Qualité des données gérées avec un ORM. De UML au DDL, en passant par Java et un outil d'ORM

Marie Christine Lafaye, Georges Louis, Antoine Wiedemann

Qualité des données gérées avec un ORM.

De UML au DDL, en passant par Java et un outil d'ORM

Marie-Christing Lafaye, Georges Louis, Antoine Wiedemann

Laboratoire L3i
Université de La Rochelle
15 rue de Vaux de Foletier
17051 LA ROCHELLE CEDEX 01
{mclafaye,glouis,awiede01}@univ-lr.fr

RÉSUMÉ. *Nous montrons comment, à l'aide de l'ingénierie dirigée par les modèles, assister un concepteur de base de données lors du pilotage d'un outil ORM. Le concepteur utilise l'outil ORM pour dériver de son modèle du domaine une implémentation Java et un schéma relationnel supportant la persistance des objets du programme. Notre assistance vise à obtenir un schéma relationnel contenant toutes les contraintes présentes dans le modèle du domaine, et ainsi faire en sorte que les données stockées soient cohérentes et donc restent utilisables même après la disparition de l'application qui les a créées.*

ABSTRACT. *We show how to support, using model driven engineering, a database designer during the monitoring of an ORM tool. The designer uses ORM to derive from his domain model a Java implementation and a relational schema enabling application's objets persistency. Our monitoring aims to get a relational schema holding every domain model contraints, hence ensuring that data remains usable after the application diseppear.*

MOTS-CLÉS : *ORM, base de données, intégrité des données, modèle du domaine, IDM, UML*

KEYWORDS: *ORM, database, data integrity, domain model, MDE, UML*

1re soumission à *INFORSID - Atelier ERTSI*, le 5 juin 2009

1. Introduction

Le modèle du domaine (Larman, 2003), noté ci-après MD est un modèle central au cours du développement des systèmes d'information, car il décrit les concepts du métier et pas une application particulière (Ambler, 2004). Dans un modèle du domaine, le type d'un attribut ne doit pas être un concept complexe du domaine, implanté par une référence dans un langage à objets. Il est possible d'y utiliser non seulement les types primitifs d'UML (Integer, Boolean, String, UnlimitedNatural) mais aussi d'autres types courants comme Date, CodePostal, Adresse, types énumérés …. Il est comparable à un diagramme E/R (Chen, 1976) a contrario des modèles objets dont les structures de données sont plus complexes (Cauvet *et al.,* 1994). Il s'agit d'un modèle multi-usages souvent utilisé comme support dans la conception des bases de données relationnelles. Nous voulons aider le concepteur à dériver d'un MD une implémentation Java des classes persistantes, un mapping ORM (Object Relational Mapping), et surtout le schéma relationnel qui en résulte. Nous faisons l'hypothèse que la cohérence des données stockées est garantie car d'une part nous déduisons des multiplicités des associations et des hiérarchies de spécialisation/généralisation du MD les contraintes référentielles, et d'autre part nous prenons en compte toutes les contraintes d'unicité collectées par le concepteur.

De nombreuses approches produisent le schéma relationnel à partir d'un diagramme E/R. Cependant, elles ne permettent pas la génération de code SQL acceptant les valeurs nulles (Codd, 1986). De plus UML est aujourd'hui largement répandu et nous souhaitons permettre au concepteur de travailler dans ce langage. On trouve dans la littérature de nombreux patrons de conception du schéma relationnel à partir d'un MD UML, mais la transformation n'est pas automatisée et la prise en compte exhaustive des contraintes d'unicité non assurée (Dorsey *et al.,* 1998, Soutou, 2002, Urban *et al.,* 2005). Les outils d'ORM, exploitent ces patrons (Hibernate, 2009), (Sun 2009), (Fowler, 2003). Ils cachent la base de données sous-jacente. La qualité du schéma relationnel dépend de la spécification du mapping. Nous proposons d'utiliser les techniques d'ingénierie dirigée par les modèles (IDM) pour obtenir automatiquement ce mapping.

