KULAKOWSKI Fabienne
BROCCOLICCHI Jérôme

Projet LPMI

OFL- ML

Réalisation d'un éditeur permettant d'éditer graphiquement (UML étendu) une application conformément au modèle OFL

Responsables : LAHIRE Philippe
CRESCENZO Pierre

Année 2001-2002

Remerciements

Nous remercions Philippe LAHIRE et Pierre CRESCENZO pour nous avoir donné l'opportunité de développer un outil s'intégrant à un projet d'ampleur : OFL.

Nous les remercions aussi pour leurs conseils, leur soutien, et leurs visites hebdomadaires, qui ont entretenu notre motivation.

Résumé

OFL (Open Flexible Languages) est un modèle métaobjets qui peut décrire les langages orientés objets à classes les plus courants (Java, Eiffel, C++ …). Chaque langage est construit à l'aide d'OFL-composants « class », « interface », « extends », « implements » …

Quatre outils mettent en œuvre le modèle OFL : OFL-DB, OFL-Meta, OFL-Parser et OFL-ML que nous développons. L'outil OFL-ML offre au programmeur la possibilité de créer des applications. C'est un outil essentiellement graphique basé sur les normes UML.

Table des matières

Introduction

Le désir de toute personne s'intéressant à la programmation informatique est d'améliorer la qualité des logiciels. Cependant, sur quoi la qualité d'un logiciel se base-t-elle ? Est-ce sa performance, c'est-à-dire sa rapidité d'exécution, sa fiabilité, sa robustesse, c'est-à-dire sa capacité à travailler dans un environnement matériel et logiciel dégradé … ? L'amélioration globale d'un logiciel dépend en fait d'un compromis entre ces qualités.

Notre objectif principal est d'améliorer la qualité du code source des logiciels. Pour cela, nous utilisons le modèle Open Flexible Languages (OFL), basé sur les langages de programmation orientés objets à classes, tels Java, C++. OFL est un modèle métaobjet qui décrit la sémantique opérationnelle, le comportement des langages de programmation à objets les plus courants. Le principal objectif du modèle OFL est d'offrir au métaprogrammeur un moyen simple de créer un nouveau langage ou de modifier le comportement d'un langage existant. En effet, traditionnellement, la métaprogrammation est une tâche difficile et fastidieuse. Pour mettre en œuvre le modèle OFL, quatre outils sont en cours de développement : OFL-DB, OFL-Meta, OFL-Parser et OFL-ML, que nous développons.

Nous vous présenterons dans un premier temps les concepts de base d'OFL pour une meilleure compréhension, puis dans une deuxième partie, nous vous décrirons l'outil OFL-ML.

1. Présentation du projet

1.1 OFL, les concepts de base

Le modèle OFL étant très vaste et complexe (il faudrait au moins 150 pages pour l'expliquer), nous n'aborderons dans cette partie uniquement les points, les concepts qui nous semblent utiles à la compréhension de l'outil que nous avons développé.

1.1.1 Concept-relation

La première notion fondamentale du modèle OFL est constituée par les concepts-relations. Un concept-relation est une abstraction d'une sorte de relation dans les langages orientés objets à classes.

Les instances d'un concept-relation sont nommées composants-relations et peuvent être vues comme des métarelations. Par exemple, dans le langage Java, nous pouvons citer comme composant-relation, les héritages implements ou extends ; et comme relation, la relation extends entre deux classes et la relation implements entre une classe et une interface. Chaque composant-relation pourrait donc être une classe, dont les instances seraient des relations.

Pour chaque relation entre descriptions (relation inter-descriptions), il existe :

- une description-source : c'est la description (la notion de description est décrite dans le 1.1.2) qui déclare la relation.
- une description-cible : c'est la description qui fournit les « informations » de la relation.

Le modèle OFL décrit les relations inter-descriptions (héritage, agrégation) et les relations entre descriptions et objets (instanciation).

