LOG660 - Bases de données de haute performance

Gestion de la persistance transparente

Hiver 2011

C. Desrosiers

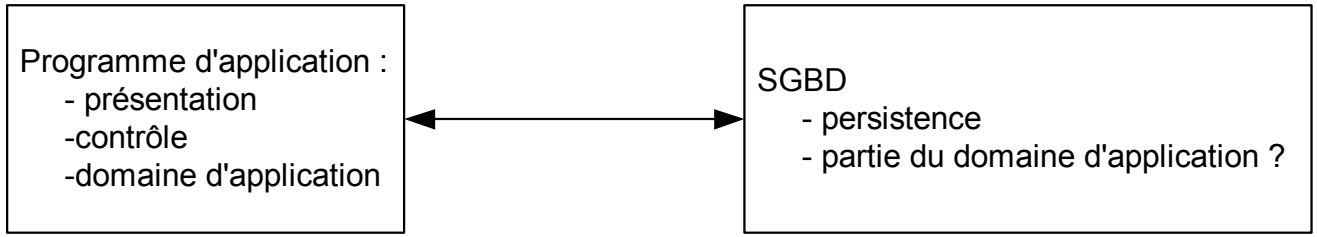
ÉTS Département de génie logiciel et des TI

Patron d'architecture en couche (layer)

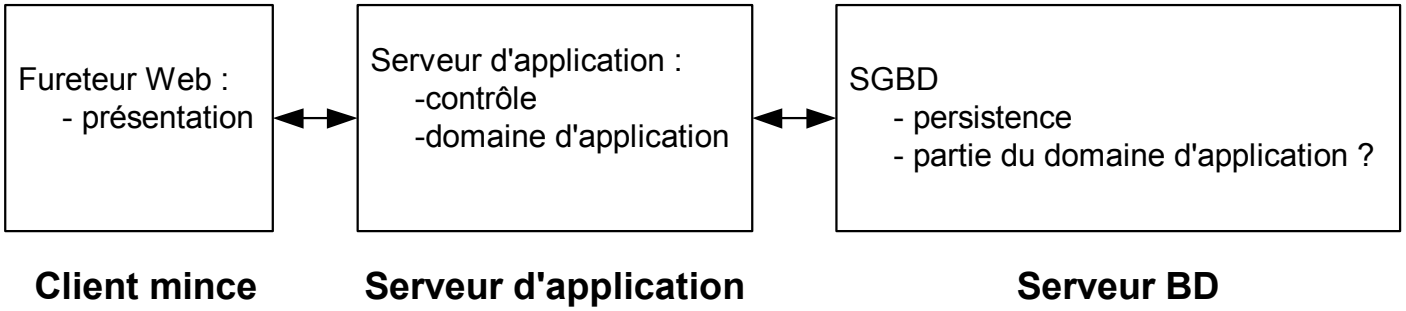
- Couche présentation
  - Ex: Servlets, JSP, ASP, PHP
- Couche contrôle (coordonnateur d'application)
- Couche domaine d'application (ou métier)
- Couche de services
  - persistance, transaction, communication, sécurité, etc.
  - Ex: EJB, Spring, Hibernate/iBatis (persistance)

Architectures physiques

- Architecture client-serveur



- Architecture Web à trois niveaux



Définition de la persistance

- Le stockage d'un objet ou d'une entité du domaine d'application sur disque ou tout autre dispositif de stockage, de manière à être utilisé d'une session à l'autre
- Possède les propriétés de *durabilité* (ne pas perdre les données en cas de panne) et d'*intégrité* (préserver cohérence des données malgré des accès/mise à jour concurrents)
- Normalement réalisée à l'aide d'une base de données
- Problème: le domaine d'application est modélisé à l'aide d'objets alors que la BD emploie un modèle relationnel (*object-relationnel impedance mismatch*)

Incompatibilité objet vs relationnel

- Granularité:
  - Ex: Une classe adresse ne correspond pas forcément à une table adresse (afin d'éviter la jointure)
- Sous-types (relation de spécialisation):
  - Concept de polymorphisme absent en relationnel
  - Plusieurs solutions possibles (délégation/fusion/concaténation)
- Identité (équivalence):
  - Valeur ou référence (objet) VS clé primaire (relationnel)
- Associations:
  - Unidirectionnel (objet) VS bidirectionnel (relationnel)
  - Plusieurs à plusieurs nécessite une table de jointure
  - Problème des n+1 sélections (ex: liste de commandes d'un client)

Gestion de la persistance

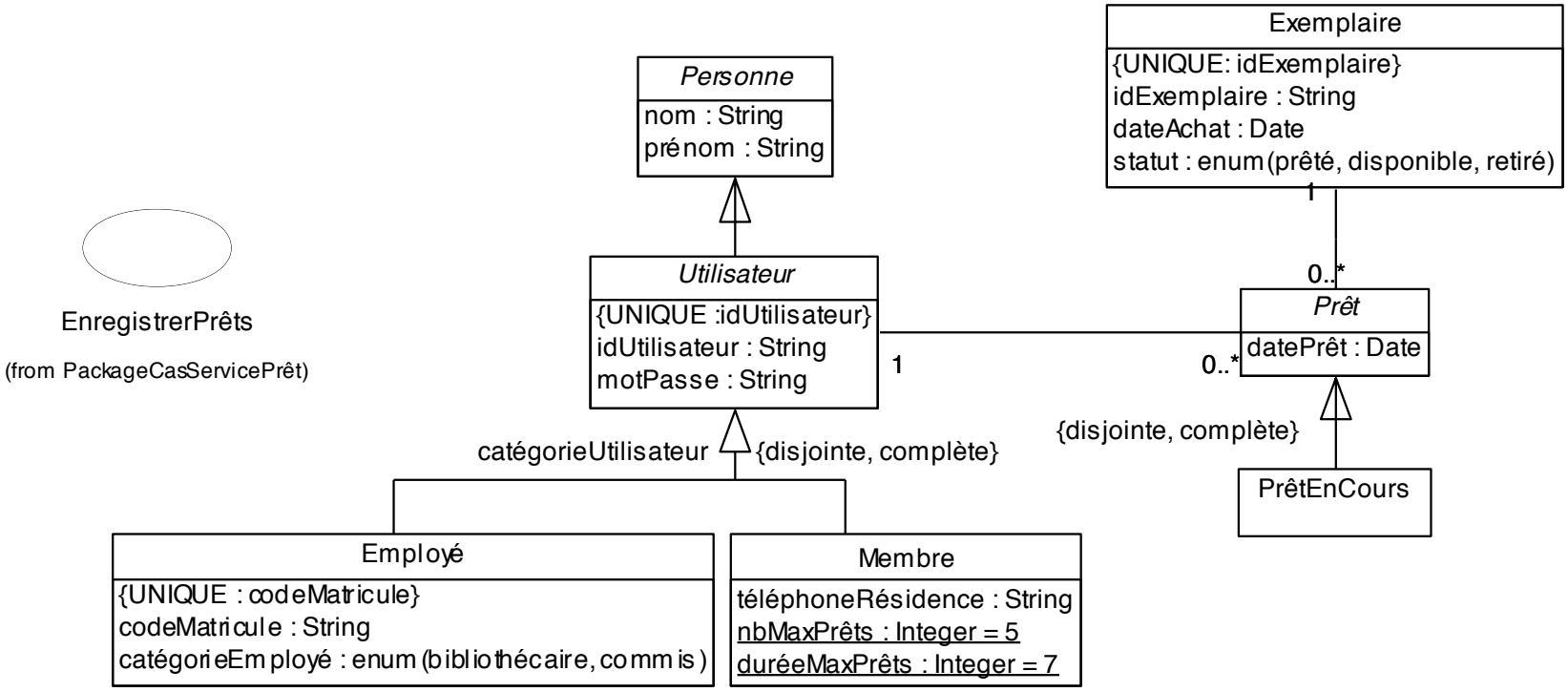
- Trois approches:
  - Matérialisation/dématérialisation à l'aide de JDBC
  - EJB entité (EJB2)
  - Framework de persistance transparente:
    - Object / relationnal mapping ORM (ex: JDO, Hibernate, EJB3)
    - Object / SQL mapping (ex: iBatis/myBatis)

