ECOLE NATIONALE SUPERIEURE DES INGENIEURS EN ETUDES ET TECHNIQUES D'ARMEMENT

RAPPORT DE PROJET DE FIN D'ETUDES

Présenté par : **Sara SELLOS**

Export de projet depuis un prototype sous Smalltalk réalisé à l'Université de Savoie vers un environnement de meta-modélisation sous Eclipse de type Kermeta

Effectué à l'Université de Savoie

sous la responsabilité de : Monsieur **Stéphane DUCASSE**

20 Août 2007

Remerciements

Ce travail a été réalisé de Avril à Août 2007 au Laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance de l'Ecole Polytech'Savoie d'Annecy.

Je voudrais tout d'abord exprimer ma plus sincère gratitude à Monsieur Stéphane DUCASSE qui a bien voulu assurer la responsabilité de mon stage.

J'adresse également mes remerciements aux chercheurs de la communauté Moose, en particulier à Monsieur Tudor GIRBA et Monsieur Adrian KUHN pour leur aide et leurs conseils qui m'ont guidée tout au long de ce stage.

Je tiens enfin à exprimer ma reconnaissance à l'ensemble des membres du LISTIC qui, pendant cinq mois, ont contribué à mon insertion au sein du laboratoire.

Table des matières

Introduction

Eclipse IDE est un environnement de développement qui s'est imposé, ces dernières années, comme un outil de référence incontournable pour le développement en JAVA. Une des ses spécificités vient du fait que son architecture donne la possibilité d'intégrer facilement de nombreux plug-in autour d'un noyau de fonctionnalités de base. Cela rend l'outil à la fois extensible et polyvalent.

Aujourd'hui, la modélisation est devenue une étape indispensable dans le processus de développement des logiciels. C'est pourquoi divers plug-in de modélisation sont disponibles pour Eclipse, notamment en logiciel libre.

Eclipse Modeling Framework (EMF) est l'un d'entre eux. Il permet de concevoir des modèles de type UML puis de générer du code en JAVA disponible pour un socle d'exécution autorisant la visualisation et l'expérimentation du modèle.

Par ailleurs, l'informatique est une part de plus en plus importante dans l'activité des entreprises. En effet, la logique de leur fonctionnement est contenue dans les logiciels qu'elles utilisent. Dans un monde en perpétuel changement, avec des logiciels de plus en plus complexes, la maintenance de ceux-ci est devenue une problématique cruciale.

Aujourd'hui, les outils de réingénierie apparaissent indispensables. C'est pourquoi les chercheurs de l'université de Bern développent depuis dix ans un puissant environnement de réingénierie appelé **Moose**. Cette plateforme propose des outils d'analyse de logiciels très performants avec, par exemple, l'analyse statique et dynamique des logiciels ou des moyens visuelles pour la compréhension des applications.

Jusqu'à présent, seuls les codes sources implémentés dans différents langages objets (Smalltalk, Python, Java, C++) étaient concernés par les outils d'analyse de Moose. En effet, l'environnement n'était pas capable d'analyser directement les modèles sous-jacents. Or, la phase de modélisation est une étape fondamentale dans le développement d'un logiciel. Un programme, conçu à partir d'un modèle bancal, a en effet toutes les chances, à terme, de provoquer des disfonctionnements. Il est donc intéressant, pour les analystes et développeurs, de disposer d'un outil qui optimiserait leur travail dans ce domaine.

De plus, un modèle ayant fait ses preuves doit pouvoir être retranscrit de façon fiable d'un système dans un autre dans un processus de réingénierie. On est alors dans un contexte der re-modélisation. Il s'agit alors de savoir reconstruire un applicatif dans une opération de maintenance approfondie sans perdre le capital acquis par les modélisations précédentes.

C'est dans ce contexte que s'inscrit mon projet de fin d'étude. L'objectif de ce projet est d'implémenter une passerelle qui lierait le monde de la modélisation, EMF, et le monde de la réingénierie, Moose.

Chapitre 1 Le LISTIC

1. Le LISTIC

Le LISTIC est un laboratoire d'informatique situé à l'université de Savoie (Annecy). Le projet scientifique du LISTIC est centré sur les systèmes de fusion d'informations. Les travaux qui y sont menés visent à la mise au point d'outils méthodologiques pour la spécification, la conception, la réalisation et l'exploitation de systèmes de fusion d'informations.

En raison de l'évolution technologique, les sources d'informations sont en effet de plus en plus nombreuses, réparties, plus ou moins fiables et d'origines diverses (capteurs, experts, ..). Ceci rend nécessaire la définition de systèmes de fusion d'informations, essentiellement logiciels, coopératifs, évolutifs et sûrs.

Les travaux du LISTIC se concentrent essentiellement sur le développement d'outils méthodologiques et logiciels pour des systèmes s'intégrant dans un processus de maîtrise de la chaîne de traitement de l'information, depuis son élaboration à partir de données issues de capteurs, sa capitalisation sous forme de connaissances et son exploitation dans quelques champs d'expérimentation privilégiés.

Le LISTIC est composé de trois équipes :

- **L'équipe TI**- Traitement de l'Information: elle contribue à la définition des composantes des systèmes de fusion d'informations (outils de représentation, combinaison, évaluation, explication, …) et à la méthodologie d'application, notamment pour le traitement d'images complexes et l'évaluation de performance industrielle (axes méthode de fusion et méthodologie).

- **L'équipe IC**- Ingénierie de la Connaissance, également appelée **équipe Condillac**. L'équipe Condillac est dédiée à la gestion et la capitalisation des connaissances et des savoir-faire. Ses travaux portent sur la modélisation des connaissances et des systèmes, de leurs fondements à leurs applications, en privilégiant une approche pluridisciplinaire: épistémologie, linguistique, intelligence artificielle et logique.

- **L'équipe LS**- Logiciels et Systèmes : l'équipe LS est chargée de la conception, de la réalisation et du déploiement des systèmes de fusion (et plus particulièrement, les aspects liés à la décentralisation). C'est au sein de cette équipe que j'ai réalisé mon stage de fin d'étude.

2. L'équipe LS

L'équipe LS intègre d'une part les axes systèmes et répartition et d'autre part, les axes logiciels et systèmes évolutifs.

L'équipe LS comporte ainsi deux axes principaux de recherches:

- L’axe **LES** : langages et évolution du logiciel (``Languages and Software Evolution") dont le responsable est l’enseignant-chercheur Stéphane Ducasse.
- L’axe **SR**: Systèmes et Répartition dont le responsable est Patrice Moreaux

Chapitre 2 Présentation du contexte technique

1. Le langage Smalltalk

1.1 Smalltalk, un langage objet

1.1.1 Objets et classes

Comme tous les langages à objets, en Smalltalk chaque objet a une identité, un état et un comportement. Le langage Smalltalk utilise le concept de **classe** pour regrouper les objets ayant même forme d'état et même comportement**.** On retrouve aussi la notion de constructeurs. Mais en Smalltalk, contrairement à JAVA et C++, les constructeurs sont des **méthodes de classe** ayant un nom défini par le développeur.

En Smalltalk, « tout est objet » : les chaînes de caractères, les entiers, les booléens, les définitions de classes, les blocs de code, les piles et la mémoire. Smalltalk, contrairement à C++ et Java, n'est pas un langage typé, ce qui rend le code plus concis.

Une des originalités importantes, par rapport à JAVA et C++, est que les blocs de contrôle (if, while, etc) n'appartiennent pas au langage mais sont développés en tant que méthode au moyen de blocs de code. Par conséquent, ils sont eux même des objets auxquels on s'adresse par l'envoi de messages.

1.1.2 Variables

En JAVA et C++, on distingue la notion de variables d'instance (variables propres à une instance) de la notion de variables de classes (variables partagées par les instances d'une classe).

En Smalltalk, le schéma est un peu plus complexe : on retrouve la notion de variables d'instance mais la notion de variable de classe prend un sens différent.

En Smalltalk, il faut distinguer la notion de variable d'instance de classe et la notion de variables partagées. Le terme de variables d'instances de classe désigne les variables d'instances définies par une méta-classe qui décrit des classes. En revanche, comme leur nom l'indique, les variables partagées (shared variables) sont communes à toutes les instances d'une classe.

1.1.3 Instanciation

En Smalltalk, la création d'objets s'effectue par l'envoi d'un message conventionnel ***new*** ou ***basicNew*** dont les méthodes sont implémentées dans la méta-classe **Behavior.** ***new*** peut être redéfinie en tant que méthode de classe dans d'autres classes tandis que ***basicNew***, comme son nom le laisse sous entendre, ne peut être réécrite**.** Ainsi, en Smalltalk contrairement à Java, ***new*** n'est pas un opérateur mais une méthode que l'on peut surcharger.
Par convention, la définition de la méthode ***new*** se fait en appelant une méthode d'instance ***`initialize`*** qui sert de constructeur

1.1.4 Visibilités des variables d'instance

En JAVA, les variables d'instance peuvent être visibles de l'extérieur d'un objet mais à condition qu'ils soient déclarés comme étant **public**. Par ailleurs, elles sont visibles uniquement par leurs sous-classes et les classes du package si elles sont définies **protected**. En outre, elles ne sont pas visibles à l'extérieur de la classe s'ils sont définies **private**. Enfin, si la visibilité n'est pas précisée dans le code, elles sont visibles du package.

En Smalltalk les variables d'instance d'une classe ne sont visibles que par ses sous-classes. Il faut donc à chaque fois écrire des méthodes pour y accéder : accesseurs en lecture et écriture.

1.1.5 Héritage

Le fonctionnement de l'héritage en Smalltalk est semblable à celui en JAVA : tous deux n'autorisent que l'**héritage simple**. Les variables d'instance et les méthodes sont systématiquement héritées des superclasses et sont redéfinissables. Par contre, une variable d'instance déclarée dans une superclasse ne peut être re-déclarée dans ses sous-classes.

L'héritage induit le **polymorphisme**. C'est un concept dans la théorie des types selon lequel un nom (par exemple une déclaration de variable) peut désigner des objets de nombreuses classes différentes qui sont reliées par une certaine superclasse commune ; par conséquent, tout objet désigné par ce nom est capable de répondre de différents façons à un certain ensemble commun d'opérations. Cette instance peut donc utiliser sans les définir les méthodes de la superclasse. Comme en JAVA, l'unique source du graphe d'héritage est la classe `Object` et toutes les classes héritent de cette classe.

1.1.6 Self et super méthode

En Smalltalk, **self** (équivalent de this en Java) représente le receveur du message. La recherche d'un message envoyé à **self** démarre dans la classe du receveur.

Lorsque l'on définit une méthode dans une sous-classe, elle peut cacher une méthode dans les superclasses. Pour avoir accès à de telles méthodes cachées d'une superclasse, les messages doivent être envoyés à **super** et pas à **self**. **super** représente aussi le receveur du message mais la recherche des messages s'effectue à partir de la superclasse de la classe de la méthode qui déclenche l'invocation du **super.**

1.1.7 Smalltalk, un langage interprété

En Smalltalk, tout est modifiable. On peut changer l'IDE en cours d'utilisation, sans recompiler et redémarrer l'application, ajouter une nouvelle instruction de contrôle dans le langage, changer la syntaxe du langage, ou la façon dont le garbage collector fonctionne. Par ailleurs, le garbage collector est intégré et transparent pour le développeur.
Les programmes Smalltalk sont généralement compilés en bytecode, exécutés par une machine virtuelle.

1.2 Les notions de Classe et de méta-classe

1.2.1 Le concept de Méta-classe

En Smalltalk, les classes sont définies comme instances d'autres classes : les méta-classes. Une méta-classe définit des propriétés de classe par des protocoles permettant de décrire et de paramétrer le comportement des classes. Ces protocoles donnent une définition quasi-réflexive des classes et des méta-classes primitives du système. Ils permettent de plus une reconfiguration partielle du système. Ceci est possible par la définition de nouvelles méta-classes ayant des comportements différents de ceux prédéfinis, par exemple, au niveau de l'héritage ou de la création et de l'initialisation des instances. Une classe peut par exemple être abstraite ou ne posséder qu'une seule instance. Elle peut également tracer des messages reçus par ses instances ou interdire la redéfinition de certaines méthodes d'instance.

En Smalltalk, chaque classe est instance d'une seule et unique méta-classe. Celle-ci lui est automatiquement associée lors de sa création.
Une classe contient ses variables et ses méthodes d'instance. En revanche, sa méta-classe contiendra la structure (variables de classes, variables d'instance de la méta-classe) et le protocole de classe (c'est-à-dire les méthodes de classes, les méthodes d'instanciation …).

Dans l'exemple suivant, la classe **Insecte** contient un certain nombre d'attributs (*abs, ord, toursFaim, toursRepas*) et de méthodes d'instance (*affichage, initialise,* les accesseurs …)
Par ailleurs, est définie une variable de classe pour la classe **Insecte** : la variable COULEUR. La méthode d'instanciation *new* est considérée en Smalltalk comme une méthode de classe. Voici donc comment sont réparties les méthodes et les variables entre la classe **Insecte** et sa méta-classe **Insecte_class.**

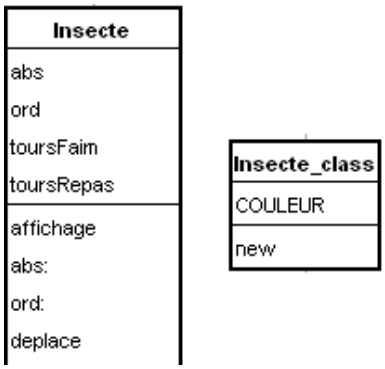


Fig 2-1 Une classe avec sa méta-classe

1.2.2 Niveaux d'abstraction

En modélisation, différents niveaux d'abstraction sont mis en jeu. Chaque niveau décrit et contrôle le niveau inférieur auquel il est connecté. Les instances terminales sont contrôlées par le premier niveau regroupant les classes. Les classes sont elles-mêmes contrôlées par des méta-classes qui sont à leur tour contrôlées par des méta méta-classes. Souvent, une classe et sa méta-classe associée sont amenées à communiquer. Cependant, des problèmes d'incompatibilité peuvent survenir du fait de ces communications. On trouve deux types de comptabilité **: la comptabilité ascendante** et **la comptabilité descendante.**

1.2.3 Comptabilité ascendante

Soit la classe A(n) qui définit la méthode **foo**. Cette méthode d'instance envoie le message **bar** à la classe. La classe B(n), sous classe de A(n), hérite donc de la méthode **foo**.
Quand **foo** est envoyée à une instance de B(n), la classe B(n) reçoit le message **bar.** La recherche du message **bar** commence à partir de B(n+1). Comment alors garantir une réponse adéquate de B(n) ? Il s'agit là d'un problème de comptabilité de méta-classes appelé problème de comptabilité ascendante.

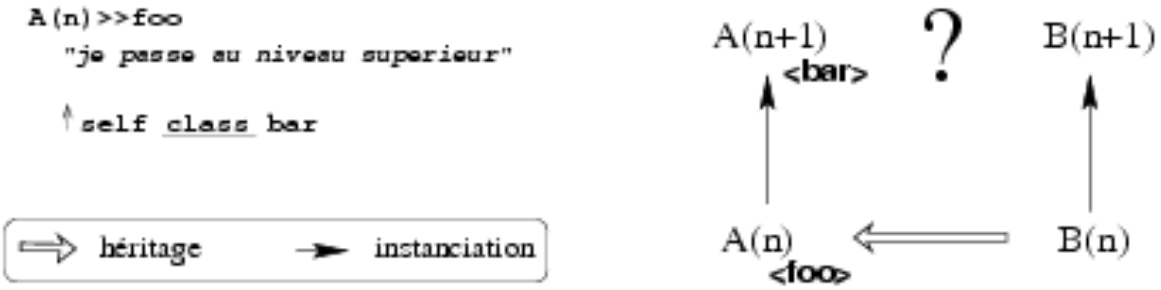


Fig 2-2 Problème de comptabilité ascendante

1.2.4 Comptabilité descendante

Le problème de comptabilité descendante correspond au problème symétrique à celui de la compatibilité ascendante.
Soit la méta-classe A(n+1) qui définit la méthode bar créant une instance de son receveur et lui envoie le message **foo**. La classe B(n+1), sous classe de A(n+1), hérite donc de la méthode **bar.**
Quand une instance de B(n+1) reçoit le message **bar** elle crée une instance et lui envoie le message **foo**. La recherche du message **foo** commence à partir de la classe B(n). Comment alors garantir que les instances de B(n) comprennent le message **foo** ? Il s'agit là d'un problème de comptabilité descendante.

