Mapping Object Relational (ORM)
Persistance des objets avec Hibernate dans Netbeans

Jacques THOORENS

1 Persistance des objets

Le modèle objet permet l'interaction d'une multitude d'objets qui s'envoient constamment des messages. Il s'agit d'un modèle idéal qui ne prend pas en compte deux contraintes du réel : l'espace mémoire dans lequel les objets évoluent n'est ni infini ni éternel. Se pose donc, à un moment donné, la question de la mémorisation des objets sur des supports stables. L'objet étant par nature éminemment changeant et interactif, cette mémorisation constitue un problème non trivial. Le modèle MVC permet d'opérer un premier tri des objets :

- les objets de type *view* sont en général relativement immuables au cours de leur vie. Les différences d'aspect qu'ils présentent résultent des caractéristiques d'autres objets dont ils donnent un aperçu. L'utilisateur pourra avoir envie de retrouver certaines préférences qu'il a paramétrées lors de son interaction avec le programme (types de classement, position de certains objets, couleur, aspect...). Par le biais de quelques paramètres habilement mémorisés dans l'environnement de chaque utilisateur, on pourra lui permettre de retrouver son environnement familier.
- les objets de *contrôle* sont par nature liés à des processus de traitement et n'ont plus de raison d'exister une fois le traitement terminé. Ce sont d'ailleurs les objets qui auront généralement la vie la plus courte.
- les objets qui constituent le *modèle*, on les désigne parfois aussi sous le nom d'objets-métier, représentent la majorité des objets qui doivent survivre à une exécution de l'application. Cette nécessité de survie sera donc une propriété de la classe, on parlera donc de classes *persistantes*.

On dispose de deux moyens pour enregistrer des objets : la sérialisation et l'utilisation de bases de données.

1.1 La sérialisation

La sérialisation est une opération de transformation d'un objet en une représentation écrite de telle sorte qu'une opération inverse puisse recréer un objet qui sera jugé identique. L'opération se réalisera facilement avec des objets simples, c'est-à-dire comportant quelques variables d'instance de type primitif. Dès que la classe comporte des variables d'instance de type objet, la sérialisation devient plus complexe, puisque la mémorisation de la seule référence d'un autre objet n'a aucun sens si l'objet est recréé dans un autre environnement (une nouvelle exécution du programme ou la recréation de l'objet ailleurs dans le réseau). L'interface `Serializable` permet de garantir que chaque objet référencé est lui aussi sérialisable. Mais cela implique des sérialisations en cascade, et malheureusement entraîne des problèmes de circularité. La sérialisation ne peut donc apporter une solution que pour des modèles simples dans lesquels les objets ne sont pas trop dépendants les uns des autres. Un objet n'est pas l'autre. On pourra plus facilement sérialiser un client qu'une facture.

La sérialisation pose également un autre problème, qui se présentera également dans l'utilisation d'une base de données : l'identité. Nous y reviendrons.

1.2 Utilisation d'une base de données orientée objet

Lorsque les objets deviennent trop nombreux ou entrent dans des associations trop complexes, la simple sérialisation est insuffisante. On va faire appel à des bases de données. Un choix stratégique se pose : quelle base de données choisir ? Le choix le plus simple, à première vue, consiste à utiliser une base de données orientée objet, seule à même de répondre efficacement aux problèmes de persistance. Plusieurs objections importantes vont venir au jour :

- ODMG est un standard défini pour la manipulation des bases de données objet. Il spécifie une API, un langage de requête, un langage de métadonnées et des associations avec différents langages. Malheureusement, il n'est implémenté complètement dans aucune base de données objet actuelle.
- De nombreux produits existent actuellement, commerciaux et Open source. Je citerai au hasard Matisse, db4Objects, Jade... Ils ont chacun leurs avantages, leurs faiblesses. Globalement, on peut leur reprocher un caractère plus expérimental que les bases de données relationnelles. En outre, sur un terrain en constante évolution, il est difficile d'assurer la pérennité des outils avec autant d'assurance que pour un des ténors relationnels.
- Le principal problème réside dans la difficulté à trouver des gestionnaires ayant une expertise d'un niveau comparable à celui des administrateurs de bases de données relationnelles.
- La nécessité de rentabiliser les investissements dans le domaine relationnel, en licence et en formation, conjuguée avec une relative incompatibilité des bases objets avec les applications traditionnelles constitue un dernier obstacle pour les entreprises ayant un parc logiciel ancien important.

Il faut reconnaître cependant que la situation rappelle le temps de l'émergence des bases de données relationnelles : l'avenir est probablement ouvert à des systèmes au moins partiellement objet. Les ténors du relationnel en sont d'ailleurs conscients et incorporent peu à peu des extensions allant dans ce sens.

1.3 Utilisation d'une base de données relationnelle

En attendant la généralisation des bases de données objet, force est de constater que l'utilisation d'une base de données relationnelle existante reste une solution plausible et souvent obligée. Cette solution n'est cependant que le début d'une série d'ennuis. Sans entrer dans trop de détails, le modèle relationnel et le modèle objet présentent pas mal de divergences qui rendent difficile le passage d'un modèle à l'autre :

La granularité

Traditionnellement, les bases de données relationnelles ont des types atomiques assez simples : nombres divers, chaînes et dates. Les objets ont des variables d'instance qui peuvent être d'autres objets. Ainsi, une facture possédera un attribut de type client, lui-même comportant un attribut adresse. Certains SGBDR permettent la définition de UDT (*user defined types*), c'est le cas d'Oracle, de MS SQL Server, mais aucun standard n'existe.

Problématique des sous-types

Les SGBDR ne fournissent aucun moyen de dériver une classe d'une autre. On devrait donc prévoir des tables différentes pour mémoriser des objets appartenant à des classes pourtant dérivées l'un de l'autre. En outre, le polymorphisme autorise une instance d'une sous-classe de figurer partout où figure une instance de sa classe-mère. Ce polymorphisme semble impossible à assurer dans une base de données, l'instance dérivée étant en principe plus « riche » que l'instance de la classe-mère.

L'identité

Un objet n'a pas besoin d'identité. L'utilisation de l'opérateur == suffit à reconnaître l'identité de deux instances. À partir du moment où on fait sortir une instance du programme, pour la faire renaître, l'identité n'est plus discernable. À l'inverse, nous savons que l'identifiant est une notion cardinale de la base de données (clé primaire). Il faut noter ici que quelque soit la solution adoptée, l'opérateur d'identité restera hors course et qu'il faudra que l'identité soit vérifié par une méthode surdéfinie : `equals()`.

Problématique des associations

Les associations entre tables sont par nature bidirectionnelles, ce qui constituait d'ailleurs une supériorité du modèle relationnel sur les modèles antérieurs où les associations figuraient sous la forme d'imbrication ou de parcours procédural (avec ou sans pointeurs). En outre, une association y est matérialisée une seule fois par une clé étrangère. Les associations sont de type un à un ou un à plusieurs. Les associations entre les objets sont directionnelles et dans les cas où la navigabilité est bidirectionnelle, cela se fait au prix d'une redondance. La multiplicité peut être de plusieurs à plusieurs.

Parcours d'un graphe d'objet

Lors d'une requête SQL, on va s'arranger pour savoir quelles associations d'une table donnée sont pertinentes pour le traitement en cours, et ainsi minimiser le nombre de jointures à réaliser. Dans une gestion objet, la récupération d'un objet devra le plus souvent anticiper les besoins et récupérer d'autres classes par le biais de plusieurs requêtes. Cela entraîne évidemment une dégradation des performances.

1.4 Le *mapping objet-relationnel*

Face à toutes ces solutions problématiques, il existe une dernière solution, qui reste la seule raisonnable actuellement, tant que les bases objet ne sont pas largement accessibles : l'utilisation d'un logiciel spécifique permettant de réaliser le *mapping* entre les objets et la base de données relationnelle. L'un des plus connus est Hibernate, conçu pour une intégration avec Java ou .net, mais il en existe d'autres, qui présentent des caractéristiques semblables. Ce logiciel présentera plusieurs avantages.

Automatisation d'un grand nombre de tâches

Par l'utilisation de métadonnées ou d'annotations placées dans la définition des classes, le système prend en main la gestion de la persistance, sans que le programmeur doive quasiment s'occuper de rien d'autre que l'inscription des objets dans le système et la gestion des sessions et des transactions. Ces métadonnées ne sont pas très lourdes à manipuler. En outre, on

pose très peu de contraintes sur les choix programmatiques lors de la réalisation des classes persistantes.

Garantie de meilleurs performances

Le logiciel est réalisé par des programmeurs spécialisés dans le ORM qui ont étudié tous les problèmes, envisagés les optimisations et tenu compte de remarques et critiques des utilisateurs des précédentes versions. Hibernate étant Open Source, on dispose également du code au cas où une optimisation critique serait nécessaire.

Indépendance par rapport au système de gestion

Le choix de la base de données et les paramètres de connexion sont fournis une seule fois et aucune routine n'en dépend. On peut donc facilement changer de base de données, lorsque l'application grandit ou change d'environnement, sans devoir réécrire la moindre ligne de code.

Disponibilité d'outils d'intégration dans les IDE

Hibernate dispose de plugins facilitant son intégration dans les grands IDE du marché (NetBeans et Eclipse). La gestion des métadonnées se fait de manière presque automatique.

2 Installation d'un environnement léger

Les démonstrations et tutoriels présents tant sur la toile que dans les ouvrages de vulgarisation présentent généralement l'utilisation d'Hibernate au sein d'un projet d'entreprise, avec Java EE, un serveur Web, des servlets et des bibliothèques nombreuses. On rajoute parfois un serveur d'application (par exemple JBoss). Dans un tel contexte, la moindre erreur demande de se plonger dans la lecture des forums où il apparaît que telle version d'un composant est incompatible avec tel autre composant, sauf s'il est modifié de telle façon.

Je me propose un peu plus modestement de présenter Hibernate dans un contexte inhabituel : celui d'une simple application Java SE. C'est sans doute artificiel de procéder ainsi, puisque l'usage d'une simple connexion JDBC et une gestion manuelle de la persistance convient pour ce genre d'application. Je veux simplement pouvoir focaliser sur l'installation d'Hibernate et son intégration avec NetBeans [1]. Le présent texte, finalisé dans le courant du mois de mai 2009 et revu en mars 2010, utilise les versions courantes des outils logiciels.

2.1 Composants nécessaires

Les dernières versions complètes de NetBeans incorporent tous les composants nécessaires à l'utilisation d'Hibernate, de même d'ailleurs que TopLink Essentials (un autre gestionnaire de persistance appartenant à Oracle dans sa version Open Source).

Les composants sont les suivants :

- NetBeans 6.8 [2] (tout récent) dont l'installation ne pose en principe aucun problème. J'ai choisi la version de complète. Afin de ne pas polluer mon environnement, j'ai refusé l'importation des préférences de la version antérieure.
- Hibernate 3.2.5. Dans sa toute dernière version : ce produit se compose de trois fichiers *zip* qu'il faudra charger si on veut réaliser une installation manuelle :

1. Beaucoup de sites Web parlent plutôt d'Eclipse.
2. Cette version ajoute également un troisième gestionnaire de persistance : EclipseLink.

