Thomas Neyraut
1812273

# LOG6302 : Ré-ingénierie du logiciel

## Rapport d'avancement final

### INTRODUCTION

Ce document présente le projet que nous avons mené au cours de cette session d'hiver 2016 dans le cadre du cours LOG6302. L'objectif final du projet était de développer un slicer à l'aide du logiciel Javacc. Pour atteindre cette fonctionnalité final (backward et forward slicing), nous avons implémenté plusieurs fonctionnalités intermédiaires. Vous trouverez le code source de notre projet au lien GitHub suivant : https://github.com/tneyraut/JavaParser.

Dans une première partie, nous présentons comment utiliser le programme ainsi que toutes ses fonctionnalités. Dans une seconde partie, l'ensemble de la méthodologie est présenté. Dans une troisième partie, nous présentons différents résultats types que nous avons obtenus, ainsi que les performances de notre programme. Enfin dans une quatrième et dernière partie, nous présentons l'ensemble des limites et critiques que nous pouvons signifier à l'égard de notre programme.

### I. UTILISATION ET FONCTIONNALITES DU PROGRAMME

Notre programme propose les fonctionnalités suivantes pour n'importe quel programme Java :

- Calcul de métriques,
- Génération du diagramme de classes,
- Génération du diagramme de flux,
- Génération du diagramme des dominateurs,
- Génération du diagramme des post-dominateurs,
- Génération du diagramme des définitions valides,
- Génération du diagramme des dépendances de données,
- Génération du diagramme des dépendances de contrôle,
- Calcul du backward slicing,
- Calcul du forward slicing.

Les obtentions des métriques, des différents diagrammes et des résultats d'un slicing sont toutes indépendantes. Il est donc nécessaire d'exécuter le programme plusieurs fois pour obtenir plusieurs diagrammes ou résultats différents. Les commandes permettant d'exécuter notre programme sont les suivantes :

- « java Javaparser1_7 path_name_file.java option »,
- « java Javaparser1_7 @path_name_fileList.txt option ».

La première commande permet d'analyser un fichier java en particulier alors que la seconde permet d'analyser l'ensemble des fichiers java dont le « path name » est présent dans le fichier .txt fournit en paramètre (un path name par ligne dans le fichier .txt). Le second paramètre « option » est un entier entre 0 et 9 permettant d'exécuter une analyse en particulier. Si ce paramètre est absent, le programme exécute par défaut le calcul des métriques. Pour obtenir l'ensemble des commandes admisses par notre programme et leurs effets, il suffit d'exécuter la commande suivante : « java Javaparser1_7 help ».

De plus pour aider l'utilisateur, nous avons implémenté un petit programme java permettant de générer un fichier .txt comportant l'ensemble des « path name » des fichiers java contenus dans un dossier passé en paramètre. La commande à employer dans le répertoire contenant le fichier FileList.class (dossier « ProgramFileList ») est la suivante : « java FileList pathNameFolder ». Un fichier fileList.txt est alors généré dans le répertoire contenant le fichier FileList.class.

Notre programme permet de calculer un ensemble de métriques. Pour chaque fichier java, le nombre de classes est calculé ainsi que le nombre d'interfaces. Pour chaque classe ou interface, le nombre de méthodes est calculé. Il est important de déterminer le nombre d'interfaces ou de classes implémentées afin de déterminer des non-conformités. En effet, il est maintenant devenu courant d'implémenter une et une seule classe ou interface par fichier. Cette pratique rend plus lisible le code et permet ainsi d'avoir une meilleure maintenabilité. Le nombre de méthodes est aussi un élément important qui permet de déterminer les fichiers qui sont surchargés de méthodes. Une autre pratique courante dans le développement de logiciels est d'éviter le plus possible de surcharger une classe de méthodes, cela rend sinon la classe peu lisible.

Enfin pour chaque méthode, notre programme calcule les métriques suivantes :

- Nombre de if, else et else if,
- Nombre de for,
- Nombre de while,
- Nombre de switch,
- Nombre de case,
- Nombre de variables locales,
- Nombre de break,
- Nombre de continue.

