IUT Lyon 1
l'excellence technologique

Java Avancé – Cours 6.2
Accès aux BD via JDBC, DAO

V. DESLANDRES

veronique.deslandres@univ-lyon1.fr

Plan de ce cours

Appeler une procédure stockée

- Procédure stockée = une procédure ou fonction inscrite en BD
- La méthode `prepareCall(String proc)`
  - Méthode de la classe `Connection`,
  - Paramètre sous la forme :
    - "{? = call nomFonction([?,?,…])}"
    - "{call nomProcédure([?,?,…])}«
- **L'interface** `CallableStatement` est dédiée à l'appel des procédures stockées

  Comme pour la méthode prepareStatement, les paramètres sont définis par des ?.

```
CallableStatement cs = connection.prepareCall("{? = call
inc_parametre(?)}");
```

Paramètres en entrée d'un CallableStatement

- Méthodes `setXXXX(indice, valeur)`

  Le passage de paramètre à un CallableStatement est identique à prepareStatement().

```
CallableStatement cs = connection.prepareCall("{? = call
  inc_parametre(?)}");
cs.setString(2,nopar);
```

Paramètres en sortie d'un CallableStatement

- Méthode `registerOutParameter(indice, type)`
  - `Indice` : position du paramètre,
  - `Type` : entier (contante ou constante nommée) identifiant le type.

```
CallableStatement cs = connection.prepareCall("{? = call
  inc_parametre(?)}");
cs.registerOutParameter(1,Types.INTEGER);
```

Les constantes nommées :

```
Types.VARCHAR, Types.DATE, Types.REAL ...
```

Récupération d'un paramètre en sortie d'un CallableStatement

- Méthode `getXXXX()`
  - variable = get*Type* (indice | "nom_colonne")

```
int nb = cs.getInt(1)
```

Paramètres de sortie

- getBigDecimal(), getBoolean(),
- getBytes(), getDate(), getDouble(),
- getFloat(), getInt(), getLong(),
- getString(), getTime(), getTimestamp()

Exécuter un CallableStatement

- Méthode `execute()`
  - Retourne un booléen :
    - `true` : l'exécution a produit un ResultSet
    - `false` : pas de retour, ou mise à jour.

```
cs.execute();
```

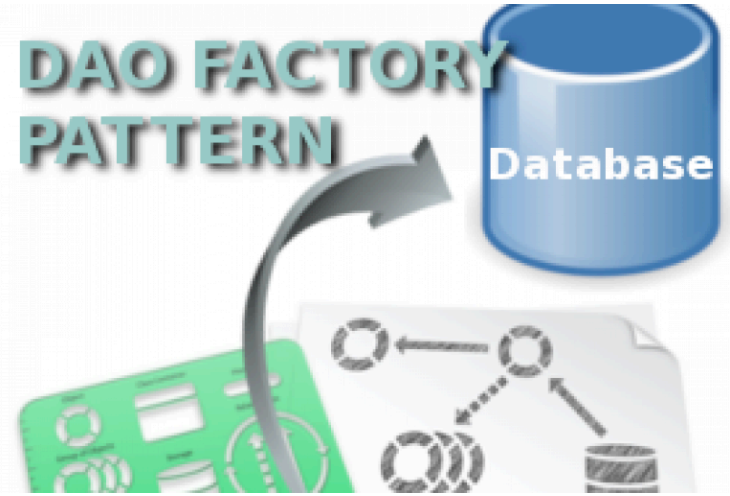
CallableStatement : illustration

Soit la fonction définie par :

```
CREATE OR REPLACE FUNCTION maFonction
(numero IN voyage.numVoyage%TYPE)
RETURN voyage.nomVoyage%TYPE IS nom voyage.nomVoyage%TYPE
BEGIN select nomvoyage into nom
FROM voyage where numvoyage = numero;
return nom;
END;
```

Illustration : appel à la fonction

```
CallableStatement cst = maConnexion.prepareCall("{? =
call maFonction(?)}");
cst.setInt(2,6);                          // le numéro de voyage
cst.registerOutParameter(1,java.sql.Types.VARCHAR);
boolean retour = cst.execute();
String nom = cst.getString(1);            // le nom du voyage
…
```



