IUT Lyon 1

l'excellence technologique

Java Avancé – Cours 5
Accès aux BD via JDBC, DAO

V. DESLANDRES, I. GUIDARA

veronique.deslandres@univ-lyon1.fr

Mai 2018

Plan de ce cours

V. Deslandres (c) - IUT de LYON

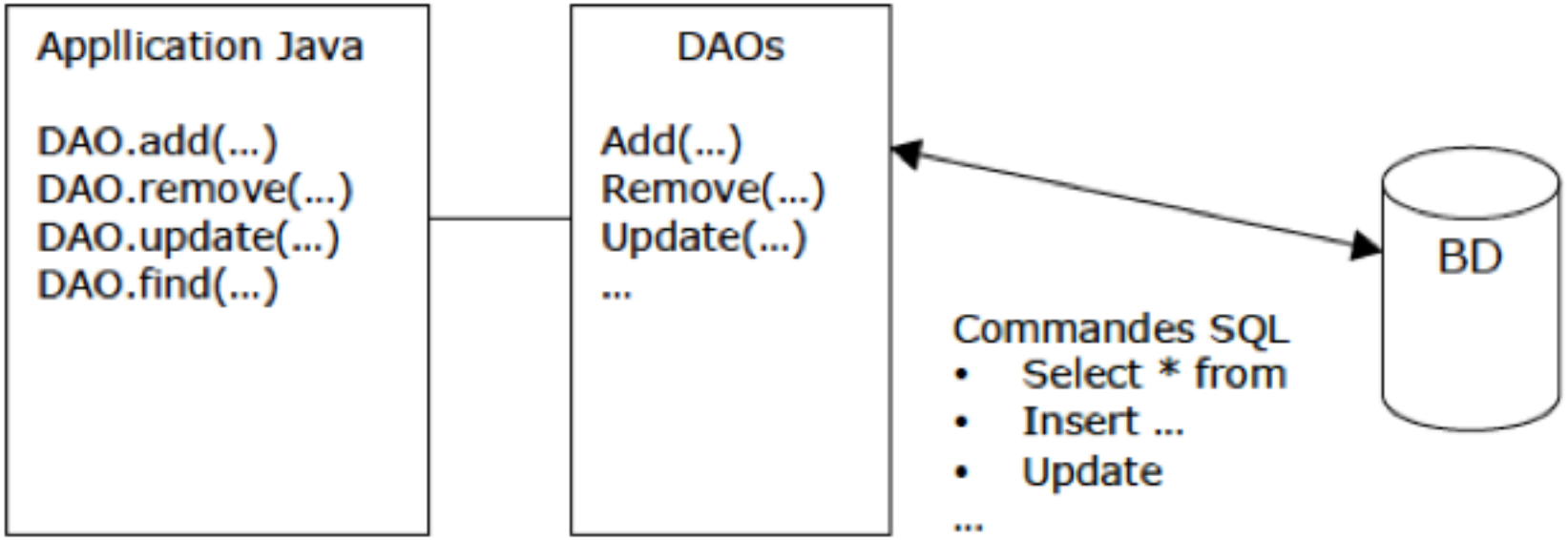
DAO – Design Pattern

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

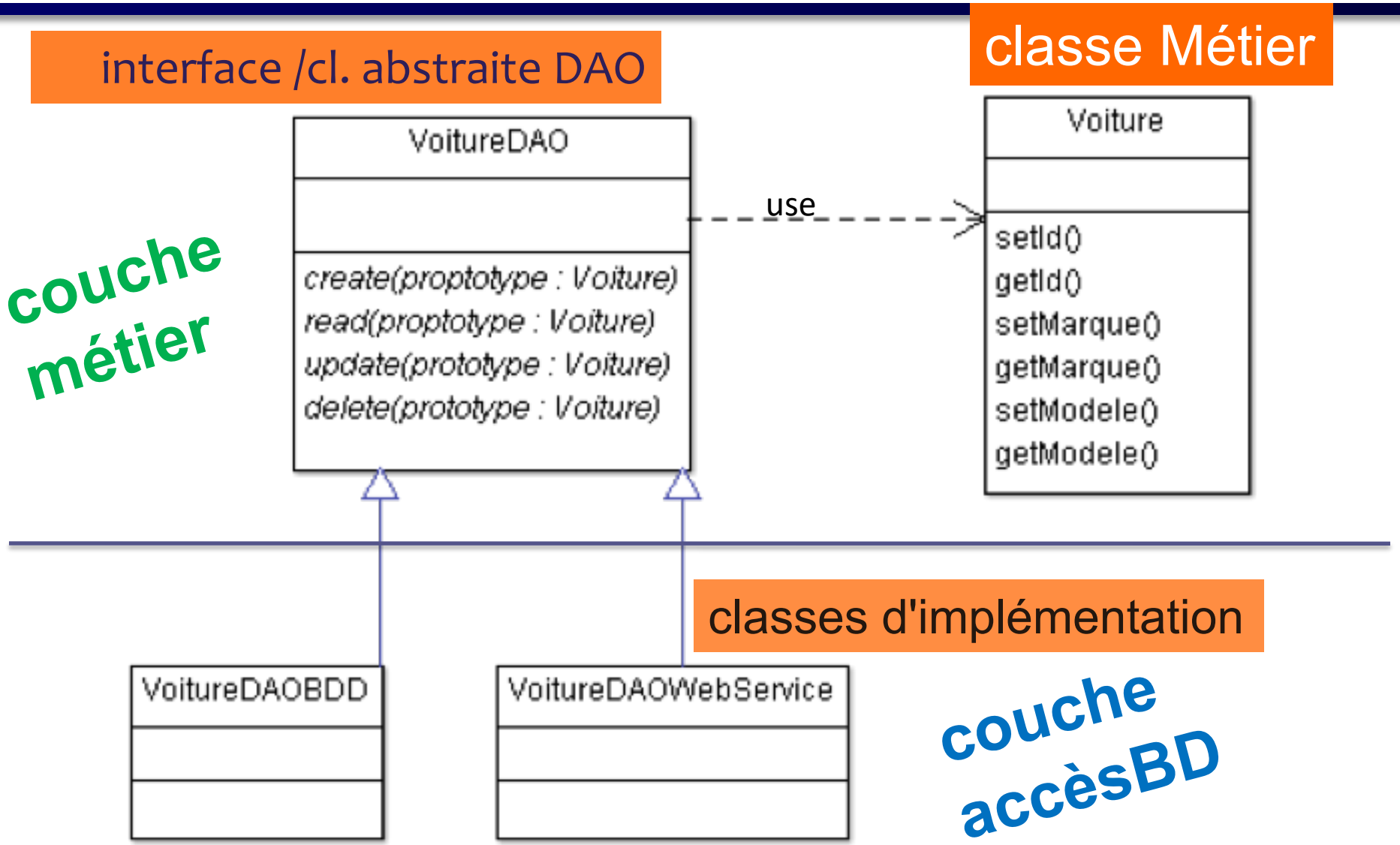
Proposer, pour les objets métiers :

