C++ Parser for PADL

par:

Sébastien Robidoux

(robidose@iro.umontreal.ca)

Ward Flores

(floresvw@iro.umontreal.ca)

date:

5 octobre 2004

Université de Montréal

Table de matières

I. Introduction.

L'analyse syntaxique (parsing) des programmes C/C++ est une tâche importante car des milliers de programmes sont écrits en C/C++ (parmi les plus connus : Mozilla, Apache, Linux) que l'on voudrait analyser pour évaluer leur qualité, leur maintenabilité, etc. Cependant, l'analyse syntaxique de programmes C/C++ est difficile car le code source de ces programmes contient des directives de pré traitement et parce que la syntaxe "lâche" de ces langages résulte parfois en du code très complexe.

À ce jour et à notre connaissance, il n'existe qu'une poignée d'analyseurs syntaxiques fiables pour C/C++ : GCC, PCCTS, Spirit (?), CppETS, Lex/Yacc. . . Cependant, aucun ne semble offrir la possibilité de facilement être appelé pour d'autres raisons et par d'autres programmes que leurs générateurs de code respectifs. C'est pourquoi, il est aujourd'hui important d'étudier les analyseurs syntaxiques existants et de les caractériser pour isoler un analyseur syntaxique à la fois rapide, fiable et qui peut être utilisé par des programmes tiers.

Nos buts par ce projet sont donc : D'un part, observer plusieurs analyseurs syntaxiques existants et comparer leur rapidité, leur fiabilité et la possibilité de les interfacer avec des programmes en Java pour réaliser des analyses de programmes C/C++. D'autre part, connecter l'analyseur syntaxique qui semble le plus prometteur avec l'outil de rétro-conception des programmes Ptidej UI qui permet d'abstraire le code source en diagrammes de classes de plus haut niveau.

II. Environnement de travail et contraintes techniques.

Le travail est effectué en Java, avec l'environnement de développement pour Java fourni avec la plate-forme Eclipse. Un partage des codes sources du projet est rendu possible grâce à CVS qui est un engin de partage de codes sources sécurisé. Cet outil nous permet de partager les codes sources entre les coéquipiers et le professeur de façon très simple et sans tracas. Nous utilisons aussi l'outil de rétro-conception des programmes Ptidej UI dont le code source nous est fourni par l'intermédiaire du CVS. Finalement, le travail se fait sur des machines qui ont Windows comme système d'exploitation.

III. Première partie : Le choix d'un analyseur syntaxique.

La première partie du projet consiste en l'observation détaillée des analyseurs syntaxiques du langage C++ avec le but précis de trouver le plus performant pour le connecter ainsi à l'outil Ptidej UI. Cette première partie se divise en deux étapes. En effet, la recherche de l'analyseur syntaxique se fait, d'un côté, parmi les parseurs et/ou compilateurs, et d'un autre côté, parmi les compilateurs de compilateurs.

a) Les parseurs et/ou les compilateurs.

Les parseurs disponibles pour le langage C++ sont très nombreux sur le web. Malheureusement, la majorité d'entre eux ont été conçue pour des besoins particuliers et donc, ils peuvent rarement être utilisés pour des programmes tiers. Par exemple, parmi les parseurs étudiés, on retrouve :

- **CINT**

Cet outil est un *interpreter* qui peut analyser du code source C++. Dans l'implantation de CINT, on trouve des classes ERTTI (Extended Run Time Type Identification) qui donnent information sur la structure des programmes analysés. Ces classes sont, par exemple : ClassInfo, BaseClassInfo, DataMemberInfo, MethodInfo, MethodArgInfo, TypeInfo, etc. Malheureusement, cet outil ne donne aucune information sur la partie interne d'une méthode et en plus l'outil contient plusieurs limitations syntaxiques (détails confirmés par l'auteur).

- **John's PCCTS-based C++ Parser**

PCCTS (*Purdue Compiler Construction Tool Set*) est un *embedded* C++ Parser aussi populaire que le GCC Parser. Cet outil a une communauté active des usagers, ce qui a contribué au parser d'être amélioré plusieurs fois. Notamment, John Lilley a contribué avec ce parser pour qu'il soit complet avec un pre-processeur, une table de symboles, etc. Malheureusement, ce parser ne génère pas d'ASA. PCCTS a été écrit initialement par Terence J. Parr, mais il a décidé de faire une re-modélisation complète de PCCTS, ce qui a donné : un compilateur de compilateurs, l'ANTLR.

- **GCC**

GCC (*GNU Compiler Collection*) est l'un de parsers le plus utilisé dans la communauté scientifique. GCC est une collection de compilateurs qui gère plusieurs langages de programmation (C, C++, Java, Fortran, Ada, etc.). Le compilateur pour le langage C++ et G++. Bien que GCC (G++) produit un arbre de syntaxe abstraite pour un programme analysé, il ne donne aucun moyen d'accès à celle-ci, car l'ASA n'est géré qu'intérieurement.

- **PUMA**

PUMA (*Pure Manipulator*) est une librairie de classes pour faire l'analyse lexicale, syntaxique et sémantique tout comme la manipulation de C/C++ code sources. Cet outil peut aussi générer des ASA et est muni d'un pre-processeur intégré. Lors du téléchargement, on a remarqué que cet outil avait été fusionné avec l'outil AspectC++ (une extension pour la programmation AOP, *Aspect-Oriented*), ce qui rendait plus complexe l'utilisation de PUMA. Toutefois, on conseille fortement d'étudier plus en détail ce parseur dans une future continuation de ce projet.

- **Sage++**

  Cet outil n'est pas un simple parseur. Il est un compilateur pre-processeur orienté objet. Cet outil peut restructurer un code source C++ en optimisant les boucles, les variables, etc. Cet outil est conçu pour travailler avec pC++, ce qui est une extension du langage C++ qui est supposément portable et qui permet les opérations parallèles des données. Donc, le problème avec cet outil est qu'il ne supporte pas encore le travail direct avec C++, il faut toujours passer par l'intermédiaire de l'extension pC++.

a) Les compilateurs de compilateurs et/ou les générateurs de parseurs.

De la même façon que les parseurs, on retrouve de nombreux compilateur de compilateurs et/ou de générateurs de parseurs sur le web. Certains supportent plusieurs langages en même temps (il suffit de se procurer une grammaire appropriée) et certains ont été implantés pour un langage en particulier. Le problème qui revenait le plus fréquemment avec ces outils était celui de se procurer une bonne grammaire (complète et fonctionnel) du langage C++, car sans la grammaire ces outils étaient inutiles pour nous.

- **CppCC**

  CppCC est un générateur de parseurs LL(k) qui est encore en développement. Cet outil vise les plateformes Unix. Bien qu'on puisse y trouver une version Windows, celle-ci ne marche pas sous la version XP de Windows, et donc on n'a pas pu vraiment tester l'outil en question.

- **Spirit**

  Spirit est aussi un générateur de parseurs. Spirit est implémenté en utilisant *template meta-programming techniques*, ce qui permet de « transformer » la spécification de la grammaire EBNF en du code C++. Ainsi de cette façon, la grammaire peut être exécutée immédiatement tandis que les générateurs de parseurs conventionnels doivent passer la source EBNF dans du code C++ avant de pouvoir exécuter la grammaire utilisée. Malheureusement, on n'a pas eu le temps de tester cet outil.

- **SLK**

  Ce générateur de parseurs se vante d'être le seul «vrai » analyseur LL(k). Les auteurs affirment que cet outil est doté des puissants algorithmes LL(k) que permettent de créer des parseurs plus rapides et plus compacts. Ce générateur de parseurs supporte C, C++, Java et C#. Malheureusement, il faut modifier manuellement SLK pour permettre l'outil de générer des ASA. Toutefois, cet outil semble être très prometteur (si on se fie à la documentation de cet outil) par conséquent une étude plus détaillée sur l'outil en question serait convenable.

- **JavaCUP**

JavaCUP est un générateur de parseurs LALR. Il est écrit en Java est, pour le moment, cet outil ne supporte pas encore C++.

- **Flex/Bison, Lex/Yacc**

Flex/Bison et Lex/Yacc supportent le langage C++. Toutefois, ces outils n'ont pas été testés pour manque du temps et surtout parce qu'on a décidé de mettre en priorité les outils écrits en Java. Notamment, ANTLR et JavaCC.

- **ANTLR**

ANTLR (*ANother Tool for Language Recognition*) est un compilateur de compilateurs qui supporte plusieurs langages et qui est le successeur de l'outil PCCTS, ces deux outils ont été créés par Terence J. Parr. ANTLR est un outil *open source* et il pourvu des plusieurs grammaires déjà fonctionnels. Avec ces 5 000 téléchargements par mois [1], cet outil est muni d'une communauté large et active.

Bref explication du fonctionnement de l'outil ANTLR

ANTLR est un outil, écrit en Java, qui accepte la spécification d'une grammaire d'un des langages supportés. Avec cette grammaire, l'outil produit un parseur qui permet l'analyse du code de ce langage tout en suivant les règles établies dans la grammaire. On peut ajouter des opérateurs et des actions à la grammaire pour pouvoir générer des ASA et des actions de sortie.