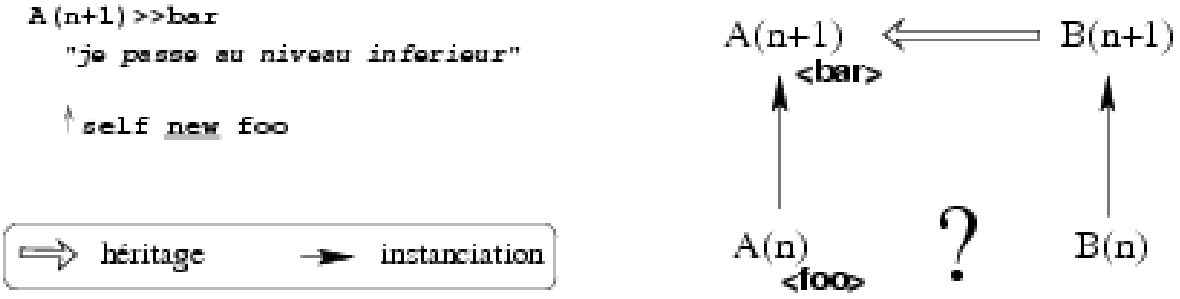


Fig 2-3 Problème de comptabilité ascendante

1.2.5 Hiérarchie parallèle

Afin de masquer une apparente complexité liée à la manipulation des classes en tant qu'objets, les concepteurs de Smalltalk ont fait le choix d'utiliser des méta-classes anonymes, non

partageables. En réalité, un langage comme Smalltalk privilégie la compatibilité entre niveaux d'abstraction au détriment de la composition des propriétés de classes. Voilà pourquoi Smalltalk a introduit une nouvelle organisation de la hiérarchie classique qui distingue les différents niveaux d'abstraction. En effet, il a placé les classes et leurs méta-classes au même niveau d'abstraction. Concrètement, les méta-classes Smalltalk sont automatiquement gérées par le système et elles sont organisées en hiérarchies que le système maintient en parallèles à celles des classes, où chaque méta-classe Smalltalk ne possède qu'une seule instance. Une méta-classe Smalltalk utilise le nom de son instance unique pour s'imprimer en le postofiant par 'class'. La méta-classe de la classe **Insecte** s'écrit donc **Insecte_Class**.
Lorsque nous définissons une classe B sous classe de A, la méta-classe de B (B_class) est automatiquement générée. Elle est définie comme sous classe de A_class, la méta class de A. Du fait de l'héritage parallèle, la classe B hérite les méthodes de A et la méta-classe B_class hérite les méthodes de A_class. Les problèmes de compatibilités descendante et ascendante sont donc vérifiés.

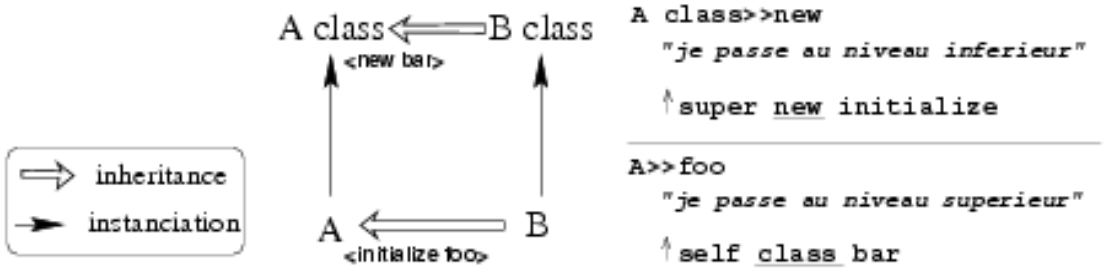


Fig 2-4 Hiérarchie Parallèle

1.2.6 Les Limites de l'héritage parallèle

L'héritage parallèle entre classes et méta-classes est la seule solution qui garantit les comptabilités ascendante et descendante. Cependant, elle ne permet pas d'ajouter une propriété à une classe sans affecter ses sous-classes. Par exemple, si, pour rendre une classe abstraite, l'utilisateur redéfinit l'allocateur (la méthode new) au niveau de sa méta-classe, les sous classes deviendraient elles aussi abstraites.

2. L'environnement Moose

La réingénierie est un concept qui consiste à repenser ce qui a été conçu dans une démarche d'ingénierie. Dans le secteur informatique, plusieurs branches sont concernées par cette discipline: la réingénierie des systèmes d'information, la réingénierie des processus, la réingénierie logicielle etc.

La réingénierie logicielle a pour objectif d'améliorer la qualité d'un logiciel existant, de l'adapter à de nouvelles contraintes réglementaires, d'ajouter des fonctionnalités ou de faciliter sa maintenance. Pour mener à bien un travail de réingénierie, des logiciels spécifiques, tel que Moose, ont été conçus.

2.1 Le projet FAMOOS

FAMOOS (*Framework-based Approach for Mastering Object-Oriented Software Evolution*) est un projet européen dont les partenaires comprennent le groupe SCG de l'université de Berne, Forschungszentrum Informatik, Daimler-Benz, Nokia Corporation, Sema Group et TakeFive Software

Le projet a pour objectif de fournir un cadre pour l'intégration d'outils de réingénierie des systèmes écrits dans différents langages orienté-objet. Au final, le résultat obtenu est un environnement nommé MOOSE qui sert de base pour un ensemble d'outils d'exploration de logiciels. Ces derniers sont divers, citons par exemple **AUDIT-RE** pour l'évaluation de qualité de logiciels, **GOOSE** pour fournir le support automatique de la détection de problèmes et **CodeCrawler** qui combine les techniques de visualisation et de métriques.

Pour l'analyse d'un code source, FAMOOS utilise une architecture en couches, présentée dans la figure ci-dessous.

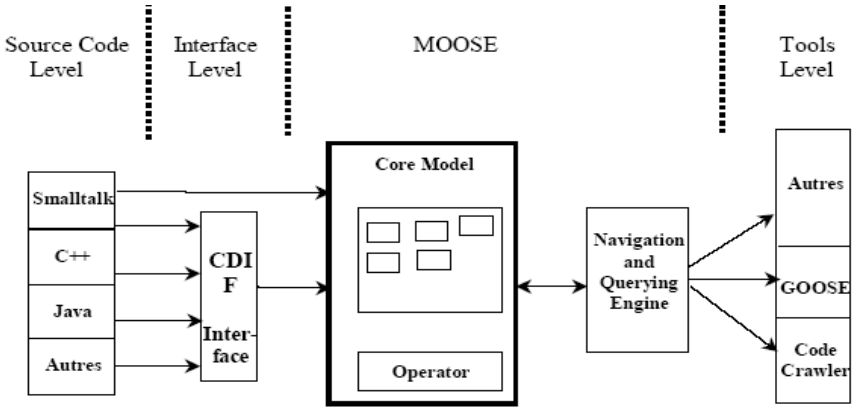


Fig 2-5 le projet FAMOOS

Comme MOOSE est lui-même en Smalltalk, tout programme écrit dans ce langage, peut être importé directement selon le méta-modèle sur lequel repose Smalltalk.

Pour d'autres langages, MOOSE fournit une interface appelée *CDIF* qui traduit le code source en un format d'échange standard compréhensible par l'environnement.

MOOSE transforme ce qu'il reçoit en entrée en une structure de données spécifique appelée *MooseModel.* Cette structure stocke les informations et leurs relations contenues dans le code importé.

MOOSE repose sur le méta modèle FAMIX que nous décrirons plus loin. D'une part, les divers outils de MOOSE sont fondés sur ce méta-modèle. D'autre part, la structure MooseModel contient des éléments également conformes à ce mét- modèle.

C'est pour cela que la réingénierie du code importé dans le noyau Moose est possible.

2.2 Moose, une plate-forme de réingénierie

Moose dispose d'un Framework de visualisation appelé **Mondrian** qui définit des techniques visuelles pour faciliter la compréhension de systèmes complexes. **Moose** est également capable d'évaluer des métriques logicielles pour la réingénierie et de détecter la duplication de code dans un programme. Par ailleurs, l'outil Moose propose une analyse dynamique et statique d'une application pour la génération de vues composables et d'information de collaboration.

Moose est un outil universel. Il capable d'importer des programmes implémentés dans différents langages objets (Smalltalk, Java, C++, Python).

Ce chapitre présente brièvement le fonctionnement de quelques outils de la plate-forme Moose.

2.2.1 Le navigateur de Moose (Moose Browser)

Moose est associé à un navigateur composé de différents panneaux.
Le panneau de gauche affiche la liste des projets importés dans Moose. Dans l'exemple ci-dessous, nous avons choisi d'analyser les caractéristiques du projet LAN (projet surligné).
Le panneau du milieu répertorie les entités spécifiques constituant du projet sélectionné. Dans le cas du projet LAN, on peut voir qu'il est composé de classes, d'attributs, de méthodes, de variables globales et locales etc...

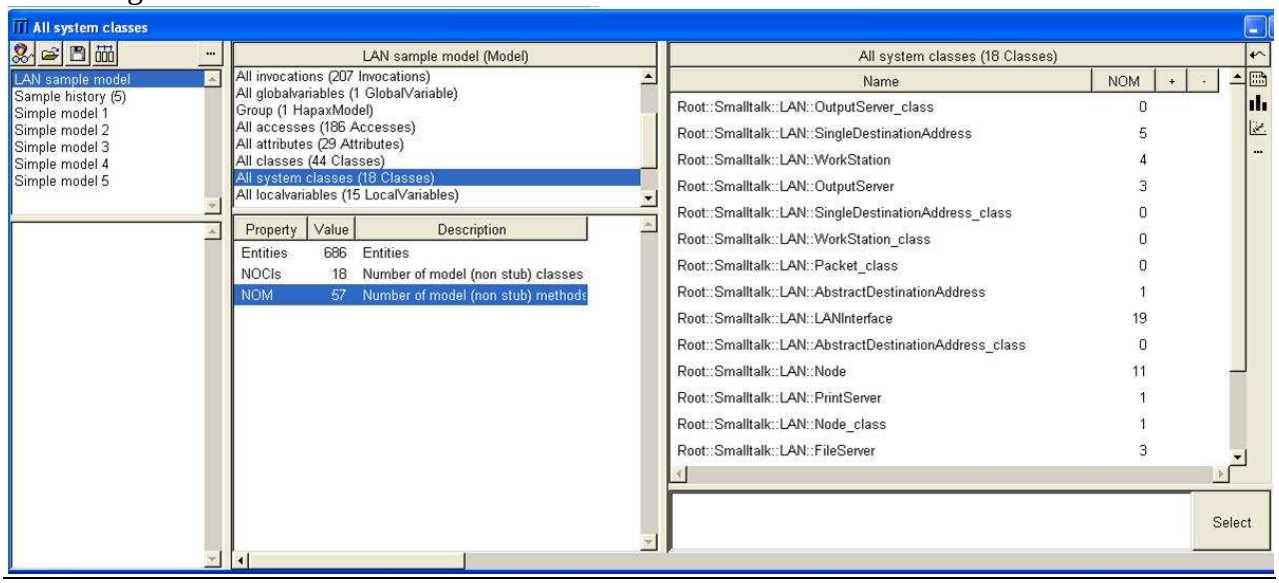


Fig 2-6 le Moose Browser

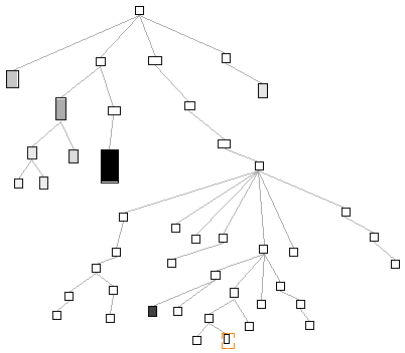
Le navigateur offre la possibilité de calculer pour chaque entité du modèle un certain nombre de métriques. Ainsi, on peut dénombrer par exemple les méthodes (option NOM), les attributs (option NOA) ou les lignes de codes dupliquées (option IDUPLINES) contenus de chaque classe du modèle (IDUPLINES).

Le panneau de droite met en relief les calculs du navigateur. Dans l'exemple ci-dessous, nous avons choisi de recenser le nombre de méthodes implémentées dans chaque classe du projet LAN. La vue que nous obtenons est composée d'une table à deux colonnes qui associe le nom des classes au nombre de méthodes qu'elles contiennent. On peut également sélectionner une entité et l'inspecter afin de connaître le méta modèle ou le méta méta modèle dont elle provient etc.

Le navigateur intégrer aussi un certain nombre d'outils d'analyse que l'on peut appeler par l'intermédiaire d'un menu volant en cliquant sur le projet concerné.

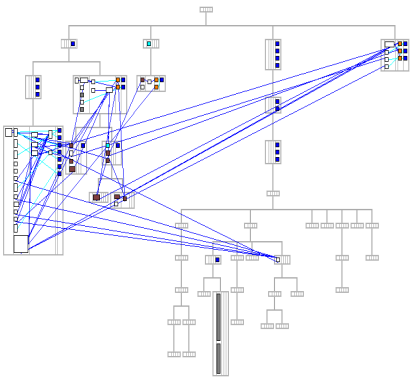
2.2.2 Mondrian

Le Framework Mondrian est capable de générer des visualisations très disparates à partir d'un système importé dans Moose. Mondrian est implémenté sous VisualWorks. Nous présentons ici quelques unes de ces visualisations.



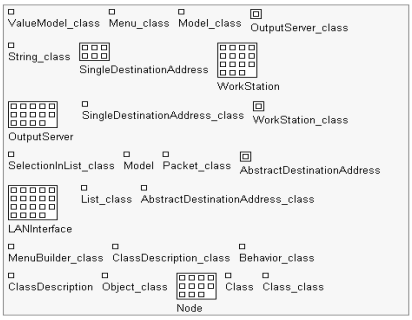
Sytem Complexity

System Complexity est une vue qui s'applique aux classes d'un modèles. Cette vue représente les hiérarchies de classe. Chaque rectangle représente une classe et sa taille est indiquée par une métrique calculée.



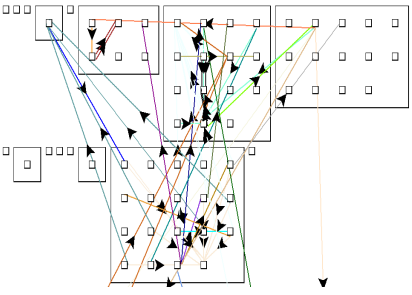
Blueprint complexity

Mondrian est capable de représenter des graphes complexes et de mettre en évidence les interactions qui existent entre les classes du modèle. Ce sont ces interactions que met en évidence le diagramme **Blueprint complexity**.



Distribution Map

Distribution Map apporte des analyses intéressantes pour la compréhension globales d'un système logiciel complexe. En effet, cette visualisation met en relief la façon dont sont distribuées les propriétés d'un logiciel au sein du système. Dans cette vue, le système est scindé en différentes parties et les propriétés sont caractérisées par des couleurs.



Method invocation

Method invocation permet de distinguer les méthodes qui sont invoquées dans le logiciel. On peut alors se demander pourquoi certaines méthodes ne sont pas invoquées, si cela est noraml ou si ces méthodes sont inutiles et leur suppression peut réduire la complexité du système.

2.2.3 Mondrian Easel

Mondrian propose un éditeur générique, appelé Mondrian Easel, qui génère lui aussi des vues particulières. Par exemple, l'éditeur de Mondrian peut produire une vue mettant en relief comment les entités sont associées entre elles. Dans cette vue, chaque noeud représente une entité dont la couleur correspond au type dont elle fait partie. Les bords gris chaque bord représente le lien unissant une entité qui fait référence à une autre entité.

