# Rapport du projet du cours Réingénierie du Logiciel

## Sigle : INF6302


## Aymen Zalila ; Matricule 1746279

## E-mail : zalila.aymen@polymtl.ca

## Chargé de laboratoire: Thierry Lavoie



**Résumé**

Ce document décrit le travail qui j'ai réalisé pour le compte du laboratoire du cours « Réingénierie du Logiciel » INF6302 au cours de la session d'hiver 2015 à l'école Polytechnique de Montréal. Le travail consiste à développer un système d'analyse de programmes à travers 5 laboratoires et en appliquant les notions de métriques, analyses syntaxiques, analyses de dépendances de contrôles , de dominance et de dépendances de programme apprises durant des séances de cours en utilisant les outils comme le diagramme de classes et les graphes de flux de contrôle.


**Introduction**

Dans ce travail, nous énonçons la demande du chargé de laboratoire pour chaque TP, nous décrivons notre réalisation détaillée pour ces requis, les résultats obtenus et les difficultés rencontrées.

Ce document se compose de cinq sections, chacune décrivant l'un des laboratoires réalisés. Le premier consiste à extraire des métriques sur les méthodes du programme Tomcat 7 de Apache. Le deuxième nous demande de générer le diagramme de classe détaillée de ce même programme. Le TP suivant a pour objectif de générer le graphe de flux de contrôle (CFG) des méthodes d'un programme. Le quatrième laboratoire consiste à créer les arbres de dominateurs et post-dominateurs immédiats et à récupérer les définitions valides du programme WordCount.java. Et le dernier suggère de dessiner le graphe de dépendance du programme et de réaliser le backword slicing.

**TP 1 : Extraction de métriques**

**Enoncé :**

*« Le premier TP consiste à compiler la grammaire Java écrite en JavaCC fournie sur le site Moodle. L'étudiant doit ensuite écrire un visiteur pour extraire les métriques de chaque méthode d'un programme Java. Le travail doit utiliser les sources de Tomcat 7. Le programme doit écrire les métriques dans un fichier en format texte qui doit respecter le format suivant:*

*ID; NOM_DU_FICHIER; NOM_DE_LA_CLASSE; NOM_DE_LA_METHODE; #IF; #WHILE; #BREAK; #VARIABLES_LOCALES (0;Exemple.java;Exemple;foo;1;1;0;3) »*

**Réalisation :**

Nous commençons par télécharger les fichiers relatifs au TP sur Moodle :

-Grammaire Java pour javacc (java1_7.jjt)

-JavaCC (javacc-5.0.tar)

Nous ajoutons, ensuite, JavaCC aux variables d'environnements et exécutons les commandes suivantes :

-« javacc-5.0/bin/jjtree java1_7.jjt » pour créer les fichiers '.java' du programme,

-« javacc java1_7.jj » pour générer le parseur,

-« javac *.java » crée la Classe JavaParser1_7.java,

-« jjtree java1_7.jjt »pour reconstruire les fichiers '.java' et créer le fichier de grammaire 'Java1_7.jj',

-« javacc java1_7.jj » reconstruit la parseur,

-« javac *.java »,

-« java JavaParser1_7 »,

-« java JavaParser1_7 CompilationUnit.java » le visiteur 'JavaParser1_7Visitor.java' est prêt à l'emploi.

Nous créons un nouveau répertoire « src/main », y plaçons le fichier « JavaParser1_7.java » qui contient le 'main' de notre projet, et créons une classe « VisitorImpl » qui implémente « JavaParser1_7Visitor » (il sera fait de même pour le reste des visiteurs créé au cours des TPs suivants). Toutes les autres classes générées sont placées dans le paquet « src/parser ». Un répertoire « src/main/test » est créé pour contenir les fichiers que nous utilisons pour les tests (le même répertoire sera utilisé pour tous les TPs). Et enfin un répertoire « src/main/methods » pour la structure de données relative à ce premier TP.

Nous implémentons aussi une méthode « write(String fileName, Sting s) » qui écrit le texte « s » passé en paramètre dans un fichier du nom « fileName ». Nous utiliserons cette même méthode toutes les fois que nous aurons besoin d'écrire sur un fichier durant tous les TPs.