Pour faciliter l'explication du fonctionnement de ANTLR, on va supposer qu'on veut écrire une grammaire « Expr ». Cette grammaire doit avoir l'extension ".g" (p.e. Expr.g). Elle doit être constitué des subclasses de ***Lexer***, ***Parser*** et ***TreeParser*** (cette dernière est implémentée seulement si on veut travailler avec l'ASA). Puis, les règles de la grammaire à l'intérieur de ces classes doivent être écrites en notation **EBNF**.

```
class ExprParser extends Parser;
expr:   mexpr ((PLUS|MINUS) mexpr)*;
mexpr:   atom (STAR atom)*;
.... suite

class ExprLexer extends Lexer;
options {....}
LPAREN: '(' ;
RPAREN: ')' ;
.... suite
```

Une fois la grammaire écrite (Expr.g), il faut générer l'analyseur lexical, syntaxique, etc. Cela se fait en utilisant la commande :

[1] Information obtenue du site http://www.antlr.org/

```
$ java antlr.Tool Expr.g
```

Puis, ANTLR générera les fichiers Java suivants :

ExprLexer.java : Ce fichier Java s'occupe de faire l'analyse lexicale suivant les règles décrites dans la grammaire. Si on regarde dans la classe, on peut remarquer qu'il y aura une méthode pour chaque règle définie dans la subclasse du *lexer.*

ExprParser.java : Ce fichier Java s'occupe de faire le *parsing* suivant les règles décrites dans la grammaire. De la même façon que pour l'analyseur lexicale, on peut noter que pour chaque règle définie dans la subclasse du *parser,* on trouvera leur méthode respective.

ExprParserTokenTypes.java : Ce fichier Java définira les chiffres à utiliser pour chaque symbole (paramètre constant) qui est utilisé dans le *parser* et/ou le *lexer.*

ExprParserTokenTypes.txt : Ce fichier texte nous montre la liste de symboles avec leurs chiffres respectifs qui ont été généré dans *ExprParserTokenTypes.java*.

Ensuite, pour pouvoir utiliser le parseur, il faut écrire une classe Main qui fera appelle aux classes générées.

```
import antlr.*;
public class Main {
   public static void main(String[] args) throws Exception {
       ExprLexer lexe r = new ExprLexer(System.in);
       ExprParser parser = new ExprParser(lexer);
       parser.expr();
    }
}
```

Maintenant, si on veut travailler avec des ASA, il faut faire quelques modifications à la grammaire. En effet, il faut activer l'option *buildAS T* dans la subclasse du *parser* et aussi ajouter de suffixes aux opérateurs pour indiquer lesquels doivent être considérés comme des sous-racines.

```
class ExprParser extends Parser;
options { buildAST=true; }
expr:   mexpr ((PLUS^|MINUS^) mexpr)*;
…. suite
```

Notre subclasse du *lexer* ne change pas, mais par contre notre classe Main doit subir aussi quelques modifications pour récupérer l'ASA. Puis, si jamais on veut parcourir l'ASA généré pour pouvoir l'analyser et/ou le modifier, il faudrait ajouter dans la grammaire une subclasse qui hérite de TreeParser avec les règles à utiliser lors du parcours de l'ASA.

Utilisation d'ANTLR avec la grammaire C++

ANTLR fourni l'ancienne grammaire C++ de PCCTS modifiée pour l'utilisation avec ANTLR. On peut observer dans cette grammaire un changement important : l'ajout d'un flag au début de la grammaire pour indiquer qu'on travaille avec le langage C++. Cela fera en sorte qu'on générera des fichiers ".cpp" au lieu de fichiers ".java".

```
header { … }
options { language = "Cpp";}
.... suite
class ExprParser extends Parser;
.... suite
class ExprLexer extends Lexer;
.... suite
```

Les fichiers générés par ANTLR pour notre grammaire C++ sont :

CPPLexer.cpp et **CPPLexer.hpp** : L'analyseur lexicale.

CPPParser.cpp et **CPPParser.hpp** : Le parseur.

STDCTokenTypes.hpp et **STDCTokenTypes.txt** : La liste de symboles.

En examinant bien la grammaire C++, on note que le flag pour générer l'ASA n'est pas actif. En plus, on voit que les opérateurs n'ont pas les suffixes indiquant s'ils doivent être des sous-racines ou non. Alors, on essaie de l'ajouter nous-mêmes pour pouvoir ainsi travailler avec l'ASA. Malheureusement, étant la grammaire très complexe, on se rend compte rapidement que cette tâche était beaucoup plus difficile qu'on l'imaginait. Mais, en faisant une recherche sur le Web, on arrive à trouver une autre grammaire[2] C++ ayant les spécifications nécessaires pour générer des ASA, mais le point faible de cette grammaire est qu'elle est dans un état incomplète et expérimental.

Bien que, à la limite, on puisse s'en passer de l'ASA pour intégrer le parseur à l'outil de rétro-conception des programmes Ptidej UI, on trouve que, en ayant l'ASA, cela facilitera notre travail grandement. Alors, on décide de continuer notre recherche avec le but d'en trouver un parseur qui permet cette action. Heureusement, l'outil JavaCC vient à notre rescousse.

- **JavaCC**

Le programme « JavaCC » (*Java Compiler Compiler*) est, comme son nom l'indique, un compilateur de compilateurs ou un générateur d'analyseur lexical et syntaxique. Il s'agit donc d'un outil qui lit les spécifications d'une grammaire et qui les convertis en un programme en «Java » qui peut ensuite analyser du code suivant

[2] Cette grammaire se trouve sur le site : http://www.imada.sdu.dk/~morling/

la grammaire. Ce programme est gratuit, disponible pour tous et depuis 2003, « JavaCC » est *Open Source*. Vous pouvez donc apporter vos propres modifications à ce programme comme bon vous semblerez.

Fonctionnement

En tout premier lieu, pour que « JavaCC » fonctionne correctement, nous avons besoin d'une grammaire qui spécifie le langage source que nous voulons analyser, dans notre cas, le langage C/C++. Cette grammaire est sous forme de fichier avec l'extension «.jj ». (Nous décrirons un peu plus tard les aspects généraux que cette grammaire doit posséder.)

Lorsque nous avons une grammaire correcte, sans erreur, et que nous exécutons « JavaCC » avec cette grammaire, il nous créera toujours une série de fichier de base qu'il crée pour chaque analyseur. Il s'agit de fichiers généraux qui sont identique peu importe la grammaire donnée.

- ***SimpleCharStream.java*** : Cette classe représente et gère l'enchaînement des caractères et des jetons (Token) du fichier contenant les sources du langage à analyser. Elle reçoit donc en paramètre pour son constructeur, un «Reader » sur le fichier de source.

- ***Token.java*** : Cette classe permet de représenter un jeton dans un code source d'un langage. Un jeton est la représentation d'un mot ou d'un symbole présent dans la grammaire.

- ***TokenMgrError.java*** : Cette classe, dérivé directement de la classe « *Error* », représente une erreur retournée par le gestionnaire de jeton qui sera expliqué plus bas.

- ***ParseException.java*** : Cette classe, dérivé directement de la classe « *Exception* », représente une exception retournée par l'analyseur syntaxique qui sera expliqué plus bas.

Ces quatre fichiers seront toujours créés peu importe la grammaire. Ils seront créés seulement s'ils ne sont pas présents dans le répertoire courant. Le seul moment où il sera nécessaire de les remplacer est si les options de « JavaCC » sont modifiées (si l'analyseur doit être statique ou non par exemple.). Donc il est fortement indiqué d'être certain des options de « JavaCC » pour la grammaire avant de faire des modifications dans ces quatre fichiers.

Le programme « JavaCC » produira aussi une série de fichiers qui seront directement dépendant de la grammaire. Il s'agit en fait de l'engin principal qui permet de faire les analyses lexicales et syntaxiques sur le fichier de code source d'après la grammaire.

• ***CPPParser.java*** : Ceci est la classe principale. Elle contient l'analyseur syntaxique. Ce fichier sera la conversion exacte du fichier de grammaire plus toutes sortes de méthodes générées automatiquement pour assurer le bon fonctionnement de l'analyseur. Elle contiendra aussi le « Main » du projet. Habituellement, ce « Main » recevra le nom du fichier source à analyser en paramètre.

• ***CPPParserTokenManager.java*** : Cette classe représente le gestionnaire de jeton, qui est aussi l'analyseur lexical du projet. Cette classe décortiquera donc le fichier source en jeton pour permettre l'analyse syntaxique par la suite. Cette classe implémente l'interface qui suit.

• ***CPPParserConstants.java*** : Une interface qui contient l'association entre les jetons et les noms symboliques. Donc, par exemple le jeton de type « 0 » représentera le symbole « EOF ».

