

Projet de diplôme 2011-2012

-

Editeur de diagrammes de classes UML

Auteur: Fabrizio Beretta Piccoli (IL2012)

Sous la direction de : Pier Donini

Date : 27.07.2012 | v1.0

Cahier des charges

L' objectif de ce travail de diplôme veut être la suite d'un travail de diplôme précédent. Le but de ce dernier était de concevoir un logiciel permettant la création de diagrammes de classes de façon rapide et intuitive. Le logiciel est nommé « SLYUM » et a été réalisé par M. Miserez pendant le semestre d'été 2010/2011.

Un apprentissage des diagrammes de classes et de la grammaire du langage Java devra être accompli par l'étudiant pendant les premières semaines.

Ayant acquis ces notions une modélisation du projet sous la forme d'un diagramme de classe UML sera présenté au responsable. Une fois la phase d'analyse et de conception terminées, une deuxième phase d'implémentation aura lieu.
Les nouvelles fonctionnalités seront codées en Java.

Un journal de travail sera maintenu à jour tout au long du projet. La période de temps prise en compte va du 13.02.2012 (semaine 1) au 23.07.2012 (semaine 24). Les premières 18 semaines sont à moitié temps, 2 jours par semaine et les semaines 19-24 à plein temps, 5 jours par semaine. Un premier rapport sera rendu le 11.06.2012 (semaine 18/24), le rapport final sera rendu la semaine 24 (23.07.2012).

La défense du diplôme est prévu pour le début de septembre, période 03.09.2012 – 16.09.2012.

Les fonctionnalités suivantes seront rajoutées à l'application :

- Depuis un ou plusieurs fichier(s) de code source présent(s) sur la machine, les importer et générer le diagramme de classe correspondant dans Slyum.
- Création des fichiers de code à partir d'un projet existant.
- Maintenir une cohérence entre le code source et le diagramme de classe si l'un des deux est modifié.

Résumé

Le but de ce projet de Bachelor est de développer une application qui puisse aider les programmeurs à manier leur diagramme uml et les relatifs fichiers de code sources de manière simple et intuitive. Souvent les deux parties sont travaillées indépendamment l'un de l'autre, ce logiciel veut alors créer un lien étroit entre ces deux parties de manière que l'une, soit toujours le reflet de l'autre avec un minimum d'énergie et de temps.

L'application se base sur l'éditeur de diagrammes de classe UML « Slyum ».

Abstract

The aim of this Bachelor project is to build an application to help developers to manage their UML diagrams and correlated source code files in an easy and intuitive way. Often these two parts are taken independently, so this software wants to create a direct link between them. We want to consider one side as the reflection of the other one with a minimum of time and energy consuming.

This software is built on the UML diagrams editor « Slyum ».

1	Introduction	1
1.1	Structure du rapport	1
1.2	Technologies utilisées	1
2	Préparation du projet	2
2.1	L'état de l'art	2
2.1.1	BeautyJ	2
2.1.2	Antlr	2
2.1.3	Java.lang.reflect	3
2.2	UML	3
2.3	Java	3
2.4	Slyum	3
3	Analyse	4
3.1	L'importation	4
3.2	La synchronisation	5
3.3	La structure de données	5
3.4	L'exportation	6
3.4.1	L'exportation avec mémoire	7
3.5	La persistance	7
3.6	Le cycle de vie	7
4	Phase de réalisation	9
4.1	L'importation	9
4.2	Le Modèle	10
4.2.1	Statements	12
4.2.2	Déclaration	13
4.2.3	Variable	14
4.3	La Vue	16
4.4	La synchronization	18
4.5	L'affichage	19
4.5.1	Les associations	19
4.5.2	La généricité	20
4.5.3	La hiérarchie	20
4.5.4	L'emplacement	21
4.6	L'exportation	22
5	La documentation Doxygen	23
6	Le langage Java	25
6.1	L'interface	25
6.2	Le type « enum »	25
6.3	La Classe	26
6.4	La méthode	27
6.5	L'attribut / le paramètre	28
6.6	Les modificateurs	29
6.7	Le commentaire	29

7	Conclusion	30
8	Liste des références	31
9	Table des illustrations.....	32
10	Le journal de travail.....	33
11	Annexe A.....	38
12	Annexe B.....	39

1 Introduction

La rétroingénierie (traduction littérale de l'anglais « reverse engineering »), également appelée rétroconception, ingénierie inversée ou ingénierie inverse, est l'activité qui consiste à étudier un objet pour en déterminer le fonctionnement interne ou la méthode de fabrication. [Tiré de 1]

Cette activité, comme elle est rapportée entre autre par Wikipedia, lorsqu'elle est appliquée dans le domaine de la programmation informatique veut indiquer la possibilité de montrer un modèle qui représente ce que la machine peut faire lorsqu'un certain code lui est soumis.

Le modèle en question est pour la plupart du temps un modèle UML [2].

Cette possibilité permet de gagner un temps considérable dans la compréhension d'un programme informatique par rapport au code source. L'examen du code dans un modèle UML permet à l'utilisateur d'identifier les modules critiques contenus dans le code, ce qui établit un point de départ pour comprendre les besoins de l'entreprise ainsi que le système préexistant afin de permettre aux développeurs d'acquérir une meilleure compréhension globale du code source. Cela permet également de mettre à jour plus facilement un diagramme UML qui a été modifié dans la phase de développement.

Le but de ce rapport est d'expliquer comment la rétroingénierie et l'exportation de diagrammes de classe UML en fichiers de code source ont été possibles en association avec l'éditeur de diagrammes UML « Slyum ».

Les langages supportés sont : Java et C++(.h).

1.1 Structure du rapport

Le rapport se divise en trois sections : une première section est destinée à expliquer les préparatifs et l'analyse de ce projet de diplôme, une deuxième section est plus technique par l'explication de l'implémentation en Java, la dernière section sert de récapitulatif pour apporter quelques notions sur le langage Java.

1.2 Technologies utilisées

Langage de programmation	Java
Environnement de développement intégré	Eclipse Indigo
Java version	1.7.0_03
Développé sous	Windows 7 ita x64
Gestionnaire de version	Subversion
Lien internet	http://code.google.com/p/slyum/

2 Préparation du projet

Avant de se lancer dans la conception du programme, il faut prendre un peu de recul pour bien cerner le problème, faire des recherches sur l'état de l'art et maîtriser les sujets en question.

2.1 L'état de l'art

Dans l'industrie ce type de logiciel n'est pas inconnu. Il existe en effet d'autres éditeurs de diagramme capables de faire l'importation et l'exportation de code, comme par exemple Enterprise Architect, mais cependant il est payant et complexe à utiliser.

De même, pour la recherche d'un bon parseur Java plusieurs semblaient, dans une première phase, répondre aux besoins du travail, mais ils ont tous fini par se révéler inadéquats: trop compliqués à transformer, trop pointus dans une direction qui n'était pas celle recherchée ou sans documentation. La solution restante était celle d'en créer un ad hoc.

Les solutions que j'avais retenues sont les suivantes :

2.1.1 BeautyJ

BeautyJ permet de parser du code Java et de le transformer en XJava XML et vice-versa. Le problème qui se pose alors est comment parser le format XML pour en extraire les informations. Même en résolvant ce problème, BeautyJ est compatible avec des versions de Java qui ne vont pas au-delà de la version 1.4. La version actuelle de Java est la 1.7 ce qui rend le logiciel pratiquement inutilisable.

Voir aussi [11]

2.1.2 Antlr

Cet outil très puissant permet de produire un parseur à partir d'une grammaire. Une bonne documentation est fournie sur comment concevoir un compilateur, comment lire des instructions comme des boucles, des additions ou des multiplications. Très peu de documentation est fournie pour la partie concernant l'extraction des attributs d'une classe, de ses méthodes, des différents objets qui composent un programme. Ce manque de documentation rend son utilisation très difficile.

Voir aussi [12]

2.1.3 Java.lang.reflect

Cette bibliothèque permet de reconnaître chaque type d'objet avec toutes ses caractéristiques, malheureusement il ne prend en compte que les fichiers déjà compilés « .class ».

Voir aussi [13]

2.2 UML

En génie logiciel, UML (Unified Modeling Language, «langage de modélisation unifié») est un langage de modélisation et de spécification basé sur un paradigme orienté objet. Le noyau du langage a été défini en 1996 par Grady Booch, Jim Rumbaugh et Ivar Jacobson (connus comme les «trois amigos») sous les auspices de l'Object Management Group, un consortium qui gère encore la norme UML. Le langage a été créé avec l'intention d'unifier les approches précédentes (en raison des trois pères d'UML et d'autres), la collecte des meilleures pratiques dans l'industrie et donc la définition d'une norme de l'industrie unifiée. Une grande partie de la littérature utilise UML pour décrire le domaine des solutions d'analyse et de conception en sorte qu'il soit compréhensible pour un large public.

Bien que UML ait une part importante dans ce projet de diplôme, il a été longuement exposé dans le travail de diplôme de M. Miserez [7] et ne fera donc pas l'objet de ce rapport.

2.3 Java

Ce logiciel se veut être indépendant de tout langage de programmation mais, comme point de départ, Java a été choisi comme le langage à traiter. Pour cette raison une étude particulière de sa grammaire a été réalisée.

2.4 Slyum

Un autre point fondamental pour le bon déroulement du projet de diplôme était celui d'étudier le logiciel « Slyum », qui va héberger les nouvelles fonctionnalités de rétroingénierie.

Slyum est un éditeur de diagramme UML développé par M. Miserez qui permet de manière intuitive de manipuler des diagrammes de classe UML.

Ses fonctionnalités sont bien expliquées dans le manuel d'utilisation [6].

3 Analyse

Dans ce chapitre est exposée la structure qui permet de parser le code source, d'en extraire toutes les informations nécessaires (les classes, les attributs, les méthodes, etc.) et de les traduire en un diagramme qui sera affiché dans la fenêtre principale du logiciel. Le parcours inverse sera également exposé.