Persistance avec JDBC

- Dématérialisation:
  - Reconstruire un objet persistant à partir des données de la BD
  - Se faite à l'aide d'un ou plusieurs SELECT
- Matérialisation:
  - Rendre/mettre à jour un objet persistant dans la BD
  - Se fait à l'aide d'un ou plusieurs INSERT/ UPDATE
- Patron de courtier BD (*database broker*)
  - Classe responsable de la matérialisation/dématérialisation des objets métiers à l'aide de requêtes à la BD
  - Permet de découpler la couche métier de la couche service de persistance (isole les interactions avec la BD)
  - Le patron Data Access Object (DAO) est une variante

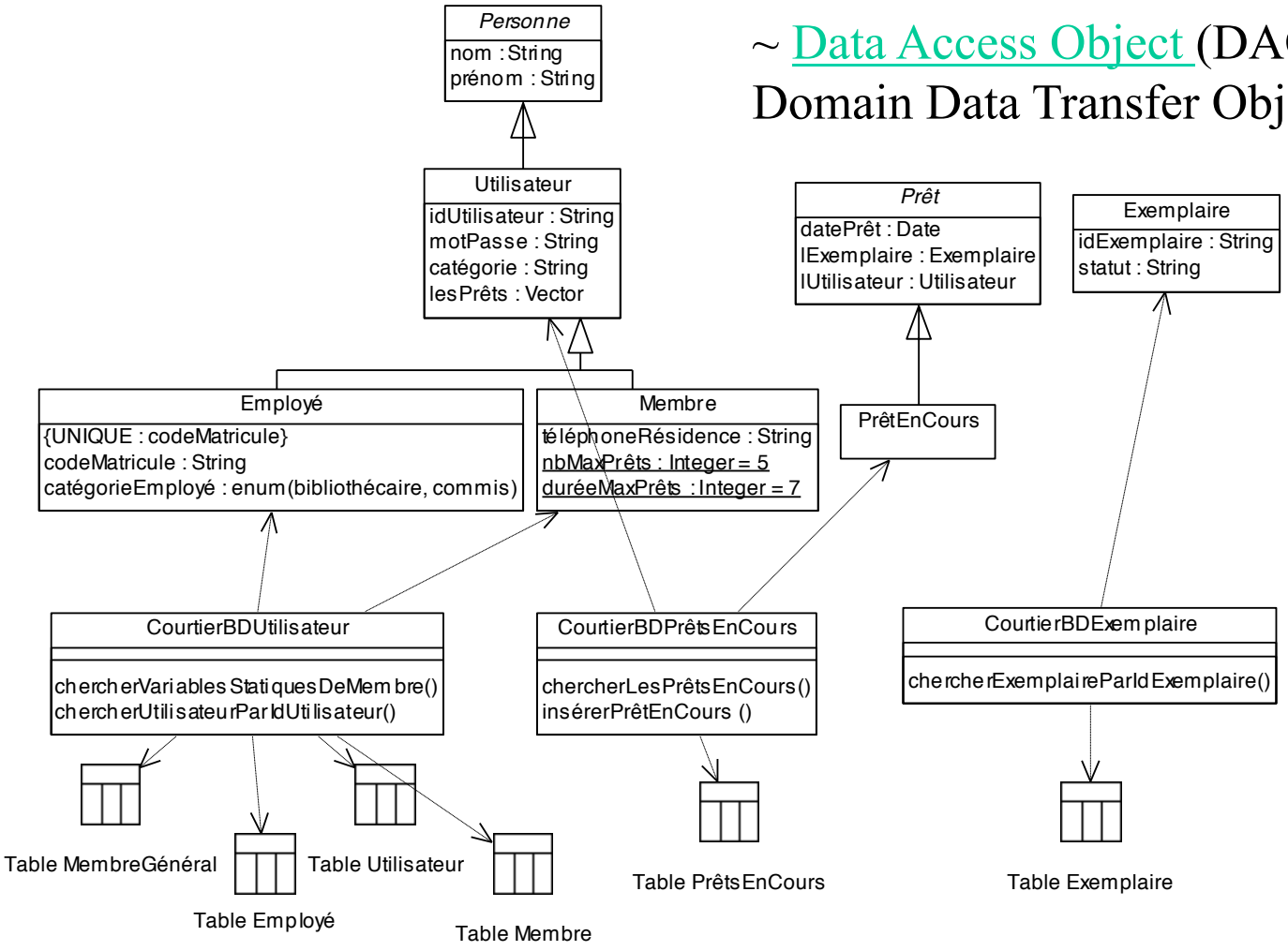
Exemple: enregistrement de prêts

- Classes métier:



Patron courtier BD et tables BD

~ Data Access Object (DAO) + Domain Data Transfer Object



Dématérialisation d'un utilisateur

```
package ExemplesJDBC.GererPrets;
import java.sql.*;
import java.util.*;

public class CourtierBDUtilisateur{
    private Connection uneConnection;
    // Constructeur pour connexion passée par le créateur
    public CourtierBDUtilisateur(Connection laConnection){
        this.uneConnection = laConnection;
    }
…
public Utilisateur chercherUtilisateurParIdUtilisateur(String idUtilisateur)
    // Retourne un objet Membre ou un Employé selon le cas
    throws Exception {
        // Verrouillage de l'utilisateur pour sérialisabilité (Oracle mode READ COMMITTED)
        PreparedStatement unEnoncéSQL = uneConnection.prepareStatement
        ("SELECT motPasse,nom,prénom,catégorieUtilisateur,téléphoneRésidence,codeMatricule,catégorieEmployé "+
        "FROM Membre m, Utilisateur u, Employé e WHERE u.idUtilisateur = ? AND "+
        "u.idUtilisateur = m.idUtilisateur (+) AND "+
        "u.idUtilisateur = e.idUtilisateur (+) FOR UPDATE");
        unEnoncéSQL.setString(1,idUtilisateur);
        ResultSet résultatSelect = unEnoncéSQL.executeQuery();
        if (résultatSelect.next ()){
            if (résultatSelect.getString ("catégorieUtilisateur").equals("membre")){
                Membre leMembre = new Membre(
                    idUtilisateur,résultatSelect.getString ("motPasse"),
                    résultatSelect.getString ("nom"),résultatSelect.getString ("prénom"),
                    résultatSelect.getString ("catégorieUtilisateur"),
                    résultatSelect.getString ("téléphoneRésidence"));
                unEnoncéSQL.close();
                return leMembre;
            }
            else{// Pas un Membre, doit être un Employé
                Employé lEmployé = new Employé(idUtilisateur,résultatSelect.getString ("motPasse");
…
```