  - hibernate-3.x
  - hibernate-entitymanager-3.x
  - hibernate-annotations-3.x
- La bibliothèque JavaEE contient les modules de persistance dans nous avons besoin. Elle fait partie intégrante de plusieurs modules de NetBeans. Elle est disponible dans de nombreux autres produits (le serveur GlassFish, Java Studio Entreprise 8.1, Sun Java System Application Server 9.1).
- Les pilotes pour les bases de données à utiliser. Malheureusement, il faut trouver et placer ces pilotes dans un chemin accessible à Hibernate. Le fait d'avoir un IDE qui fonctionne ne garantit pas qu'Hibernate pourra utiliser la base de données. La découverte de la situation réelle de ces pilotes ressemble parfois à un parcours du combattant. Je propose d'installer des pilotes pour mySQL, Derby (base interne de Java), HSQL, Oracle et SQLite [3]. Il n'est malheureusement pas possible d'expliquer dans ces notes comment créer et gérer des bases dans tous ces systèmes (les systèmes sans serveurs sont particulièrement mal connus et peu documentés).

Les pilotes pour les différents bases de données :

Base	Pilote
mySQL	`mysql-connector-java-5.0.7-bin.jar`
Oracle	`ojdbc14.jar`
Derby	`derbyclient.jar`
HSQL	`hsqldb.jar`
SQLite	`sqlitejdbc-v047-nested.jar`

2.2 Préparation de l'IDE

En principe, les assistants de NetBeans se chargent de référencer les bibliothèques nécessaires à l'application. À titre d'information, il s'agit de tous les fichiers .jar présents dans les entrées *Hibernate*, *Hibernate JPA* et *Persistence* du gestionnaire de bibliothèques (voir figure 1) .

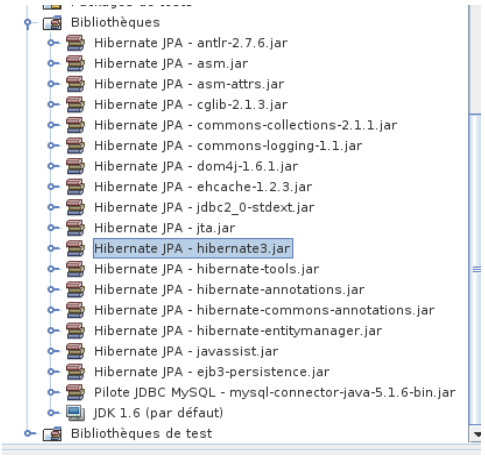


FIGURE 1 – Les bibliothèques nécessaires

3. Bien que j'aie pu faire fonctionner ce pilote avec NetBeans, je ne suis pas parvenu à ouvrir une connexion avec Hibernate. Si on veut utiliser une base légère, il vaut mieux utiliser Derby, bien mieux intégré dans Java.

3 Présentation rapide d'Hibernate

Les principaux éléments de la philosophie d'Hibernate ont été récemment intégré dans l'API de persistance de Java (*Java Persistence API*). Hibernate s'interface donc particulièrement bien avec cette dernière. J'ai choisi de privilégier l'approche définie au moyen d'annotations, bien qu'il reste possible d'envisager une gestion propriétaire de la persistance au moyen de fichiers XML. De même, Hibernate dispose d'une API propre, proche de celle de Java, mais plus riche et plus puissante dans une utilisation avancée. J'ai gardé, dans la mesure du possible, uniquement les aspects qui se conforment au standard JPA. Pour un programmeur avancé, il est possible de garder le respect du standard Java pour la gestion courante et de n'utiliser les fonctionnalités propres d'Hibernate que pour les cas où la richesse fonctionnelle ou les performances l'imposent.

3.1 Contraintes

Hibernate se veut aussi peu intrusif que possible et entend séparer la couche de persistance du modèle objet standard. Les objets manipulés par Hibernate, abstraction faite de leur persistance, devraient pouvoir fonctionner de manière totalement normale. D'autres solutions classiques n'ont pas cette discrétion et imposent l'implémentation d'interface complexes, qui rendent les classes totalement dépendantes de cet environnement (c'est le cas de EJB *Enterprise Java Beans*). Hibernate propose un retour aux *Plain Old/Ordinary Java Objects* (POJO). Hibernate est capable d'analyser un objet par réflexion, mais ses concepteurs conseillent néanmoins d'adopter quelques habitudes de manière à faciliter l'interaction avec les classes d'Hibernate.

constructeur sans argument

Comme Hibernate est amené à recréer les objets par des moyens détournés, il faut qu'il existe au moins un constructeur sans argument pour qu'il réussisse à instancier un ancien objet. Notons que ce constructeur ne doit pas nécessairement être public. Cela signifie aussi que les propriétés persistantes vont devoir être initialisées par des méthodes. C'est l'objet de la contrainte suivante.

convention de nommage JavaBean : variable privée, deux accesseurs `get` et `set`

Cette habitude n'a rien de très gênant, vu qu'elle est employée par la plupart des autres outils : les outils UML ou NetBeans disposent d'assistants qui génèrent automatiquement ce genre de propriétés. Une fois de plus, l'accès à ces variables et à ces méthodes est laissé à l'appréciation du programmeur. Comme dans toute bonne programmation objet, il est déconseillé de laisser un accès public aux variables d'instance.

propriété id (numérique ou String)

L'absence d'identifiant des objets a été soulignée plus haut. Java propose deux concepts pour vérifier l'identité de deux objets. Nous allons voir qu'Hibernate introduit une troisième notion.

– l'*identité* de deux objets permet de vérifier que deux références d'objet pointent sur la même instance. On dispose pour la tester de l'opérateur ==. Ce n'est pas une notion triviale, dans la mesure où deux objets créés indépendamment peuvent parfaitement posséder la même référence, par suite d'une optimisation. C'est parfois le cas des instances

de la classe `String`. En général, le test d'identité donne des résultats peu intéressants pour le déroulement du programme.

- l'*égalité* est une notion plus complexe et plus intéressante. Souvent, il importe de savoir si les informations contenues dans deux instances sont identiques, ce qui permet de dire que les objets sont égaux. L'exemple le plus fréquent est l'égalité de deux chaînes de caractères. La méthode `equals()` définie dans la classe `Object` permet de réaliser le test de cette égalité. Pour la plupart des classes, il convient de surdéfinir la méthode de la classe mère, qui renvoie une valeur basée sur l'identité. Dans le cas des chaînes, par exemple, on opère une comparaison de tous les caractères contenus dans les chaînes. En cas de surdéfinition de la méthode `equals()`, il faut également modifier la méthode `hashCode()` de manière à ce qu'elle se comporte de la même manière [4].
- l'*identité de base de données* nous informe que deux objets sont représentés par la même ligne de la base de données. De manière évidente, cette identité se vérifie en comparant les valeurs des clés primaires. Cette valeur peut être obtenue par la valeur d'une propriété particulière d'une instance persistante (en général, par une méthode habituellement nommée `getId()`).

En pratique, il n'est pas toujours simple de gérer le problème de l'identité. Plusieurs méthodes sont possibles, mais celle qui présente le plus d'avantages consiste à créer une clé d'identité de base de données arbitraire (automatique) et une clé d'égalité basée sur le métier : en utilisant un ou plusieurs champs pour éviter les homonymes et en veillant à choisir des propriétés qui changent rarement. Il reste néanmoins pas mal de questions en suspens (que je ne peux pas envisager de traiter dans une rapide introduction) : par exemple, l'identité de base de données ne peut être testée qu'après transfert de l'objet dans la base.

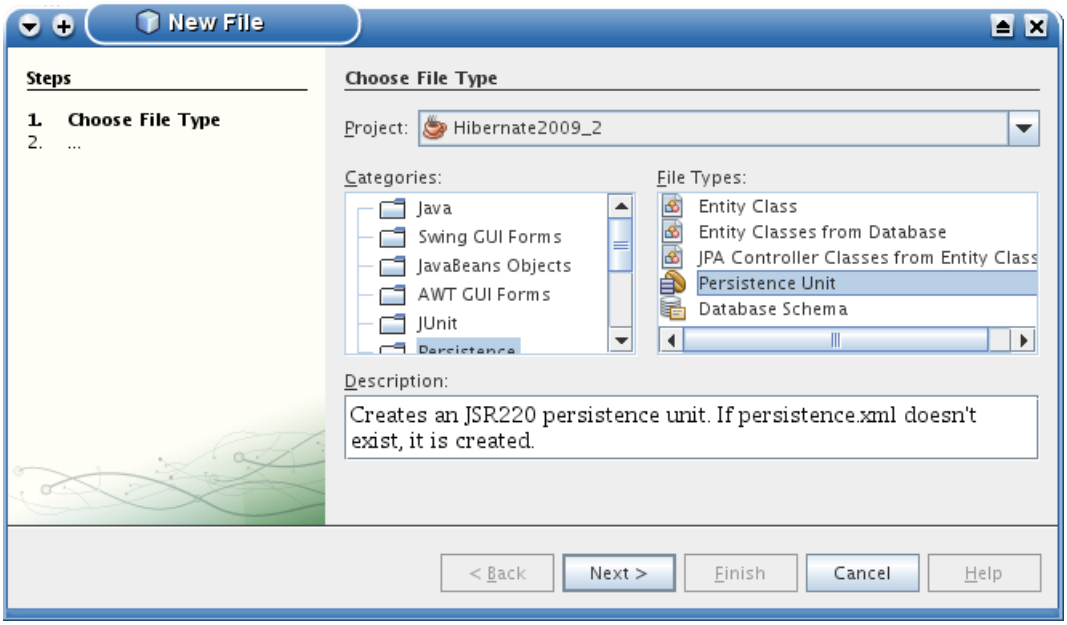
3.2 Création d'un projet d'exploration

Notre projet se basera sur une série de classes utilisées par une facturation simple. Pour éviter les problèmes de conflits de noms, les différentes versions des classes seront placées dans des paquetages nommés *p1*, *p2* etc... Outres les classes métier, chaque paquetage contient une classe *Run* comprenant une méthode statique *main()* permettant de réaliser un test. On pourra disposer du code source des exemples sur mon site (*http://thoorens.net*), ainsi que des exemples réalisés en classe.

