe `Prof` on connaitra le `Module` pour lequel il est spécialiste mais pour un `Module` on ne peut pas savoir quels sont les `Prof` spécialistes ;
- pour chaque instance de la classe `Module` on connaitra le `Prof` responsable mais pour un `Prof` on ne peut pas savoir quels sont les `Modules` dont il est responsable ;
- pour chaque instance de la classe `Module` on connaitra son `Module` père mais pour un `Module` donné on ne peut pas retrouver l'ensemble de ses fils.

Question 3 : Implémenter en respectant le sens de navigation imposé l'association *« est spécialiste »* entre `Prof` et `Module`. Un objet `Prof` n'étant associé qu'à un seul `Module`, il suffit d'ajouter à la classe `Prof` un attribut `specialite` (sans oublier les accesseurs associés) qui est une référence vers un `Module`. Il permet de lier un objet `Prof` à sa spécialité.

Faire de même pour les deux autres TA hiérarchiques en respectant à chaque fois les sens de navigation de la figure 5. Copier la classe `testJDBC` dans la nouvelle classe `testAsso1`. Modifier le code de cette classe pour remplir un `ArrayList` d'objets `Prof`. Pour chacun d'eux construire un objet `Module` représentant sa spécialité et conserver une référence vers cet objet dans l'attribut `specialite`. Afficher chacun des profs et le module dont il est spécialiste.

2.2.3 Modélisation des types d'association non hiérarchiques

Contrairement aux types d'association hiérarchiques qui peuvent être implémentés simplement par des références (pointeurs en $C++$), les types d'association non hiérarchiques nécessitent une structure supplémentaire (Milicev, 2007; Génova et al., 2003; Barbier et al., 2003). Nous allons présenter trois manières d'implémenter ces types d'association : les collections de pointeurs de chaque coté de l'association, les objets d'association et la promotion d'une association en classe. Chacune de ces méthodes d'implémentation a des avantages et des inconvénients qu'il faudra prendre en compte avant de faire un choix.

Collections de pointeurs aux extrémités de l'association : La première méthode est en quelque sorte une extension de la technique d'implémentation du paragraphe précédent. Pour simplifier la présentation, cette approche est appliquée dans un premier temps sur l'association `Notation`, dans laquelle on ne considère pas les données portées par l'association (*cf.* figure 6). L'implémentation complète (en rajoutant la classe d'association) de cette association sera faite dans un second temps.

