RAPPORT DE PROJET

Titre :

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

Auteurs
Alonzo Laurent
Colletin Guillaume
Garoste Fanny

Encadreurs
Crescenzo Pierre
Lahire Philippe

Table des matières

1. Présentation du contexte

1.1 Le laboratoire I3S

Le laboratoire I3S (Informatique, Signaux et Systèmes de Sophia-Antipolis) est une Unité Mixte de Recherche (UMR 6070) associant l'Université de Nice Sophia-Antipolis (UNSA) au CNRS.

Ses activités couvrent de nombreux domaines de l'informatique, de l'automatique et du traitement du signal et de l'image. Elles sont menées par environ 150 personnes. Les recherches réalisées, de caractères fondamental et appliqué, donnent lieu à des collaborations importantes aux niveaux régional, national et international.

L'I3S bénéficie de l'environnement exceptionnel de Sophia-Antipolis, avec en particulier l'ESSI (Ecole Supérieure en Sciences Informatiques), l'ESINSA (Ecole Supérieure d'Ingénieurs de Nice Sophia-Antipolis), l'institut EURECOM, le centre INRIA (Institut National de Recherche en Informatique et en Automatique), le CNET (Centre National d'Etudes en Télécommunications), le CMA (Centre de Mathématiques Appliquées de l'Ecole Nationale Supérieure des Mines de Paris) et de nombreuses entreprises.
D'autre part, les membres d'I3S participent de façon active à plusieurs formations de troisième cycle regroupées au sein de l'Ecole Doctorale STIC, et à un moindre titre l'école doctorale de mathématiques.

De par ses thèmes de recherche, sa production scientifique, et ses nombreuses collaborations dont beaucoup entrent dans le cadre de projets européens et de contrats industriels, le laboratoire I3S contribue à faire de la technopole de Sophia-Antipolis un pôle en sciences et technologies de l'information.

Le laboratoire I3S regroupe trois équipes de recherche faisant partie de l'unité de recherche de Sophia-antipolis du centre de l'**INRIA** Sophia-Antipolis.

Ses activités tournent autour de 5 thèmes de recherche :

1. architecture logicielle et matérielle
2. Images numériques et vidéos
3. Informatique fondamentale et applications
4. Sciences et techniques du logiciel *
5. Signal robotique, communication, contrôle et optimisation

* Le projet sur lequel nous avons travaillé, le modèle OFL, a fait l'objet d'une thèse (décrite au chapitre suivant). Ses concepteurs et toutes les personnes qui travaillent dessus font partie de l'équipe composant le projet OCL.
Lui-même est un des projets du thème de recherche *Sciences et techniques du logiciel*.

1.2 les unités de recherche à L'INRIA

Créé en 1967 à Rocquencourt près de Paris, l'INRIA (Institut national de recherche en informatique et en automatique) est un établissement public à caractère scientifique et technologique (***EPST***) placé sous la double tutelle du ministre de la recherche et du ministère de l'économie, des finances et de l'industrie.

L'INRIA, composé de 6 unités de recherche a développé, depuis sa création, son implantation sur le territoire national.

Son ambition d'être au plan mondial**,** un institut de recherche au cœur de la société de l'information le pousse à poursuivre sa politique d'ouverture et de partenariats avec les autres organismes de recherche, les universités et les grandes écoles.

Plus de la moitié des recherches de l'institut (47 projets de recherche sur 87) sont aujourd'hui communes avec d'autres établissements (universités, grandes écoles et organismes de recherche), et plus du quart sont dirigées par des chefs de projets relevant d'autres établissements.

Sa volonté est de mettre en réseau des compétences et des talents de l'ensemble du dispositif de recherche français dans le domaine des STIC. Ce réseau permet de mettre l'excellence scientifique au service des progrès technologiques, créateurs d'emplois, de richesse et de nouveaux usages répondant à des besoins sociaux économiques.

Son organisation décentralisée en 6 unités de recherche, ses petites équipes autonomes et évaluées régulièrement, permettent à l'INRIA d'amplifier ses partenariats; il renforce son implication dans les travaux de valorisation des résultats de recherche et le transfert technologique : 600 contrats R&D avec l'industrie et un peu moins d'une cinquantaine de sociétés sont issues de l'INRIA.

Quelques chiffres (décembre 2000)

Ressources budgétaires

- budget total: 577 MF HT
- ressources propres : 1/4

Ressources humaines

- titulaires INRIA : 724
- post-Doctorants et Contractuels: 256
- doctorants : 550
- chercheurs et enseignants d'autres organismes : 230
- conseillers, collaborateurs divers et invités : 430

1.3 Thèse OFL

OFL est un modèle méta objet pour paramétrer la sémantique opérationnelle des langages à objets et le comportement des langages de programmation à objets les plus courants comme Java, C++, Eiffel, etc.

OFL tourne autour de trois concepts qui sont :

- Les concepts-descriptions permettant de générer des composants-descriptions qui offrent une description des méta-classes du langage
- Les concepts-relations pour décrire des composants-relations qui vont permettre de définir les méta-relations du langage.
- Les concepts-langages permettant de rassembler pour un langage donné l'ensemble de ses concepts-descriptions et de ses concept-relations.

Les concepteurs du projet OFL sont :

Adeline Capouillez
Robert Chignoli
Pierre Crescenzo
Philippe Lahire

Plus précisément, M. Pierre Crescenzo, docteur en informatique de l'UNSA et encadreur durant toute la durée de ce projet avec M. Philippe Lahire, a effectué une thèse sur la méta-programmation selon le concept OFL.

Le mémoire de thèse peut être téléchargé et consulté sur : http://www.crescenzo.nom.fr

2. Présentation du sujet

2.1 L'existant

Il existe certains logiciels qui permettent de programmer graphiquement une application, comme RATIONAL ROSE ou ObjectEering par exemple, qui offrent à l'utilisateur la possibilité de créer un schéma basé sur UML et ensuite de générer le code correspondant au schéma. Bien qu'efficaces dans leur domaine d'application, ces logiciels sont limités par la sémantique opérationnelle d'un langage et ne proposent à son utilisateur que l'ensemble des spécificités propres à ce langage. De telles applications restent beaucoup trop générales, contrairement à l'application qui nous a été demandé de développer, qui met à la disposition de chaque utilisateur des «Component-Relationship» et des «Component-Descriptions» spécifiques à ses besoins.

Notre objectif n'était pas d'essayer de créer un éditeur graphique permettant de la modélisation UML pour rivaliser avec le grand nombre de logiciels existant aujourd'hui sur le marché (Rational Rose, ObjectEering, …) mais de développer une interface graphique simple, intuitive et surtout évolutive.

Elle se devait d'être 'générique' c'est à dire indépendante d'un langage particulier et capable de proposer à son utilisateur les seules fonctionnalités décidées préalablement par le travail du méta programmeur.

Ce concept nouveau permet une programmation plus rapide et plus propre du code, mais offre aussi une plus grande souplesse de réutilisation que des éditeurs graphique classiques.

2.2 Programmation et méta-programmation

Le terme de '**méta programmeur**' désigne le développeur non pas d'une application mais d'un langage de programmation, avec ses propres spécificités.

L'interface que nous avons développé est appelée interface OFL-ML, ML faisant référence au langage de modélisation UML dont elle conserve le formalisme graphique.

OFL-ML s'interface avec l'application OFL-META développée par une seconde équipe, et destinée, quant à elle, à la méta programmation. Cette dernière va permettre au méta-programmeur , par l'intermédiaire de menus proposant des choix multiples, de développer un langage de programmation personnalisé, c'est à dire, offrant des caractéristiques représentant les choix du méta programmeur en termes de concepts-langages, concepts-descriptions et concepts-relations (cf. thèse OFL).

Ainsi le méta programmeur décrit son propre langage grâce à l'outil OFL-META.
Par la suite tous ses choix seront enregistrés et classés dans un fichier selon une convention XML.

Ce dernier va ensuite être chargé puis analysé par OFL-ML qui met ainsi à disposition les descriptions et les relations autorisées dans ledit langage.

OFL-ML va ainsi reprendre les choix du méta programmeur et offrir les spécificités propres au langage qui vient d'être défini.
Il ne reste plus au programmeur qu'à utiliser l'interface pour le développement de son application.

On voit donc bien ici que le méta programmeur et le programmeur bien que travaillant à un niveau différent doivent collaborer pour obtenir au final un langage adapté au mieux au besoin du projet qu'ils ont à réaliser ensemble.

Pour résumer, OFL-ML est une interface souple, flexible, capable d'offrir à ses utilisateurs l'ensemble des spécificités d'un langage particulier. Cela signifie également que OFL-ML n'est pas bridée par des contraintes imposées par un langage donné et n'est limitée que par l'imagination de ses utilisateurs.

OFL-ML est donc une interface ouverte dans le sens où elle ne se limite plus à un langage existant comme c'est le cas pour les applications actuelles.
Elle est capable de s'adapter au mieux aux exigences d'un utilisateur et d'offrir ainsi la puissance pour cibler de manière efficace les spécificités nécessaires au développement d'une application donnée en créant un nouveau langage ou en modifiant un langage existant..