3.1 L'importation

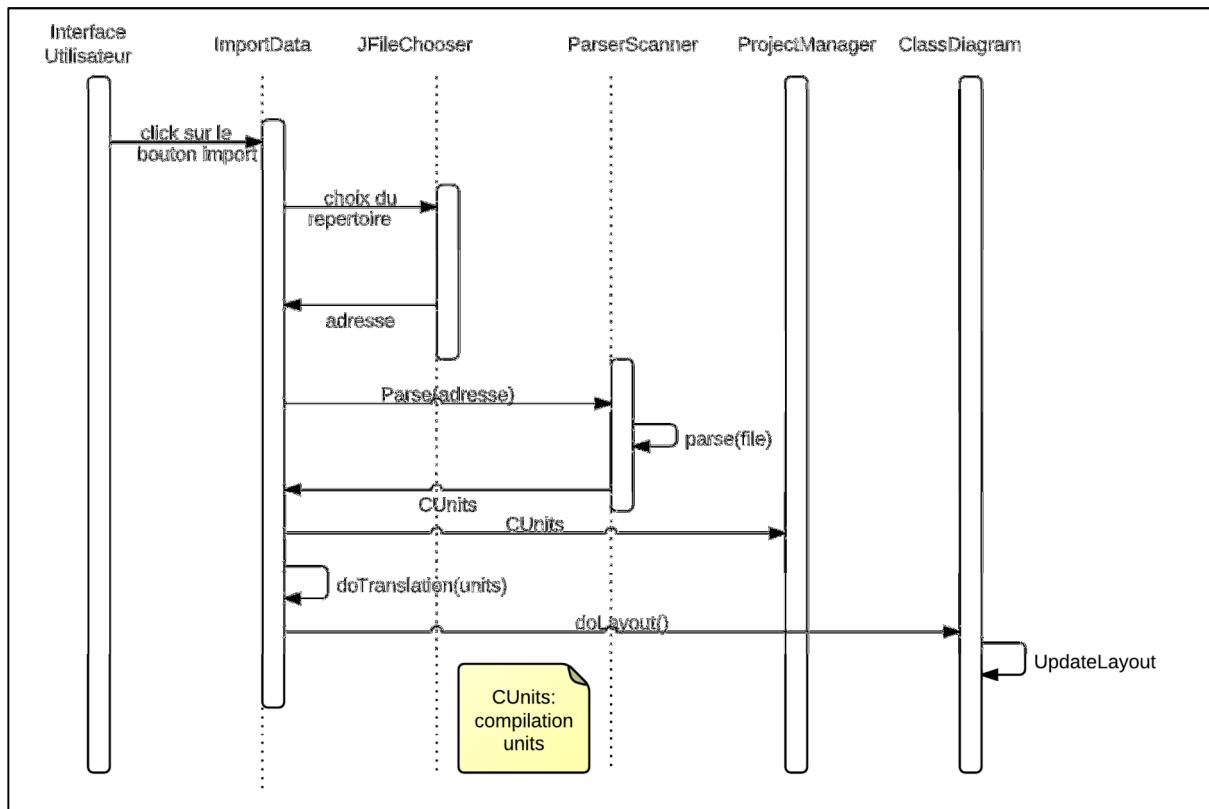


Figure 1: Depuis le fichier jusqu'au diagramme

Dans la figure 1 on peut voir le chemin accompli par l'importation du code source dans l'application. Une fois la procédure d'importation lancée, le parseur prend en charge les fichiers, les scanne et envoie les informations au *ProjectManager* ; cette phase accomplie, ces informations sont reprises, traduites et affichées dans le *ClassDiagram*. Une autre possibilité pour démarrer ce processus est celui du « Drag&Drop », c'est-à-dire de déplacer un dossier à l'aide de la souris depuis le système d'exploitation jusqu'à la fenêtre de l'application.

Les six parties visibles dans la figure 1 sont les suivantes :

- *ParserScanner* : prend en charge les fichiers de code source pour en extraire le contenu.
- *ProjectManager* : contient la structure de données.
- *ImportData* : l'activité qui est démarrée depuis l'interface utilisateur.

- *ClassDiagram* : correspond au modèle. Contient la structure des classes et leurs relations (héritages, associations, dépendances, etc.).
- *JFileChooser* : permet de choisir un dossier dans le système d'exploitation.
- *Interface Utilisateur* : partie visible et interactive de l'application.

3.2 La synchronisation

Une fois un projet importé dans l'application il est fort probable que ses fichiers originaires de code source soient modifiés, une synchronisation depuis l'interface utilisateur est alors possible. La synchronisation aura pour effet de parser à nouveau le fichier en question et d'apporter toutes les modifications nécessaires sur son alter ego version uml.

3.3 La structure de données

La structure de données nécessite plusieurs éléments pour supporter la complexité du code source. Les éléments qui vont être nécessaires sont présents dans la figure 2.

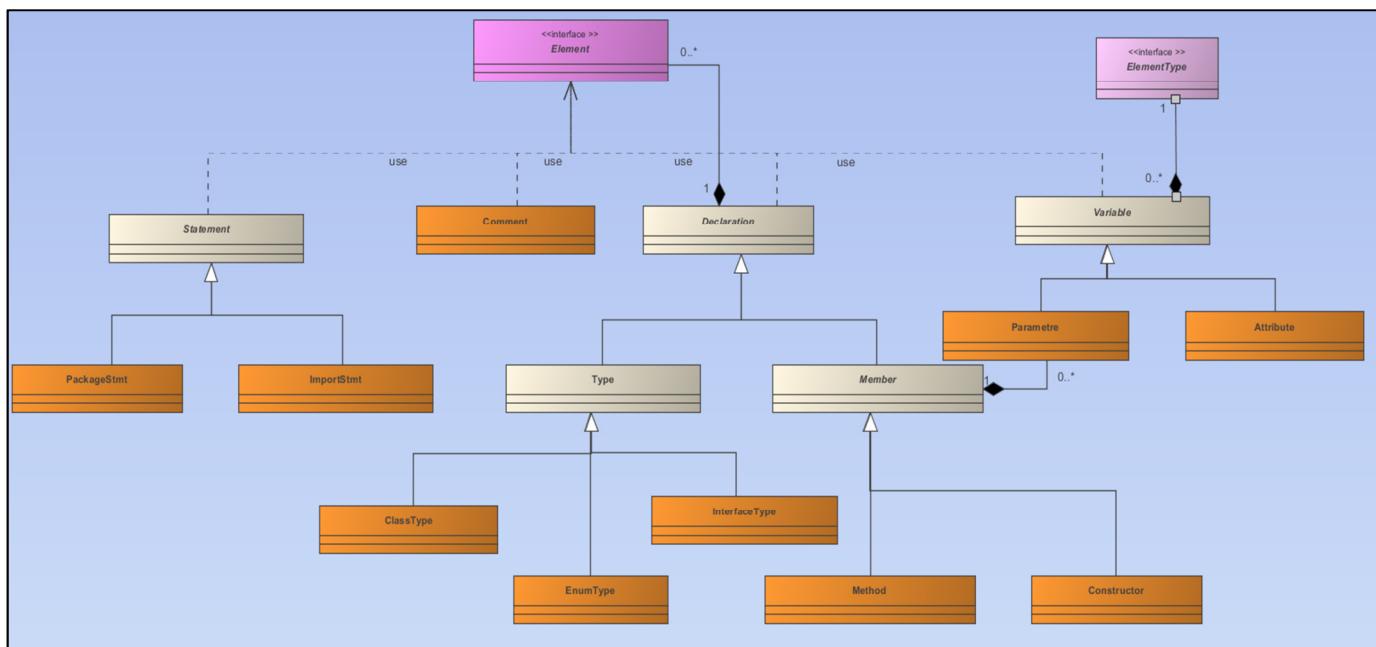


Figure 2: la structure de données

En parcourant tous les éléments (en orange) de gauche à droite on peut apercevoir :

- *PackageStmt* : le paquetage d'appartenance (chap. 4.2.1).
- *ImportStmt* : la classe/le paquetage de référence (chap. 4.2.1).
- *Comment* : un commentaire (chap. 4.2, 5.7)
- *ClassType* : une classe (chap. 4.2.2, 5.3)
- *InterfaceType* : une interface (chap. 4.2.2, 5.1)
- *EnumType* : un type énuméré (chap. 4.2.2, 5.2)
- *Constructor* : un constructeur (chap. 4.2.2, 5.4)

- *Method* : une méthode (chap. 4.2.2, 5.4)
- *Parametre* : un paramètre (chap. 4.2.3, 5.5)
- *Attribute* : un attribut (chap. 4.2.3, 5.5)

Un projet est habituellement composé de plusieurs fichiers contenant du code, appelés aussi unités de compilation. Chaque unité de compilation contient plusieurs éléments. En reprenant le même concept : le conteneur principal est le *ProjectManager*, qui contient plusieurs *CompilationUnit*, qui englobent plusieurs *Element*.

3.4 L'exportation

La Figure 3 montre le processus d'exportation d'un diagramme UML.

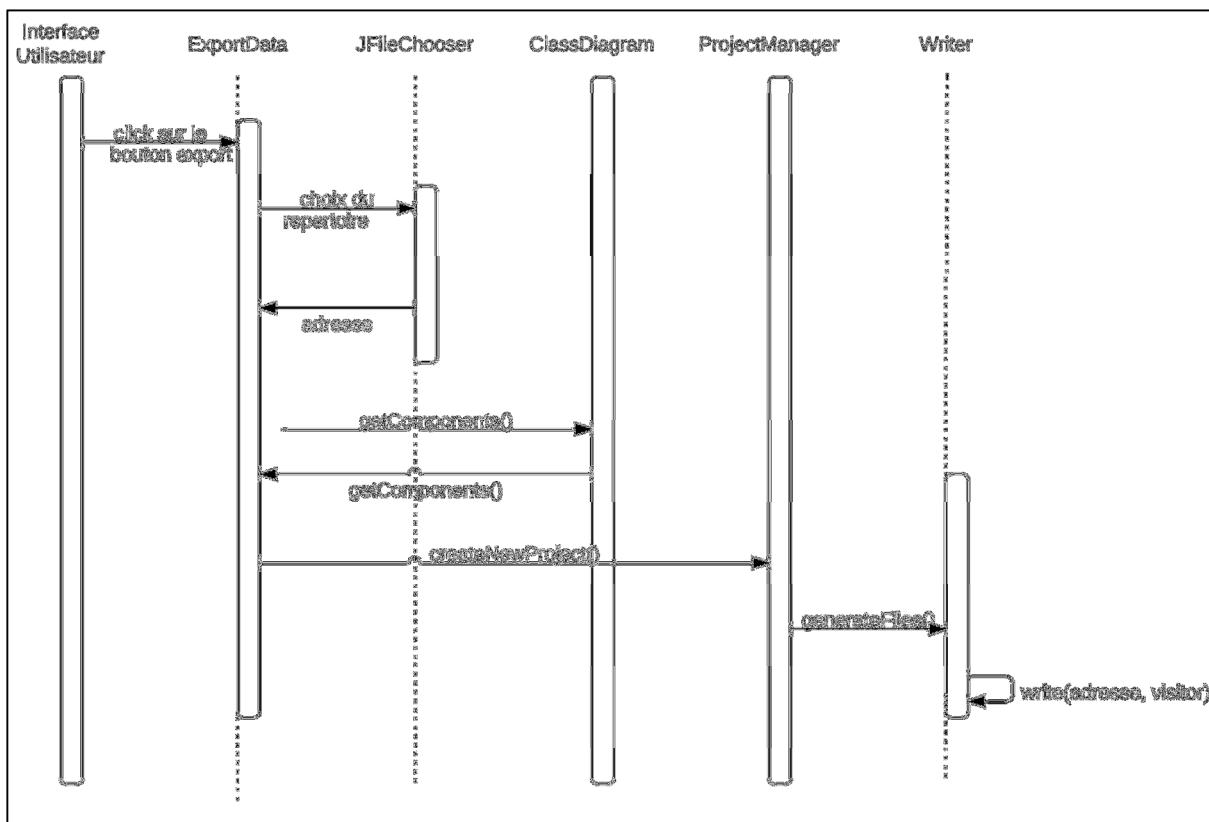


Figure 3 processus d'exportation

Dans le processus de la figure 3 on peut voir que certaines entités sont présentes tout au long du parcours et d'autres ont une durée de vie limitée. Les entités qui ont une durée de vie indéterminée sont *Interface Utilisateur*, *ClassDiagram* et *ProjectManager*. Cela n'est pas surprenant car le premier représente la partie visible et interactive de l'application tandis que les deux derniers gardent la structure de données. Pour ce qui concerne les autres entités, elles ont une mission à accomplir et une fois celle-ci terminée, elles doivent se terminer aussi. Cette mission peut être soit d'afficher une fenêtre qui montre le système de fichier pour trouver la bonne adresse de destination

(*JFileChooser*), soit de créer concrètement les fichiers et d'y écrire les données (*Writer*), ou encore de gérer le tout (*ExportData*).