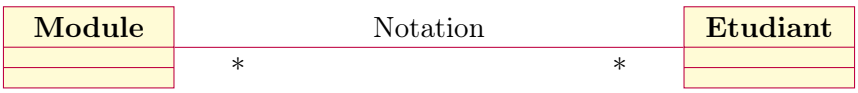

FIGURE 6 – Association `Notation` sans considérer la classe d'association

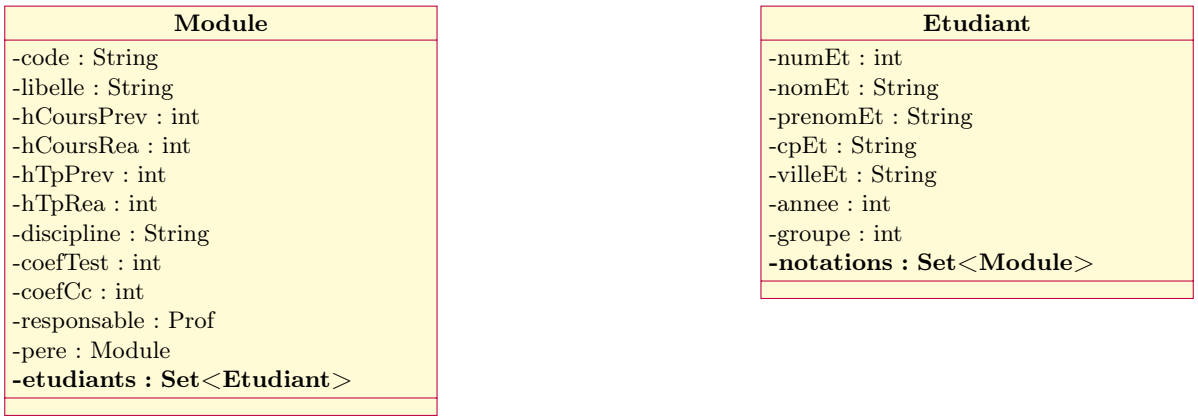

FIGURE 7 – Classes `Module` et `Etudiant` utilisant deux `Set`

Dans le cas de l'association « `est spécialiste` » où un `Prof` n'était lié qu'à un seul `Module`, il a suffi d'ajouter dans `Prof` une référence vers une instance de `Module`. Ici, un `Etudiant` peut être lié à plusieurs `Module`. On ajoute donc non pas une seule référence, mais un ensemble (ou collection) de références, nommé `notations`, vers des objets `Module`. Cette collection doit être d'un type implémentant l'interface `Set`[11] tel que `HashSet` ou `TreeSet`. Cette contrainte garantit l'unicité des objets contenus dans la collection. Ainsi, un même `Etudiant` ne peut pas être lié plusieurs fois à un même `Module`, ce qui indispensable pour modéliser correctement une association.

Aucun sens de navigation n'étant privilégié, il faut rajouter de manière symétrique une collection appelée `etudiants` dans `Module`. Cet ensemble de références vers des objets `Etudiant` rend possible la navigation dans le sens inverse. Le diagramme de la figure 7 décrit les changements apportés aux classes `Etudiant` et `Module` pour implémenter l'association.

L'implémentation proposée permet de savoir à quel `Module` un `Etudiant` est lié (et inversement) mais elle ne permet pas d'ajouter des informations supplémentaires aux liens. Pour implémenter l'association comme dans la figure 8, il faut prendre en compte la classe d'association `Notation`. Les ensembles de références sont remplacés par des dictionnaires (des conteneurs associatifs) pour atteindre cet objectif. Un dictionnaire peut être globalement perçu, d'un point de vue fonctionnel, comme une sorte de tableau indexable par n'importe quel type d'objet (plus

11. Les spécificités de l'interface `Set` sont présentées sur la page suivante : http://java.sun.com/developer/onlineTraining/collections/Collection.html#SetInterface

