L’approche ORM
Objet Relationnel Mapping

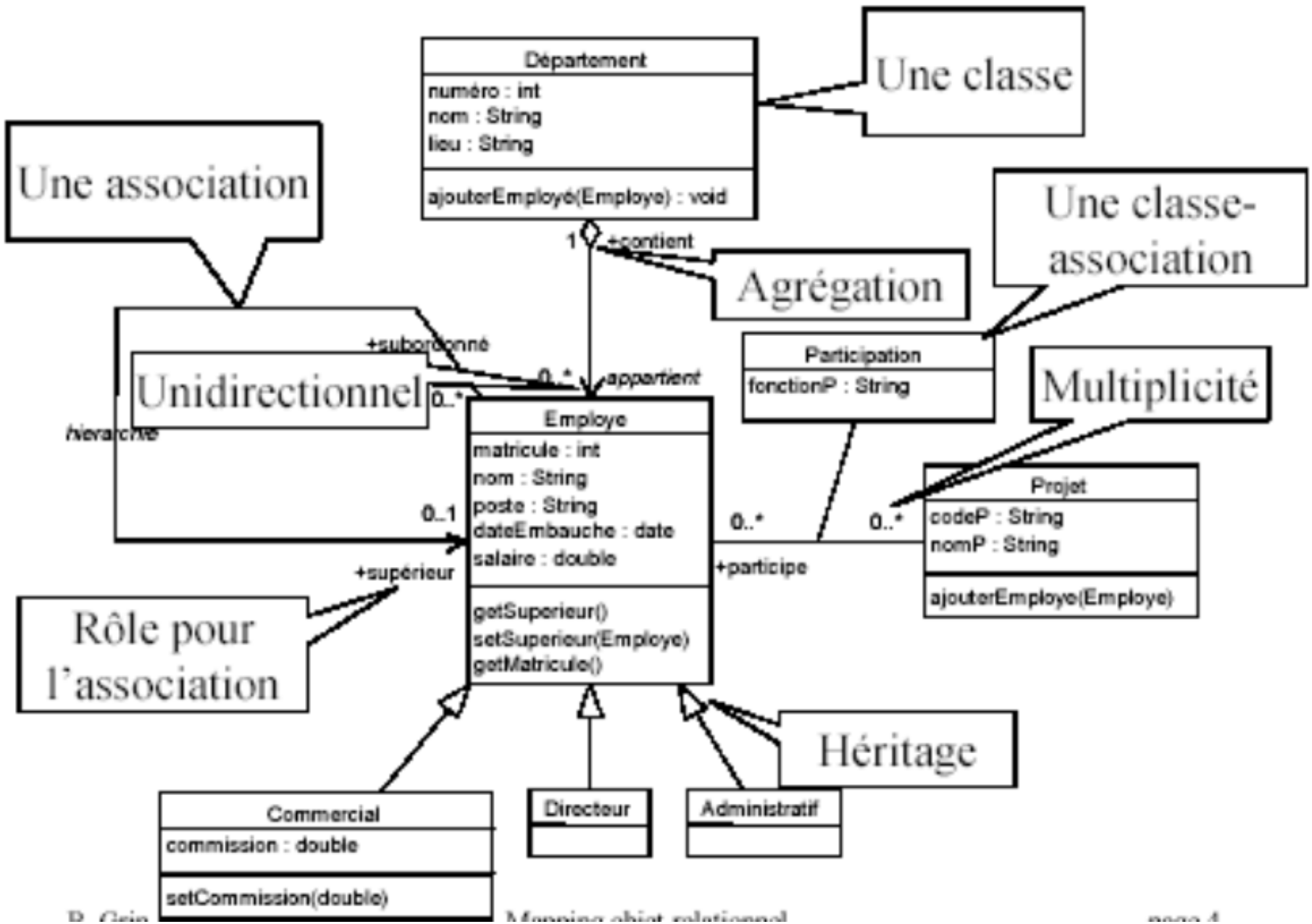
M. Bouneffa

Master Info

Introduction

Pour le schéma relationnel des données manipulées par une application objet, 2 possibilités :

- *le schéma existe déjà (construit souvent avec un schéma entités – associations)*
- *on part du modèle objet écrit dans les premières étapes de la construction de l'application* Cette partie du cours s'intéresse à cette dernière situation


