Rapport de projet

Par
Tachier – Christophe

Date : le 11 avril 2002

Persistance de données dans le projet OFL

Lieu où est effectué le projet : ***Laboratoire I3S***
Encadreurs(s) : ***M Pierre Crescenzo***
M Philippe Lahire

OFL est le sigle de Open Flexible Languages et le nom d'un méta-modèle des langages de programmation objets à classes. Il repose sur trois concepts essentiels de ces langages : les descriptions qui sont une généralisation de la notion de classe, les relations telles l'héritage ou l'agrégation et les langages eux-mêmes. OFL offre un paramétrage de ces trois concepts dans le but d'adapter leur sémantique opérationnelle aux besoins du programmeur.
OFL a besoin de conserver certaines entités dans un formalisme standard, pour permettre un partage aisé et durable pour pouvoir envisager leur réutilisation.
OFL-DB constitue donc l'interface au socle de données sur lequel reposent les autres outils OFL. C'est par OFL-DB que l'on fera persister ces données. Pour cela, il faudra étudier les différents systèmes persistants, en tirer les avantages et inconvénients de chacun. Par la suite, l'outil sera développé en Java et implémentera donc le choix du système persistant.

Sommaire

I. Introduction

OFL (Open Flexible Language) est un modèle méta-objet qui peut décrire les langages à objets les plus courants (Java, Eiffel, C++, …).

Dans le projet OFL, il existe quatre outils pour sa mise en œuvre.
Les quatre outils sont :

- OFL-META
- OFL-ML
- OFL-PARSER
- OFL-DB

Nous allons présenter succinctement chacun de ces outils.

I.1°) OFL-META

Son but est de donner au méta-programmeur la capacité de décrire un langage. Pour cela, il crée un OFL-Composant instance d'OFL-Concepts, qui peut être un composant-description ou un composant-relation. Par exemple, la relation d'héritage est un composant-relation ou encore une interface Java est un composant-description. En fait, le méta-programmeur doit pouvoir créer ses composants en les paramétrant. Chaque composant n'est qu'une (longue) liste de paramètres avec, des valeurs par défaut si le méta-programmeur crée de toutes pièces son composant ou avec des valeurs à modifier s'il l'a importé au préalable.

L'outil se présente sous forme d'interface graphique. En créant un nouveau composant, une liste de paramètres devient accessible dans une frame séparée. Cette liste de paramètres est fixée et le méta-programmeur n'a plus qu'à donner les valeurs souhaitées aux paramètres.

I.2°) OFL-ML

A ce stade, le méta-programmeur est intervenu et a crée son langage. Il laisse maintenant la place au programmeur. Le nom d'OFL-ML dérive d'UML car il emprunte une partie de son formalisme graphique. OFL-ML permet de créer graphiquement des descriptions, de décrire des relations entre ces descriptions et de donner un corps aux méthodes, initialiseurs, etc…
Le programmeur doit avant tout importer le langage crée avec OFL-META et peut ensuite dessiner le graphe de descriptions et de relations, de son application.
Cet outil est basé sur le mécanisme d'utilisation de « Rational Rose ». C'est un outil dynamique puisqu'il change de comportement (vis à vis de l'utilisateur) selon le langage d'OFL-META mis en œuvre. Concrètement, si dans un langage on a défini que l'héritage multiple était possible, alors OFL-ML créera une nouvelle représentation qui correspond à cet héritage multiple, au lieu d'avoir plusieurs flèches simples de généralisation/spécialisation (et donc il n'y aurait aucune distinction avec l'héritage simple). D'une autre manière, si maintenant le langage interdit l'héritage multiple, la flèche créée spécialement pour l'héritage multiple deviendra inaccessible.

OFL-ML sert à concevoir une application (par dessin), qui sera codée plus tard, et dont le squelette d'application a été généré par OFL-PARSER à partir du schéma dessiné dans OFL-ML. Il se présente de la même façon que beaucoup d'outils de conception, à savoir une frame avec la liste des entités que l'on ait en train de créer (en mode texte), et une autre frame avec le dessin composé de ces entités. Il existe aussi une barre, séparant les deux frames, contenant les icônes des entités constructibles (définies selon le langage importé) et sur lesquelles il suffira de cliquer pour les dessiner.

I.3°) OFL-PARSER

Cet outil vient à la suite d'OFL-ML. Une fois les descriptions et relations de l'application dessinées, on peut dans OFL-ML intégrer du code, mais celui-ci ne sera pas interprété. C'est le rôle d'OFL-PARSER de générer le code en Java pour rendre l'application exécutable. C'est également lui, qui va intégrer ce code dans les méthodes appropriées, compiler l'ensemble, pour finalement l'exécuter. L'implémentation d'OFL-PARSER est de loin la plus exigeante. En effet, il ne s'agit pas d'un simple compilateur (qui est déjà lourd à écrire), il doit en plus prendre en compte le langage crée avec OFL-META. Là encore, la structure dynamique rend difficile la tâche.

OFL-PARSER compile et réalise un exécutable.

I.4°) OFL-DB

Il s'agit du sujet sur lequel je travaille. En fait, OFL-DB n'est pas à la suite de l'un des outils précédents. Il les englobe tous les trois. Il intervient à différents niveaux et plus exactement entre chaque outil. L'existence de plusieurs outils implique d'une part une communication entre eux, d'autre part la persistance des données qu'ils manipulent.

OFL-DB intervient, comme on peut le voir sur la Figure1 ci-dessous, à différents stades qui sont en fait les passages d'un outil à un autre.

Tout d'abord, OFL-DB intervient à la fin d'OFL-META. En effet, lorsque le méta-programmeur a terminé son langage, il doit le sauvegarder pour le mettre à la disposition du programmeur (c'est à dire OFL-ML). OFL-MATA doit aussi pouvoir charger et recouvrer le langage qu'il a pu créer quelques temps auparavant. OFL-DB doit aussi permettre d'importer des bibliothèques d'OFL-Composants dans OFL-META.