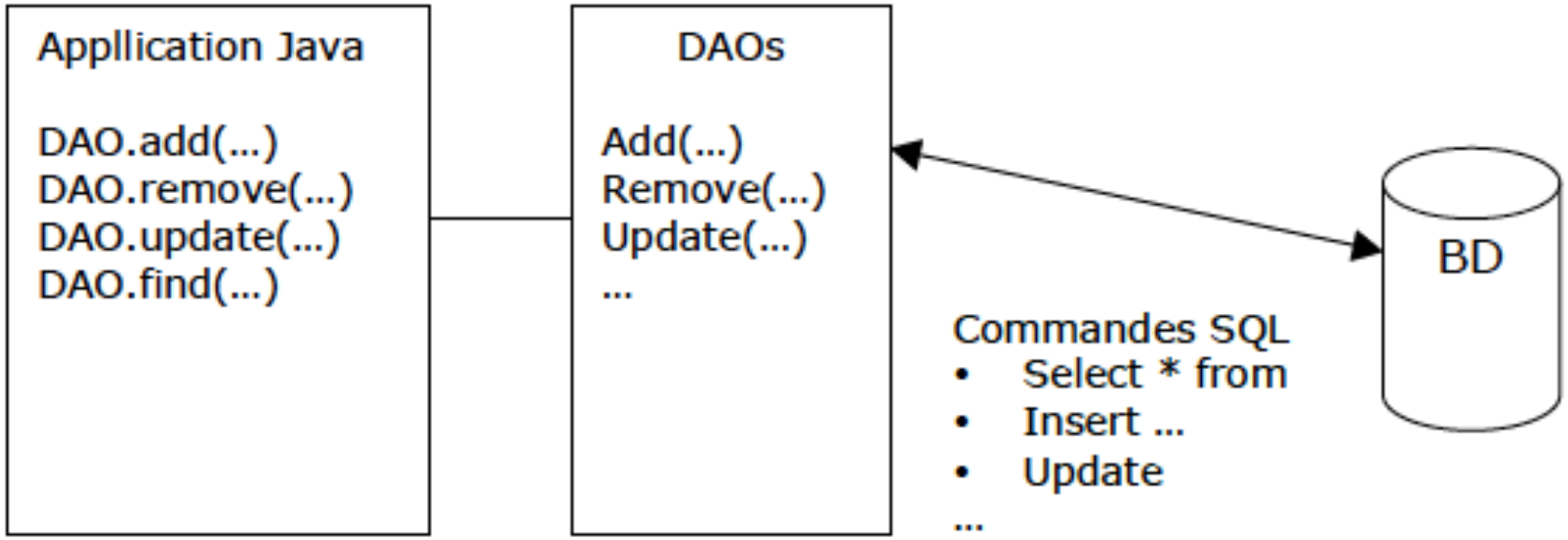
DAO – Design Pattern

PATRON POUR LA PERSISTANCE DES DONNÉES (DATA ACCESS OBJECT)

Le Pattern DAO

- Rappel : pattern MVC est utilisé pour l'interaction entre les couches **métiers** et **présentation**
- Le pattern DAO : pour le modèle, on détaille **l'implémentation des accès aux données**
  - *Data Access Object*
- Isoler la gestion de la **persistance** dans des objets spécifiques
  - Découpler Métier / Persistance

Architecture DAO



L'application java n'a aucun n'information sur l'accès de BD, elle utilise des DAOs