Une classe
Département
numéro : int
nom : String
lieu : String
ajouterEmployé(Employe) : void
Une association
1
+contient
Agrégation
Une classe-association
+subordonné
Unidirectionnel
0..*
appartient
Participation
fonctionP : String
Multiplicité
hiérarchie
Employe
matricule : int
nom : String
poste : String
dateEmbauche : date
salaire : double
0..1
0..*
0..*
Projet
codeP : String
nomP : String
ajouterEmploye(Employe)
+supérieur
+participe
Rôle pour l'association
getSuperieur()
setSuperieur(Employe)
getMatricule()
Héritage
Commercial
commission : double
setCommission(double)
Directeur
Administratif

2 paradigmes

- Les paradigmes objet et relationnel sont bien distincts :
  - Le modèle objet est bien plus riche que le modèle relationnel et il n’est pas toujours simple de faire correspondre des objets et des données de tables relationnelles

Quelques problèmes du passage R □ □ Objet

- Identité des objets
- Traduction des associations
- Traduction de l'héritage
- Navigation entre les objets

Problèmes de base

- Les objets persistants doivent être enregistrés dans la base de données relationnelle
- Un objet a une structure complexe qui peut être représentée par un graphe
- Le plus souvent ce graphe est un arbre dont la racine correspond à l'objet et les fils correspondent aux valeurs des variables d' instance qui sont persistantes
- Il faut « aplatir » ce graphe pour le ranger dans la base de données relationnelle

Plan

1. Exemple simple de traduction d'une classe en une table
2. Les identificateurs
3. Traduction des associations
4. Traduction de l'héritage
5. Navigation entre les objets

Traduction d'une classe en une table

- Dans les cas les plus simples une classe est traduite en une table
- Chaque objet est conservé dans une ligne de la table

Exemple : la classe Département est traduite par la table

DÉPARTEMENT(numéro, nom, lieu)

En SQL

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

L’identification dans le monde de l’objet

- Tout objet est identifiable simplement dans les langages objets (adresse de l’ emplacement mémoire)
- 2 objets distincts peuvent avoir exactement les mêmes valeurs pour leurs propriétés

L’identification dans le monde du relationnel

- Dans les schémas relationnels toute table doit

comporter une clé primaire qui permet d’identifier une ligne parmi toutes les autres lignes de la même table

- Si on n’enregistre que les propriétés des

objets, on peut avoir des lignes identiques dans la base de données

2 solutions

- On peut être amené à ajouter un identificateur

à un objet, qui servira de clé primaire dans la base de données

- Les instances de certaines classes peuvent être sauvegardées dans la même table qu'une autre classe ; ces instances sont appelées des objets dépendants et ne nécessitent pas d'être identifiées par leurs propriétés

Exemple d’objets dépendants

- Les adresses des clients peuvent être enregistrées dans la table CLIENT, et pas dans une table à part

Eviter les identifiants significatifs

- Même si les objets peuvent être distingués par leurs propriétés, un identificateur sera souvent rajouté pour éviter les identificateurs significatifs
- Surtout si la clé significative est composée de plusieurs propriétés

Problème des duplications

- Une même ligne de la base de données ne doit pas être représentée par plusieurs objets en mémoire centrale (ou alors le code doit le savoir et en tenir compte)
- Si elle n'est pas gérée, cette situation peut conduire à des pertes de données

Exemple de duplication

- Un objet **Produit** correspondant au produit « s003 » 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

Pour éviter les duplications

- En mémoire, un objet « cache » répertorie tous les objets créés et conserve l'identité qu'ils ont dans la base (la clé primaire)
- Lors d'une navigation ou d'une recherche dans la base, le cache intervient
- Si l'objet est déjà en mémoire le cache le fournit, sinon, l'objet est créé avec les données récupérées dans la base de données

Traduction des associations

Dans le code objet une association peut être représentée par

- une variable d'instance représentant « l'autre » objet avec lequel se fait l'association (associations 1:1 ou N:1)
- une variable d'instance de type collection

représentant tous les autres objets avec lesquels se fait l'association (associations 1:N ou M:N)

- une classe « association » (M:N)

Exemples de classes pour une association 1:N (ou N:1)

- **class Département {**

private Collection<Employé> employes;

. . .

