Chapitre 3

Persistance de données dans le contexte d'applications orientées objet

Master II STIC

Année universitaire 2025/2026

3.1. Introduction

Dans le développement d'applications orientées objet (OO), les données sont représentées sous forme d'objets (instances de classes), tandis que les bases de données relationnelles stockent les informations sous forme de tables.

- Cette différence de paradigme crée une difficulté appelée :

Object–Relational Impedance Mismatch→ le décalage entre les modèles de programmation (objets) et de stockage (relations).

3.1. Introduction

Pour résoudre ce problème, on utilise des mécanismes de persistance d'objets : des techniques permettant de sauvegarder, retrouver et manipuler des objets en base de données tout en minimisant la différence entre les deux mondes.

3.2. Qu’est-ce que la persistance d’objet ?

3.2.1. Définition: Un objet est persistant lorsqu’il continue d’exister au-delà de la durée de vie du programme qui l’a créé, c’est-à-dire lorsqu’il est stocké de façon durable dans une base de données.

3.2.2. Objectifs de la persistance:

- Sauvegarder les états des objets de manière durable.

- Restaurer les objets lors de la relance de l’application.Maintenir les relations entre objets.Assurer la cohérence entre mémoire et base de données.

3.3. Problématique : le décalage objet–relation

Aspect	Modèle Objet	Modèle Relationnel
Structure	Classes et objets	Tables et tuples
Identité	Référence (OID)	Clé primaire
Relations	Références entre objets	Clés étrangères
Héritage	Classes dérivées	Tables séparées / jointes
Méthodes	Encapsulation	Aucune équivalence directe
Collections	Listes, ensembles	Tables liées

- Le rôle du mapping objet-relationnel (ORM) est de combler ce décalage.

3.4. Le Mapping Objet–Relationnel (ORM)

Le mapping objet–relationnel (Object-Relational Mapping) est le processus qui définit la correspondance entre :

- les classes du modèle objet, et
- les tables du modèle relationnel.

Il permet à l'application de manipuler des objets sans se soucier de la manière dont ils sont stockés.

3.5. Types de mapping

3.5.1. Mapping semi-transparent d'objets

Définition :

- L'application gère une partie du passage entre objets et tables.
- Le développeur écrit manuellement les instructions SQL (INSERT, SELECT, etc.).
- Les objets sont convertis (sérialisés/désérialisés) explicitement.

- Le mapping est **manuel**. Le programmeur traduit les objets en tuples.

3.5.1. Mapping semi-transparent d’objets

Avantages :

- Contrôle total sur les requêtes SQL.
- Performance optimisable.

Inconvénients :

- Code redondant et lourd.
- Couplage fort entre code et base de données.

3.5.2. Mapping transparent d’objets
3.5.1. Mapping transparent d’objets

Définition :

- Le mapping est géré automatiquement par un **framework ORM**.
- L’application manipule uniquement des objets.
- Le framework se charge de générer et exécuter les requêtes SQL correspondantes.

3.5.2. Mapping transparent d’objets

Avantages :

- Simplicité du code (orienté objet pur).
- Portabilité entre SGBD.
- Maintenabilité accrue.

Inconvénients :

- Moins de contrôle sur les requêtes.
- Légère perte de performance possible.

3.6. Stratégies de mapping objet–relationnel

Élément OO	Équivalent relationnel	Stratégie de mapping
Classe	Table	1 classe → 1 table
Attribut simple	Colonne	Directe
Attribut composé	Colonnes multiples ou table séparée	Dénormalisation ou relation
Association (1–N, N–N)	Clé étrangère, table associative	JOIN
Héritage	Tables séparées / combinées	3 approches (voir ci-dessous)

3.6.1. Mapping de l’héritage

Stratégie	Description	Avantage	Inconvénient
Table par hiérarchie	Une seule table pour toutes les classes	Simple	Champs NULL
Table par sous-classe	Une table pour chaque sous-classe	Flexible	Nécessite jointures
Table par classe concrète	Une table par classe concrète	Direct	Redondance

3.7. Technologies ORM populaires

Langage	Framework ORM	Description
Java	Hibernate, JPA	Standard industriel pour Java EE
Python	SQLAlchemy, Django ORM	Mapping souple et intuitif
C# / .NET	Entity Framework	Intégré à .NET
PHP	Doctrine ORM	Utilisé avec Symfony / Laravel
C++	ODB ORM	ORM natif performant

3.8. Cycle de vie d’un objet persistant

Transient: L’objet est créé en mémoire, non encore sauvegardé.