L'énoncé du laboratoire demande de récupérer les données suivantes : le nom du fichier, le nom de la classe, le nom de la méthode, et le nombre de if, while, break et de variables locales qu'elle contient. Pour répondre à notre besoin, une structure contenant tous ces paramètres est requise. Nous créons la classe « method.java » décrite dans la Figure 1 :

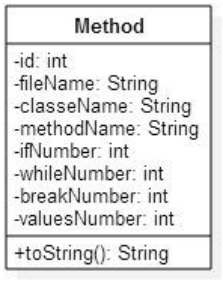


**Figure 1: Diagramme de classes "main.methods"**

Pour comprendre le comportement du visiteur, nous créons une classe « VisitorTest » et implémentons le code suivant dans toutes ses méthodes :

```
System.out.println(node.getClass() + " = " + node.jjtGetFirstToken() + " " + node.jjtGetLastToken() + "\n");
node.childrenAccept(this, data);
return null;
```

-« `node.getClass()` » : récupère le type de la classe,

-« `node.jjtGetFirstToken()` » : récupère le premier jeton du nœud,

-« `node.jjtGetLastToken()` » : récupère le dernier jeton du nœud,

-« `node.childrenAccept(this, data)` » : permet de visiter les fils.

Le résultat de cette implémentation nous donne une trace des appels de fonctions qui est réalisée lors d'une exécution précise, avec les données contenus dans chaque nœud.

Nous ajoutons à notre visiteur deux variables :

-Method method : qui est la structure de données,

-int number : qui décrit l'ID de la méthode en cours.

La structure du visiteur n'est pas vraiment utilisée dans ce TP, nous récupérons seulement les données dont nous avons besoin durant le parcours. Au nœud :

-« Identifier » : nous vérifions si le nœud parent est, soit un « ConstructorDecl », soit un « MethodOrFieldDecl », soit un « VoidMethodDecl » ou autre chose. Si l'un des trois est verifié, nous instancions de nouveau « method », y ajoutons les noms de classe et de fichier et incrémentons « number »,

-« IfStatement » nous incrémentons la valeur « ifNumber »,

-« WhileStatement » nous incrémentons la valeur « whileNumber »,

-« BreakStatement » nous incrémentons la valeur « breakNumber »,

-« LocalVariableDeclarationStatement » nous incrémentons la valeur « valuesNumber ».

-« ConstructorDecl », « MethodOrFieldDecl », ou « VoidMethodDecl » nous écrivons les valeurs de « method » dans le fichier (méthode write()).

**Difficultés rencontrées :**

La difficulté principale pour ce TP est la compréhension de la grammaire du

parseur donné par le chargé de laboratoire.

**TP 2 : Génération du diagramme de classes**

**Enoncé :**

*« Le TP2 consiste à extraire le diagramme de classe du système Tomcat 7.0 (voir TP1) à l'aide d'un nouveau visiteur. Votre diagramme de classe UML devra représenter les classes avec leurs membres et leurs méthodes ainsi que les relations d'héritages et les relations de composition. La visibilité ainsi que les types des membres, des méthodes et de leurs paramètres sont exigés dans la représentation de votre diagramme.*

*On vous suggère d'utiliser GraphViz pour réaliser la sortie graphique de votre diagramme. »*

**Réalisation :**

Nous commençons par créer notre structure de données par rapport aux demandes du TP. Le but est de générer le diagramme de classes d'un programme. Il nous faut donc, pour chaque classe son nom, la liste de ses méthodes et attributs ainsi que le paquet qui la contient, sa classe mère et la liste de ses imports. La Figure 2 représente cette structure.

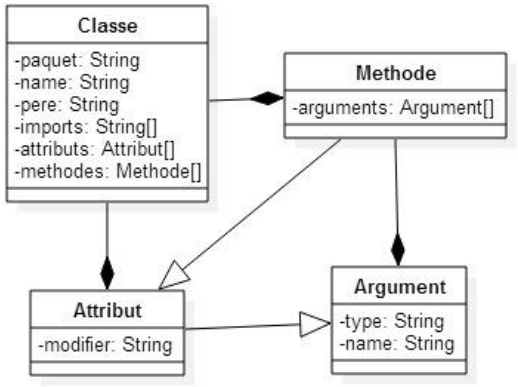


**Figure 2: Diagramme de classes "main.diagram"**

Nous étudions ensuite le langage GraphViz, et nous modifions les méthodes toString()de toutes les classes de notre structure pour que l'appel toString() de la classe « Classe » écrive un fichier sous une grammaire correcte de GraphViz.

Pour ce deuxième TP, nous ajoutons un nouveau visiteur « VisitorDiagram », comme demander par l'énoncé. Dans cette version du visiteur, nous lui ajoutons deux attributs : un attribut de type « Classe » qui représente la classe explorée actuellement, et un attribut « Vector<Classe> » qui contient toutes les classes déjà parcourues.