Dans la section 2 nous présentons les principes de fonctionnement des outils ORM. Dans la section 3 nous décrivons notre proposition d'utilisation de l'IDM pour la transformation d'un MD en un schéma relationnel. Nous avons expérimenté l'outil ORM Hibernate, et nous détaillons dans la section 4 le mode d'utilisation des annotations disponibles pour que cet outil génère automatiquement le DDL cible souhaité. La section 5 est la conclusion.

2. Object Relational Mapping (ORM)

Le but d'un outil d'ORM est de fournir au développeur d'applications en langage à objet un service de persistance dans une base de données relationnelle. Faire fonctionner un application Java avec un support persistant relationnel suppose de résoudre

les problèmes posés par les différences entre le modèle objet de la programmation et le modèle relationnel. Il faut par exemple rendre cohérentes les façons dont l'un et l'autre des paradigmes mettent en œuvre l'identification des individus. Les objets voient leur identification gérée de manière transparente par le système, tandis que les tuples relationnels sont identifiés explicitement par la valeur des attributs composant leur clé primaire. L'association dans UML et la référence de la programmation objet s'appuient sur ces identifiants implicites, au contraire de la dépendance de référence du modèle relationnel qui repose sur des identifiants explicites.

Pour exécuter les ordres SQL insert, select, update et delete (CRUD) requis par les opérations CRUD sur les objets, les solutions ORM utilisent l'API Java Database Connectivity (JDBC). (Fowler, 2003) propose d'inclure le code SQL soit directement dans la classe du domaine (patron Active Record), soit dans une classe annexe à la classe du domaine (patron Data Mapper ou patron Data Access Object (DAO) de Sun (Sun 2002)). Ce mapping fixe la correspondance entre les attributs des classes Java et les colonnes du schéma relationnel. Il peut être utilisé pour générer le DDL de la base de données, puis il pilote l'instanciation des objets et l'enregistrement des tuples par l'outil ORM (matérialisation/dématérialisation des objets de l'application (Larman, 2003)).

La plateforme Java EE 5 défini l'API Java Persistence (JPA) comme standard pour l'ORM (EJB 2006). JPA exploite des solutions présentes dans les plateformes de persistance répandues sur le marché comme Hibernate, Orable TopLink, Java Data Objects (JDO), et la persistance gérée par conteneur des Enterprise JavaBeans (EJB). L'API définit un modèle de persistance standard pour les applications Java. JPA admet que les métadonnées de mapping puissent être fournies de deux manières : soit par un fichier XML, soit par des annotations directement sur le code Java. Hibernate, développé depuis 2001, implante complètement JPA. Mais il possède un jeu d'annotations supplémentaire sur le code Java : l'API *Hibernate Annotations* (Hibernate, 2009). Ces annotations restent moins puissantes qu'un document XML pour spécifier un mapping.

3. Ingénierie dirigée par les modèles pour la conception du schéma relationnel à partir du modèle du domaine

Nous détaillons dans la figure 1 les différentes étapes du processus de conception assistée que nous proposons. Trois espaces techniques (Bézivin, 2006) se côtoient. Dans l'espace UML le concepteur construit manuellement un MD. L'utilisation de stéréotypes UML permet de sélectionner les classes dont les objets seront persistants. Nous fournissons des annotations spécifiques pour exprimer les contraintes d'unicité sur le MD (un stéréotype « UC » sur une contrainte UML associée à une classe persistante). Nous tentons de garder ces annotations aussi légères et ergonomiques que possible. Ce MD annoté, dont un exemple est fourni figure 3, est la variable d'entrée des différentes transformations de modèle opérées dans l'espace IDM. Les règles de transformation que nous utilisons sont les suivantes :