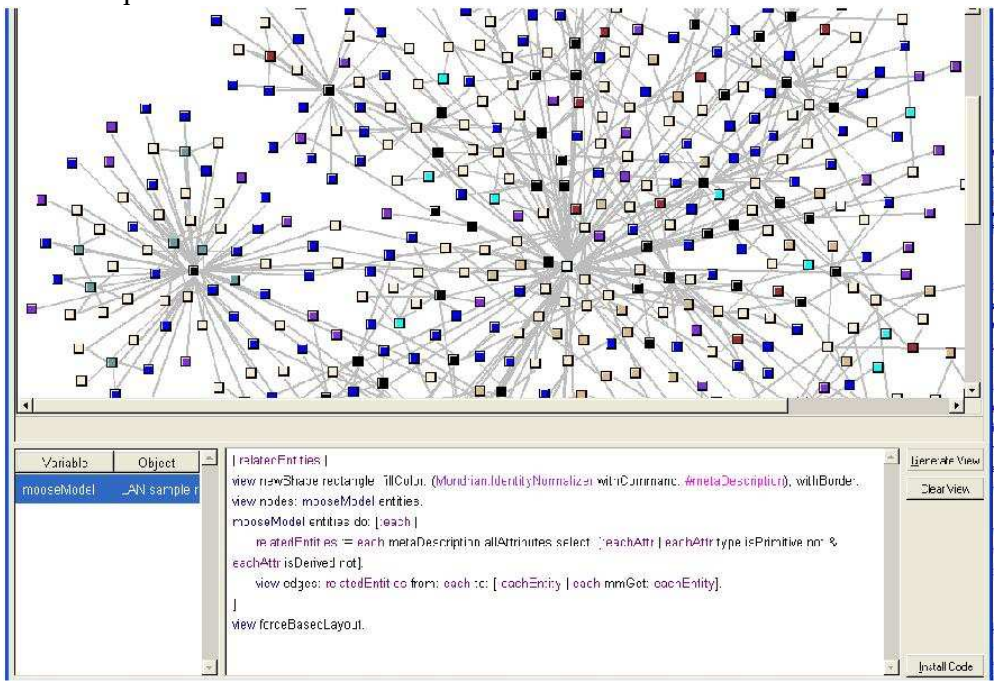


Fig 2-7 Références entre objets

2.2.4 Chronia

Chronia est un l'outil Moose qui établie un lien avec CVS. Cet outil propose une vue qui met en relief la façon dont le travail a été réparti entre les progammeurs participant au développement d'un grand projet. Chaque programmeur est associé à une couleur, l'axe horizontal représente le temps et les traits horizontaux correspondent aux fichiers développés pendant une durée précise.

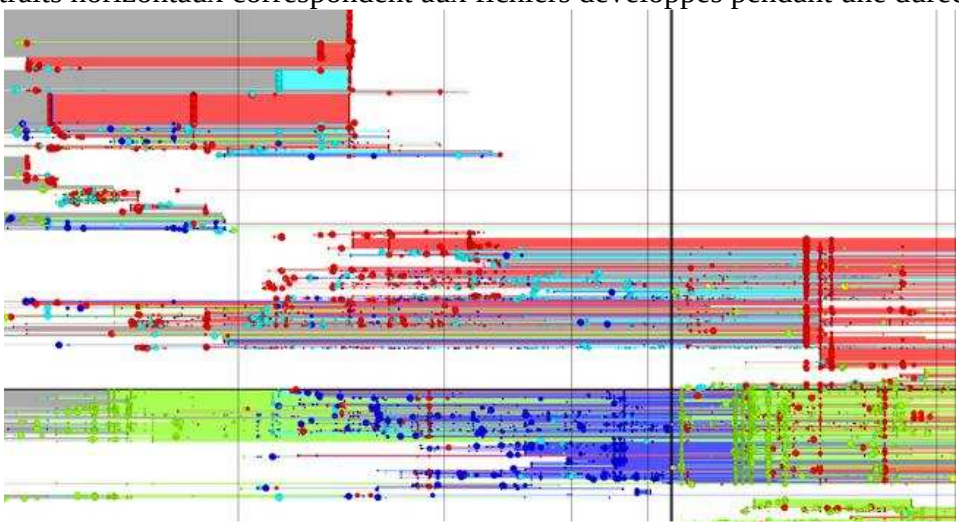


Fig 2-8 Chroni

2.3 Le méta-méta modèle EMOF

Le **Meta-Object Facility** (MOF) est un standard de l'OMG adressant la représentation des métamodèles et leur manipulation. Le langage MOF est un langage auto- descriptif .

Le standard MOF est situé au sommet d'une architecture de modélisation en 4 couches:

- M3: le méta-méta-modèle MOF (couche auto descriptive)
- M2: les méta-modèles
- M1: les modèles
- M0: Le monde réel

EMOF est un méta-méta-modèle repose sur la version MOF 2.0. Il est utilisé pour définir des méta-modèles fondés sur les concepts orientés objets.

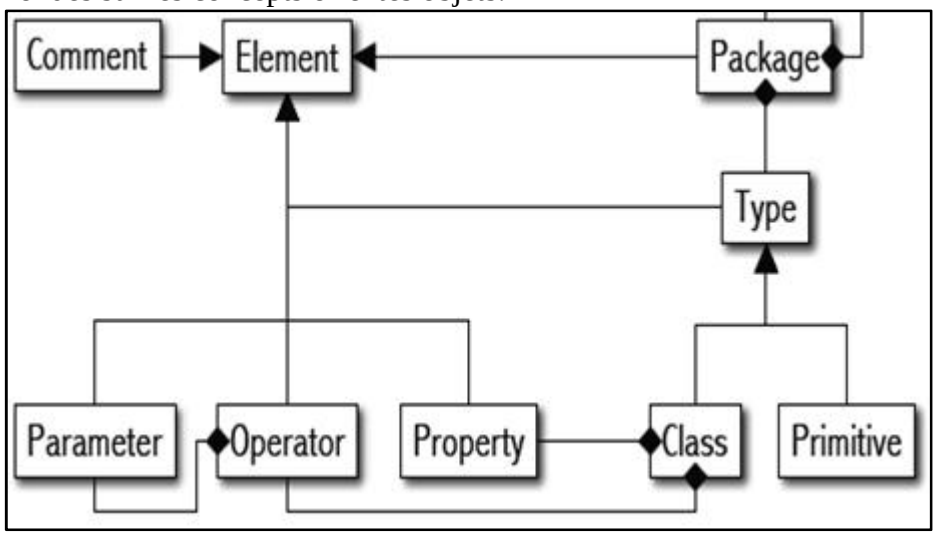


Fig 2-9 Méta-méta-modèle EMOF

2.4 Le méta modèle FAMIX

2.4.1 FAMIX un méta modèle exécutable

Les méta-modèles MOF, EMOF et ECORE décrivent les entités structurelles et leurs relations mais n'autorisent qu'une description informelle et non exécutable des opérations. En effet, ces méta-modèles ne définissent pas la notion de **comportement**. Ils ne peuvent donc pas être utilisés pour spécifier les sémantiques opérationnelles (comportement, conformité) des méta-modèles. Or, pour qu'un méta-modèle soit **exécutable,** il doit absolument être renforcé par des opérations et des propriétés qui dénotent de sa sémantique opérationnelle.

L'utilisation d'un méta-modèle exécutable dans un procédé de réingénierie permet de le rendre plus efficace et plus flexible. Les développeurs ont ainsi la possibilité d'utiliser leur environnement de développement pour travailler aussi bien sur le langage de base que sur le méta-modèle.

C'est pour cette raison que les concepteurs de Moose ont implémenté leur propre méta modèle exécutable: le méta-modèle FAMIX. C'est aussi pour cela que le langage SMALLTALK a été

choisi : il permet de décrire précisément le comportement des objets du système étudié directement dans le méta- modèle.

2.4.2 FAMIX, un méta modèle indépendant des langages

Comme nous l'avons vu précédemment, Moose supporte l'évolution des systèmes orienté-objet écrits dans différents langages. Pour pallier à la diversité des langages de programmation utilisés, MOOSE a donc besoin d'être lié à un méta modèle **modèle indépendant des langages**. Par conséquent, FAMIX a été conçu comme un méta-modèle indépendant des langages. Il permet ainsi de représenter les aspects centraux des langages objets tels que Smalltalk, C++, Java etc.

FAMIX comprend un modèle de noyau (*core model*) qui est commun pour tous les langages orienté-objet utilisés dans le code source Ce modèle contient un noyau, qui sert de base pour tous les langages. Par ailleurs, ce noyau est extensible et peut donc être adapté aux propriétés spécifiques d'un langage et aux différents besoins de rétro-ingénierie.

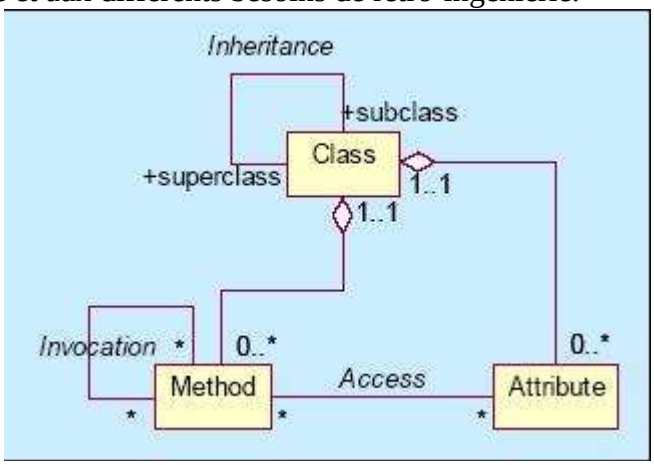


Fig 2-10 Cœur de FAMI

Le noyau contient trois types d'entités : les classes, les méthodes et les attributs. Une classe est composée d'attributs et de méthodes. Une classe peut hériter d'une autre classe. Une méthode peut invoquer d'autres méthodes ou accéder à des attributs.

2.4.3 FAMIX, un méta-modèle qui définit la notion de dépendance interne

Contrairement à UML, FAMIX définit une notion primordiale dans le cadre de la réingénierie : la notion de dépendance interne comme **les invocations de méthode** et **les accès aux variables**. Ces dépendances sont nécessaires pour résoudre les problèmes de phases de détection et de réorganisation du cycle de vie de la réingénierie Un méta-modèle comme UML n'aurait pas été suffisant pour définir les opérations de réingénierie. FAMIX s'inspire néanmoins étroitement d'UML, au niveau de la terminologie et des conventions.

2.4.4 Lien entre FAMIX et EMOF

FAMIX est un méta-modèle. Les entités FAMIX sont décrites par des instances du méta-méta-modèle EMOF. Par exemple, FAMIXClass est décrite à la fois par les instances de

EMOFClass et de EMOFAttibute. Dans l'exemple ci-dessous, la classe Point est représentée par une instance de FAMIXClass.

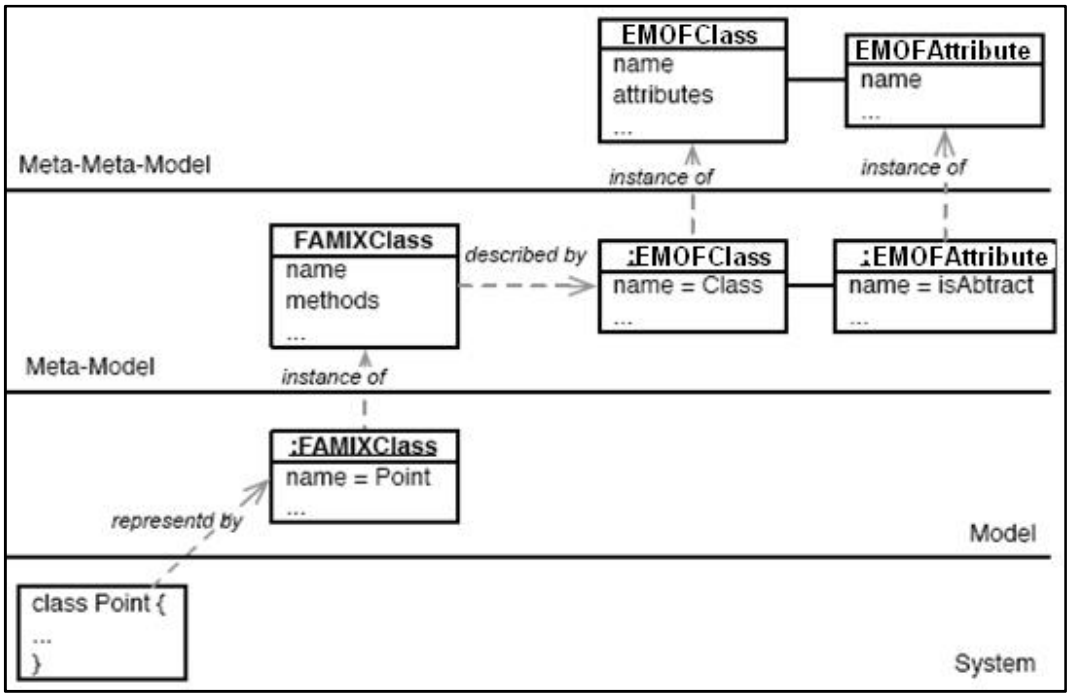


Fig 2-11 EMOF et FAMIX

3. Le plug-in EMF

2.1 L'outil EMF

Les logiciels informatiques sont devenus si complexes que l'on ne peut plus mener un projet sans faire de la modélisation. Pour un outil de développement comme Eclipse, une ouverture à l'ingénierie des modèles était donc indispensable. On a alors ajouté des plug-ins à Eclipse pour pouvoir faire de la modélisation.

En informatique, un plugin ou plug-in, de l'anglais to plug in (brancher), est un logiciel tiers venant se greffer à un logiciel principal afin de lui apporter de nouvelles fonctionnalités. EMF est donc le *plug-in* que l'on a ajouté à l'outil Eclipse pour faire de la modélisation

EMF est constitué trois parties :

EMF – Le cœur du framework EMF inclut un méta modèle (Ecore) décrivant les modèles et fournissant les supports à l'exécution des modèles comme les notifications des modifications, la persistance grâce à une sérialisation par défaut en XMI, ainsi qu'une API très efficace pour manipuler les objets EMF de façon générique.

EMF.Edit – Ce framework comprend des classes génériques réutilisables permettant de créer des éditeurs pour des modèles EMF. Il fournit :

- Des classes de contenu et de libellé, un support de source des propriétés, et d'autres classes utiles permettant aux modèles EMF d'être affichés par des visionneurs standards (JFace) et des fenêtres de propriétés.

- Un framework de commandes, comprenant un jeu de classes d'implémentation de commandes génériques, destinées à créer des éditeurs qui supportent entièrement et automatiquement les fonctionnalités *Annuler* et *Répéter*.

EMF.Edit permet ainsi de générer des modèles conformes à un méta modèle donné. En effet, La partie EMF.Edit génére automatiquement les interfaces et les classes d'implémentation (appelées *ItemProvider*) correspondant au méta modèle Ecore et qui vont permettre l'affichage et l'édition de ses classes ainsi qu'un éditeur arborescent sous la forme d'un plugin pour Eclipse.

L'éditeur ainsi généré va permettre de créer des modèles qui respecteront le méta modèle de départ, notamment en adaptant les menus contextuels (par exemple le menu *New Child*). Ceci est possible en implémentant l'interface *IEditingDomainItemProvider* dans les *ItemProvider*.

EMF.Codegen – Cet utilitaire de génération de code est en mesure de générer n'importe quel code nécessaire pour créer un éditeur complet pour un modèle EMF. Il comprend un GUI à partir duquel les options de génération peuvent être spécifiées et d'où les générateurs peuvent être invoqués. Cette fonctionnalité vint en plus du composant JDT (Java *Development Tooling*) d'Eclipse.

Trois niveaux de génération de code sont supportés :

- Model : fournit des interfaces Java ainsi que les classes d'implémentation pour toutes les classes du modèle, plus un factory et une classe d'implémentation de package (méta donnée).

- Adapters : génèrent les classes d'implémentation (appelées *ItemProviders*) qui adaptent les classes du modèle à l'affichage et l'édition.

- Editor : produit un éditeur bien structuré conforme aux recommandations pour les éditeurs de modèle EMF pour Eclipse, servant comme point de départ pour une personnalisation.

Tous les générateurs supportent la regénération du code tout en préservant les modifications de l'utilisateur. Les générateurs peuvent être invoqués soit à travers le GUI, soit par le biais d'une ligne de commande.

2.2 Le Méta-modèle ECORE

Ecore est basé sur MOF1.3. Cependant, il a connu de nombreuses évolutions et simplifications pour permettre l'implantation de nombreux outils. Par exemple, les concepts d'Association ou de Constraint présent dans MOF1.3 ont disparu de Ecore, alors que le concept de Factory a fait son apparition.