Le nombre de if et de else sont deux métriques très importantes à déterminer car elles permettent de localiser des défauts d'implémentation. En effet, un défaut courant dans l'implémentation d'un programme est la succession de if et de else if qui pose aussi un problème de maintenabilité, et qui rend le code source peu lisible. Pour les mêmes raisons, la présence trop abondante de switch peut aussi avoir des répercussions sur la maintenabilité du programme. D'autre part, le nombre de for et de while sont aussi calculés. La connaissance du nombre de boucle présente dans une méthode permet de déterminer si une méthode n'est pas trop surchargée (ce qui nuit à sa lisibilité) et permet notamment de déterminer les potentiels cas de boucles imbriquées qui coûtent chers en

temps d'exécution. Ces deux métriques permettent de déterminer les méthodes ayant de fortes chances d'être peu performantes par la présence de nombreuses boucles. De plus, le calcul du nombre de if, case (switch), while et for permet par la suite de déterminer la complexité cyclomatique d'une méthode. La complexité cyclomatique doit de manière générale être inférieure à 10 ou 20 pour une méthode. Si ce n'est pas le cas, il est alors préférable de factoriser la méthode. Enfin, le calcul du nombre de variables locales permet d'identifier les méthodes gourmandes en mémoire.

Une fois toutes les métriques calculées, elles sont ensuite exportées dans un fichier csv ou json dans le dossier « MetricFile ». Le fichier csv permet à l'utilisateur d'analyser les métriques. Le fichier json pourrait permettre à l'utilisateur d'utiliser un autre programme permettant d'analyser automatiquement les métriques.

Notre programme permet notamment de générer le diagramme de classes de n'importe quel logiciel java. Cependant, l'ensemble des types de relations entre classes pouvant apparaitre dans un diagramme de classes ne sont pas tous générés par notre programme. En effet, notre programme génère uniquement les liens de compositions internes, les liens d'implements et les liens d'héritage entre les classe. Notre programme prend bien sûr en considération :

- Le nom des classes,
- Les liens d'héritage des classes,
- Mes liens d'implements des classes,
- Le package des classes,
- Les variables de classe (nom, type et accessibilité),
- Les méthodes de classe (nom, type de retour, accessibilité, nom des paramètres et type des paramètres).

Le diagramme de classes est généré en plusieurs fichiers .dot sous le format GraphViz dans le dossier « ClassDiagram ». Le diagramme est divisé en plusieurs parties afin d'assurer son ouverture via le logiciel GraphViz.

Notre programme propose aussi de générer d'autres diagrammes énoncés précédemment. Tout comme le diagramme de classes, ces diagrammes sont générés en plusieurs fichiers .dot sous le format GraphViz respectivement dans les dossiers suivants :

- Pour le diagramme de flux, le dossier « CFG »,
- Pour le diagramme des dominateurs, le dossier « Dominateurs »,
- Pour le diagramme des post-dominateurs, le dossier « PostDominateurs »,
- Pour le diagramme des définitions valides, le dossier « GraphEntryOut »,
- Pour le diagramme des dépendances de données, le dossier « GraphDependancesDonnees »,
- Pour le diagramme des dépendances de contrôle, le dossier « GraphDependancesControle ».

L'ensemble de ces diagrammes sont générés pour l'ensemble du code du ou des fichiers java fournis en paramètre, c'est-à-dire pour l'ensemble des méthodes.

Pour finir, le programme que nous avons implémenté permet d'effectuer le backward slicing et le forward slicing. C'est deux slicing sont effectués sur du code du ou des fichiers java fournis en paramètre, c'est-à-dire sur :

- Pour le backward slicing, l'ensemble des variables intervenant dans les différentes structures (condition, assignation de variable…),
- Pour le forward slicing, l'ensemble des assignations de variables.

Pour les mêmes raisons que les diagrammes précédents, les résultats du backward slicing et du forward slicing sont générés sous la forme de plusieurs fichiers .dot sous format GraphViz dans les dossiers respectifs « BackwardSlicing » et « ForwardSlicing ». Les résultats du backward slicing sont aussi sauvegardés dans un fichier csv pour une meilleure compréhension des résultats dans certains cas.

## II. METHODOLOGIE

Pour réaliser les différentes fonctionnalités présentées dans la partie précédente, nous avons implémenté un certain nombre de classes que nous pouvons répartir en trois catégories :

- Les visiteurs permettant l'analyse du code java du ou des fichiers java fournis en paramètre,
- Les générateurs des fichiers .dot, csv et json,
- Les classes des structures supportées par notre programme.

Ainsi, nous avons implémenté 3 visiteurs différents héritant de la classe AbstractVisitor fournie par Javacc :

- ExampleVisitor.java : permet de calculer les métriques,
- UmlVisitor.java : permet de récupérer l'ensemble des données nécessaires à la génération du diagramme de classes,
- CFGVisitor.java : permet de récupérer l'ensemble des données nécessaires à la génération des autres diagrammes et nécessaires à la réalisation des slicing.