Au commencement, l'algorithme va visiter le nœud « CompilationUnit », là il crée une nouvelle « Classe » puis visite le reste des nœuds, et au retour la classe est ajoutée au vecteur de classes (cette méthode s'avèrera erronée pour les cas où il y'a des classes internes). Dans les nœuds « KeywordModifier » et « ClassBodyDeclaration », il faut les initialisations des attributs « imports » et « attribut », respectivement. Dans

« MethodOrFieldDecl », il ajoute le type du dernier attribut récupéré, et dans « FormalParameterDecls » l'algorithme récupère les arguments d'une méthode. C'est dans le nœud « Identifier » que le reste des données seront récupérées. Ici, toutes les données sont sélectionnées par un switch sur les parents et par des conditions sur les parents des parents d'un nœud. Tous les nœuds restants ne feront que propager le visiteur à leurs enfants.

A la fin la classe « JavaParser1_7 » fait appel à la classe « main.diagram.Graphe » en passant en paramètre le vecteur de Classes. Cette classe écrit les données des classes dans un fichier « nom_du_programme.dot » sous la grammaire GraphViz.

Nous exécutons ensuite ce ficher avec la commande « dot –Tps nom_du_fichier.dot > out.ps ». out.ps étant le nom du fichier en sortie, l'extension peut être les formats .png, .jpeg ou .pdf.

Suite à une remarque du chargé de laboratoire sur la non-utilisation du passage de la structure en paramètre dans la visite des nœuds, nous décidons de créer un nouveau visiteur « VisitorDiag » pour commencer à utiliser cette propriété.

Ce nouveau visiteur, propage l'extraction des données sur plusieurs autres nœuds. Le nœud « Identifier » dans ce cas, ne fait que retourner la valeur de son premier jeton. Les autres nœuds décrits dans le visiteur « VisitorDiagram » ont exactement le même fonctionnement. Les nouveaux nœuds ajoutés sont :

-« QualifiedIdentifier » : récupère le package,

-« ImportDeclaration » : récupère les imports,

-« NormalClassDeclaration » : récupère le nom de la classe, et crée un nouvelle classe dans le cas où il y'a une classe interne (ceci règle le problème rencontré précédemment du cas des classes internes),

-« ReferenceType » : récupère la classe mère,

-« VoidMethodDecl » : instancie une nouvelle méthode, récupère son nom, son 'Modifier', met son type à 'void' et l ajoute à la liste des méthodes,

-« MethodOrFieldDecl » : crée un nouvel attribut ou récupère les données relatives à une méthode avec un retour,

-« VariableDeclaratorId » : récupère les données relatives aux arguments et les ajoute à la méthode concernée,

-« VariableDeclarator » : récupère les données d'un attribut.

L'algorithme a le même fonctionnement à la fin de l'exécution, en faisant appel à la classe « Graphe ».

**Difficultés rencontrées :**

-Comprendre le fonctionnement du visiteur,

-Identifier l'ordre de passage de visiteur par les nœuds, et le passage de données entre eux

**TP 3 : Génération des Graphes du Flux de Contrôle**

**Enoncé :**

*« Le TP3 consiste à extraire le CFG intra-procédural de Tomcat7.0. Vous pouvez limiter votre CFG aux structures If, While, Do, For et Switch. On vous demande également de traiter de manière appropriée les continue, break et return. Les autres énoncés peuvent être approximés dans le CFG, mais vous devez préserver le plus d'information possible, en faisant des hypothèses adéquates. Toutes vos fonctions devraient commencer par un nœud "Entry" et se terminer par un nœud "Exit"; vous devez rajouter ces nœuds dans votre CFG, même s'ils ne sont pas présents dans l'AST.*

*Le CFG doit être calculé en mémoire dans une structure de graphe et ensuite imprimé en format GraphViz, comme au TP2. »*

**Réalisation :**

Comme pour les autres TPs, nous commençons par mettre en place notre structure de données. Elle est composée d'une façon assez simple, une classe « Projet » qui contient une liste de classe « Classe » qui renferme les listes de ses attributs et de ses méthodes. La classe « Methode » contient à son tour la liste des « Statements » qui inclue une liste de « Statements » qui décrit ses fils « next », ainsi que un attribut « Expression ». Ce dernier est décrit par une partie gauche de l'expression, une partie droite et l'opérateur. La structure décrite ci-dessous, est implémentée dans le répertoire « src/main/cfg ».

