

## **CHAPITRE 2**

# **OUTILS SUPPORTE POUR LA TRANSFORMATION DES MODELES**

### **2.1 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 (d'analyse vers conception, par exemple), 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 nous allons voir 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 eclipse EMF et GMF. Certaines tentatives de transformations des modèles sont réalisées, Les travaux connexes représentent un survol sur ces approches. Et mon travail basé sur les outils EMF et GMF donc dans ce chapitre, nous allons présenter en générale quelque outils et on passe à détaillé les outils de notre travail.

## 2.2 Quelques outils disponibles pour la transformation

Outils	Description	Transformation	
		Langage	Expression graphique (G) ou textuel (T)
<b>AndroMDA</b>	Outil basé sur les Template pour générer du code J2EE à partir d'UML/XML. <a href="http://galaxy.andromda.org/index.php?option=com_frontpage&amp;Itemid=48">http://galaxy.andromda.org/index.php?option=com_frontpage&amp;Itemid=48</a>	<b>ATL, MofScript</b>	<b>T</b>
<b>AToM3</b>	(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.). Un Outil écrit en Python pour méta-modélisation et méta-transformation soutenant la modélisation de systèmes complexes, les formalismes et les modèles sont décrits comme des graphes <a href="http://atom3.cs.mcgill.ca/">http://atom3.cs.mcgill.ca/</a>	<b>Multi formalism (python)</b>	<b>G</b>
<b>DSL Tools</b>	DSL Tools permet la construction d'éditeurs graphiques personnalisés et la génération de code source à partir d'une modélisation des concepts métiers exprimée dans un langage spécifique. <a href="http://msdn2.microsoft.com/enus/vstudio/aa718368.aspx">http://msdn2.microsoft.com/enus/vstudio/aa718368.aspx</a>	<b>Notation XML</b>	<b>T</b>
<b>Merlin</b>	Un outil de modélisation basé sur la transformation de modèle vers modèle et la génération de code. <a href="http://merlingenerator.sourceforge.net/merlin/index.php">http://merlingenerator.sourceforge.net/merlin/index.php</a>	<b>QVT, JET</b>	<b>T</b>
<b>AGG</b>	AGG (Attributed Graph Grammars) est un outil capable de représenter des graphes, des graphes	<b>Graphic editor, textual editor</b>	

	de type et des règles de transformation, et permet d'éditer les graphes visuellement <sup>3</sup> , il est capable d'appliquer des <b>transformations</b> et d'assurer la conformité d'un graphe par rapport à un Type graph donné. <a href="http://tfs.cs.tu-berlin.de/agg/">http://tfs.cs.tu-berlin.de/agg/</a>		<b>G</b>
<b>VIATRA2</b>	Le principal objectif du cadre VIATRA2 (Visual transformations de modèles automatisés) est de fournir un support à usage général pour le cycle de vie complet des transformations de modèles d'ingénierie, y compris la spécification, la conception, l'exécution, la validation et la maintenance des transformations à l'intérieur et entre les différents langages de modélisation et de domaines.	<b>ASM, GT</b>	<b>T</b>

**Tableau 2.1 : quelque outil de transformation. [17][19]**

## 2.3 Plate-forme d'accueil

### 2.3.1 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 (notamment environnement de développement intégré et Framework) 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 bibliothèques 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. Figurant parmi les grandes réussites de l'Open Source, Eclipse est devenu un standard du marché des logiciels de développement, intégré par de grands éditeurs logiciels et sociétés de services. Les logiciels commerciaux Lotus Notes, IBM Lotus Symphony ou WebSphere Studio Application Developer sont notamment basés sur Eclipse. [11]

### 2.3.2 Architecture d'Eclipse

Le méta-projet Eclipse est constitué et organisé en une galaxie de projets logiciels. Sa spécificité tient à son architecture totalement développée autour de la notion de plugin en conformité avec la norme OSGI : toutes les fonctionnalités de l'atelier logiciel doivent être développées en tant que plug-in bâti autour de l'Eclipse Platform. [11]