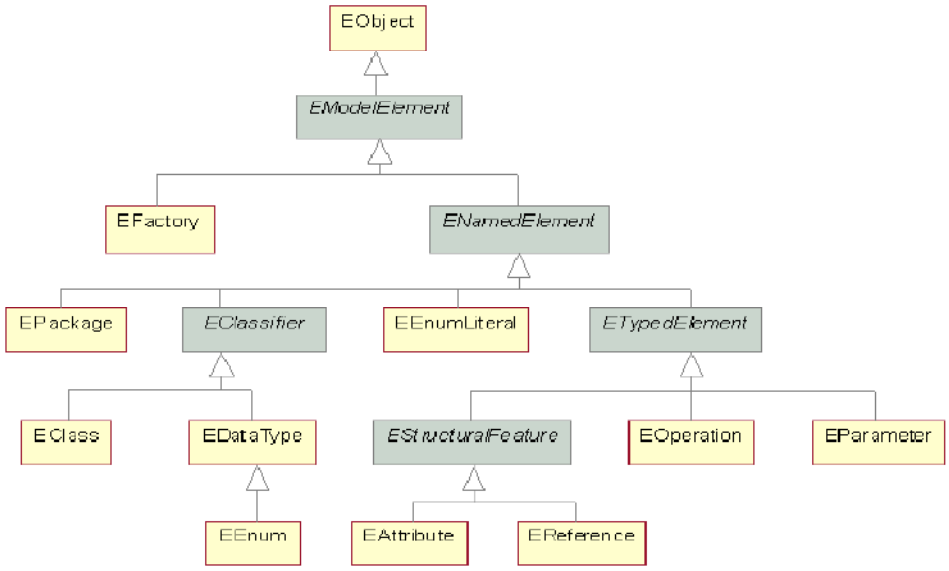


Fig 2-12 Méta-méta-modèle ECORE

2.3 Le format XMI

Les modèles EMF sont enregistrés sous le format ECORE. En réalité, tout modèle Ecore est totalement équivalent à sa sérialisation XMI, ce qui signifie qu'on peut lire un modèle EMF en format XMI.

Chapitre 3 Génération de Code

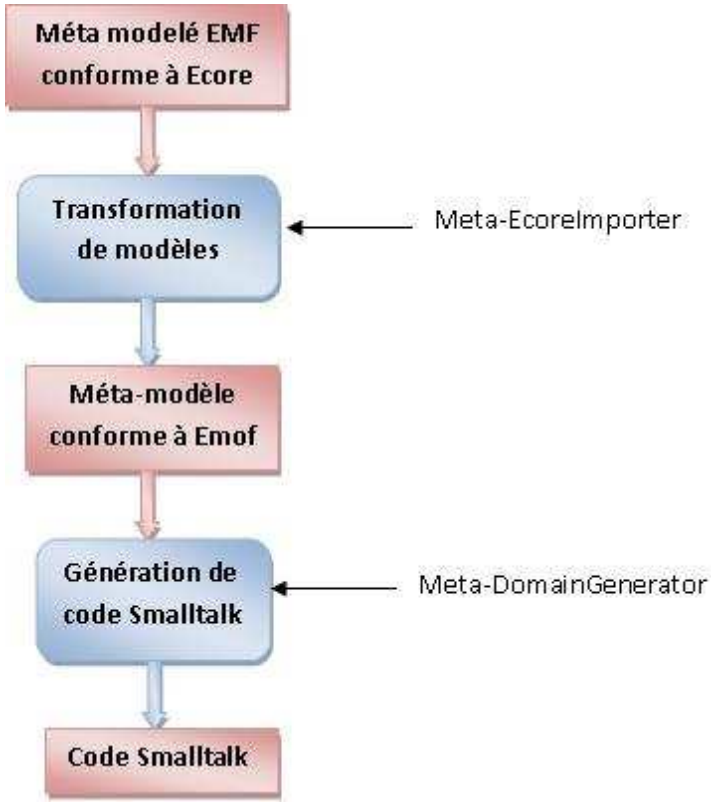
L'objectif de ce projet est d'implémenter une passerelle reliant le plug-in EMF à l'outil Moose. Cette passerelle doit être bidirectionnelle: elle doit permettre d'une part l'importation de modèles EMF dans Moose et d'autre part l'exportation de projets Moose vers EMF. Ce chapitre traite le cas de l'importation et développe en particulier l'aspect 'génération de code'.

1. Analyse du problème

L'importation de modèles EMF comprend deux cas d'utilisation différents. Dans le premier cas, l'utilisateur souhaite importer un méta-modèle EMF afin d'évaluer, grâce à l'outil Moose, la qualité de sa conception. Dans le deuxième cas, l'utilisateur souhaite importer et analyser un modèle EMF généré à partir d'un méta-modèle qu'il a lui-même réalisé. Ces deux cas d'utilisation requièrent évidemment un traitement bien distinct.

1.1 Cas de l'importation de méta-modèle EMF

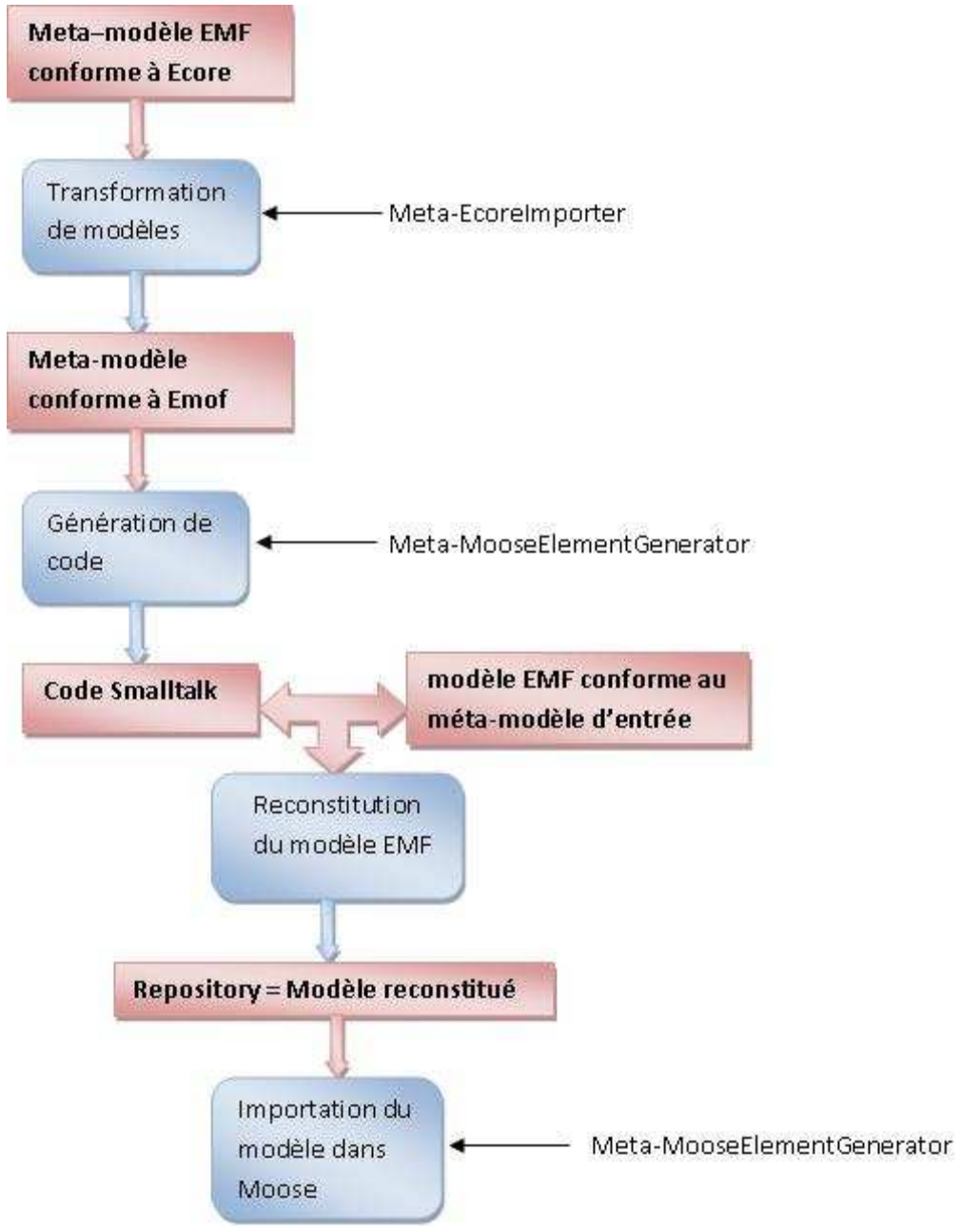
L'importation d'un méta-modèle EMF dans Moose comprend deux étapes. La première étape consiste en une transformation de modèles. Il s'agit de transformer un méta-modèle EMF conforme à ECORE en un méta-modèle conforme à EMOF. Cette transformation est assurée par le package **Meta-EcoreImporter**. La deuxième étape consiste en une génération de code. Il s'agit de générer les classes Smalltalk correspondant au méta-modèle précédemment transformé. La génération de code est assurée par le package **Meta-DomainGenerator**. Une fois le code généré, l'utilisateur peut l'importer dans Moose pour y exploiter les différents outils d'analyse qu'il propose.



1.2 Cas de l'imporation de modèle EMF conçu à partir d'un méta-modèle personnel

L'importation d'un modèle conforme à un méta-modèle EMF comprend quatre étapes. Les deux premières étapes sont similaires à celles décrites précédemment car pour pouvoir importer un modèle conforme, il est nécessaire au préalable de générer le code du méta-modèle correspondant. Cependant, le générateur de code utilisé est cette fois différent : il est implémenté dans le package **Meta-MooseElementGenerator**. La raison de ce changement est expliquée dans les paragraphes qui suivent.

Cette étape de génération de code est indispensable car pour reconstituer le modèle EMF conforme à un méta-modèle particulier, il est nécessaire de créer les instances des classes précédemment générées à partir de ce méta-modèle. Ces instances sont ensuite regroupées dans une même structure, qu'on appelle un **repository**, et qui devient alors une représentation du modèle EMF d'entrée que l'environnement de Smalltalk peut comprendre. Enfin, la dernière étape consiste à importer ce repository dans Moose.



Dans le cadre de ce projet, je me suis concentré sur la génération de code. J'ai donc implémenté les packages **Meta-DomainGenerator** et **Meta-MooseElementGenerator**.

2. Analyse de l'existant

Dans Moose, il existe déjà un générateur de code. La classe qui l'implémente s'appelle **BasicCodeGenerator**. Cette classe génère les **namespaces** (espaces de nommage) ainsi que les classes et leurs variables d'instance.

La classe **BasicCodeGenerator** est conforme au motif de conception « Visiteur ». Elle a été créée pour s'appliquer aux entités **EMOF**. En effet, par conception, toutes les entités EMOF doivent mettre à disposition une méthode appelée *acceptMetamodelVisitor*. Cette dernière fait en retour, appel à la méthode « visite » de la classe **BasicCodeGenerator** qui lui correspond. Par exemple, la méthode *acceptMetamodelVisitor:* de la classe **Package** appelle *visitPackage* de la classe **BasicCodeGenerator** tandis que celle de la classe **Class** appelle *visitClass* etc. De cette manière, les données nécessaires sont obtenues pour générer le code spécifique à l'entité **EMOF** visitée.

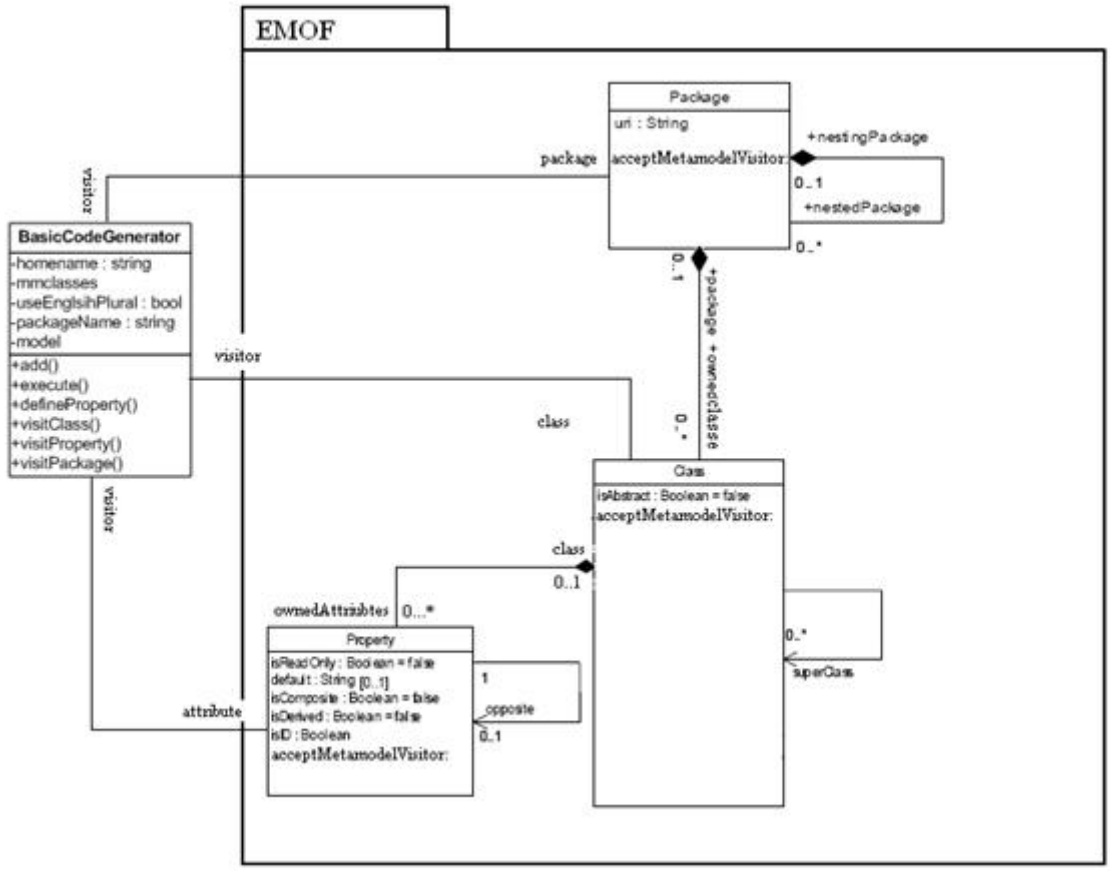


Fig 3-1 Diagramme de classes

Cependant, **BasicCodeGenerator** reste un générateur incomplet car il ne s'applique pas à toutes les entités du méta-modèle **EMOF**. En effet, il ne gère pas les opérations, les paramètres, les commentaires, les énumérations et les littéraux. J'ai donc implémenté les classes **DomainCodeGenerator** et **MooseElementCodeGenerato**r qui s'inspirent directement de la classe **BasicCodeGenerator**.

3. Deux types de générateur

Dans de nombreux langages orientés objet, l'arbre d'héritage possède une unique classe racine, la seule à ne pas hériter d'une superclasse. En Smalltalk, comme en Java, cette classe particulière s'appelle **Object**. Elle rassemble les attributs et les méthodes communs à l'ensemble des objets du système. En Smalltalk, en particulier, c'est dans cette classe que sont définies les méthodes associées aux sélecteurs des messages, les traitements des erreurs, etc.

Normalement, lorsqu'on souhaite implémenter une classe ne possédant pas de superclasse spécifique, par défaut, on l'a fait hériter de la classe **Object**. Néanmoins, dans certains cas, il est nécessaire de la faire hériter de la classe **MooseElement**. En effet, **MooseElement** est une classe particulière de VisualWorks qui est directement liée à l'outil Moose. Par exemple, toutes les entités **FAMIX** implémentés dans Visual Works héritent directement ou indirectement de la classe **MooseElement**. C'est grâce à cela que l'outil Moose est capable d'importer des programmes implémentés en Smalltalk.

Ainsi, la différence entre **DomainCodeGenerator** et **MooseElementCodeGenerator** se situe au niveau de la génération des classes. En effet, **DomainCodeGenerator** génère simplement les classes qui héritent d'Object tandis que **MooseElementCodeGenerator** crée les classes à partir de **MooseElement**.

Concrètement, si l'utilisateur souhaite importer dans Moose un modèle conforme à un méta modèle, il devra utiliser le générateur **MooseElementCodeGenerator** (cf. 1.2). En revanche, s'il souhaite simplement générer du code Smalltalk à partir d'un modèle Ecore sans se soucier des instances du modèle, il utilisera de préférence le générateur **DomainCodeGenerator** évitant ainsi une surcharge inutile (cf. 1.1).