• *Il est à noter que « CPPParser » au début de chaque fichier est différent, tout dépendant du nom du projet donné dans la grammaire. Dans notre cas, il est « CPPParser », mais pourrait être tout autre.*

Ces trois fichiers seront générés automatiquement à chaque exécution de « JavaCC », peu importe s'ils sont présents dans le répertoire courant ou non. Il est donc fortement déconseillé de faire une quelconque modification dans ces fichiers. De toute manière, les modifications doivent être faites dans le fichier de grammaire vu que ces trois fichiers en dépendent directement. Donc si une modification doit être faite dans un de ces trois fichiers, elle peut être faite dans le fichier de grammaire. Sinon, il faudra refaire la modification à chaque exécution de « JavaCC ».

<u>Spécification de la grammaire de « JavaCC »</u>

La grammaire de « JavaCC » est un fichier texte suivant une syntaxe particulière. La syntaxe peut s'apparenter beaucoup à des langages de programmation impérative ou concurrente comme C/C++ et Java. Comme par exemple :

```
void simple_type_specifier() :
{}
{
   (
   builtin_type_specifier()
   |
   qualified_type()
   )
}

void builtin_type_specifier() :
{}
{
   “void” | “char” | “short” | “int” | “long” | “float” |
   “double” | “signed” | “unsigned”
}
```

Ici, on veut représenter un nœud de type «simple_type_specifier() » qui est soit un « builtin_type_specifier() » ou un «qualified_type() ». Et ensuite, on voit que « builtin_type_specifier() » peut être soit « void », « char », etc. Il est donc assez simple de lire ou modifier une grammaire de « JavaCC ».

Donc, chaque nœud ou élément de la grammaire est représenté par une méthode ou fonction dans la syntaxe. Dans le premier couple d'accolade, nous pouvons inscrire du code «Java » d'initialisation, comme des variables locales par exemple. Ce code sera simplement copier dans l'analyseur résultant sans aucun changement. Dans la deuxième paire d'accolade, il s'agit de code suivant la syntaxe de « JavaCC ». Comme exemple simple, la barre «| » représente un «ou » entre des choix possibles pour la grammaire. Si nous avons besoin d'inscrire du code source « Java » dans la deuxième paire d'accolade, comme ajouter un « return … » par exemple, nous pouvons le faire en ajoutant une autre paire d'accolade. Tout ce qui se retrouvera dans cette nouvelle paire sera automatiquement inscrit dans le résultant sans aucun changement. Il est à noter de bien faire attention où vous placez les accolades, car « JavaCC » pourrait interpréter autre chose que ce que vous voulez faire.

« JavaCC » construit des grammaires de type « LL(k) ». Donc dans la grammaire, nous pourrons inscrire des appels à une fonction spéciale «LOOKAHEAD() » qui permet de faire des tests qui nous permet de suivre les spécifications.

Pour la représentation des jetons, le mot clé « TOKEN » est utilisé dans la grammaire, comme par exemple :

```
TOKEN :
{
  < LCURLYBRACE: "{" >
| < RCURLYBRACE: "}" >
| < LSQUAREBRACKET: "[" >
| < RSQUAREBRACKET: "]" >
| < LPARENTHESIS: "(" >
| < RPARENTHESIS: ")" >
...
}
```

Nous pouvons aussi représenter les informations qui seront laissé de côté par l'analyseur, comme les commentaires ou les espaces inutiles par exemple.

Au début de chaque grammaire, nous pouvons aussi ajouter un bloc d'option qui nous permet d'exécuter « JavaCC » avec une grammaire sans en connaître les paramètres importants. Comme par exemple, si une grammaire construit un analyseur qui est statique ou dynamique.

```
options {
  STATIC=true;
}
```

Pour ensuite définir le nom de la classe de l'analyseur, la méthode « main » de cette classe ou toutes autres méthodes qui ne seront pas créé explicitement par « JavaCC », nous devons les inscrire dans un bloc spécial. Ce bloc pourra contenir aussi toutes les importations nécessaires au code de l'analyseur, le nom du « package » auquel il va appartenir, etc.

```
PARSER_BEGIN(CPPParser)
package test.exemple;

public final class CPPParser {
  ...
}
PARSER_END(CPPParser)
```

Bien sur, tous les détails plus important se retrouvent sur le site de « JavaCC », https://javacc.dev.java.net/doc/.

JJTree

« JJTree » est un utilitaire qui fonctionne de concert avec «JavaCC ». Comme son nom l'indique, il permet de générer un arbre de syntaxe abstraite, ce que « JavaCC » ne fait pas explicitement. «JJTree » prend en paramètre une grammaire de « JavaCC », puis il retourne un fichier de type « .jj.jj » qui est exactement la même grammaire, mais avec du code en plus.

Voici sommairement, comment il fonctionne. Si nous reprenons l'exemple de grammaire précédente :

```
void simple_type_specifier() :
{ /* Création d'un noeud */ }
{
   (
   builtin_type_specifier()
   |
   qualified_type()
   )
{ /* Ajout du noeud dans l'arbre */ }
}

void builtin_type_specifier() :
{ /* Création d'un noeud */ }
{
   "void" | "char" | "short" | "int" | "long" | "float" |
   "double" | "signed" | "unsigned"
{ /* Ajout du noeud dans l'arbre */ }
}
```

Ici, quand il est indiqué « Création d'un nœud », « JJTree » ajoute du code « Java » qui permet la création d'un nœud et lorsqu'il est indiqué «Ajout du nœud

dans l'arbre », « JJTree » ajoute simplement le code nécessaire pour ajouter ce nœud dans l'arbre de syntaxe abstraite qu'il est entrain de créer.

Donc, pour que le tout fonctionne, « JJTree » ajoute plusieurs classes au projet pour l'analyseur, donc la plus important, « node.java ». Cette classe permet de représenté un nœud dans l'arbre. Avec des options de « JJTree », nous pouvons faire qu'il nous crée une entité différente de « node » ayant le nom de chaque méthode dans la grammaire. Donc dans l'exemple, il y aurait deux entités, « simple_type_specifier.java » et « builtin_type_specifier.java » qui hériteraient tous de « node.java ». Sinon, une entité nommée « simplenode.java » sera ajoutée au projet et représentera tous les nœuds. Une option permet aussi d'implanter le patron de conception visiteur pour chaque nœud de l'arbre. Les méthodes nécessaires seront implantées automatiquement à chaque nœud.

À chaque fois qu'un nœud est ajouté dans l'arbre, la référence à ce nœud est envoyée dans la liste des nœuds enfants du nœud qui est en cours de création. C'est ainsi que la hiérarchie dans l'arbre de syntaxe est créée.

Grammaire C++ pour « JavaCC »

Malheureusement, il ne suffit pas d'un bon compilateur de compilateur et d'un bon engin pour bâtir des arbres de syntaxe abstraite pour analyser du code source C++. Il nous faut une bonne grammaire pour C++. Et pour cela, nous avons du rechercher ailleurs que dans la liste de grammaire du site de « JavaCC ».

Nous avons du utilisé une grammaire disponible dans un paquet de source nommée « PMD ». Cette grammaire, créé en 1997 par le concepteur de « JavaCC », est une grammaire qui fonctionne bien et qui analyse le code C++ standard à cette date. Il construit un analyseur statique qui contient un genre de base de données des symboles supplémentaire. En plus de toutes les classes créées automatiquement par « JavaCC », cette grammaire requiert plusieurs classes qui sont fournis dans le paquet « PMD ».

-*SymtabManager.java* : Cette classe permet de conserver chaque nom de classe ou type spécial. Ceci permet de faire des recherches simples pour les noms des constructeurs ou des destructeurs ou de savoir si un nom de classe est présent dans le code source analyser, etc. Il s'agit d'un engin qui est semblable avec les « IdiomLevelModel ». Il est pratique, mais cet engin nous a causé quelques problèmes que nous avons pu, heureusement, régler.

-*Scope.java* : Contient la définition d'un environnement de variable et de nom de méthode. Des recherches sont faites pour savoir si un nom est contenu dans un scope.

-*ClassScope.java* : Classe qui hérite de « Scope.java » pour définir une classe comme un environnement.

Ces trois classes sont pratiques et nécessaire dans l'implémentation de la grammaire, mais ils pourraient sûrement, avec une bonne analyse, être totalement être remplacer par les «IdiomLevelModel». Cette notion de table de symbole nous a aussi créé quelque problème. Vu qu'il cherchait si chaque symbole (nom de classe, de méthode, de type variable, etc.) était présent dans la table, si, par exemple, une classe faisait référence à une autre classe qui n'est pas présente dans le code source qui est présentement analyser, et bien il sortait une erreur d'analyse. Donc, pour analyser un projet, il fallait que tout soit dans un seul fichier. Ce qui n'était pas très pratique dans « Ptidej UI », alors que nous pouvons recevoir un projet en plusieurs fichiers et même une seule classe par fichier. Heureusement, il fut très facile de court-circuiter le test qui sortait l'erreur d'analyse. Donc maintenant, l'analyseur construit avec cette grammaire peut recevoir un projet en plusieurs fichiers sans erreur.