3. Découverte du modèle OFL

3.1 Présentation du concept OFL

Face à la multitude des langages objets et à leurs particularités, le nouveau concept OFL propose à un utilisateur un méta langage qui englobe tous les langages objets existants, lequel lui permet de caractériser et de créer son propre langage. L'utilisateur a ainsi la possibilité de créer un langage avec un comportement qui lui est spécifique et qui correspond exactement à l'attente de celui-ci.
Le concept OFL est donc indépendant d'un langage particulier et englobe tous les composants des langages objets existants. Ces composants permettent de paramétrer les caractéristiques et le comportement des éléments de type « description » et « relation ».
La notion de « description » décrit les objets d'une application et la notion de « relation » décrit les liens entre les éléments de type « description ».
Ce concept nécessite le développement de deux applications distinctes que sont la création d'un langage à partir d'un ensembles de composants (les paramètres hyper génériques) et la création d'un programme à partir du langage créé.

Ces deux applications décrivent trois niveaux :

- Le niveau « **OFL** » qui contient toutes les caractéristiques possibles d'un langage objet.Ce niveau est un méta-modèle au niveau « LANGAGE », une sorte de « créateur de langage ». Il contient des éléments de type OFL-concept (Concept-Description, Concept-Relationship et Concept-Language) qui permettent de paramétrer le niveau « LANGAGE ».

- Le niveau « **LANGAGE** » qui décrit un langage objet particulier. Il spécifie les caractéristiques des différents éléments de type « description » et « relation » que l'on peut trouver dans ce langage grâce à des éléments de type OFL-composant (Component-Description, Component-Relationship et Component-Language). Ces éléments de type OFL-composant sont donc des instances des éléments de type OFL-concept vu précédemment.

- Le niveau « **APPLICATION** » qui possède tous les objets créés ainsi que les relations entre ces objets (ce sont les éléments Description et Relationship)L'utilisateur peut créer une application en créant des éléments de type « description » et « relation » et doit respecter les caractéristiques décrites au niveau « LANGAGE ».Les éléments de ce niveau sont donc des instances des éléments de type OFL-composant vus précédemment.

Afin de bien comprendre comment ces trois niveaux interagissent entre eux, nous avons regroupé les différents éléments de chaque niveau dans un même schéma.
Les classes représentées par des rectangles et une écriture normale font partie du niveau « OFL ».
Les classes représentées par des rectangles et une écriture en italique font partie du niveau « LANGAGE ».
Les classes représentées par des formes arrondies font partie du niveau APPLICATION ».
Les flèches continues représentent l'héritage et les flèches en pointillés représentent l'instanciation.

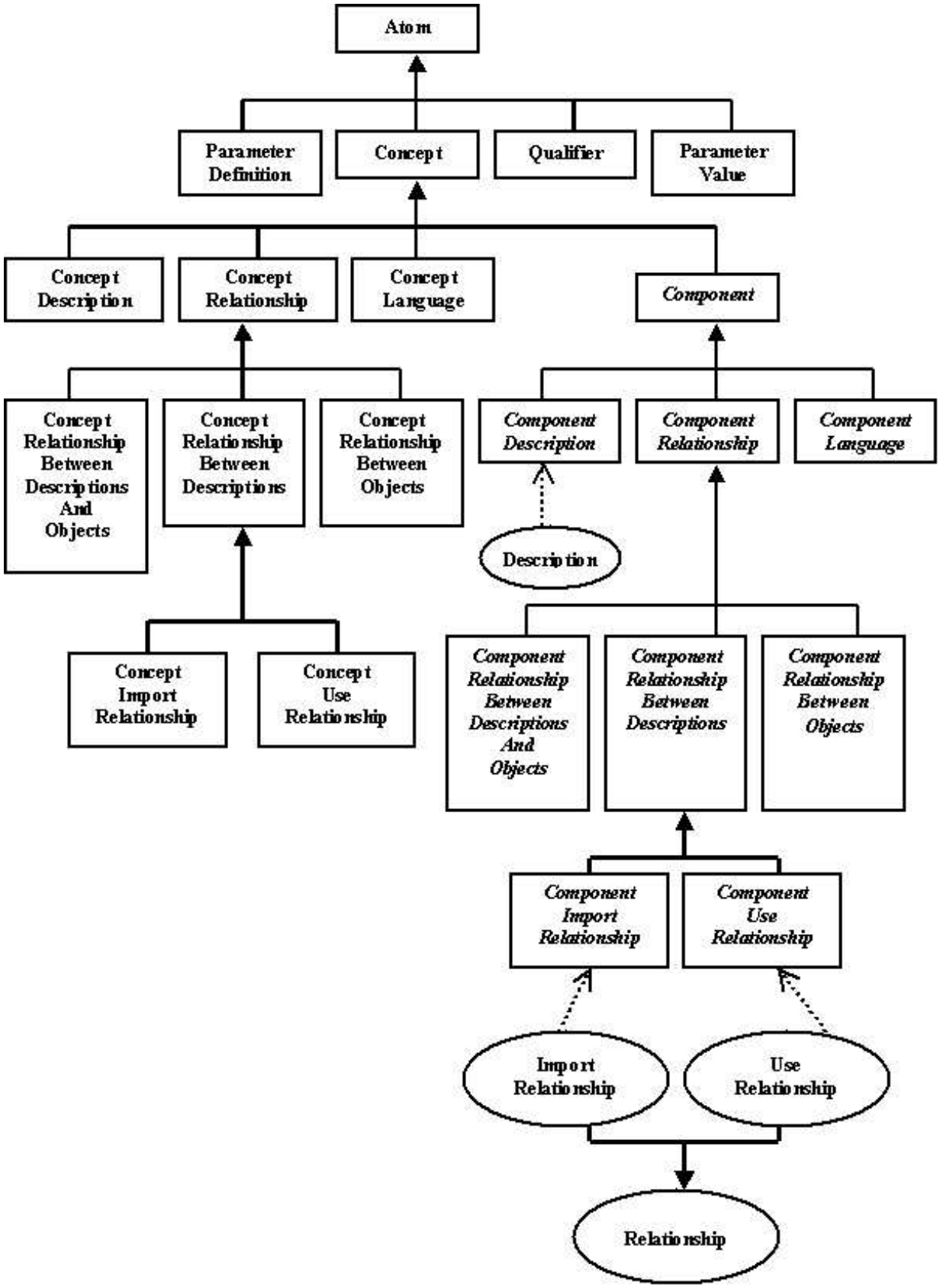

figure 1 : schéma de classe des trois niveaux.

Pour des raisons de structuration (contraintes de cohérence), les éléments de type OFL-composant ne sont pas directement instance des éléments de type OFL-concept; chaque OFL-composant possède un attribut de type OFL-concept dont il est conceptuellement « instance ».

Notre projet porte sur le développement de l'application permettant de créer un programme à partir d'un langage créé. Il est donc fondamental de bien comprendre le concept de « description », de « relation » et de « langage » et d'analyser les paramètres qui décrivent leur comportement.

3.2 Le concept « description »

3.2.1 Description

Le concept « description » symbolise la notion de classe. On retrouve donc ce concept dans les trois niveaux : Concept-Description -> Component-Description -> Description.
Il est maintenant important de bien comprendre le contenu d'un élément de type Component-Description. Ces éléments de type Component-Description sont des méta classes, ils permettent en effet de créer des classes qui respectent un comportement défini par les paramètres de ces éléments.
Nous allons donc présenter dans la section suivante une liste brève des paramètres d'un élément de type Component-Description. Cette liste regroupe les principaux paramètres que nous avons dû analyser lors de la création d'une description.
Chaque paramètre a un nom et un type (qui est un ensemble de valeurs possibles de type String).

3.2.2 Paramètres utilisés d'un élément de type Component-Description

- **Valid source relationship-components** : Ce paramètre regroupe l'ensemble des relations valides qui peuvent partir d'une description.

- **Valid target relationship-components** : Ce paramètre regroupe l'ensemble des relations valides qui peuvent partir d'une description.

- **Valid qualifiers** : Ce paramètre définit l'ensemble des qualifieurs que l'on peut trouver devant une classe.
Exemple de valeurs : {<description, {''static''}>}.

- **Name** : Ce paramètre définit le type des descriptions que l'on peut créer.
Exemple de valeurs : « class » et « interface ».

- **Genericity** : Ce paramètre permet de définir ou non des descriptions génériques (template).
Exemple de valeurs : « true ».

- **Attribute** : Ce paramètre permet de savoir si les attributs sont autorisés ou non à l'intérieur d'une description.
Exemple de valeur : « allowed ».

- **Method** : Ce paramètre permet de savoir si les méthodes sont autorisées ou non à l'intérieur d'une description.
Exemple de valeur : « forbidden ».

- Overloading : Ce paramètre permet de savoir si la surcharge est autorisée ou non.

