# Rapport de TER

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

Maîtrise d'Informatique 2001-2002
Université de Nice Sophia Antipolis

**Participants au projet :**
Christophe GARABEDIAN – garabedc@echo.unice.fr
Gabriel SPINEK – spinekg@echo.unice.fr
Thierry TEBOUL – teboult@echo.unice.fr

26 juin 2002


**Encadreurs :**
Pierre CRESCENZO
Philippe LAHIRE

## Remerciements

Nous remercions particulièrement nos encadreurs Pierre CRESCENZO et Philippe LAHIRE, enseignants-chercheurs à l'I3S, qui nous ont souvent consacré leur temps précieux, et qui se sont mis à notre disposition pour suivre l'avancée de notre travail et répondre à nos nombreuses questions.

# Table des matières

# Table des figures

# 1 Introduction

## 1.1 Le laboratoire I3S

Le laboratoire I3S (Informatique, Signaux et Systèmes de Sophia-Antipolis) est une Unité Mixte de Recherche (UMR6070) associant l'Université de Nice Sophia-Antipolis au CNRS.

Ses activités couvrent de nombreux domaines notamment celui de l'informatique. Le labotatoire I3S regroupe environ 150 personnes, dont l'équipe du **Projet OCL**[1]. Notre TER (travail d'étude et de recherche) portant sur le **Projet OFL**[2], il est donc important de préciser que les concepteurs, ainsi que toutes les personnes qui travaillent sur le projet OFL, font partie du projet OCL ; c'est notamment le cas de nos encadreurs Pierre Crescenzo (auteur le la thèse sur le modèle OFL) et Phillipe Lahire.

Les concepteurs du projet OFL sont :
- Adeline CAPOUILLEZ
- Robert CHIGNOLI
- Pierre CRESCENZO
- Philippe LAHIRE

## 1.2 Contexte

L'un des buts de tout programmeur informatique est de fournir du code de qualité, c'est-à-dire rapide, robuste, performant, fiable et lisible. Ces qualités ne sont pas toujours compatibles, c'est pourquoi le programmeur doit souvent faire des compromis. Le Projet OCL, en s'appuyant sur le Génie Logiciel, vise à offrir au programmeur des outils et des techniques nécessaires pour produire du code de bonne qualité. Les assertions sont par exemple une des préoccupations du projet OCL, ainsi que les *relations entre classes*. Le modèle OFL propose, en se basant sur les langages à objets (Java, C++ ou Eiffel par exemple), un modèle méta-objets permettant de décrire la sémantique et le comportement de tels langages. Le métaprogrammeur peut alors, à l'aide de ce modèle, créer un nouveau langage adapté à ses besoins, ou modifier de même un langage existant.

## 1.3 But du projet

Notre travail a consisté à développer, en Java, l'un des quatre outils présents dans le modèle OFL : l'outil **OFL-ML**. Cette application permet de créer la représentation graphique d'une application, aux normes UML (uniquement orienté vers les diagrammes de Classe), et selon les composants

[1] Objets et Composants Logiciels
[2] Open Flexible Language

existants dans le langage à objets choisi. L'application OFL-ML pourra faire apparaître les types de relations possibles entre les classes, les caractéristiques des classes, des relations, des méthodes et attributs. Ainsi, plusieurs contrôles pourront vérifier les droits et les devoirs de l'utilisateur, afin d'autoriser ou d'interdire certaines de ses actions.

## 1.4 Contraintes

Nous devions dans un premier temps étudier le modèle OFL avec notamment comme support la thèse de Pierre Crescenzo. Suite à cette étude, nous devions concevoir une application où il était imposé d'utiliser le modèle de conception MVC (Modèle-Vue-Contrôleur), le polymorphisme (au possible) et le langage XML afin de fournir du code propre et réutilisable.

Par ailleurs, nous devions utiliser une bibliothèque extérieure, nommée OFL-J (car écrite en Java), qui représente une réification des langages à objets. Nous la détaillerons dans la suite de ce rapport. Cette bibliothèque nous a été fourni régulièrement, et à plusieurs reprises, car il évoluait parallèlement ; cette bibliothèque regroupe donc tous les composants de plusieurs langages à objets. Nous devions ainsi dévélopper de nouveaux composants pour le langage Java, afin de les utiliser dans OFL-ML. Il a donc fallu s'adapter à chacune de ses évolutions.

En ce qui concerne la programmation en elle-même, il était imposé de n'avoir que des attributs privés sauf exception justifiée, ce qui a nécessité un grand nombre d'accesseurs et de modificateurs.
Nous devions aussi utiliser des identificateurs pertinents et en anglais, tout comme la documentation et les commentaires. La documentation devait contenir des pré-conditions et post-conditions pour chaque méthode, exprimées en langage naturel.
Il nous était aussi demandé de rendre l'interface simple, intuitive et conviviale, afin que l'utilisateur puisse rapidement créer le diagramme de son application, et d'organiser soigneusement nos classes afin de faciliter leur relecture et leur réutilisation. De plus, il fallait implanter le chargement et la sauvegarde des diagrammes créés, suivant le format XML.

# 2 Etude du contexte

## 2.1 Le modèle OFL

Afin de mieux comprendre ce qu'est le modèle OFL, prenons un exemple simple. L'héritage est souvent incontournable en programmation orientée objets, mais il est parfois peu explicite et peu adapté. Dans ce cas, l'idée du modèle OFL est de paramétrer l'héritage pour traiter ces cas particuliers, en modifiant le moins possible les comportements par défaut du langage adapté. Ceci constitue l'**hypergénéricité**, c'est à dire le paramétrage des relations inter-classes afin d'obtenir le comportement souhaité par le programmeur (exemple : fournir une sorte d'héritage utilisé pour de la réutilisation de code, mais en évitant un polymorphisme qui n'aurait pas de sens dans ce contexte).

Le modèle OFL est donc un modèle méta-Objets qui sert à décrire la sémantique opérationnelle et le comportement des langages à objets tels que Java, C++ ou Eiffel. Le principal objectif du modèle OFL est d'offrir au méta-programmeur un moyen simple de créer un nouveau langage ou de modifier le comportement d'un langage existant. Ainsi les principaux langages de programmation pourront être construits à l'aide d'**OFL-composants**, classifiés en trois concepts : les concepts-descriptions, les concepts-relations et les concepts-langages ; les instances de ces concepts seront nommées repectivement **composants-descriptions**, **composants-relations** et **composants-langages**.

- Un concept-relation est une abstraction d'une sorte de relation dans les langages orientés objets. Les instances (les composants-relations) peuvent être vues comme des méta-relations (en java par exemple, les héritages *extends* et *implements* sont des composants-relations. Une relation *extends* peut être définie comme une relation entre deux classes. Une relation *implements* peut être définie comme une relation entre une classe et une interface). Chaque composant-relation pourrait donc être une classe dont les instances seraient des relations. Le modèle OFL définit des relations inter-description comme l'héritage par exemple, et des relations entre descriptions et objets comme l'instanciation.

- Un concept-description permet de généraliser la notion de classe, d'interface (pour le langage Java) et de valeur de base pour les langages à objets. Un composant-description peut être vu comme une méta-classe. Il peut être imcompatible avec certains composants-relations (exemple en java : le composant-description *interface* est compatible avec le composant relation *implements*).

– Un concept-langage est une réification de la notion de langage. Les concepts-langages servent donc à modéliser les nouveaux langages et permettent de fédérer un ensemble de concepts-relations et un ensemble de concepts-descriptions.