Pourquoi avons nous choisis « JavaCC » ?

Voici un petit tableau démontrant les principaux critères qui nous on permis de faire un choix éclairé :

Critères	ANTLR	JavaCC
Analyse une version standard de C++	Oui	Oui
Retourne un ASA facilement utilisable	ASA possible*	Oui
Utilisable par d'autre	Oui	Oui
Sans interface graphique	Oui	Oui
Outil implémenté en C++ ou Java	Java	Java
La grammaire C++ produit des fichiers C++ ou Java	C++	Java
Grammaire facile à utiliser	Un peu	Oui

*ASA est possible, mais doit être implanté manuellement. Plus compliqué à faire.

Donc, comme on peut le remarquer, les deux analyseurs sont presque semblables sur tous les points. Ils sont tous deux en « Java », un aspect très important pour l'implémentation de cet analyseur avec « Ptidej UI » qui est en « Java ».

Ensuite, l'autre aspect très important était la fabrication d'un arbre de syntaxe abstraite. À ce moment du projet, nous pensions que c'était la seule façon de pouvoir intégré le résultat de l'analyse avec le méta-modèle de « Ptidej UI ». Donc il fallait absolument avoir accès à l'ASA. «JavaCC » nous offrait la meilleure possibilité avec l'utilitaire «JJTree » qui faisait tout le travail. Il était bien sur possible de faire un ASA avec « ANTLR », mais il fallait le faire manuellement. Une tâche ardue à

comparer de «JavaCC ». En plus, « JavaCC » offrait une grammaire un peu plus facile à modifier que « ANTLR ».

Donc, en conclusion, on voit assez bien pourquoi nous avons choisit « JavaCC » au lieu de « ANTLR » pour analyser le code C++ pour « Ptidej UI ».

III. Deuxième partie : La connexion avec l'outil « Ptidej UI ».

Rendu à cette étape, nous nous sommes rendu compte qu'il était probablement plus facile de modifier la grammaire avec les actions sémantiques plutôt que de relire un arbre de syntaxe abstraite. Nous sauvions ainsi beaucoup plus de temps. Bien sur, l'analyse de l'ASA pour chaque code source nous permet de voir le fonctionnement de la grammaire beaucoup plus facilement. C'est pourquoi, ici, nous montrerons comment nous sommes arrivé à la conclusion d'où mettre chaque action sémantique dans la grammaire.

a) Ajout des actions sémantiques a l'analyseur syntaxique

Premièrement, définissons un peu comment fonctionne le méta-modèle de « Ptidej UI ». Il s'agit d'un engin qui fonctionne suivant le patron de conception «Usine Abstraite » (ou « Abstract Factory » en anglais). Donc dans le projet «Ptidej UI », il existe plusieurs classes et interfaces qui représentent les entités ou les éléments de la grammaires «Java », qui est très semblable à C++. Bien sûr nous avons dû ajouter des classes et interfaces au méta-modèle, ce qui sera discuté plus bas.

Donc, dans le méta-modèle, nous retrouvons les interfaces « IClass », « IMethod », « IConstructor », etc. Chacune de ces interfaces seront donc utilisées dans la grammaire pour représenter chacune des représentations.

Pour pouvoir utiliser ces interfaces, avec l'*usine abstraite*, nous devons utiliser les méthodes « CreateClass() », « CreateMethod() », « CreateConstructor() », etc. Ces méthodes vont donc créer, indépendamment de la version du méta-modèle, les objets voulus. Mais pour utiliser ces méthodes, il faut avoir un réceptacle qui va pouvoir recevoir les entités ou éléments créés par la grammaire. Ce réceptacle est « IdiomLevelModel » et il va contenir l'usine (le méta-modèle) à utiliser pour pouvoir recevoir ainsi tous les éléments créés par les méthodes de création de l'usine.

Donc la première étape pour intégrer notre analyseur avec «Ptidej UI » est de recevoir un «IdiomLevelModel » et une liste de nom de fichier à analyser. La classe qui sera appelé par « Ptidej UI » et qui recevra ces paramètres devra aussi implémenter l'interfaces « IIdiomLevelModelCreator ». Cette classe contiendra donc une méthode « Create() » qui appellera l'analyseur pour chaque fichier de la liste et avec le « IdiomLevelModel » qui contiendra la représentation du fichier. Cette classe est la classe « CppCreator.java ». Ceci fait, il nous reste simplement à recevoir un

« IdiomLevelModel » dans notre code source de l'analyseur et d'ajouter chaque élément dedans.

Les Classes

La principale entité de tout code source C++ est la classe. Il s'agit de l'entité la plus complexe que nous avions à analyser. Elle comporte plusieurs sous-éléments : des méthodes et des membres.

Voici la déclaration d'une classe que l'on retrouve souvent dans le « header » d'une classe :

```
CLASS <classe> {
public:
  constructeur
  destructeur
  méthode
private:
protected:
};
```

et voici la représentation fait par l'ASA :

```
->external_declaration
-> declaration
->  declaration_specifiers
->   class_specifier
```

Ici, le nom de la classe se retrouve dans le nœud « class_specifier » de la grammaire. Ensuite, chaque enfant du noeaud « class_specifier » sera de type « member_declaration ». Ce type de nœud peut contenir autant un membre, une fonction que la description d'accès comme « public », « private » et « protected ». Ce type de nœud peut contenir la déclaration d'une variable, d'un constructeur, d'un destructeur, d'une fonction et la description du type d'accès. En d'autres mots, ce type de nœud contient toutes les informations propres qui seront définit dans une classe.

Pour les descriptions d'accès, nous aurons donc

```
-> member_declaration
->  access_specifier
```

Donc dans la méthode « access_specifier() », nous pouvons à chaque fois stocké dans une variable le type des éléments de la classe qui seront déclaré ensuite.Voyons maintenant chaque élément contenu dans une classe :

Les membres

Voici la description générale d'un membre d'une classe d'après l'ASA.

```
-> member_declaration
->  declaration_specifiers
->   builtin_type_specifier
```

```
->  member_declarator_list
->   member_declarator
->    declarator
->     direct_declarator
->      qualified_id
```

Ici, le nom du membre sera stocké dans le nœud «qualified_id » et son type dans le nœud « builtin_type_specifier ». Il est a noter que ceci est pour le cas d'un type primitif. S'il s'agit d'un type conçu par le concepteur (comme une classe par exemple) et bien la structure sera :

```
-> declaration_specifiers
->  qualified_type
->   qualified_id
```

au lieu de

```
-> declaration_specifiers
->  builtin_type_specifier
```

Ceci est vrai pour toutes les déclarations de type dans le programme. Que ce soit une fonction ou un membre ou la déclaration d'une variable locale ou globale. Il est aussi à noter que le nœud «member_declarator_list » peut contenir plusieurs enfants de type « member_declarator ». Ceci représente les déclarations de type « int a, b, c, d; ». Si le membre est un tableau, nous aurons donc, suivant le nœud « member_declarator_list »

```
-> member_declarator_list
->  member_declarator
->   declarator
->    direct_declarator
->     qualified_id
->     declarator_suffixes
->      constant_expression
->       conditional_expression
->        logical_or_expression
->         logical_and_expression
->          inclusive_or_expression
->           exclusive_or_expression
->            and_expression
->             equality_expression
->              relational_expression
->               shift_expression
->                additive_expression
->                 multiplicative_expression
->                  pm_expression
->                   cast_expression
->                    unary_expression
->                     postfix_expression
->                      primary_expression
->                       constant
```

Ici nous remarquons l'ajout du nœud «declarator_suffixes » qui nous indique la présence de []. La grandeur du tableau sera contenu dans le nœud «constant ». Si la taille du tableau est une variable, alors le nœud « constant » sera remplacé par un nœud « id_expression ». Le nom du tableau est toujours, comme tous les membres, dans le nœud « qualified_id », frère du nœud « declarator_suffixes ».

Si le membre est un pointeur, alors le nœud «direct_declarator » sera décalé. Il sera remplacer par d'autres nœud. Le nœud «direct_declarator » existe toujours, il est seulement décalé, car on ajoute le nœud «ptr_operator » pour indiquer la présence d'un pointeur.

```
-> member_declarator_list
->  member_declarator
->   declarator
->    ptr_operator
->     cv_qualifier_seq
->    declarator
->     direct_declarator
->      qualified_id
```

Le nom du membre sera toujours contenu dans «qualified_id », mais le chemin sera différent.

Donc en résumé, la création d'un membre dans le méta-modèle sera toujours fait à la fin de la méthode «membre_declarator() », le type du membre sera toujours trouvé dans la méthode «declaration_specifier() » et il sera toujours facile de savoir s'il s'agit d'un pointeur ou d'un tableau.

