Principes de base du mapping objet-relationnel JPA

© Olivier Caron - Polytech Lille

15 février 2024

Merci à Bernard Carré, Bastien Cazaux, Frédéric Hoogstoel et Walter Rudametkin pour les remarques, commentaires et discussions sur la rédaction de ce document.

Les sources Java des classes décrites dans ce poly sont accessibles dans le projet GIT :
`https://gitlab.univ-lille.fr/olivier.caron/polyjpa.git`

1 JPA en quelques mots

JPA est l'acronyme pour Java Persistence API. Cette interface de programmation permet de manipuler des données relationnelles issues de bases de données à l'aide d'objets Java. Cette interface est essentiellement constituée d'annotations Java qui permettent d'établir le lien entre objets Java et données relationnelles. JPA permet donc de disposer d'une vue orientée objet à partir d'une base de données relationnelles. JPA fournit également un langage de requêtes qui permet de charger en mémoire des objets Java à partir de données stockées en base.

1.1 JPA versus EJB

On utilise parfois le terme d'entités JPA et parfois d'entités EJB (Enterprise Java Beans). A partir de la version 3.0 des EJB, la programmation Java des entités EJB est rigoureusement identique à celle des entités JPA. La différence est que les entités EJB ne peuvent s'exécuter qu'au sein d'un serveur d'applications JEE (Java Enterprise Edition) alors que les entités JPA peuvent s'exécuter dans une application Java simple.

2 Ce document

Ce document **non exhaustif** décrit les principales annotations de JPA toutes définies dans le paquetage Java : `jakarta.persistence` (avant 2017 : `javax.persistence`).

La section 3 décrit un système d'information simple qui illustrera les différents concepts. La section 4 décrit les annotations JPA de base. Le framework JPA permet de créer de toutes pièces une base de données relationnelle grâce à ces annotations et un ensemble de règles de génération.

Dans le cas où une base de données existe déjà, il est nécessaire d'ajouter des informations afin de relier correctement objets Java avec les données relationnelles. Le jeu d'annotations JPA de la section 5 permet de spécifier ces informations.

3 Application fil rouge

Afin d'illustrer l'API JPA, le système d'information d'une gestion d'un parc informatique sera utilisé. Dans les salles du parc informatique se trouvent des ordinateurs de type PC ou MAC. Un ordinateur est identifié par un code et se caractérise par son nom de machine ainsi que son numéro IP associé. Pour les ordinateurs de type PC, on désire également savoir si le système Linux est installé.

Différents logiciels sont installés sur les différents ordinateurs. Plusieurs versions d'un même logiciel peuvent être installées sur un même ordinateur. Enfin, les salles du parc informatique peuvent être utilisées pour plusieurs formations identifiées par un code. Le schéma UML de ce système d'information est conforme à la figure 1

4 Traduction d'un schéma UML aux entités JPA

Lors de la phase de conception, les spécificités technologiques ne sont en général pas prises en compte. Le schéma UML est donc indépendant de toute caractéristique technologique. Lors de la traduction d'un schéma UML vers une technologie donnée, certaines caractéristiques ou limitations de la plateforme retenue peuvent apparaître, un second schéma UML peut alors être spécifié pour adapter le modèle UML initial en fonction de ces contraintes. Voici une liste des adaptations requises :