Persistent: L’objet est associé à une ligne de table (sauvegardé).

Detached: L’objet est déconnecté de la session mais existe encore.

Removed: L’objet est supprimé de la base.

3.9. Avantages de la persistance objet et de l'ORM

- Réduction du code SQL manuel
- Intégration naturelle avec la programmation objet
- Support automatique des transactions
- Cohérence entre modèle applicatif et modèle de données
- Portabilité entre bases de données

3.10. Limites et inconvénients

- Difficulté à optimiser certaines requêtes complexes
- Génération SQL parfois inefficace
- Courbe d’apprentissage des frameworks ORM
- Problèmes de synchronisation en cas de caches multiples

Exemple pratique (JBDC)

Table et classe Personne

1. Création de table

```
CREATE TABLE personne (
    id SERIAL PRIMARY KEY,
    nom VARCHAR(50),
    age INT
);
```

Classe Java

```
public class Personne {
    private int id;
    private String nom;
    private int age;
    public Personne(int id, String nom, int age) {
        this.id = id;
        this.nom = nom;
        this.age = age;
    }
    public Personne(String nom, int age) {
        this.nom = nom;
        this.age = age;
    }
    // Getters et Setters
}
```

3. Connexion à la base (ODBC)

```
Class.forName("oracle.jdbc.driver.OracleDriver");
            // URL de connexion : format général
            // jdbc:oracle:thin:@<host>:<port>:<SID>
            // ou jdbc:oracle:thin:@//<host>:<port>/<service_name>
            String url = "jdbc:oracle:thin:@localhost:1521/BDA";
            String user = "system";
            String password = "oracle";
            // Connexion à la base
            conn= DriverManager.getConnection(url, user, password);
            System.out.println("✅ Connexion réussie à Oracle
Database !");
```

Exemple CRUD complet avec JDBC

INSERT

```
public void insertPersonne(Personne p) throws Exception {
    Connection con = ConnexionBD.getConnection();
    String sql = "INSERT INTO personne (nom, age) VALUES (?, ?)";
    PreparedStatement ps = con.prepareStatement(sql);
    ps.setString(1, p.getNom());
    ps.setInt(2, p.getAge());
    ps.executeUpdate();
    con.close();
}
```

Exemple CRUD complet avec JDBC

UPDATE

```
public void updatePersonne(Personne p) throws Exception {
    Connection con = ConnexionBD.getConnection();
    String sql = "UPDATE personne SET nom=?, age=? WHERE id=?";
    PreparedStatement ps = con.prepareStatement(sql);
    ps.setString(1, p.getNom());
    ps.setInt(2, p.getAge());
    ps.setInt(3, p.getId());
    ps.executeUpdate();
    con.close();
}
```

Exemple CRUD complet avec JDBC

DELETE

```
public void deletePersonne(int id) throws Exception {
    Connection con = ConnexionBD.getConnection();
    PreparedStatement ps = con.prepareStatement("DELETE FROM
personne WHERE id=?");
    ps.setInt(1, id);
    ps.executeUpdate();
    con.close();
}
```

Exemple CRUD complet avec JDBC

SELECT (un enregistrement)

```
public Personne getPersonne(int id) throws Exception {
    Connection con = ConnexionBD.getConnection();
    PreparedStatement ps = con.prepareStatement("SELECT * FROM personne WHERE id=?");
    ps.setInt(1, id);
    ResultSet rs = ps.executeQuery();
    Personne p = null;
    if (rs.next()) {
        p = new Personne(rs.getInt("id"), rs.getString("nom"), rs.getInt("age"));
    }
    con.close();
    return p;
}
```

