Évolution et restructuration des logiciels

Rapport

SAR Alexandre

October 2024

Lien vers le dépôt GitHub : https://github.com/texao/tp1EvolutionRestructurationLogiciel

Contents

1 Ananlyse

L'objectif est d'écrire une application, pour obtenir certaines informations d'une application. Pour cela, des calculs statistiques ont été mis en place. Par exemple, calculer le nombre total de méthodes de l'application, calculer le nombre de classes ou calculer les 10% des classes qui possèdent le plus grand nombre de méthodes. Cette section présente alors une série de calculs statistiques, afin de mieux comprendre la structure du code source d'une application.

1.1 Implémentation

Pour analyser le code source Java, qui se trouve dans un package, j'ai implémenté un analyseur basé sur l'API AST (Abstract Syntax Tree) de Java. Il parcourt les fichiers source à l'aide d'un modèle de visiteur (Visitor Pattern) pour extraire des informations statistiques. Le visiteur est utilisé pour identifier les classes, méthodes et packages dans chaque fichier, en comptant leur occurrence et en calculant des statistiques comme le nombre de lignes de code et le nombre de paramètres par méthode.

1.2 Calculs statistiques

Voici des exemples de calculs statistiques réalisés.

1. **Les 10% des classes qui possèdent le plus grand nombre de méthodes**:

```
List<TypeDeclaration> top10PourcentMehtode = classDeclarations.stream()
.sorted((a, b) -> Integer.compare(b.getMethods().length, a.getMethods().length)) // Trier par nombre de me
.limit(Math.max(1, classDeclarations.size() / 10)) // Prendre les 10% (au moins 1)
.collect(Collectors.toList());

System.out.println("Les 10% des classes avec le plus de méthodes:");
for (TypeDeclaration typeDecl : top10PourcentMehtode) {
    System.out.println("Classe : " + typeDecl.getName() + ", Méthodes : " + typeDecl.getMethods().length);
}
```

Figure 1: Classes avec le plus de méthodes

Explication:

- **Filtrer les classes** : Utilisation de TypeDeclaration, qui permet d'accéder aux informations sur chaque classe, comme le nombre de méthodes.

- **Tri des classes par nombre de méthodes** : La méthode sorted trie les classes en fonction du nombre de méthodes qu'elles contiennent, de la plus grande à la plus petite.

- **Limiter la sélection à 10%** : La méthode limit réduit la sélection aux 10% des classes ayant le plus grand nombre de méthodes.

- **Collecter le résultat** : Les classes filtrées et triées sont collectées dans une nouvelle liste.

Résultats:

```
Les 10% des classes avec le plus de méthodes:
Classe : ObjetPostal, Méthodes : 20
Les 10% des classes avec le plus de d'attributs:
Classe : ColisExpress, attributs : 6
```

2. Nombre de lignes de code de l'application:

```
// Compter les lignes de code par classe
int startLine = cu.getLineNumber(node.getStartPosition());
int endLine = cu.getLineNumber(node.getStartPosition() + node.getLength());
totalLinesOfCode += (endLine - startLine + 1);
```

Figure 2: Nombre total de lignes

Explication :

- `cu.getLineNumber(node.getStartPosition())` : Cette ligne récupère la ligne de départ où la classe commence dans le fichier source. La

méthode `getStartPosition()` renvoie la position du début de la classe, et `getLineNumber()` convertit cette position en numéro de ligne.

- `cu.getLineNumber(node.getStartPosition() + node.getLength())` : Cette ligne calcule la ligne de fin de la classe. La méthode `getLength()` renvoie la longueur du nœud de la classe (en nombre de caractères), et `getLineNumber()` donne le numéro de la ligne où la classe se termine.
- `totalLinesOfCode += (endLine - startLine + 1)` : Ici, on calcule le nombre total de lignes de code pour la classe en soustrayant la ligne de début de la ligne de fin et en ajoutant 1 (car les deux lignes sont incluses).

Résultats

```
Nombre total de lignes de code : 570
Nombre moyen de méthodes par classe : 10.0
```

3. Nombre moyen de méthodes par classe.

```
"Nombre moyen de méthodes par classe : " + (totalMethods / (float) totalClasses));
```

Figure 3: Nombre moyen de méthode par classe

Explication :

Le **nombre moyen de méthodes par classe** est calculé en divisant le nombre total de méthodes par le nombre total de classes dans l'application.

- `totalMethods` représente le nombre total de méthodes dans toutes les classes de l'application. Sachant que:

  `totalMethods += node.getMethods().length;`

- `totalClasses` représente le nombre total de classes analysées dans le projet.