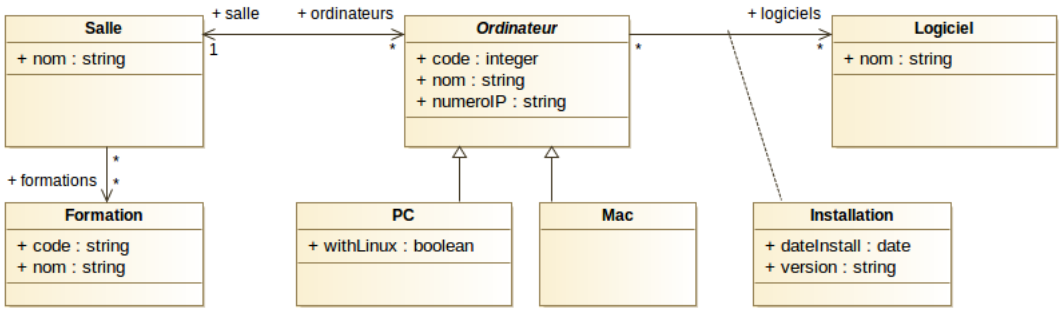

FIGURE 1 – schéma UML du parc informatique

1. Les entités JPA ne supportent que l'héritage simple de classes. C'est une réelle limitation et il est souvent conseillé de modéliser le problème en ne recourant pas à l'héritage multiple de classes.
2. Les entités JPA ne supportent que les associations binaires. Les associations n-aires (avec $n > 2$) peuvent être transformées en plusieurs associations binaires et fournir un résultat conceptuellement équivalent. Il est parfois nécessaire de compléter ces associations binaires par des contraintes pour obtenir le même niveau d'information sémantique du schéma initial (informations indiquées par exemple dans les cardinalités de l'association n-aire).
3. Les entités JPA nécessitent d'avoir un attribut qui joue le rôle de clé primaire. Le standard UML dispose d'un mécanisme d'extension qui autorise l'annotation d'éléments du modèle. On utilisera une annotation `Identifiant` (les annotations sont des stéréotypes en UML) pour caractériser cet attribut (cette annotation sera représentée par une image dessinant une clé dans les figures de ce document). Si aucun des attributs d'une classe UML ne peut jouer le rôle de clé primaire, il est nécessaire d'en ajouter un.
4. Les associations binaires JPA ne peuvent pas avoir de propriétés d'associations. Dans ce cas, on substitue l'association par une classe. Cette classe sera reliée par associations binaires aux deux classes de l'association initiale et contiendra les propriétés de l'association initiale. Un attribut de cette classe devra jouer le rôle de clé primaire.

Pour le cas du parc informatique, les adaptations retenues sont les suivantes :

— Transformation de la classe-association `Installation` en véritable classe UML (adaptation 3). Cette classe sera reliée à `Ordinateur` et `Logiciel` pour suppléer à l'association supprimée entre `Ordinateur` et `Logiciel`.
— Ajout des attributs `Logiciel.codeLogiciel` et `Installation.codeInstall` puis ajout de l'annotation `Identifiant` à un attribut de chaque classe UML (adaptation 2)

La figure 2 décrit le second schéma UML adapté aux contraintes technologiques JPA.

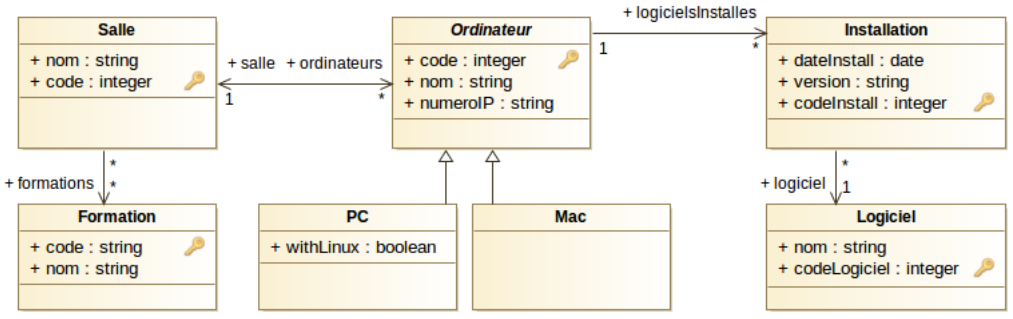

FIGURE 2 – schéma UML annoté du parc informatique - compatible JPA

4.1 Traduction des classes, attributs et clés primaires

Toute entité JPA correspond à une classe Java qui doit respecter les critères suivants :

— La classe est annotée par l'annotation `@Entity` (exemple ligne 6 - listing 1)
— La classe dispose d'une fonction constructeur sans paramètres, elle peut contenir d'autres fonctions constructeurs
— Ni la classe ni ses méthodes ne doivent être déclarées `final`

— Les attributs des classes du schéma UML sont représentés en Java à l'aide d'attributs Java[1].
— La classe dispose d'un attribut annoté par `@Id` (exemple ligne 9 - listing 1).
— La classe doit implémenter l'interface `java.io.Serializable` dans le cas d'applications Java en réseau.