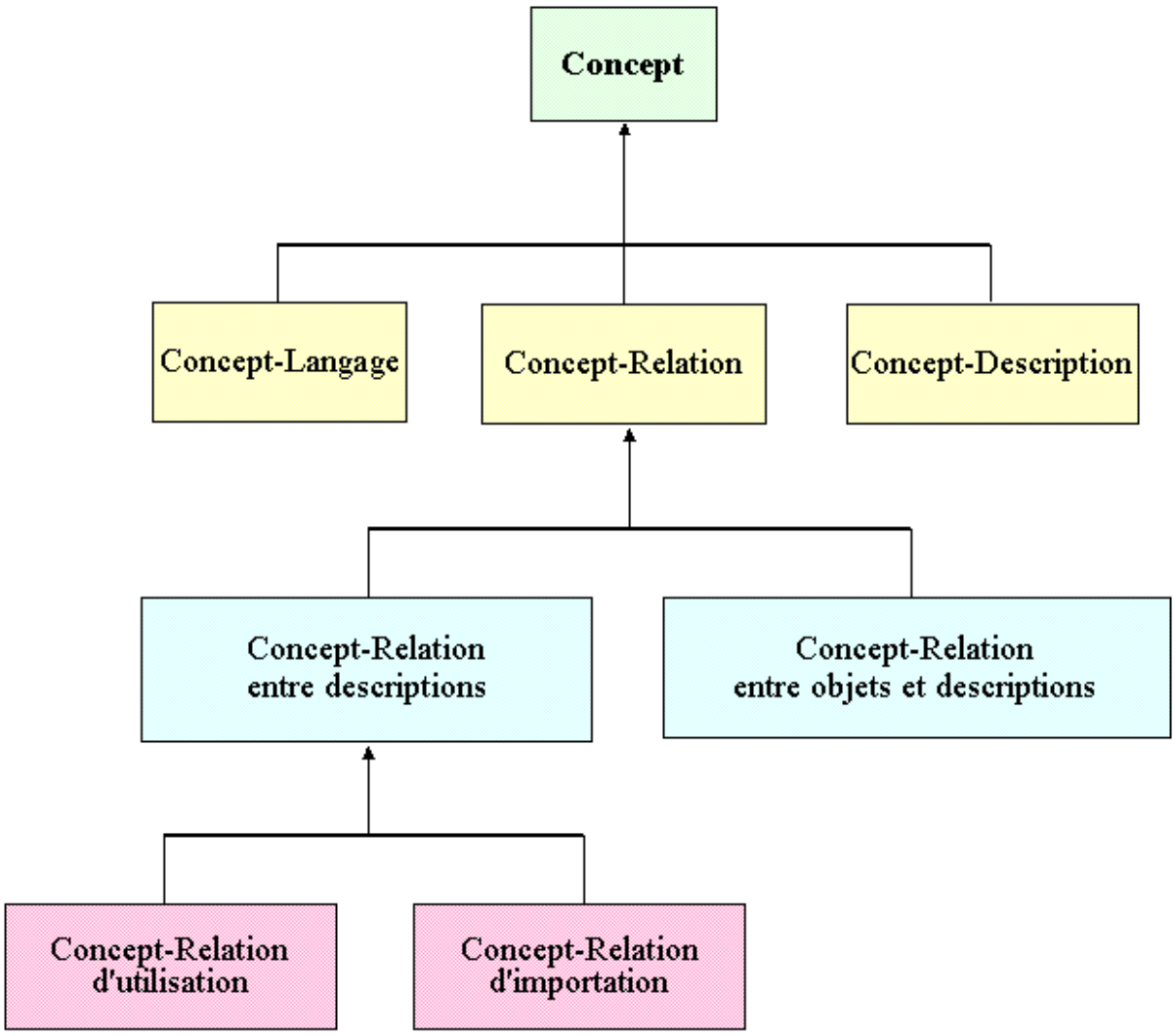


FIG. 1 – Diagramme OFL-concepts

Chaque application basée sur le modèle OFL sera réifiée. Chaque élement de cette réification sera relié à un aspect méta. Par exemple, la réification d'une classe (nommée *Description*) sera reliée à une méta-classe (nommée *composant-description*). *Description* contiendra l'ensemble de ses paramètres (une liste d'attributs, de méthodes, de classes héritées, etc...). Les objects "méta" contiendront aussi des paramètres et des actions qui permettront une sémantique plus précise. On pourra ainsi faire du contrôle, de la génération de code ou des métriques. Dans notre cas, cette sémantique plus précise permettra principalement de faire du contrôle sur les droits du programmeur. Par exemple, s'il manipule une relation reliée à une méta-relation (nommée *composant-relation*) qui décrit un héritage non multiple, les actions sémantiques interdiront au programmeur de faire un héritage multiple. (cf schéma ci-dessous)

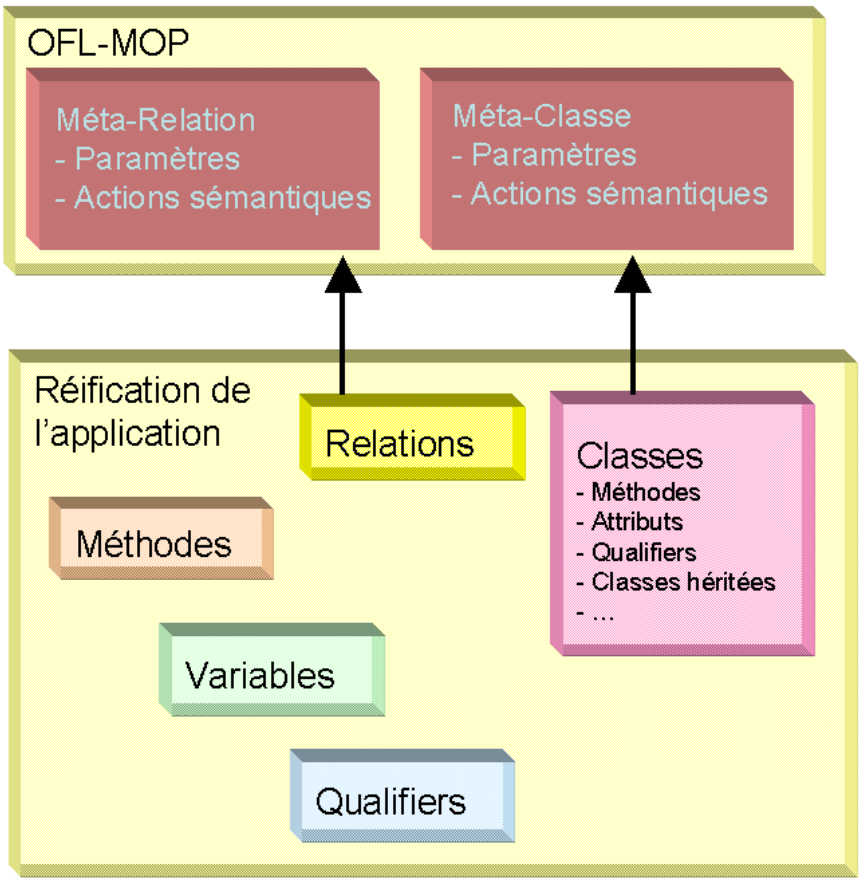


FIG. 2 – Réification d'une application

## 2.2 Les objectifs d'OFL

La réalisation du modèle a pour but d'améliorer la qualité du code et donc celle du logiciel.

Deux idées ont oeuvré à la conception du modèle OFL :
- si le programmeur écrit moins de lignes de codes (pouvant être répétitives), il y aura moins d'erreurs.
- s'il a plus de sémantique pour décrire une description ou une relation, les spécifications seront alors plus précises lors de la phase de conception/programmation.

Plusieurs objectifs sont donc mis en place au moment de la réalisation d'OFL pour offrir de nombreux avantages.

Les deux premiers sont d'augmenter la **lisibilité** des codes sources, et permettre une meilleure **documentation**. Plus le code est précis, plus il est lisible. Par exemple en Java, lorsque l'on spécifie pour une classe une relation d'importation, on peut lire "extends" ou "implements" cela nous renseigne d'avantage qu'un simple héritage "inherits" en Eiffel par exemple. Ainsi, en ayant des descriptions et des relations plus explicites que l'héritage et l'agrégation, la documentation du code de l'application générée est beaucoup plus précise et pertinente. Et comme une liste de paramètres a été établie, permettant de définir la sémantique et les informations essentielles des descriptions et des relations, le programmeur est mieux renseigné sur les aspects du langage.

Un avantage important apporté par le modèle OFL est celui de la **flexibilité** (on le retrouve dans le nom *Open Flexible Languages*). Comme nous l'avons vu dans le chapitre précédent, le modèle OFL offre la possibilité d'adapter son langage de programmation à ses besoins, soit en modifiant sa sémantique opérationnelle, soit en ajoutant ou supprimant des descriptions ou relations utiles.

OFL apporte aussi une augmentation de la **maintenabilité** des logiciels. Grâce à une meilleure lisibilité et documentation, c'est à dire à une meilleure spécification de la sémantique associée aux descriptions et relations, le programmeur pourrait par exemple poursuivre plus facilement un programme déjà écrit. De même OFL apporte une meilleure **évolutivité** des applications. Par ailleurs, comme nous l'avons dit, OFL offre la possibilité d'adapter son langage de programmation à ses besoins, ou d'étendre un langage pour l'améliorer, et ainsi le faire évoluer.

OFL permet aussi de réaliser des **contrôles**, qui seront plus ou moins

étendus selon la précision de la sémantique des descriptions et des relations (exemple : la réutilisation de code interdit le polymorphisme). OFL offre ainsi la possibilité d'obtenir des composants logiciels robustes et fiables. De même, grâce aux meilleures indications effectuées par le programmeur cela entraîne de meilleures **performances** de l'interprète ou du compilateur, ainsi une optimisation plus efficace du code généré.

Un dernier objectif important est l'**adéquation**. Le but est de réduire le fossé entre les méthodes de conception telles UML et les langages de programmation, ce qui a pour avantage de diminuer les risques de conception d'un logiciel qui ne répond pas aux spécifications.

L'idée générale est de ne pas reposer sur un langage de programmation donné (plateforme indépendante d'un langage particulier), et de simplifier le travail du méta programmeur. En augmentant la qualité des logiciels et en spécifiant plus précisément les usages des descriptions et relations, les programmeurs peuvent plus facilement exprimer leurs intentions, et adapter un langage de programmation à des besoins spécifiques, tout en gardant un cadre contrôlé.

## 2.3 La bibliothèque OFL-J

### 2.3.1 Modèle OFL

Le paquetage OFL-J représente la réification du modèle OFL. Comme nous l'avons vu précédemment, OFL est indépendant d'un langage particulier et englobe les composants des langages à objets existants. Ces composants permettent de paramétrer les caractéristiques et le comportement de type "description" et "relation". Nous rappellons que la notion de description décrit les objets d'une application, et la notion de relation décrit les liens entre les éléments de types description.

Le modèle OFL nécessite alors le développement de deux travaux distincts que sont la création d'un langage à partir d'un ensemble de composants (les paramètres hyper génériques) et la création d'un programme à partir du précédent langage. Ces deux travaux décrivent ainsi trois niveaux. (Nous avons contribué aux développements de ces trois niveaux).

OFL-J regroupe deux niveaux :

- Le **niveau OFL** constitue un méta modèle pour le langage de programme (niveau langage), qui décrit la partie paramétrable des descriptions, relations, et langages, ce sont des "OFL-concept". Il constitue aussi un méta modèle pour les programmes (niveau application), qui décrit la partie non paramétrable des trois concepts vus précédemment, ce sont des "OFL-atomes".

- Le **niveau LANGAGE** décrit un langage à objets 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".

L'outil OFL-ML décrit :

- Le **niveau APPLICATION** qui possède tous les objets créés et les relations entre ces objets. Les éléments créés sont de type "description" et "relation", et doivent respecter les caractéristiques décrites au niveau Langage.

La figure 3 nous montre l'architecture du modèle OFL, représentée par la bibliothèque OFL-J.

### 2.3.2 Utilisation d'OFL-J

Grâce à la classe OFL-EXTERNAL contenue dans le paquetage OFL-J, nous avons accès aux composants des langages, basés sur le modèle OFL.

Ces composants sont rangés dans un nouveau sous-paquetage d'OFL-J au niveau des "OFL-composants". Ces nouvelles classes définissent un langage de programmation personnalisé. Elles sont décrites à partir des bibliothèques existantes dans OLF-J, et différentes valeurs ont été attribuées aux différents paramètres possibles pour décrire la sémantique du langage.

Pour créer des composants, nous nous sommes basées sur le langage Java. Ces composants offrent alors des caractéristiques propres et seront utilisés lors de la conception d'une application avec OFL-ML.

Les composants décrits seront chargés et analysés par l'application OFL-ML, qui mettra ainsi à disposition les descriptions et les relations disponibles suivant le langage choisi. Ainsi, OFL-ML pourra récupérer les choix de l'utilisateur parmi tous les composants disponibles, pour offrir, en utilisant OFL-J, la possibilité d'instancier des descriptions et des relations.

