Applications orientées données (NSY135)

8 – JPA : le mapping

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

Plan du cours

S3 Associations bidirectionnelles

Introduction

- On souhaite représenter l'association de manière bi-directionnelle, ce qui sera souhaité dans une bonne partie des cas
- dans notre exemple, on voudra naviguer dans l'association **réalise** à partir d'un metteur en scène ou d'un film
- On peut tout à fait représenter l'association des deux côtés, en plaçant donc simultanément dans la classe **Film** la spécification:

```
@ManyToOne
@JoinColumn (name="id_realisateur")
private Artiste realisateur;
public void setRealisateur(Artiste a) {realisateur = a;}
public Artiste getRealisateur() {return realisateur;}
```

- et dans la classe **Artiste** la spécification:

```
@OneToMany
@JoinColumn(name="id_realisateur")
private Set<Film> filmsRealises = new HashSet<Film>();
public void addFilmsRealise(Film f) {filmsRealises.add(f) ;}
public Set<Film> getFilmsRealises() {return filmsRealises;}
```

- Cette fois l'association est représentée des deux côtés.

Problème avec cette méthode

- Cela soulève cependant un problème dû au fait que là où il y a deux emplacements distincts en java, il n'y en a qu'un en relationnel.
- Effectuons une dernière modification de **insertFilm()** en affectant les **deux** côtés de l'assocation.

```
// Le réalisateur de Gravity est Alfonso Cuar\'on
gravity.setRealisateur(cuaron);
// Films réalisés par A. Cuaron?
for (Film f : cuaron.getFilmsRealises()) {
System.out.println("Curaron a réalisé " + f.getTitre());
}
// Alfonso Cuaron a réalisé Gravity
cuaron.addFilmsRealise(gravity);
```

- Au passage, on a aussi affiché la liste des films réalisés par Alfonso Cuarón.
- Première remarque: le code est alourdi par la nécessité d'appeler **setRealisateur()** **et addFilmRealises()** pour attacher les deux bouts de l'association aux objets correspondants.

Problème (suite)

- Exécutons à nouveau **insertFilm()** (après avoir supprimé à nouveau de la base l'artiste et le film) et regardons ce qui se passe.

```
Hibernate: select genre_.code from Genre genre_ where genre_.code=?
Hibernate: insert into Film (annee, genre, code_pays,
              id_realisateur, resume, titre) values (?, ?, ?, ?, ?, ?)
Hibernate: insert into Artiste (annee_naissance, nom, prenom) values (?, ?, ?)
Hibernate: update Film set annee=?, genre=?, code_pays=?,
              id_realisateur=?, resume=?, titre=? where id=?
Hibernate: update Film set id_realisateur=? where id=?
```

- Ce que **l'on ne voit pas** : notre code devrait afficher la liste des films réalisés par Alfonso Cuarón. Rien ne s'affiche, car nous avons demandé cet affichage **après** avoir dit que Cuarón est le réalisateur de **Gravity**, mais **avant** d'avoir dit que **Gravity** fait partie des films réalisés par Cuarón.
- En d'autres termes, Java **ne sait pas**, quand on renseigne un côté de l'association, **déduire** l'autre côté.
- Il faut donc bien appeler **setRealisateur()** et **addFilmRealises()** pour que notre graphe d'objets Java soit cohérent. Si on ne le fait pas, l'incohérence du graphe est potentiellement une source d'erreur pour l'application.

Problème (suite)

- Exécutons à nouveau **insertFilm()** (après avoir supprimé à nouveau de la base l'artiste et le film) et regardons ce qui se passe.

```
Hibernate: select genre_.code from Genre genre_ where genre_.code=?
Hibernate: insert into Film (annee, genre, code_pays,
              id_realisateur, resume, titre) values (?, ?, ?, ?, ?, ?)
Hibernate: insert into Artiste (annee_naissance, nom, prenom) values (?, ?, ?)
Hibernate: update Film set annee=?, genre=?, code_pays=?,
              id_realisateur=?, resume=?, titre=? where id=?
Hibernate: update Film set id_realisateur=? where id=?
```

- Seconde remarque: Hibernate effectue deux **update**, alors qu'un seul suffirait. Là encore, c'est parce que les deux spécifications de chaque bout de l'association sont indépendantes l'une de l'autre.
- **Nous sommes en train d'analyser une des principales difficultés conceptuelles du *mapping* objet-relationnel**

La solution : déclarer un responsable

- Nous avons deux objets java qui sont déclarés comme persistants (le fim, l'artiste)
- Hibernate surveille ces objets et déclenche une mise à jour dans la base dès que leur état est modifié
- Comme la réalisation d'une association implique la modification des **deux** objets, alors que la base relationnelle représente l'association en un seul endroit (la clé étrangère), on obtient des mises à jour redondantes.
- Est-ce **vraiment** grave? A priori on pourrait vivre avec
- le principal inconvénient prévisible étant un surplus de requêtes transmises à la base, ce qui peut éventuellement pénaliser les performances.
- Hibernate propose une solution: déclarer qu'un côté de l'association est **responsable** de la mise à jour.
- Cela se fait avec l'attribut **mappedBy** :

```
@OneToMany(mappedBy="realisateur")
private Set<Film> filmsRealises = new HashSet<Film>();
```

