HAI913I
Rapport TP1 Partie 2

SAID Adam
COSSU Arnaud

Table des matières

1 Introduction

1.1 Préambule

Ce rapport présente le travail effectué pour le TP1 Partie 2 de l'UE HAI913I. Il s'agit ici de la création d'un outil d'analyse de programmes `Java` permettant d'en déduire certaines informations comme le nombre de classes, de lignes de codes, de méthodes...
Vous retrouverez ici les différents éléments développés pour notre solution, un guide d'utilisation de l'application et les liens vers notre répertoire avec le code source et le lien vers une vidéo de démonstration.

Répertoire GitHub du code source

Vidéo de démonstration

1.2 Guide d'utilisation

Voici les étapes pour lancer et utiliser notre outil :

1. Ouvrez le projet source téléchargeable depuis notre GitHub avec votre IDE.
2. Si nécessaire résolvez les problèmes de build path en ajoutant `commons-io-2.4.jar` présent à la racine, au build path du projet.
3. Lancez le fichier `CodeAnalyserGUI.java`
4. Le projet étant lancé, vous vous retrouvez avec une GUI
5. Cliquez sur le bouton en haut "Choose directory" en sélectionnez un projet Java
6. Cliquez en suite sur "Analyse" pour lancer l'analyse
7. Au bout de quelques secondes vous aurez toutes les statistiques de votre projet

Liste des informations affichées par l'outil :
Informations numériques :

— Nombre de classe
— Nombre de lignes de code
— Nombre de méthode
— Nombre de package
— Nombre moyen de lignes par méthode
— Nombre moyen de méthodes par classe
— Nombre moyen d'attributs par classe
— Nombre max de paramètres par classe

Graphiques :
— Les 10% de classes avec le plus de méthodes
— Les 10% de classes avec le plus d'attributs
— Les 10% de méthodes les plus longues
— Les 10% qui ont le plus de méthodes et le plus d'attributs

En bas de page vous retrouverez une CLI avec plus de détails, notamment le nom de ces méthodes et attributs.

Vous pouvez afficher le graphe d'appel en appuyant sur le bouton orange "See graph".

2 Calcul statistique pour une application OO

Pour la première partie du projet, nous avons créé des méthodes faisant appel à des classes visiteurs, ce qui permet d'analyser ces classes en profondeur. Suivant le but de la méthode, elle va récupérer la liste des fichiers sources et boucler sur chacune des classes pour en extraire les informations importantes. Voici un exemple permettant la récupération des méthodes du projet.

```
public static List<String> printMethodsInfo(CompilationUnit parse) {

            MethodDeclarationVisitor visitor1 = new MethodDeclarationVisitor();
            parse.accept(visitor1);
            List<String> methodsList = new ArrayList<String>();
            for (MethodDeclaration methode : visitor1.getMethods()) {
                    methodsList.add(methode.getName().toString());
            }
            return methodsList;
    }
```

On peut y voir la récupération du visiteur et son parcours des méthodes. Ces méthodes sont en suite appelées sur le projet et leur résultat est stocké en mémoire.
Après analyse les différentes informations sont envoyées à la GUI ou bien à la CLI à travers des méthodes permettant un affichage dynamique comme par exemple :

```
public static void displayListString(String nomObjet, List<String> listObjet) {
            System.out.println("Nombre de "+ nomObjet +": "+ listObjet.size());
            for (int i = 0; i < listObjet.size(); i++) {
                    System.out.println("-> Nom "+nomObjet+": "+listObjet.get(i));
                    cmd += "-> Nom "+nomObjet+": "+listObjet.get(i) + "\n";
            }
    }
```

Ces méthodes génériques permettent un affichage dynamique suivant le type d'objet.