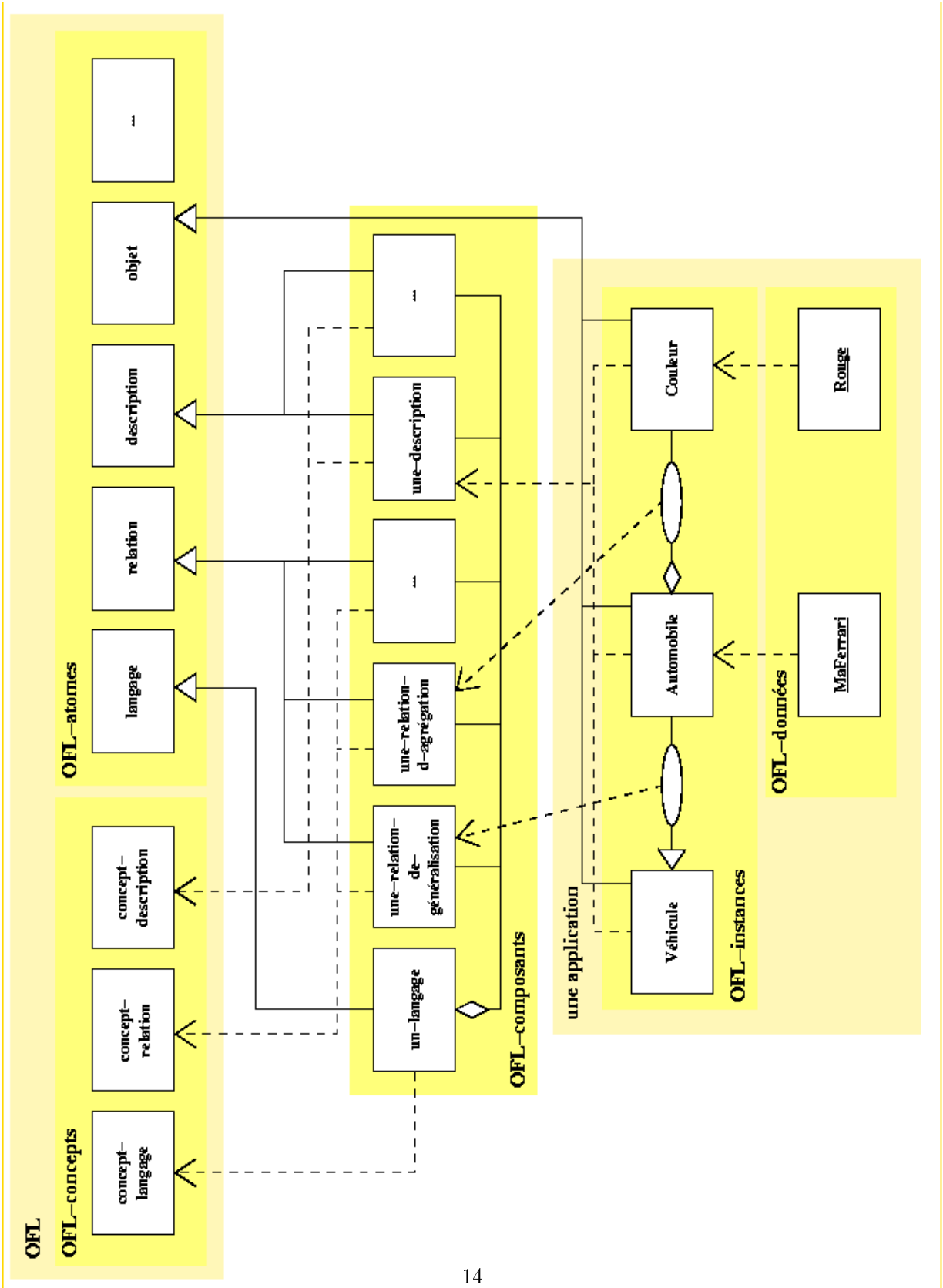




FIG. 3 – Architecture du modèle OFL

## 2.4 Description du Modèle MVC

### 2.4.1 Définition

MVC est constitué de trois types d'objets ; le modèle, la vue, et le contrôleur. Le principe est simple, mais son application n'est pas triviale. Il s'agit en fait de séparer les données de l'interface graphique en utilisant des contrôleurs qui joueront un rôle d'intermédiaire.
Généralement dans une interface, il y a plusieurs modèles, plusieurs contrôleurs et plusieurs vues qui sont souvent en interaction. Mais l'architecture MVC ne nous est pas tout a fait inconnue. Elle est largement utilisée dans les langages à objet (JAVA ou Eiffel par exemple) pour modéliser les composants graphiques qui contiennent des données.

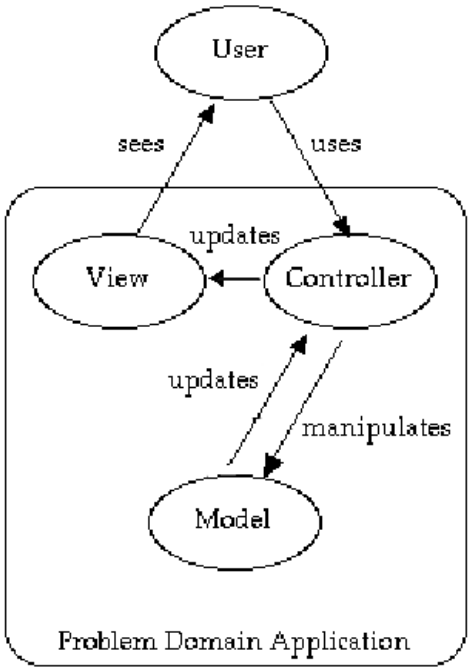


FIG. 4 – Principe du modèle MVC

### 2.4.2 Pourquoi séparer ?

La séparation d'un programme en Modèle, Vue, et Contrôleur possède un grand nombre d'avantages. Tout d'abord et principalement, il fournit un ensemble naturel de frontières d'encapsulation, qui permet de réduire les interdépendances du programme et les interactions, et donc réduit les erreurs et améliore la compréhension du programme. Ensuite, la séparation encourage et facilite la production de multiples vues d'un même modèle.

# 3 OFL-ML

## 3.1 L'existant

Il existe certains logiciels qui permettent de programmer graphiquement une application, comme par exemple RATIONAL-ROSE, et qui offrent à l'utilisateur la possibilité de créer un diagramme de classe UML, puis ensuite de générer le code correspondant. Bien qu'efficaces dans leur domaine d'application, ces logiciels sont limités par la sémantique opérationnelle d'un langage, et ne proposent à l'utilisateur que l'ensemble des spécificités propres à ce langage. De telles applications restent de plus beaucoup trop générales.

Notre objectif n'était pas d'essayer de rivaliser avec le grand nombre d'éditeurs UML existants, mais de développer une application simple, intuitive et évolutive, prenant en compte plusieurs langages avec leur propre sémantique opérationnelle.

## 3.2 La structure d'OFL-ML

Voici la structure d'OFL-ML. Un arbre hiérarchique représente les composants disponibles et la zone de dessin contriendra les représentations des descriptions et des relations.

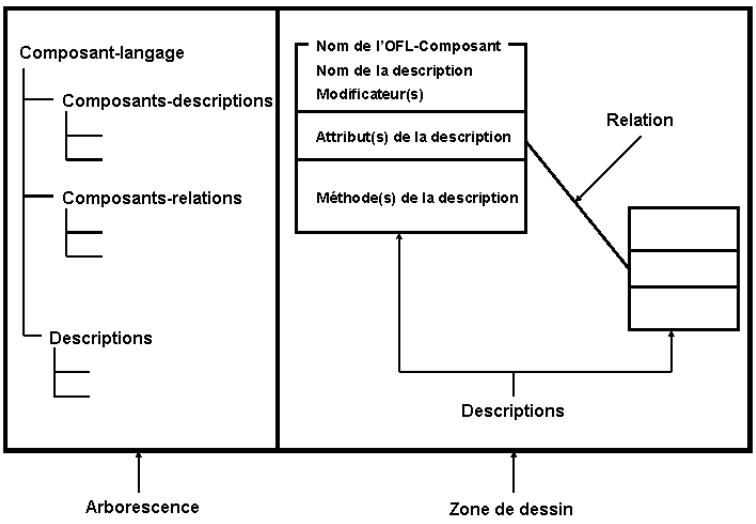


FIG. 5 – Structure de l'interface graphique d'OFL-ML

### 3.3 Le modèle MVC

Maintenant que nous avons décrit, dans une précédente partie, ce qu'est le modèle MVC, nous allons étudier ses avantages et ses besoins dans OFL-ML, dans un soucis de clarté et d'évolutivité.

Dans le cadre de notre projet, le fait d'avoir des ensembles d'objets indépendants facilite par exemple l'ajout ultérieur de vues et de contrôleurs. Nous avons notamment développé deux vues différentes pour la représentation d'une description. Il est donc possible dans l'application de changer de vue "à chaud" ce qui nous montre explicitement les avantages du modèle MVC.

L'image suivante représente le menu grâce auquel on peut changer de vue : il faut sélectionner la vue choisie, puis cliquer sur "Change view for any Description" pour l'appliquer. Les images suivantes montrent le passage d'une vue à l'autre.