1.1.2 Concept-description

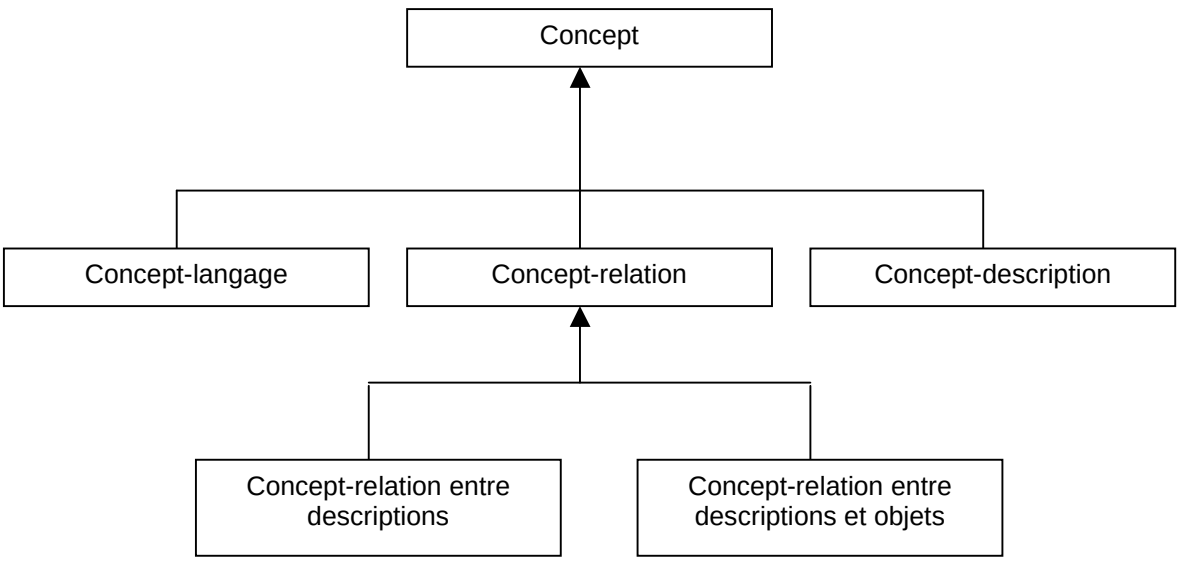
Un concept-description permet de généraliser la notion de classe pour représenter aussi bien une classe, qu'une interface (pour Java par exemple), ou qu'une valeur de base.

Les instances d'un concept-description sont nommées composants-descriptions et peuvent être vues comme des métaclasses. Chaque composant-description est compatible ou non avec un ensemble de composants-relations. Par exemple, en Java, le composant-description interface est compatible avec le composant-relation implements.

1.1.3 Concept-langage

Un concept-langage est une notion simple et importante. Un concept-langage est une réification de la notion de langage. C'est un méta méta langage dont les instances d'instances, ou composants-langages, sont les langages. Il permet de fédérer les composants-relations et les composants-descriptions. Chaque langage est constitué en particulier d'un ensemble de concepts-descriptions et d'un ensemble de concepts-relations, chacun étant compatible avec au moins un des concepts-descriptions sélectionnés.

1.1.4 Hiérarchie des concepts



1.2 Les outils associés à OFL

Quatre outils sont en cours de développement afin de mettre en œuvre l'environnement de métaprogrammation et de programmation associé au modèle OFL..

- OFL-DB : Tous les outils qui gravitent autour d'OFL doivent partager des données. Ces données sont par exemple les OFL- concepts (concepts-descriptions, concepts-relations). Mais n'oublions pas également celles qui seront les plus importantes pour les programmeurs, les bibliothèques d'OFL-composants (composant-description, composant-relation, composant-langage). Nous avons donc besoin de conserver ces entités dans un formalisme standard, pour permettre un partage aisé, et durable, pour pouvoir envisager leur réutilisation. C'est ce que permet OFL-DB.

- OFL-Meta : Il est destiné aux métaprogrammeurs. Son rôle est de permettre et de faciliter la création de nouveaux OFL-composants (composant-description, composant-relation, composant-langage). Le but d'OFL-Meta est d'offrir au métaprogrammeur la capacité de décrire un langage.

- OFL-ML : Il permet au programmeur de créer graphiquement des descriptions, de décrire les relations entre elles et de définir les méthodes et attributs des descriptions, après que le métaprogrammeur est intervenu et décrit un langage à l'aide de OFL-Meta. Les entités de base du modèle sont conservées par OFL-DB.