}

- **class Employé {**

private Département département;

. . .

}

Représentation d'une association

Dans le monde relationnel, une association peut être représentée par

- une ou plusieurs clés étrangères (associations 1:1, N:1 ou 1:N)
- une table association (associations M:N)

Navigabilité des associations

- Dans un modèle objet, une association peut être bi ou mono directionnelle
- Exemple d'association mono directionnelle : la classe **Employé** peut avoir une variable d' instance donnant son département mais la classe **Département** peut n'avoir aucune collection d'employés
- En partant d'un département, on n'a alors pas de moyen simple de retrouver ses employés

- Dans un schéma 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

On va d'abord étudier le cas des associations M:N qui nécessitent une table association dans le monde relationnel

Associations binaires M:N

- Dans le monde objet, on peut représenter une telle association de 2 façons différentes :
  - une collection dans chacune des classes associées, qui référence les objets associés
  - une classe association qui représente une instance de l'association
- Si l'association contient des propriétés, seule la 2ème façon convient

Exemples de classes solution 1

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

Exemples de classes : solution 2

- **class Employé { . . .}**
- **class Projet {. . .}**
- **class Participation {**

private Employé employé;

private Projet projet;

. . .}

Employé *ou* ***Projet*** *peuvent éventuellement contenir une collection de* ***Participation*** *pour faciliter la navigation*

Traduction d'une association binaire M:N

- Dans le relationnel, on doit créer une table pour traduire l'association
- La clé primaire de cette table est formée des 2 clés des tables qui représentent les classes qui interviennent dans l'association

Exemple de traduction d'une association binaire M:N

- PARTICIPATION (matr, codeP)
- En SQL :

create table participation (

matr integer *references emp* ,

codeP varchar(2) *references projet* ,

***primary key(matr, dept)*)**

Classe association (cas M:N)

- Si l'association a des propriétés, elles sont ajoutées dans la nouvelle table qui traduit l’association ou dans la classe association

En Objet

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

En relationnel

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

create table participation (

matr integer *references emp* ,

codeP varchar(2) *references projet* ,

fonctionP varchar(15),

primary key(matr, dept) **)**

Traduction d'une association binaire dont une multiplicité max est 1 (1:N ou 1:1)

- On peut traduire les associations N:1 ou 1:1 en ajoutant une nouvelle relation, comme pour une association M:N
- Mais le plus simple est d'ajouter la clé de l' autre classe dans la relation qui traduit la classe placée du côté opposé à la multiplicité 1
- La 1ère solution est plus souple mais plus coûteuse (jointures)

Exemple

- EMP(*matr* ,…, **dept**)
- En SQL :

create table emp (

matr integer primary key,

…

***dept smallint references dept*)**

Classe association (cas 1:N ou 1:1)

- Si l'association est représentée par une classe association, on peut ajouter les attributs de classe dans la relation qui reçoit la clé de l'autre classe (ou dans la nouvelle relation si on a choisi cette solution)
- Si les attributs de la classe association sont nombreux, il peut être préférable de traduire l'association par une relation à part (comme pour une association M:N)

Associations de degré > 2

- Le code objet peut traduire une association de degré > 2 de plusieurs façons différentes
- Le plus simple est sans doute de représenter l'association par une classe « association » qui contient des références vers les objets qui participent à l'association
- Mais on peut aussi représenter une telle association par des collections de classes qui contiennent des collections ou des références vers des classes qui contiennent des collections

Exemple

- Une réservation dans une compagnie aérienne peut être considérée comme une association entre les avions, les passagers et les numéros de siège
- En effet, un passager peut occuper plusieurs sièges (problème de santé par exemple)

Représentation par une classe

```
class Reservation {
private Vol vol;
private Passager passager;
private int siège;
. . .
}
```

Autre représentation

```
class Vol {
private
Collection<Reservation> résas;
. . .
}
class Reservation {
private Passager passager;
private int siège;
. . .
}
```

Traduction d’une association de degré > 2

- Dans le monde relationnel on crée une table pour traduire l’association
- La clé primaire de cette table est formée d'un sous-ensemble des clés des tables qui traduisent les classes qui interviennent dans l’association
- Le sous-ensemble peut être strict si une dépendance fonctionnelle existe entre ces clés