3 Construction du graphe d'appel d'une application

Pour la seconde partie, nous avons travaillé comme dans la précédente. Nous avons mis en place une méthode qui parcours toutes les classes et pour chacune va lister les méthodes ainsi que les appels de méthodes qu'elles contiennent, en récupérant la classe

à laquelle elles appartiennent. Voici ladite méthode :

```
public static Map<String, Map<String, Map<String, String>>>
buildCallGraph(CompilationUnit parse) {
    ClassDeclarationVisitor classVisitor = new ClassDeclarationVisitor();
    parse.accept(classVisitor);

    Map<String, Map<String, Map<String, String>>> callGraph =
    new HashMap<String, Map<String, Map<String, String>>>();

    for (TypeDeclaration classDeclaration : classVisitor.getClasses()) {
        String className = classDeclaration.getName().getIdentifier();
        Map<String, Map<String, String>> methodCalls =
        new HashMap<String, Map<String, String>>();

        MethodDeclarationVisitor methodVisitor = new MethodDeclarationVisitor();
        classDeclaration.accept(methodVisitor);

        for (MethodDeclaration methodDeclaration : methodVisitor.getMethods()) {
            String methodName = methodDeclaration.getName().getIdentifier();
            Map<String, String> calledMethods =
            new HashMap<String, String>();

            MethodInvocationVisitor invocationVisitor =
            new MethodInvocationVisitor();
            methodDeclaration.accept(invocationVisitor);

            ...

            methodCalls.put(methodName, calledMethods);
        }

        callGraph.put(className, methodCalls);
    }

    return callGraph;
}
```

Cette méthode récupère toutes les informations en bouclant sur chacun des types d'objets (classes et méthodes) avant de tout stocker dans une Map.

4 Interface Graphique du projet

4.1 Technologie utilisée

Pour la conception de cette interface utilisateur, nous avons opté pour l'utilisation de Swing avec WindowBuilder, ainsi que JFreeChart pour générer des graphiques correspondant aux dernières données requises dans notre analyse. La visualisation sous forme de graphiques favorise une meilleure interaction avec les données chiffrées, facilitant ainsi leur compréhension. La GUI est divisée en quatre sections distinctes, lesquelles sont les suivantes :

1. Partie **Chiffres Clés** : Comprends les chiffres clés de l'analyse comme le nombre de lignes de code du projet, le nombres de classes et d'autres informations utiles. Cette partie est mise en avant afin de faciliter la lecture de l'analyse.
2. Partie **Graphes** : L'utilisation de graphiques permet de présenter de manière concise plusieurs données combinées. En effet, diverses analyses produisent des résultats regroupés en différentes classes. Cette représentation graphique offre également la possibilité d'obtenir des informations plus détaillées, telles que le nombre précis de classes ou de méthodes pour chaque ensemble de données.
3. Partie **Trace d'exécution (cmd)** : Cette section offre la possibilité de parcourir la trace d'exécution de l'analyse, ce qui permet d'examiner de manière plus approfondie les données récupérées dans les projets.
4. Partie **Graphe d'appel** : Cette section est présentée dans une fenêtre distincte qui affiche, sous forme de mise en page textuelle, le graphe d'appel du projet.