Exemple CRUD complet avec JDBC

SELECT (liste)

```
public List<Personne> getAllPersonnes() throws Exception {
    Connection con = ConnexionBD.getConnection();
    Statement st = con.createStatement();
    ResultSet rs = st.executeQuery("SELECT * FROM personne");
    List<Personne> list = new ArrayList<>();
    while (rs.next()) {
        list.add(new Personne(rs.getInt("id"), rs.getString("nom"),
rs.getInt("age")));
    }
    con.close();
    return list;
}
```

Limites de JDBC

- Code répétitif et verbeux.
- SQL mêlé à la logique applicative.
- Difficile à maintenir et à faire évoluer.
- Gestion manuelle des transactions et exceptions.
- Aucune gestion automatique des relations entre objets.

Exemple CRUD Hibernate

Classe Personne

```
import jakarta.persistence.*;

@Entity

@Table(name="personne")

public class Personne {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private int id;

    private String nom;

    private int age;

    // Getters, Setters, Constructeurs

}
```

Configuration Hibernate (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/BDA</property>
  <property name="hibernate.connection.username">system</property>
  <property name="hibernate.connection.password">oracle</property>
  <property              name="hibernate.dialect">            org.hibernate.dialect.
Oracle10gDialect</property>
  <property name="hibernate.hbm2ddl.auto">update</property>
  <mapping class="Personne"/>
 </session-factory>
</hibernate-configuration>
```

Exemple CRUD complet

```
SessionFactory factory = new Configuration().configure().buildSessionFactory();

Session session = factory.openSession();

Transaction tx = session.beginTransaction();

// INSERT

Personne p = new Personne();

p.setNom("Karim");

p.setAge(22);

session.persist(p);
```

```
// SELECT (un)

Personne p2 = session.get(Personne.class, 1);

// UPDATE

p2.setAge(25);

session.merge(p2);
```

Exemple CRUD complet

```
// SELECT liste
List<Personne> personnes = session.createQuery("from Personne", Personne.class).list();
// DELETE
session.remove(p2);


tx.commit();
session.close();
factory.close();
```

3.4. Relations en Hibernate

Relation 1–1

```
@Entity
public class Personne {
  @Id @GeneratedValue
  private int id;
  private String nom;

  @OneToOne(cascade=CascadeType.ALL)
  @JoinColumn(name="passeport_id")
  private Passeport passeport;
}
```

```
@Entity
public class Passeport {
  @Id @GeneratedValue
  private int id;
  private String numero;

  @OneToOne(mappedBy="passeport")
  private Personne personne;
}
```

3.4. Relations en Hibernate

Relation 1–N

```
@Entity
public class Departement {
  @Id @GeneratedValue
  private int id;
  private String nom;

@OneToMany(mappedBy="departement",
cascade=CascadeType.ALL)
  private List<Personne> personnes;
}
```

```
@Entity
public class Personne {
  @Id @GeneratedValue
  private int id;
  private String nom;
  @ManyToOne
  @JoinColumn(name="departement_id")
  private Departement departement;
}
```

3.4. Relations en Hibernate

Relation N–N

```
@Entity
public class Etudiant {
  @Id @GeneratedValue
  private int id;
  private String nom;
  @ManyToMany
  @JoinTable(
    name="etudiant_cours",
    joinColumns=@JoinColumn(name="etudiant_id"),
inverseJoinColumns=@JoinColumn(name="cours_id"))
  private List<Cours> cours;
}
```

```
@Entity
public class Cours {
  @Id @GeneratedValue
  private int id;
  private String titre;

