

Rapport V&V NullPointerException

Introduction :

Dans le cadre du module V&V du Master 2 Ingénierie logiciel en alternance, nous avons travaillé sur un outil d'analyse de code Java. Le sujet que nous avons choisi est la détection des NullPointerException. Il s'agissait donc d'analyser statiquement un code source afin de rendre un rapport à l'utilisateur, permettant d'avoir un avis sur les risques de NullPointerException sur ses fichiers Java.

Pour ce faire, nous avons utilisé la librairie Spoon (créée par l'INRIA).

Vous trouverez notre dépôt GitHub à cette adresse :

<https://github.com/lalzin/V-V-Static-Analysis>

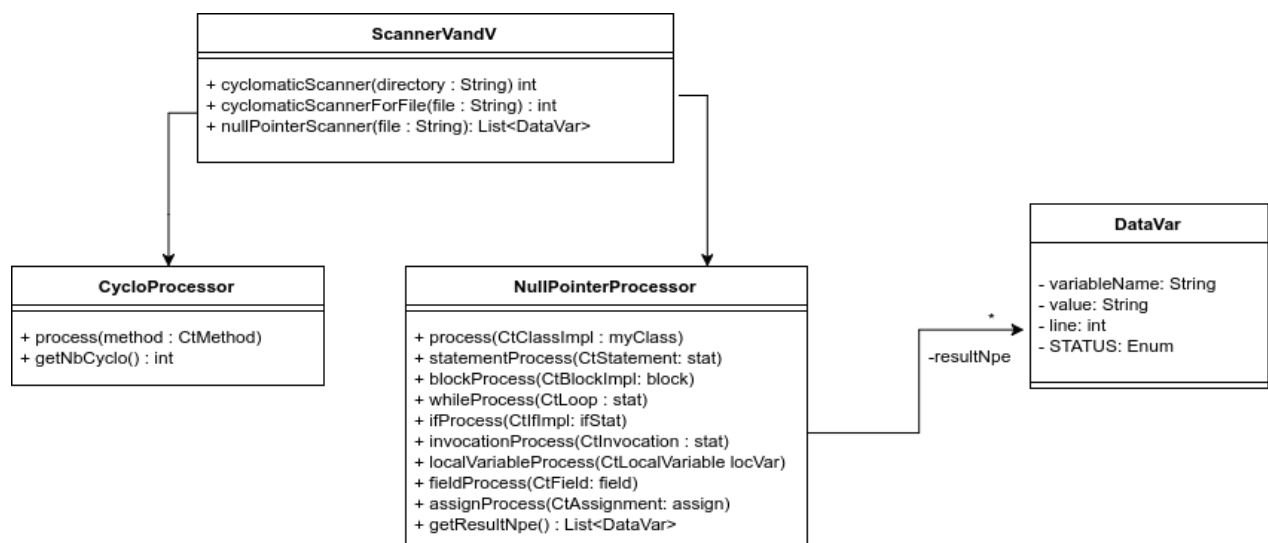
Ce dépôt contient :

- Le code source de notre outil.
- Un README.
- Une suite de test unitaire validant les fonctionnalités.

Solution :

Architecture de notre projet :

Voici l'architecture de notre projet :



La classe **ScannerVandV** est le point d'entrée de l'application. Il implémente 3 fonctionnalités :

- **cyclomaticScanner** : Cette méthode prend un paramètre : le chemin vers le projet devant être analysé. Il renvoie un entier, le nombre cyclomatique du projet. Cette méthode utilise le processeur Spoon, **CycloProcessor**.
- **cyclomaticScannerForFile** : Cette méthode prend un paramètre : le chemin vers le fichier JAVA devant être analysé. Il renvoie un entier, le nombre cyclomatique du fichier JAVA. Cette méthode utilise le processeur Spoon, **CycloProcessor**.
- **nullPointerScanner** : Cette méthode prend un paramètre : le chemin vers le fichier à analyser pour trouver les éventuels NullPointerException. Cette méthode utilise le processeur Spoon, **NullPointerExceptionProcessor**.

La classe **CycloProcessor** a pour rôle de d'analyser les fichiers pour renvoyer la complexité cyclomatique d'un projet ou d'un fichier. La complexité cyclomatique correspond au nombre de points de décision de la méthode (if, case, while, ...) + 1 (le chemin principal).

La classe **NullPointerExceptionProcessor** a pour rôle d'identifier les risques que le code lance un NullPointerException. Un fois l'analyse effectuée, l'utilisateur peut récupérer un rapport de l'analyse (avec *getResultNPE()*), qui est en une liste de **DataVar** contenant les appels.