Ensuite, le programmeur pourra créer son application via OFL-ML en important le langage. Comme pour OFL-META, le programmeur doit sauvegarder son application (et la charger ultérieurement si des modifications sont à apporter).

Enfin, il intervient pour permettre à OFL-PARSER de générer le code à partir de l'application dessinée.

Les entités de base du modèle sont conservées par OFL-DB.

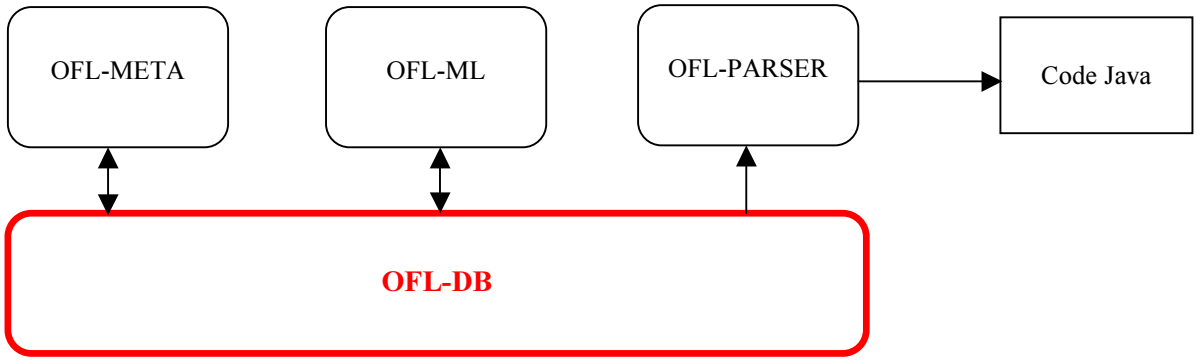


Figure 1

II. Cahier des charges

Ce projet consiste, tout d'abord à étudier, les différentes alternatives possibles de conservation des données (bases de données, systèmes persistants, XML, …) et d'en tirer les avantages et inconvénients. Par la suite, cette étude mènera à la réalisation d'un outil logiciel (OFL-DB), serveur d'objets persistants pour les autres constituants de ce nouvel environnement de programmation.

II.1°) Etude des moyens de conservation de données

La première partie consiste à étudier les différents moyens de persistance de données. Cependant, un format existe déjà. Le format est une base de données orientée objets nommées POET. POET respecte également les spécifications de l'ODMG et est capable d'exporter ses données sous un format XML, capacité dont OFL ne tire pas parti. Le choix de POET reste cependant anecdotique en regard des bases ODMG, XML et Java optées pour leur propriété de standards.

En attendant la fin de cette étude préalable, un format texte devra être proposé.

II.2°) Bibliothèque OFL

Une bibliothèque OFL très simplifiée, commune aux trois projets, devra être réalisée. Cette bibliothèque est un travail collectif et se fera en collaboration avec OFL-META et OFL-ML. Cette bibliothèque contient, en fait, la description du langage. C'est en instanciant les classes de cette bibliothèque que l'on pourra créer un langage.

II.3°) Réalisation de l'outil OFL-DB

Après l'étude préalable, une implémentation du système devra être proposée. L'objectif initial est de couvrir les trois outils d'OFL avec évidemment un ordre logique qui est de commencer par la sauvegarde et le chargement d'OFL-META pour terminer par le chargement et la sauvegarde d'OFL-ML.

III. Travail réalisé

III.1°) Etude des moyens de conservation

Dans un premier temps, il faut définir le format de la sauvegarde sachant que les objets à sauvegarder seront des objets Java. Pour cela, il faut recenser les différents moyens de persistance et voici la liste :

- base de données relationnelles
- base de données orientées objets
- fichier texte
- fichier XML

III.1°) a. Bases de données

Il faut séparer les deux types de SGBD que nous sommes susceptibles d'utiliser : les SGDB relationnels (SGBDR) et les SGBD objets (SGBDO). Les SGBDR ne sont pas adaptés pour sauvegarder directement des objets. Il faut recourir à des identificateurs pour lire/écrire les objets. Le schéma de la base n'est pas évident à déduire de la hiérarchie des objets. Les SGBDO s'adaptent plus facilement à la POO. La persistance des objets est beaucoup plus facile à gérer.

Avantages :

- requêtes sur de grosses quantités de données
- outils existants
- cohérence assurée à l'intérieur de la BDD
- adapté aux très grandes quantités de données
- 'scalable'

Inconvénients :

- peu portable
- lenteur/lourdeur
- nécessite une BDD
- administration de la BDD
- indépendance limitée
- cohérence mémoire/BDD difficile à gérer, à écrire
- transfert via le réseau pénible (nécessite une forme intermédiaire)

Limites des SGBD relationnels et nouveaux besoins

Les SGBD relationnels ont à leur avantage:

- le modèle de données est très simple et donc facile à comprendre pour les utilisateurs
- le modèle repose sur une base formellement définie ce qui a permis de définir des méthodes de conception de schémas (théorie de la normalisation) et des langages de manipulation de données (LMD) standardisés (SQL, QUEL...).

