Chapitre 6 Eclipse Modeling Framework

1. Introduction

EMF est une structure Java et une fonction de génération de code permettant de construire des outils et d'autres applications basées sur un modèle structuré.

Qu'entend-t-on par modèle ? Lorsque nous évoquons la modélisation, nous pensons généralement à des diagrammes de classes, des diagrammes de collaboration, des diagrammes d'état, etc. Le langage UML (Unified Modeling Language) définit une (la) notation standard pour ces types de diagrammes. En utilisant une combinaison de diagrammes UML, il est possible de spécifier un modèle complet d'une application. Ce modèle peut être utilisé uniquement à titre de documentation ou, si employé avec les outils appropriés, il peut être utilisé comme l'entrée permettant de générer une partie ou, dans des cas simples, la totalité d'une application.

Etant donné que ce type de modélisation nécessite généralement des outils d'analyse et de conception orienté objet OOA/D (Object Oriented Analysis and Design) coûteux, vous devez être réservé quant à notre assertion précédente relative au coût d'entrée faible d'EMF. Nous pouvons déjà dire qu'un modèle EMF requiert seulement un petit sous-ensemble des choses que vous pouvez modéliser dans UML, des définitions particulièrement simples des classes et de leurs attributs et relations, pour lesquelles il n'est pas nécessaire d'employer un outil de modélisation graphique exhaustif.

2. Eclipse Modeling Framework

2. 1 Définition EMF

• EMF est un « framework » qui traite des modèles : cela peut s'entendre ici sous le sens que 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 de traiter différents types de fichiers : conformes à des standards reconnus (XML, XMI) et aussi sous des formes spécifiques (code Java) ou tout simplement sur mesure (au bon gré du concepteur),

• EMF ne propose pas d'outil graphique (de dessin) pour la modélisation

2.2 Objectif de EMF

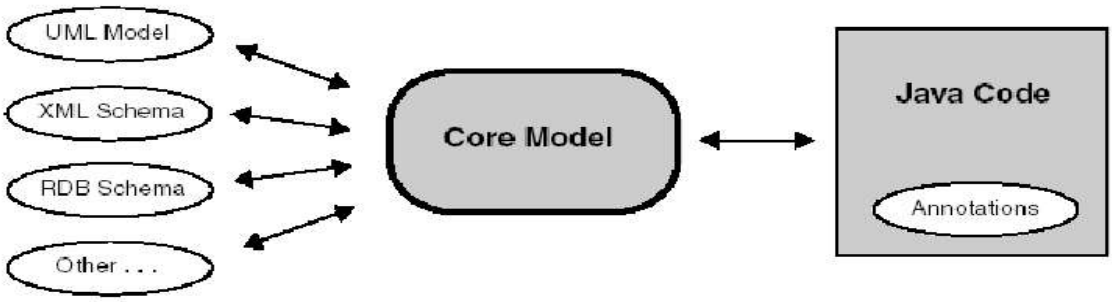


Illustration 1 - Organisation générale de EMF ([BUDINSKY et al. 2004])

L'objectif général de 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 (à gauche sur l'Illustration 1), présentés sous diverse formes, en Core Model,
2. la gestion de la persistance du Core Model,
3. la transformation du Core Model en code Java.

Les doubles flèches symbolisent que les transformations inverses (ou les « imports/exports ») sont aussi gérées par EMF.

Certains considèrent que EMF est à la fois :

• Un outil de ré-unification de divers standards de modèles (XMI, UML, code Java - si du moins on considère ce dernier comme un modèle),

• Un pont entre deux mondes du génie logiciel (celui des gourous de la modélisation et celui des partisans du code avant tout) en prenant une position médiane et en prenant le meilleur de chaque monde.