- 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

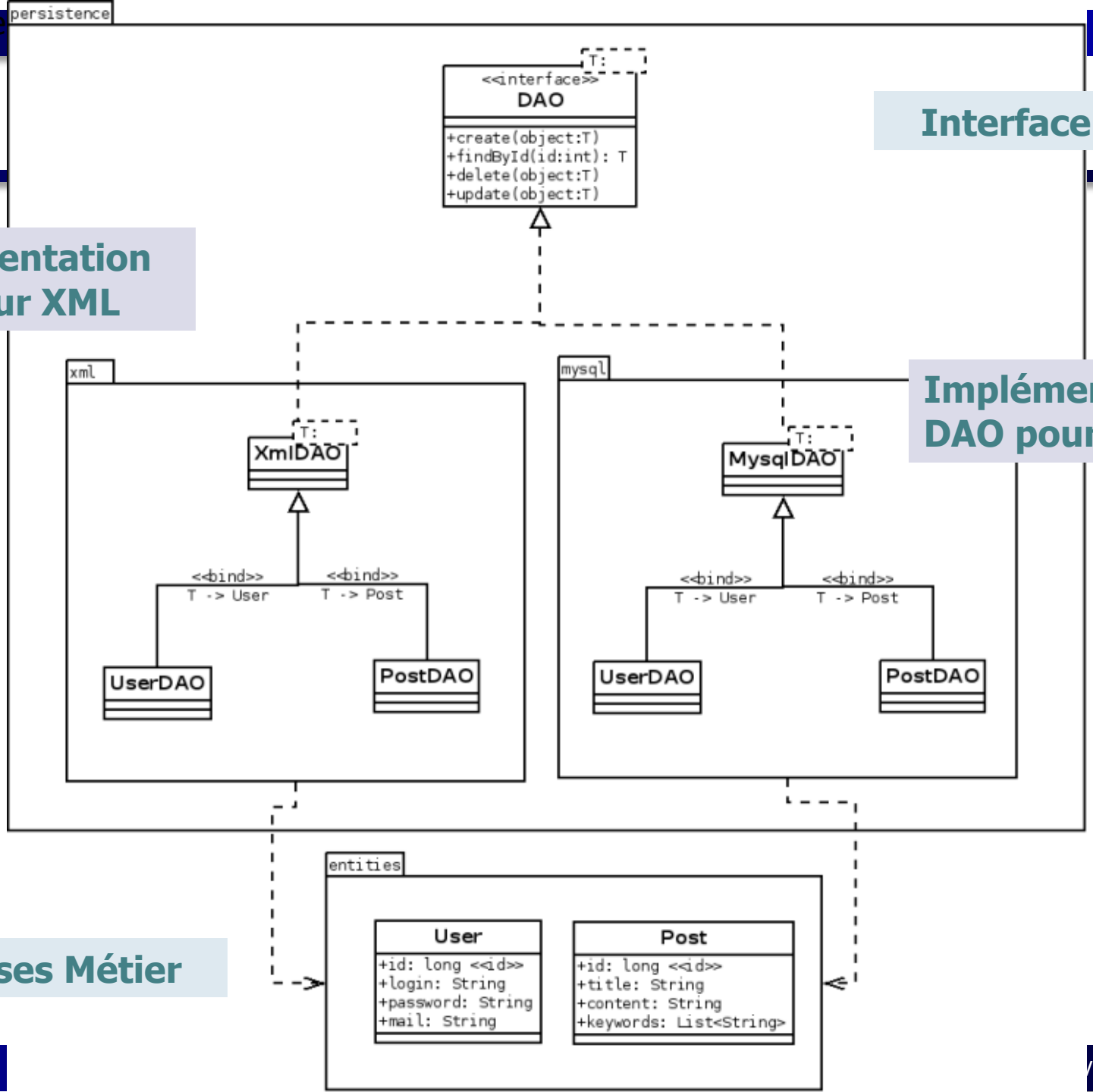

interface /cl. abstraite DAO
classe Métier
VoitureDAO
create(proptotype : Voiture)
read(proptotype : Voiture)
update(prototype : Voiture)
delete(prototype : Voiture)
use
Voiture
setId()
getId()
setMarque()
getMarque()
setModele()
getModele()
couche métier
classes d'implémentation
VoitureDAOBDD
VoitureDAOWebService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



