SIR

Persistance des objets

Olivier Barais
ISTIC
Université de Rennes 1
barais@irisa.fr

Au sommaire du cours

- Introduction au cours sur la persistance des objets
  - Problématiques
  - Rappel : organisation générale d'une application d'entreprise
- Mapping Objet / Relationnel
  - Exemple simple de traduction d'une classe en une table
  - Identification
  - Traduction des associations
  - Objet dépendant
  - Traduction de l'héritage
  - Navigation entre les objets
- JPA (Hibernate / Toplink) 2 cours
- XML et les objets

Cours bâti à partir de :

- Cours de Richard Grin (Université de Nice)
- Livre Hibernate 3.0 (Eyrolles)

Au sommaire aujourd'hui

- Des Objets aux bases de données
- Mapping Objet - Relationnel

Problématiques

On part des objets

- Vous avez vu jusque là comment concevoir la structure d'une base de données en partant d'une analyse/conception de type entité-association (E-A)

- Le point de vue de ce cours est différent :
  - On part d'une application objet et on veut rendre certaines données de l'application persistantes
  - La structure de la base de données devra donc correspondre au modèle objet de l'application

Le contexte

- Une application complexe multi-tiers
- Le cours s'intéresse à l'interface entre le tiers «métier» et le tiers « base de données »
- L'énorme majorité des bases de données sont relationnelles et de nombreuses applications modernes sont écrites avec des langages objet
- Comment une application écrite dans un langage objet peut effectuer la persistance de ses objets dans une base de données relationnelle

Problématique de la persistance des objets

- A la fin d'une session d'utilisation d'une application toutes les données des objets existant dans la mémoire vive de l'ordinateur sont perdues

- Rendre persistant un objet c'est sauvegarder ses données sur un support non volatile de telle sorte qu'un objet identique à cet objet pourra être recréé lors d'une session ultérieure

Pourquoi les BD relationnelles ?

- Position dominante
- Grande facilité et efficacité pour effectuer des recherches complexes dans des grandes bases de données
- Facilité d'adaptation à des besoins différents (applications avec des vues différentes des données)
- Facilité pour spécifier des contraintes d'intégrité sans programmation
- Une théorie solide et des normes reconnues

Pourquoi pas un SGBD Objet ?

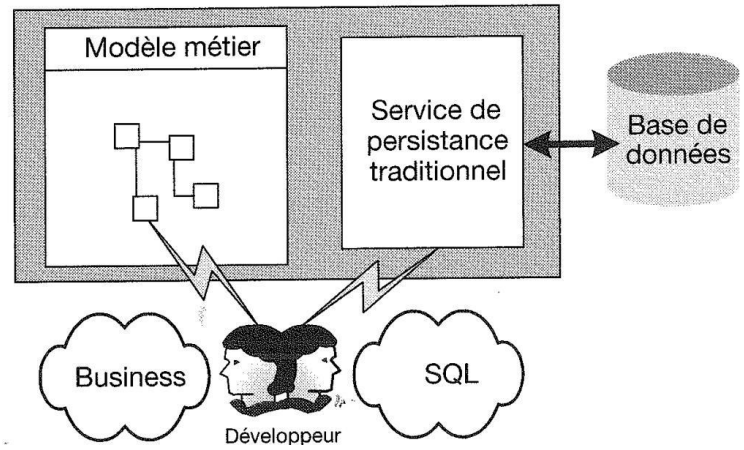
- Principale raison : de très nombreuses bases relationnelles en fonctionnement
- Les SGBD objet n'offre pas la capacité d'adaptation à la charge des SGBD relationnels
- Moins de souplesse pour s'adapter aux besoins d'applications différentes
- Manque de normalisation pour les SGBDO ;
  - trop de solutions propriétaires
- Peu d'informaticiens formés aux SGBDO (conséquence des autres raisons)

Objet -> relationnel => pas toujours possible

- Une partie des données peut être contenue dans une base de données préexistante
- Ces données sont déjà utilisées par d'autres applications et leur structure ne peut donc être modifiée
- Il n'est donc pas toujours possible de suivre une démarche « objet vers relationnel »

2 paradigmes différents

- L'objet et le relationnel sont 2 paradigmes bien différents
  - Faire la correspondance entre les données modélisées par un modèle objet et par un modèle relationnel n'est pas simple
  - Le modèle relationnel est moins riche que le modèle objet
  - pas d'héritage, ni de références, ni de collections dans le modèle relationnel