Mais, le développement de nouvelles applications, différentes des applications de gestion classiques, avec de nouveaux besoins, ont révélé les limites du modèle relationnel, notamment:

- le modèle de données est trop simple et ne permet pas de représenter facilement les entités du monde réel qui sont souvent plus complexes qu'une relation. Dans les schémas, les entités du monde réel sont éclatées en plusieurs relations, ce qui multiplie les jointures dans les requêtes des utilisateurs. De plus le modèle relationnel ne permet pas de représenter explicitement les différents types de liens sémantiques qui peuvent lier des entités : composition, généralisation / spécialisation, association...
- l'incompatibilité des LMD relationnels et des langages de programmation:
  - les LMD sont déclaratifs et fournissent en résultat un ensemble de tuples, alors que les langages de programmation sont impératifs et travaillent sur un élément à la fois
  - les types de **données** manipulés par les langages de programmation sont plus complets et plus complexes que ceux des LMD relationnels.
- le développement d'applications n'est pas satisfaisant : lenteur du développement, résultat souvent décevant (ne correspond pas ou mal aux besoins), applications difficilement maintenables. Ces nouvelles applications sont, par exemple, la conception ou production assistée par ordinateur, le génie logiciel, la bureautique, les systèmes d'informations géographiques... qui gèrent des objets plus complexes et plus volumineux, comme les textes, graphiques, cartes, images, dessins multidimensionnels, vidéos, sons...

Les bases de données orientées objets (BDO) sont nées de la convergence de deux domaines :

- les bases de données;
- les langages de programmation orientés objets, tels que Eiffel, Smalltalk, C++, Java...

L'objectif de ces langages est d'accroître la productivité des développeurs en permettant de créer des logiciels structurés, extensibles, réutilisables et de maintenance aisée.

Leurs principes essentiels sont :

- les objets, qui comportent deux parties: leur valeur, et les opérations, appelées méthodes, qui permettent de les manipuler. La valeur est cachée. L'accès et la mise à jour des objets se font par appel des méthodes. Cela rend plus facile la maintenance des logiciels construits selon ce paradigme.

- l'héritage, qui permet à une classe d'objets d'avoir les mêmes propriétés (structure de données et méthodes) qu'une autre classe sans avoir à les redéfinir. C'est l'héritage qui permet d'étendre et de réutiliser facilement des logiciels.

Les bases de données orientées objets sont caractérisées par quatre points essentiels:

- un modèle de données qui permet de représenter des structures de données complexes
- les données et les traitements ne sont plus séparés. La dynamique (les méthodes) fait partie de la déclaration des objets;
- l'héritage;
- tout objet possède une identité qui le distingue de tout autre objet, même s'ils ont la même valeur.
- il n'y a plus d'incompatibilité entre le langage de programmation et le langage de manipulation des données.

Les SGBDO sont un domaine en évolution et il n'y a pas de consensus, tant sur le modèle de BDO que sur le LMD objets, alors que de nombreux produits sont proposés sur le marché. Il y a actuellement deux propositions rivales de norme : ODMG qui a été défini en 1993 et est promu par un groupe de constructeurs, et SQL3 qui est une extension de SQL, compatible avec SQL, proposé par le comité de normalisation ANSI, mais dont la spécification n'est pas encore terminée.

III.1°) b. Format texte

Il n'y a aucun intérêt à utiliser à l'heure actuelle un format ASCII simple.
La sauvegarde et le chargement du graphe des objets à maintenir doivent être programmés. Un format doit être défini ce qui peut être une source d'erreurs et de travail supplémentaire inutile.

Concernant les mécanismes, nous devons observer que :

- La sauvegarde sous forme de fichiers de texte est transparente mais demande une grande attention. Il ne faut pas que les caractères spéciaux appartenant aux chaînes de caractères à sauvegarder (*e. g.* caractère de nouvelle ligne, espace, tabulation etc.) soient confondus avec les caractères spéciaux, qui servent de délimiteurs de champs dans les fichiers de sauvegarde. Nous pouvons imaginer utiliser une suite improbable de deux caractères spéciaux pour créer les délimiteurs de champs, et même nous pouvons imaginer mettre en œuvre un mécanisme de dédoublement de caractères pour être vraiment certain que la sauvegarde textuelle sera toujours correcte, ce qui est clairement inadmissible.
- D'autre part, la sauvegarde sous forme de fichiers de texte n'est pas *sécurisée*. Les fichiers peuvent facilement être modifiés en dehors du programme.

III.1°) c. Format XML

Après plus de 30 ans de bons et loyaux services, l'ASCII et ses différentes adaptations pour les caractères non américains (comme nos accents) n'ont plus la cote. Il est incapable de gérer correctement les différents caractères nationaux sans s'encombrer des infâmes " code page " qui précisent la nature du texte. Plus grave, il n'offre aucun format standard permettant de structurer le document qu'il contient.

Descendant du SGML (Structured Genaral Markup Language), XML (Extensible Markup Language) a tous les atouts pour lui succéder. Compatible UNICODE (jeu de caractères sur 16 ou 32 bits) et normalisé par le W3C, XML décrit un document en entourant ses différents éléments par des balises (tags) comme celles utilisées dans un document HTML. Contrairement à celui-ci, le nom de ces balises, leurs attributs et le niveau d'imbrication sont librement choisis par l'utilisateur. XML n'est donc pas un format de plus mais bien un langage permettant de décrire et manipuler le contenu des documents.

De plus en plus d'outils permettent de manipuler de l'XML et les avantages de l'XML par rapport à l'ASCII simple sont nombreux.

Avantages :

- lisibilité, facilité d'édition
- peu sensible aux changements du code
- indépendant de la plate-forme ou du système de fichiers
- outils existants pour lire/écrire l'XML
- validation (déjà écrit et testé)
- standard

Inconvénients :

- entièrement à définir et à mettre en place
- format verbeux, donc long à manipuler

Conclusion

Inutile de faire durer le suspens, c'est le format XML qui a été choisi.

XML est largement de plus en plus utilisé, de plus beaucoup d'outils sont disponibles, ce qui le rend d'autant plus attrayant.