2. 3 Architecture et particularités

EMF est avant tout un framework open source complètement ouvert. Il représente un ensemble de classes pouvant être utilisées dans tous types d'applications nécessitant de

prendre en compte la notion de modèle. Ces classes mettent en oeuvre un ensemble de mécanismes prédéfinis, et transparents à l'utilisateur, pour le contrôle des modifications (i.e. EMF utilise une communication par événements) ou la sérialisation des modèles aux format XML. Les classes du framework EMF sont intéressantes car elles permettent de générer automatiquement d'autres classes qui fournissent à leur tour :

1. Une implémentation par défaut conforme aux spécifications fournies par du code Java annoté. Par exemple, une interface I marquée par la balise "@model" va être traduite en une classe I_implementation qui hérite et surcharge les méthodes de EObject fournie par EMF. Cette implémentation définit ensuite des utilitaires pour faciliter l'instanciation des éléments générés avec ici une façade I_Factory.
2. Des classes pour l'édition, i.e. l'utilisation et la manipulation au sein d'un programme, de l'implémentation précédente. Ces classes se caractérisent des éléments précédents par le fait qu'elles mettent en oeuvre les mécanismes utilisables pour la synchronisation et la persistance des modèles.
3. Un éditeur par défaut qui exploite le code précédent et qui se présente sous la forme d'un nouveau plugin pouvant être intégré dynamiquement à l'environnement Eclipse.

L'éditeur par défaut composé principalement d'une vue "arborescente" et d'une vue "propriétés". Il est représenté au niveau du code par une classe Java pouvant être adaptée au besoin de l'utilisateur. Par exemple, cette classe peut profiter du plugin GEF pour ajouter des vues graphiques ou JET pour définir des générations automatiques. Il faut noter que l'exécution de cet éditeur, et son intégration à Eclipse va avoir pour effet, de créer un fichier XML par défaut utilisé à la fois comme support à la persistance et comme mémoire de travail (i.e. toute action sur l'éditeur va modifier les données contenues dans ce fichier).

Tout comme Eclipse, EMF est implémenté en, et utilise, le langage Java. En effet, utiliser EMF revient à utiliser une API composées de classes spécifiques qui utilisent elles-mêmes du code Java complété par des annotations; une annotation se présente sous la forme d'un commentaire Java qui précède une déclaration (de package, de classe, de méthode, d'attribut, etc.) et qui fait appel à des balises particulières ("@model" en particulier).

EMF peut être décrit de manière plus détaillée sous la forme d'une architecture à trois niveaux représentant chacun un aspect ou une fonctionnalité. Ces trois niveaux correspondent chacun aux :

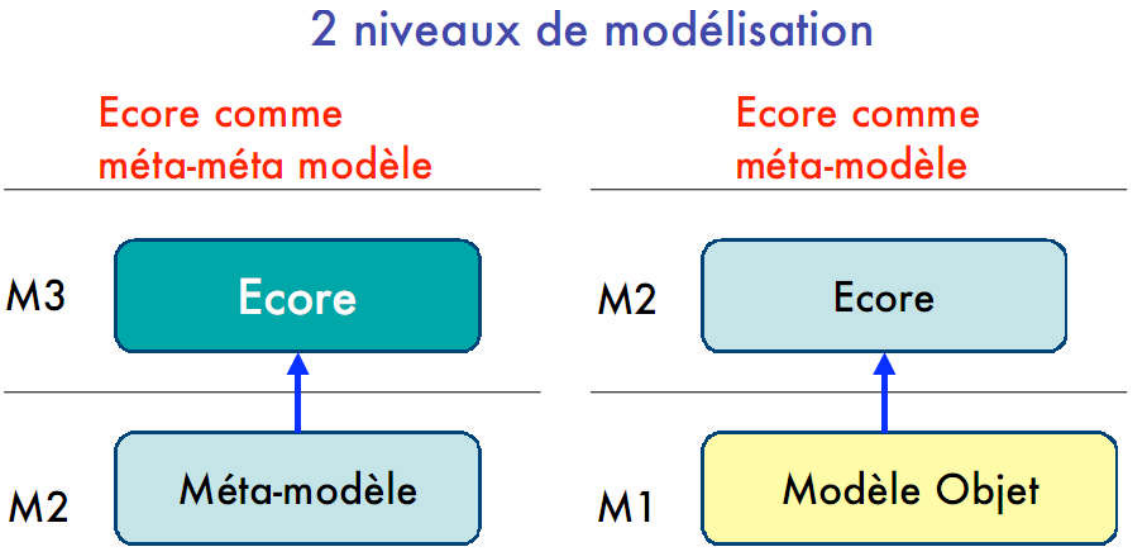
1. Modèles (package EMF). Ce niveau correspond aux données fournies en entrée d'EMF qui peuvent provenir de différentes sources : XMI, Java annoté ou diagrammes . Les différentes

balises utilisées dans le fichier XMI ou comme annotations correspondent aux éléments du méta modèle ECore (i.e. Eclipse Core) sur lequel repose Eclipse-EMF.

