Université de Montpellier
Master 2 Informatique Génie Logiciel

Évolution et restructuration des logiciels :

Analyse statique et dynamique : TP 1 Partie 2

Rapport du TP

Étudiante :
GARCIA Léa

1 Exercice 1 : Calculs statistiques pour une application OO

Pour lancer mon projet il suffit d'ouvrir le projet dans IntelliJ et de lancer la classe main du fichier Parser.java et d'intéragir avec la CLI qui se lance dans le terminal.

Le projet que j'ai analysé est le projet d'analyse lui même.

Pour clone mon projet il faut utiliser cette URL HTTPS :

`https://github.com/Ravenn2203/Rendu_TP_1.git`

Pour concevoir mon application d'analyse de code source j'ai utilisé comme base, le code proposé en correction. J'ai créé une classe InterfaceUtilisateur.java pour réaliser ma CLI qui propose d'exécuter l'un des deux exercices, pour le premier elle propose chaque façon d'analyser l'application passée en paramètres et pour le deuxième elle affiche le graphe d'appel.

1.1 Question 1.1

J'ai utilisé la classe Parser pour d'abord afficher la CLI, proposer à l'utilisateur de choisir, et ensuite lancer l'exercice demandé. Pour l'exercice 1, ma fonction "choixAnalyse()" récupère tout d'abord le choix de l'utilisateur et avec un switch case, lance le code correspondant. J'ai énuméré ma conception pour chaque analyse différente dans les sections suivantes.

Dans chaque cas, je parcours chaque fichier java trouvé dans l'application, je récupère le contenu, j'applique l'extracteur de modèle, un parseur et je récupère l'AST du fichier. Ensuite j'applique un extracteur de propriétés, un visiteur (on va voir ensuite ceux que j'ai créés), qui va extraire les propriétés des noeuds de l'AST. Enfin, selon chaque cas, je fais les calculs nécessaires à l'analyse.

Les visiteurs que j'ai créés avant mon switch sont les suivants :
- TypeDeclarationVisitor
- MethodDeclarationVisitor
- PackageDeclarationVisitor

Le premier permet de visiter les noeuds classes/interfaces, le deuxième permet de visiter les noeuds méthodes et enfin le dernier permet de visiter les noeuds packages.

```
public static void choixAnalyse(ArrayList<File> javaFiles, int choix) throws IOException {

    //Je récupère mes visiteurs
    TypeDeclarationVisitor visiteurClassesEtInterfaces = new TypeDeclarationVisitor();
    MethodDeclarationVisitor visiteurDeclarationMethodes= new MethodDeclarationVisitor();
    PackageDeclarationVisitor visiteurPackages = new PackageDeclarationVisitor();

    switch (interfaceUtilisateur.getChoixAnalyse()) {
        case 1: {
```

1.1.1 Le nombre de classes

Dans ce premier cas, j'accepte tout d'abord un visiteur de type TypeDeclarationVisitor, puis j'appelle la méthode getNbClassesEtInterfaces() que j'ai défini sur la classe de ce visiteur et qui récupère simplement la taille de la variable types, attribut du visiteur contenant l'ensemble des noeuds de type classe ou interface.

1.1.2 Le nombre de lignes de code

Dans ce cas là, je n'utilise pas le visiteur, je récupère le fichier .java parsé puis j'incrémente une variable avec pour chaque fichier l'entier qui représente le numéro de la dernière ligne :

```
parse.getLineNumber(parse.getLength() -1);
```

1.1.3 Le nombre total de méthodes

Dans ce cas, j'accepte tout d'abord un visiteur de type MethodDeclarationVisitor, puis j'appelle la méthode getNbMethodes() que j'ai défini sur la classe de ce visiteur qui récupère simplement la taille de la variable methods, attribut du visiteur contenant l'ensemble des noeuds de type méthode, comme pour le cas numéro 1.

1.1.4 Le nombre total de packages

Exactement comme pour les classes et les méthodes, j'utiliser le visiteur de package et je récupère la taille de la variable packages.

1.1.5 Le nombre moyen de méthodes par classe

Pour ce cas, j'accepte le visiteur des classes TypeDeclarationVisitor et j'appelle la méthode nbMoyenAttributsParClasse() que j'ai définie sur le visiteur et qui récupère pour chaque classe récupérée dans la variable types, le nombre de méthodes (avec classe.getMethods().length). Au moment de retourner le nombre total de méthodes de toutes les classes, je divise ce nombre par le nombre de classes donc la taille de la variable types.

1.1.6 Le nombre moyen de lignes de code par méthode

Afin de pouvoir compter le nombre de ligne de code par méthode j'ai, dans la boucle qui récupère tous les fichiers java, créé autant de visiteurs de type DeclarationMethode que de fichiers. J'accepte à chaque fois le visiteur et j'appelle ensuite la méthode nbMoyenLigneCodeParMethode() qui prend en paramètre le fichier parsé afin de pouvoir le parcourir et récupérer le numéro des lignes pour ensuite calculer le nombre de lignes de code de la méthode. Dans la méthode définie sur le visiteur, je récupère dans le fichier parsé le numéro de la ligne où commence la méthode, puis je récupère le numéro de la dernière ligne (égale à la ligne de départ + la longueur de la méthode) et je fais une soustraction pour récupérer la longueur de la méthode.

```java
public int nbMoyenLigneCodeParMethode(CompilationUnit parse){
    int nbTotalLigneCodeDansMethodes = 0;
    int debutMethode = 0;
    int finMethode = 0;
    int longueurMethode = 0;
    for(MethodDeclaration method : methods){
        debutMethode = parse.getLineNumber(method.getStartPosition());
        finMethode = parse.getLineNumber( position: method.getStartPosition() + method.getLength());
        longueurMethode = finMethode - debutMethode + 1;
        nbTotalLigneCodeDansMethodes += longueurMethode;
        System.out.println(longueurMethode);

    }
    return nbTotalLigneCodeDansMethodes;

}
```