Matérialisation d'un *PrêtEnCours*

```
package ExemplesJDBC.GererPrets;
import java.sql.*;
import java.util.*;

public class CourtierBDPrêtEnCours {
    private Connection uneConnection;
…
    public void insérerPrêtEnCours (PrêtEnCours unPrêtEnCours)
    // Matérialise un nouveau PrêtEnCours dans la BD
        throws Exception, SQLException {
        PreparedStatement unEnoncéSQL = uneConnection.prepareStatement
        ("INSERT INTO PrêtEnCours(idExemplaire,idUtilisateur)"+
        "VALUES(?,?)");
        unEnoncéSQL.setString(1,unPrêtEnCours.getIdExemplaire());
        unEnoncéSQL.setString(2,unPrêtEnCours.getIdUtilisateur());
        //NB la date est générée automatiquement par le serveur de BD (default sysdate)
        // et le statut est modifié par un TRIGGER
        int n = unEnoncéSQL.executeUpdate();
        unEnoncéSQL.close();
        if (n != 1){throw new Exception("Insertion dans PrêtEnCours a échoué");}
    }
}
```

Classe de contrôle *ControleEnregistrerPrets*

```
package ExemplesJDBC.GererPrets;
/* Classe de contrôle simple pour EnregistrerPrêts
 * Interface à l'utilisateur minimaliste
 * NB Vérifie les conditions de prêt et le statut de l'exemplaire au niveau du programme client
 */
import java.sql.*;
import javax.swing.JOptionPane;
import java.util.*;

class ControleEnregistrerPrets {
    public static void main (String args []) throws Exception {
        // Création d'une Connection globale pour l'application
        UsineConnection uneUsineConnection = new UsineConnection();
        Connection uneConnection = uneUsineConnection.getConnectionSansAutoCommit(
            "oracle.jdbc.driver.OracleDriver",
            "jdbc:oracle:thin:@localhost:1521:ora817i",
            "clerat","oracle");
        try{
            // Dématérialiser l'Utilisateur et ses PrêtsEnCours
            Membre unMembre = null;
            String idUtilisateur =
                JOptionPane.showInputDialog("Entrez l'identificateur de l'utilisateur: ");

            // Création du courtier et dématérialisation de l'Utilisateur
            CourtierBDUtilisateur unCourtierBDUtilisateur = new CourtierBDUtilisateur(uneConnection);
            Utilisateur unUtilisateur =
                unCourtierBDUtilisateur.chercherUtilisateurParIdUtilisateur(idUtilisateur);

            //Création du courtier et dématérialisation des PrêtsEnCours
            CourtierBDPrêtEnCours unCourtierBDPrêtEnCours = new CourtierBDPrêtEnCours(uneConnection);
            unCourtierBDPrêtEnCours.chercherLesPrêtsEnCours(unUtilisateur);
            JOptionPane.showMessageDialog(null,
            "Nombre de prêts en cours :" + unUtilisateur.getNbPrêtsEnCours());
```

…

Granularité des courtiers

- Certaines opérations nécessitent la dématérialisation de plusieurs objets en même temps
  - Ex: recherche d'un utilisateur et de tous ses prêts
- Option 1: un courtier par classe / objet
  - Meilleure cohésion
  - Permet un contrôle plus fin le moment de dématérialisation
  - Peut exiger un grand nombre d'appels important au serveur
- Option 2: un courtier pour plusieurs classes / objets
  - Limite le nombre d'appels au serveurs
  - Peut extraire des données inutilement
- On cherche le meilleur compromis entre 1 et 2

Problème d'accès concurrents

Transaction T_1	Transaction T_2

`SQL> SELECT * FROM PrêtEnCours`
`  2  WHERE idUtilisateur = 2;`
 	
`IDEXEMPLAI DATEPRÊT IDUTILISAT`
`---------- -------- ----------`
`3          02-03-21 2`	

	`SQL> SELECT * FROM PrêtEnCours`
	`  2  WHERE idUtilisateur = 2;`
	
	`IDEXEMPLAI DATEPRÊT IDUTILISAT`
	`---------- -------- ----------`
	`3          02-03-21 2`

`SQL> INSERT INTO`
`  2  PrêtEnCours(idExemplaire,idUtilisateur)`
`  3  VALUES(4,2);`
 	
`1 row created.`
 	
`SQL> COMMIT;`
 	
`Commit complete.`	

	`SQL> INSERT INTO`
	`  2  PrêtEnCours(idExemplaire,idUtilisateur)`
	`  3  VALUES(5,2);`
	
	`1 row created.`
	
	`SQL> COMMIT;`
	
	`Commit complete.`
	
	Dépasse la limite du nombre de prêts (maximum 2)

Stratégies de gestion de la concurrence

- Verrouillage pessimiste (*pessimist locking*)
  - On verrouille les ressources de la transaction jusqu'à ce qu'elle termine (*commit*)
  - Au niveau de la BD:
    - SELECT ... FOR UPDATE, LOCK TABLE, etc.
  - Au niveau de l'application:
    - Fonctionnalités JDO, Hibernate, etc.
  - Problématique si la transaction dépend du temps de réponse/réflexion de l'utilisateur (*think time*)

Stratégies de gestion de la concurrence

- Verrouillage optimiste (*optimist locking*)
  - Se fait au niveau de l'application
  - On ajoute à chaque table (classe) une colonne contenant l'instant de la dernière mise à jour
  - On annule une transaction si la donnée à mettre à jour a été modifiée depuis la dernière lecture (lecture impropre)
  - Permet d'accélérer grandement le traitement des transactions
  - Ne garantie pas l'intégrité des données
  - Populaire dans les applications Web

Autres stratégies

- Limiter la durée des transactions
  - Découper une longue transaction en plusieurs courtes transactions
  - Utiliser *autocommit*
- Vérifier les contraintes par le serveur
  - Ex: TRIGGER Oracle
- Réduire l'interactivité
  - Limiter la dépendance au temps de réflexion
  - Employer seulement en dernier recours

Vérification des contraintes par le serveur

Transaction T_1	Transaction T_2
SQL> SELECT * FROM PrêtEnCours 2 WHERE idUtilisateur = 2; IDEXEMPLAI DATEPRÊT IDUTILISAT ---------- -------- ---------- 3 02-03-21 2	
	SQL> SELECT * FROM PrêtEnCours 2 WHERE idUtilisateur = 2; IDEXEMPLAI DATEPRÊT IDUTILISAT ---------- -------- ---------- 3 02-03-21 2

SQL> INSERT INTO
  2 PrêtEnCours(idExemplaire,idUtilisateur)
  3 VALUES(4,2);
 	
1 row created.
 	
SQL> COMMIT;
 	
Commit complete.	

	SQL> INSERT INTO
	  2 PrêtEnCours(idExemplaire,idUtilisateur)
	  3 VALUES(5,2);
	
	INSERT INTO
	PrêtEnCours(idExemplaire,idUtilisateur)
	  *
	ERROR at line 1:
	ORA-20101: le nombre maximal d'emprunts est atteint
	ORA-06512: at "CLERAT.BIPRÊTENCOURSVÉRIFIER", line 59
	ORA-04088: error during execution of trigger 'CLERAT.BIPRÊTENCOURSVÉRIFIER'
	
	Un TRIGGER empêche l'insertion fautive.