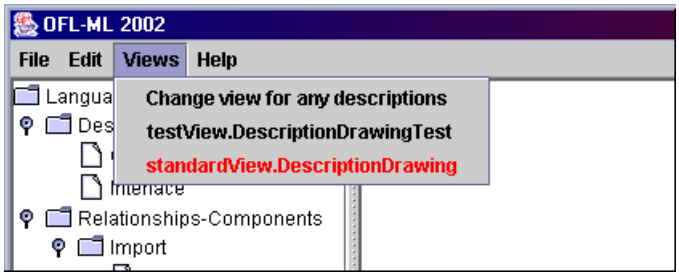


FIG. 6 – Menu permettant de changer de vue

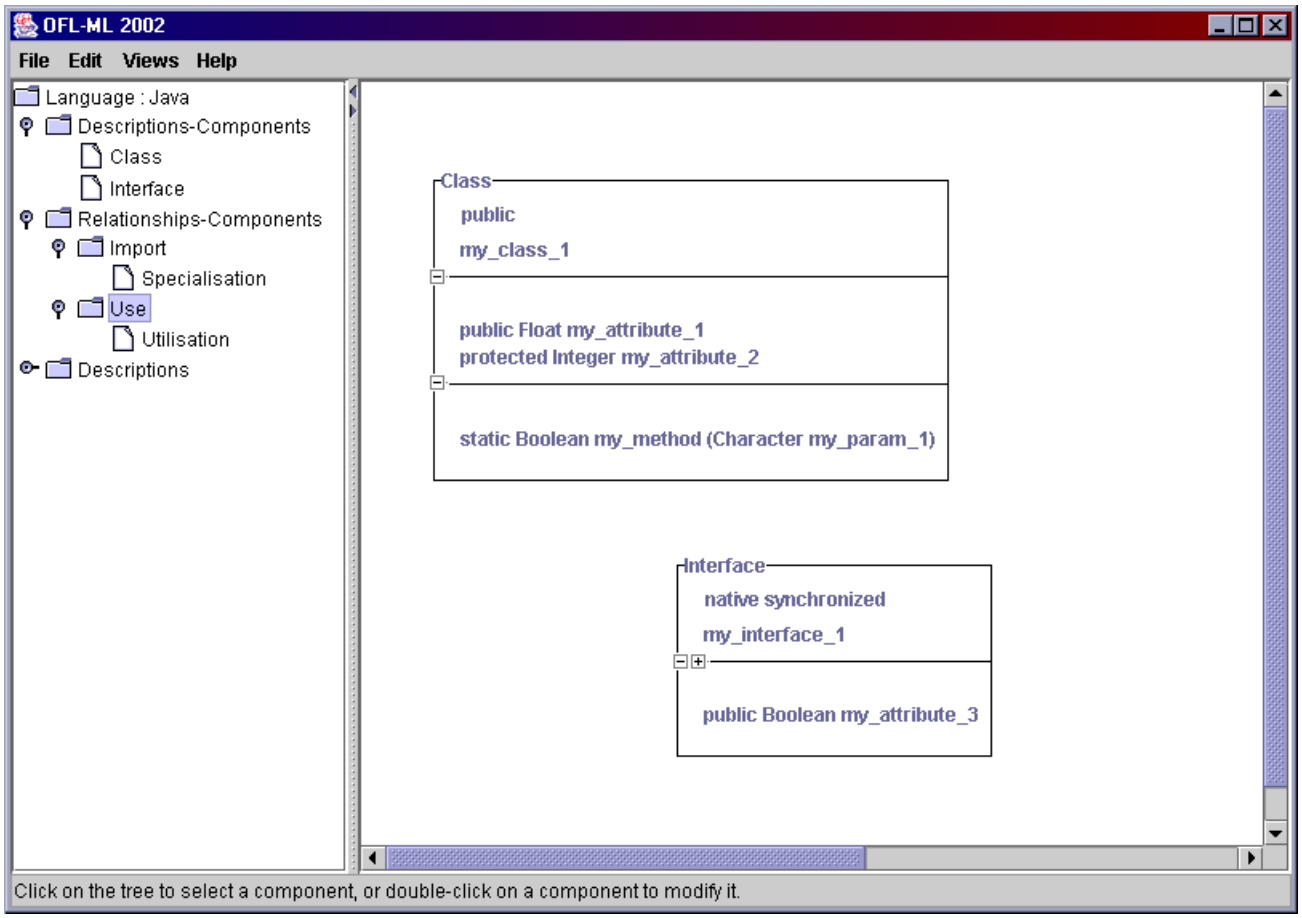


FIG. 7 – Affichage avant le changement de la vue

Voici ce qu'on obtient suite à un changement de vue :

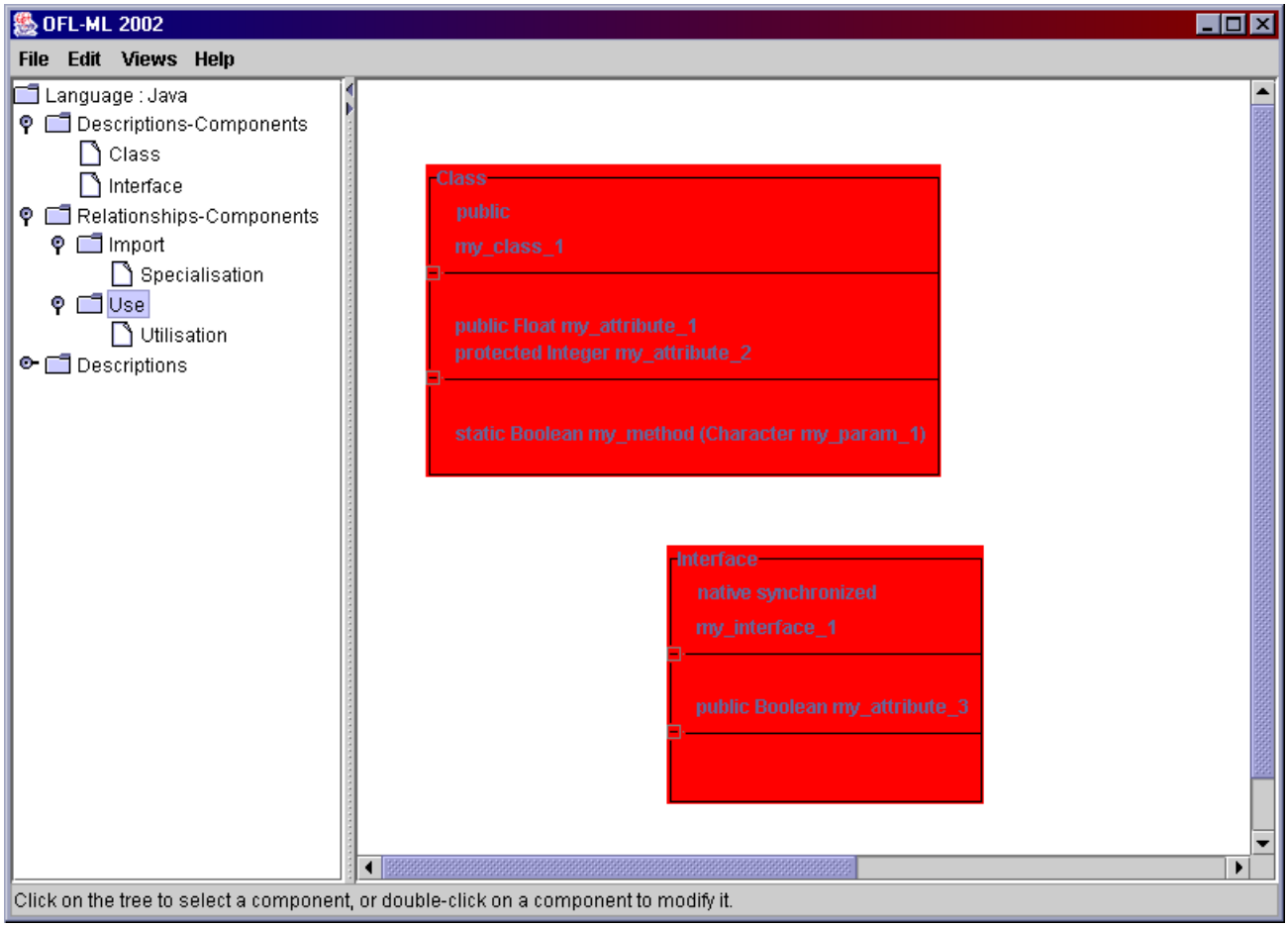


FIG. 8 – Affichage après le changement de la vue

Précisons que nous avons réalisé un chargement dynamique de vues, ce qui permet de charger des vues (pour les descriptions et les relations) sans recompiler notre application. Il suffira donc de déposer dans le répertoire `src/gui/views` un répertoire contenant toutes les classes nécessaires à la prise en compte de nouvelles vues. Celles-ci seront ajoutées dans le menu "Views" de notre application pour être directement utilisables.

### 3.3.1 MVC dans notre application

- **Le modèle** représente nos données, c'est à dire dans notre application les classes qui dérivent de *DataComponent.java*, donc *DataDescription.java* pour les données relatives aux descriptions, et *DataRelationship* pour les données relatives aux relations. Ces classes sont isolées du reste de l'application, et ne possèdent qu'une "liaison" vers le paquetage OFL-J, qui, nous le rappellons, représente une sorte de base de données des caractéristiques des différents languages utilisables.

- **La vue** décrit une représenation graphique d'un modèle. Dans notre application, les vues représentent les classes qui dérivent de *ViewComponent*. Par exemple, on associera une vue *ViewDescription.java* à une donnée de type *DataDescription.java*. Cette vue sera chargée de redessiner une description, dans une zone de dessin, lorsqu'une modification aura lieu sur le modèle correspondant.

- **Le contrôleur** joue un rôle intermédiaire entre le modèle et la vue. Par exemple, un utilisateur voulant modifier une donnée passera par un contrôleur. Une fois la donnée modifiée, cette dernière notifiera les vues pour qu'elles se réaffichent. Ce mécanisme se réalise facilement en JAVA grâce au design-pattern[3] **observer-observable**, les données étant les observables et les vues les observeurs. Une modification d'une donnée entraîne une notification aux observeurs correspondants.

  Ni la vue ni le modèle ne disposent du contrôleur. C'est uniquement l'application, en dehors du modèle MVC, qui dispose d'une liste de contrôleurs.

Chaque contrôleur est mis en relation avec une instance de *DataDescription.java* (ou plutôt d'une de ses classes dérivées), et avec une instance de *ViewComponent.java* (d'une de ses classes dérivées). Voici un diagramme mettant en évidence, dans notre application, les différentes interfaces nécéssaires au modèle MVC.

[3]Patrons de conception

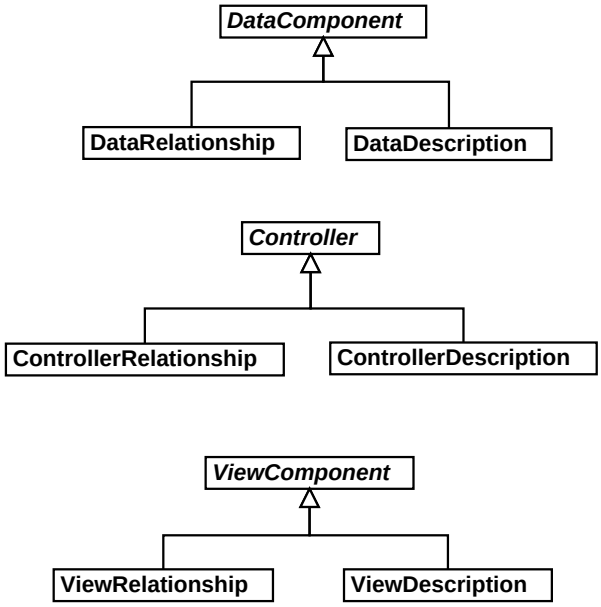


FIG. 9 – Les différentes interfaces présentes dans notre modèle MVC

**Pour conclure :**

- Une donnée ne dit que : *"Je notifie les observeurs que j'ai changé"*.