Exemple

- RESERVATION(***nVol*** , ***nSiège*** , ***codePassager*** , …)
- En SQL :

create table reservation(nvol varchar(10) *references vol* , nsiege integer, codePassager varchar(10) *references passager* , *primary key(nVol, nSiege, codePassager)* , …)

Gestion des association bidirectionnelles

- Gérer une association est plus complexe dans le monde objet que dans le monde relationnel, surtout si elle est bidirectionnelle
- Par exemple, si un employé change de département, il suffit de changer le numéro de département dans la ligne de l'employé
- En objet, il faut en plus enlever l'employé de la collection des employés du département et le rajouter dans la collection de son nouveau département

Gestion automatique de l’ autre bout d’une association

- Certains framework (EJB par exemple), automatisent une partie de la gestion des associations
- Ainsi, si le champ « dept » d’un employé est modifié, l’employé est passé automatiquement de la collection des employés du département d’origine dans celle du nouveau département
- D’autres frameworks comme Hibernate préfèrent ne rien automatiser

Traduction de l'héritage

Pour fixer les idées, et pour reprendre des dessins du très bon livre de Martin Fowler cité dans la bibliographie, on travaillera sur l'exemple suivant tiré du monde anglophone :

- des sportifs qui peuvent être des footballeurs ou des joueurs de crickets
- parmi les joueurs de crickets, on distingue les joueurs qui lancent la balle : les *bowlers*

Plusieurs méthodes de traduction

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

Remarque

- Cette remarque est valable quelle que soit la méthode de traduction de l'héritage
- Toutes les variables d'instance ne sont pas nécessairement accessibles par toutes les Classes filles (si elles ne sont pas **protected** et si elles ne possèdent pas d'accesseur **protected** ou **public**)
- Le code devra donc souvent trouver un moyen de demander à chaque classe de la hiérarchie d'héritage de participer à la persistance pour ses propres variables