Voici le format de **DataVar** :

- Nom de la variable appelée : String
- Valeur de la variable appelée : String
- Status de la variable (Enumération : ALERT / WARNING / OK)
- Ligne de l'appel : int

Voici un exemple d'affichage d'une liste de **DataVar** :

```
OK [var4] with value ["titi"] at line 16
WARNING for [var4] with value [null] at line 20
OK for [var3] with value ["tutu"] at line 27
ALERT for [var1] with value [null] at line 29
OK for [var2] with value ["toto"] at line 31
```

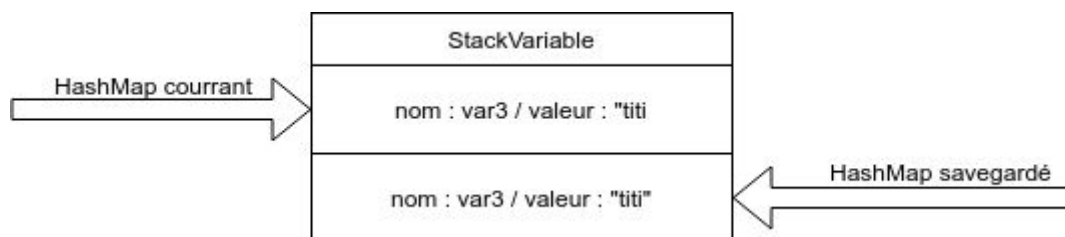
Scénario d'exécution :

Ci-dessous, nous présentons le fonctionnement de notre analyseur statique a la recherche de NPE :

Voici un fichier JAVA très simple. Nous allons effectué notre analyse pas à pas :

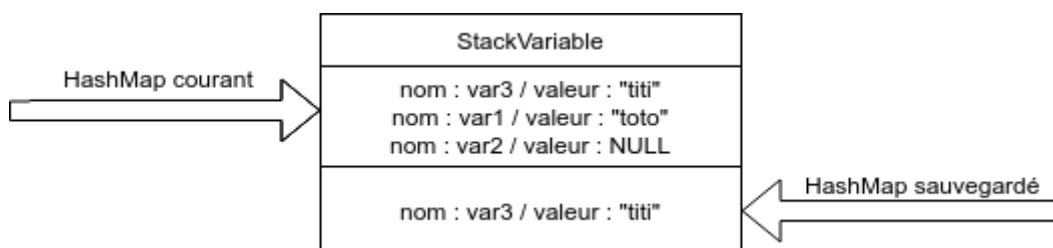
```
1 String var3 = "titi";
2
3 public static void test1() {
4     String var1 = "toto"
5     String var2 = null;
6     var1.toString();
7     var2.toString();
8 }
9
10 public static void test2() {
11     var3.toString();
12 }
```

1 - Notre analyseur commence par initialiser un HashMap<String, String> stockant les valeurs des variables. Il va ensuite stocker ce hashmap dans une stack. Il ajoute dans ce hashmap les variables globales (attributs de la classe).Voici l'état de la stack juste après être entrée dans la méthode test1() :



2- Nous donc une duplication de notre hashmap afin de sauvegarder son état. Son état sera restauré à la fin de la méthode test1().

3- Voici l'état de la stackVariable après avoir analysé la ligne *String var1 = "toto"* et *String var2 = null;*



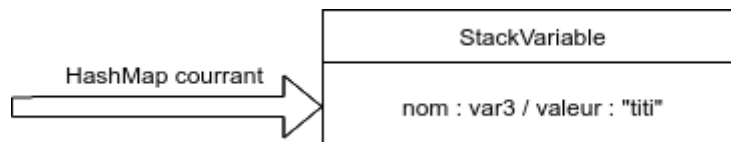
4- Lors de l'évaluation de la ligne *var1.toString()*; notre programme vérifie dans son hashmap courant la valeur de var1. la variable var1 est donc évalué à "toto", donc aucun NPE. Nous pouvons enregistrer cette appel dans notre resultatNPE :

résultatNPE
[OK] - var1 - "toto" - ligne 6

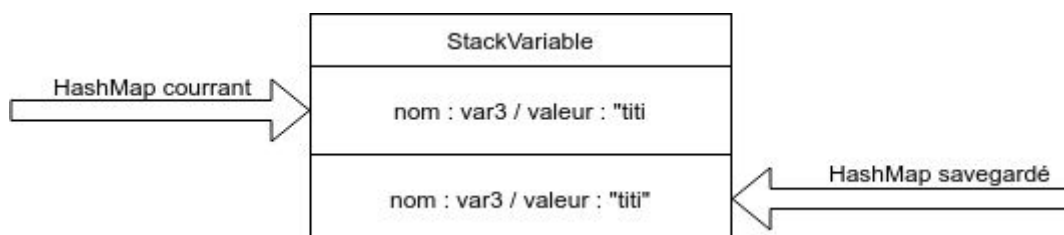
5- Lors de l'évaluation de la ligne *var2.toString()*; notre programme vérifie dans son hashmap courant la valeur de var2. la variable var2 est donc évalué à NULL, donc nous avons ici un NPE. Nous pouvons enregistrer cette appel dangereux dans notre resultatNPE :

résultatNPE
[OK] - var1 - "toto" - ligne 6
[ALERT] - var2- NULL - ligne 7

6- A la fin de la méthode test1(), notre hashmap courant est supprimé on peut à nouveau pointer sur notre hashmap de départ :