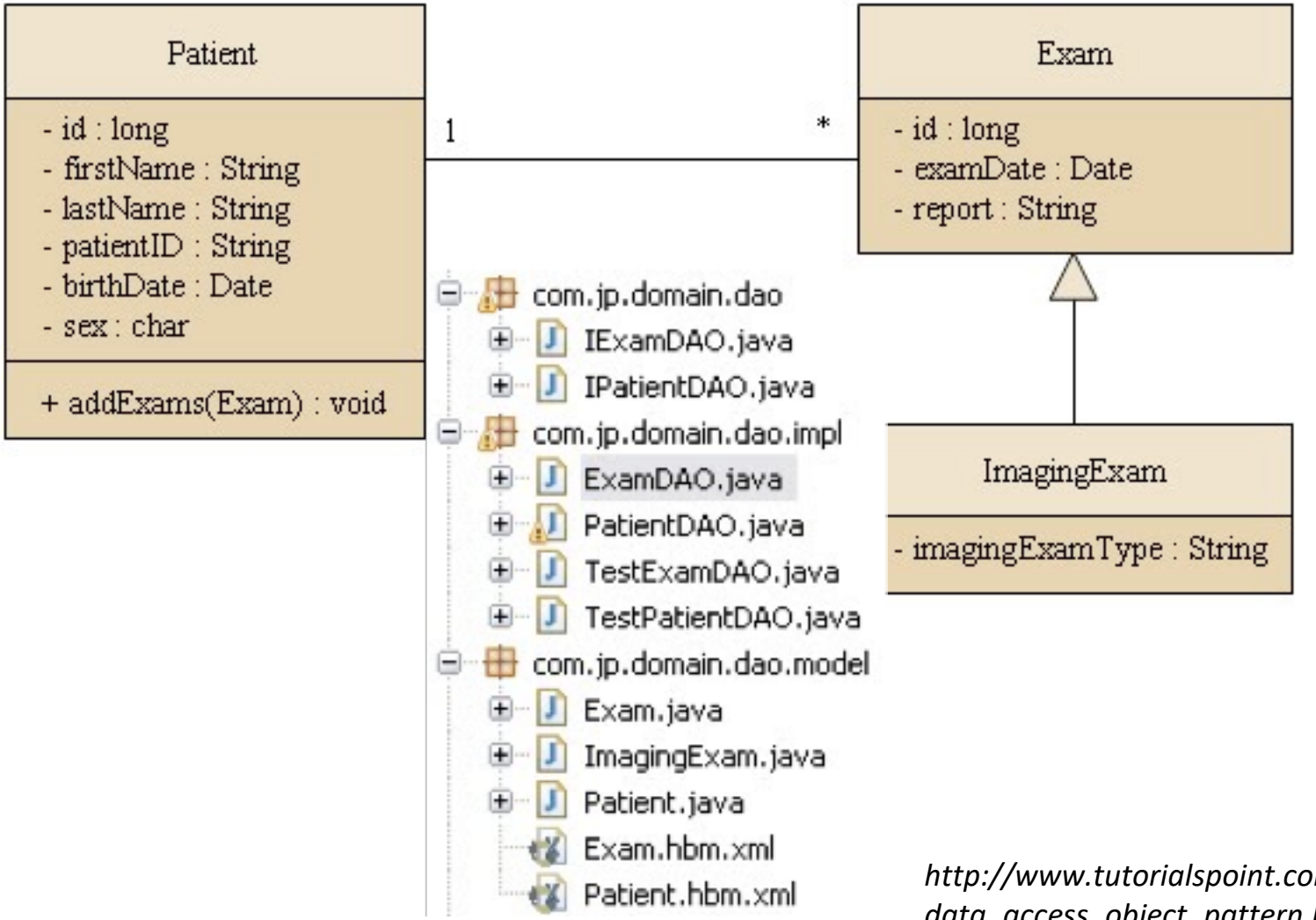
Interface DAO

Implémentation DAO pour XML

Implémentation DAO pour MySQL

Classes Métier

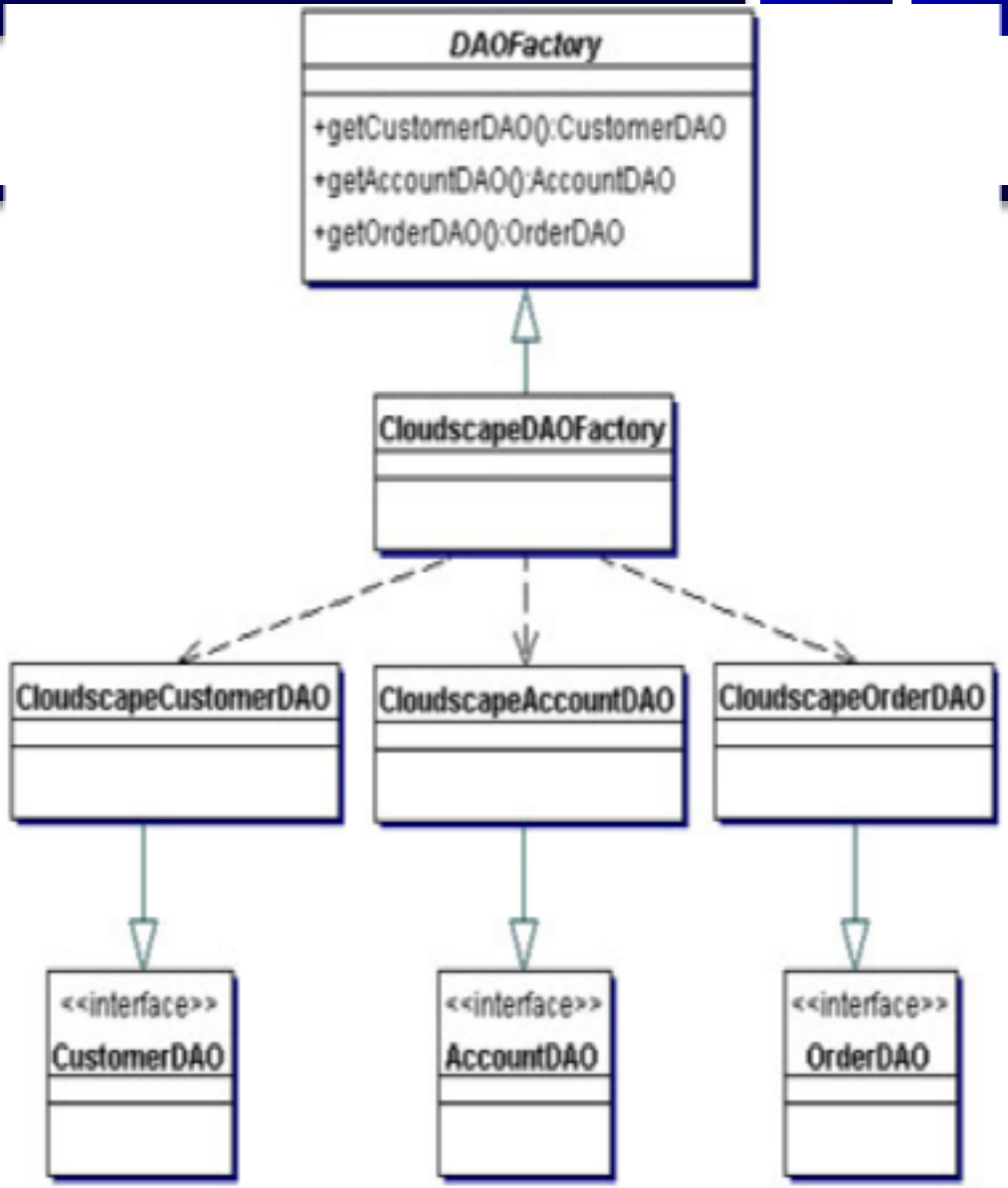
Architecture DAO



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)*



Cf http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

Ex. Factory / Fabrique