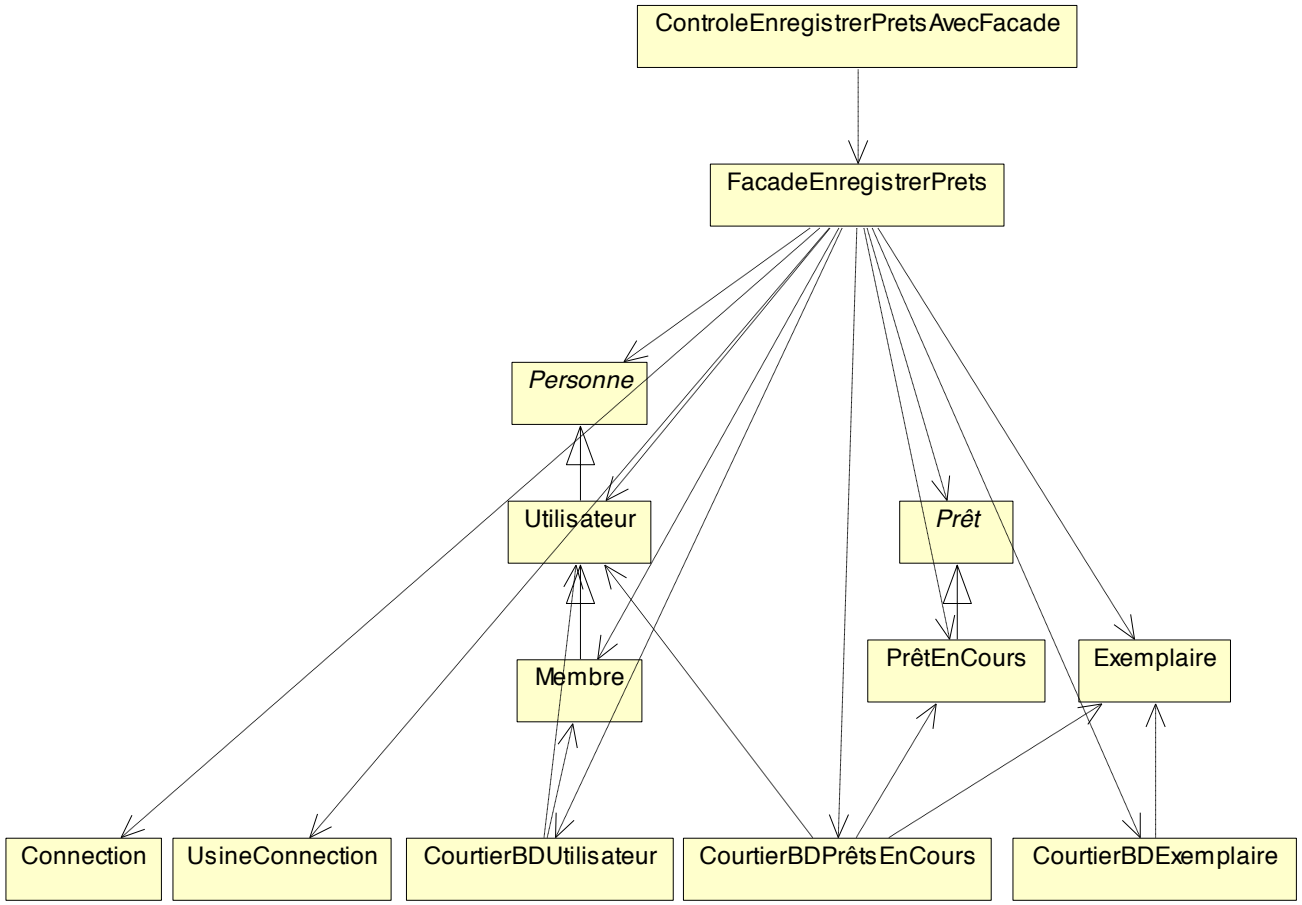
Gestion de la sécurité

- Deux approches
  - Par le SGBD:
    - La connexion à la BD se fait avec le login de l'utilisateur
    - On utilise les mécanismes du SGBD pour contrôler l'accès
  - Par l'application:
    - La connexion à la BD se fait avec un login générique
    - L'accès est géré soit par la couche de contrôle ou la couche de service (ex: EJB, Spring, etc.)

Patron de conception façade

- Fournit une interface simple à un sous-système complexe
- Masque les détails de la gestion de la persistance (connexion JDBC, matérialisation/dématérialisation, etc.) à la couche de contrôle
- Souvent employé dans les applications multi-niveaux (*multi-tier*)

Façade pour les services de la couche application



Objet de transfert de données (OTD)

- Patron *Data Transfer Objet* (DTO) ou *Value Object* (VO)
- Réduit le nombre de transferts entre deux sous-systèmes (ex: de la couche métier à la couche de contrôle) en regroupant un ensemble de données provenant de plusieurs objets
- Les OTD ne possèdent que des attributs (aucune méthode)

Exemple d'utilisation du patron OTD

```
public class FacadeEnregistrerPrêts
{
  …
  public OTDUtilisateurPrets chercherOTDUtilisateurPrets(String idUtilisateur) throws Exception
  {
    // Création du courtier et dématérialisation de l'Utilisateur
    CourtierBDUtilisateur unCourtierBDUtilisateur = new CourtierBDUtilisateur(uneConnection);
    unUtilisateur =
        unCourtierBDUtilisateur.chercherUtilisateurParIdUtilisateur(idUtilisateur);

    //Création du courtier et dématérialisation des PrêtsEnCours
    unCourtierBDPretEnCours = new CourtierBDPrêtEnCours(uneConnection);
    unCourtierBDPretEnCours.chercherLesPrêtsEnCours(unUtilisateur);

    // Retourner l'objet de transfert de données OTDUtilisateurPrets
    if (unUtilisateur instanceof Membre){
      unCourtierBDUtilisateur.chercherVariablesStatiquesDeMembre();
      Membre unMembre = (Membre)unUtilisateur;
      return new OTDUtilisateurPrets(
                unMembre.getCatégorie(),
                unMembre.conditionsPrêtAcceptées(),
                unMembre.getNbPrêtsEnCours(),
                Membre.getNbMaxPrêts(),
                unMembre.getNbRetards());
    } else {
      return new OTDUtilisateurPrets(
        unUtilisateur.getCatégorie(),true,unUtilisateur.getNbPrêtsEnCours(),0,0);
    }
  }
  …
}
```

Problème d'utiliser JDBC

- Le développement et le maintient de code SQL peut être complexe
- La portabilité est réduite par le code SQL spécifique au SGBD (ex: séquences Oracle)
- Problèmes spécifiques à l'utilisation de EJB
  - Longs cycles édition-compilation-debug parce que les EJB résident du côté serveur
  - Ajoute beaucoup de code (interfaces *Home*, *Remote*, etc.)
  - Encourage l'écriture de code procédurale monolithique

Architecture pour la persistance transparente

- La persistance objet est gérée par un service externe aux couches métier et contrôle, de manière transparente
- Le client travaille avec de simples objets Java (*Plain Old Java Objects – POJOs*)
- Deux types d'architectures:
  - Object / relationnal mapping ORM (ex: JDO, Hibernate, EJB3)
  - Object / SQL mapping (ex: iBatis/myBatis)

Frameworks ORM

- Mappage du domaine d'application à BD relationnelle
  - Classes ⇔ tables
  - Attributs ⇔ colonnes
  - Relations inter-objets (1-1, 1-plusieurs, etc.)
  - Héritage
- Gestion du cycle de vie des objets
- Gestion de l'identité des objets

Frameworks ORM : services

- Mappage déclaratif entre modèle objet et schéma de BD relationnelle
- API pour création, lecture, modification et destruction d'objets
- Langage de requête (objet) pour trouver efficacement les objets basé sur des critères
- Support pour la gestion de transactions
- Chargement hâtif ou tardif (eager/lazy loading)
- Gestion de cache pour limiter les accès à la BD
- Manipulation d'objets "détachés"

Frameworks ORM : avantages et limitations

- Avantages:
  - Productivité:
    - Aucun code SQL / JDBC à écrire
    - Le découplage de la BD facilite les tests et la maintenance
  - Performance:
    - Gestion avancée de la cache, chargement hâtif, etc.
  - Portabilité:
    - Code SQL généré spécifique au SGDB employé (ex: Oracle)
- Limitations:
  - Pas toujours évident de mapper le domaine d'application au schéma relationnel (ex: systèmes legacy, dénormalisation, etc.)
  - Pas toujours le contrôle sur la définition et l'accès au schéma relationnel (ex: accès par procédures stockées seulement)