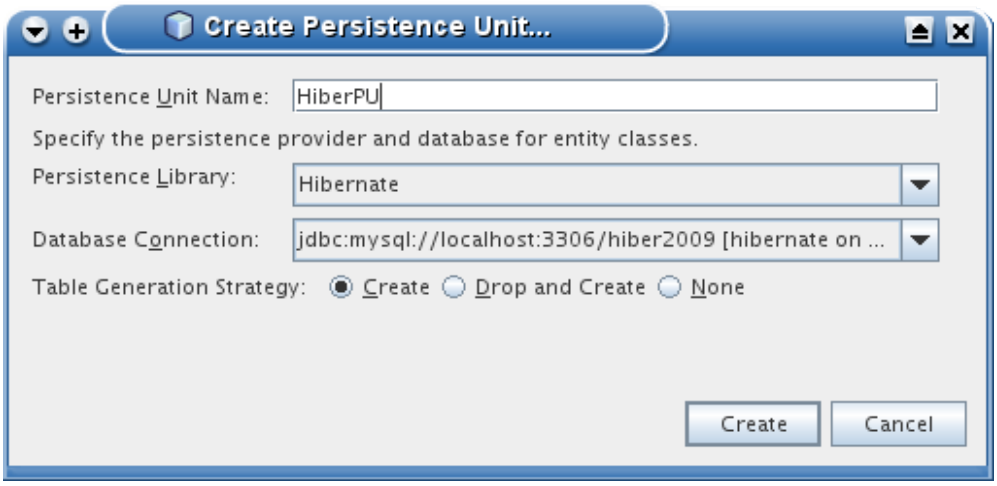
Le contexte

1. s'assurer qu'on dispose d'une version complète de NetBeans, incluant les bibliothèques Hibernate.
2. s'assurer qu'on dispose d'une connexion sur un SGBDR. J'ai choisi un serveur local mySQL avec l'utilisateur *hibernate*, mot de passe *codd*, disposant de droit absolu sur la base *hiber2009*. Créer une connexion pour cette configuration dans l'onglet service.
3. créer un projet normal (avec une classe Main)
4. créer une unité de persistance (*persistence unit*) en spécifiant la bibliothèque de persistance et la connexion à la base de données. NetBeans nous offre pour cela un assistant (*File/New)* :

4. Il faut noter que la méthode `compareTo()` imposée par l'interface `Comparable` permet aussi quand elle renvoie 0 de déterminer l'identité de deux objets. Il convient de bien vérifier que `equals()` et `compareTo()` renvoient bien des valeurs concordantes. Cette exigence se vérifiera pour toutes les classes qui doivent intervenir dans des collections de type `Tree`.



La stratégie de génération des tables par défaut est la création.



Le fichier généré aura le nom *src/META-INF/persistence.xml*. Il s'écrit en XML, mais un assistant en dévoile la substantifique moelle (voir figure 2).

Cet assistant impose d'utiliser une connexion JDBC préalablement préparée dans la fenêtre service de NetBeans (elles se trouvent dans la liste déroulante). Le nom « PU » est arbitraire, mais sera utilisé lors de l'initialisation du gestionnaire de persistance. Les classes seront automatiquement ajoutées par NetBeans, mais cela peut être fait manuellement. Il est possible de créer plusieurs modèles de persistance au sein d'une même application, mais nous n'utiliseront pas cette possibilité.

La stratégie de génération doit être soigneusement choisie :

- *create* crée les tables et les contraintes qui n'existent pas
- *drop and create* servira souvent pendant les premiers tests, mais pas en production
- *none* doit être réservé à une base de données préexistante (le cas le plus fréquent en entreprise).

Voici le fichier pour une connexion avec mySQL. L'utilisation d'une autre base de données nécessite une chaîne de connexion différente, dont le libellé exact sort du cadre de cet exposé.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/
   persistence"
```

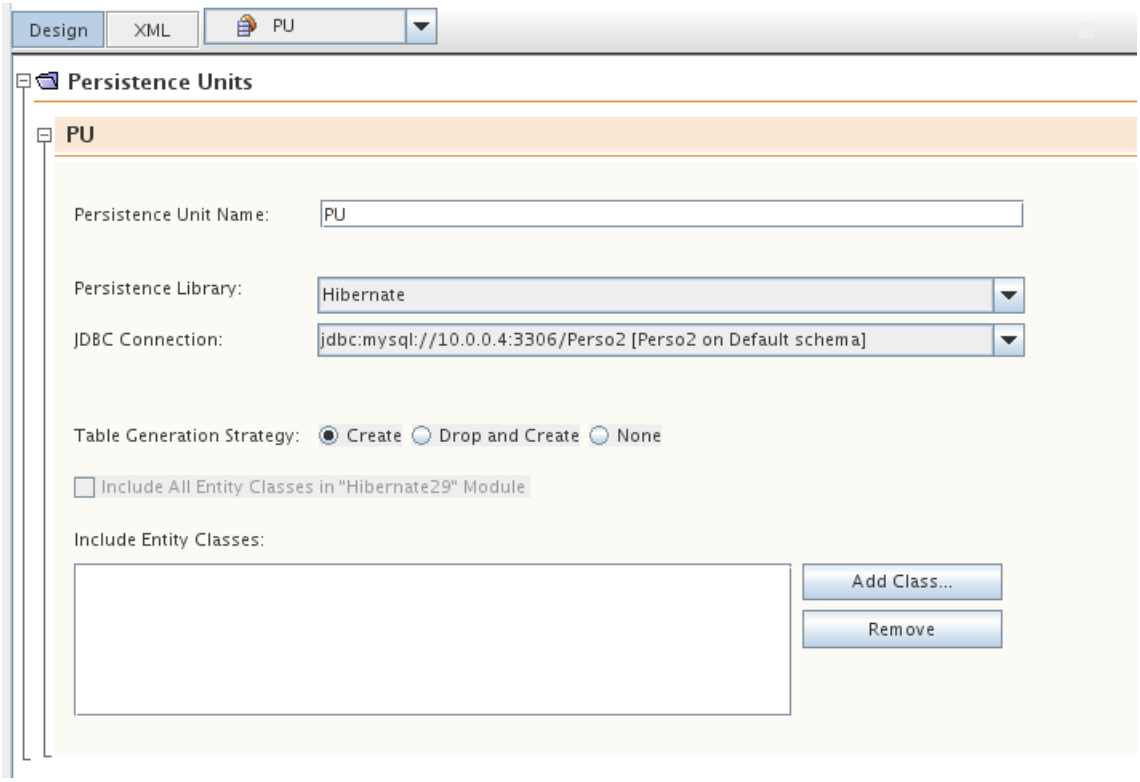


FIGURE 2 – L'assistant de création d'unité de persistance

```
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <property name="hibernate.connection.username"
          value="Perso2"/>
      <property name="hibernate.connection.driver_class"
          value="com.mysql.jdbc.Driver"/>
      <property name="hibernate.connection.password" value="codd"/>
      <property name="hibernate.connection.url"
          value="jdbc:mysql://10.0.0.4:3306/Perso2"/>
      <property name="hibernate.cache.provider_class"
          value="org.hibernate.cache.NoCacheProvider"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
    </properties>
  </persistence-unit>
</persistence>
```

Première classe persistante (*entity class*)

NetBeans offre un moyen agréable de créer des classes persistantes : il crée la classe et propose un assistant pour créer les propriétés. Il suffit de choisir de créer une entité dans les propositions de la catégorie « persistence ». Un embryon de classe sera créé sur base du choix.

```
@Entity
public class Client1 implements Serializable {
```

```
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (id != null ? id.hashCode() : 0);
        return hash;
    }

    @Override
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in the case the id
            fields are not set
        if (!(object instanceof Client1)) {
            return false;
        }
        Client1 other = (Client1) object;
        if ((this.id == null && other.id != null) || (this.id !=
           null && !this.id.equals(other.id))) {
            return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return "modeles_1.Client1[id=" + id + "]";
    }

}
```

Quelques remarques à propos du code obtenu :

- on notera les annotations spécifiques à l'entité :
  - `@Entity` devant l'en-tête de la classe. On peut, si nécessaire, ajouter le paramètre `(name="XYZ")` si on désire que la table porte un autre nom que la classe [5].
  - `@Id @GeneratedValue(strategy = GenerationType.AUTO)` devant le champ `Id`. Ceci signifie que l'on utilisera un générateur automatique pour créer des identifiants. La manière d'y parvenir dépend du type de SGBD.
- les accesseurs de `id` sont automatiquement prévus

5. Dans ces notes, je préfère donner des noms différents aux tables en ajoutant un nombre au nom des classes homonymes dans les différents paquetages.

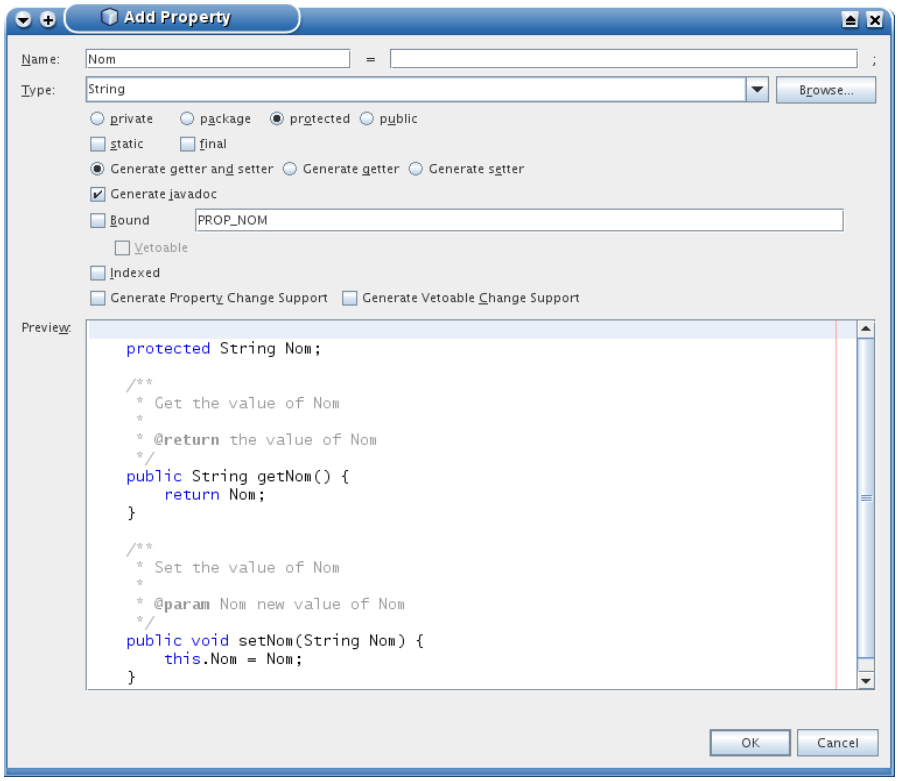


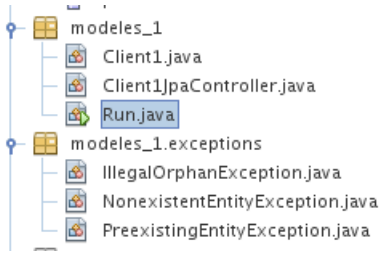
FIGURE 3 – Ajout d'une propriété

– les méthodes `hashCode()`, `equals()` et `toString()` sont redéfines. Une note fait remarquer que `equals()` ne peut fonctionner qu'après sauvegarde de l'instance dans la base de données. On peut choisir de redéfinir la méthode autrement.

Il reste à définir les propriétés, entendues comme des attributs protégés, pourvus d'accesseurs et considérés comme persistants (par défaut tous les attributs sont persistants). Le raccourci *alt-insert* propose un assistant pour définir la propriété (voir figure 3).

Il reste a définir un gestionnaire de persistance pour pouvoir utiliser la classe entité. Un assistant le génère pour nous (voir figure 4).

Quatre classes sont générées :



L'écriture d'un test est immédiate :