- On indique donc que le **mapping** de l'association est pris en charge par la classe **Film**, **et on supprime l'annotation @JoinColumn puisqu'elle devient inutile**.
- Ce que dit **mappedBy** en l'occurrence, c'est qu'un objet associé de la classe **Film** maintient un lien avec une instance de la classe courante (un **Artiste**) grâce à une propriété nommée **realisateur**.
- Hibernate, en allant inspecter la classe **Film**, trouvera que **realisateur** est **mappé** avec la base de données par le biais de la clé étrangère **id_realisateur**. Dans **Film**, on trouve en effet:

```
@ManyToOne
@JoinColumn (name="id_realisateur")
private Artiste realisateur;
```

- Toutes les informations nécessaires sont donc disponibles, et représentées une seule fois

La solution (suite)

- En conséquence, une modification de l'état de l'association au niveau d'un artiste **ne déclenchera pas** d'**update** dans la base de données. En d'autres termes, l'instruction

```
// Alfonso Cuaron a réalisé Gravity
cuaron.addFilmsRealise(gravity);
```

  ne sera pas synchronisée dans la base, et rendue persistante. On a sauvé un **update**, mais introduit une source d'incohérence.

- Une petite modification de la méthode **addFilmsRealises()** suffit à prévenir le problème :

```
public void addFilmsRealise(Film f) {
f.setRealisateur(this);
filmsRealises.add(f) ;
}
```

- L'astuce consiste à appeler **setRealisateur()** pour garantir la mise à jour dans la base dans tous les cas.

Récapitulons le “un à plusieurs”

- Voici donc la méthode recommandée pour une association **un à plusieurs**.
- Nous prenons comme guide l’association entre les films et leurs réalisateurs.
- **Film** est du côté **plusieurs** (un réalisateur met en scène **plusieurs** films), **Artiste** du côté **un** (un film a **un** metteur en scène).
- Dans la base relationnelle, c’est du côté **plusieurs** que l’on trouve la clé étrangère.

- Du côté “plusieurs” :
  - On donne la spécification du **mapping** avec la base de données. Les annotations sont **@ManyToOne** et **@JoinColumn**.

```
@ManyToOne
@JoinColumn (name="id_realisateur")
private Artiste realisateur;
```

  - Dans le jargon ORM, ce côté est "responsable" de la gestion du **mapping**.

Récapitulons le “un à plusieurs”

- Voici donc la méthode recommandée pour une association **un à plusieurs**.
- Nous prenons comme guide l’association entre les films et leurs réalisateurs.
- **Film** est du côté **plusieurs** (un réalisateur met en scène **plusieurs** films), **Artiste** du côté **un** (un film a **un** metteur en scène).
- Dans la base relationnelle, c’est du côté **plusieurs** que l’on trouve la clé étrangère.

- Du côté “un” :
  - On indique avec quelle **classe responsable** on est associé, et quelle **propriété** dans cette classe représente l’association.
  - Aucune référence directe à la base de données n’est faite.
  - L’annotation est **@OneToMany** avec l’option **mappedBy** qui indique que la responsabilité de l’association est déléguée à la classe référencée, dans laquelle l’association elle-même est représentée par la valeur de l’attribut **mappedBy**. Par exemple:

```
@OneToMany(mappedBy="realisateur")
private Set<Film> filmsRealises = new HashSet<Film>();
```

  - Hibernate inspectera la propriété **realisateur** dans la classe responsable pour déterminer les informations de jointure si nécessaire.

Les accesseurs

- Du côté **plusieurs**, les accesseurs sont standards. Pour la classe **Film**.
- et pour la classe **Artiste**, on prend soin d'appeler l'accesseur de la classe responsable pour garantir la synchronisation avec la base et la cohérence du graphe.

```
public void addFilmsRealise(Film f) {
  f.setRealisateur(this);
  filmsRealises.add(f) ;
}
public Set<Film> getFilmsRealises() {return filmsRealises;}
```

Code d'association

- Dans ces conditions, voici le code pour créer une association.

```
1   public void insertFilm() {
2     session.beginTransaction();
3     Film gravity = new Film();
4     gravity.setTitre("Gravity");
5     gravity.setAnnee(2013);
6
7     Genre genre = new Genre();
8     genre.setCode("Science-fiction");
9     gravity.setGenre(genre);
10
11    Artiste cuaron = new Artiste();
12    cuaron.setPrenom("Alfonso");
13    cuaron.setNom("Cuaron");
14
15    // Alfonso Cuaron a réalisé Gravity
16    cuaron.addFilmsRealise(gravity);
17
18    // Sauvegardons dans la base
19    session.save(gravity);
20    session.save(cuaron);
21    session.getTransaction().commit();
22  }
```

- L'association est réalisée des deux côtés en java par un seul appel à la méthode **addFilmsRealises()**.
- il reste nécessaire de sauvegarder individuellement le film et l'artiste.
- Cela peut se gérer avec des annotations **Cascade**.