TP2 : Compréhension des programmes
Évolution et restructuration des logiciels

Mohamad Satea Almallouhi Tony Nguyen

10 novembre 2024

Résumé

Rapport d'exercice sur l'analyse d'un programme par l'analyse statique et de la notion de couplage afin d'en déduire des modules

Table des matières

List of Algorithms

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. L'étape d'extraction des informations a été réalisé précédement. Nous nous trouvons à présent dans l'étape de traitement des propriétés dans le workflow. Nous allons étudier le concept de couplage des classes.

Tout d'abord, à partir du travail précédent, nous allons nous servir du graph d'appel des méthodes écrites dans les classes du projet analysé. Cela nous permettra de calculer le couplage entre les différentes classes.

Ainsi, grâce au graph de couplage, nous allons partitioné notre ensemble de classes en différent modules.

Démonstration vidéo

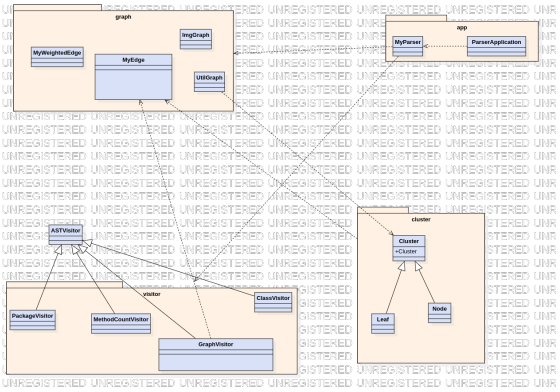
En ligne sur Youtube, à l'adresse URL `https://youtu.be/4WYid4mVgWk` une démonstration vidéo de notre travail.

Installation

Vous trouverez les instructions dans le README.md

To Do
— expliqué la solution implémenté
— add code picture
— add resultat picture
— diagram class visiteur
— diagram class de l'app
— definition du couplage avec écritures math DONE
— tous les algo en latex stylé DONE
— →!!! → vidéo ←!!! ←
Faire une vidéo, le rapport avec des screenshot des résultats et du code et enfin un read.md(instruction). En plus, pour le bonus, faire une belle application, des tests unitaires, faire le rapport en Latex.

1 UML (juste pour montrer l'organisation)




```java
public boolean visit(MethodDeclaration node) {
    // Récupérer le nom de la méthode
    String methodName = node.getName().getIdentifier();

    // Récupérer le nom de la classe en parent
    TypeDeclaration typeDecl = (TypeDeclaration) node.getParent();
    String className = typeDecl.getName().getIdentifier();

    // Combiner le package, la classe et la méthode
    String fullMethodName = currentPackage + "." + className + "::" + methodName;
      System.out.println("Nom complet de la méthode: " + fullMethodName);

    g.addVertex(fullMethodName);

    // Parcourir le corps de la méthode pour trouver les MethodInvocation
    node.getBody().accept(new ASTVisitor() {
        @Override
        public boolean visit(MethodInvocation methodInvocation) {
            // Récupérer le nom de la méthode appelée
             String calledMethodName = methodInvocation.getName().getFullyQualifiedName();



            IMethodBinding methodBinding = methodInvocation.resolveMethodBinding();

            if (methodBinding != null) {
                // Récupérer le nom de la méthode appelée
                String methodName2 = methodBinding.getName();

                // Récupérer la classe ou l'interface dans laquelle la méthode est définie
                ITypeBinding declaringClass = methodBinding.getDeclaringClass();
                if (declaringClass != null) { // isExternalMethod
                    String declaringClassName = declaringClass.getQualifiedName();
                    String source = fullMethodName;
                    String target = declaringClassName+"::"+methodName2;

                    // trick : ajout de la source et la cible -> ajout de l'arête plus facile
                    g.addVertex(source);
                    g.addVertex(target);


                    MyEdge e = g.addEdge(source, target);

                    e.setClassNameSource(currentPackage + "." + className);
                    e.setMethodNameSource(methodName);

                    e.setClassTargetSource(declaringClassName);
                    e.setMethodNameTarget(methodName2);
                }
            } else {
                System.out.println("resolveMethodBinding not succesful");
            }

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

2 Graphes

Nous nous intéressons au couplage entre les classes de notre application. Il serait logique de réunir les classes fortement dépendantes les unes des autres. De la même façon, les classes qui n'ont aucun rapport entre elle, pour des raisons de clarté, peuvent être isolé.

2.1 Algorithmes de création du graphe d'appel

La construction de ce graph est la base de ce travail. Il nous permettra ensuite de calculer le couplage entre les classes ...

Lors du parcours de l'arbre syntaxique abstrait (AST), quand on atterit sur un noeud qui correspond à une méthod, on ajoute un noeud au graph et si il y a une méthode invoqué interne au projet, on l'ajoute au graph et on crée une arête.

Les méthodes qui se surchage entre elles (les méthodes ayant le même nom dans une classe mais avec une signature différente) sont confondues.

L'algorithme n°1 décrivant cela se trouvre à la page 6

Résultat Nous obtenous ainsi le graph suivant :

2.2 Le couplage

Définition Étant donné une application composé de n classes. Le nombre de méthode d'une classe est noté nbMethod.

$nbMethodTotal = \sum_{i=0}^{n} nbMethod_i$

$nbRelationBinaire = nbMethodTotal * nbMethodTotal$

$nbRelation_{A->B} \neq nbRelation_{B->A}$

$nbRelation(A,B) = nbRelation_{A->B} + nbRelation_{B->A}$

$Couplage(A,B) = nbRelation(A,B)/nbRelationBinai$

2.3 Graphe de couplage inter-classes

À partir du graphe d'appel, nous allons maintenant créer le graph pondéré par le couplage entre les différentes classes.

Explication Pour cela, nous allons créer un graph pondéré où les sommet seront les class et les arête auront un poid égal au couplage entre les class.

L'algorithme n°2 décrivant cela se trouvre à la page 6