Résultats:

```
Nombre moyen de méthodes par classe : 10.0
Nombre moyen d'attributs par classe : 3.0
Nombre moyen line code par classe : 71.25
```

2 Interface

Une interface utilisateur graphique permet d'analyser le code source d'un projet Java. Elle utilise la bibliothèque **Swing**, qui est une partie intégrante de Java Foundation Classes (JFC) et fournit des composants graphiques pour créer des interfaces utilisateur riches et interactives.

2.1 Principales Caractéristiques

- **Fenêtre Principale** : La fenêtre principale est créée en utilisant `JFrame`, qui agit comme le conteneur principal de l'application. Elle est configurée pour se fermer proprement lorsque l'utilisateur clique sur la croix de fermeture.
- **Composants de l'Interface** : Divers composants Swing, tels que `JLabel`, `JTextField`, `JButton`, et `JTextArea`, sont utilisés pour afficher des textes, recevoir des entrées de l'utilisateur, et afficher les résultats de l'analyse. `FlowLayout` est utilisé pour organiser les composants de manière fluide, permettant à l'interface de s'adapter à différentes tailles de fenêtre.
- **Interaction Utilisateur** : L'utilisateur doit entrer le chemin du répertoire à analyser et spécifier un nombre de méthodes (X). En effet, c'est pour connaître les classes qui possèdent plus de X méthodes. Un bouton "Analyser" déclenche l'analyse du code, et les résultats sont affichés dans une zone de texte non éditable.
- **Gestion des Événements** : L'application utilise des `ActionListener` pour gérer les événements tels que le clic sur le bouton. Cela permet de déclencher des actions spécifiques, comme l'analyse du code, et d'afficher les résultats à l'utilisateur.

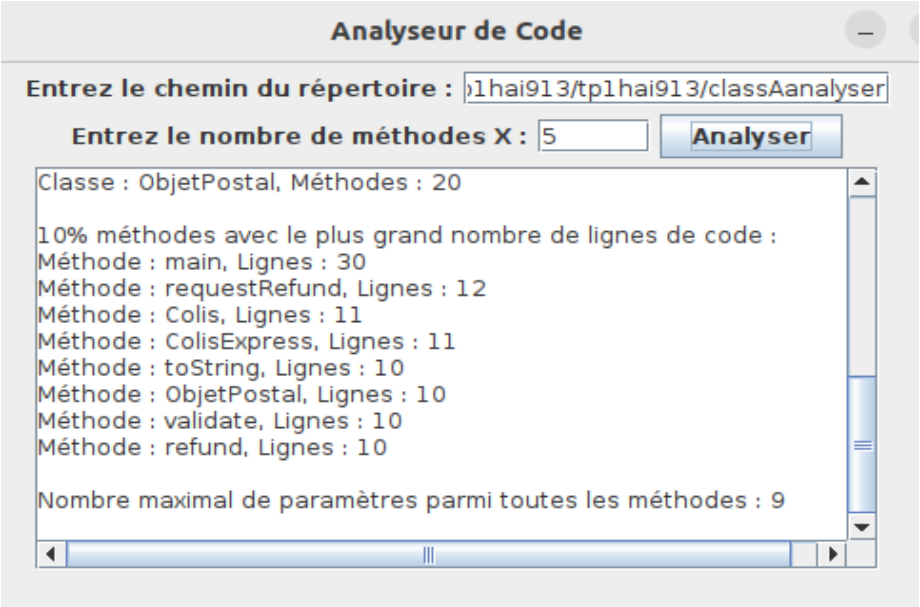

Figure 4: Interface graphique

Sur la figure ci dessus, nous voyons l'interface graphique. L'utilisateur doit rentrer le chemin vers le projet, ainsi qu'un nombre X. Il s'agit de connaître les classes qui possèdent plus de X méthodes. On voit certaines informations, comme les 10% des méthodes avec le plus grand nombre de lignes de code, ou encore le nombre maximal de paramètres, parmi toutes les méthodes.

3 Graphe d'appel

3.1 Structure graphe

```
Appel global :
    -> setDeclContenu()
    -> setValeurDecl()
    -> getVolume()
    -> getTauxRec() -> getTauxRec() (récursif)
    -> getValeurDecl() -> getValeurDecl() (récursif)
```

Figure 5: Graphe d'appel

Sur la figure ci-dessus, nous pouvons voir un graphe d'appel correspondant à du code analysé.