L'extrait de code source pour l'entité JPA `Logiciel` se trouve dans le listing 1

```
1  package parcInfo ;
2
3  import jakarta.persistence.Entity ;
4  import jakarta.persistence.Id ;
5
6  @Entity
7  public class Logiciel implements java.io.Serializable {
8
9    @Id private int codeLogiciel ;
10   private String nom ;
11
12   public Logiciel() { }
13
14   public int getCodeLogiciel() { return this.codeLogiciel ; }
15   public void setCodeLogiciel(int value) { this.codeLogiciel=value ; }
16
17   public String getNom() { return this.nom ; }
18   public void setNom(String value) { this.nom=value ; }
19   ...
20 }
```

Listing 1 – code Java de Logiciel

Toute entité JPA a un équivalent dans le modèle relationnel. Une traduction Java-relationnel (on parle de « mapping ») existe par défaut. Voici quelques notions élémentaires de ce mapping :

— Une entité JPA est traduite par une table de même nom.
— Chaque **attribut** Java est traduit par une colonne de même nom. Il existe un mapping par défaut pour chaque type Java : le type Java `int` ou `Integer` se traduit par le type SQL `integer`, le type `String` se traduit par le type SQL `varchar(255)`, le type `java.sql.Date` se traduit par le type SQL `date`, ...
— L'attribut annoté par `@Id` est traduit par une colonne qui joue le rôle de clé primaire de la table. Pour représenter une clé composite, il faut définir une classe sérializable incluant les attributs concernés (non développé dans ce document). Il est possible de générer automatiquement une valeur de clé. Pour cela, l'attribut clé primaire sera également annoté par `@GeneratedValue` (cette fonctionnalité n'est valable que pour les clés entières).

Il est possible d'adapter ce mapping par défaut. Par exemple, avoir un nom de table ou de colonne différent du concept Java associé, une taille de chaîne de caractères différente de 255, ...

L'API JPA propose un jeu d'annotations Java qui permet l'adaptation de ce mapping par défaut, quelques-unes de ces annotations seront détaillées dans la section 5.

Par défaut, les propriétés d'une entité JPA sont considérées comme persistantes. Si l'on veut qu'une propriété ne soit pas persistante, il est nécessaire de la préfixer par `@Transient`.

4.2 Traduction de l'héritage de classes

Le concept d'héritage n'existe pas dans le modèle relationnel. L'API JPA propose 3 stratégies d'implantation de l'héritage décrites dans les sous-sections suivantes.

4.2.1 Stratégie d'héritage : une table unique par hiérarchie de classes JPA.

Cette stratégie est le mode par défaut. Classe racine et sous-classes sont traduites par une unique table qui contiendra toutes les instances de ces classes. Cette table contient les colonnes qui correspondent aux propriétés de la classe racine ainsi qu'aux propriétés de ces sous-classes. Une colonne additionnelle de nom par défaut `dtype` de type chaîne de caractères est employée pour connaître la classe d'instanciation de la ligne (le nom de la classe).

La hiérarchie de classes de `Ordinateur` est fournie dans le listing 2.

1. Il existe une traduction alternative qui exploite la notion de propriété bien connue du modèle Java Bean, cette alternative ne sera pas traitée dans ce document.

```java
/**** fichier Ordinateur.java ****/
package polyJPA;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;

@Entity public abstract class Ordinateur implements java.io.Serializable {
  @Id private int code ;
  private String nom ;
  private String numeroIP ;

  public Ordinateur() {}

  public int getCode() { return this.code ; }
  public void setCode(int value) { this.code=value ; }
  ...
}
/**** fichier Mac.java ****/
package polyJPA;

import jakarta.persistence.Entity;
@Entity public class Mac extends Ordinateur {
  public Mac() {}
}
/**** fichier PC.java ****/
package polyJPA;

import jakarta.persistence.Entity;
@Entity public class PC extends Ordinateur {
  private boolean withLinux ;
  public PC() {}  ...
}
```