```
public static SimpleWeightedGraph<String, MyWeightedEdge> getGraphCouplage(
        Set<String> classNameSet,
        DefaultDirectedGraph<String, MyEdge> masterGraph,
        int nbMethodTotal) {

    // instanciation d'un graph pondéré
    SimpleWeightedGraph <String, MyWeightedEdge> graphPondere =
            new SimpleWeightedGraph<>(MyWeightedEdge.class);

    // ajout des différentes classes de l'application
    classNameSet.stream().forEach(aClassName -> graphPondere.addVertex(aClassName));

    // ---- Produit cartésien de 2 ensembles privé des doublons
    for (String s : classNameSet) {
        for (String t : classNameSet) {
            if (!s.equals(t)) {
                // ----

                System.out.println("("+s+", "+t+")");

                aux_createEdge(masterGraph, graphPondere, nbMethodTotal*nbMethodTotal, s, t);
            }
        }
    }

    return graphPondere;
}
```

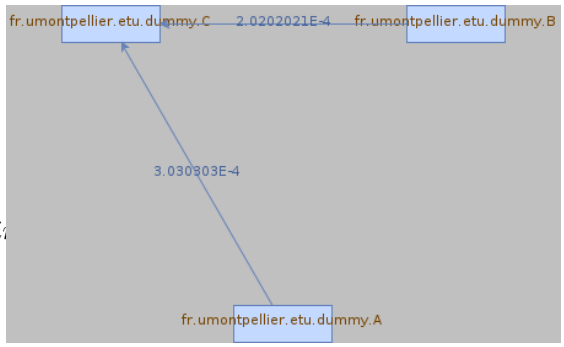
```
private static void aux_createEdge(
        DefaultDirectedGraph<String, MyEdge> masterGraph,
        SimpleWeightedGraph <String, MyWeightedEdge> graphPondere,
        int nbMethodTotal,
        String s,
        String t) {

    float n = UtilGraph.nbRelation(masterGraph, t, s);

    float nbTotal = nbMethodTotal*(nbMethodTotal-1);

    float res = n / nbTotal;

    if (n != 0) {
        MyWeightedEdge e = graphPondere.addEdge(s, t);
        if (e != null) { e.setWheight(res); }
    }
}
```

```
public static int nbRelation(DefaultDirectedGraph<String, MyEdge> g, String A, String B) {
    int nbAppel = 0;

    nbAppel += g.edgeSet().stream().filter(e -> e.getClassNameSource().equals(B)).filter(e -> e.getClassNameTarget().equals(A)).count();
    nbAppel += g.edgeSet().stream().filter(e -> e.getClassNameSource().equals(A)).filter(e -> e.getClassNameTarget().equals(B)).count();

    return nbAppel;
}
```

Remarquons que le graph d'appel nous aide à calculer le couplage.

Par la suite, le graphe pondéré de couplage entre les classes ne sera pas exploité, seul le graph d'appel sera utilisé.

Résultat Nous obtenous ainsi le graph suivant :



3 Clusturing

Nous allons maintenant voir comment nous avons rassemblé les classes entre elles.

3.1 Algorithme de clustering

Rassemblons les classes les plus proches entre elles à l'aide du couplage et créons un arbre dendrogramme

Cluster L'algorithme n°3 décrivant cela se trouvre à la page 7 avec l'implémentation en java :

```
public static Cluster clustering_hierarchique(
        DefaultDirectedGraph<String, MyEdge> g,
        Set<String> classNameSet) throws IOException {

    // l'arbre dendrogramme qui représentes les différents clusters
    ArrayList<Cluster> clusters = new ArrayList<>(); // au début, toutes les classes sont des clusters de 1


    classNameSet.stream().forEach(x ->
            clusters.add(new Leaf(x))
    );