- **Encapsulation** : Ce paramètre permet de savoir si l'encapsulation d'attributs et de méthodes dans les descriptions est possible ou non.
Exemple de valeur : <true, true>.

3.3 Le concept « relation »

3.3.1 Description

Le concept « relation » symbolise la notion de lien entre classes. On retrouve donc ce concept dans les trois niveaux :
Concept-Relationship -> Component-Relationship -> Relationship.
Il est maintenant important de bien comprendre le contenu d'un élément de type Component-Relationship. Ces éléments de type Component-Relationship sont des méta relations, ils permettent en effet de créer des relations qui respectent un comportement défini par les paramètres de ces éléments.
Nous allons donc présenter dans la section suivante une liste brève des paramètres d'un élément de type Component-Relationship. Cette liste regroupe les principaux paramètres que nous avons dû analyser lors de la création d'une relation.
Chaque paramètre a un nom et un type (qui est un ensemble de valeurs possibles de type String).

3.3.2 Paramètres utilisés d'un élément de type Component-Relationship

- **Valid source description-concepts** : Ce paramètre regroupe l'ensemble des descriptions valides depuis lesquels une relation peut partir.
Exemple de valeur : {Class, Interface}.

- **Valid target description-concepts** : Ce paramètre regroupe l'ensemble des descriptions valides depuis lesquelles une relation peut arriver.

- **Valid qualifier** : Ce paramètre définit l'ensemble des qualifieurs que l'on peut trouver devant une relation.

- **Name** : Ce paramètre définit le nom de la relation.
Exemple de valeur : « spécialisation ».

- **Kind** : Ce paramètre définit le type des relations que l'on peut créer.
Exemple de valeurs : « import » et « use ».

- **Cardinality** : Ce paramètre permet de définir la cardinalité de départ et d'arrivée d'une relation.
Exemple de valeur : <1, 10>.

- **Repetition** : Ce paramètre permet de définir si un héritage identique est possible ou non.
Exemple de valeur : <forbidden, forbidden>.

- **Circularity** : Ce paramètre permet de définir si un ensemble de relations peuvent décrire un graphe circulaire.
Exemple de valeur : « allowed ».

- **Symmetry** : Ce paramètre permet de définir si une relation est symétrique ou non.
Exemple de valeur : « false ».

- **Adding** : Ce paramètre permet d'ajouter dans la description d'arrivée certaines méthodes de la description de départ.

- **Removing** : Ce paramètre permet de supprimer dans la description d'arrivée certaines méthodes de la description de départ.

- **Renaming** : Ce paramètre permet de modifier dans la description d'arrivée certaines méthodes de la description de départ.

- **Masking** : Ce paramètre permet de masquer dans la description d'arrivée certaines méthodes de la description de départ.

- **Showing** : Ce paramètre permet de rendre visible dans la description d'arrivée certaines méthodes de la description de départ.

- **Redefining** : Ce paramètre permet de modifier une méthode appartenant à la description de départ dans la description d'arrivée.

- **Abstrating** : Ce paramètre permet de rendre abstrait dans la description d'arrivée une méthode concrète de la description de départ.

- **Effecting** : Ce paramètre permet de rendre concret dans la description d'arrivée une méthode abstraite de la description de départ.

3.4 Le concept « langage »

3.4.1 Description

Le concept « langage » représente un langage à part entière. On retrouve ce concept dans les deux premiers niveaux (« OFL » et « LANGAGE ») :
Concept-Language -> Component-Language.

Il est maintenant important de bien comprendre le contenu d'un élément de type Component-Language. Un élément de type Component-Language contient l'ensemble des descriptions et des relations valides avec leurs caractéristiques, il possède une liste d'éléments de type Component-Description et une liste d'éléments de type Component-Relationship. Ce sera donc un élément de type Component-Language que nous récupèrerons afin de pouvoir l'analyser et créer une application conforme à ce langage.
Nous allons donc présenter dans la section suivante une liste brève des paramètres d'un élément de type Component-Language.
Chaque paramètre a un nom et un type (qui est un ensemble de valeurs possibles de type String).

3.4.2 Paramètres utilisés d'un élément de type Component-Language

- **Valid** description-components : Ce paramètre contient la liste des éléments de type Component-Description qui définissent ce langage.

- **Valid relationship-components** : Ce paramètre contient la liste des éléments de type Component-Relationship qui définissent ce langage.

- **Valid relationships** : Ce paramètre contient la liste des départs et des arrivées possibles pour une relation donnée.

- **Name** : Ce paramètre contient le nom du langage créé.

Maintenant que nous avons détaillé les paramètres des éléments méta (OFL-composant) qui caractérisent un langage; nous allons présenter les différents logiciels qu'il faudrait mettre en place afin de gérer le modèle OFL.

3.5 Les applications OFL-META, OFL-DB, OFL-ML et OFL-PARSER

Autour des trois niveaux que nous avons vu précédemment, quatre applications sont nécessaires afin de gérer ce modèle.

- L'application **OFL-META** permet en premier lieu de passer du niveau « OFL » au niveau « LANGAGE ». Cette application permet donc de créer un langage, c'est à dire un élément de type Component-Language.

- L'application **OFL-DB** permet de gérer la conservation d'un modèle défini en utilisant une norme bien précise, en l'occurrence « XML ».

- L'application **OFL-ML** permet de passer du niveau « LANGAGE » au niveau « APPLICATION ». Cette application permet donc à partir d'un élément de type Component-Language de créer un ensemble de descriptions (classes) et un ensemble de relations (liens) entre ces descriptions.

- L'application **OFL-PARSER** permet enfin de traduire l'application décrite dans l'application OFL-ML vers un langage cible. Cette application est une sorte de compilateur d'applications.

Les trois premières applications ont donné lieu à trois projets, développés par des élèves de l'ESSI. Et nous rappelons que notre travail consiste à développer un logiciel permettant de gérer l'application OFL-ML. Nous avons donc travaillé en collaboration avec les deux autres groupes (OFL-META et OFL-DB).
Afin de mieux comprendre les interactions entre ces quatre applications, nous les avons représenté sur un même schéma.

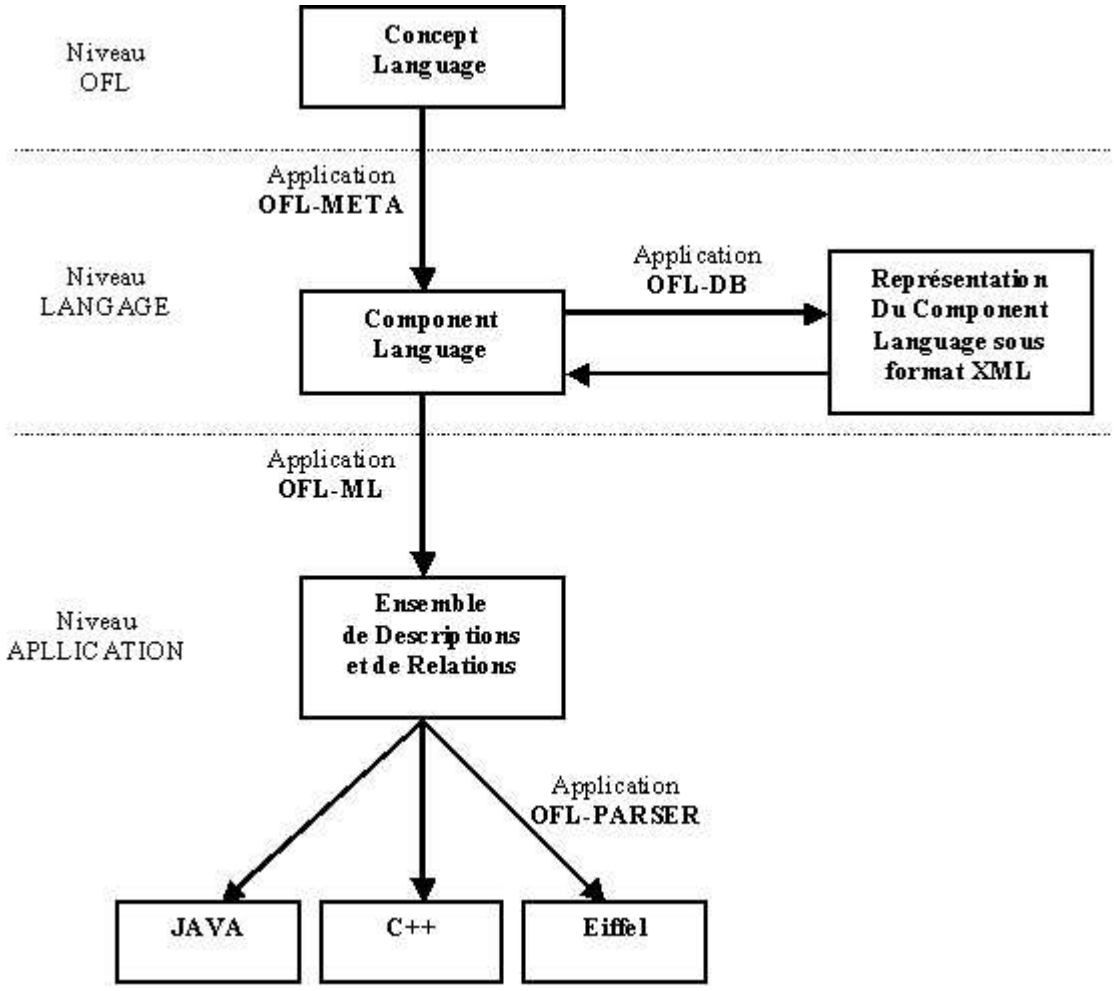

Figure 2 : interactions entre les différentes applications.