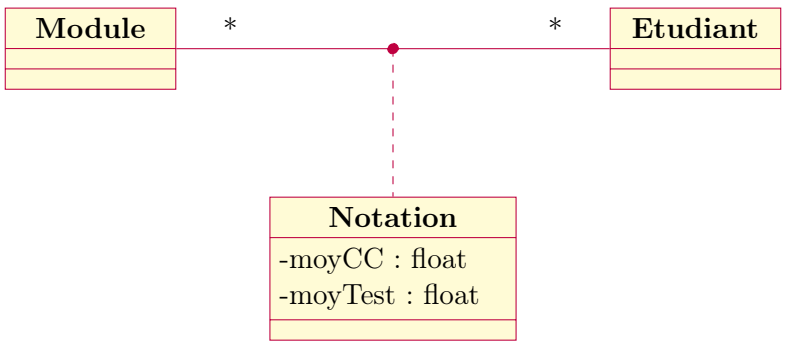

FIGURE 8 – Association `Notation` en considérerant les attributs portés

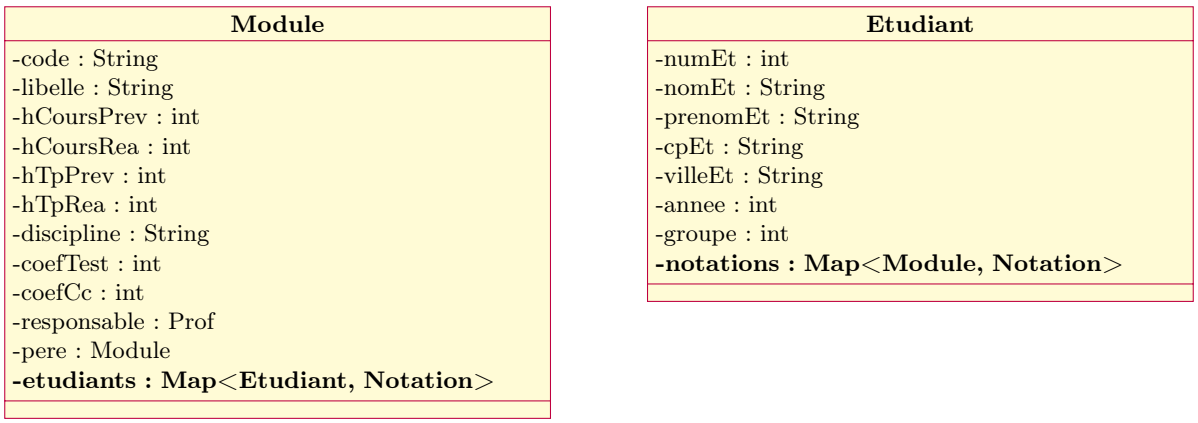

FIGURE 9 – Classe `Module`, `Etudiant` avec prise en compte des notations

seulement par des entiers). Malgré leur simplicité d'utilisation, ils ont un coût d'accès plus élevé qu'un tableau classique (Sedgewick and Schidlowsky, 1998). En Java, les conteneurs associatifs sont des classes implémentant l'interface `Map` [12] (tel que `HashMap`). Ces classes permettent d'associer un objet clef (l'objet servant d'index) à un objet valeur (n'importe quel autre objet). D'après le diagramme de classe de la figure 8, cet objet valeur sera une référence vers un objet de la classe `Notation`. Les modifications à apporter aux classes `Module` et `Etudiant` pour prendre en compte ces changements sont décrites dans la figure 9. Le lien `Notation` entre un `Etudiant` et un `Module` est ainsi représenté sous forme de collections (associatives) de pointeurs de part et d'autre de l'association.

Objets d'association : L'approche précédente est relativement simple à mettre en œuvre du point de vue des modifications à apporter aux différentes classes. La principale difficulté provient de l'interdépendance entre objets qu'elle introduit. En effet, chacun des objets participant à une association a la responsabilité de construire et de maintenir à jour sa propre liste de liens. Si l'on souhaite supprimer un objet, il faut avant cela supprimer cet objet dans chacune des listes des objets avec lequel il est lié. La responsabilité de la cohérence (réciprocité) d'un lien est partagée entre plusieurs objets de classes différentes, il y a donc éparpillement du code de gestion l'association ce qui implique un risque plus important d'erreur.

Dans notre cas (application avec objets persistants en BD) une telle approche n'est pas envisageable, car lorsque l'on doit rendre persistant un objet dans une base de données, cela implique de vérifier si les objets avec lesquels il est lié sont déjà stockés dans la base de données. Or, cette tache n'est pas du tout évidente d'un point de vue algorithmique et a un coût important s'il existe un grand nombre de liens.

La seconde solution (Gessenharter, 2009) consiste à créer un unique objet qui aura la responsabilité de conserver et gérer tous les liens d'une association. Cet objet ayant une vision globale des liens existants, il peut facilement supprimer tous les liens entretenus par un seul et même objet. De plus, un objet n'a plus à connaitre tous les objets qui lui sont liés mais uniquement l'objet association qui pourra retrouver au besoin tous ces liens. En quelque sorte, cette approche est une solution globale qui décharge les différents objets de la responsabilité de gérer chacun des liens localement. Pour rendre persistante une association, il suffit de stocker tous les objets connus par l'association avant de stocker l'objet d'association lui même.

L'implémentation d'un objet d'association se fait en utilisant un ensemble (`Set`) d'objet lien. Chaque objet lien est un n-uplet de références vers les différentes classes participant à l'association. La figure 10 donne le diagramme de classe de l'objet d'association `AssociationNotation`. Un `Lien` est dans notre cas un triplet d'étudiant, module et notation. Pour gérer correctement

12. Les spécificités de l'interface `Map` sont présentées à la page http://java.sun.com/developer/onlineTraining/collections/Collection.html#MapInterface

