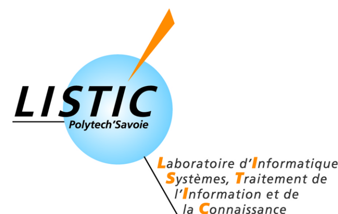
École ENSIETA
Laboratoire Listic, Ecole Polytech' Savoie
Lieu Annecy, France
Maître de stage Stéphane Ducasse
Tuteur Joel Champeau

Rapport de Projet de Fin d'Etudes

Import de métamodèles et de modèles conformes depuis EMF vers un outil de visualisation

BOUAZZA Pierrick

Avril à Août 2007



Remerciements

Tout d'abord je tiens à remercier mon maître de stage : Monsieur Stéphane DUCASSE pour sa disponibilité, ses conseils, ses critiques et sa patience.

Je suis également très reconnaissant à Monsieur Philippe BOTON, directeur du LISTIC, de m'avoir accepté dans son laboratoire pour ce stage.

Je remercie également l'ensemble des personnels du LISTIC et plus particulièrement l'équipe LS (notamment Damien Cassou, Damien Pollet et Mathieu Suen avec lesquels j'ai passé la plus grande partie de mon stage) pour leur accueil chaleureux et l'aide qu'ils m'ont apporté tout au long de mon stage.

Je tiens aussi à souligner l'aide que m'a apportée Mlle Sara SELLOS pendant la durée de ce stage.

Enfin, je remercie Monsieur Joel CHAMPEAU pour le soutien et le suivi qu'il nous a fourni au cours des cinq mois qu'ont duré ce stage.

Résumé

Ce document présente le travail que j'ai réalisé lors de mon stage de fin d'étude. Ce stage s'est déroulé d'avril à août 2007 sous la responsabilité de M. Stéphane DUCASSE dans le laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance de l'école Polytech' Savoie à Annecy.

Le sujet de ce stage était la réalisation d'une passerelle entre l'environnement de développement Eclipse (et en particulier son plugin dédié à la métamodèlisation EMF) et un outil de re-engineering nommé Moose et codé en Smalltalk. Dans un premier temps, j'ai réalisé un importeur de métamodèles qui, en collaboration avec le générateur code de Mlle SELLOS, permet soit d'éditer un métamodèle réalisé avec EMF dans l'environnement de développement en Smalltalk VisualWorks, soit d'utiliser les outils de visualisation intégrés à Moose pour mieux comprendre ce métamodèle. Ensuite, j'ai réalisé un deuxième importeur destiné aux modèles conformes à un métamodèle personnel. Cependant, cet importeur ne pourra générer que des élément destinés à être visualisé dans Moose : en effet, en tant qu'instances des entités crées par mon premier importeur ces entités n'ont pas de code associé.

Table des matières

1. Introduction 6
1.1. Le Listic. 6
1.1.1. Présentation 6
1.1.2. Organisation 6
1.2. Eclipse et EMF. 7
1.2.1. Eclipse. 7
1.2.2. EMF. 8
1.3. Moose. 9

2. Le projet 12
2.1. Intérêt 12
2.2. Conduite du projet 12

3. L'importeur de métamodèle 16
3.1. Étude préliminaire 16
3.2. Réalisation 18
3.2.1. Processus de développement 18
3.2.2. L'architecture 20
3.2.3. Le développement 22

4. L'importeur de modèles conformes 28
4.1. Étude préliminaire 28
4.2. Réalisation 30
4.2.1. L'architecture 30
4.2.2. Le développement 30

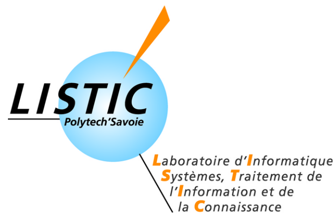
5. Conclusion 33
5.1. Apports 33
5.2. Perspectives 33

A. Syntaxe Smalltalk 35

1. Introduction

1.1. Le Listic.

1.1.1. Présentation



Le Laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance (LISTIC) de l'Ecole Polytech' Savoie est un laboratoire d'informatique dont l'activité est centrée sur la fusion des systèmes d'information. Ses principaux champs d'investigation sont :

les systèmes complexes et l'instrumentation intelligente Instrumentation d'un gant numérique permettant de commander les mouvements d'un robot grâce aux mouvements de la main.

le traitement d'image Détection automatiques des changements de plan dans les films d'animation.

les logiciels complexes Ré-ingénierie d'applications complexes.

la gestion des connaissances et des compétences Modélisation des connaissances et des systèmes, de leurs fondements à leurs applications.

1.1.2. Organisation

le laboratoire se structure en trois équipes :

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

Ingénierie de la Connaissance. Les recherches de l'équipe IC portent sur la modélisation des connaissances et des systèmes, de leurs fondements à leurs applications ;

Logiciels et Systèmes. Chargée de la conception, de la réalisation et du déploiement des systèmes de fusion (et plus particulièrement les aspects liés à la décentralisation), l'équipe LS intègre essentiellement les axes systèmes et répartition d'une part, logiciels et systèmes évolutifs d'autre part.

L'équipe LS C'est au sein de cette équipe que j'ai réalisé mon stage. Elle est divisée en deux entités :

Systèmes et Distribution. Cette équipe conduit principalement ses recherches dans les domaines du contrôle intelligent des systèmes industriels, des systèmes dynamiques à événements discrets et des systèmes distribués.

Language et Evolution des Logiciels. Cette équipe travaille sur l'architecture et le design logiciels et la ré-ingénirie. Elle est dirigée par le professeur Stéphane DUCASSE qui fut mon maître de stage et accueille de nombreux stagiaires et thésards qui ont pu m'apporter leurs conseils et leur aide tout au long de mon stage.

1.2. Eclipse et EMF.

1.2.1. Eclipse.

Eclipse est un environnement de développement intégré (Integrated Development Environment) dont le but est de fournir une plate-forme modulaire pour permettre de réaliser des développements informatiques.

I.B.M. est à l'origine du développement d'Eclipse qui est d'ailleurs toujours le coeur de son outil Websphere Studio Workbench (WSW), lui même à la base de la famille des derniers outils de développement en Java d'I.B.M. Tout le code d'Eclipse a été donné à la communauté par I.B.M afin de poursuivre son développement.

Eclipse utilise énormément le concept de modules nommés "plug-ins" dans son architecture. D'ailleurs, hormis le noyau de la plate-forme nommé "Runtime", tout le reste de la plate-forme est développé sous la forme de plug-ins. Ce concept permet de fournir un mécanisme pour l'extension de la plate-forme et ainsi fournir la possiblité à des tiers de développer des fonctionnalités qui ne sont pas fournies en standard par Eclipse.

Les principaux modules fournis en standard avec Eclipse concernent Java mais des modules ont été développé (ou sont en cours de développement) pour d'autres langages notamment C++, Cobol, mais aussi pour d'autres aspects du développement (base de données, conception avec UML, ...). Ils sont tous développés en Java soit par le projet Eclipse soit par des tiers commerciaux ou en open source.

Bien que développé en Java, les performances à l'exécution d'Eclipse sont très bonnes car il n'utilise pas Swing pour l'interface homme-machine mais un toolkit particulier nommé SWT associé à la bibliothèque JFace. SWT (Standard Widget Toolkit) est développé en Java par IBM en utilisant au maximum les composants natifs fournis par le système d'exploitation sous jacent. JFace utilise SWT et propose une API pour faciliter le développement d'interfaces graphiques.

C'est cette modularité, la liberté d'extension ainsi que sa simplicité d'utilisation couplé avec l'essor de JAVA qui ont fait qu'Eclipse est l'un des IDE le plus couramment utilisé dans le milieu professionnel.

1.2.2. EMF.

EMF est constitué trois parties :

EMF. Le coeur du framework EMF inclut un métamodèle (Ecore : figure 1.1) 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.

Les éléments générés se présente sous la forme d'un plugin pour Eclipse, qui une fois chargé va permettre d'utiliser l'éditeur réfléxif disponible dans EMF pour générer des modèles conformes au métamodèle de notre choix : au moment d'ajouter un fils à un élément (en cliquant droit sur cet élément) l'éditeur proposera uniquement les choix définis dans le métamodèle (à la place des habituels EPackage, EClass, etc).

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 vient 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.

