Applications orientées données (NSY135)

7 – Introduction à JPA et Hibernate

Auteurs: Raphaël Fournier-S'niehotta et Philippe Rigaux
(philippe.rigaux@cnam.fr,fournier@cnam.fr)

Département d'informatique
Conservatoire National des Arts & Métiers, Paris, France

Introduction

- Avec les principes d'indépendance de MVC, nous avons *encapsulé les lignes d'une table relationnelle sous forme d'objets*
- Les accès à la base sont isolés dans la couche Modèle
- Ils sont transparents pour le reste de l'application
- Avantage
  - possibilité de faire évoluer la couche d'accès aux données indépendamment du reste de l'application;
  - possibilité de tester séparément chaque couche, et même la couche modèle sans avoir à se connecter à la base (par création d'entités non persistantes).

Plan du cours

Incompatibilité relationnel et objet

Les représentations utilisées dans la base de données et dans une application objet sont incompatibles :

- Le rôle d'un système ORM est de convertir *automatiquement*, *à la demande*, la base de données sous forme d'un *graphe d'objet*.
- L'ORM s'appuie pour cela sur une *configuration* associant les *classes* du modèle fonctionnel et le schéma de la base de donnés.
- L'ORM génère des requêtes SQL qui permettent de matérialiser ce graphe ou une partie de ce graphe en fonction des besoins.

Exemple de structure relationnelle

id	titre	année	idReal
17	Pulp Fiction	1994	37
54	Le Parrain	1972	64
57	Jackie Brown	1997	37

id	nom	prénom	naissance
1	Coppola	Sofia	1971
37	Tarantino	Quentin	1963
64	Coppola	Francis	1939

- la représentation d'une association en relationnel est un mécanisme de référence **par valeur** (et pas par adresse),
- la clé primaire d'une entité (ligne) dans une table est utilisée comme attribut (dit **clé étrangère**) d'une autre entité (ligne).

- Ici, le réalisateur d'un film est un artiste dans la table **Artiste**, identifié par une clé nommée **id**.
- Pour faire référence à cet artiste dans la table **Film**, on ajoute un attribut **clé étrangère idReal** dont la valeur est l'identifiant de l'artiste.
- Dans l'exemple ci-dessus cet identifiant est 37 pour *Pulp Fiction* et *Jackie Brown*, 64 pour *Le parrain*, avec la correspondance à une et une seule ligne dans la table **Artiste**.

Représentation équivalente en langage objet

- Nous avons des objets, et la capacité à référencer un objet depuis un autre objet (cette fois par un système d'adressage).
- supposant que nous avons une classe **Artiste** et une classe **Film** en java
- le fait qu'un film ait un réalisateur se représente par une référence à un objet **Artiste** sous forme d'une propriété de la classe **Film**.

```
class Film {
  (...)
  Artiste realisateur;
  (...)
}
```