Eclipse propose un Framework de développement logiciel fournissant des briques logicielles pour développer ces outils. En fait Eclipse est à la fois considéré comme un EDI, un Framework ou une plateforme, selon que l'on considère le projet, ses composants, les EDI résultant de leur assemblage :

En effet le projet Eclipse propose également des 'packages' en téléchargement : il peut s'agir :

- ✓ d'applications 'prêtes à l'emploi' facilitant la diffusion d'Eclipse, en intégrant chacune un ensemble cohérent de plugins autour de l'Eclipse Platform pour répondre à différents besoins spécifiques.
- ✓ Il s'agit essentiellement d'IDE spécialisés, tels que Eclipse Classique, Eclipse IDE for Java EE Developers, Eclipse IDE for C/C++ Developers, Eclipse for Mobile Developers,
- ✓ mais également d'AGL comme Eclipse Modeling Tools.

De Framework : **Eclipse RCP** constitue ainsi la plateforme type pour tout environnement de développement Eclipse orienté client riche : constitué des 2 plugins fondamentaux org.eclipse.ui et org.eclipse.core.runtime, il constitue la base de tout IDE Eclipse 'RCP', mais peut aussi être utilisé à partir d'autres IDE.

Eclipse RCP n'est traité ni comme un sous-projet de Eclipse, ni comme un package, mais est présenté comme une plateforme

### 2.3.3 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.

Tous ces composants de base peuvent être réutilisés pour développer des clients lourds indépendants d'Eclipse grâce au 'package' Eclipse RCP (Rich Client Platform). [11]

### 2.3.4 Eclipse projet

Eclipse Project constitue le projet fondateur autour duquel s'agrègent les autres composants Eclipse ; il comprend en 2012 les sous-projets :

- Platform, qui définit les composants communs 'de base' de l'ensemble du modèle de développement Eclipse ;
- Le Plug-in development environment, ou 'PDE', ensemble de plugins Eclipse permettant de développer et tester d'autres plugins Eclipse, en conformité avec OSGi sur lequel repose la philosophie Eclipse. PDE permet ainsi de créer les plugins Eclipse comme des Bundle OSGi puis, par assemblage, des applications RCP complètes ;
- Les Java Development Tools, ou 'JDT' : cet ensemble regroupe les plugins couvrant les fonctionnalités usuelles et spécifiques aux environnements de développement tel que l'exécution de code avec débogage. [11]

## 2.4 Les concepts clés de la manipulation des modèles

Les interfaces de manipulation de modèles proposées par ces standards sont de deux types : les interfaces dites Taylored, c'est-à-dire taillées sur mesure pour un méta-modèle donné, et les interfaces dites réflexives, qui permettent l'accès au niveau du méta-modèle. Les interfaces Taylored sont parfaitement adaptées à la manipulation d'un type de modèle, c'est-à-dire à un ensemble de modèles instances d'un même méta-modèle. Par exemple, les interfaces Taylored pour UML2.0 offriront les opérations permettant d'obtenir les attributs d'une classe ou les connexions entre composants UML. On parlera alors d'interface Taylored pour le méta-modèle UML2.0. Les interfaces réflexives sont utilisables sur tous types de modèles, les opérations qu'elles proposent étant totalement indépendantes de la structure des modèles. Le point fort de ces interfaces est qu'elles permettent d'obtenir des informations sur le méta-modèle d'un modèle et ainsi de connaître dynamiquement la structure du modèle. Les sections suivantes présentent plus en détail ces deux sortes d'interfaces de manipulation des modèles. [8]