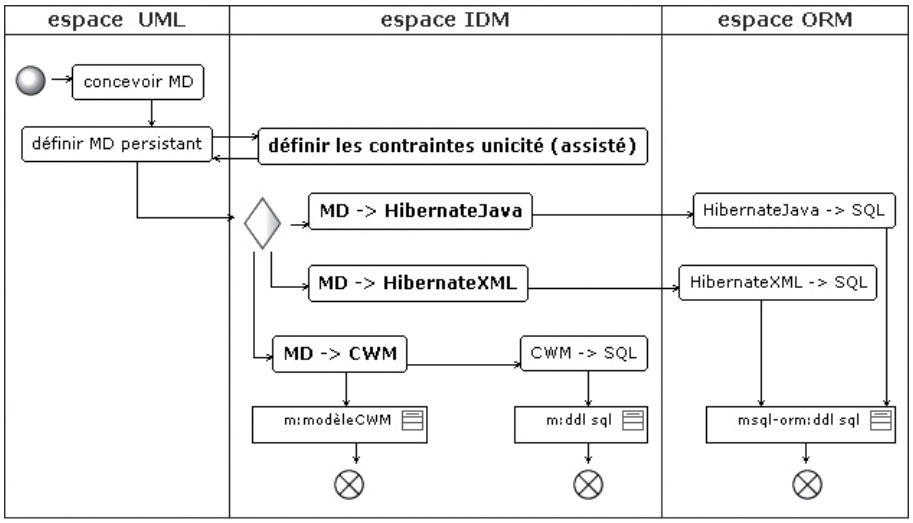


Figure 1 – Diagramme d'activité du processus de conception, depuis le modèle du domaine jusqu'aux artefacts relationnels

1) Chaque classe persistante devient une table. Pour obtenir un DDL avec des contraintes plus lisibles, nous utilisons un alias constitué des 4 premières lettres du nom de la classe, en majuscules. Cet alias sera aussi utilisé pour construire les noms des clés primaire des tables du schéma, et les attributs correspondants dans les classes Java.

2) Chaque attribut de type primitif devient une colonne sur laquelle une contrainte NOT NULL est définie par défaut. Nous acceptons les types primitifs UML et SQL (notamment Date).

3) Chaque association binaire est transformée soit en une clé étrangère, soit en une table de jointure avec deux clés étrangères (cf table 1). En fonction des multiplicités de l'association, il est nécessaire de créer des contraintes NOT NULL, UNIQUE et des triggers.

4) Chaque association n-aire devient une table de jointure composée de n clés étrangères.

5) La transformation des classes association s'effectue par l'application des règles 1 et 3 ou 1 et 4 ci-dessus.

6) Chaque généralisation est transformée en une clé étrangère (qui est aussi clé primaire) dans chaque spécialisation (ceci s'applique à l'héritage multiple).

7) Chaque contrainte « UC » (UML) est transformée en une contrainte UNIQUE (SQL).

8) Par défaut, l'identification par valeur des instances est réalisée en adoptant le patron de conception « Identity Field »(Fowler, 2003). Un attribut artificiel « id_<alias

classe> » utilisé comme clé primaire est automatiquement géré par le SGBD (surrogate key). Dans certains cas, la clé primaire d'une table est construite autrement. Dans l'exemple de la figure 3, les spécialisations Teacher et Student héritent de l'identifiant de la généralisation Person. La classe association ternaire Eval possède un identifiant construit par la concaténation des clés primaires des trois classes associées. Enfin, compte tenu de la multiplicité [0..1-1..1] de l'association Training—Defence, la classe Defence utilise la clé étrangère référençant Training comme clé primaire. Quand une contrainte d'unicité définie par le concepteur est bâtie sur plusieurs attributs, nous pensons qu'elle ne constitue pas un bon candidat à l'identification. Sur la classe Training, la contrainte d'unicité `{trainee, start}` est composée. Ici, `trainee` est un rôle et a pu être utilisé par le concepteur pour la construction de sa contrainte en raison des multiplicités de l'association Training—Student.

Nous obtenons deux types de résultats : une représentation CWM d'un schéma relationnel (Lafaye *et al.,* 2007, Wiedemann, 2008), ou un mapping ORM spécifié au choix sous forme d'un fichier XML ou d'annotations Java. Les règles de transformation que nous appliquons sont identiques dans les deux cas.