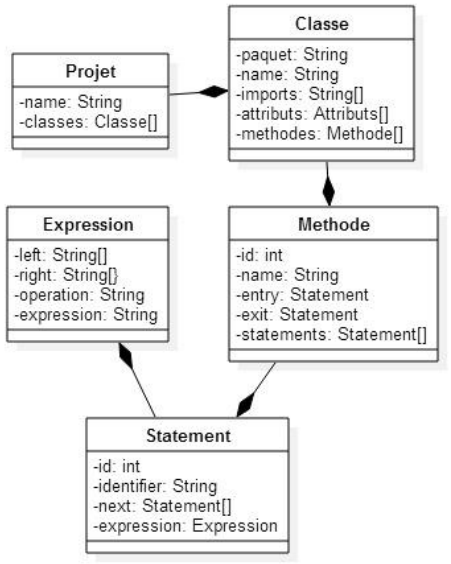


**Figure 3: Diagramme de classes "main.cfg"**

Cette structure est si détaillée pour avoir le plus d'information que possible comme demandé dans l'énoncé, et pour pouvoir répondre aux exigences des prochains TPs en même temps.

Comme pour le TP2, les méthodes toString() des classes ont été écrite en fonction de la grammaire GraphViz.

Nous créons un nouveau visiteur pour ce travail, nommé « VisitorCFG ».

Dans le main, la classe « Projet » est instanciée puis passée en paramètre au vecteur. L'algorithme fonctionne comme précédemment, à chaque nœud, il récupère les informations dont il a besoin :

-« CompilationUnit » : crée un nouveau Projet et le passe en paramètre à ses enfants,

-« NormalClassDeclaration » : crée une nouvelle Classe, récupère son nom et la passe en paramètre à ses enfants, au retour la classe est ajoutée au projet,

-« Type » : crée une expression et la passe en paramètre à ses enfants,

-« ReferenceType » : récupère le type des variables locales,

-« StaticInitBlock » : crée une méthode pour les blocks et l'ajoute à la liste des méthodes,

-« VoidMethodDecl », « ConstructorDecl » et « MethodOrFieldDecl » : crée une méthode, récupère son nom, met le Modifier adéquat, et la passe en paramètre à ses enfants,

-« VariableDeclaratorId » : récupère la partie gauche d'une expression,

-« VariableDeclarator », « VariableDeclaratorRest » : récupère l'expression sous forme de String,

-« BlockStatement » : lie les « Statements » qui doivent l'être au Statement "retrun",

- « LocalVariableDeclarationStatement » : crée un « Statement » pour les déclarations de variables locales,

-« Statement » : crée un « Statement », le passe en paramètre à ses enfants et ajoute le Statement suivant au "next" s'il y'a lieu,

- «IdentifierStatement », « StatementExpression », « IfStatement », « ElseStatement », « NoElseStatement », « SwitchStatement », « WhileStatement », « DoStatement », « ForStatement », « LabeledBreak », « UnlabeledBreak », « LabeledContinue », « UnlabeledContinue » et « ReturnStatement » : créent un « Statement » et récupère les données relatives à son type,

-« SwitchLabel » : récupère la variable sur laquelle va être exécutée le switch,

-« ForVarControlRest » : récupère l'expression de condition de la boucle "for",

-« ForVariableDeclaratorsRest » : termine l'expression de la boucle,

-« Expression », « Literal » et leurs enfants : récupèrent une à une les parties composant l'expression,

-« ParExpression » : récupère la condition du "if", du "while" et du "do",

-« Arguments » : récupère la partie droite de l'expression,

-« Creator » et ses enfants : récupère l'expression d'instanciation d'un Objet.

A la fin de l'algorithme, le « main » fait appel à la classe « main.cfg.Graphe » qui génère le fichier.dot en passant le projet.

**Résultat :**

Nous exécutons l'algorithme sur un programme test « WordCount.java » envoyé pas le chargé de laboratoire. Le résultat de l'exécution donne le graphe ci-dessous :

