Rapport TP1 - Évolution et Restructuration des logiciels

HAI913I

M1 Informatique - Génie Logiciel

Faculté des Sciences de Montpellier

ROMERO Gaétan

14 octobre 2022

1 Exercice 1

1.1 Visitor

Dans notre application, il y a 4 classes utilisant le design pattern Visitor qui ont été utilisées. Parmi celles-ci, se trouve PackageVisitor, TypeDeclarationVisitor, MethodDeclarationVisitor et MethodInvocationVisitor. Respectivement, ces visitor servent à visiter les packages, classes et méthodes de l'application que l'on visite.

Notre approche consiste donc à parser une application java et en extraire des informations, à l'aide de ces visitor. Ceux-ci sont en fait des sous classes de ASTVisitor, que l'on a créées afin de rajouter des fonctionnalités à certaines classes déjà existantes dans AST JDT.

PackageDeclarationVisitor

Par exemple, notre classe PackageVisitor nous permet de récupérer la liste des packages de l'application, puis de les ajouter dans une liste pour au final afficher la liste de tous les packages de l'application. Nous avons créé une méthode *sortedPackagesList* pour palier au fait que l'on se retrouvait avec plusieurs fois le même package dans la liste. C'est à dire que dans une application, on ne peut pas avoir deux fois la même classe, ni deux fois la même méthode, mais on peut avoir deux classes qui appartiennent au même package, ce qui nous conduisait à avoir deux fois le package dans la liste. Du coup, cette méthode empêche cela de se produire.

TypeDeclarationVisitor

Dans cette classe, nous avons créé quelques méthodes pour répondre à nos besoins. Premièrement, elle contient deux attributs importants pour comprendre *List<TypeDeclaration> types* et *List<Pair> listMethodsWithLines*. L'un stocke tous les TypeDeclaration (donc les classes) et l'autre des paires de MethodDeclaration avec le nombre de lignes leur étant associées. Pour stocker les nombres de lignes avec leur méthode, on utilise la méthode *lineNumberPerMethod* :

```
1  // Add to a list a pair of a method and the lines number of it
2  public void lineNumberPerMethod(CompilationUnit parse, MethodDeclaration[] methods) {
3      int startLineNumber, endLineNumber;
4
5      for(MethodDeclaration method : methods) {
6          startLineNumber = parse.getLineNumber(method.getStartPosition());
7          endLineNumber = parse.getLineNumber(method.getStartPosition() +
8                                              method.getLength());
9          listMethodsWithLines.add(new Pair(method,
10                                           (endLineNumber - startLineNumber + 1)));
11     }
12 }
13
```

Le principe est que l'on calcule le nombre de lignes d'une MethodDeclaration en utilisant les méthodes *getLineNumber()* et *getStartPosition()* de CompilationUnit. En fait, on calcule la position de départ de la méthode et la position de fin de celle-ci, dans le fichier. Une fois cela fait, on en fait la différence pour obtenir le nombre de lignes total.

Pour afficher les 10% de méthodes avec le plus de lignes par classe, il faut utiliser la méthode *print10PourcentMethodPerClass()*, qui appelle à son tour *lineNumberPerMethod()* et *bestChoiceMethodLines()*. Le fonctionnement est simple. La méthode calcule le nombre de méthodes nécessaire pour avoir 10% de la classe, et choisit celle qui a le plus de lignes à chaque itération jusqu'à avoir le nombre de méthodes maximal pour atteindre 10%. *bestChoiceMethodLines()* s'occupe justement de faire la sélection de la méthode la plus grande, tout en vérifiant, grâce à une liste passée en paramêtre et contenant les méthodes déjà sélectionnées, que l'on ne sélectionne pas deux fois la même méthode. Il faut aussi préter attention à l'indice *indiceClass*. Son utilité est flagrante quand on réfléchit à la façon dont nous parsons les fichiers et à la façon dont sont implémentées les méthodes. Tout d'abord, l'attribut parser de type CompilationUnit sert à parser les fichiers un par un de notre application cible. Or, avec cette utilisation, si on appelle la méthode *print10PourcentMethodPerClass()* à chaque fois, on va premièrement calculer le nombre de lignes de chaque méthode plusieurs fois, mais aussi afficher les 10% plusieurs fois, et ce n'est ni optimal, ni intéressant, ni ce que nous voulons. C'est donc ici que *indiceClass* rentre en jeu. Il permet de garder en mémoire l'endroit où l'on se situe dans la liste des TypeDeclaration, et donc de ne pas refaire les calculs sur les ceux que l'on a déjà fait. Ce raisonnement reste plus ou moins identique pour les autres méthodes calculant 10%, que ce soit les 10% des classes qui ont le plus d'attributs, ou le plus de de méthodes.

MethodDeclarationVisitor

Ici, seulement deux classes méritent notre attention. Premièrement, *numberOfLinesOfMethodsPerFiles()*. Une fois que l'on a stocké toutes nos méthodes, celle-ci nous permet de compter le nombre de lignes total de toutes les méthodes de l'application et le stocker, en faisant le même calcul que celui pour compter le nombre de lignes de chaque méthode dans TypeDeclarationVisitor. Deuxièmement, il y a *printMethodWithMaxParam()*. Comme son nom l'indique, elle nous permet d'afficher la méthode avec le plus de paramètre(s). Le principe est simple : on stocke la première méthode trouvée dans une variable, puis on la compare à chaque fois à la méthode suivante, et on la remplace si elle a moins de paramètre que celle-ci.