Il n'était pas sérieusement envisageable de sauvegarder ces informations dans une base de données. La première raison, c'est qu'il faut posséder le SGBD correspondant. Or, on ne peut pas imposer aux utilisateurs de se fournir un SGBD surtout s'il est payant comme ORACLE par exemple. De même, dans le cas d'un SGBD gratuit, la maintenance est beaucoup plus lourde. D'ailleurs, il suffit de remarquer que la très grande majorité des produits de conception UML créer un format spécifique de sauvegarde.

Remarque :

Le nombre de sauvegardes est très supérieur au nombre de chargements. Typiquement il y a un chargement au démarrage et des sauvegardes régulières jusqu'à l'arrêt ou au plantage.

L'utilisation d'une BDD est un gros désavantage vis à vis du point énoncé. De plus, cela oblige à installer et à administrer une BDD, ce qui n'est pas à la portée de tout le monde. Néanmoins la BDD est le seul système 'scalable'. Dans le cas d'une grande quantité de données, seule la BDD (qui est fait pour ça...) est capable de gérer le volume et d'offrir des performances acceptables. Le fait de cumuler le stockage tout en permettant l'accès est clairement un avantage. Cependant, la séparation des deux fonctions coûte trop cher pour les gros volumes.

Le format XML n'est pas très adapté à des sauvegardes fréquentes et rapides en raison de sa verbosité. Par contre, il est pratique pour le stockage ou le transfert.

Quant au format texte, il n'est pas nécessaire d'en parler puisque XML reprend l'intérêt d'un format texte et y ajoute une multitude d'avantages.

Pour voir le format texte qui a été proposé, se référer à l'annexe. Ce format n'est bien évidemment plus d'actualité.

III.2°) Bibliothèque OFL

III.2°) a. Evolution de la bibliothèque

Au début du projet, nous devions créer notre propre bibliothèque simplifiée qui nous a aidés à approcher les mécanismes des différentes classes et notions du projet.
Plus tard, une version (une fois de plus simplifiée) de la bibliothèque nous a été proposée, par Pierre Crescenzo et Philippe Lahire, puisque le but est d'intégrer le travail effectué à la vraie bibliothèque. Petit à petit, cette bibliothèque s'est étoffée par l'apport d'OFL-META et OFL-DB (OFL-ML étant moins impliqué dans celle-ci, de part son sujet).

Cette bibliothèque a constamment évoluée ce qui a rendu son implémentation difficile. En effet, au fur et à mesure de nouvelles méthodes sont venues se greffer. Les changements les plus brutaux se sont fait au niveau de l'architecture même, notamment lorsque des attributs ont été déplacés dans une super-classe. Cette manipulation n'a absolument rien à voir avec une mauvaise conception. En fait, ces attributs n'ont pas été déplacés mais « cloner ». Le but de la manœuvre était de pouvoir permettre la séparation des composants et des concepts lors du chargement et de la sauvegarde.

III.2°) b. Structure de la bibliothèque

Chaque classe (et donc chaque objet de la classe) est responsable de sa propre sauvegarde. Ceci donne un caractère *modulaire* qui permet la modification locale de l'information sauvegardée (ajouter un attribut persistant) sans avoir à modifier la manière dont est effectuée la sauvegarde des autres classes. Toute l'information indispensable à la reconstitution du graphe est sauvegardée.

Chaque entité n'est sauvegardée qu'une seule fois afin de réduire la taille du fichier et afin de donner de meilleures performances aussi bien au chargement qu'à la sauvegarde.

Une explication sommaire de la structure OFL (Open Flexible Languages) s'impose pour comprendre la suite du rapport. Commençons par dire que dans OFL, il y a des concepts et des composants. Il existe trois types de concepts : des concepts-descriptions, des concepts-relations et des concepts-langage. De même, il existe trois types de composants : : des composants-descriptions, des composants-relations et des composants-langage. En réalité, ils ne sont pas au nombre de trois mais de quatre (le concept-relation et le composant-relation se spécialisant) mais nous garderons en tête qu'il n'y en a que trois.

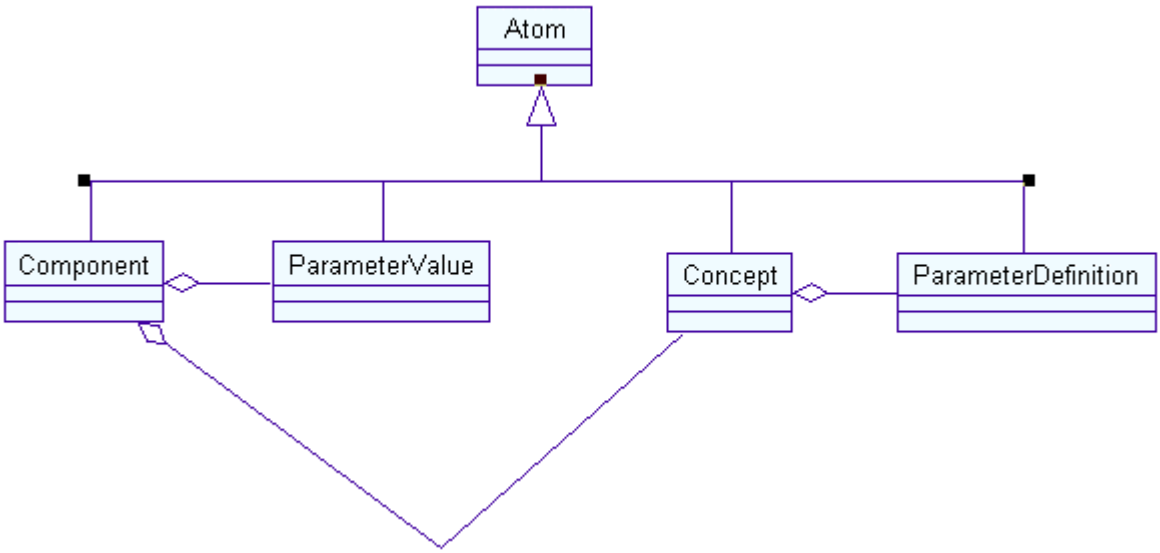
Un composant-langage est un composant logiciel qui offre une définition d'un langage objet à classes. C'est un méta-langage.