```
public static void main(String[] args) {
  System.out.println("Démo␣d'une␣classe␣persistante,␣avec␣
     JPAController");
  System.out.println("
     ================================================");
  Client1JpaController jpac = new Client1JpaController();
  Client1 client = new Client1();
  client.setName("Dupont");
```

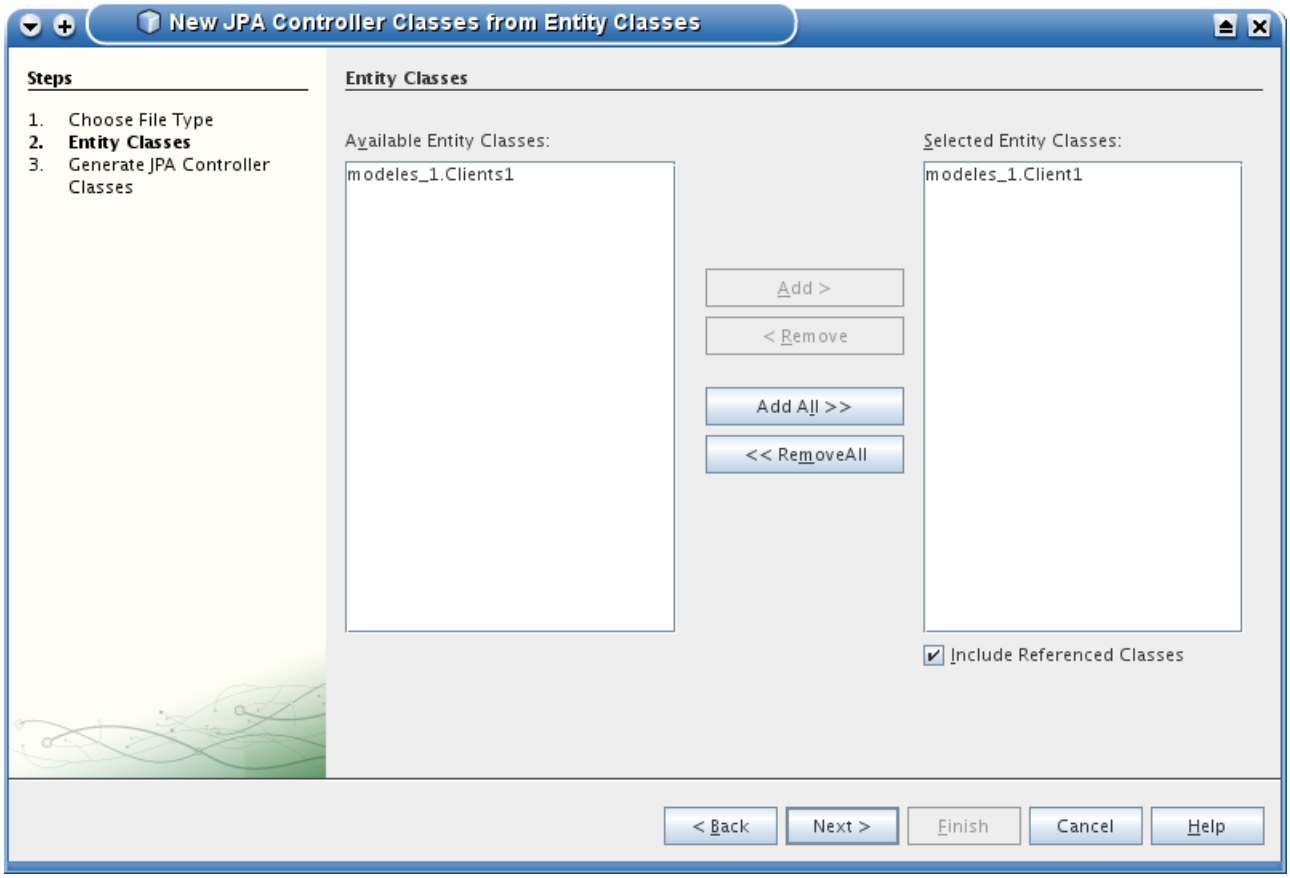


FIGURE 4 – Création d'un gestionnaire

```
  jpac.create(client);
  Client1 cl = jpac.findClient1(1L);
 System.out.println("Nom␣du␣nouveau␣client␣(après␣récupération)␣" +
    cl.getName());
  if(client.equals(cl)){
     System.out.println("Et␣les␣deux␣instances␣sont␣égales");
}
```

La classe `Client1JPAController` permet de créer le gestionnaire de persistance, la table, si nécessaire, et permet la sauvegarde de l'instance grâce à sa méthode `create()`. On constate que l'instance relue porte le nom voulu et qu'elle est « égale » à la première.

Finition de la classe métier

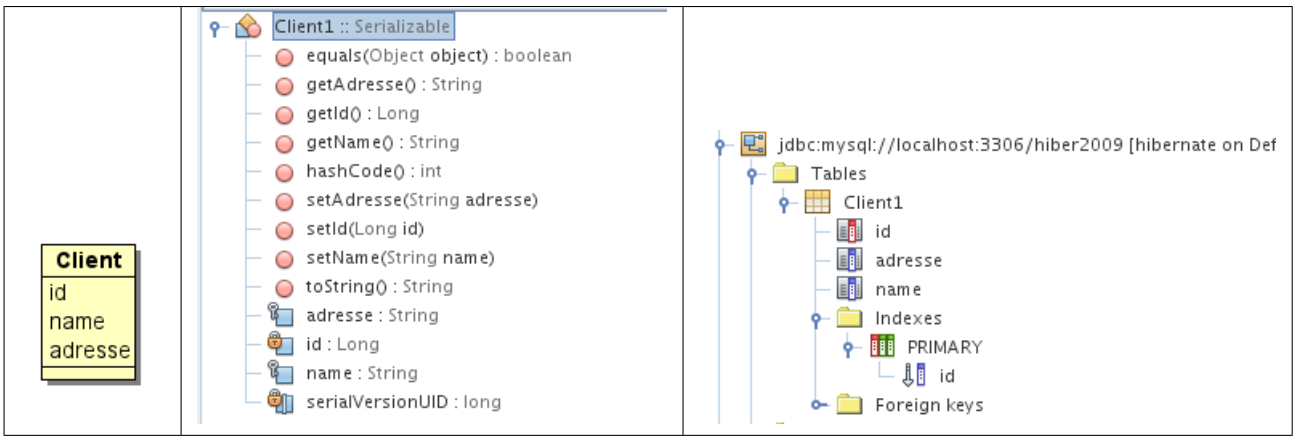
À ce squelette, nous pourrions appliquer quelques modifications :

1. ajouter deux attributs métier `nom` et `adresse`, tous deux de types `String`, ainsi que leurs accesseurs, par factorisation.
2. fournir un constructeur qui initialise le nom du client. La présence d'un constructeur avec un argument nous oblige à redéfinir un constructeur sans argument qui sera nécessaire à Hibernate. Notons que l'interface `Serializable` nous y forcera en cas d'oubli.
3. modifier les trois premières méthodes surdéfinies de manière à ce qu'elles prennent en compte les caractéristiques métier de la classe.
4. ajouter une propriété `name` à l'annotation `@Entity` afin que la classe générée porte un nom adapté au paquetage, puisque nous allons définir d'autres classes *Client* dans notre

exploration. Nous choisirons *Client1*.

Mapping en base de données

Lors de la compilation du programme, si on a activé la génération des tables dans la définition de la persistance, une table sera générée automatiquement. On remarque qu'elle comporte trois champs correspondant aux trois variables d'instance de la classe.



Utilisation de la classe persistante

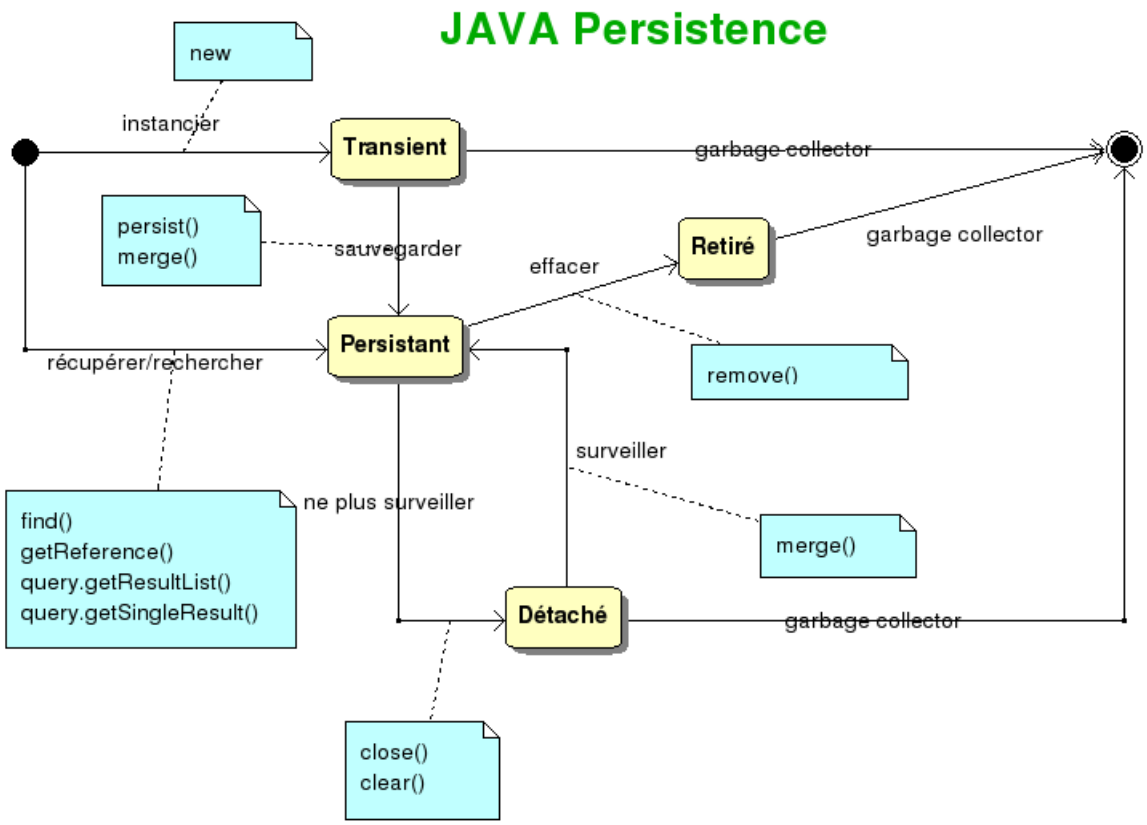


FIGURE 5 – Vie d'un objet persistant

On peut représenter le cycle de vie d'une objet en Java par un diagramme d'état (voir figure 5). Le schéma reprend les méthodes propres au paquetage *javax/persistence*.

Les objets *transients*[6] sont inconnus pour le gestionnaire d'entités. Ils sont créés par un appel de `new` ou une classe ou méthode *factory*. Ces objets peuvent devenir persistants si on appelle la méthode `persist()` du gestionnaire d'entités. On peut récupérer les objets dans la base de données, les réinstancier, au moyen de plusieurs méthodes. La méthode `find()` sert à récuper un objet au moyen de son identifiant. Notons que si on ferme le gestionnaire d'entités, par la méthode `close()`, l'objet cesse d'être géré et les modifications qui le concerne ne sont pas mémorisées. On peut également retirer un objet par la méthode `remove()`. Dans ce cas, les données de l'objet dans la base sont supprimées. Il subsiste en mémoire, mais il est conseillé de l'abandonner.

Il existe une autre bibliothèque basée sur la terminologie d'Hibernate, qui reprend le même parcours de vie, mais avec des méthodes nommées autrement et en plus grand nombre.

3.3 Création d'une classe utilitaire

Nous avons vu que NetBeans permet de gérer les classes/entités à l'aide d'une classe annexe. Cette méthode présente des avantages et des inconvénients. Bien écrit, il a pour mérite de laisser le système de persistance dans un état stable à chaque utilisation. Il double néanmoins le nombre de classes, puisque chaque entité se voit dotée d'un contrôleur. Il me semble aussi qu'en ouvrant et fermant sans cesse les gestionnaires d'entités, en ouvrant et validant les transactions pour ne gérer qu'une seule classe, il ne sera sans doute pas très performant.

D'où l'idée de proposer un gestionnaire d'entités plus générique, capable de donner à n'importe quelle classe le moyen de gérer ses instances de manière simple.

Je propose ici une classe à méthodes statiques capable de récupérer facilement le gestionnaire d'entités (*entity manager*), le gestionnaire de transactions, voire même de disposer de toutes les méthodes nécessaires à une gestion simple de la persistance de quelques classes. La plupart de ces méthodes seront statiques. Il suffira d'appeler la méthode `setName` qui permettra d'initialiser le nom de l'unité de persistance à utiliser[7].