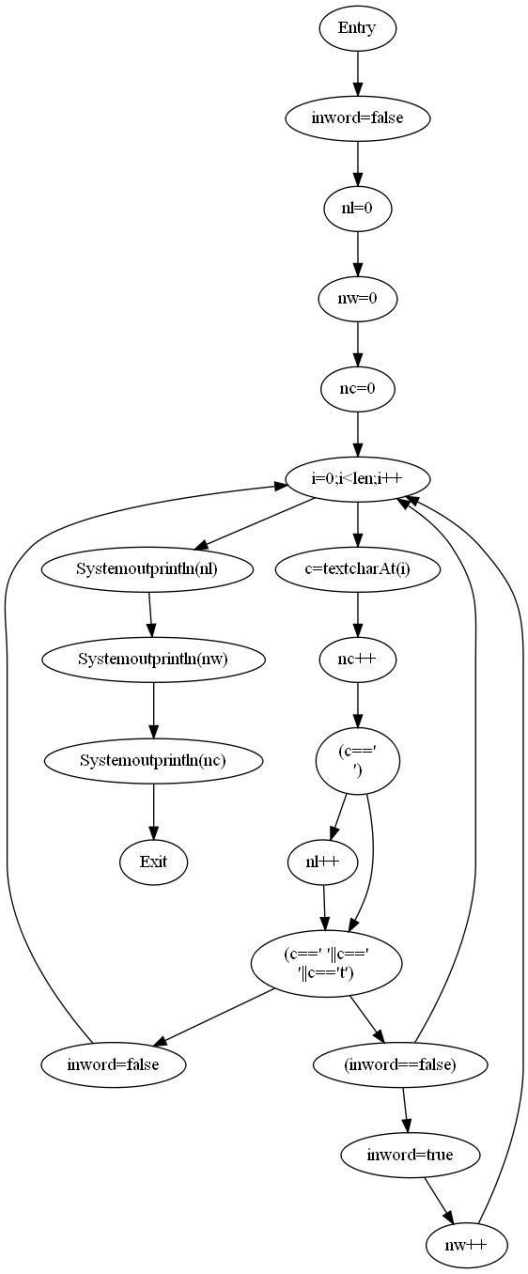


**Figure 4: CFG "WordCount.java"**

**Difficultés rencontrées :**

-identifier la forme des différents CFG : if, while, do et switch

-identifier la donnée adéquate à passer en paramètre.

-suivre le visiteur dans son passage par la totalité des nœuds.

**TP 4 : Dominateur, Post-Dominateur, et Définitions Valides**

**Enoncé :**

*« Vous devez extraire l'arbre des dominateurs immédiats et des post-dominateurs immédiats ainsi qu'effectuer le calcul des définitions valides. »*

**Réalisation :**

Pour ce TP, nous n'avons pas besoin de créer, ni de structure de données, ni de visiteur, vu que toutes les données récupérées par « VisitorCFG » suffisent à générer l'arbre des dominateurs immédiats, celui des post-dominateurs immédiats et les définitions valides.

Nous commençons par la partie Dominateur. Pour cela nous créons une nouvelle classe à cet effet, la classe « Dominator » sous le répertoire « src/main/dominator » (nous travaillons dans ce même répertoire pour le reste de ce TP). Dans « Dominator », l'algorithme parcoure le CFG, nœud par nœud, pour chacun d'entre eux, il récupère tous ses dominateurs. Ensuite, il utilise la méthode récursive « findDom() » dans le cas où le nœud a plusieurs branches d'entrée, qui calcul le

dominateur immédiat sur toutes ces branches. Enfin, le toString() affiche un résultat sous la grammaire GraphViz, qui exécuté donne le résultat Figure 5 sur le programme WordCount.java.

Pour le Post-Dominateur, l'algorithme est similaire à celui des dominateurs, à part qu'au début nous inversons les arcs de notre CFG. Le résultat sur WordCount.java est représenté sur la Figure 6 (dans l'annexe).

L'algorithme des définitions valides est composé de deux parties. La première parcoure le CFG en cherchant toutes les définitions de chaque variable :

```
c [6,] nc [4,7,] nw [3,14,] i [5,] inword [1,11,13,] nl [2,9,]
```

La deuxième parcoure le CFG pour chacune des paires (variables, nœud de définition), en ajoutant les nœuds où la définition est toujours valide.

```
(4, nc) = [4, 5, 6, 7, 15, 16, 17, 18]

(2, nl) = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]

(5, i) = [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]

(14, nw) = [14, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 16, 17, 18]

(6, c) = [6, 7, 8, 9, 10, 11, 5, 15, 16, 17, 18, 12, 13, 14]

(3, nw) = [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]

(7, nc) = [7, 8, 9, 10, 11, 5, 6, 15, 16, 17, 18, 12, 13, 14]

(9, nl) = [9, 10, 11, 5, 6, 7, 8, 15, 16, 17, 18, 12, 13, 14]

(11, inword) = [11, 5, 6, 7, 8, 9, 10, 12, 13, 15, 16, 17, 18]

(1, inword) = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 16, 17, 18]

(13, inword) = [13, 14, 5, 6, 7, 8, 9, 10, 11, 12, 15, 16, 17, 18]
```

Les résultats ci-dessus sont extraits de WordCount.java.