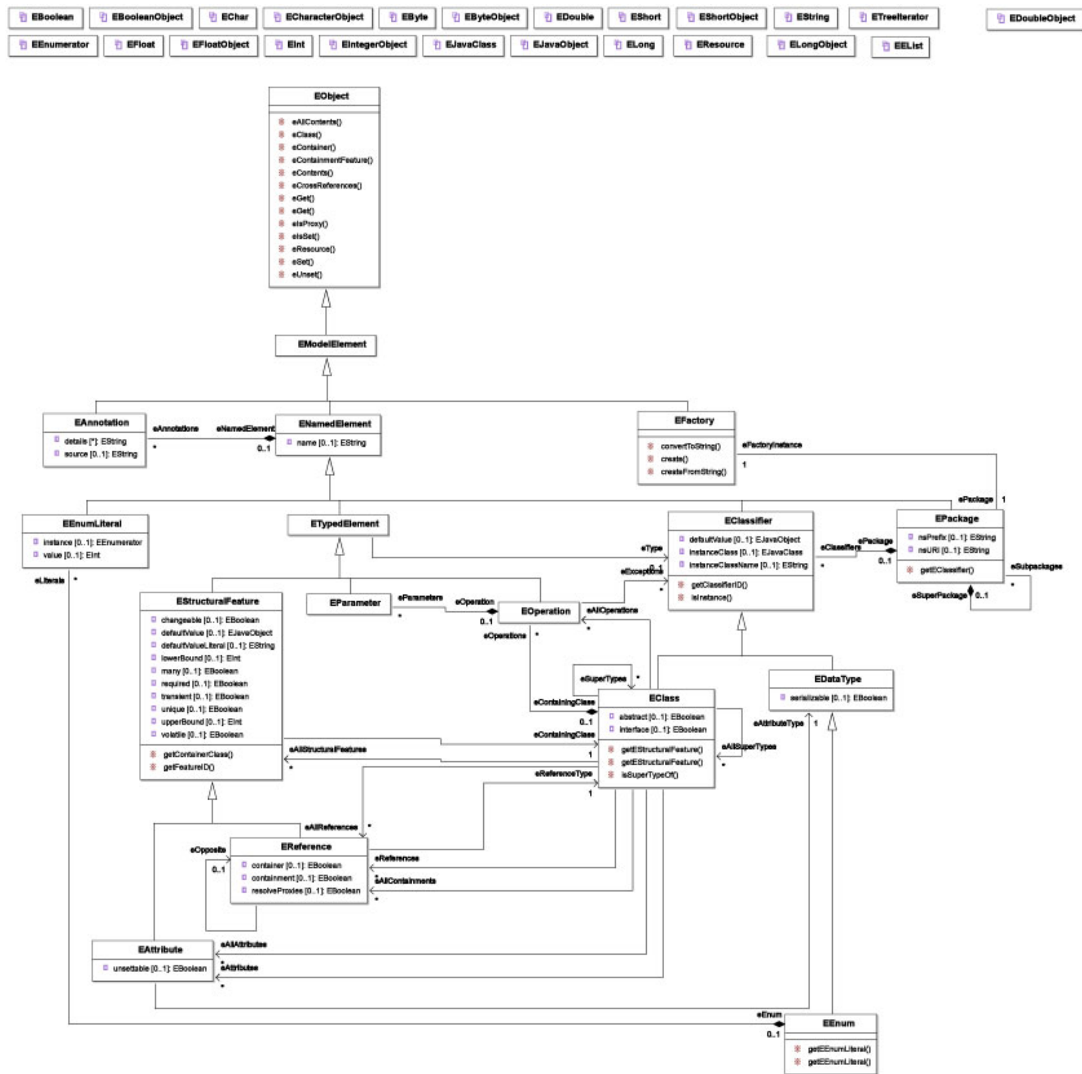
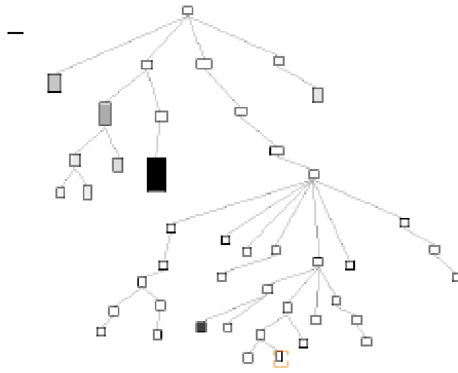


FIG. 1.1.: Le métamodèle ECORE

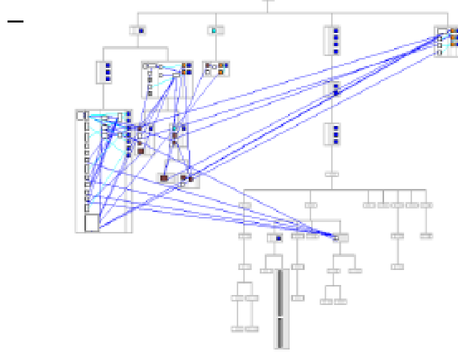
1.3. Moose.

Moose est un environnement de visualisation open-source qui propose un panel d'outils pour faciliter l'analyse de logiciels complexes. Ces visualisations permettent d'analyser beaucoup plus rapidement un programme auquel ont souhaite appliquer un processus de refactoring. Il peut également être utilisé pendant le développement d'un programme afin d'identifier au plus tôt les différents problèmes de design et pouvoir les corriger et ainsi de limiter leur impact sur le développement futur.

Parmi les visualisations disponibles, on trouve notamment :

– 

Le diagramme de complexité du système (System complexity) :
Cette vue ressemble au premier abord a un diagramme UML simplifié (on ne voit pas ni les attribut, ni les méthode). On y retrouve en effet la hiérarchie du code étudiés. Cependant, les caractéristiques (dimensions, couleur) du rectangle représentant chaque classe sont définies par des métriques calculées a partir de la classe elle-même (par exemple le nombre de ligne de code, le nombre de méthode, etc). Elle permets donc de repérer rapidement les classes importantes du système étudié : elles seront plus longues, plus larges, plus foncées ou un mélanges de ces caractéristiques suivant les métriques associées a chaque caractéristique.

– 

Le modèle des classes (Class blueprint)
Cette vue va présenter la structure interne des classes. Chaque grand rectangle représente une classe tandis que les plus petits représentent soit les méthodes (dans la partie gauche de chaque classe), soit les attributs (dans la partie droite). Ici, la taille des rectangles représentant les méthodes est calculée à partir de métriques. Leur couleur a aussi une signification particulière (par exemple une méthode dont le rectangle est représenté en orange accède à des données a l'extérieur de la classe à laquelle elle appartient). Les liens entres les classes et les attributs correspondent aux invocations et aux accès. Cette vue permet de comprendre le fonctionnement d'un système, notamment en identifiant rapidement quels sont les objets les plus référencés ou faisant le plus de références.

De nombreuses autres vues sont également disponibles notamment pour suivre l'évolution d'un système au cours du temps, ou encore étudier la duplication de code au sein du système.

De même que EMF, Moose dispose d'une implémentation d'un métamodèle. Les développeurs du projet ont choisi EMOF (figure 1.2). Contrairement à Ecore qui a été créé spécialement pour EMF, EMOF a été défini par l'Object Management Group. Cependant, pour correspondre mieux au besoin des programmes développés en SmallTalk, certaines légères modifications ont été apportées. Par exemple, les énumérations et leurs littéraux n'étaient pas implémentés à l'origine, alors qu'elles apparaissent dans les spécifications de l'OMG. J'ai été amené, dans le cadre de mon projet a intégrer ces éléments à l'implémentation utilisée par Moose.

Plan. Au chapitre 2, je décrirais le déroulement de mon stage. Puis on verra comment a été réalisé l'importeur de métamodèle au chapitre 3. Le chapitre 4 montrera

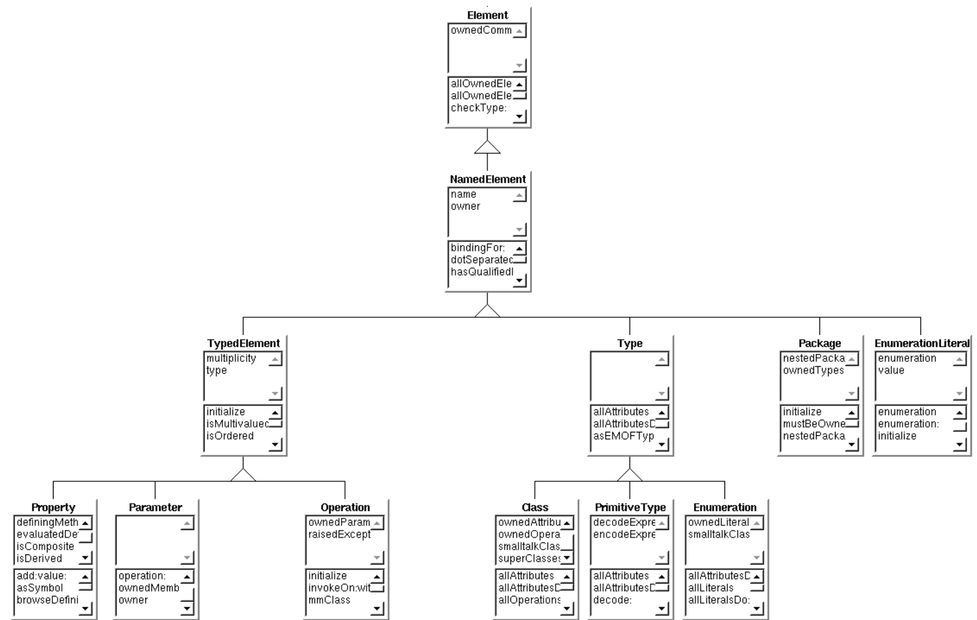


FIG. 1.2.: Le métamodèle EMOF implanté dans MOOSE

ensuite la réalisation de l'importeur de modèles conformes. Enfin au chapitre 5 je présenterais les possibilités d'évolutions du programme.

2. Le projet

2.1. Intérêt

Aujourd'hui, le processus de développement dirigé par l'architecture (Model Driven Architecture) est de plus en plus couramment utilisé. Ceci tend a généraliser l'utilisation de la métamodèlisation et la modélisation.

De plus, l'utilisation du langage JAVA est elle aussi de plus en plus répandu. L'outil Eclipse étant déjà très utilisé, il n'a pas tardé a se voir doté d'un plugin permettant la métamodèlisation. Avec ce plugin, EMF (Eclipse Modeling Framework), l'utilisation d'Eclipse a encore augmenté. De ce fait, une grande partie des développements actuels commencent par une phase de modélisation sous EMF. Il était donc important de doter l'outil Moose d'un module permettant d'analyser tous les modèles réalisés dans cet environnement afin de le rendre plus universel.

Il est également intéressant pour une personne ayant a appréhender un métamodèle ou un modèle qu'il n'a pas conçu (soit parce qu'il vient d'intégrer un projet, ou dans le cadre d'un procédé de re-engineering) de pouvoir profiter des visualisations disponibles dans Moose pour pouvoir comprendre mieux et plus rapidement le fonctionnement du code qu'il vient de recevoir. En effet, il n'est pas toujours simple de comprendre les liens qui unissent certaines entités simplement en analysant le code. Une visualisation appropriée peut faire gagner beaucoup de temps et éviter de nombreuses erreurs.

Une autre utilisation possible de cet importeur, est pour une personne disposant d'un métamodèle réalisé grâce à EMF, mais souhaitant poursuivre le développement en Smalltalk.

2.2. Conduite du projet

Comme le montre le diagramme 2.1, le premier mois a été consacré a de la bibliographie ainsi qu'à une étude de l'existant afin d'envisager les différentes approches possibles pour notre projet. C'est aussi pendant cette période que Sara SELLOS et moi nous sommes partagé le travail à réalisé. En effet, après en avoir parlé avec notre maître de stage, nous avons fixé comme priorité la réalisation de l'importeur de métamodèles, ainsi que son architecture générale :

une transformation de modèle Pour passer du métamodèle Ecore au métamodèle EMOF implanté dans MOOSE. C'est cette partie que j'ai réalisé dans un premier temps.

de la génération de code Pour rendre le métamodèle obtenu lors de la transformation de modèles utilisable. C'est la partie dont s'est chargée Sara SELLOS.

