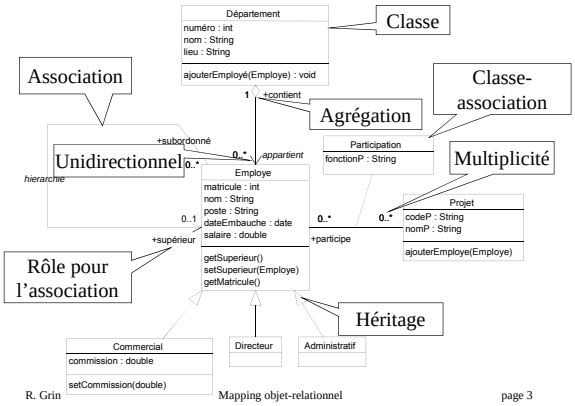
Correspondance Objet - Relationnel

Université Française d'Egypte
Version O 0.9 – 7/10/12
Richard Grin

Introduction

- Ce support expose les notions utiles pour faire correspondre
  - les données d'un modèle objet
  - avec les données d'une BD relationnelle



2 paradigmes bien différents

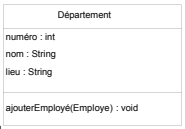
- Le paradigme objet est plus riche
- Dès qu'un modèle objet est complexe, il n'est pas simple de lui faire correspondre un modèle relationnel
- Par exemple, pas de notion d'héritage ou d'association dans le paradigme relationnel

Plan

1. Exemple simple de traduction d'une classe en une table
2. Identification
3. Traduction des associations
4. Traduction de l'héritage
5. Récupération des objets associés à un objet

Exemple simple de traduction d'une classe en une table

Traduction d'une classe en une table



- Dans le cas les plus simples
  une classe ↔ une table
  un objet ↔ une ligne de la table
- Classe Département traduite par la table
  DÉPARTEMENT(numéro, nom, lieu)

Création de la table en SQL

```
create table departement (
numero integer
    constraint pk_dept primary key,
nom varchar(15),
lieu varchar(15))
```


Identification

Identification dans le monde Objet

- Tout objet est identifiable indépendamment des données qu'il contient
- 2 objets distincts peuvent avoir exactement les mêmes valeurs pour leurs propriétés

Identification dans le monde relationnel

- Si 2 lignes ont les mêmes valeurs, il est impossible de les différencier
- Toute table devrait avoir une clé primaire
  (rappel : préférable de travailler avec une clé primaire non significative)

Propriété « clé primaire » pour les objets

- On ajoute souvent une propriété à une classe persistante pour représenter la clé primaire non significative de la table correspondante (facilite la correspondance objet-relationnel)

Problème des duplications

- Une ligne de la BD ne doit pas être représentée par plusieurs objets distincts
- Possibilité de pertes de données sinon

Exemple de duplication

- Un objet **`p1`** de la classe **`Produit`** de code « 1003 » est créé en mémoire à l'occasion d'une navigation à partir d'une ligne de facture
- On peut retrouver le même produit en navigant depuis une autre facture, ou en cherchant tous les produits qui vérifient un certain critère
- Il ne faut pas créer un autre objet **`p2`** différent du premier objet **`p1`**

Pour éviter des duplications

- En mémoire, un objet « cache » répertorie tous les objets créés liés à la BD et conserve leur clé primaire
- Lors d'une recherche dans la base, le cache intervient :
  - il fournit l'objet cherché s'il le contient déjà
  - sinon, l'objet est créé avec les données récupérées dans la BD, et ajouté au cache

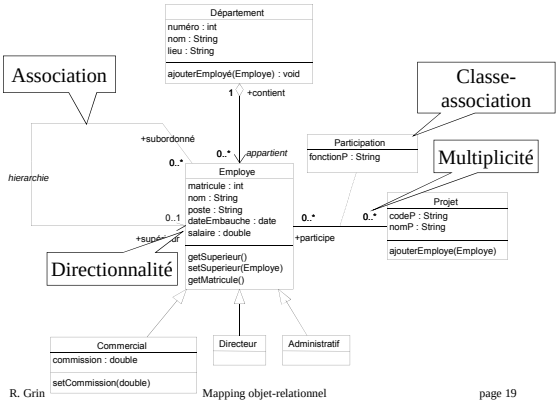
Objets intégrés

- Le modèle objet peut avoir une granularité plus fine que le modèle relationnel
- Les instances de certaines classes peuvent être sauvegardées dans la même table qu'une autre classe
- Ces instances sont appelées des objets intégrés (*embedded*) et ne nécessitent pas d'identificateur « clé primaire »