  @ManyToMany(mappedBy="cours")
  private List<Etudiant> etudiants;
}
```

3.5. Héritage

Introduction

En programmation orientée objet, les classes peuvent être organisées selon des relations d'héritage.
Or, les bases de données relationnelles **ne supportent pas directement l'héritage**.
Hibernate propose donc **plusieurs stratégies** pour mapper ces hiérarchies de classes sur des tables relationnelles.

3.5. Héritage

Stratégies d'héritage dans Hibernate

Hibernate (et JPA) propose trois stratégies principales :

Stratégie	Annotation	Description
Single Table	`@Inheritance(strategy = InheritanceType.SINGLE_TABLE)`	Une seule table pour toutes les classes, avec une colonne discriminante.
Joined	`@Inheritance(strategy = InheritanceType.JOINED)`	Une table pour la classe mère et une table pour chaque sous-classe, liées par clé primaire.
Table per Class	`@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)`	Chaque sous-classe a sa propre table indépendante contenant toutes les colonnes.

3.5. Héritage

Exemple commun : Personne, Étudiant, Enseignant

```
public abstract class Personne {
    private Long id;
    private String nom;
    private String email;
}

public class Etudiant extends Personne {
    private String matricule;
}

public class Enseignant extends Personne {
    private String grade;
}
```

3.5. Héritage

A. Stratégie 1 : SINGLE_TABLE

Une seule table pour toutes les classes de la hiérarchie.

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "type_personne", discriminatorType = DiscriminatorType.STRING)
public abstract class Personne {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom;
    private String email;
}
```

```
@Entity
@DiscriminatorValue("ETU")
public class Etudiant extends Personne {
    private String matricule;
}
```

```
@Entity
@DiscriminatorValue("ENS")
public class Enseignant extends Personne {
    private String grade;
}
```

3.5. Héritage

A. Stratégie 1 : SINGLE_TABLE

Structure de table générée :

Table PERSONNE
---
id | nom | email | type_personne | matricule | grade

Avantage : très rapide pour la lecture.
Inconvénient : colonnes inutilisées pour certaines lignes.

3.5. Héritage

B. Stratégie 2 : JOINED

Chaque sous-classe a sa table propre, reliée par clé primaire.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Personne {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom;
    private String email;
}
```

```
@Entity
public class Etudiant extends Personne {
    private String matricule;
}
```

```
@Entity
public class Enseignant extends Personne {
    private String grade;
}
```

3.5. Héritage

B. Stratégie 2 : JOINED

Structure générée :

PERSONNE(id, nom, email)
ETUDIANT(id, matricule)
ENSEIGNANT(id, grade)

Avantage : modèle propre, bien normalisé.
Inconvénient : nécessite des jointures pour les requêtes globales.

3.5. Héritage

C. Stratégie 3 : TABLE_PER_CLASS

Chaque sous-classe possède sa propre table complète.

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Personne {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String nom;
    private String email;
}
```

```
@Entity
public class Etudiant extends Personne {
    private String matricule;
}
```

```
@Entity
public class Enseignant extends Personne {
    private String grade;
}
```

3.5. Héritage

C. Stratégie 3 : TABLE_PER_CLASS

Structure générée :

ETUDIANT(id, nom, email, matricule)
ENSEIGNANT(id, nom, email, grade)

Avantage : pas de jointure.
Inconvénient : duplication des colonnes communes.

3.6. Performance et stratégies Hibernate

1. Stratégies de chargement

EAGER loading : chargement immédiat (par défaut pour @OneToOne, @ManyToOne).

LAZY loading : chargement différé (par défaut pour @OneToMany, @ManyToMany).

Exemple

```
@OneToMany(fetch = FetchType.LAZY)

private List<Personne> personnes;
```

3.6. Performance et stratégies Hibernate

2. Cache Hibernate

Cache de premier niveau

- Activé par défaut.
- Propre à la session Hibernate.
- Réduit les accès répétés à la base.