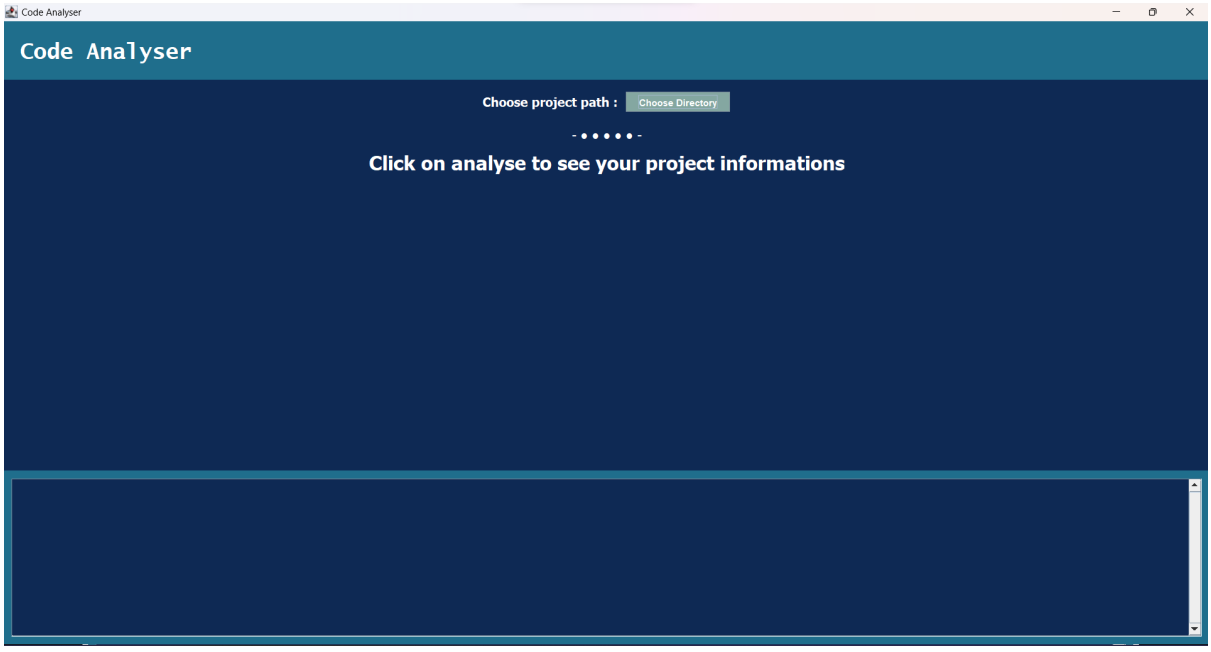

FIGURE 1 – Page d'accueil de la GUI

4.2 Représentation sous forme de graphes

Dans le cadre de ce projet, nous avons voulu réaliser une interface graphique utilisateur (GUI) utilisant des graphes, fonctionnalité clé de cette GUI permettant la représentation graphique des données, réalisée grâce à la bibliothèque JFreeChart.

Cette section 'Représentation sous forme de graphes', offre une visualisation claire et informative des résultats de l'analyse. Elle présente graphiquement les 10% de classes avec le plus de méthodes, les 10% de classes avec le plus d'attributs, les 10% de méthodes les plus longues, ainsi que les 10% de classes qui cumulent à la fois le plus grand nombre de méthodes et d'attributs.

Grâce à JFreeChart, nous sommes en mesure de créer des graphiques interactifs et dynamiques qui permettent aux utilisateurs d'explorer les données de manière plus rapide. Cette représentation graphique facilite la compréhension des données extraites et des points clés de l'analyse, ce qui s'avère extrêmement utile pour les utilisateurs analysant les projets Java.

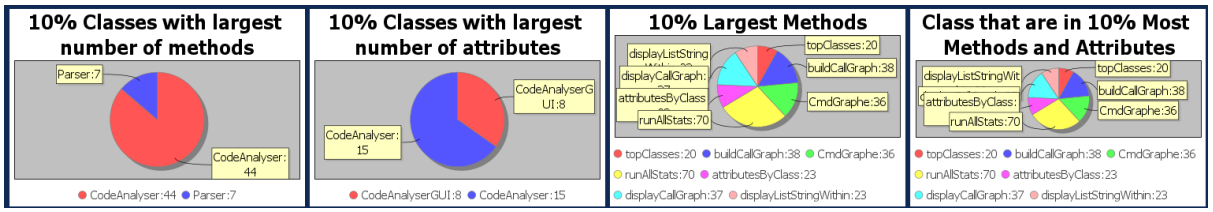

FIGURE 2 – Exemples de graphes générés lors d'une analyse

À noter que tous ces graphes sont dynamiques, et sont générés lors de l'analyse, ils sont donc totalement modelables.

4.3 Affichage graphe d'appel

Dans cette section, nous proposons une fonctionnalité permettant de générer un graphe d'appel du projet Java analysé. Le graphe d'appel est représenté sous forme de texte, et il est affiché dans une fenêtre distincte équipée d'un composant JTexte Area, lui même imbriqué dans un JScrollPanel. Cette visualisation textuelle du graphe d'appel offre une manière pratique d'explorer la structure d'appel du projet.

Le graphe d'appel nous permettra de comprendre les relations entre les différentes classes et méthodes du projet, ce qui peut être particulièrement utile pour analyser la complexité du code, la dépendance entre les composants, et identifier les points critiques de notre application Java.

Cette fonctionnalité nous donne un aperçu clair et détaillé de la manière dont les classes et les méthodes interagissent au sein de notre projet Java, ce qui peut faciliter la compréhension du code et la prise de décisions pour son amélioration.