Un composant-description est un composant logiciel qui offre une définition d'une description. C'est une méta-classe. Une description est une entité qui offre une définition d'une instance. Les classes sont les descriptions les plus connues du langage mais les interfaces ou types de tableau Java sont aussi des descriptions.

Un composant-relation est un composant logiciel qui offre une définition d'une relation. C'est une méta-relation. Une relation est une entité d'une application qui assure la liaison entre d'autres entités, que ces dernières soient des descriptions ou des instances finales. L'hériatge, l'aggrégation ou l'instanciation sont des relations communes dans les langages objets à classes.

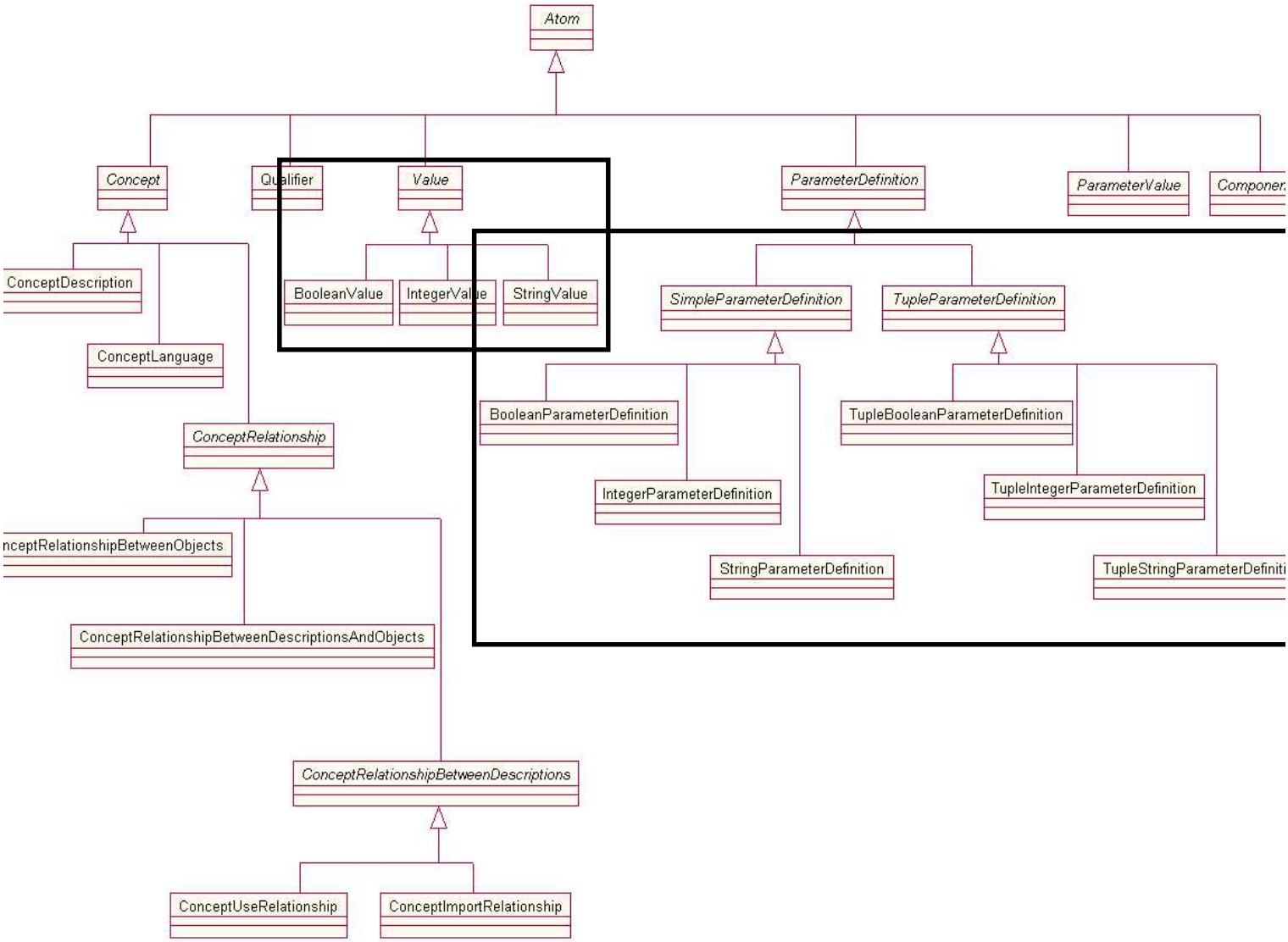
Un concept est un méta-composant.

Voici un schéma (ultra-simplifié) qui va nous permettre de situer toutes ces entités



- Dans le langage que l'on crée, il n'y a qu'une seule instance des trois types de concepts.
- Les *Component* ont un attribut qui est un vecteur de *ParameterValue.*
- Les *Concept* ont un attribut qui est un vecteur de *ParameterDefinition.*
- Un *Component* a un attribut qui est une référence sur un *Concept*.
- La classe ParameterValue est intimement liée à la classe ParameterDefinition, de part le fait qu'un paramètre de définition est un méta-paramètre valeur (méta-ParameterValue)

Dans le schéma ci-dessous, représentant la structure **d'héritage** de la bibliothèque OFL, certaines classes sont manquantes pour des raisons de visualisation. En fait, il faut rajouter pour la classe abstraite *Component*, les mêmes sous-classes que celles de *Concept* (mais en remplaçant *concept* par *component*). De même pour les classes *ParameterDefinition* et *ParameterValue*. Dans cette bibliothèque, il y a exactement 42 classes.
Remarque : ce qui encadré dans le schéma, correspond aux classes ajoutées par OFL-DB (20 classes), par rapport à la bibliothèque OFL simplifiée de départ.