Les méthodes

Les méthodes se divise en type. Il y a les méthodes ordinaires, les constructeurs et les destructeurs. Chaque méthode comprend un nœud, « compound_statement » qui contient le code de la méthode en question. La suite sera toujours tronqué, car elle ne nous est pas utile dans notre cas. Pour le cas de constructeurs et de destructeurs, le nœud « compound_statement » indique le début du corps de ceux-ci et dans le cas de fonctions, c'est plutot le nœud «func_decl_def ». Par exemple, une fonction abstraite aura quand même le nœud « func_decl_def » mais elle n'aura pas le nœud « compound_statement ».

Paramètre d'une méthode

La déclaration de paramètres d'une méthode expliquer ici-bas s'applique aussi lorsqu'il s'agit de constructeurs et destructeurs. La liste de paramètre se déclare comme suit :

```
-> parameter_list
->  parameter_declaration_list
->   parameter_declaration
->    declaration_specifiers
->     builtin_type_specifier
->    declarator
->     direct_declarator
->      qualified_id
```

Ici, s'il y a plusieurs paramètres, le nœud «parameter_declaration » se répétera, tout en restant enfant de « parameter_declaration_list ». Le type sera contenu dans « builtin_type_specifier » (voir membre pour plus de détaille sur les types) et le nom du paramètre dans «qualified_id ». Si les paramètres ne sont pas définit, s'ils sont abstraits (ex.: void foo (char * , int){ };), la déclaration diffère un peu.

```
-> parameter_declaration
->  declaration_specifiers
->    builtin_type_specifier
->  abstract_declarator
```

Ici le nœud « declarator » est remplacer par « abstract_declarator ». Si la description abstraite du paramètre est un pointeur, alors nous trouvons :

```
-> parameter_declaration_list
->  parameter_declaration
->   declaration_specifiers
->    builtin_type_specifier
->   abstract_declarator
->    ptr_operator
->     cv_qualifier_seq
->    abstract_declarator
```

Bien sur, dans les deux cas précédents, on ne peut avoir le nom du paramètre. Donc en résumé, nous pouvons créer chacun des paramètres avec l'interface « IParameter » dans la méthode « parameter_declaration » et sont type dans « declaration_specifier ». Un pointeur ou un tableau se trouve aussi facilement que pour une membre. Ensuite, nous pouvons envoyer chaque paramètre dans une liste. Comme cela, ils seront stockés et ajoutés à la prochaine méthode qui sera créée.

Les constructeurs

Les constructeurs se déclare comme suit dans l'ASA :

```
-> member_declaration
->  ctor_definition
->   dtor_ctor_decl_spec
->   ctor_declarator
->    qualified_id
->    parameter_list
->     parameter_declaration_list
->      parameter_declaration
->       declaration_specifiers
->        builtin_type_specifier
->       declarator
->        direct_declarator
->         qualified_id
->   compound_statement
```

Ici, le nom du constructeur se trouve dans « qualified_id » suivant immédiatement « ctor_declarator ». Il est a noter ici que si le constructeur ne contient pas de paramètre, le nœud « parameter_list », ainsi que ces enfants, ne seront pas là.

```
-> member_declaration
->  ctor_definition
->   dtor_ctor_decl_spec
->   ctor_declarator
->    qualified_id
->   compound_statement
```

Donc en résumé, pour créer un constructeur avec l'interface «IConstructor », nous allons dans la méthode «ctor_declarator ». Chaque paramètre ayant déjà été trouvé

et ajouté à une liste avant la création du constructeur, il ne suffira qu'à les ajouter itérativement au constructeur.

Les destructeurs

Les destructeurs se déclare comme suit dans l'ASA :

```
-> member_declaration
->  dtor_definition
->   dtor_ctor_decl_spec
->   dtor_declarator
->    simple_dtor_declarator
->   compound_statement
```

Ici le nom du destructeur n'est pas stocké. Mais bien sur, il s'agit du même nom que la classe avec le symbole « ~ » avant le nom. Si le destructeur possèdes des paramètres, il aura la structure suivante dans l'arbre.

```
-> member_declaration
->  dtor_definition
->   dtor_ctor_decl_spec
->   dtor_declarator
->    simple_dtor_declarator
->     parameter_list
->      parameter_declaration_list
->       parameter_declaration
->        declaration_specifiers
->         builtin_type_specifier
->        declarator
->         direct_declarator
->          qualified_id
->   compound_statement
```

Donc en résumé, pour créer un destructeur avec l'interface « IDestructor », nous allons dans la méthode « simple_dtor_declarator ». Chaque paramètre ayant déjà été trouvé et ajouté à une liste avant la création du constructeur, il ne suffira qu'à les ajouter itérativement au constructeur.

Les méthodes « ordinaires »

Les méthodes "ordinaire" se déclare comme suit dans l'ASA :

```
-> member_declaration
->  function_definition
->   declaration_specifiers
->    builtin_type_specifier
->   function_declarator
->    function_direct_declarator
->     qualified_id
->     parameter_list
->      parameter_declaration_list
->       parameter_declaration
->        declaration_specifiers
->         builtin_type_specifier
->        declarator
->         direct_declarator
->          qualified_id
->   func_decl_def
```

Ici, le nom de la méthode sera stocké dans le nœud «qualified_id » et son type dans le nœud « builtin_type_specifier ». Il est a noter que ceci est pour le cas d'un type primitif. S'il s'agit d'un type conçu par le concepteur (comme une classe par exemple) et bien il sera différent.

```
-> declaration_specifiers
->  qualified_type
->   qualified_id
```

au lieu de

```
-> declaration_specifiers
->  builtin_type_specifier
```

Tous les paramètres seront contenus dans le nœud « parameter_list ». Et bien sur, s'il n'y a pas de paramètre à cette méthode, aucun nœud « parameter_list » ne sera présent. Et aussi, tout ce qui suit le nœud « func_decl_def » sera ignoré pour nos besoins.

Donc, pour la création d'une méthode « ordinaire » se fait dans la méthode « function_direct_declarator ». Sont type se retrouvera dans la méthode « declaration_specifier » et chaque paramètre qui seront déjà trouvé et stocké dans une liste seront ajouter dans la méthode.

En conclusion, pour ajouter une classe dans le modèle, nous devons d'abord ajouter, à l'entité, tous les éléments de cette classe (les méthodes et les membres), qui seront toujours déclaré et stocké avant. Cette entité sera ensuite ajouter au modèle.

Les structures/unions

Pour les structures et les unions, nous avons exactement la même structure qu'une classe. La différence sera essentiellement qu'il n'y aura jamais de méthode de définie dans une structure ou un union. Il y aura simplement un « switch/case » dans la méthode « class_specifier » qui fera la distinction entre une classe, une structure ou une union.

Les énumérations

Les énumérations sont assez simples à découvrir dans la grammaire et à ajouter. Leur structure dans l'ASA est comme suit.

```
-> external_declaration
->  enum_specifier
->   enumerator_list
->    enumerator
```

Ici, le nom se situera dans le nœud « enum_specifier ». Ensuite, chaque nœud « enumerator » contient chaque élément de l'énumération. Donc, la création d'une énumération dans le modèle se fait dans la méthode «enum_specifier() ».

Les variables globales

Nous pouvons imaginer déjà que nous trouverons les variables globales dans une déclaration externe, car les variables globales sont déclarées justement à l'extérieur des classes. Alors, une variable globale est représentée par l'ASA de la façon suivante :

```
-> external_declaration
->  declaration
->   declaration_specifiers
->    builtin_type_specifier
->   init_declarator_list
->    init_declarator
->     declarator
->      direct_declarator
->       qualified_id
```

Tout comme pour les paramètres, si nous avons une liste de globalField, elle sera représentée après *init_declarator_list*.

Les liens d'héritage

L'héritage se retrouve toujours dans la déclaration d'une classe.

```
-> external_declaration
->  declaration
->   declaration_specifiers
->    class_specifier
->     base_clause
->      base_specifier
->       access_specifier
->      base_specifier
->       access_specifier
```

Ici, nous trouvons le nom de la super classe dans le nœud «base_specifier » et l'accès à cette super classe se trouve dans le nœud « access_specifier » qui le suit directement. On remarque ici qu'une classe en C++ peut hériter de plusieurs classes. Donc, pour ajouter une super classe dans une classe, dans la méthode « base_specifier() », on ajoute le nom de la super classe dans une liste. Lors de la création de la classe, nous pourrons ajouter chaque élément de la liste à la classe avant de l'ajouter dans le modèle par un lien d'héritage.

Les autres liens

Pour l'instant, tous les autres liens sont considérés comme des liens « UseRelationship ». Donc ce que nous voulons faire avec ces liens est de représenté qu'une classe utilise une autre classe, énumération, structure ou union. Comme nous avons expliqué précédemment, tous les types de variable, de membre, ou de retour de méthode se divise en deux types : des types primitifs et des types définis. Les types primitifs sont « int », «char », «double », etc. Les types défini représente une classe, un énumération, etc.