    Node root = null;

    int i = 0;
    while (i < clusters.size()) {
        Node n = Cluster.clusterProche(clusters, g);

        clusters.remove(n.getLeftChild());
        clusters.remove(n.getRightChild());

        clusters.add(n);

        root = n;
        i++;
    }

    return root;
}
```

```
public static Node clusterProche(
        ArrayList<Cluster> clusters,
        DefaultDirectedGraph<String, MyEdge> g) {

    assert clusters.size() > 1;

    Set<String> setClassNameA, setClassNameB;
    Node bestCluster = null;
    int max = -1;

    for (Cluster c1 : clusters) {
        for (Cluster c2 : clusters) {
            if (c1 != c2) {
                setClassNameA = new HashSet<>();
                setClassNameB = new HashSet<>();

                //mettre les noms des classes du clusters c1 dans son set
                setClassNameA = c1.getNames();
                setClassNameB = c2.getNames();

                // comparer
                int nbRelation = UtilGraph.nbRelation(g, setClassNameA, setClassNameB);

                if (nbRelation > max) {
                    max = nbRelation;
                    bestCluster = new Node(c1, c2, nbRelation);
                }
            }
        }
    }

    return bestCluster;
}
```

```
public static int nbRelation(DefaultDirectedGraph<String, MyEdge> g, Set<String> setA, Set<String> setB) {
    int nbAppel = 0;

    nbAppel += g.edgeSet().stream().filter(e -> setB.contains(e.getClassNameSource())).filter(e -> setA.contains(e.getClassNameTarget())).count();
    nbAppel += g.edgeSet().stream().filter(e -> setA.contains(e.getClassNameSource())).filter(e -> setB.contains(e.getClassNameTarget())).count();

    return nbAppel;
}
```

Remarque Nous utilisons uniquement nbRelation(A,B) et non pas Couplage(A,B). De plus, A et B représente ici un ensemble de class.

3.2 Identification des modules / partitionnement des classes

Cluster L'algorithme n°3 décrivant cela se trouvre à la page 7.

```
public static ArrayList<Set<String>> getModule(double cp, Cluster root, int nbClasses) {
    //Descendre dans le dendo depuis le noeud racine.
    Si la cpValue du Noeud courant est inférieur à CP, on essaye de descendre.
    // quand le cp du noeud courant est supérieur, s'arrêter et partitioner en module


    ArrayList<Set<String>> result = new ArrayList<>();
    aux_getModule(root, result);

    return result;
}
private static void aux_getModule(Cluster root, List<Set<String>> res) {
    if (root.check(2.5)) {
        res.add(root.getNames());
    } else {
        aux_getModule(root.getLeftChild(), res);
        aux_getModule(root.getRightChild(), res);
    }
}
```

```
/**
 * Vérifie si le cluster this possède une valeur cp acceptable pour pouvoir les unifiés
 * @param i
 * @return
 */
public abstract boolean check(double i);
```

```
@Override
public boolean check(double i) {
    return (this.cpValue >= i);
}
```

```
@Override
public boolean check(double i) {
    return true;
}
```

4 Spoon

L'un des problème rencontrer est de trouver un moyen pour savoir si une méthode est interne ou externe au programme.

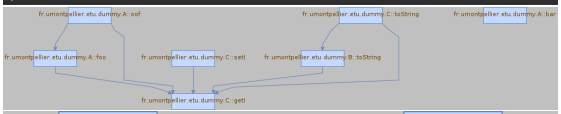
La solution que nous avons trouvé est de comparer avec le package de la class où la méthode est déclarée.

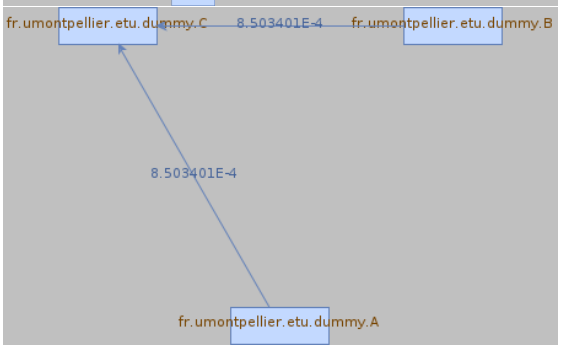
On remarque que le résultat différe entre Spoon et jdt.core.dom.AST. C'est due au fait que Spoon n'a capturé les constructeurs comme des méthodes. Si on regarde les graph d'appel, ils sont identique à l'exception des constructeurs.

