Student Name: ROMERO Gaétan
Student ID: 21701877

Rapport TP2
HAI913I

Fonctionnement

Premièrement, il faut savoir que l'application se base sur le graphe créé lors du TP1. C'est à dire que l'on utilise le fichier .dot du graphe et on le transforme en une structure de graphe que l'on peut manipuler. Pour des raisons de simpliciter, ce fichier se trouve déjà dans `src/main/resources` mais on peut tout à fait relancer le tp1 sur une autre application pour obtenir un autre graphe. Il faut cependant savoir qu'il faut ajouter une ligne au tp1 pour enregistrer le graphe créé au format DOT, dans un fichier .dot. La voici :

```
exporter.exportGraph(graph, new File("example/callGraph.dot")); // To DOT File
```

Ensuite, il suffit de lancer le main et de suivre le déroulement de la CLI :

- En premier nous avons le couplage entre deux classes qui affichera donc le couplage avec 5 décimaux maximum après la virgule.
- En deuxième, nous avons le graphe de couplage de l'application qui va sauvegarder le tout dans un fichier qui se trouvera dans `/example` .
- En troisième, nous avons le clustering. Celui ci réalise l'exercice 2 en une seule fois. C'est à dire qu'il fait le clustering des classes de l'application et ensuite, demande un couplage minimum puis trie parmi les clusters trouvés juste avant pour respecter les contraintes imposées par l'exercice et retourne une liste de clusters.
- En dernier, nous avons Spoon. En fait, ca réalise exactement les trois premiers mais avec Spoon.
- Si nous souhaitons terminer la cli, il suffit de choisir la dernière option, qui est la 5ème.

Exercice 1

Couplage

Le principe est simplement que l'on a une liste de couples de classes appelantes et de classes appelées concaténées avec leurs méthodes. Lors de la recherche du couplage entre deux classes, on parcours juste cette liste de couples (ou paires) et si une paire contient le nom des deux classes que l'on recherche, on incremente un indice qui représente le nombre d'appel entre les deux. A la fin, on retourne cet indice divisé par le nombre total d'éléments de la liste.

Graphe de couplage

Pour réaliser le graphe, nous appelons simplement sur chaque couple de classes possible, la méthode de couplage entre les deux. Puis, à la fin, on exporte simplement le graphe en PNG en affichant bien le couplage comme poids des arêtes entre les classes, qui correspondent aux noeuds du graphe. Voici comment on exporte en précisant le poids des arêtes avec Graphviz.

```
// It allows us to limit a double to 5 decimal places.
    DecimalFormat sizeCoupling = new DecimalFormat();
    sizeCoupling.setMaximumFractionDigits(5);

    // We use the DOTExporter to export the graph in DOT Format and PNG Format
    DOTExporter<String, DefaultWeightedEdge> exporter = new DOTExporter<String,
        DefaultWeightedEdge>();
    //Put vertices in a form which is known by the DOT Format -> "label"
    exporter.setVertexAttributeProvider((v) -> {
      Map<String, Attribute> map = new LinkedHashMap<String, Attribute>();
      map.put("label", DefaultAttribute.createAttribute(v.toString()));
      return map;
    });
    // Put edges in a form which is known by the DOT Format -> "weight"
    // And in order to print it in the graph we create a "label" with his value
    exporter.setEdgeAttributeProvider((v) -> {
      Map<String, Attribute> map = new LinkedHashMap<String, Attribute>();
      map.put("weight", DefaultAttribute.createAttribute(graph.getEdgeWeight(v)));
      map.put("label", DefaultAttribute.createAttribute(sizeCoupling.format(graph.
          getEdgeWeight(v))));
      return map;
    });
    Writer writer = new StringWriter();
    exporter.exportGraph(graph, writer);
    exporter.exportGraph(graph, new File("target/couplingGraph.dot")); // To DOT File
    MutableGraph g = new guru.nidi.graphviz.parse.Parser().read(writer.toString());
    Graphviz.fromGraph(g).height(1000).render(Format.PNG).toFile(new File("target/
        couplingGraph.png"));
```

Section 2

Clustering

Le principe est que l'on compare tous les couples possibles dans la liste de clusters déjà trouvés, et le couple avec le plus de couplage est transformé en un cluster, et on supprime les deux anciens. Pour créer un cluster, on concatène les deux anciens clusters, car il s'agit de string. Si lors de la recherche du cluster avec le plus de couplage expliquée précédemment, un des deux clusters est une concaténation, on découpe la string en tokens correspondant chacun à une classe, et on ajoute le couplage de chaque classe pour obtenir le couplage du cluster entier.