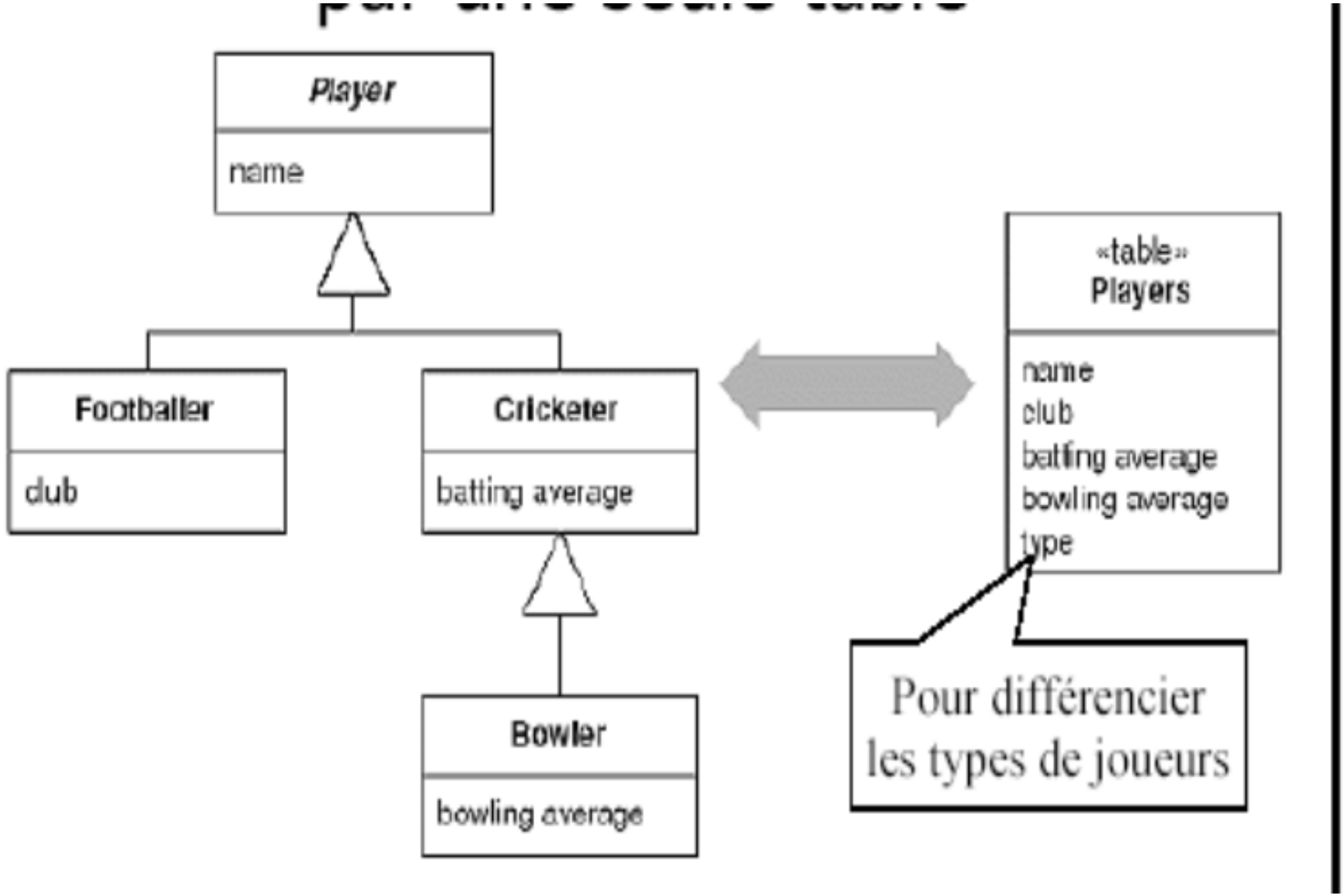
Remarque

- Par exemple, si une instance d'une classe fille veut se sauvegarder dans la base, il lui faudra trouver un moyen de récupérer les variables d'instance de ses classes mères
- Un moyen est de faire appel à une méthode de la classe mère par « super. », cette méthode récupérant les variables d'instance de la classe mère

Clés primaires

- Avec les 2 dernières méthodes, un objet peut avoir ses attributs répartis sur plusieurs tables (celles qui correspondent aux classes d'une même hiérarchie d'héritage)
- Son identité est alors préservée en donnant la même clé primaire aux lignes qui correspondent à l'objet dans les différentes tables
- Les clés primaires des tables correspondant aux classes filles sont des clés étrangères vers la clé primaire de la classe mère

Traduction de l'arborescence par une seule table



Avantages

- Souvent la solution la plus simple à mettre en place
- C’est d’ailleurs la solution la plus fréquemment choisie
- Permet les requêtes et associations polymorphes