Dans cet article, nous détaillons les mappings qui serviront d'entrée dans l'espace ORM à la transformation permettant d'obtenir *in fine* le DDL cible. Ce DDL, équivalent à celui représenté dans le modèle CWM, contient la description des tables et des contraintes d'intégrité : NOT NULL, PRIMARY KEY, UNIQUE, FOREIGN KEY. . .et minimise le nombre d'identifiants systèmes (surrogate key).

4. Spécification du mapping Hibernate

Nous illustrons notre travail à l'aide du modèle du domaine présenté figure 3, dans lequel un stage est suivi par un étudiant, et donne lieu à une soutenance. Plusieurs critères sont utilisés pour évaluer un stage, chacun d'entre eux fait l'objet d'une note attribuée par un enseignant. Le concepteur a rajouté sur ce modèle des contraintes d'unicité qui seront exploitées dans le modèle relationnel. Tous les attributs de la base de données sont NOT NULL, sauf les attributs Training.end et Training.mark à cause de la multiplicité [0..1] dans le MD UML. Dans la figure 4, le modèle du domaine UML est implémenté en Java. Le mapping Hibernate y est ajouté par des annotations sur le code Java, et par un complément en XML. Le DDL obtenu est présenté figure 2.

4.1. *Traitement d'une classe*

Le traitement d'une classe du modèle du domaine comprend toujours celui de ses attributs et de son identification. L'annotation JPA `@Entity` indique qu'une classe doit être persistante (classe Training des lignes 34 à 54 de la figure 4).

Dans le MD, les attributs possèdent par défaut une multiplicité [1..1]. Notre mapping ajoute donc par défaut l'annotation JPA `@Column{nullable=false}`. Nous ne

Tables :

```
create table Criterion ( ... );
create table Eval (
  criterion int8 not null,
  teacher int8 not null,
  training int8 not null,
  mark int8 not null,
  primary key (criterion,
    teacher, training) );
create table Person ( ... );
create table Student (
  no int4 not null,
  id_PERS int8 not null,
  primary key (id_PERS),
  unique (no) );
create table Teacher ( ... );
create table Training (
  id_TRAI int8 not null,
  "end" timestamp,
  mark int4,
  start timestamp not null,
  topic varchar(255) not null,
  trainee int8 not null,
  primary key (id_TRAI),
  unique (trainee, start) );
create table Defence (
  training int8 not null,
  date timestamp not null,
  room varchar(255) not null,
  mark float4,
  primary key (training) );
```

Dépendances de référence :

```
alter table Eval
  add constraint FK_TRAI
  foreign key (training)
  references Training;
alter table Eval
  add constraint FK_CRIT
  foreign key (criterion)
  references Criterion;
alter table Eval
  add constraint FK_TEAC
  foreign key (teacher)
  references Teacher;
alter table Student
  add constraint
    FKF3371A1B9D5CEDAD
  foreign key (id_PERS)
  references Person;
alter table Teacher
  add constraint
    FKD6A63C29D5CEDAD
  foreign key (id_PERS)
  references Person;
alter table Training
  add constraint FK_STUD
  foreign key (trainee)
  references Student;
alter table Defence
  add constraint FK_TRAI
  foreign key (training)
  references Training;
```

Figure 2 – DDL PostgreSQL cible du schéma relationnel

l'ajoutons pas dans les cas de multiplicité d'attribut [0..1] (exemple `Training.end` ligne 45 figure 4). Pour garder la possibilité d'avoir dans l'application des instances Java dont les attributs NULL n'ont pas de valeur, nous utilisons des types objet et non des types primitifs. Par exemple, `Training.mark` est déclaré du type objet Java `Integer`, et non pas du type primitif `int` (ligne 46) et `Eval.mark` (ligne 80).

4.2. *Traitement de l'identification et des contraintes d'unicité*

Pour transformer les contraintes UML « UC » en contraintes SQL UNIQUE conformément à la règle 7 de la section 3, le mapping Hibernate contient des annotations @UniqueConstraint (lignes 19 et 34 de la figure 4).