Frameworks de mappage SQL-objet

- Utilisé dans les cas où on ne contrôle pas le schéma de la BD (ex: code SQL ultra-optimisé, accès restreint)
- Élimine la dépendance de l'application à la couche de transparente en encapsulant les accès à la BD dans un fichier de configuration externe (XML)
- Le code JDBC est géré par le framework
- Ressemble aux procédures stockées, mais les paramètres et valeurs de retour sont des objets du domaine d'application

Frameworks de mappage SQL-objet

- Exemple iBatis/myBatis
  - Définition:

```
<select id="getLivre"
        parameterClass="java.lang.String"
        resultClass="Livre">
  SELECT ISBN, TITRE, ANNEE_PARUTION, NOM_EDITEUR
  FROM LIVRE l, EDITEUR e
  WHERE e.ID_EDITEUR = l.ID_EDITEUR AND
        ISBN = #value#
</select>
```

  - Appel:

```
Livre leLivre = (Livre) sqlMapClient.queryForObject(
        "getLivre", "1-111-1111");
```

Frameworks de mappage SQL-objet

- Tutoriel iBatis – SQL Maps 2.0

http://ibatis.apache.org/docs/java/pdf/iBATIS-SqlMaps-2-Tutorial_en.pdf (EN)

http://ibatis.apache.org/docs/java/pdf/iBATIS-SqlMaps-2-Tutorial_fr.pdf (FR)

- Exemple JPetStore 5

http://apache.sunsite.ualberta.ca/ibatis/binaries/ibatis.java/JPetStore-5.0.zip

Hibernate

- Outil de persistence transparente Java
- Open source
- Améliorations vs EJB 2 et JDO 1
  - Plain Old Java Objects (POJO)
    - Héritage, associations par attributs Java
  - Langage de requête HQL plus proche de SQL
  - Pas de manipulation de code (introspection Java)
- Support de l'API de persistance Java de la nouvelle norme EJB3 (JSR220)

POJO dans Hibernate

- Simple objet Java
- N'hérite pas d'interfaces complexes (ex: EJB 2)
- Facilite les tests et portables à d'autres applications
- Nécessite:
  - Constructeur sans paramètre
  - Définition d'un attribut identifiant (ID) si clé artificielle
  - Définition de méthodes d'accès (*get*/*set*) pour les attributs
  - La méthode *setId* doit être privée (Hibernate gère l'assignation de valeurs aux Ids)
  - Les collections doivent être définies comme des interfaces (ex: *Set* au lien de *HashSet*)

Exemple: classe Editeur

```
public class Editeur
{
   // Les attributs de la classe
   private Integer id;
   private String nom;
   private String ville ;

   // La collection des livres de l'éditeur
   private Set lesLivres = new HashSet();

   // Constructeur vide REQUIS
   public Editeur(){}

   // Lecteurs
   public Integer getId(){return id;}
   public String getNom (){return nom;}
   public String getVille(){return ville;}
   public Set getLesLivres(){return lesLivres;}

   // Modifieurs
   private void setId(Integer id) {this.id = id;}
   public void setNom (String nomEditeur){this.nom = nom;}
   public void setVille(String ville){this.ville = ville;}
   public void setLesLivres(Set lesLivres){this.lesLivres = lesLivres;}
}
```

Exemple: classe Livre

```
public class Livre
{
   // Les attributs de la classe
   private String ISBN;
   private String titre;
   private  int anneeParution ;

   // Pour l'association avec Editeur
   private Editeur editeur;

   // Constructeur vide requis
   public Livre(){}

   // Lecteurs
   public String getISBN(){return ISBN;}
   public String getTitre(){return titre;}
   public int getAnneeParution(){return anneeParution;}
   public Editeur getEditeur (){return editeur;}

   // Modifieurs
   public void setISBN(String ISBN){this.ISBN = ISBN;}
   public void setTitre(String titre){this.titre = titre;}
   public void setAnneeParution(int anneeParution){this.anneeParution = anneeParution;}
   public void setEditeur(Editeur editeur){this.editeur = editeur;}
}
```

Exemple: schéma relationnel

```
CREATE TABLE EDITEUR
(ID_EDITEUR                 INTEGER     NOT NULL,
 NOM_EDITEUR                VARCHAR(20) NOT NULL,
 VILLE                      VARCHAR(20) NOT NULL,
 PRIMARY KEY (ID_EDITEUR)
)

CREATE TABLE LIVRE
(ISBN                       CHAR(13)    NOT NULL,
 TITRE                      VARCHAR(50) NOT NULL,
 ANNEE_PARUTION             NUMBER(4)   NOT NULL,
 ID_EDITEUR                 INTEGER     NOT NULL,
 PRIMARY KEY (ISBN),
 FOREIGN KEY (ID_EDITEUR) REFERENCES EDITEUR
)
```

Mappage de la classe Editeur

- Mappage de la classe à une table:

```
<hibernate-mapping>
    <class name="Editeur" table="EDITEUR">
     ...
    </class>
</hibernate-mapping>
```

- Mappage de l'identifiant:

```
<class name="Editeur" table="EDITEUR">
   <id name="id" column="ID_EDITEUR"/>
      <generator class="native"/>
   </id>
   ...
</class>
```

  - Stratégie de génération (*generator class*)
    - *identity* : DB2, MySQL, MS SQL Server, Sybase, HypersonicSQL
    - *sequence* : DB2, PostgreSQL, Oracle
    - *hilo* : algorithm hi/lo
    - *native* : choisit la meilleure stratégie pour le SGBD employé
    - *select* : si les clés sont générées automatiquement (ex: TRIGGER)
    - *assigned* : ID géré par une clé naturelle (défaut si aucune stratégie spécifiée)

Mappage de la classe Editeur

- Mappage des attributs:

```
<class name="Editeur" table="EDITEUR">
   ...
   <property name="nom" column="NOM_EDITEUR"/>
   <property name="ville" column="VILLE"/>
   ...
</class>
```

  - L'attribut peut être public ou privée
  - Conversion du type de l'attribut:
    - Implicite pour les types simples, sinon il faut spécifier le type:

```
<property name="datePublication"
    type="TIMESTAMP"
    column="DATE_PUBLICATION"/>
```

  - Accès par champs (*field*) ou par méthode (*property*):
    - Accès par méthode (défaut) favorise l'encapsulation, mais demande de définir les get/set
    - Pour que l'accès se fasse directement par les champs:

```
<property name="nom"
   column="NOM_EDITEUR"
   access="field"/>
```

Mappage de la classe Editeur

- Mappage des relations:

```
<class name="Editeur" table="EDITEUR">
   ...
   <set name="lesLivres" inverse="true" cascade="all">
       <key column="ID_EDITEUR"/>
       <one-to-many class="Livre"/>
   </set>
   ...
</class>
```

  - Collections supportées: *set*, *list*, *map*, *bag*, *array*, *primitive-array*
  - Relations peuvent être bi-directionnelles
    - *inverse* : indique que l'accès se fait par la relation inverse allant de Livre à Editeur
  - Propagation des sauvegardes/suppressions des objets référencés avec *cascade*:
    - *none* : aucune propagation
    - *save-update* : lors de la sauvegarde de l'objet référençant
    - *delete* : lors de la suppression de l'objet référençant
    - *all*: lors de la sauvegarde ou la suppression de l'objet référençant