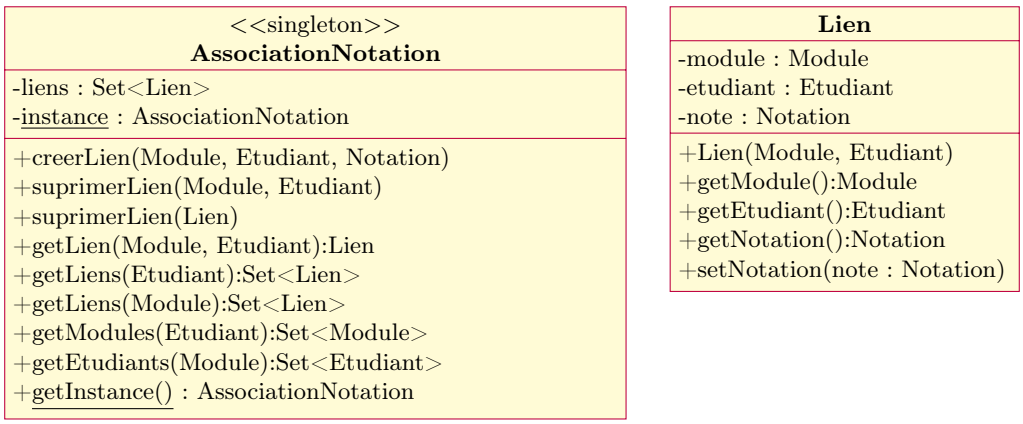

FIGURE 10 – Diagramme de classe de `AssociationNotation`

la contrainte d'unicité de l'association, la classe `Lien` doit surcharger les méthodes `equals()` et `hashCode()` héritées de `Object`[13]. Deux liens sont considérés comme égaux s'ils référencent le même étudiant et le même module (peu importe la note).

Question 4 : Implémenter l'association « `Notation` » entre `Etudiant` et `Module` en utilisant l'objet d'association `AssociationNotation`. Copier la classe `testJDBC` dans la nouvelle classe `testAsso2`. Modifier le code de cette classe pour charger toutes les notes des différents étudiants aux différents modules dans l'objet d'association `AssociationNotation`. Pour simplifier les traitements, penser à charger l'ensemble des étudiants et des modules à l'avance. Afficher les étudiants et leurs notes pour le module 'ACSI'.

Promotion d'une association en classe : La dernière approche présentée a pour objectif de simplifier le diagramme de classe pour contourner le problème des associations trop complexes à matérialiser. Comme nous venons de le voir, implémenter une association bidirectionnelle non-hiérarchique demande un travail important. Généralement lorsque l'on rencontre des associations n-aires (avec $n > 2$), l'une des techniques employées est de promouvoir cette association en une classe. Celle-ci sera liée par une association hiérarchique à chacune des classes participant à l'ancienne association. Cette modification du diagramme de classe modifie aussi partiellement sa sémantique. En effet, la contrainte d'unicité de l'association n'est plus vérifiée structurellement, la responsabilité de cette contrainte revient au code de l'utilisateur. Il faudra en être conscient avant de faire le choix d'utiliser cette solution.

Dans notre base de données « Gestion Pédagogique » il n'y a qu'une seule association ternaire : `Enseignement`(*cf.* figure 11). Elle sera donc notre support pour mettre en pratique cette technique. La figure 12 montre les modifications à apporter à cette association pour la promouvoir en classe.

Question 5 : Implémenter l'association « `Enseignement` » entre `Etudiant`, `Module` et `Prof` en utilisant le diagramme de classe donné par la figure 12. Modifier chacune des classes participantes pour que les associations A_i soient navigables dans les deux sens. Copier la classe `testJDBC` dans la nouvelle classe `testAsso2`. Modifier le code de cette classe pour charger tous les enseignements. Afficher tous les enseignements suivis par les étudiants du groupe 1.

2.3 Construction de la couche d'accès aux données

Les paragraphes précédents ont présenté comment construire le modèle objet miroir du modèle relationnel. L'objectif est maintenant d'écrire le code permettant de faire communiquer ces deux