Rappel : organisation générale d'une application d'entreprise

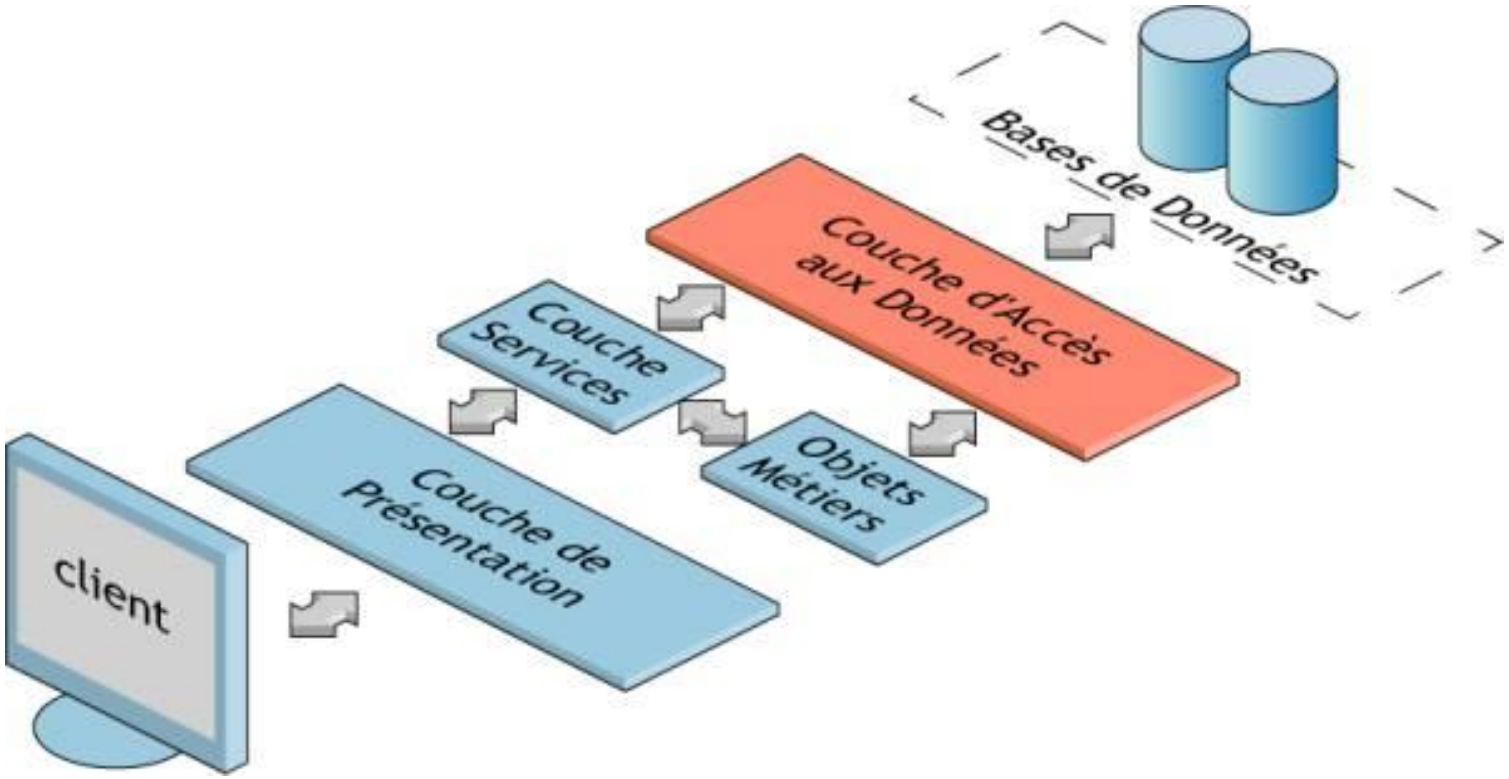
Client - Serveur

- Architecture aux capacités limitées dans le cadre des applications d'entreprise complexes :
  - client riche difficile à installer / déployer
  - dépendance forte vis-à-vis des outils de persistance et de la couche de présentation
  - enchevêtrement des services techniques (concurrence, persistance, transactionnel, sécurité,…) avec les processus métier

Architecture 3-tiers (ou n-tiers)

- Les nouvelles architectures préconisées ajoutent au moins une couche (tiers, niveau) entre le client et le serveur
- Dans cette couche tournent
  - les traitements « métiers » des applications
  - les processus techniques qui gèrent la sécurité, le transactionnel,…

Architecture 3-tiers (ou n-tiers)



Les différentes couches

Les différentes couches

- Présentation, interface homme-machine (IHM)
- Traitements relatifs au domaine de l'application (traitements « métiers ») avec les règles de gestion en vigueur
- Enregistrement, récupération et gestion des données persistantes dans une base de données

Les traitements « métier »

- Ce sont les « vrais » traitements : ils font le travail essentiel liés au domaine de l'application (comptabilité, facturation, calculs scientifiques,…)
- Ils nécessitent des traitements techniques, non fonctionnels, pour gérer la sécurité, le transactionnel, la concurrence,…
- Ces traitements techniques sont souvent réunis sous le nom de couche « services » et pris en charge par le *framework* de développement (container EJB par exemple)

La couche de persistance

- Elle est composée de la base de données
- Le plus souvent on y ajoute une couche qui effectue la correspondance (*mapping*) entre les objets et la base de données (DAO, ORM)
- En effet, le passage du monde objet au monde relationnel (ou plus généralement des objets « métier » aux données enregistrées) peut nécessiter des traitements complexes
- Souvent cette couche sert aussi de cache pour les objets récupérés dans la BD et améliore donc les performances

Relations entre les couches

- Aucune dépendance des 2 autres couches avec la couche présentation

- Couche présentation = modification régulière. Important que ces modifications soient sans impact pour les autres couches

Indépendance de la couche métier

- Pas de dépendance non plus entre la couche métier et l'implémentation de la couche de persistance
- Le plus souvent un processus métier reste stable durant toute la vie de l'application
- Modification que pour l'améliorer ou le corriger mais pas pour tenir compte d'une modification (le plus souvent technique) dans l'une des autres couches

Séparation entre les couches

- Pas de logique métier dans la couche présentation
  - éviter de la duplication de code
  - améliorer l'indépendance entre les couches
- => la couche de présentation doit
  - appeler des méthodes des objets métiers
  - mais que le contenu de ces méthodes doit être encapsuler dans les objets métiers

Séparation entre les couches

- Pas de logique métier dans la couche de persistance
  - Pas de dépendance entre la couche de persistance et la couche « métier »
  - Indépendance totale est évidemment impossible
- Une bonne indépendance
  - \+ éviter qu'une modification du modèle objet n'implique automatiquement une modification de la couche de persistance
    - (par l'utilisation de vues par exemple)