1.2 Parser

La classe Parser est la classe principale de notre application. C'est elle qui contient le main et qui met une CLI à disposition de l'utilisateur, par l'intermédiaire de la classe UserInterface. La plupart des méthodes nécessaire au développement du TP sont contenues dans celle-ci. On y retrouve *classInfo()*, *methodInfo()*, *packageInfo()* qui permettent de faire appel à la méthode *accept()*. Ensuite, *parseFilesClass()*, *parseFilesMethod()* et *parseFilesPackage()*, qui permettent de parser les fichiers de l'application cible en appelant les bonne méthodes en fonction de nos besoins. Par la suite, il y en a des plus spécifiques comme *BestMethodPerClass()* qui appelle la méthode *print10PourcentMethodPerClass()* lors de chaque itération du parsage de nos fichiers. On peut voir son contenu ci-après :

```
1  public static void BestMethodPerClass(TypeDeclarationVisitor visitorClass,
2                                         ArrayList<File> javaFiles)throws IOException {
3      System.out.println("Les 10% des méthodes qui ont le plus grand nombre de lignes,
       par classe");
4      for (File fileEntry : javaFiles) {
5          String content = FileUtils.readFileToString(fileEntry);
6          parse = parse(content.toCharArray());
7          classInfo(visitorClass);
8          visitorClass.print10PourcentMethodPerClass(parse);
9      }
10 }
11
```

Pour ce qui est de la CLI, le choix de l'utilisateur quant à lui est géré par un switch case qui récupère l'entrée clavier par le biais d'un Scanner, et agit en fonction. Le reste est implémenté par la classe UserInterface qui met à disposition une liste des questions avec leur numéro, pour que l'utilisateur puisse intéragir simplement en inscrivant un numéro, et choisir l'exercice et la question adéquate.

2 Exercice 2

Pour cet exercice, nous avons utilisé la librairie JGraphT pour réaliser notre graphe d'appel et l'exporter en .dot, ainsi que l'API Graphviz pour transformer le .dot en png.

Pour construire notre graphe, nous avons dû créer deux classes. L'une, Node, nous permet de créer des objets pour les noeuds du graphe et l'autre, Edge, s'occupe des arêtes de notre graphe. Avec cela, il suffit de stocker les noeuds et les arêtes en récupérant toutes les méthodes dans chaque classe, ainsi que les appels à d'autres méthodes à l'intérieur de celles-ci, puis de créer un graphe de type *Graph<String, DefaultEdge>* avec la librairie JGraphT, et d'y ajouter les noeuds et les arêtes, de la manière suivante :

```
1  List<Node> listNode = new ArrayList<Node>();
2  List<Edge> listEdge = new ArrayList<Edge>();
3
4      for(TypeDeclaration type : visitorClass.getTypes()) {
5
6          for(MethodDeclaration method : type.getMethods()) {
7
8              MethodInvocationVisitor visitor2 = new MethodInvocationVisitor();
9              method.accept(visitor2);
10
11             Node nodeMethodDecla = new Node(type.getName().toString() + "." +
12                                             method.getName().toString());
13             listNode.add(nodeMethodDecla);
14
15
16             if (visitor2.getMethods().size() != 0) {
17 //              System.out.println("\nMethod : " + nodeMethodDecla.getNode());
18
19                 for (MethodInvocation methodInvocation : visitor2.getMethods()) {
20
21                     Node nodeMethodInv = new Node(type.getName().toString() + "." +
                                                   methodInvocation.getName().toString());
22                     listNode.add(nodeMethodInv);
23                     listEdge.add(new Edge(nodeMethodDecla.getNode(),
24                                           nodeMethodInv.getNode()));
25
26 //                  System.out.println(" --- calls : " + nodeMethodInv.getNode());
27                 }
28             }
29         }
30     }
31     // Add vertex and edges to the graph
32     Graph<String, DefaultEdge> graph = new DefaultDirectedGraph<String,
33                                                 DefaultEdge>(DefaultEdge.class);
34     for (Node node : listNode) {
35         graph.addVertex(node.getNode());
36     }
37     for (Edge edge : listEdge) {
38         graph.addEdge(edge.getNodeSource(), edge.getNodeTarget());
39     }
40
```

On voit que les méthodes utilisées pour ajouter les noeuds et les arêtes sont : *addVertex()* et *addEdge()*.

De même, pour exporter en .dot, on utilise :

```
DOTExporter<String, DefaultEdge> exporter = new DOTExporter<String, DefaultEdge>();
exporter.setVertexAttributeProvider((v) -> {
    Map<String, Attribute> map = new LinkedHashMap<String, Attribute>();
    map.put("label", DefaultAttribute.createAttribute(v.toString()));
    return map;
});

```

Il s'agit encore aussi d'un type de la librairie JGraphT.
Enfin, pour transformer le tout en .png, on utilise une API de Graphviz :

```
Writer writer = new StringWriter();
exporter.exportGraph(graph, writer);
MutableGraph g = new guru.nidi.graphviz.parse.Parser().read(writer.toString());
Graphviz.fromGraph(g).height(1000).render(Format.PNG).toFile(
                                                new File("example/callGraph.png"));

```

3 Lien Github

Lien du dépôt Github