Nous avons aussi implémenté 10 classes générant les fichiers résultats :

- MetricFileGenerator : permet de générer le fichier csv et json comportant les résultats des métriques,
- ClassDiagramGenerator : permet de générer les fichiers .dot formant le diagramme de classes,
- CFGGenerator : permet de générer les fichiers .dot formant l'ensemble des diagrammes de flux de contrôle,
- DominateurGenerator : permet de générer les fichiers .dot formant l'ensemble des diagrammes des dominateurs,
- PostDominateurGenerator : permet de générer les fichiers .dot formant l'ensemble des diagrammes des post-dominateurs,

- GraphEntryOutGenerator : permet de générer les fichiers .dot formant l'ensemble des diagrammes des définitions valides,
- GraphDependancesControleGenerator : permet de générer les fichiers .dot formant l'ensemble des diagrammes des dépendances de contrôle,
- GraphDependancesDonneesGenerator : permet de générer les fichiers .dot formant l'ensemble des diagrammes des dépendances de données,
- GraphBackwardSlicingGenerator : permet de générer les fichiers .dot et le fichier csv représentant les résultats du backward slicing,
- GraphForwardSlicingGenerator : permet de générer les fichiers .dot représentant les résultats du forward slicing.

Enfin, nous avons implémenté 17 classes représentant les différentes structures couvertes par notre programme :

- Classe : Cette classe permet de définir les instances représentant les classes pour la génération du diagramme de classes,
- Method : Cette classe permet de définir les instances représentant les méthodes présentes dans une classe pour la génération du diagramme de classes, ainsi que pour le reste des diagrammes et des slicing,
- Structure : Cette classe est une super classe permettant de définir les différents éléments (attributs et méthodes) communes aux structures couvertes par notre programme. Cette classe est utilisée pour la génération des diagrammes (excepté le diagramme de classes) et la réalisation des slicing,
- Break ; Case ; Catch ; Continue ; Do ; FinallyStructure ; For ; If ; Return ; Switch ; Throw ; Try ; Variable ; While : Ces classes héritent de la classe Structure, et sont utilisées pour la génération des diagrammes (excepté le diagramme de classes) et la réalisation des slicing.

Nous avons utilisé notre programme pour analyser notre programme lui-même afin de déterminer son diagramme de classes présenté en plusieurs parties en annexes (**figure 1, figure 2 et figure 3**).

## III. Resultats

Nous allons à présent présenter des résultats que nous obtenons à l'aide de notre programme. Pour cela nous allons considérer le code suivant (**figure 4**) :

```
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 = nw + 1;
        }
    }
    System.out.println(nl);
    System.out.println(nw);
    System.out.println(nc);
}
```

**Figure 4. code exemple**

Pour ce code, nous avons obtenu les diagrammes de flux de contrôle (**figure 5**), des dominateurs (**figure 6**), des post-dominateurs (**figure 7**), des définitions valides (**figure 8**), des dépendances de contrôle (**figure 9**) et des dépendances de données (**figure 10**) présentés en annexes. Les résultats du backward slicing (**figure 11**) et du forward slicing (**figure 12**) sont aussi présentés en annexes. Les résultats du backward slicing sont aussi visualisables dans un fichier csv.

Nous avons pu tester notre programme sur un ensemble de petit exemple de code comme celui que nous avons présenté précédemment. Nous avons notamment testé notre programme à l'aide de notre propre programme contenant 30 fichiers. Nous avons aussi testé notre programme sur le logiciel Eclipse JDT core. Ce logiciel est composé de 6895 fichiers et de 1,33 millions de lignes de code. L'utilisation de ce logiciel nous a permis de mettre en évidence tout un ensemble de limites à notre programme. Les performances observées dans ces deux cas sont résumées dans le tableau suivant (figure 13).

| | | Projet de réingénierie (30 fichiers) | Eclipse JDT core (6895 fichiers / 1,33 MLOC) |
|---|---|---|---|
| Temps d'exécution (en s) | Génération des métriques | 0,619 | 66,189 |
| | Diagramme de classes | 0,722 | 54,555 |
| | Diagramme de flux de contrôle | 0,855 | 84,219 |
| | Diagramme des dominateurs | 0,848 | 96,072 |
| | Diagramme des post-dominateurs | 0,880 | 97,781 |
| | Diagramme des définitions valides | 1,277 | 149,843 |
| | Diagramme des dépendances de données | 1,219 | 143, 954 |
| | Diagramme des dépendances de contrôle | 1,277 | 146,335 |
| | Backward slicing | 1,269 | 144,435 |
| | Forward slicing | 1,627 | 175,956 |

**Figure 13. Tableau des performances du programme**

Comme on peut le remarquer dans le tableau précédant, notre programme dispose de performances tout à fait acceptables pour la génération des métriques, du diagramme de classes et du diagramme de flux de contrôle. Les générations du diagramme des dominateurs, des post-dominateurs et des définitions valides nécessitent un temps plus élevé mais encore convenable car le programme doit avant générer en mémoire le diagramme de flux de contrôle. De même, le diagramme des dépendances de contrôle nécessite préalablement le calcul des post-dominateurs et du diagramme de flux de contrôle. Enfin, le diagramme des dépendances de données nécessite pour sa part le calcul des définitions valides et du diagramme de flux de contrôle.