13. http://download.oracle.com/javase/6/docs/api/java/lang/Object.html

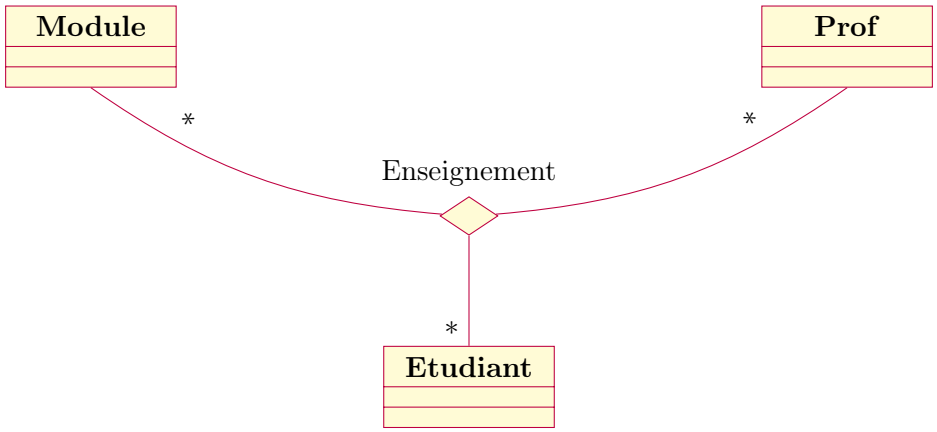

FIGURE 11 – Diagramme de classe de l'association ternaire `Enseignement`

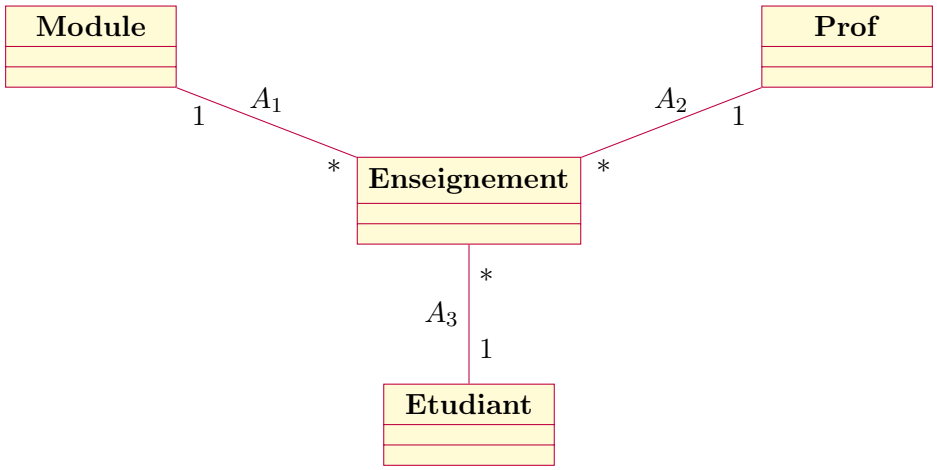

FIGURE 12 – Diagramme de classe de l'association ternaire `Enseignement`

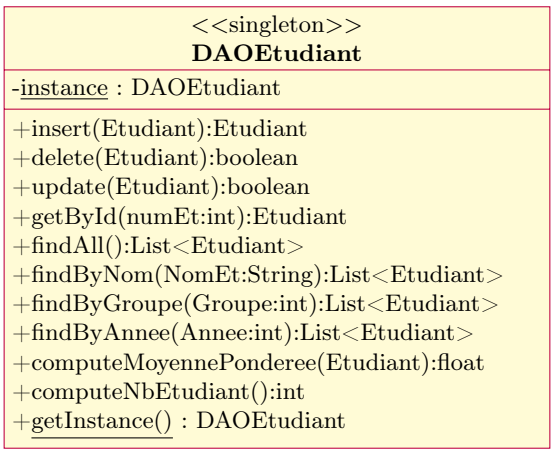

FIGURE 13 – Diagramme de classe de `DAOEtudiant`

modèles. Les questions ont mis en évidence la difficulté (et l'aspect répétitif) d'écrire un tel code avec JDBC . Utiliser directement JDBC à chaque accès aux données produirait deux effets très négatifs :

- Une pollution importante du code métier par du code JDBC. Cela implique donc une moins grande lisibilité du code et ainsi un risque d'erreur plus important.
- Une moins grande indépendance vis à vis du SGBD. L'intrication forte entre code métier et code d'accès au données rend le changement de SGBD (par exemple le remplacement de Oracle par Postgres) très délicat voir impossible.