Editeur.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="Editeur" table="EDITEUR">
        <id name="id" column="ID_EDITEUR"/>
            <generator class="native"/>
        </id>
        <property name="nom" column="NOM_EDITEUR"/>
        <property name="ville" column="VILLE"/>
            <set name="lesLivres" inverse="true" cascade = "all">
                <key column="ID_EDITEUR"/>
                <one-to-many class="Livre"/>
            </set>
    </class>
</hibernate-mapping>
```

Livre.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="Livre" table="LIVRE">
        <id name="ISBN" column="ISBN"/>
        <property name="titre" column="TITRE"/>
        <property name="anneeParution" column="ANNEE_PARUTION"/>
        <many-to-one name="editeur"
            class="Editeur"
            column="ID_EDITEUR"
            not-null="true"/>
    </class>
</hibernate-mapping>
```

Fichier de configuration *hibernate.cfg.xml*

```
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">
            oracle.jdbc.driver.OracleDriver
        </property>
        <property name="hibernate.connection.url">
            jdbc:oracle:thin:@localhost:1521:ORCL
        </property>
        <property name="hibernate.connection.username">godin</property>
        <property name="hibernate.connection.password">oracle</property>
        <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
        <property name="hibernate.c3p0.min_size">5</property>
        <property name="hibernate.c3p0.max_size">20</property>
        <property name="hibernate.c3p0.timeout">300</property>
        <property name="hibernate.c3p0.max_statements">50</property>
        <property name="hibernate.c3p0.idle_test_period">3000</property>
        <!-- SQL to stdout logging -->
        <property name="show_sql">true</property>
        <property name="format_sql">true</property>
        <property name="use_sql_comments">true</property>

        <mapping resource="Editeur.hbm.xml"/>
        <mapping resource="Livre.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Singleton qui démarre Hibernate et fournit l'objet SessionFactory

```
public class HibernateUtil {
  private static SessionFactory sessionFactory;
  static {
    try {
       sessionFactory = new Configuration().configure().buildSessionFactory();
    } catch (Throwable ex) {
       throw new ExceptionInInitializerError(ex);
    }
  }
  public static SessionFactory getSessionFactory() {
      return sessionFactory;
  }
  public static void shutdown() {
      // Ferme les antémémoires et les bassins (pool) de connexions
      getSessionFactory().close();
  }
}
```

Session Hibernate qui insère un éditeur et un livre

```
Session uneSession = HibernateUtil.getSessionFactory().openSession();
uneSession.beginTransaction();

Editeur unEditeur = new Editeur();
unEditeur.setNom("Addison-Wesley");
unEditeur.setVille("Reading, MA");

Livre unLivre = new Livre();
unLivre.setISBN("1-111-1111");
unLivre.setTitre("SGBD");
unLivre.setAnneeParution(2000);
unLivre.setEditeur(unEditeur);

unEditeur.getLesLivres().add(unLivre);

// Rendre unEditeur persistant
uneSession.save(unEditeur);
uneSession.getTransaction().commit();
uneSession.close();
```

Persistance par référence des livres associés (voir attribut cascade dans fichier de mappage *Editeur.hbm.xml*)

Session Hibernate qui lit les données de la première session et les affiche

```
uneSession = HibernateUtil.getSessionFactory().openSession();
uneSession.beginTransaction();
List lesEditeurs = uneSession.createQuery(
    "FROM Editeur e
     WHERE e.ville = 'Paris' OR e.ville = 'Longueuil'
     ORDER BY e.nom ASC").list();

System.out.println( lesEditeurs.size() + " éditeurs trouvés:" );

for ( Iterator iterEditeurs = lesEditeurs.iterator(); iterEditeurs.hasNext(); ) {
  Editeur unEditeurCharge = (Editeur) iterEditeurs.next();
  System.out.println("Éditeur:" + unEditeurCharge.getNom() );
  System.out.println("Livres de l'éditeur:" );

  for ( Iterator iterLivres = unEditeurCharge.getLesLivres().iterator();
        iterLivres.hasNext(); ) {
    Livre unLivreCharge = (Livre) iterLivres.next();
    System.out.println("   " + unLivreCharge.getTitre() );
  }
}
uneSession.getTransaction().commit();
uneSession.close();
```

Requêtes dans Hibernate

- Requête HQL (dialecte objets)

```
Query maRequete = uneSession.createQuery(
        ”FROM Livre l WHERE l.ISBN = '1-111-1111' ");
List lesLivres = maRequete.list();
```

- Requête SQL (dialecte de la BD)

```
Query maRequete = uneSession.createSQLQuery(
        ”SELECT * FROM Livre l
          WHERE l.ISBN = '1-111-1111' ");
List lesLivres = maRequete.list();
```

- Critère de sélection (dialecte programmatif)

```
// Tous les objets de type Livre
Criteria monCritere = uneSession.createCriteria(Livre.class);
List lesLivres = monCritere.list();
```

Session Hibernate qui modifie le titre de ISBN: 1-111-1111

```
uneSession.beginTransaction();
List lesLivres = uneSession.createQuery(
        ”FROM Livre l WHERE l.ISBN = '1-111-1111' ").list();
unLivre = (Livre)lesLivres.iterator().next();
unLivre.setTitre("UnNouveauTitre");
uneSession.getTransaction().commit();
```

- Pas besoin de faire un *save* ou un *update* explicite après la modification de unLivre car c'est un objet persistant
- Par contre, il faut le faire si l'objet est dans un état détaché

```
uneSession.beginTransaction();
List lesLivres = uneSession.createQuery(
    ”FROM Livre l WHERE l.ISBN = '1-111-1111' ").list();
unLivre = (Livre)lesLivres.iterator().next();
uneTransaction.commit();

// unLivre est maintenant en état "détaché"
unLivre.setTitre("UnNouveauTitre");
uneAutreSession = HibernateUtil.getSessionFactory().openSession();
uneAutreSession.beginTransaction();
uneAutreSession.update(unLivre);           // unLivre est mis a jour
uneAutreSession.getTransaction().commit();
```

Session Hibernate qui supprime un livre

```
uneTransaction = uneSession.beginTransaction();
lesLivres = uneSession.createQuery("
  FROM Livre l WHERE l.ISBN = '1-111-1111' ").list();
unLivre = (Livre)lesLivres.iterator().next();
uneSession.delete(unLivre);
uneTransaction.commit();
```

- La suppression du livre ne sera pas propagée à son éditeur car la propriété *cascade* n'est pas *all* ou *delete*
- Par contre, la suppression de l'éditeur entraînerait celle de ses livres

Sauvegarde intelligence

- Par défaut, Hibernate n'écrit sur la BD que les données qui ont été modifiées (*dirty checking*)
- Pour un objet simple, la comparaison utilise la **valeur** des attributs de l'objet
- Pour une collection d'objets, la comparaison est faite à l'aide des références de ces objets
- À éviter:

```
...
public List getLesLivres() {
   return Arrays.asList(lesLivres);}
...
```

Traduction de relations

- Association plusieurs à 1
- Association 1 à 1
- Association 1 à plusieurs
- Association plusieurs à plusieurs
- Composition 1 à plusieurs

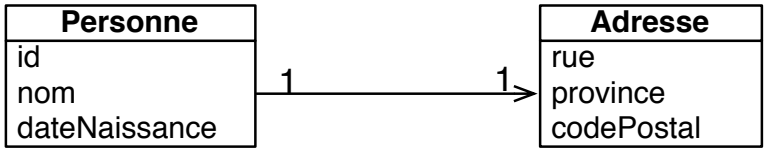
Association plusieurs à 1

```
<class name="Personne" table="PERSONNE">
   <id name="id" column="ID_PERSONNE">
      <generator class="native"/>
   </id>
   ...
   <many-to-one name="adresse"
      column="ID_ADRESSE"
      not-null="true"/>
</class>

<class name="Adresse" table="ADRESSE">
   <id name="id" column="ID_ADRESSE">
      <generator class="native"/>
   </id>
   ...
</class>
```


```
CREATE TABLE PERSONNE
(ID_PERSONNE INTEGER NOT NULL PRIMARY KEY,
 ID_ADRESSE  INTEGER NOT NULL,
 FOREIGN KEY (ID_ADRESSE) REFERENCES ADRESSE
 ...)

