

## Quelques patterns pour la persistance des objets avec DAO

Université de Nice Sophia-Antipolis  
Version 1.4 – 30/8/07  
Richard Grin

- ❑ Ce cours présente des modèles de conception utilisés pour effectuer la persistance des objets

## Principe de base

- ❑ Il est plus fréquent de changer la façon d'effectuer la persistance que de changer le modèle « métier »
- ❑ Pour faciliter les changements dans la persistance il faut isoler le plus possible le code qui gère la persistance

## DAO

- ❑ La persistance est isolée dans des objets spécifiques, les DAO (*Data Access Objects*)

## Le modèle de conception DTO (*Data Transfer Object*)

## Utilité des DTOs

- ❑ Les DAOs sont situés dans une couche proche de la base de données
- ❑ Le code utilisateur des DAOs est souvent situé sur une autre couche distante
- ❑ Les DTOs peuvent être utilisés pour transporter les données entre les différentes couches distantes

## Un fait important

- ❑ Les appels de méthode distants sont beaucoup plus coûteux que les appels locaux
- ❑ Le coût dépend peu de la quantité de données transférée à chaque appel

## Le problème à résoudre

- ❑ Un client souhaite récupérer des données en interrogeant des objets distants **non facilement transportables sur le réseau**
- ❑ Exemple : récupérer le nom, prénom, salaire et lieu de travail d'un employé
- ❑ S'il utilise les accesseurs des classes des objets (`getNom`, `getPrenom`, `getSalaire`, `getLieu`), plusieurs appels distants sont nécessaires

## La solution DTO

- ❑ Le client demande un objet qui contient toutes les valeurs dont il a besoin
- ❑ Cet objet, un *Data Transfer Object* (DTO), est construit sur le site distant et passé en une seule fois au client
- ❑ Un DTO contient l'état d'un ou de plusieurs objets métier, mais pas leur comportement
- ❑ Synonyme : *Transfer Object* (TO)

## Exemples d'utilisation des DTO

- ❑ Transporter les données d'un objet distant pas transportable sur le réseau (pas sérialisable)
- ❑ Transporter plusieurs objets distants en un seul appel distant ; par exemple une facture avec toutes les lignes de facture et les informations sur les produits
- ❑ Pour éviter les complications inutiles il faut éviter les DTOs si l'application est locale (pas distribuée)

## DTO pour modifier

- ❑ Un DTO peut aussi être utilisé, plus généralement pour modifier un ou plusieurs objets distants (ou les données de la base de données) :
  - le DTO est créé ou modifié sur une couche de l'application
  - il est passé à une couche distante qui utilise ses données pour modifier un ou plusieurs objets distants (ou la base de données)

## Le modèle de conception DAO (*Data Access Object*)

## Le problème à résoudre

- ❑ Le code pour la persistance varie beaucoup
  - avec le type de stockage (BD relationnelles, BD objet, fichiers simples, etc.)
  - avec les implémentations des fournisseurs de SGBD
- ❑ Si les ordres de persistance sont imbriqués avec le code « métier », il est difficile de changer de source de données

## La solution

- ❑ Encapsuler le code lié à la persistance des données dans des objets DAO dont l'interface est indépendante du support de la persistance
- ❑ Le reste de l'application utilise les DAOs pour gérer la persistance, en utilisant des interfaces abstraites, indépendantes du support de persistance ; par exemple,  
`Employe getEmploye(int matricule)`

## DAO

- ❑ Quand l'application a besoin d'effectuer une opération liée à la persistance d'un objet, elle fait appel à un objet DAO à qui elle passe les informations nécessaires pour effectuer l'opération
- ❑ Chaque classe d'objet métier a son propre type de DAO (`DAOEmploye`, `DAODepartement`, ...)
- ❑ Mais le même objet DAO peut être utilisé pour tous les objets d'une classe d'objet métier

## Utilité des DAOs

- ❑ Plus facile de modifier le code qui gère la persistance (changement de SGBD ou même de modèle de données)
- ❑ Factorise le code d'accès à la base de données
- ❑ Plus facile pour le spécialiste des BD d'optimiser les accès (ils n'ont pas à parcourir toute l'application pour examiner les ordres SQL)
- ❑ Sans doute le modèle de conception le plus utilisé dans le monde de la persistance