2. Editeurs (package EMF.Edit). Ce niveau propose une API pour réaliser des éditeurs permettant eux d'afficher des modèles suivant différentes vues (simple texte, vue arborescente, vue sous forme de tables, etc.) Ou d'éditer de propriétés. Un éditeur par défaut pouvant être facilement adapté est proposé.

3. Générateurs (package EMF.Codegen). Ce dernier niveau permet d'étendre le niveau précédent en offrant les moyens de générer du code. Celui-ci n'est pas fondamental dans la mesure où le niveau précédent intègre déjà un mécanisme de persistance au format XML et qu'il est possible d'utiliser d'autres plugins (XML Schema, en particulier) pour générer n'importe quel code à partir de ce fichier. Le plugin JET représente une alternative plus conviviale pour réaliser des générations de code et est présenté dans la suite de cette partie.

2.4 Modèle EMF



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

2.5 Les formats d'entrée standard

2 .5 .1 U ML

Pour cette option, il existe trois possibilités.

a. L'édition directe conformément au méta-modèle Ecore

Il s'agit là, d'éditer des modèles graphiques UML conformément au méta-modèl *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.

b. L'importation de modèles UML

Il s'agit d'importer, à laide de la fonction ad hoc de 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.

c. 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.

2.5 .2. X MI

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.

Malgré quelques problèmes de maturité (en phase de résolution) qui demandent une attention particulière quand 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'*Illustation 1* ne fait pas directement référence à ce format, mais il est permit de penser que ce choix est pertinent pour les quelques raisons suivantes :

• 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.

2.5 .3 J ava 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 Le (méta ?)méta-modèle pivot : Ecore

le *Core Model* pivot des transformations possibles avec EMF, doit pouvoir modéliser les correspondances entre plusieurs types de modèles. On dirait, pour être puriste, que ces différents modèles sont conformes à autant de méta-modèles. Le tableau ci-dessous synthétise les cas en présence pour EMF.

Tableau II - Les espaces techniques de modélisation à fédérer.

Modèles (M1)[5]	Méta-modèles (M2)	Méta-méta-modèles (M3)
Diagramme de classes (entrée)	UML	MOF
Fichier XML (entrée)	XMI	MOF
Ensemble d'interfaces Java (entrée)	Java Annoté	EBNF[6]
Programme Java (sortie)	Java	EBNF

Pour être fidèle à l'IDM, les transformations sont définies au niveau des méta modèles. Cette stratégie permet de modéliser les règles de transformation (il faut là aussi un méta-modèle), et ainsi de capitaliser les efforts faits pour leur définition.

Le tableau ci-dessous représente l'espace technique standard de EMF.

Tableau III - L'espace technique de modélisation fédérateur.

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

Il est hors de question de faire ici un exposé détaillé sur Ecore, mais voici quelques éléments de compréhension.

Ecore est **un méta-méta-modèle** : très proche de MOF. Il est en fait un sous ensemble de MOF : il restreint celui-ci.

En effet, une des particularités de 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.

En l'occurrence, la seule entrée UML possible est le diagramme de classe, mais on peut imaginer de méta-modéliser d'autres diagrammes (cas d'utilisation, séquence,...) : cela permettrait d'utiliser EMF à d'autres stades du cycle de développement (transformation CIM vers PIM de MDA)

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 II*) 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.

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.

Comme l'illustre les exemples pratiques proposés, EMF répond bien a 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 demanderais, au préalable, une étude plus approfondie de EMF et de Ecore.

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 Packetage

Cette classe apporte des facilités pour accéder aux méta-données Ecore du modèle. Il contient des accesseurs aux *EClasses*, EA*ttributes* et *EReferences :* 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 pourvoir 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 *get* 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 L e 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. Autres services proposés par EMF

En plus d'être un outil d'amélioration de la productivité, EMF a d'autres apports, tels que : la notification de modification du modèle, la gestion de la persistance par une sérialisation XMI, une API réflective pour la manipulation générique des objets EMF, l'EMF dynamique. La combinaison de tous ces services (et d'autres non cités) fait qu'EMF offre un socle pour l'interopérabilité avec d'autres outils et applications basés sur EMF.
Nous ferons ici une simple évocation de ces possibilités.