La réalisation de cet importeur a duré jusqu'à la mi-juin et sera détaillée au chapitre 3. La fin de cette période a coïncidé avec la prise de contact avec monsieur Jean-Rémy FALLERY du LIRMM (Laboratoire d'Informatique, de Robotique et de Micro-électronique de Montpellier) qui nous a fourni quelques métamodèles afin que nous puissions tester la validité de notre code. Nous avons également cherché à obtenir d'autres métamodèles auprès des enseignants de l'ENSIETA afin de vérifier le plus possible le bon fonctionnement de notre importeur. Des tests qui ont donc été menés pendant environ deux semaines. Ils m'ont permis d'identifier un certain nombre d'erreurs de conception et des cas particuliers que je n'avais pas envisagés. J'ai donc commencer à rectifier ceci au fur et à de leur découvertes. Corriger la plupart des erreurs ainsi découvertes m'a pris jusqu'au début du mois de juillet.

Après avoir fini mon premier importeur, nous nous sommes interrogés sur la poursuite du projet. EMF offrant la possibilité de générer des modèles conformes à un méta modèle particulier, il a tout naturellement convenu que je devrais m'intéresser à l'importation de ces modèles. J'ai commencer à réfléchir au design de cet importeur pendant la période de tests que j'ai conduit sur le premier importeur. J'ai également étudier plus en détails la structure des modèles générés, car je me suis rendu compte qu'elle était différente de celle des métamodèles. J'ai commencé à réaliser ce second importeur en même temps que je finissais de tester le premier et cela m'a pris environ 1 mois. Comme pour le premier importeur, j'ai ensuite testé celui ci, et je me suis aperçu qu'il restait quelques erreurs de conception du à une mauvaise compréhension de la structure des fichiers correspondant aux modèles conforme. De ce fait, la correction de cette erreur fut un peu plus longue, et l'importeur de modèles conformes ne fut donc disponible que début août.

Une fois cet importeur terminé, je me suis occupé d'un cas particulier du premier que j'avais laissé de coté lors de la phase de tests : en effet, certains fichiers de M. FALLERY comportaient des références 'croisées' : un fichier A comportait des références à des éléments d'un fichier B. Ce cas particulier m'a posé de nombreux problèmes c'est pourquoi j'y ai consacré le début du mois d'août.

Début juillet j'ai également commencé à rédiger mon rapport. J'ai profité du fait que certains stagiaires utilisaient couramment LateX pour apprendre moi aussi à utiliser cet environnement. En effet, tout au long de ma scolarité a l'ENSIETA, j'avais eut envie de profiter de la puissance de ce langage mais je n'avais jamais trouvé le temps nécessaire pour me lancer dans son apprentissage. De plus, le début est très déroutant par rapport aux logiciels de traitement de texte qui sont habituellement utilisés. Heureusement, les autres stagiaires m'ont grandement facilité la prise en main, notamment en m'aidant à configurer Emacs pour une utiliation très intuitive. J'ai terminé mon rapport pendant la fin du mois d'août (ceci fut d'ailleurs ma principale activité pendant cette période).

Enfin, la dernière semaine d'août, j'assisterais à la conférence ESUG (European Smalltalk User Group)qui aura lieu à Lugano en Suisse. Cette conférence a pour but

de présenter et de promouvoir l'utilisation de Smalltalk dans l'industrie.

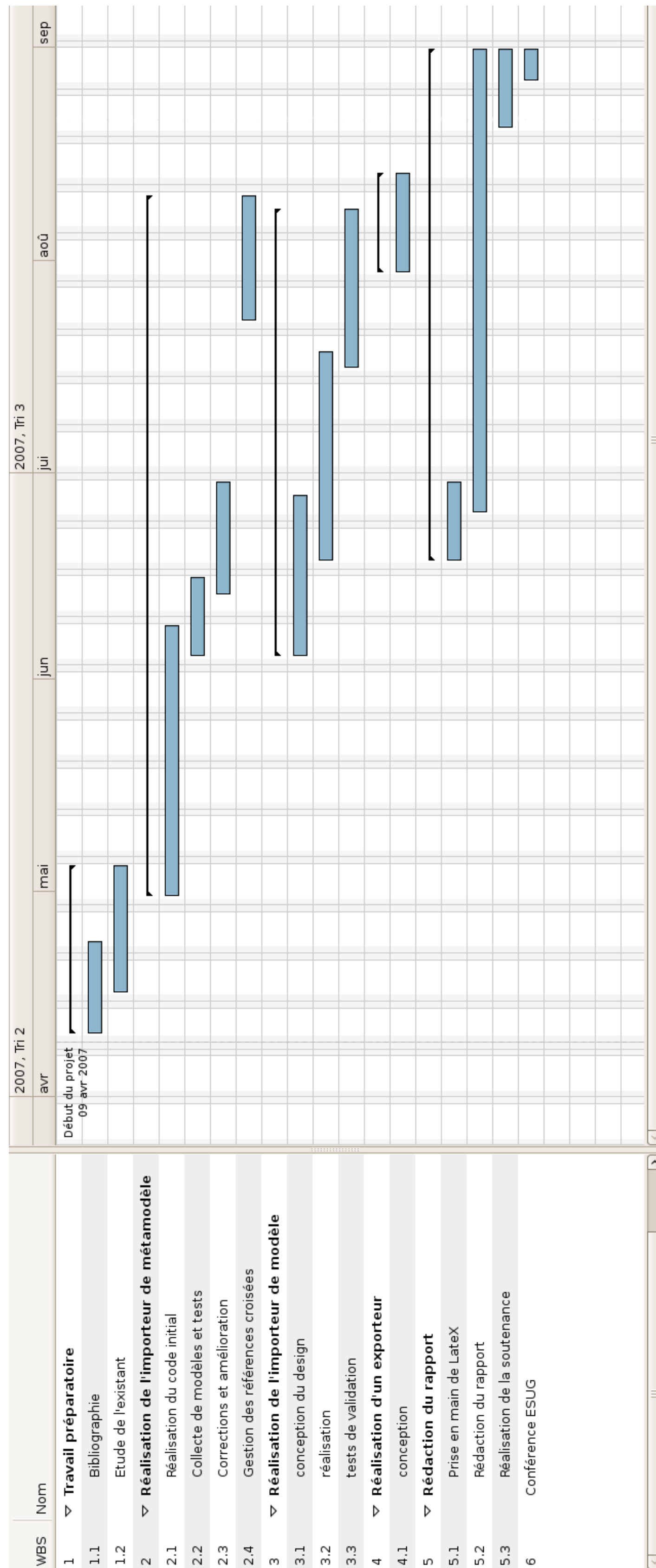


FIG. 2.1.: Diagramme de GANT

3. L'importeur de métamodèle

3.1. Étude préliminaire

Le premier mois de mon stage a été consacré à l'étude du problème ainsi qu'a l'amélioration de mes connaissances du langage Smalltalk, ainsi que de l'environnement de développement VisualWorks. En effet, pendant mon projet d'application système, j'avais déjà eut l'opportunité d'acquérir les bases de ce langage, mais le cadre du Listic était plus propice pour apronfondir mes connaissances. J'ai ainsi pu assister a quelques cours donné par M. Ducasse à l'université de Savoie, et les conseils apportés par les autres stagiaires du Listic m'ont été d'une grande aide pour progresser rapidement.

Comme je l'ai dit plus tôt, lors du partage des taches, il a été convenu que je devais me charger le l'import des métamodèles sauvegardés au format Ecore afin de les rendre utilisables dans un environnement Smalltalk. J'ai donc commencer par étudier plus en détails ce format de donnée afin de définir la meilleure solution. Le format Ecore étant conforme aux spécifications XML, j'ai dans un premier temps cherché quels étaient les outils déjà implémentés dans VisualWorks pour gérer ce format. C'est ainsi que je me suis aperçu qu'il existait un parser XML (correspondant à la classe XMLParser) implanté nativement dans Visualworks. Après avoir étudier cette classe, j'en suis arrivé aux résultats suivants :

- en passant le message parse: avec en argument un stream correspondant a un fichier ecore à une instance de XMLParser, on obtient un Document.
- un Document va permettre la représentation d'un fichier XML sous forme d'un arbre. Cet arbre est constitué d'objets Nodes. Chaque Node représente un élément du fichier d'origine : il est caractérisé par son 'parent', les éléments qu'il contient ainsi que des attributs qui lui sont propres.
- Il existe plusieurs sous-classes de Node (figure 3.1). Cependant, le parser ne crée

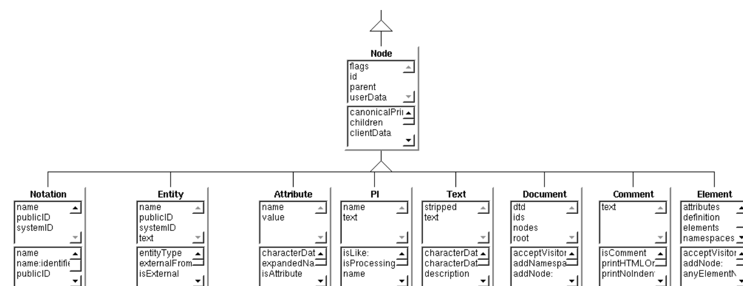


FIG. 3.1.: Hierarchie de la classe Node

que des Elements et des Texts. Seuls les Elements ont un intérêt dans le cadre de

notre projet car ils correspondent aux entités ecore du fichier original. Les Texts ne contiennent que des caractères de saut de ligne et servent à la mise en page du fichier ecore.