## Emplacement des DAOs

- ❑ Les DAOs sont placés dans la couche dite « d'accès aux données » qui est souvent sur une autre machine que la couche des objets métiers
- ❑ Les échanges de messages entre les DAOs et les objets métiers engendrent donc souvent des appels distants et des DTO peuvent donc être utilisés pour améliorer la vitesse des échanges

## CRUD

- ❑ Cet acronyme désigne les opérations de base de la persistance : *create*, *retrieve*, *update* et *delete*
- ❑ Ces 4 opérations de base sont implémentées dans un DAO

## CRUD

- ❑ Create pour créer une nouvelle entité dans la base
- ❑ Retrieve pour retrouver une ou plusieurs entités de la base
- ❑ Update pour modifier une entités de la base
- ❑ Delete pour supprimer une entité de la base
- ❑ Plusieurs variantes pour les signatures de ces méthodes dans les DAOs

R. Grin

Mapping objet-relationnel

page 19

## create - paramètres

- ❑ Prend en paramètre l'état de la nouvelle entité
- ❑ Cet état peut être donné
  - par une série de paramètres des types des données : `create(int id, String nom,...)`
  - par un DTO : `create(DTOxxx dto)`
  - par l'objet métier que l'on veut rendre persistant : `create(Article article)`

R. Grin

Mapping objet-relationnel

page 20

## create – type retour

- ❑ Le type retour peut être
  - `void` (la variante la plus utilisée)
  - `boolean` pour indiquer si la création a pu avoir lieu (on peut utiliser une exception à la place)
  - l'identificateur de l'entité ajoutée (utile si la clé primaire est générée automatiquement dans la base de données)
  - un objet métier ou un DTO correspondant à l'entité ajoutée

R. Grin

Mapping objet-relationnel

page 21

## create – comparaison des variantes

- ❑ Les variantes qui passent les différentes valeurs individuellement sont les plus souples
- ❑ Lorsque les DTOs sont utilisées par ailleurs, les variantes avec DTO sont souvent rencontrées
- ❑ Les variantes avec objet métier ne conviennent que si l'objet métier a toutes ses propriétés publiques et s'il est facilement transportable
- ❑ Mais elle peut être pratique et performante dans les cas où elle sont applicables

R. Grin

Mapping objet-relationnel

page 22

## Exemple de code JDBC pour create

```
private String insert = variable d'instance
"insert into STYLO (ref, nom, prix, couleur)
 values(?, ?, ?, ?)";

...
PreparedStatement ps = méthode create
    connexion.prepareStatement(insert);
ps.setString(1, reference);
ps.setString(2, nom);
ps.setString(3, couleur);
ps.setBigDecimal(4, prix);
ps.executeUpdate();
```

R. Grin

Mapping objet-relationnel

page 23

## retrieve

- ❑ 3 types de méthode, suivant qu'elle retourne
  - un seul objet
  - une collection d'objets
  - une valeur calculée à partir de plusieurs entités (agrégation)
- ❑ Une méthode (ou un objet) qui retourne des données de la base de données est souvent appelée *finder*

R. Grin

Mapping objet-relationnel

page 24

## Exemples de *finders*

- On trouvera le plus souvent
  - une méthode `findById(id)` (ou d'un nom semblable...) qui retrouve une entité en donnant son identificateur dans la base
  - une méthode `findAll()` qui retrouve toutes les entités du type géré par le DAO
- Mais on trouvera aussi des *finders* qui cherchent suivant des critères quelconques ou suivant des critères bien précis (ces *finders* dépendent des traitements métier)

R. Grin

Mapping objet-relationnel

page 25

## *Finder* qui retourne un objet

- On lui passe en paramètre un identificateur de l'entité cherchée
- Il retourne un objet métier qui correspond à l'entité cherchée, ou un DTO qui contient les données de l'entité cherchée
- Si le *finder* retourne un objet unique, il retourne `null` si rien n'a été trouvé