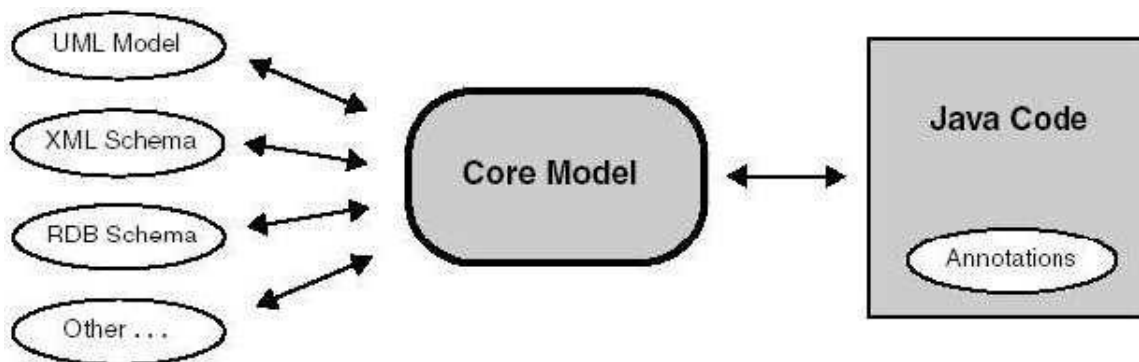
## 2.5 Manipuler des modèles avec EMF :

L'OMG a proposé un format de représentation permettant la manipulation des modèles dans les langages de programmation orientée objet. Ce format était défini dans le standard MOF. Le Framework EMF (Eclipse Modeling Framework) a été conçu pour la manipulation des modèles dans l'environnement ouvert Eclipse.

L'idée est de fournir un ensemble d'interfaces offrant les opérations nécessaires à la manipulation des modèles. Le développement d'une opération sur les modèles consiste simplement à développer une application utilisant ces interfaces de manipulation des modèles. [12]

### 2.5.1 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 2.1 : organisation générale d’EMF**

### 2.5.2 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. [18]

### 2.5.3 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. [12]

➤ ***UML***

Pour cette option, il existe trois possibilités.

✓ ***L'édition directe conformément au méta-modèle Ecore***

Il s'agit là, d'éditer des modèles graphiques UML conformément au méta-modèle Ecore. Cela sous-entend l'utilisation d'un outil de modélisation qui en soit capable. Par voie de conséquence, la transformation d'entrée du modèle n'est plus : on dispose du Core Model sous le bon format.

✓ ***L'importation de modèles UML***

Il s'agit d'importer, à l'aide de la fonction ad hoc d'EMF, un modèle dans son format natif (qui dépend de l'outil de modélisation utilisé). Seul le format IBM Rational (.mdl) permet de profiter de cet avantage. En effet, la gamme Rational et Eclipse sont des projets « frères » donc « génétiquement » compatibles.

✓ ***L'exportation de modèles UML***

C'est à peu près le même principe que l'option d'importation, sauf que la conversion du format natif en Ecore ne se fait pas avec EMF, mais avec l'outil de modélisation d'origine.

✓ ***Discussion à propos des trois options***

La première option ci-dessus présente l'avantage d'être simple et directe, de ne pas nécessiter d'opération d'importation ou d'exportation. Aucune synchronisation entre l'outil de modélisation natif et EMF n'est nécessaire. Les deux autres options (équivalentes du point de vue de la conversion) offre l'avantage que l'outil de modélisation peut servir à autre chose que la « simple » modélisation. Par exemple, il peut proposer sa propre fonction de génération de code. Il donne aussi la possibilité de créer son propre Core Model, de le transformer en Ecore pour l'utiliser ainsi par la suite.



➤ **XMI :**

Ce format de fichier, standard de l'OMG (Object Management 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.

Malgré quelques problèmes de maturité (en phase de résolution) qui demandent une attention particulière quant à l'association de différentes versions de UML avec les différentes versions de XMI, ce format tend à devenir le standard pour l'échange de données (et notamment des modèles) entre les différents outils du génie logiciel.