```
package outils;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.persistence.Query;

    /**
 * La classe permet une utilisation simplifiée du gestionnaire d'
    entités
 * @author jacques
 */
public class GP {
    private static EntityManager em;
    private static EntityTransaction tx;
    private static String persistenceName;
    private static EntityManagerFactory emf;

    /**
```

6. *Ce terme peut être traduit par* transitoire, éphémère, passager.
7. Cette classe utilitaire n'est donc pas utilisable avec plusieurs gestionnaires de persistances simultanés.

```
     * Initialise la classe de gestion des entités au moyen du nom
        de l'unité
     * de persistance
     * @param persistenceName
     */
    public static void setName(String persistenceName){
        GP.persistenceName=persistenceName;
    }

    /**
     * Renvoie la référence du gestionnaire d'entités actif.
     * Au besoin, l'initialise.
     * @return Le gestionnaire d'entités actif
     */
    public static EntityManager getEM(){
        if(em!=null)
            return em;
        else
        {
            emf = Persistence.createEntityManagerFactory(
               persistenceName);
            em = emf.createEntityManager();
            tx = em.getTransaction();
            return em;
        }
    }

    /**
     * Renvoie la transaction en cours pour invoquer une de ses
        méthodes.
     * Si nécessaire, le gestionnaire d'entités est initialisé.
     * @return la transaction en cours
     */
    public static EntityTransaction getTX(){
        if(tx!=null)
            return tx;
        else
            return getEM().getTransaction();
    }

    /**
     * Ferme le gestionnaire d'entités
     * @param transaction spécifie si la transaction doit être
        fermée
     */
    public static void close(){
        if(tx.isActive())
            tx.commit();
        em.close();
        emf.close();
        em=null;
        tx=null;
    }
```

```
/**
 * Accès direct à la méthode commit de la transaction en cours
 * Le gestionnaire d'entités est supposé actif
 */
public static void transactionCommit(){
    getTX().commit();
}

/**
 * Accès direct à la méthode begin de la transaction en cours
 * Si nécessaire le gestionnaire d'entités est initialisé
 */
public static void transactionBegin(){
    getTX().begin();
}

/**
 * Accès direct à ma méthode rollback de la transaction en cours
 * Le gestionnaire d'entités est supposé actif
 */
public static void transactionRollback(){
    getTX().rollback();
}


/**
 * Accès direct à la méthode persist du gestionnaire d'entités
 * Le gestionnaire d'entités est supposé actif
 * @param objet Objet à sauvegarder
 */
public static void persist(Object objet){
    getEM().persist(objet);
}

/**
 * Recréation d'une instance archivée dans la base de données
 * Le gestionnaire d'entités est supposé actif
 * @param <T>
 * @param classe la classe de l'instance à rechercher
 * @param id identificant de l'instance
 * @return l'instance retrouvée
 */
public static <T> T find(Class<T> classe,Long id){
    return getEM().find(classe,id);
}

/**
 * Accès direct à la méthode merge du gestionnaire d'entités
 * Le gestionnaire d'entités est supposé actif
 * @param objet Objet à sauvegarder
 */
public static void merge(Object objet){
```

```
        getEM().merge(objet);
    }


    /**
     * Permet d'effacer les données d'une table avant la
        démonstration
     * Le gestionnaire d'entités est supposé actif
     * @param tableName table à affacer
     */
    public static void deleteTable(String tableName){
        //try {
            //String SQL="TRUNCATE TABLE "+tableName;
            String SQL="␣DELETE␣FROM␣"+tableName;
            Query query = GP.getEM().createQuery(SQL);
            System.err.println("Tentative␣de␣suppression␣du␣contenu␣
               de␣"+tableName+"␣par␣"+SQL);
            GP.transactionBegin();
            int lignes=query.executeUpdate();
            System.out.printf("Suppression␣de␣%d␣ligne␣dans␣%s\n",
               lignes,tableName);
            GP.transactionCommit();
        /*} catch (Exception e) {
            System.out.println("Erreur : pas de suppression de
               lignes dans "+tableName);
            System.out.println(e.getMessage());
        }*/
    }


}
```

Toutes les autres méthodes pourront appel à cette dernière au cas où l'initialisation n'a pas eu lieu.

Le lecteur intéressé pourra facilement étendre les fonctionnalités de cette classe par le biais de nouvelles méthodes statiques.

Vérification

Les instructions suivantes doivent s'exécuter sans erreur. Les tables seront créées dans la base de données si elles n'existent pas encore.

```
private static void verification() {
        GP.begin();
        GP.close();
    }
```

Voici un exemple de code qui va utiliser plusieurs des fonctions. On notera l'appel à la démo vue plus haut pour initialiser la table (il faut noter que la commande « `TRUNCATE TABLE Client1` » devrait être lancée avant d'exécuter le programme. En effet, la suppression de la table ne remet pas la clé automatique à 0. Malheureusement, Hibernate n'accepte pas cette commande.

```
    public static void main(String[] args) {
```

```
        GP.getEM();
        GP.deleteTable("Client1");
        // Création
        modeles_1.Run.main(args);
        // Première manip
        GP.setName("Hibernate2009_2PU");
        Client1 client=GP.find(Client1.class, 1L);
        System.out.println("Nom du client 1:"+client.getName());
        client.setAdresse("rue de l'Eglise");
        GP.close();
        // Nouvelle gestion
        GP.merge(client);
        Client1 cl2 = GP.find(Client1.class, 1L);
        System.out.println("Adresse : "+cl2.getAdresse());
        GP.close();
    }
```

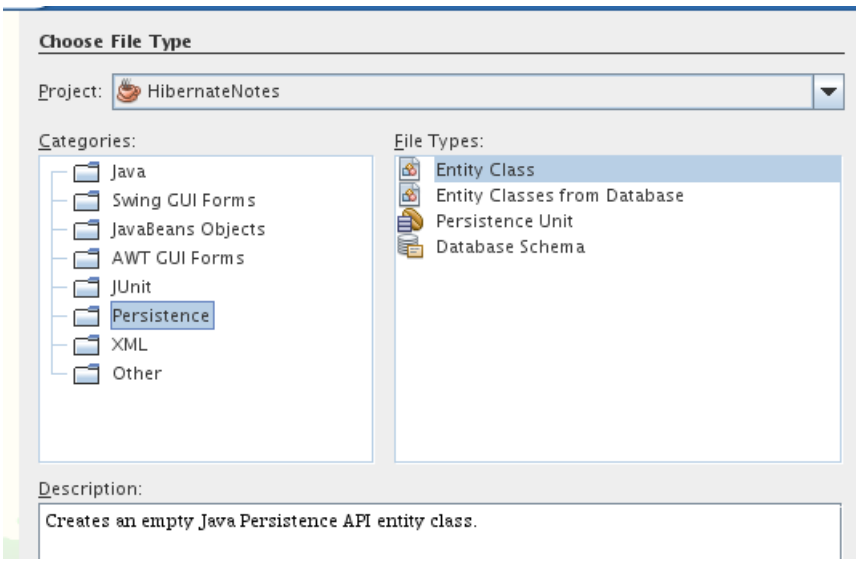
4 Classes et associations

La persistance d'une seule classe constitue un problème assez trivial. Nous allons voir que la conception d'un modèle de classes riches, avec des associations et des généralisation mérite un examen détaillé de différents types de modélisation. Cette section va se limiter aux associations. Nous verrons les problèmes d'héritage et de polymorphisme dans la section suivante.

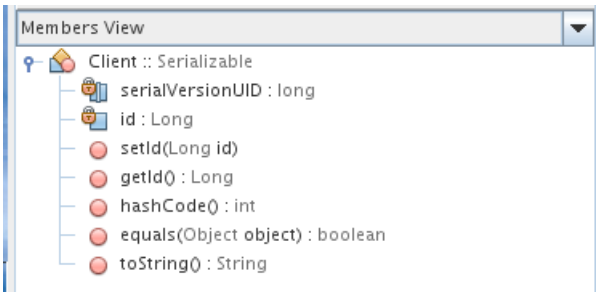
4.1 Une classe simple (paquetage *p1*)

Annotation de la classe Java

Réexaminons la classe `Client1` vue plus haut. Rappelons qu'elle a été créée à l'aide de NetBeans.



L'assistant nous produit une classe déjà consistante :



L'attribut `id` jouera un rôle fondamental dans la persistence (il deviendra la clé primaire de la table). Il est doté de ses accesseurs. La classe implémente l'interface sérialisable (ce n'est pas requis, mais conseillé) et surdéfinit trois méthodes de la classe *Object*. Répétons que le code automatiquement généré pour les méthodes `equals()` et `hashCode()` n'est qu'une approximation de ce qu'il devrait être en finale : en effet, il s'appuie sur `id` qui n'a de sens qu'après une première sauvegarde dans la base de données. Il faudrait se servir d'attributs métiers de la classe pour écrire ces méthodes. La présence de `hashCode` permettra de placer cette classe dans des collections de type `HashSet`. Si on veut utiliser une collection de type `TreeSet` il faudra en outre implémenter l'interface `Comparable` et définir la méthode `compareTo()`.

Trois annotations importantes ont été définies :

- `@Entity` manifeste que nous avons affaire à une classe persistante. Toutes les valeurs des attributs de types simples seront conservées dans la base de données, à moins de spécifier `@Transient`.
- `@Id` indique l'attribut qui correspondra à la clé primaire. Cette annotation est placée devant l'accesseur `getId()`.
- `@GeneratedValue(strategy = GenerationType.AUTO)` manifeste que nous désirons une génération automatique de cette clé primaire [8].