R. Grin

Mapping objet-relationnel

page 26

## *Finder* qui retourne une collection

- On lui passe en paramètre le critère de sélection, sous une forme quelconque
  - objet ou valeurs « critère de sélection »
  - objet « exemple » (à la « *query by example* »)
- Le type retour peut être très divers :
  - `ResultSet`
  - `RowSet`
  - Collection (`Collection`, `List`, `Set`,...)  
d'objets métier ou de DTOs
  - tableau (rare)

R. Grin

Mapping objet-relationnel

page 27

## Résultat vide

- Si le critère de la requête n'est vérifiée par aucune valeur, le *finder* doit retourner une « collection » (collection, resultset rowset ou tableau) vide
- Retourner la valeur `null` obligerait à un cas particulier pour le traitement de la valeur retournée (« `if (result == null)` » avant une boucle qui parcourt le résultat)

R. Grin

Mapping objet-relationnel

page 28

## *Finder* qui retourne une valeur calculée

- Les valeurs calculées à partir des données de plusieurs entités (exemple : total des salaires) peuvent s'obtenir à partir d'objets chargés en mémoire
- Mais il peut être préférable de ne pas créer les objets et d'interroger directement la base de données qui est optimisée pour ce type de requête
- Un DAO peut ainsi comporter une méthode qui renvoie le total des salaires des employés

R. Grin

Mapping objet-relationnel

page 29

## update

- Des variantes diverses pour les paramètres :
  - identificateur + valeurs (plusieurs paramètres pour les valeurs ou un seul DTO)
  - l'objet métier dont on veut sauvegarder les modifications (nécessite un accès public aux valeurs qui seront modifiées)
- Le type retour peut être
  - `void`
  - `boolean` pour indiquer si la modification a pu avoir lieu

R. Grin

Mapping objet-relationnel

page 30

## delete

- Variantes pour les paramètres :
  - identificateur de l'entité à supprimer dans la base
  - l'objet métier (ou un DTO) correspondant à l'entité à supprimer dans la base
- Variantes pour le type retour :
  - `void`
  - `boolean` pour indiquer si la suppression a pu avoir lieu

R. Grin

Mapping objet-relationnel

page 31

## Autres méthodes des DAOs

- Outre les opérations CRUD, les DAO peuvent aussi implémenter des méthodes spécifiques au modèle métier de l'application
- Le plus souvent ce sont des variantes de l'opération « *retrieve* »
- Par exemple une méthode qui renvoie les candidats qui ont une mention à un examen

R. Grin

Mapping objet-relationnel

page 32

## 2 stratégies d'utilisation des DAOs

1. Chaque objet métier a une référence à son DAO et l'utilise pour sa propre persistance. Le programme qui manipule les objets métier ne connaît pas les DAOs
2. Le programme qui manipule les objets métier utilise directement les DAOs. Les objets métier n'ont pas de référence à un DAO (stratégie sans doute la plus fréquemment utilisée)

R. Grin

Mapping objet-relationnel

page 33

## Stratégie 1

- Les programmes qui manipulent les objets métier ne sont pas modifiés par rapport à un programme qui n'utilise pas de DAO
- Seuls les objets métier connaissent leur DAO
- Les objets métier doivent avoir une référence vers le DAO qu'ils utilisent
- Cette référence peut être obtenue par une méthode `static` de la classe DAO (ce qui peut permettre de partager un DAO entre tous les objets métier d'une même classe)

R. Grin

Mapping objet-relationnel

page 34

## Exemple de code

```
class Stylo {
    private StyloDAO dao;
    ...
    public void sauvegardeToi() {
        dao = getDAO();
        dao.insertOrUpdate(this);
    }
    private StyloDAO getDAO() {
        if (dao == null)
            StyloDAO.getDAO();
        return dao;
    }
}
```

on peut aussi construire un DTO pour le passer au DAO

Pour simplifier, on ne tient pas compte des exceptions

R. Grin

Mapping objet-relationnel

page 35

## Stratégie 2

- On rencontre le plus souvent la stratégie 2
- On perd sans doute de la pureté de la programmation objet