- OFL-Parser : Le métaprogrammeur puis le programmeur ayant fait leur office, il reste maintenant à traduire les différents éléments du modèle et la définition de l'application pour qu'elle devienne exécutable. Le but d'OFL-Parser est d'exécuter l'application décrite dans OFL-ML.

1.3 L'outil OFL-ML

Nous décrirons dans cette partie le travail que nous avons dû réaliser avec ses contraintes, et le but à atteindre.

1.3.1 But du projet

Notre rôle dans le projet OFL était de réaliser l'outil OFL-ML. Le but était de commencer la programmation de l'outil, et non pas de la finaliser. Nous avons réalisé cet outil en parallèle avec l'équipe chargée de développer l'outil OFL-Meta dans un souci de cohérence avec les fichiers créés dans OFL-Meta et dont se sert OFL-ML.

L'outil a été développé avec la version 1.1.3 du Java Development Kit (JDK) et le logiciel JCreator.

1.3.2 Contraintes du projet

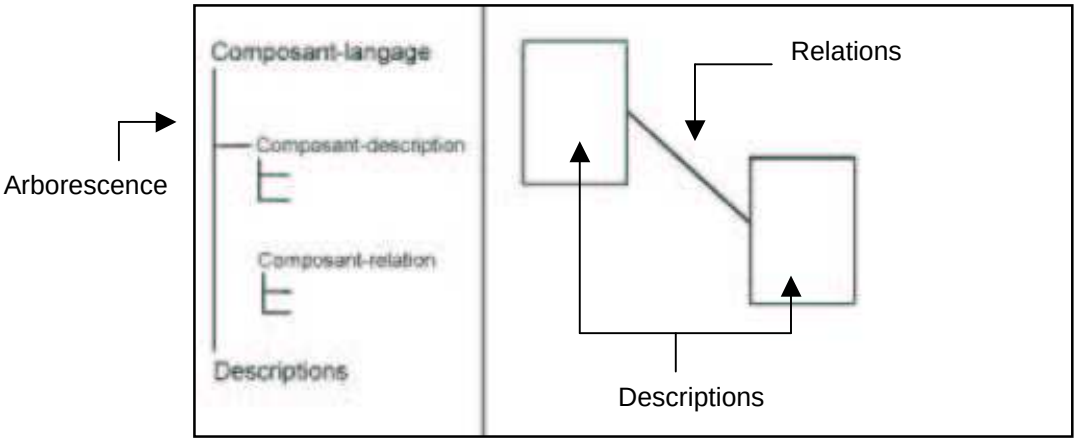
Pour le développement de l'outil, certaines contraintes nous ont été imposées. Il y avait tout d'abord une contrainte temporelle : notre projet devait se dérouler sur environ 4 mois. Par ailleurs, l'outil devait être développé en Java. En effet, par rapport aux concepts développés par le modèle OFL, qui se basent sur les langages orientés objets, il semblait primordial de coder l'outil OFL-ML dans un langage à objets, qu'est le langage Java. De plus, les autres outils d'OFL (OFL-DB, OFL-Meta, OFL-Parser) sont développés en Java, donc pourquoi créer l'exception avec OFL-ML.

Aussi, toutes les classes créées devaient être commentées en utilisant les tags de la documentation de Java afin de la générer. Vous pouvez le constater en vous référant aux annexes. De plus tous les attributs des classes devaient être privés, c'est-à-dire qu'ils ne pouvaient être accessibles uniquement dans les méthodes de la classe où ils ont été créés, afin d'éviter les changements incongrus de valeur. Il a donc fallu créer un ou plusieurs accesseurs en consultation (méthode get…) et en modification (méthode set…) pour chaque attribut. Enfin, il fallait produire un code propre, compréhensible et commenté, le code pouvant être réutilisé.

Ces nombreuses contraintes nous ont été bénéfiques en tant que développeurs, et l'application que nous avons codée est stable. Cependant, elles ont ralenti l'avancement du projet : il est plus rapide et plus facile de faire du code « sale », que du code « propre ».