Pour éviter ces problèmes, nous allons construire une couche dédiée à l'accès aux données qui utilisera le pattern DAO [14] (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. Elle sera donc une sorte d'abstraction du modèle de données indépendante de la solution de stockage des données. La couche DAO contiendra au moins autant de classes de DAO que d'entités du MCD (classe d'objet métier) [15].

La source de données étant une ressource unique, il n'existera qu'une seule instance de chacune des classes de DAO. Elles devront donc toutes être des singletons (*cf.* classes `ConnexionUnique` et `AssociationNotation`). 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* [16]. Par convention, chacune des classes de DAO devra être nommée par `"DAO"` suivi due nom de la classe métier associée. La figure 13 décrit la classe `DAOEtudiant` qui est le DAO associé à la classe d'objet métier `Etudiant`. Cette classe est constituée des méthodes suivantes :

- `insert` qui à 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 `hashCode()`) ne devra participer à aucune association avant d'être inséré dans la base avec cette méthode [17].
- `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.

14. http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html
15. L'écriture et la maintenance d'une telle couche est donc une opération généralement fastidieuse. C'est l'une des raisons pour lesquelles les solutions de persistance actuelles génèrent automatiquement une grande partie du code (Java et/ou SQL).
16. Généralement désigné par l'acronyme anglais CRUD pour *Create, Retrieve, Update et Delete*
17. Ces objets sans identité jouent le rôle des objets de transfert de données (*Data Transfer Object*) du pattern DAO original.

- `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.

En utilisant `DAOEtudiant`, la récupération par l'application de l'étudiant d'identifiant 1 dans la base de données se déroule comme suit :

1. L'application demande un objet `Etudiant` correspondant au tuple d'identifiant 1 dans la base de données à l'unique instance de `DAOEtudiant`.
2. L'objet `DAOEtudiant` récupère cette demande (méthode `getByID(1)`) et il s'occupe d'exécuter la requête SQL avec JDBC.
3. Le SGBD interprète la requête SQL et retourne le résultat attendu (s'il existe).
4. L'objet `DAOEtudiant` récupère ces informations.
5. L'objet `DAOEtudiant` instancie un objet `Etudiant` avec les données récupérées.
6. Enfin, l'objet `DAOEtudiant` retourne l'instance de l'objet `Etudiant`.

Cette séquence d'opération illustre bien le rôle central de l'objet DAO dans l'accès aux données. Les opérations de mise à jour et de suppression se dérouleront à peu près de la même manière. Pour l'insertion d'un nouveau tuple, il faudra d'abord créer un objet sans identité (avec le constructeur par défaut) puis appeler la méthode `insert()` qui nous retournera notre objet définitif (avec un identifiant valide). Le code ci-dessous illustre l'utilisation typique du DAO pour l'ajout d'un nouvel étudiant et sa modification :

```
public class Main {
  public static void main(String[] args){
    DAOEtudiant dao = DAOEtudiant.getInstance();
    Etudiant e = new Etudiant();//e est un Etudiant temporaire
    e.setNom("Dupont");
    e.setPrenom("Paul");
    e.setCp("13100");
    e.setVille("Aix-en-Provence");
    e.setAnnee(1);//Modification des attributs de e
    e.setGroupe(5);
    e = dao.insert(e);//e referencie maintenant un Etudiant definitif
    ...
    e.setAnnee(2);// Modification des attributs de e
    e.setGroupe(3);
    ...
    boolean updateOk = dao.update(e);//Sauvegarde des modifications
    ...
  }
}
```

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 14 donne le diagramme de classe de