Listing 2 – hiérarchie d'héritage

Le code de cette hiérarchie de classes sera associé à la table relationnelle `ordinateur` décrite dans la table 1.

dtype	code (primary key)	nom	numeroip	withlinux
varchar(31)	integer	varchar(255)	varchar(255)	boolean
Mac	112	Mac Aron	192.168.38.01	null
PC	113	PC V	192.168.38.02	true

TABLE 1 – table ordinateur, stratégie "une table par hiérarchie"

Avantages/Inconvénients :

Cette stratégie évite la prolifération de tables lors d'une hiérarchie importante de classes. Mais elle engendre une table complexe (beaucoup de colonnes) avec des valeurs souvent positionnées à `null` (exemple : colonne `withlinux` vaut `null` pour les lignes dont `dtype`='`Mac`'). Cette stratégie par défaut ne nécessite pas d'annotations Java supplémentaires. Le développeur Java peut cependant expliciter cette stratégie d'implantation de l'héritage par l'annotation `@Inheritance`. L'attribut `strategy` de cette annotation permet d'expliciter la stratégie désirée. Le listing 3 suivant illustre l'emploi de cette annotation (exemple lignes 3-4 du listing 3)

```java
1 package polyJPA;
2
3 @jakarta.persistence.Inheritance
4   (strategy=jakarta.persistence.InheritanceType.SINGLE_TABLE)
5 @jakarta.persistence.Entity
6 public abstract class Ordinateur implements java.io.Serializable { ... }
```

Listing 3 – stratégie : une table par hiérarchie

`InheritanceType` est un type Java énuméré qui identifie les 3 mappings possibles (exemple ligne 3 - listing 3).

4.2.2 Stratégie d'héritage : une table par classe

Cette stratégie associe chaque classe **concrète** à une table relationnelle. La programmation de cette stratégie est simple : il suffit d'expliciter ce mode via l'annotation `@Inheritance` comme le montre cet exemple (lignes 3-4 - listing 4) :

```
1 package polyJPA;
2
3 @jakarta.persistence.Inheritance
4   (strategy=jakarta.persistence.InheritanceType.TABLE_PER_CLASS)
5 @jakarta.persistence.Entity
6 public abstract class Ordinateur implements java.io.Serializable { ... }
```

Listing 4 – stratégie : une table par classe

Avec cette stratégie, la table relationnelle `pc` aura la structure décrite dans la table 2.

code (primary key)	**nom**	**numeroip**	**withlinux**
integer	varchar(255)	varchar(255)	boolean
113	PC V	192.168.38.02	true

TABLE 2 – table pc, stratégie "une table par classe"

Avantages/Inconvénients :
Cette stratégie permet de localiser facilement les instances de chaque classe. Il n'y a plus de valeurs nulles. En revanche, cette stratégie duplique toutes les propriétés héritées dans chaque « sous-table ». La récupération de toutes les instances d'une hiérarchie est également moins efficace (nécessite l'emploi de l'opérateur SQL d'union pour collecter les instances).

4.2.3 Stratégie d'héritage : jointure de tables

Cette stratégie associe chaque classe (**abstraite ou pas**) à une table relationnelle. La colonne correspondant à la clé primaire est dupliquée dans toutes les tables. Des clés étrangères sont générées afin de garantir la cohérence entre sur-classe et sous-classes. En plus de la clé primaire, les tables ne contiennent que les colonnes représentant les attributs de la classe associée. Au niveau de la table correspondant à la classe racine de la hiérarchie, la colonne `dtype` permet de connaître la classe d'instanciation de la ligne (le nom de la classe). La programmation de cette stratégie se fait aussi simplement que les précédentes (cf lignes 3-4 du code ci-dessous) :

```
1 package polyJPA;
2
3 @jakarta.persistence.Inheritance(
4   strategy=jakarta.persistence.InheritanceType.JOINED)
5 @jakarta.persistence.Entity
6 public abstract class Ordinateur implements java.io.Serializable { ... }
```