R. Grin

Mapping objet-relationnel

page 36

## Exemple de code

```
// ou styloDAO = new StyloDAO()
StyloDAO styloDAO = StyloDAO.getDAO();
int idStylo =
    styloDAO.create("Marker", "noir ",
        . . .
        styloDAO.findById(idStylo);
styloDAO.update(idStylo, . . .);
List<Stylo> l = styloDAO.findAll();
```

DTO ou objet métier

Nouvelles valeurs pour le stylo

R. Grin

Mapping objet-relationnel

page 37

## Exemple de code (variante)

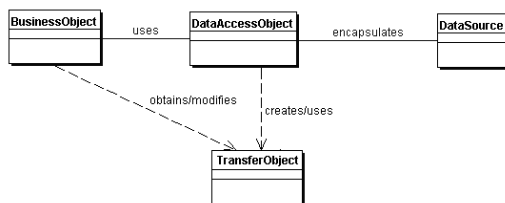
```
// ou styloDAO = new StyloDAO()
StyloDAO styloDAO = StyloDAO.getDAO();
styloDAO.create(145, "Marker",
    "noir ", 120, . . .);
. . .
Stylo stylo = styloDAO.findById(1234);
stylo.setPrix(45);
styloDAO.update(stylo);
List<Stylo> l = styloDAO.findAll();
```

R. Grin

Mapping objet-relationnel

page 38

## Diagramme de classes (avec utilisation de TO)



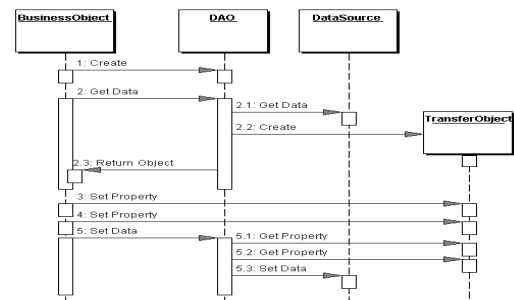
Cette image (et les suivantes) sont extraites du « Core J2EE Pattern Catalog » de Sun

R. Grin

Mapping objet-relationnel

page 39

## Diagramme de séquences



Modification de plusieurs attributs persistants en utilisant un DTO : création du DAO, puis récupération des valeurs actuelles, puis modification de ces valeurs

## Problèmes avancés sur les DAO

R. Grin

Mapping objet-relationnel

page 41

## Problèmes abordés

- DAO et exceptions
- DAO et connexions
- DAO et transactions
- DAO et objets composés
- DAO et héritage

R. Grin

Mapping objet-relationnel

page 42

## DAO et exceptions (1)

- ❑ Les méthodes des DAO peuvent lancer des exceptions puisqu'elles effectuent des opérations d'entrées-sorties
- ❑ Les exceptions ne doivent pas être liées à un type de DAO particulier si on veut pouvoir changer facilement de type de DAO

## DAO et exceptions (2)

- ❑ Pour cela, on crée une ou plusieurs classes d'exception indépendantes du support de persistance, désignons-les par **DAException** (ou **DataAccessException** ou **DaoException**)
- ❑ Les méthodes des DAO attrapent les exceptions particulières, par exemple les **SQLException**, et relancent des **DAException** (auxquels sont chaînées les exceptions d'origine pour faciliter la mise au point)

## DAO et connexions (1)

- ❑ Une connexion peut être ouverte au début des méthodes du DAO, et fermée à la fin des méthodes
- ❑ Cette stratégie va coûter cher si un pool de connexions n'est pas utilisé

## DAO et connexions (2)

- ❑ Il est préférable que les connexions soient ouvertes par les clients du DAO
- ❑ En ce cas, les connexions ouvertes doivent être passées au DAO
- ❑ Pour cela le DAO peut comporter une méthode **setConnection(Connection c)** (la façon de faire dépend de l'API de persistance que l'on utilise ; avec JPA on passera le manager d'entité et avec Hibernate la session)