La couche DAO s'occupe de tous les accès à la base de données

Principe du pattern DAO

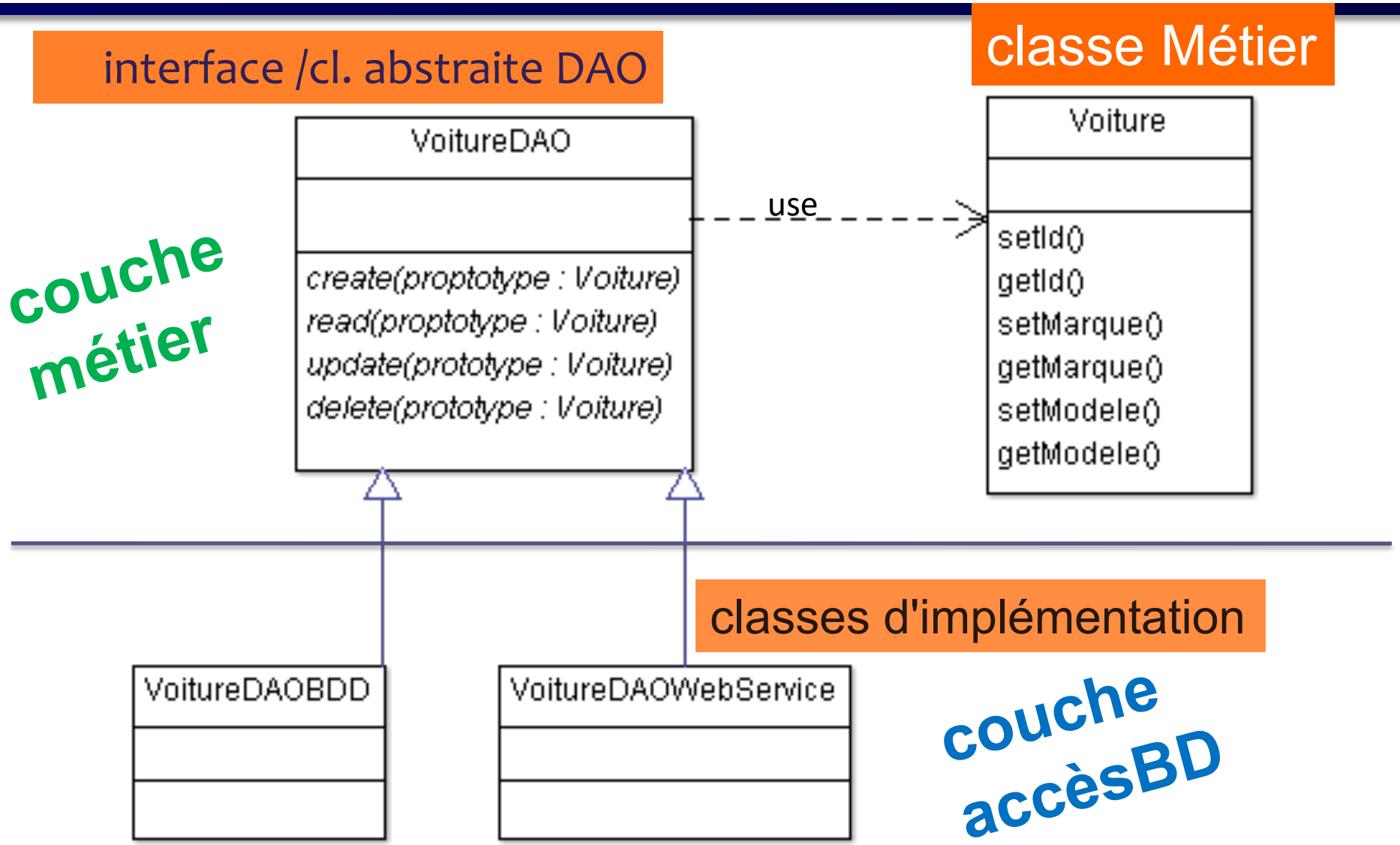
Proposer, pour les objets métiers :

- une interface de la gestion de la persistance (services **CRUD**)
  - `Create` : création d'une nouvelle entité
  - `Read/ Retrieve` : lire / rechercher des entités
  - `Update` : modifier une entité
  - `Delete` : supprimer un entité
- **indépendante de la source de données**
- et **d'autres fonctionnalités** métier

Intérêts du DAO

- Au niveau de la **couche métier**
  - Il n'y a plus de requêtes SQL ou d'ordres de connexion spécifiques codés en dur dans les objets Métier
  - L'accès aux données se fait uniquement par des objets DAO
  - On masque le mapping objet-relationnel
- Au niveau de la **couche d'accès aux données**
  - La maintenance des accès BD est facilitée : la source de données peut être modifiée sans impacter la couche métier
  - Factorisation du code d'accès aux données
  - Sécurité, fiabilité des accès

Ex. DAO


interface /cl. abstraite DAO
classe Métier
VoitureDAO
create(proptotype : Voiture)
read(proptotype : Voiture)
update(prototype : Voiture)
delete(prototype : Voiture)
use
Voiture
setId()
getId()
setMarque()
getMarque()
setModele()
getModele()
couche métier
classes d'implémentation
VoitureDAOBDD
VoitureDAOWebService
couche accèsBD

L'interface d'objet DAO

- Expose les **fonctionnalités CRUD** indépendantes de l'implémentation
- Aucune des méthodes spécifiées ne doit contenir de requête SQL en paramètre
- Doit proposer une gestion personnalisée des exceptions
  - les exceptions standards sont attrapées et redirigées vers une ou des exceptions particulières
  - gérées par une ou plusieurs **classes d'exceptions indépendantes du support** de persistance