Nous avons vu et détaillé le concept OFL ainsi que les applications qui en découlent, nous pouvons maintenant aborder en détail la description du projet et du travail qui nous est demandé.

4. Description du projet et du travail réalisé

4.1 Le cahier des charges

Notre projet a pour but de réaliser un éditeur permettant de programmer graphiquement une application conformément au modèle OFL. La partie graphique de l'application sera basée sur le concept UML.
Suite à ce que l'on a vu dans le chapitre 3, notre application devra être capable de récupérer un élément de type Component-Language (cf. annexe 1 et 2). Après l'analyse de cet élément de type Component-Language, l'application récupèrera la liste des descriptions valides qu'un utilisateur pourra créer (liste de Component-Description); ainsi que la liste de relations valides pour ce langage (liste de Component-Relationship).
Nous distinguerons alors deux types de relations donnant lieu à deux sous listes, les relations de type utilisation (liste de Component-UseRelationship) et les relations de type importation (liste de Component-ImportRelationship).
L'application proposera donc à l'utilisateur la possibilité de créer des éléments de son langage et il convient alors de distinguer deux parties que sont la création d'une description et la création d'une relation.

La création d'une description :
L'utilisateur devra pouvoir créer une nouvelle description simplement en cliquant dans une zone graphique réservée au développement de son application. Un label permettra de saisir le nom de la nouvelle description, laquelle aura les caractéristiques du Component-Description sélectionné dans la liste de tous les éléments de type Component-Description du langage en cours. De plus la description créée sera automatiquement ajoutée à une liste de description en cours afin que l'utilisateur puisse facilement visualiser et retrouver une description créée précédemment.

La création d'une relation :
L'utilisateur devra pouvoir créer une nouvelle relation en définissant un ensemble de descriptions de départs et un ensemble de descriptions d'arrivées. Ces deux ensembles seront créés par simple clic à l'intérieur d'une description et une option permettra d'arrêter la création de l'ensemble en cours. La relation créée aura les caractéristiques du Component-Relationship sélectionné dans la liste de tous les éléments de type Component-Relationship du langage en cours.

Dans les deux cas, les éléments créés devront respecter le comportement décrit par leurs composants respectifs et auront donc des caractéristiques bien précises. Ces caractéristiques sont celles décrites dans la partie 3.2.2, 3.3.2 et 3.4.2 Il faudra donc vérifier la conformité de la création et de la manipulation d'un élément « Description » ou « Relationship » et, éventuellement, avertir par un message l'utilisateur d'une opération non conforme avec les caractéristiques du langage utilisé. Comme pour la création, l'application proposera à un utilisateur la possibilité de manipuler les éléments créés. Il convient là aussi de distinguer deux parties que sont la manipulation d'une description et la manipulation d'une relation.

La manipulation d'une description :
L'utilisateur pourra à tout moment modifier le nom d'une description. Il aura par ailleurs la possibilité d'ajouter, de modifier et de supprimer un attribut ou une méthode. La validité de cette opération dépendra des paramètres Attribute et Method de l'élément Component-Description associé à cette description. De plus l'utilisateur pourra utiliser les notions de généricité, d'encapsulation et de surcharge si cela est autorisé par les paramètres vus dans la partie II.1.
La suppression d'une description donnée entraînera la suppression de toutes les relations qui la référençaient. Celle-ci sera bien évidemment retirée de la liste de toutes les descriptions en cours.
Dans un soucis de lisibilité, l'utilisateur pourra cacher la liste des attributs et la liste des méthodes d'une description.

La manipulation d'une relation :
L'utilisateur pourra sélectionner une relation parmi la liste des relations créées entre deux descriptions données. Dans un soucis de lisibilité, seule la relation sélectionnée sera visible entre deux descriptions données. L'utilisateur pourra visualiser l'ensemble des relations ou bien sélectionner uniquement les relations de type utilisation ou les relations de type importation.
L'utilisateur pourra ajouter une description à l'ensemble des descriptions de départ ou à l'ensemble des descriptions d'arrivée et ce pour une relation sélectionnée.
L'utilisateur pourra supprimer une relation; si celle-ci est constituée de plusieurs descriptions de départ et d'arrivée, alors l'ensemble des liens entre descriptions représentant cette relation devront être effacés.

Plus généralement une interface graphique relativement simple et facile d'utilisation devra être mise en place. Cette interface permettra à l'utilisateur de faire toutes les opérations vues précédemment.

- Un menu contextuel permettra d'accéder aux fonctions classiques (copier, couper, coller, open, save, exit, help …).
- Un arbre structuré permettra d'accéder aux différents éléments de type composant et de visualiser rapidement les éléments de type description déjà créés.
- Une zone graphique permettra de visualiser l'application en cours de création et devra offrir la possibilité d'arranger la disposition des descriptions et des relations (l'utilisateur pourra déplacer la partie graphique d'une description et les liens entre descriptions devront être précis et ordonnés).

Face à la complexité des paramètres des éléments de type composant, il est entendu avec les encadreurs que le but de ce projet n'est évidemment pas de réaliser l'ensemble des fonctionnalités décrites par le modèle OFL, mais de proposer un début d'implémentation d'un logiciel permettant de programmer une application. Notre projet a donc pour but de proposer une première maquette, et pour cela de bien comprendre le concept OFL.
En conclusion, nous avons pour but de réaliser un travail de qualité (code propre avec des commentaires en anglais et une javadoc) plutôt qu'un travail de quantité afin que celui-ci puisse être repris et poursuivis facilement.

Maintenant que le cahier des charges a été présenté, nous allons détailler les grandes étapes que nous avons suivi durant le déroulement de notre projet.

4.2 Les grandes étapes et le déroulement du projet

La réalisation de notre projet a nécessité quatre grandes étapes que sont :

- La compréhension du concept OFL et la réalisation d'une première interface

Dans un premier temps, nous avons dû découvrir et comprendre le concept OFL, ainsi que le travail demandé. Afin de faciliter cette compréhension, nous avons développé en parallèle une première interface permettant de créer des objets graphiques qui symboliseraient par la suite les éléments de type OFL. En effet, cette première maquette permettait de créer des rectangles et des flèches représentant respectivement les éléments de type Component-Description et Component-Relationship.

- L'intégration des éléments OFL dans notre application

Une fois la structure OFL comprise, nous avons implémenté les éléments OFL (les éléments descriptions et relationships) associés aux éléments graphiques vus précédemment (les rectangles et les flèches). A la fin de cette étape un élément OFL avait donc une représentation graphique, laquelle permettait de visualiser leur l'état.
Nous avions donc à ce stade la possibilité d'influencer le comportement simple d'un objet OFL (création, caractéristiques simples, suppression).

- Le développement en parallèle du comportement complexe des éléments OFL

Cette étape a été la plus fastidieuse, en effet nous avons dû nous plonger au cœur du concept OFL afin de comprendre les caractéristiques d'un composant OFL.
Ces caractéristiques sont comme nous l'avons vu dans les chapitres précédents dictées par les différentes valeurs des paramètres d'un composant OFL (cf. chapitre 3.2.2, 3.3.2 et 3.4.2). Chaque paramètre a été développé indépendamment, ce qui nous a permis de travailler en parallèle.

- La réalisation de test sur notre application

Durant tout le développement de l'étape 3 et 4, nous avons testé notre application en récupérant un élément de type Component-Language réalisé par l'application OFL-META. Ces tests ont permis de valider notre application par un élément de type Component-Language valide.

Après avoir détaillé les grandes étapes du développement de notre application, nous allons maintenant présenter notre travail.

4.3 Présentation de notre application

Nous avons développé une application en Java qui implémente la plus part des points vus dans le cahier des charges (chapitre 4.1), dont voici quelques aperçus.