Conformément à la règle 8 de la section 3, l'attribut artificiel sur lequel est définie la clé primaire est aussi ajouté à la classe Java, avec les annotations JPA `@Id` et `@GeneratedValue` (exemple `id_TRAI` lignes 38 et 39 figure 4). Pour définir une clé primaire composée, il est nécessaire de définir une classe interne `Id` et d'indiquer son rôle d'identifiant grâce aux annotations `Embeddable` (exemple Eval lignes 55 à 80).

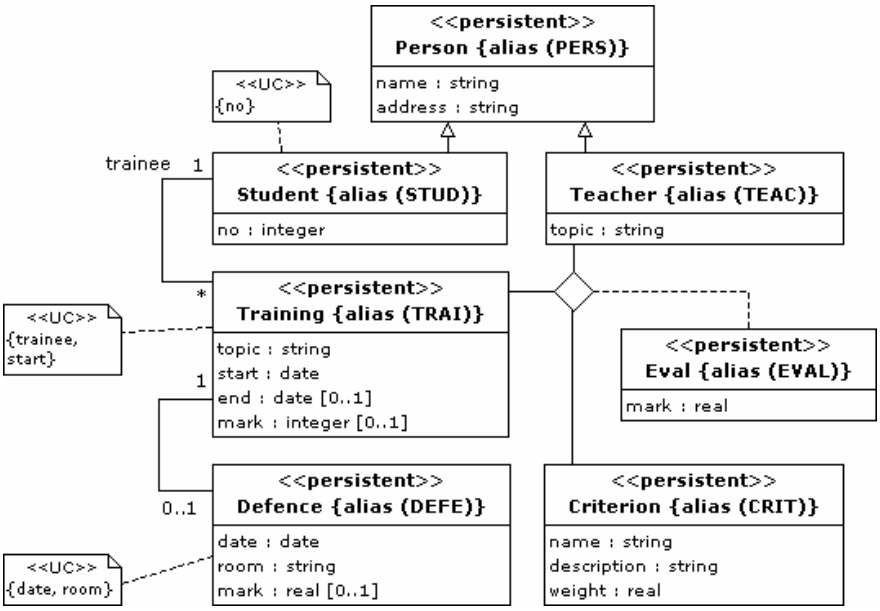


Figure 3 – Exemple de modèle du domaine annoté

Pour que la classe *Defence* utilise comme clé primaire la clé étrangère référençant *Training*, il faut utiliser le mapping XML des lignes 83 et 89 de la figure 4.

4.3. *Traitement des associations*

Le traitement des associations comprend la définition de références Java adéquates dans le code des classes métier, et le mapping Hibernate. Nous distinguons trois familles d'associations binaires : 1—1, 1—N et N—M, comme c'est indiqué dans la première colonne de la table 1. La troisième colonne indique dans quelle table la clé étrangère représentant l'association est ajoutée. Par exemple, pour la multiplicité [0..1-0..1] le concepteur peut choisir d'implanter la clé étrangère soit dans la table obtenue par transformation pour la classe A soit dans la table pour la classe B. Pour les associations de type N—M, il est nécessaire de rajouter un table supplémentaire (table de jointure), avec deux clés étrangères. La quatrième colonne précise le cas où la clé étrangère admet des valeurs nulles, et la cinquième le cas où il nécessaire de créer une contrainte UNIQUE sur la clé étrangère. Les septième, et huitième colonnes détaillent le code et les annotations Hibernate qui doivent être ajoutés aux classes Java pour obtenir le DDL souhaité.