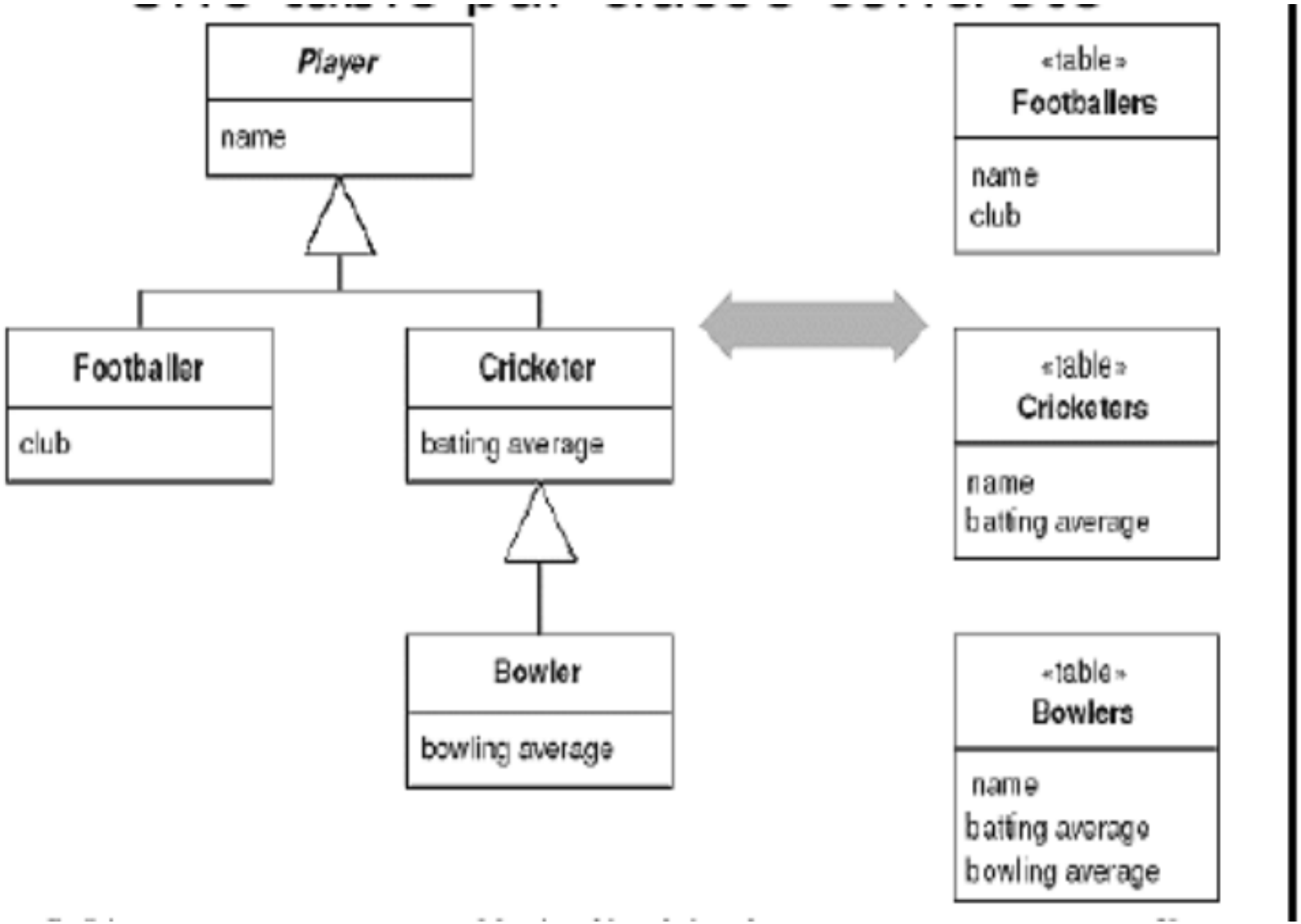
Polymorphisme

- Association polymorphes : une classe contient une référence vers une classe mère abstraite
- Par exemple, une société garde une référence vers un sportif (d'un sport quelconque) qu'elle sponsorise
- Requête polymorphe : cas où on veut une information qui correspond à une propriété qui appartient à une classe mère
- Par exemple, on veut les noms de tous les joueurs

Problèmes

- 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 concrète



Remarque

- Si une classe concrète se retrouve au milieu de l'arbre d'héritage avec des classes filles (comme ici la classe Cricketer), il faudra que les classes filles concrètes aient la même clé primaire que la classe mère concrète dans les tables correspondantes

Avantage

- C’est la méthode la plus naturelle : une table par type d’entité

Problème

- Ne peut traduire simplement les associations polymorphes
- Par exemple, une classe qui référence un joueur d'un sport quelconque (joueur de football ou de cricket)
- En effet, aucune table relationnelle ne correspond à un joueur d'un sport quelconque et on ne peut donc imposer une contrainte d'intégrité référentielle (clé étrangère)

Solution

- Aucune solution vraiment satisfaisante
- Des solutions partielles :
  - ignorer la contrainte d'intégrité référentielle
  - mettre plusieurs colonnes dans la table qui référence, une pour chaque table concrète référencée (mais ça sera difficile d'imposer l'unicité de la référence ; sur l' exemple, ça sera difficile d'imposer d' avoir une seule référence vers un joueur)