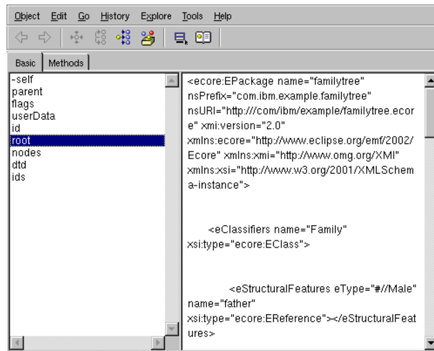


FIG. 3.2.: Un document obtenu en scannant un fichier ecore

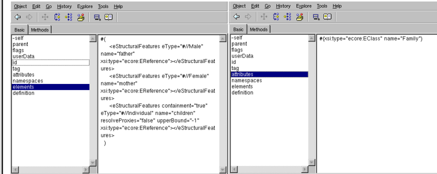


FIG. 3.3.: Un element XML ainsi que ses attributs et ses enfants

Ces résultats convenaient parfaitement à se que j'attendais : ainsi, je n'ai pas eut à réaliser moi même ce parser, ce qui m'aurais certainement retardé pour le reste du projet.

J'ai par la suite étudié le package META fourni avec l'outil Moose. Ce package fourni un ensemble d'outils destinés à la métamodélisation dans Moose. C'est notamment lui qui contient l'implémentation du modèle EMOF utilisées par Moose(figure 1.2).

J'y ai également découvert des importeurs pour d'autres formats de données (KM3 et MSE). Ces importeurs utilisent tous les deux le design pattern (modèle de conception) Visitor. Ce pattern permet de séparer une opération portant sur des objets de la classe même de ces objets. Ainsi, on peut facilement ajouter ou modifier des opérations concernant ces objets, sans les modifier.

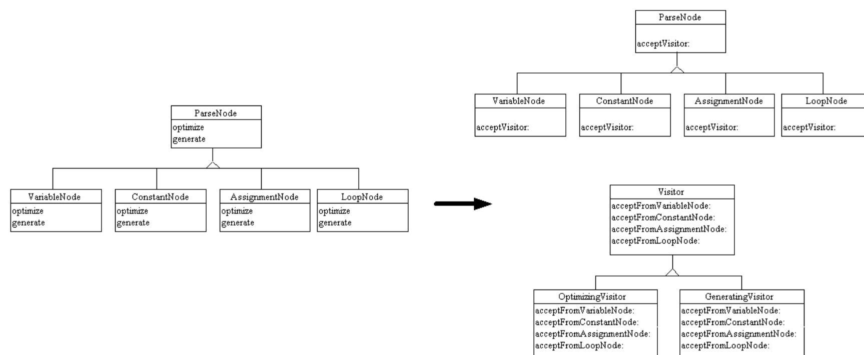


FIG. 3.4.: Un exemple d'utilisation du pattern visitor

Chaque classe pouvant être visitée devra implémenter une méthode `accept:` prenant en paramètre un visiteur et appelant une méthode `visit:` de ce dernier. Cette méthode

visit: doit avoir en paramètre l'objet visité pour que le visiteur puisse accéder aux méthodes de l'objet visité ainsi qu'à ses différents attributs.

3.2. Réalisation

3.2.1. Processus de développement

Une des exigence de M. DUCASSE concernant la réalisation de notre projet était que nous basions notre développement sur les tests unitaires, en utilisant autant que possible l'approche Test First. C'est un processus de développement itératif comportant deux phases :

- Dans un premier temps, on écrit le test correspondant à la fonctionnalité que l'on souhaite implémenter. Un test est un ensemble d'affirmation que doit satisfaire le code que l'on va produire. Il ne fournit aucune fonctionnalité, mais va nous obliger à réfléchir à l'implémentation que l'on va choisir. Par exemple, si l'on souhaite implémenter un dictionnaire, un test simple ressemblera à :

```
monDictionnaire := Dictionnary new.
self assert: (monDictionnaire frenchTranslationFor('dictionnary') = 'dictionnaire').
```

- Dans un second temps, on va réaliser le code a proprement parlé de la manière la plus simple possible (on ne doit pas chercher à anticiper sur les fonctionnalité autres) en le modifiant jusqu'à ce que les tests soient vérifiés. Le code se construit donc par étape successives en remplissant à chaque fois de nouvelles fonctions. A chaque étape, les tests précédents sont relancés pour vérifier que les nouvelles fonctionnalités n'interfèrent pas avec les anciennes.

Pour être utiles, les tests doivent remplir un certains nombres de critères :

- ils doivent être répétables : c'est à dire que un même test doit pouvoir être lancé plusieurs fois et produire toujours le même résultat. En effet, des résultats non constants seraient révélateurs de problème de conception.
- ils doivent pouvoir être lancés sans intervention humaine : de cette façon, un utilisateur peut lancer une longue suite de tests et ne pas être obligé de suivre leur évolution. C'est aussi un critère de répétabilité : en effet, on ne peut garantir qu'un utilisateur fera toujours le bon choix.

Pour nous aider dans ce processus, il existe un framework dédié à Smalltalk nommé SUnit (le même type de framework existe pour de nombreux autres langages). Ce framework permet de regrouper nos tests dans une classe unique sous classe de TestCase (chaque test est alors contenu dans une méthode comme le montre la figure 3.6). L'ensemble des TestCases est ensuite regroupé dans une suite de tests (classe TestSuite). En appelant la méthode run de cette suite de tests, on va exécuter à la suite tous les tests contenus dans tous les TestCases qu'elle contient. Chaque TestCase va retourné un TestResult indiquant le nombre de tests lancé, le nombre de tests ayant échoués et le nombre de tests ayant conduit à une erreur. Un test échoué signifie que le paramètre de la méthode assert: est faux, ce qui est différent d'un test ayant rencontré une erreur pendant son exécution. Un échec signifie que la fonctionnalité testée

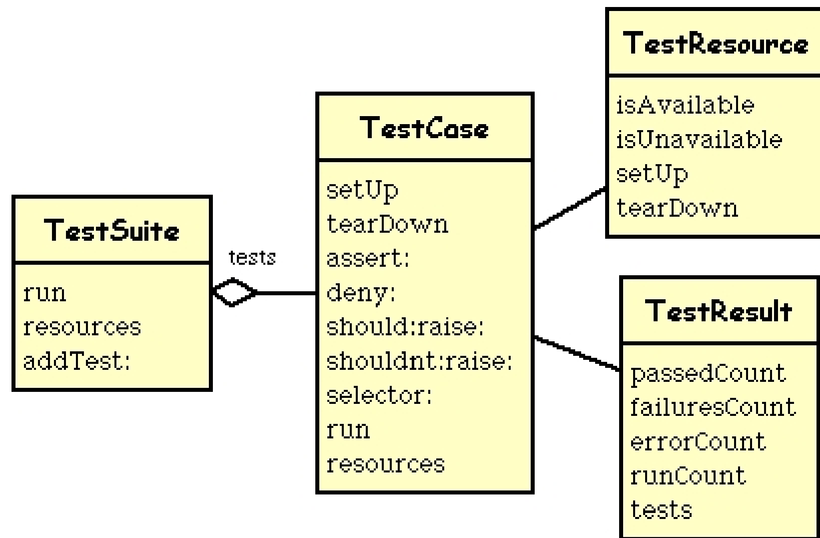


FIG. 3.5.: Le noyau de SUnit

n'est pas remplie, alors qu'un test signalé comme ayant rencontré une erreur signifie que la fonctionnalité a été mal codée. Il faut noté que si le code doit lever des exceptions au cours de son fonctionnement normal, la méthode should: expression raise: exception permet de le vérifier en évitant d'obtenir des résultats du type erreur.

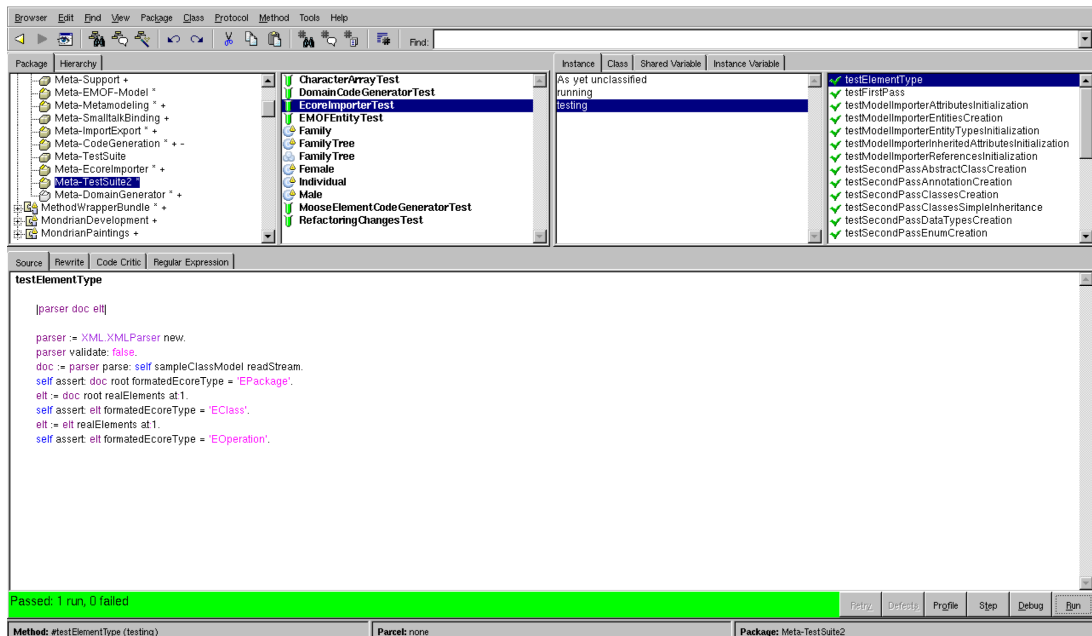


FIG. 3.6.: L'interface du framework SUnit. Ici, un test a été lancé et a réussi

Ce processus offre plusieurs avantages :

– Tout d'abord, le code ainsi développé est en général beaucoup plus fonctionnel

que du code développé selon un processus classique (sans tests). Le fait de relancer tous les tests à chaque itérations va garantir un code avec très peu de bugs. L'enchaînement des itérations successives en ajoutant à chaque étape le code le plus simple possible pour fournir la nouvelle fonctionnalité garanti également un code généralement plus simple et plus réfléchi.

