TP1 : Analyse statique
Évolution et restructuration des logiciels

Mohamad Satea Almallouhi - Tony Nguyen
M1 Génie Logiciel
Faculté des Sciences
Université de Montpellier.

6 octobre 2024

Résumé

Rapport d'exercice sur l'analyse statique

Table des matières

Introduction

Dans le cadre de l'Unité d'Enseignement Évolution et restructuration des logiciels, nous allons analysés un programme en observant son code source de manière statique.

Tout d'abord, l'approche consistera a extraire un modèle du code source. Cela prendra la forme d'un AST.

Par la suite, à l'aide du Patron de Conception Visiteur, nous allons parcourir l'arbre pour en extraire des propriétés.

Démonstration vidéo

En ligne sur Youtube, à l'adresse URL `https://youtu.be/fDvHlrK8IRg` une démonstration vidéo de notre travail.

Installation

Vous trouverez les instructions dans le README.md

1 Extraction de l'AST

Cette partie est prise en charge par la classe ASTParser.

Cependant, un programme étant composé d'une mutltitude de fichier, nous allons procédé en les analysant un par un.

Nous indiquons le chemin vers le programme à analyser.

```
public MyParser(String pathToProject) {
    final File folder = new File(pathToProject);
    javaFiles = listJavaFilesForFolder(folder);
    nbFile = javaFiles.size();
```

Ensuite, la fonction suivante va fouiller récursivement le repertoire pour trouver tous les fichiers du programme.

```
// read all java files from specific folder
public static ArrayList<File> listJavaFilesForFolder(final File folder) {
    ArrayList<File> javaFiles = new ArrayList<File>();
    for (File fileEntry : folder.listFiles()) {
        if (fileEntry.isDirectory()) {
            javaFiles.addAll(listJavaFilesForFolder(fileEntry));
        } else if (fileEntry.getName().contains(".java")) {
            // System.out.println(fileEntry.getName());
            javaFiles.add(fileEntry);
        }
    }

    return javaFiles;
}
```

Après ça, il sera possible de transmettre le contenu du fichier au parser.

```
for (File fileEntry : javaFiles) {

    int nbMethod, nbAttr;
    String nom = fileEntry.getName();
    int end = nom.lastIndexOf(".java");
    nom = nom.substring(0, end);

    String content = FileUtils.readFileToString(fileEntry);
    CompilationUnit parse = parse(content.toCharArray());

    // compute and extract here
    LineCountVisitor visitor = new LineCountVisitor(parse);
    parse.accept(visitor);
```

2 Implementation du visiteur

Afin de parcourir l'AST, nous allons utilisé un visiteur. Pour en implémenter une compatible avec ASTParser, nous spécialisons la classe abstraite ASTVisitor.

Pour pouvoir naviguer précisement dans l'arbre, il nous faut masquer la méthode visit(T node), avec T extends ASTNode. C'est en choisisant soignement le type T que nous pouvons atteindre l'informations recherché.

2.1 Le nombre de classe

Dans la classe MasterVisitor, nous écrivons la méthode visit(TypeDeclaration node). Ici la variable node va correspondre à une classe ou une interface. Ainsi, types.add(node) nous permettra de compter le nombre de classe dans un fichier.

```
@Override
public boolean visit(TypeDeclaration node) {
    nomClass = node.getName().getIdentifier();

    // Récupérer le nombre de lignes de la classe (ou interface)
    int startLine = compilationUnit.getLineNumber(node.getStartPosition());
    int endLine = compilationUnit.getLineNumber(node.getStartPosition() + node.getLength());
    int classLineCount = endLine - startLine + 1;
    totalLines += classLineCount;
    ////////////////////////////

    if (!node.isInterface()) { types.add(node); }

    return super.visit(node);
}
```

Pour chaque fichier, on compte le nombre de classe, afin de s'assurer de bien prendre en compte les classes imbriqués.

```
nbClass += visitor.getTypes().size();
```

2.2 La méthode avec le plus de paramètre

Pour chaque fichier à analyser, lors d'une visite d'un noeud MethodDeclaration, nous instancions un tuple sous la forme d'un object CoupleNomData avec comme nom le nom de la méthode et en data, son nombre d'argument. Ils sont mémoriser dans une liste.

```
@Override
public boolean visit(MethodDeclaration node) {
    this.nbMethod++;

    // Récupérer le nombre de lignes d'une méthode
    int startLine = compilationUnit.getLineNumber(node.getStartPosition());
    int endLine = compilationUnit.getLineNumber(node.getStartPosition() + node.getLength());
    int methodLineCount = endLine - startLine + 1;
    this.nbMethodLine += methodLineCount;

    // Récupérer le nom de la méthode
    String methodName = node.getName().getFullyQualifiedName();

    // Récupérer le nombre de paramètres
    int argumentCount = node.parameters().size();

    // Afficher le nom de la méthode et le nombre d'arguments
      System.out.println("Méthode: " + methodName + " -> Nombre d'arguments: " + argumentCoun
    list.add(new CoupleNomData(this.nomClass+"::"+methodName, argumentCount));


    return super.visit(node);
}
```

À la fin de l'analyse d'un fichier, la liste est trié à l'aide des bibliothèques de bases de Java. Il sera ajouté à listTopParam, une liste pour sauvegarder le meilleur candidat de chaque fichier.

```
ArrayList<MyParser.CoupleNomData> l = visitor.getList();
Collections.sort(l);

listTopParam.add(l.get(0));
```

Finalement, cette liste sera également triée de la même façon afin de garder la méthode avec le plus d'argument.

```
Collections.sort(listTopParam);
```

3 Exploitation

Rien de particulier, on se contente de simplement afficher les informations.

4 Graphe d'appel

Nous avons réalisé le graphe d'appel, mais à 1 seul niveau. On ne va pas regarde de façon récur-

sive.

Nous allons regarde chaque déclaration de méthode, pour chacune d'entre elle, nous enregistrons quels méthodes sont invoqués avec le type MethodInvocation.

```java
public static void printMethodInvocationInfo(CompilationUnit parse) {

    MethodDeclarationVisitor visitor1 = new MethodDeclarationVisitor();
    parse.accept(visitor1);
    for (MethodDeclaration method : visitor1.getMethods()) {

        MethodInvocationVisitor visitor2 = new MethodInvocationVisitor();

        method.accept(visitor2);

        HashSet<String> setMethod = new HashSet();
        for (MethodInvocation methodInvocation : visitor2.getMethods()) {

            setMethod.add(methodInvocation.getName().getIdentifier());
        }
        if (setMethod.isEmpty()) {
            System.out.println(method.getName() + " n'appel pas de méthode");
        } else {
            System.out.println(method.getName() + " appel : ");
            for (String s : setMethod) {
                System.out.println("  "+s);
            }
        }

    }
}
```

```java
public class MethodDeclarationVisitor extends ASTVisitor {
    List<MethodDeclaration> methods = new ArrayList<MethodDeclaration>();

    @Override
    public boolean visit(MethodDeclaration node) {
        methods.add(node);
        return super.visit(node);
    }

    public List<MethodDeclaration> getMethodsDeclared() {
        return methods;
    }
}
```

```java
public class MethodInvocationVisitor extends ASTVisitor {
    List<MethodInvocation> methods = new ArrayList<MethodInvocation>();
    public boolean visit(MethodInvocation node) {
        methods.add(node);
        return super.visit(node);
    }


    public List<MethodInvocation> getMethods() {
        return methods;
    }
}
```