```
DefaultDirectedGraph<String,MyEdge> graphAppel = uneApp.getGraphAppel();
ImgGraph.writeGraphIntoImgTopToBottomNoLabel(graphAppel, "target/img/","graphAppel.png");

SimpleWeightedGraph <String, MyWeightedEdge> graphPondere =
        UtilGraph.getGraphCouplage(
                uneApp.getClassNameSet(),
                graphAppel,
                uneApp.model.getElements(new TypeFilter<>(CtMethod.class)).size());
ImgGraph.showGraphPond(graphPondere,"target/img/","graphPondereCouplageClass.png");
```

```
public static boolean isExternalMethod(String declaringType) {
    // Supposons que votre application a un package racine "com.monapp"
    // Vous pouvez ajuster ce check selon votre projet
    return !declaringType.startsWith("fr.umontpellier.etu");
}
```

```
private DefaultDirectedGraph<String,MyEdge> getGraphAppel() {

    DefaultDirectedGraph<String,MyEdge> graphAppel = new DefaultDirectedGraph<String, MyEdge>(MyEdge.class);


    // Filtre pour obtenir les classes
        for (CtClass<?> clazz : this.model.getElements(new TypeFilter<>(CtClass.class))) {

            // Pour chaque méthode dans la classe
            for (CtMethod<?> method : clazz.getMethods()) {
                graphAppel.addVertex(
                        clazz.getPackage().getQualifiedName()+"."+clazz.getSimpleName()+"::"+method.getSimpleName());

                // Filtrer pour obtenir les appels de méthodes dans cette méthode
                for (CtInvocation<?> invocation : method.getElements(new TypeFilter<>(CtInvocation.class))) {
                    CtExecutableReference<?> executable = invocation.getExecutable();
                    String methodName = executable.getSimpleName();
                    String declaringType = executable.getDeclaringType().getQualifiedName();

                    // Vérifier si la méthode appartient à votre projet ou à une lib externe
                    if (!isExternalMethod(declaringType)) {
                        graphAppel.addVertex(declaringType+"::"+methodName);
                        MyEdge e = graphAppel.addEdge(
                                clazz.getPackage().getQualifiedName()+"."+clazz.getSimpleName()+"::"+method.getSimpleName(),
                                declaringType+"::"+methodName);

                        if (e!=null) {
                         e.setClassNameSource(clazz.getPackage().getQualifiedName()+"."+clazz.getSimpleName());
                         e.setMethodNameSource(method.getSimpleName());

                         e.setClassTargetSource(declaringType);
                         e.setMethodNameTarget(methodName);
                        }
                    }
                }
            }
        }

    return graphAppel;
}
```





Algorithm 1: An algorithm to make call graph

```
Data: programme
Result: graphe d'appel
g ← new Graph();
for each class c do
    for each method m implemented in c do
        for each method invoqued i in m do
            if !isExternalMethod(i) then
                g.addSommet(m);
                g.addSommet(i) ;                    /* ajout des sommets m et i */
                g.addVertex(m, i) ;                 /* ajoute une arete de m vers i */
            end
        end
    end
end
return g;
```

Algorithm 2: An algorithm to make a weighted graph corresponding to the coupling between class

```
Data: le graphe d'appel
Result: graphe pondere
graphPondere ← new GraphPondere();
/* Produit cartésion entre toutes les classes de l'application en retirant les
   couples identiques (x,x)                                                    */
for each String aClassName1 in classNameSet do
    for each String aClassName2 in classNameSet do
        if aClassName1 != aClassName2 then
            cpValue ← calculCouplageValueEntre(aClassName1, aClassName2);
            if cpValue > 0 then
                graphPondere.addArete(aClassName1, aClassName2, cpValue);
            end
        end
    end
end
return graphPondere;
```

Algorithm 3: Clustering algorithm (Creating the Dendrogramme)

UML INSERT UML CLASS DIAGRAM OF Cluster composite pattern
Data: *graph d'appel*
Result: *the dendrogramme*
`/* Stratégie: toutes les class sont leurs propres cluster. On va essayer de fusioner les clusters entre eux en fonction du couplage */`
clusters ← *Cluster*[];
for *each String aClassName in classNameSet* **do**
  clusters.add(*new Leaf*(*aClassName*));
end
i ← 0;
Noderoot ← *null*;
while *i* < *size(clusters)* **do**
  n ← *laFusionEntreLes*2*CLustersLesPlusProche*();
  on retire les 2 *enfants du noeud n*;
  clusters.add(*n*);
  i + +;
end
return root;

Algorithm 4: Identification des modules

Data: *cluster root*
Result: *Partitionnement en modules*
result`/* une liste d'ensemble de String */`
auxGetModule(*root*, *result*);
returnresult;

Algorithm 5: auxGetModule

Data: *dendrogramme root*
Result: *Partitionnement en modules*
if *root possède une valeur de couplage suffisante ou si c'est une feuille* **then**
  on a trouvé un module
else
  on fait la même chose au enfants de root de façon récursive
end