Analyse statique du code JAVA d'une application et modélisation sous formes de graphes

Manuel Chataigner, Jimmy Lopez et Morgane Vidal

1 Analyse du code source java en utilisant l'arbre syntaxique abstrait (AST)

1.1 Présentation de l'architecture générale

Dans un premier temps, nous avons créé l'AST associé à une classe Java. Pour cela, on utilise un AST parser qui nous permet d'obtenir les attributs, méthodes, variables locales et méthodes appelées d'une classe.

Afin d'enregistrer les informations de chacun de ces éléments, nous avons choisi de mettre en place trois différentes classes représentant les classes analysées, les méthodes et les variables (regroupant variables locales, attributs et paramètres). Ainsi, comme on peut le voir dans le diagramme suivant, une classe contient un nom ainsi que la liste de ses attributs et méthodes représentés respectivement par les classes **ASTVariable** et **ASTMethod**.

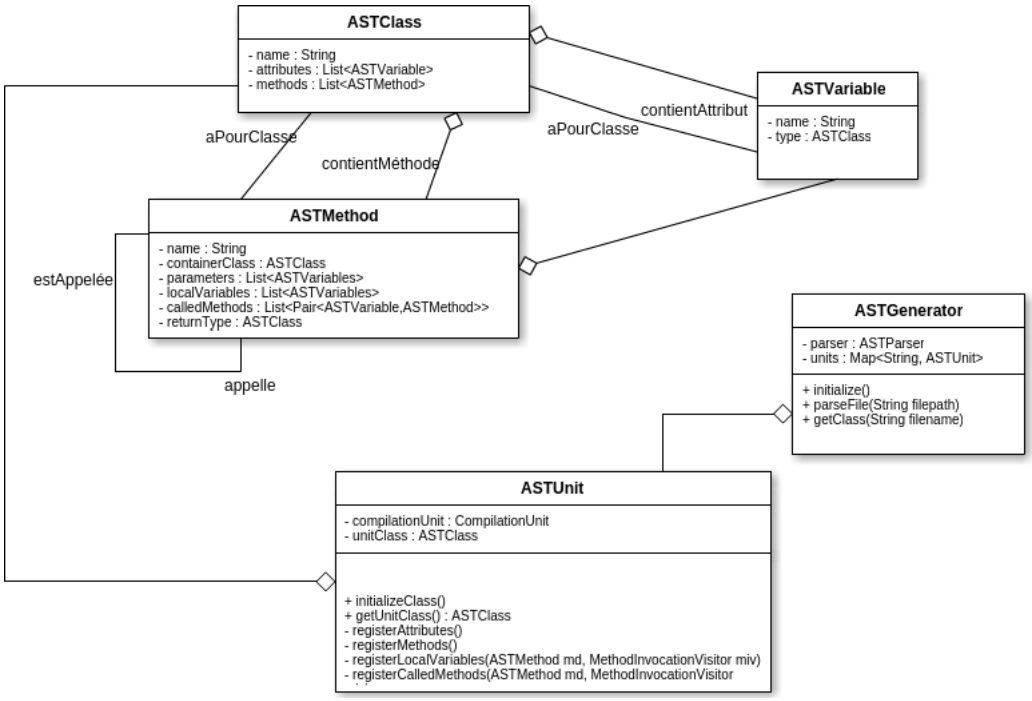

Figure 1: Diagramme de classes

A l'inverse, une méthode contient une référence vers la classe à laquelle elle appartient, un nom, la liste de ses paramètres, de ses variables locales, ainsi que la liste de ses méthodes appelées.

Enfin, une variable est représentée par son nom et son type, correspondant à une référence à la classe correspondante.

Le but étant de représenter les différents appels aux sein d'une classe, mais aussi entre les différentes classes, nous avons choisi d'enregistrer les variables locales, paramètres et attributs afin de pouvoir d'une part récupérer les types associés aux receveurs des appels de méthode et donc savoir à quelle classe appartiennent les méthodes appelées, et d'autre part, connaître les types des arguments pour déterminer les bonnes signatures de méthode.

Pour récupérer l'ensemble des informations de l'AST, nous utilisons la classe **ASTParser**. Dans le but de pouvoir gérer plusieurs classes java, nous avons créé une classe **ASTGenerator** contenant un attribut **ASTParser**, ainsi que l'ensemble des **CompilationUnit** générés par le parser. Pour récupérer plus facilement les informations contenues dans une unité, nous avons également mis en place une classe **ASTUnit** encapsulant une unité et l'objet **ASTClass** créé, et récupérant l'ensemble des informations nécessaires à travers la méthode publique *initializeClass()* appelant les méthodes privées *registerAttributes()* et *registerMethods()* faisant elle-même appel aux méthodes privées *registerLocalVariables()* et *registerCalledMethods()*.