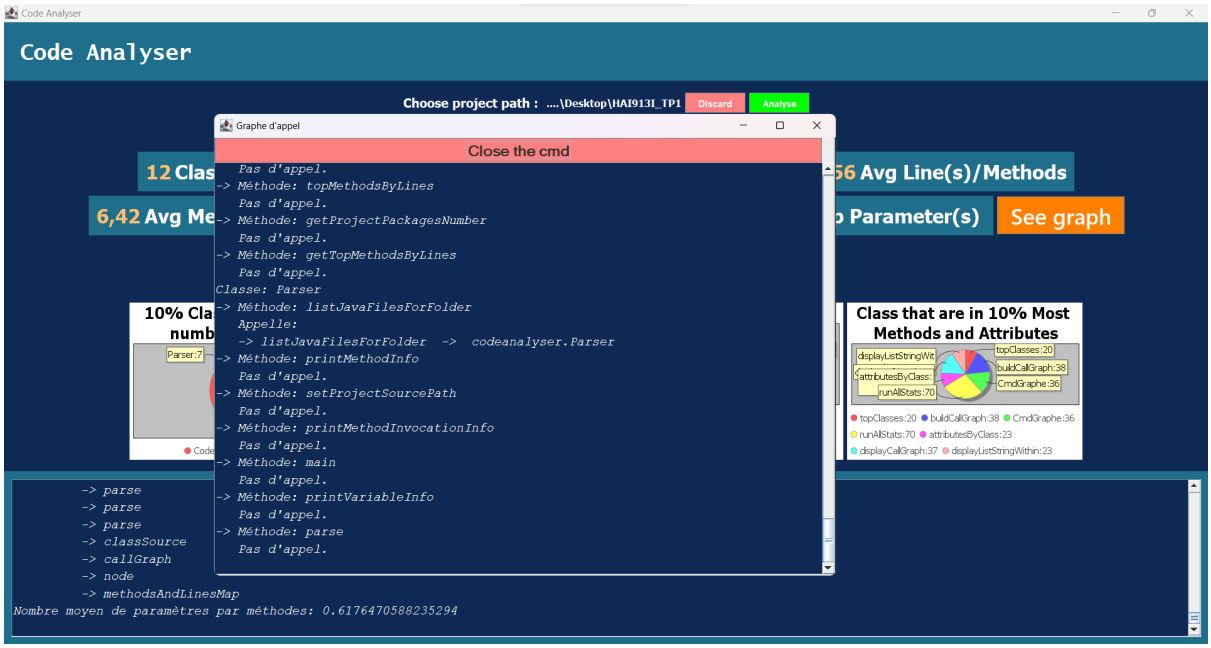

FIGURE 3 – Vue de la fenêtre cmd d'affichage du graphe d'appel

4.4 Gestion des erreurs

Nous avons également implémenté les divers éléments de code nécessaires à la gestion des erreurs, créant ainsi une approche robuste pour traiter les cas où le projet analysé ne correspond pas à un projet Java conforme à nos attentes. Afin d'assurer une expérience utilisateur sans heurts, nous avons employé des structures de blocs try...catch pour intercepter et gérer les exceptions éventuelles. Dans le cas où le projet ne serait pas un projet Java valide, notre interface détectera l'erreur et nous invitera à modifier la source du projet.