– De plus, si au cours de tests ou de l'utilisation normal du code, on découvre malgré tout une erreur, on écrira un test reproduisant cette erreur avant de tenter de la corriger. Dans le même esprit, avant toute modification du code, on écrira également un test : les tests vont ainsi servir de documentation pour le code que l'on écrit : les tests écrit initialement servant à décrire le fonctionnement normal du code, les tests écrit par la suite traduisant plutôt l'évolution du système. Même si ils ne remplacent pas une vraie documentation, les tests en sont un complément intéressant : tout d'abord parce qu'ils sont indissociables du code (et donc ne peuvent pas être perdus) et de plus, si le processus de développement est respecté par tous, ils sont toujours à jour par rapport au code.

3.2.2. L'architecture

Suite à l'étude préliminaire, j'avais remarqué que les importeurs existant utilisaient tous le pattern Visitor. J'ai donc décidé de suivre le même schéma. J'ai par conséquent créer la classe EcoreMetaModelImporter, qui hérite de la classe Importer du package Meta car cette classe contenait un certain nombre de fonctions intéressantes pour mon projet, en particulier la méthode newInstanceOf: aName qui me permet de créer des instances d'entités EMOF. Un autre avantage lié à ce choix d'héritage est que la classe Importer hérite de la classe Repository. Cette classe représente un conteneur pour un modèle, un métamodèle ou un meta-métamodèle. C'est pour cela que le générateur de code de Mlle SELLOS l'utilisait comme entrée pour son générateur de code. La classe EcoreMetamodelImporter sert donc de conteneur pour les entités EMOF que je génère.

J'ai ensuite créé la classe EcoreVisitor qui va servir de base à la hiérarchie de mes visiteurs. Cette classe ne dispose que d'une seule variable d'instance pointant sur l'importeur auquel sera rattaché notre visiteur. Elle contient aussi les fonctions communes à tous les visiteurs possibles notamment les méthodes d'instanciation, d'initialisation ainsi que la méthode runOn: aDocument. Cette méthode correspond à un design particulier dans lequel c'est le visiteur qui demande dans un premier temps à sa 'cible' la possibilité de la visiter, le code est donc le suivant :

```
runOn: aDocument

  ^aDocument acceptVisitor: self.
```

Ma classe EcoreVisitor implémentait aussi les méthodes visitElement: aXMLElement et visitText: aXMLText en utilisant le code :

```
self subclassResponsibility.
```

Ainsi, les sous classes de EcoreVisitor devaient impérativement implémenter ces deux méthodes. Par la suite, je me suis aperçu que la classe Element du package XML contenait une méthode realElement qui retournait uniquement les fils de type Element.La méthode visitText est alors devenue inutile, je l'ai donc supprimée.

Il ne me restait plus qu'à implémenter les méthodes accept: aVisitor dans les classes Document et Element pour pouvoir réellement commencer à réaliser le projet. Cependant, comme ces classes font partie du coeur de VisualWorks, il était hors de question de les modifier directement. J'ai donc profité du mécanisme d'extension de classe de Smalltalk : Une classe peut être étendue dans un package différent de son package d'origine. Un utilisateur chargeant le package original aura accès aux fonctionnalités de base, alors qu'un utilisateur chargeant le package dans lequel a été étendu la classe pourra profiter des fonctionnalités supplémentaires. Ainsi, j'ai étendu Document et Element dans mon package afin d'implémenter la méthode acceptVisitor dans chacune de ces deux classes. Celle de Document se contente de d'appeler la méthode acceptVisitor sur l'Element racine de l'arbre XML (qui correspond à son attribut root). Elle n'a donc pas vraiment d'importance. Au contraire, la méthode définie dans la classe Element va avoir une importance capitale : c'est en effet elle qui va définir le sens de parcours de l'arbre pour tous les visiteurs quels qu'ils soient.. Deux choix d'implémentation étaient possibles :

– un parcours en profondeur préfixé en utilisant le code :

```
acceptVisitor: anEcoreVisitor

anEcoreVisitor visitElement: self.
self realElements notNil ifTrue: [
self realElements do: [ :each |
each acceptVisitor: anEcoreVisitor]].
```

Chaque élément est visité avant ses fils. L'arbre représenté à la figure 3.7 sera donc parcouru dans le sens A B C D E F G.

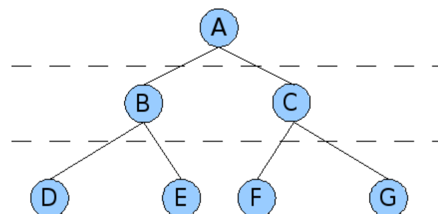


FIG. 3.7.: Un exemple d'arbre

– un parcours en profondeur suffixé en utilisant le code :

```
acceptVisitor: anEcoreVisitor

self realElements notNil ifTrue: [
self realElements do: [ :each |
each acceptVisitor: anEcoreVisitor]].
```

```
anEcoreVisitor visitElement: self.
```

Dans ce cas de figure, l'arbre 3.7 sera parcouru dans le sens D E B F G C A. J'avais dans un premier temps, et par un choix purement arbitraire, choisi la deuxième solution, mais j'ai rapidement été confronté à des problèmes lors de l'utilisation des visiteurs de mon importeur : en effet, de nombreux types d'éléments Ecore possèdent un pointeur vers leur parents (e.g. les classes connaissent leur package, les paramètres leur opérations, etc.). Lors du parcours du fichier Ecore, la résolution de ces liens étaient fortement compliquée par le parcours suffixé, j'ai donc modifié la méthode pour passer à un parcours préfixé.

J'ai décidé de réaliser l'importation en deux phases : la première phase va simplement créer une entité EMOF pour chaque entité Ecore de l'arbre XML. L'entité EMOF créer ne disposera d'aucun des attributs de l'entité Ecore d'origine : seul le nom sera initialisé. Toutes les autres références seront initialisées lors de la seconde passe. Ce choix a grandement simplifié le code. En effet, lors de la seconde passe, à chaque fois que je vais rencontrer une référence à un élément Ecore (par exemple le lien entre une classe et son package) je suis assuré que l'élément référencé existe bien : je peut donc le rechercher sans prendre de précautions particulière.

Si j'avais choisi de réaliser l'importation en une seule phase, il aurait en effet fallu prévoir le cas particulier où l'objet référencé n'a pas encore été initialisé. Ceci implique de réaliser des tests supplémentaires qui risquent de ralentir le système. De plus, le fait de réaliser les deux phases séparément simplifie aussi le debugging. En réalisant les deux phases séparément on facilite l'identification des erreurs et donc on accélère leur résolution. Pour réaliser ces deux phases, j'ai créé deux visiteurs : MetamodelFirstPassVisitor et MetamodelSecondPassVisitor.

3.2.3. Le développement

La première tache que j'ai réalisé consistais à trouver la correspondance entre chaque type d'élément Ecore et un élément EMOF. Pour identifier les différents éléments Ecore, j'ai réaliser avec EMF un modèle comprenant la totalité des éléments mis à notre disposition. En étudiant le fichier ainsi obtenu, je me suis aperçu que l'accès au type d'un Element XML provenant d'un fichier ecore n'étais pas homogène. L'exemple suivant présente le contenu simplifié du fichier ecore correspondant à un package nommé rootPackage contenant une classe nommée classA :

```
<ecore:EPackage xmi:version="2.0" name="rootPackage">
  <eClassifiers xsi:type="ecore:EClass" name="classA">
  </eClassifiers>
</ecore:EPackage>
```

On constate rapidement que la classe possède un attribut 'xsi :type' ayant pour valeur 'ecore :EClass' (figure 3.9) alors que le type du package est défini dans son 'étiquette' (figure 3.8). Heureusement, aucun des éléments dont le type est défini dans l'étiquette

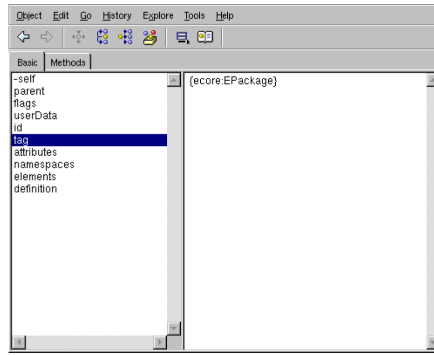


FIG. 3.8.: L'Element correspondant à rootPackage. On constate que le 'tag' (l'étiquette) correspond à 'ecore :EPackage'

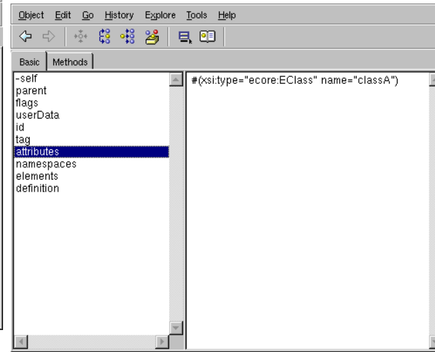


FIG. 3.9.: L'Element classA. Ici on voit que c'est un attribut qui contient 'ecore :EClass'

ne possède d'attribut 'xsi :type'. Par conséquent, j'ai pu implémenté dans mon extension de la classe Element une méthode ecoreType retournant soit la valeur de l'attribut 'xsi :type' si celui ci existait, soit l'étiquette dans le cas contraire. J'ai par la suite modifié légèrement cette méthode pour obtenir une mise en forme identique quelque soit le type de l'élément : en effet, pour un package j'obtenais le résultat 'ecore :EPackage' alors que pour un commentaire, j'avais 'eAnnotations'. La méthode formatedEcoreType effectue donc une mise en forme pour que quelque soit le type, la mise en forme soit la même.

Une fois cette étape réalisée, j'ai inclus dans la classe EcoreMetamodelImporter un dictionnaire associant à chaque type Ecore que j'ai identifié une entité EMOF. C'est à cette période que j'ai implémenté les classes EMOF.Enumeration et EMOF.EnumerationLiteral.