Exemple

- Une classe **`Adresse`** peut ne pas avoir de correspondance sous la forme d'une table séparée
- Les attributs de la classe **`Adresse`** sont intégrés dans la table qui représente la classe **`Client`**

Traduction des associations



Une association en objet

- Peut être représentée par
  - une variable d'instance référençant l'objet associé (association 1:1 ou N:1)
  - une variable d'instance de type collection contenant les objets associés (association 1:N ou M:N)
  - une classe « association » (M:N)

Exemple pour une association 1:N (ou N:1)

```
class Département {
  private Collection<Employe> employes;
  . . .
}
```

```
class Employé {
  private Département département;
  . . .
}
```


Association en relationnel

- Peut être représentée par
  - une clé étrangère (association 1:1, N:1 ou 1:N)
  - une table association (M:N)

Exemple pour une association 1:N

```
create table DEPT ( . . .)
```

```
create table EMPLOYE (
  . . .
  num_dept integer references DEPT)
```


Erreur de débutant

- Ne pas confondre le monde des objets et le relationnel
- En POO les associations se font entre les objets et sont représentées par des objets ou des collections d'objets, pas par des clés ou des collections de clés
- L'association de **`Employé`** vers **`Département`** :
  `private Département département;`
  Faux :
  ~~**`private int numéroDépartement;`**~~

Navigabilité des associations - Objet

- Une association peut être bi ou unidirectionnelle
- Unidirectionnelle : la classe **`Employé`** a une variable d'instance **`departement`** mais la classe **`Département`** n'a pas de collection d'employés
- En partant d'un département, on n'a alors pas de moyen simple de retrouver ses employés

Navigabilité des associations - relationnel

- Une association est toujours bidirectionnelle : on peut toujours « naviguer » vers l'autre bout d'une association
- Par exemple, pour trouver les employés du département 10 :

```
select matr, nomE from employe
where dept = 10
```


Associations M:N - Objet

- 2 façons de représenter une association M:N
  - une collection dans chacune des classes associées
  - une classe association qui représente une instance de l'association
- Si l'association contient des propriétés, seule la 2ème façon convient

Exemple – solution 1

```
class Employé {
  private Collection<Projet> projets;
  . . .
}
```

```
class Projet {
  private Collection<Employé> employés;
  . . .
}
```


Exemple – solution 2

```
class Employé {
  . . .
}
```

```
class Projet {
  . . .
}
```

`Employé` et **`Projet`** peuvent contenir une collection de **`Participation`** pour permettre la navigation

```
class Participation {
  private Employé employé;
  private Projet projet;
  . . .
}
```


Association M:N - Relationnel

- Une association M:N est traduite par une table association
- La clé primaire de cette table est souvent formée des 2 clés des tables associées

Exemple

- PARTICIPATION(***matr***, ***codeP***)
- En SQL :

```
create table participation (
  matr integer references emp,
  codeP varchar(2) references projet,
  primary key(matr, codeP))
```


Association M:N avec propriétés

- Si l'association a des propriétés, elles sont ajoutées dans la classe association et dans la table association

En objet

```
class Participation {
    private Employé employé;
    private Projet projet;
    private String fonction;
    . . .
}
```


En relationnel

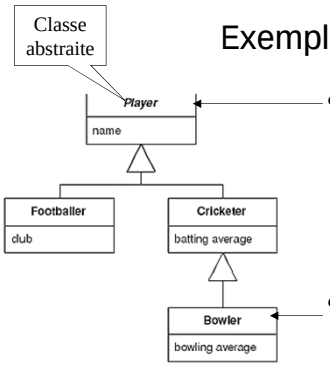
- PARTICIPATION(***matr***, ***codeP***, fonctionP)

Association bidirectionnelle

- Plus complexe à gérer dans le monde objet
- En relationnel si un employé change de département, il suffit de changer le numéro de département dans la ligne de l'employé
- En objet, il faut
  - modifier le département dans la classe des employés
  - enlever l'employé de la collection des employés de l'ancien département
  - le rajouter dans la collection de son nouveau département

Traduction de l'héritage

Exemple



- Sportifs qui peuvent être des footballeurs ou des joueurs de cricket
- Les joueurs de cricket qui lancent la balle : les *bowlers*