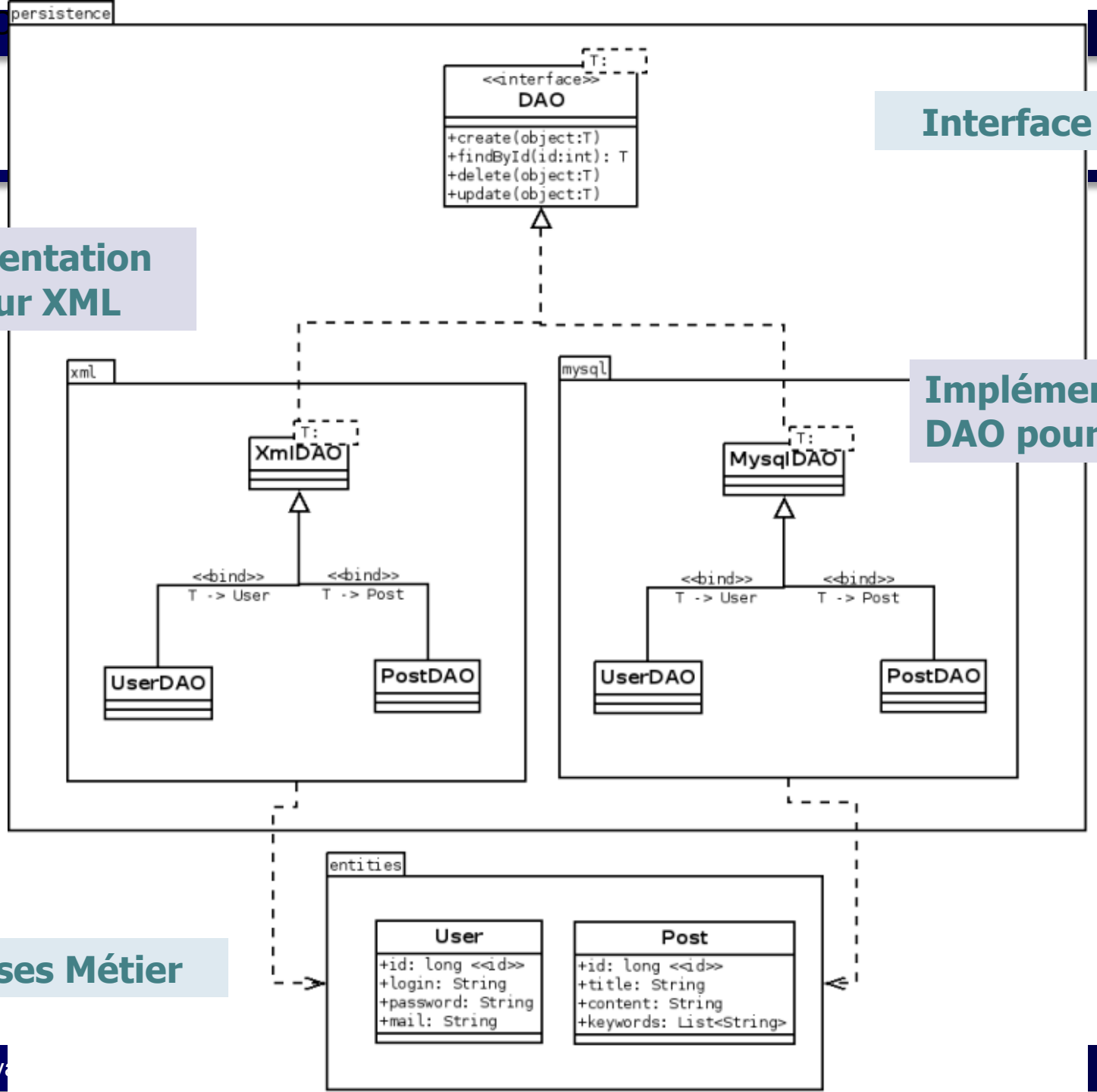
L'implémentation d'objet DAO

- Un seul objet DAO créé par classe métier
  - il y a donc autant d'objets DAO que de classes métier
- L'application ne manipule **que** les objets métier
  - seuls les objets métier utilisent les services de **leur DAO**
- Création des objets DAO
  - les objets DAO dépendent de la source de données
  - il existe donc une famille d'objets DAO par type de source de données