Les deux types de slicing demandent eux aussi un certain temps d'exécution. Cela est dû à la nécessité de calculer préalablement les dépendances de données et de contrôle.

## IV. Limites

Notre programme dispose de performances suffisamment convenables. Il propose aussi à l'utilisateur un certain nombre de fonctionnalités. Cependant, notre programme dispose malheureusement d'un très grand nombre de limites.

Tout d'abord, les différents diagrammes sont générés en plusieurs fichiers .dot. Cela dû à la limitation technologique du logiciel GraphViz qui ne parvient pas à ouvrir des fichiers .dot de taille trop importante. L'analyse et la compréhension des diagrammes sont de ce fait très difficiles. Par exemple, plusieurs parties d'un diagramme de classes directement liées peuvent se trouver dans des fichiers .dot différents. Les diagrammes sont tous divisés de manières arbitraires. Chaque fichier .dot ne comporte qu'au maximum 20 entités (20 classes pour les diagrammes de classes, 20 méthodes pour les autres diagrammes). Cela nous permet d'être quasiment certain de pouvoir ouvrir et visualiser les diagrammes. Initialement, nous avions opté pour diviser les diagrammes de classes par package. Malheureusement pour des logiciels de taille réelle, une très large majorité des fichiers était impossible à ouvrir. Pour les autres diagrammes, nous les avions divisés par fichier. Cependant tout comme pour les diagrammes de classes, une très large majorité des fichiers était impossible à ouvrir. C'est pourquoi nous avons choisi de mettre en place ce système de division totalement arbitraire des diagrammes en plusieurs fichiers .dot. Il est tout de même possible dans certains cas très particulier (par exemple dans le cas d'une God Class) que des fichiers .dot soient quand même impossible à ouvrir. Ainsi, il serait intéressé de concevoir à l'avenir un système permettant de diviser les différents diagrammes de la manière la plus optimale possible.

Deuxièmement, nous avons utilisé le logiciel GraphViz afin de représenter sous la forme de diagramme les résultats que nous obtenions. Ces représentations nous ont permis de vérifier le bon fonctionnement de notre programme. Ces représentations pourraient être utilisées par les utilisateurs pour analyser à la main leur logiciel. Cependant, nous doutons de la consistions, de la compréhensibilité et de l'exploitabilité des représentations des résultats. Dans un grand nombre de cas, les diagrammes sont trop complexes et trop peu lisibles pourraient être facilement exploités par l'utilisateur. Il serait intéressant de concevoir à l'avenir des représentations bien plus claires, lisibles et compréhensibles pour une majorité des utilisateurs.

Notre programme se base sur l'analyse du code java. De ce fait, il intègre un certain nombre de structure que l'on retrouve dans du code java (for, while, if…). Malheureusement, notre programme n'intègre pas l'ensemble des structures que l'on peut trouver dans du code java. Au cours de notre projet, nous avons remarqué que notre programme ne supportait pas certaines structures basiques comme : « int a=1,b=2,c=3 ; ». Nous avons dû ajouter au fur et à mesure modifier notre programme pour prendre en compte de nouvelles structures. De nombreuses structures ne sont tout de même pas encore couvertes par notre programme. De ce fait, de très nombreuses exceptions sont rencontrées au cours de l'exécution de notre programme avec le logiciel Eclipse

JDT core. En effet, Eclipse JDT core utilise des structures de code assez particulières comme les expressions lambda et d'autres que nous n'avons pas eu le temps de déterminer et de couvrir. On peut donc considérer que notre programme n'est d'aucune utilité pour certains logiciels.

Enfin, notre programme ne permet pas à l'utilisateur de lancer le backward slicing ou le forward slicing sur une variable et une ligne en particulier. A la place, notre programme effectue le slicing sur l'ensemble des éléments du code possibles. Les résultats obtenus des slicing sont en grande partie peu lisibles et exploitables à cause de cette particularité.

Pour finir, notre programme est loin d'être parfait en termes de performances. Nous sommes convaincus qu'il est possible d'optimiser notre code ainsi que les structures de données que nous avons mises en place. De plus, certains éléments de notre architecture ne nous semblent pas recommandables. On peut citer par exemple la présence de plusieurs classes nous permettant de générer les différents diagrammes et fichiers résultats. On peut en effet compter une bonne partie de code dupliquer dans notre implémentation. Ceci est en grande partie dû à l'absence de vision globale du projet à son commencement et tout au long de son déroulement.