III.2°) c. Maintenance de la bibliothèque

Contrôler totalement la persistance des données devient vite lourd et difficile à maintenir. Si l'on veut ajouter un nouvel attribut à une classe, alors il faut modifier la manière dont les objets de la classe écrivent leur données dans les fichiers de sauvegarde, et symétriquement, la manière dont ils lisent ces données pendant le chargement.
Remarque : A priori, les classes sont définies et ne bougeront plus. Il est toutefois légitime de se poser la question, étant donné notre expérience sur ce projet (voir III.3°) a)

III.3°) Réalisation de l'outil OFL-DB

III.3°) a. Le parser XML

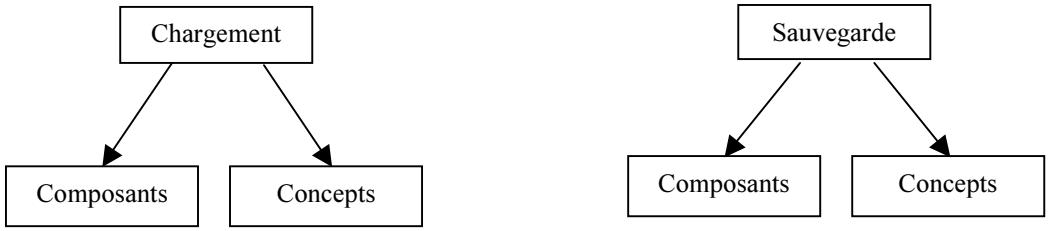
Il est plus aisé d'utiliser un analyseur/générateur XML. Le choix s'est porté vers JDOM qui est une bibliothèque Java capable de lire et écrire du XML.
Il a fallu donc étudier les méthodes utilisables, seules quelques méthodes sont nécessaires pour écrire un fichier XML.
Cette partie d'étude a été assez courte dans l'ensemble.

III.3°) b. Le XML Schema

Les fichiers XML sont généralement accompagnés d'une DTD ou plus récemment d'un XML Schema. Un schéma a donc été écrit pour donner la structure des fichiers XML. Aucune vérification automatique sur les fichiers XML n'est faite à partir du schéma. Il n'est là que pour présenter la structure. Ceci n'exclut pas le fait que les fichiers XML doivent s'y conformer. Vous pouvez vous référer à l'annexe.

III.3°) c. L'implémentation

OFL-DB doit sauvegarder et charger les composants et les concepts du langage. Il existe, donc, des méthodes séparées, qui traitent du chargement d'une part, et de la sauvegarde d'autre part, dont chacune d'elle peut ne sauvegarder/charger que les composants ou concepts.



Les classes de chargement et de sauvegarde sont dans un package séparé. Les méthodes « load » et « save » existent dans toutes les classes de la bibliothèque OFL.

Lors du chargement, une vérification de conformité est effectuée. En effet, les *ParameterValue* sont des instances des *ParameterDefinition*. Plus précisément, les *ParameterDefinition* donnent le nom, la valeur par défaut, les valeurs possibles, les redéfinitions possibles d'un *ParameterValue*. On peut donc, par exemple, vérifier que le *ParameterValue* du fichier XML existe et contient une valeur valide. Pourquoi faire cette vérification ? Tout simplement parce que le fichier peut être modifié aisément avec un éditeur de texte. Les fichiers XML sont lisibles et aussi modifiables. Par contre, on ne peut pas se prémunir d'une modification sur les concepts.

III.3°) d. Les difficultés

Une très grosse partie du travail a été d'essayer de produire du code propre, permettant une grande modularité et souplesse. Les commentaires réguliers dans le code étaient aussi une priorité, ainsi que les commentaires Javadoc.

De part l'existence de nombreuses classes abstraites (contenant également des méthodes abstraites), le polymorphisme fut le mécanisme objet de loin le plus utilisé. Ce qui en terme de modularité convient parfaitement, mais ajoute aussi de la complexité surtout lorsque les sous-classes de classes abstraites sont aussi des classes abstraites. On peut dire que ce point fut difficile à gérer, mais qu'une fois les méthodes d'une classe écrite, alors toutes les autres, se situant au même niveau dans la hiérarchie d'héritage, en bénéficient.

Le dernier point, où la difficulté fut grande, est la sauvegarde de paramètres « complexes ». Par « complexe », on entend tous les types contenant eux-mêmes des objets (par exemple, le type « Vector »).

Les *ParameterDefinition* encapsulés dans les *Concept* et les *ParameterValue* encapsulés dans les *Component* peuvent avoir trois types d'instances :