Listing 5 – stratégie : jointure de tables

La table 3 décrit la structure de la base de données associée à cette stratégie pour les tables `ordinateur` et `pc`.

table ordinateur			
code (primary key) (foreign key : pc) (foreign key : mac)	**dtype**	**nom**	**numeroip**
integer	varchar(31)	varchar(255)	varchar(255)
112	Mac	Mac Aron	192.168.38.01
113	PC	PC V	192.168.38.02

table pc	
code (primary key)	**withlinux**
integer	boolean
113	true

TABLE 3 – tables ordinateur et pc, stratégie "jointure de tables"

Avantages/Inconvénients :
Cette stratégie permet de ne plus dupliquer les propriétés héritées dans chaque classe. La récupération de toutes les instances d'une hiérarchie est également moins efficace que la première stratégie (nécessite l'emploi des opérateurs SQL algébriques `join` et `union` pour collecter les instances)

4.3 Traduction des associations

Le concept d'association n'existe pas en tant que tel dans les langages orienté-objet. Les associations vont donc être traduites par une ou plusieurs références d'objet annotées.

Pour traduire les associations binaires UML en Java, il est nécessaire de considérer les éléments de modélisation suivants :

— *La navigation des associations UML.* Chaque rôle navigable de l'association (présence d'une flèche [2]) se traduit par un attribut Java. Si l'association est bidirectionnelle, il y a donc un attribut défini dans chaque classe participante de l'association. Si l'association est unidirectionnelle, seule la classe qui permet de naviguer vers la classe opposée possède un attribut qui référence cette classe navigable. En général, le nom du rôle de l'association est utilisé pour le nom de l'attribut correspondant.

— *La cardinalité des rôles des associations.* Le type de l'attribut associé au rôle dépend de la cardinalité max de ce rôle. Si la cardinalité max est 1, ce type est la classe associée au rôle. Si cette cardinalité est '*' (ou bien toute valeur > 1), le type est une `Collection` (où l'un de ses sous-types) de la classe associée au rôle. Le choix de ce sous-type est guidé par les caractéristiques des types Java de `Collection` (voir encadré 1).

— *L'association elle-même.* Chaque attribut doit être annoté par l'une de ces 4 annotations : `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`. Ces annotations décrivent les deux cardinalités max de l'association. Pour chaque annotation `@XtoY`, `X` désigne la cardinalité max de la classe qui contient cette annotation, `Y` désigne la cardinalité max de la classe opposée (indiquée par le type de l'attribut annoté). Dans le cas d'association bidirectionnelle, il est également nécessaire d'indiquer quel est le rôle qui sera le *responsable de la mise à jour* de l'association, (pour chaque association, il n'en existe qu'un, voir discussion de la section 4.3.1). Cela sera réalisé avec l'attribut `mappedBy` disponible dans les 3 annotations `@OneToOne`, `@OneToMany` et `@ManyToMany`. Le second apport de l'attribut `mappedBy` est qu'il permet d'indiquer que les deux propriétés Java correspondent aux rôles de la **même** association. Sans cela, JPA considère que les deux attributs Java sont des rôles de deux associations unidirectionnelles **distinctes**.

Encadré 1: les collections Java

Les sous-types principaux de la classe `Collection` se distinguent par les caractéristiques suivantes :

`java.util.Set` est utilisé pour définir un groupe d'éléments uniques.

`java.util.List` est utilisé pour définir une suite d'éléments ordonnés accessibles par leur rang dans la liste sans garantie d'unicité des éléments de cette liste.

`java.util.Map` est utilisé pour définir une collection de couples clé-valeur. Les clés sont uniques mais la même valeur peut-être associée à plusieurs clés.