2.8.1 L a notification et les Adaptateurs

Lors de l'observation du code des classes d'implémentation générées par EMF, on trouve dans ses accesseurs de type *set* exemple de code ci-dessous:

```
public void setNom(String newNom) {
String oldNom = nom;
nom = newNom;
if (eNotificationRequired())
eNotify(new ENotificationImpl(this, Notification.SET,
BibliothequePackage.AUTEUR__NOM, oldNom, nom));
}
```

Le test conditionnel permet de vérifier si une notification de modification a été demandée sur l'élément (ici un attribut) et, le cas échéant, de l'opérer.

Cette possibilité de EMF est il intéressante : l'on va pouvoir transmettre automatiquement aux objets dépendants une modification faite sur un attribut ou une référence.

Dans EMF les observateurs de notification (les *listener* Java) son appelés *Adaptaters* (Adaptateurs). En effet, en plus de leur statut d'observateur, ils peuvent modifier les comportements des objets auxquels ils sont attachés.

2.8.2 L a gestion de la persistance

EMF fourni un mécanisme simple, et néanmoins puissant, pour la gestion de la persistance des objets.

Comme nous l'avons vu, le standard générique utilisé par EMF pour stocker les modèles est XMI. EMF propose aussi un mécanisme pour utiliser cette possibilité, déjà implémentée, pour sérialiser les objets des applications. Cela dégage l'utilisateur de la mise en place d'une sérialisation propre.

Le principe général consiste à créer des entités *Ressources* (classe *RessourceSetImpl*) et d'y placer les objets que l'on veut sérialiser. Ils sont ainsi à disposition par la suite.

2.8.3 L'API réflective

Comme tous les éléments des modèles EMF héritent de la classe *EObject* du méta métamodèle Ecore, l'utilisateur peut se servir de l'API réflective pour manipuler leurs instances. Il pourra, par exemple, accéder directement à la valeur d'un attribut d'objet sans faire appel aux accesseurs (*get*, *set*) de sa classe. Des méthodes d'accès plus génériques existent. Bien que cette technique d'accès est moins performante que l'accès direct par les

accesseurs des classes, elle permet une ouverture vers l'extérieur des modèles grâce à la généricité des méthodes de l'API réflective.

2.8.4 L'EMF dynamique

Jusqu'à présent nous avons utilisé EMF pour générer une implémentation de nos modèles.
Dans de nombreux cas, nous ne désirerons pas implémenter le modèle Ecore de notre application. Il suffit pour une observation de l'architecture.
La particularité de l'API réflective est qu'elle peut générer dynamiquement les classes nécessaires (et uniquement celles là) lors de l'utilisation d'une de ses méthodes.
En fait, le modèle Ecore est le seul nécessaire pour pouvoir utiliser l'application : seul un allongement du temps d'accès sera perçu.

2.9. 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.

2.9.1. 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).

2.9.2. 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
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.

2.9.3. 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 toute 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*).

2.9. 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 de EMF).

Conclusion

À travers cette étude introductive de EMF, ainsi que par l'exemple décrit dans le paragraphe précédent, nous avons pu mieux cerner les buts et le fonctionnement du plugin. Ce que l'on peut retenir :

1. EMF intervient au moment de la génération du code à partir d'un modèle : son domaine est le langage Java (donc aussi le monde objet) en standard,

2. EMF permet de prendre en entrée plusieurs formats de modèles (UML, Java Annoté, Schéma XML, XMI) : il peut donc être compatibles avec d'autres outils utilisés en amont dans le cycle de vie du logiciel,

3. une fois que l'on dispose du modèle d'entrée, les transformations se passent très facilement : quelques clics suffisent,

4. le code généré sera aussi précis et complet que le sera le modèle d'entrée : EMF pourra donc s'adapter à l'évolution des recherches (et donc des outils) qui fabriquerons les modèles du futur,

5. la génération du code à partir du modèle n'empêche pas son raffinage manuel : une re-génération après une modification du modèle n'écrase pas les compléments.