## Qui gère les transactions ?

- ❑ Un DAO pourrait démarrer et terminer lui-même les transactions à chaque méthode
- ❑ Cependant il n'est pas rare de vouloir inclure un ou plusieurs appels de méthodes de DAOs dans une seule transaction
- ❑ L'implémentation des DAOs doit donc permettre cette dernière possibilité : ce sont les clients du DAO qui vont gérer les transactions

## Transactions gérées par les clients

- ❑ C'est le client du DAO, et pas le DAO qui va indiquer quand une transaction doit être validée ou invalidée
- ❑ Le DAO utilise la transaction en cours si elle existe

## Exemple schématique JDBC - DAO

```
public class StyloDao {
    private Connection conn;
    public void setConnection(Connection c) {
        this.conn = c;
    }
    public long create(...) {
        PreparedStatement pstmt =
            conn.prepareStatement(...);
        pstmt.setString(...);
        ...
        pstmt.executeUpdate();
    }
}
```

R. Grin

Mapping objet-relationalnel

page 49

## Exemple schématique JDBC - client

```
Connection conn = ... ;
daoStylo.setConnection(conn);
daoRamette.setConnection(conn);
...
daoStylo.create(...);
daoFacture.update(...);
conn.commit();
```

R. Grin

Mapping objet-relationalnel

page 50

## Tout n'est pas parfait !

- ❑ On vient de voir qu'avec un DAO JDBC, il faut passer une connexion ; avec un DAO JPA (étudié dans une autre partie du cours) il faut passer un gestionnaire d'entités
- ❑ Il est donc difficile de rendre l'utilisation des DAOs totalement indépendante du type de persistance si on veut gérer des types de persistance très différents
- ❑ Malgré tout, l'utilisation des DAOs diminue fortement la dépendance vis-à-vis des types de persistance

R. Grin

Mapping objet-relationalnel

page 51

## Solution partielle

- ❑ Par exemple, pour la gestion des transactions, le code différent concernera l'initialisation des DAOs (avec une connexion ou avec un autre objet)
- ❑ La solution est de ne pas mettre la méthode `setConnection` dans l'interface du DAO et de *caster* le DAO dans un type concret, le temps de l'initialiser

R. Grin

Mapping objet-relationalnel

page 52

## DAO et objets composés

- ❑ Par exemple, pour une facture, la question doit être posée : le dao pour les factures doit-il retourner les lignes de la facture ?
- ❑ Il n'y a pas de réponse générale ; la réponse dépend du contexte

R. Grin

Mapping objet-relationalnel

page 53

## Le modèle de conception « fabrique abstraite »

R. Grin

Mapping objet-relationalnel

page 54

## Un cas d'utilisation

- ❑ Une application fonctionne sur un ordinateur portable
- ❑ Une application peut utiliser une base locale MySQL si l'ordinateur n'est pas connecté à Internet, ou une base Oracle distante s'il est connecté
- ❑ Comment utiliser le bon DAO pour chaque classe métier (par exemple, utiliser un **DAOSTyloMySQL** ou un **DAOSTyloOracle** suivant le cas) ?

R. Grin

Mapping objet-relationnel

page 55

## Avec un constructeur

- ❑ Le code

```
DaoStylo dao = new DaoStyloOracle();
```

fixe le type de DAO créé

- ❑ Il devra être modifié si on veut un autre type

R. Grin

Mapping objet-relationnel

page 56

## Pattern « fabrique » pour récupérer les DAO

- ❑ Le pattern « fabrique » (*factory*) permet de créer des instances en cachant le type concret des instances créées

R. Grin

Mapping objet-relationnel

page 57

## Avec une fabrique

- ❑ `DaoStylo dao = fabriqueDaoStylo.getDao();`
- ❑ `dao` sera du type **DaoStyloOracle** ou **DaoStyloMySQL** selon le cas
- ❑ **DaoStylo** est une interface implémentée par les classes concrètes de DAO **DaoStyloOracle** et **DaoStyloMySQL**
- ❑ Signature de `getDao()` :  
`DaoStylo getDao()`

R. Grin

Mapping objet-relationnel

page 58

## Avec une fabrique

- ❑ Pour fixer le type renvoyé, il suffit de l'indiquer auparavant à la fabrique par le code `fabriqueDaoStylo.setTypeDao(typeDao);`
- ❑ `typeDao` peut, par exemple, être déterminé en testant si l'ordinateur est connecté ou non à Internet