Considérons la classe `Salle` de notre exemple, cette classe est concernée par l'association bidirectionnelle 1-* avec `Ordinateur` et par l'association unidirectionnelle *-* avec `Formation`, deux attributs annotés seront définis dans cette classe (cf lignes 7-8 du listing 6). Les noms des rôles des associations UML sont employés pour les noms des attributs Java. Dans le cas de l'association bidirectionnelle, l'attribut `mappedBy` est spécifié et précise l'attribut opposé `salle`, cet attribut doit être défini dans la classe Ordinateur (cf ligne 6 du listing 7) (attention : un seul `mappedBy` par association). Le type Java `Set` est employé pour ces deux variables.

```
1  package polyJPA;
2
3  import ...
4
5  @Entity
6  public class Salle implements java.io.Serializable {
7    @OneToMany(mappedBy="salle") private Set<Ordinateur> ordinateurs ;
8    @ManyToMany private Set<Formation> formations
9
10   public Salle() {
11     this.ordinateurs = new java.util.HashSet<Ordinateur>() ;
12     this.formations = new java.util.HashSet<Formation>() ;
13   }
14
15   public Set<Ordinateur> getOrdinateurs() { return this.ordinateurs ;}
16   public void setOrdinateurs(Set<Ordinateur> value) { this.ordinateurs=value ; }
17
18   public Set<Formation> formations { return this.formations ; }
```

2. En UML, une association binaire ne contenant aucune flèche est considérée comme bidirectionnelle : les deux extrémités de l'association sont donc considérées comme navigables)

```
19    public void setFormations(Set<Formation> value) { this.formations=value ; }
20    ...
21  }
```

Listing 6 – Gestion des associations pour `Salle`

Le listing 7 indique la programmation JPA des associations (lignes 6-7) de la classe `Ordinateur`.

```
1   package polyJPA;
2
3   import ...
4
5   @Entity public abstract class Ordinateur implements java.io.Serializable {
6     @ManyToOne private Salle salle ;
7     @OneToMany private Set<Installation> logicielsInstalles ;
8     ...
9
10    public Ordinateur() {
11      this.logicielsInstalles = new java.util .HashSet<Installation>() ;
12    }
13    public Salle getSalle() { return this.salle ;}
14    public void setSalle(Salle value) { this.salle=value ; }
15
16    public Set<Installation> getLogicielsInstalles() {
17      return this.logicielsInstalles ;
18    }
19    public void setLogicielsInstalles(Set<Installation> value) {
20      this.logicielsInstalles=value ;
21    }
22  }
```

Listing 7 – Gestion des associations pour `Ordinateur`

Enfin, le listing 8 indique la programmation JPA de l'association entre `Installation` et `Logiciel` (ligne 6) .

```
1   package polyJPA;
2
3   import ...
4
5   @Entity public class Installation implements java.io.Serializable {
6     @ManyToOne private Logiciel logiciel ;
7     ...
8     public Logiciel getLogiciel() { return this.logiciel ; }
9     public void setLogiciel(Logiciel value) { this.logiciel=value ; }
10  }
```

Listing 8 – Gestion des associations pour `Installation`

En n'utilisant que les seules annotations `@XtoY`, JPA propose un mapping relationnel par défaut. Les règles de ce mapping sont les suivantes.

Dans le cas d'association *-* entre deux classes `C1` et `C2` , une table-jointure de nom `C1_C2` est créée, cette table contient deux colonnes qui sont des clés étrangères vers `C1` et `C2`. Ces deux colonnes forment la clé primaire de cette table-jointure. Le nom des colonnes sera soit `nomDuRole_nomCléPrimaire` si le rôle-propriété existe, soit `nomTable_nomCléPrimaire` si le rôle-propriété n'existe pas (cas d'association uni-directionnelle). Pour notre exemple, l'implantation relationnelle de l'association unidirectionnelle *-* entre `Salle` et `Formation` est décrite dans la table 4

salle_code	**formations_code**
primary key : (salle_code, formations_code)	
foreign key : salle	foreign key : formation
integer	integer

TABLE 4 – Table salle_formation (association *-*)

Dans le cas d'association 1-*, deux cas de figure se présentent :

1. Le rôle de cardinalité max égal à 1 est un rôle navigable : dans ce cas, une colonne clé étrangère de nom `nomTable_nomRôle` correspondant à ce rôle sera créé. L'association entre `Ordinateur` et `Salle` ainsi que l'association entre `Installation` et `Logiciel` font partie de ce cas de figure : création des colonnes `salle_code` (table `ordinateur`) et `logiciel_codelogiciel` (table `installation`).
2. Le rôle de cardinalité max égal à 1 est un rôle non navigable : une table-jointure est créée avec les mêmes règles de création que pour les associations *-* avec une contrainte d'unicité supplémentaire sur la colonne qui référence la table côté *. L'association entre `Ordinateur` et `Installation` fait partie de ce cas de figure : création de la table `ordinateur_installation`.