L'Illustration 1 ne fait pas directement référence à ce format, mais il est permis de penser que ce choix est pertinent pour les quelques raisons suivantes :

- comme nous l'avons vu ci-dessus, il tend à devenir un standard, du moins pour le développement orienté objet (aussi le credo de Java),
- il est le standard utilisé par Ecore pour sa propre persistance,
- tout cela concourt 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),
- cela permet une évolution « douce » vers la modélisation des plus irréductibles codeurs.

Les annotations sont des tags @model placés dans la javadoc des interfaces. Ces tags, et leurs attributs éventuels, sont détectés par EMF qui considère ainsi les entités concernées (interfaces, méthodes) comme des éléments de modélisation.

## 2.6 EMF and 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.

### 2.6.1 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. [8]

### 2.6.2 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 2.2) ont leur méta-modèle Ecore (ceux cités sont fournis en standard) 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. [13]

<b>Modèle(M1)</b>	<b>Méta-modèle(M2)</b>	<b>Méta-méta-modèles(M3)</b>
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 2.2 : 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.

<b>Modèle (M1)</b>	<b>Méta-modèle (M2)</b>	<b>Méta-méta-modèle (M3)</b>
MEcore (pivot)	MMEcore	Ecore

**Tableau 2.3 : l'espace technique standard d'EMF**

### 2.6.3 Les interfaces réflexives d'EMF

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

La particularité d'EMF est que toutes les interfaces réflexives qu'il propose sont spécifiées dans le méta-méta-modèle Ecore. [8]

➤ **EObject**

L'interface réflexive 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.

L'interface EObject 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 réflexive 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.

Cette interface offre aussi l'opération getEStructuralFeature (), qui permet d'obtenir une propriété (attribut ou référence) d'une EClass à partir de son nom.

➤ **EPackage**

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

➤ **EFactory**

L'interface réflexive 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.

#### **2.6.4 Les interface Taylored d'EMF :**

Les interfaces taylored sont parfaitement adaptées à la manipulation d'un unique type de modèle, c'est-à-dire à un ensemble de modèles instances d'un même métamodèle. Elles offrent des opérations spécifiques permettant une navigation dans les modèles instances d'un unique métamodèle.

Les interfaces tailored relatives au métamodèle représentant la structure des diagrammes de cas d'utilisation et permettent d'effectuer quelques opérations sur les cas d'utilisations. [8]

### 2.6.5 Règles de génération d'interfaces tailored

Les règles de génération d'interfaces tailored d'EMF sont similaires aux règles de génération JMI. Elles sont même un peu plus simples car elles ne souffrent pas du problème délicat de la traduction en Java des méta-associations, puisque ces dernières n'existent pas. Nous présentons ici une version simplifiée de ces règles. [8]

#### ➤ Règles EClass

Une EClass donne lieu à la création d'une seule interface (contrairement à JMI qui génère deux interfaces par métaclasse).

Cette interface présente les caractéristiques suivantes :

- A pour nom `nom_EClass`.
- Offre des opérations de lecture et d'écriture pour chaque EAttribute de l'EClass.
- Offre des opérations de lecture et d'écriture pour chaque EReference de l'EClass.
- Hérite de l'interface réflexive EObject.

#### ➤ Règles EPackage

Un EPackage donne lieu à la création de deux interfaces : une interface Factory, permettant la création de toute instance des EClass contenues dans l'EPackage, et une interface Package offrant les opérations de navigation entre toutes les EClass d'un package d'un méta-modèle.

L'interface Factory présente les caractéristiques suivantes :

- A pour nom `nom_packageFactory`.
- Offre une opération de création pour chaque EClass contenue dans l'EPackage
- Hérite de l'interface réflexive EFactory.

L'interface Package présente les caractéristiques suivantes :

- A pour nom `nom_packagePackage`.
- Offre une opération de navigation pour chaque EClass contenue dans l'EPackage permettant d'obtenir l'EClass correspondante.

- Hérite de l'interface réflexive EPackage.