## CONCLUSION

En conclusion, nous avons réussi au cours de ce projet à implémenter un programme respectant les exigences fonctionnelles et fonctionnant parfaitement bien pour certains programmes java ne comportant pas de structures non supportées. Ce projet nous a permis d'appréhender, de comprendre et d'assimiler l'ensemble des notions vues en cours théorique du cours LOG6302.

ANNEXES

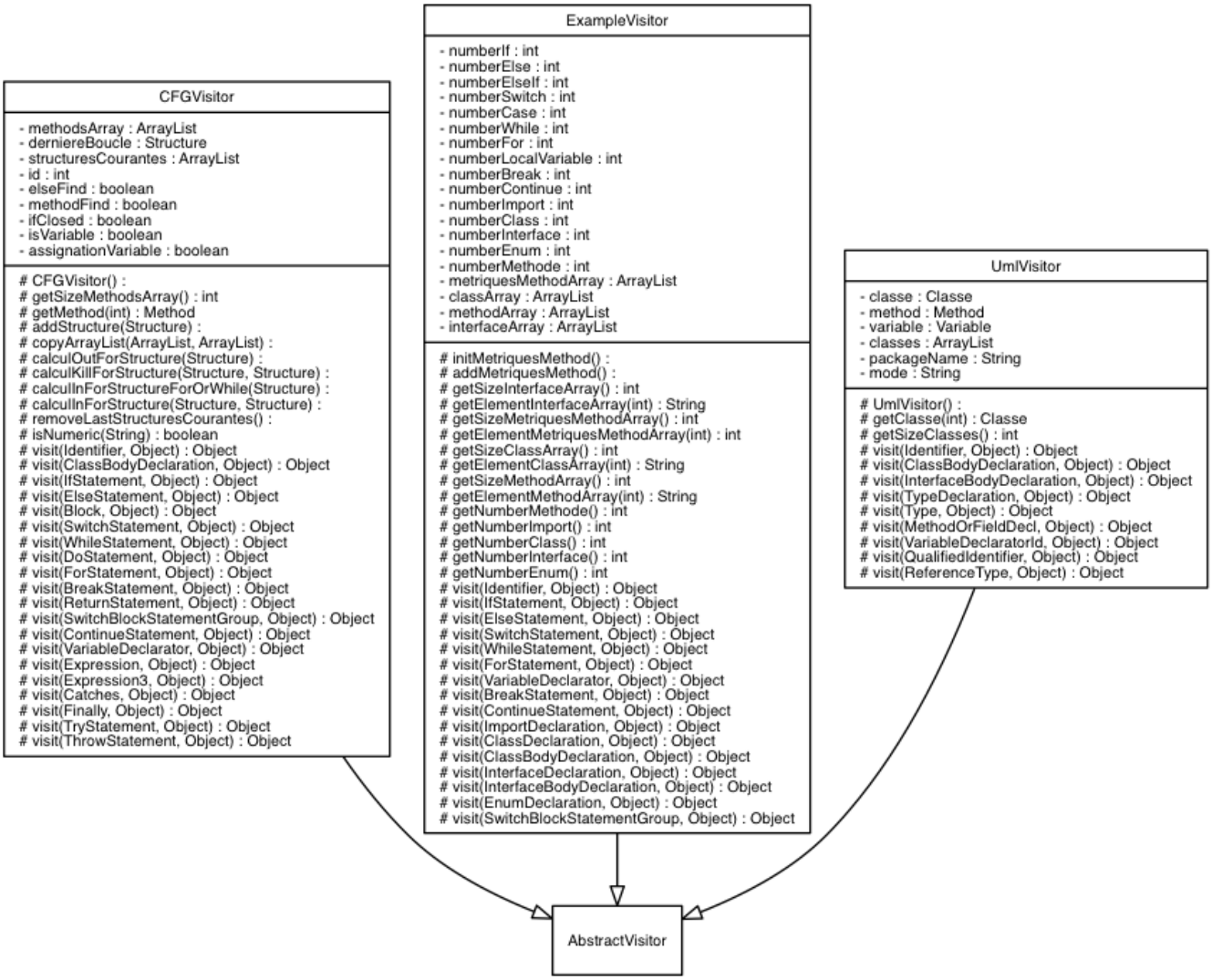


**Figure 1. Diagramme de classes du programme (partie visiteurs)**

**Figure 2. Diagramme de classes du programme (partie Structures)**

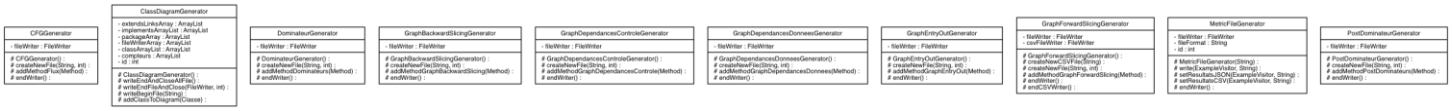


**Figure 3. Diagramme de classes du programme (partie Générateur de fichiers)**

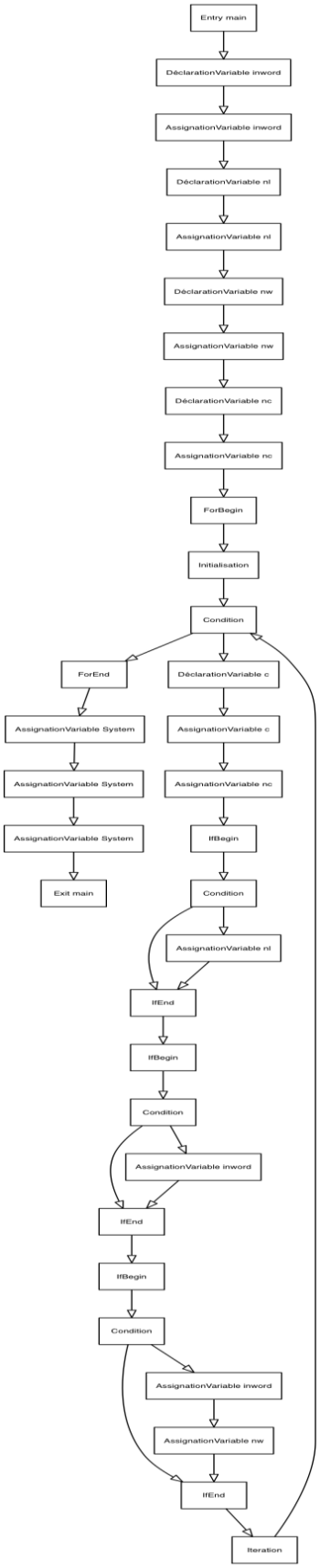