La table 5 décrit les structures de tables relationnelles impactées par les associations 1-*.

table ordinateur				
dtype	code	nom	numeroip	salle_code
	primary key			foreign key :salle
varchar(31)	integer	varchar(255)	varchar(255)	integer

table installation			
codeinstall	dateinstall	version	logiciel_codelogiciel
primary key			foreign key :logiciel
integer	date	varchar(255)	integer

table ordinateur_installation	
ordinateur_code	logicielsinstalles_codeinstall
primary key : (ordinateur_code,logicielsinstalles_codeinstall)	
foreign key : ordinateur	foreign key : installation
	unique
integer	integer

TABLE 5 – Implantation des associations 1-*

4.3.1 Discussion sur le rôle de l'attribut `mappedBy` .

Prenons l'exemple de l'association entre salle et Ordinateur. Soient deux objets java persistants qui vont ajouter un lien d'association :

```
1 unOrdinateur.setSalle(uneSalle) ;
2 uneSalle.getOrdinateurs().add(unOrdinateur) ;
```

Listing 9 – Extrait Java, ajout d'un lien d'association

Logiquement, on peut s'attendre à ce que le gestionnaire d'entités déclenche deux « update » dans la base (un pour chaque ligne de code). Or, l'implantation de cette association est une simple clé étrangère. L'exécution de la première ligne de code Java est suffisante pour modifier la base. Pour éviter la multiplication d'updates inutiles dans le cas d'associations bidirectionnelles, le gestionnaire d'entités ne met à jour la base que pour le rôle défini par la propriété indiquée par `mappedBy` (ici `salle`, cf lignes 9-11 du listing 7). Le gestionnaire d'entités ne produira donc pas de mises à jour pour la seconde ligne de code. Cette solution optimise l'accès aux données relationnelles mais introduit une source d'incohérence au niveau Java.

5 Personnalisation du mapping objet-relationnel

Les règles du mapping objet-relationnel par défaut offrent l'avantage au programmeur Java d'utiliser un minimum d'annotations Java. Lors de la construction d'un nouvelle base de données, ce mapping par défaut ne pose aucun problème. Mais lorsqu'une base de données existe déjà, il faut pouvoir configurer ce mapping afin de relier parfaitement objets Java et données relationnelles existantes. Le framework JPA propose un jeu d'annotations Java qui permet de modifier le mapping par défaut. Cette section décrit quelques-unes de ces annotations.

5.1 Adaptation du mapping des classes.

L'annotation `@Table` permet de relier une entité Java à une table relationnelle de nom différent du nom de l'entité. Le nom de la table sera spécifié à l'aide de l'attribut `name` de cette annotation. La ligne 5 du listing 10 permet de relier l'entité `Logiciel` à la table `t_logiciel`.

5.2 Adaptation des attributs

On utilisera l'annotation `@Column` pour modifier certaines caractéristiques de la colonne reliée à l'attribut Java. L'attribut `name` permet de spécifier le nom de la colonne relationnelle, l'attribut booléen `length` permet de modifier la taille (cas des chaînes de caractères), l'attribut booléen `unique` ajoute une contrainte d'unicité sur la colonne, l'attribut booléen `nullable` permet d'autoriser ou d'empêcher des valeurs nulles.