A ce stade, j'avais tous les outils pour faire le premier visiteur. J'ai donc écrit la méthode visitElement: aXMLElement suivante :

```
visitElement: anElement

| element |
  element := importer newInstanceFromEcore: anElement.
  element name: (importer adequateNameOf: anElement).
  importer add: element.
```

La méthode newInstanceFromEcore: aXMLElement fait appelle à la méthode newInstanceOf:aName de la classe Importer dont j'ai parler précédemment. Elle utilise également le dictionnaire associant les types Ecore au nom des entités EMOF qui leur sont associé. La méthode adequateNameOf: retourne le nom de l'entité Ecore, en mettant une majuscule au début du nom des classes et des packages pour respecter les conventions de Smalltalk.

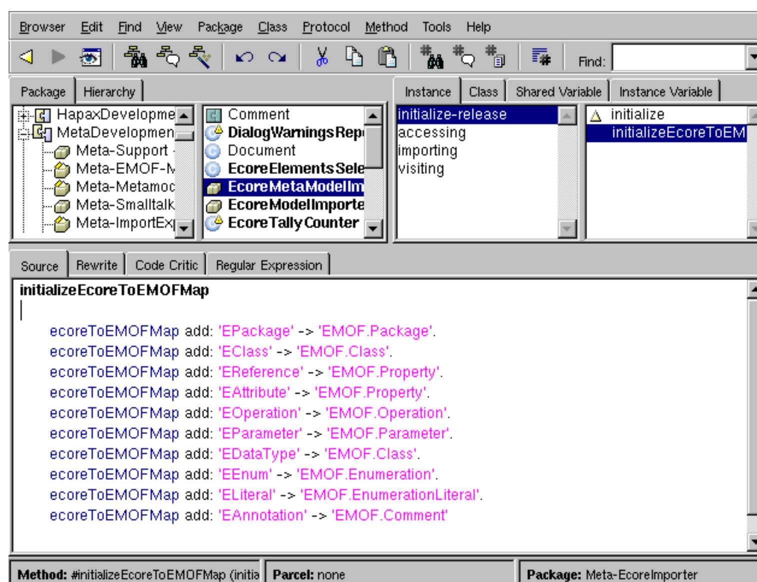


FIG. 3.10.: Le dictionnaire associant entité Ecore et EMOF

Grâce à cette fonction, j'obtiens un EcoreMetamodelImporter contenant des entités EMOF correspondant au métamodèle Ecore d'entrée. Cependant, ces entités EMOF ne possède aucun des attributs (autre que le nom) des entités Ecore.

Je me suis donc occupé de la deuxième passe. Un problème c'est rapidement posé : chaque type Ecore possédant des attributs différents, mon visiteur devait adapter son comportement à chaque entité ; en effet, une classe ne possède pas les mêmes attribut qu'un paramètre : ils doivent donc avoir une initialisation qui leur est propre.. Cependant, toutes ces entités étant de la classe Element, elles appelaient obligatoirement la méthode visitElement du visiteur. J'ai donc eut recours à la méthode perform:aSymbol implémentée dans la classe Object. Cette méthode va exécuter la méthode nommée 'aSymbol' sur son receveur. Le code

```
visitElement: anElement

    self perform: ('visit', anElement formatedEcoreType, ':') asSymbol with: anElement.
```

va donc appeler une méthode dont le nom sera de la forme visitType (par exemple visitEClass :) avec pour argument l'élément XML à visiter (le passage de l'argument se fait grâce à with: anElement>>). Chaque type pouvant être traité différemment des autres, et ceci sans tester le type de l'élément à visiter. Cette architecture, même si elle fait perdre un peu de lisibilité au code (il n'est pas toujours facile d'identifier ce qui va effectivement se passer lors de l'exécution) le rend beaucoup plus simple et plus rapide..

Cependant j'ai rencontrer un nouveau problème : en effet, les références se font entre entités Ecore ; et mon importeur ne disposait que des entités EMOF. Plutôt que de tenter de recréer l'entité EMOF pour la rechercher dans l'importeur, j'ai préféré lui

ajouter un dictionnaire reliant chaque entité Ecore à son entité EMOF correspondante. Ce dictionnaire, que j'ai appelé emofElementMap est initialisé pendant la première passe afin de pouvoir être utilisé pendant la deuxième. J'ai donc modifié la méthode visitElement de mon premier visiteur afin de prendre en compte cette modification.

De la même façon, j'ai rajouté un dictionnaire reliant le nom complet d'une entité Ecore (i.e. le nom de l'entité ainsi que le chemin pour y accéder) à l'entité EMOF correspondante. En effet, les commentaires (EAnnotations) possèdent un attribut nommé 'references' qui corresponds à une liste des éléments concernés par ce commentaire. Malheureusement, cette liste ne contient que les noms des entités référencés, et pas les entités elles-mêmes. Par conséquent, je ne pouvais pas utiliser l'attribut emofElementMap que j'avais créer précédemment. De plus, cette liste de noms étant obtenu en une seule chaine de caractère, il m'a fallu dans un premier temps écrire une méthode pour séparer les noms et obtenir une collection des noms référencés.

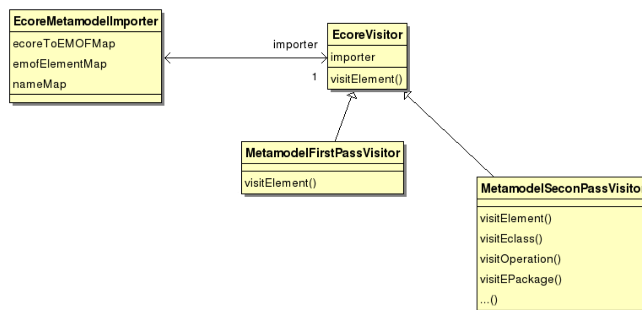


FIG. 3.11.: Le diagramme de classe de mon importeur de métamodèle

Le rapport d'erreurs A ce stade du développement, j'ai commencé à tester l'importeur avec les différents métamodèles que j'avais obtenu de Mrs FALLERY et CHAMPEAU. Ces tests m'ont permis de découvrir un certains nombre de cas particuliers que je n'avais pas envisagés. En effet, EMF est très permissif, il autorise certaines actions que je n'avais pas envisagées lors de la conception de mon importeur, comme par exemple nommer deux classe de la même façon. Certains de ces cas nous paraissaient être des erreurs, nous avons quand même décidé de les gérer. Cependant, j'ai décidé qu'il pouvait être intéressant d'avertir l'utilisateur quand un tel cas de figure se présentait. J'ai donc ajouter à ma classe EcoreMetamodelImporter une collection de Strings nommée warningCollection. J'ai ensuite créé la méthode addWarning: aString qui ajoute simplement la chaîne de caractères passée en paramètre à warningCollection Cette méthode était appelé dans le code servant à résoudre les différents cas particulier rencontrés. A la fin de l'importation, une fenêtre s'ouvrait pour présenter à l'utilisateur les différents problèmes rencontrés (si il y en avait) et lui offrir la possibilité de sauvegarder ce rapport dans un fichier texte.

Cependant, cette fenêtre était problématique pour les tests puisqu'elle nécessitait une intervention humaine pour la fermer ce qui est contradictoire avec le coté automatique des tests. J'ai par conséquent modifié le design en créant la classe Warn-

ingReporter. C'est dans cette classe que j'ai déplacé la collection warningsCollection, ainsi que la méthode addWarning: aString ; j'ai également ajouté la méthode reportWarnings défini de manière abstraite. De son coté, la classe EcoreMetamodelImporter ne possède plus qu'un attribut warningReporter. La classe WarningReporter possède plusieurs sous-classes qui seront utilisée selon le contexte, chacune de ses sous-classes implémentant reportWarnings d'une manière différente. Ce design correspond au pattern Strategy. Par défaut, un EcoreMetamodelImporter est initialisé avec un DialogWarningReporter qui ouvrira une fenêtre à la fin de l'import d'un métamodèle. Lors des tests, il utilisera un TestWarningReporter qui retournera une chaine de caractères sur laquelle je pourrais faire des tests pour tester son fonctionnement.

Les références croisées L'autre principale difficulté que j'ai découvert pendant les tests était les références croisées : certains fichiers que nous a fourni M. FALLERY comportaient en effet des références à des fichiers Ecore différents (e.g. une classe ayant pour super-classe une classe appartenant à un autre métamodèle). Malheureusement, les fichiers de départ ne comportent aucune information concernant le fichier cible. EMF va automatiquement aller chercher dans les autres fichiers du projet celui qui correspond. J'ai longtemps cherché comment avoir moi aussi accès aux entités de ces fichiers cible, mais ne trouvant pas de solution, je me suis résolu à adopter une autre stratégie. J'ai ajouter un EMOF.Package nommé 'UnresolvedElements à la racine du métamodèle en cours d'importation. Ce package dispose d'une méthode d'accès réalisant une initialisation paresseuse (Lazy Initialization), c'est à dire que ce package ne sera créé qu'à la première utilisation de cette méthode d'accès. Ainsi, si un métamodèle ne possède aucune référence croisée, ce package n'apparaîtra pas dans mon importeur.

Une fois ce package mis en place, il pouvait contenir des entités EMOF représentant les cibles des références croisées. Dans un premier temps, je me suis contenté de créer une entité dont le type correspondait et dont le nom était le chemin complet permettant d'accéder à la cible de la référence.

Par exemple, une classe dont la super-classe est référencée par :

fileB.ecore#//PackageA/ClassC

signifie que la super-classe en question se nomme ClassC et qu'elle appartient au package PackageA du fichier fileB.ecore. Lorsque mon visiteur rencontrait une telle référence, il créait dans le package UnresolvedElement une EMOF.Class nommée 'fileB.ecore#//PackageA/ClassC'. Cependant, cela ne faisait pas vraiment de sens. De plus, les classes n'étaient pas les seuls éléments référencés de la sorte : leurs attributs aussi l'était parfois. Et avec cette approche, les attributs n'avait pas de lien avec leur classe d'origine, ce qui n'était pas très représentatif.

J'ai donc modifié mes méthodes de création de ces entités externes en leur incluant un très simple parser de chaînes de caractères pour que la hiérarchie du fichier soit recréée. Actuellement, la référence précédente va créer une EMOF.Class nommée 'ClassC' appartenant à un EMOF.Package nommé 'PackageA', lui même inclus dans

un autre EMOF.Package nommé fileB ; ce dernier package appartenant au package 'UnresolvedElements' défini un peu plus haut.