- Une vue ne dit que : *"Je me redessine suite à une notification"*.

- Un contrôleur possède une instance de *DataComponent* et de *ViewComponent*, et ne sait dire que : *"DataComponent, je change tes valeurs, ou donne moi tes valeurs."*. Il peut bien sûr fournir les deux instances précédente qu'il possède. Le contrôleur ne retourne que des types de base (ou une composition de types de base) pour que l'on n'accède pas directement aux données depuis l'endroit où l'on dispose d'un contrôleur.

### 3.3.2 Développer des modèles, des vues et des contrôleurs

Comme nous l'avons vu dans la figure 9, chaque élément principal du modèle MVC est représenté dans notre application par une interface. L'interface *Controlleur.java* pour les contrôleurs, l'interface *DataComponent.java* pour les données, et l'interface *ViewComponent* pour les vues. Pour développer une nouvelle vue, il suffira d'implementer l'interface *ViewComponent.java*, et de rajouter si nécessaire des attributs et des méthodes supplémentaires. Le programmeur devra implémenter toutes les méthodes des interfaces afin de définir et d'intégrer de nouvelles vues, à OFL-ML.

Cette technique, qui permet de developper de nouveaux éléments grâce à des interfaces, représente en fait un schéma de conception souvent utilisé dans le modèle MVC : le schéma de conception **Stratégie**. On pourrait donc dire que ce schéma de conception définit une sorte de famille d'algorithmes, et encapsule chacun d'eux pour les rendre interchangeables. Il permet alors aux algorithmes d'évoluer indépendamment des clients qui les utilisent.

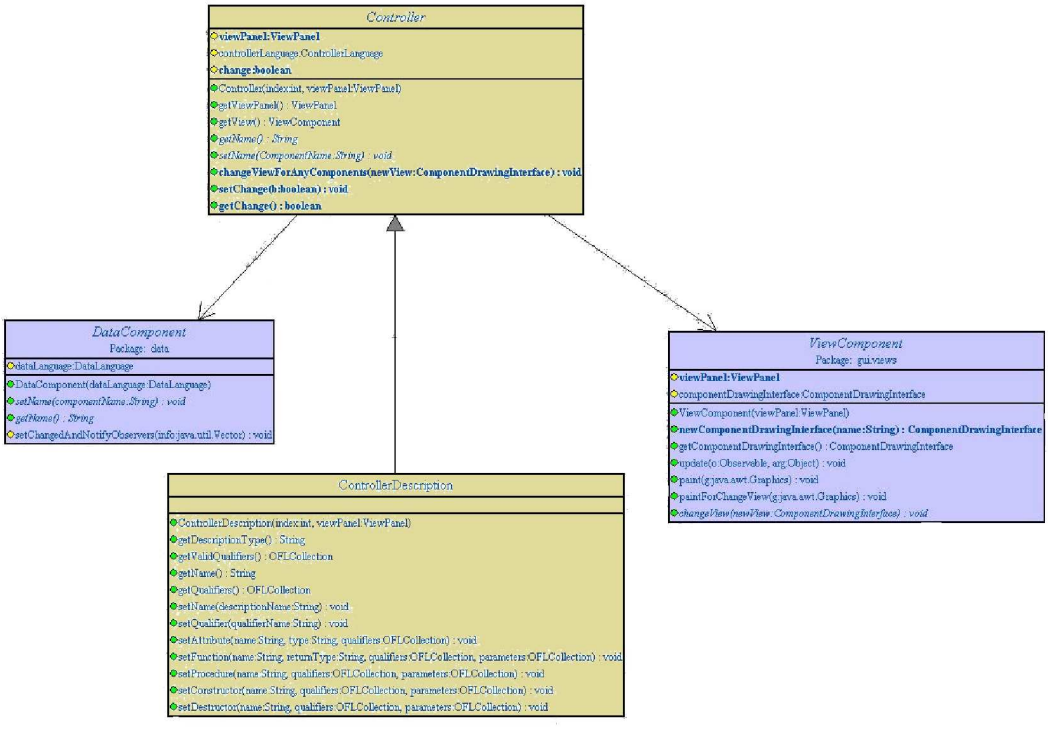


FIG. 10 – Diagramme de classes du modèle MVC

## 3.4 Description et fonctionnalités

Cette partie permet de prendre connaissance avec le logiciel OFL-ML, en découvrant ses possibilités par le biais de nombreuses captures d'écran. Ce "manuel", nous l'espérons, permettra au lecteur de se familiariser rapidement avec l'application et d'apprécier son intuitivité et son intérêt.

### 3.4.1 Lancement de l'application


FIG. 11 – Menu permettant de charger un langage


FIG. 12 – Fenêtre de chargement d'un langage

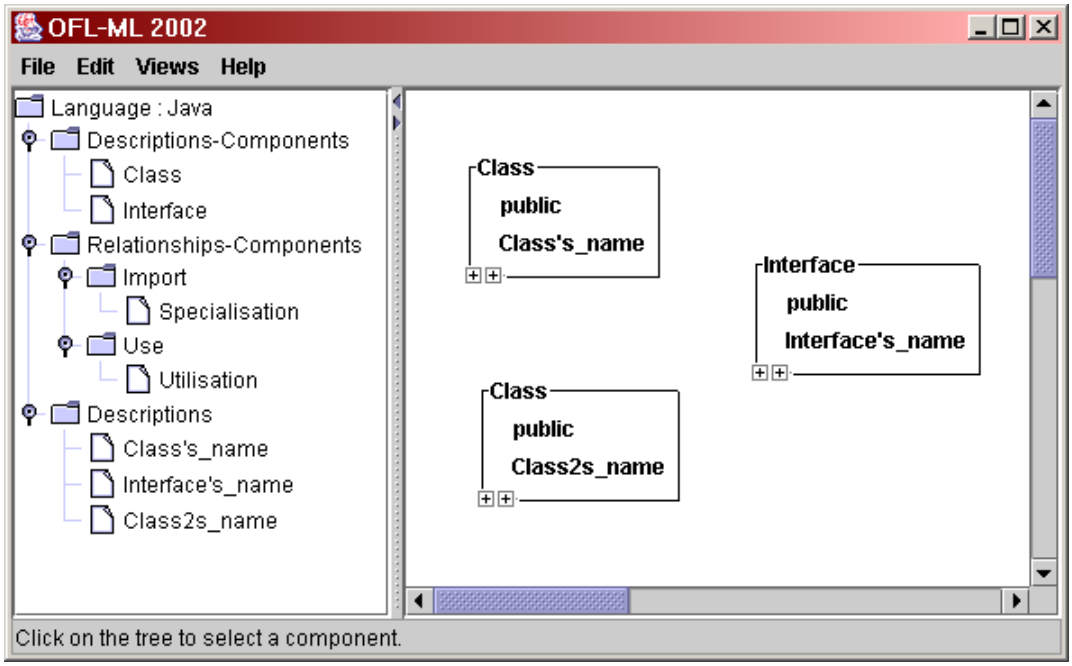


FIG. 13 – Arbre hiérarchique après chargement du langage Java

Le logiciel OFL-ML permet de créer graphiquement une application, contenant des descriptions (classes, interfaces...) et des relations (importations...). Pour cela, un langage à objets est tout d'abord chargé, grâce au menu *File* et à une fenêtre de choix de langages (voir les figures 11 et 12) ; le langage Java est d'ailleurs chargé au démarrage, c'est le langage par défaut. Une fois le langage chargé, selon les éléments présents dans la bibliothèque OFL-J, ses composants sont affichés dans l'arbre hiérarchique (figure 13) de la façon suivante :

- Le noeud "Language : ..." indique le nom du langage chargé, ici Java.
- Le noeud "Descriptions-Components" contient la liste des descriptions présentes dans le langage en cours, donc dans l'exemple ci-contre les descriptions représentent les Classes et les Interfaces.
- Le noeud "Relationship-Components" contient la liste des types de relations faisant partie du langage. Ces relations sont séparées en deux ensembles : celles de type "Importation" et celles de type "Utilisation", qui sont imposées par le modèle OFL. Nous avons créé pour chacune de ces deux catégories une "relation-exemple" appelées respectivement "Specialisation" et "Utilisation".
- Enfin, le dernier noeud, vide au départ, regroupera les descriptions créées. Donc, à chaque création ou modification d'une description, ce noeud sera mis à jour. Ce noeud est en fait un observeur des descriptions crées dans la zone de dessin.

De plus, l'arbre hiérarchique jouera aussi un rôle de sélecteur afin d'ajouter ou de supprimer une description ou une relation. Par exemple, suite au choix d'une description (depuis le dernier noeud), sa suppression (depuis *Menu Edition –> delete description*) entrainera aussi la suppression des instances du contrôleur, de la vue et du modèle associés.

Maintenant que nous avons expliqué le fonctionnement et l'utilité de l'interface, nous pouvons passer à la création, ou plus précisément, à l'instanciation des composants présents dans l'arbre.

### 3.4.2 Création (ou modification) de descriptions

On peut désormais choisir un composant pour l'*"instancier"*, après quoi sa représentation graphique s'affichera dans la partie droite de l'éditeur. On peut noter qu'une aide basique est présente dans la zone de texte au bas de l'interface principale, indiquant les actions possibles.

La création des composants se fera au moyen de la fenêtre de la figure 14 (pour les descriptions). Une fenêtre similaire apparaît lorsqu'on double-clique sur une description existante : on pourra alors modifier celle-ci, ses valeurs courantes étant chargées dans la fenêtre.
Mais il est aussi possible d'utiliser l'arbre hiérarchique pour cela en sélectionnant un composant, puis en choisissant une action dans le menu d'édition.
Les figures 15 et 16 montrent les interfaces permettant de créer les attributs et les méthodes des descriptions.