```
package p1;
import  ...

@Entity
public class Client1 implements Serializable {
    private static final long serialVersionUID = 1L;
    private Long id;

    public void setId(Long id) {
        this.id = id;
    }

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public Long getId() {
        return id;
    }
    @Override
    public int hashCode() {
    ...
```

8. D'autres méthodes de génération sont disponibles et expliquées dans la documentation d'Hibernate et de JAP, elles prennent notamment en compte l'existence des séquences. On peut aussi choisir de ne pas initialiser l'attribut `id` automatiquement (c'est ce que je ferai plus loin pour les factures, où l'initialisation se fera par le code de l'application). Dans un vrai programme, il faudra trouver un moyen sûr d'éviter les doublons.

```
    @Override
    public boolean equals(Object object) {
        ...
    @Override
    public String toString() {
        ...
}
```

Test de la classe

La méthode statique `p1/Run/classesimple()` crée un objet, le rend persistant, ferme la transaction en cours et oublie l'objet. On rouvre ensuite une nouvelle transaction pour récupérer l'objet mis en mémoire (ici sur base de son identifiant). Nous allons utiliser les méthodes `persist()` et `find()` pour placer une instance dans la base de données et la récupérer. Il est également possible de récupérer une instance sur base d'un critère quelconque en utilisant une requête et une requête HQL, dont la syntaxe est proche de SQL.

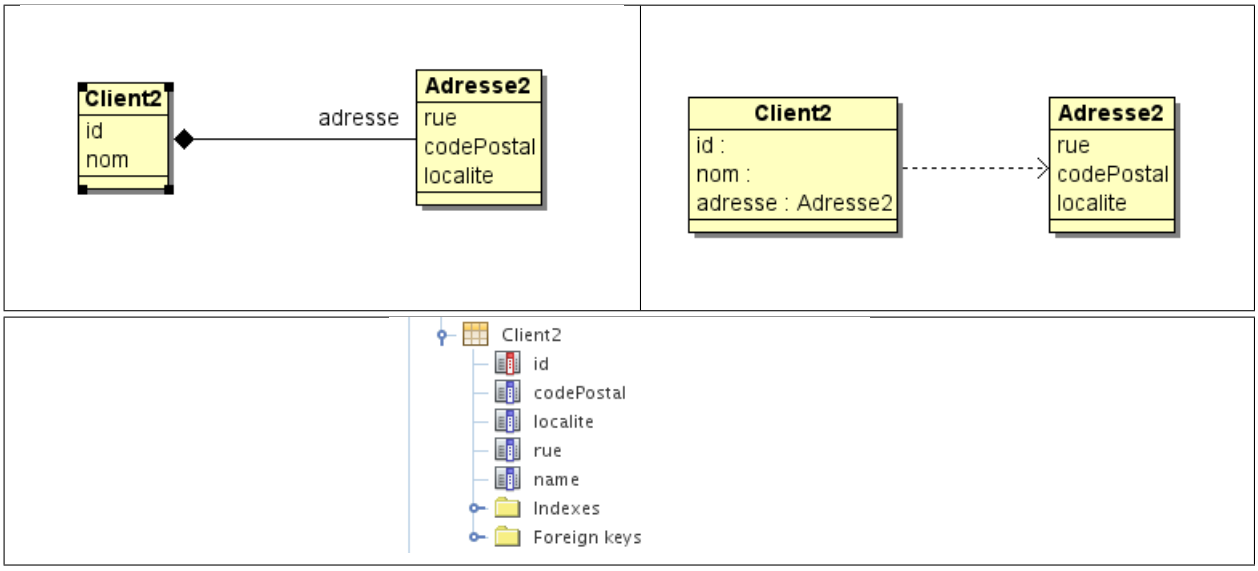
```
public static void classeSimple() {
   Long lastID;

   // nettoyage de la table si nécessaire
   GP.deleteTable("Client1");

   System.out.println("Création d'une classe simple");
   GP.begin();
   Client1 client = new Client1("Dupont");
   GP.persist(client);
   lastID=client.getId();
   client.setAdresse("rue de l'Eglise");
   GP.close();
   client = null;

   GP.begin();
   //récupération du client par son identifiant
   client = GP.find(Client1.class, lastID);
   System.out.println("Le client récupéré " + client.getNom() +
            " habite " + client.getAdresse());

   // Récupération sur base d'un critère quelconque
   Query query = GP.getEM().createQuery(
   "SELECT x FROM Client1 x WHERE nom = :nom");
   query.setParameter("nom","Dupont");
   Client1 copie = (Client1)query.getSingleResult();

   // comparaison des deux instances
   if(copie.equals(client))
       System.out.println("Deux objets identiques sont en mémoire");
   if(copie.getId()==client.getId())
       System.out.println("Les deux copies ont le même ID.");
   if(copie==client)
       System.out.println("Les deux copies sont le même objet.");
   client.setAdresse("rue du Temple");
```

```
    System.out.println("L'adresse de la copie est "
       + copie.getAdresse());
    GP.close(true);
}
```

La deuxième instance (`copie`) porte le même nom que la première (le nom est supposé unique dans le système). La méthode `equals()` les déclare donc identiques. Elles partagent le même identifiant. La dernière comparaison est plus étonnante : en fait lors de la récupération par `getSingleResult()`, Hibernate a constaté que l'instance existait déjà en mémoire et s'est contenté de fournir une référence vers l'objet existant. L'opérateur `==` constate donc cette équivalence. La modification de l'adresse de l'instance `client` se répercute sur la « copie ».

4.2 Composition (paquetage p2)

Certaines classes ont une existence assez faible. Elles constituent un moyen commode de regrouper des données (une sorte de structure). Parfois, elles offrent des méthodes de manipulation, comme dans le cas des dates. Mais, à aucun moment, on ne peut parler d'entités. J'ai choisi l'exemple de l'adresse. En modélisation, on peut représenter cela au moyen d'une composition unidirectionnelle (la représentation utilisant une simple dépendance est également possible et produit le même code en Java).



Dans la gestion de la persistance, on considère que les différents attributs de l'adresse doivent être incorporés dans les différentes entités qui en font usage. On utilise simplement les annotations `@Embeddable` du côté de la classe contenue et `@Embedded` au niveau du champ [9] inclus dans l'autre classe (en l'occurrence, le client).

```
@Embeddable
public class Adresse2 implements Serializable {
    private static final long serialVersionUID = 1L;

    private String rue;
```

9. Java permet de placer les annotations soit avant la variable d'instance, soit avant son accesseur en écriture (solution meilleure). NetBeans propose de refactoriser les programmes pour donner un traitement cohérent à toutes les annotations si on a disposé les annotations au petit bonheur.

```
    private String localite;
    private String cp;
...
}
```

Dans la classe Client2, on veillera à initialiser la variable d'instance avant toute écriture dans la base de données. Comme cette écriture n'est pas toujours prévisible, une adresse dont les champs sont laissés sans valeur sera une sage précaution.

```
    private Adresse2 adresse=new Adresse2();
    @Embedded
    public Adresse2 getAdresse() {
        return adresse;
    }
    public void setAdresse(Adresse2 adresse) {
        this.adresse = adresse;
    }
```

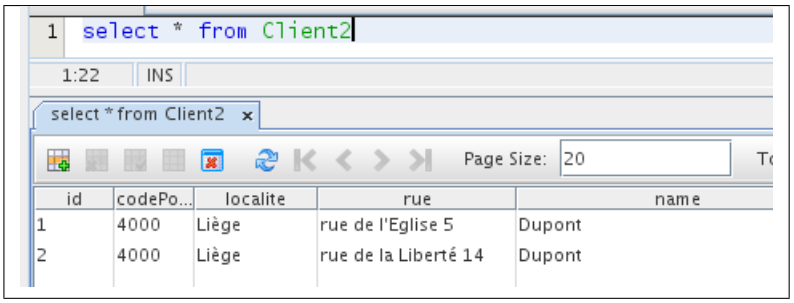
Une conséquence de cette manière de modéliser est que l'adresse est mémorisée dans chaque client. Si on utilise une même adresse pour deux clients différents, la modification de l'adresse entraînera la modification de l'adresse de l'autre.