**Figure 5. Diagramme de flux de contrôle du code exemple**

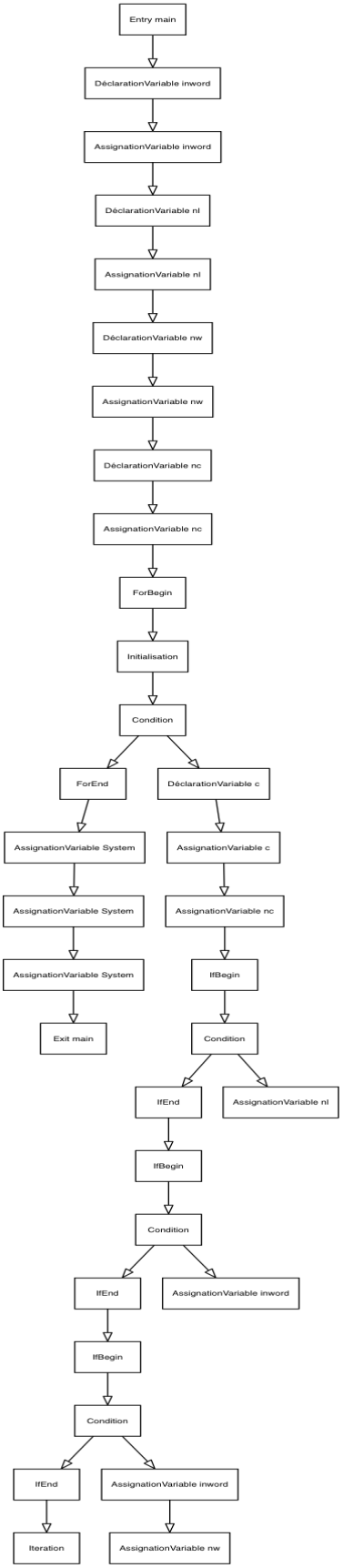


**Figure 6. Diagramme des dominateurs du code exemple**


**Figure 7. Diagramme des post-dominateurs du code exemple**

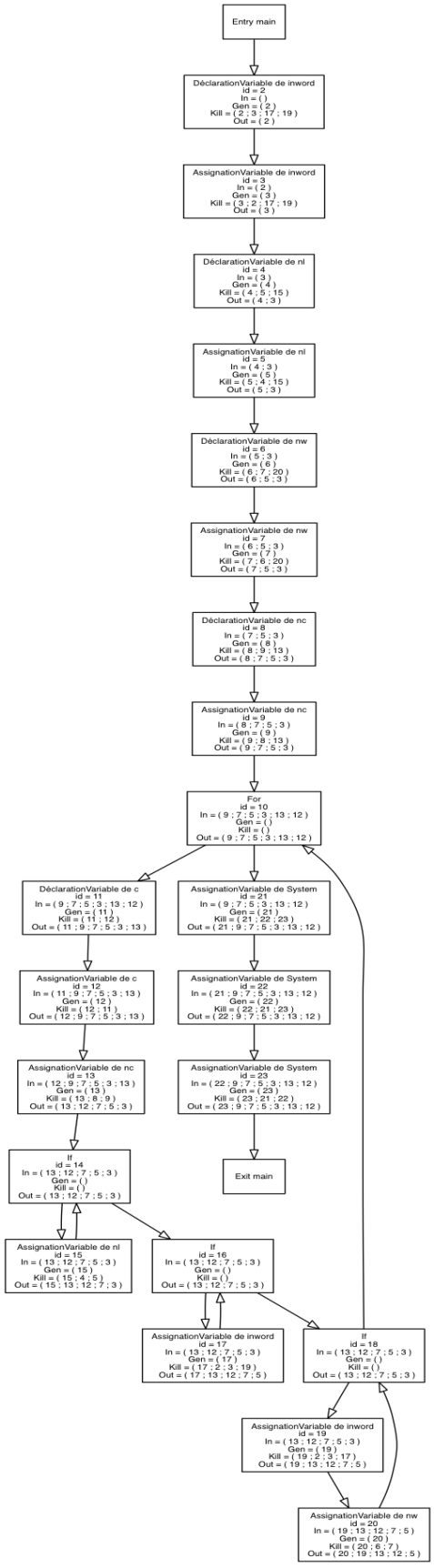


**Figure 8. Diagramme des définitions valides du code exemple**

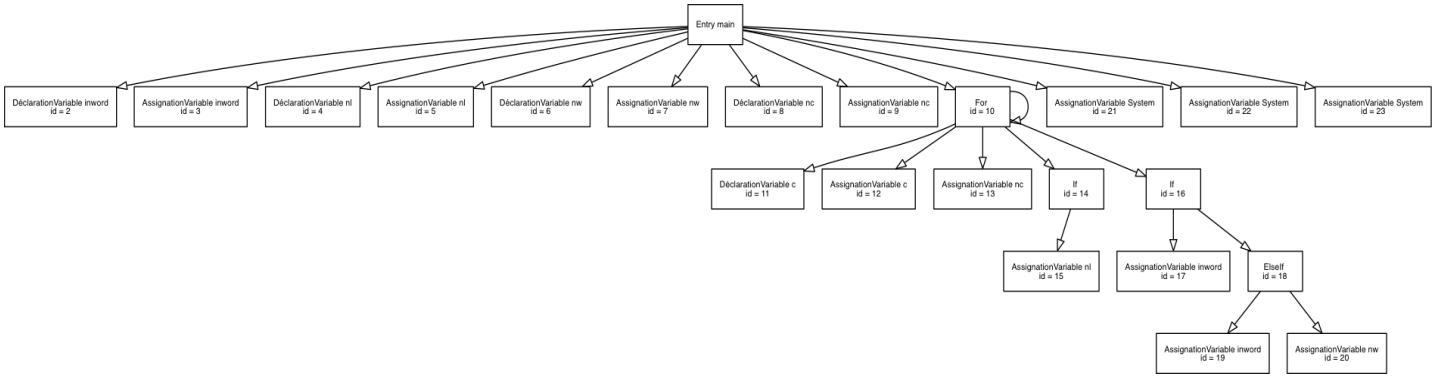