Donc, nous devons faire un lien avec tout ce qui est un type défini. Les type défini se retrouve tous dans une structure comme celle-ci.

```
-> declaration_specifiers
->  qualified_type
->   qualified_id
```

Donc nous avons qu'à ajouter chaque nom de type que nous trouvons dans le nœud « qualified_type » dans une liste et d'ensuite créer un lien entre cette entité trouvé et la classe qui est créé.

Les « Ghost »

Que devons nous faire si jamais, une classe hérite d'une autre classe, mais que cette classe n'est pas compris dans le modèle qui est créé? Simple, nous devons ajouter une entité fantôme qui va représenter symboliquement la classe manquante. Donc à chaque fois que nous tentons de créer un lien entre deux entités, si une entité n'est pas dans le modèle, nous créons une entité fantôme.

Bien sur, il faut s'assurer que si par après, nous ajoutons une entité qui a été représenté par une entité fantôme, nous devons enlever le fantôme et le remplacer par la bonne entité. Donc, nous avons ajouté une méthode qui permet de remplacer un fantôme par la vraie entité et de remplacer tous les liens qui auraient pu être fait avec ce fantôme.

b) Ajout des structures au Méta-modèle (PADL).

Le méta-modèle fourni par le professeur Guéhéneuc est très complet par rapport au code Java. En effet, ce méta-modèle est pourvu de la spécification nécessaire pour gérer tous les détails propre d'une implémentation dans ce langage, comme par exemple : l'héritage, les interfaces, etc.

Or, bien que le langage C++ soit semblable au langage Java, il est aussi différent dans plusieurs aspects et par conséquent, nous avons dû ajouter certaines structures propres au langage C++ et introuvables dans le langage Java. Ces différences sont, par exemple, la présence des «Struct » dans le langage C++ tandis qu'en Java on ne trouve pas cette entité.

En plus, la différence entre ces deux langages ne s'arrête pas là parce que ces deux langages peuvent supporter un même principe, comme par exemple l'héritage, mais ce même principe est implémenté d'une façon différente dans chaque langage. En effet, en C++ on peut trouver de l'héritage multiple tandis qu'en Java l'héritage existe mais l'héritage multiple n'est pas supporté.

Alors, en ayant retracé les différences importantes entre ces deux langages et en tenant compte du comportement spécifique de chaque langage face à certains principes qui ont en commun, nous avons décidé d'ajouter et/ou modifier quelques entités et éléments dans le méta-modèle.

Pour chaque entité ou élément à ajouter dans le méta-modèle il faut suivre plusieurs étapes, pour faciliter l'explication on supposera qu'on veut ajouter un entité dans le méta-modèle (le même procédé doit se faire pour l'ajout d'un élément). Alors, il faut ajouter une interface de cette entité dans le package ***padl.kernel*** du projet **PADL**, et il faut ajouter aussi une classe de cette entité dans le package ***padl.kernel.impl*** du projet **PADL**. Ensuite, il faut modifier l'interface **IFactory** et la classe **Factory** dans ***padl.kernel*** et ***padl.kernel.impl*** respectivement pour ainsi ajouter les méthodes qui permettront d'ajouter l'entité dans notre réceptacle **IdiomLevelModel**.

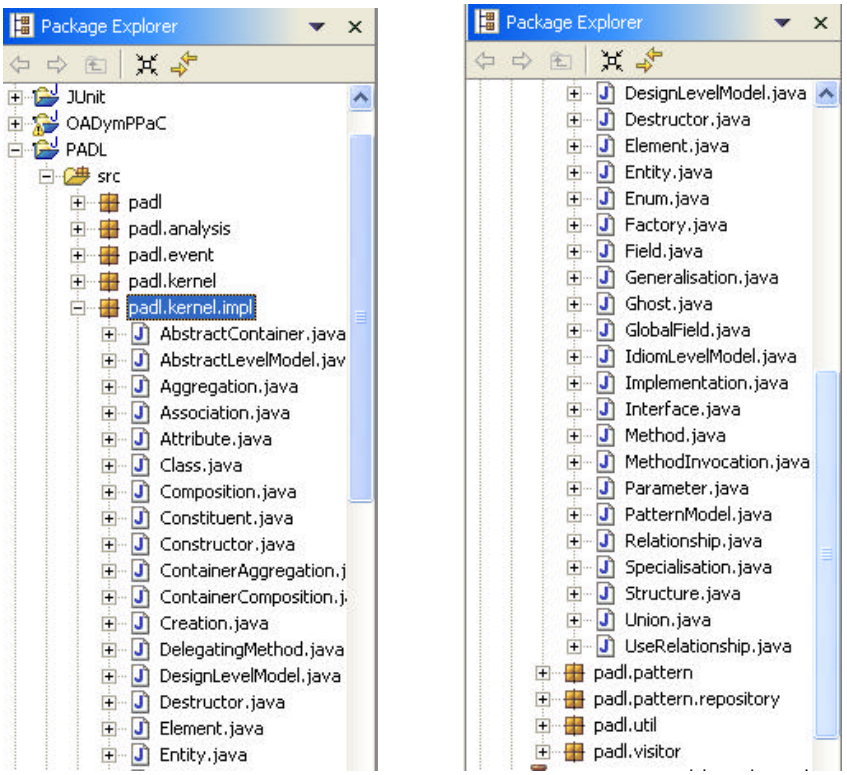

Figure 1. Ajout des structures manquantes dans le méta-modèle

Une fois terminé l'ajout de l'entité dans le méta-modèle, il faut ajouter une classe de cette entité dans le package ***ptidej.ui.kernel*** du projet **Ptidej UI**. À ce point, la démarche à suivre est différente pour le cas de l'ajout d'une entité et pour le cas de l'ajout d'un élément.

D'un côté, si on veut ajouter une entité, il faut modifier le fichier **AbstractModelGraph.java** du package ***ptidej.ui.kernel***. En effet, cette classe s'occupe de créer les entités à ajouter dans le modèle abstrait graphique. Nous pouvons trouver dans cette classe la méthode **createEntity()**, laquelle fait un « **cast** » des différentes entités. Alors, c'est dans cette méthode qu'il faut ajouter une condition pour que la méthode reconnaisse notre nouvelle entité et fasse l'ajout correspondant dans le modèle.

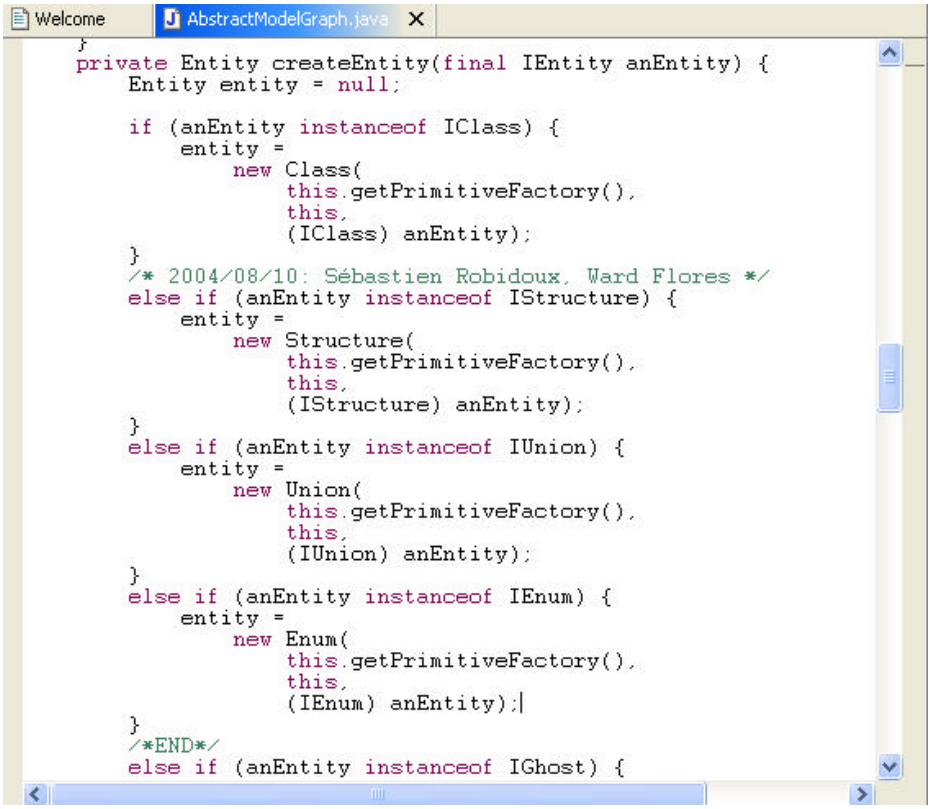

Figure 2. Ajout des conditions pour la reconnaissance des nouvelles entités dans le modèle à afficher par l'outil Ptidej UI.