Arrivé à ce stade, j'ai essayer de remplacer les entités que j'avais ainsi créées par les véritables cibles de la référence en proposant à l'utilisateur d'importer les fichiers manquants. Ceci aurait permis, notamment pour la définition des super-classes, d'être plus complet, en ajoutant notamment les méthodes et les attributs aux classes que je crée automatiquement. Cependant, bien que j'arrive à retrouver les entités cibles au sein du deuxième métamodèle importé, je n'est pas réussi à les substituer aux entités que je crée moi-même.

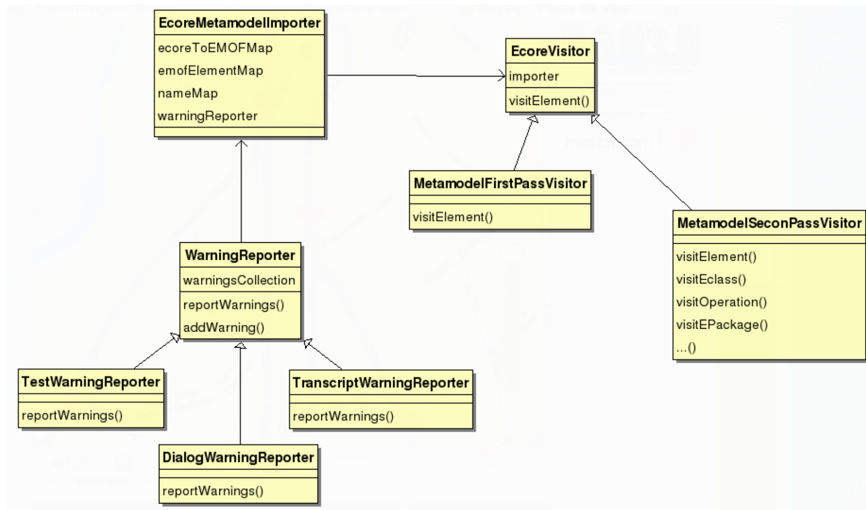


FIG. 3.12.: Le diagramme de classe simplifié de l'importeur de métamodèle

4. L’importeur de modèles conformes

4.1. Étude préliminaire

Pour mon deuxième importeur, l’étude préliminaire a été plus rapide. L’expérience que j’avais acquis lors de la réalisation de l’importeur de métamodèle m’a été utile. En effet, ayant déjà étudier le package Meta, j’avais déjà envisagé l’utilisation de métamodèles personnels. J’avais par exemple remarqué que la classe Repository possédait un attribut metaRepository. Cet attribut est lui-même un Repository qui contient le métamodèle auquel doivent se conformer les entités importées. Dans le cas de l’importeur de métamodèle, cet attribut contenait le métamodèle EMOF. La principale différence vient du fait que le format sous lequel sont sauvegardés les modèles conformes est beaucoup plus alambiqué que celui des métamodèles.

Prenons le métamodèle représenté par figure 4.1. A partir de ce métamodèle, j’ai

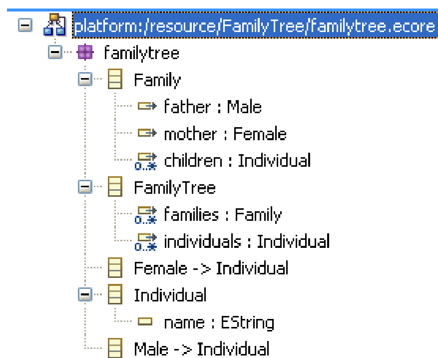


FIG. 4.1.: Un métamodèle d’arbre généalogique

créer le modèle conforme représenté figure 4.2

On peut tout d’abord remarquer la structure d’un modèle : elle sera toujours présenté sous cette forme. Lors de la réalisation d’un modèle, EMF ne laisse la possibilité que d’avoir un seul et unique élément à la base de la hiérarchie. Dans notre exemple, il aurait été impossible de modéliser deux arbres généalogiques (FamilyTree) en parallèle.

En regardant le code figure 4.3, on comprend un peu mieux la manières dont sont faites les références. Le premier point que l’on peut remarquer est que le type des entités rencontrés n’est pas accessible directement (que ce soit par l’intermédiaire d’un attribut ou dans l’étiquette). La seule information que nous possédons est le nom de

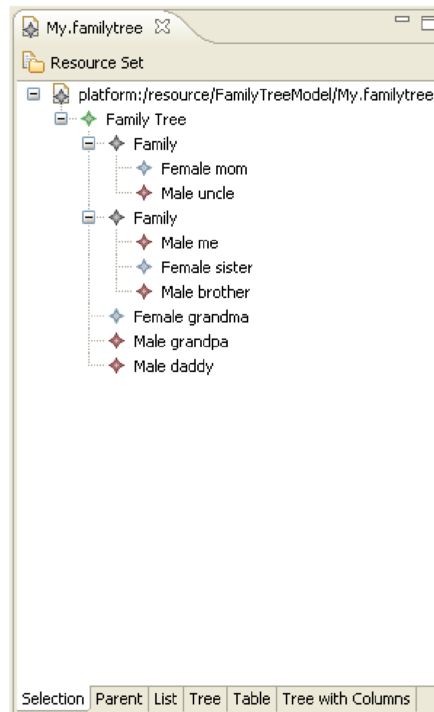


FIG. 4.2.: Un modèle conforme au métamodèle figure 4.1

```
*My.familytree
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.example.familytree:FamilyTree xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance
xmlns:com.ibm.example.familytree="http:///com/ibm/example/familytree.ecore">
  <families father="//@individuals.1" mother="//@individuals.0">
    <children xsi:type="com.ibm.example.familytree:Female" name="mom"/>
    <children xsi:type="com.ibm.example.familytree:Male" name="uncle"/>
  </families>
  <families father="//@individuals.2" mother="//@families.0/@children.0">
    <children xsi:type="com.ibm.example.familytree:Male" name="me"/>
    <children xsi:type="com.ibm.example.familytree:Female" name="sister"/>
    <children xsi:type="com.ibm.example.familytree:Male" name="brother"/>
  </families>
  <individuals xsi:type="com.ibm.example.familytree:Female" name="grandma"/>
  <individuals xsi:type="com.ibm.example.familytree:Male" name="grandpa"/>
  <individuals xsi:type="com.ibm.example.familytree:Male" name="daddy"/>
</com.ibm.example.familytree:FamilyTree>
```

FIG. 4.3.: Le code associé au modèle

l'attribut dans le métamodèle. Dans notre exemple, les entités instance de la classe Family possèdent le type 'families' qui est le nom de l'attribut de FamilyTree possédant le type Family. Ce moyen de faire n'est pas très intuitif, car avant de pouvoir créer une entité d'un modèle, il va falloir déterminer son type en allant vérifier dans le métamodèle courant.

De plus, comme on peut le lire dans le code, les attributs 'mother' and 'father' pointant vers des individus déjà défini se font en désignant la position de l'individu dans la hiérarchie du modèle. Ainsi, l'attribut 'mother' de la deuxième famille est référencé par : '"//families.0/children.0' ce qui signifie que la mère de cette famille est le premier enfant (en java, la numérotation des tableaux commence à 0) de la première famille de l'élément racine de notre modèle.

4.2. Réalisation

4.2.1. L'architecture

De même que pour l'importeur de métamodèle, j'ai décidé de faire hériter celui ci de la classe Importer du package Meta. En effet, j'avais encore besoin de créer des instances d'entités existantes, par conséquent la méthode newInstanceOf: aName était toujours très utile. Et comme je l'ai dit dans mon étude préliminaire, la structure de Repository me permet de définir assez simplement le métamodèle auquel doivent être conformes les entités créées.

Dans un premier temps, j'ai également adopté le pattern Visitor. J'ai donc créé les classes EcoreModelImporter et ModelVisitor en suivant le même modèle que pour l'importeur de métamodèle. Cependant, au fur et à mesure que le développement avançait, je me suis rendu compte que ce design n'était pas forcément adapté pour cet importeur. En effet, mon visiteur faisait très souvent appel à des méthodes de l'importeur, avec pour conséquence de ralentir le système. J'ai donc supprimé la classe ModelVisitor et déporté les méthodes directement dans l'importeur.

J'ai également découpé l'import de modèle en deux phases :

une phase de création des entités pendant laquelle je vais définir le type des éléments XML obtenu lorsque l'on utilise le XMLParser sur un fichier de modèle et créer les objets correspondants. C'est également pendant cette phase que je reconstruis la structure du modèle importé.

une phase d'initialisation des objets pendant laquelle je vais me charger d'initialiser les attributs des différentes entités du modèle importé.

4.2.2. Le développement

J'ai, pour commencer, étudier le moyen d'implanter un métamodèle personnel dans mon importeur. Comme je l'ai dit dans mon étude préliminaire, j'ai constaté que la classe Repository (qui fait parti des super-classes de mon importeur) possède un attribut metaRepository dans lequel est stocké le métamodèle auquel doivent

correspondre les entités importées. J'ai donc tout naturellement choisi d'utiliser un EcoreMetamodelImporter comme metaRepository. Cependant, ce choix c'est rapidement révélé n'être pas si judicieux que cela. En effet, un Repository possède un dictionnaire faisant le lien entre les entités qu'il contient, et le code qui les représente. Et ce lien est utilisé pendant la création d'instance grâce à la méthode newInstanceOf:. Heureusement, le générateur de code de Mlle SELLOS était fonctionnel. Les classes générées étaient rangées dans des Namespaces. Il m'a donc fallu étudier le moyen d'utiliser cette structure pour définir mon metaRepository. C'est en effet de la méta-description des classes dont j'avais besoin, et non pas des classes elles-mêmes. Heureusement, lors de la génération de son code, ma camarade avait implémenté tous les protocoles de métamodélisation nécessaires à l'obtention de cette méta-description. En ajoutant la méta-description de chaque classe au metaRepository de mon importeur, j'obtins le même résultat qu'en utilisant un EcoreMetamodelImporter, mais le lien entre les entités du metaRepository et le code est fait ce qui permet de générer correctement les entités des modèles.