4. Génération de code

Le générateur de code applique un traitement spécifique à chaque entité du méta-modèle EMOF qu'il rencontre.

4.1 Génération d'un Bundle

Dans Visual Works, il existe un répertoire capable de contenir tous les packages d'un projet. Le générateur de code crée systématiquement ce répertoire, qu'on appelle un *bundle*. Il offre d'ailleurs à l'utilisateur la possibilité de nommer lui-même le *bundle* dans lequel il souhaite générer ses packages.

4.2 Génération d'un package

4.2.1 Package et Namespace

Un **package** est une structure qui regroupe un ensemble d'éléments de programmation (procédures, classes, d'autres packages etc.) qui sont liés et cohérents entre eux. Les packages créent un niveau d'abstraction supérieur aux classes permettant de mieux appréhender le développement de programmes

complexes. Ils permettent aussi de mieux construire des modules indépendants que l'on peut extraire et maintenir séparément. Ils facilitent également le travail en équipe, chacun pouvant travailler sur un package séparément.

Un **namespace** est une zone de déclaration qui permet de délimiter la recherche des noms des identificateurs par le compilateur. Le but des namespaces est d'enfermer, dans une zone délimitée, des noms ayant un sens sémantique précis pour un espace de nommage donné.

Le langage C ne possède pas la notion de namespace. Ainsi, dans ce langage, lorsque deux programmeurs utilisent le même nom dans deux modules séparés, un « conflit de noms » peut apparaître à l'édition des liens, empêchant son aboutissement. Ce conflit peut être problématique à résoudre si le nom utilisé fait référence à des concepts différents. En C++, la notion de namespace a été introduite pour résoudre ce genre de problème.

En Java, les notions de package et de namespace sont indissociables. En effet, lorsqu'un package est crée en Java, son namespace lui est systématiquement associé. Par conséquent, l'ensemble des classes d'un package sont regroupées au sein d'un unique namespace et ainsi, ne rentreront pas en conflit éventuel avec d'autres classes issues d'un package différent.

Dans VisualWorks, les notions de package et de namespace sont distinctes. En effet, un namespace est considéré comme un objet contenu dans son package tandis qu'un package est un répertoire où sont stockés les éléments de programmation (classes, variables partagées, namespaces). Le programmeur a donc la liberté de créer ses propres namespaces ou bien d'utiliser ceux déjà définis par le système.

Dans VisualWorks, le namespace par défaut s'appelle 'Smalltalk'. Le générateur **DomainCodeGenerator** génère donc les éléments dans ce namespace par défaut. Par contre, le générateur **MooseElementCodeGenerator** utilise le namespace dédié à Moose, c'est-à-dire 'SCG.Moose'.

4.2.2 Gestion des packages par le générateur de code

Le générateur propose générer les espaces de nommages en tant qu'option. En effet, si l'utilisateur choisit l'option « namespaces », le générateur de code créera, pour chaque package d'un modèle, un répertoire « package » associé à un namespace qui lui est propre. Si au contraire l'utilisateur préfère ignorer cette option, le générateur produira toujours des « packages ». En revanche, il ne génèrera qu'un seul namespace commun à tous les éléments du modèle.

4.3 Génération d'une classe

4.3.1 Définition d'une classe Smalltalk sous Visual Works

Dans VisualWorks, la définition d'une classe Smalltalk se présente sous la forme suivante :

```
Smalltalk.Library defineClass: #Book
superclass: #{Core.Object}
indexedType: #none
private: false
```

```
instanceVariableNames: ''
classInstanceVariableNames: ''
imports: ''
category: 'Library'
```

Smalltalk.Library est le namespace auquel la classe Book est liée. **Book** correspond au nom de la classe. superclass : est le champ associé au nom de sa super classe. category la relie au package auquel elle appartient. Enfin, instanceVariablesNames regroupe les attributs de la classe.

Pour chaque classe d'un modèle, le générateur de code génère ainsi la classe Smalltalk qui lui est associée, en la plaçant dans le package approprié et en la reliant au namespace qui convient. Les variables d'instances de la classe lui sont adjointes une fois que le générateur de code a rencontré les entités EMOF correspondant aux attributs de la classe qu'il est en train de traiter (cf. paragraphe 4.5).

4.3.2 Héritage

En Smalltalk, l'héritage est *simple* contrairement à C++, pour lequel l'héritage est *multiple*. Dans certains modèles, on peut donc rencontrer des classes possédant plusieurs classes mères. Dans un tel cas de figure, le générateur de code n'a pas d'autre alternative que de sélectionner une seule super classe parmi les autres.

Lorsqu'une classe ne possède pas de classe mère particulière, le générateur de code lui affecte la superclasse **Object** ou **MooseElement,** cela dépend du générateur de code que l'utilisateur souhaite utiliser (cf. 3).

4.4 Gestion des Enumération et des littéraux

Dans le méta-modèle Ecore, une énumération EEnum est une structure contenant des littéraux ELiteral associés à des entiers. En réalité, une énumération sert à créer et à regrouper des types. Prenons l'exemple d'un modèle de librairie. L'énumération **BookCategory** regroupe trois principales catégories de livres : les bandes dessinées, associées au littéral **comic**, les thrillers associés à **thriller** et les livres fantastiques associés à **fantastic**.

Fig 3-2 Enumération BookCategory

En Smalltalk, les énumérations ne font pas partie des éléments de programmation. C'est pourquoi il a fallut trouver un équivalent. J'ai donc décidé de transformer une énumération en classe et ses littéraux en méthodes de classes. Concrètement, ces méthodes de classes portent le nom du littéral auxquels elles sont associées et qui retournent une chaine de caractère associant le nom du littéral et le nom de l'énumération auquel il appartient.

Par exemple, pour l'énumération BookCategory, la classe générée se présentera comme suit :

```
Smalltalk.Library defineClass: #BookCategory
superclass: #{Core.Object}
indexedType: #none
private: false
instanceVariableNames: ''
classInstanceVariableNames: ''
imports: ''
category: 'Library'
```

Par ailleurs, les méthodes de classe sont générées de la manière suivante :

```
BookCategory>> fantastic
        ^'BookCategory.fantastic'.

BookCategory>> thriller
        ^'BookCategory.thriller'

BookCategory>> comic
        ^'BookCategory.comic'
```

4.5 Génération des variables d'instance

Le méta-modèle ECORE définit deux catégories de variables d'instances : les **attributs** et les **références**. Les attributs sont associés à des types primitifs tels que int, long, double, byte etc. Les références, au contraire, sont des variables d'instances dont le type dérive d'une classe ou d'une énumération.

Le méta-modèle EMOF ne fait pas la différence entre les notions d'**attribut** et de **référence**. Dans EMOF, ces deux entités ne font qu'un et portent le nom de **propriété** (Property). Par contre, le méta-modèle fait bien la distinction entre les types primitifs et les types non primitifs. Dans Visual Works, il existe de nombreux types primitifs. Parmi eux, on distingue quatre types primitifs par défaut: **Number** pour les entiers, **UnlimitedNatura**l pour les réels, **Boolean** pour les booléens, **String** pour les chaînes de caractères. Les types non primitifs, quant à eux, proviennent en général des classes et des énumérations.

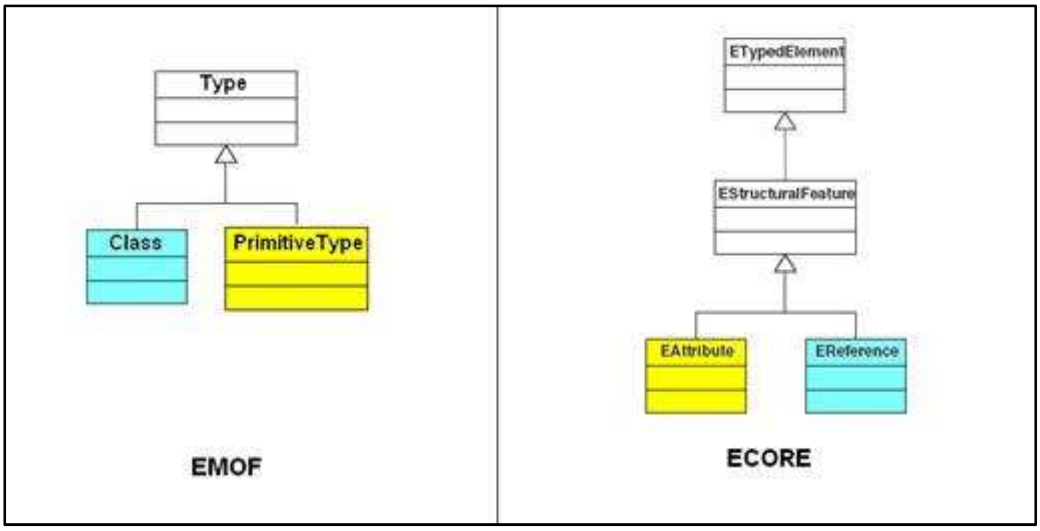


Fig 3-3 Comparaison entre EMOF et ECORE

Lorsque le générateur de code rencontre une **propriété**, il lui applique un traitement spécifique selon sa cardinalité et la nature de son type. En outre, lors la génération de code, les entités **Property** sont utilisées de différentes manières.

4.5.1 Définition d'une classe

Tout d'abord, les propriétés sont utilisées pour remplir le champ **instanceVariableNames** (cf 3.1) à la définition d'une classe. Par exemple, si la classe Book possède les variables d'instance 'title' et 'author', la classe Book se générera de la façon suivante :

```
Smalltalk.Library defineClass: #Book
  superclass: #{Core.Object}
  indexedType: #none
  private: false
  instanceVariableNames: 'title author'
  classInstanceVariableNames: ''
  imports: ''
  category: 'Library'
```

4.5.2 Méthode d'initialisation

Par ailleurs, les propriétés sont utilisées pour générer la méthode ***initialize,*** appelée à la création d'une instance, qui initialise les valeurs par défaut d'un objet.
En Smalltalk, la méthode d'initialisation s'écrit, par exemple pour la classe Book, sous la forme suivante ;

```
class Book>> initialize
                title:='the Bible'
                author:='anonymous'
```

Bien entendu, un traitement spécifique est effectué selon la cardinalité, la valeur par défaut et le type de la propriété.

Cardinalité

En effet, lorsqu'un attribut ou une référence EMF est de cardinalité multiple, le générateur de code initialise par défaut la propriété correspondante avec une collection Smalltalk appelée **OrderedCollection**. En revanche, lorsque l'attribut est simple, le générateur de code lui applique un traitement spécifique selon son type.

Les types primitifs

EMF définit 32 types primitifs différents que j'ai regroupés dans quatre catégories principales :

Tout d'abord, les types EMF qui représentent des collections: **ByteArray**, **List**, **FeatureMap**, **Map** et **ResourceSet**. Un attribut de type **ByteArray** est initialisé par un tableau (objet Array en Smalltalk) de taille 8 puisque un octet (Byte) est constitué de 8 bits. Un attribut de type **List** est associé à la collection Smalltalk également appelée List. De même, les attributs de type **FeatureMap** et **Map** correspondent à la collection Smalltalk Map. Enfin, un attribut de type **ResourceSet** est initialisé avec une collection Smalltalk appelée Set.

Ensuite, viennent les types EMF de base : **Int**, **IntegerObject**, **Long, LongObject**, **Double**, **Short, ShortObject**, **Float**, **FloatObject**, **BigDecimal**, **BigInteger**, **Boolean**, **BooleanObject**, **Byte**, **ByteObject**. Si ces attributs sont initialisés dans le modèle EMF, le générateur affecte aux propriétés correspondantes la valeur par défaut qui convient à l'initialisation. Si au contraire, les propriétés ne sont pas initialisées dans le modèle, alors le générateur ne les fait pas apparaître concrètement dans le code pour ne pas le surcharger. Néanmoins, concrètement, ces propriétés sont initialisées à nil.

Ensuite, on trouve les types EMF qui représentent un caractère: **Char** et **CharacterObject**. En Smalltalk, un caractère correspond au type Character et s'initialise de la manière suivante, par exemple pour le caractère 'a' :

```
var :=$a.
```

Ainsi, lorsqu'un attribut EMF de type **Char** et **CharacterObject** possède une valeur par défaut, le générateur de code initialise la propriété associée avec cette valeur mais en la préfixant par le symbol '$'. En revanche, si cet attribut n'a pas de valeur par défaut, le générateur de code la traitera comme pour les types de base : il l'initialisera à nil mais ne la fera pas apparaître dans le code.

La dernière catégorie regroupe les types EMF suivant : **String, Date, Enumerator, FeatureMapEntry, JavaClass, JavaObject, TreeIterator, DiagnosticChain, Resource.** En Smalltalk, une chaine de caractères est associée au type String et s'initialise de la manière suivante, par exemple pour la chaîne de caractères « Smalltalk » :

```
var :='Smalltalk'.
```

Par conséquent, lorsqu'un attribut EMF de type String possède une valeur par défaut, le générateur de code initialise la propriété associée avec cette valeur en la mettant entre deux cotes. Par ailleurs, un attribut de type **Date** dans EMF est traité par le générateur comme une chaîne de caractère. Par exemple, l'attribut EMF nommé **bithday** de type Date initialisé à **08-21-06**, sera généré dans la méthode *initialize* comme suit :

```
initialize
        birthday :='08-21-06'.
```

Enfin, j'ai décidé de traiter les types **Enumerator, FeatureMapEntry, JavaClass, JavaObject, TreeIterator, DiagnosticChain, Resource** comme les chaines de caractère et ce, pour éviter les erreurs dans la génération de code. En effet, nous n'avons pas trouvé les équivalents Smalltalk appropriés à ces types d'entité. Ainsi, lorsque ces types de propriété sont initialisés avec une valeur par défaut, c'est une chaine de caractère qui est générée. Ceci permet à l'utilisateur de retrouver dans le code la valeur qu'il a affecté à son attribut dans son modèle initial. Il pourra ensuite à sa propre initiative implémenter la classe Smalltalk appropriée pour initialiser sa propriété comme il le souhaite.

Les types non primitifs

Les types dérivant d'une énumération

Lorsque les attributs dérivent d'une énumération, l'initialisation est un peu plus complexe. En effet, comme nous l'avons décrit précédemment dans le paragraphe 4.3.4, les énumérations sont transformées en classe et ses littéraux en méthodes de classes. Par conséquent, la valeur par défaut de l'attribut en question correspond en fait une méthode de classe auquel le générateur doit faire appel pour initialiser la propriété correspondante. Par exemple, prenons la classe Book définie dans le modèle EMF de librairie :

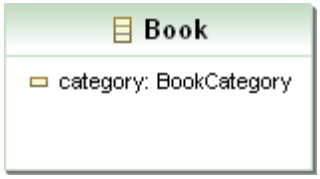


Fig 3-4 Classe Book

Ci-dessus, la classe Book possède un attribut appelé category de type BookCategory (il s'agit de l'énumération définie dans le paragraphe 4.4.1). Dans le modèle EMF, l'attribut category est initialisé avec une valeur par défaut correspondant au littéral thriller. Ainsi, le générateur de code initialisera la propriété category en faisant appel à la méthode de classe *thriller* définie dans la classe BookCategory :

```
Book>> initialize
        category := Library.BookCategory thriller.
```

les types dérivant d'une classe

Dans EMF, les références sont des variables d'instance qui dérivent d'une classe. Pour ce type de variable d'instance, le générateur de code crée la propriété associée et l'initialise systématiquement à nil.

4.5.3 Les accesseurs, les itérateurs, les méthodes « add », « remove » et « isEmpty »

Par ailleurs, le générateur de code propose plusieurs options : il laisse à l'utilisateur le choix de générer les accesseurs pour chaque propriété du modèle qu'il souhaite importer. Toutefois, si la propriété est spécifiée comme étant en lecture seule (isReadOnly), le générateur ne générera pas les « setters » pour la propriété.

Pour les propriétés à cardinalités multiples, il offre également la possibilité de générer les itérateurs suivants :

- **do:** pour parcourir une collection
- **collect:** qui applique un traitement à chaque item d'une collection dont le résultat est ensuite placé dans une nouvelle collection
- **select:** pour sélectionner des objets particuliers dans une collection
- **detect:** pour détecter un objet particulier dans la collection

Outre cela, le générateur donne la possibilité de générer les méthodes add : pour ajouter un élément dans une collection, **remove:** pour supprimer un élément d'une collection et **isEmpty:** pour tester si la collection est vide.