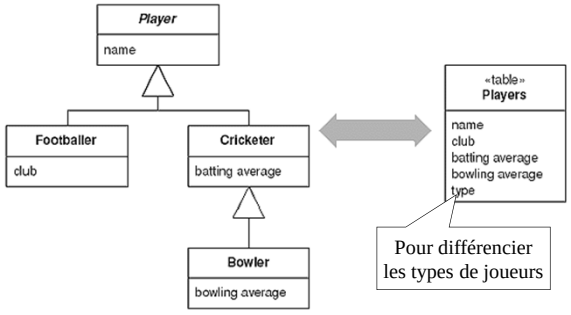
3 stratégies de traduction

- Représenter toutes les classes d'une arborescence d'héritage par une seule table
- Représenter chaque classe instanciable (concrète) par une table
- Représenter chaque classe, même les classes abstraites, par une table

Vocabulaire : « polymorphe »

- Association polymorphe : association vers une instance d'une classe C *ou d'une des classes filles* de C
- Exemple : une société garde une référence vers un sportif (sport quelconque) qu'elle sponsorise
- Requête polymorphe : renvoie une information conservée dans les instances d'une classe C *ou des classes filles* de C
- Exemple : requête qui renvoie les noms de tous les sportifs

Toutes les classes traduites par une seule table



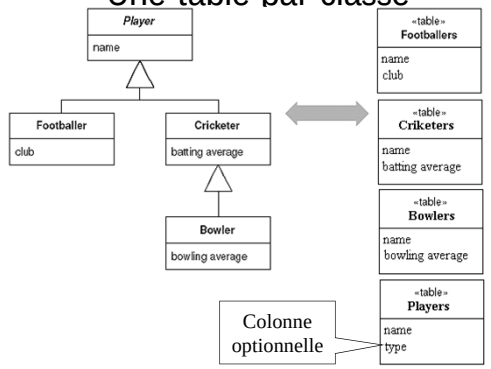
Avantages

- Souvent la solution la plus simple à mettre en place
- Permet les requêtes et associations polymorphes
- Solution la plus fréquemment choisie

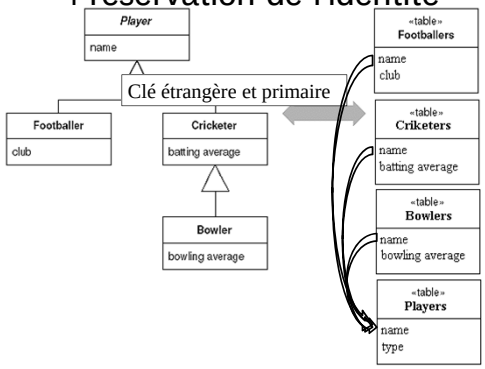
Inconvénients

- Oblige à avoir de nombreuses colonnes qui contiennent la valeur **`NULL`**
- On ne peut déclarer ces colonnes « **`NOT NULL`** », même si cette contrainte est vraie pour une des sous-classes

Une table par classe



Préservation de l'identité



Pour représenter un bowler

- Table Player : (Bill, 'bowler')
- Table Criketer : (Bill, 50)
- Table Bowler : (Bill, 48)
- En réalité il faudrait utiliser une clé non significative ; par exemple :
- Table Player : (158, Bill, 'bowler')
- Table Criketer : (158, 50)
- Table Bowler : (158, 48)

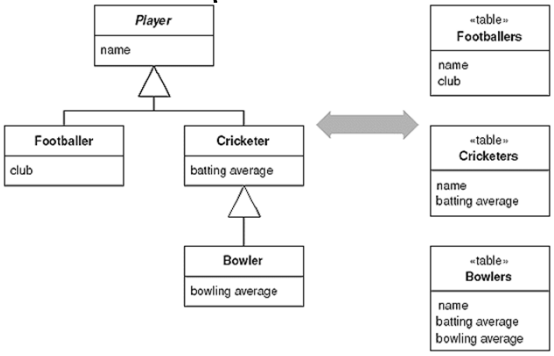
Avantage

- Simple : bijection entre les classes et les tables
- Permet les requêtes et associations polymorphes

Inconvénient

- Si l'héritage est complexe, mauvaises performances à cause des nombreuses jointures pour reconstituer les informations éparpillées dans les tables

Une table par classe concrète



Avantages

- C'est la méthode la plus naturelle : une table par type d'entité
- Pas de jointures pour retrouver les informations

Inconvénient

- Ne peut traduire simplement les associations et les requêtes polymorphes
- Par exemple, traduire une classe qui référence un joueur d'un sport quelconque
- En effet, aucune table ne correspond à un joueur d'un sport quelconque
- On ne peut pas, par exemple, imposer une contrainte d'intégrité référentielle (clé étrangère) dans la table qui référence le joueur

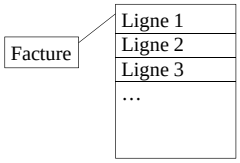
Récupération des objets associés à un objet