Au sommaire aujourd'hui

- Des Objets aux bases de données
- Mapping Objet - Relationnel

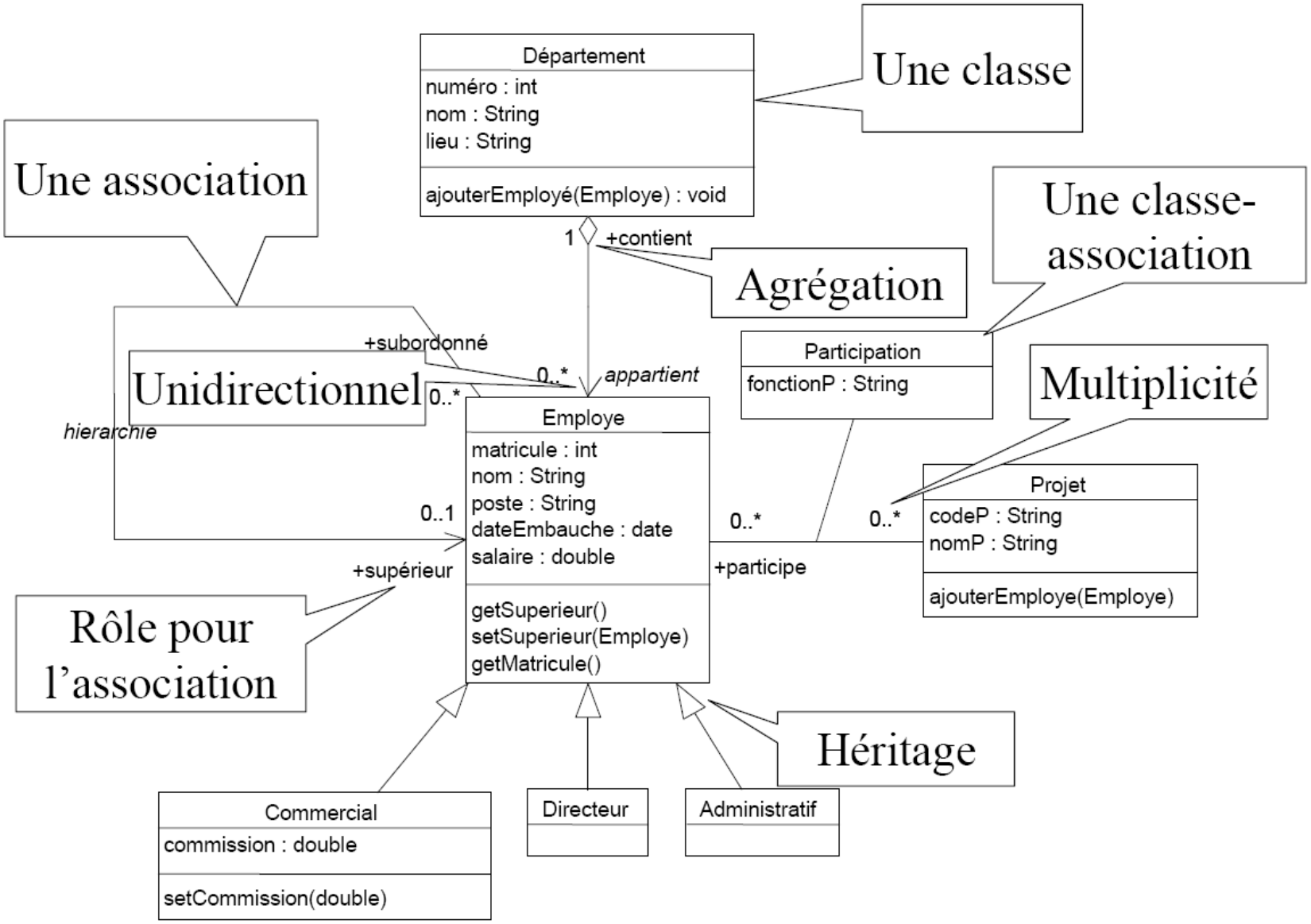
Introduction

- Ce cours présente :
  - Introduction problèmes de base qui se posent quand on veut faire correspondre
    - les données contenues dans un modèle objet
    - avec les données contenues dans une base de données relationnelle
- Les cours sur JPA (*Java Persistence API*) et Hibernate/TopLink donneront un exemple concret d'outil qui automatise en grande partie ce mapping

Contexte d'utilisation

- La base de données relationnelle manipulée par une application objet peut déjà exister ou non au moment de la conception de l'application objet
- Dans ce cours, on suppose plutôt que la base **n'existe pas**
- Dans la majorité des cas, les adaptations à une base existante ne sont pas très complexes et sont prises en charge par les outils de mapping

Rappel sur UML



2 paradigmes

- Les paradigmes objet et relationnel sont bien distincts
  - Le modèle objet est plus riche (héritage, associations en particulier ;)
  - => Pas toujours simple de faire correspondre des objets et des données de tables relationnelles

Problèmes du passage R ↔ Objet

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

Problème de base

- Les objets persistants doivent être enregistrés dans la base de données relationnelle
- Un objet a une structure complexe ≈ 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
- => Besoin d'« aplatir » ce graphe pour le ranger dans la base de données relationnelle

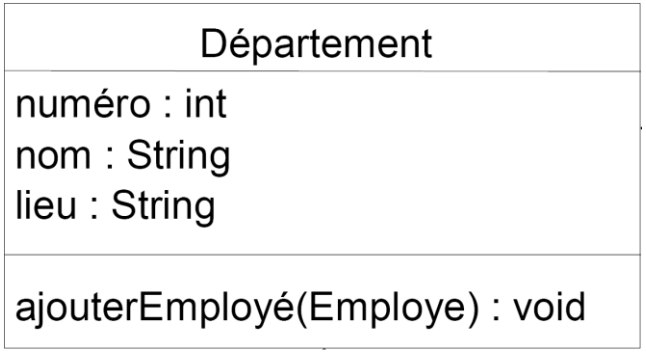
Plan

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

1. De la classe à la table

1. 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(<u>numéro</u>, nom, lieu)