Ce procédé est réitéré autant de fois que l'on peut créer des clusters. A la fin, on a soit un seul cluster, soit plusieurs si ils ne sont pas couplés entre eux.

Clustering avec couplage minimum

Les contraintes imposées sont que le nombre de clusters rendus doit être inférieur au nombre de classes de l'application divisé par 2, que chaque cluster ne contient des classes que d'une seule branche du dendogramme, c'est à dire que d'un seul clusters trouvé précédemment ou de ses sous clusters, et dernièrement, que la moyenne du couplage du cluster soit inférieur à un nombre rentré par l'utilisateur.

Pour ce faire, nous avons donc d'abord trié la liste des clusters en supprimant ceux qui ont un couplage inférieur au nombrre choisi par l'utilisateur, puis enlevé les clusters qui étaient contenus dans d'autres, et pour finir, si le nombre de clusters est encore trop élevé, enlevé ceux qui sont les plus petits.

Exercice 3

Le fonctionnement est identique que les 2 premiers exercices. Le seul aspect qui change est que la liste de couples de classes que la méthode utilise n'est plus créée à partir du fichier .dot expliqué au début, mais en utilisant Spoon. C'est à dire que l'on reprend le chemin vers les fichiers d'une application que l'on veut parser, et on lit chaque fichiers avec les méthodes internes à Spoon pour les parser et les transformer en AST, que l'on peut ensuite utiliser pour recréer la liste de couples de classes que l'on avait précédemment.

```
1  public class SpoonParser {
2
3    private static boolean spoonOrNot = false;
4
5    // Return a list formed with pairs of classes
6    // Each pair is like that : (calling class, called class)
7    public static ArrayList<Pair<String, String>> getListPairFromFileWithSpoon (ArrayList<
         File> files) {
8
9      ArrayList<CtModel> models = new ArrayList<CtModel>();
10
11     for (File file : files) {
12
13       Launcher launcher = new Launcher();
14
15       // addInputResource() add a resource, so a file or a folder, in the launcher
16       // It allows us to add as many resources as we want in the launcher
17       // In order to parse them
18       launcher.addInputResource(file.getPath());
19       models.add(launcher.buildModel());
20     }
21     return takeModelAndreturnList(models);
22
23   }
24
25   // Take a List of CtModel created with a spoon launcher
26   // And return a list of pairs of classes
27   protected static ArrayList<Pair<String, String>> takeModelAndreturnList(ArrayList<
         CtModel> models) {
28
```

```
    ArrayList<Pair<String, String>> listPairs = new ArrayList<Pair<String, String>>();

    // We get each class in each model
    // Then we search each time a method is called into each method of the class
    // And we create a pair
    for (CtModel model : models) {

      for (CtType<?> type : model.getAllTypes()) {

        for (Object method : type.getAllMethods()) {

          CtMethod<?> ctmethod = null;

          if (method instanceof CtMethod) {

            ctmethod = (CtMethod<?>) method;
          }
          if (ctmethod != null) {

            String methodName = ctmethod.getSimpleName();

            // We filter the children in order to only keep the invocation ones
            ctmethod.filterChildren(new TypeFilter<CtInvocation<?>>(CtInvocation.class)).
                forEach(inv -> {
              CtInvocation<?> newInvocation = (CtInvocation<?>) inv;

              // If the method called is not in the same class that the one calling
              if (newInvocation.getExecutable().getSimpleName() != type.getSimpleName())
                  {
                if (newInvocation.getExecutable().getDeclaringType() != null) {

                  listPairs.add(new Pair<String, String>(type.getSimpleName() + "." +
                      methodName,
                      newInvocation.getExecutable().getDeclaringType().getSimpleName() +
                      "." + newInvocation.getExecutable().getSimpleName()));
                }
              }
            });
          }
        }
      }
    }
    return listPairs;
  }
}
```