Puis, d'un autre côté, si on veut ajouter un élément, il faut modifier la classe **Entity** du package ***ptidej.ui.kernel*** parce que cette classe s'occupe d'ajouter les éléments dans l'entité respective. Ensuite, ces entités seront ajoutées au modèle abstrait comme on l'a vu dans le paragraphe précédent. La classe **Entity** contient la méthode **addElement()**, laquelle fait un «**cast** » des différentes éléments pour faire après l'ajout de l'élément dans l'entité. Alors, il faut ajouter une condition dans cette méthode pour pouvoir reconnaître notre nouvel élément et pouvoir ainsi l' ajouter dans l'entité.

À continuation, la liste des toutes les entités et éléments ajoutés au méta-modèle avec une description de leur hiérarchie supérieur (*supertype hierarchy*).

1. IDestructor : Ajouté au même niveau que l'entité Constructor.

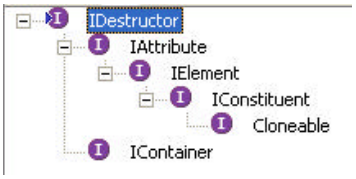

Figure 3. Hiérarchie supérieure de IDestructor.

2. IStructure : Au même niveau que l'entité Class.

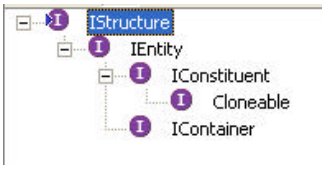

Figure 4. Hiérarchie supérieure de IStructure.

3. IUnion : Au même niveau que l'entité Class. La hiérarchie de cette entité est égale à celle de la structure

4. IEnum : Au même niveau que l'entité Class. La hiérarchie de cette entité est égale à celle de la structure.

5. IGlobalField : Au même niveau que l'entité Class, hérite des entités Entity et Field.

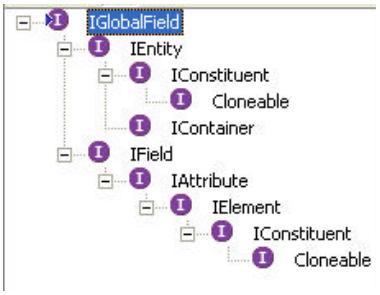

Figure 5. Hiérarchie supérieure de IGlobalField.

Quant aux classes de ces nouvelles entités qui ont été ajoutées au méta-modèle, c'est clair qu'elles suivront le même principe de leurs interfaces par rapport à la hiérarchie à utiliser lors de l'ajout au méta-modèle.

Finalement, l'outil **Ptidej** représente graphiquement chaque entité par une « boîte » contenant toute l'information associée à celle-ci. Par exemple, pour l'affichage d'une classe, nous aurons trois séparations dans notre «boîte » : la première partie sera pour le nom de la classe, la deuxième pour les champs et la troisième pour les méthodes de la classe. Or, pour faciliter la différenciation d'une classe avec une interface, le professeur Guéhéneuc ajoute le texte « interface » avec le titre de l'interface. Donc, nous avons utilisé le même procédé en ajoutant des textes indiquant si l'entité en question est un Struct, un GlobalField, etc. Pour faire cela, nous avons modifié la méthode **build()** du fichier **Entity.java** dans le package ***ptidej.ui.kernel*** du projet **Ptidej UI**.

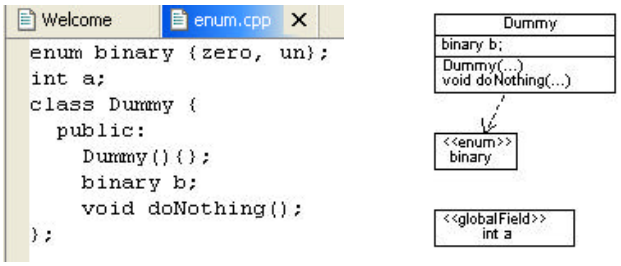

Figure 6. Affichage graphique du fichier enum.cpp avec l'outil Ptidej UI.

c) Ajout des tests JUnit

Lorsqu'on programme, il faut souvent séparer les tâches pour faciliter le travail. Ainsi, nous pouvons nous attaquer à nos objectives une à la fois. Bien que ce procédé puisse être très utile, il peut amener aussi une source d'erreurs et cela parce que, par exemple, lorsqu'on travaille sur une deuxième partie, nous pouvons commettre des erreurs qui feront « bugger » notre première partie, laquelle pourtant fonctionnait correctement.

Heureusement, l'outil JUnit nous aide avec ce problème. JUnit est une *framework* de tests que permet de construire des suites de tests qui nous permettront de vérifier à tout moment le bon fonctionnement de toutes les parties du projet. En effet, en ajoutant une série de tests unitaires pour chaque problème résolu, on peut rester avec l'esprit tranquille, car nous pourrons vérifier facilement si la modification qu'on est en train de faire nuira ou non les autre parties du projet déjà résolues. En plus, en admettant qu'une de ces modifications crée un problème, nous pourrons retracer facilement la source de l'erreur, car les tests unitaires nous indiquent où l'erreur se trouve.

Alors, nous avons ajouté le projet **PADL C++ Creator Tests**, dans lequel on trouvera quatre suites de test dans le package ***padl.test.example***. Le premier test (*TestWorld.java*) s'occupe de tester à fond le fichier **word.cpp** que nous pouvons trouver dans le dossier ***test_src***. Ce fichier **word.cpp** est très complet par rapport aux caractéristiques du langage C++. En effet, dans ce fichier, on a deux classes, une enumeration, une structure, un globalField, des constructers, des destructeurs, des champs, des méthodes et quelques relations (*useRelationship*) entre les classes. Nous avons créé des tests dans cette première suite de tests pour chacune de ces caractéristiques. Pour accomplir cet objectif, nous avons récupéré la liste des entités pour ainsi pouvoir vérifier que ces entités et leurs types d'accès correspondent à celles du fichier.

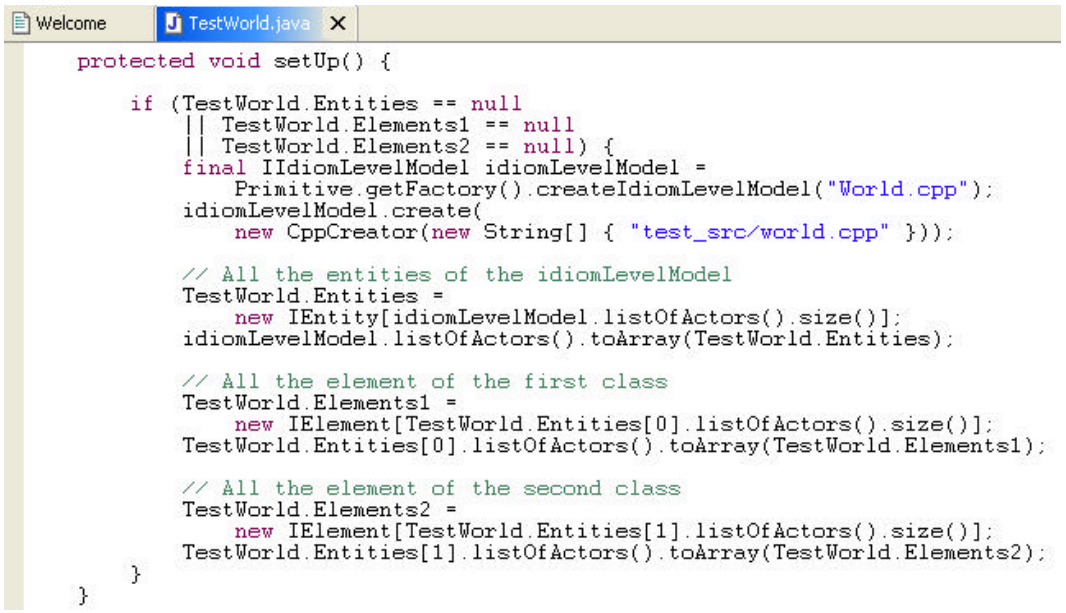

Figure 7. Nous récupérons les entités et les éléments pour chaque entité classe pour pouvoir ainsi vérifier que l'information soit correcte.

Pour l'entité globalField, nous vérifions aussi que le type de la variable soit le même. Ensuite, pour chaque entité classe, nous parcourons la liste de ces éléments pour vérifier au fur et à mesure que les constructeurs, les destructeurs (s'il y en a), les champs et les méthodes sont corrects. Bien sûr, pour chacune de ces éléments, nous comparons le nom, le type, l'accessibilité et le nombre de paramètres pour les méthodes. Finalement, nous vérifions le nombre de relations d'usage pour chaque entité.