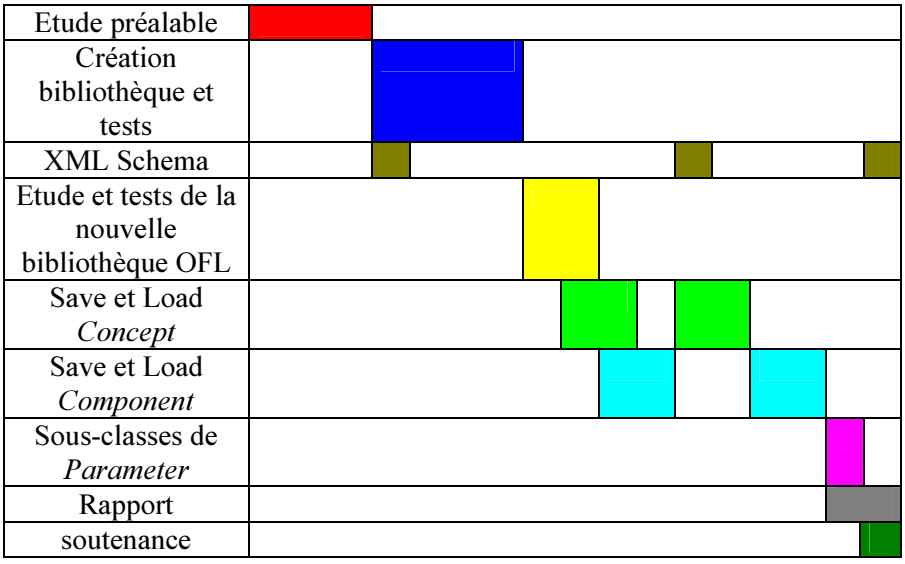
- simple (exemples : chaîne, booléen, entier, énuméré,...)
- tuple, autrement dit "collection d'un nombre fixe de valeur" (exemples : <>, <true, false>, <allowed, forbidden, allowed>, <<true, false>, <true, false>>, <{1, 3}, {3, 4}>,...)
- ensemble, autrement dit "collection d'un nombre indéterminé de valeur" (exemples : {3, 2, 1}, {{3, 3}, {5, 6}}, {{<"toto", false>}, {<"tata", false>, <"tutu", false>}},...)

La question est : comment faire pour créer les objets Java lors du chargement des paramètres de type "ensemble" quand on ne connaît pas leur type ? Bien évidemment, on ne sauvegardera pas les types en XML car sinon on se retrouverait dans le cas où l'on testerait tous les types possibles avant de décider quel constructeur appeler. On aurait quelque chose comme : "if(estDuType(String))then ….else ...
Ce qui n'est pas acceptable.
Vous parlerez alors de polymorphisme, ce qui est exact, et c'est ce qui a été mis en œuvre. Pourtant, le mécanisme rencontre tout de même des difficultés, dès lors que les instances d'ensemble ne contiennent pas le même type de données. Au moment de charger le paramètre, nous ne pouvons pas faire appel à la bonne classe car le type des éléments du vecteur reste *Object*. En réalité, nous devons produire de nouvelles classes pour permettre le chargement et la sauvegarde d'ensembles (et en utilisant le polymorphisme).

III.3°) e. Le travail à faire

La partie concernant la sauvegarde des objets d'OFL-ML n'a pas été implémentée.
Pour ce qui est de la partie d'OFL-META, les paramètres (*ParameterDefinition* et *ParameterValue*) dont le type est une collection d'objets de types différents ne sont pas pris en compte. Toutes les sous-classes des classes *ParameterDefinition* et *ParameterValue* ne sont pas intégrées aux deux autres projets.

IV. Planning



V. Conclusion

En conclusion, ce projet a permis d'obtenir une première expérience sur un travail « long ». En effet, c'est la première fois que nous avons travaillé sur un sujet de six mois environ. Il est vrai que ce n'était pas un travail à temps plein. Toutefois, nous remarquerons qu'en terme de gestion du temps, c'est la même chose. Plus clairement, les objectifs à la fin de chaque semaine sont moindres que s'il s'agissait d'un travail à temps complet, mais le déroulement est exactement le même. Il a fallu planifier (bien que cette contrainte ait été très dure à tenir), avoir quelques rencontres avec les autres groupes pour mettre en coordination le travail, observer les éventuelles modifications selon les besoins de chacun, faire un code propre, lisible et commenté (qui sera relu par une autre personne, ou bien soi-même !).

De plus, une expérience non négligeable en Java, et plus généralement sur la technologie objet, est un gain très appréciable dans la poursuite de ma carrière. Je pourrais mettre à contribution cette expérience aussi bien pour mon stage que pour mes futurs emplois, puisque c'est dans les technologies objets que je souhaiterais poursuivre.

De même, l'étude préalable sur la persistance sera un plus lors de mon stage, pour percevoir les différents types de stockage ainsi que les avantages et inconvénients de chacun, puisque le stage intègre également une partie base de données. Au-delà du stage, la recherche sur des technologies ne peut être qu'un « plus ».

VI. Bibliographie

http://www.w3.org/TR/REC-xml (site du W3C définissant le langage XML et sa structuration)

http://www.w3.org/XML/Schema (site du W3C définissant le XML Schema)

http://www.jdom.org/ (site permettant de télécharger la bibliothèque JDOM ainsi que sa documentation)

Thèse de Pierre Crescenzo (décembre 2001)
Titre : OFL, modèle pour paramétrer la sémantique opérationnelle des langages à objets
Université : Université de Nice-Sophia Antipolis

VII. Annexe

VII.1°) Format texte proposé

De part l'architecture actuelle et de son évolution future, on peut proposer un format texte simple d'utilisation et peu lourd. En effet, le langage crée par OFL-Meta peut se résumer en une liste d'OFL-Concept comportant plusieurs paramètres, eux-mêmes constitués d'un nom et d'une valeur.

#Nombre d' OFL-Concept :
<1> Type du 1er OFL-Concept , Nom du 1er OFL-Concept
#Nombre de paramètres de l'OFL-Concept
<1.1>Nom du 1er paramètre, Type du 1er paramètre, Valeur du 1er paramètre
<1.2>Nom du 2eme paramètre, Type du 2eme paramètre, Valeur du 2eme paramètre
<1.3>Nom du 3eme paramètre, Type du 3eme paramètre, Valeur du 3eme paramètre
... ...

<2> Type du 2eme OFL-Concept , Nom du 2eme OFL-Concept
#Nombre de paramètres de l'OFL-Concept
<2.1>Nom du 1er paramètre, Type du 1er paramètre, Valeur du 1er paramètre
<2.2>Nom du 2eme paramètre, Type du 1er paramètre, Valeur du 2eme paramètre
<2.3>Nom du 3eme paramètre, Type du 1er paramètre, Valeur du 3eme paramètre
... ...
...