Interface DAO

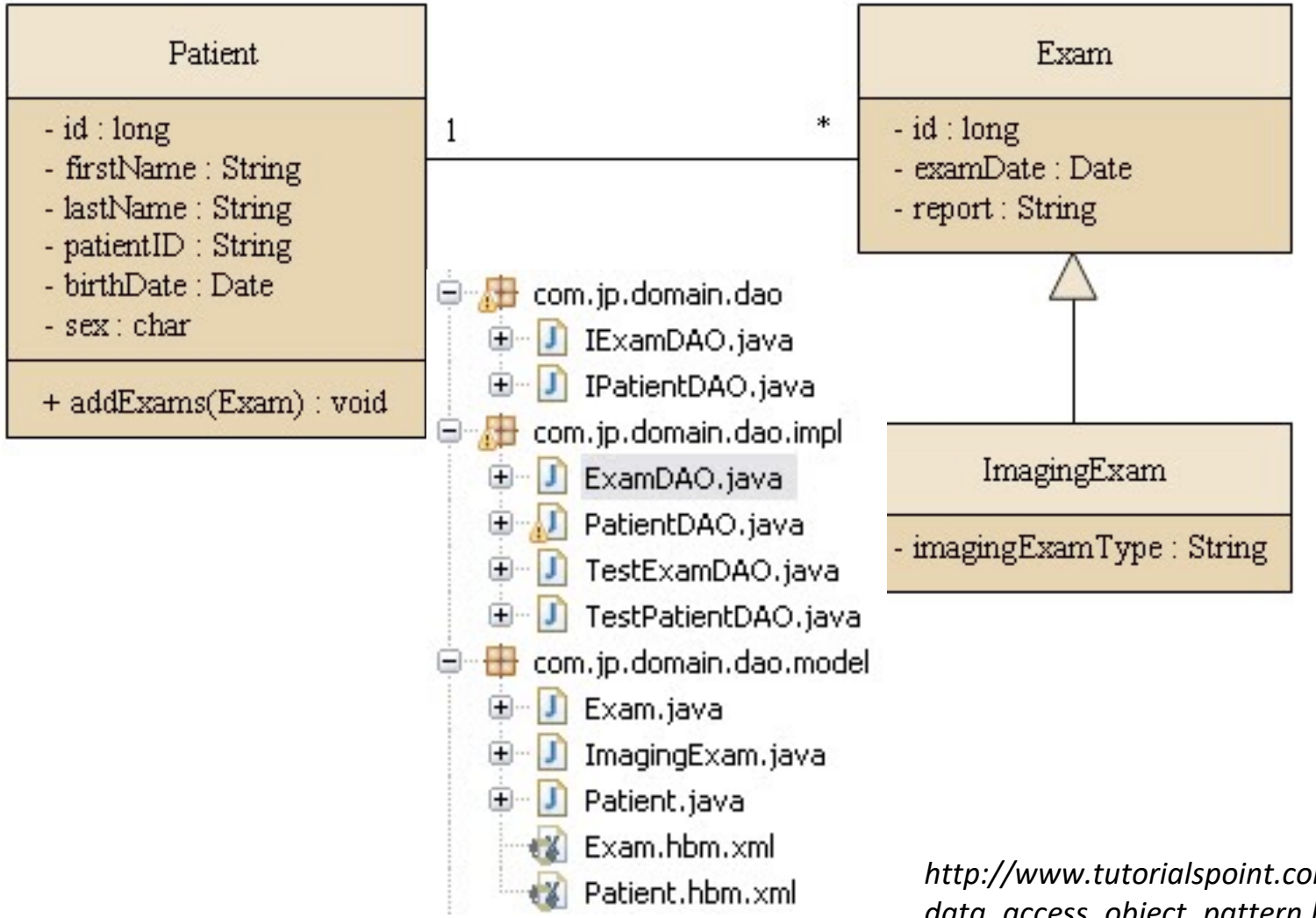
Implémentation DAO pour XML

Implémentation DAO pour MySQL



Classes Métier

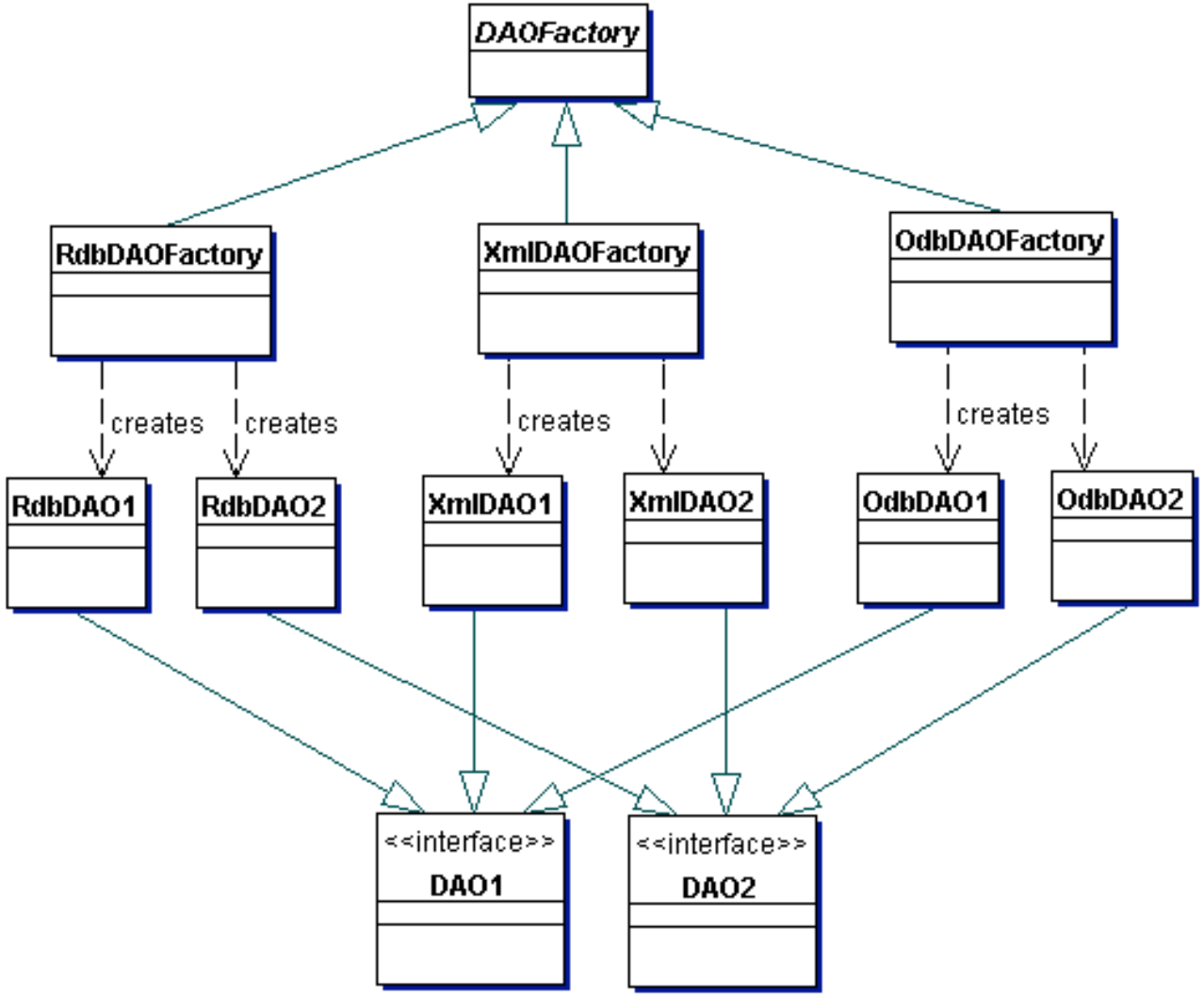
Architecture DAO



http://www.tutorialspoint.com/design_pattern/data_access_object_pattern.htm

Factory & DAO

- On utilise souvent un **FactoryMethod** (ici, DAOFactory) pour implémenter les accès possibles (ici, getCustomer, getAccount, getOrder) pour une même connexion à un SGBD
- *(Ou AbstractFactory Method lorsqu'on a plusieurs SGBD)*



Source : http://lia.deis.unibo.it/Courses/TecnologieWeb0708/materiale/laboratorio/guide/dao/PatternDAO.pdf