3.4.1 L'exportation avec mémoire

L'exportation peut se faire de deux manières différentes : avec ou sans mémoire.

Si elle n'a pas de mémoire cela consiste simplement à reproduire le diagramme UML dans le langage souhaité. Si par contre elle a de la mémoire, alors le code généré sera complété par des informations extraites pendant l'importation (pas de mémoire possible sans une importation préalable). L'information peut porter sur des commentaires, des corps de méthodes, des bibliothèques d'importation, etc.

3.5 La persistance

Pour maintenir les données d'une session à l'autre il faut les enregistrer sur le disque dur. Cet enregistrement se fait au moyen d'un fichier XML qui permet de sauvegarder un diagramme UML et ses fichiers de code source. Pour les fichiers de code source seule leur adresse est écrite dans le XML.

3.6 Le cycle de vie

Le cycle de vie d'un projet graphiquement :

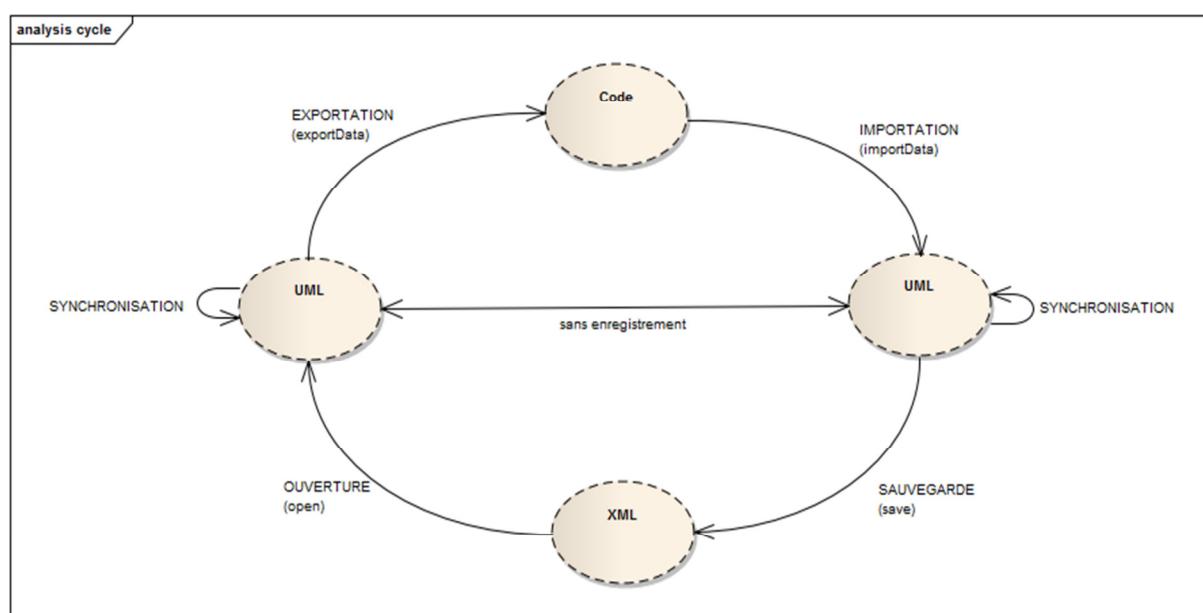


Figure 4 : le cycle de vie

L'état « XML » est un état de transition, il ne peut exister qu'en association avec l'état « UML ». Les autres états peuvent exister de façon indépendante.

Le cycle de vie montre bien qu'une transition entre les deux états, « CODE » et « UML », est faisable. On peut toujours revenir à l'état de départ depuis les deux autres états. Il n'existe pas d'état de « départ » ou de « fin » mais seulement une évolution entre les deux états.

Le point à retenir est que l'on puisse faire le lien entre les deux états « CODE » et « UML », le tout en gardant l'information à jour dès qu'un des deux états est modifié.

4 Phase de réalisation

Les pages de ce chapitre sont dédiées à représenter tous les éléments du chapitre 3, sous un angle d'implémentation. On rappelle que ce logiciel a été écrit entièrement en Java et donc les extraits de code feront référence à ce langage.

4.1 L'importation

L'importation commence par l'extraction des informations à l'aide d'un parseur, ces informations seront ensuite traduites dans la structure de données UML (*ClassDiagram*). La phase finale consiste à afficher le diagramme dans la fenêtre de l'application.

Le parseur est un algorithme qui permet d'analyser un texte et d'en déterminer sa structure syntaxique. Le texte est parcouru à l'aide de deux Scanners¹ : le premier parcourt toutes les lignes et le deuxième parcourt le contenu de chaque ligne. Il est ainsi possible de déterminer ce que la ligne représente, s'il s'agit par exemple de l'en-tête d'une classe, de l'en-tête d'une méthode ou d'un attribut.

Un seul passage n'est pas suffisant pour reconstruire le code parsé, il faudra faire un passage supplémentaire pour arriver à un résultat optimal. Le premier est nécessaire pour connaître l'ensemble des éléments. Le deuxième passage sert à reconstruire les relations d'héritage, d'implémentation est pour reconnaître le type des attributs / paramètres.

La reconnaissance d'une ligne est possible grâce à un signe distinctif (sauf si elle est vide ou si elle fait partie du corps d'une méthode). Ces signes peuvent être résumés dans le tableau 1.

Signe	Déduction
Mot clef « class »	Une classe
Mot clef « interface »	Une interface
Mot clef « enum »	Un type énuméré
Mot clef « import »	Une importation
Mot clef « package »	Le nom du package
« { »	Début du bloc
« (»	Une méthode
« // »	Un commentaire (1 ligne)
« /* »	Un commentaire (n lignes)
« @ »	Une annotation
sinon	Un attribut

Tableau 1 : les signes distinctifs.

¹ La classe Scanner est une API Java : java.util.Scanner

4.2 Le Modèle

Le modèle consiste dans une structure qui permet d'emmageriner les données reçues par le parseur.

Si on regarde la grammaire du langage Java on voit qu'une classe peut contenir plusieurs attributs, plusieurs méthodes, même plusieurs autres classes, et qu'un fichier peut également contenir plusieurs classes; il faut donc trouver une solution pour gérer ce niveau d'imbrication d'éléments les uns dans les autres.

Pour gérer cette structure très imbriquée, il a été nécessaire de se diriger vers le patron de conception « Composite » qui permet de gérer un objet simple ou un objet composé de plusieurs autres objets de la même manière.

Le patron de conception « Composite » s'associe très bien avec l'autre patron de conception « Visiteur ». Ce dernier permet de manipuler le même objet de différentes façons tout en gardant le principe d'ouverture/fermeture.

Le « Visiteur » devient utile, par exemple, dans le cas suivant : le concept de classe dans un langage orienté objet est le même mais son écriture est différente.

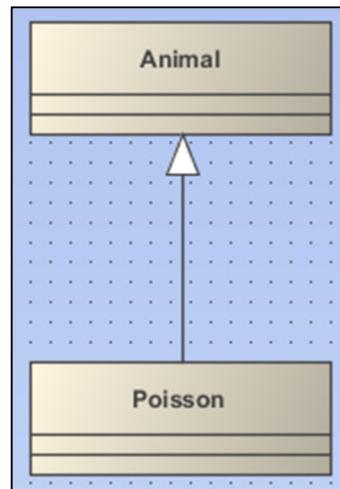


Figure 5: exemple héritage

Si on veut écrire l'en-tête de la classe *Poisson* en Java et C++ :

Java

```
public class Poisson extends Animal
```

C++

```
class Poisson : public Animal
```

Le point central du modèle est *ProjectManager* qui garde la totalité de la structure. C'est à travers lui que sont faites les requêtes d'exportation et d'importation. Chaque entité figurant dans le diagramme UML est représentée par un élément implémentant l'interface *Element* comme l'indique la figure 6.

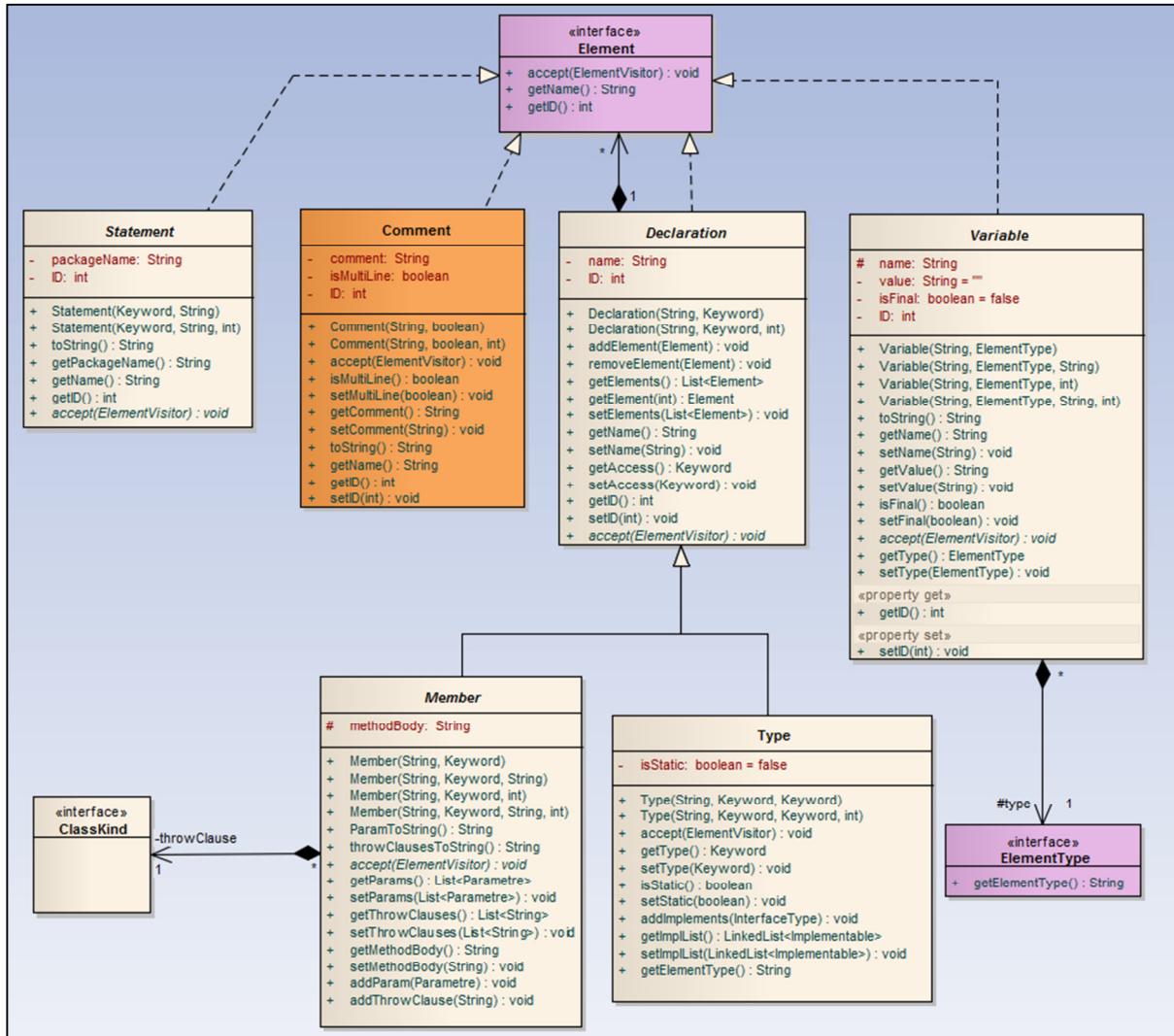


Figure 6: Eléments (branche)

Les éléments sont affichés en orange, tandis que les branches sont affichées en blanc (3 branches et 2 sous-branches). Le seul élément qui est rattaché directement à *Element* est *Comment* qui représente un commentaire; cela est dû à la nature très simple de *Comment* qui comporte seulement une chaîne de caractères et une valeur booléenne indiquant s'il se trouve sur plusieurs lignes ou non.