1.3.3 Cahier des charges

L'interface graphique de l'outil OFL-ML est constituée de deux parties, avec à sa gauche, une arborescence de fichiers et à sa droite une zone de dessin. L'arborescence est créée à partir d'un fichier généré par l'outil OFL-Meta. Voici un schéma de l'interface :

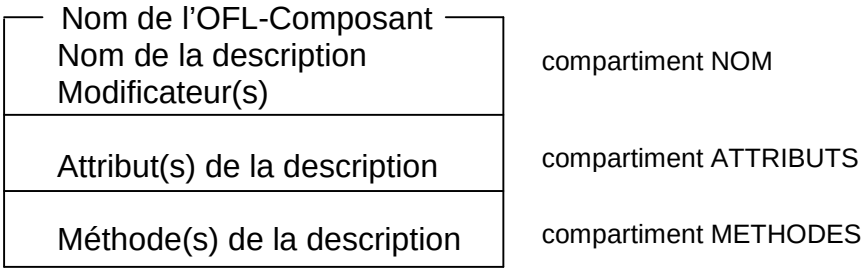


Les descriptions et les relations sont créées respectivement à partir des composants-descriptions et des composants-relations du composant-langage de l'arborescence.

L'outil OFL-ML se base sur les normes de représentation d'UML (d'où il tire son nom). Il permet au programmeur de représenter graphiquement des composants-descriptions, composants-relations, d'en décrire les attributs et méthodes. L'outil est adapté à tous les langages : il n'est pas nécessaire que le composant-langage soit toujours identique.

1.3.3.1 Représentation UML des composants-descriptions

Un composant-description est représenté de la manière suivante (nous avons divisé sa représentation en compartiment afin d'en faciliter la compréhension):



Le nom de l'OFL-composant est le type de la description.

1.3.3.1.1 Compartiment nom

- Le nom de la description est défini par le programmeur et est différent d'une description à l'autre.
- Son type est celui d'un type du composant-langage.
- Les modificateurs sont celui du composant-langage. Il peut y en avoir aucun, un seul ou plusieurs. Dans ce cas, le programmeur doit respecter les règles de programmation du composant-langage. Par exemple, pour Java, on ne peut pas mettre public private côte à côte.

1.3.3.1.2 Compartiment attributs

- Le nom de l'attribut est défini par le programmeur.
- Son type peut être soit celui d'un type du composant-langage, soit celui d'un composant-description du composant-langage.
- Les modificateurs sont celui du composant-langage. Il ne peut y en avoir aucun, un seul ou plusieurs.
- La syntaxe de représentation d'un attribut est la suivante : [modificateur(s)] type_de_l'attribut nom_de_l'attribut. La présence d'attributs dans une description n'est pas obligatoire.

1.3.3.1.3 Compartiment méthodes

- Le nom de la méthode est défini par le programmeur.

- Son type de retour peut être soit celui d'un type du composant-langage, soit celui d'un composant-description du composant-langage. Il peut ne pas y en avoir.
- Les modificateurs sont celui du composant-langage. Il ne peut y en avoir aucun, un seul ou plusieurs.
- Les paramètres de la méthode suivent les mêmes règles que les attributs d'une description.
- La syntaxe de représentation d'une méthode est la suivante : [modificateur(s)] [type_de_retour_de_la_méthode] nom_de_la_méthode ([paramètre(s)]). La présence de méthodes dans une description n'est pas obligatoire.

2. L'outil OFL- ML

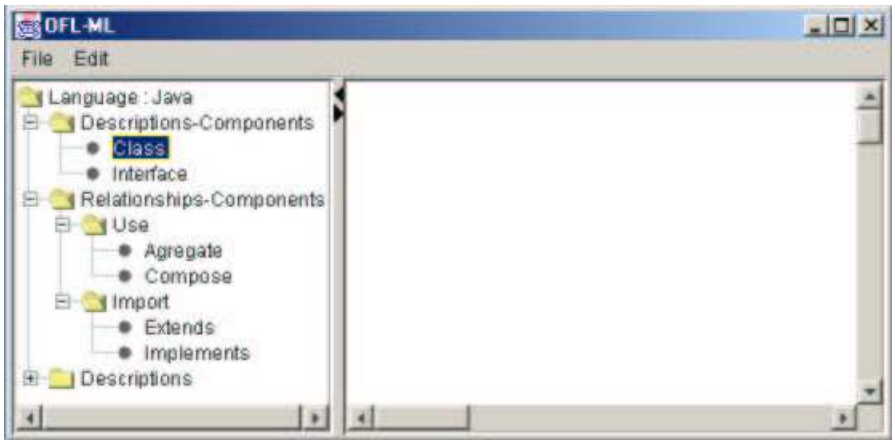
Le cahier des charges, qui nous a été fourni au commencement du projet, ne spécifiait pas les propriétés de l'interface graphique. Notre rôle a donc été de la concevoir, et ce pour une utilisation intuitive. Nous présenterons ici l'outil et ses fonctionnalités (imposées par le cahier des charges) : la création et la représentation graphique de descriptions, leur modification…