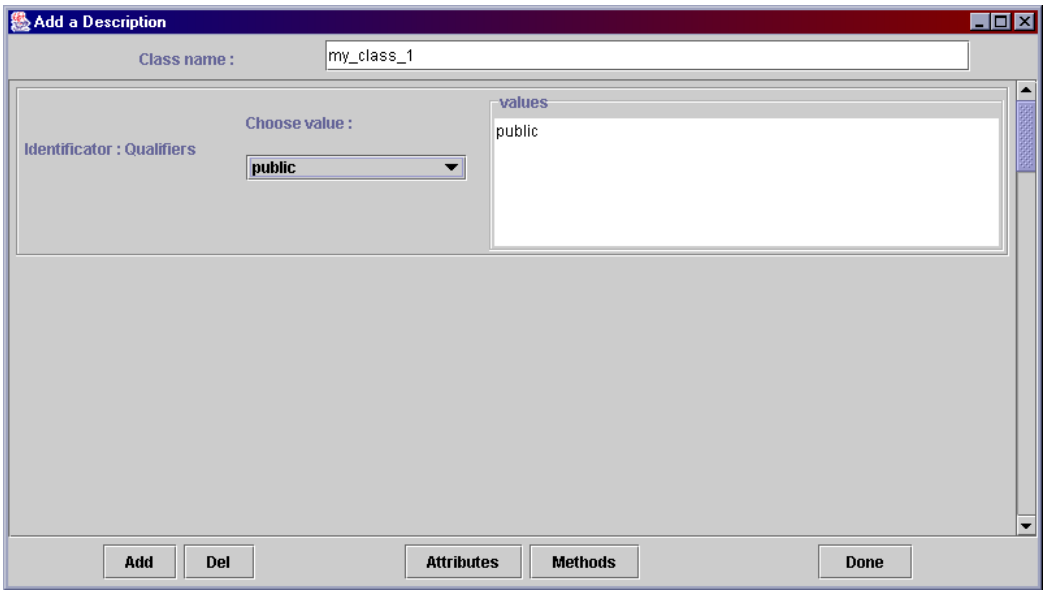


FIG. 14 – Fenêtre de création de descriptions

On pourra ici entrer le nom de la description, choisir son (ses) qualifier(s) parmi la liste proposée, et appeler les fenêtres des attributs et des méthodes.

La liste de tous les qualifiers disponibles est récupérée depuis OFL-J. Elle n'est pas construite depuis l'interface graphique, mais elle est récupérée grâce à un *controlleurDescription*.

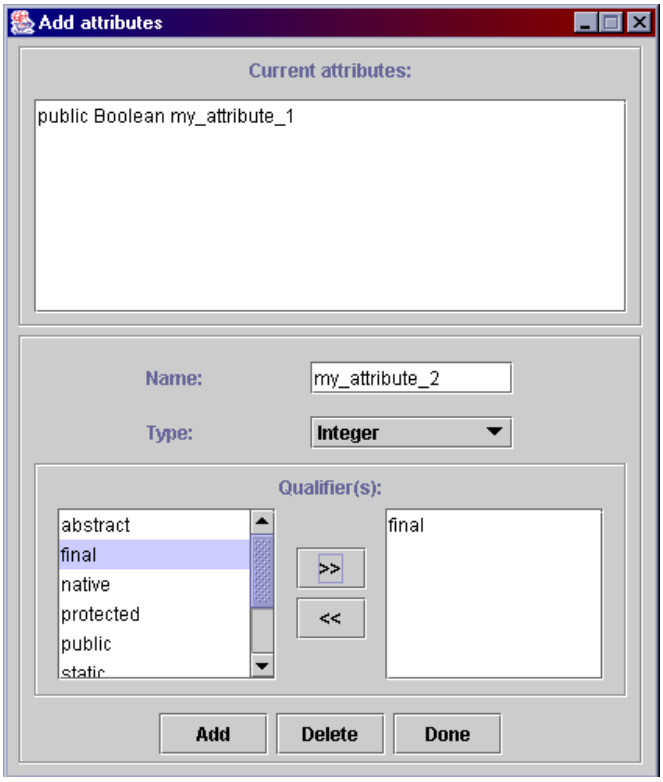


FIG. 15 – Fenêtre de modification des attributs d'une description

Il est à noter qu'un contôle permet d'empêcher l'utilisateur de définir deux attributs ayant le même nom. En effet, pour une description, nous vérifions à l'aide du contrôleur associé que l'attribut à ajouter ou à modifier n'est pas déjà présent dans la listes des attributs de la description. Pour réaliser un tel contrôle, le contrôleur appelle une méthode depuis son modèle associé qui se chargera de vérifier, dans OFL-J, la validité de l'action de l'utilisateur. Le contrôleur joue donc un rôle intermédiaire.

Il en est de même pour l'ajout ou la modification d'une méthode. Une vérification est donc réalisée par l'intermédiaire du contrôleur de la description afin d'éviter les méthodes identiques (sur leur signature). D'autres contrôles sont réalisés au niveau d'une méthode, notamment celui qui empêche deux de ses paramètres d'avoir le même nom et le même type.

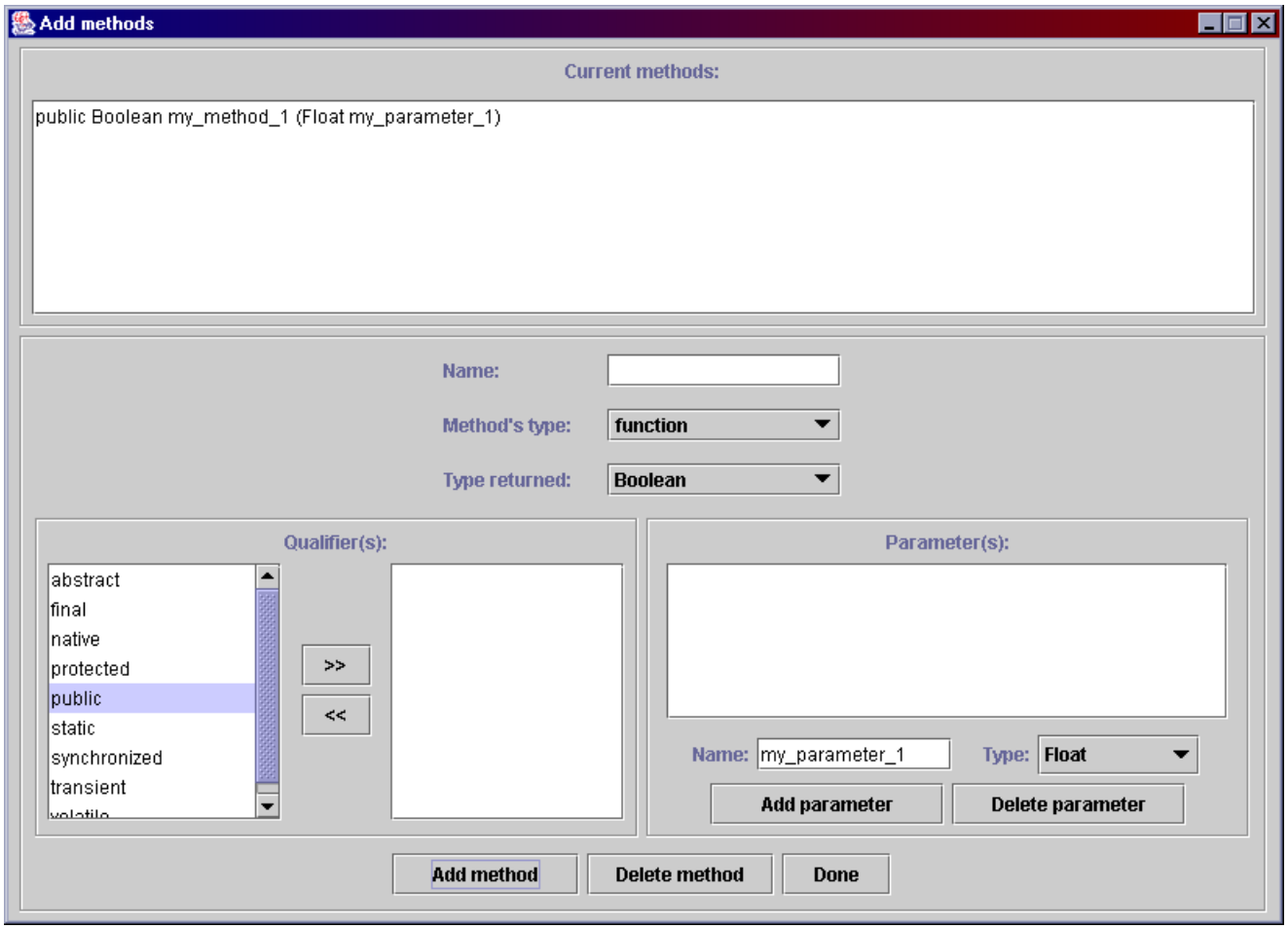


FIG. 16 – Fenêtre de modification des méthodes d'une description

### 3.4.3 Suppressions

Toutes les suppressions sont possibles depuis l'arbre hiérarchique ; la suppression d'une description et la suppression d'attibuts et de methodes d'une description. Comme nous l'avons dit précédemment, la suppression d'une description entraînera aussi la suppression des instances du contrôleur, de la vue et du modèle associés. Par contre, une suppression d'attribut ou de methode paraît certes triviale, ce qui n'est pas toujours le cas. Par exemple, pour supprimer une methode de type *procédure* (c'est à dire considérée sans valeur de retour) depuis une description, il faut retrouver dans la liste de ses méthodes celle qui correspond à la signature de la procédure à supprimer. Cette tâche nécéssite un travail de composition et de décomposition de la signature de la méthode, car la bibliothèque OFL-J décrit de manière très rigoureuse ce que représente une méthode de type *procédure*.