Chaque branche possède plusieurs feuilles :

4.2.1 Statements

Statement se divise en trois parties, le nom du paquetage (*PackageStmt*), les différentes références à d'autres classes ou paquetages (*ImportStmt*) et les instructions au préprocesseur.

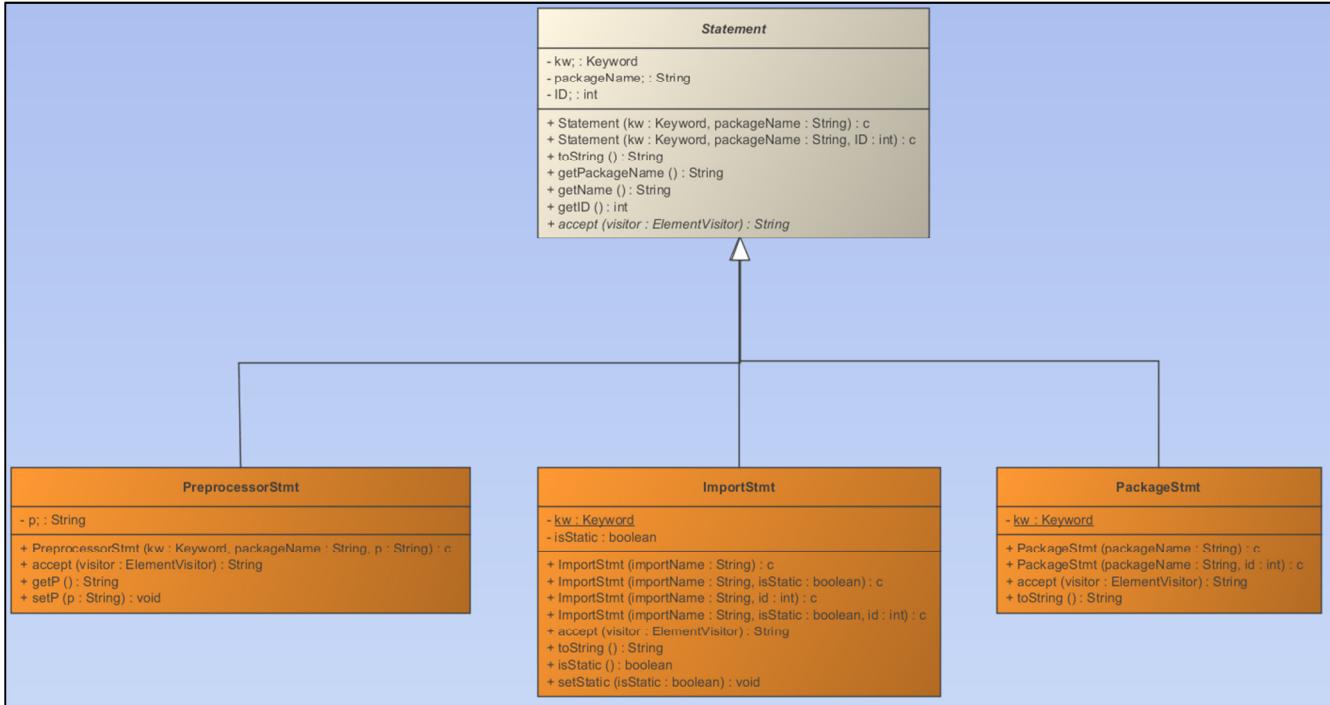


Figure 7: Statement

4.2.2 Déclaration

Une déclaration est la création d'un nouvel objet (sous-branche *Type*) ou encore la création d'une nouvelle fonction membre (sous-branche *Member*).

La sous-branche *Type* peut prendre 3 formes : soit d'une classe, soit d'une interface ou encore d'un type énuméré.

La sous-branche *Member* prend 2 formes : celle d'une méthode ou celle d'un constructeur.

L'association *Member-ClassKind* représente les exceptions qu'une méthode peut lever.

Se référer au chapitre 5 pour une explication détaillée des *Type* et des *Member*.

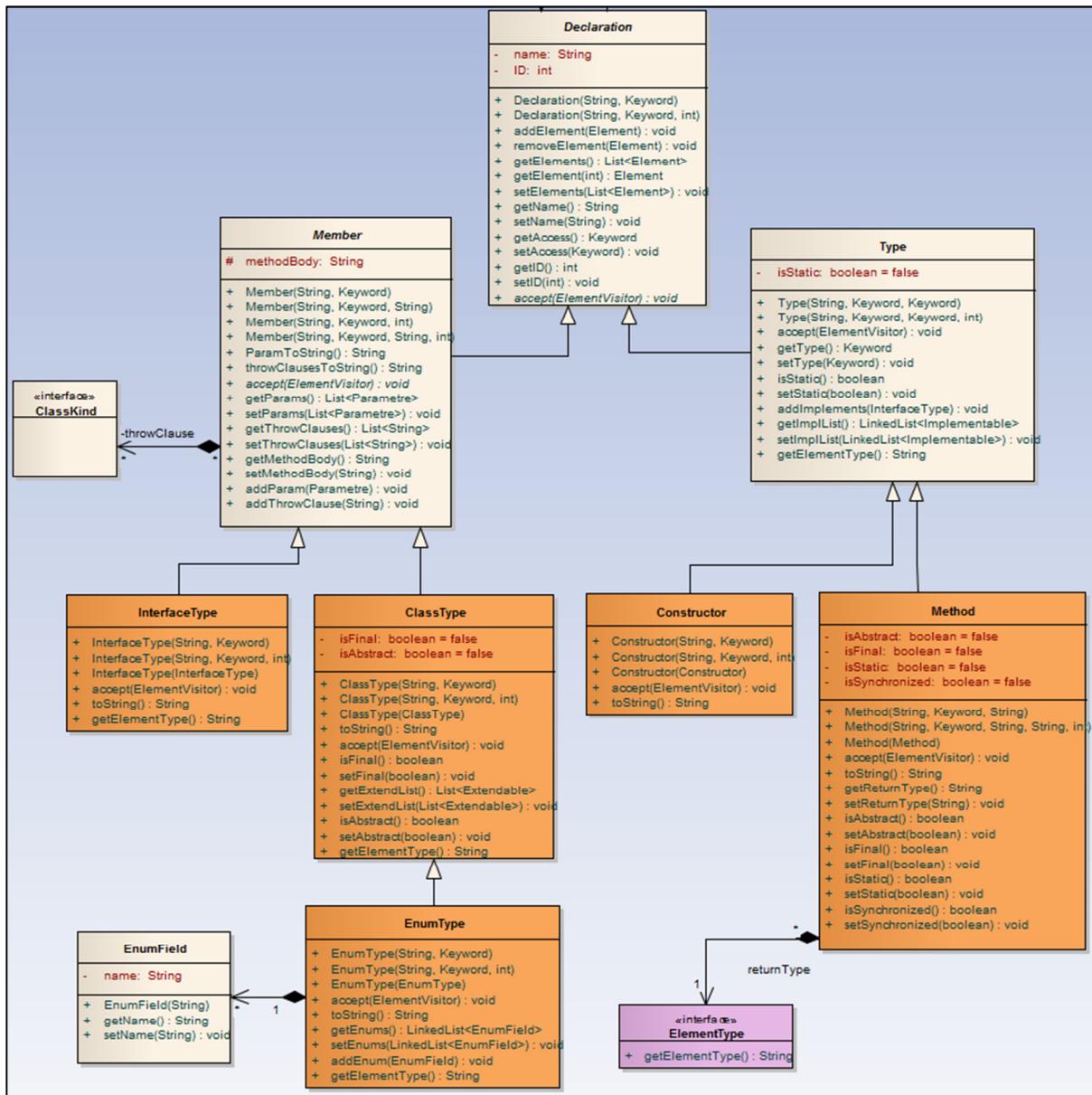


Figure 8: Declaration

4.2.3 Variable

La dernière branche, *Variable*, est celle qui représente une variable. Une variable peut être reconnue comme un paramètre dans une méthode, ou encore comme un attribut dans une classe.

La particularité de *Variable* est qu'il représente une nouvelle instance d'un type de données déjà existant. Pour le cerner, il faut prendre en compte les types primitifs (*byte*, *char*, *short*, *int*, *long*, *float*, *double* et *boolean*), les types créés par l'utilisateur et les types présents dans les librairies ou les bibliothèques (p.ex. : *java.util.String*). Tous ces types sont regroupés dans la catégorie *ElementType*, qui entre autre prend en charge les notions de tableau et de collection.

Dans la figure 9 on peut apercevoir deux interfaces, *ClassKind* et *InterfaceKind*, qui n'ont ni attribut ni méthodes, cela peut paraître inusuel mais le but de ces deux interfaces est de regrouper deux types d'objets sous un seul toit de manière à les traiter indistinctement. L'interface *ClassKind* représente toutes les classes depuis les quelles on peut hériter; l'héritage est donc possible depuis les classes créées par l'utilisateur et les classes présentes dans l'API Java. Les exceptions sont aussi représentées par *ClassKind* parce qu'elles héritent de *Exception* (et/ou *Throwable*). Le même principe s'applique à *InterfaceKind* qui représente toutes les interfaces.

Les classes *PointerType* et *ReferenceType* représentent les pointeurs et les valeurs par référence.