1. En SQL

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

2. Identification des objets

2. Identification dans le monde 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

2. Identification dans le monde relationnel

- Seules les valeurs des colonnes peuvent servir à identifier une ligne
- Si 2 lignes ont les mêmes valeurs, il est impossible de les différencier
- Dans les schémas relationnels besoin d'une clé primaire pour toute table

2. Propriété « clé primaire » pour les objets

- Pour effectuer la correspondance objet-relationnel,
  - Besoin d'ajouter un identificateur à un objet

- Cet identificateur permet de conserver la clé primaire de la ligne de la base de données qui correspond à l'objet

2. Éviter les identificateurs 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

2. 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

2. Exemple de duplication

- Un objet **p1** de la classe **Produit** de code « 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
- Une erreur serait de créer en mémoire centrale un autre objet **p2** indépendant du premier objet **p1** déjà créé

2. Pour éviter des duplications

- En mémoire, un objet « cache » répertorie toutes 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

2. Objets « embarqués »

- 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 insérés (*embedded*) et ne nécessitent pas d'identificateur « clé primaire »

2. Exemple

- Une classe **Adresse** n'a pas de correspondance sous forme d'une table séparée dans le modèle relationnel
- Les attributs de la classe **Adresse** sont insérées dans la table qui représente la classe **Client**
- Les objets de la classe **Adresse** n'ont pas d'identification liée à la base de données

3. Traduction des associations

3. Exemple

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

```
class Département {
  private Collection<Employé> employes;
  …
}

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

3. 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)

3. Exemples de tables

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

)
```

3. Navigabilité des associations

- Dans un modèle objet, une association peut être bi ou unidirectionnelle
- Exemple d'association unidirectionnelle :
  - 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

3. Navigabilité des associations

- 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
```

3. Association binaire M:N - Objet

- 2 façons différentes pour représenter une association M:N :
  - une collection ou une *map* dans chacune (ou dans une, selon la directionnalité) 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

3. Exemples de classes – solution 1

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

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

3. Solution 1 – variante avec une Map

```
class Projet {
  private Map<Integer, Employé>
          employés;
  …
}
```

- La clé peut, par exemple, représenter le matricule d'un employé
- La variante avec *map* permet un accès rapide à un des employés participant à un projet
- La classe *Employé* reste identique à la première variante

3. Exemples de classes – solution 2

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

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

3. Association M:N - Relationnel

- Une association M:N est traduite par une table association

- La clé primaire de cette table est formée des 2 clés des tables qui représentent les classes qui interviennent dans l'association

3. Exemple de traduction

- 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, codeP))
```

Pour faciliter la lecture, la syntaxe est simplifiée pour les contraintes

3. Classe association (cas M:N)

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

3. En objet

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

3. 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)
)
```

3. Association de degré > 2 - Objet

- Une association de degré > 2 peut être représentée de plusieurs façons différentes :

1. par une classe « association » qui contient des références vers les objets qui participent à l'association

2. par des collections de classes qui contiennent des collections ou des références vers des classes qui contiennent des collections

3. Exemple d'association de degré > 2

- Une réservation dans une compagnie aérienne peut être considérée comme une association entre
  - les avions,
  - les passagers
  - et les numéros de siège

un passager peut occuper plusieurs sièges (problème de santé par exemple)

3. Représentation par une classe

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

3. Autre représentation

```
class Vol {
    private
    Collection<Reservation> résas;
    …
}

class Reservation {
    private Passager passager;
    private int siège;
    …
}
```

3. Association de degré > 2 en relationnel

- 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

3. Exemple de traduction

- Exemple de traduction d'une association de degré > 2
  - RESERVATION(***nVol***, ***nSiège***, ***codePassager***, …)
  - En SQL :