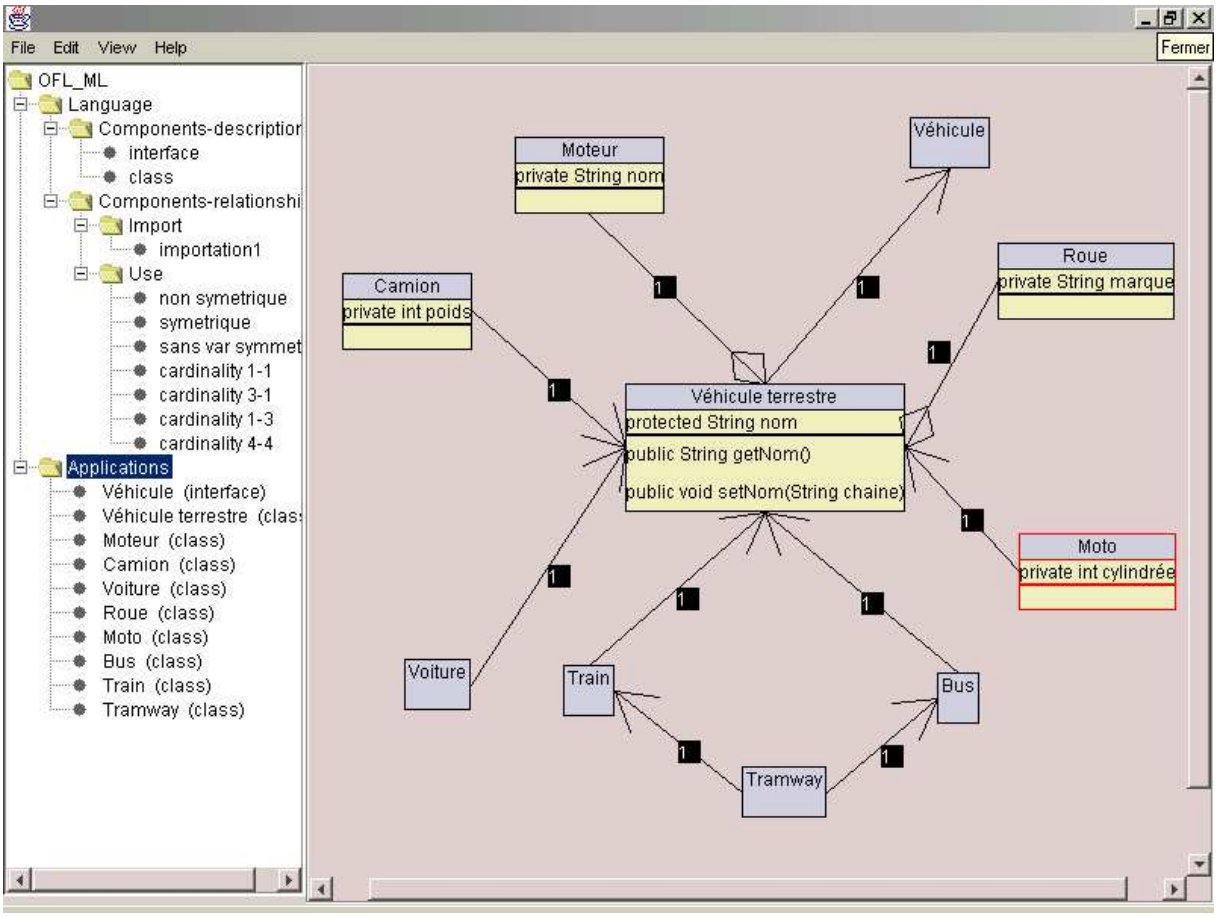

Figure 3 : Création d'une application avec notre logiciel.

Dans l'exemple de la figure 3, l'arbre (OFL-ML) propose à l'utilisateur l'ensemble des descriptions et des relations décrivant ce langage. Cet arbre est construit à partir d'un élément de type Component-Language récupérer en entrée du programme. Nous pouvons voir que le nœud « OFL-ML » possède deux fils, un nœud contenant la description du langage et un autre nœud décrivant l'ensemble des descriptions déjà créées par l'utilisateur. Le nœud « Language » propose la liste des descriptions, la liste des relations de type use et la liste des relations de type import valides pour ce langage.
La zone graphique permet à l'utilisateur de créer son application. Les descriptions possèdent trois parties, une partie contenant leur nom, une partie contenant leurs attributs et une partie contenant leurs méthodes. Nous détaillerons par la suite les fonctionnalités possibles sur une description donnée. En ce qui concerne les relations, nous avons distingué graphiquement une relation de type use par rapport à une relation de type import. Cette représentation a été imposée par le concept OFL. Les flèches traditionnelles représentent donc des relations de type « use » et les flèches à base de losange

représentent des relations de type « import ». Là aussi nous verrons ultérieurement les fonctionnalités offertes pour une relations donnée.
Le menu contextuel offre à l'utilisateur toutes les fonctionnalités classiques d'une application (copier, couper, coller), ainsi que d'autres plus spécifiques qui seront abordées plus tard.
Maintenant qu'une première approche de notre logiciel est faite, nous allons rentrer dans les détails en commençant par les fonctionnalités autour des éléments de type descriptions.

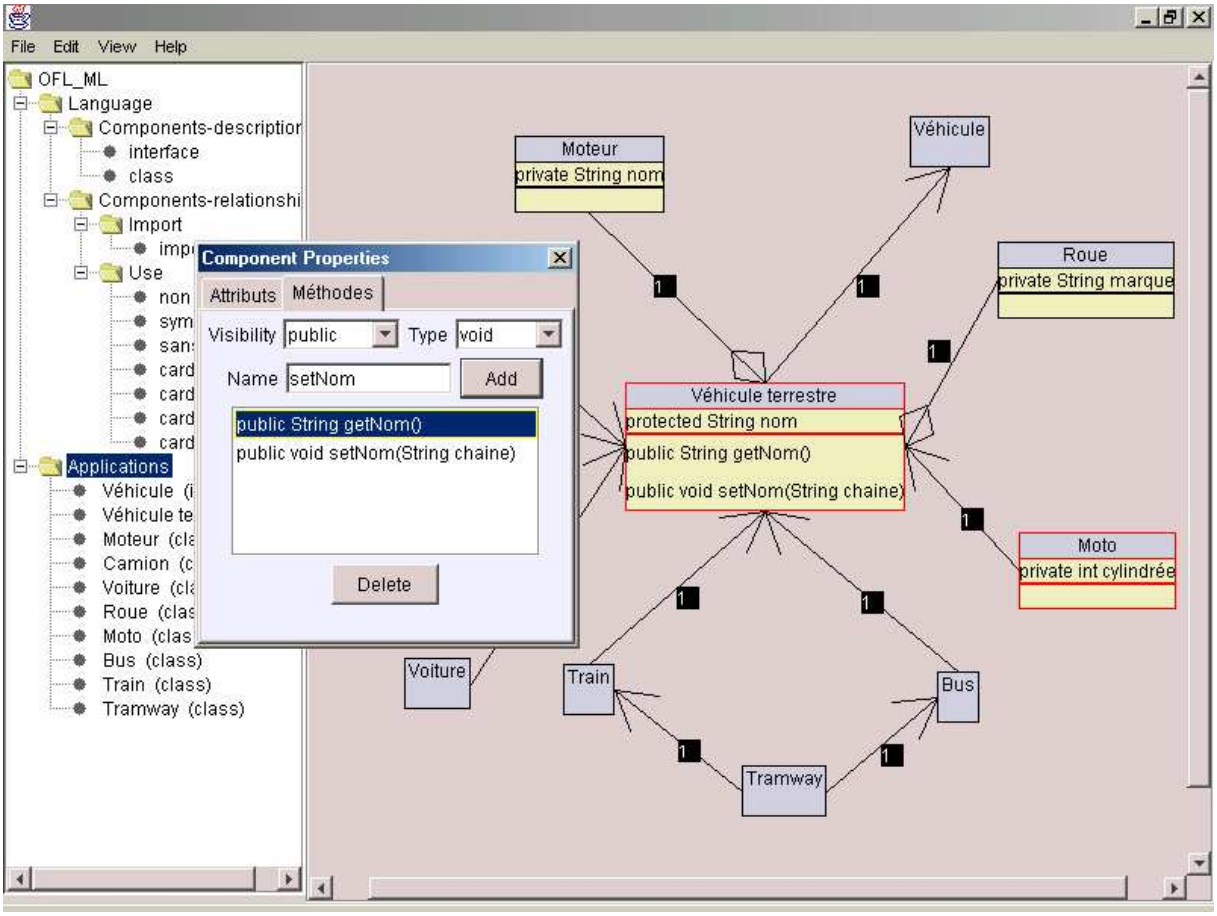

Figure 4 : Visualisation de la propriété d'une description donnée.

Une fenêtre « Component Properties » décrivant les propriétés d'une description peut être activée par l'utilisateur. Il suffit de choisir l'option « properties » du menu proposé suite au clic droit de la souris sur une description. Nous reviendrons sur ce menu plus tard.
Cette fenêtre permet à l'utilisateur de créer, supprimer ou modifier des attributs et des méthodes pour une description donnée. Les fonctionnalités autour d'un attribut et d'une méthode sont les mêmes et nous ne présenterons donc que le fonctionnement basé autour d'une méthode.
Deux listes permettent de choisir respectivement la visibilité d'une fonction et le type de retour de celle-ci. Une zone est réservée au choix du nom et un bouton « **Add** » permet de créer une nouvelle méthode avec les paramètres précédemment choisis par l'utilisateur.
Un bouton « **Delete** » permet d'effacer une méthode sélectionnée parmi la liste des méthodes courantes. Dans notre exemple (cf. Figure 4), la méthode getNom() est sélectionnée.

