MicroservicesRepository

Études empiriques pour analyser les odeurs de code et les odeurs d'architecture des projets adoptant l'architecture microservices.

Le contexte de notre étude empirique est mené sur des applications basées sur style architectural de microservices conçus et développé en tant que projet open source et hébergées sur GitHub. Nous avons analysé deux catégories de mauvaises odeurs (1) d'architecture, et (2) de code.

Nous ciblons des applications développées principalement avec java, nous avons considéré par la suite 10 projets qui constituent notre jeu de données final.

En particulier, nous avons analysé ces applications afin de détecter la présence d'un ensemble d'odeurs courantes, traditionnelles en conception et en développement, ayant un impact négatif sur les attributs fonctionnels et non fonctionnels des applications.

Par la suite on étudie la densité de chaque type d'odeur par KLOC et on finit par déduire la relation de corrélation entre les odeurs de code et les odeurs d'architecture dans les microservices du même système.

Pour chaque application de notre jeu de données, nous avons utilisé les outils suivants :

1- Designite :

Pour chaque application de notre jeu de données, nous avons utilisé l'outil Designite pour sa capacité d'évaluer la qualité code des applications, Designite permet de détecter les odeurs de conception spécifiques aux projets développés à base de java. Il suffit d'utiliser la commande suivante sur l'invite de commande :

C:\Users\DELL\Designite> java -jar Designitejava.jar -i **myProjetct** -o **myCSVResult**

Le majeur avantage que l'outil Designite offre, c'est qu'il affiche des résultats clairs des odeurs d'architecture détectée sur l'invite de commande directement, accompagnée des métriques de détection utilisées. Designite permet également d'exporter des fichiers CSV résultants.

Le catalogue des odeurs d'architecture détectables avec Designite :

- **Dépendance cyclique :** Dépendance directe ou indirecte entre deux composants architecturaux ou plus.
- **Dépendance instable :** Un composant instable qui dépend d'un autre composant encore moins stable que lui. Le principe stipule que la dépendance doit aller dans le sens de stabilité des packages, un package ne doit jamais dépendre d'un package moins stable que lui.

- **Interfaces ambiguës :** Une interface n'offrant qu'un seul point d'enterrée pour un composant. Apparais lorsque les interactions ne sont pas explicitement modélisées.
- **Composant God :** un composant dans le nombre de lignes de code LOC est très élevé, ou qu'il contient beaucoup de classes.

  Valeurs des métriques considérées sont : min classes =30 et min LOC d'un composant = 27.

- **Concentration de fonctionnalités :** Un composant dont sa responsabilité est importante, autrement dit, il se charge de réaliser plusieurs opérations.
  Valeurs des métriques utilisées sont : min classes = 5 et Min LCC = 0,20.
- **Fonctionnalité dispersée :** Un ensemble de composants qui résolvent tous le même problème de grande valeur. Une odeur qui se produit lorsque des composants sont responsables de la réalisation de la même responsabilité de haut niveau.
  Valeurs de la métrique utilisée est : min accès au composant externe =1.
- **Structure dense :** Dépendance excessive entre les composants architecturaux dont la structure n'est pas précise.
  Valeurs de la métrique utilisée est : min degré moyen du graphique des composants =5.

2- L'outil *Organic tool* :

Est un outil de détection des mauvaises odeurs de code source. Pour exécuter *Organic*, il vous suffit de lancer une commande sur un terminal dans le répertoire de chaque un de nos projets. En sortie en aura un fichier Json avec tous les détails concernant l'odeur détectée qu'on traduit par la suite en fichier Excel pour une meilleure lisibilité de lecture de données.

C:\Users\DELL\organic-standalone-main\>gradle run --args="-sf '**Myresult.json**' -src '**MyProject**'"

Voici les odeurs de code que l'outil nous a permis d'analyser :

- **Méthode longue :** Une méthode contient beaucoup de lignes de code. La complexité d'une méthode commence dès 10 lignes. Plus le nombre de lignes de code est élevé, plus la méthode devient difficile à comprendre.
- **Méthode complexe :** La complexité cyclomatique d'une méthode se calcule en fonction du nombre de chemins d'exécution, Plus le nombre de chemins est élevé, plus le nombre de tests nécessaires pour couvrir tous les chemins possibles devient important.
- **Longue liste de paramètres** : signifie qu'une fonction avec beaucoup de paramètres est complexe. Plus de 3 à 4 paramètres pour une méthode. Ou une

longue liste de paramètre signifie que la méthode s'en charge d'un nombre excessif de responsabilités.

- **Jalousie de fonctionnalités** : Une classe qui s'intéresse à accéder aux méthodes et attribue d'autres classes plus que ses attributs et méthodes à elle.
- **Classe de données :** Une classe ayant des champs, des getters et des setters, mais qu'elle n'implémente pas de fonctionnalités qui comportent sur ses champs, elle est généralement manipulée par d'autres classes vu qu'elle manque de méthodes pertinentes pour justifier le fait qu'elle soit une classe.
- **Chaînes de message :** Un ensemble d'appels ($ a->b ()->c ()->d ()) signifie qu'un client demande à un objet, que cet objet demande à son tour encore à un autre objet, ce qui rend difficile de voir ou la fonction se produit réellement.
- **Classe paresseuse :** Une classe qui n'est pas utilisée, qui ne fait pas assez ou même inutile, elle ne fait qu'augmenter la complexité du code, donc elle doit être supprimée ou bien faire partie d'une classe utile.

3- Intellij :

L'outil nous permet de générer la matrice de dépendance « Dependancy Structure Matrix », une analyse DSM qui nous permet de visualiser les relations complexes entre les packages de chaque projet, ainsi nous pouvons suivre la propagation de tous les changements sur les systèmes de composants. Nous la également considérer comme outil de validation des résultats de l'odeur architecturale « Dépendance cyclique ».

4- Understand:

En ce qui concerne la validation de nos données collectées, nous avons utilisé l'outil Understand pour vérifier un ensemble de métrique comme le LOC et nombre de méthodes, de fichiers, etc., nous l'avons utilisé également pour déduire le nombre de lignes de code de chaque projet et le nombre de chaque composant constituant de chaque projet (Classes, méthodes…).