```
create table reservation(
  vol varchar(10) references vol,
  siege integer,
  codePassager varchar(10)
  references passager,
  primary key(nVol, nSiege,
  codePassager),
  ...
)
```

3. Gestion des associations bidirectionnelles

- Gérer une association est plus complexe dans le monde objet que dans le monde relationnel, surtout si elle est bidirectionnelle

- 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

3. Gestion automatique de l'extrémité d'une association

- Certains framework (EJB 2 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 ou EJB 3/JPA préfèrent ne rien automatiser

4. Objet dépendant

4. Objet dépendant

- Un objet dont le cycle de vie dépend du cycle de vie d'un autre objet auquel il est associé, appelé objet propriétaire

- Aucun autre objet que le propriétaire ne doit avoir de référence directe vers un objet dépendant

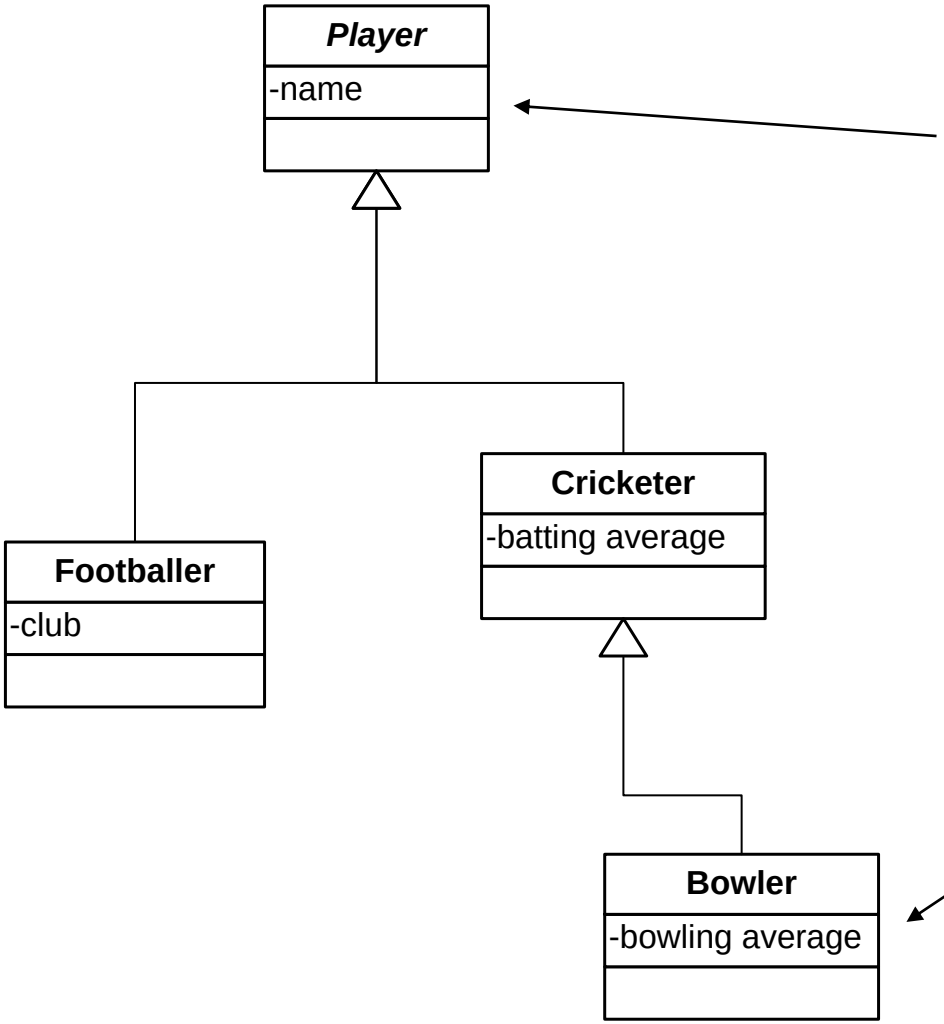
- En ce cas, la suppression de l'objet propriétaire doit déclencher la suppression des objets qui en dépendent (déclenchement « en cascade »)

4. Exemple

- Une ligne de facture ne peut exister qu'associée à une (en-tête de) facture
- Si on supprime une facture => toutes les lignes de la facture doivent être supprimées
- Tous les outils de mapping, comme JPA ou Hibernate, offrent la possibilité d'automatiser les suppressions en cascade d'objets dépendants

5. Traduction de l'héritage

5. Exemple



Sportifs qui peuvent être des footballeur ou des joueurs de crickets

Parmi les joueurs de crickets, les joueurs qui lancent la balle : les *bowlers*

5. 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

5. Accessibilité des variables

- 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

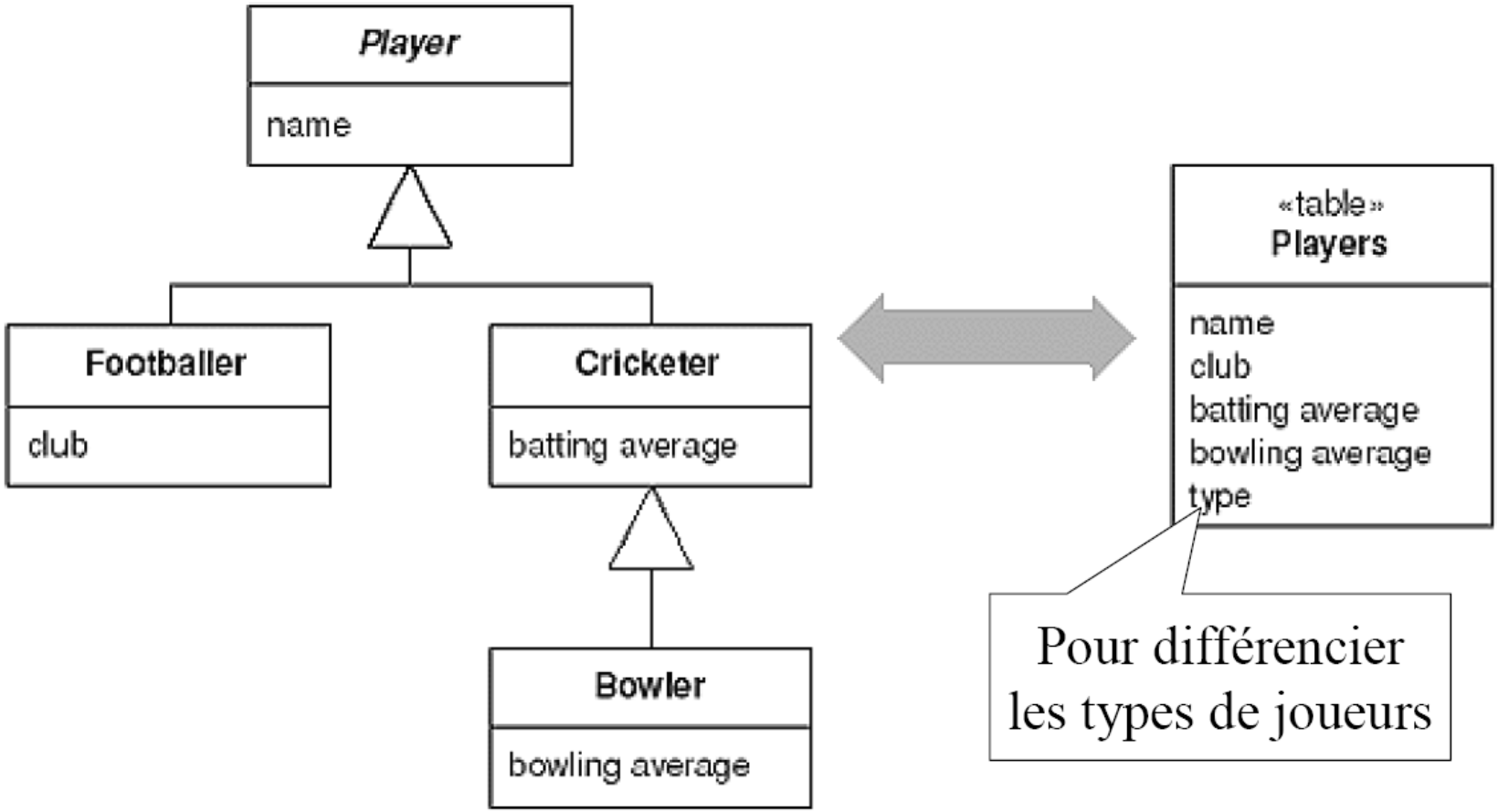
5. Accessibilité des variables

- 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