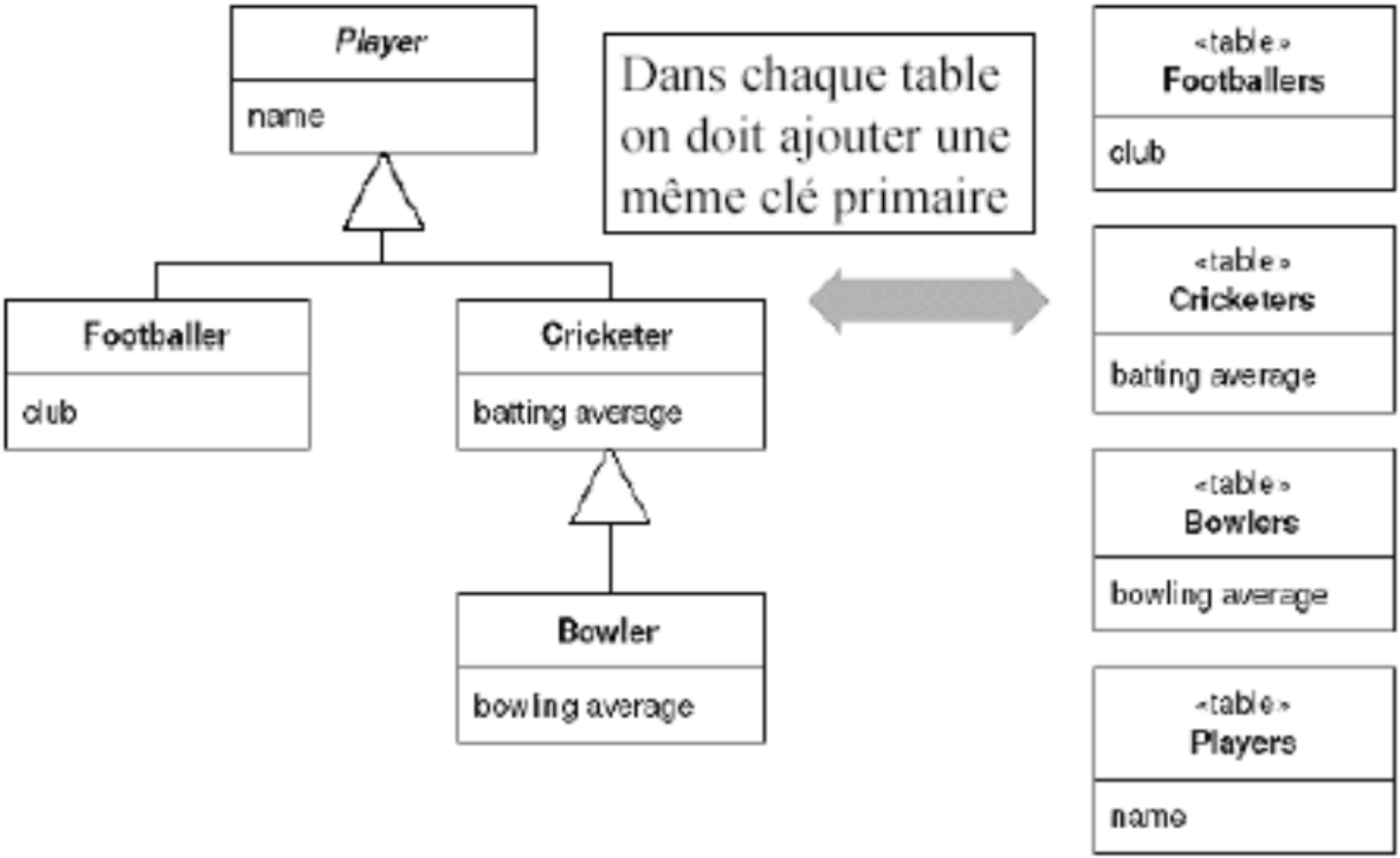
Autre problème

- Dans le même ordre d'idée, il est difficile d'interroger la base pour effectuer une requête polymorphe
- Par exemple, avoir le nom de tous les joueurs

Solution

- On devra lancer plusieurs selects (un pour chaque sous-classe concrète) et utiliser une union de ces selects
- Le select sera donc plus complexe et sans doute moins performant

Une table par classe



Remarque

- On a vu que toutes les tables qui correspondent à une hiérarchie d’ héritage ont la même clé primaire
- Pour récupérer les informations sur une instance d’une classe fille, il suffit de faire une jointure sur ces clés primaires

Avantage

- Simple : bijection entre les classes et les tables

Problèmes

- Si la hiérarchie d'héritage est complexe on sera amené à faire beaucoup de jointures pour reconstituer les informations éparpillées dans de nombreuses tables
- D'où des instructions complexes, mais surtout de mauvaises performances