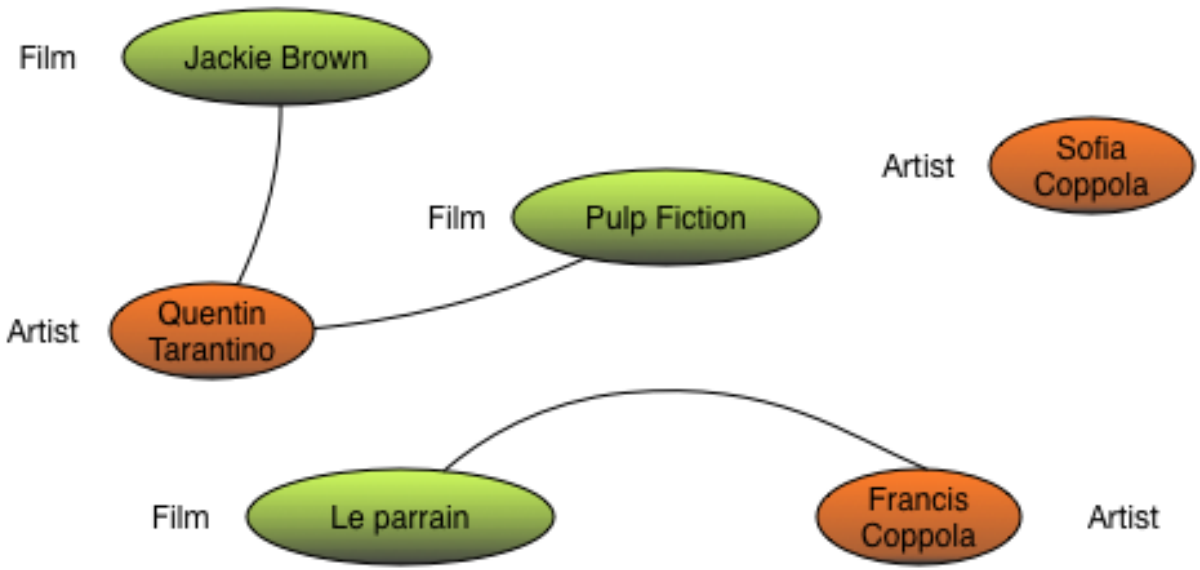
- Et du côté de la classe **Artiste**, nous pouvons représenter les films réalisés par un artiste par un ensemble.

```
class Artiste {
  (...)
  Set<Film> filmsDiriges;
  (...)
}
```

Mapping

- Différence importante entre les deux représentations des associations :
  - En relationnel, elles sont bi-directionnelles
  - Il est toujours possible par une requête SQL (une jointure) de trouver les films réalisés par un artiste, ou le réalisateur d'un film
  - En java, le lien peut être représenté de chaque côté de l'association, ou des deux.
  - Par exemple, on pourrait mettre la propriété **réalisateur** dans la classe **Film**, mais pas **filmsDiriges** dans la classe **Artiste**, ou l'inverse, ou les deux.
  - Cette subtilité est la source de quelques options plus ou moins obscures dans les systèmes ORM, nous y reviendrons.