  - Un autre moyen, utilisé par les outils de *mapping*, est la réflexivité Java qui permet d'outrepasser les restrictions d'accès accolées aux variables d'instance

5. Polymorphisme

- Association polymorphe : 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

5. Toutes les classes traduites par une seule table



5. Avantages

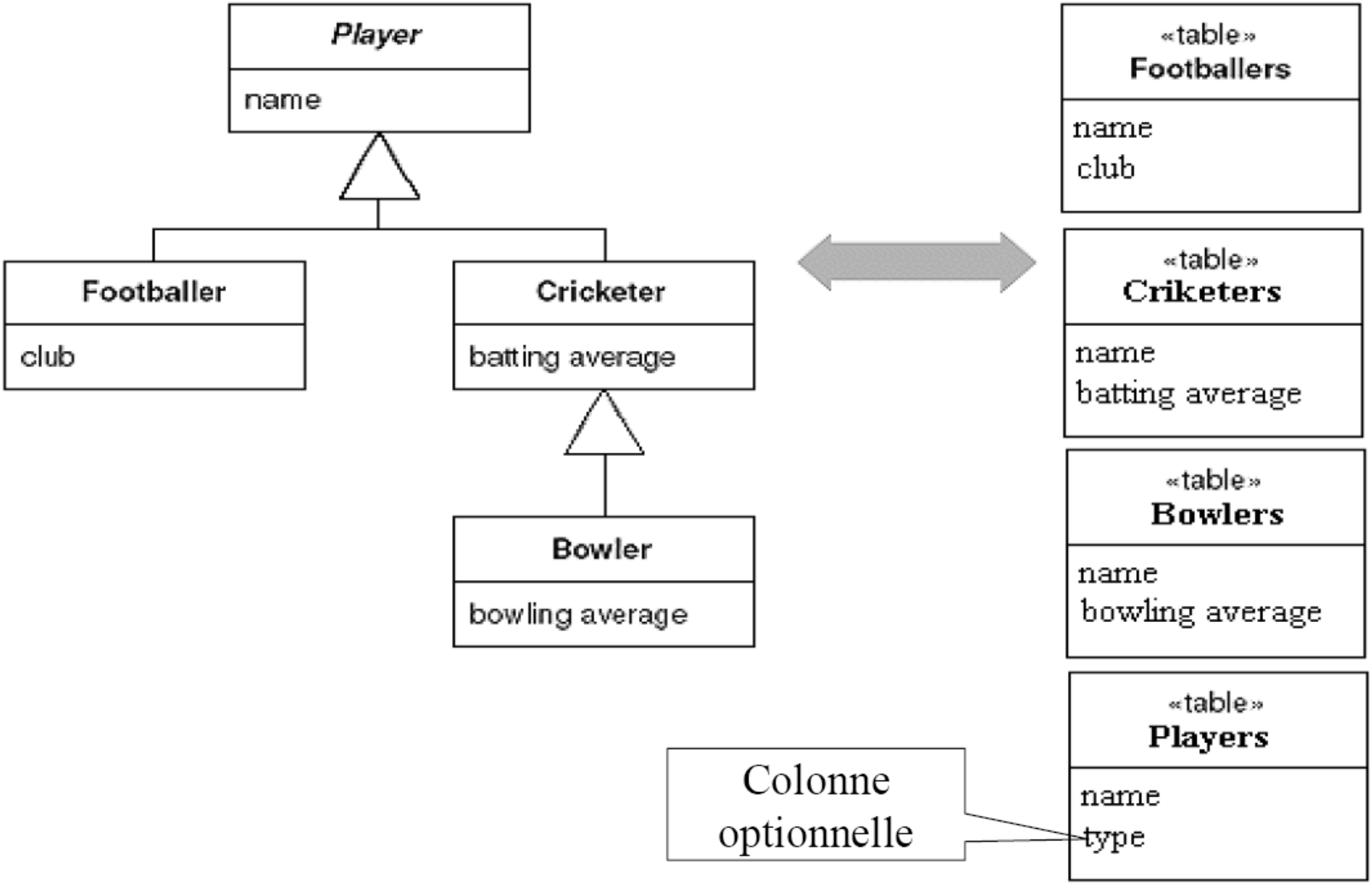
- Souvent la solution la plus simple à mettre en place
- C’est d’ailleurs la solution la plus fréquemment choisie
- Permet les requêtes et associations polymorphes

5. 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

5. Autre solution : une table par classe



5. Préservation de l'identité

- 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

5. Préservation de l'identité

- Pour récupérer les informations sur une instance d'une classe fille, il suffit de faire une jointure sur ces clés primaires

5. Avantages

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

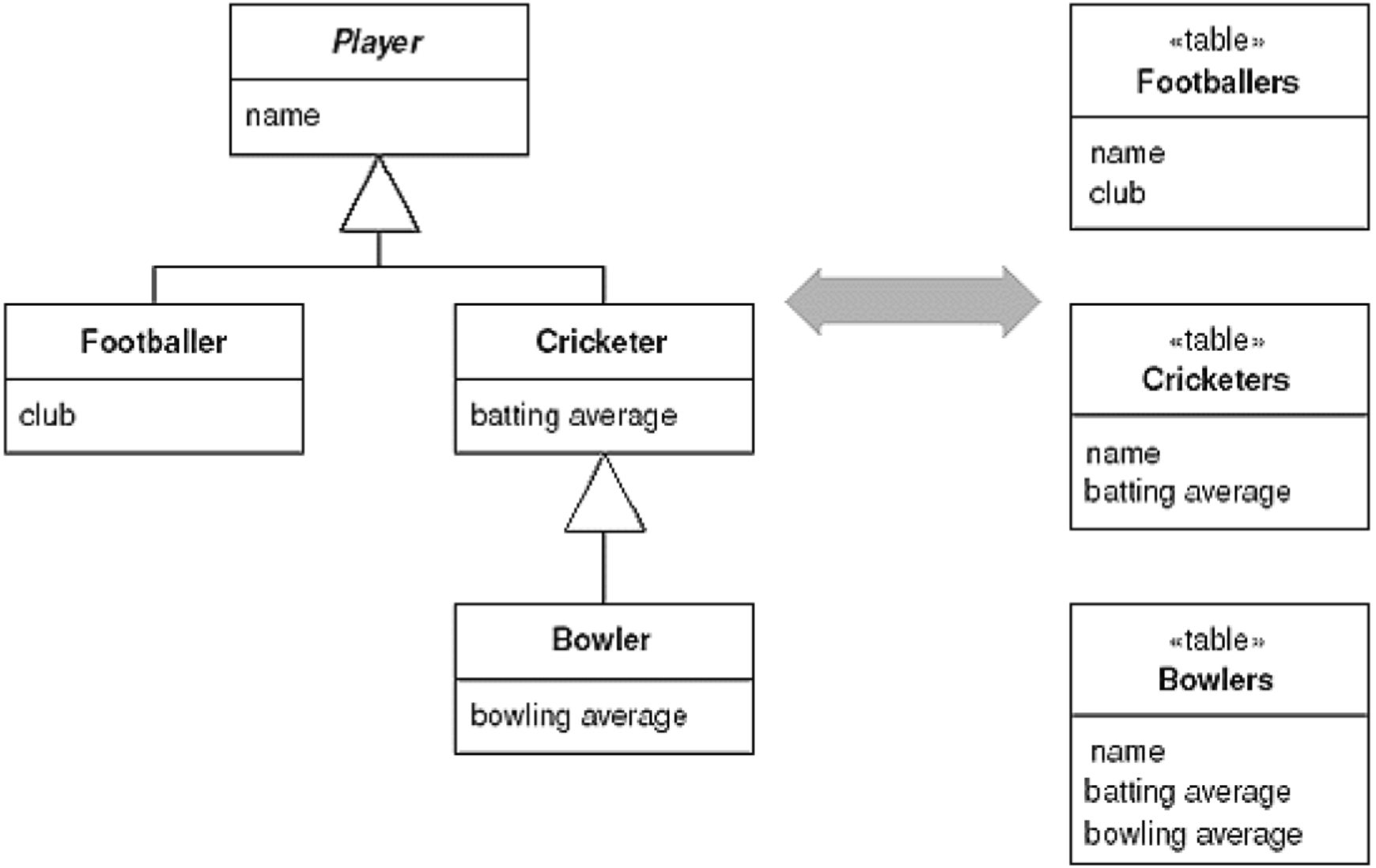
5. Inconvénient

- Si la hiérarchie d'héritage est complexe nécessite de nombreuses jointures pour reconstituer les informations éparpillées dans de nombreuses tables

- D'où des instructions complexes, mais surtout de mauvaises performances

- La colonne « type » de la table *Players* permet de limiter un peu ce problème ; il est possible, par exemple, de retrouver les noms des footballers sans faire de jointure

5. Une table par classe concrète



5. 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

5. Avantages

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

5. Inconvénients

- 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)

5. 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)

5. 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

5. 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

5. Une solution à éviter