Le listing suivant décrit un exemple d'utilisation de cette annotation (lignes 8 et 11).

```
1  package polyJPA;
2
3  import jakarta.persistence.*;
4
5  @Table(name="t_logiciel")
6  @Entity
7  public class Logiciel  implements java.io.Serializable  {
8    @Column(name="t_nom", length=40, unique=true, nullable=false)
9    private String nom ;
10   @GeneratedValue @Id
11   @Column (name="code")
12   private int codeLogiciel ;
13
14   public int getCodeLogiciel() { return this.codeLogiciel ;}
15   public void setCodeLogiciel(int value) { this.codeLogiciel=value ; }
16
17   public String getNom() { return this.nom ; }
18   public void setNom(String value) { this.nom=value ; }
19   ...
20 }
```

Listing 10 – Adaptation du mapping des classes et attributs

Pour les propriétés entières annotées par `@Id`, il est possible de générer une valeur de clé automatiquement, pour cela on utilisera l'annotation `@GeneratedValue` (cf ligne 10 du listing 10)

5.3 Adaptation de l'héritage

Par défaut, une colonne de nom `dtype` est employée pour distinguer le type de l'instance. Les valeurs possibles sont les noms des entités. L'annotation `@DiscriminatorColumn` permet d'expliciter le nom de cette colonne tandis que l'annotation `@DiscriminatorValue` permet de donner une valeur pour cette colonne (annotation à définir pour chaque classe de la hiérarchie). Un exemple est donné dans le listing 11 (lignes 5-7)

```
1  package polyJPA;
2
3  import jakarta.persistence.* ;
4
5  @DiscriminatorColumn
6    (name="type_ordi", discriminatorType=DiscriminatorType.STRING, length=30)
7  @DiscriminatorValue("ordinateur")
8  @Entity
9  public abstract class Ordinateur implements java.io.Serializable {
10   ...
11 }
```

Listing 11 – Exemple adaptation de l'héritage

5.4 Adaptation des associations

L'annotation `@JoinColumn` qui s'applique aux attributs utilisés pour les associations, permet de modifier le nom par défaut de la colonne associée grâce à l'attribut `name` et permet de préciser la cardinalité min (0 ou 1) grâce à l'attribut booléen `nullable`. L'exemple du listing 12 a pour effet de créer une clé étrangère non nulle de nom `ref_logiciel` dans la table `installation` (ligne 6).

```
1  package polyJPA;
2
3  import jakarta.persistence.* ;
```

```
4
5  @Entity public class Installation implements java.io.Serializable {
6    @JoinColumn(name="ref_logiciel", nullable=false)
7    @ManyToOne private Logiciel logiciel ;
8    ...
9  }
```

Listing 12 – Adaptation des associations

Lorsqu'une table-jointure est générée pour représenter l'association, on utilisera l'annotation `@JoinTable`. Cette annotation permet de définir le nom de la table ainsi que les deux noms de colonnes qui sont des clés étrangères vers les deux tables qui composent l'association. Le listing 13 fournit un exemple (lignes 8-10). La table-jointure créée par cette annotation est décrite dans la table 6.

```
1  package polyJPA;
2
3  import jakarta.persistence.* ;
4
5  @Entity
6  public class Salle implements java.io.Serializable {
7    ...
8    @JoinTable(name="asso_salle_formations",
9               joinColumns=@JoinColumn(name="ref_salle"),
10              inverseJoinColumns = @JoinColumn(name = "ref_formation"))
11   @ManyToMany private Set<Formation> formations ;
12   ...
13 }
```

Listing 13 – Adaptation de la table-jointure

ref_salle	**ref_formation**
primary key : (ref_salle,ref_formation)	
foreign key : salle	foreign key : formation
integer	integer

TABLE 6 – Table asso_salle_formations (association *-*)

6 Conclusion

Rappelons que ce document est un **extrait** de la norme JPA. Ainsi, toutes les caractéristiques des annotations présentées ne sont pas étudiées. De même, il existe d'autres annotations qui permettent d'assouplir les liens entre objets Java et données relationnelles. Citons en exemple la possibilité de relier une classe Java à une ou plusieurs tables reliées par jointure.

Ce document ne traite également que le mapping objet-relationnel. La spécification JPA décrit également le comportement du gestionnaire d'entités (`EntityManager`) qui propose des fonctionnalités pour manipuler des entités Java (ajout, modification suppression, recherche). Ce gestionnaire est responsable de la gestion de l'état des entités et de leur persistance dans la base de données.