Cache de second niveau

- Partagé entre plusieurs sessions.
- Nécessite une configuration (EhCache, Infinispan, etc.).

```
<property name="hibernate.cache.use_second_level_cache">true</property>

<property
name="hibernate.cache.region.factory_class">org.hibernate.cache.ehcache.EhCacheRegionFact
ory</property>
```

Mappage par fichiers XML

3.7.1. Exemple simple :

Mapping de la classe Personne

Classe Java

```
public class Personne {
    private int id;
    private String nom;
    private int age;

    // Getters et Setters
}
```

Fichier XML : Personne.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN«  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="Personne" table="personne">
    <id name="id" column="id">
      <generator class="identity"/>
    </id>
    <property name="nom" column="nom" type="string"/>
    <property name="age" column="age" type="int"/>
  </class>
</hibernate-mapping>
```

Et dans le fichier hibernate.cfg.xml : <mapping resource="Personne.hbm.xml"/>

3.7.2. Relations entre objets via XML

Relation 1–1 (Personne ↔ Passeport)

```
public class Passeport {
    private int id;
    private String numero;
    private Personne personne;
    // Getters et Setters
}
public class Personne {
    private int id;
    private String nom;
    private Passeport passeport;
    // Getters et Setters
}
```

3.7.2. Relations entre objets via XML

Relation 1–1 (Personne ↔ Passeport)

Fichiers XML: Passeport.hbm.xml

```
<hibernate-mapping>
  <class name="Passeport" table="passeport">
    <id name="id" column="id">
      <generator class="identity"/>
    </id>
    <property name="numero" column="numero"/>
    <one-to-one name="personne" class="Personne"/>
  </class>
</hibernate-mapping>
```

Fichiers XML: Personne.hbm.xml

```
<hibernate-mapping>
  <class name="Personne" table="personne">
    <id name="id" column="id">
      <generator class="identity"/>
    </id>
    <property name="nom" column="nom"/>
    <one-to-one name="passeport«
class="Passeport" cascade="all"/>
  </class>
</hibernate-mapping>
```

3.7.2. Relations entre objets via XML Relation 1–N (Département ↔ Personne)

```
public class Departement {
    private int id;
    private String nom;
    private Set<Personne> personnes;
    // Getters et Setters
}

public class Personne {
    private int id;
    private String nom;
    private Departement departement;
    // Getters et Setters
}
```

3.7.2. Relations entre objets via XML

Relation 1–N (Département ↔ Personne)

Fichiers XML: Departement.hbm.xml

```
<hibernate-mapping>
  <class name="Departement" table="departement">
    <id name="id" column="id">
      <generator class="identity"/>
    </id>
    <property name="nom" column="nom"/>

    <set name="personnes" inverse="true" cascade="all">
      <key column="departement_id"/>
      <one-to-many class="Personne"/>
    </set>
  </class>
</hibernate-mapping>
```

Fichiers XML: Personne.hbm.xml

```
<hibernate-mapping>
  <class name="Personne" table="personne">
    <id name="id" column="id">
      <generator class="identity"/>
    </id>
    <property name="nom" column="nom"/>

    <many-to-one name="departement" class="Departement"
column="departement_id"/>
  </class>
</hibernate-mapping>
```

3.7.2. Relations entre objets via XML

Relation N–N (Étudiant ↔ Cours)

```
public class Etudiant {
    private int id;
    private String nom;
    private Set<Cours> cours;
    // Getters et Setters
}

public class Cours {
    private int id;
    private String titre;
    private Set<Etudiant> etudiants;
    // Getters et Setters
}
```

3.7.2. Relations entre objets via XML

Relation N–N (Étudiant ↔ Cours)

Fichiers XML: Etudiant.hbm.xml

```
<hibernate-mapping>
  <class name="Etudiant" table="etudiant">
    <id name="id" column="id"><generator class="identity"/></id>
    <property name="nom" column="nom"/>
    <set name="cours" table="etudiant_cours" cascade="all">
      <key column="etudiant_id"/>
      <many-to-many class="Cours" column="cours_id"/>
    </set>
  </class>
</hibernate-mapping>
```

Fichiers XML: Cours.hbm.xml