Remarque 1 : navigabilité des associations UML
L'association UML de type 1—N (Training—Student par exemple) est l'association la plus communément rencontrée en modélisation. En UML, elle est implicitement

bidirectionnelle, de même, dans le modèle relationnel, puisque la jointure est commutative. La référence Java n'autorise cependant la navigation que dans un sens, de la classe qui possède la référence vers la classe référencée. Représenter le double sens de navigation en Java oblige à définir deux références mutuelles dans `Training` et `Student`. Pour tenir compte de la cardinalité maximum `*` du côté de Training dans le modèle métier UML, la référence `Student.trainings` est déclarée comme une collection. Mais contrairement à UML, où les deux extrémités sont liées par l'association à laquelle elles appartiennent, il n'en est rien en Java : les deux références définissent à elle deux non pas *une* association *bidirectionnelle* Training $\longleftrightarrow$ Student, mais *deux* associations *unidirectionnelles*, de sens opposé, Training $\longrightarrow$ Student et Training $\longleftarrow$ Student. L'implémentation Java de la figure 4 ne contraint pas une instance `t` de `Training` appartenant à la collection `s.trainings` d'une instance de `Student` d'avoir sa référence `t.trainee` égale à `s`. Au mapping de la référence `Training.trainee` par l'annotation `ManyToOne`, lignes 47 à 51, et au mapping de `Student.training` par l'annotation `OneToMany`, ligne 25, il est nécessaire d'ajouter la propriété `mappedBy="trainee"` pour préciser que les deux références Java représentent les deux extrémités d'une même association.

Remarque 2 : cardinalités minimum
Nous proposons de traiter les cardinalités minimum à 1 des multiplicités 1—1 en ajoutant une contrainte supplémentaire, qu'Hibernate prend en charge avec la propriété `nullable = false`. La classe `Training` avec l'annotation JPA `@JoinColumn`, et sa propriété `nullable = false`, ligne 49 précise qu'une instance de Training *doit* être reliée à une instance de Student. Pour les cardinalités minimums à 1 des multiplicités 1—N il faudrait ajouter un trigger dans le code SQL (Hibernate n'en génère pas).

Remarque 3 : Associations [1..1-1..1]
Pour nous la meilleure implantation relationnelle de ces associations est une table résultante de la fusion des deux classes, et de la construction sur cette table de deux vues, recréant par projection les deux entités initiales. La BCNF (Codd, 1974) sera ainsi respectée. JPA et Hibernate permettent l'obtention d'un table unique pour deux classes Java, mais pas la définition des vues.

Remarque 4 : Association [0..1-1..1]
Pour pouvoir faire en sorte que la clé étrangère soit aussi clé primaire (exemple : Defence), nous devons utiliser un mapping XML comme celui des lignes 82 à 89 de la figure 4.

4.4. *Généralisations/Spécialisations*

Plusieurs stratégies sont proposées dans la littérature (Ambler, 2002, Soutou, 2002, Urban *et al.,* 2005) pour implanter une hiérarchie de spécialisations/généralisations dans un schéma relationnel. Hibernate permet trois d'entre elles : une table par hiérarchie, une table par classe et une table par spécialisation. Nous choisissons la stratégie une table par classe par l'attribut `strategy` de l'annotation `Inheritance`, ligne 13 de la figure 4.

```
@Entity public class Criterion {
  ...
}
//mapping fourni en XML dans un
// fichier séparé
public class Defence {
  long id_DEFE;
  Date date;
  Float mark;
  String room;
  Training training;
}
@Entity @Inheritance( strategy =
  InheritanceType.JOINED )
public class Person {
 ...
}
@Entity @Table(uniqueConstraints =
  {@UniqueConstraint(
    columnNames = {"no"})})
public class Student
  extends Person {
  @Column(nullable=false)
  Integer no;
  @OneToMany(mappedBy="trainee")
  Set<Training> trainings =
    new HashSet<Training>(0);
}
@Entity public class Teacher
  extends Person {
  @Column(nullable=false)
  String topic;
}
@Entity @Table(uniqueConstraints =
  {@UniqueConstraint(columnNames =
    { "trainee" , "start" })})
public class Training {
  @Id @GeneratedValue
  long id_TRAI;
  @Column(nullable=false)
  String topic;
  @Column(nullable=false)
  Date start;
  @Column(name = "`end`")
  Date end;
  Integer mark;
  @ManyToOne
  @JoinColumn(name="trainee",
    nullable=false)
  @ForeignKey(name="FK_STUD")
  Student trainee;
  @OneToOne(mappedBy = "training")
    Defence defence;
}
@Entity public class Eval {
  @Embeddable
  public static class Id
    implements Serializable {
    @Column(name = "teacher")
    private long id_TEAC;
    @Column(name = "criterion")
    private long id_CRIT;
    @Column(name = "training")
    private long id_TRAI; }
  @EmbeddedId
  private Id id = new Id();
  @ManyToOne
  @ForeignKey(name = "FK_TEAC")
  @JoinColumn(name = "teacher")
  Teacher teacher;
  @ManyToOne
  @ForeignKey(name = "FK_CRIT")
  @JoinColumn(name = "criterion")
  Criterion criterion;
  @ManyToOne
  @ForeignKey(name = "FK_TRAI")
  @JoinColumn(name = "training")
  Training training;
  @Column(nullable=false)
  long mark;
}
```