2.1 Fonctionnement de l'outil

2.1.1 Initialisation

L'outil est initialisé à son ouverture avec plusieurs paramètres : le nom du composant-langage, les composants-descriptions et les composants-relations (voir ci-dessous). Ces paramètres proviennent d'un fichier créé par l'équipe du projet OFL-Meta.

Arborescence initialisée au lancement de l'outil.

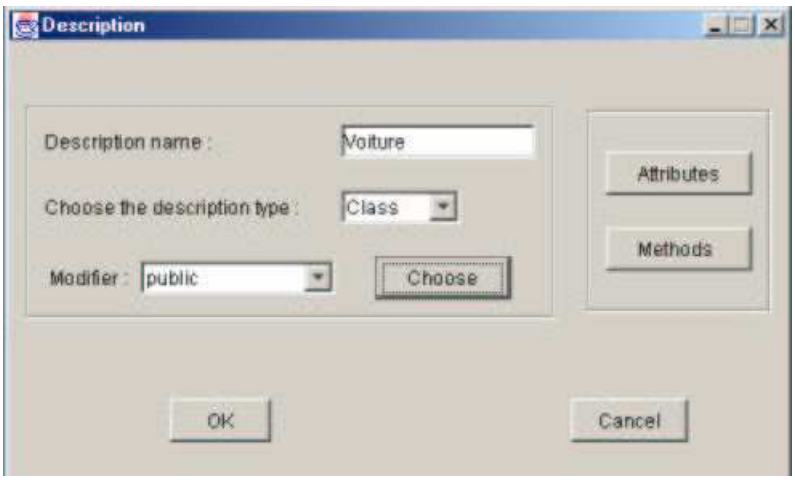


Ce fichier contient le nom du composant-langage (ici C++), le nom des composants-descriptions et des composants-relations du composant-langage. Toutes ces informations peuvent donc être différentes d'un fichier à un autre, sans altérer le fonctionnement de l'outil.

Nous avons pour ceci créé une classe LoadObject (voir annexe). Cette classe charge et sauvegarde ces différentes informations. Ces informations sont stockées dynamiquement.

2.1.2 Création d'une description

Pour la création d'une description, l'utilisateur doit tout d'abord choisir son type (ici : Class ou Interface). Il a ensuite la possibilité de saisir son nom, ses modificateurs et d'y ajouter des attributs et des méthodes.



Si l'utilisateur n'a pas sélectionné le bon type de sa description, il a la possibilité de le modifier grâce à une liste déroulante qui contient tous les types de composants-descriptions (elle est initialisée avec le contenu du fichier).

Les modificateurs peuvent changer selon le composant-langage. C'est pour cela que le choix des modificateurs se fait un à un (bouton Choose) par le biais d'une liste initialisée dynamiquement.

Un même nom ne peut pas être affecté à deux descriptions différentes. Si c'est le cas, un message d'erreur apparaît.