CREATE TABLE ADRESSE
(ID_ADRESSE INTEGER NOT NULL PRIMARY KEY,
 ...)
```

Association 1 à 1

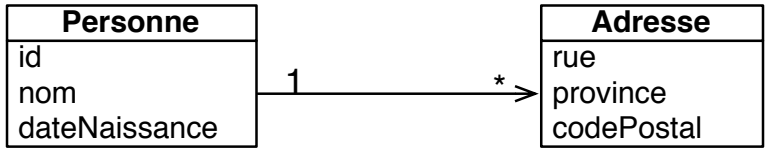
```
<class name="Personne" table="PERSONNE">
   <id name="id" column="ID_PERSONNE">
      <generator class="native"/>
   </id>
   ...
   <many-to-one name="adresse"
      column="ID_ADRESSE"
      unique="true"
      not-null="true"/>
</class>

<class name="Adresse" table="ADRESSE">
   <id name="id" column="ID_ADRESSE">
      <generator class="native"/>
   </id>
   ...
</class>
```




```
CREATE TABLE PERSONNE
(ID_PERSONNE INTEGER NOT NULL PRIMARY KEY,
 ID_ADRESSE  INTEGER NOT NULL UNIQUE,
 FOREIGN KEY (ID_ADRESSE) REFERENCES ADRESSE
 ...)

CREATE TABLE Address
(ID_ADRESSE INTEGER NOT NULL PRIMARY KEY,
 ...)
```

Association 1 à plusieurs

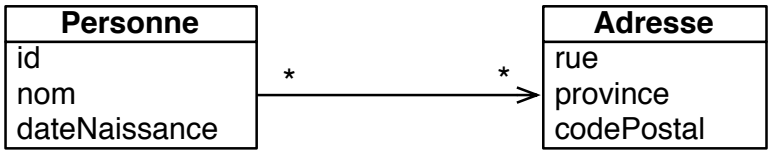
```
<class name="Personne" table="PERSONNE">
   <id name="id" column="ID_PERSONNE">
      <generator class="native"/>
   </id>
   ...
   <set name="lesAdresses">
      <key column="ID_PERSONNE"
         not-null="true"/>
      <one-to-many class="Adresse"/>
   </set>
</class>

<class name="Adresse" table="ADRESSE">
   <id name="id" column="ID_ADRESSE">
      <generator class="native"/>
   </id>
   ...
</class>
```




```
CREATE TABLE PERSONNE
(ID_PERSONNE INTEGER NOT NULL PRIMARY KEY,
 ...)

CREATE TABLE ADRESSE
(ID_ADRESSE  INTEGER NOT NULL PRIMARY KEY,
 ID_PERSONNE INTEGER NOT NULL,
 FOREIGN KEY (ID_PERSONNE) REFERENCES PERSONNE
 ...)
```

Association plusieurs à plusieurs (unidirectionnelle)

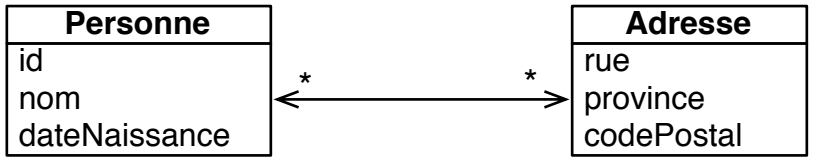
```
<class name="Personne" table="PERSONNE">
   <id name="id" column="ID_PERSONNE">
      <generator class="native"/>
   </id>
   <set name="lesAdresses" table="PERSONNE_ADRESSE">
      <key column="ID_PERSONNE"/>
      <many-to-many column="ID_ADRESSE"
         class="Adresse"/>
   </set>
</class>

<class name="Adresse" table="ADRESSE">
   <id name="id" column="ID_ADRESSE">
      <generator class="native"/>
   </id>
</class>
```




```
CREATE TABLE PERSONNE
(ID_PERSONNE INTEGER NOT NULL PRIMARY KEY,
 ...)

CREATE TABLE ADRESSE
(ID_ADRESSE INTEGER NOT NULL PRIMARY KEY,
 ...)

CREATE TABLE PERSONNE_ADRESSE
(ID_PERSONNE INTEGER NOT NULL,
 ID_ADRESSE  INTEGER NOT NULL,
 FOREIGN KEY(ID_PERSONNE) REFERENCES PERSONNE,
 FOREIGN KEY(ID_ADRESSE) REFERENCES ADRESSE)
```

Association plusieurs à plusieurs (bidirectionnelle)

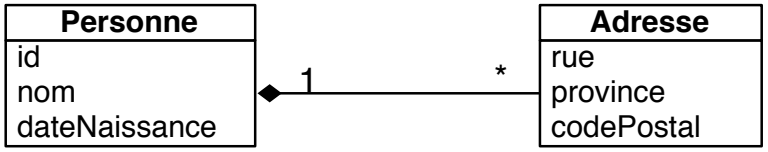
```
<class name="Personne" table="PERSONNE">
   <id name="id" column="ID_PERSONNE">
      <generator class="native"/>
   </id>
   <set name="lesAdresses" table="PERSONNE_ADRESSE">
      <key column="ID_PERSONNE"/>
      <many-to-many column="ID_ADRESSE"
         class="Adresse"/>
   </set>
</class>

<class name="Adresse" table="ADRESSE">
   <id name="id" column="ID_ADRESSE">
      <generator class="native"/>
   </id>
   <set name="lesPersonnes" inverse="true" table="PERSONNE_ADRESSE">
      <key column="ID_ADRESSE"/>
      <many-to-many column="ID_PERSONNE"
         class="Personne">
   </set>
</class>
```




```
CREATE TABLE PERSONNE
(ID_PERSONNE INTEGER NOT NULL PRIMARY KEY,
 ...)

CREATE TABLE ADRESSE
(ID_ADRESSE INTEGER NOT NULL PRIMARY KEY,
 ...)

CREATE TABLE PERSONNE_ADRESSE
(ID_PERSONNE INTEGER NOT NULL,
 ID_ADRESSE  INTEGER NOT NULL,
 FOREIGN KEY(ID_PERSONNE) REFERENCES PERSONNE,
 FOREIGN KEY(ID_ADRESSE) REFERENCES ADRESSE)
```

Composition 1 à plusieurs

```
<class name="Personne" table="PERSONNE">
   <id name="id" column="ID_PERSONNE">
      <generator class="native"/>
   </id>
   ...
   <set name="lesAddresses" table="PERSONNE_ADRESSE">
      <key column="ID_PERSONNE"/>
      <composite-element class="Adresse">
         <property name="rue" column="RUE"/>
         <property name="province" column="PROVINCE"/>
         <property name="codePostal" column="CODE_POSTAL"/>
      </composite-element>
   </set>
</class>
```



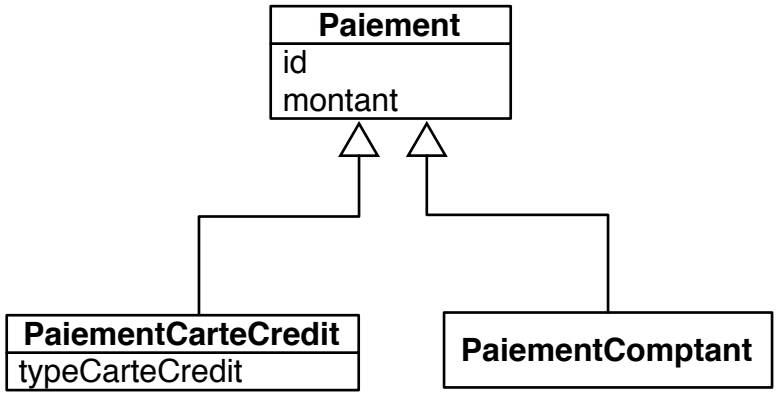

```
CREATE TABLE PERSONNE
(ID_PERSONNE INTEGER NOT NULL PRIMARY KEY,
 ...)