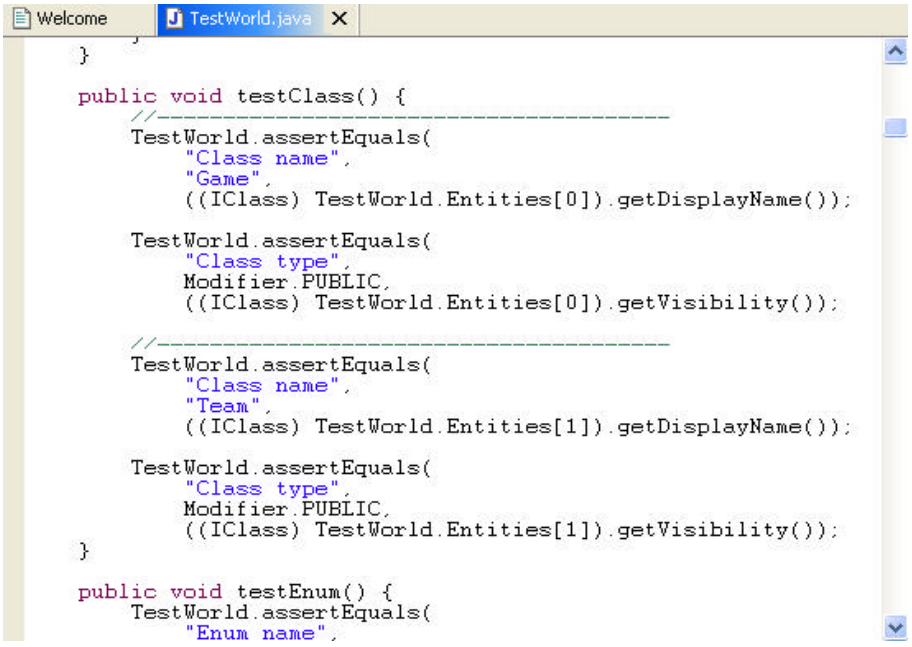

Figure 8. Tests unitaires pour vérifier les noms et types d'accessibilité des classes du fichier *world.cpp*

Quant à la deuxième suite de test, c'est un test pour vérifier l'exactitude du fichier **csegment2.cpp**. La particularité dans ce fichier est que le fichier utilise des «ghost », ce que nous n'avions pas trouvé dans le fichier **world.cpp**. Nous profitons également pour tester encore que tous les éléments des entités sont corrects. En plus, nous vérifions une fois de plus la relation d'usage (*useRelationship*) de la classe **csegment2.cpp** avec sa cible qui est dans ce cas, le *ghost*.

Notre troisième suite de test est basée sur le fichier **her.cpp**. Ce fichier nous montre un exemple très simple mais pourtant très important sur une des caractéristiques du langage C++. Nous parlons de l'héritage multiple. Ce fichier C++ est composé de trois classes où une de ces classes fait utilisation de l'héritage multiple. Alors, on vérifie pour chacune de ces entités classes, ses classes «enfants » et ses classes «parents ».

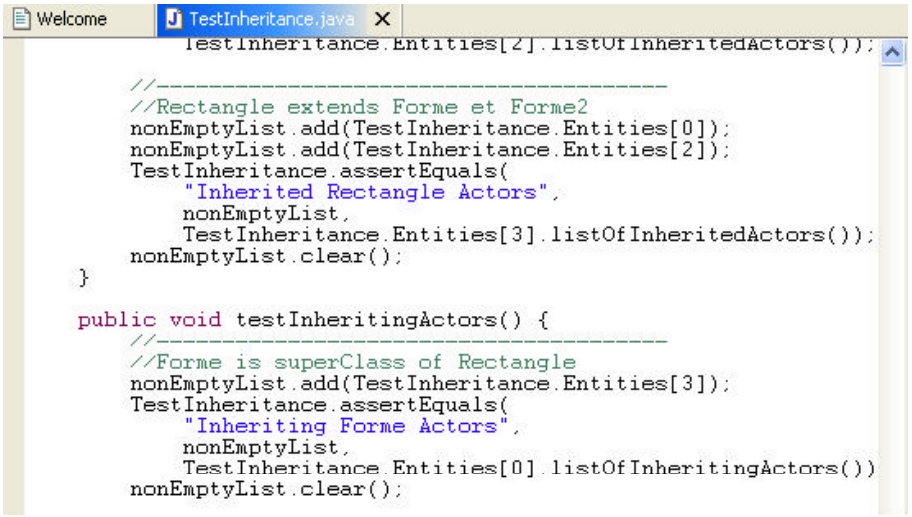

Figure 9. Nous pouvons voir dans cet image les tests faits pour vérifier que la classe **Rectangle** hérite des classes **Forme** et **Forme2** et que la classe **Forme** doit avoir comme enfant la classe **Rectangle**.

Puis, notre quatrième suite de test se base sur le fichier **Cpoint.cpp**. Dans ce fichier nous avons une classe avec des champs et des méthodes, alors dans cette suite de test nous vérifions l'exactitude des données de cette classe.

V. Discussion.

a) Problèmes rencontrés.

Le plus gros problème que nous avons rencontré dans ce projet fut la première avec la recherche d'un bon analyseur de C++. Les deux première semaines on été un peu décourageante, car nous en avions trouvé aucun qui satisfaisait au critère que nous nous étions donnés. C'est pourquoi nous avons dépassé notre échéance que nous nous étions donnés pour trouvé notre analyseur.

Heureusement, ensuite, nous n'avons pas vraiment rencontré de problème avec ce projet. Tout à semblé bien se déroulé, ce qui nous a permis de respecter notre échéance malgré le retard accumulé après la première partie. Le seul petit problème que nous avons rencontré fut d'adapter l'analyseur créé par « JavaCC » pour qu'il nous permettent d'analyser des projet incomplet. (voir « Grammaire C++ pour « JavaCC » » plus haut.).

b) Améliorations possibles/futures.

Bien sur, comme tout projet informatique, il y a des améliorations qui pourront être effectué sur notre projet. Comme nous l'avons toujours appris en informatique, aucun projet n'est jamais terminé. Voici une petite liste de ce que nous croyons être les améliorations principales :

- Spécification des liens entre les entités :

Pour l'instant, lorsque nous créons des liens entre les entités, il s'agit du lien de type « UseRelationship ». Mais dans le méta-modèle, plusieurs type de liens existe, comme des liens d'agrégation, d'association, de contenants, etc. Donc une amélioration serait de différentier ces liens au lieu de toujours créer des liens « UseRelationship ».

- Modernisation de la grammaire C++ :

Pour l'instant, la grammaire fonctionne. Mais il se peut très bien que des modifications soit fait sur le langage C++ d'ici là. Comme par exemple, la commande « using namespace std; » n'est pas valide avec notre grammaire. Et bien sur, aucune grammaire n'est toujours parfaite pour un langage aussi complexe que C++.

c) Évolution personnelles

Ce projet fut très intéressant pour nous. Nous avons découvert un peu plus en profondeur un domaine que nous n'avions que vu que rapidement dans nos cours. La compilation est un domaine qui nous intéresse beaucoup et ce projet nous a permis d'approfondir nos connaissance en ce domaine. Bien sur, nous avons pu aussi apprendre à utilisé un merveilleux environnement de travail qui est « Eclipse » et « CVS ». Bien coder en « Java » ne sera jamais pareil sans «Eclipse ». Et encore plus, nous avons pu renforcer nos notions sur le langage de programmation C++. Il n'y a pas de meilleur façon de bien apprendre un langage qu'en apprenant presque par cœur sa grammaire.

d) Remerciements.

- Premièrement, nous voulons remercier notre directeur de projet, M. Yann-Gaël Guéhéneuc. Grâce à son aide et à ses nombreux emails, même lorsqu'il était à l'autre bout du monde, en vacances loin de l'université. Aussi pour sa bonne humeur et son intérêt profond dans le projet qui nous motivait à chaque fois. Il y a aussi tout le code du

programme qui était déjà fourni ainsi que les nombreux exemples avec d'autre langage de programmation qui nous ont grandement aidé pour le bon déroulement du projet.

- Nous voulons aussi remercier les concepteurs d' « Eclipse » pour avoir créé un outil qui réduit énormément le temps de conception de tout projet «Java ».

- Nous voulons aussi remercier toute personne impliqué de proche ou de loin, télépathiquement ou autre dans ce projet. Tout spécialement notre ami à tous, Duc-Loc, pour avoir amener un peu d'animation dans le local informatique et pour toutes les invitations à dîner qu'il a refusé, ou plutôt oublié!

VI. Références.

ANTLR	http://www.antlr.org/
CINT	http://root.cern.ch/root/Cint.html
CoCoLab	http://www.cocolab.de/
CppCC	http://cppcc.sourceforge.net/
FOG	http://www.computing.surrey.ac.uk/research/dsrg/fog/
GCC	http://gcc.gnu.org/
JavaCC	https://javacc.dev.java.net/
LEX/YACC	http://dinosaur.compilertools.net/
	http://www.bykeyword.com/pages/detail8/download-8937.html
	http://howtos.linuxbroker.com/Lex-YACC-HOWTO.html
PCCTS	http://www.empathy.com/pccts/
	http://www.polhode.com/pccts.html
PMD	http://pmd.sourceforge.net/
PUMA	http://ivs.cs.uni-magdeburg.de/~puma/
SAGE++	http://www.extreme.indiana.edu/sage/
SLK	http://home.earthlink.net/~slkpg/
SPIRIT	http://spirit.sourceforge.net/