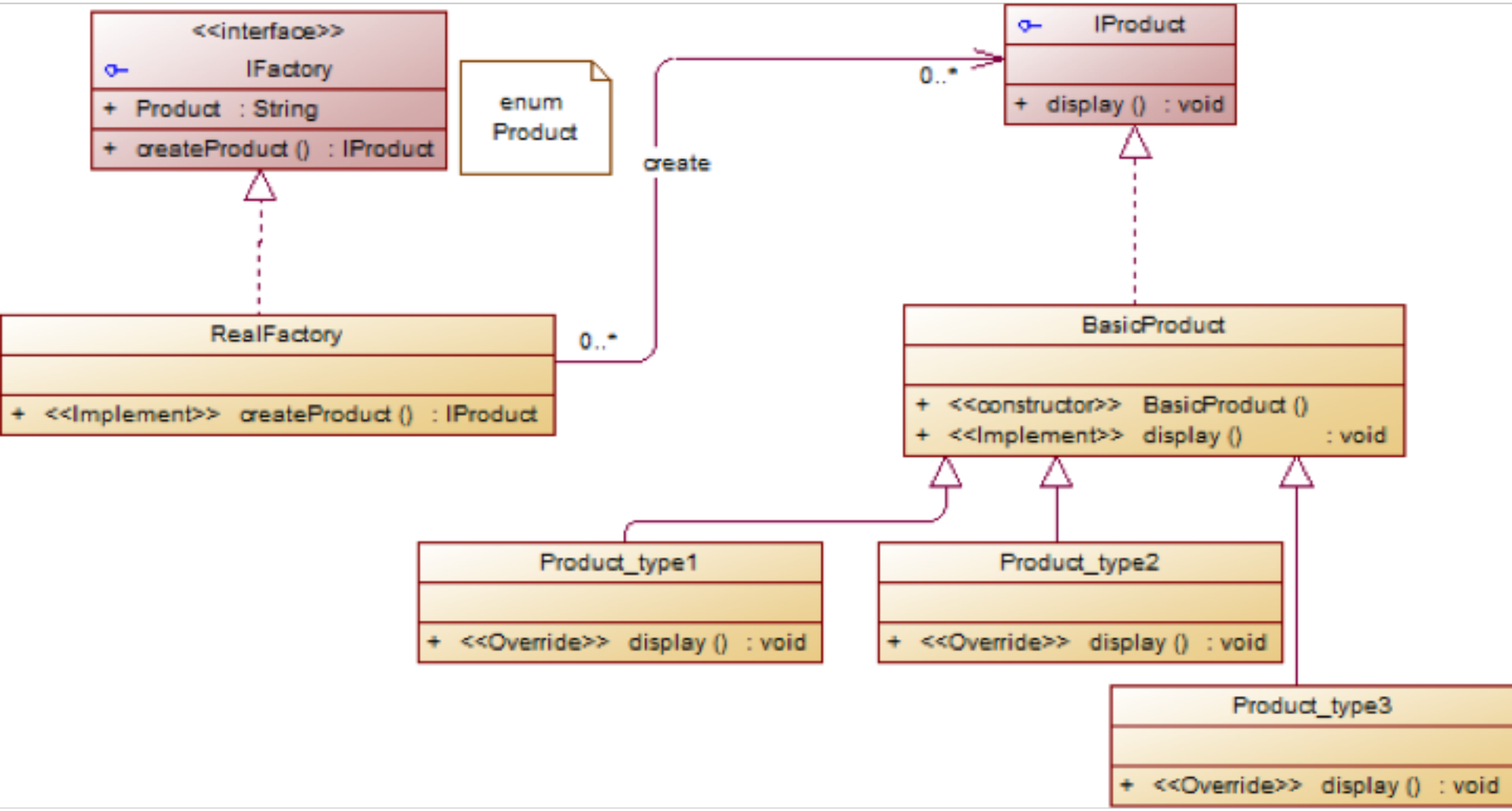
Confier la création d'une instance à une **Fabrique générique**, l'instance sera choisie parmi une arborescence donnée de classes, en fonction de paramètre fourni à l'exécution :

```
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, qq soit le type

(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

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

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 );
}
}
```

JDBC : je retiens

- Les principes : API Java, drivers des BD
  - Les principaux éléments, leurs noms
- Les 2 types de connexions
  - *DriverManager*
  - 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