CREATE TABLE PERSONNE_ADRESSE
(ID_PERSONNE INTEGER NOT NULL PRIMARY KEY,
 RUE          VARCHAR2(50),
 PROVINCE     VARCHAR2(15),
 CODE_POSTAL VARCHAR2(6),
 FOREIGN KEY (ID_PERSONNE) REFERENCES PERSONNE)
```

Traduction de la spécialisation (héritage)

- Trois stratégies (*voir cours conception*):
  1. Une table pour la hiérarchie:
     - Toutes les sous-classes sont dans la même table
     - Le type de la sous-classe est déterminé par une colonne discriminant
  2. Une table par sous-classe:
     - Chaque sous-classe possède sa propre table contenant ses attributs ajoutés et une référence vers la table parent
  3. Une table par classe concrète:
     - Chaque sous-classe (sauf les classes abstraites et interfaces) possède sa propre table contenant ses attributs ajoutés
     - Les attributs de la classe parent sont recopiés dans les tables des sous-classes

Exemple de hiérarchie

- Modes de paiement



Une table pour la hiérarchie

```
<class name="Paiement" table="PAIEMENT">
   <id name="id" type="long" column="ID_PAIEMENT">
      <generator class="native"/>
   </id>
   <discriminator column="TYPE_PAIEMENT" type="string"/>
   <property name="montant" column="MONTANT"/>
   ...
   <subclass name="PaiementCarteCredit" discriminator-value="CREDIT">
      <property name="typeCarteCredit" column="TYPE_CC"/>
      ...
   </subclass>
   <subclass name="PaiementComptant" discriminator-value="COMPTANT">
      ...
   </subclass>
</class >
```

- **Note**: Ne pas utiliser *not-null=true* pour les attributs dérivés

```
CREATE TABLE PAIEMENT
(ID_PAIEMENT   INTEGER      NOT NULL PRIMARY KEY,
 TYPE_PAIEMENT VARCHAR2(10) NOT NULL,
 MONTANT       NUMBER(9,2)  NOT NULL,
 TYPE_CC       VARCHAR2(20)
)
```

Une table par sous-classe

```
<class name="Paiement" table="PAIEMENT">
   <id name="id" type="long" column="ID_PAIEMENT">
      <generator class="native"/>
   </id>
   <property name="montant" column="MONTANT"/>
   ...
   <joined-subclass name="PaiementCarteCredit" table="PAIEMENT_CREDIT">
      <key column="ID_PAIEMENT"/>
      <property name="typeCarteCredit" column="TYPE_CC"/>
      ...
   </joined-subclass>
   <joined-subclass name="PaiementCarteCredit" table="PAIEMENT_COMPTANT">
      <key column="ID_PAIEMENT"/>
      ...
   </joined-subclass>
</class >
```

```
CREATE TABLE PAIEMENT
(ID_PAIEMENT   INTEGER      NOT NULL PRIMARY KEY,
 MONTANT       NUMBER(9,2)  NOT NULL)

CREATE TABLE PAIEMENT_CREDIT
(ID_PAIEMENT   INTEGER      NOT NULL PRIMARY KEY,
 TYPE_CC       VARCHAR2(20) NOT NULL
 FOREIGN KEY(ID_PAIEMENT) REFERENCES PAIEMENT)

CREATE TABLE PAIEMENT_COMPTANT
(ID_PAIEMENT   INTEGER      NOT NULL PRIMARY KEY,
 FOREIGN KEY(ID_PAIEMENT) REFERENCES PAIEMENT)
```

Une table par classe concrète

```
<class name="Paiement" table="PAIEMENT">
   <id name="id" type="long" column="ID_PAIEMENT">
      <generator class="native"/>
   </id>
   <property name="montant" column="MONTANT"/>
   ...
   <union-subclass name="PaiementCarteCredit" table="PAIEMENT_CREDIT">
      <key column="ID_PAIEMENT"/>
      <property name="typeCarteCredit" column="TYPE_CC"/>
      ...
   </union-subclass>
   <union-subclass name="PaiementCarteCredit" table="PAIEMENT_COMPTANT">
      <key column="ID_PAIEMENT"/>
      ...
   </union-subclass>
</class >
```

- **Note**: Pas besoin de redéfinir les attributs de la classe parent dans les sous-classes (ex: *montant*)

```
CREATE TABLE PAIEMENT_CREDIT
(ID_PAIEMENT   INTEGER      NOT NULL PRIMARY KEY,
 MONTANT       NUMBER(9,2)  NOT NULL,
 TYPE_CC       VARCHAR2(20) NOT NULL)

CREATE TABLE PAIEMENT_COMPTANT
(ID_PAIEMENT   INTEGER      NOT NULL PRIMARY KEY,
 MONTANT       NUMBER(9,2)  NOT NULL)
```

Optimisation de la performance

- Verrouillage optimiste (*optimistic locking*)
  - Accélère le traitement de transactions concurrentes en évitant de bloquer les ressources
  - En cas de conflit, la transaction gagnante est celle qui a écrit en premier
  - Verrouillage à l'aide d'une version

```
class Commande {
  ...
  private Long laVersion;
  ...
}
```

```
<class name="Commande" table="COMMANDE">
    <version name="laVersion" column="VERSION"
       access="field"/>
   ...
</class>
```

  - Peut également se faire à l'aide d'une estampille (*timestamp*)
  - Le verrouillage d'une table, d'un attribut ou d'une relation peut être contrôlé à l'aide du paramètre *optimistic-lock*.

Optimisation de la performance

- Chargement hâtif:
  - Charge les objets référencés lorsque l'objet référençant est chargé
  - Exemple: charger les livres d'un éditeur lorsqu'on charge l'éditeur
  - Le chargement se fait à l'aide d'une jointure externe
  - Limite le nombre d'accès à la BD, mais peut entraîner le chargement inutile d'objets

```
<class name="Editeur" table="EDITEUR">
   ...
   <set name="lesLivres" inverse="true" cascade="all"
      fetch="join">
       <key column="ID_EDITEUR"/>
       <one-to-many class="Livre"/>
   </set>
   ...
</class>
```

  - Le paramètre *lazy* peut être employé pour contrôler le moment de chargement des classes, attributs et relations

Annotations

```
import javax.persistence;

@Entity
@Table(name="EDITEUR")
public class Editeur
{
   private Integer id;
   private String nom;
   private String ville ;
   private Set lesLivres = new HashSet();

   public Editeur(){}

   @Id
   @GeneratedValue
   @Column(name="ID_EDITEUR")
   public Integer getId(){return id;}

   @Column(name="NOM_EDITEUR")
   public String getNom (){return nom;}

   @Column(name="VILLE")
   public String getVille(){return ville;}

   @OneToMany(cascade = CascadeType.ALL)
   @JoinColumn(name = "ID_EDITEUR")
   public Set getLesLivres(){return lesLivres;}

   ...
}
```

- Avantages:
  - Moins encombrant que le XML
  - Flexibilité (ex: *refactoring* des noms)
  - Pas besoin de décoder les fichiers (performance)
- Inconvénient:
  - Crée une dépendance entre le code source et le mappage