4.5.4 Les méthodes de description de modèle

Le générateur de code associe à chaque propriété une méthode de classe appartenant à un protocole de méta-modélisation. Ces méthodes sont utilisées pour fournir un certain nombre d'information sur le comportement d'un objet car elles décrivent précisément ses propriétés. Voici par exemple l'allure de ces méthodes de classe pour la classe Book modélisée ci-dessous:

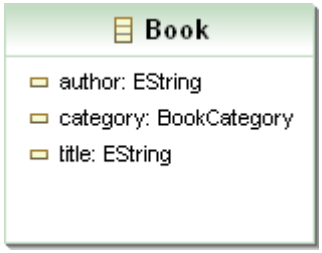


Fig 3-5 Classe Book

```
Book_class>>metamodelTitle
        ^EMOF.Property name: #title type: String
```

```
Book_class>>metamodelAuthor
        ^EMOF.Property name: #author type: String

Book_class>> metamodelCategory
^EMOF.Property name: #category type: Library.BookCategory
```

Ces méthodes sont en fait des méta-annotations utilisées pour construire des méta-modèles. En effet, comme elles décrivent et renvoient des entités EMOF, on peut par leur intermédiaire remonter au méta-modèle dont le code provient.
En outre, ces méthodes donnent un moyen de vérifier la fiabilité du code généré puisque grâce à elles, on peut directement avoir accès aux entités EMOF de type **Property** dont le code des classes générées est issue.

4.6 Génération des opérations

Lorsque le générateur de code rencontre une entité EMOF de type Operation, il génère la méthode associée dans la classe correspondante en prenant en compte ses paramètres et ses commentaires.

4.6.1 Les paramètres

En Java et en C, les paramètres des méthodes sont mis entre parenthèses les uns à la suite des autres et sont séparés par des virgules. En Smalltalk, la syntaxe est différente. En effet, les paramètres sont séparés par des connecteurs logiques qui donnent un sens plus précis à la méthode auquel ils appartiennent. Par exemple, la méthode sum(a,b), qui additionne a et b, s'écrirait en Smalltalk sum:a with :b. Ainsi, dans le cadre de la génération de code, lorsque l'opération à générer possède plusieurs paramètres, le générateur les sépare à l'aide du connecteur logique par défaut '**and** '.

4.6.2 Les commentaires

Une entité **Operation** possède un attribut qui regroupe l'ensemble de ses commentaires. Ces commentaires sont pris en compte par le générateur de code lors de la création d'une méthode. Les commentaires sont ainsi mis entre guillemets à la suite de la déclaration de la méthode correspondante.

4.6.3 Les méthodes de description de modèle

Comme pour les propriétés, le générateur de code associe à chaque opération une méthode de classe qui la décrit. Comme précédemment, ces méthodes renvoient l'entité EMOF dont provient l'opération générée

Chapitre 4 Export de projets Smalltalk vers EMF

L'objectif de ce projet est également de pouvoir permettre le passage du monde de la réingénierie vers le monde de la modélisation. Ce chapitre traite de l'exportation de projets Smalltalk implémentés sous VisualWorks vers l'outil de modélisation EMF.

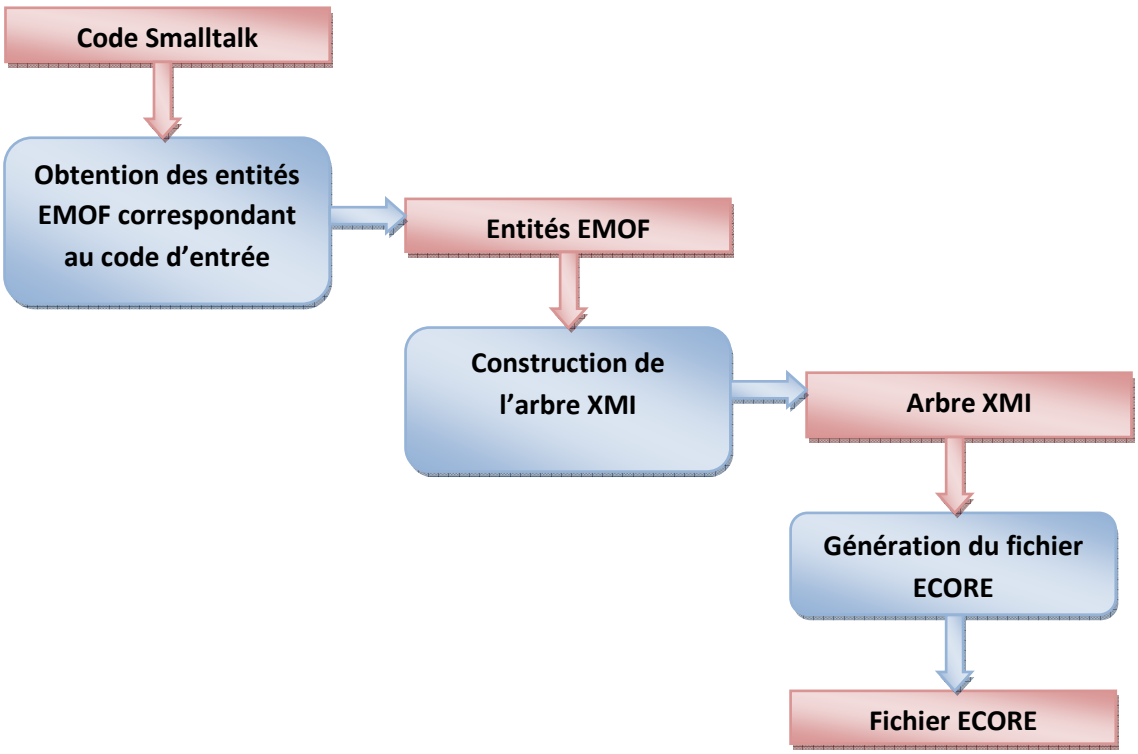
1. Analyse de problème

L'objectif de l'Exporteur est de constituer un fichier au format Ecore (XMI) à partir d'un code Smalltalk que l'on souhaite exporter. Pour atteindre cet objectif, trois étapes sont nécessaires :

La première étape consiste à obtenir les entités EMOF correspondant au code Smalltalk que l'on cherche à exporter (cf. chapitre 2, paragraphe 2.3).

La deuxième étape consiste construire un arbre XMI à partir des entités EMOF obtenues à l'étape précédente.

Enfin, la dernière étape doit permettre d'obtenir le fichier au format ECORE correspondant que code Smalltalk d'entrée. Ce fichier peut ensuite pris en compte par le plug-in EMF pour générer un modèle.



2. Analyse de l'existant

2.1 Les méta-descriptions

Grâce au package **Meta** implémenté sous Visual Works, il est possible d'obtenir le méta-modèle qui dérive d'un code source Smalltalk. En effet, pour obtenir les entités EMOF du méta-modèle, il suffit d'envoyer le message *asMetaDescription* aux namespaces ou aux classes du code Smalltalk qui le constituent. Chaque classe doit cependant mettre à disposition les méthodes de classes qui décrivent ses variables d'instances et ses méthodes (cf. chaptire 3 paragraphe 4.5.4) sans quoi le méta-modèle obtenu risque d'être incomplet. En effet, sans ces méta-descriptions, le méta-modèle obtenu ne fera pas apparaître les entités EMOF de type **Property** et **Operation**.

2.2 L'arbre XMI

2.2.1 Le Format XMI

Le langage UML a été pensé pour permettre la modélisation mais n'a pas été conçu dans un premier temps pour communiquer autrement que visuellement. Pour permettre la communication, il lui a été adjoint un formalisme de persistance: le langage XMI.
XML Metadata Interchange est un standard pour l'échange d'informations de métadonnées UML basé sur XML. Ce standard a été créé par l'OMG. Ce format standard permet d'exprimer les concepts de la modélisation objet. Pour transcrire les structures de graphe, très répandues dans ce domaine, on a recours à un mécanisme d'identifiants et de références à ces identifiants car un document XML a les caractéristiques d'un arbre. Il est alors possible d'encoder un modèle UML dans un fichier au format XMI.
Tous les modèles sont normalement disponibles en format XMI, mais la représentation sous forme de fichier texte fait que, pour la manipulation, les formats graphiques UML et les modèles Ecore d'EMF sont préférés. Le but de ces standards est de permettre à des ateliers logiciels d'explorer et d'échanger les définitions des structures de données, leurs propriétés, les relations les unissant, etc.

2.2.2 L'arbre XML implémenté dans Visual Works

Les documents XML sont basés sur des entités élémentaires: les **éléments** et les **attributs**. Pour créer un arbre XML, il faut définir ses éléments et ses attributs et leur assigner des types valides. Les éléments décrivent les données alors que les attributs définissent les éléments qui les contiennent. Un arbre XMI est constitué d'éléments XML dont les spécifications sont conformes au standard XMI
Dans VisualWorks, le package XML définit les classes qui permettent la mise en œuvre d'arbres XML et XMI.

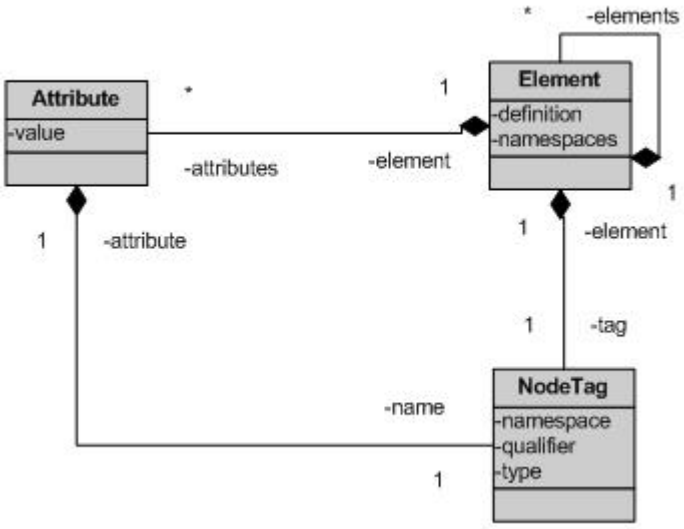


Fig 4-1 diagramme de classes associé à l'implémentation de l'arbre XML dans VW

La classe Element

Dans Visual Works, un arbre XML est constitué d'objets de type **Element**. Un **Element** peut en contenir d'autres. On distingue l'**Element** racine des autres nœuds de l'arbre. Les objets de type **Element** contiennent des données et encapsulent des sous-ensembles de données. Ils sont représentatifs des entités d'un modèle : packages, classes, références, attributs, méthodes.

La classe Attribute

Les **Attributes** apportent des informations complémentaires relatives à un **Element** auquel il appartient. Les **Attributes** sont utilisés pour contenir des informations simples comme des données fixes ou des valeurs par défaut par exemple.

La classe NodeTag

Les **NodeTags** permettent de caractériser le type des **Elements** et des **Attributes** qui constituent l'arbre.
En effet, d'une part, les **NodTags** spécifient le type de l'entité du modèle que l'**Element** représente (Tab 4.1).

Element	Type du NodeTag
Package racine	'EPackage'
Sous-package	'eSubPackage'
Classe ou Enumération	'eClassifiers'
Attribut ou Référence	'eStructuralFeatures'
Méthode	'eOperations'
Annotation	'eAnnotations'
Paramètre	'eParameters'

Tab 4-1 Relation entre Element et NodeTag

D’autre part, il spécifie le type d’information définie par un **Attribute** (Tab 4.2).

Attribute	Type du NodeTag
URI d’un package	‘nsURI’
Nom d’une entité	‘name’
Superclasse d’une Classe	‘eSuperTypes’
Type d’un attribut, d’une référence, d’une méthode ou d’un paramètre	‘eType’
Valeur par défaut d’une référence ou d’ un attribut	‘defaultValueLiteral’
Source d’un commentaire	‘source’

Tab 4-2 Relation entre Attribute et NodeTag

3.L’Exporteur

3.1 Un visiteur

La classe qui implémente l’Exporteur s’appelle **EcoreExporter**. Tout comme le générateur de code, cette classe est conforme au motif de conception « Visiteur ». Ce visiteur crée, pour chaque entité EMOF, un élément XMI qui lui correspond.

3.2 Les Packages

Une entité EMOF de type **Package** est dite ‘racine’ si elle n’est pas contenu par un autre package. Lorsqu’un package n’est pas racine, on dit qu’il s’agit d’un sous-package. Les projets implémentés en Smalltalk peuvent souvent contenir plusieurs packages racines.

Dans un modèle EMF cependant, il existe un unique package racine contenant plusieurs sous-packages. Concrètement, le package racine est associé à l’élément racine de l’arbre XML alors que les sous-packages correspondent à ses enfants.

Pour exporter du code Smalltalk vers EMF, il faut donc prendre en compte cette différence.

3.2.1 Le package racine

l’**Element** racine de l’arbre XML généré provient en réalité d’un package par défaut qui est crée à à l’initialisation de l’Exporteur. Ce package par défaut est en fait à une variable d’instance de l’Exporteur et s’appelle rootPackage. L’Exporteur donne à l’utilisateur la possibilité de nommer lui-même ce package par défaut. L’Element racine est donc généré de la façon suivante :
Tout d’abord, son **NodeTag** indique que l’Element généré est de type‘**EPackage’**.

Ensuite, sa variable d'instance **namespaces** est initialisée avec un dictionnaire contenant les trois espaces de nommage préfixés. En effet, dans un fichier XMI, l'élément racine d'un modèle EMF contient trois déclarations d'espaces de nommage**:**

- **'http://www.omg.org/XMI**' associé au préfixe **xmi**: ce namespace est un espace de nommage par défaut. On le trouve communément dans de nombreux fichiers XMI.

- **'http://www.w3.org/2001/XMLSchema-instance'** associé au préfixe **xsi**: ce namespace définit le schéma XML utilisé.

- **'http://www.eclipse.org/emf/2002/Ecore'** associé au préfixe **ecore**: ce namespace caractérise le méta-modèle auquel est conforme le modèle que représente le fichier XMI correspondant. Dans le cas d'un modèle EMF, c'est le méta-modèle Ecore qui est utilisé.

L'Exporteur initialise ensuite l'**Element** racine avec une collection de 4 **Attributes**:

- Le premier **Attribute** définit un URI. Dans de nombreux documents XMI, les éléments du modèle possèdent un i*dentifiant uniforme de ressource*, c'est-à-dire une courte chaîne de caractères identifiant une ressource Web physique ou abstraite, et dont la syntaxe respecte une norme d'Internet mise en place pour le World Wide Web. Dans un fichier XMI, un URI sert à identifier les namespaces. Quelquefois en effet, un problème peut apparait si on mélange deux documents XML dont les éléments ont le même nom mais pas la même définition. Pour éviter ce conflit de noms, on enrichit le nom de l'élément avec un espace de nommage. Chaque rédacteur de document XMI doit choisir l'espace de noms dans lequel il va créer ses éléments. L'Exporteur propose à l'utilisateur de nommer lui-même l'URI de l'élément racine du fichier XMI qu'il souhaite constituer.

- Le second **Attribute** définit un préfixe. Pour faciliter plusieurs utilisations d'un espace de noms, on peut lui associer un surnom appelé le **préfixe**. Contrairement à la définition de l'espace de noms par défaut, le mécanisme des préfixes ne porte que sur l'élément courant. L'Exporter donne au préfixe de l'**Element** racine le nom du package par défaut dont il provient.

- Le troisième **Attribute** définit la version du langage XMI utilisée. En effet, tous les fichiers XMI précisent au début la version du langage qu'ils utilisent. C'est d'ailleurs pourquoi cette information est révélée dans l'**Element** racine de l'arbre. EMF utilise la version 2.0 du langage XMI.

- Enfin, un dernier **Attribute** permet de préciser le nom du package auquel l'**Element** XMI correspond.

L'**Element** racine ainsi généré pourra alors contenir d'autres **Elements** associés aux sous-packages, aux classes et aux énumérations du package racine qu'il représente.

3.2.2 Les sous-packages

Lorsque l'Exporteur rencontre un package EMOF racine, il la transforme en sous-package dont le parent est le package par défaut rootPackage précédement décrit. Il traite donc cette entité EMOF comme un sous-package.

La création d'un **Element** à partir d'un sous package EMOF est très simple.
Tout d'abord, son **NodeTag** indique que l'Element est de type '**eSubPackages**'.
Ensuite, l'**Element** est initialisé avec un unique attribut qui spécifie le nom du sous-package qu'il représente.Les Classes et les Enumérations

L'Exporteur crée, à partir d'une classe ou d'une énumération, un **Element** dont le **NodeTag** indique que l'entité EMOF qu'il représente est de type '**eClassifiers'**.
D'autre part, l'**Element** est initialisé avec deux principaux **Attributes** :

- Le premier **Attribute** spécifie le nom de classe ou de l'énumération que l'**Element** symbolise.