**Figure 9. Diagramme des dépendances de contrôle du code exemple**

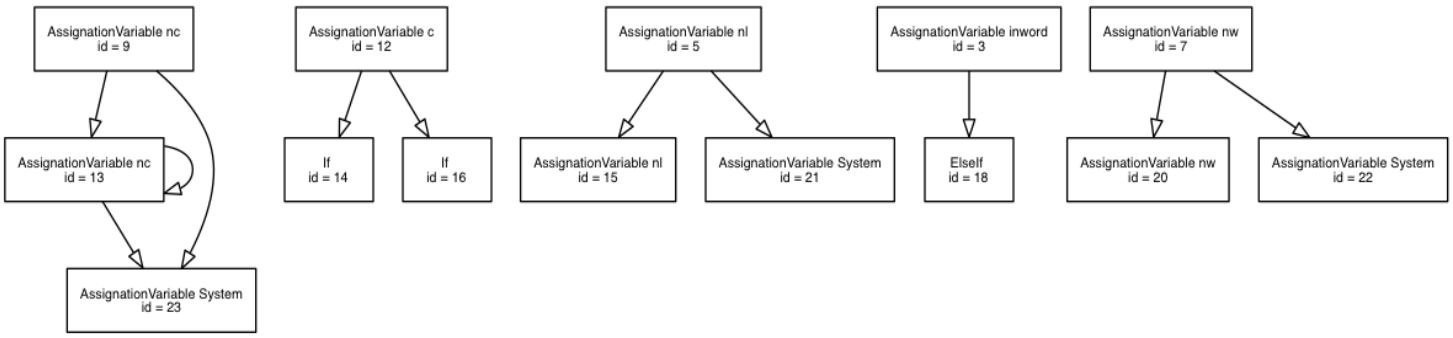


**Figure 10. Diagramme des dépendances de données du code exemple**

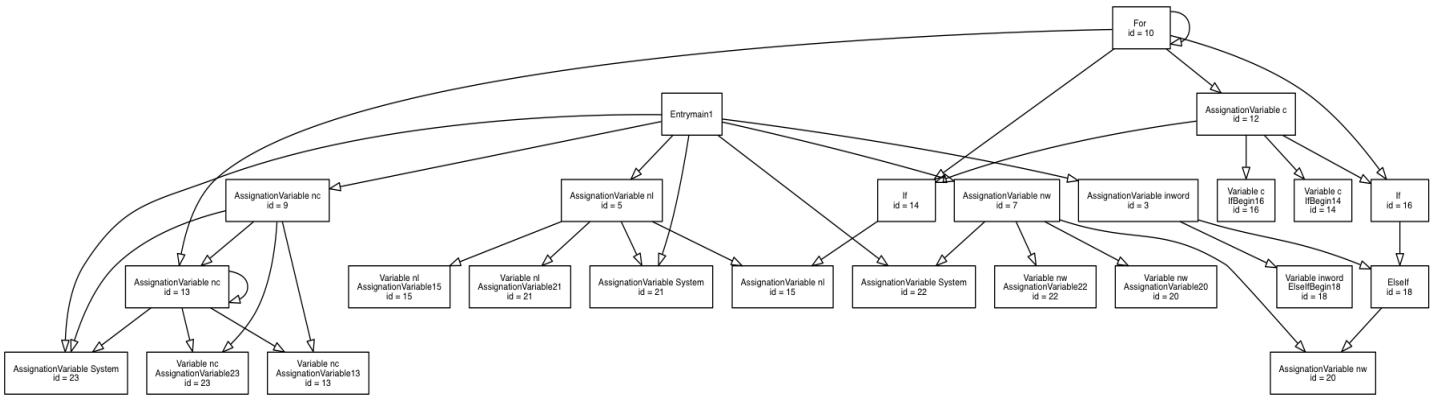


**Figure 11. Résultats du backward slicing du code exemple**

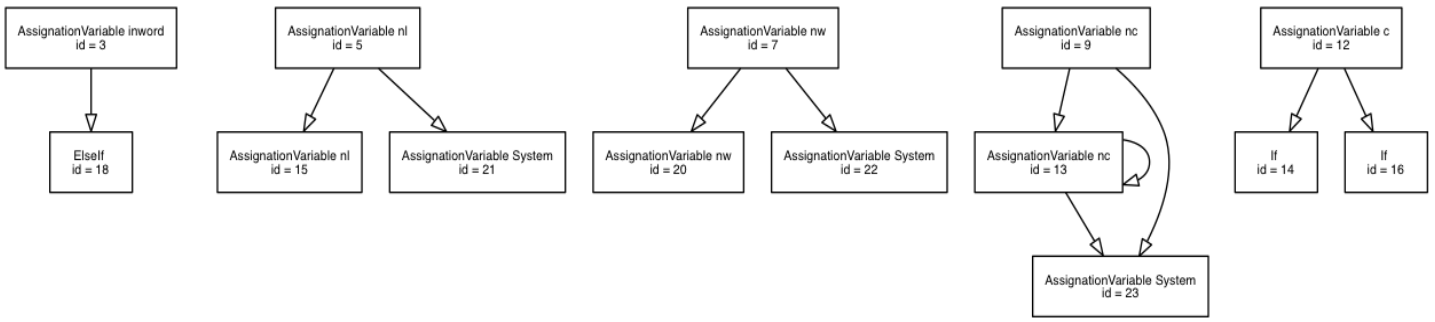


**Figure 12. Résultats du forward slicing du code exemple**