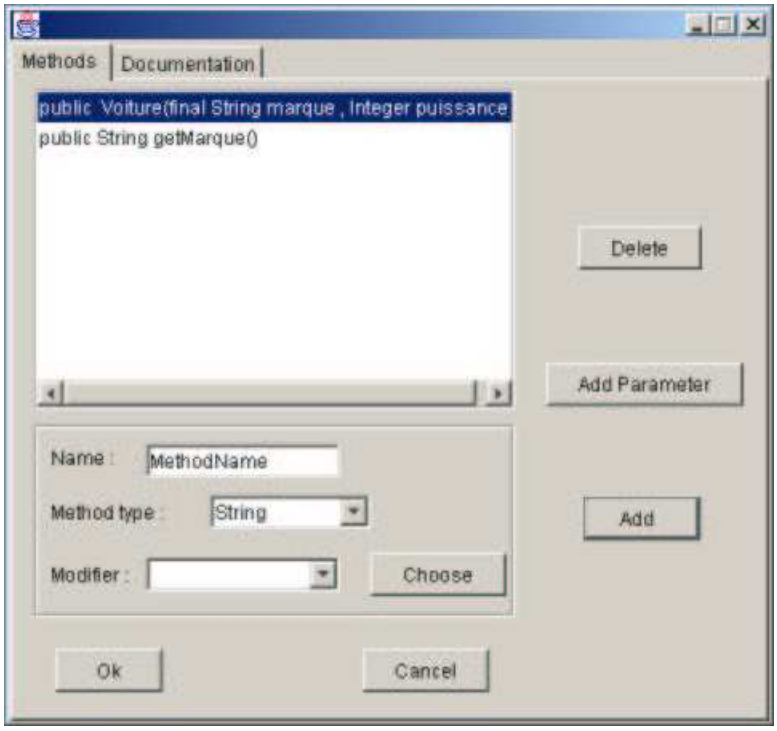
Dès cet instant, l'utilisateur peut alors choisir de créer sa description, ou d'y ajouter des attributs et/ou des méthodes. Supposons qu'il veuille ajouter des méthodes.

2.1.3 Ajout de méthodes

L'utilisateur doit saisir le nom de la méthode dans un champ de texte. Le choix des modificateurs fonctionne sur le même principe que les modificateurs de la description (cf. 2.1.2).

Deux points nous paraissent intéressants. Tout d'abord, l'utilisateur peut choisir le type de sa méthode par le biais d'un menu déroulant. Ce menu déroulant est initialisé dynamiquement par le fichier et la classe LoadObject. De plus il contient le nom de la description en cours de création et le nom de toutes les descriptions créées auparavant. Par ailleurs, pour le choix du type de retour et le type des paramètres de la méthode, la présence des noms de toutes les descriptions était obligatoire et primordiale. Ces deux menus déroulants évoluent donc en fonction des descriptions créées.

Le deuxième point est l'ajout de paramètres à la méthode. Un paramètre est constitué de trois champs : son/ses modificateur(s), son type, son nom. Le choix des paramètres suit le même fonctionnement que le choix des propriétés (son/ses modificateur(s), son type de retour, son nom) d'une méthode.



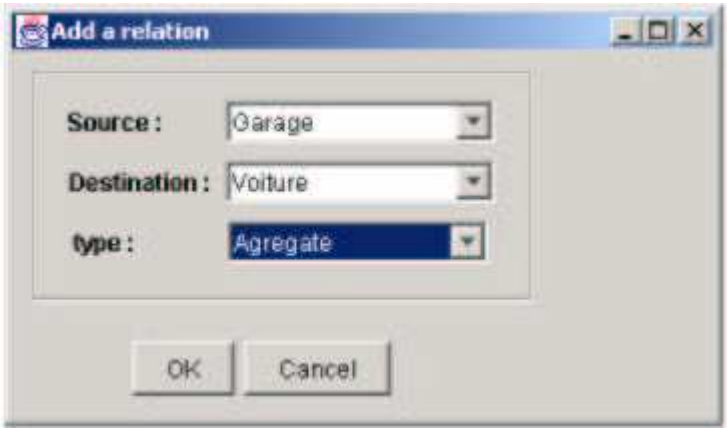
Lorsque l'utilisateur ajoute la méthode à la description, elle est enregistrée et s'affiche au coté des autres méthodes dans une liste. Cette liste permet à l'utilisateur de visualiser les différentes méthodes avec leurs paramètres. Il a la possibilité d'en effacer ou d'en modifier une.

Les attributs fonctionnent sur les mêmes principes que les paramètres.