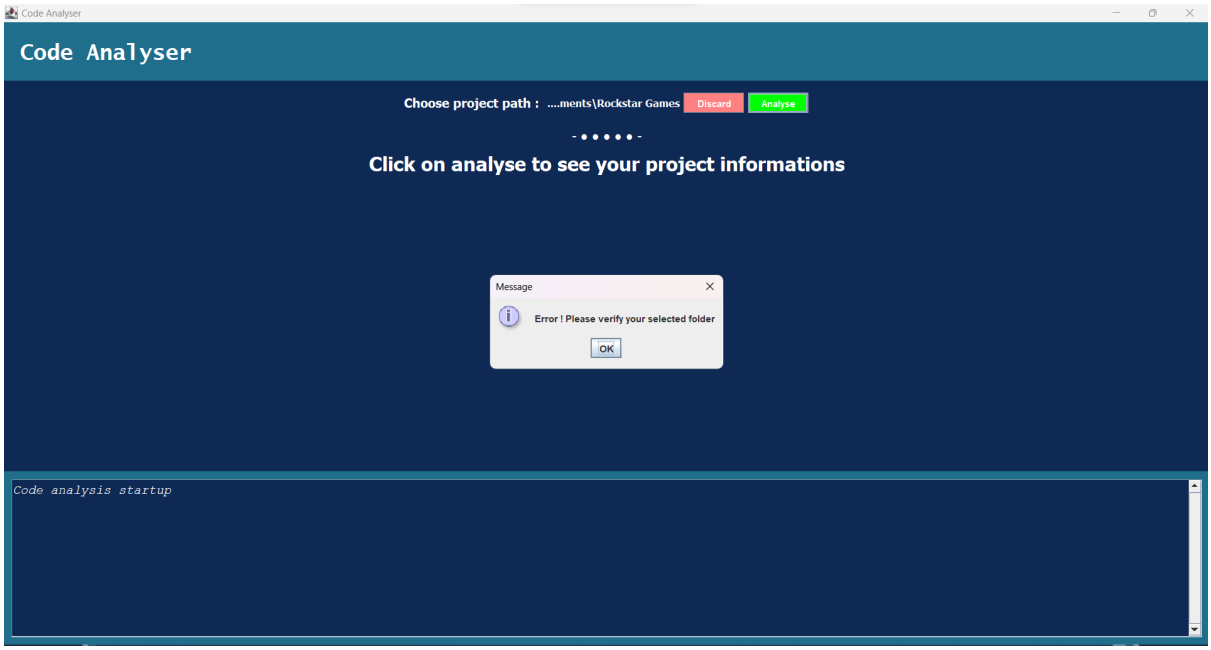

FIGURE 4 – Erreur lors de l'analyse d'un mauvais projet (Non Java)

4.5 Exemple d'une analyse de projet complexe

Voici pour finir la représentation de l'analyse d'un projet complexe, avec les différentes parties. Comme on peut le voir les graphes et les données clés permettent de bien voir les différents résultats d'analyse.

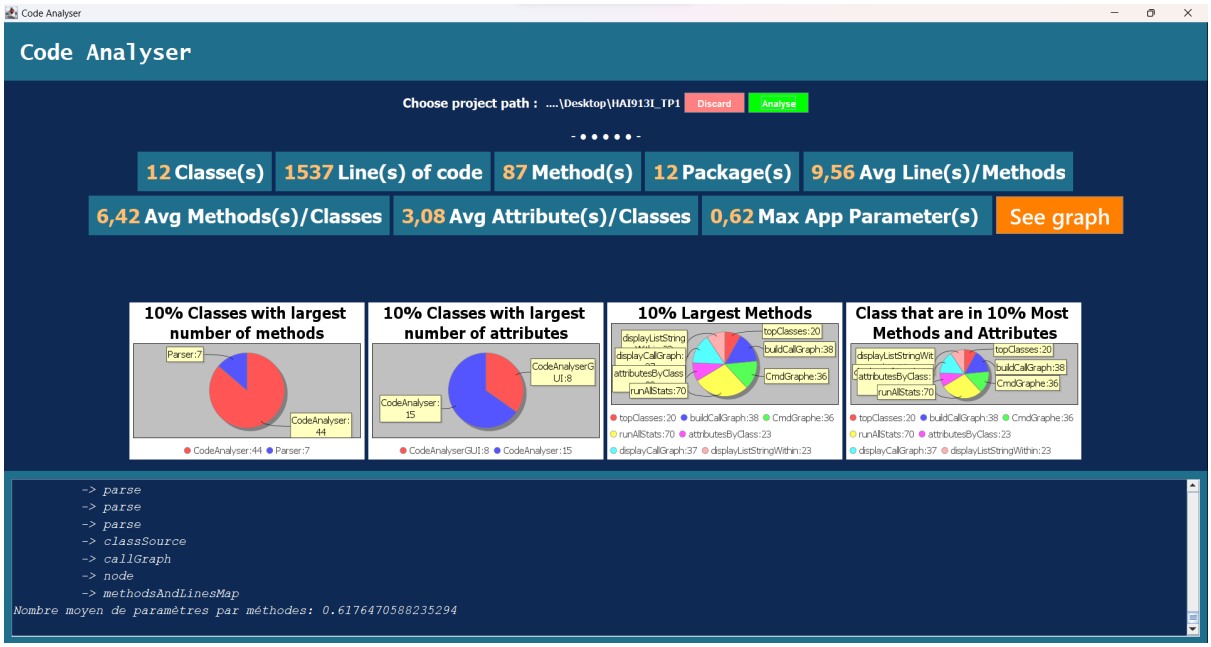

FIGURE 5 – Exemple de l'analyse d'un projet complexe