Se référer aussi au chapitre 5.5

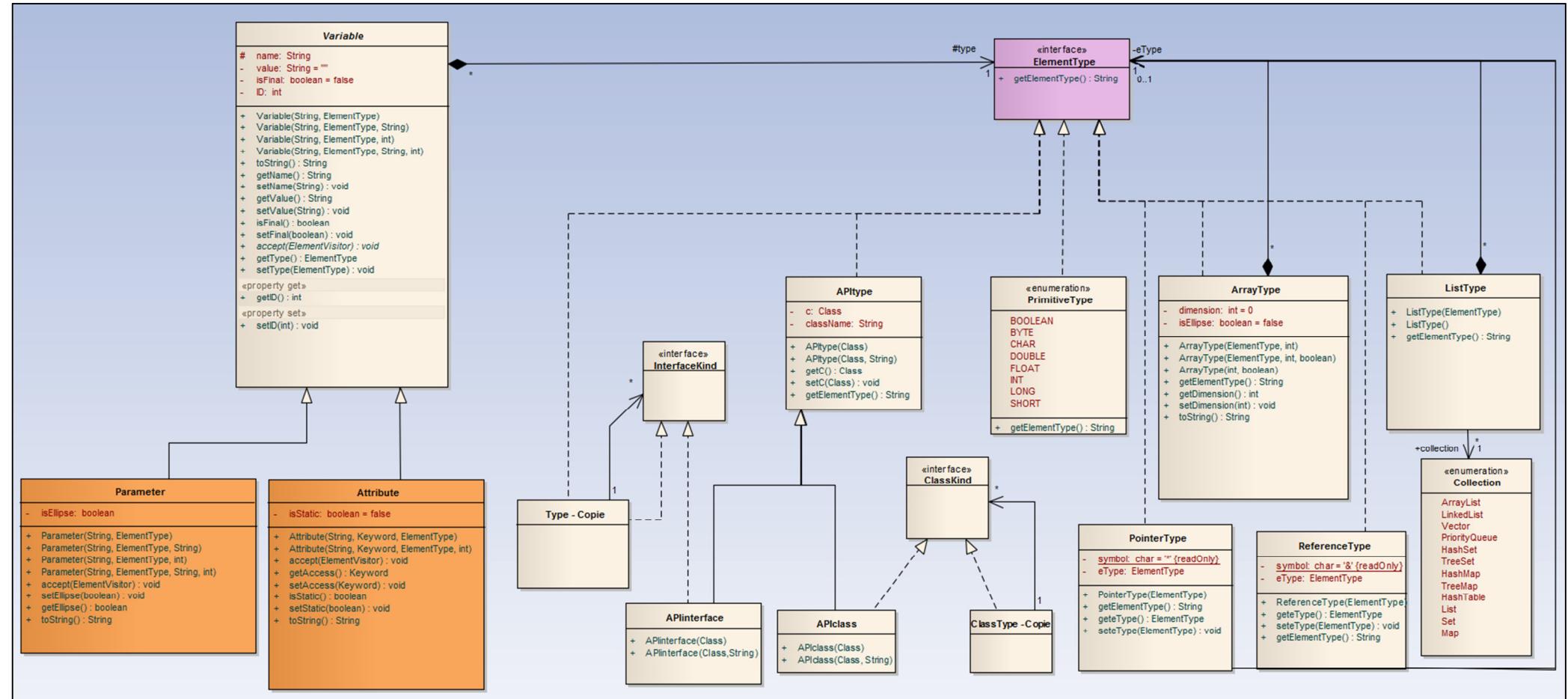


Figure 9: Variable et ElementType

4.3 La Vue

La partie visible de l'application n'a pas été beaucoup modifiée comparé à la version de base. Des boutons et des options supplémentaires ont été rajoutés dans la barre de menu, respectivement dans le menu, pour induire l'utilisateur à importer du code ou à exporter un diagramme.

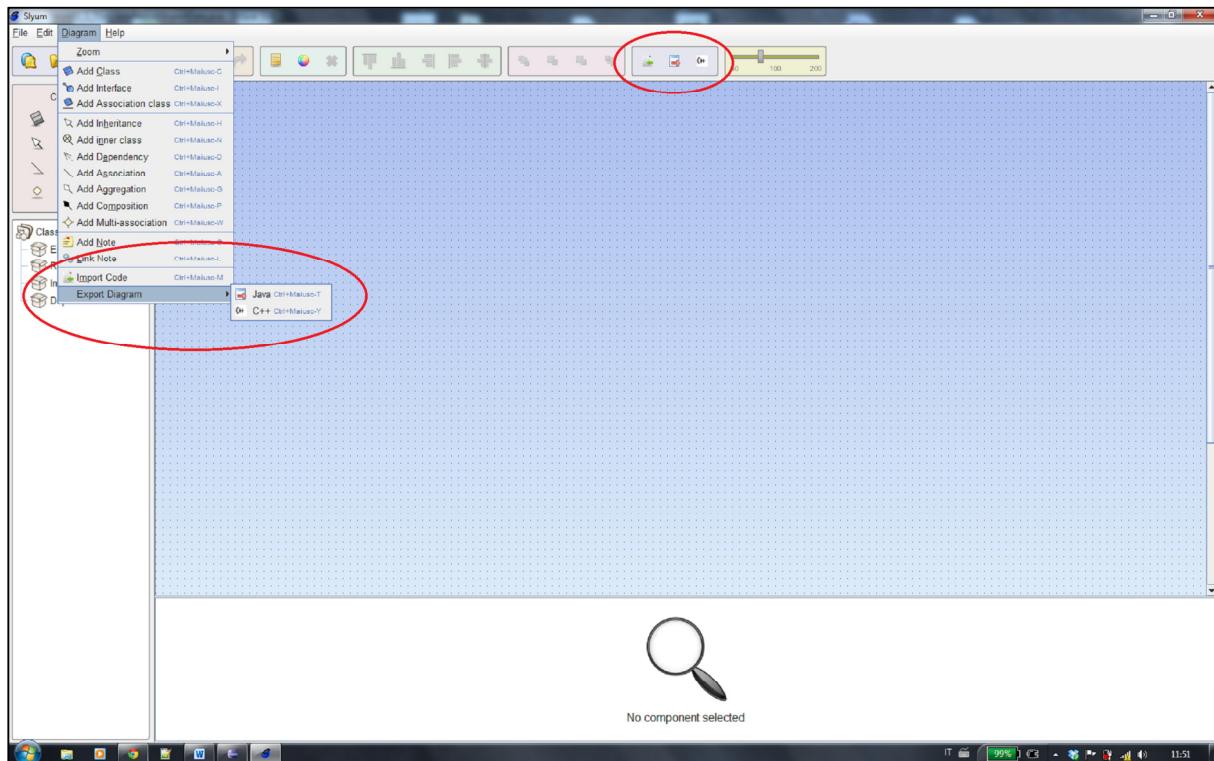


Figure 10: Slyum

Avec le « Drag&Drop »

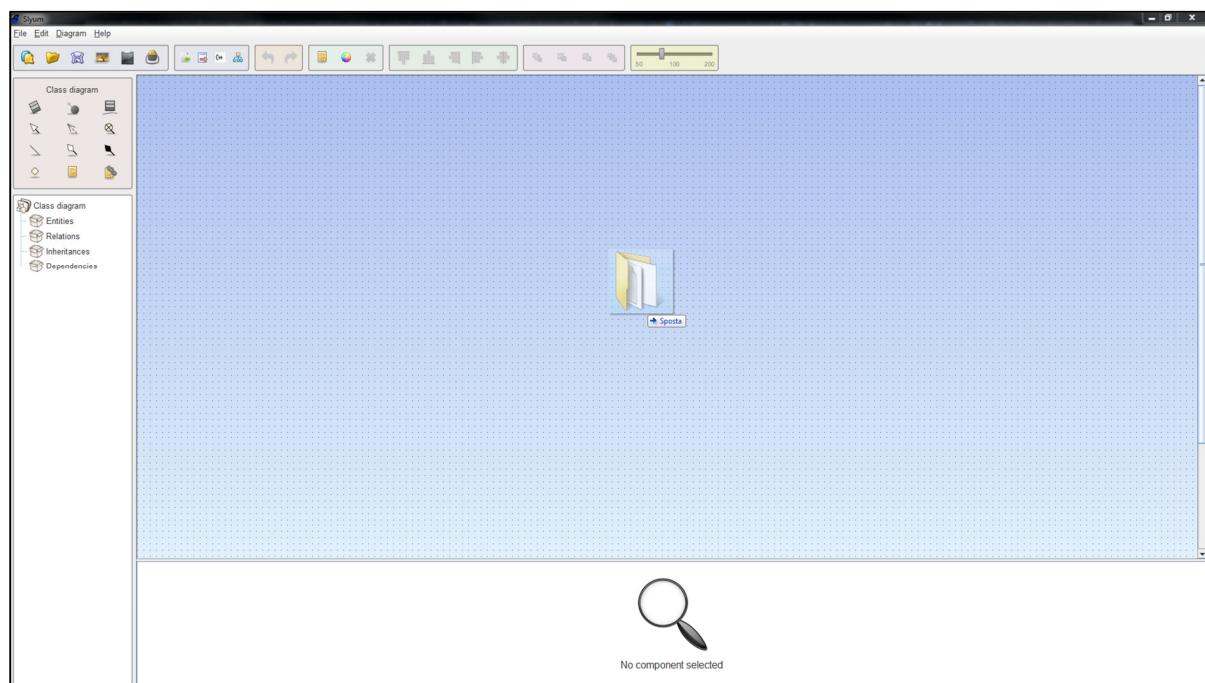


Figure 11 Drag&Drop

Une fois l'une de ces actions déclenchée, une fenêtre permet de choisir le bon répertoire. Un message apparaît lors de la bonne exécution d'une exportation indiquant le nom des nouveaux fichiers créés.

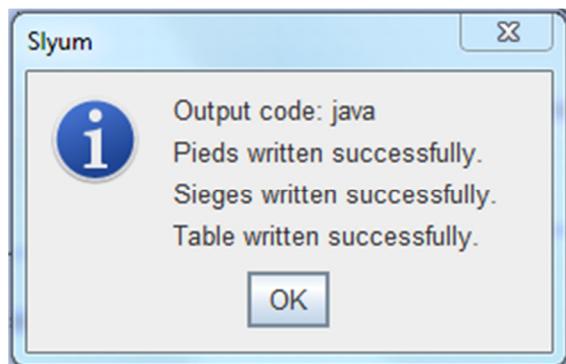


Figure 12 exportation réussie!

Lors de l'importation illégale d'un fichier (par ex.: un pdf), une fenêtre affiche l'erreur :

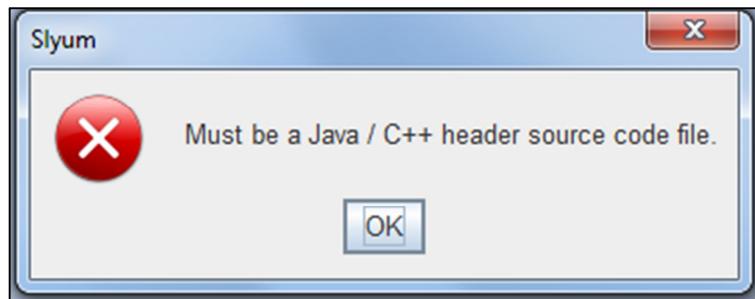


Figure 13 importation illégale

Une fenêtre affiche également une erreur lors d'un fichier qui n'est pas compilable :

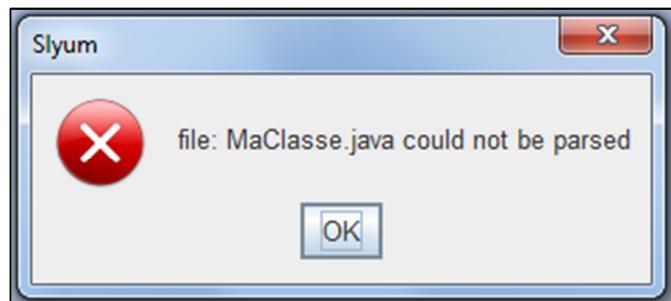


Figure 14 erreur de parsage

Pour rajouter la généricté dans les termes du langage Java il faut sélectionner (clic droit) une entité et choisir « add genericity » ou « remove genericity » dans le menu.