```
public static void modificationAdresse(){
    Long id1,id2;

    System.out.println("Modification d'adresse");
    GP.begin();
    Client2 client1=new Client2("Durant");
    Client2 client2 =new Client2("Dumont");
    GP.persist(client1);
    GP.persist(client2);
    id1=client1.getId();
    id2=client2.getId();
    Adresse2 adresse1 = new Adresse2("rue de l'Eglise");
    Adresse2 adresse2 = new Adresse2("rue du Temple");
    adresse2=adresse1;
    client1.setAdresse(adresse1);
    client2.setAdresse(adresse2);
    adresse1.setCP("4000");
    adresse2.setCP("4000");
    System.out.printf("Les deux clients ont le même code postal:  %s
        et %s \n",
            client1.getAdresse().getCP(), client2.getAdresse().getCP
               ());
    GP.close();
    // voir suite plus loin
```

Par contre, quand on travaillera avec un nouveau gestionnaire d'entités (après fermeture du premier à l'aide de `close()` ou dans un autre programme), les données seront lues dans la table dans deux lignes différentes et le lien entre les adresses sera perdu.

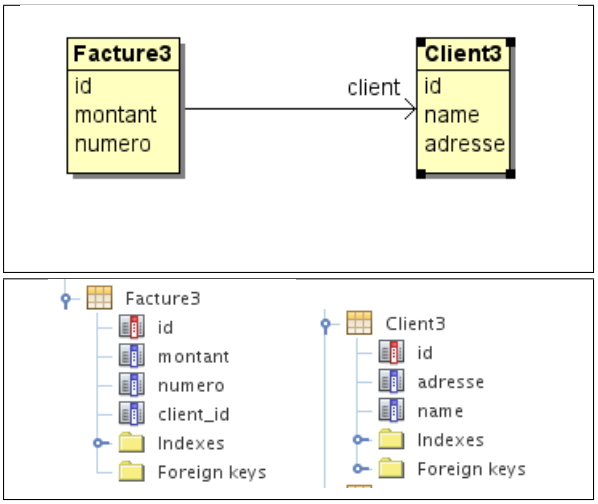
```
    // suite de modificationAdresse
    adresse1=adresse2=null;
    client1=client2=null;
    GP.begin();
```

```
        client1=GP.find(Client.class, id1);
        client2=GP.find(Client.class, id2);
        client1.getAdresse().setCP("4020");
        System.out.printf("CP␣des␣deux␣clients␣différents:␣%s␣et␣%s␣\n",
                client1.getAdresse().getCP(),client2.getAdresse().getCP
                    ());
        GP.close();
}
```

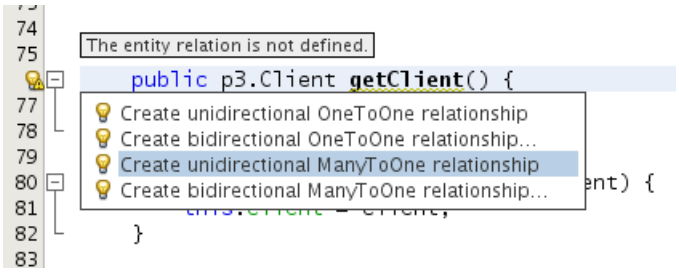


4.3 Association unidirectionnelle N:1 (paquetage p3) `@ManyToOne`

Dans la modélisation d'une facture, on trouvera trace du client à qui elle est adressée. Si le programme veut simplement connaître le client correspondant à une facture, il sera nécessaire de créer une association unidirectionnelle avec la multiplicité 1 sur la destination (le nom de rôle *client* permettra de donner automatiquement un nom à la variable d'instance). Nous lui donnerons par factorisation les accesseurs `getClient()` et `setClient()`. En base de données, on sait que le lien entre la facture et le client sera matérialisé par une clé étrangère servant à mémoriser l'identifiant du client.



Les deux classes Facture et Client seront créées par l'assistant comme des classes d'entité. Lorsqu'on ajoute à la classe Facture la variable d'instance `client` et ses accesseurs, NetBeans réagit immédiatement en signalant que la relation entre entités n'est pas définie. Il nous propose le choix entre quatre type de relations, dont nous pouvons d'emblée exclure les relations bidirectionnelles. Comme un client peut avoir plusieurs factures, nous choisissons la relation unidirectionnelle *ManyToOne*.



Ce modèle illustre bien le divorce entre le modèle objet où une relation bidirectionnelle doit faire l'objet d'un traitement complexe et le modèle relationnel où ce type de relation est automatiquement obtenu.

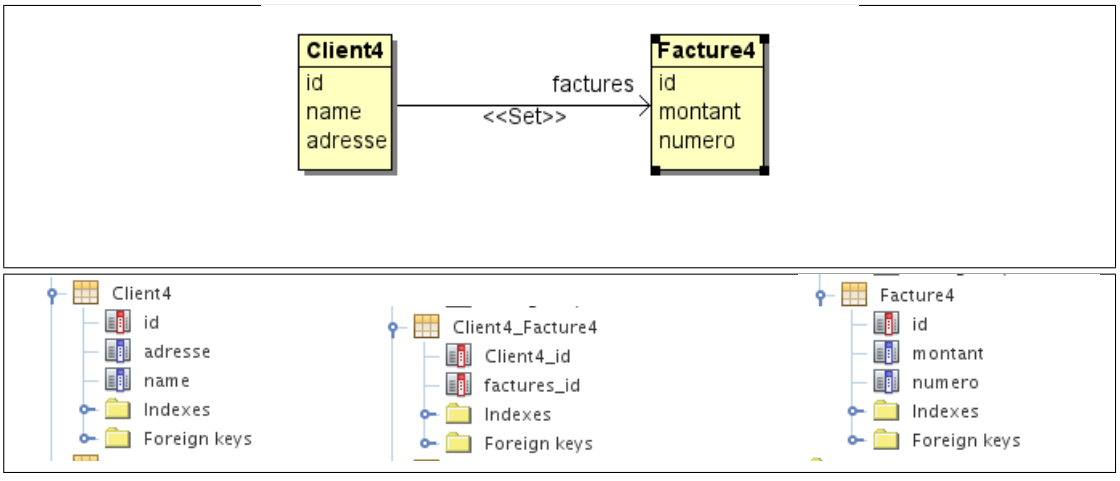
L'assistant nous génère automatiquement la seule annotation nécessaire (`@ManyToOne`).

Dans `Facture3`

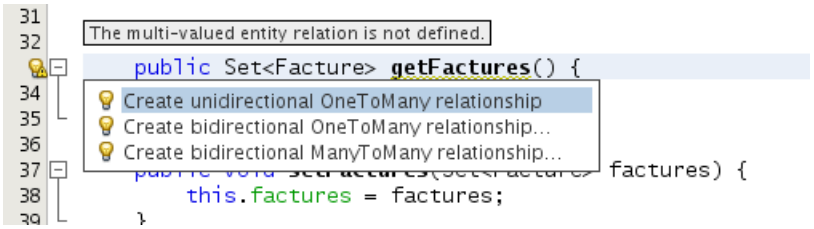
```
    private Client3 client;
    @ManyToOne
    public Client3 getClient() {
        return client;
    }
    public void setClient(Client3 client) {
        this.client = client;
    }
```

4.4 Association unidirectionnelle 1:N (paquetages p4 et p4b)`@OneToMany`

Dans le cas inverse d'un programme centré sur les clients, on désirera connaître les différentes factures d'un client (une navigabilité unidirectionnelle, une multiplicité $\star$, et un stéréotype *Set* pour marquer le choix de la collection). Notons que le modèle relationnel attendu est le même que dans le cas précédent, mais que ce qui est généré en diffère un peu.



L'ajout d'une variable de type `Set` et de ses accesseurs va provoquer la même réaction de NetBeans et, cette fois, nous choisirons l'option « unidirectional OneToMany ».



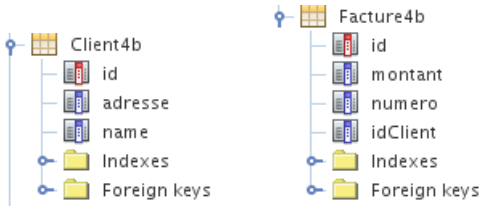
Dans `Client4`

```
private Set<Facture4> factures = new HashSet<Facture4>();

@OneToMany
public Set<Facture4> getFactures() {
    return factures;
}
public void setFactures(Set<Facture4> factures) {
    this.factures = factures;
}
```

Il faut reconnaître que le modèle relationnel généré, avec sa table intermédiaire, est pour le moins curieux. Pour obtenir un modèle plus classique, il faut préciser un nom de colonne de jointure (le système ne peut le déduire puisqu'il n'existe pas de variables d'instance pour référencer le client dans la table Facture4).

Dans `Client4b`

```
private Set<Facture4b> factures = new HashSet<Facture4b>();

@OneToMany
@JoinColumn(name="idClient")
public Set<Facture4b> getFactures() {
    return factures;
}

public void setFactures(Set<Facture4b> factures) {
    this.factures = factures;
}
```



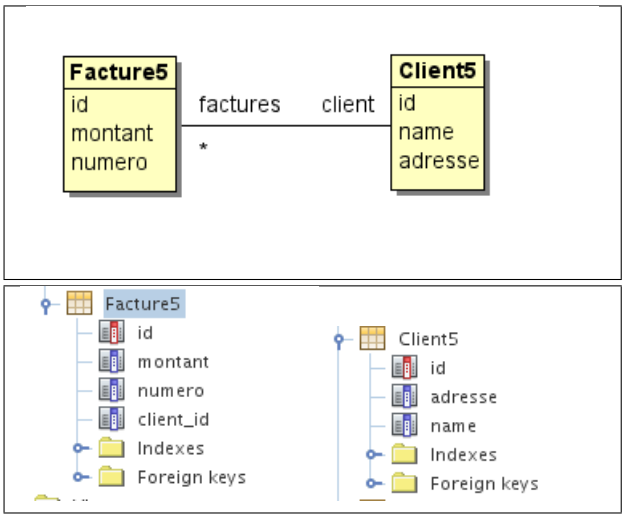
Remarques :

1. la version à trois tables est assez lourde à gérer au moment de l'effacement des données (impossibilité de supprimer les tables données de la table de jointure à l'aide d'Hibernate, sans doute à cause de la perte de contenu que cela implique pour des objets existants).

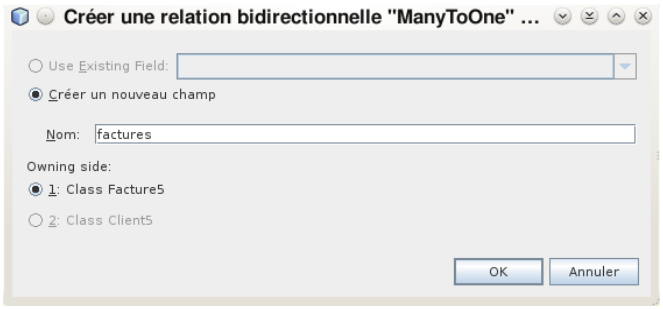
2. il n'est pas commode de gérer les ajouts de factures au moyen des accesseurs de la collection. On conseille d'écrire une méthode qui facilite le travail.

4.5 Association bidirectionnelle 1:N (paquetage p5) `@ManyToOne`

Le modèle objet rejoint ici la puissance du modèle relationnel, mais au prix d'une gestion de doubles références : une variable simple dans la facture et une collection dans le client. Le modèle relationnel est le même que précédemment.



Le simple ajout d'un attribut `client` et de ses accesseurs dans la classe *Facture5* provoque l'apparition de l'assistant vu précédemment. Il faut choisir l'option « bidirectional ManyToOne ». Une boîte de dialogue propose de créer l'attribut correspondant dans l'autre classe (si l'attibut existe déjà, on peut le choisir dans une liste). On doit également choisir le côté qui « possède » la relation.



Dans `Facture5`