- Sauf pour des cas bien précis où le polymorphisme n'est pas important, il vaut mieux éviter cette solution

- D'ailleurs, la version actuelle de la spécification EJB 3 n'impose pas aux serveurs d'application d'offrir cette possibilité de traduction de l'héritage

5. Variantes

- Dans une arborescence d'héritage ces stratégies peuvent être mélangées
- On peut, par exemple, créer plusieurs tables pour plusieurs branches de l'arborescence d'héritage
- Mais le risque est de retomber sur les problèmes de la stratégie 3 (une table par classe concrète) avec les requêtes et les associations polymorphes

5. Héritage multiple

- Dans les cas où l'héritage est traduit par des tables séparées, l'identifiant dans les classes descendantes est l'ensemble des identifiants des classes mères

6. Navigation entre les objets

6. 2 stratégies pour les associations

- Lorsqu'un objet est créé à partir des données récupérées dans la base de données, 2 stratégies vis-à-vis des objets associés à cet objet :
  - 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

6. Une situation

- Recherche dans la base de données d'une facture qui vérifie un certain critère
- Un objet de la classe **Facture** qui correspond à cette facture est créé
- Est-ce que les objets **LigneFacture** associés à cette facture doivent aussi être créés ?
- 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, …

6. Le problème

- Le risque est de créer un très grand nombre d'objets, peut-être inutiles
- Par exemple, si on veut récupérer cette facture pour connaître seulement la date de facturation, on voit que tous ces objets associés sont totalement inutiles
- D'où de mauvaises performances, sans raisons valables

6. Chargement « paresseux »

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

6. Chargement « paresseux »