R. Grin

Mapping objet-relationnel

page 59

## Fabrique abstraite

- ❑ Si on veut changer de base de données, le type de DAO doit être fixé pour toutes les fabriques de DAOs
- ❑ Le pattern « fabrique abstraite » permet de changer plus facilement de base de données
- ❑ En une seule ligne de code tous les DAOs peuvent être remplacés par des DAOs adaptés à la nouvelle base de données choisie

R. Grin

Mapping objet-relationnel

page 60

## Fabrique abstraite

- Une fabrique abstraite est un type abstrait qui permet de cacher les types réels d'un ensemble de fabriques concrètes
- Chaque fabrique concrète fournit tous les DAOs (DAOStylo, DAORamette,...) associés à une certaine source de données
- Dans la ligne de code, on récupère la bonne fabrique de DAOs, associée à la bonne source de données

R. Grin

Mapping objet-relationnel

page 61

## Code pour fabrique abstraite

- Le code qui suit est un exemple schématique de l'utilisation du pattern DAO, avec le pattern fabrique abstraite (inspiré fortement d'un exemple donné par Sun)
- Il utilise le pattern DTO/TO exposé dans la 1<sup>ère</sup> partie de ce cours
- Ce pattern DTO peut être évité si les objets persistants peuvent être transportés entre les différentes couches d'une application (par exemple, pour une application locale, ou en utilisant les objets « détachés » de JPA ou de Hibernate)

R. Grin

Mapping objet-relationnel

page 62

Le type abstrait

## Code client (début)

```

DAOFactory daoFactory =
    DAOFactory.getDAOFactory(
        DAOFactory.TypeDao.MYSQL);
styloDAO = daoFactory.getStyloDAO();
// crée un nouveau stylo dans la base
int styloNo = styloDAO.create(...);
// Trouve un stylo
StyloTO styloTO = styloDAO.find(...);
// Modifie des valeurs du DTO
styloTO.setPrix(125);
// Modifie le stylo dans la base
styloDAO.update(styloTO);
    
```

La seule ligne de code pour changer de SGBD

R. Grin

Mapping objet-relationnel

page 63

## Code client (suite)

```

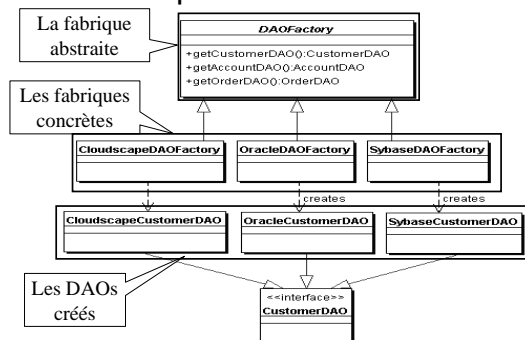
// Supprime un stylo de la base
styloDAO.delete(styloNo);
// Trouve tous les stylos d'une marque
// Utilise un stylo « exemple » de ce
// que l'on cherche
StyloTO styloEx = new StyloTO();
styloEx.setMarque("Marker");
Collection<StyloTO> listeStylos =
    styloDAO.find(styloEx);
...
    
```