### 3.4.4 Représentation des descriptions

Voici la manière dont se décompose la vue d'une description :

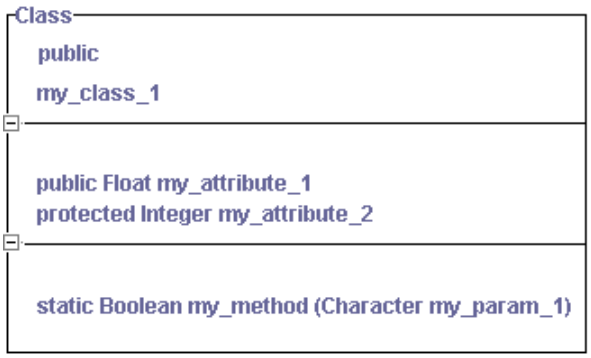


FIG. 17 – Exemple de description

- La zone supérieure représente l'*"en-tête"* de la relation, c'est-à-dire son type (par exemple *Class* ou *Interface*), son nom, et enfin son (ses) qualifier(s).
- La zone du milieu contient la liste des attributs de la méthode, avec, pour chacun, sa liste de qualifiers, son type, puis son nom.
- La zone inférieure contient la liste des méthodes de la description. Pour chaque méthode, on verra la liste de qualifiers associés, le type de retour (seulement pour les méthodes de type "function", et enfin, entre parenthèses, la liste des paramètres passés à la méthode. Ces paramètres sont constitués de la même façon que les attributs (qualifiers, type, nom).

Des clics sur le "+" et le "-" permettent d'afficher ou non la zone réservée aux attributs ou celle réservée aux méthodes.

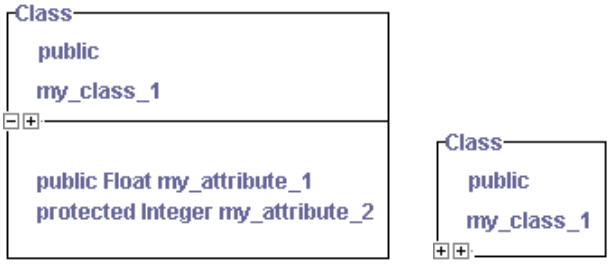


FIG. 18 – Exemple de descriptions sous forme compactée

### 3.4.5 Notes sur la généricité de l'interface

L'interface graphique permettant de créer ou de modifier des descriptions est générique sur le nombre de paramètres qu'elle peut afficher. Plus simplement, cela veut dire qu'elle affichera autant de paramètres que possède la description considérée (comme la liste des qualifiers disponibles, des relations sources et cibles disponibles...). Un choix de valeurs est alors proposé pour chaque paramètre de la description, à l'exception du nom, des attributs et des méthodes présents dans tous les langages à objets.

Malheureusement, la bibliothèque OFL-J ne nous permet pas encore de gérer l'association entre une donnée et sa correspondante méta. Donc, malgré la généricité de l'interface, nous sommes obligés de prévoir exactement le nombre et la nature des paramètres à prendre en compte pour chaque description. Cela dit, il sera possible d'inclure autant de panels d'informations qu'il y aura de paramètres pour une description donnée. Dans le cadre de notre projet, nous avons choisi de proposer uniquement les qualifiers d'une description en attendant qu'OFL-J évolue, pour nous permettre de mettre en relation une donnée et sa correspondante méta (cf figure 14).

## 3.5 Limitations

### 3.5.1 Les relations

A l'heure actuelle, la gestion des relations est incomplète : l'architecture à respecter permet de développer facilement des relations, mais faute de temps nous n'avons pas pu implanter les classes correspondantes. Tout de même, un squelette des fichiers est présent, ce qui facilitera l'éventuelle poursuite de notre application. Pour l'heure, nous ne pouvons donc pas créer ou modifier des relations.

### 3.5.2 Les vues

L'un des intérêts du modèle MVC, nous l'avons vu précédemment, est la possibilité d'interchanger des vues sur les composants description et relation. Cependant, seule la vue "standard", chargée par défaut, est élaborée. Nous avons créé une autre vue qui ne diffère que par la couleur d'arrière-plan (rouge au lieu de blanc). En effet cela paraît inutile et trivial, mais le principe est intéressant ; l'utilisateur pourrait facilement créer ses propres vues, en s'aidant de nos modèles. Celles-ci pourraient par exemple gérer des icones ou plus de couleurs pour différencier les signatures des méthodes...
Par contre, la limitation principale est l'obligation de gérer les "+" et les "-" ; c'est-à-dire que chaque vue doit se réserver "deux espaces", qui seront le plus souvent utilisés pour faire apparaitre ou disparaitre les attributs et les méthodes d'une description. Dans le cas des relations, on pourrait décider de ne pas implanter les methodes qui gèrent l'apparition et la disparition d'éléments.

### 3.5.3 Les contrôles

Nous avons réalisé assez peu de contrôles. Etant donné qu'OFL-J ne fournissait pas de contrôles spécifiques pour un langage particulier, nous avons décidé de ne pas coder "en dur" certaines interdictions qui pourraient s'avérer justes pour un autre langage. Mais nous aurions dû implanter certains contrôles qui étaient considérés comme valides quelque soit le langage à objects (par exemple interdire qu'un attribut soit à la fois public et private).
De plus, les relations n'étant pas complètement terminées, les principaux contrôles n'ont pas pu être implantés (comme un héritage multiple autorisé...).

### 3.5.4 XML

Faute de temps, nous n'avons pas implanté le chargement ni la sauvegarde des applications en XML. C'est pourquoi, les éléments des menus concernant cette partie ne sont pas fonctionnels.

## 3.6 Outils et documentation

Nous avons programmé sous emacs et grâce à l'outil JBuilder de Borland. En ce qui concerne l'environnement Java, nous nous sommes servis du JDK 1.3.1 (ou supérieur). Mais nous avons rencontré quelques difficultés avec le JDK 1.4 car ce dernier posait des problèmes avec certains *lookAndFeels*[4], et de plus, il semblait ne pas gérer le chargement dynamique de classes (à l'aide d'un *classloader*) depuis un répertoire contenant des espaces (ce qui est souvent le cas sous Windows) ; le JDK 1.3.x, lui, fonctionnait.

La documentation a été générée grâce à l'outil Javadoc à l'aide du module `iDoclet.jar` qui permettait de construire une documentation HTML avec des préconditions, postconditions et des invariants.

Nous avons utilisé pour ce TER de nombreux documents, dont la thèse de Pierre CRESCENZO, qui porte sur la méta-programmation selon le modèle OFL, les prototypes d'étudiants de LPMI et DESS ISI, ainsi qu'un ouvrage sur les design-patterns[5] en Java.

## 3.7 Structure d'OFL-ML sous forme de diagrammes de classes

Le diagramme de la figure 19 représente l'interface d'OFL-ML, c'est-à-dire l'arbre hiérarchique et de la zone d'affichage.

Le diagramme de la figure 20 représente l'architecture d'OFL-ML, c'est-à-dire le modèle MVC ainsi que ses interfaces.

[4] Les styles et les caractéristiques spécifiques d'une interface graphique
[5] Patrons de conception