1.2 Méthode de création de l'AST

Pour générer les graphes d'appel, nous parcourons récursivement un dossier donné. Pour chaque fichier source java, nous créons l'AST correspondant à l'aide de l'**ASTParser**, puis nous récupérons l'objet **CompilationUnit** retourné. Nous créons donc une instance de notre classe **ASTUnit** qui nous permettra de traiter les informations de l'AST pour le fichier source en question.

En premier lieu, nous créons l'objet **ASTClass** associé au fichier java, à partir du nom du **TypeDeclaration** s'il existe, sinon cela veut dire qu'il n'y a pas de classe dans le fichier parsé.
Puis nous lançons les méthodes *registerAttributes()* et *registerMethods()* sur notre objet **ASTUnit**.

Pour chaque attribut de la classe parcourue, nous récupérons le nom et le type associé, que nous stockons dans un objet **ASTVariable**. On ajoute ensuite l'objet représentant l'attribut dans la liste des attributs de l'**ASTClass** correspondant à la classe visitée.

Lors du traitement des méthodes, nous récupérons dans un objet **ASTMethod** le nom, le type de retour et la liste des paramètres stockés dans des objets **ASTVariable**. Le reste des informations de la méthode est récupéré dans son corps. Pour cela nous utilisons un visiteur qui récupère dans le corps

de la méthode parsée les instances, déclarations de variable et invocations de méthodes.
Enfin nous ajoutons la méthode dans l'**ASTClass** correspondante.

A partir des informations du visiteur, nous stockons ensuite dans la méthode l'ensemble des variables locales et appels de méthode. Pour ces derniers, nous récupérons le receveur, en mettant *this* s'il n'existe pas, puis le nom de la méthode appelée.
Par la suite, nous récupérons la liste des arguments envoyées à la méthode, pour stocker la liste des types des paramètres de la méthode. Nous vérifions donc pour chaque argument de l'appel, s'il s'agit d'un littéral, d'une instance nouvellement crée ou d'une variable afin d'en déduire les types associés. Pour rechercher les variables du receveur et des paramètres, nous vérifions s'il s'agit de *this*, puis, afin d'obtenir le type, nous cherchons dans les variables locales, dans les paramètres, puis dans les attributs.

Pour le stockage des méthodes, deux possibilités s'offrent à nous. Nous pouvons dans un premier temps rencontrer un appel, auquel cas nous stockons dans la liste des paramètres de l'**ASTMethod** les types et noms des arguments. Puis nous mettons à jour les paramètres lorsque l'on arrive sur la définition de la méthode, afin d'enregistrer les noms corrects des paramètres. Sinon, c'est la signature qui est parsée en premier, nous la gardons alors pour vérifier les appels de méthodes.

1.3 Structure de graphe

Une fois les informations stockées dans notre ensemble de classes, il nous est alors possible de récupérer l'ensemble des appels au sein d'une même classe, puis entre les différentes classes.

Nous avons donc mis en place un système de graphes pour stocker les données des différents appels. Ainsi, nous avons une classe **Digraph** qui permet de représenter un graphe orienté. Cette classe est composée d'une liste de nœuds. Chacun des nœuds possède un ensemble d'arêtes entrantes et sortantes, et chaque arête possède un seul noeud, le noeud cible, ainsi que son poids. Ces trois classes sont génériques et disposent de quelques méthodes abstraites à redéfinir pour avoir un comportement précis par rapport au type T.

Nous avons créé deux nouveaux **DiGraph**, **DigraphASTClass** dont chaque noeud va contenir des **ASTClass** et **DiGraphASTMethod** dont chaque noeud contiendra des **ASTMethod**.

La classe **CallGraph** permet de générer un **DiGraph** d'**ASTMethod** à partir d'une **ASTClass**, on crée donc un graphe d'appels au sein d'une classe. Elle permet aussi à partir d'une liste d'**ASTClass**, d'avoir un **Di-**

Graph d'**ASTClass** qui représentera le graphe d'appels entre chaque classe de l'application.