Un format similaire a été proposé au projet OFL-ML.
Pour OFL-ML, on a besoin des informations suivantes :

- *pour chaque entité, le nom, les attributs, les méthodes, la taille de la fenêtre, son emplacement sur le schéma*
- *pour chaque relation, le nom, la source, la destination. Si la relation (sous forme de flèches) peut se courber ou se déformer pour ne peut plus être une simple ligne, il faut alors enregistrer sa taille, ses points de déformation,... (dans le but de rendre le schéma plus clair)*

#Nombre d'entités
<1> Nom de la 1ere entité
<taille> Taille en X de la fenêtre , Taille en Y de la fenêtre
<coord> Abscisse de la fenêtre , Ordonnée de la fenêtre
#Nombre d'attributs
Nom du 1er attribut
Nom du 2eme attribut
...
#Nombre de méthodes
Nom de la 1ere méthode
Nom de la 2eme méthode
...
<2> Nom de la 2eme entité
<taille> Taille en X de la fenêtre , Taille en Y de la fenêtre
<coord> Abscisse de la fenêtre , Ordonnée de la fenêtre
#Nombre d'attributs
<2.1a>Nom du 1er attribut
<2.2a>Nom du 2eme attribut
...
#Nombre de méthodes
<2.1m>Nom de la 1ere méthode
<2.2m>Nom de la 2eme méthode
...
...
#Nombre de relations
<1> Nom de la 1ere relation, Type de relation
<entité> Nom de l'entité source, Nom de l'entité destination
<2> Nom de la 2eme relation, Type de relation
<entité> Nom de l'entité source, Nom de l'entité destination
...
...

VII.2°) XML schema

Attention : ce XML Schema est une version qui ne prend pas en compte les dernières modifications.

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">

<!-- definition of simple type elements -->

<!-- definition of attributes -->

<!-- definition of groups -->

<!-- definition of complex type elements -->

<xsd:complexType name="componentLanguage">
 <xsd:sequence>
  <xsd:element ref="parameterValue"
        minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="conceptLanguage" ref="conceptLanguage">
  <xsd:element name="conceptDescription" ref="conceptDescription">
```

```
 <xsd:element name="conceptUseRelationship" ref="conceptUseRelationship">
 <xsd:element name="conceptImportRelationship" ref="conceptImportRelationship">
 <xsd:element ref="componentDescription"
        minOccurs="0" maxOccurs="unbounded"/>
 <xsd:element ref="componentUseRelationship"
        minOccurs="0" maxOccurs="unbounded"/>
 <xsd:element ref="componentImportRelationship"
        minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>


<xsd:complexType name="conceptLanguage">
 <xsd:sequence>
  <xsd:element ref="parameterDefinition"
        minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>


<xsd:complexType name="conceptDescription">
 <xsd:sequence>
  <xsd:element ref="parameterDefinition"
        minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>


<xsd:complexType name="conceptUseRelationship">
 <xsd:sequence>
  <xsd:element ref="parameterDefinition"
        minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>


<xsd:complexType name="conceptImportRelationship">
 <xsd:sequence>
  <xsd:element ref="parameterDefinition"
        minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>


<xsd:complexType name="parameterDefinition">
 <xsd:sequence>
  <xsd:element name="parameterDefinitionName" type="xsd:string"/>
  <xsd:element name="parameterDefinitionTypeValue" type="xsd:string"/>
  <xsd:element name="nbElements" type="xsd:integer"
        minOccurs="0" maxOccurs="1"/>
  <xsd:choice>
   <xsd:element name="parameterDefinitionDefaultValue" type="xsd:string"/>
   <xsd:complexType name="parameterDefinitionDefaultValue">
    <xsd:sequence>
     <xsd:element name="uplet" type="xsd:string"
```

```
            minOccurs="0" maxOccurs="unbounded"/>
     </xsd:sequence>
    </xsd:complexType>
   </xsd:choice>
   <xsd:choice>
    <xsd:element name="parameterDefinitionValidValue" type="xsd:string"
            minOccurs="0" maxOccurs="unbounded"/>
    <xsd:complexType name="parameterDefinitionValidValue">
     <xsd:sequence>
      <xsd:element name="uplet" type="xsd:string"
            minOccurs="0" maxOccurs="unbounded"/>
     </xsd:sequence>
    </xsd:complexType>
   </xsd:choice>
   <xsd:choice>
    <xsd:complexType name="parameterDefinitionValidRedefinition"
            minOccurs="0" maxOccurs="unbounded"/>
     <xsd:sequence>
      <xsd:element name="from" type="xsd:string">
      <xsd:element name="to" type="xsd:string">
     </xsd:sequence>
    <xsd:complexType name="parameterDefinitionValidRedefinition">
     <xsd:sequence>
      <xsd:complexType name="element" type="xsd:string"
            minOccurs="0" maxOccurs="unbounded"/>
       <xsd:complexType name="redefinition"
                    minOccurs="0" maxOccurs="unbounded"/>
        <xsd:sequence>
         <xsd:element name="from" type="xsd:string">
         <xsd:element name="to" type="xsd:string">
        </xsd:sequence>
       </xsd:complexType>
      </xsd:complexType>
     </xsd:sequence>
    </xsd:complexType>
   </xsd:choice>
</xsd:element>


<xsd:complexType name="parameterValue">
 <xsd:element name="parameterValueName" type="xsd:string">
 <xsd:choice>
  <xsd:element name="parameterValueValue" type="xsd:string">
  <xsd:complexType name="parameterValueValue">
   <xsd:sequence>
    <xsd:element name="nbElements" type="xsd:string">
    <xsd:element name="element" type="xsd:string"
            minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
  <xsd:complexType>
 </xsd:choice>
</xsd:element>


</xsd:schema>
```