Ex. Factory / Fabrique

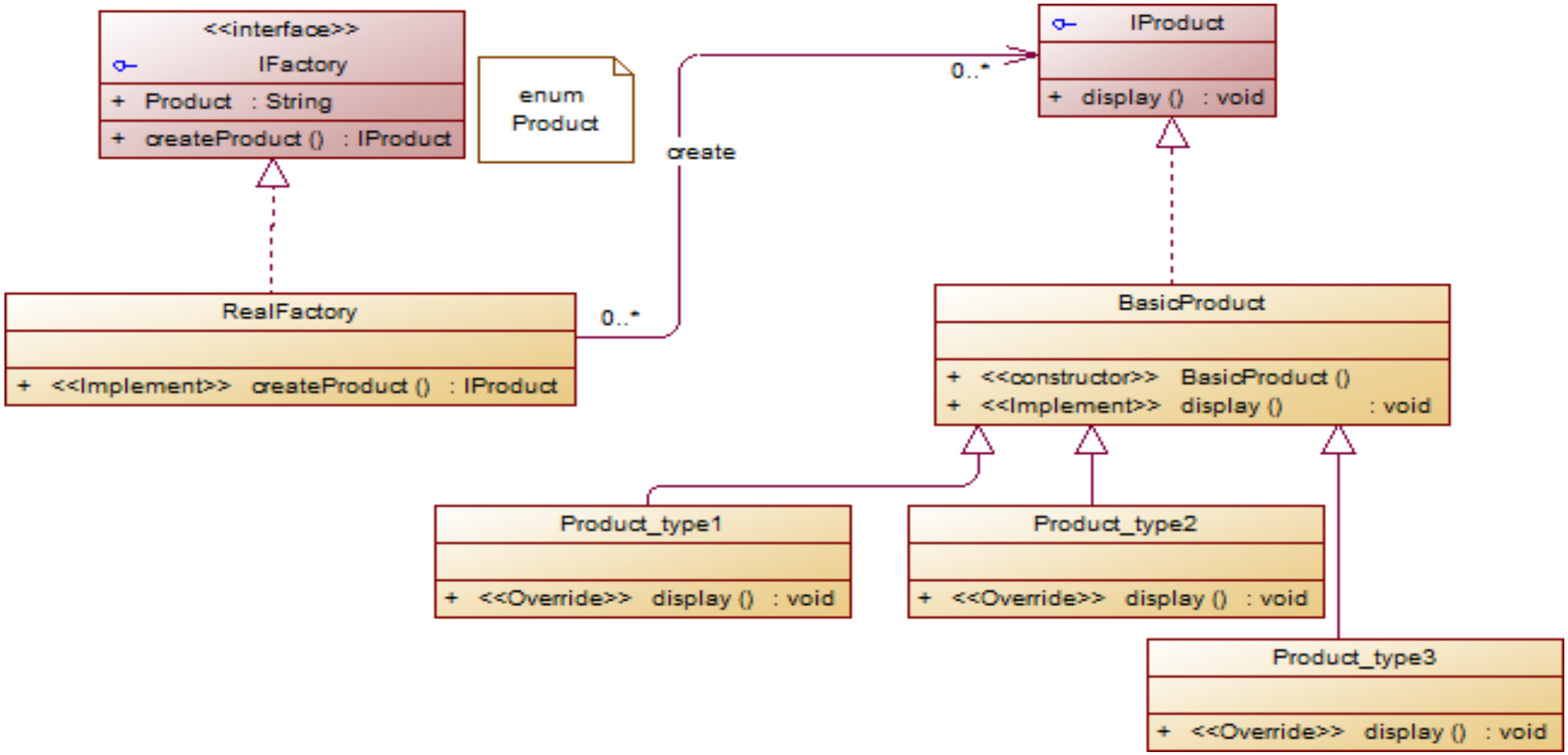
Confier la création d'une instance à une **Fabrique générique**, l'instance sera choisie parmi une arborescence donnée de classes, en fonction de paramètre fourni à l'exécution :

```
public class FabriqueSauvegarde {
   Sauvegarde getSauvegarde(String typePersistance) throws ExceptionCreation {
     if (typePersistance.equals( "fichier")) {
           return new SauvegardeFichier();     }
     else if (typePersistance.equals( "SGBD")) {
           return new SauvegardeSGBD();     }
     throw new ExceptionCreation("Impossible de créer une sauvegarde " + typePersistance);
   }
}
```

Permet à une classe métier de faire une sauvegarde, quelque soit le type de sauvegarde

(On utilisera le pattern FactoryMethod pour créer un pool de connexions aux BD avec JDBC)

Factory : exemple
un affichage ciblé de produits



JDBC

POUR ALLER PLUS LOIN

SQLException

- C'est en fait une **liste d'exceptions**

- Exploitation :

```
catch (SQLException e) {
   while (e != null) {
     System.out.println(e.getSQLState());
     System.out.println(e.getMessage());
     System.out.println(e.getErrorCode());
     e = e.getNextException();
   }
}
```

Gestion de pool : quand fermer une connexion ?