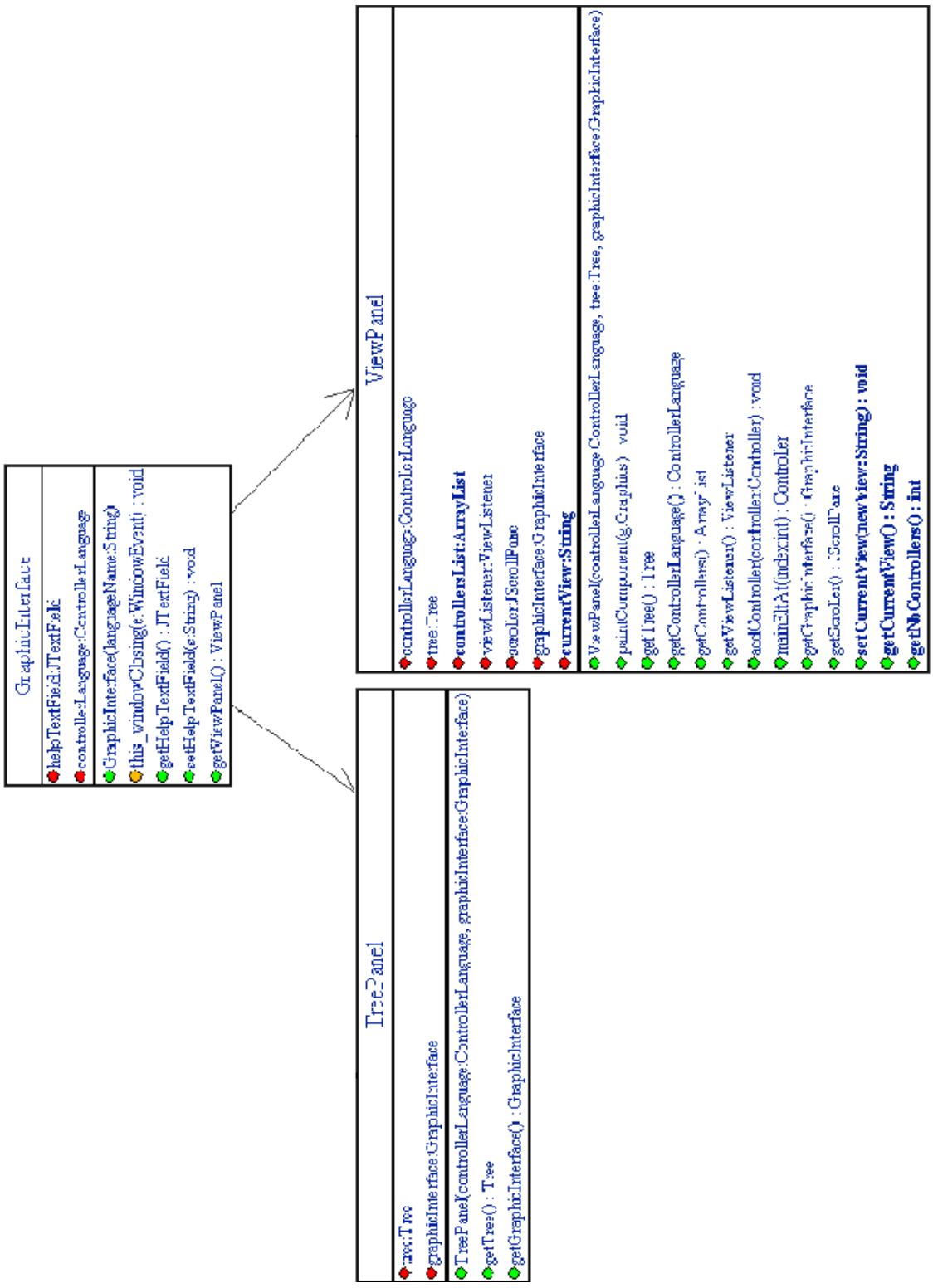


FIG. 19 – Les classes qui représentent l'interface graphique

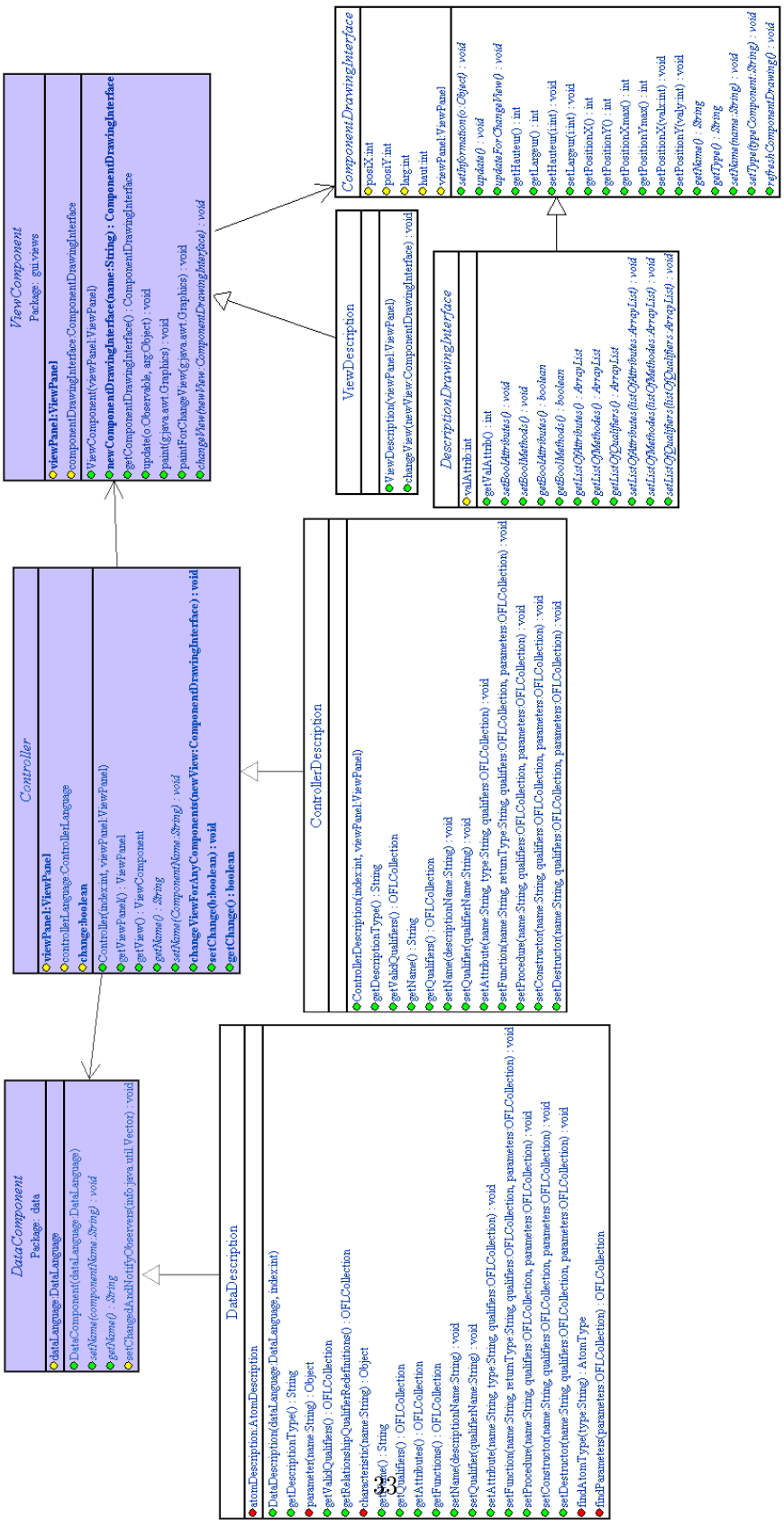




FIG. 20 – La structure des classes mettant en évidence le modèle MVC

## 3.8 Gestion du temps

### 3.8.1 Travail accompli en fonction du planning initial

Le tableau ci-dessous représentait le planning initial que nous nous étions fixé. Ces estimations ont plus ou moins été respectées hormi le volume horaire assigné pour l'étude du code existant. En effet il a été bien supérieur à ce que l'on avait imaginé, car comprendre la bibliothèque OFLJ (composé de plus de 300 classes) ainsi que récupérer efficacement le code d'autrui, n'ont pas été des tâches évidentes.

Comme nous étions les premiers "beta-testeurs" d'OFL-J (car les premiers à instancier certaines classes), nous avons décelé quelques problèmes. Nous avions le devoir de prevenir nos encadrants à chaque problème et à chaque erreur rencontrés. Ainsi, nous avons participé à l'évolution et au développement de la bibliothèque OFL-J. Nous ne pouvions, théoriquement en aucun cas, nous permettre de modifier le paquetage OFLJ, car il devait être extérieur à notre application, et nous devions tout simplement utiliser la réification des languages qu'il proposait. Ainsi, autant que possible, nous communiquions avec nos encadrants.

Pour ces multiples raisons, notre TER s'est vu retardé considérablement, bien qu'il était quasiment impossible de remplir la totalité du cahier des charges.

| | | |
|---|---|---|
| Etude des modèles : | **Modèle OFL (thèse)**<br>*80 heures* | **Modèle MVC (design-patterns)**<br>*60 heures* |
| Etude du code existant : | **Prototypes d'OFL-ML**<br>*90 heures* | **Bibliothèque OFLJ**<br>*? ? ? heures* |
| Conception : | **Architecture MVC**<br>*50 heures* | **Interface graphique**<br>*50 heures* |
| Développement : | **IHM**<br>*70 heures* | **Vues, données, contrôleurs**<br>*90 heures* |
| Développement : | **Composants d'OFL-J**<br>*? ? ? heures* | **RTTI, reflexivité**<br>*40 heures* |
| Documentation : | **HTML**<br>*40 heures* | **Pré-postconditions**<br>*15 heures* |

FIG. 21 – Planning initial

Le tableau ci-dessous permet de comparer notre planning initial avec le temps de travail effectif fourni.

| | | |
|---|---|---|
| Etude des modèles : | **Modèle OFL (thèse)**<br>*120 heures* | **Modèle MVC (design-patterns)**<br>*40 heures* |
| Etude du code existant : | **Prototypes d'OFL-ML**<br>*125 heures* | **Bibliothèque OFLJ**<br>*60 heures* |
| Conception : | **Architecture MVC**<br>*55 heures* | **Interface graphique**<br>*55 heures* |
| Développement : | **IHM**<br>*75 heures* | **Vues, données, contrôleurs**<br>*75 heures* |
| Développement : | **Composants d'OFL-J**<br>*110 heures* | **RTTI, reflexivité**<br>*100 heures* |
| Documentation : | **HTML**<br>*40 heures* | **Pré-postconditions**<br>*20 heures* |

FIG. 22 – Estimation du temps de travail effectif

### 3.8.2 Diagramme de Gantt

Voici un diagramme de Gantt qui fait apparaitre la répartition des activités dans le temps ainsi que l'affectation des individus, durant l'élaboration de ce projet.

**Légende :**
C : Christophe GARABEDIAN ; G : Gabriel SPINEK ; T : Thierry TEBOUL

| **Activités / Semaines** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | **pers. :** |
|---|---|---|---|---|---|---|---|---|---|
| Etude modèle OFL (thèse) | C,G,T | C,G,T | | | | | | | 6 |
| Etude prototypes | | C,G,T | C,G,T | G,T | | | | | 8 |
| Etude modèle MVC | | | C,G,T | C | C | | | | 5 |
| Etude bibliothèque OFL-J | | | C,G,T | G | G | G | | | 6 |
| Conception Architecture MVC | | | | C,G,T | C | C | | | 5 |
| Conception interface graphique | | | | C,G,T | C,T | G,T | | | 7 |
| Dével. IHM | | | | C,G,T | T | T | T | | 6 |
| Dével. modèles, vues, contrôleurs | | | | C,G,T | C,G | C | C | | 7 |
| Dével. et utilisation d'OFL-J | | | | C,G,T | C,G,T | G | G | G | 9 |
| Dével.t RTTI, reflexivité | | | | | C,G,T | C,G,T | C,G | C | 9 |
| Documentation javadoc | | | | | C,G,T | T | T | T | 6 |
| Documentation pré-postconditions | | | | | | C,G,T | T | T | 5 |
| **pers. :** | 3 | 6 | 9 | 19 | 17 | 14 | 7 | 4 | p/sem : 79 |

FIG. 23 – Diagramme de Gantt

# 4 Conclusion

## Bilan

Afin de poursuivre l'implantation d'OFL-ML, il serait conseillé d'utiliser notre squelette fourni afin de développer des relations, en respectant le modèle MVC mis en place, c'est-à-dire développer leur vue, leur modèle et leur contrôleur. Suite à cela, il resterait à développer quelques contrôles sur ces relations. Une autre perspective serait de générer un squelette du code suivant le diagramme de classe depuis d'OFL-ML, suivant le langage choisi.

La réalisation d'OFL-ML fut très enrichissant sur de nombreux critères. Tout d'abord, il a fallu intégrer un univers de Recherche en perpétuelle évolution où la profonde compréhension d'un sujet est de rigueur. Puis il a fallu travailler en équipe, et notamment plannifier le travail et l'ordonnancer en tâches, afin de se les répartir au mieux pour avancer rapidement et régulièrement.

## Perspectives

Aujourd'hui, un des principaux problèmes afin de développer une application réside en le choix d'un langage de programmation. En effet, choisir parmi la quantité de langages disponibles est une tâche délicate, étant donné que chacun d'eux possèdent leurs propres spécificités, avec leurs qualités et leurs faiblesses.
L'idéal serait d'avoir un langage universel, où l'on puisse définir sa propre sémantique opérationnelle. Un tel langage est une utopie. Par contre, en proposant la possibilité de modifier un langage existant ou encore de créer son propre langage adapté, le modèle OFL permet de résoudre ces problèmes. L'interface OFL-ML permet d'utiliser le modèle OFL pour ouvrir la voie vers un nouveau type de programmation, enfin libéré des contraintes trop souvent problématiques imposées par les lagages de programmation actuels.