1.4 Visualisation de graphe

Afin de visualiser les graphes générés, nous avons choisi d'utiliser la librairie GraphStream. Celle-ci permet de représenter graphiquement l'ensemble des nœuds de notre graphe ainsi que les arrêtes représentant les appels de méthodes. Il est possible de déplacer les nœuds dans le graphe généré afin d'avoir de meilleurs points de vues sur certaines parties.

2 Analyse du code source d'un exemple

Nous avons choisi d'analyser le code source de notre projet de génération d'arbres syntaxiques.

2.1 Graphe d'appels au sein d'une classe

Figure 2: Graphe d'appels au sein de ASTUnit

La figure ci-dessus représente le graphe d'appels au sein de la classe **ASTUnit**. Comme on peut le voir par exemple, *registerCalledMethod()* fait appel à la méthode *addParamOfCalledMethod()*. De même, *registerMethods()* fait appel aux méthodes *addParamOfMethod()*, *registerCalledMethod()* et *registerLocalVariables()*.

2.2 Graphe d'appels de l'application

La figure ci-dessous correspond au graphe d'appels entre les différentes classes de l'application, avec les différents poids des arêtes. Ainsi, on peut voir par exemple que la classe **DiGraphASTMethod** possède quatre appels vers la classe **NodeMethod**. De ce fait, **DiGraphASTMethod** ne peut pas être utilisé sans **NodeMethod** alors que l'on peut utiliser **NodeMethod** sans **DiGraphASTMethod**. On peut donc dire que la classe **DiGraphASTMethod** est dépendante de la classe **NodeMethod**.

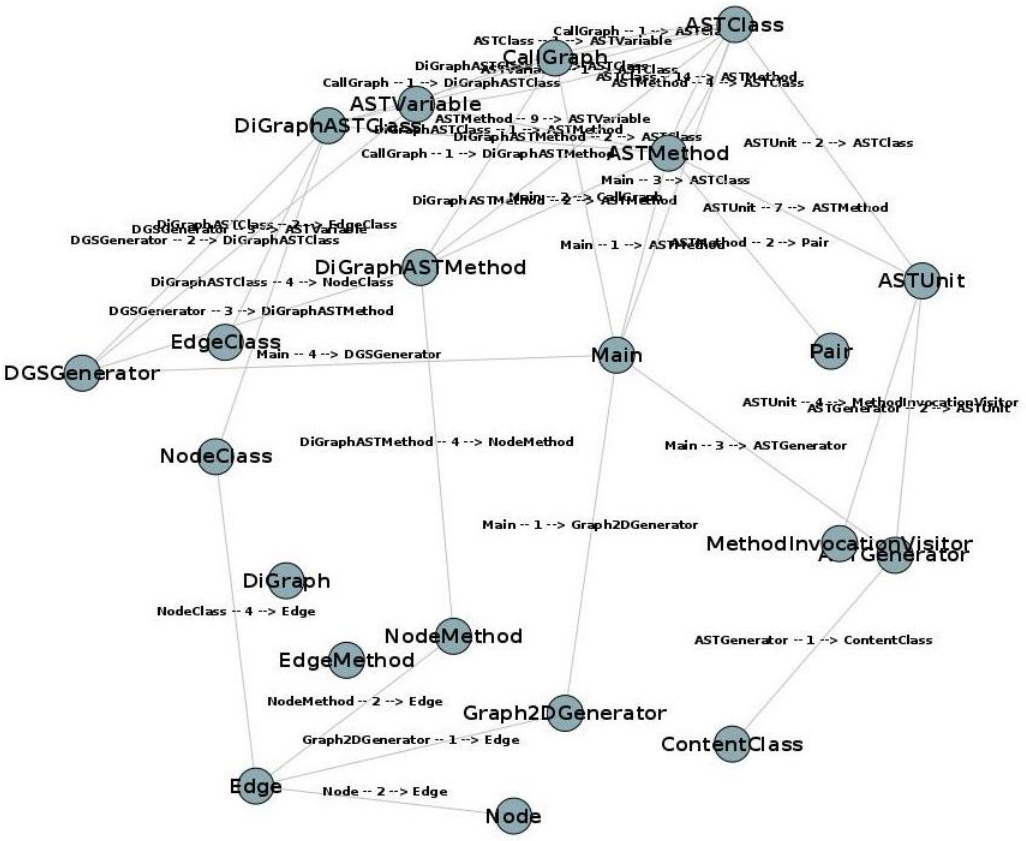

Figure 3: Graphe d'appels de l'application