Une autre fenêtre « Parameters » permet d'ajouter, de modifier ou de supprimer des paramètres à une méthode donnée. Cette fenêtre peut être accédée simplement par un double clic de la souris sur une méthode parmi la liste. (cf. Figure 5).

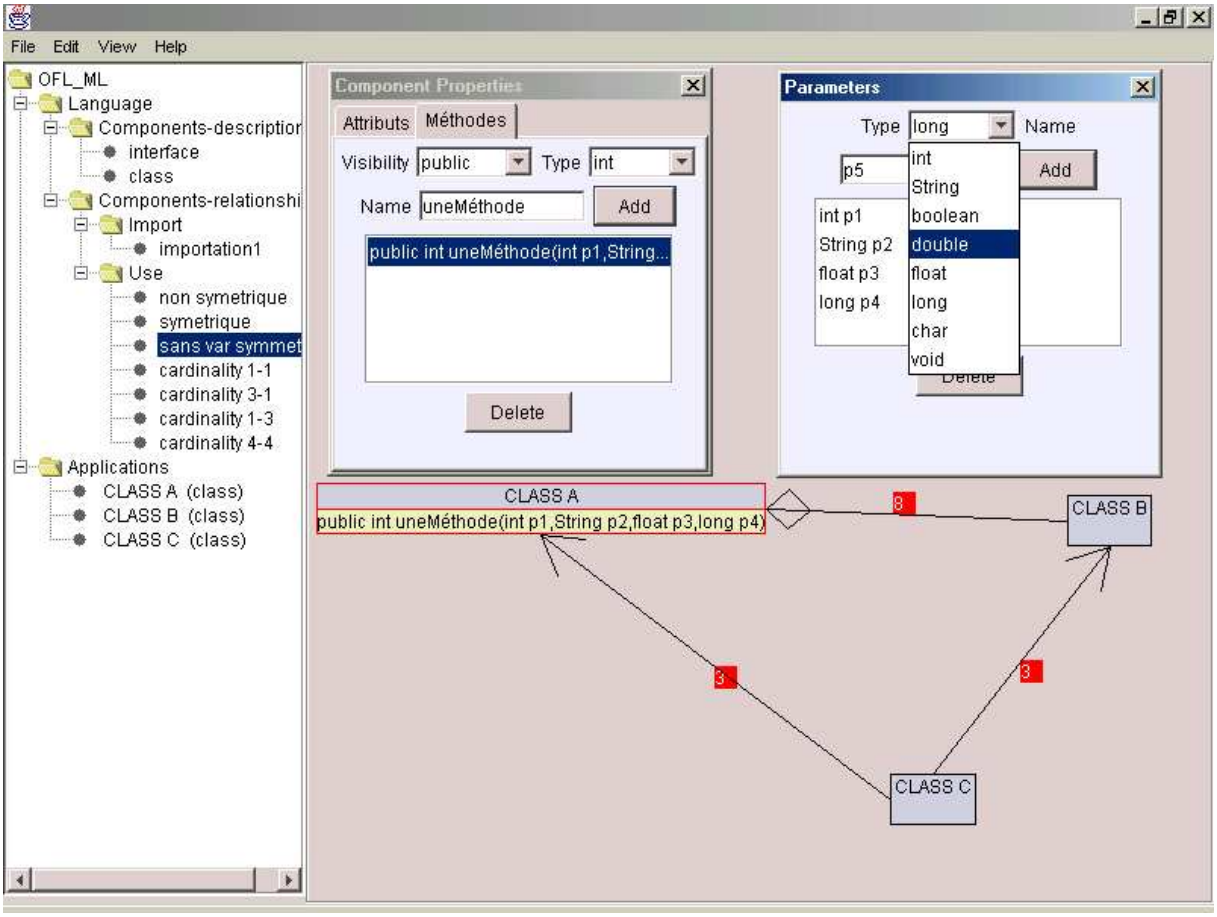

Figure 5 : Visualisation de la liste des paramètres pour une méthode donnée.

Une liste de types permet de choisir le type d'un nouveau paramètre, lequel est ajouté grâce au bouton « Add ». Le bouton « Delete » permet de supprimer un paramètre sélectionné parmi la liste des paramètres courants à une méthode.

L'ensemble des fonctionnalités d'une description a été vu et nous pouvons passer aux fonctionnalités concernant une relation.

Pour créer une relation, l'utilisateur doit sélectionner une ou plusieurs descriptions de départ par un clic gauche sur celles-ci, puis sélectionner une ou plusieurs descriptions d'arrivée.
A un ensemble de relations entre deux descriptions est associé un label contenant le nombre de relations courantes.
Une option du menu contextuel principal permet de ne sélectionner que les relations de type importation ou que les relations de type utilisation ou les deux à la fois. Cette option a pour but d'alléger la compréhension d'une application. De la même façon seule la dernière relation créée est visible entre deux descriptions.

La sélection d'une relation parmi l'ensemble des relations entre deux descriptions permet de la rendre visible au dépend de celle qui l'était déjà (cf. Figure 6).

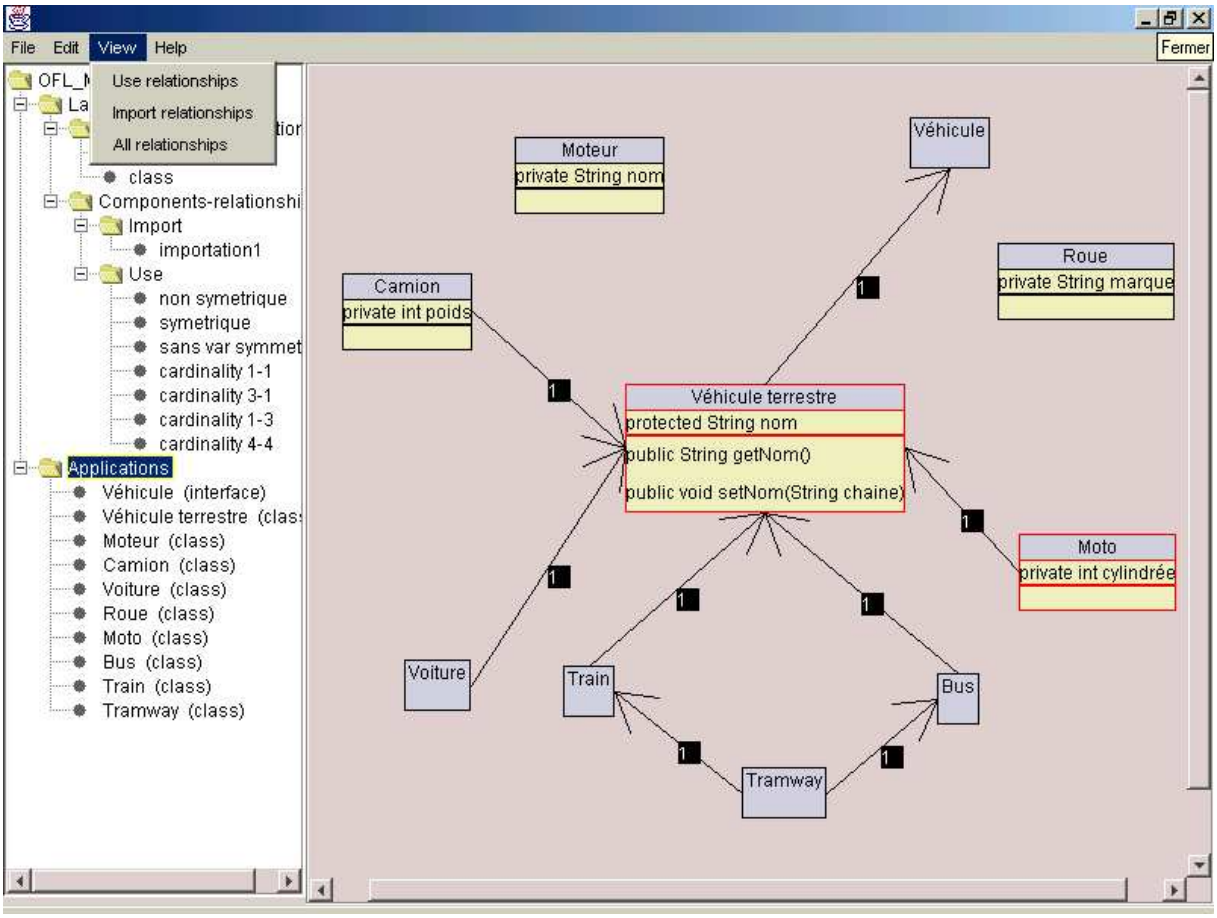

Figure 6 : Visualisation des relations de type importation uniquement.

L'utilisateur peut accéder aux propriétés d'une relation donnée simplement par un clic droit de la souris sur un des labels. Une fenêtre « Relationship Properties » (cf. Figure 7) permet alors de visualiser l'ensemble des relations en cours entre deux descriptions. L'utilisateur peut alors choisir parmi la liste des relations celle qu'il veut visualiser. Dès lors que le label contient plus d'une relation, celui-ci devient rouge afin de mettre en valeur le fait qu'il y a plusieurs relations entre deux descriptions.