- Le deuxième **Attribute** indique si l'**Element** est associé à une classe ou à une énumération. En effet, s'il s'agit d'une classe, l'**Attribute** précise alors que l'eClassifiers est de type EClass. En revanche, s'il s'agit d'une énumération, l'Attribut révèle que l'eClassifiers est de type EEnum.

- Dans le cas d'une classe, un troisième **Attribute** se rajoute pour indiquer le nom de la superclasse dont elle provient.

L'**Element** ainsi crée pourra contenir les **Elements** associés aux variables d'instances et aux méthodes de la classe qu'il représente.

3.3 Les Propriétés

Dans le méta-modèle EMOF, une entité de type **Class** est composée d'un ou plusieurs objets de type **Property**. Les Properties représentent en fait les variables d'instance d'une classe donnée.

L'Exporteur crée, à partir d'une **Property**, un **Element** dont le **NodeTag** indique que qu'il s'agitu d'un **eStructuralFeatures**'.

De plus, l'**Element** est initialisé par le biais de 6 **Attributes** :

- Le premier **Attribute** spécifie le nom de la **Property** auquel l'**Element** est associé.

- Ensuite, deux autres **Attributes** viennent souligner la cardinalité de la **Property** : ‘lower’ pour la borne inférieur et ‘upper’ pour la borne supérieure.

- En outre, le quatrième **Attribute** indique le type d’eStructuralFeatures auquel on à faire. En effet, le méta-modèle Ecore fait la distinction entre attribut et référence (cf. chapitre 3 paragraphe 4.3). Dans un fichier XMI, un **Attribute** peut donc spécifier si un Element représente un attribut (il s’agit d’un EAttribute) ou une référence (il s’agit d’un EReference). Dans le méta-modèle EMOF, cette distinction n’existe pas. Pourtant, l’Exporteur traite le problème de la manière suivante : lorsqu’une entité **Property** est de type primitif (**PrimitifType**) ou dérive d’une énumération, l’**Element** généré est un attribut. Par contre, lorsqu’une **Property** dérive d’une classe, l’**Elemen**t associé est une référence.

- Le cinquième **Attribute** désigne le type de donnée de la **Property**.

- Enfin, le sixième **Attribute** associe à l’**Element** la valeur par défaut de la **Property** qu’il représente.

3.4 Les Littéraux

Dans le méta-modèle EMOF, une entité de type **Enumeration** est composée d’un ou plusieurs objets de type **Literal**. Les littéraux sont associés à une valeur entière.
L’Exporteur crée, à partir d’une entité de type **Literal**, un **Element** de type ‘**eLiterals**’.
L’**Element** est ensuite initialisé avec 2 **Attributes** :

- Le premier **Attribute** spécifie le nom du littéral que l’**Element** représente.
- Le second **Attribute** indique la valeur qui est associé au littéral.

3.5 Les Opérations

Dans le méta-modèle EMOF, une classe peut être composée de plusieurs entités de type **Operation**. Les opérations sont en fait les méthodes d’une classe donnée.
L’Exporteur crée, à partir d’une **Operation**, un **Element** de type ‘**eOperations**’.
Ensuite, l’**Element** est initialisé avec deux **Attributes** :

- Le premier Attribute inscrit dans l’arbre le nom de l’opération que l’**Element** symbolise.
- Le second précise le type de données renvoyé par l’opération.

L’**Element** crée pourra ensuite contenir les **Elements** associés à ses paramètres

3.6 Les Paramètres

L'Exporteur crée, à partir d'une entité de type **Parameter**, un **Element** dont le **NodeTag** indique qu'il s'agit d'un **Element** de type '**eParameters**'.
Ensuite, L'**Element** est initialisé avec deux **Attributes** :

- Le premier lui associé le nom du paramètre.
- Le second précise le type de données du paramètre.

3.7 Génération du fichier XMI

Une fois l'arbre XMI obtenu, il est possible de le convertir en chaîne de caractère en appliquant le message *printString* à l'Element racine. Cette chaîne de caractère est ensuite enregistrée dans un fichier .ecore. Le fichier ainsi obtenue possède toutes les caractéristiques propres à un fichier XMI. Il peut donc enfin être pris en compte par l'outil EMF

4. Critiques et améliorations possibles

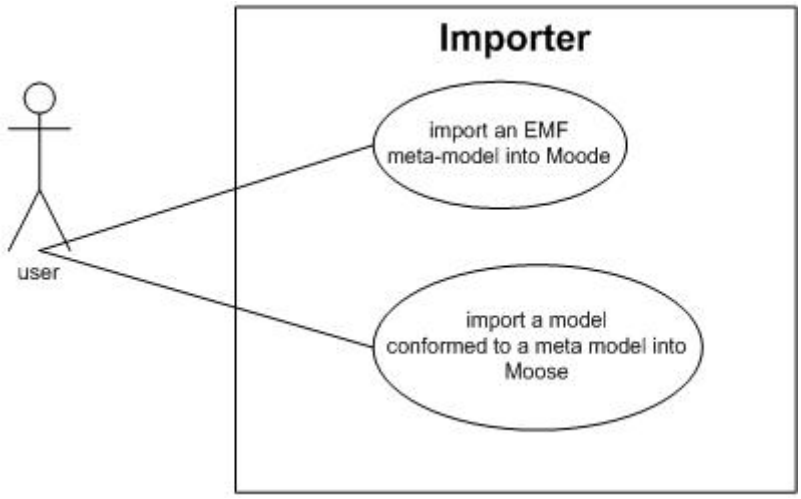
Le travail que j'ai réalisé sur l'exportation mériterait un certain nombre d'améliorations. L'Exporteur que j'ai implémenté reste en effet incomplet.
D'une part, je n'ai pas eu le temps de gérer convenablement les types de données associés aux entités de type **Property**, **Operation** et **Parameter**. Pour l'instant, les attributs, les références, les méthodes et les paramètres générées dans EMF possèdent des types de données indéfinies.
D'autre part, les entités EMOF de type **Comment** n'ont pas encore été traitées. Ces entités auraient dû être utilisées pour générer des Annotations dans le modèle EMF. Cependant, j'ai rencontré quelques difficultés à gérer ce type d'entité et je n'ai pas eu le temps de résoudre ces problèmes.
Cet Exporteur est toutefois capable de gérer un bon nombre d'entités EMOF et parvient à produire un modèle EMF convenable à partir d'un code source Smalltalk. Cet Exporteur constitue donc une bonne base pour des améliorations futures.

Conclusion

Annexes

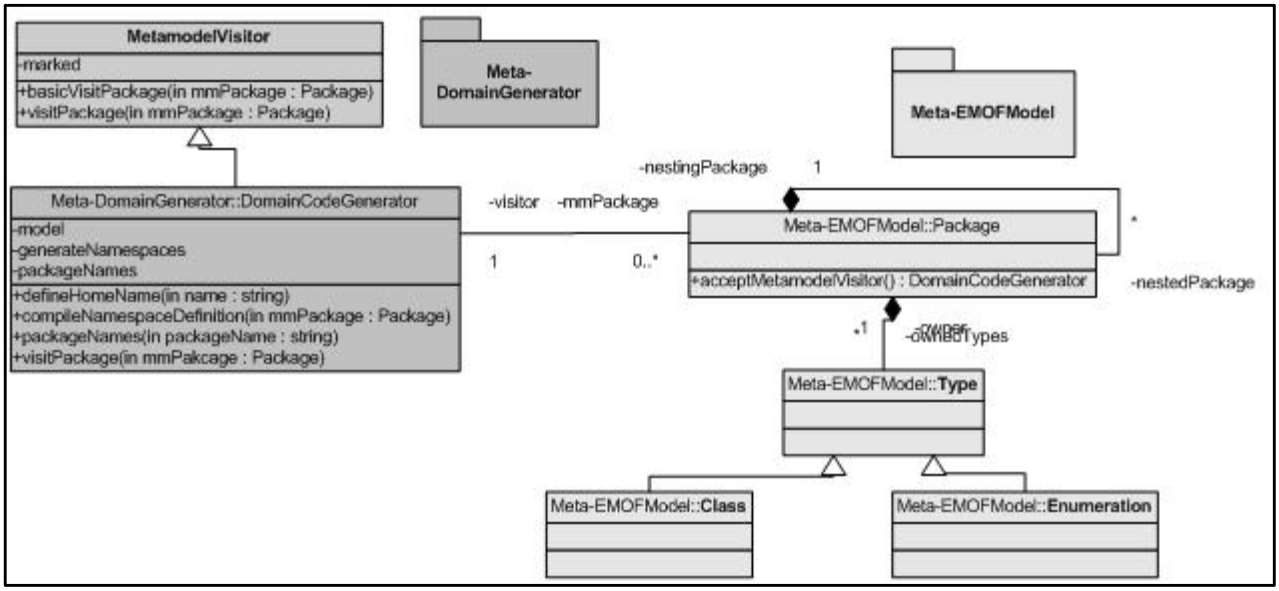
1. Modélisation du Générateur de code

1.1 Cas d'utilisation

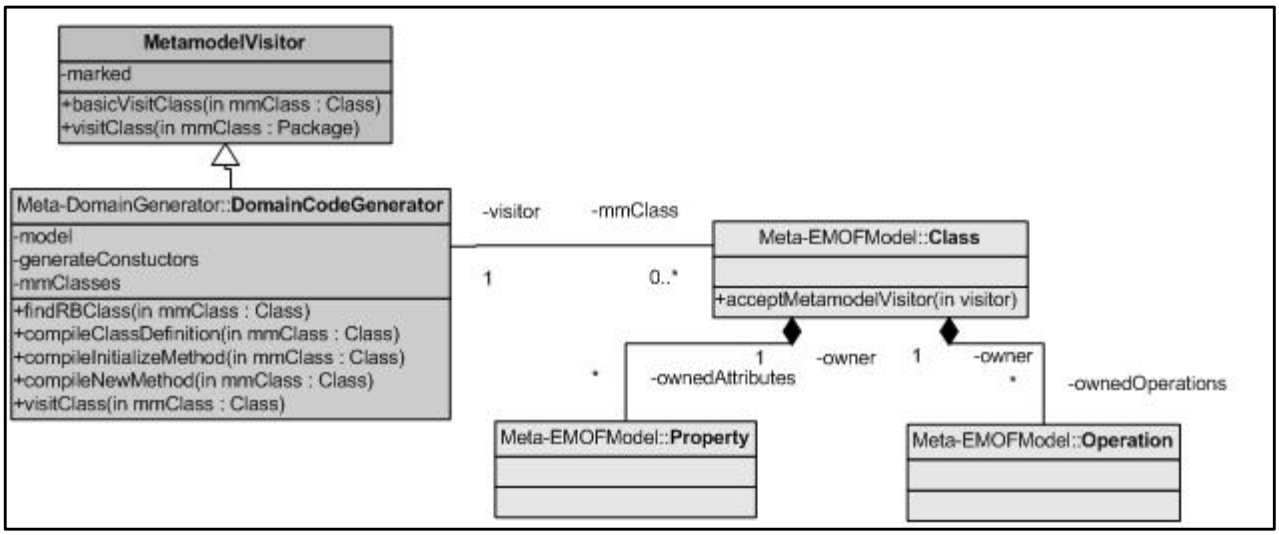


1.2 Diagrammes de classes

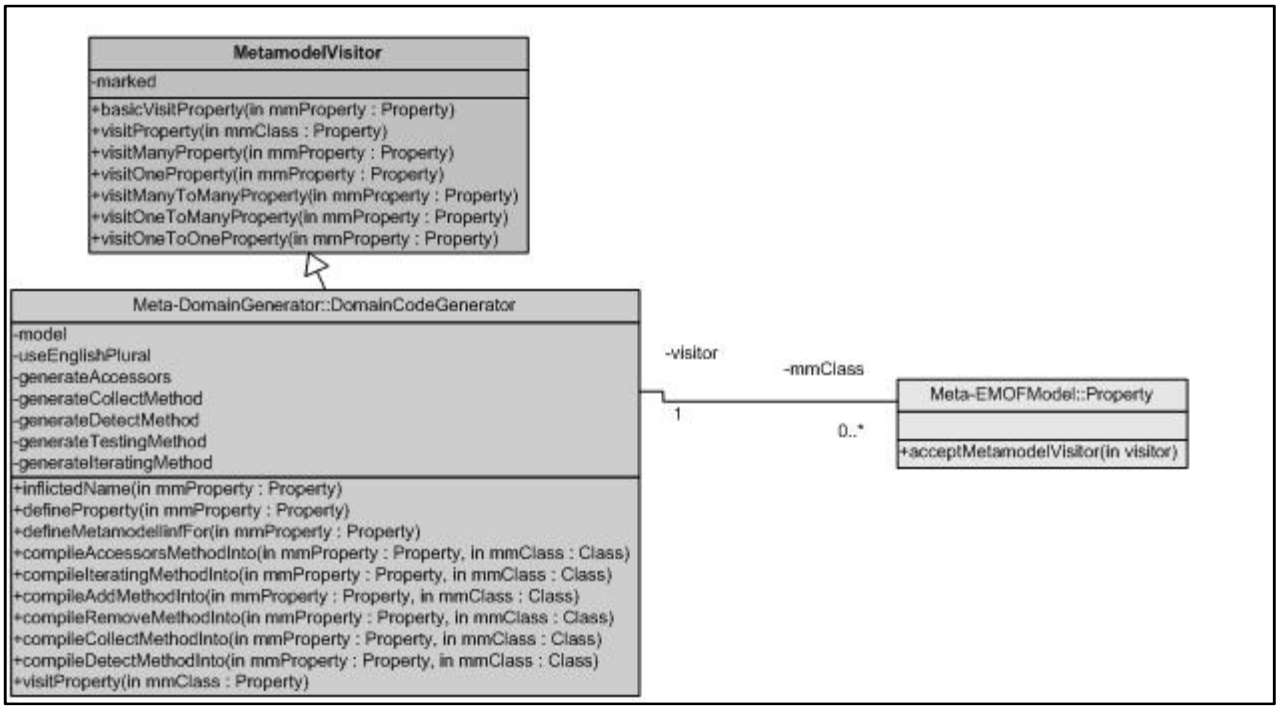
1.2.1 Génération des packages



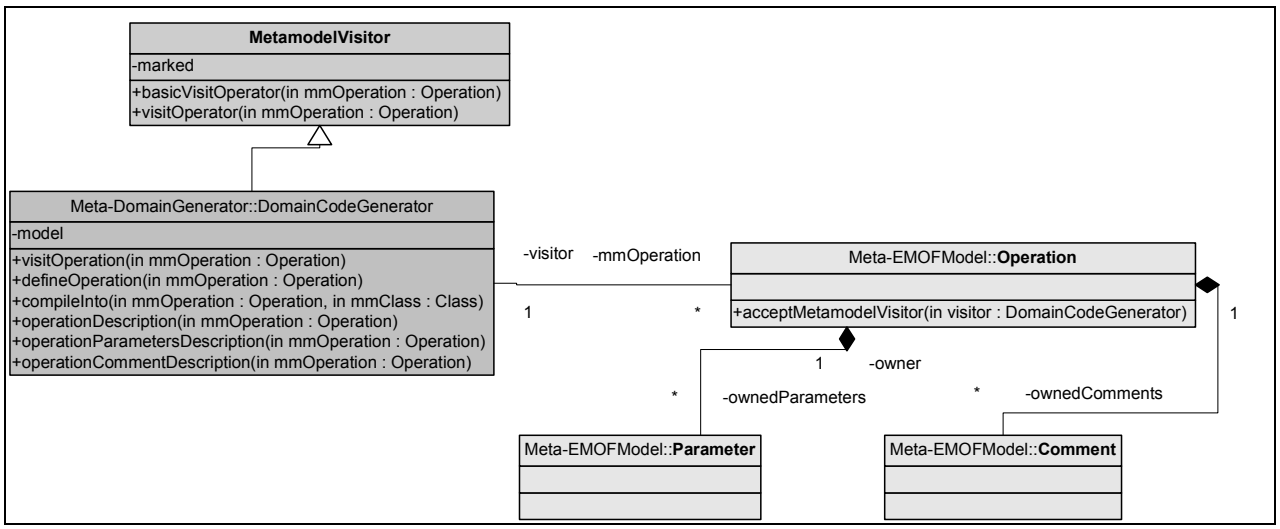
1.2.2 Génération des classes



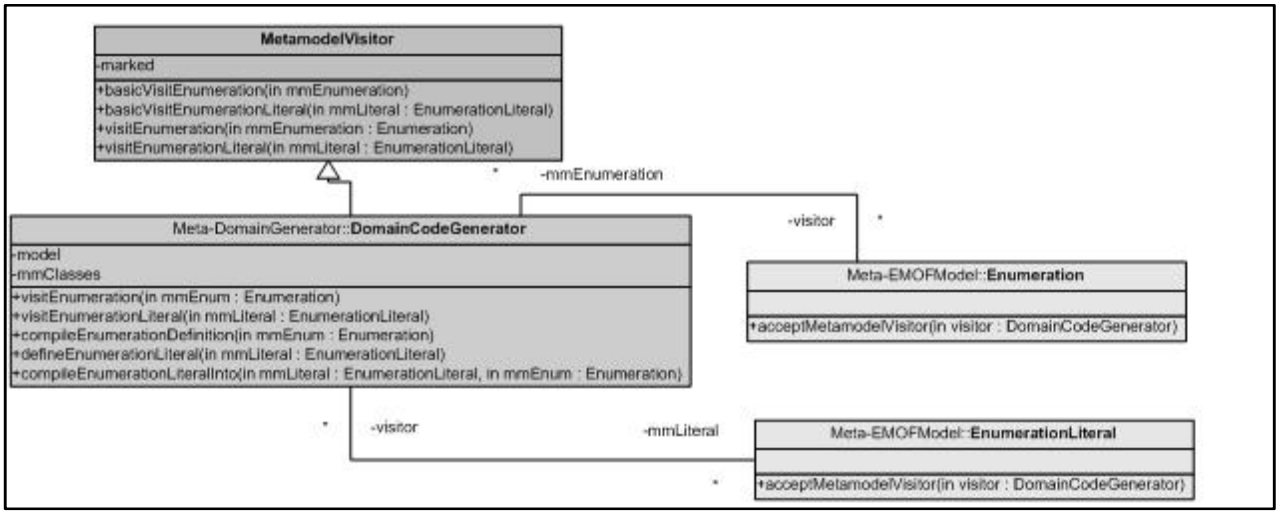
1.2.3 Génération des variables d'instances