**Difficultés rencontrées :**

-la mise en place d'une structure sur laquelle nous pouvons réaliser toutes nos manipulations.

**TP 5 : Graphe de Dépendances du Programme et Slicing**

**Enoncé :**

*« Vous devez générer le graphe de dépendances du programme (Graphe CD + Graphe DD) et créer le Backword-Slicing (au moins sur la paire (16, nw)). »*

**Réalisation :**

Dans ce dernier TP, nous décidons de suivre la démarche suivie dans le cours. Nous commençons par générer le graphe de l'union entre l'arbre des dominateurs immédiats et celui des post-dominateurs, puis le graphe des super blocs (classe SuperBlocs), suivis de l'analyse des

dépendances de contrôle et la construction des dépendances pour générer le graphe de dépendance de contrôle (la classe GrapheCD). Ensuite, nous cherchons les paires (définition, utilisation) pour générer le graphe de dépendance de données (la classe GrapheDD). Enfin, l'union des deux derniers graphes nous donne le graphe de dépendance du programme dans la classe GrapheP.( voir Figure 7: Graphe de dépendances du programme dans l'annexe)

La dernière partie fait le backword-slicing sur la paire (nœud, variable) passée en paramètre de la méthode « slice() » dans la classe « Slicing ». Elle est réalisée en parcourant les branches des graphes CD et DD en même temps en utilisant le tableau « buffer » qui contient les nœuds à parcourir et « slice » qui contient les nœuds déjà visités :

```
Slice(16,nw) = [16, -1, 14, 3, 12, 10, 11, 1, 13, 5, 6]
```

Le travail est réalisé dans le répertoire « src/main/slicing ».

**Difficultés rencontrées :**

Dans ce TP, nous n'avons pas vraiment rencontré de difficultés vu que toutes les données dont nous avons besoin sont déjà présentes après l'extraction du TP3 et le traitement dans le TP4.

**Conclusion**

Au terme de cette session hiver 2015, nous jugeons avoir répondu à tous les requis des cinq laboratoires. Le premier TP récupère toutes les métriques demandées en un temps (varie entre 20 et 40 secondes) assez satisfaisant sur Tomcat. Pour le second, le diagramme de classes est complet. Le troisième laboratoire nous donne un CFG détaillé avec la totalité des « Statements ». L'avant dernier TP, remis en retard, répond aussi aux requis et affiche les arbres de dominance et post-dominance et les définitions valides. Et le dernier, génére le graphe de dépendances du programme, le slicing et même toutes les étapes intermédiaires. Nous avons réalisé plusieurs jeux de tests sur les différents TPs et les résultats obtenus répondent à nos attentes.

Nous aurions aimé avoir plus de temps pour retourner un code plus clair, commenté et plus ordonné.

**Référence**

[Moodle]

https://moodle.polymtl.ca/course/view.php?id=1219

[JavaCC] https://javacc.java.net/

[Tomcat 7.52]

https://archive.apache.org/dist/tomcat/tomcat-7/v7.0.52/bin/

[GraphViz]

http://www.ffnn.nl/pages/articles/media/uml-diagrams-using-graphviz-dot.php

# Annexe

Le programme WordCount.java utilisé dans les TP 3 à 5 :

```
package main.test;

class WordCount {

	static final String text = "Cycle de re-ingŽnierie : re-
ingenierie, restructuration, \n" +
	"retro-ingenierie, recuperation de la conception \n" +
	"(Design Recovery) et re-documentation.\n";

	static final int len = text.length();

	static public void main (String args[]) {
		boolean inword = false;
		int nl = 0;
		int nw = 0;
		int nc = 0;

		for (int i = 0; i<len; i++) {
			final char c = text.charAt(i);
			nc++;
			if( c == '\n')
				nl++;
			if( c == ' ' || c == '\n' || c == '\t')
				inword = false;
			else if( inword == false ){
				inword = true;
				nw++;
			}
		}

		System.out.println ( nl);
		System.out.println ( nw);
		System.out.println ( nc);
	}
}
```