Fichier de mapping XML de la classe `Defence` :

```
<hibernate-mapping default-access="field" ><class name="Defence">
  <id name="id_DEFE"><generator class="foreign"><param name="property">
    training</param></generator></id>
  <property name="date" not-null="true"/>
  <property name="room" not-null="true"/>
  <property name="mark"/>
  <one-to-one name="training", foreign-key="FK_TRAI",
    constrained="true"/></class></hibernate-mapping>
```

Figure 4 – Implémentation Java et mapping Hibernate des classes du domaine

5. Conclusion

Lorsque plusieurs applications vont partager un même modèle du domaine, l'effort nécessaire pour définir ce modèle et pour lui donner une implémentation de qualité est tout-à-fait justifié. Nous avons montré ici une technique pour obtenir ce résultat en utilisant les patrons de conception et les outils habituellement appliqués pour le passage d'un modèle UML à des classes Java dont la persistance est assurée par une base de données relationnelle (ORM). Un des aspects sur lesquels nous insistons est la qualité du schéma relationnel, et l'obtention des contraintes qui assureront la cohérence des données enregistrées.

Notre proposition peut se résumer ainsi :

1) Annoter le modèle UML des données persistantes, en définissant des contraintes d'unicité, puis

2) Transformer systématiquement le modèle UML en classes Java, annotées de manière à

3) Obtenir un schéma relationnel de qualité, avec un maximum de contraintes d'intégrité.

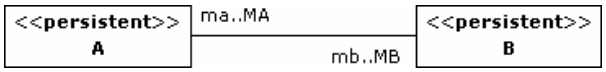
La qualité du modèle relationnel obtenu se caractérise comme suit :

1) Les valeurs nulles, dans leur variété inapplicable au sens de (Codd, 1986), sont utilisées pour limiter le nombre de tables de jointure (les seules associations binaires qui donnent lieu à une table sont les associations N-N). Ceci limite le nombre de jointures à l'exécution.

2) Les contraintes d'unicité collectées sur le modèle UML sont traduites en contraintes d'unicité (UNIQUE) dans le schéma relationnel.

3) On évite la multiplication de clés artificielles inutiles. C'est notamment le cas pour le traitement des généralisations/spécialisations d'UML.

Le traitement des valeurs nulles « inapplicables » est lié à la dynamique des objets. Spécifier cette dynamique au niveau du modèle UML devrait permettre de compléter notre outillage afin d'introduire la vérification automatique de nouvelles contraintes sur la base de données, qui assureraient que les valeurs nulles dans les tables restent bien en permanence cohérentes avec l'état des instances du modèle.



multiplicités		DDL cible				Annotations Hibernate dans Java	
[ma..MA-mb..MB]		clé étrangère dans	FK is NULL	FK UNIQUE	trigger dans	dans la classe Java A	dans la classe Java B
1—1	[0..1-0..1]	A ou B	✓	✓		`@OneToOne @JoinColumn ( unique = true ) B b ;`	`@OneToOne(mappedBy="b") A a ;`
	[0..1-1..1]	A		✓		`@OneToOne @JoinColumn ( nullable = false, unique = true ) B b ;`	
	[1..1-1..1]	Cas particulier : une seule table et deux vues				Impossible avec Hibernate	
1—N	[0..1-0..*]	B	✓			`@OneToMany(mappedBy="a") Set<B> b ;`	`@ManyToOne @JoinColumn( unique=true) A a ;`
	[0..1-1..*]	B	✓		A		
	[1..1-0..*]	B					`@ManyToOne @JoinColumn ( unique = true, nullable = false) A a ;`
	[1..1-1..*]	B			A		
N—M	[0..*-0..*]	table de jointure				`@ManyToMany ( mappedBy = "a" ) Set<B> b ;`	`@ManyToMany @JoinTable Set<A> a ;`
	[0..*-1..*]				A		
	[1..*-1..*]				A et B		

Tableau 1 – DDL cible et annotations Hibernate pour les associations binaires

6. Bibliographie

Ambler S., « Mapping Objects to Relationnal Databases : O/R Mapping in detail », *http ://www.agiledata.org/essays/mappingObjects.html*, 2002.

Ambler S., *The Object Primer : Agile Model-Driven Development with UML 2.0*, Cambridge University Press, 2004.

Bézivin J., « Model Driven Engineering : An Emerging Technical Space », *Generative and Transformational Techniques in Software Engineering*, LNCS, p. 36-64, 2006.

Cauvet C., Djillali M., « Processus de conception orientée objet : transformation d'un schéma conceptuel en un schéma logique », *in* , INFORSID (ed.), *Actes du Xème congrès INFORSID*, 1994.

Chen P. P.-S., « The Entity-Relationship Model-Toward a Unified View of Data », *ACM Transactions on Database Systems*, vol. 1, n° 1, p. 9-36, 1976.

Codd E. F., « Recent investigations in relational database systems », *in* , N. Holland (ed.), *Information processing*, p. 1017-1021, 1974.

Codd E. F., « Missing information (applicable and inapplicable) in relational databases », *SIGMOD Rec.*, vol. 15, n° 4, p. 53-53, 1986.

Dorsey P., Hudicka J. R., *Oracle8 Database Design Using UML Object Modeling*, Oracle Press, 1998.

EJB 3.0 Expert Group, « JSR 220 : Enterprise JavaBeansTM,Version 3.0 », May 2, 2006.

Fowler M., *Patterns of Enterprise Application Architecture*, Addison-Wesley Professional, 2003.

Hibernate, « Hibernate Annotations - Reference Guide », 2009.

Lafaye M. C., Lafaye J. Y., Louis G., Wiedemann A., « L'approche MDA pour la conception des bases de données relationnelles, une illusoire simplicité », *Actes des 3èmes journées sur l'Ingénierie Dirigée par les Modèles*, vol. 1, p. 167-181, 2007.

Lafaye M. C., Lafaye J. Y., Paillard J. P., Téouri S., « Qualité de conception des bases de données relationnelles avec UML », *Ingénierie des Systèmes d'Information*, vol. 9, n° 5-6, p. 69-101, 2004.

Larman C., *UML 2 et les Design Patterns*, 2ème edn, CampusPress, 2003.

Soutou C., *De UML à SQL*, Eyrolles, 2002.

Sun Microsystems, Inc, « Desing Patterns : Data Access Object », 2002.

Sun Microsystems, Inc, « Java Data Objects (JDO) », 2009.

Urban S. D., Dietrich S. W., *An advanced course in database systems, beyond relational databases*, Prentice Hall, 2005.

Wiedemann A., « Approche MDA pour la transformaion d'un modèle UML en schéma relationnel », *Actes du XXVème congrès INFORSID, forum jeunes chercheurs*, Association INFORSID, p. 587-588, 2007.

Wiedemann A., « Relational Database Modeling », *MODELS'08*, Doctoral Symposium, Springer, LNCS, Toulouse, 2008.