Variantes

- Dans une arborescence d'héritage on peut mélanger toutes ces possibilités
- On peut, par exemple, créer plusieurs tables pour plusieurs branches de l'arborescence d'héritage
- Mais on risque de retomber sur les problèmes de la solution 2 (une table par classe concrète) avec les requêtes et les associations polymorphes

Héritage multiple

- Dans les cas où on traduit l'héritage par des tables séparées, on met comme identifiant dans les classes « filles » l'ensemble des identifiants des classes mères

Navigation entre les objets

Lorsque l'on crée un objet à partir des données récupérées dans la base de données, 2 stratégies vis-à-vis des objets dépendants de cet objet :

- récupération immédiate et création des objets dépendants
- on ne crée ces objets que lorsque l' application en a vraiment besoin

Une situation

- On recherche dans la base de données une facture qui vérifie un certain critère
- On va créer un objet de la classe **Facture** qui correspond à cette facture
- Est-ce qu'il faut aussi créer les objets **LigneFacture** associés à cette facture ?
- Et si la réponse est positive, faut-il créer aussi les objets **Produit** correspondants à chaque ligne de facture ?
- Et si la réponse est positive, …

Le problème

- On voit qu'ainsi on risque de créer un très grand nombre d'objets dépendants
- Si on veut récupérer cette facture pour connaître seulement la date de facturation, on voit que tous ces objets dépendants sont totalement inutiles
- On aura ainsi de mauvaises performances sans raisons valables

Récupération paresseuse

- La solution est le « *lazy loading* », mot à mot « récupération paresseuse », que l'on peut traduire par « récupération à la demande » ou « récupération retardée »

Récupération paresseuse

- Quand on crée un objet en mémoire à partir des données enregistrées dans la base de données, on ne charge pas immédiatement les objets dépendants
- Ces objets sont remplacés par des pseudoobjetsb qui permettront de ne charger les « vrais » objets (avec toutes leurs données) que si c'est vraiment indispensable

Exemple

- Si on recherche la facture de numéro 456, un objet **f** de la classe **Facture** est créé mais les objets correspondants aux lignes de la facture **f** ne sont pas créés
- Mais si le programme contient le code **f.getLigne(i).getQuantite()** l'objet **LigneFacture** correspondant à la ième ligne de facture est créé et le message **getQuantite()** lui est envoyé

Problème des N+1 selects

- On peut alors tomber sur le problème des « N + 1 selects » avec les instanciations paresseuses

Exemple

- On veut récupérer la facture qui a le plus gros total, parmi 5000 factures
- Si la récupération se fait en mode paresseux, on commence par récupérer les objets « Facture » (1 select), puis, pour chacune des 5000 factures, on récupère les lignes de facture associées pour calculer le total (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 pour cette association doit toujours être immédiate (dans les fichiers de configuration) ne convient pas le plus souvent
- En effet, dans d'autres circonstances, on ne souhaitera pas charger les lignes de factures pour éviter de créer trop d' objets inutiles

Solution possible (1)

- Lancer un ordre SQL ad hoc qui renvoie le total de la facture pour résoudre le problème ponctuel (sans création d'objets)
- C'est la solution la plus simple, mais elle peut ne pas convenir dans le cas où on veut des informations plus complexes sur les objets associés à l'objet

Solution possible (2)

- La plupart des outils de mapping permettent de spécifier une stratégie spéciale pour une requête particulière
- On peut alors indiquer que le mode de récupération sera « paresseux » par défaut, tout en spécifiant que la récupération des objets dépendants devra être immédiate pour cette requête particulière

Code avec Hibernate

```
List results = session.createCriteria(Item.class)
.add( Expression.eq("item.seller", user) )
.setFetchMode("bids", FetchMode.EAGER)
.list();
// évite la duplication de résultats (due au outer join)
Iterator items = new HashSet(results).iterator();
List maxAmounts = new ArrayList();
while (items.hasNext()) {
Item item = (Item) items.next();
BigDecimal maxAmount = new BigDecimal("0");
Iterator b = item.getBids().iterator();
while (b.hasNext()) {
Bid bid = (Bid) b.next();
if ( bid.getAmount().compareTo(maxAmount) == 1 )
maxAmount = bid.getAmount();
}
maxAmounts.add( new MaxAmount( item.getId(), maxAmount ) );
}
```