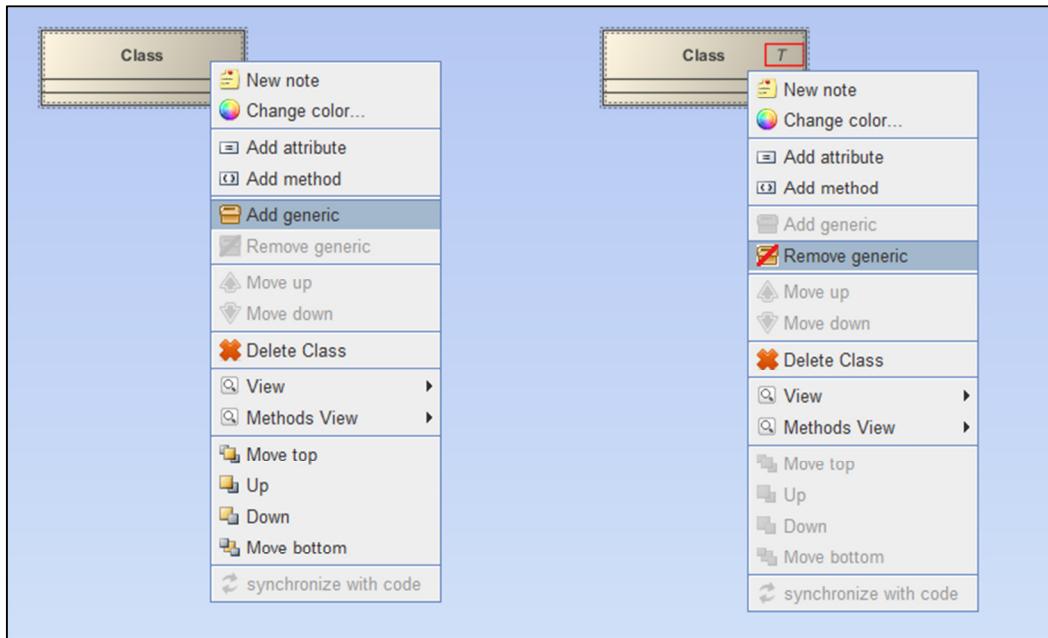


Figure 15 add/remove genericity

4.4 La synchronisation

La synchronisation est effectuée depuis le menu qui apparaît suite à la sélection d'un élément (clic droit sur élément). Si l'élément n'a pas d'alter ego « code source » la synchronisation n'est pas possible.

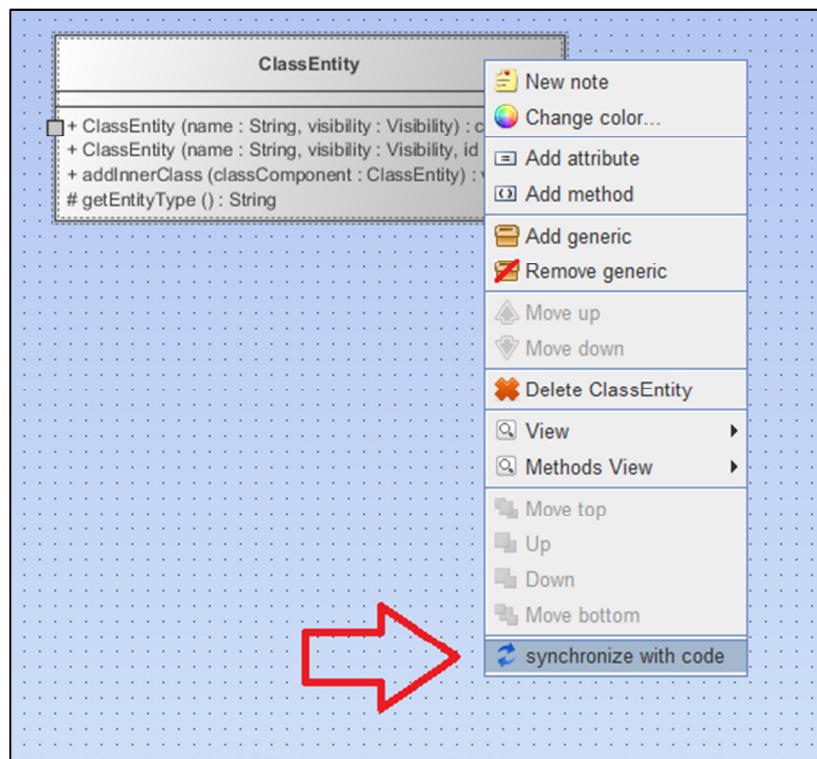


Figure 16 synchroniser

4.5 L'affichage

La compréhension d'un diagramme est beaucoup plus simple quand il est bien affiché.
Ce sous-chapitre veut présenter comment les différents éléments sont représentés dans l'application.

4.5.1 Les associations

Les associations sont représentées de la manière suivante :

Association 1-1 :

```
class Voiture
{
    private Personne proprietaire;
}
```

```
class Personne
{ }
```

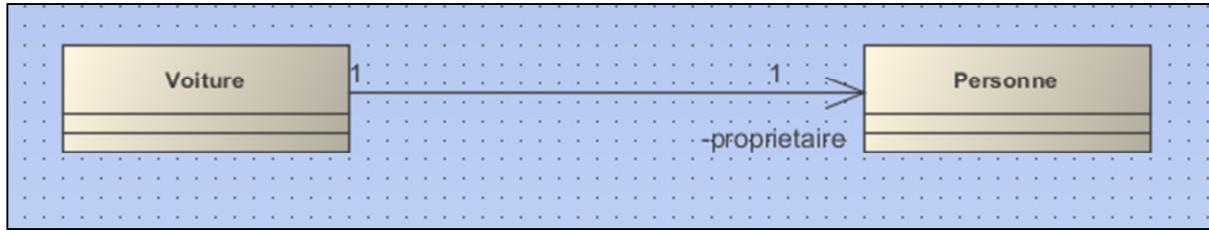


Figure 17 association 1-1

Association 1 – N :

```
class Voiture
{
    private Pneu[] pneus = new Pneu[4];
}
```

```
class Pneu
{ }
```

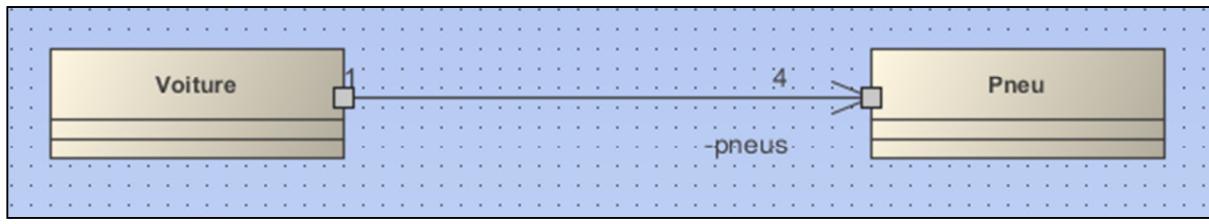


Figure 18 association 1-N

Association 1 – N..N :

```
class Voiture
{
    private LinkedList <Gadget> gadgets;
}
```

```
class Gadget
{ }
```

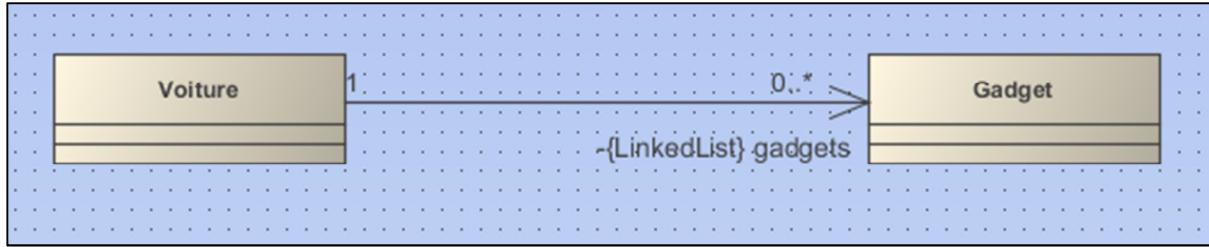


Figure 19 association 1-N..N

On peut remarquer que l'attribut a été enlevé de la classe et remplacé par le lien.

4.5.2 La généricité

Dans le langage Java la généricité peut être exprimée de la manière suivante :

```

public class ClassGenerique< T >
{
}
  ou
  ...
  public < T> void foo(){}
  ...
  ...
  
```

ClassGenerique sera représentée ainsi :

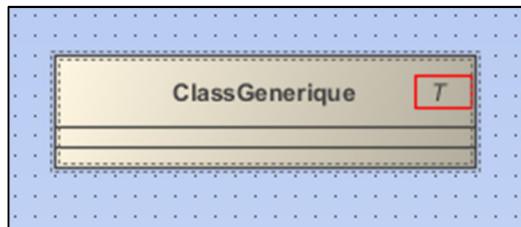


Figure 20 classGenerique

4.5.3 La hiérarchie

Les diagrammes UML permettent de représenter également l'héritage et l'implémentation d'interfaces. Si on prend comme père la superclasse ou l'interface et comme fils la classe qui hérite/implémente alors les fils sont toujours représentés à un niveau plus bas que les parents. Les liens qui relient les pères avec les fils ne sont pas sur une ligne droite mais sont divisés en trois parties et possèdent des angles de 90°, ce choix a été opté parce que cela améliore beaucoup la lisibilité du diagramme. On peut le voir dans la figure 19.

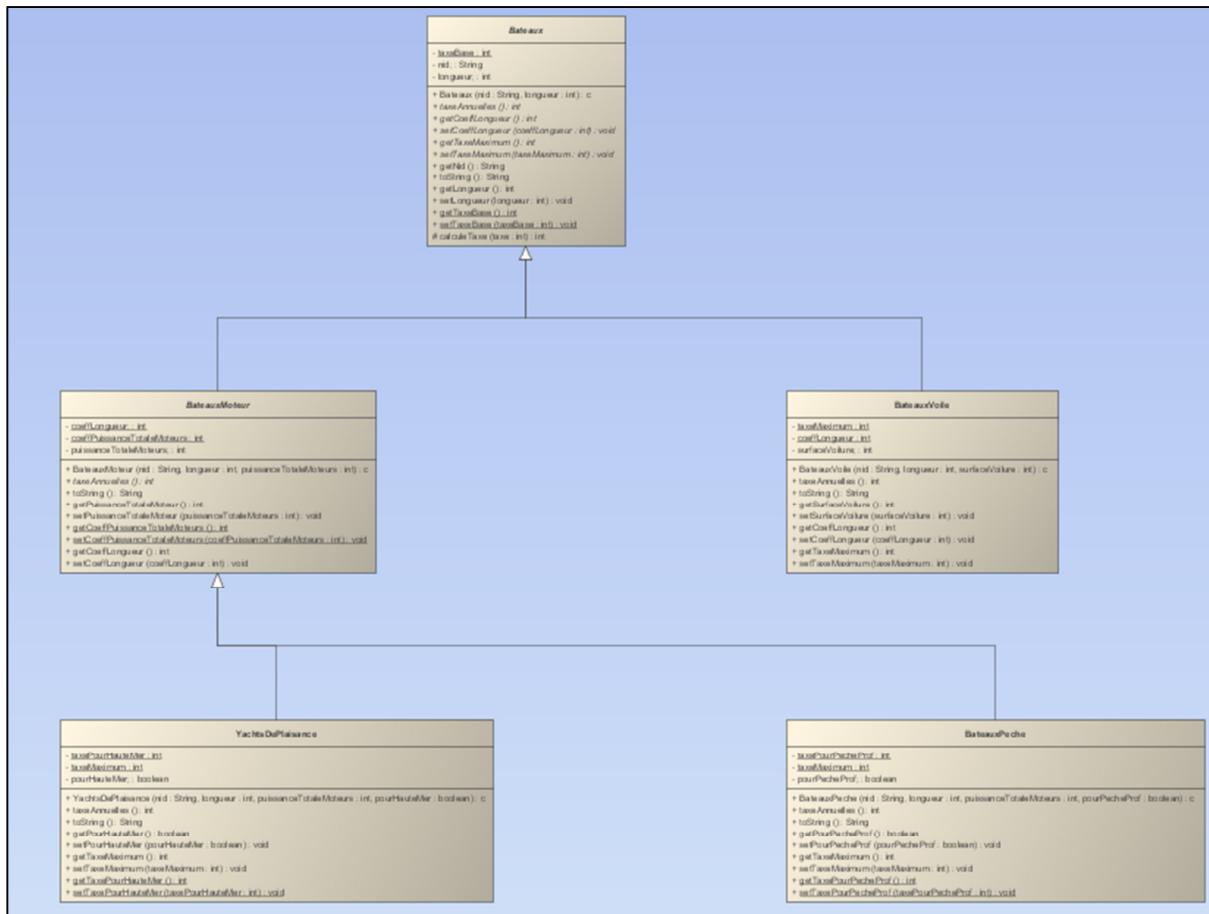


Figure 21 : hiérarchie entre classes

4.5.4 L'emplacement

Le choix d'emplacement des différentes entités est plutôt complexe parce qu'il faut gérer plusieurs variables : les relations de hiérarchie, les relations d'association et la taille de la fenêtre de l'application. Les relations créent plusieurs liens qui risquent de se croiser et il est parfois impossible d'éviter les croisements. Le nombre d'entités et leur taille nécessite de temps en temps une superposition des éléments pour avoir la totalité des entités sous l'œil.