- La réponse dépend des logiciels utilisés…
- Certains gèrent d'eux-mêmes les connexions du pool disponibles, d'autres supposent qu'on les ferme explicitement

Types Wrappers plutôt que primitifs

- C'est une bonne pratique de toujours utiliser les objets *Wrapper* dans les classes dont les propriétés correspondent à des champs d'une base de données, afin d'éviter les erreurs de *mapping* quand un champ est vide
  - (null retourné, or pas de null pour les types primitifs)
- Exemples :
  - Préférer : `Long id` plutôt que `long id;`

DAO: distinguer les types d'exceptions (1)

Objectif : séparer les informations de configuration du reste de l'application

```
public class DAOException extends RuntimeException {
/*
* Constructeurs
*/
   public DAOException( String message ) {
          super( message );
   }
   public DAOException( String message, Throwable cause ) {
          super( message, cause );
   }
   public DAOException( Throwable cause ) {
          super( cause );
   }
}
```

https://openclassrooms.com/courses/creez-votre-application-web-avec-java-ee/le-modele-dao

Distinguer les types d'exceptions (2)

```
public class DAOConfigurationException extends RuntimeException {
/*
* Constructeurs
*/
public DAOConfigurationException( String message ) {
super( message );
}
public DAOConfigurationException( String message, Throwable cause ) {
super( message, cause );
}
public DAOConfigurationException( Throwable cause ) {
super( cause );
}
}
```

(ancienne méthode de connexion : « Legacy code »)

CONNEXION AVEC DRIVER MANAGER

Processus de Connexion

```
import java.sql.DriverManager;          // gestion des pilotes
import java.sql.Connection;             // une connexion à la BD
import java.sql.Statement;              // une instruction
import java.sql.ResultSet;              // un résultat (lignes/colonnes)
import java.sql.SQLException;           // les erreurs liées à la BD

public class JDBCSample {
// 0- chargement du pilote (pour JDBC antérieur à JDBC4)
// 1- ouverture de connexion
// 2- exécution d'une requête
// 3- programme principal
}
```

Créer une connexion

- Méthode getConnection() de DriverManager :
- `Connection conn =` DriverManager.getConnection(URL, userid, password);

- Toute la difficulté réside dans la définition de la chaîne de connexion :

```
final String url = "oracle.jdbc.driver.OracleDriver";

// "org.mariadb.jdbc.Driver" ; pour MariaDB
// "sun.jdbc.odbc.JdbcOdbcDriver" ; pour ACCESS
```

À l'IUT

- **URL pour MySQL sur le serveur IUT :**

```
String url="jdbc:mysql://iutdoua-web.univ-lyon1.fr/pxxxxx";
String userid="pxxxx";
String password="votrePwd";
```

- **URL pour Oracle sur le serveur IUT :**

```
jdbc:oracle:thin:iutdoua-oracle.univ-lyon1.fr:1521:orcl
```

En local avec MySQL ou MariaDB

- ```
  String url = "jdbc:mariaDB://localhost:3306/maBD";
  Connection conn = DriverManager.getConnection(url,"root",
  "root");
  ```

Paramètres indiqués par votre environnement de serveur APACHE local

MySQL

MySQL peut être administrée via phpMyAdmin.

Pour vous connecter au serveur MySQL dans vos propres scripts PHP, utilisez les paramètres suivants:

Hôte	localhost
Port	3306
Utilisateur	root
Mot de passe	root
Socket	/Applications/MAMP/tmp/mysql/mysql.sock
  ```

NOTA

- On peut aussi utiliser un fichier de propriétés avec DriverManager :

  DriverManager.getConnection(String url, Properties info);

- Cf http://www.tutorialspoint.com/jdbc/jdbc-db-connections.htm

Enregistrement d'un pilote JDBC

- Chargement explicite d'un pilote antérieur à JDBC4 :

```
private String driverName = "com.mysql.jdbc.Driver";

void loadDriver() throws ClassNotFoundException {
    Class.forName(driverName);
}
```

- L'appel à `forName()` déclenche un chargement dynamique du pilote.
- Un programme peut utiliser plusieurs pilotes, un pour chaque base de données.
- Le pilote doit être accessible à partir de la variable d'environnement CLASSPATH.
- Le chargement explicite est inutile à partir de JDBC 4
  - L'application charge le 1er pilote JDBC4 trouvé dans les librairies

JDBC : je retiens

- Les principes : API Java, drivers des BD
  - Les principaux éléments, leurs noms
- La connexion
  - Pool de connexions (*DataSource*)
- Le principes d'exécution de requêtes
  - Les interfaces Java *Statement, PreparedStatement, CallableStatement*, etc.
  - Les méthodes execute(), executeUpdate()
  - L'exploitation des résultats avec *ResultSet*
- L'utilisation de DAO, le principe d'un Singleton