```
private Client5 client;
@ManyToOne
public Client5 getClient() {
    return client;
}
public void setClient(Client5 client) {
    this.client = client;
}
```

Le propriétaire de la relation n'ajoute rien à l'annotation. Du côté du Client5, on ajoute le propriété `mappedBy` [10].

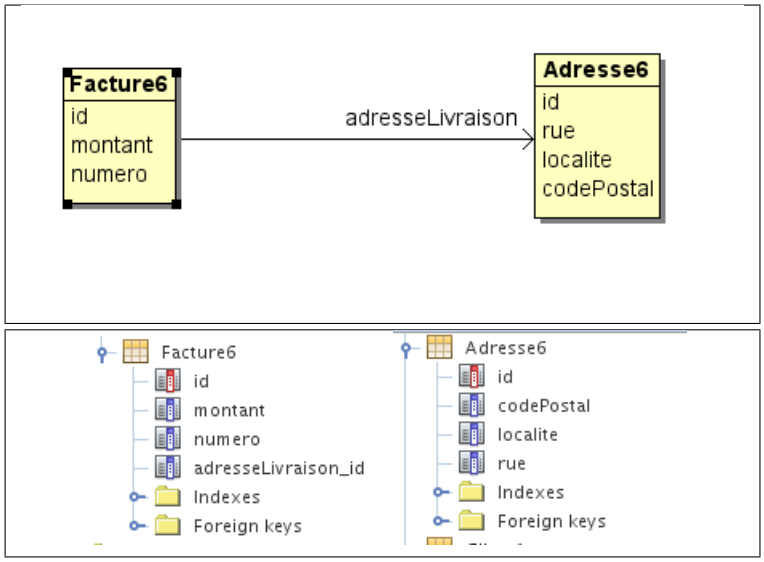
Dans `Client5`

```
private Set<Facture5> factures=new HashSet<Facture5>();
@OneToMany(mappedBy="client")
public Set<Facture5> getFactures() {
    return factures;
}

public void setFactures(Set<Facture5> factures) {
    this.factures = factures;
}

public void ajouterFacture(Facture5 facture){
    factures.add(facture);
    facture.setClient(this);
}
```

La méthode `ajouterFacture()` est un moyen commode de gérer la double référence dans les deux classes [11].

4.6 Association unidirectionnelle 1:1 (paquetage 6) `@OneToOne`

Ce cas plus rare prend en compte des associations qui lient par exemple une facture à son adresse de livraison



Il y a de grandes analogies avec le cas N :1 tant au niveau du modèle relationnel que dans les assistants et les annotations.

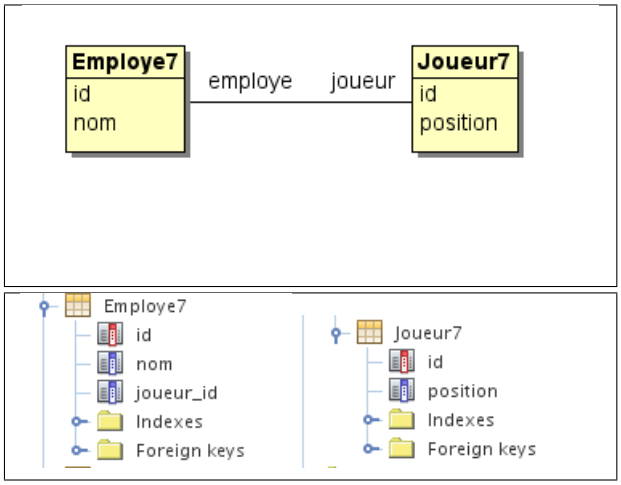
10. NetBeans propose un type `List<Facture5>`. Je préfère utiliser `Set` qui empêche les doublons. Notons que l'initialisation avec un `HashSet` est indispensable. Le choix de `HashSet` permet de profiter du soin mis à surdéfinir la méthode `hash`.
11. Dans une application réelle, il faudrait également gérer la suppression.

Dans `Facture6`

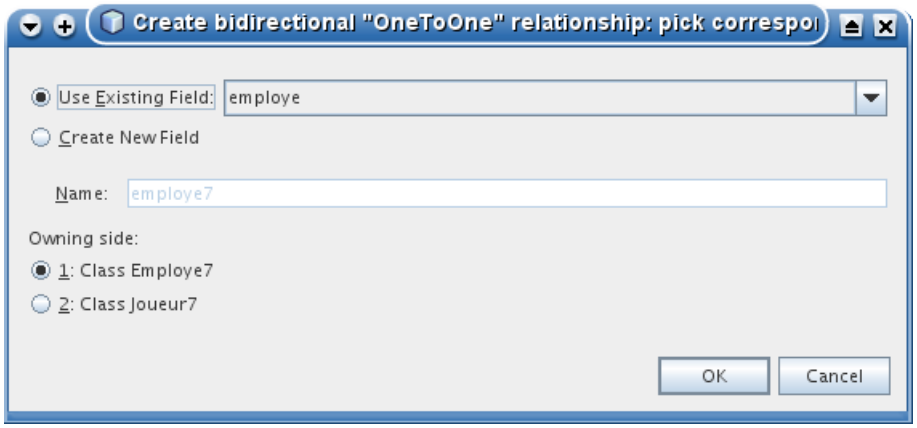
```
private Adresse adresse;
@OneToOne
public Adresse getAdresse() {
    return adresse;
}
public void setAdresse(Adresse adresse) {
    this.adresse = adresse;
}
```

4.7 Association bidirectionnelle 1:1 (paquetage 7) `@OneToOne`

Nous allons imaginer un match de football qui concerne une entreprise. Nous disposons de classes pour remprésenter les employés. Nous allons créer des joueurs, caractérisés par leur position sur le terrain. Chaque joueur correspond à un employé et certains employés sont des joueurs.



On retrouve encore le même genre de démarche et quasiment le même assistant que pour la relation 1:N. Il faut répondre à une question supplémentaire, puisque le lien relationnel peut figurer indifféremment du côté de l'employé (clé étrangère vers les joueurs) ou du côté du joueur (clé étrangère vers les employés). Dans le modèle relationnel, deux clés étrangères seraient non seulement inutiles, mais source de problèmes, au cas où elles se désynchroniseraient. Dans le modèle objet, les deux objets doivent posséder une variable d'instance pour se référencer mutuellement. Voici encore une illustration de l'incompatibilité des deux modèles.



Notons que l'assistant est capable de créer le champ dans l'autre classe, mais qu'il faut créer les accesseurs avec un autre assistant.

Dans `Employe7`

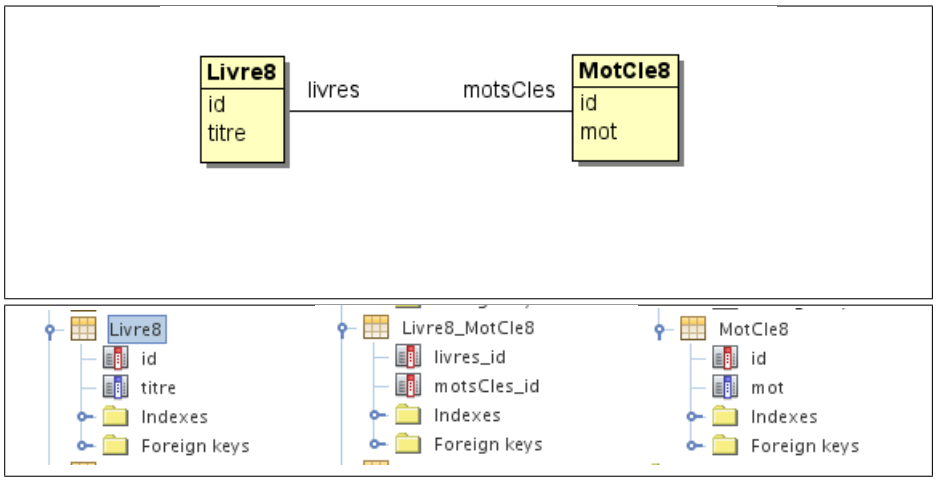
```
@OneToOne
protected Joueur7 joueur;
```

Dans `Joueur7`

```
@OneToOne(mappedBy = "joueur")
private Employe7 employe=new Employe7();
```

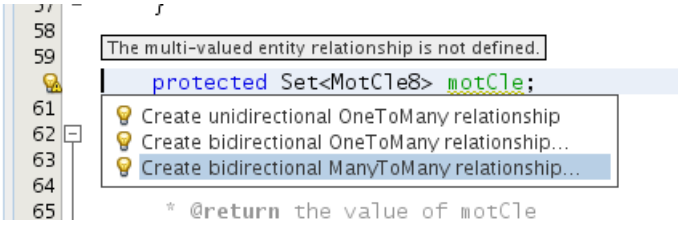
4.8 Association bidirectionnelle N:M (paquetage 8) `@ManyToMany`

Je propose ici un exemple tiré d'une bibliothèque : un livre peut avoir un ou plusieurs mots-clés et un mot-clé peut caractériser un ou plusieurs livres.

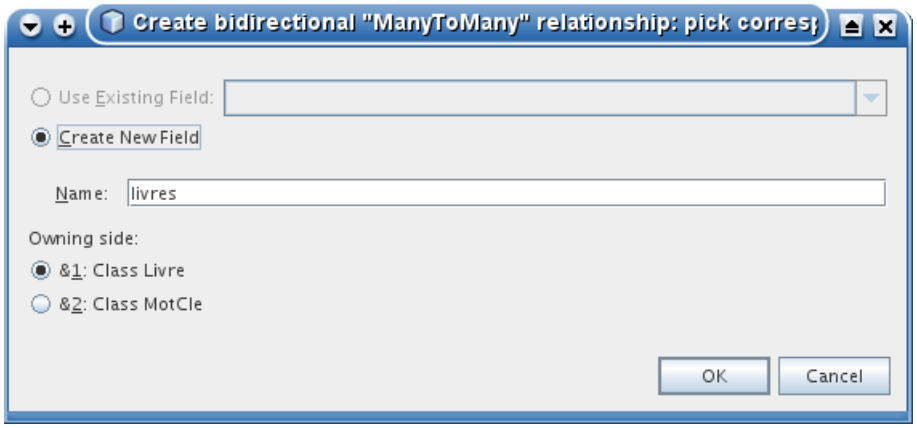


La table de jointure du modèle relationnel est ici parfaitement justifiée.

Pour la persistance, il suffit cette fois de créer une association ManyToMany (elles sont forcément bidirectionnelles).



Nous retrouvons un assistant pour la création de la collection de l'autre côté (par défaut, NeatBeans propose une liste).



Notons que l'assistant génère un champ de type `Liste<MotCle8>` ce qui ne m'arrangeait pas. Il faut également prévoir les accesseurs, ainsi qu'une méthode permettant de réaliser la double référence lors de l'ajout d'un mot clé.

Dans `Livre8`

```
private Set<MotCle8> motscles = new HashSet<MotCle8>();
@ManyToMany
public Set<MotCle8> getMotscles() {
    return motscles;
}

public void setMotscles(Set<MotCle8> motscles) {
    this.motscles = motscles;
}
public void ajouterMotCle(MotCle8 motCle){
    motsCles.add(motCle);
    motCle.getLivres().add(this);
}
```

Dans `MotCle8`

```
private List<Livre8> livres = new ArrayList<Livre8>();
@ManyToMany(mappedBy = "motscles")
public List<Livre8> getLivres() {
     return livres;
}

public void setLivres(List<Livre8> livres) {
    this.livres = livres;
}
```