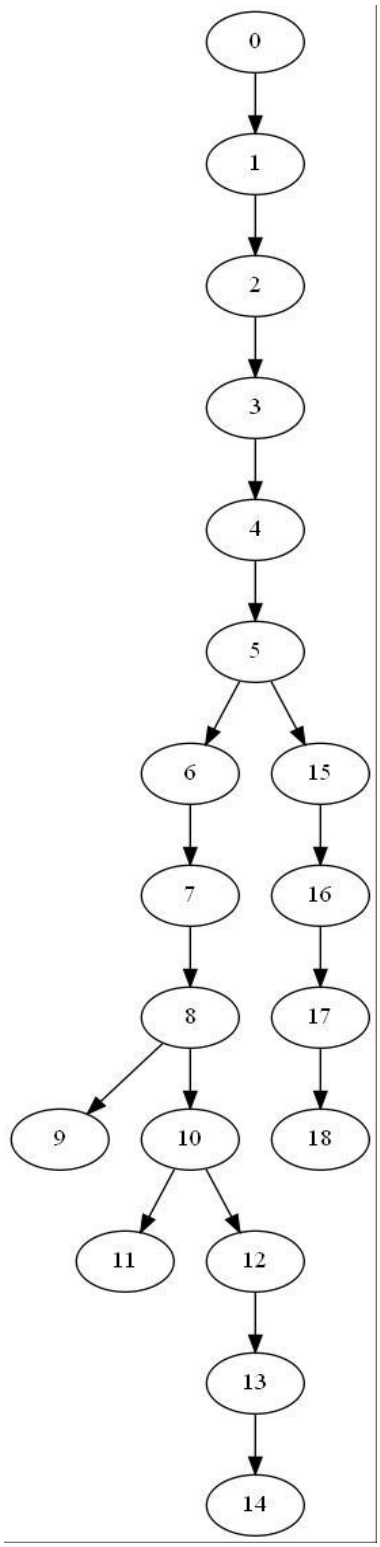


**Figure 5: Arbre de dominateur**

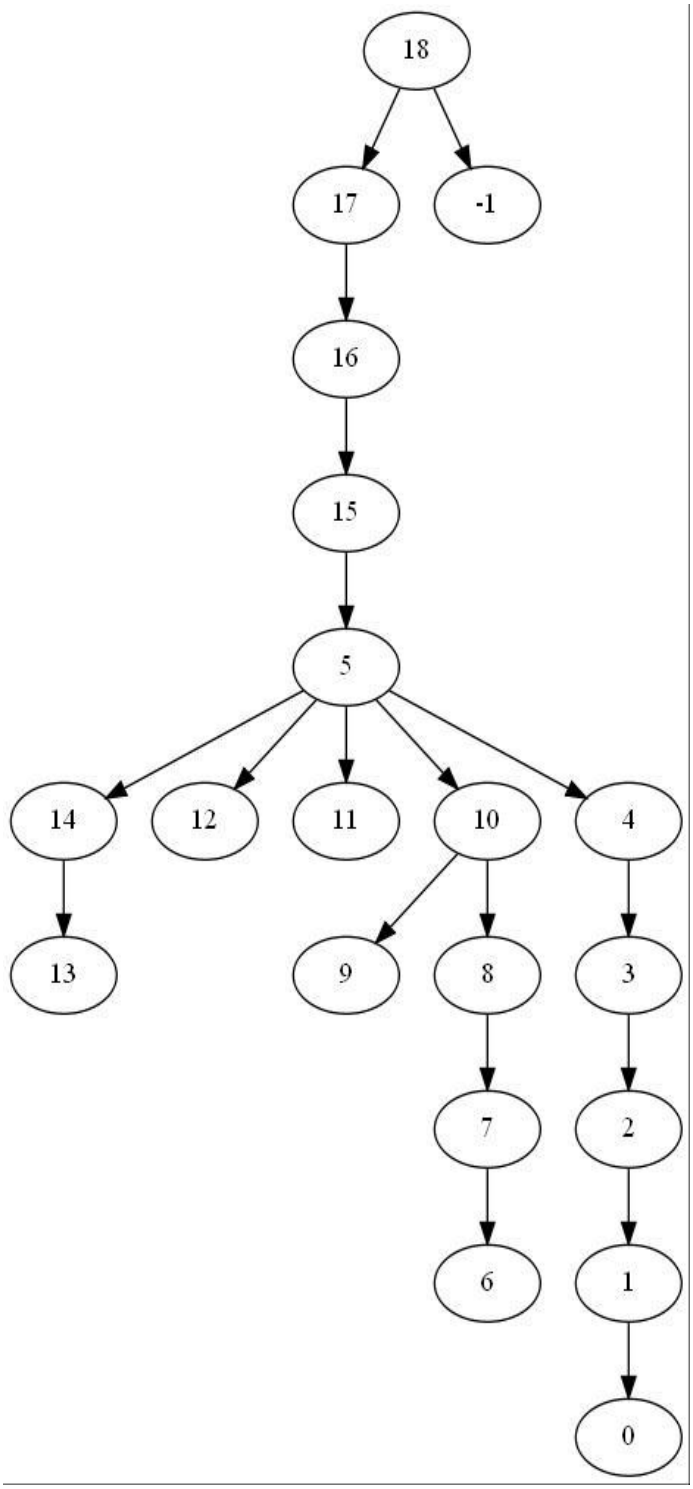


**Figure 6: Arbre de post-dominateur**

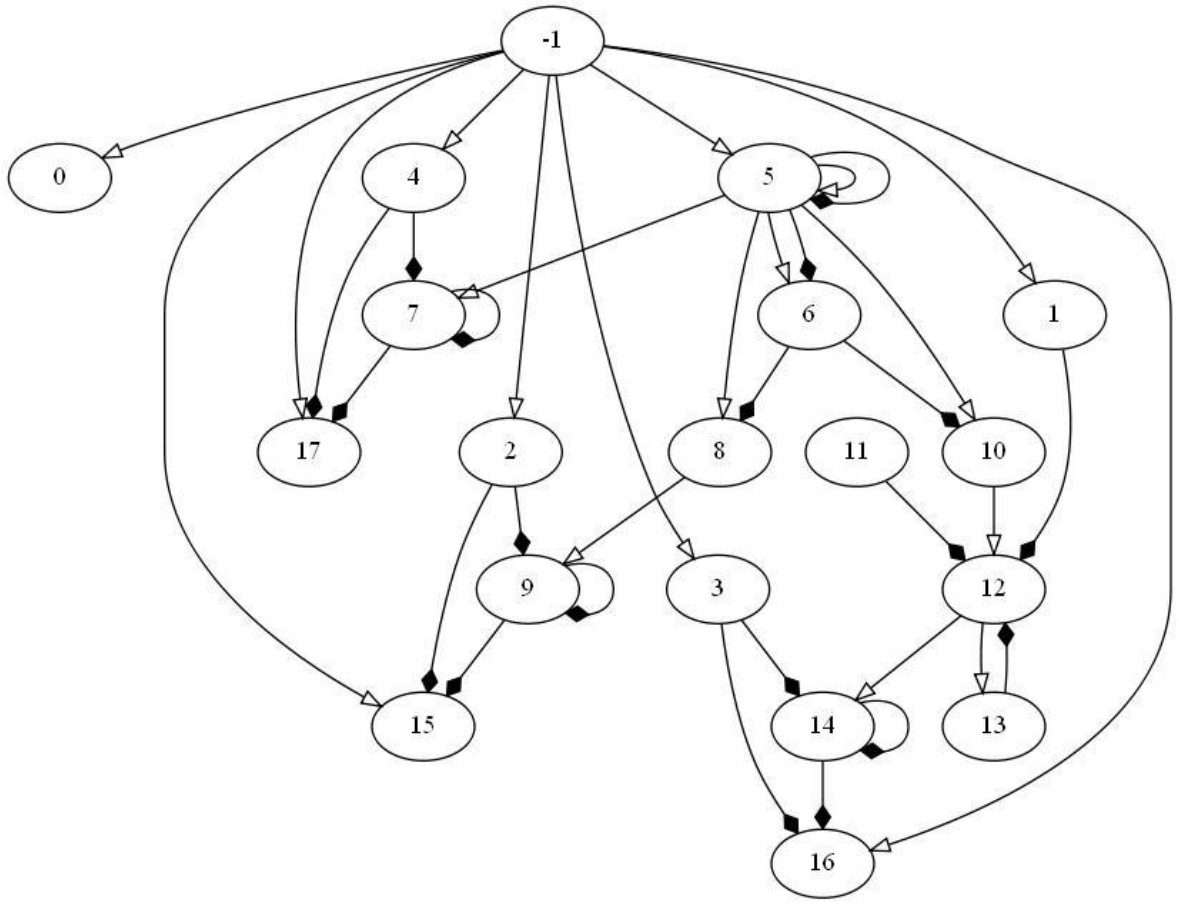


**Figure 7: Graphe de dépendances du programme**

Dans le graphe de la Figure 7, les branches avec des flèches vides sont les arcs du graphe CD (dépendance de contrôle) et celles plaines sont les arcs du graphe DD (dépendance de données).