2.1.4 Création de relation inter-descriptions

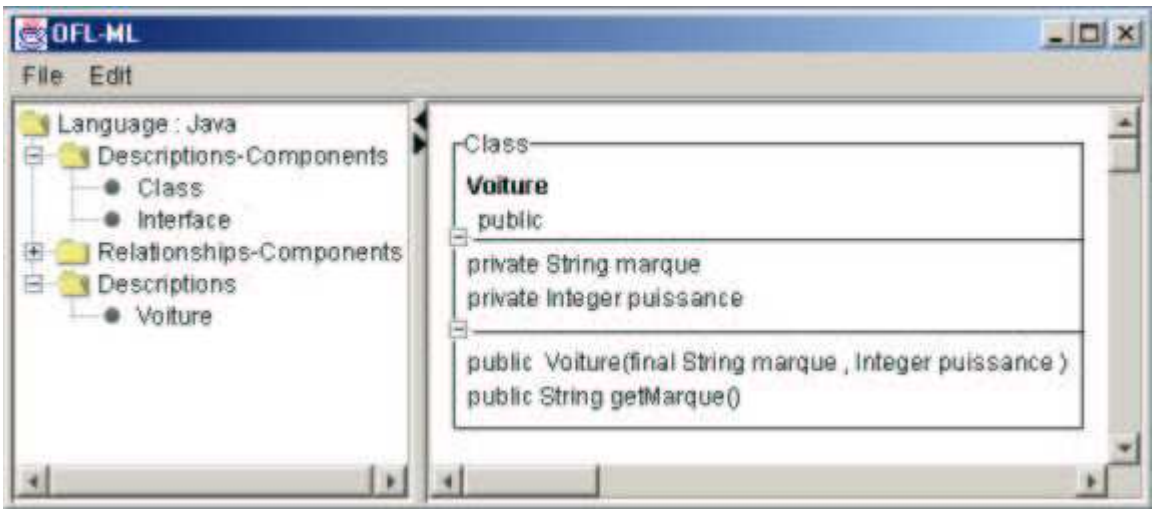
L'utilisateur choisit un type de relation dans les composants-relations. Il sélectionne ensuite les deux descriptions sur lesquelles il veut appliquer la relation. Un changement du curseur lui signale qu'il a sélectionné soit la description-source, soit la description-cible.

La relation est alors créée et une fenêtre apparaît pour que l'utilisateur confirme ou modifie (changement de description-source et/ou de description-cible) ses choix.



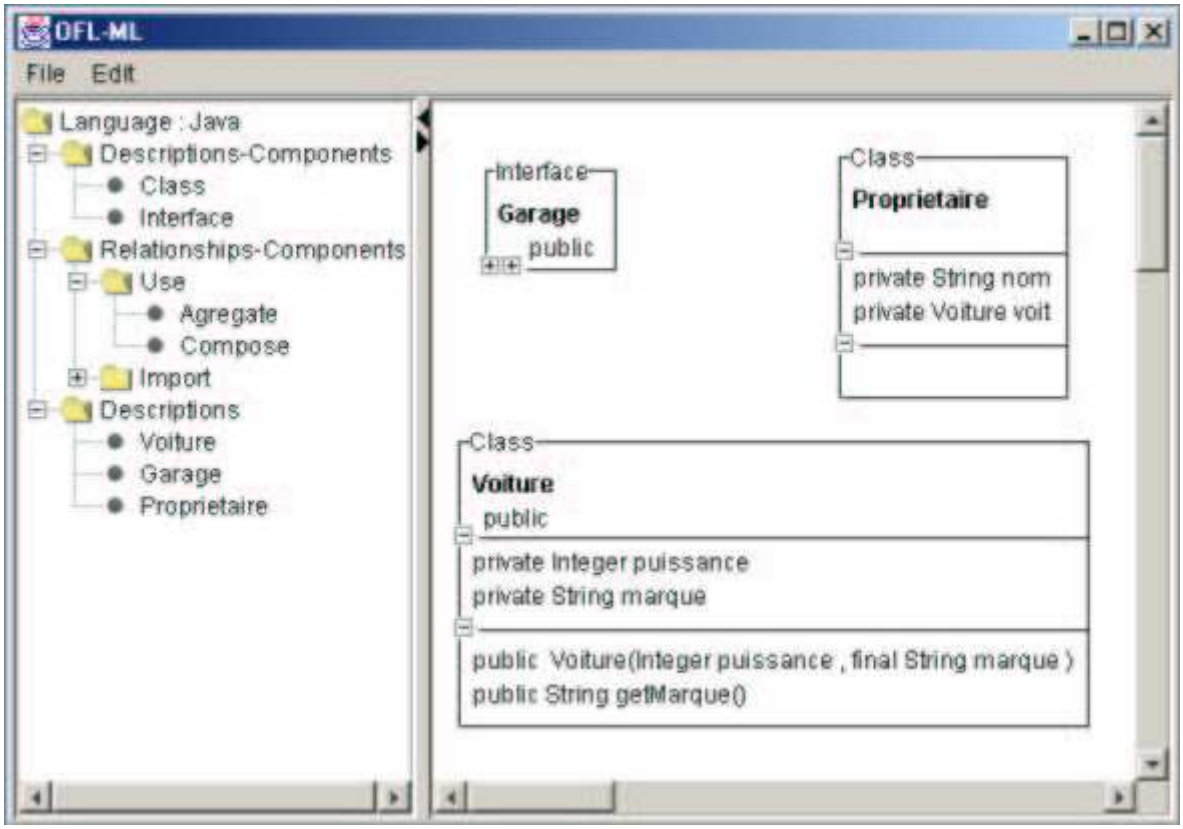
2.2 Représentation graphique des descriptions