- **Appels internes à la classe *Colis* :**
  - Appel de la méthode *setDeclContenu* avec *this* comme receveur.
  - Appel de la méthode *setValeurDecl* avec *this* comme receveur.
  - Appel de la méthode *getVolume* avec *this* comme receveur.
  - Appels multiples de la méthode *getTauxRec* avec *this* comme receveur.
  - Appels multiples de la méthode *getValeurDecl* avec *this* comme receveur.

3.2 Génération graphe d'appel

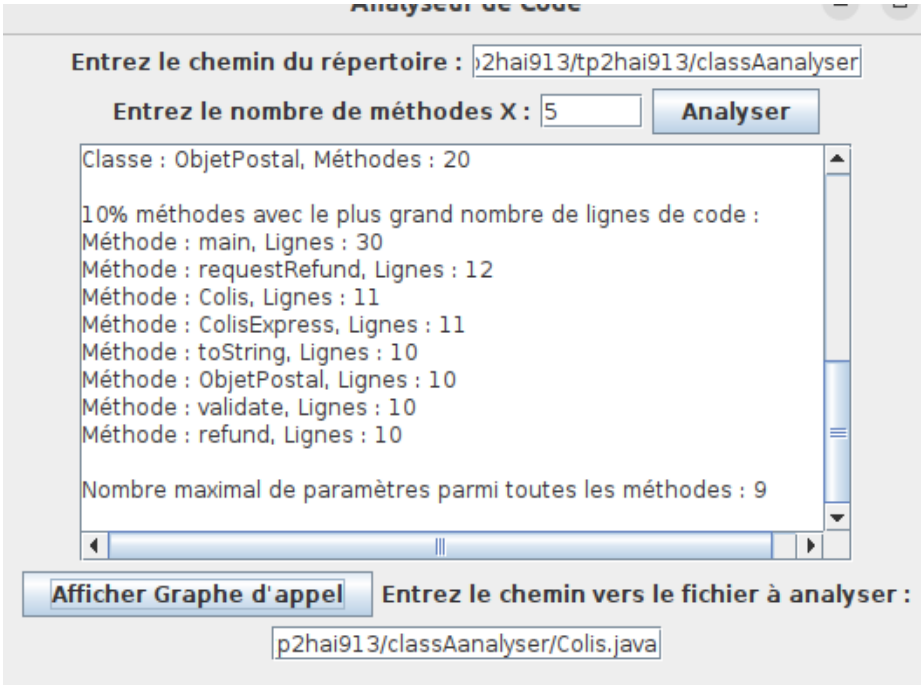

Figure 6: Interface

La figure ci-dessus présente l'interface de l'application. En bas de l'interface, un champ de saisie à été rajouté. Il permet à l'utilisateur d'entrer le chemin vers le fichier à analyser. Ce chemin est essentiel pour générer le graphe d'appel.

Lorsque l'utilisateur clique sur le bouton **"Afficher Graphe d'appel"**, l'application exécute les étapes suivantes :

1. **Génération du fichier `.dot` :** L'application analyse le code source du fichier spécifié et crée un fichier `.dot`, qui contient les relations d'appel entre les différentes méthodes et classes de la source analysée.

2. **Création du fichier PNG :** Le fichier `.dot` est ensuite utilisé pour générer une image au format PNG. Cette image représente visuellement le graphe d'appel, permettant à l'utilisateur de visualiser facilement les interactions entre les méthodes et les classes.

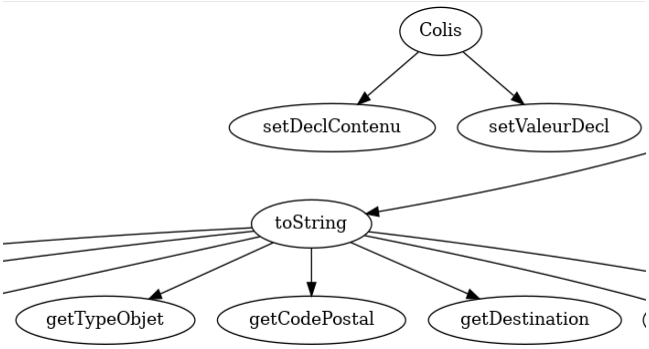

Figure 7: image graphe d'appel

La figure ci dessus présente une partie du graphe d'appel pour la classe "Colis". Ce graphe illustre les différentes méthodes de la classe et les relations d'appel qui les relient entre elles. Chaque flèche indique comment les méthodes interagissent, offrant ainsi une visualisation claire des dépendances au sein de la classe.