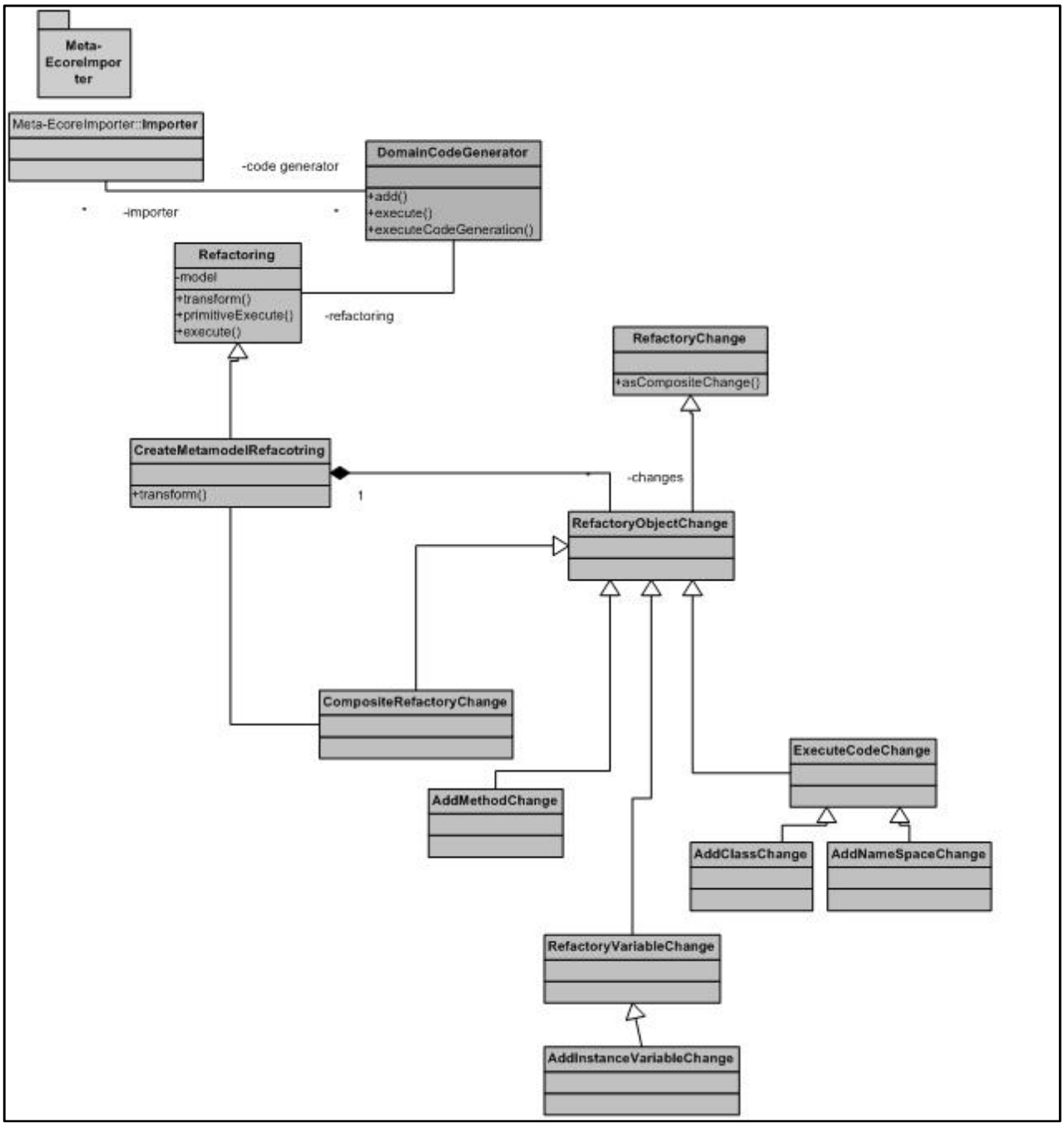
1.2.4 Génération des Opérations



1.2.5 Génération des énumérations et des litéraux

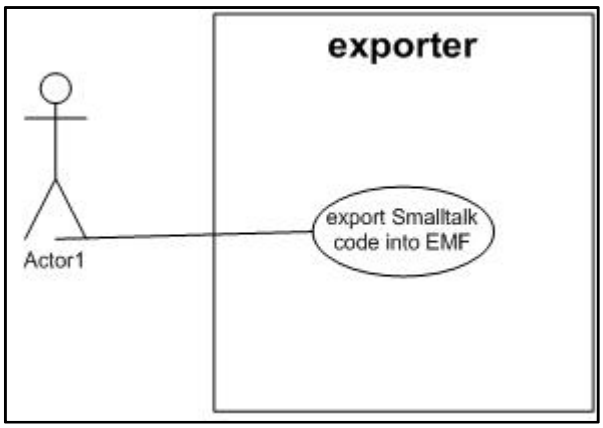


1.2.6 Génération de code Smalltalk



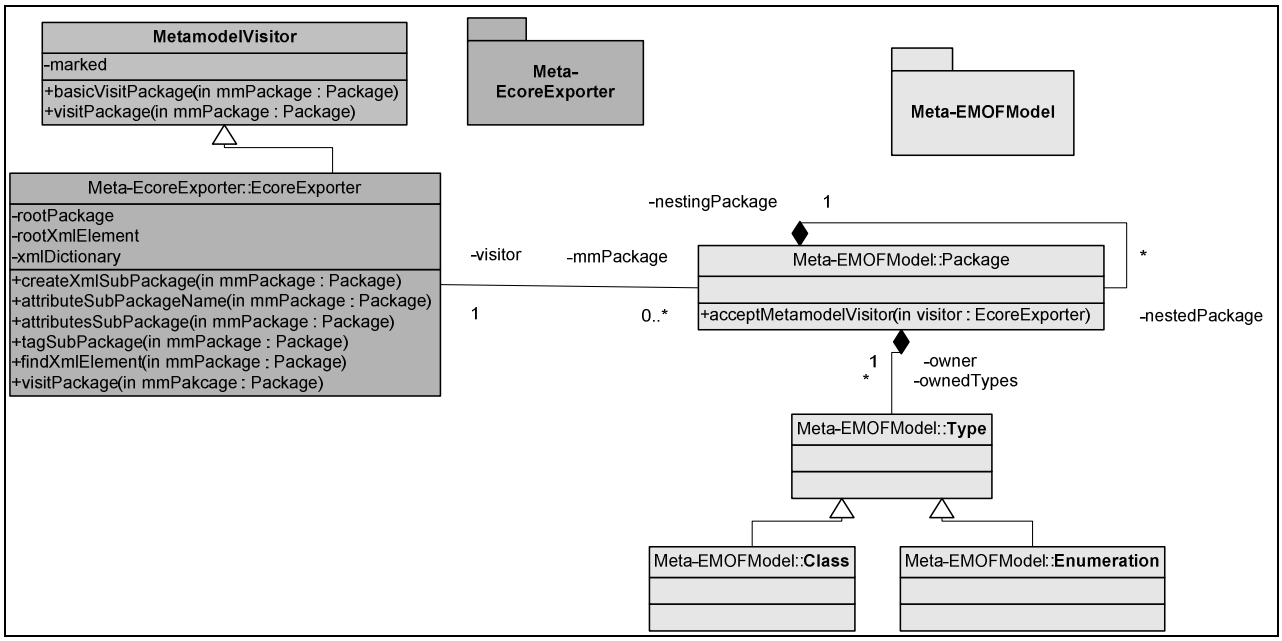
2. Modélisation de l'Exporteur

2.1 Cas d'utilisation

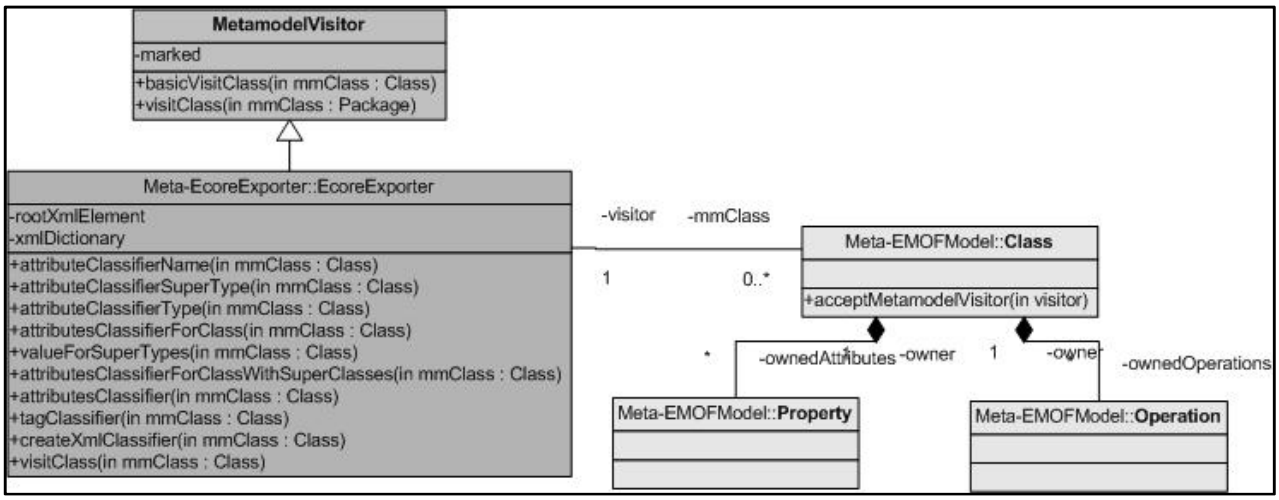


2.2 Diagrammes de classes

2.2.1 Les packages

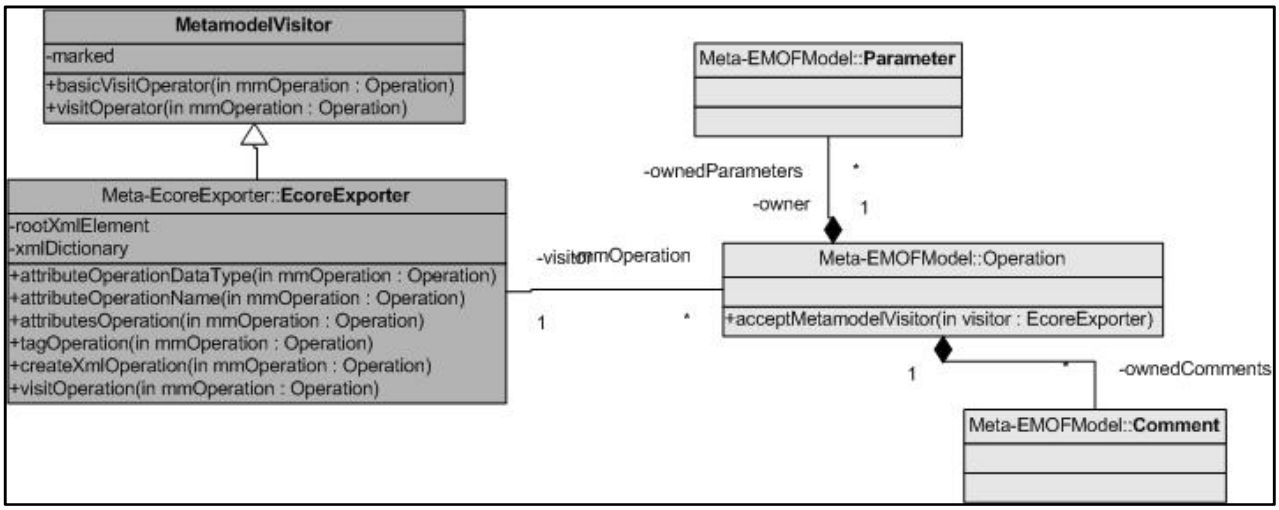


2.2.2 Les classes

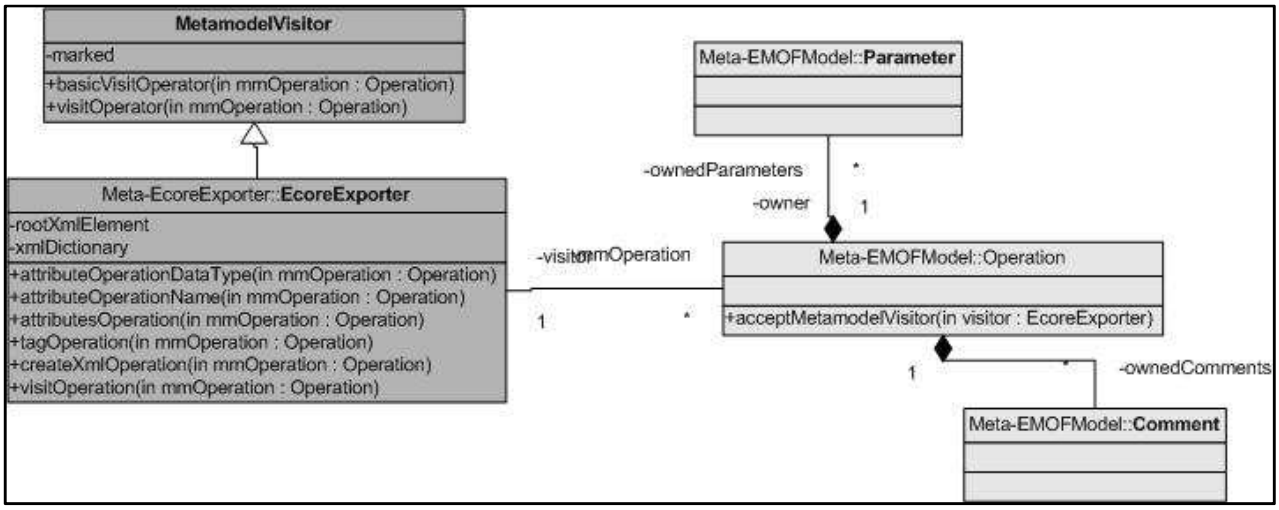


2.2.3 Les variables d'instances

2.2.4 Les Opérations



2.2.5 Les paramètres



2.2.6 Les énumérations et les litéraux