Le menu proposé suite au clic droit de la souris sur une description permet d'accéder à six options (cf. Figure 8) :
La première option permet de l'effacer.
La deuxième option permet d'accéder à ses propriétés vues ci-dessus.
La troisième option permet d'arrêter la sélection de l'ensemble des descriptions de départ.
La quatrième option permet d'arrêter la sélection de l'ensemble des descriptions d'arrivée.
La cinquième option permet d'ajouter celle-ci à l'ensemble des descriptions de départ.
La sixième option permet d'ajouter celle-ci à l'ensemble des descriptions d'arrivée.

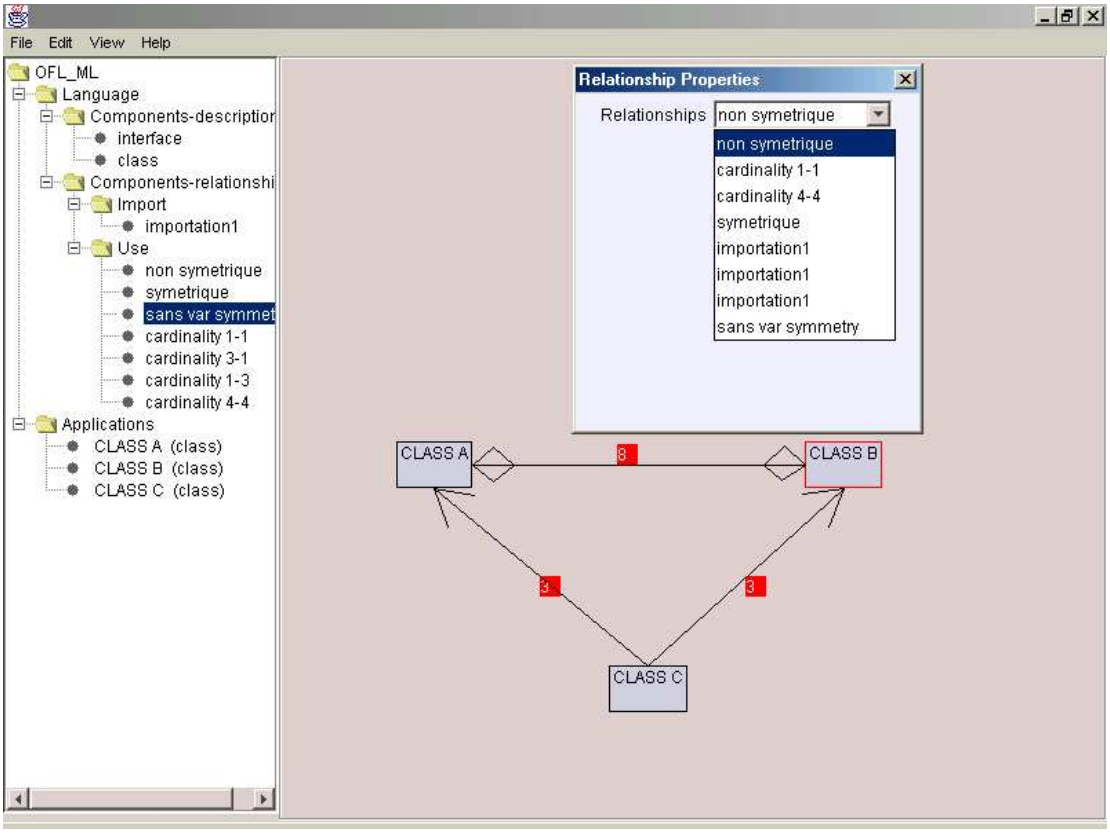

Figure 7 : Visualisation de la propriété d'une relation donnée.

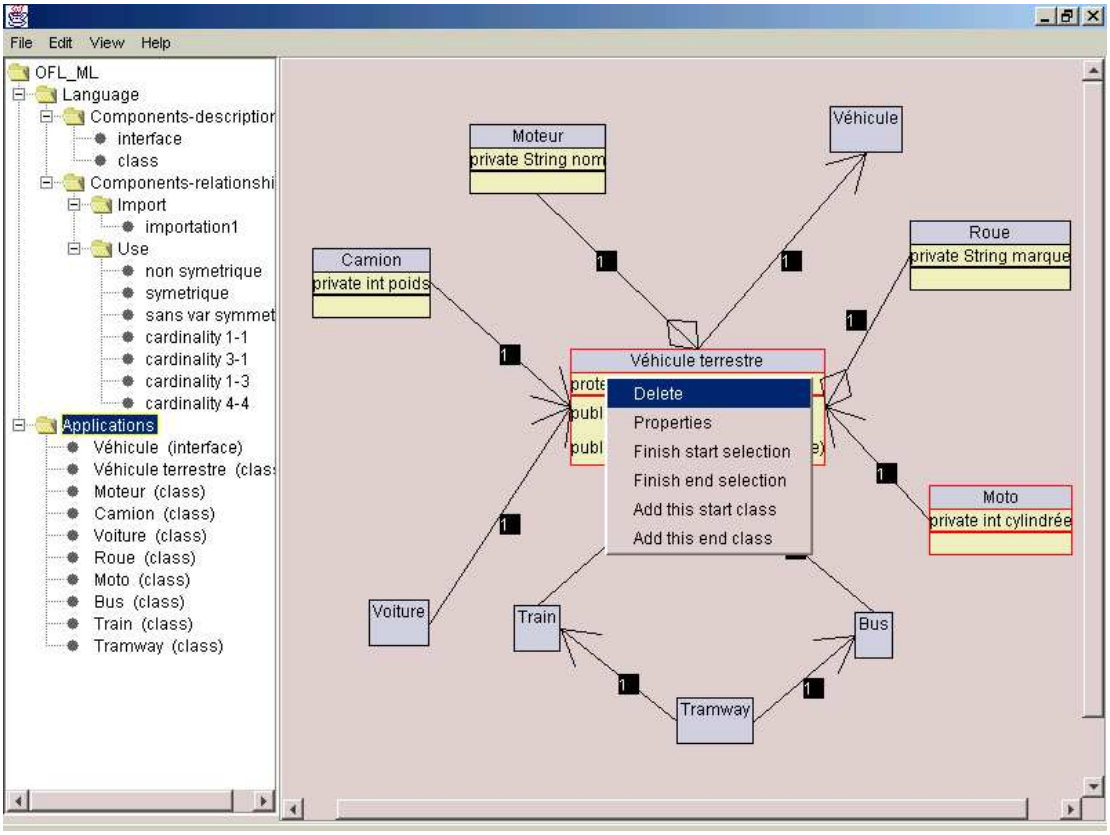

Figure 8 : Visualisation du menu suite au clic droit sur une description.

L'utilisateur est averti à tout moment d'une fonctionnalité non conforme avec un élément particulier. En effet, lors de chaque manipulation nous parcourons la liste des paramètres d'un composant afin d'avertir si nécessaire l'utilisateur d'une non conformité.

4.4 Problèmes rencontrés

La difficulté principale de ce projet a résidé dans la compréhension du concept OFL.
Il a fallu en effet découvrir et comprendre l'ensemble des paramètres d'un composant.
Plus particulièrement, la gestion des ensembles des descriptions de départ et d'arrivée a été très fastidieuse.

5. Conclusion

Aujourd'hui la quantité de langages de programmation qui s'offre au programmeur constitue un large éventail dans lequel le concepteur doit s'immerger pour trouver celui qui conviendra le mieux au projet qu'il dirige.
De plus, l'évolution de la programmation vers les langages à objets a provoqué une révolution dans le domaine du développement.
Ainsi de nombreux langages sont nés comme le C++, Eiffel, plus récemment Java et tout dernièrement des langages adaptés à des plates-formes de développement comme c# de la plate-forme .net (Microsoft).
Comment s'y retrouver parmi ce nombre toujours croissant de langages ?
Comment choisir celui qui permettra de développer une application donnée en proposant le comportement le mieux adapté au type de cette application ?
Chaque langage apporte son lot de nouveautés, ses propres spécificités avec ses qualités et ses faiblesses.
En partant de ce constat, il est toujours difficile de choisir celui qui offrira le meilleur compromis entre fiabilité, puissance et sémantique adaptée.
Les décisions à prendre quant au langage à utiliser peuvent alors devenir des dilemmes car un choix non judicieux peut s'avérer catastrophique pour la suite du projet.
Il est quasiment impossible de changer de langage en cours de développement surtout si ce dernier est déjà bien avancé. Malgré tout, si elle n'est pas impossible, tenter l'expérience peut conduire à une perte de temps considérable voire à l'échec pur et simple du projet.
L'idéal serait d'avoir un langage proposant les seules spécificités et le seul comportement nécessaires au développement de l'application désirée.
Ce langage idéal ne saurait répondre à un autre type d'application car il serait adapté de la manière la plus précise à l'utilisation qu'on veut en faire.
Pour exister, ce type de langage devrait être flexible c'est à dire offrir la possibilité de modifier sa sémantique opérationnelle, rajouter ou supprimer certains de ses comportements.
Aucun langage n'offre aujourd'hui une solution simple pour obtenir ce genre de comportement.