1.1.7 Le nombre moyen d'attributs par classe

Exactement comme pour le nombre moyen de méthodes par classe, mais ctte fois je récupère le nombre d'attributs avec classe.getFields().size().

1.1.8 Les 10 pourcents des classes qui possèdent le plus grand nombre de méthodes

Pour ce cas, j'accepte tout d'abord un visiteur de classes et interfaces puis je récupère dans une ArrayList, l'appel de la méthode DixPourcentsClassesMaxMethodes() qui permet de renvoyer les 10 pourcents demandés de l'analyse. Dans le visiteur de classes et interfaces j'ai défini la méthode DixPourcentsClassesMaxMethodes() qui définit tout d'abord ce que représente 10 pourcents (un entier), puis qui fait deux forEach qui parcourent tous deux l'ensemble des classes de l'attribut types du visiteur. La première boucle récupère une classe et dans la deuxième boucle, je compare le nombre de méthodes de ma première classe au nombre de méthodes de toutes les autres classes. Si ma classe contient moins de méthodes que n autres classes (n = l'entier qui représente les dix pourcents - 1) alors je ne l'ajoute pas dans ma liste, si au contraire ma classe sélectionnée a plus de méthodes qu'au moins 3 classes par exemple et que 10 pourcents de classes du projet est équivalent à 4 classes, alors j'ajoute cette classe à ma liste des 10 pourcents.

```java
public ArrayList<TypeDeclaration> DixPourcentsClassesMaxMethodes() {

    int DixPourcentsClasses = (int) (types.size() * 0.1 + 1);
    System.out.println("(10% de " + types.size() + " classes c'est " + DixPourcentsClasses + " classe(s), si d
    //Ma liste à renvoyer
    ArrayList<TypeDeclaration> listeClasses = new ArrayList<>();
    int MoinsQue = 0;

    //Tant que j'ai pas taille = 5 (si 5 c'est 10% des classes de l'application), je continue d'ajouter
    for (TypeDeclaration classe : types) {
        MoinsQue = 0;
        //Je reparcours toutes mes classes
        for (TypeDeclaration test : types) {
            if (classe.getName() != test.getName()) {
                //Si je mets <=, les classes de même taille ne pourront pas être choisies
                if (classe.getMethods().length < test.getMethods().length) {
                    MoinsQue++;
                }
            }
        }
        //Si la classe courante fait partie des 10%, je l'ajoute
        if (MoinsQue < DixPourcentsClasses) {
            listeClasses.add(classe);
        }
    }
```

1.1.9 Les 10 pourcents des classes qui possèdent le plus grand nombre d'attributs

Comme pour le cas précédent, j'utilise la même façon de faire et au lieu de récupérer le nombre de méthodes, je récupère le nombre d'attributs

1.1.10 Les classes qui font partie des deux catégories précédentes (deux 10 pourcents de classes)

Comme pour les autres cas d'analyse, j'accepte le visiteur puis j'accepte une méthode déifnie sur lui. Là j'ai défini une méthode qui parcourt simplement une des deux ArrayList (premiers 10 pourcents) et qui regarde si la classe est présente dans l'autre ArrayList (deuxièmes dix pourcents). Si cette classe est présente, je la rajoute à mon ArrayList que je renvoie

1.1.11 Les classes qui possèdent plus de X méthodes et vous devrez nous donner X ensuite

Dans ce cas je demande à l'utilisateur de rentrer ce X. J'accepte le visiteur de classes et interfaces puis j'appelle la méthode classePlusDeXMethodes() qui est défini sur le visiteur également. Le corps étant simple je ne vais pas l'expliquer.

1.1.12 Les 10 pourcents des méthodes qui possèdent le plus grand nombre de lignes de code par classe

Cette question utilise le code utilisé dans la question 6 pour compter les lignes et le code utilisé dans les questions 8 et 9 pour récupérer les 10 pourcents de classes sauf qu'ici je récupère des méthodes.

1.1.13 Le nombre maximal de paramètres parmi ceux de toutes les méthodes

Dans ce cas j'accepte mon visiteur de méthodes puis j'appelle la méthode getMaxParametresDeMethode() qui a également un corps simple que je ne vais pas expliquer.

2 Exercice 2 : Construction du graphe d'appel d'une application

2.1 Question 1.1

Afin de construire mon graphe, je parcours à l'aide d'un visiteur sur les méthodes, l'ensemble des méthodes de mon application, puis je récupère dans chaque méthode, les invocations de méthodes. Pour chaque méthode et chaque appel de méthode je construis un noeud et entre une méthode et la méthode qui est invoquée dans son corps, je mets une arrête. Enfin, j'utilise DotExporter pour exporter le graphe en .dot puis j'utilise une API Grapheviz qui permet de convertir le .dot et .png