Graphe d'objets Java



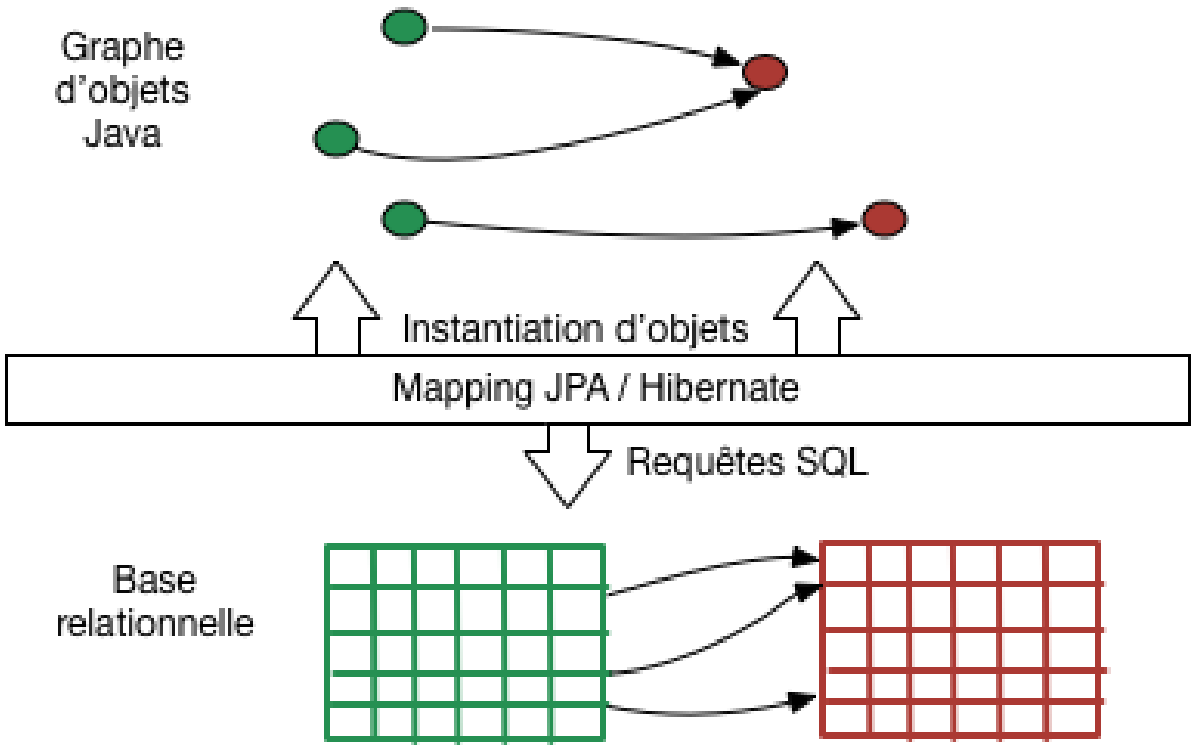
Gestion des différences

- pour bien gérer les différences entre représentations, il nous faut une approche systématique
- on pourrait choisir que :
  - on crée une classe pour chaque entité;
  - on équipe cette classe avec des méthodes *create()*, *get()*, *delete()*, *search()*, ..., que l'on implante en SQL/JDBC;
  - on implante la navigation d'une entité à une autre, à l'aide de requêtes SQL codées dans les classes et exécutées en JDBC.
- Avec cette approche :
  - la méthode *get(id)* serait clairement implantée par un **SELECT * From <table> WHERE id=:id**.
  - Le *create()* serait implanté par un **INSERT INTO ()**
  - etc.

Couche ORM

- Nous n'utiliserons pas cette approche pour au moins trois raisons:
  - concevoir nous-mêmes un tel mécanisme est un moyen très sûr de commettre des **erreurs de conception** (qui seraient coûteuses à réparer, une fois que des milliers de lignes de code auraient été produites)
  - la tâche répétitive de produire des requêtes toujours semblables nous inciterait rapidement à chercher un **moyen automatisé**;
  - et enfin - et surtout - ce mécanisme dit **d'association objet-relationnel** (ORM pour **Objet-relational mapping**) **existe**, il a été mis au point, implanté, et validé depuis de longues années
- Non seulement de tels outils existent, mais de plus ils sont normalisés en Java, sous le nom générique de **Java Persistence API** ou **JPA**.
- JPA est essentiellement une spécification intégrée au JEE qui vise à standardiser la couche d'association entre une base relationnelle et une application Java construite sur des objets.
- JPA n'est qu'une spécification visant à la standardisation

Mapping d'une BD relationnelle en graphe d'objets Java



JPA et Hibernate

- Cette standardisation vient après l'émergence de plusieurs *frameworks* qui ont exploré les meilleures pratiques pour la réalisation d'un système ORM.
- Le plus connu est sans doute **Hibernate**, que nous allons utiliser dans ce qui suit.
- Avec l'apparition de JPA, ces frameworks vont tendre à devenir des *implémantations* particulières de la norme
- Comme JPA est une sorte de facteur commun, il s'avère que chaque *framework* propose des extensions spécifiques, souvent très utiles.
- Il existe une implantation de référence de JPA, EclipseLink `http://www.eclipse.org/eclipselink/`

Introduction à JPA

- JPA définit une interface dans le package **javax.persistence.***.
- La définition des associations avec la base de données s'appuie essentiellement sur les annotations Java,
- évite d'avoir de longs fichiers de configuration XML qui doivent être maintenus en parallèle aux fichiers de code.
- On obtient une manière assez légère de définir une sorte de base de données objet *virtuelle*
- qui est matérialisée au cours de l'exécution d'une application par des requêtes SQL produites par le *framework* sous-jacent.

- dans la suite, on utilise Hibernate comme *framework* de persistance, et l'on tâche de respecter JPA autant que possible
- cela devrait faciliter la transition vers d'autres frameworks, pour vous plus tard