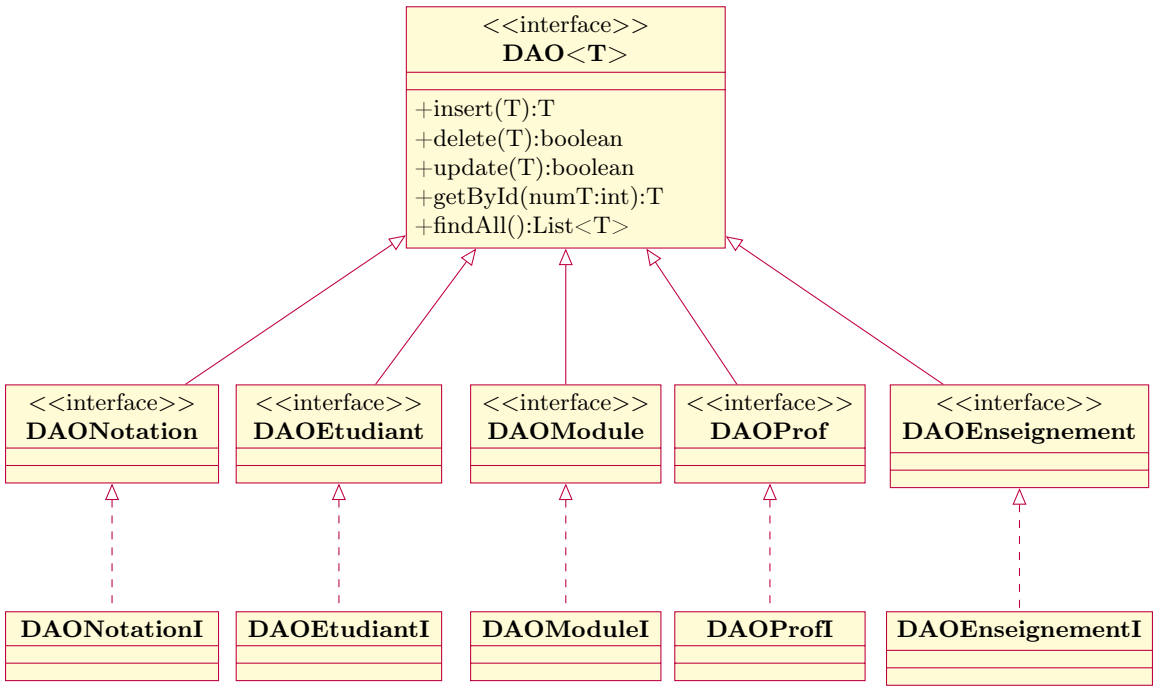

FIGURE 14 – Diagramme de classe de la couche DAO de l'application gestion pédagogique

l'ensemble des DAO de l'application gestion pédagogique. Dans sa version complète, le pattern présenté utilise des `Factory` pour améliorer encore la modularité de la couche de persistance.

Question 6 : Implémenter les classes `DAOEtudiant`[18] dont le diagramme de classe incomplet vous est donné par la figure 13.

Question 7 : Implémenter toutes les classes `DAO` en prenant en compte intéligement les associations existant entre les différentes classes métiers. Copier la classe `testJDBC` dans la nouvelle classe `testDAO`. Modifier le code de celle-ci pour que sa boucle principale récupère tous les étudiants de deuxième années, les affiche, puis augmente toutes leurs notes pour le module « ACSI » d'un point et enfin sauvegarde les résultats dans la base.

Références

Barbier, F., Henderson-Sellers, B., Le Parc-Lacayrelle, A., and Bruel, J.-M. (2003). Formalization of the whole-part relationship in the unified modeling language. *IEEE Trans. Softw. Eng.*, 29 :459–470.

Booch, G., Rumbaugh, J., and Jacobson, I. (1999). *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Gessenharter, D. (2009). Implementing uml associations in java : a slim code pattern for a complex modeling concept. In *Proceedings of the Workshop on Relationships and Associations in Object-Oriented Languages*, RAOOL '09, pages 17–24, New York, NY, USA. ACM.

Génova, G., Castillo, C. R. D., and Llorens, J. (2003). Mapping uml associations into java code. *JOURNAL OF OBJECT TECHNOLOGY*, 2(5) :135–162.

18. Pour ceux qui n'auraient pas terminé la première partie du TP, un squelette des classes métiers est disponible à l'adresse suivante : http://allegro.iut.univ-aix.fr/~nedjar/testJdbc.tar.bz2

Milicev, D. (2007). On the semantics of associations and association ends in uml. *IEEE Trans. Softw. Eng.*, 33 :238–251.

Rumbaugh, J., Jacobson, I., and Booch, G., editors (1999). *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., Essex, UK, UK.

Sedgewick, R. and Schidlowsky, M. (1998). *Algorithms in Java, Third Edition, Parts 1-4 : Fundamentals, Data Structures, Sorting, Searching*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition.