REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

N° d'ordre : .....

UNIVERSITE DE M'SILA

FACULTE DES MATHEMATIQUES ET DE L'INFORMATIQUE

Département d'Informatique

MEMOIRE de fin d'étude

Présenté pour l'obtention du diplôme de MASTER

Domaine : Mathématiques et Informatique
Filière : Informatique
Spécialité : Systèmes d'Informations Avancés

Par : AMOUR Nasreddine

SUJET

Une approche MDA pour la transformation des diagrammes de classe UML vers une base de données Basant sur l'outil EMF

Soutenu publiquement le : / /2014 devant le jury composé de :

……	**Université de M'sila**	**Président**
Mr. MOKHTARI Rabah	**Université de M'sila**	**Rapporteur**
……	**Université de M'sila**	**Examinateur**
……	**Université de M'sila**	**Examinateur**

Promotion : 2013 /2014

Dédicaces

A mes parents

Je vous dois ce que je suis aujourd'hui grâce à votre amour, à votre patience et vos innombrables sacrifices. Que ce modeste travail, soit pour vous une petite compensation et reconnaissance envers ce que vous avez fait d'incroyable pour moi.

Que Dieu, le tout puissant, vous préserve et vous procure santé et longue vie afin que je puisse à mon tour vous combler.

A mes très chers frères

Aucune dédicace ne serait exprimer assez profondément ce que je ressens envers vous, je vous dirais tout simplement, un grand merci, je vous aime.

A mes très chers amis

En témoignage de l'amitié sincère qui nous a liées et des bons moments passés ensemble. Je vous dédie ce travail en vous souhaitant un avenir radieux et plein de bonnes promesses.

AMOUR NASREDDINE

Remerciements

En tout premier lieu, je remercie Allah le tout puissant, à la sagesse et au savoir infinis, « Gloire à toi ! Nous n'avons de savoir que ce que Tu nous as appris. Certes c'est Toi l'Omniscient, le sage, le tout miséricordieux le très miséricordieux » (Sourate al-Baqarah, verset 32).

Je tiens à remercier mon encadreur M[r] Mokhtari Rabah pour le grand honneur qu'il m'a fait en me proposant le sujet de ce mémoire de fin d'étude. J'ai eu l'honneur et le privilège de travailler sous son assistance et de profiter de ses qualités humaines, professionnelles et de sa grande expérience, il m'a guidé tout au long de ce travail. L'élaboration avec amabilité et dynamisme le caractérisant. Que ce modeste travail puisse satisfaire mes examinateurs, pour qu'ils en témoignent ma gratitude et reconnaissance pour l'aide et les conseils qu'il ma prodigué, ainsi que pour la savoir qu'il ma inculqué.

Je remercie tous mes enseignants de l'université de M'sila.

Mes remerciements vont également aux membres de jury d'avoir accepté de juger mon travail.

Je remercie vivement toute ma famille, en particulier mes parents, pour m'avoir toujours soutenu au cours de mes études. Qu'ils trouvent ici le fruit de leur patience et du soutien permanent qu'ils m'ont prodigué pour affronter tous les moments difficiles.

Je tiens également à remercier mon collègue Saadi Aimad pour son aide et ses conseils précieux.

Merci pour tous ceux qui, m'ont aidé de près ou de loin à réaliser ce travail.

Table des Matières

Introduction générale 1

Chapitre 1 – IDM et L'approche MDA et les outils de modélisation et de transformation

1.1 Introduction……..4
1.2 Principes généraux de l'IDM……..…….4
1.3 OMG et MDA ……...5
1.4 Définition des concepts fondamentaux ……..5
1.5 Les approches de l'Ingénierie dirigée par les modèles ……7
1.5.1 L'Architecture Dirigée par les Modèles (MDA) ……7
1.5.1.1 Architecture MDA à quatre niveaux ……8
1.5.2 Les standards de L'OMG ……9
1.5.2.1 MOF : Meta Object Facility ……....9
1.5.2.2 UML: Unified Modeling Language ……...10
1.5.2.3 OCL: Object Constraint Language ……10
1.5.2.4 XMI : Xml Meta data Interchange ……10
1.6 Modèles de MDA ……11
1.6.1 Modèle d'exigence CIM (Computation Independent Model) ……...11
1.6.2 PIM (Platform Independent Model) ……..11
1.6.3 PSM (Platform Specific Model) ……11
1.6.4 PDM (Platform Description Model) ……11
1.7 La transformation de modèle ……12
1.7.1 Définition d'une transformation de modèle ……..12
1.7.2 Généralités ……13
1.8 Standards et langages pour la transformation de modèle ……15
1.8.1 ATL : ATLAS Transformation Language ……...15
1.8.2 Description du langage ……15
1.8.3 Transformations ……...15
1.9 Outils supporté pour la transformation des modèles ……16
1.9.1 Introduction ……16
1.10 Plate-forme d'accueil ……17

1.10.1 Généralité sur Eclipse ……17
1.10.2 Eclipse plate-forme ……17
1.11 Manipuler des modéles avec EMF ……..17
1.11.1 Eclipse Modeling Framework (EMF) ……17
1.11.2 Objectif d'EMF ……...18
1.12 EMF and MDA ……19
1.12.1 Le méta-méta-modèle d'EMF ……19
1.12.2 Le méta-modèle Ecore ……20
1.12.3 Les interfaces réflectives d'EMF ……21
1.13 GMF& GEF ……..22
1.14 Topcased ……22
1.15 ATOM3 ……22
1.16 Conclusion ……23

Chapitre 2 – Les langages de Modélisation

2.1 Introduction ……25
2.2 Le Langage UML ……...25
2.2.1 Définition d'UML ……26
2.2.2 La modélisation UML ……...26
2.2.3 Diagramme de classe ……26
2.3 Généralites sur les bases de données ……34
2.3.1 Introduction ……...34
2.3.2 Qu'est-ce qu'une base de données ? ……..34
2.3.3 Critères d'une base de données ……35
2.3.4 Utilité d'une base de données ……...35
2.3.5 Avantages de la base de données ……36
2.3.6 Sécurité et confidentialité de la base de données ……36
2.3.7 Niveau de description des données ANSI/SPARC ……..36
2.3.8 Conception de Base de Données ……..37
2.3.8.1 MERISE ……38
2.3.8.2 Le modèle relationnel ……38
2.3.9 Normalisation ……...39
2.3.10 Le Système de Gestion de Base de Données ……..40

2.3.10.1 Modéles de SGBD ……...40
2.3.10.2 Objectif de SGBD ……40
2.3.10.3 Caractéristiques de la conception de BDD en SGBD ……..41
2.3.10.4 Méthode d'accès aux données ……41
2.3.10.5 Les principaux SGBD (système de gestion de bases de données) ……..42
2.4 Conclusion ……43

Chapitre 3 – Génération des outils pour la transformation des DC vers BDDR

3.1 Introduction ……45
3.2 Génération d'un modèle relationnel à partir des diagrammes de classe UML ……..45
3.2.1. Les formes de normalisation ……45
3.2.2 Diagramme de classe VS Modèle Relationnel ……..46
3.3 Génération d'un outil EMF pour « les diagrammes de classe » ……..47
3.3.1 Un méta-modèle pour les diagrammes de classe ……47
3.3.2 La génération d'un outil pour les diagrammes de classe ……49
3.3.3 La génération d'un outil pour la transformation d'un DC BDD ……54
3.3.4 Utilisation d'ATL ……..60
3.3.5 Le module ATL ……61
3.3.6 Requêtes et la génération de texte ……62
3.4 Transformation des diagrammes de classe vers une Base de données ……...63
3.4.1 Les règles de transformation ……63
3.5 Exemple ……...67
3.6 Conclusions à l'issu de l'expérimentation ……71
3.6.1 Les avantage d'ATL ……..71
3.6.2 Inconvénients d'ATL ……71
3.7 Travaux futurs ……72
3.7.1 Explorer les possibilités d'interfaçage ……72
3.7.2 Terminer l'extraction de règles de l'algorithme ……72
3.7.3 éliminer les circuits dans les dépendances référentielles ……..72
3.8 Conclusion ……73
Conclusion Générale 75
Bibliographie 77
Annexe 80

Table des Figures

Figure 1.1: Relations entre système, modèle, méta-modèle et langage ……....6
Figure 1.2: Aperçu de l'approche OMG de l'IDM : MDA ……...8
Figure 1.3: Les quatre niveaux d'abstraction pour MDA …….....9
Figure 1.4: Principe du Processus MDA ……12
Figure 1.5 : Processus de transformation de modèles ……..13
Figure 1.6 : Opérations de transformation sur les modèles MDA ……...13
Figure 1.7: Unification *PIM* et *PDM* pour produire le *PSM* puis le code…….....14
Figure 1.8: Un exemple de règle ATL.……...…….....15
Figure 1.9 : organisation générale d'EMF ……...18

Figure 2.1 : classe abstraite …….....27
Figure 2.2 : Classe non abstraite …….....28
Figure 2.3 : Représentation UML d'une classe. ……..28
Figure 2.4 : Exemple d'association binaire…….……30
Figure 2.5 : exemple d'association n-aire …….……...31
Figure 2.6 : Exemple de classe-association. …….……...32
Figure 2.7 : Exemple de relation d'agrégation. …….……..32
Figure 2.8 : Exemple de relation …….……33
Figure 2.9 : Exemple d'une Bases de données MySQL ……..35
Figure 2.10 : Processus de conception d'une base de données…….37
Figure 2.11 : l'accès à une BDD par plusieurs utilisateurs simultanément……..42

Figure 3.1 : un méta-modèle pour les diagrammes de classe ……...…….48
Figure 3.2 : Crée projet EMF vide. …….……...……..49
Figure 3.3 : Crée diagramme Ecore. …….…….……49
Figure 3.4 : les éléments du méta-modèle de DC. …….……..50
Figure 3.5 : génération de « genmodel »…….……..…51
Figure 3.6 : les éléments de « genmodel» de DC.…….……..……52

Figure 3.7 : génération des projets .edit, .editor, .tests ……52
Figure 3.8 : exécuter les plugins ……..…….53
Figure 3.9 : plugin détails de classe modèle …….……...53
Figure 3.10 : création d'une classe modèle …….……..…..54
Figure 3.11 : créer un nouveau projet ATL …….……54
Figure 3.12 : création des dossiers ……...55
Figure 3.13 : création les méta-modèles ……..55
Figure 3.14 : création des instances dynamiques ……56
Figure 3.15 : choisir le nom de modèle et le dossier de l'emplacement ……..57
Figure 3.16 : modèles généré ……...57
Figure 3.17 : La création des éléments de modèle à transformer ……58
Figure 3.18 : création ATL file …….……...59
Figure 3.19 : sélection le nom d'ATL file …….……..59
Figure 3.20 : sélection le type query pour fichier ATL ……..60
Figure 3.21 : ATL file généré …….……60
Figure 3.22 : Extrait du code transformation du module ATL ……61
Figure 3.23 : Requêtes et la génération de texte « .sql » ……..…….……62
Figure 3.24 : transformation de ClassModel à RDBMSModel ……..63
Figure 3.25 : générer le fichier SQL de la BDD en ATL ……...63
Figure 3.26 : transformation de Class à Une table …….……64
Figure 3.27 : transformation class vers une table …….……64
Figure 3.28 : transformation attribue à colonne …….…….……65
Figure 3.29 : transformation Attribue vers Colonne ……...65
Figure 3.30 : transformation Association à clé étrangère …….……..65
Figure 3.31 : Transformation Association vers FK + Colonne…….……...66
Figure 3.32 : exemple d'une instance d'un diagramme de classe ……...67
Figure 3.33 : Diagramme de classe simple par l'outil généré ……68
Figure 3.34 : Configuration de fichier ATL …….……...69
Figure 3.35 : Génération d'une BDD…...…….……...69
Figure 3.36 : La consultation de fichier SQL généré ……...70
Figure 3.37 : La création de BDD par l'intermédiaire phpMyAdmin ……71

Liste des tableaux

Tableau 1.1 : les cas en présence pour EMF…….20
Tableau 1.2 : l'espace technique standard d'EMF……20

Tableau 2.1 : listes des principaux SGBD ……....42

Liste des abréviations

ATL : *Atlas Transformation Langage*

ATOM3 : *A Tool for Multi-formalism Meta-Modelling*

BDDR : *Base de données* relationnelle

CIM : *Computation Independent Models*

CORBA : *Common Object Request Broker Architecture*

DC : *Diagramme de Classe*

DTD : *Document Type Definition*

EBNF : *Extended Backus-Naur Form*

EMF : *Eclipse Modeling Framework*

GEF : *Graphical Editing Framework*

GMF : *Graphical Modeling Framework*

IBM : *International Business Machines*

IDE : *integrated development environment*

IDL : *Interface description language*

IDM : *Ingénierie Dirigée par les Modèles*

LDD : *Langage de définition de données*

LMD : *Langage de manipulation de données*

MDA : *Model Driven Architecture*

MERISE : *Méthode d'étude et de réalisation informatique* pour les systèmes d'entreprise

MOF : *Meta Object Facility*

OMG : *Object Management Group*

PDM : *platform description model*

PIM : *platform independent models*

PSM : *platform specific models*

QVT : *Query – View – Transformation*

RCP : *Rich Client Platform*

SGBD : *Système de gestion de base de données*

SQL : *Structured Query Language*

SWT : *Standard Widget Toolkit*

UML : *Unified Modeling Language*

XMI : *XML Metadata Interchange*

XML : *eXtensible Markup Language*

Introduction générale

Introduction générale

Dans un contexte d'accroissement exponentiel de la complexité des systèmes informatiques, la modélisation de ces systèmes est devenue un enjeu majeur de la réussite des projets : bonne prise en compte du besoin fonctionnel, réduction des délais et des coûts par la réutilisation des conceptions et des liens avec le code et, enfin, souplesse nécessaire pour l'adaptation des applications aux différentes technologies actuelles ou futures.

Et pour faire un modèle conceptuel d'un système complexe, le diagramme de classe d'UML offre une bonne solution, le problème de la génération automatique de base de données à partir d'un diagramme de classe a déjà été largement étudié et enseigné sur le plan théorique. Souvent, on se contente d'étudier les différents cas de figure qui peuvent se présenter en termes de multiplicités des associations entre classes, et de déterminer dans chaque cas le schéma relationnel adapté pour implanter ces associations.

Pour valider les avantages de ce processus et de la méthode qu'il incarne, il a été choisi de tenter une partie de son implémentation dans le langage ATL spécialement conçu pour la transformation de modèles. Ce langage est développé dans le cadre du projet ATLAS [38] mené à l'université de Nantes par une équipe de chercheurs . Le travail présenté ici est une expérimentation et une évaluation de l'utilisation d'ATL pour réaliser une transformation automatique d'un diagramme de classe vers une base de données relationnelle on se basant sur l'outil EMF qui nous permet de construire des applications basées sur des modèles. Pour cela ce mémoire est organisé en trois chapitres :

Le chapitre 1 est consacré à l'étude des concepts de base de l'ingénierie dirigée par les modèles (IDM) et leur approche l'architecture dirigée par les modèles (MDA), suivi par la transformation des modèles et le processus de la vérification dans l'IDM, puis nous exprimons la manipulation des modèles par le standard EMF comme un outil de modélisation, et ATL comme un langage pour la transformation des modèles.

Dans le deuxième chapitre nous avons fait premièrement une présentation sur les langages de modélisation et découvert leurs composants et leurs propriétés.

Dans le troisième chapitre, nous avons adopté un mécanisme de transformation automatique de diagramme de classe à une base de données relationnelle. La méthode proposée est basée sur le langage de transformation ATL, et utilise l'outil de modélisation et de méta-modélisation EMF.

Chapitre 01 :

IDM et L'approche MDA et les outils de modélisation et de transformation

CHAPITRE 1
IDM ET l'APPROCHE MDA ET LES OUTILS DE MODELISATION ET DE TRANSFORMATION

1.1 Introduction

Au cours des dernières décennies, le développement de grands projets d'ingénierie, toujours plus complexes, a mis en évidence la nécessité de disposer d'outils, de méthodes et de processus permettant d'en assurer la maîtrise tout au long de leur cycle de vie.

L'ingénierie dirigée par les modèles (IDM), d'abord utilisée principalement dans le domaine des systèmes logiciel, a permis plusieurs améliorations significatives dans le processus de développement de systèmes complexes en se concentrant sur des préoccupations plus abstraites autour des modèles utilisés que sur la programmation classique (le code). Il s'agit donc d'une forme d'ingénierie générative dans laquelle tout ou partie d'une application est engendrée à partir de modèles. Un modèle est une abstraction, une simplification d'un système qui est suffisante, non seulement pour comprendre le système modélisé, mais également pour garantir son bon fonctionnement.

Dans ce chapitre nous présentons de façon globale l'ingénierie dirigée par les modèles (IDM). Nous abordons les principes clés de l'ingénierie IDM et de ses différentes centrées sur les modèles, ensuite nous entreprenons les approches de l'IDM et plus particulièrement l'architecture dirigée par les modèles (MDA), la transformation de modèle et son langage ATL, le standard EMF comme un outil de modélisation.

1.2 Principes généraux de l'IDM

Suite à l'approche objet des années 80 et de son principe du « tout est objet», l'ingénierie du logiciel s'oriente aujourd'hui vers l'ingénierie dirigée par les modèles (IDM) et le principe du « tout est modèle ». Cette nouvelle approche peut être considérée à la fois en continuité et en rupture avec les précédents travaux. Tout d'abord en continuité car c'est la technologie objet qui a déclenché l'évolution vers les modèles. En effet, une fois acquise la conception des systèmes informatiques sous la forme d'objets communicant entre eux, il s'est posé la question de les classifier en fonction de leurs différentes origines (objets métiers, techniques, etc.).[1]

L'IDM vise donc, de manière plus radicale que pouvaient l'être les approches des patterns et des aspects, à fournir un grand nombre de modèles pour exprimer séparément chacune des préoccupations des utilisateurs, des concepteurs, des architectes, etc. C'est par ce principe de base fondamentalement différent que l'IDM peut être considérée en rupture par rapport aux travaux de l'approche objet.

Alors que l'approche objet est fondée sur deux relations essentielles, «Instance De » et « Hérite De», l'IDM est basée sur un autre jeu de concepts et de relations. Le concept central de l'IDM est la notion de modèle pour laquelle il n'existe pas à ce jour de définition universelle. Néanmoins, de nombreux travaux s'accordent à un relatif consensus d'une certaine compréhension.

La démarche de l'IDM s'articule donc autour de cinq concepts fondamentaux : les modèles, les méta-modèles, les méta-méta-modelés, les transformations de modèles et les plateformes. [2]

1.3 OMG et MDA

L'Object Management Group est une association américaine, fondée en 1989 et compte aujourd'hui un consortium de plus de 800 entreprises du monde entier. Parmi les réalisations de l'OMG sont le Request Broker Architecture Common Object ™ (CORBA ™), Unified Modeling Language ™ (UML ™), Meta Object Facility ™ (MOF ™), IDL (Interface Definition Language), XML Metadata Interchange (XMI ™) et le méta-modèle d'entrepôt commun ™ (MCG ™). Toutes ces normes contribuent à rendre l'idée d'un modèle de développement axé sur une réalité. Environ vers 2001 OMG adopté un nouveau cadre appelé le modèle Driven Architecture (MDA). Contrairement aux autres standards de l'OMG le MDA offre un moyen d'utiliser des modèles au lieu du code source traditionnelle. Il reste à voir si cette nouvelle façon de développement logiciel sera acceptée parmi les développeurs et les entreprises. [3]

1.4 Définition des concepts fondamentaux :

a- Un modèle : en réalité il n'existe pas à ce jour de définition universelle :

"***A model*** *represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality*". [29].

"***Un modèle*** *représente un système réel en se basant sur la sémantique et les règles qui conditionnent ses éléments; en d'autres termes il ne doit en aucun cas briser la structure ou les contraintes que les éléments du système réel respectent*." [4]

b- Méta-modèle : Un méta-modèle est un modèle qui définit le langage d'expression d'un modèle, c.-à-d. le langage de modélisation. La notion de méta-modèle conduit à l'identification d'une seconde relation, liant le modèle et le langage utilisé pour le construire, appelée *conformeA*.

En cartographie, il est effectivement indispensable d'associer à chaque carte la description du « *langage*» utilisé pour réaliser cette carte. Ceci se fait notamment sous la forme d'une légende explicite. La carte doit, pour être utilisable, être conforme à cette légende.

Plusieurs cartes peuvent être conformes à une même légende, La légende est alors considérée comme un modèle représentant cet ensemble de cartes (μ) et à laquelle chacune d'entre elles doit se conformer (c). [1]

Ces deux relations permettent ainsi de bien distinguer le langage qui joue le rôle de système, du (ou des) méta-modèle(s) qui jouent le rôle de modèle(s) de ce langage.

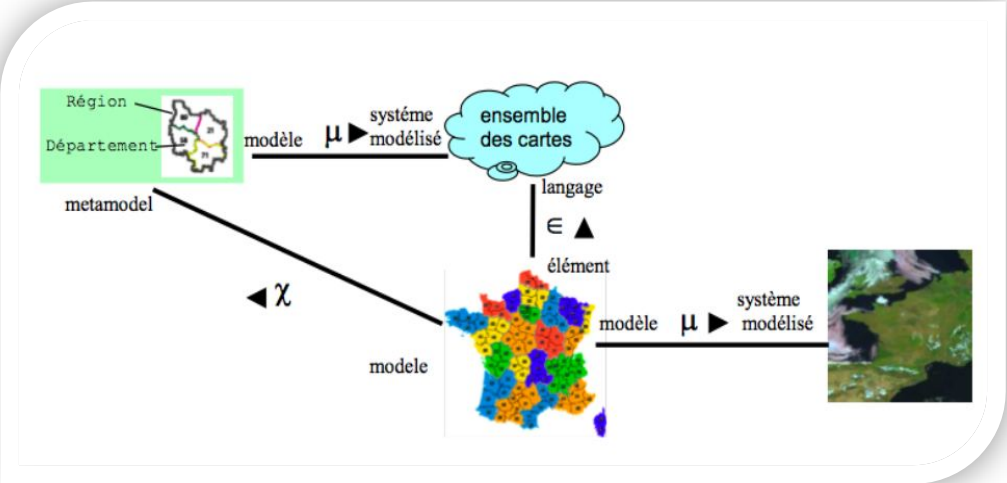

Figure 1.1: Relations entre système, modèle, méta-modèle et langage.

c- Méta-méta-modèle : Un méta-méta-modèle est un modèle qui décrit un langage de méta-modélisation, c.-à-d. les éléments de modélisation nécessaires à la définition des langages de modélisation. Il a de plus la capacité de se décrire lui-même.

C'est sur ces principes que se base l'organisation de la modélisation de l'OMG généralement décrite sous une forme pyramidale (cf. figure 3). Le monde réel est représenté à

la base de la pyramide (niveau M0). Les modèles représentant cette réalité constituent le niveau M1. Les méta-modèles permettant la définition de ces modèles (p. ex. UML) constituent le niveau M2. Enfin, le méta-méta-modèle, unique et méta-circulaire, est représenté au sommet de la pyramide (niveau M3). [4]

d- Transformation de modèles

D'après Hubert Kadima [5]:

– Une transformation de modèles est la génération d'un ou plusieurs modèles cibles à partir d'un ou plusieurs modèles sources conformément à une définition de transformation.

– Une définition de transformation est un ensemble de règles de transformation qui décrivent globalement comment un modèle décrit dans un langage source peut être transformé en un modèle décrit dans un langage cible.

e- Plate-forme

Une plate-forme est un système offrant des services nécessaires à la construction, La réalisation ou l'exécution d'autres systèmes [6].

On distingue deux types de plateformes :

Dans la pratique : les plates-formes logicielles et les plates-formes matérielles. Dans la Démarche de l'IDM, les plates-formes doivent être définies dans des modèles pour permettre leur prise en compte dans le processus de construction d'un produit.

1.5 Les approches de l'Ingénierie dirigée par les modèles :

L'IDM peut être considérée comme un domaine qui a émergé avec les technologies liées à l'instrumentation des modèles. Il existe différentes approches concrétisant différentes façons d'utiliser les modèles dans leur processus de développement des systèmes. L'approche la plus connue et peut-être la plus développée est l'approche MDA. Nous présentons cette approche dans la sous-section suivante.

1.5.1 L'Architecture Dirigée par les Modèles (MDA) :

En novembre 2000, l'OMG (Object Management Group), initie l'approche MDA (Model Driven Architecture). Cette approche a pour but d'apporter une nouvelle vision unifiée de concevoir des applications en séparant la logique métier de l'entreprise, de toute plateforme technique. En effet, la logique métier est stable et subit peu de modifications au cours du temps, contrairement à l'architecture technique.

Il est donc évident de séparer les deux pour faire face à la complexité des systèmes d'information et aux couts excessifs de migration technologique.

La proposition de l'OMG recommande l'utilisation d'UML, MOF (Meta Object Facility) et XMI (XML Meta data Interchange). Le MOF est le méta-méta-modèle standard unique. XMI est le format qui va permettre l'échange de modèles et de méta-modèles. Il est basé sur XML. Enfin, UML est l'un des premiers méta-modèles basé sur le MOF adopté par l'OMG. [7].

La figure 1.2 réprésente une aperçu de l'approche OMG de l'IDM : MDA

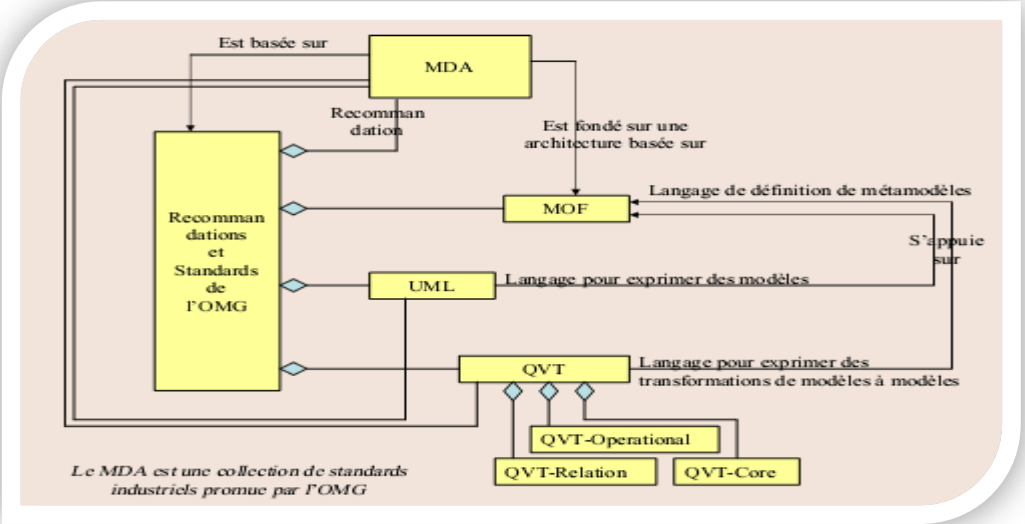

Figure 1.2 : Aperçu de l'approche OMG de l'IDM : MDA

1.5.1.1 Architecture MDA à quatre niveaux:

L'OMG a défini une architecture à quatre niveaux d'abstraction, comme carde général pour l'intégration des méta-modèles, en se basant sur l'MOF comme le montre la figure 1.3. Dans cette architecture, les modèles de deux niveaux adjacents sont liés par une relation d'instanciation : [8]

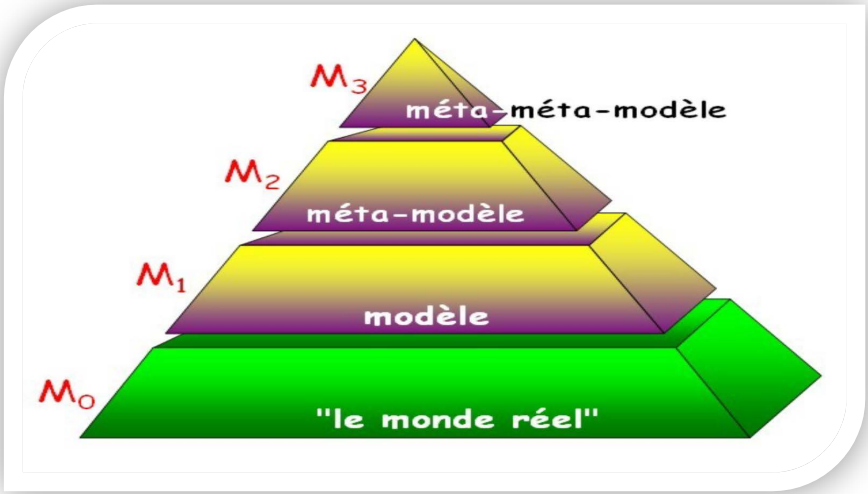

Figure 1.3 : Les quatre niveaux d'abstraction pour MDA

• ***Le niveau M0*** : Niveau des instances des modèles. Il définit des informations pour la modélisation des objets du monde réel.

• ***Le niveau M1*** : Ce niveau représente toutes les instances d'un méta-modèle. Les Modèles du niveau M1 doivent être exprimés dans un langage défini au niveau M2. UML est un exemple de modèles du niveau M1.

• ***Le niveau M2*** : Ce niveau représente toutes les instances d'un méta-méta-modèle. Il est composé de langages de spécifications de modèles d'information. Le méta-modèle UML qui est décrit dans le standard UML et qui définit la structure interne des modèles UML, appartient au niveau M2.

•***Le niveau M3*** : Ce niveau définit un langage unique pour la spécification des

Méta-modèles. Le MOF élément réflexif du niveau M3, définit la structure de tous les Méta-modèles du niveau M2.

1.5.2 Les standards de L'OMG :

L'OMG a déjà défini plusieurs standards pour le MDA : nous en dressons ici une liste des plus importants. [9]

1.5.2.1 MOF : Meta Object Facility

MOF pour (Meta-Object Facility) est un ensemble d'interfaces permettant de définir la syntaxe et la sémantique d'un langage de modélisation. Il a été créé par l'OMG afin de définir des méta-modèles et leurs modèles correspondants.

Il fait partie des standards définis par l'OMG et il peut être vu comme un sous-ensemble d'UML. Le MOF et l'UML constituent le cœur de l'approche MDA car ces deux standards permettent de créer des modèles technologiquement neutres. Le MOF spécifie la structure et la syntaxe de tous les méta-modèles comme UML, CWM (Common Warehouse Meta-model) et SPEM (Software Process Engineering Meta-model.

1.5.2.2 UML: Unified Modeling Language

La notation UML est décrite sous forme d'un ensemble de diagrammes. La première génération d'UML (UML 1.x), définit neuf diagrammes pour la spécification des applications. Dans UML 2.0, quatre nouveaux diagrammes ont été ajoutés : il s'agit des diagrammes de structure composite (Composite structure diagrams), les diagrammes de paquetages (Packages diagrams), les diagrammes de vue d'ensemble d'interaction (Interaction overview diagrams) et les diagrammes de synchronisation (Timing diagrams). Les diagrammes UML sont regroupés dans deux classes principales :

– **Les diagrammes dynamiques** : regroupent les diagrammes de séquence, les diagrammes de communication (nouvelle appellation des diagrammes de collaboration d'UML), les diagrammes d'activités, les machines à états, les diagrammes de vue d'ensemble d'interaction, et les diagrammes de synchronisation.

– **Les diagrammes statiques** : regroupent les diagrammes de classes, les diagrammes d'objets, les diagrammes de structure composite, les diagrammes de composants, les diagrammes de déploiement, et les diagrammes de paquetages.

1.5.2.3 OCL: Object Constraint Language [30]

En utilisant uniquement UML, il n'est pas possible d'exprimer toutes les contraintes souhaitées. Fort de ce constat, l'OMG a défini formellement le langage textuel de contraintes OCL, qui permet de définir n'importe quelle contrainte sur des modèles UML : le langage OCL (Object Constraint Language) qui est un langage d'expression permettant de décrire des contraintes sur des modèles. Une contrainte est une restriction sur une ou plusieurs valeurs d'un modèle non représentable en UML.

1.5.2.4 XMI : Xml Meta data Interchange [31]

XMI est le langage d'échange entre le monde des modèles et le monde XML (eXtensible Markup Language). C'est le format d'échange standard entre les outils compatibles MDA. XMI décrit comment utiliser les balises XML pour représenter un modèle UML en XML.

Cette représentation facilite les échanges de données entre les différents outils ou plateformes de modélisation. En effet, XMI définit des règles permettant de construire des DTD (Document Type Définition) et des schémas XML à partir de méta-modèles, et inversement.

1.6 Modèles de MDA

1.6.1 Modèle d'exigence CIM (Computation Independent Model)

Le CIM est l'abréviation anglaise de Computation Independent Model. Les exigences du Système sont modélisées dans ce modèle qui décrit la situation dans laquelle le système sera utilisé, Un tel modèle est parfois appelé Business Model ou Domain Model. Il ne montre pas les détails de la structure du système. Typiquement ce modèle est indépendant de l'implémentation du système. Le CIM correspond à la modélisation de l'entreprise sans parler encore de système informatique. [10]

1.6.2 PIM (Platform Independent Model)

Le terme PIM est l'acronyme de l'anglicisme Platform Independent Model soit le modèle Indépendant de la plate-forme. Cela signifie que ce type de modèle n'a pas de dépendance Avec la plate-forme technique. Il décrit le système mais ne montre pas les détails de son utilisation sur la plate-forme. Le PIM représente ainsi le logique métier, spécifique au système, mais indépendant de la technique et de la technologie.

Le PIM correspond à la modélisation du système de manière indépendante à la plate-forme. [10]

1.6.3 PSM (Platform Specific Model)

Le PSM, pour Platform Specific Model, est, quant à lui, un modèle dépendant de la Plate-forme technique. Ce type de modèle sert essentiellement de base à la génération de Code exécutable.

Ce modèle, produit à partir d'une transformation du PIM, est un modèle du même système que le PIM mais spécifié par rapport à la plate-forme. Il décrit aussi comment ce système utilisera la plate-forme choisie. [10]

1.6.4 PDM (Platform Description Model)

Ce modèle est désigné par l'acronyme PDM pour Platform Description Model. Il correspond à un modèle de transformation du PIM vers un PSM d'implémentation. L'architecte doit choisir une ou plusieurs plates-formes pour l'implémentation du système

avec les qualités architecturales désirées. Ce modèle propre à la plate-forme est utile pour la transformation du PIM en PSM. La démarche MDA est ainsi basée sur le détail des modèles dépendant de la plate-forme. Il représente les particularités de chaque plate-forme. Il devrait être fourni par le créateur de la plate-forme, le principe du processus MDA est montré dans la (figure 1.4). [10]

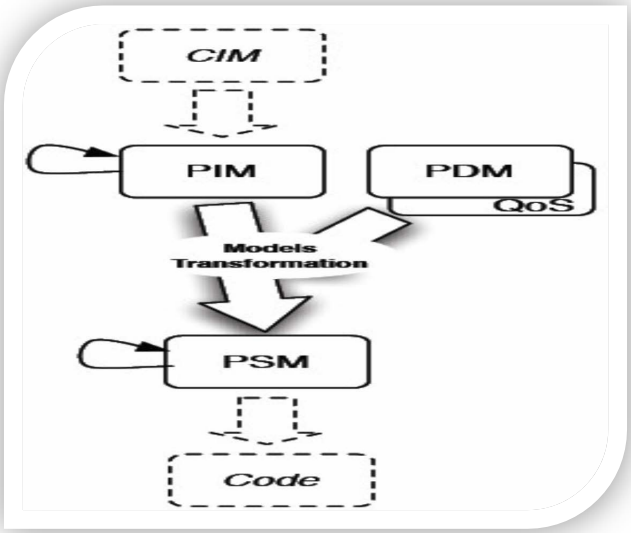

Figure 1.4 : Principe du Processus MDA

1.7 La transformation de modèle

1.7.1 Définition d'une transformation de modèle

La transformation : est une génération automatique d'un modèle cible (Target model) à partir d'un modèle source (source Model).

- D'après la définition de la transformation, La transformation est un ensemble de règles de transformation qui décrivent ensemble comment un modèle dans un langage source est transformé en modèle dans un langage cible.

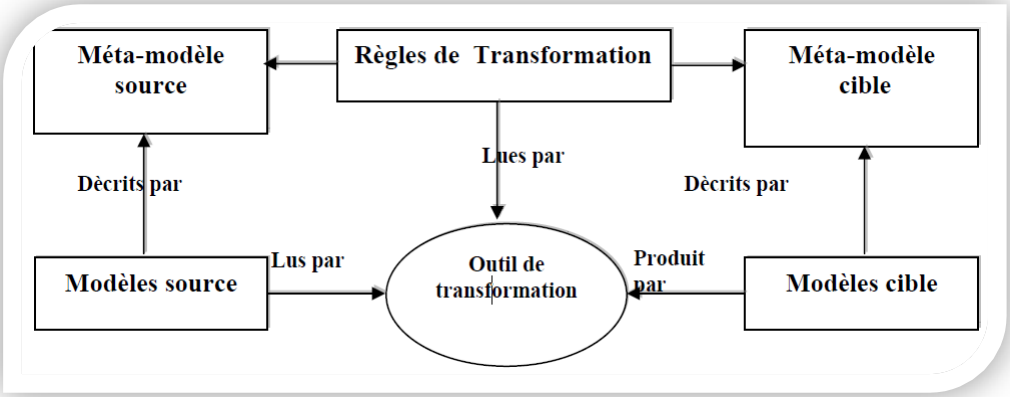

Figure 1.5 : Processus de transformation de modèles

1.7.2 Généralités

MDA étant basée sur la manipulation de modèles, le passage de l'un à l'autre (et vice versa) est une activité centrale de la méthode. La figure suivante présente les différentes opérations de transformation de modèles que l'on peut trouver dans MDA.

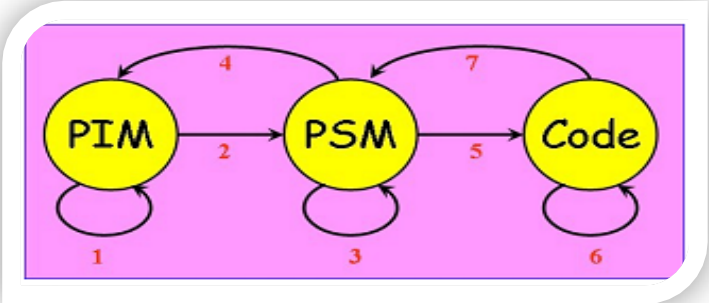

Figure 1.6 : Opérations de transformation sur les modèles MDA

Le passage d'un modèle type UML à du code écrit en langage évolué (Java, C++,...), que l'on considère ici comme un modèle, est déjà bien implémenté dans les outils de modélisation (génération de squelette d'application).

Les transformations réflexives de modèles (PIM/PIM,PSM/PSM, Code/Code) seront effectuées soit lors d'ajouts d'éléments pour représenter des niveaux différents d'abstraction, soit pour optimiser un modèle lorsqu'une transformation non réflexive est incomplète ou pas assez précise (raffinage).Il est important dénoter que ces transformations réflexives ne sont pas toujours automatisables. La transformation centrale de MDA (et aussi la plus délicate) est le passage du PIM vers le PSM Il est temps maintenant de s'y arrêter plus longuement. [11]

Transformations PIM PIM et PSM PSM: Les transformations de type PIM vers PIM ou PSM vers PSM visent à enrichir, filtrer ou spécialiser le modèle. Il s'agit de transformations de modèle à modèle .

Transformation PIM PSM: La transformation de PIM vers PSM permet de spécialiser le PIM en fonction de la plate-forme cible choisie. Elle n'est effectuée qu'une fois le PIM suffisamment raffiné. Cette transformation de modèle à modèle est réalisée en s'appuyant sur les informations fournies par le PDM.

Transformation PSM Code: La transformation de PSM vers l'implémentation (le code) est une transformation de type modèle à texte. Le code est parfois assimilé à un PSM exécutable. Dans la pratique, il n'est généralement pas possible d'obtenir la totalité du code à partir du modèle et il est alors nécessaire de le compléter manuellement.

Transformations inverses PSM PIM et code PSM: Ces transformations sont des opérations de rétro-ingénierie (reverse engineering). Ce type de transformations pose de nombreuses difficultés mais il est essentiel pour la réutilisation de l'existant dans le cadre de l'approche MDA.

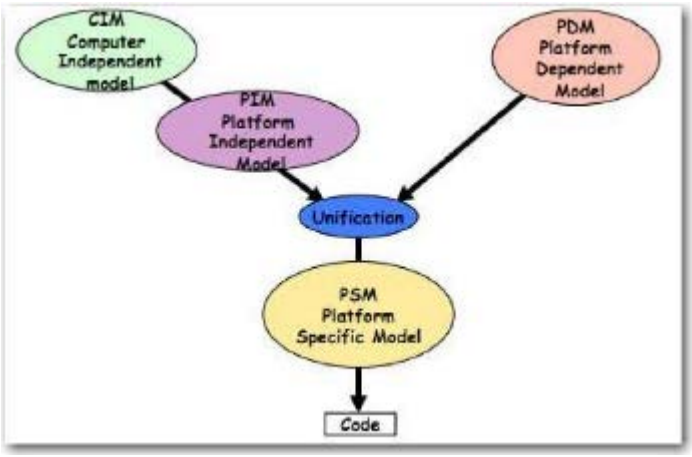

Figure 1.7: Unification *PIM* et *PDM* pour produire le *PSM* puis le code

1.8 Standards et langages pour la transformation de modèle

1.8.1 ATL : ATLAS Transformation Language

ATL (ATLAS Transformation Language) est le langage de transformation développé dans le cadre du projet ATLAS, il est développé au LINA à Nantes par l'équipe de Jean Bézivin, elle se compose : [12]

- ✓ d'un langage de transformation.
- ✓ d'un compilateur et d'une machine virtuelle.
- ✓ d'un IDE s'appuyant sur Eclipse.

1.8.2 Description du langage

ATL est un langage de transformation de modèles crée pour s'inscrire dans une approche MDA. Sa syntaxe abstraite a été décrite comme un méta-modèle MOF. Sa syntaxe concrète textuelle quant à elle, a été dénie en correspondance avec ce méta-modèle.

1.8.3 Transformations :

Un modèle de transformation ATL peut transformer un ensemble de modèles sources en un ensemble de modèles cibles, à la condition que les méta-modèles sources et cibles lui soient connus. Actuellement, ATL est capable de manipuler tous les modèles conformes à un méta-modèle MOF, ce qui implique aussi bien la transformation de méta-modèles que du méta-méta-modèle, grâce à la rétivité de celui-ci.

Un programme de transformation écrit en ATL est composé de règles qui spécifient comment les éléments du modèle source sont reconnus et parcourus pour créer et initialiser les éléments du modèle cible. Ces règles sont de la forme générale suivante :

```
rule ForExample {
     from
          i : InputMetaModel!InputElement
     to
          o : OutputMetaModel!OutputElement(
               attributeA <- i.attributeB,
               attributeB <- i.attributeC + i.attributeD
          )
}
```

Figure 1.8: Un exemple de règle ATL

- "ForExample" est le nom de la règle de transformation.
- i (resp. o) est le nom de la variable qui dans le corps de la règle va représenter l'élément source identifié (resp. l'élément cible créé).
- InputMetaModel (resp. OutputMetaModel) est le méta modèle auquel le modèle source (resp. le modèle cible) de la transformation est conforme.

- InputElement désigne la métaclasse des éléments du modèle source auxquels cette règle va s'appliquer.
- OutputElement désigne la métaclasse à partir de laquelle la règle va instancier les éléments cibles.
- attributeA et attributeB sont des attributs de la métaclasse OutputElement.
- Leur valeur est initialisée à l'aide des valeurs des attributs i.attributeB, i.attributeC et i.attributeD de la métaclasse InputElement.

1.9 Outils supporté pour la transformation des modèles

1.9.1 Introduction

La technologie MDA (Model Driven Architecture) est un standard issu de l'OMG qui permet d'exploiter et de transformer les modèles UML. Ceci permet notamment de produire d'autres modèles, produire automatiquement des documentations issues du modèle, générer automatiquement du code à partir du modèle.

Une fonctionnalité essentielle de la MDA est la notion de transformation. Une transformation regroupe un ensemble de règles et de techniques pour transformer un modèle en un autre, et pour faire la modélisation, la méta-modélisation, et la transformation de ces modèles on a besoin des outils de modélisation et de transformation, et on vue Plusieurs outils de modélisation ont été proposés pour effectuer la transformation des modèles soit transformation modèle vers modèle ou modèle vers code (génération du code), par exemple AGG, AToM3, VIATRA2, et des plug-ins éclipse EMF et GMF. Certaines tentatives de transformations des modèles sont réalisées, Les travaux connexes représentent un survol sur ces approches. Et ce travail basé sur les outils EMF et ATL donc dans ce chapitre nousallons présenter en générale quelque outils et nous passons à détailler les outils de notre travail.

1.10 Plate-forme d'accueil

1.10.1 Généralité sur Eclipse

Eclipse est un projet créé en 2001 par IBM, décliné et organisé en un ensemble de sous-projets de développements logiciels, de la Fondation Eclipse visant à développer un environnement de production de logiciels libres qui soit extensible, universel et polyvalent, en s'appuyant principalement sur Java. Son objectif est de produire et fournir des outils pour la réalisation de logiciels, englobant les activités de programmation mais aussi d'ATL recouvrant modélisation, conception, testing, gestion de configuration, reporting... Son EDI, partie

intégrante du projet, vise notamment à supporter tout langage de programmation à l'instar de Microsoft Visual Studio. Bien que conçu initialement uniquement pour produire des environnements de développement, ses utilisateurs et contributeurs se sont rapidement mis à réutiliser ses librairies logicielles pour des applications clientes classiques. Cela a conduit à une extension du périmètre initial d'Eclipse à toute production de logiciel : c'est l'apparition du framework Eclipse RCP en 2004. [13]

1.10.2 Eclipse plate-forme

L'IDE Eclipse Platform est principalement écrit en Java (à l'aide de la bibliothèque graphique SWT, d'IBM); ce langage, grâce à des bibliothèques spécifiques, est également utilisé pour écrire les extensions. La base de l'environnement de développement intégré que constitue l'EDI Eclipse Platform est composée de :

- Platform Runtime démarrant la plateforme et gérant les plug-ins
- SWT, la bibliothèque graphique de base de l'EDI
- JFace, une bibliothèque graphique de plus haut niveau basée sur SWT
- Eclipse Workbench, la dernière couche graphique permettant d'organiser et manipuler des composants, tels que des vues, des éditeurs et des perspectives. [13]

1.11 Manipuler des modéles avec EMF

1.11.1 Eclipse Modeling Framework (EMF)

EMF existe depuis 2002, est un framework qui traite des modèles c'est-à-dire qu'EMF offre à ces utilisateurs un cadre de travail pour la manipulation des modèles. EMF permet de stocker les modèles sous forme de fichier pour en assurer la persistance. EMF permet également de traiter différents types de fichiers : conformes à des standards reconnus (UML, XML, XMI) et aussi sous des formes spécifiques (code Java) ou tout simplement sur mesure (au bon gré du concepteur).

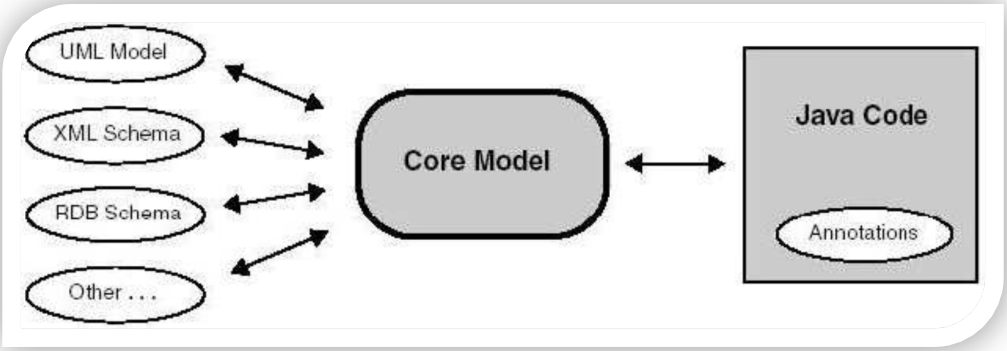

Figure 1.9 : organisation générale d'EMF

1.11.2 Objectif d'EMF :

L'objectif général d'EMF est de proposer un outillage qui permet de passer du modèle au code Java automatiquement. Pour cela le framework s'articule autour d'un modèle (le Core Model). EMF va proposer plusieurs services :

1. la transformation des modèles d'entrées, présentés sous diverses formes, en Core Model.
2. la gestion de la persistance du Core Model.
3. la transformation du Core Model en code Java.

EMF peut de base gérer en entrée des modèles présentés sous trois formats :

- ✓ UML.
- ✓ XMI.
- ✓ Code Java Annoté.

Dans EMF, il est possible de définir un méta-modèle et de générer les interfaces afin de pouvoir manipuler les instances du méta-modèle dans Eclipse. [14]

1.11.3 Les formats d'entrée standards

EMF peut de base gérer en entrée des modèles présentés sous trois formats : UML, XMI et en code Java Annoté. [15]

- ➢ **UML** Pour cette option, il existe trois possibilités.
  - ✓ L'édition directe conformément au méta-modèle Ecore
  - ✓ L'importation de modèles UML
  - ✓ L'exportation de modèles UML

- **XMI :**

Ce format de fichier, standard de l'OMG (Object Managment Group), est utilisé conjointement à UML :

• UML se charge de décrire les contenus des modèles,

• XMI se charge de formater ces contenus pour permettre de leur assurer une persistance standardisée.

• comme nous l'avons vu ci-dessus, il tend à devenir un standard, du moins pour le développement orienté objet.

• il est le standard utilisé par Ecore pour sa propre persistance,

• tout cela concoure à combler le fossé qui existe entre les modèles UML et les fichiers de code Java.

- **Java annoté :**

Une des solutions tentantes pour modéliser les classes, qui vont être concrétisées par une application Java, est d'utiliser les interfaces Java :

• elles n'implémentent pas les méthodes : on s'abstrait donc de cette implémentation,

• les méthodes get/set peuvent être utilisées pour modéliser les attributs,

• une classe pourra implémenter plusieurs interfaces, ce qui est une manière détournée d'autoriser l'héritage multiple (impossible en Java de classe à classe et possible en UML).

1.12 EMF et MDA

Le processus d'utiliser EMF est compatible avec l'approche MDA de OMG, cependant il en manque quelques-uns de les propriétés essentielles d'un MDA, l'outil EMF produit le code de modèles. Parler strictement, un outil MDA, devez produire plate-forme - spécifique les modèles (PSM) avant de produire le code. Bien qu'il ne puisse pas s'aligner avec OMG 100%, EMF est un du la plupart des ambassadeurs puissants pour le MDA approchent.

1.12.1 Le méta-méta-modèle d'EMF

EMF propose son propre méta-méta-modèle, le méta-méta-modèle Ecore, qui ressemble fortement au méta-méta-modèle EMOF, car il ne supporte que la notion de méta-classe sans méta-association. Ecore est légèrement différent du standard EMOF en ce qu'il est entièrement intégré à la plate-forme Eclipse. [16]

1.12.2 Le méta-modèle Ecore :

Ecore est un méta-méta-modèle très proche de MOF. Il est en fait un sous ensemble de MOF. En effet, une des particularités d'Ecore est qu'il accepte des méta-classes (dans le niveau M2) sans associations. Pour associer deux classes, il faudra stéréotyper un attribut comme étant une association. Cette possibilité a été mise au point car en langage Java le concept d'association n'existe pas.

En Java, les associations d'un diagramme de classes UML s'implémentent par la création d'un attribut ayant pour type la classe partenaire. L'attribut doit être créé dans l'une, dans l'autre ou dans les deux classes partenaires (dans le cas d'une association binaire) selon la navigabilité de l'association, au niveau du méta-modèle l'on définira les caractéristiques du modèle de niveau M1. Par exemple, on définira les concepts d'un diagramme de classes UML si c'est cela que l'on veut traiter.

Le caractère fédérateur de EMF et de son méta-méta-modèle Ecore tient dans le fait que les différents modèles d'entrée et de sortie (voir tableau 1.1) ont leur méta-modèle Ecore et les transformations de l'un à l'autre se feront en appuyant sur un (ou quelques) bouton. Le tableau ci-dessous synthétise les cas en présence pour EMF. [17]

Modèle(M1)	**Méta-modèle(M2)**	**Méta-méta-modèles(M3)**
Diagramme de classes (entrée)	UML	MOF
Fichier XML (entrée)	XMI	MOF
Ensemble d'interfaces Java (entrée)	Java Annoté	EBNF
Programme Java (sortie)	Java	EBNF

Tableau 1.1 : les cas en présence pour EMF

Dans MDA, les transformations sont définies au niveau des méta-modèles. Cette stratégie permet de modéliser les règles de transformation, et ainsi de capitaliser les efforts faits pour leur définition, Le tableau ci-dessous représente l'espace technique standard d'EMF.

Modèle (M1)	**Méta-modèle (M2)**	**Méta-méta-modèle (M3)**
MEcore (pivot)	MMEcore	Ecore

Tableau 1.2 : l'espace technique standard d'EMF

1.12.3 Les interfaces réflectives d'EMF

Tout comme JMI, EMF fourni des interfaces réflectives. Celles-ci permettent la manipulation des modèles d'une façon indépendante de leur méta-modèle.

La particularité d'EMF est que toutes les interfaces réflectives qu'il propose sont spécifiées dans le méta-méta-modèle Ecore. [16]

- **EObject**

L'interface réflective la plus importante est EObject. Elle représente n'importe quel élément, qu'il appartienne à un modèle ou à un métamodèle, cette interface offre l'opération eClass(), qui permet d'obtenir l'EClass de l'élément. Cette opération retourne un objet Java de type EClass, elle offre aussi les opérations eGet() et eSet(), qui permettent respectivement de lire et d'écrire les valeurs des différentes propriétés de l'élément (attributs et références).

- **EClass**

L'interface réflective EClass représente une métaclasse d'un métamodèle, Cette interface offre les opérations getEAttributes() et getEReferences(), qui permettent d'obtenir la liste respectivement de tous les attributs et références contenus dans la métaclasse.

- **EPackage**

L'interface réflective EPackage représente le moyen d'accès à toutes les EClass définies dans un package. Cette interface offre l'opération getEClassifier(String name), qui permet de récupérer la référence vers une EClass d'un métamodèle à partir de son nom.

- **EFactory**

L'interface réflective EFactory représente le moyen de créer des instances des EClass définies dans un package. Cette interface offre l'opération create(), qui permet de créer une instance d'une EClass.

1.13 GMF& GEF :

L'utilisation de deux GMF et GEF pour la construction d'une fonctionnalité basée sur Eclipse est assez fréquent. Beaucoup de références ci-dessous fournissent des informations sur la façon d'utiliser ces cadres ensemble, dont certaines inspirées du projet GMF lui-même. [18]

1.14 Topcased

TOPCASED est un logiciel d'ingénierie assistée par ordinateur. Il contient un IDE basé sur le framework de la plateforme de développement Eclipse, à laquelle il ajoute des fonctionnalités essentiellement liées à la mise en œuvre de la première branche du cycle en V pour l'ingénierie du logiciel, du matériel ou de systèmes mixtes logiciel/matériel.

Sont implémentés des moyens d'analyse d'exigences, modélisation, simulation de modèles, implémentation, test, validation, rétro-ingénierie, génération de code, de modèles et de documentation et gestion de projet.

S'appuyant principalement sur des langages standardisés pour la modélisation du logiciel (UML, SysML, AADL...), TOPCASED travaille avec des fichiers XMI. Tous ces standards sont implémentés dans leurs dernières versions stables, soit directement par le projet TOPCASED, soit par les modules de la dernière version stable de la plate-forme Eclipse. TOPCASED UML est ainsi le modeleur UML le plus complet parmi les solutions gratuites, et le plus respectueux des standards actuels. [19]

1.15 ATOM3

AToM3 (A Tool for Multi-formalism and Meta-Modelling) est un outil de modélisation multi-paradigmes développé par le laboratoire MSDL (Modelling, Simulation and Design Lab.) à l'université de McGill Montréal, Canada. Cet outil a été conçu en collaboration avec Juan de Lara de Universidad Autónoma de Madrid (UAM), Espagne.

Les deux principales fonctionnalités d'AToM3 sont **la méta-modélisation** et la **transformation de modèles**. La méta-modélisation est la description ou la modélisation de différents types de formalismes. La transformation de modèles est une technique consistant à transformer, traduire ou modéliser automatiquement un modèle décrit dans un certain formalisme, vers un autre modèle qui peut être décrit soit dans le même formalisme soit dans un autre, Il permet aussi de définir la syntaxe abstraite et concrète des langages visuels.[20]

1.16 Conclusion

Nous avons présenté dans ce chapitre un état de l'art sur le développement logiciel dirigé par les modèles. Comme nous l'avons dit dans l'introduction, nous avons plus particulièrement mis l'accent dans cette étude sur les concepts, langages et outils associés à la transformation de modèles – paradigme central de l'IDM.

Le développement logiciel dirigé par les modèles a pour principal objectif de concevoir des applications en séparant les préoccupations et en plaçant les notions de modèles, méta-modèles et transformations de modèles au centre du processus de développement. Les transformations nécessitent des langages dédiés (ATL, QVT,…), des outils de méta-modélisation flexibles (Kermeta, EMF/Ecore, TOPCASED,…) qui pourront assister les concepteurs de systèmes logiciels complexes.

Beaucoup de thèmes de recherche émergent au sein de la sphère IDM ; il devient donc essentiel de créer une synergie entre les différents groupes de recherche dans ce domaine. L'action transversale IDM (action-idm-website) a ainsi été lancée pour servir de cadre de réflexion et d'échange entre les communautés issues de domaines technologiques différents.

Chapitre 02 :

Les langages de modélisation

CHAPITRE 2

LES LANGAGES DE MODELISATION

2.1 Introduction

Ayant à notre disposition les méta-modèles de diagramme de classe et de base de données, nous allons définir, dans ce chapitre, un chemin de migration horizontal entre ces deux différentes technologies, par la définition d'une transformation qui permet de passer de modèle statique et abstrait (diagramme de classe) à un modèle statique et exécutable (base de données).

Nous allons utiliser un langage bien défini pour exprimer l'ensemble des règles qui constituent la définition de la transformation. La définition des règles de transformation entre modèles se fera au niveau des méta-modèles

2.2 Le Langage UML

Durant les années 70 et 80, on comptait tout au plus dix langages de spécification d'applications orientées objet. Afin d'enrayer la multiplication des langages de spécification, G. Booch et J. Rumbaugh ont décidé en 1994 d'unifier leurs méthodes respectives OOADA (Object-Oriented Analysis and Design with Applications) et OMT (Object Management Technique) au sein de la société Rational Software Corporation.

La première version de ces travaux est sortie en octobre 1995 sous le nom d'UML0.8. Après 1995, l'initiative de Rational Software Corporation a intéressé d'autres industriels, qui ont vite compris les avantages qu'ils pouvaient tirer de l'utilisation d'UML. C'est ainsi qu'une RFP (Request For Proposal) a été émise à l'OMG en 1996 pour la standardisation d'UML.

Des débuts de l'initiative UML jusqu'à la version 1.4 élaborée par l'OMG, l'objectif fondamental était de résoudre le problème de l'hétérogénéité des spécifications. L'approche envisagée a consisté à proposer un langage unifiant tous les langages de spécification.

La version 2.0 vise à faire entrer ce langage dans une nouvelle ère en faisant en sorte que les modèles UML soient au centre de MDA. Cela revient à rendre les modèles UML pérennes et productifs et à leur permettre de prendre en compte les plates-formes d'exécution.

2.2.1 Définition d'UML :

Tout à fait simplement, l'UML « Unified Modeling Language » ou « langage de modélisation unifié » est une langue visuelle pour modeliser et communiquer au sujet de systèmes à travers l'usage de diagrammes et texte secondaire.

UML n'est pas une méthode ou un processus, est un langage de modélisation objet, UML a été adopté par toutes les méthodes Objet. [21]

2.2.2 La modélisation UML

Le méta-modèle UML fournit une panoplie d'outils permettant de représenter l'ensemble des éléments du monde objet (classes, objets, ...) ainsi que les liens qui les relie. Toutefois, étant donné qu'une seule représentation est trop subjective, UML fournit un moyen astucieux permettant de représenter diverses projections d'une même représentation grâce aux *vues*. Une vue est constitué d'un ou plusieurs *diagrammes*, On distingue deux types de vues :

- **Les vues statiques** : c'est-à-dire représentant le système physiquement
- diagrammes d'objets.
- diagrammes de classes.
- diagrammes de cas d'utilisation.
- diagrammes de composants.
- diagrammes de déploiement.
- **Les vues dynamiques** : montrant le fonctionnement du système
- diagrammes de séquence.
- diagrammes de collaboration.
- diagrammes d'états-transitions.
- diagrammes d'activités.

2.2.3 Diagramme de classe

Le diagramme de classes constitue un élément très important de la modélisation : il permet de définir quelles seront les composantes du système final : il ne permet en revanche pas de définir le nombre et l'état des instances individuelles. Néanmoins, on constate souvent qu'un diagramme de classes proprement réalisé permet de structurer le travail de développement de manière très efficace, il permet aussi, dans le cas de travaux réalisés en groupe, de séparer les

composantes de manière à pouvoir répartir le travail de développement entre les membres du groupe, enfin, il permet de construire le système de manière correcte. [22]

- **Les classes**

La notion de classe est essentielle en programmation orientée objets : elle définit une abstraction, un type abstrait qui permettra plus tard d'instancier des objets. On distingue généralement entre classes abstraites (qui ne peuvent pas être instanciées) et classes "normales", qui servent à définir des objets.

- ✓ **Classes abstraites**

Une classe abstraite ne peut donc pas être utilisée pour fabriquer des instances d'objets, elle sert uniquement de modèle, que l'on pourra utiliser pour créer des classes plus spécialisées par dérivation (héritage). Une classe abstraite est assez proche de la notion d'interface, d'ailleurs, la notion d'interface est généralement implémentée par une classe abstraite, dont toutes les méthodes sont purement virtuelles. [22]

Sa représentation en UML correspond à la figure 2.1 :

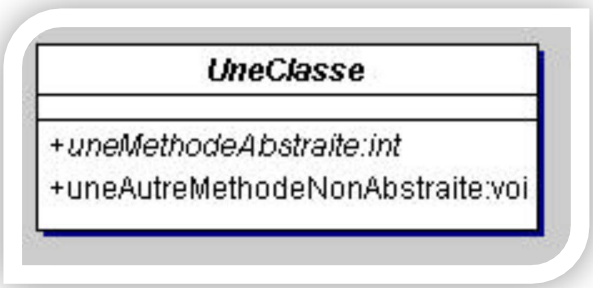

Figure 2.1 : classe abstraite

Le + dénote la publication du membre concerné. Une classe abstraite peut posséder des membres privés ou des attributs privés ou publics, d'autre part, les méthodes peuvent faire l'objet d'une implémentation, même si la méthode est purement virtuelle.

- ✓ **Classes non abstraites**

Une classe "normale" ne contient pas de membres abstraits. Sa représentation en UML correspond à la figure 2.2.

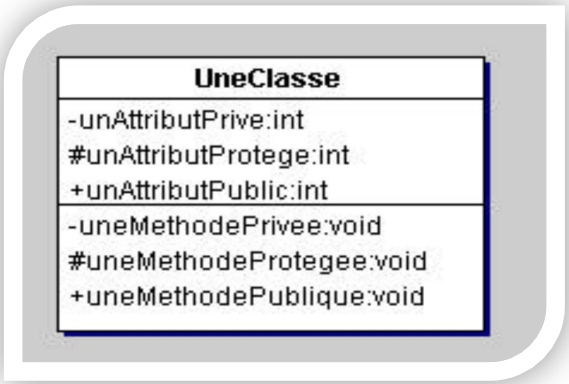

Figure 2.2 : Classe non abstraite

Noter les petits signes "cabalistiques" précédant l'identification des Méthodes : [22]

- \- dénote un membre privé
- \+ dénote un membre public
- # Dénote un membre protégé

✓ **Représentation graphique**

Une classe est un classeur. Elle est représentée par un rectangle divisé en trois à cinq compartiments.

Le premier indique le nom de la classe, le deuxième ses attributs et le troisième ses opérations. Un compartiment des responsabilités peut être ajouté pour énumérer l'ensemble de tâches devant être assurées par la classe mais pour lesquelles on ne dispose pas encore assez d'informations. Un compartiment des exceptions peut également être ajouté pour énumérer les situations exceptionnelles devant être gérées par la classe.

Nom de la classe
- **Attribut_1 : type_1** - **Attribut_2 : type_2**
+ Opération_1 ( ) :type1 **+ Opération_2 ( ) :type2**

Figure 2.3 : Représentation UML d'une classe.

✓ **Caractéristiques d'une classe**

- **Nom d'une classe**

Le nom de la classe doit évoquer le concept décrit par la classe. Il commence par une majuscule. Pour indiquer qu'une classe est abstraite, il faut ajouter le mot-clef **abstract**.
Il est écrit dans le rectangle du haut. Dans une classe classique, le nom est écrit en romain.
Le nom des classes abstraites est écrit en italique. Les classes Template ont, dans leur angle supérieur droit, un rectangle dont la bordure est en pointillé et qui contient les types des paramètres.

- **Les attributs de la classe**

Les attributs (on parle parfois de propriétés) définissent des informations qu'une classe ou un objet doit connaître. Ils représentent les données encapsulées dans les objets de cette classe. Chacune de ces informations est définie par un nom, un type de données, une visibilité et peut être initialisé. Le nom de l'attribut doit être unique dans la classe.

Le type de l'attribut peut être un nom de classe, un nom d'interface ou un type de donné prédéfini. La multiplicité d'un attribut précise le nombre de valeurs que l'attribut peut contenir. Lorsqu'une multiplicité supérieure à 1 est précisée, il est possible d'ajouter une contrainte pour préciser si les valeurs sont ordonnées ou non.

Par défaut, chaque instance d'une classe possède sa propre copie des attributs de la classe. Les valeurs des attributs peuvent donc différer d'un objet à un autre. Cependant, il est parfois nécessaire de définir un attribut de classe (*statique* en Java ou en C++) qui garde une valeur unique et partagée par toutes les instances de la classe. Les instances ont accès à cet attribut mais n'en possèdent pas une copie. Un attribut de classe n'est donc pas une propriété d'une instance mais une propriété de la classe et l'accès à cet attribut ne nécessite pas l'existence d'une instance. Graphiquement, un attribut de classe est souligné.

- **Les Méthodes (Opération) de la classe**

Dans une classe, une opération (même nom et même types de paramètres) doit être unique. Quand le nom d'une opération apparaît plusieurs fois avec des paramètres différents, on dit que l'opération est surchargée. En revanche, il est impossible que deux opérations ne se distinguent que par leur valeur retournée.

Comme pour les attributs de classe, il est possible de déclarer des méthodes de classe. Une méthode de classe ne peut manipuler que des attributs de classe et ses propres paramètres.

Cette méthode n'a pas accès aux attributs de la classe. L'accès à une méthode de classe ne nécessite pas l'existence d'une instance de cette classe. Graphiquement, une méthode de classe est soulignée.

✓ **Classe active :**

Une classe est passive par défaut, elle sauvegarde les données et offre des services aux autres. Une classe active initie et contrôle le flux d'activités. Graphiquement, une classe active est représentée comme une classe standard dont les lignes verticales du cadre, sur les côtés droit et gauche, sont doublées.

➢ **Relations entre classes**

✓ **Notion d'association :**

Une association est une relation entre deux classes (association binaire) ou plus (association n-aire), qui décrit les connexions structurelles entre leurs instances. Une association indique donc qu'il peut y avoir des liens entre des instances des classes associées. Un attribut peut être considéré comme une association dégénérée dans laquelle une terminaison d'association est détenue par un classeur.

✓ **Association binaire :**

Une association binaire est matérialisée par un trait plein entre les classes associées. Elle peut être ornée d'un nom, avec éventuellement une précision du sens de lecture (► ou ◄). Quand les deux extrémités de l'association pointent vers la même classe, l'association est dite réflexive.

Figure 2.4 : Exemple d'association binaire.

✓ **Association n-aire :**

Une association n-aire lie plus de deux classes. On représente une association n-aire par un grand losange avec un chemin partant vers chaque classe participante. Le nom de l'association, le cas échéant, apparaît à proximité du losange.

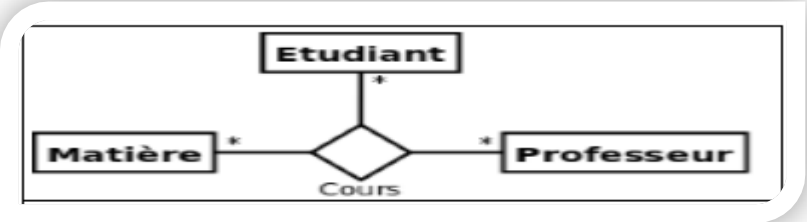

Figure 2.5 : exemple d'association n-aire

✓ **Multiplicité ou cardinalité :**

La multiplicité associée à une terminaison d'association, d'agrégation ou de composition déclare le nombre d'objets susceptibles d'occuper la position définie par la terminaison d'association. Voici quelques exemples de multiplicité :

- 1**…**1 noté 1 : Un et un seul
- 0**…**1 : Zéro ou un
- 0**…*** noté * : De Zéro à n
- 1**…*** : De un à n
- n**…**m : De n à m

✓ **Classe-association :**

Il peut arriver que l'on ait besoin de garder des informations (attributs ou opérations) propres à une association. Une classe de ce type est appelée classe association. Une classe association est une classe comme une autre qui peut entretenir des relations avec d'autres classes.

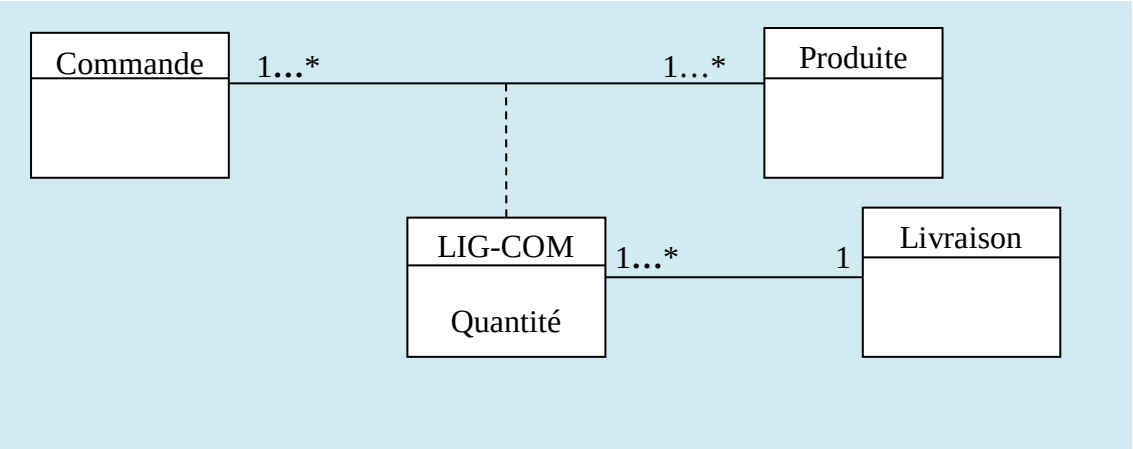

Figure 2.6 : Exemple de classe-association.

✓ **Agrégation :**

Une agrégation est un type particulier d'association. Elle traduit la volonté de renforcer la dépendance entre les classes. C'est une association non symétrique dans laquelle une des extrémités joue un rôle prédominant par rapport à l'autre extrémité. Graphiquement, on ajoute un losange vide (◊) du côté de l'agrégat. [23]

Les critères suivants impliquent une agrégation :

- une classe fait partie d'une autre classe,
- une action sur une classe implique une action sur une autre classe,
- les objets d'une classe sont subordonnés aux objets d'une autre classe.

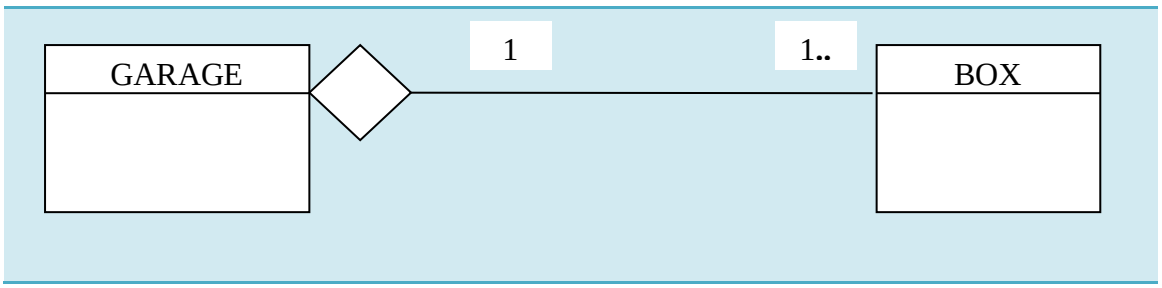

Figure 2.7 : Exemple de relation d'agrégation.

L'inverse n'est pas toujours vrai, l'agrégation n'implique pas nécessairement tous les critères ci-dessus. Un agrégat peut être multiple.

✓ **Composition :**

La composition est un cas particulier d'agrégation dans laquelle la vie des composants est liée à celle de l'agrégat. Dans la composition, l'agrégat ne peut être multiple. La composition se représente par un losange noir. [23]

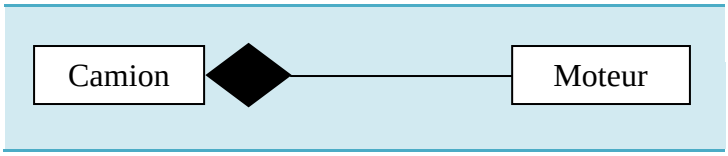

Figure 2.8 : Exemple de relationde composition**.**

Une composition est une association contraignante : la suppression d'un objet agrégat entraîne la suppression des objets agrégés.

✓ **Généralisation et Héritage :**

La généralisation décrit une relation entre une classe générale (classe de base ou classe mére) et une classe spécialisée (sous-classe). Dans le langage UML, la relation de généralisation se traduit par le concept de relation d'héritage. [24]

Les propriétés principales de l'héritage sont :

- La classe enfant possède toutes les caractéristiques de ses classes parents, mais elle ne peut accéder aux caractéristiques privées de cette dernière.
- Une classe enfant peut redéfinir une ou plusieurs méthodes de la classe parent.
- Toutes les associations de la classe parent s'appliquent aux classes dérivées.
- Une instance d'une classe peut être utilisée partout où une instance de sa classe parent est attendue.
- Une classe peut avoir plusieurs parents, on parle alors d'héritage multiple.

✓ **Dépendance :**

Une dépendance est une relation unidirectionnelle exprimant une dépendance sémantique entre des éléments du modèle. Elle est représentée par un trait discontinu orienté. Elle indique que la modification de la cible peut impliquer une modification de la source.

2.3 Généralités sur les bases de données

2.3.1 Introduction

Le domaine informatique bien qu'étant jeune, a une évolution croisière. Jadis, la gestion et le traitement des données se faisaient par la méthode classique à laquelle l'on a pu dégager ces défauts suivants :

- La redondance de données.
- La dépendance pleine entre données et traitement.
- Le manque de normalisation au niveau de stockage de données.

- Pour remédier à cette situation, il a été mis au point la notion de base de données répondant aux questions suivantes :
  - L'accès aux données selon les multiples critères.
  - L'intégration des données.
  - La relation entre les données.

2.3.2 Qu'est-ce qu'une base de données ?

Définition *1.1.*- Base de données -Un ensemble organisé d'informations avec un objectif commun.

Plus précisément, on appelle base de données un ensemble structuré et organisé permettant le stockage de grandes quantités d'informations afin d'en faciliter l'exploitation (ajout, mise à jour, recherche et consultions de données) [25].

Définition *1.2.* Base de données informatisée - Une base de données informatisée est un ensemble structuré de données enregistrées sur des supports accessibles par l'ordinateur, représentant des informations du monde réel et pouvant être interrogées et mises à jour par une communauté d'utilisateurs. [25]

La gestion et l'accès à une base de données sont assurés par un ensemble de programme que constitue le système de gestion de base de données (SGBD).

Ainsi la notion de base de données est généralement couplée à celle des réseaux informatiques afin de pouvoir mettre en commun les informations d'où le nom de « base ».

On parle souvent de système d'information pour désigner toute structure regroupant les moyens mis en place pour partager les données.

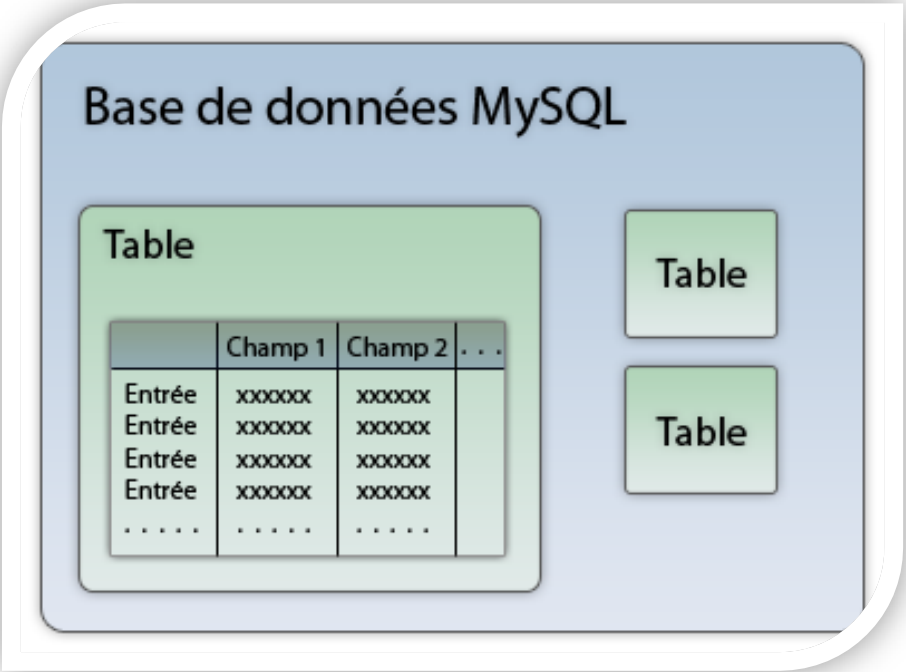

Figure 2.9 : Exemple d'une Bases de données MySQL

2.3.3 Critères d'une base de données

Une base de données doit répondre aux trois critères suivants :

- ✓ L'exhaustivité : C'est la présence dans cette base de tous les enseignements qui ont trait aux applications en question.
- ✓ La non redondance des données : Non répétition d'une donnée plusieurs fois.
- ✓ La structure : C'est l'adaptation du mode de stockage de données au traitement de structuration que la base doit avoir est liée à l'évolution de la technologie. [26]

2.3.4 Utilité d'une base de données

Une base de données permet de mettre des données à la disposition des utilisateurs pour une consultation, une saisie ou bien une mise à jour, tout en assurant des droits accordés à ces derniers. Cela est d'autant plus utile que les données informatiques soient de plus en plus nombreuses.

2.3.5 Avantages de la base de données

La base de données présente les avantages ci - après :

- L'indépendance entre données et traitements.
- La duplication des données est réduite.
- La base de données dote l'entreprise d'un contrôle centralisé de données opérationnelles qui représentent d'après H.S. MELZER le capital important de l'entreprise.
- L'ordre dans le stockage de données.
- L'utilisation simultanée des données par différents utilisateurs.

2.3.6 Sécurité et confidentialité de la base de données

La base de données doit être sécurisée contre :

- ✓ Les indiscrétions : Par un mot de passe
- ✓ Les erreurs : Des contrôles doivent être mis en place pour vérifier que des contraintes d'intégrités sont respectées.
- ✓ Les destructions : En cas d'incident (panne logicielle, panne matérielle ou panne d'électricité), des procédures de sauvegarde et reprise doivent être prévues afin de relancer le système sans avoir recommencé les saisies par la transaction.

2.3.7 Niveau de description des données ANSI/SPARC

Pour atteindre certains de ces objectifs, trois niveaux de description de données ont été définis par la norme ANSI/SPARC. [25]

- Le niveau externe :

Correspond à la perception de tout ou partie de la base pour un groupe donné d'utilisateurs, indépendamment des autres. On appelle cette description le schéma externe ou "vue".

- Le niveau conceptuel :

Décrit la structure de toutes les données de la base, leurs propriétés (les relations qui existent entre elles : leur sémantique inhérente), sans se soucier de l'implémentation physique ni de la façon chaque groupe de travail voudra s'en servir. On appelle cette description le schéma conceptuel.

- Le niveau interne ou physique :

Correspond à la manière dont la base est implantée sur les ordinateurs, définit la politique de stockage ainsi que le placement des données (description des enregistrements contenant les données des index, etc.).

2.3.8 Conception de Base de Données

La conception d'un système d'information n'est pas évidente car, il faut réfléchir sur l'ensemble de l'organisation, que l'on doit mettre en place. La phase de conception nécessite des méthodes permettant de mettre en place un modèle sur lequel il faut s'appuyer. La modélisation consiste à créer une représentation virtuelle d'une réalité de telle façon à faire ressortir les points auxquels l'on s'intéresse. La méthode la plus utilisée et repandue est la méthode "MERISE". [25]

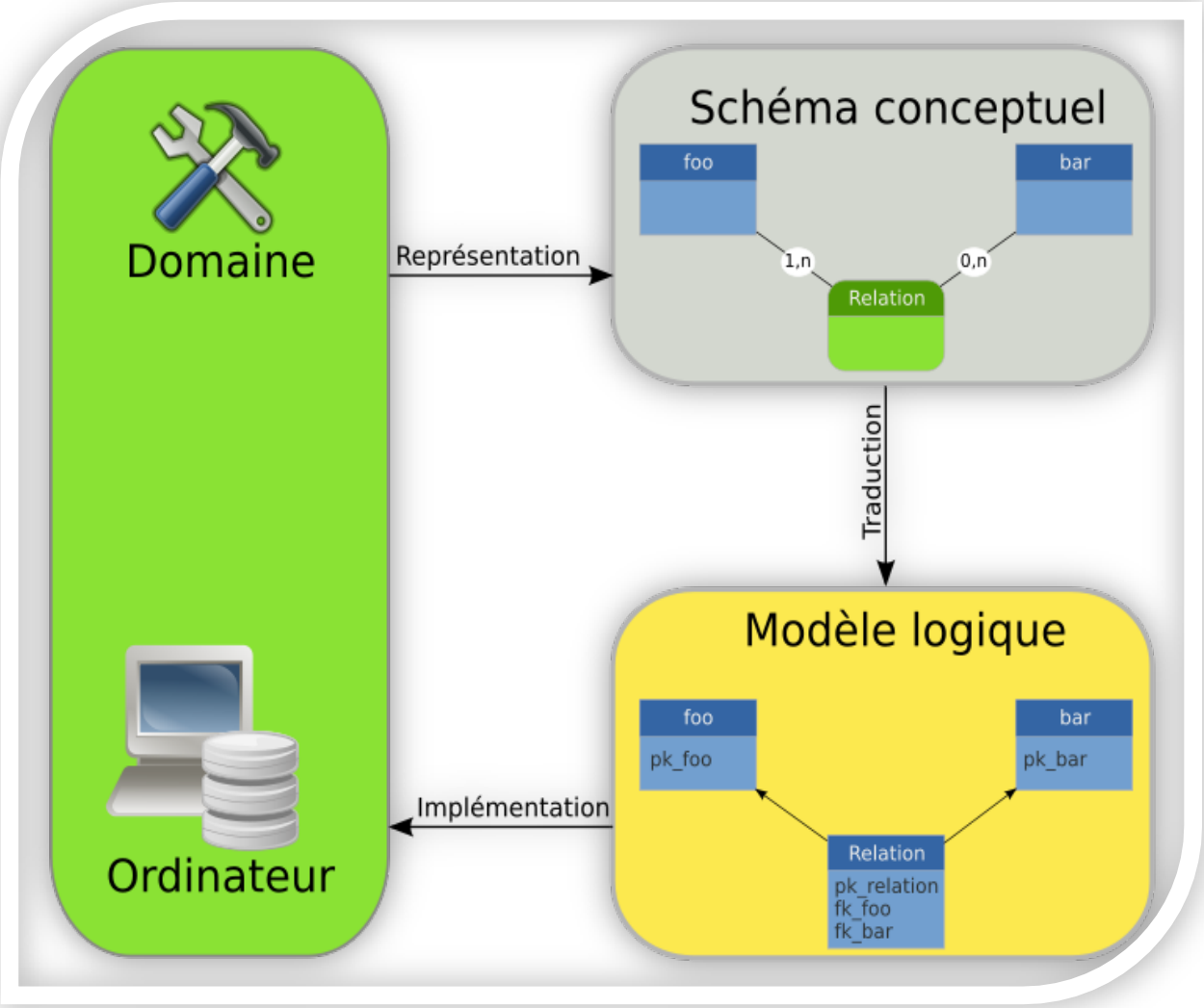

Figure 2.10 : Processus de conception d'une base de données

2.3.8.1 MERISE

MERISE (Méthode d'Etude et de Réalisation Informatique pour les Systèmes d'Entreprise). MERISE est une méthode de conception, de développement et de réalisation des projets informatiques.

- ✓ Caractéristiques générales :
  - o Cohérence interne et avec l'industrie.
  - o Séparation des données et des traitements.
  - o Description de l'existant pour aller vers le futur.
  - o Prise en compte du système d'information.

2.3.8.2 Le modèle relationnel

Le modèle relationnel représente la base de données comme un ensemble de tables, sans préjuger de la façon dont les informations sont stockées dans la machine. Les tables constituent donc la structure logique du modèle relationnel. Au niveau physique, le système est libre d'utiliser n'importe quelle technique de stockage (fichiers séquentiels, indexa ge, adressage dispersé, séries de pointeurs, compression…) dès lors qu'il est possible de relier ces structures à des tables au niveau logique. Les tables ne représentent donc qu'une abstraction de l'enregistrement physique des données en mémoire. De façon informelle, le modèle relationnel peut être défini de la manière suivante :

- les données sont organisées sous forme de tables à deux dimensions, encore appelées relations, dont les lignes sont appelées n-uplet ou tuple en anglais.
- les données sont manipulées par des opérateurs de l'algèbre relationnelle.
- l'état cohérent de la base est défini par un ensemble de contraintes d'intégrité.

- ✓ **Eléments constructifs du modèle relationnel**

Dans ce modèle, les données sont représentées par de tables, sans préjuger de la façon dont les informations sont stockées dans la machines. Les tables constituent donc la structure logique du modèle relationnel. Les tables ne représentent donc qu'une abstraction de l'enregistrement physique des données en mémoire. [27]

Le père des bases relationnelles est Edgar Frank CODD. Chercheur chez IBM à la fin des années 1960, Les objectifs du modèle relationnel sont :

- Proposer des schémas de données faciles à utiliser.
- Améliorer l'indépendance logique et physique.

- Mettre à la disposition des utilisateurs des langages de haut niveau.

Dans un modèle relationnel on trouve :

- Attribut : Un attribut est un identifiant (un nom) décrivant une information stockée dans une base.
- Domaine : Le domaine d'un attribut est l'ensemble, fini ou infini, de ses valeurs possibles.
- Relation : Une relation est un sous-ensemble du point cartésien de n domaines d'attributs (n>0).
- Schéma de relation : Un schéma de relation précise le nom de la relation ainsi que la liste des attributs avec leurs domaines.
- Degré : Le degré d'une relation est son nombre d'attributs.
- Occurrence ou n-tuples ou tuples : Une occurrence, ou n-tuples ou tuples, est un élément de l'ensemble figuré par une relation.
- Cardinalité : La cardinalité d'une relation est son nombre d'occurrences.
- Clé candidate : Une clé candidate d'une relation est un ensemble minimal des attributs de la relation dont les valeurs identifient à coup sur l'occurrence.
- Clé primaire : Une clé primaire d'une relation est une de ses clés candidate. Pour signaler la clé primaire, ses attributs sont généralement soulignés.
- Clé étrangère : Une clé étrangère dans une relation est formée d'un ou plusieurs attributs qui constituent une clé primaire dans une autre relation.
- Schéma relationnel : Un schéma relationnel est constitué par l'ensemble des schémas de relation.
- Base de données relationnelle : Une base de données relationnelle est constituée par l'ensemble des n-uplets des relations du schéma relationnel.

2.3.9 Normalisation

Les formes normales sont différents stades de qualité qui permettent d'éviter la redondance dans les bases de données relationnelles afin d'éviter ou limiter : les pertes de données, les incohérences au sein des données, l'effondrement des performances des traitements.

Le processus de normalisation consiste à remplacer une relation donnée par certaines projections afin que la jointure de ces projections permette de retrouver la relation initiale.[25]

2.3.10 Le Système de Gestion de Base de Données

Définition : Afin de pouvoir contrôler les données ainsi que les utilisateurs, le besoin d'un système de gestion s'est vite fait ressentir. La gestion de la base de données) ou en anglais DBMS (DataBase Management System). Le SGBD est un ensemble de services (applications logicielles) permettant de gérer les bases de données c'est-à-dire :

- ✓ Permettre l'accès aux données de façon simple.
- ✓ Autoriser un accès aux informations à de multiples utilisateurs.
- ✓ Manipuler les données présentes dans la base de données (insert, suppr, modif).

2.3.10.1 Modéles de SGBD

Les bases de données sont apparues à la fin des années 60, à une époque où la nécessité d'un système de gestion de l'information souple se fait ressentir. Il existe cinq modèle de SGBD, les différenciés selon la représentation des données qu'elle contient :

- Modèle Hiérarchique : Le modèle hiérarchique est une forme de système de gestion de base de données qui lie des enregistrements dans une structure arborescente de façon à ce que chaque enregistrement n'ait qu'un seul processeur Il s'agit du premier modèle de SGBD.
- Modèle réseau : Ce modèle utilise des pointeurs vers des enregistrements.
- Modèle relationnel : Dans ce modèle les données sont structurées suivant les principes de l'algèbre relationnel. En d'autre terme, les données sont enregistrées dans des tables.
- Modèle déductif : Les données sont représentées sont forme de table, mais leur manipulation se fait par calcul de prédicats.
- Modèle objet : Les données sont stockées sous forme d'objets, c'est-à-dire de structures appelées classes présentant des données membres, les champs sont des instances de ces classes.

2.3.10.2 Objectif de SGBD

Des objectifs principaux ont été fixés aux SGBD dès l'origine de ceux-ci et ce, afin de résoudre les problèmes causés par la démarche classique.
Ces objectifs sont ci-après : [25]

- ✓ Indépendance physique : La façon dont les données sont définies doit être indépendante des structures de stockage utilisées.
- ✓ Indépendance logique : Un même ensemble de données peut être vu différemment par des utilisateurs différents.

- ✓ Accès aux données : Se fait par intermédiaire d'un langage de manipulation de données (LMD).
- ✓ Administration centralisée de données (intégration) : toutes les données doivent être centralisées dans un réservoir unique commun à toutes les applications.
- ✓ Non redondance : Chaque donnée doit être présente qu'une seule fois dans la base.
- ✓ Cohérence de données : Les données sont soumises à un certain nombre de contraintes d'intégrité qui définissent un état cohérent de la base.
- ✓ Partage de données : Il s'agit de permettre à plusieurs utilisateurs d'accéder aux données au même moment de manière transparente.
- ✓ Sécurité des données : Les données doivent pouvoir être protégées contre les accès non autorisés.
- ✓ Résistance aux pannes.

2.3.10.3 Caractéristiques de la conception de BDD en SGBD

- o Les entités deviennent des tables dans le modèle relationnel.
- o Les identifiants deviennent des clés primaires et secondaires.
- o Les propriétés deviennent des attributs de la table.
- o L'ensemble des propriétés devient une ligne.

✓ les types de relations qu'on peut rencontrer sont au nombre de quatre :

- o un à un (1,1).
- o un à plusieurs (1, n).
- o plusieurs à plusieurs (n, n).
- o plusieurs à un (n, 1).

2.3.10.4 Méthode d'accès aux données

La base de données étant créée, il sera nécessaire que les utilisateurs trouvent bien un moyen d'y accéder par diverses opérations telles que :

- ✓ L'ajout de données.
- ✓ La mise à jour des données.
- ✓ La suppression des données et tant d'autres.

Cela se fait grâce à une méthode d'accès aux données.

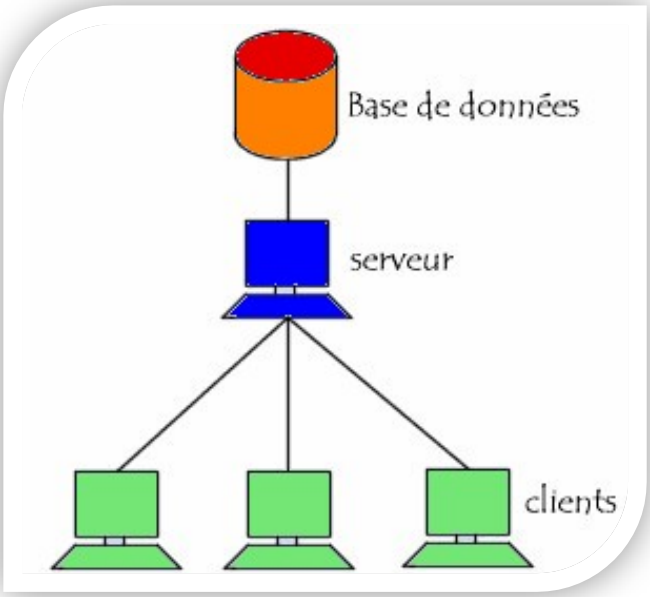

Figure 2.11 : l'accès à une BDD par plusieurs utilisateurs simultanément

2.3.10.5 Les principaux SGBD (système de gestion de bases de données)

Un Système de Gestion de Base de Données (SGBD) est un logiciel (ou un ensemble de logiciels) permettant de manipuler les données d'une base de données. Manipuler, c'est-à-dire sélectionner et afficher des informations tirées de cette base, modifier des données, en ajouter ou en supprimer (ce groupe de quatre opérations étant souvent appelé "CRUD", pour Create, Read, Update, Delete) parmi les SGBD les plus importants on cite :

- **Borland Paradox**
- **Filemaker**
- **IBM DB2**
- **Ingres**
- **Interbase**
- **Microsoft SQL server**
- **Microsoft Access**
- **Microsoft FoxPro**
- **Oracle**
- **Sybase**
- **MySQL**
- **PostgreSQL**
- **mSQL**

Tableau 2.1 : listes des principaux SGBD

2.4 Conclusion

Dans ce chapitre nous avons essayé de présenter une introduction au langage de modélisation UML, on a présenté leurs diagrammes et précisément le diagramme de classe où nous avons souligné tous les éléments principaux qui le compose.

Pour bénéficier des avantages d'UML dans la modélisation des systèmes sans omettre la tâche de vérification des modèles résultants d'une telle modélisation, plusieurs travaux se sont focalisés sur le principe de transformation de modèles pour obtenir des modèles dont la vérification est abordable.

Nous avons présenté Les bases de données et le modèle relationnelles d'une façon générale, Il est important de maîtriser les différentes notions de ce modèle qui est aujourd'hui le plus enseigné au niveau des formations, mais aussi le plus utilisé en entreprise.

Chapitre 03 :

Génération des outils pour la transformation des DC vers BDDR

CHAPITRE 3
GENERATION DES OUTILS POUR LA TRANSFORMATION DES DC VERS BDDR

3.1 Introduction

Avant d'aborder les techniques de transformation de modèle et les règles de la transformation, il est nécessaire de définir le méta-modèle de « diagrammes de classe UML ».

Pour définir et réaliser notre méta-modèle nous allons utiliser le standard EMF qu'on a déjà abordé dans le chapitre 01.

3.2 Génération d'un modèle relationnel à partir des diagrammes de classes UML :

3.2.1. Les formes de normalisation

- **Première forme normale**

Une relation est en première forme normale si et seulement si tout attribut de cette relation est atomique, c'est-à-dire qu'un attribut n'est composé que d'une valeur significative à la fois (par opposition à une adresse complexe, par exemple) et mono value.

- **Deuxième forme normale**

Une relation est en deuxième forme normale si et seulement si elle est en première forme normale, qu'elle n'a qu'une seule clé, et que tout attribut n'appartenant pas à la clé ne dépend pas d'une partie de cette clé.

- **Troisième forme normale**

Une relation est en troisième forme normale si et seulement si elle est en deuxième forme normale et si chaque attribut non clé est en dépendance non transitive avec la clé primaire.

- **Forme normale de Boyce-Codd (BCNF)**

Une relation est en forme normale de Boyce-Codd si et seulement si seuls les déterminants sont des clés candidates.

- **Quatrième et cinquième formes normales**

Pour traiter les cas plus complexes ou apparaissent des associations n-m entre les relations du schéma relationnel, la théorie de la normalisation introduit, au-delà des simples dépendances fonctionnelles qui sont utilisés dans les énoncés de ces premières formes normales, les notions de dépendances multi-valuées et de composantes de jointure (ou dépendance de jointure). L'étude de ces notions sort du cadre du présent travail. Elles sont simplement mentionnées ici puisqu'elles font partie de l'arsenal des outils qui ont étés utilisés par les auteurs du processus pour garantir sa qualité.

L'application de ces principes permet une amélioration considérable de la qualité de la base obtenue à partir du schéma ainsi normalisé. Elle facilite le respect des contraintes d'intégrité imposées lors de la conception en évitant la redondance des données et donc en rendant les mises à jour plus aisées.

3.2.2 Diagramme de classe VS Modèle Relationnel

Il est possible de traduire un diagramme de classe en modèle relationnel. Bien entendu, les méthodes des classes ne sont pas traduites. Aujourd'hui, lors de la conception de base de données, il devient de plus en plus courant d'utiliser la modélisation UML plutôt que le traditionnel modèle entités-association.

- **Classe avec attributs**

Chaque classe devient une relation. Les attributs de la classe deviennent des attributs de la relation. Si la classe possède un identifiant, il devient la clé primaire de la relation, sinon, il faut ajouter une clé primaire arbitraire.

- **Association 1 vers 1**

Pour représenter une association 1 vers 1 entre deux relations, la clé primaire de l'une des relations doit figurer comme clé étrangère dans l'autre relation.

- **Association 1 vers plusieurs**

Pour représenter une association 1 vers plusieurs, on procède comme pour une association 1 vers 1, excepté que c'est forcément la relation du côté plusieurs qui reçoit comme clé étrangère la clé primaire de la relation du côté 1.

- **Association plusieurs vers plusieurs**

Pour représenter une association du type plusieurs vers plusieurs, il faut introduire une nouvelle relation dont les attributs sont les clés primaires des relations en association et dont la clé primaire est la concaténation de ces deux attributs.

- **Classe-association plusieurs vers plusieurs**

Le cas est proche de celui d'une association plusieurs vers plusieurs, les attributs de la classe-association étant ajoutés à la troisième relation qui représente, cette fois ci, la classe-association elle-même.

3.3 Génération d'un outil EMF pour « les diagrammes de classe »

Basant sur le méta_modèl décrit dans la figure 3.1, nous avons généré un outil permettant de créer des diagrammes de classe sous format XMI.

3.3.1 Un méta-modèle pour le diagramme de classe

Pour définir un méta-modèle (ou une ecore) EMF utilise le diagramme de classe. Donc, un DC (comme le montre le méta-modèle de la figure 3.1) est composée de huit classes :

- ✓ ***ClassModel***
- ✓ ***Association***
- ✓ ***Class***
- ✓ ***Classifier***
- ✓ ***PrimitiveDataType***
- ✓ ***Attribute***
- ✓ ***Méthode***
- ✓ ***Paramètre***

- Pour construire un modèle EMF plusieurs formats disponibles
  - ✓ Modèle Ecore (voir la suite)
  - ✓ Classes Java annotées
  - ✓ Modèle de classes Rose
  - ✓ Modèle UML
  - ✓ XML Schéma

Nous utiliserons par la suite un modèle Ecore puisque l'outillage fourni par EMF facilite la construction.

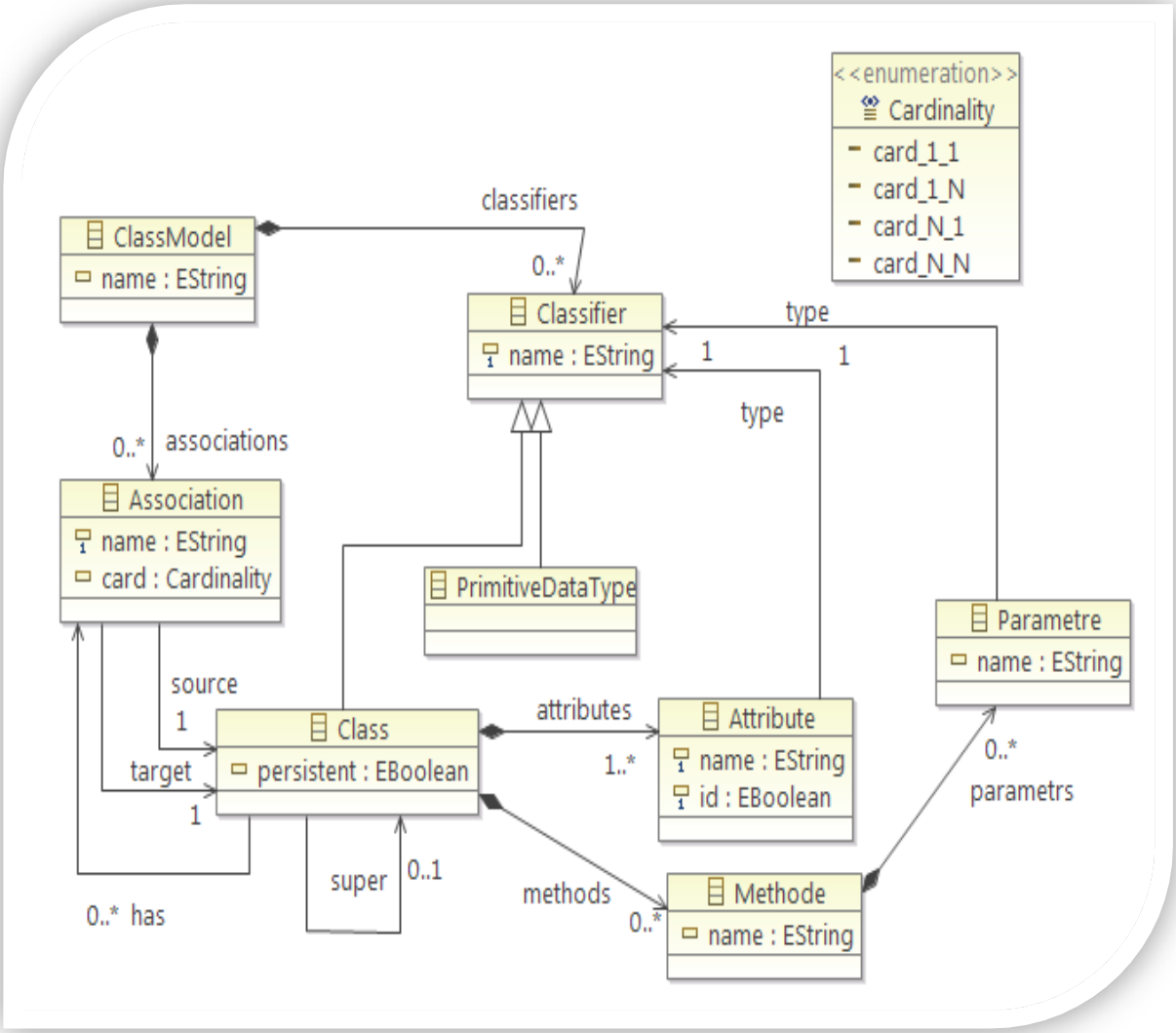

Figure 3.1 : un méta-modèle pour les diagrammes de classe

3.3.2 La génération d'un outil pour les diagrammes de classe

En se basant sur le méta-modèle (voir la figure 3.1) on peut générer un outil qui nous permettra de créer des exemples de diagramme de classe (des instances) avec les étapes suivantes :

- **Création d'un projet EMF vide (File → New → Project…)**

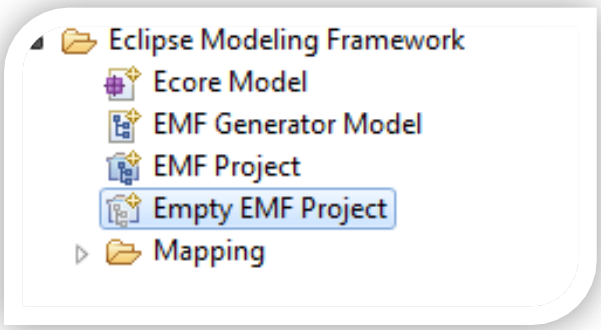

Figure 3.2 : Crée projet EMF vide.

- **Création d'un diagramme Ecore (New → Other →)**

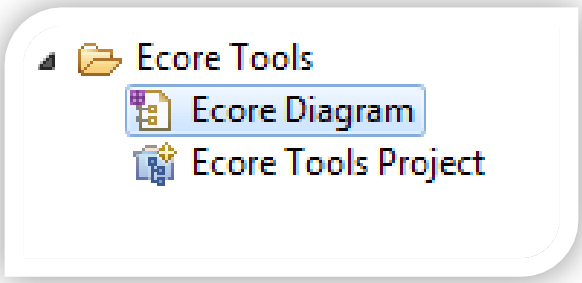

Figure 3.3 : Crée diagramme Ecore.

- **Défini les éléments de notre méta-modèle de DC**

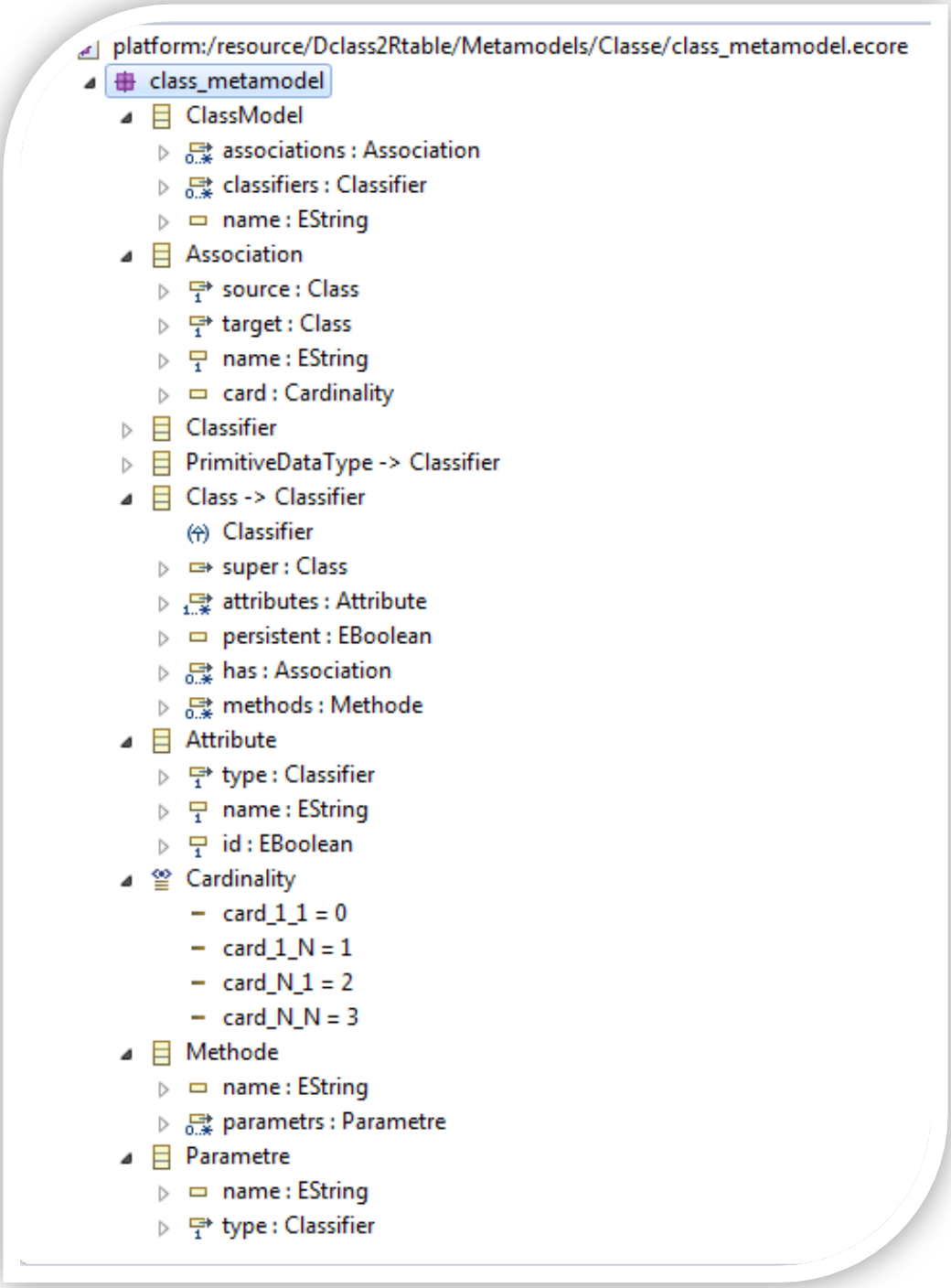

Figure 3.4 : les éléments du méta-modèle de DC.

➢ Création de *genmodel*

À partir du modèle défini précédemment, il est possible de générer du code Java dédié à la création des instances de ce modèle.

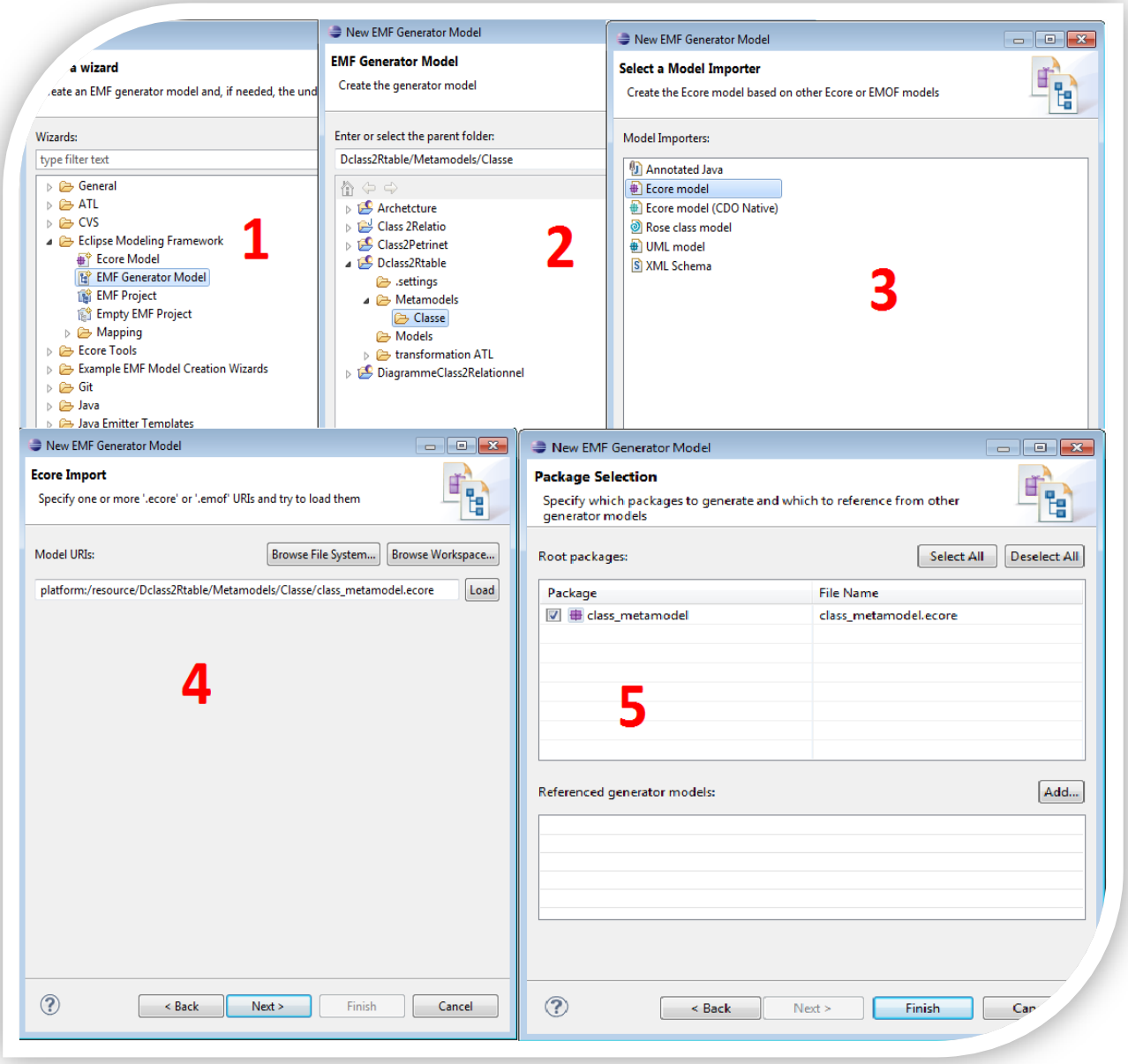

Figure 3.5 : génération de « genmodel »

- **Eléments pour la création d'un DC**

La génération de code nécessite la création d'un modèle de génération appelé « genmodel », Ce modèle contient des informations dédiées uniquement à la génération et qui ne pourraient pas être intégrées au modèle.

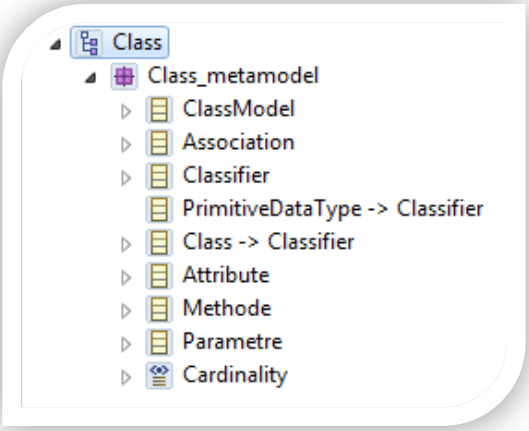

Figure 3.6 : les éléments de « genmodel » de DC.

- **Génération des projets (édit, editor, et tests)**

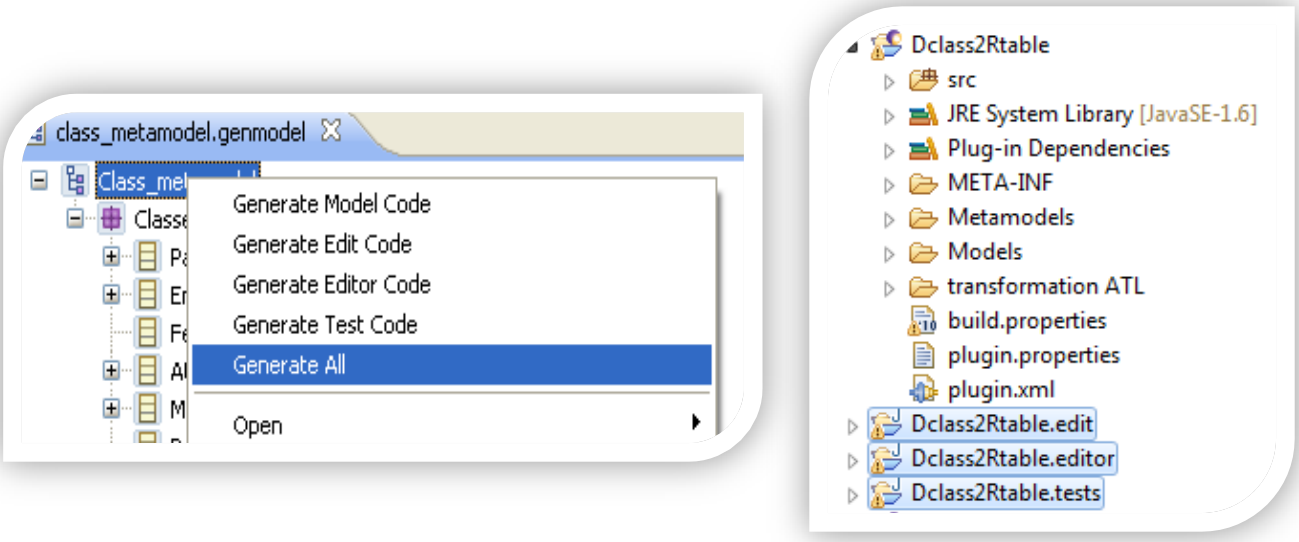

Figure 3.7 : génération des projets :edit, .editor, .tests

- **Exécuter les plugins :**

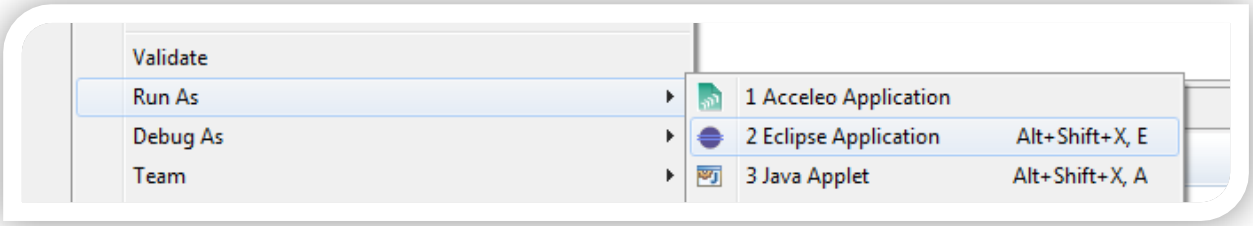

Figure 3.8 : exécuter les plugins

Pour tester les nouveaux plugins, une deuxième instance d'Eclipse doit être lancée, et pour vérifie que les plugins déjà produit, amenez le « Help/About Eclipse plate-forme » dialogue, clique sur le bouton « plug-in Détails », et vérifie.

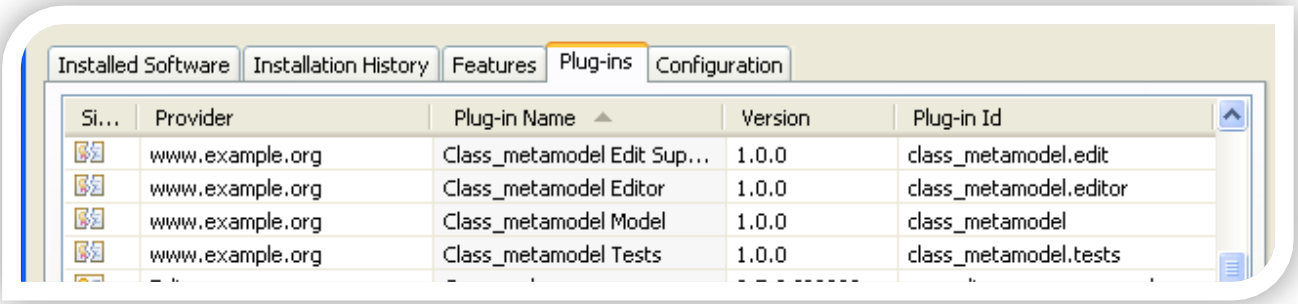

Figure 3.9 : plugin détails de classe modèle

- **Création d'une nouvelle modèle de diagramme de classe**

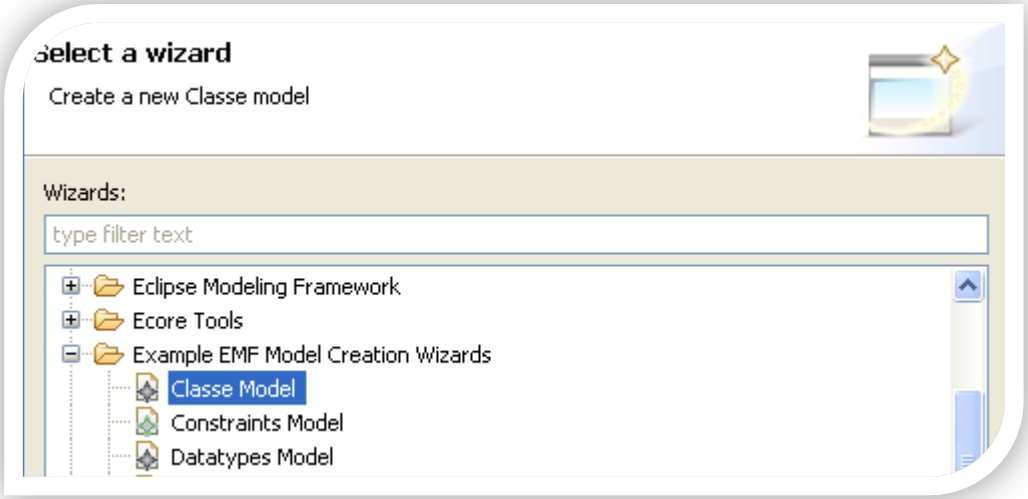

Figure 3.10 : création d'une classe modèle

3.3.3 La génération d'un outil pour la transformation d'un DC BDD

- **Création d'un ATL projet (File → New → Project…)**

Figure 3.11 : créer un nouveau projet ATL

- **Création des dossiers (metamodels, models, transformations)**

Pour l'organisation de travail on va créer trois dossiers dans le projet ATL.

Clic droit sur le projet puis (New → Other …).et choisi folder.

Figure 3.12 : création des dossiers

- **Création les méta-modèles Ecore**

Nous allons créer les méta-modèles de DC Ecore.

On a déjà vue comment faire ça.

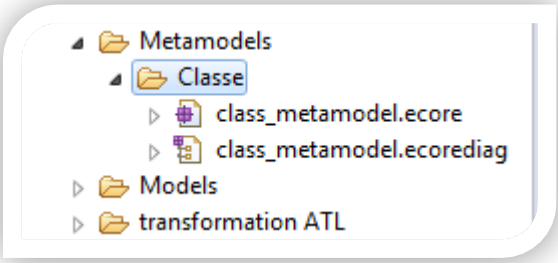

Figure 3.13 : création les méta-modèles

➢ **Génération de modèle conforme à méta-modèles de DC**

On va générer le modèle conformément au méta-modèle des DC.

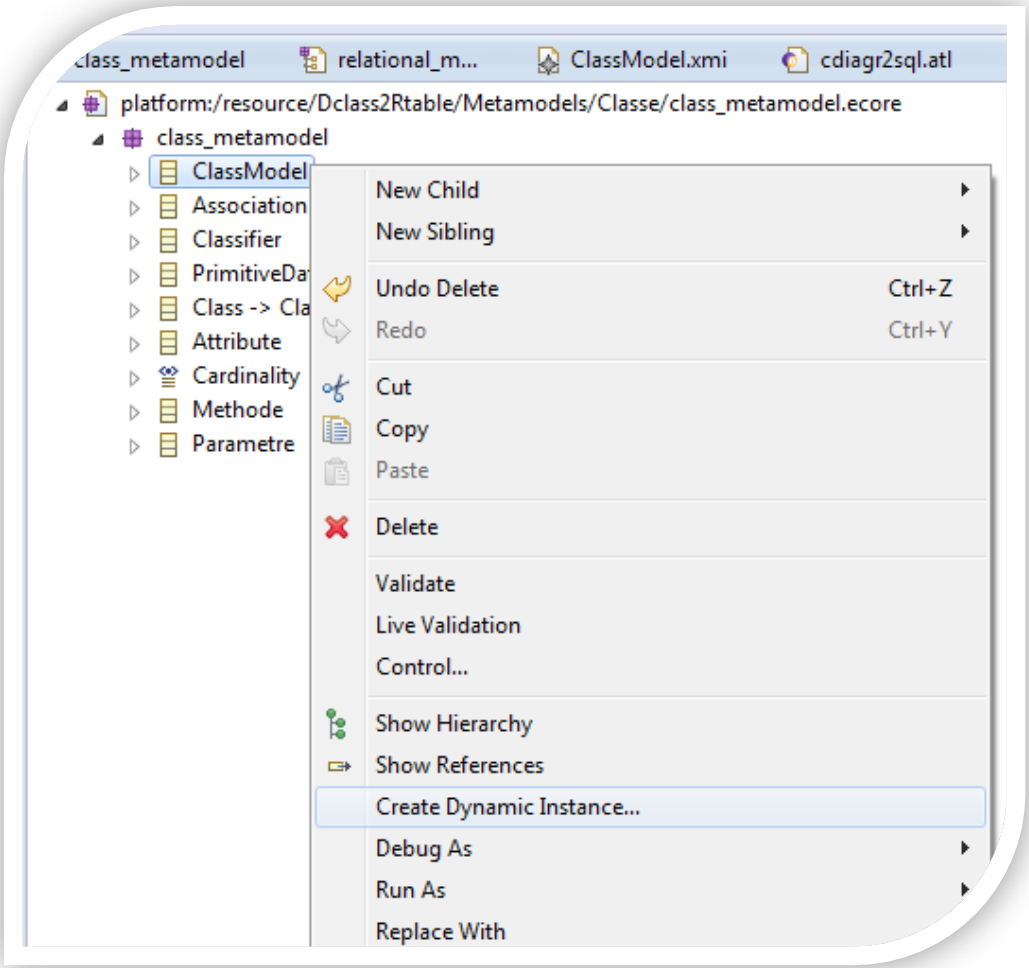

Figure 3.14 : création des instances dynamiques

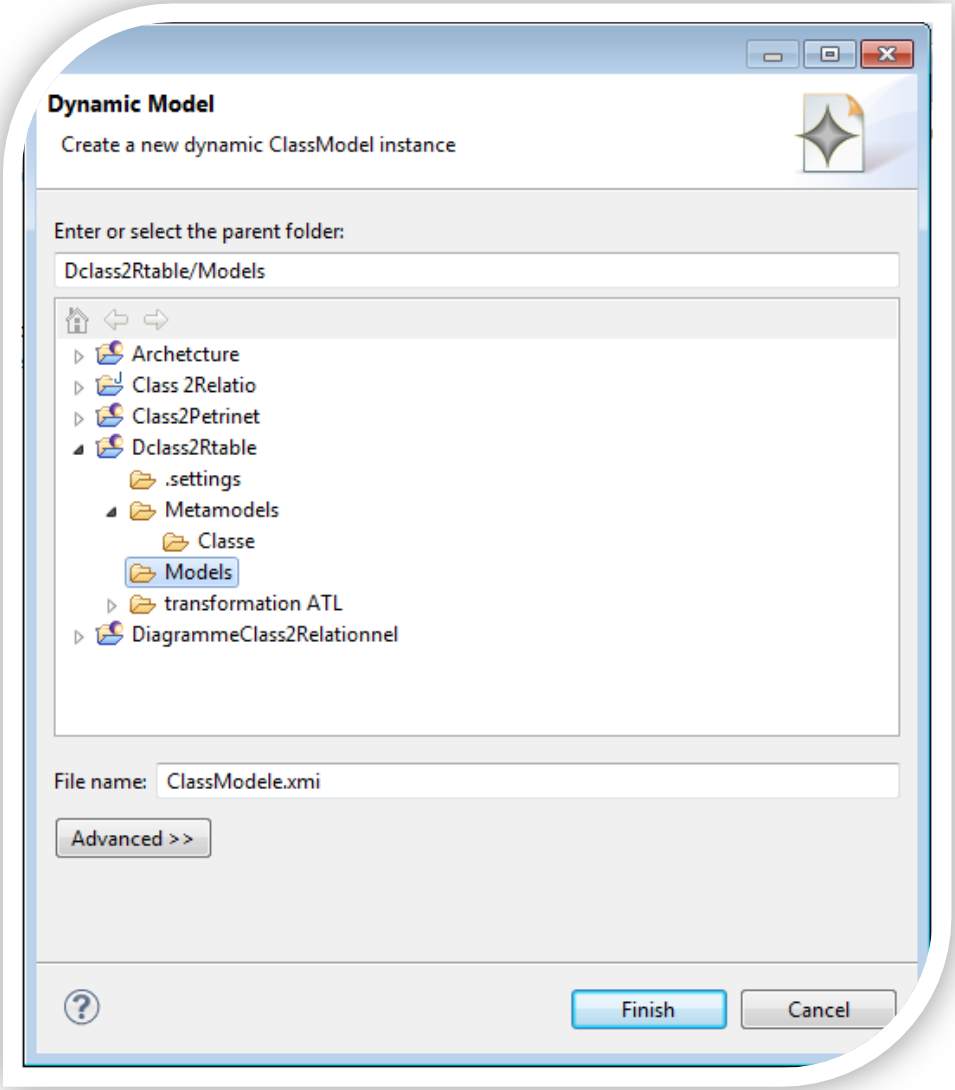

Figure 3.15 : choisir le nom de modèle et le dossier de l'emplacement

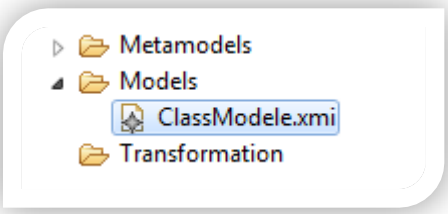

Figure 3.16 : modèles généré

- **La création des éléments de modèle à transformer**

La figure 3.17 décrit un exemple, au format XMI instancié à partir de notre outil basant sur le méta_modèl proposé pour les DCs, cet exemple contient six classes, après avoir discuté l'ensemble des «helpers» ATL on va l'utiliser pour montrer l'utilité de notre démarche.

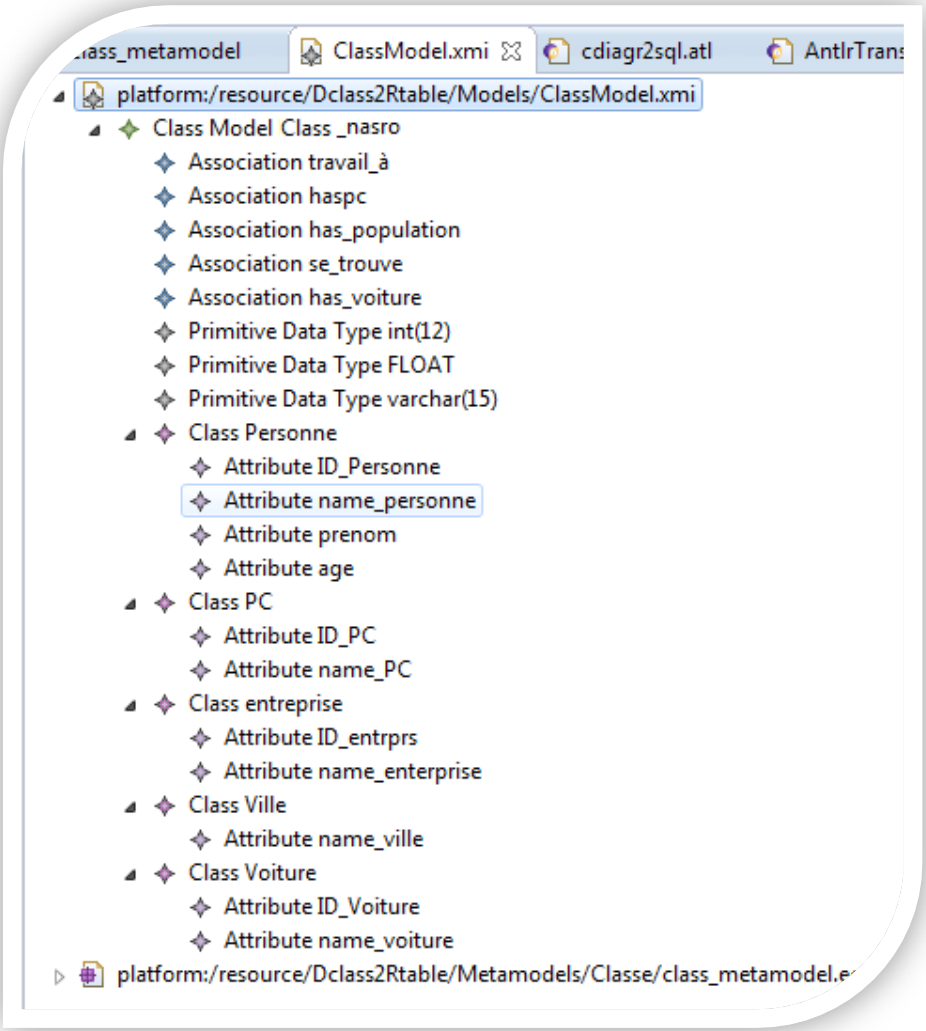

Figure 3.17 : La création des éléments de modèle à transformer

- **Transformation « Model To Text »**

Il existe deux type de transformation ATL : **« Model To Model » et « Model To Text »**, dans notre cas, nous utilisons la deuxième.

Après avoir créé un fichier ATL (figure 3.18) l'ensemble de nos helpers permettant de transformer un DC à un fichier de « BDD.sql ».

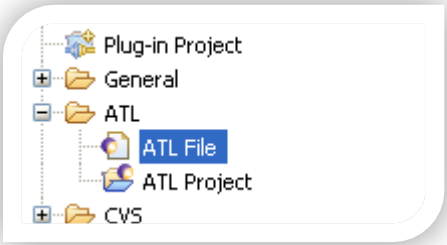

Figure 3.18 : création ATL file

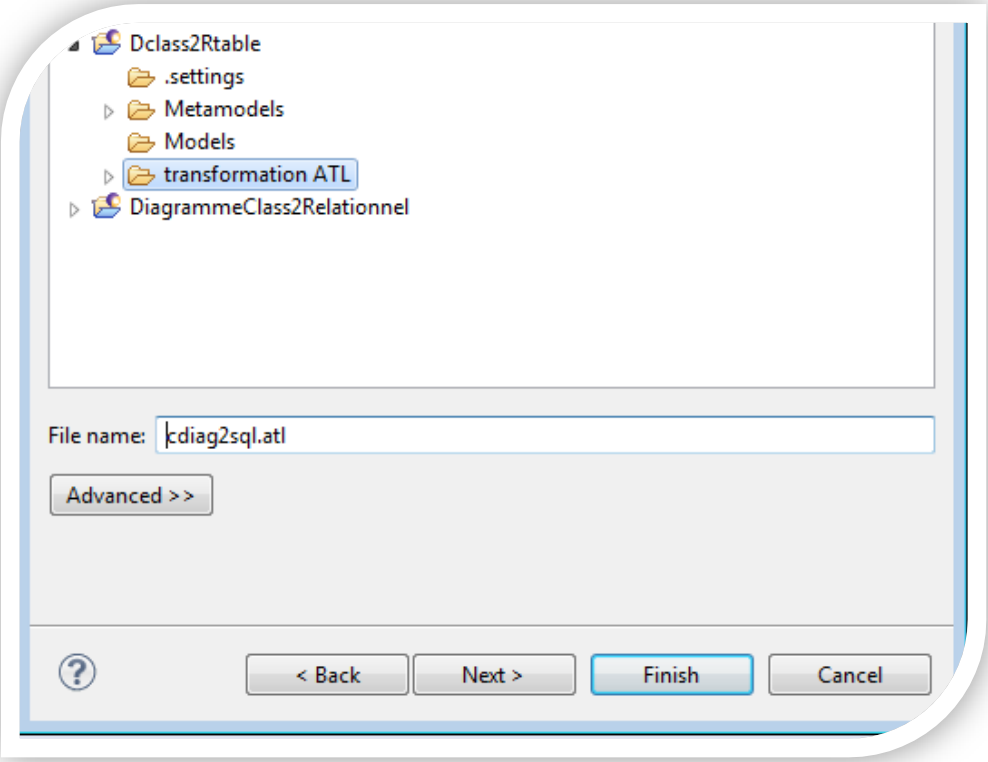

Figure 3.19 : sélection le nom d'ATL file

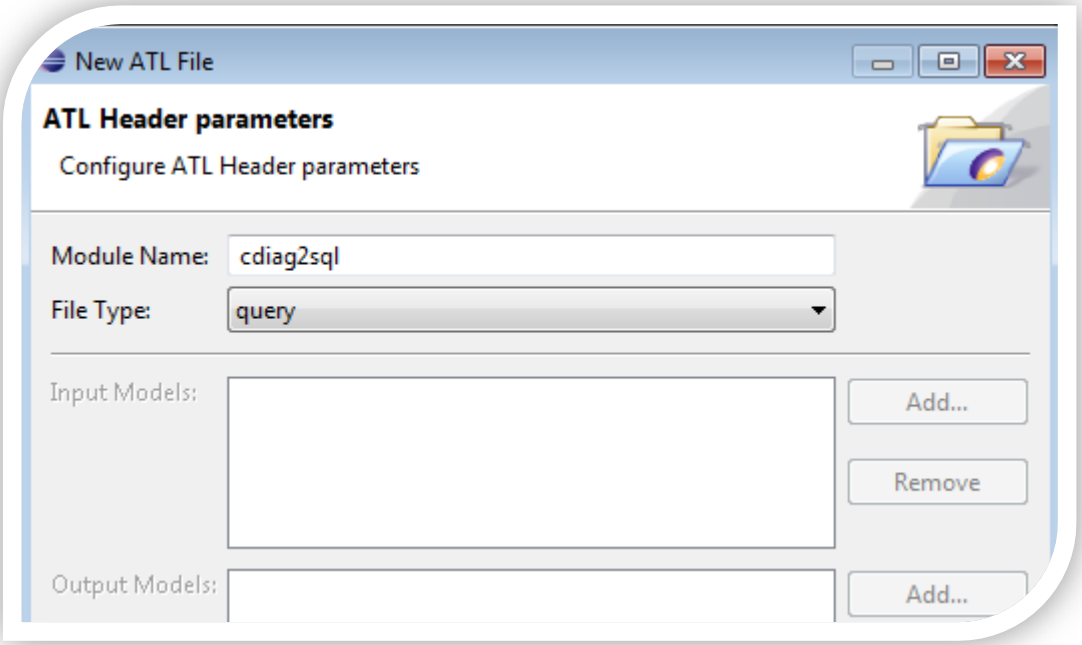

Figure 3.20 : sélection le type query pour fichier ATL

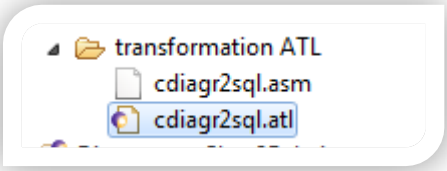

Figure 3.21 : ATL file généré

3.3.4 Utilisation d'ATL

Considérons notre problème « Génération d'un code LDD SQL normalisé à partir de diagrammes de classes UML ». Faisons abstraction des calculs indiqués dans le processus de conception décrit dans le mémoire, des nécessaires interactions avec l'utilisateur, et concentrons-nous sur la traduction proprement dite du modèle UML en code LDD SQL.

Les informations que nous devons apporter au système pour nourrir le moteur de traduction ATL sont :

• Le méta modèle UML 2.0

• Le modèle UML (le modèle du domaine conforme au méta modèle UML 2.0)

• Le code ATL de transformation

Ces éléments doivent permettre de générer automatiquement

• Le code LDD SQL

3.3.5 Le module ATL :

L'écriture du programme de transformation proprement dit (annexe 1) ne pose en pratique aucun problème. Elle se résume simplement à la création d'un fichier texte avec l'extension atl, où l'on écrit les helpers qui transforme le modèle de diagramme de classe vers le code LDD SQL.

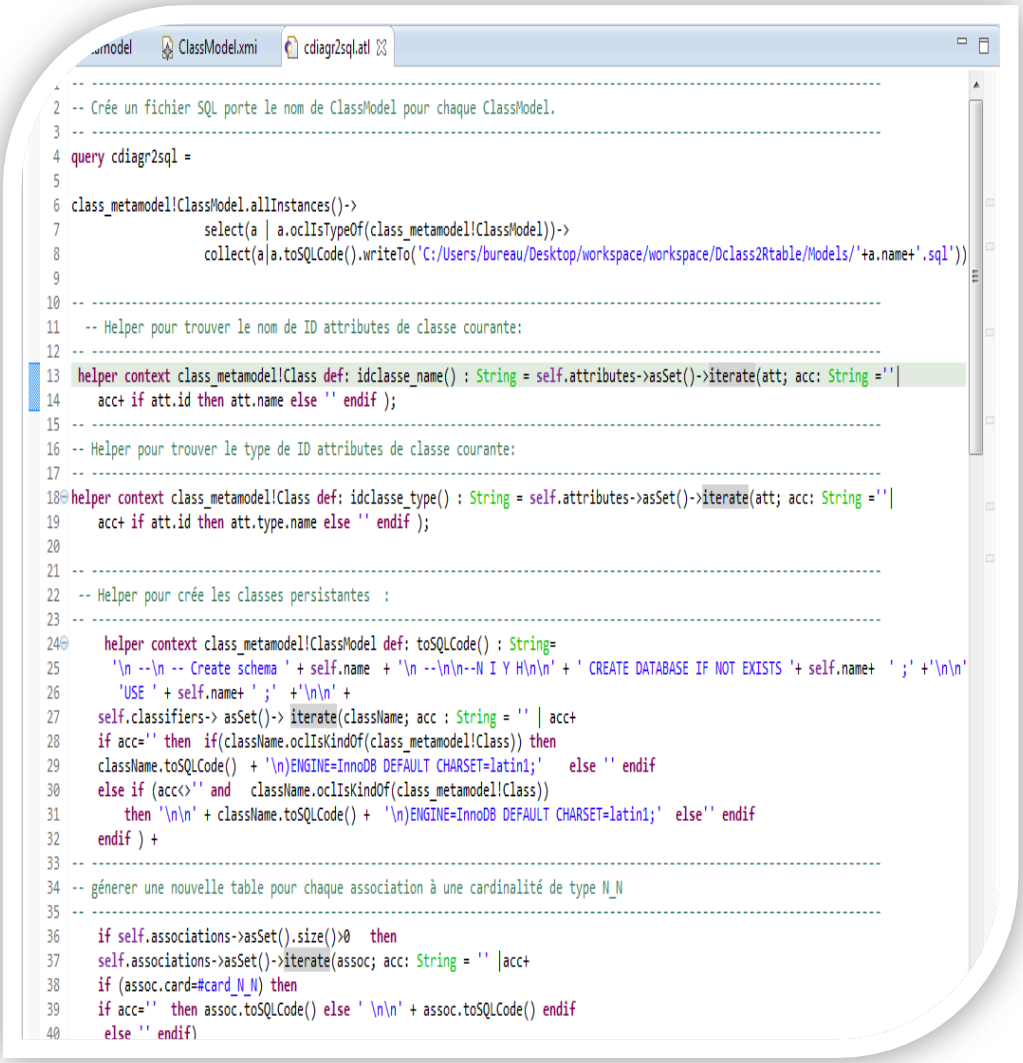

Figure 3.22 : Extrait du code transformation du module ATL

3.3.6 Requêtes et la génération de texte :

Les programmes ATL sont généralement basés sur les règles de transformation. Cependant, il y'a aussi ATL requête. Les requêtes permettent d'analyser les modèles et pour calculer une sortie qui n'est pas nécessairement un modèle, Cela les rend très pratique pour générer du texte ou code à partir d'un modèle.

Programmes de requête ALT doivent commencer par une instanciation de requête qui se composent de la clé « query », une variable de requête, un signe égal et une expression OCL initialisation de la variable de requête. Dans l'exemple suivant, vous voyez un extrait du programme de CDiagr2Sql qui transforme les modèles de class_metamodel code. Avec la fonction all Instances il traverse tous les éléments de modèles class_metamodel. La fonction collect appelle les fonctions d'assistance toSQLCode et concatène les valeurs de chaîne qu'ils renvoient. La fonction WriteTo écrit la concaténation dans un fichier dédié.

On note qu'il y'a plusieurs fonctions toSQLCode auxiliaires. Pendant l'exécution, la fonction d'assistance avec le type de contexte qui correspond le mieux sera choisi pour l'exécution.

En général, on peut dire que cette approche simplifie la génération de texte ou de code parce que le programmeur est pris en charge dans le traitement de codage de chaque type de méta modèle séparément.

```
query CDiagr2sql =
class_metamodel!ClassModel.allInstances()->
                collect(a|a.toSQLCode().
                writeTo('C:/'+a.name+'.sql');
```

Figure 3.23 : Requêtes et la génération de texte « .sql »

3.4 Transformation des diagrammes de classe vers une Base de données

3.4.1 Les règles de transformation

Afin de produire les méta-modèles générés par notre outil (EMF), nous avons proposé des requêtes de transformation avec le langage ATL qui seront exécutées dans un ordre ascendant. L'application de ces requêtes au diagramme de classe crée par notre outil conduit à la génération du Base de données équivalent.

✓ ***Règle 1 :*** **transformation de ClassModel à RDBMSModel**

Cette règle permet de transformer la classe ClassModel à la classe RDBMSModel, dans cette étape on va affecter les attributs de la classe package à des attributs de la classe RDBMSModel, ainsi les contenants classes de la classe package à la classe RDBMSModel.

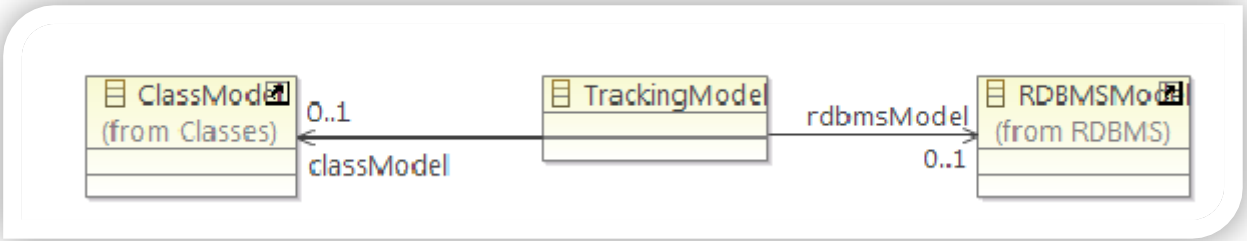

Figure 3.24 : transformation de ClassModel à RDBMSModel

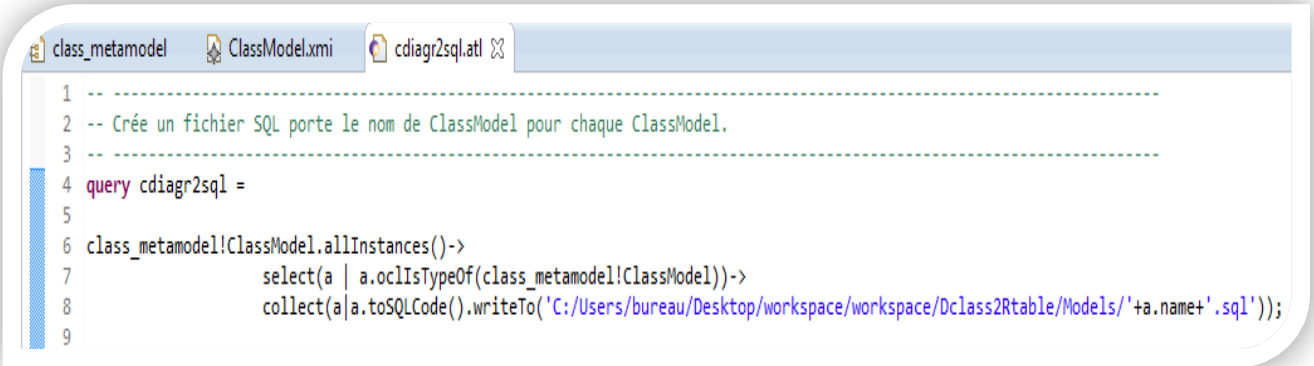

Figure 3.25 : générer le fichier SQL de la BDD en ATL

✓ ***Règle 2 :*** **transformation de Class à *Une table***

Cette règle permet de transformer la classe class à une table, donc faire une équivalence entre les classes class et les tables SQL, à chaque classe on peut faire correspondre une relation telle que :

- le nom de la relation reprend le nom de la classe,
- le nom des attributs de la relation sont issus des noms des attributs de la classe.

Figure 3.26 : transformation de Class à Une table

La figure suivant décrit cette règle.

```
21 -- ------------------------------------------------------------------------------------------------------------------
22  -- Helper pour crée les classes persistantes  :
23 -- ------------------------------------------------------------------------------------------------------------------
24     helper context class_metamodel!ClassModel def: toSQLCode() : String=
25       '\n --\n -- Create schema ' + self.name+'\n --\n\n--N I Y H\n\n'+' CREATE DATABASE IF NOT EXISTS '+self.name+' ;'+'\n\n'+
26        'USE ' + self.name+ ' ;'  +'\n\n' +
27     self.classifiers-> asSet()-> iterate(className; acc : String = '' | acc+
28     if acc='' then  if(className.oclIsKindOf(class_metamodel!Class)) then
29     className.toSQLCode()  + '\n)ENGINE=InnoDB DEFAULT CHARSET=latin1;'    else '' endif
30     else if (acc<>'' and   className.oclIsKindOf(class_metamodel!Class))
31         then '\n\n' + className.toSQLCode() +  '\n)ENGINE=InnoDB DEFAULT CHARSET=latin1;'  else'' endif
32     endif ) +
```

Figure 3.27 : transformation class vers une table

✓ ***Règle 03 :*** **transformation attribue à colonne**

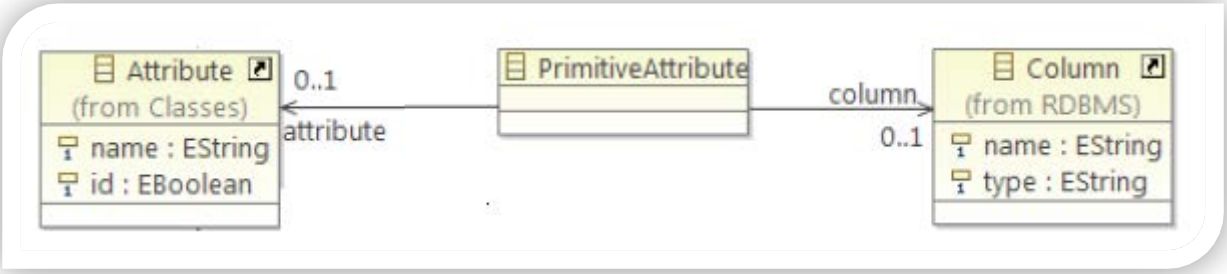

Figure 3.28 : transformation attribue à colonne

Dans cette règle l'équivalence est entre la classe Attribute et la classe Column, c'est –à-dire Pour chaque attribut primitif dans une classe créer une colonne dans le tableau correspondant. Et la figure 3.29 représenter cette règle.

```
45  helper context class_metamodel!Class def: toSQLCode() : String=
46 if self.persistent then  '--\n-- Definition of table '+ '`'+self.name+'`\n--'+  '\n\n create table `' +
47 self.name  +'` (\n'   else '' endif
48   +  self.attributes-> collect(d | d.debug())->iterate(att; acc : String = '' | acc+
49  if acc = '' then  '     '+att.toSQLCode() else  ',\n     ' + att.toSQLCode()
50  endif ) +
```

Figure 3.29 : transformation Attribue vers Colonne

✓ ***Règle 04*** **: transformation Association à clé étrangère**

Figure 3.30 : transformation Association à clé étrangère

Cette règle permet de transformer une association entre les classes à une clé étrangère « ForeignKey » dans la table correspondant. La figure 3.31 représente le code de transformation dédié.

```
if self.has->asSet().size()>0 then

    self.has.asSet()->
    iterate(a; acc: String = '' |
        acc+
        if(a.card=#card_1_1) then
            if(self.name=a.source.name)then',\n     `'+ a.target.idclasse_name()+'` '+a.target.idclasse_type()+
            ' NOT NULL'+',\n     CONSTRAINT `' +a.source.name+'_'+a.target.name
            + '` FOREIGN KEY (`'+ a.target.idclasse_name() +'`) REFERENCES `'+ a.target.name+'`'
            + '(`' +a.target.idclasse_name() + '`)'

            else',\n     `'+ a.source.idclasse_name()+'` '+a.source.idclasse_type()+ ' NOT NULL '+',\n     CONSTRAINT `' +
            a.target.name+'_'+a.source.name
            + '` FOREIGN KEY (`'+ a.source.idclasse_name() +'`) REFERENCES `'+ a.source.name+'`' + '(`' +
                    a.source.idclasse_name() + '`)'

            endif
        else
                if(a.card=#card_1_N) then
                if(self.name=a.target.name) then

                ',\n     `'+ a.source.idclasse_name()+'` '+a.source.idclasse_type()+ ' NOT NULL '+',\n     CONSTRAINT `' +
                a.source.name+'_'+a.target.name
            + '` FOREIGN KEY (`'+ a.source.idclasse_name() +'`) REFERENCES `'+ a.source.name+'`' + '(`' +
                    a.source.idclasse_name() + '`)'

                else '' endif
                else
                if(a.card=#card_N_1) then
                if(self.name=a.source.name) then
                ',\n     `'+ a.target.idclasse_name()+'` '+a.target.idclasse_type()+ ' NOT NULL '+',\n     CONSTRAINT `' +
                a.source.name+'_'+a.target.name
            + '` FOREIGN KEY (`'+ a.target.idclasse_name() +'`) REFERENCES `'+ a.target.name+'`' + '(`' +
                    a.target.idclasse_name() + '`)'
```

Figure 3.31 : Transformation Association vers FK + Colonne

3.5 Exemple

- nous prouvont l'utilité de notre démarche.

La figure suivante (3.32) décrit l'exemple illustré dans la figure (3.33)

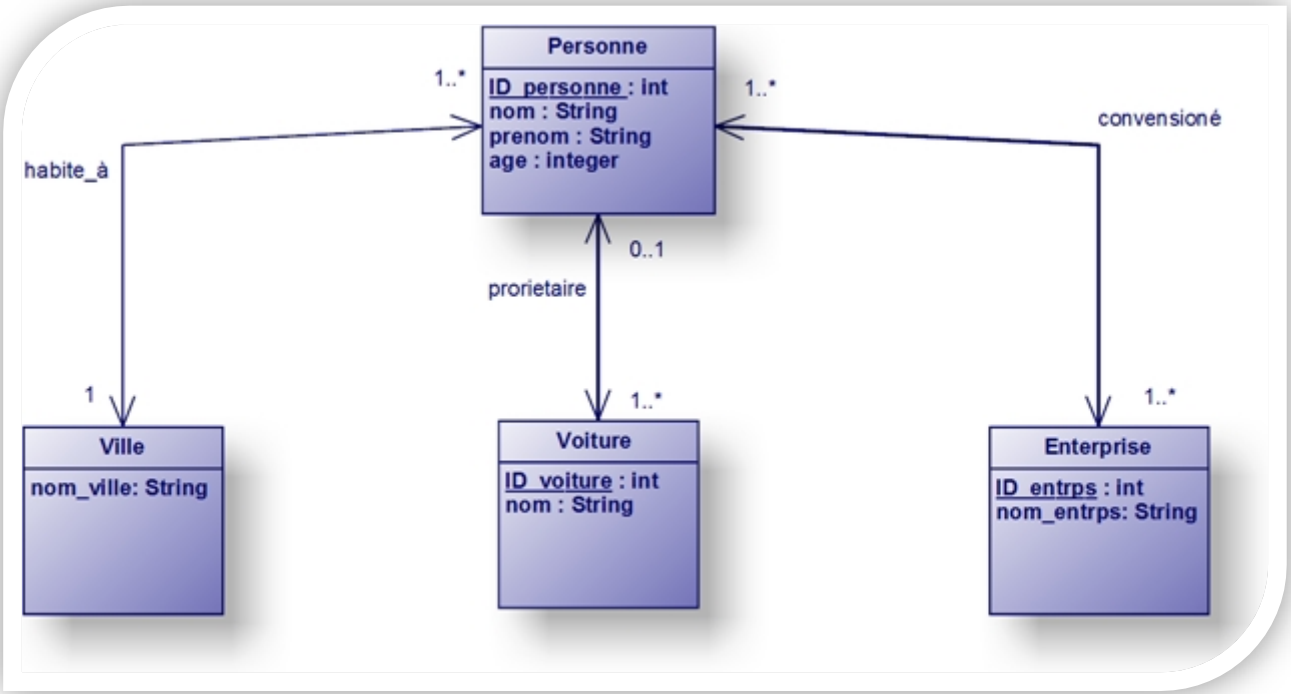

Figure 3.32 : exemple d'une instance d'un diagramme de classe

- **La création d'un diagramme de classe par l'outil généré**

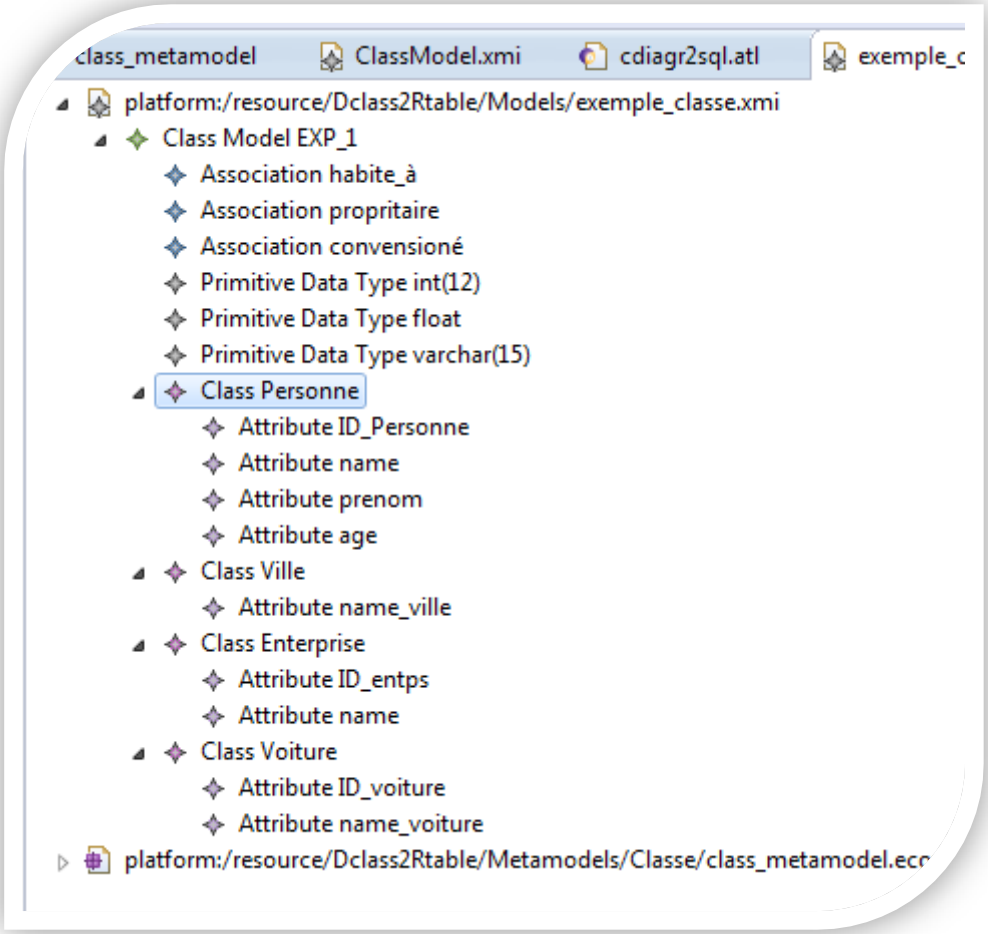

Figure 3.33 : Diagramme de classe simple par l'outil généré

- **Configuration de fichier ATL**

Pour que la transformation de notre modèle s'exécute il faut d'abord créer une configuration pour notre module ATL et lui donné le méta modèle et le modèle que nous voulons transformer.

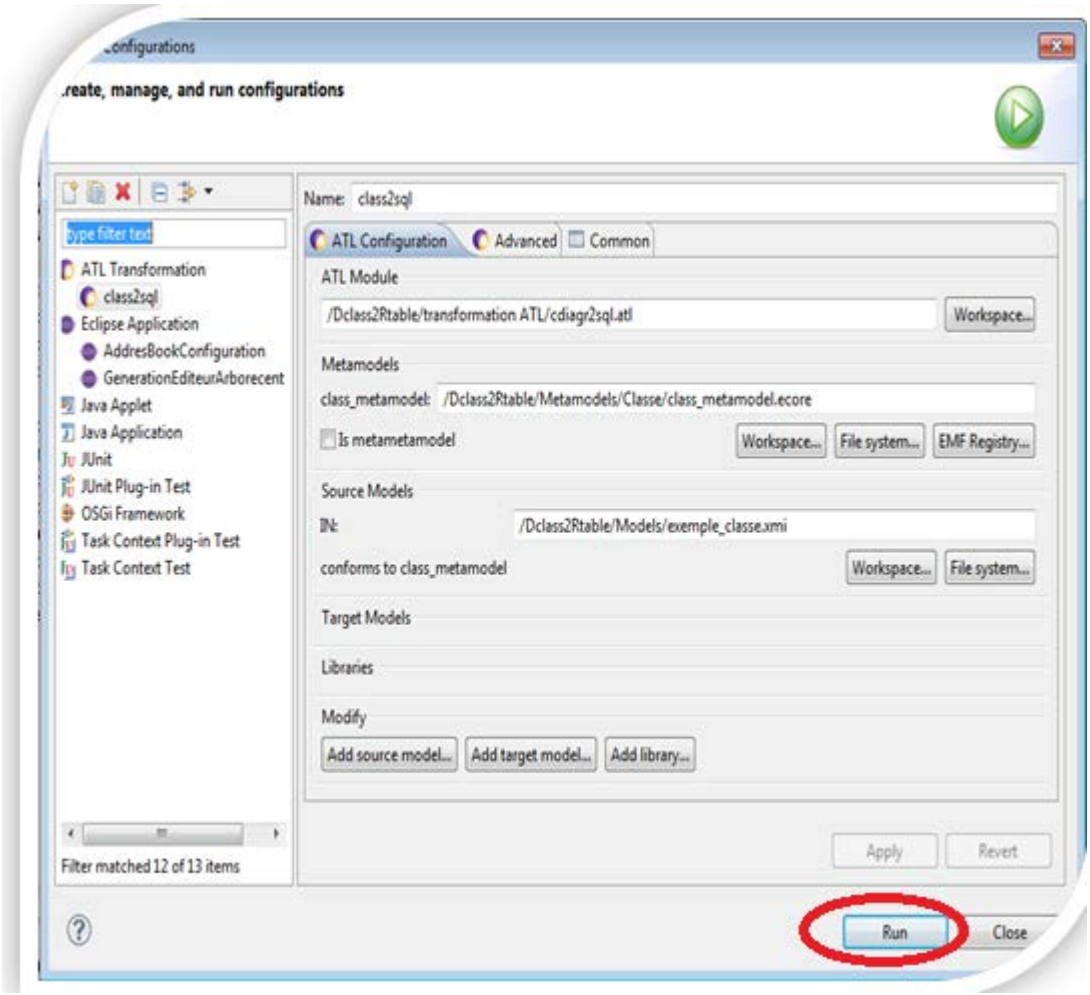

Figure 3.34 : Configuration de fichier ATL

- **La création de fichier SQL**

Après l'exécution « Run » de notre transformation ATL, on a obtenu le fichier « EXP_1.sql » montré dans la figure (3.35)

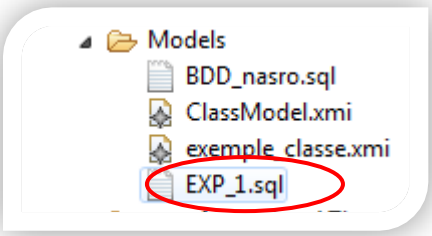

Figure 3.35 : Génération d'une BDD

- **La consultation de fichier SQL généré**

Le code suivant décrit le contenu de fichier généré.

```
--
-- Create schema EXP_1
--

-- N I Y H

CREATE DATABASE IF NOT EXISTS EXP_1 ;

USE EXP_1 ;

--
-- Definition of table `Enterprise`
--

create table `Enterprise` (
    `ID_entps`  int(12) NOT NULL PRIMARY KEY ,
    `name` varchar(15) NOT NULL
)ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Definition of table `Ville`
--

create table `Ville` (
    `name_ville`  varchar(15) NOT NULL PRIMARY KEY
)ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Definition of table `Personne`
--

create table `Personne` (
    `ID_Personne`  int(12) NOT NULL PRIMARY KEY ,
    `name` varchar(15) NOT NULL,
    `prenom` varchar(15) NOT NULL,
    `age` int(12) NOT NULL,
    `name_ville` varchar(15) NOT NULL ,
```

Figure 3.36 : La consultation de fichier SQL généré

- **La création de BDD par l'intermédiaire phpMyAdmin**

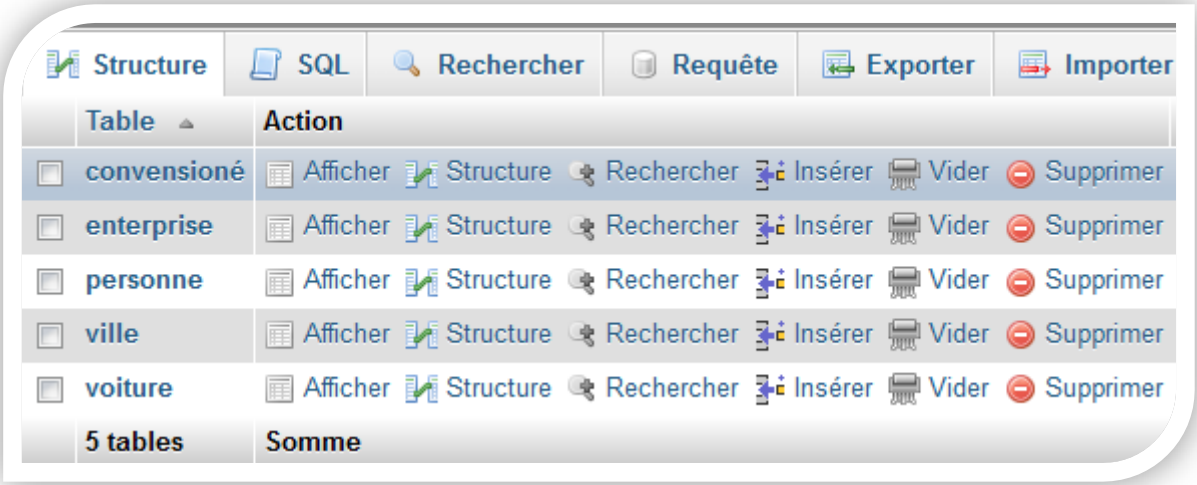

Figure 3.37 : La création de BDD par l'intermédiaire phpMyAdmin

3.6 Conclusions à l'issu de l'expérimentation

3.6.1 Les avantage d'ATL

Au moment de la rédaction de ces lignes, le bilan quant à l'utilisation du langage ATL pour aborder notre problème est positif. Après une prise en main du langage et des concepts qui l'entourent, ATL a pleinement révélé son adéquation avec notre besoin de transformer les modèles qui nous intéressent,

ATL est un Langage déclaratif simple avec le traçage automatique avec une utilisation intensive du OCL : bien soutenu et expressive, Une grande partie du code est souvent dans les aides et pas les règles de transformation réels.

3.6.2 Inconvénients d'ATL

Au stade actuel de son développement, ATL souffre cependant d'une mise en œuvre fastidieuse, en tout cas sur le plan des outils techniques à utiliser pour sa programmation. Son intégration sous forme de plugin dans la plateforme eclipse oblige en effet le programmeur à procéder à l'installation de plusieurs éléments de sources diverses, dont les versions évoluent de manière permanente et souvent incompatibles les unes avec les autres. Ainsi, trouver à un instant t une combinaison des versions de ces éléments qui permettent d'obtenir un environnement de développement à la fois fonctionnel, robuste et permettant de profiter de toutes les fonctionnalités de développement peut parfois relever de la gageure. La situation a cependant tendance à s'arranger avec le souci des développeurs Nantais du langage de livrer des « bundle » complets,

intégrant dans un seul paquet d'installation tous les éléments nécessaires et compatibles entre eux. Un de ces bundle est déjà disponible [28], mais pour l'instant exclusivement pour le système d'exploitation Microsoft Windows.

3.7 Travaux futurs

3.7.1 Explorer les possibilités d'interfaçage

A terme, on sait que le but du processus de conception est destiné à permettre à un utilisateur humain de maîtriser toute la chaîne de conception d'une base de données. Au-delà de la transformation de modèle, ce long processus sera aussi jalonné de nombreuses étapes de modélisation proprement dite et d'annotation. Ces étapes ne sauraient avoir lieu dans de bonnes conditions sans le recours à des interfaces graphiques ergonomiques permettant à l'utilisateur d'entrer ses choix, ses besoins dans le système. Les possibilités d'interfaçage graphique d'ATL ou plus largement les conditions d'articulation d'ATL avec d'autres technologies seront des éléments déterminants pour la réalisation d'un outil complet et performant conforme aux attentes des auteurs de ce mémoire.

3.7.2 Terminer l'extraction de règles de l'algorithme

Cette expérimentation d'ATL a abouti à la création d'un module où quelques « *helpers* » de transformation sont codées. Bien que fonctionnelles, ces helpers sont cependant incomplèts par rapport à l'ensemble des cas de figure traités dans ce mémoire. Bien qu'aucun verrou technique ne soit en vue, et qu'il nous semble qu'ATL soit à même de traiter tous les cas, il reste encore beaucoup à faire pour complètement extraire de ce mémoire toute la sémantique de transformation et l'implémenter concrètement et étoffant le module déjà existant.

3.7.3 éliminer les circuits dans les dépendances référentielles

La génération du code du LDD SQL se fait à partir du schéma relationnel, ce dernier étant obtenu en appliquant pour chaque classe et pour chaque association les « *helpers*» définies dans l'annexe (page 80) . Pour effectuer cette génération de code, on ne tient pas compte du double-sens des dépendances référentielles.

L'ordre de création des relations se fait en respectant les références en avant des dépendances référentielles. Il peut exister un (ou plusieurs circuits) dans les dépendances référentielles. Dans ce cas, il faut que le concepteur 'coupe' ce circuit, et reporte la contrainte correspondante au niveau applicatif, Les possibilités de définie des règles ATL pour la détection des circuits dans le diagramme de classe avant démarrer le processus de transformation sera une

bonne solution pour éviter les problèmes des circuits de dépendances référentielles afin de réaliser des outils complets et performants conforme aux besoins des utilisateurs.

3.8 Conclusion

Dans ce chapitre nous avons proposé une approche automatique pour transformer les Diagrammes de classe vers un code SQL. La méthode proposée se base sur langage de transformation ATL et utilise l'outil de modélisation et de méta-modélisation EMF.

Afin de réaliser cette méthode, nous avons proposé un méta-modèle (ecore EMF) qui nous a permis de générer un outil visuel pour la réalisation (écriture) des Diagrammes de classe.

Enfin, nous avons donné un exemple illustratif pour montrer notre démarche, et générer le code LDD SQL et le passer à un SGBD pour créer la base de données équivalente.

Enfin, la documentation d'ATL et un dialogue avec les développeurs nous donnent à penser qu'il faudra articuler sur ce langage avec d'autres outils pour améliorer l'ergonomie du programme de transformation global.

Conclusion

Conclusion générale

La démarche MDA, bien qu'assez récente, suscite un réel intérêt chez bon nombre d'industriels et de développeurs. En effet, cette démarche est prometteuse et répond à des attentes légitimes non comblées par les technologies objet ou composant, elle autorise la séparation du logique métier de l'entreprise de son implémentation physique.

Pour que le MDA se diffuse auprès des développeurs, le savoir doit être essaimé de nouveaux outils doivent être développés. Pour produire ces outils qui font encore aujourd'hui défaut, plusieurs projets ont été lancés pour que l'approche MDA tienne ses promesses. Ces outils permettront l'automatisation des transformations ainsi que la génération automatique de code à partir de modèles.

Le travail présenté dans ce mémoire s'inscrit dans le domaine de l'ingénierie dirigée par les modèles. Il se base essentiellement sur l'utilisation combinée de méta – modélisation et de transformation de modèle. Plus précisément, la méta-modélisation et transformation des diagrammes de classe, à l'aide d'EMF et ATL.

L'EMF présente un Framework complet et extensible pour le développement MDA. L'objectif de notre travail est de suivre cette démarche pour résoudre certains problèmes rencontrés lorsqu'on utilise le langage de modélisation le plus populaire (UML).

Aujourd'hui, lors de la conception de base de données, il devient de plus en plus courant d'utiliser la modélisation UML plutôt que le traditionnel modèle entités-association.On peut donc transformer automatiquement des diagrammes de classe en BDD.

Nous cherchons dans un travail futur d'étendre notre approche pour couvrir tous les autres concepts utilisés dans les diagrammes de classe (héritage, composition, …etc) et de générer le code java convenable permettant d'accéder et de manipuler la base de données déjà générée.

Enfin nous souhaitons que ce travail puisse être une base pour d'éventuels travaux et puisse être amélioré et enrichi davantage.

Bibliographie

BIBLIOGRAPHIE

[1]- Benoit combemale , ingénierie dirigée par les modelés (idm) état de l'art, université de toulouse article publie le 12 aout 2008.

[2]- Marcus Alanen and Ivan Porres. Difference and union of models. In UML Conference, pages 217, San Francisco, California, Octobre 2003. Springer-Verlag LNCS 2863.

[3]- Wikipedia http://fr.wikipedia.org/wiki/Object_Management_Group.

[4]- Jamal abd-ali, méta modélisation et transformation automatique de psm dans une approche mda, mai 2006.

[5]- Hubert Kadima. Conception orienté objet guidée par les mod`eles. Dunod, 2005.

[6]- Mireille Blay-Fornarino Jean-Marie Favre, Jacky Estublier. L'ing´enierie dirig´ee par les mod`eles - Au-del`a du MDA. Herm`es - Lavoisier, 2006.

[7]- Jean Bezivin & Xavier blanc, mda : vers un important changement De paradigme en génie logiciel, Juillet 2002.

[8]- Jean Bézivin & Xavier Blanc, MDA : VERS UN IMPORTANT CHANGEMENT DE PARADIGME EN GENIE LOGICIEL, Université de Nantes.

[9]- Mohamed HADJ KACEM , Modélisation des applications distribuées à architecture dynamique : Conception et Validation , Novembre 2008.

[10]- NGUYEN Viet HOA, Capitalisation des architectures métiers pour une implémentation sur Différentes plates –formes techniques en utilisant la démarche MDA, Hanoï, Juin 2008.

[11]- Jacques barzic, model driven architecture (mda), conservatoire national des arts et métiers, fevrier 2007.

[12]- Benoit Combemale, Xavier Cr_egut, Marc Pantel , Transformation de Modèles Introduction _a ATL , IRISA CNRS Laboratory, University of Rennes 1, derniére mise _a jour le 24 octobre 2010

[13]- Wikipedia http://fr.wikipedia.org/wiki/Eclipse_%28projet%29

[14]- LAVIGNASSE Karen - LÉPINE Nathalie - MOLLIÈRE Hélène SROUR Youssef - SUDRE Raphaël, Génération de fichiers de paramétrage Projet opérationnel, Nantes, 24 mars 2007.

[15]- Jacques Barzic, Eclipse et ses plugins de modélisation (EMF – GEF – GMF), PDF créé le 12 janvier 2008.

[16]- XAVIER Blanc, MDA en Action, Groupe Eyrolles, 2005

[17]- BLANC X., 2005. MDA en action. Ingénierie logicielle guidée par les modèles. Eyrolles, Architecte logiciel, PARIS. 269 p,

[18]- http://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial/Part. 04.06.2013

[19]- Wikipedia http://fr.wikipedia.org/wiki/Topcased . 04.06.2013

[20]- http://atom3.cs.mcgill.ca/

[21]- Xin Jin, Applying Model Driven Architecture approach to Model Role Based Access Control System, Master of Science in System Science, University of Ottawa, Ontario, Canada ©2006 Xin Jin.

[22]- UML, diagrammes de classes, Introduction à la programmation orientée objets, chapitre 9

[23]- Méthodologie des systèmes d'information UML course dispenser par Annick Lassus. CNAM ANGOULEME 2000-2001

[24]- Laurent Audibert, UML 2 Edition 2007/2008,

[25]- http://laurent-audibert.developpez.com/cours-bd/.

[26]- Ass. MANYA F., Note du cours de SGBD, G2 INFO/U.KA. 2007 - 2008.

[27]- Guézelo, P. (2006). ModElisation des données : Approche pour la conception des bases des données (http://philippe.guezelou.fr/mcd.htm)..

[28]- Le bundle ATL pour Windows: http://www.sciences.univ-nantes.fr/lina/atl/atldemo/adt/ .

[29]- JeffRothenberg, the Nature of Modeling, 1989 .

[30]- OMG, Object Constraint Language OMG Available Specification, Version 2.0, formal/06-05-01

[31]- OMG MOF 2.0/XMI Mapping, Version 2.1.1, formal/2007-12-01

Annexe

Annexe

Le code complet de la transformation (module cdiagr2sql.atl)

```
-- ------------------------------------------------------------------------------
-- Crée un fichier SQL porte le nom de ClassModel pour chaque ClassModel.
-- ------------------------------------------------------------------------------
query cdiagr2sql =
class_metamodel!ClassModel.allInstances()->
                        select(a | a.oclIsTypeOf(class_metamodel!ClassModel))->
      collect(a|a.toSQLCode().writeTo('C:/Users/bureau/Desktop/workspace/workspace/Dc
lass2Rtable/Models/'+a.name+'.sql'));
-- ------------------------------------------------------------------------------
  -- Helper pour trouver le nom de ID attributes de classe courante:
-- ------------------------------------------------------------------------------
helper context class_metamodel!Class def: idclasse_name(): String= self.attributes>
asSet()->iterate(att; acc: String =''|
      acc+ if att.id then att.name else '' endif );
-- ------------------------------------------------------------------------------
-- Helper pour trouver le type de ID attributes de classe courante:
-- ------------------------------------------------------------------------------
helper context class_metamodel!Class def: idclasse_type() : String = self.attributes-
>asSet()->iterate(att; acc: String =''|
      acc+ if att.id then att.type.name else '' endif );
-- ------------------------------------------------------------------------------
 -- Helper pour crée les classes persistantes :
-- ------------------------------------------------------------------------------
     helper context class_metamodel!ClassModel def: toSQLCode() : String=
        '\n --\n -- Create schema ' + self.name+'\n --\n\n-- N I Y H\n\n'+' CREATE
DATABASE IF NOT EXISTS '+self.name+' ;'+'\n\n'+ 'USE ' + self.name+ ' ;'  +'\n\n' +
      self.classifiers-> asSet()-> iterate(className; acc : String = '' | acc+
      if acc='' then  if(className.oclIsKindOf(class_metamodel!Class)) then
      className.toSQLCode()  + '\n)ENGINE=InnoDB DEFAULT CHARSET=latin1;'    else ''
endif
    else if (acc<>'' and   className.oclIsKindOf(class_metamodel!Class))
      then '\n\n' + className.toSQLCode() +  '\n)ENGINE=InnoDB DEFAULT
CHARSET=latin1;'  else'' endif
      endif ) +
```

```
-- ------------------------------------------------------------------------------
-- génerer une nouvelle table pour chaque association à une cardinalité de type N_N
-- ------------------------------------------------------------------------------
      if self.associations->asSet().size()>0   then
      self.associations->asSet()->iterate(assoc; acc: String = '' |acc+
      if (assoc.card=#card_N_N) then
      if acc=''  then assoc.toSQLCode() else ' \n\n' + assoc.toSQLCode() endif
     else '' endif)
       else '' endif ;
-- ------------------------------------------------------------------------------
  -- Helper pour crée les attributs  des classes et les clés étrangères
-- ------------------------------------------------------------------------------
     helper context class_metamodel!Class def: toSQLCode() : String=
      if self.persistent then  '--\n-- Definition of table '+ '`'+self.name+'`\n--'+
'\n\n create table `' + self.name  +'` (\n'   else '' endif
        +  self.attributes-> collect(d | d.debug())->iterate(att; acc : String = '' |
acc+ if acc = '' then  ' '+att.toSQLCode() else  ',\n ' + att.toSQLCode() endif )
+ if self.has->asSet().size()>0 then
                  self.has.asSet()-> iterate(a; acc: String = '' | acc+
                        if(a.card=#card_1_1) then
                        if(self.name=a.source.name)then',\n     `'
+ a.target.idclasse_name()+'` '+a.target.idclasse_type()+
' NOT NULL'+',\n     CONSTRAINT `' +a.source.name+'_'+a.target.name
+ '` FOREIGN KEY (`'+ a.target.idclasse_name() +'`) REFERENCES `'+ a.target.name+'`' +
'(`' +a.target.idclasse_name() + '`)'

else',\n     `'+ a.source.idclasse_name()+'` '+a.source.idclasse_type()+ ' NOT NULL
'+',\n     CONSTRAINT `' +
a.target.name+'_'+a.source.name + '` FOREIGN KEY (`'+ a.source.idclasse_name() +'`)
REFERENCES `'+ a.source.name+'`' + '(`' +
a.source.idclasse_name() + '`)' endif
                        else if(a.card=#card_1_N) then
if(self.name=a.target.name) then ',\n     `'+ a.source.idclasse_name()+'`
'+a.source.idclasse_type()+ ' NOT NULL '+',\n     CONSTRAINT `' +
a.source.name+'_'+a.target.name  + '` FOREIGN KEY (`'+ a.source.idclasse_name() +'`)
REFERENCES `'+ a.source.name+'`' + '(`' + a.source.idclasse_name() + '`)'

else '' endif
else
```

```
if(a.card=#card_N_1) then if(self.name=a.source.name) then
                                        ',\n     `'+ a.target.idclasse_name()+``
'+a.target.idclasse_type()+ ' NOT NULL '+',\n     CONSTRAINT `' +
                                        a.source.name+'_'+a.target.name
                                    + `` FOREIGN KEY (`'+ a.target.idclasse_name() +'`)
REFERENCES `'+ a.target.name+'`' + '(`' +
                                              a.target.idclasse_name() + '`)'

                                    else '' endif
                                          else '' endif
                                          endif
                              endif)
                 else '' endif;
-- ------------------------------------------------------------------------------------
-- Helper pour écrire les attributs des  classes persistantes ainsi leur clé primaire
-- ------------------------------------------------------------------------------------
       helper context class_metamodel!Attribute def: toSQLCode() : String=

      if (self.id) then '`'+self.name+''  ' + self.type.name + ' NOT NULL PRIMARY KEY
' else ''+self.name+''+' ' +
             self.type.name + ' NOT NULL'   endif   ;


-- ------------------------------------------------------------------------------------
-- écrire une nouvelle table pour chaque association à une cardinalité de type N_N
-- ------------------------------------------------------------------------------------
       helper context class_metamodel!Association def: toSQLCode() : String=
       '\n\n' +'--\n-- Definition of table '+ '`' +self.name+'`\n--'+ '\n\n'+ '
create table ' + '`'+self.name+'`' +
       ' (\n' +     `'+ self.target.idclasse_name()+'` '+
self.target.idclasse_type()+ ' NOT NULL ,\n'+
                   '     `'+self.source.idclasse_name()+'` '+
self.source.idclasse_type()+ ' NOT NULL ,\n'+ '     PRIMARY KEY '+
                   '(`'+self.target.idclasse_name()+'`,
`'+self.source.idclasse_name()+'`),\n'+

                   '     CONSTRAINT `' +self.source.name+'_'+self.target.name
                                    + `` FOREIGN KEY (`'+ self.target.idclasse_name() +'`)
REFERENCES `'+ self.target.name+'`' + '(`' +
                                              self.target.idclasse_name() + '`)'
```

```
                +
                ',\n     CONSTRAINT `' +self.target.name+'_'+self.source.name
                        + '` FOREIGN KEY (`'+ self.source.idclasse_name() +'`)
REFERENCES `'+ self.source.name+'`' + '(`' +
                                        self.source.idclasse_name() + '`)'

      + '\n) ENGINE=InnoDB DEFAULT CHARSET=latin1;' ;
```

ملخص:

نأمل من خلال هذه المذكرة, في تقديم مقاربة MDA بغية معالجة وتحويل مخططات الفئات UML إلى قاعدة بيانات علائقية ,واليوم اصبح اكثر شيوعا استخدام النماذج فى تصميم قاعدة البيانات بدلا من النموذج التقليدى الكيانات - رابطة

تعتمد المقاربة المطروحة على المعيار EMF ونموذج تعريفي خاص بمخططات الفئات .

من اجل تحويل مخطط الفئات إلى شبكة قاعدة بيانات علائقية موافقة, تدعو الحاجة إلى استخدام لغة ATL , ولإتمام العملية لا بد من تحديد مجموعة قواعد للتحويل بمقدورها انجاز الإجراء بصفة آلية.

كلمات مفتاحية: IDM , MDA,UML , Méta-modèle, Diagrammes de classe, Base de données relationnelle, Modele de transformation, ATL, MOF

Résumé :

On cherche à présenter une approche MDA pour la manipulation et la transformation des diagrammes de classe vers les bases de données relationnelles.

Il est possible de traduire un diagramme de classe en modèle relationnel. Aujourd'hui, lors de la conception de base de données, il devient de plus en plus courant d'utiliser la modélisation UML plutôt que le traditionnel modèle entités-association.

Notre approche est basée sur le standard EMF et un méta-modèle pour les diagrammes de classe.

Pour transformer un diagramme de classe vers une base de données relationnelle équivalente on veut utiliser le langage ATL (Atlas Transformation Langage) pour cela il nous faudra définir un ensemble de « *helpers* » de transformation qui vont permettre de réaliser automatiquement ce processus.

Mots clés : IDM, MDA, UML, Diagrammes de classe, Base de données relationnelle, Modele de transformation, ATL, Méta-modèle, MOF.

Abstract :

We intend to introduce an MDA approach for handling and transforming UML class diagrams into relational Data base.

It is possible to translate a class diagram into relational model. Today, during the design of database, it is becoming more and more common to use the UML modeling rather than the traditional model entity-relationship.

Our approach is based on the EMF standard and a meta-model for UML class diagrams . To transform a class diagram into an equivalent relational data base, we need to use the ATL language (Atlas Transformation Language) for this purpose, we need a set of transformation helpers wich will allow us to automatically perform this process.

Keywords: IDM, MDA, UML, Class diagrams, Relational data base, Model transformation, ATL, Meta-model, MOF