R. Grin

Mapping objet-relationnel

page 64

## Fabrique abstraite – 1 DAO

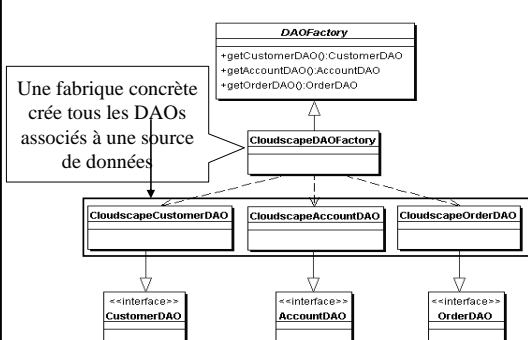


R. Grin

Mapping objet-relationnel

page 65

## Fabrique abstraite – 1 source de données

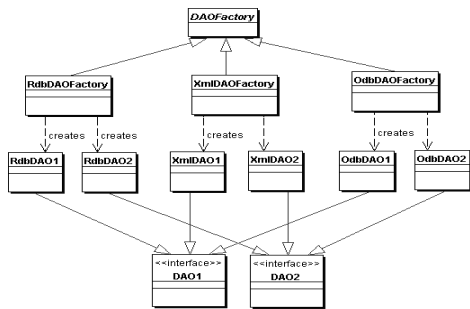


R. Grin

Mapping objet-relationnel

page 66

## Fabrique abstraite – schéma global



R. Grin

Mapping objet-relationnel

page 67

## Code pour fabrique abstraite

- Le code qui suit est un exemple schématique de l'utilisation du pattern DAO, avec le pattern fabrique abstraite (inspiré fortement d'un exemple donné par Sun)
- Il utilise aussi le pattern DTO/TO
- Important : ce pattern DTO peut être évité si les objets persistants peuvent être transportés entre les différentes couches d'une application (par exemple, pour une application locale, ou en utilisant les objets « détachés » de JPA ou de Hibernate)

R. Grin

Mapping objet-relationnel

page 68

## La fabrique abstraite

```

public abstract class DAOFactory {
    public enum TypeFabrique {MYSQL, ORACLE};
    public abstract StyloDAO getStyloDAO();
    public abstract FactureDAO getFactureDAO();
    ...
    public static DAOFactory getDAOFactory(
        TypeFabrique typeFabrique) {
        switch (typeFabrique) {
            case MYSQL: return new MySQLDAOFactory();
            case ORACLE: return new OracleDAOFactory();
            default: ... ; // erreur
        }
    }
}
    
```

R. Grin

Mapping objet-relationnel

page 69

## Une fabrique concrète

```

public class MySQLDAOFactory
    extends DAOFactory {
    @Override
    public StyloDAO getStyloDAO() {
        return new MySQLStyloDAO();
    }
    @Override
    public FactureDAO getFactureDAO() {
        return new MySQLFactureDAO();
    }
    ...
}
    
```

R. Grin

Mapping objet-relationnel

page 70

## Interface des DAO pour Stylo

```

public interface StyloDAO {
    public int insert(...);
    public boolean delete(...);
    public StyloDTO find(...);
    public boolean update(...);
    public Collection<StyloTO> findAll(...);
    public Collection<StyloTO> find(...);
    ...
}
    
```

référence

objet exemple

R. Grin

Mapping objet-relationnel

page 71

## DAO concret pour Stylo

```

import java.sql.*;
public class MySQLStyloDAO
    implements StyloDAO {
    public MySQLStyloDAO() { ... }
    public int insert(...) { ... }
    public boolean delete(...) { ... }
    public StyloTO find(...) { ... }
    public boolean update(...);
    public Collection<StyloTO> findAll(...);
    ...
}
    
```

R. Grin

Mapping objet-relationnel

page 72

## TO pour Stylo

```
import java.io.Serializable;
public class StyloTO implements Serializable {
    private String reference;
    private String name;
    ...
    // Accesseurs et modificateurs
    public String getReference() {...}
    public void setReference(String ref) {...}
    ...
}
```

indispensable pour être transporté d'une couche à une autre

R. Grin

Mapping objet-relational

page 73

## Inconvénient de ce pattern

- ❑ Si on veut ajouter un nouveau type de DAO (par exemple, un DAO pour un autre article), il faut modifier le code de toutes les fabriques abstraites et concrètes

R. Grin

Mapping objet-relational

page 74

## DAO et EJB

- ❑ Dans les applications construites selon la spécification EJB, les DAOs seront le plus souvent des beans sessions sans état (`@Stateless`) avec des contextes de persistance limités à une transaction (voir cours sur JPA)

R. Grin

Mapping objet-relational

page 75

## Bibliographie

- ❑ Patterns of Enterprise Application Architecture de Martin Fowler – Addison Wesley
- ❑ Présentation du pattern DAO et implémentation en Java, par Sun : <http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html>

R. Grin

Mapping objet-relational

page 76