- Quand un objet est créé à partir des données enregistrées dans la BD, les objets associés ne sont pas immédiatement créés
- L'objet créé contient juste l'information nécessaire pour retrouver ces objets associés (clé primaire)
- Ces objets ne seront créés que si c'est vraiment indispensable

6. 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

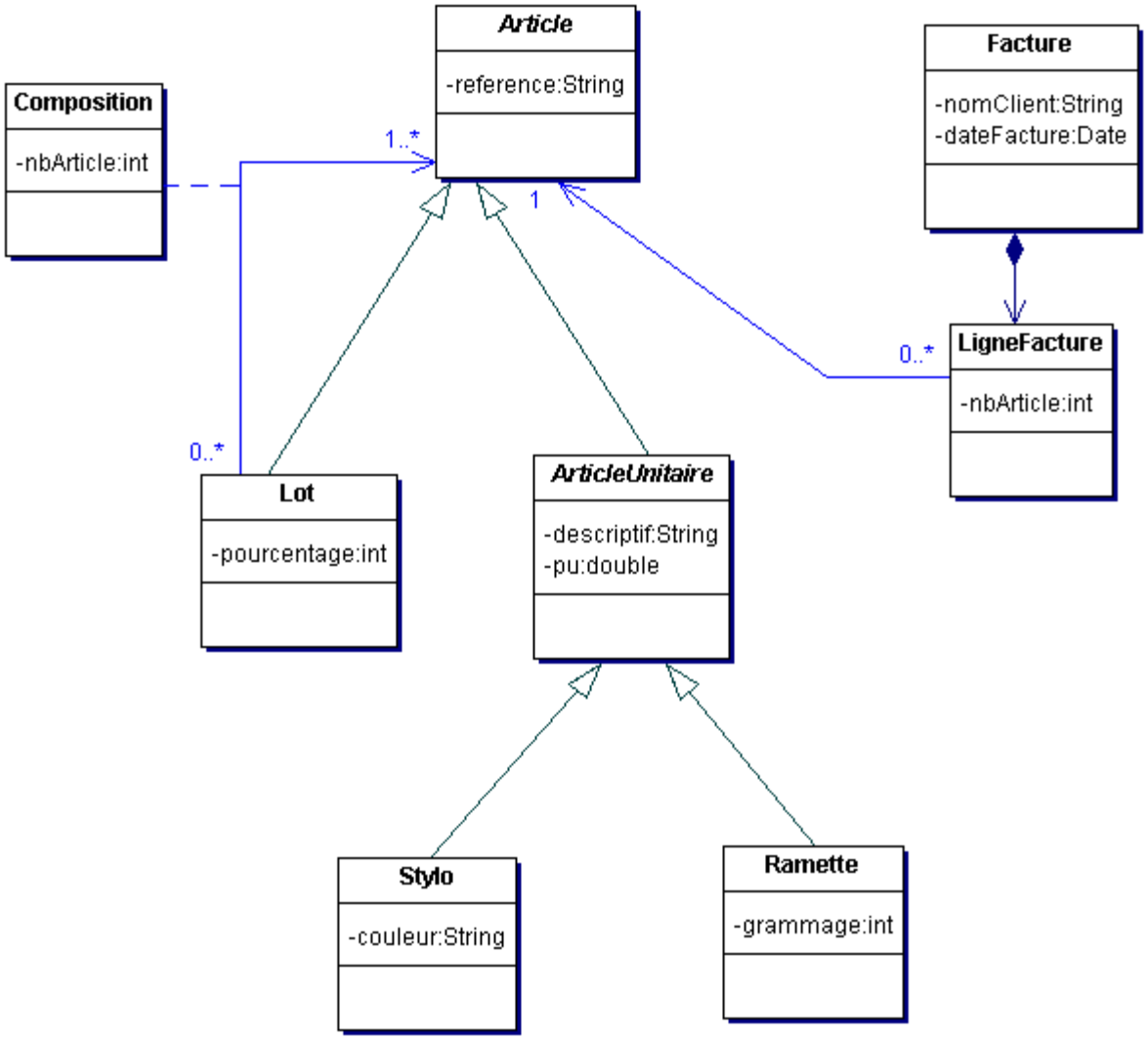
- Si le programme contient le code **f.getLigne(i).getQuantite()** l'objet **LigneFacture** correspondant à la ième ligne de facture est créé et le message **getQuantite()** lui est envoyé

Conclusion pour aujourd'hui

- La correspondance entre le modèle objet et le modèle relationnel n'est pas une tâche facile

- Les outils et *framework* ORM (étudiés à la fin de ce cours) réduisent grandement le code qui permet cette correspondance et facilitent donc les adaptations mutuelles des 2 modèles

- Questions ?
- Exercice



Problème des « N + 1 selects »

- On peut tomber sur le problème des « N + 1 selects »
- Exemple :
  - On veut récupérer la facture qui a le plus gros total, parmi 5000 factures
  - Si la récupération des lignes se fait à la demande, les objets « Facture » sont d'abord récupérées (1 select), puis, pour chacune des 5000 factures, les lignes de facture associées sont récupérées 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

Solutions possibles (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

- Il faut cependant retenir que, dans certaines circonstances, une bonne requête SQL vaut mieux que la création de nombreux objets

Solutions possibles (2)

- La plupart des outils de mapping permettent de spécifier une stratégie spéciale pour une requête particulière

- On peut alors indiquer que, par défaut, le mode de récupération ne sera pas immédiate, tout en spécifiant que la récupération des objets associés devra être immédiate pour une requête particulière