L'algorithme qui permet l'affichage accompli les étapes suivantes : en premier lieu il établit les relations de hiérarchie, ensuite il ajoute les relations d'association en prenant soin de ne pas modifier les relations déjà existantes. Il essaye ensuite d'enlever un maximum de croisements entre les liens créés. Il rajoute alors tous les éléments qui n'ont pas de liens. Les éléments sont toujours affichés par rapport au milieu de la largeur de la fenêtre.

4.6 L'exportation

Après avoir vu l'importation au point 4.1, passons maintenant à l'exportation. Afin de rendre l'exportation souple pour plusieurs types de langage, chaque langage définit son propre visiteur² (*ElementVisitor*), lequel permet de définir avec précision comment le code sera généré et imprimé sur « papier » par le *Writer*. Il existe donc un unique *Writer* pour tous types de langage, la différence est dans le visiteur (*ElementVisitor*) qui lui est passé en paramètre.

L'écriture sur fichier se fait grâce à la classe *java.io.FileWriter*[9], qui permet avec *java.io.BufferedWriter*[10] d'écrire un flux de caractères dans un fichier en faisant appel à la méthode *write*. Il faut encore se rappeler de fermer le flux avec la méthode *close*.

```
FileWriter fstream = new FileWriter (file);
BufferedWriter out = new BufferedWriter (fstream);
    out.write (unit.visit(v));
    out.close();
```

Voici un exemple de génération de code Java après un 'exportation avec mémoire :

```
import bateaux.*;
/**
 * Classe de test pour le laboratoire bateaux.
 *
 * @author Fabrizio Beretta Piccoli
 * @version 1.0 | 13.11.2009
 */
public class Test
{
    public static void main(String[] args)
    {
        StationBalneaire newBeach = new StationBalneaire(10);
        newBeach.ajouterBateau(new YachtsDePlaisance("Brigitte", 20, 25, true));
        newBeach.ajouterBateau(new YachtsDePlaisance("Coti", 200, 75, false));
        newBeach.ajouterBateau(new BateauxPeche("Pechi", 10, 20, false));
        newBeach.ajouterBateau(new BateauxPeche("Proets", 5, 10, true));
        newBeach.ajouterBateau(new BateauxVoile("Voili", 10, 20));
        newBeach.ajouterBateau(new BateauxVoile("Fabri", 5, 16));

        System.out.println("----- Affichage des bateaux -----");
        newBeach.afficherTousLesBateaux();

        // Suppression de bateaux.
        newBeach.supprimerBateau("Brigitte");
        newBeach.supprimerBateau("Fabri");
        newBeach.supprimerBateau("Voili");
        newBeach.supprimerBateau("Pechi");
        newBeach.supprimerBateau("Coti");

        System.out.println("----- Affichage du bateau restant -----");
        newBeach.afficherTousLesBateaux();
    }
}
```

Figure 22 exemple exportation

Remarque : la couleur du texte est due au formatage de Notepad++.

² Le visiteur fait référence au patron de conception "Visiteur"

5 La documentation Doxygen

En annexe à ce document on peut trouver une documentation réalisée avec Doxygen [14].

Cette documentation en ligne (HTML) permet, à travers un navigateur, de parcourir les fichiers de code source de cette application.

Cette documentation est très similaire à une documentation javadoc avec un ajout de diagrammes.

Cette documentation est disponible aussi sous : <http://code.google.com/p/slyum/>

La page d'index :



Figure 23 doxygen - index

La page relative à ProjectManager :

The screenshot shows the Slyum 2.0 documentation interface for the `dataRecord.ProjectManager` class. The title bar reads "Slyum: dataRecord.Project". The address bar shows the URL: `file:///C:/Users/Fabrizio/Desktop/doxygen%20doc/html/classdata_record_1_1_project_manager.html`. The main content area is titled "dataRecord.ProjectManager Class Reference". It includes a "Collaboration diagram for dataRecord.ProjectManager:" which shows a `LinkedList< CompilationUnit >` named `i_filesRecord` associated with the `dataRecord.ProjectManager` class via an `instance` dependency. Below the diagram is a "List of all members." section. Under "Public Member Functions", there are 14 listed methods, each with a brief description. Under "Static Public Member Functions", there are two static methods. Under "Private Member Functions", there is one private constructor. Under "Private Attributes", there is one attribute listed. The left sidebar shows a tree view of the project structure, with `dataRecord` expanded to show `ProjectManager`.

Figure 24 doxygen - ProjectManager

6 Le langage Java

Cette section ne va pas donner une description exhaustive du langage Java, elle va plutôt détailler quelques aspects du langage fondamental pour la compréhension du projet.

6.1 L'interface

Une interface est un objet abstrait qui ne peut contenir que des constantes et des méthodes abstraites. Toutes ces méthodes (publiques par default) doivent être redéfinies dans la classe qui implémente la dite interface.

La syntaxe :

```
[visibility] interface InterfaceName [extends other interfaces]
{
    constant declarations
    abstract public method declarations
    inner declarations
}
```

Des exemples :

```
public interface MonInterface
{
    int i = 0 ;
    public void foo();
}
```

```
public interface MonInterface2 extends MonInterface
{
    int i = 0 ;
    public void foo();

    class Inner {}
}
```

Une interface peut implémenter une autre interface avec le mot-réservé « extends »; le mot-réservé « implements » lui est par contre interdit.

Elle peut aussi contenir des classes internes qui seront déclarées statiques, même en absence du mot- réservé « static » des types énumérés et des autres interfaces.

Les attributs n'ont pas besoin d'être déclarés « final static », ils le sont par default, tout comme les méthodes qui sont abstraites par default.

6.2 Le type « enum »

Un type énuméré est un type de données qui consiste en un ensemble de constantes appelées énumérateurs. Lorsque l'on crée un type énuméré on définit ainsi une énumération. Lorsqu'un identificateur tel qu'une variable est déclarée comme étant de

type énuméré, cette variable peut recevoir n'importe quel énumérateur (lié à ce type énuméré) comme valeur [tiré de 3].

La syntaxe :

```
[visibility] enum EnumName [implements other interfaces]
{
    constant declarations
    field declarations
    method declarations
    inner declarations
}
```

Des exemples :

```
public enum MonEnum
{
    HIVER, ETE, PRIMETEMPS, AUTONNE;
}
```

```
public enum MonEnum implements MonInterface
{
    HIVER, ETE, PRIMETEMPS, AUTONNE;

    public void foo(){}
    class A{}
}
```

Un type énuméré est très similaire à une classe. Ce qui différencie un type énuméré d'une classe sont sa liste de constantes, le mot réservé « enum » à la place de « class » et un constructeur qui ne peut être que privé.

6.3 La Classe

Une classe représente le concept d'objet dans un langage orienté objets comme Java.

La syntaxe:

```
[visibility] class ClassName [implements other interfaces]
            [extends other classe]
{
    field declarations
    method declarations
    inner declarations
}
```

Des exemples :

```
public abstract class MaClasse
{
    private int unAttribut ;
    public MaClasse(){}
    public abstract void foo();
    protected int foo2(){}
}
```

```
public class MaClasse2 extends MaClasse implements MonInterface
{
    enum unEnumEnInterne{}
    class Inner{}

    public void foo(){}
}
```

Une classe peut offrir des attributs et des fonctions pour garder et traiter l'information.
Elle peut hériter d'une autre classe comme implémenter plusieurs autres interfaces.
La création de classe/enum/interface internes est permise.

6.4 La méthode

Une méthode permet d'effectuer des traitements sur (ou avec) les données membres des objets.

La syntaxe :

```
[Modificateurs] TypeDeRetour nomDeLaMethode [{Parametres}][throw {exceptions}]
{
    liste d'instructions
    [return]
}
```

Des exemples :

```
public int foo()
{
    int valeur = 2 ;
    return valeur ;
}
```

```
MaClasse(int value) throws Exception
{
    this.value = value;
}
```

Un constructeur est une méthode particulière qui ne possède pas de type de retour et dont le nom est celui de la classe qui le définit.

Une méthode peut (re)lancer des exceptions en ajoutant « throws » plus le nom de l'exception à la fin de son entête.

6.5 L'attribut / le paramètre

L'information inhérente d'un objet peut être retenue grâce aux attributs membres d'une classe. Si un attribut est passé en paramètre dans une méthode on parle alors de paramètre; celui-ci ne peut pas avoir de modificateur de visibilité et ne peut pas être initialisé.

La syntaxe :

```
[Modificateurs] Type nomDeL'attribut [= initialisation]3 ;
```

Des exemples :

```
public final int i;
protected int tab[] = {1,2,3};
```

³ Exceptées les paramètres.

6.6 Les modificateurs

Il reste encore à voir les modificateurs des différents types cités ci-dessus :

		Modifiers-Elements Matrix in Java							
element	modifier	Data field	Method	Constructor	Class		Interface		
					top level (outer)	nested (inner)	top level (outer)	nested (inner)	
abstract	no	yes	no		yes	yes	yes	yes	
final	yes	yes	no		yes	yes	no	no	
native	no	yes	no		no	no	no	no	
private	yes	yes	yes		no	yes	no	yes	
protected	yes	yes	yes		no	yes	no	yes	
public	yes	yes	yes		yes	yes	yes	yes	
static	yes	yes	no		no	yes	no	yes	
synchronized	no	yes	no		no	no	no	no	
transient	yes	no	no		no	no	no	no	
volatile	yes	no	no		no	no	no	no	
strictfp	no	yes	no		yes	yes	yes	yes	

Figure 25 : les modificateurs[2]

Dans la figure 4 on peut voir quel modificateur peut être appliqué à quel élément.

Quelques règles supplémentaires [4] :

- Chaque élément peut contenir au maximum un modificateur entre : « public », « protected » et « private ».
- Une classe ne peut pas être déclarée « abstract » et « final » en même temps.
- Une méthode abstraite n'a pas de corps.
- Une classe qui contient une méthode abstraite est une classe abstraite.