Lorsque l'utilisateur a créé sa description, la représentation UML de celle-ci s'affiche dans la partie droite de la fenêtre principale, et son nom est ajouté à l'arbre (ici : nous avons créé une description Voiture, avec deux attributs : marque et puissance, un constructeur Voiture et une méthode getMarque()).



L'utilisateur a la possibilité de modifier une description. Un double clic sur celle-ci ouvre les fenêtres précédemment vues, initialisées avec les informations de cette description. L'utilisateur peut alors rajouter/ effacer/ changer des attributs et des méthodes. Il peut aussi modifier le type, les modificateurs ou tout simplement le nom de la description. Lorsque l'utilisateur confirme tous les changements à effectuer, alors ceux-ci sont répercutés à la fois sur la représentation graphique de la description, mais aussi dans l'arborescence, si le nom a été modifié.
Les différents vecteurs de sauvegarde sont eux aussi modifiés. Toutes les informations concernant la description sont mises à jour.

L'utilisateur peut créer autant de descriptions qu'il le souhaite. Dans un souci de clarté de la partie graphique, les descriptions peuvent être déplacées par un simple » drag and drop » et il est possible de cacher leurs attributs et/ou leurs méthodes.



2.3 Etat du projet

Toute la partie graphique de l'outil concernant les descriptions a été programmée et est fonctionnelle. Il reste à traiter le dessin des relations inter-descriptions. Il existe deux relations : Use et Import. La première correspond à la relation d'utilisation, et la seconde à la relation d'importation.

Les relations Use seraient représentées de la manière suivante :

description-source —Nom de la relation◆— description-cible

et pour les relations Import :

description-source —Nom de la relation→ description-cible

Les noms des relations sont affichés afin de les différencier entre elles, étant donné qu'il peut exister plusieurs relations d'un même composant-relation.
Les relations seraient dessinées de telle sorte qu'elles ne chevauchent pas des descriptions, ce dans un souci d'ergonomie de l'interface graphique.

De plus, il n'y a pour pas le moment de sauvegarde possible. Celle-ci s'effectuerait dans un fichier au format XML.

Conclusion

OFL-ML permet au programmeur de créer facilement des applications grâce à une interface graphique. L'outil permet au programmeur de dessiner des descriptions et des relations inter-descriptions à partir de composants-descriptions et de composants-relations. Ces composants ont été créés au préalable dans un outil appelé OFL-Meta.

Ce projet nous a permis, certes d'approfondir nos connaissances du langage Java, mais aussi et surtout de comprendre le fonctionnement d'un langage de programmation orienté objet, et comment il se constitue.

Nous envisageons prochainement de publier cet outil sur le Web.

Vous pouvez nous contacter aux adresses e-mails suivantes :
Fabienne KULAKOWSKI : fabkula@yahoo.fr
Jérôme BROCCOLICCHI : broccolicchi@yahoo.com