## **2.7 Génération du code :**

Il est temps maintenant d'observer ce que va produire EMF : le code. C'est bien là son objectif premier. EMF répond bien à son objectif d'améliorer la productivité du développement d'application. Il y arrive en automatisant la génération du code à partir du modèle. Effectivement, une fois le modèle créé, « quelques clics » suffisent à cette génération.

Nous n'allons pas ici entrer dans une analyse détaillée du code contenu dans les éléments générés : nous nous contenterons, dans cette introduction à EMF, d'une liste (non exhaustive) de ce qui est généré. Une analyse détaillée demanderait, au préalable, une étude plus approfondie d'EMF et d'Ecore. [12]

### **2.7.1 Organisation du code généré :**

Un choix de conception a été fait par ses concepteurs et est imposé par EMF :

La séparation interface/implémentation dans le code généré.

Cela va se concrétiser par la génération de deux ensembles (issus du modèle d'entrée, assimilable à un diagramme de classes UML) :

1. un ensemble d'interfaces Java,
2. un ensemble de classes implémentant ces interfaces.

Les raisons principales qui justifient ce choix sont : la correspondance avec un pattern utilisé par de nombreuses API, la nécessité de pouvoir disposer d'un héritage multiple (impossible en Java sans la notion d'interface).

En plus des classes correspondant au modèle d'entrée, EMF génère deux autres éléments importants : une interface Factory et une interface Package ainsi que leurs classes d'implémentation.

#### **➤ La Factory**

Cette interface comprend une méthode create pour chacune des classes du modèle d'entrée.

Cela va permettre de créer des instances (des objets) des classes de l'application.

Le modèle de programmation EMF incite fortement à utiliser ces méthodes pour créer les objets lors de l'utilisation de l'application, en lieu et place de l'opérateur new.

➤ **Le Paquetage**

Cette classe apporte des facilités pour accéder aux méta-données Ecore du modèle. Il contient des accesseurs aux EClasses, EAttributes et EReferences9 implémentées dans le modèle, par exemple.

### **2.7.2 La re-génération et la fusion :**

Une des caractéristiques avantageuses de EMF est qu'il va permettre de compléter manuellement le code obtenu automatiquement, de pouvoir re-générer le code à partir du modèle modifié sans perdre les ajouts faits manuellement.

En effet, il est indispensable de pouvoir ajouter des méthodes et des attributs au code généré automatiquement. Celui-ci s'occupe uniquement (mais c'est déjà pas mal !) de générer le squelette des classes, les références aux autres classes (les associations) et les accesseurs aux attributs (les gets et les set).

Les éléments générés automatiquement seront repérés par le tag @generated dans la javadoc.

Lors d'une re-génération, suite à une modification du modèle, seuls ces éléments seront retouchés. En cas de conflit avec une modification manuelle, c'est cette dernière qui prime.

### **2.7.3 Le modèle générateur :**

En plus du modèle conforme au méta-méta-modèle Ecore, EMF utilise un modèle dit générateur (fichier d'extension .genmodel). Ce modèle, comme le modèle Ecore, est généré automatiquement (donc de manière transparente pour l'utilisateur) lors de la transformation du modèle d'entrée en modèle Ecore.

La plupart des informations nécessaires sont contenues dans le modèle « core » : le nom des classes les attributs, les références,...

Mais un certain nombre d'informations n'y sont pas telles que : les règles de préfixation du nom des classes, où mettre le code généré,...

Ces informations de paramétrage de la génération seront stockées dans le modèle générateur.

L'avantage de cette séparation (du générateur et du « core ») est que le méta-méta-modèle Ecore reste indépendant de toutes informations relevant de la stricte génération du code.

L'inconvénient est qu'il faut assurer une synchronisation des deux modèles en cas de modification (pour en garder la cohérence). EMF assure automatiquement cette synchronisation.

## **2.8 EMF et les standards OMG**

Des discussions ont cours à propos des relations qu'entretient EMF avec les différents standards de l'OMG que sont UML, XMI, MOF et MDA.