Facture

Facture

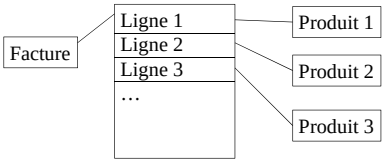
- Recherche d'une facture qui vérifie un certain critère
- Un objet de la classe **`Facture`** est créé

Facture



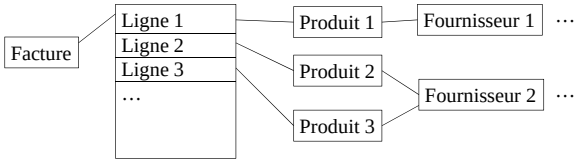
- Est-ce que les objets **`LigneFacture`** associés à cette facture doivent aussi être créés ?

Facture



- Si la réponse est positive, faut-il créer aussi les objets **`Produit`** correspondants à chaque ligne de facture ?

Facture



- Et si la réponse est positive, …

Le problème

- Le risque est de créer un très grand nombre d'objets associés
- Si on veut connaître seulement la date de facturation, ces objets associés sont totalement inutiles
- D'où de mauvaises performances, sans raisons valables

2 stratégies

- Un objet est créé à partir des données de la base
- 2 stratégies pour les objets associés :
  - récupération immédiate et création des objets associés
  - les objets associés ne sont pas créés tout de suite, mais seulement lorsque l'application en a vraiment besoin

Récupération « à la demande »

- Traduction de *lazy loading*
- Les objets associés ne sont pas immédiatement créés
- Ils sont remplacés par des objets « *proxy* » qui ne contiennent que l'information nécessaire (clé primaire) pour créer les « vrais » objets associés
- Les « vrais » objets ne seront créés par la suite que si le code en a besoin, en lançant une requête dans la base de données

Exemple (1/2)

- Le code recherche la facture de numéro 456
- Un objet **`f`** de la classe **`Facture`** est créé
- Les objets correspondants aux lignes de la facture **`f`** sont remplacés par des objets d'une classe « proxy »
- Ces objets « proxy » ne contiennent que la clé primaire des lignes de la facture dans la table LIGNE_FACTURE

Exemple (2/2)

- Si le programme contient ensuite le code **`f.getLigne(i).getQuantite()`**

1. L'objet **`LigneFacture`** correspondant à la i^{ème} ligne de facture est créé en allant chercher les informations dans la base de données
2. Le message « **`getQuantite()`** » lui est envoyé

Problème des « N + 1 selects »

- On peut alors tomber sur le problème des « N + 1 selects »

Un exemple de « N + 1 selects »

- Afficher les lignes des 5000 factures éditées après une certaine date, avec une récupération « à la demande » des lignes :
- Les 5000 objets « Facture » sont d'abord récupérés (1 select)
- Dans une 2ème temps, pour chacune des 5000 factures, les lignes de facture associées sont récupérées (5000 selects)
- D'où un total de 5001 selects, alors qu'on peut obtenir le résultat avec un seul select !

Mauvaise solution pour les « N + 1 selects »

- Indiquer que la récupération des lignes de facture doit toujours être immédiate (dans des fichiers de configuration par exemple) ne convient pas
- En effet, dans d'autres circonstances, créer les lignes de factures ne servira à rien

Solutions possibles (1)

- Dans une méthode, lancer un ordre SQL ad hoc qui renvoie uniquement ce qui est recherché
- Exemple : pour trouver la facture qui a le plus gros total, inutile de charger toutes les factures et lignes de factures en mémoire
- Le plus simple et le plus performant (pas de création de nombreux objets)
- Ne convient pas si on veut des informations plus complexes sur les objets associés

Solutions possibles (2)

- Permettre de spécifier une stratégie spéciale pour une requête particulière
- On peut indiquer que le mode de récupération ne sera pas immédiat par défaut, mais que ce mode devra être immédiat pour une requête particulière

Code avec JPA

```
String texteQuery =
  "select f from Facture f "
  + " join fetch f.lignes "
  + " where f.date > '1/1/06'";
Query query = em.createQuery(texteQuery);
List<Facture> liste =
  (List<Facture>)query.getResultList();
Ligne ligne = liste.get(0).getLigne(0));
```

Ne génère aucune requête SQL

Conclusion de cette section

- La correspondance entre le modèle objet et le modèle relationnel n'est pas une tâche facile
- Cependant on verra que les outils et *framework* ORM réduisent grandement la difficulté et le nombre de lignes de code liés à cette tâche