En proposant la possibilité de modifier un langage existant ou encore de créer son propre langage adapté, le modèle OFL permet enfin de résoudre les problèmes de ce genre.
Avec son interface graphique OFL_ML, qui s'adapte au langage qui vient d'être crée, le modèle OFL ouvre la voie vers un nouveau type de programmation, enfin libéré des contraintes souvent trop problématiques imposées par les langages de programmation actuels.

6. Annexe

Annexe 1 : Définition d'un Component-Language

```
// ********************** Définition de ParameterDefinition **************//
Vector parameterDefinitions = new Vector();

// ************************ Name ************************************//
Vector validValues = new Vector();
validValues.addElement("class");
validValues.addElement("interface");
parameterDefinitions.addElement(new ParameterDefinition("Name", validValues,
"class", new Vector()));

// ************************ Genericity ******************************//
validValues = new Vector();
validValues.addElement("true");
validValues.addElement("false");
parameterDefinitions.addElement(new ParameterDefinition("Genericity", validValues,
"false", new Vector()));

// ************************ Encapsulation ****************************//
validValues = new Vector();
Vector value = new Vector();
value.addElement("true");
value.addElement("false");
validValues.addElement(value);
validValues.addElement(value);
parameterDefinitions.addElement(new ParameterDefinition("Encapsulation",
validValues, value, new Vector()));

// ************************ Attribute ********************************//
validValues = new Vector();
validValues.addElement("allowed");
validValues.addElement("forbidden");
parameterDefinitions.addElement(new ParameterDefinition("Attribute", validValues,
"forbidden", new Vector()));

// ************************ Method ***********************************//
validValues = new Vector();
validValues.addElement("allowed");
validValues.addElement("forbidden");
parameterDefinitions.addElement(new ParameterDefinition("Method", validValues,
"forbidden", new Vector()));

// ************************ Overloading ******************************//
validValues = new Vector();
value = new Vector();
```

```
value.addElement("allowed");
value.addElement("forbidden");
validValues.add(value);
validValues.add(value);
validValues.add(value);
validValues.add(value);
Vector defValue = new Vector();
defValue.addElement("forbidden");
defValue.addElement("forbidden");
defValue.addElement("forbidden");
defValue.addElement("forbidden");
parameterDefinitions.addElement(new ParameterDefinition("Overloading",
validValues, defValue, new Vector()));

// ************************** Qualifiers ****************************//
Vector q = new Vector();
q.addElement(new Qualifier("public"));
q.addElement(new Qualifier("protected"));
q.addElement(new Qualifier("private"));
q.addElement(new Qualifier("final"));
q.addElement(new Qualifier("static"));

// ************************ Création d'un Concept-Description *************//
ConceptDescription conceptDescription = new
ConceptDescription(parameterDefinitions, new Vector(), new Vector(), q);

// ************************ Ensemble de ParameterValue ******************//
Vector parameterValues = new Vector();
parameterValues.addElement(new ParameterValue("Name", "interface"));
parameterValues.addElement(new ParameterValue("Genericity", "false"));
value = new Vector();
value.addElement("true");
value.addElement("true");
parameterValues.addElement(new ParameterValue("Encapsulation", value));
value = new Vector();
value.addElement("description");
value.addElement("instance");
parameterValues.addElement(new ParameterValue("Attribute", "forbidden"));
parameterValues.addElement(new ParameterValue("Method", "forbidden"));
value = new Vector();
value.addElement("forbidden");
value.addElement("forbidden");
value.addElement("forbidden");
value.addElement("forbidden");
parameterValues.addElement(new ParameterValue("Overloading", value));

// ********************* Création d'un Component-Description *************//
ComponentDescription componentDescription
   = new ComponentDescription(conceptDescription, parameterValues);
```

```
// ********************** Ensemble de ParameterValue *****************//
Vector parameterValues2 = new Vector();
ParameterValues2.addElement(new ParameterValue("Name", "class"));
ParameterValues2.addElement(new ParameterValue("Attribute", "allowed"));

// ********************** Création d'un Component-Description **********//
ComponentDescription componentDescription2
   = new ComponentDescription(conceptDescription, parameterValues2);

//*********************************************************************//
//********************** Définition du concept relation ********************//
// *** ***************** Définition de ParameterDefinition *****************//
parameterDefinitions = new Vector();
parameterDefinitions.addElement(new ParameterDefinition("Name", new Vector(), "",
new Vector()));
parameterDefinitions.addElement(new ParameterDefinition("Kind", new Vector(), "",
new Vector()));
parameterDefinitions.addElement(new ParameterDefinition("Circularity", new
Vector(), "", new Vector()));
parameterDefinitions.addElement(new ParameterDefinition("Symmetry", new
Vector(), "", new Vector()));
parameterDefinitions.addElement(new ParameterDefinition("Cardinality", new
Vector(), "", new Vector()));

// ******************* Définition du Concept Import Relationship ***********//
ConceptImportRelationship conceptImportRelationship
  = new ConceptImportRelationship(parameterDefinitions, new Vector(), new
Vector(), q);

// ***************** Définition d'un ensemble de ParameterValue *************//
parameterValues = new Vector();
parameterValues.addElement(new ParameterValue("Name", "importation1"));
parameterValues.addElement(new ParameterValue("Kind", "import"));
parameterValues.addElement(new ParameterValue("Circularity", "forbidden"));
parameterValues.addElement(new ParameterValue("Symmetry", "false"));
value = new Vector();
value.addElement("3");
value.addElement("4");
parameterValues.addElement(new ParameterValue("Cardinality", value));

// ****************** Définition du Component-Import-Relationship **********//
ComponentImportRelationship componentImportRelationship = new
ComponentImportRelationship(conceptImportRelationship, parameterValues);

// ****************** Définition du Concept Use Relationship **************//
ConceptUseRelationship conceptUseRelationship
= new ConceptUseRelationship(parameterDefinitions, new Vector(), new Vector(), q);
```

```
// ****************** Définition d'un ensemble de ParameterValue ***********//
parameterValues = new Vector();
parameterValues.addElement(new ParameterValue("Name", "non symetrique"));
parameterValues.addElement(new ParameterValue("Kind", "use"));
parameterValues.addElement(new ParameterValue("Circularity", "forbidden"));
parameterValues.addElement(new ParameterValue("Symmetry", "false"));
value = new Vector();
value.addElement("1");
value.addElement("1");
parameterValues.addElement(new ParameterValue("Cardinality", value));

// ****************** Définition du Component-Use-Relationship ***********//
ComponentUseRelationship componentUseRelationship
= new ComponentUseRelationship(conceptUseRelationship, parameterValues);

//****************** Création du Component-Language *****************//
Vector listeComposantDescription = new Vector();
listeComposantDescription.addElement(componentDescription);
listeComposantDescription.addElement(componentDescription2);
Vector listeImportRelationship = new Vector();
listeImportRelationship.addElement(componentImportRelationship);
Vector listeUseRelationship = new Vector();
listeUseRelationship.addElement(componentUseRelationship);

ConceptLanguage cl = new ConceptLanguage(new Vector(), new Vector(), new
Vector());

ComponentLanguage componentLanguage
= new ComponentLanguage(cl, new Vector(), listeComposantDescription,
                        listeImportRelationship, listeUseRelationship);
```

Annexe 2 : contenu de la classe ComponentLanguage

```
package ofl;

import java.util.Vector;

/* ComponentLanguage décrit un composant-langage (exemples : "java", "c++",
   ...). Il référence le concept-langage pour retrouver les données communes à
   tous les composants-langages. */

public class ComponentLanguage extends Component {
        private ConceptLanguage conceptLanguage;
        private Vector parameterValues; // ensemble de ParameterValue
        private Vector listeComposantDescription; // ensemble de composants description
        private Vector listeImportRelationship; // ensemble de relationship de type import
        private Vector listeUseRelationship; // ensemble de relationship de type use

        public ComponentLanguage() {}

        public ComponentLanguage(ConceptLanguage cl, Vector pv, Vector lcd, Vector lir,
Vector lur) {
        conceptLanguage = cl;
        parameterValues = pv;
        listeComposantDescription = lcd;
        listeImportRelationship = lir;
        listeUseRelationship = lur;
        }

        public Vector getComponentDescription () {
        return listeComposantDescription;
        }

        public Vector getComponentImportRelationship() {
        return listeImportRelationship;
        }

        public Vector getComponentUseRelationship() {
        return listeUseRelationship;
        }

        public Concept getConcept() {
        return getconceptLanguage();
        }
        public ConceptLanguage getconceptLanguage() {
        return conceptLanguage;
        }
        public Vector getParameterValues() {
        return parameterValues;
        }
}
```