```
<hibernate-mapping>
  <class name="Cours" table="cours">
    <id name="id" column="id"><generator class="identity"/></id>
    <property name="titre" column="titre"/>
    <set name="etudiants" table="etudiant_cours" inverse="true">
      <key column="cours_id"/>
      <many-to-many class="Etudiant" column="etudiant_id"/>
    </set>
  </class>
</hibernate-mapping>
```

3.7.3. Héritage en mapping XML

Hibernate permet aussi de décrire ces mêmes stratégies via des fichiers XML.

C’est utile dans les projets sans annotations ou lorsqu’on souhaite séparer le code Java de la configuration.

3.7.3. Héritage en mapping XML

A. Stratégie SINGLE_TABLE (XML)

Fichier `Personne.hbm.xml`

```
<hibernate-mapping>
 <class name="Personne" table="personne" discriminator-value="P">
  <id name="id" column="id">
   <generator class="increment"/>
  </id>
  <discriminator column="type_personne" type="string"/>
  <property name="nom" column="nom"/>
  <property name="email" column="email"/>
  <subclass name="Etudiant" discriminator-value="E">
   <property name="matricule" column="matricule"/>
  </subclass>
  <subclass name="Enseignant" discriminator-value="N">
   <property name="grade" column="grade"/>
  </subclass>
 </class>
</hibernate-mapping>
```

3.7.3. Héritage en mapping XML

B. Stratégie JOINED (XML)

Fichier `Personne.hbm.xml`

```
<hibernate-mapping>
 <class name="Personne" table="personne">
  <id name="id" column="id">
   <generator class="increment"/>
  </id>
  <property name="nom" column="nom"/>
  <property name="email" column="email"/>
  <joined-subclass name="Etudiant" table="etudiant">
   <key column="id"/>
   <property name="matricule" column="matricule"/>
  </joined-subclass>
  <joined-subclass name="Enseignant" table="enseignant">
   <key column="id"/>
   <property name="grade" column="grade"/>
  </joined-subclass>
 </class>
</hibernate-mapping>
```

3.7.3. Héritage en mapping XML

C. Stratégie TABLE_PER_CLASS (XML)

Fichier `Personne.hbm.xml`

```
<hibernate-mapping>
 <class name="Personne" abstract="true" table="personne">
  <id name="id" column="id">
   <generator class="increment"/>
  </id>
  <property name="nom" column="nom"/>
  <property name="email" column="email"/>
 </class>
 <union-subclass name="Etudiant" table="etudiant">
  <property name="matricule" column="matricule"/>
 </union-subclass>
 <union-subclass name="Enseignant" table="enseignant">
  <property name="grade" column="grade"/>
 </union-subclass>
</hibernate-mapping>
```

Avantages du mapping XML

Aspect	Avantage
Séparation du code	Le mapping est externe au code source.
Maintenance	Facile à modifier sans recompiler.
Compatibilité	Utilisable avec du code non-annoté (hérité).
Souplesse	Permet des configurations complexes (héritages multiples, vues, jointures personnalisées).

Inconvénients

Aspect	Limite
Verbosité	Les fichiers XML sont longs et répétitifs.
Synchronisation	Risque d'incohérence avec le code Java.
Popularité	Les annotations JPA sont désormais préférées.

3.7.4. Comparatif global des stratégies

Stratégie	Performances lecture	Normalisation BD	Complexité requêtes	Recommandée pour
SINGLE_TABLE	Excellente	Faible	Simple	Hiérarchies légères
JOINED	Moyenne	Excellente	Moyenne	Grands modèles
TABLE_PER_CLASS	Moyenne	Moyenne	Simple	Classes indépendantes

Conclusion générale sur la persistance Hibernate

Hibernate offre deux modes complémentaires :

- **Mapping par annotations** (plus concis, standard JPA),
- **Mapping par XML** (plus flexible et compatible avec les anciens projets).

Dans les architectures modernes, on combine souvent les deux :
classes métier annotées + quelques mappings XML spécifiques pour les cas complexes (héritage, vues matérialisées, etc.).

Points clés à retenir

- Les annotations JPA simplifient le code et favorisent la maintenance.
- Le mappage XML est utile pour les projets configurables et les environnements sans code source modifiable.
- Le choix de la stratégie dépend du compromis entre **performance**, **normalisation** et **souplesse**.