Ce paragraphe est de pure culture, en faire l'impasse ne nuira pas à la compréhension et l'expérimentation de EMF. [12]

### **2.8.1 Pour UML**

UML est un méta-modèle très utilisé pour modéliser les applications du monde objet. Les différents diagrammes conformes à UML sont prévus pour permettre autant de représentations d'une même application. Entre autres :

- la vue utilisateur : les cas d'utilisation,
- la vue dynamique : le diagramme de séquence,
- la vue architecturale : le diagramme de composants,
- la vue statique de conception : le diagramme de classes.

C'est sur cette dernière qu'intervient actuellement EMF : c'est le diagramme de classes qui sera utilisé pour générer le code (Java en l'occurrence).

### **2.8.2 Pour MOF**

MOF (Meta Object Facility) est le méta-méta-modèle standard de l'OMG. Il est utilisé pour définir les méta-modèles promus par cette organisation. Citons les plus caractéristiques de l'IDM : UML, QVT, CWM (Commun Warehouse Metamodel).

Ecore et MOF ont de nombreuses similitudes. Les différences se situent au niveau de la couverture des différents concepts tels que ceux de classes, de types de données, d'attributs, de relations entre paquetages et classes.



L'on peut aussi noter que le projet EMF, enfant du projet Eclipse, et son retour d'expérience ont une influence non négligeable sur les travaux de standardisation de MOF et/ou UML.

### **2.8.3 Pour XMI**

XMI est un standard créé par l'OMG basé sur XML. Ce dernier est lui-même un standard porté par le W3C (World Wide Web Consortium).

Ce standard a été créé pour faciliter la sérialisation et les échanges de données, dans le cadre de la modélisation, entre les différents outils qui interviennent dans le cycle de vie d'une application informatique.

Il s'appuie sur les mécanismes de DTD (Document Type Definition) ou de schéma XML pour définir les structurations de balises nécessaires et suffisantes à la représentation des modèles MOF au format XML.

XMI peut être utilisé pour sérialiser toutes sortes de modèles utilisés par EMF, il est aussi utilisé pour le méta-méta-modèle Ecore lui-même et comme forme canonique des fichiers « Ecore » (.ecore). [13]

### **2.8.4 Pour MDA**

Model Driven Architecture est une démarche d'ingénierie promue par l'OMG. Elle est basée sur la manipulation de différents modèles représentant l'application cible (indépendant de l'informatisation, indépendant de la plate-forme d'exécution, spécifique à cette plate-forme, de la plate-forme elle-même, le code) et, par voie de conséquence, sur des transformations de modèles. EMF est bien dans la philosophie de cette démarche et peut s'intégrer dans l'outillage nécessaire comme générateur de code à partir de modèles (ce qui est l'objectif premier d'EMF). [15]

## **2.9 Langage de transformation ATL**

### **2.9.1 Définition**

ATL (Atlas Transformation Language), a été proposé par le groupe ATLAS de L'Université de Nantes et par la compagnie TNI-Valiosis. Les outils de transformation Liés à

ATL sont intégrés sous forme du plugin ADT (ATL Development Tools) pour L'environnement de développement Eclipse. [15]

### **2.9.2 Description des méta-modèles**

Un modèle de transformation ATL se base sur des définitions de méta-modèles au Format XMI. Sachant qu'il existe plusieurs dialectes d'XMI, ADT est adapté pour interpréter des méta-modèles décrits à l'aide d'EMF (Eclipse) ou de MDR (NetBeans). Afin d'assurer l'indépendance d'ADT par rapport aux autres outils de modélisation, ADT met à disposition le langage KM3 (Kernel Meta-MetaModel), qui est une forme textuelle et simplifiée d'EMOF (voir figure 2), et permet de décrire des méta-modèles et des modèles.

### **2.9.3 Description des modèles à transformer**

Comme pour les méta-modèles, ADT supporte les modèles décrits dans différents dialectes d'XMI, qui peuvent par exemple être décrits au format KM3 et transformés au format XMI voulu.