Cependant, avant de les générer, il fallait résoudre le problème du type des éléments a générer. Seul le type de l'élément à la racine du modèle est directement accessible, toutes les autres entités sont référencées par le nom de l'attribut qu'elle représente. J'ai écris la EcoreModelImporter>>typeOfAttribute: qui prend en paramètre un élément XML correspondant à une entité d'un modèle. Cette méthode va aller chercher dans la metaDescription du conteneur de l'élément passer en paramètre l'attribut correspondant et ainsi déterminer le type. Ainsi, dans l'exemple précédent, en exécutant la méthode sur un élément représentant une des familles, on aura la séquence suivante :

1. l'importeur va déterminer que le conteneur de l'élément étudié est la racine du modèle.
2. il va ensuite consulter dans la metaDescription de cet élément (voir figure 4.1 si un attribut nommé 'families' est bien présent.
3. un fois cet attribut retrouvé, on connaîtra la classe de l'entité à créer pour représenter notre famille (ici Family).

Une fois l'entité créée, j'utilise les accesseurs générés automatiquement par le générateurs de code pour régénérer la hiérarchie du modèle.

Une fois les entités créées, nous avons tenter de les visualiser dans Mooose. Nous nous sommes alors rendu compte que pour pouvoir être utiliser dans Moose, les entités de nos modèles devait être instance de classe héritant de MooseElement. Mlle SELLOS à donc modifier son générateur de code afin que les modèles que je génère soit utilisables dans Moose.

Une fois cette étape réalisé, je me suis occupé de l'initialisation des attributs de chaque entité créée précédemment. Cette initialisation repose sur une méthode capable, à partir d'une chaîne de caractère telle que celle que j'ai présentée au chapitre 4.1, de parcourir le modèle pour retourner l'entité ciblée par la référence. Cette méthode va parcourir la chaîne de caractère à la recherche des caractères '/'. De cette manière, elle va pouvoir accéder successivement à chaque entités citées, et tenter de

les retrouver parmi les éléments créer précédemment. Bien entendu, si la valeur de l'attribut n'est pas une référence, mais une chaîne classique, c'est cette chaîne qui sera renvoyée. Il a également fallu que je différencie l'initialisation des attributs de multiplicité unique et multiple : en effet, comme j'utilise ici encore les accesseurs générés par le générateur de code de Mlle SELLOS, pour initialiser un attribut ayant une multiplicité unique, je vais utiliser une méthode du type nomDeLAttribut: alors que si cet attribut est de multiplicité multiple, il sera initialisé comme une Collection. Je devrais donc employé la méthode add:

5. Conclusion

5.1. Apports

D'un point de vue personnel, j'ai trouvé ce stage très enrichissant à plusieurs points de vue. Tout d'abord, il m'a permit de découvrir le monde de la recherche que je ne connaissais pas, ayant réalisé mes deux précédents stages dans le milieu industriel. L'état d'esprit y est complètement différent, la curiosité y est encouragée beaucoup plus que dans l'industrie où l'on doit malheureusement souvent se cantonner à faire uniquement ce qui nous est demandé.

De plus, j'ai découvert un nouveau langage de programmation. Le langage Smalltalk est d'autant plus intéressant qu'il est vraiment différent des autres langages que j'avais appris jusqu'à présent. Il m'a notamment permis d'acquérir une certaine rigueur, principalement en ce qui concerne le nommage des attributs, des classes etc. En effet, le typage dynamique oblige à être beaucoup plus attentif aux noms donnés sous peine de rendre le code produit complètement incompréhensible.

Le fait d'avoir utilisé le processus de développement dirigé par les tests est également un atout. Même si cela peut sembler contraignant au premier abord, ce style de développement devient rapidement naturel et permet d'améliorer grandement la qualité du code produit. Je m'en suis personnellement rendu compte en comparant le code produit pendant ce stage et le code qui avait été réalisé pendant mon projet d'application système.

5.2. Perspectives

Le code que nous avons produit durant la durée de ce stage permet de générer du code Smalltalk à partir d'un métamodèle au format Ecore. Ce code peut servir de base à une personne souhaitant poursuivre le développement du projet en utilisant le langage Smalltalk. Il peut également être importer dans Moose, afin d'être analysé, compris et si possible amélioré. Un utilisateur peut également charger un modèle conforme dans Moose afin de profiter de certaines visualisations.

Cependant, quelques points seraient perfectibles pour rendre notre outils plus utile. Par exemple, il serait certainement très intéressant d'étudier la possibilité de générer de nouvelles visualisations adaptées aux modèles conformes. Celles disponibles actuellement ne sont en effet pas toujours très fonctionnelles.

Une autre possibilité d'amélioration serait de trouver le moyen de générer du code associé aux modèles conformes. En effet, les entités actuellement générées ne sont que les instances d'autre classe : elles n'ont donc pas de code à proprement parlé. Il

serait certainement possible de modifier l'importeur pour pouvoir générer des modèles exécutables.

Une dernière possibilité d'amélioration serait d'offrir une gestion plus avancée des commentaire, en offrant par exemple la possibilité de reconnaître du code OCL. Ce code pourrait ensuite être traduit pour générer les méthodes Smalltalk appropriées et ainsi rendre nos métamodèles exécutables.

A. Syntaxe Smalltalk

Le tableau A.1 présente les éléments syntaxiques de Smalltalk :

.	permet de séparer les expressions.
:=	affectation.
ˆ	symbole de retour.
'une chaine de caractère'	les simples qu'ôter délimitent les chaînes de caractère.
,	opérateur de concaténation de chaine.
[expression]	délimitent les blocs de code.
[:arg1 :arg2 | bloc]	les blocs peuvent posséder un ou plusieurs arguments.
| tmpVar1 tmpVar2 |	les 'pipes' servent à déclarer les variables temporaires. Leur durée de vie et leur portée est limitée à la méthode
"commentaire"	les double quotes marquent les commentaires.
$a	le $ indique une instance de la classe Character.
#(a b)	une collection contenant les éléments a et b.

TAB. A.1.: Éléments syntaxique de base

En Smalltalk, il existe six pseudo variables. Elles sont données dans le tableau A.2.

true et false	booléen, instance de True and False.
nil	valeur donnée à une variable non définie.
self	objet courant (équivalent de this en Java).
super	représente également l'objet courant. Cependant la recherche de méthode doit se fa
thisContext	pile d'exécution.

TAB. A.2.: Pseudo Variables

En Smalltalk, Il n'existe que des méthodes que l'on exécute sur des objets. L'appel à une méthode ce fait en envoyant un message du même nom que la méthode à l'objet.

Il existe trois types de méthodes :

Les méthodes unitaires Ces méthodes ne prennent pas d'arguments.

Appel de la méthode removeAll sur l'objet 'aCollection' :

```
aCollection removeAll.
```

Les méthodes binaires Les méthodes binaires ne prennent qu'un argument et leur nom est composé de symboles (',' '+' '-'...).

Appel de la méthode + sur l'objet 1 avec le paramètre 3 :

```
1 + 3.
```

Les méthodes à mots-clés ce sont des méthodes qui peuvent prendre autant de paramètres que vous le souhaitez. Leurs noms sont composés de plusieurs groupes de caractères (autant qu'il y a d'arguments) se terminant chacun par le caractère ' :'. Les arguments se placent après chaque caractère ' :'.

Appel de la méthode `replaceFrom:to:with:` qui prend trois arguments sur l'objet 'aCollection'

```
aCollection replaceFrom: 1 to: 6 with: anOtherCollection.
```

L'équivalent en Java serait :

```
aCollection.replaceFromToWith(1, 6, anOtherCollection);
```

Priorité des messages Lors de la lecture d'une expression les messages sont évalués dans l'ordre suivant :

1. méthodes unitaires ;
2. méthodes binaires ;
3. méthodes à mots-clés.

Par exemple :

```
5 factorial + 5 gcd: 5
```

Doit être lu :

```
((5 factorial) + 5) gcd: 5
```

Les opérations mathématiques n'ont pas de priorité particulière, ils respectent exactement la même règle. Par conséquent `3 + 4 * 3` vaut 21 et pas 15. Il faut donc utiliser des parenthèses pour rétablir la priorité classique : `3 + (4 * 3)`

Cascades Il est également possible d'envoyer plusieurs messages à la suite à un même objet. Pour cela, il faut utiliser l'opérateur ' ;' au lieu du '.'.

```
aCollection
    add: anObject;
    add: anOtherObject;
    add: aThirdObject.
```

Définition de méthode La définition d'une méthode se fait de la manière suivante :

```
String>>lineCount
  "Answer the number of lines represented by the receiver (a string), where every
   carriage return (cr) adds one line."

  | count |
  count := 1.
  self do:
    [:c | (c == Character cr)
            ifTrue: [count := count + 1]].
  ^ count
```

Le code ci-dessus doit être compris comme :

1. une nouvelle méthode nommée lineCount est définie dans la classe String.
2. le commentaire donne des informations complémentaires sur le but et l'utilisation de la méthode.
3. la variable temporaire count est déclarée.
4. elle est initialisée avec la valeur 1.
5. le message #do: est envoyé à l'objet courant (self). Il prend comme argument un bloc de code (qui doit posséder lui aussi un argument). Ce bloc sera évalué pour chaque caractère de la chaîne sur laquelle sera exécutée cette méthode. La méthode #do: est l'équivalent de foreach en C#.
6. le bloc défini un argument c. c prendra la valeur de chacun des caractères de la chaîne courante.
7. Le corps du bloc commence par comparer c avec le caractère de fin de ligne. Ce caractère s'obtient en appelant la méthode cr sur la classe Character.
8. le résultat de la comparaison est un booléen : soit true soit false.
9. true et false implémente tous les deux la méthode #ifTrue:. #ifTrue: prend en argument un bloc qui ne sera évalué que si le receveur est true.
10. par conséquent, si c est un caractère de fin de ligne, le bloc est exécuté, et on ajoute 1 à la variable count
11. quand la méthode do: a fini d'itérer sur tous les caractères, elle se termine puis la valeur de count est retourné.