6.7 Le commentaire

Les commentaires sont possibles à tout moment dans le code et peuvent être d'une seule ligne ou de plusieurs lignes :

Un exemple :

```
// un commentaire d'une ligne
```

```
/** un commentaire
    sur 2 lignes */
```

7 Conclusion

Le moment est venu de faire le bilan du travail accompli. Dans l'introduction il était mentionné que ce projet prenait comme base l'éditeur de diagrammes de classe UML « Slyum ». Voyons une synthèse des améliorations qui ont été apportées à cette application : la possibilité de faire des aller - retour entre les diagrammes de classe et les respectifs fichiers de code source Java. Cela implique la synchronisation des éléments du diagramme avec le code et l'exportation du diagramme en incluant les informations apportées par le programmeur au code mais qui ne sont pas présentées sur le diagramme (exportation avec mémoire). Un algorithme gère le placement des éléments suite à l'importation pour faciliter la lecture du diagramme. Dans l'enregistrement du diagramme dans Slyum, une trace des emplacements des fichiers (si existants) est gardée de manière à les repérer lors de sa prochaine ouverture. Même si une importation / exportation est possible pour le C++, la prise en charge de ce langage est d'une qualité moyenne et mérirerait une reprise.

Fabrizio Beretta Piccoli

8 Liste des références

- [1] La rétroingénierie
<http://fr.wikipedia.org/wiki/R%C3%A9troing%C3%A9nierie>
- [2] UML - http://fr.wikipedia.org/wiki/Unified_Modeling_Language
- [3] Type énumérée
http://fr.wikipedia.org/wiki>Type_%C3%A9num%C3%A9r%C3%A9
- [4] Java modifiers - <http://bmanolov.free.fr/javamodifiers.php>
- [6] Manuel d'utilisation Slyum
- [7] Slyum rapport
- [8] Liste chaînée - http://fr.wikipedia.org/wiki/Liste_cha%C3%AEn%C3%A9e
- [9] Java API FileWriter
<http://docs.oracle.com/javase/1.4.2/docs/api/java/io/FileWriter.html>
- [10] Java API BufferedWriter
<http://docs.oracle.com/javase/1.5.0/docs/api/java/io/BufferedWriter.html>
- [11] BeautiJ - <http://beautyj.berlios.de/>
- [12] Antlr - <http://www.antlr.org/>
- [13] Java API reflect
http://docs.oracle.com/javase/6/docs/api/java/lang/reflect/package-summary.html#package_description
- [14] Doxygen - <http://www.stack.nl/~dimitri/doxygen/>

P.S.: dernière visite des sites : 19.07.12

9 Table des illustrations

<i>Figure 1: Depuis le fichier jusqu'au diagramme.....</i>	4
<i>Figure 2: la structure de données.....</i>	5
<i>Figure 3 processus d'exportation</i>	6
<i>Figure 4 : le cycle de vie.....</i>	7
<i>Figure 5: exemple heritage</i>	10
<i>Figure 6: Eléments (branche)</i>	11
<i>Figure 7: Statement</i>	12
<i>Figure 8: Declaration</i>	13
<i>Figure 9:Variable et ElementType</i>	15
<i>Figure 10: Slyum</i>	16
<i>Figure 11Drag&Drop</i>	16
<i>Figure 12 exportation réussie!.....</i>	17
<i>Figure 13 importation illégale</i>	17
<i>Figure 14 erreur de parсage.....</i>	17
<i>Figure 15 add/remove genericity</i>	18
<i>Figure 16 synchroniser</i>	18
<i>Figure 17 association 1-1</i>	19
<i>Figure 18 association 1-N.....</i>	19
<i>Figure 19 association 1-N..N</i>	20
<i>Figure 20 classGenerique.....</i>	20
<i>Figure 21 : hiérarchie entre classes.....</i>	21
<i>Figure 22 exemple exportation</i>	22
<i>Figure 23 doxygen - index</i>	23
<i>Figure 24 doxygen - ProjectManager</i>	24
<i>Figure 25 : les modificateurs[2]</i>	29

10 Le journal de travail

Semaine 13.02 - 19.02.12:

- Création du planning pour le prochain semestre.
- Découverte de l'éditeur de diagramme de classes "Slyum".

Semaine 20.02 - 26.02.12:

- Installation de TurtoiseHG, création d'un clone du projet.
- Lecture de la documentation inhérente à TurtoiseHG.
- Installation du JDK 1.7.
- Un nouveau projet a été ouvert dans Eclipse à partir du clone, le projet est compilable et fonctionnel.
- Découverte de l'éditeur de diagramme de classes "Slyum".

Semaine 27.02 - 04.03.12:

- Semaine de relâches.

Semaine 05.03 - 11.03.12 :

- Rédaction du cahier des charges.
- Une recherche sur l'état de l'art des parseurs et des logiciels qui font des opérations similaires à Slyum est entreprise.
- Création d'une nouvelle branche sur Google code pour la sauvegarde du projet et la possibilité de gérer l'évolution du code.
- Une première version du diagramme UML est établie.

Semaine 12.03 - 18.03.12 :

- Correction du diagramme UML.
- Poursuite de la recherche sur l'état de l'art.

Semaine 19.03 - 25.03.12 :

- Une étude sur la grammaire Java est menée pour bien cerner tous les cas à prévoir lors du parсage et la construction de la structure de données.
- Amélioration du diagramme UML.

 **Semaine 26.03 - 01.04.12 :**

- La structure des éléments a subi un changement dans la bonne direction.
Elle est maintenant prête à être implémentée.
- Analyse sur comment générer l'importation et l'exportation de fichiers de code source.
- L'importation et l'exportation doivent être assez souples pour accepter plusieurs types de codage.

 **Semaine 02.04 - 08.04.12 :**

- Un regard approfondit est mené sur le code du logiciel pour savoir comment faire le lien entre le code existant et le code qui va être rajouté par la suite.
- Amélioration du diagramme UML.

 **Semaine 09.04 - 15.04.12 :**

- Semaine de relâches.

 **Semaine 16.04 - 22.04.12 :**

- Début de l'implémentation: création des éléments principaux de l'application, création de l'interface *Element* laquelle sera implémentée par tous les éléments.
- Les classes Attribute.java, ClassType.java, Comment.java, Constructor.java, Declaration.java, Element.java, EnumType.java, importStmt.java, InterfaceType.java, Member.java, Method.java, PackageStmt.java, Parameter.java, PrimitiveType.java, Statement.java, Type.java, Field.java sont rajoutées au projet.
- Création aussi des classes ProjectManager.java et CompilationUnit.java, la première sert de point de départ pour toute la structure de données et la deuxième modélise un fichier de code source.

 **Semaine 23.04 - 29.04.12 :**

- Dans chaque classe a été implémentée la méthode `toString()` pour effectuer un premier rendu de la structure, cette méthode sera maintenue mais seulement pour effectuer du debugging.
- Un writer est implémenté pour la génération de code source Java. Ce writer est pour le moment spécifique au langage Java.

 **Semaine 30.04 - 06.05.12 :**

- Création du parseur (ParserScanner.java). Cette première version du parseur est capable d'extraire les entêtes des classes, des interfaces et des méthodes, il peut extraire aussi les différents attributs d'une classe/interface.

 **Semaine 07.05 - 13.05.12 :**

- Changement dans *Field* : un attribut possède un type, ce type n'est plus représenté par un String mais pas une sous-structure qui arrive à déterminer son appartenance (type primitif, API, ou un objet créé par l'utilisateur) et s'il s'agit d'un objet simple ou d'un tableau ou encore d'un ensemble d'élément (par exemple une liste).

 **Semaine 14.05 - 20.05.12 :**

- La connexion entre ProjectManager et ClassDiagram est établie, c'est-à-dire entre les deux structures de données. Dès qu'un projet se fait parser il est aussi traduit dans la structure gérant la partie UML. Aucun algorithme d'affichage n'est présent pour le moment, tout est affiché dans le coin en haut à gauche de la fenêtre.
- Création de la classe ImportData.java.

 **Semaine 21.05 - 27.05.12 :**

- Une amélioration a été apportée au parseur Java
- Mise à jour du diagramme UML
- La création de la classe ExportData.java permet de démarrer le processus de génération de fichier de code source à partir du diagramme UML. L'utilisateur peut décider dans quel dossier il veut placer ces nouveaux fichiers.
- Dans L'interface utilisateur de nouveaux boutons ont été créés dans la barre du menu et dans le menu pour effectuer les opérations d'importation et d'exportation.

 **Semaine 28.05 - 03.06.12 :**

- Modification de la vue.
- Rédaction du rapport intermédiaire.
- Le système d'exportation a été modifié : maintenant un seul *Writer* est présent et il peut gérer tous les types d'affichage.
- Création de JavaVisitor.java et de CppVisitor.java, les visiteurs pour gérer la génération de code source Java respectivement C++.

 **Semaine 04.06 - 10.06.12 :**

- Création d'un nouveau projet avec la version 1.7 (JRE 1.7)
- Meilleure gestion des associations 1-N..N avec affichage dans la vue du type de l'association (LinkedList, HashMap, etc), création du type énuméré Collection.java, mise à jour du parseur et de la structure de données.

 **Semaine 11.06 - 17.06.12 :**

- La rédaction du rapport intermédiaire est reprise, corrigée et envoyée au professeur responsable.
- Création de la classe Layout.java. Cette classe permet d'afficher convenablement un diagramme de classe UML dans l'éditeur après importation.
- Un bouton a été rajouté dans l'éditeur pour permettre l'appel à la fonction d'affichage.

 **Semaine 18.06 - 24.06.12 :**

- Mise à jour de l'affichage.
- Création du Powerpoint pour la présentation.
- Présentation du logiciel devant le professeur responsable et ses assistants.
- Amélioration du parseur Java.

 **Semaine 25.06 - 01.07.12 :**

- Mise en place de la synchronisation uml --> code : la vue à dû être modifiée avec l'insertion d'un bouton « sync » dans le menu. Dans le fichier xml de sauvegarde une ligne a été rajoutée avec l'adresse du fichier à synchroniser. Modification de plusieurs autres fichiers pour permettre la synchronisation.
- Pour commencer l'importation possibilité de la faire grâce au Drag&Drop.
- Possibilité de faire l'exportation avec mémoire.
- Séparations des fichiers de code source en 4 différents paquetages :
 - dataRecord
 - dataRecord.io
 - dataRecord.Elements
 - dataRecord.ElementType

 **Semaine 02.07 - 08.07.12 :**

- Amélioration de la synchronisation entre l'UML et le code.
- Dans l'affichage les liens entre les pères et les fils ont été modifiés pour une meilleure lisibilité du diagramme.
- Ajout de la javadoc dans la plupart des classes et des méthodes
- Amélioration du Parseur.

 **Semaine 09.07 - 15.07.12 :**

- Rédaction du rapport final.
- Finalisation de la javadoc.
- Génération de la documentation avec Doxygen.
- Test de l'application.
- La classe Field change de nom. Nouveau nom : Variable.

 **Semaine 16.07 - 22.07.12 :**

- Rédaction du rapport final.
- Changement de Extendable et Implementable en ClassKind et InterfaceKind.
- Meilleur gestion des exceptions.
- Création du parseur des fichiers « .h ».
- Création de Parser pour une interchangeabilité des parseurs.
- Création de PreprocessorStmt.java dans la branche de Statement.
- Création de PointerType et ReferenceType pour la gestion des pointeurs et des references.
- Test de l'application.

 **Semaine 23.07 - 29.07.12 :**

- Finalisation du rapport.
- Test de l'application.
- Relecture des directives du projet de diplôme de Bachelor
- Création du Dvd.
- Contenu du Dvd :
 - Slyum2.0.jar
 - Readme.txt
 - Rapport.pdf
 - Cahier des charges
 - Sources
 - Doxygen documentation
- Rendu final.

11 Annexe A

La Planification

12 Annexe B

DVD avec :

- ❖ Slyum2.0.jar
- ❖ Readme.txt
- ❖ Rapport.pdf
- ❖ Cahier des charges
- ❖ Sources
- ❖ Documentation Doxygen