### **2.9.4 Langage pour les règles de transformation**

Il s'agit du langage ATL, qui est défini par un modèle MOF (Meta Object Facility) pour sa syntaxe abstraite et possède également une syntaxe concrète présentée sous forme textuelle. Pour accéder aux éléments d'un modèle, ATL utilise des requêtes sous forme d'expressions OCL. Une requête peut retourner une valeur de type primitif ou complexe (élément d'un modèle, collection ou tuple). Elle permet de naviguer entre les éléments d'un modèle et d'appeler des opérations sur ceux-ci. La navigation ne peut se faire que sur des éléments initialisés : on ne pourra jamais naviguer sur des éléments du modèle cible.

Les règles appliquées sur reconnaissance sont exécutées. Pour chacune, on recherche dans le modèle source les éléments qui suivent son pattern source et le filtre correspondant ; la règle est alors reconnue pour un ensemble d'éléments.

La définition d'un modèle de transformation se fait dans un fichier portant l'extension « .atl ». L'entête du fichier se déclare de la manière suivante : [15]

Module <nom du module>

Create OUT : <méta-modèle cible> from IN : <méta-modèle source>

Pour décrire la transformation d'un élément du méta-modèle source en un élément du méta-modèle cible, on utilise une règle appliquée sur reconnaissance, qui se définit selon la syntaxe :

```
rule <nom de la règle> {  
  From  
  <Pattern source>  
  To  
  <Un ou plusieurs patterns cibles>  
}
```

Le pattern source est constitué de variables (avec éventuellement une valeur par défaut) et peut comporter des expressions OCL qui, portant sur ces variables, constituent le filtre. Voici la forme générale :

```
<Identifiant> : <type> = <valeur par défaut> (  
  <Filtre OCL portant sur les variables>  
)
```

Le pattern cible est constitué de déclarations de variables qui correspondent aux éléments cibles à générer, suivies d'attachements qui indiquent pour chaque élément cible, comment instancier ses propriétés :

```
<Nom élément cible> : <type> mapsTo <élément source associé> (  
  <Attachement>...)
```

Un attachement à la forme :

```
<Propriété de l'élément cible> <- <élément source>
```

Implicitement, la propriété de l'élément cible généré prendra comme valeur le résultat de la transformation de l'élément source selon la règle correspondant à ce dernier.

ATL offre également la possibilité de créer et de détruire des éléments intermédiaires selon les besoins de la transformation, ainsi que des constructions conditionnelles (if, switch), et itératives (while) avec une syntaxe proche de Java.

Les opérations (helpers) ATL peuvent éventuellement prendre des paramètres et ont un type de retour ; définies à l'aide d'expressions OCL, elles prennent la forme suivante :

```
Helper def : (<paramètre> : <type>) : <type retour> =  
<Expression OCL> ;
```

### 2.9.5 Les avantages 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, et aucun verrou technique ne laisse présager de difficulté à exprimer dans une transformation automatique. Comme il a déjà été évoqué, plus que sur le codage, le travail du programmeur portera sans doute sur un patient dialogue avec les concepteurs du processus de transformation pour précisément spécifier les méta-modèles sources et cibles de la transformation, qui devront être assez complets et précis pour permettre d'exprimer l'ensemble des concepts utilisés dans les deux mondes de modélisation. [16]

### 2.9.6 Les 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, 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, mais pour l'instant exclusivement pour le système d'exploitation Microsoft Windows. [16]

## **2.10 Conclusion**

Ce chapitre s'est exprimé la manipulation des modèles par le standard EMF comme un outil de modélisaion. EMF est un Framework open source de la plate-forme Eclipse, qui vise à permettre la manipulation dans Eclipse des modèles instances de méta-modèle Ecore.

Nous avons vu comment l'EMF permet de prendre en entrée plusieurs formats de modèles (UML, Java Annoté, Schéma XML, XMI).

Nous avons présenté ATL comme un langage pour la transformation des modèles et découvre leur règles de transformation.