7- Une fois entrée la méthode test2(), voici l'état de notre stackVariable :



8- Lors de l'évaluation de la ligne *var3.toString()*; notre programme vérifie dans son hashmap courant la valeur de *var3*. la variable *var3* est donc évalué à "titi", donc aucun NPE. Nous pouvons enregistrer cette appel dans notre resultatNPE :

résultatNPE
[OK] - var1 - "toto" - ligne 6 [ALERT] - var2- NULL - ligne 7 [OK] - var3- "titi" - ligne 11

9- Une fois sortie de la méthode *test2()*, notre analyse se termine et l'utilisateur peut récupérer le resultatNPE :

OK for [var1] with value ["toto"] at line 6
ALERT for [var2] with value [null] at line 7
OK for [var3] with value ["titi"] at line 11











Evaluation :

Description du sujet :

Nous avons choisis le sujet sur la détection de NPE dans un programme tout en portant un intérêt à l'analyse cyclomatique de ce dernier. Nous avons utilisé l'IDE Eclipse car étant tout deux habitués à cette interface et avons développé notre programme en java. Pour pouvoir effectuer nos traitements nous avons utilisé la librairie Spoon permettant de faire de l'analyse de code.

De plus pour tester notre programme nous avons utilisé JUnit ainsi que Covertura pour la couverture de code.

Ci dessous la couverture de code du projet :

V-V-Static-Analysis (2) (21 déc. 2017 15:44:11)		
Element	Coverage	Covered Instruct
- V-V-Static-Analysis	 94,6 %	1 295
src/main/java	 91,6 %	804
istic.vv	 91,6 %	804
DataVar.java	 66,5 %	113
ScannerVandV.java	 96,6 %	252
App.java	0,0 %	0
CycloProcessor.java	100,0 %	34
NullProcessor.java	 100,0 %	405
src/test/java	 100,0 %	491
istic.vv.test	 100,0 %	491
AppTest.java	100,0 %	12
CycloProcessorTest.java	 100,0 %	80
NullPointerProcessorTest.java	 100,0 %	399

La couverture de code Covertura nous indique que 94,6% de notre code est couvert par des tests, ce qui est très correcte.

Complexité / Performance:

Le programme que nous avons réalisé parcourt les fichiers java à la recherche d'un NullPointer. Il n'est pas en mesure de parcourir en profondeur l'ensemble d'un projet pour en trouver les NullPointer car nous avons restreint le programme à l'analyse d'un seul fichier à la fois. Il aurait été cependant intéressant d'avoir cette fonctionnalité. De ce fait, la complexité est lié à la taille du fichier analysé.

Pour l'analyse cyclomatique elle peut s'effectuer de deux manière : sur un répertoire sur l'ensemble des fichier java ou sur un fichier unique. La complexité est donc corrélé au nombre de fichier à analyser.

Discussion :

Notre travail sur ce sujet peut analyser du code pour trouver des NPE. Comme indiqué précédemment, il serait intéressant de pouvoir analyser l'intégralité d'un projet tout en effectuant une analyse poussée pour pouvoir trouver toutes les instances de NullPointerException possible. Nous n'avons pas été en mesure de pouvoir effectuer ces traitements par manque de temps.

De plus notre analyse s'effectue de haut en bas de manière simpliste, mais aurait pu être plus judicieux d'opter pour une analyse dite de bas en haut qui consisterait à identifier les appels et de remonter le graphe du code jusqu'aux affectations de la variable concernée.

Problème rencontré :

Notre choix de conception de l'analyse des NPE (de haut en bas) ne nous permettait pas de faire une analyse complète du problème posé, notamment l'analyse d'un projet. Nous avons essayé de régler ce problème en effectuant un traitement différent, une analyse dite de bas en haut. Cette amélioration n'étant toujours pas au point, nous avons préféré rester sur notre première solution pour rendre un projet qui fonctionne selon nos critères.

Axe d'amélioration :

Comme vu précédemment, l'analyse de bas en haut devrait être rajoutée ainsi que la possibilité de gérer l'intégralité d'un projet.

Les résultats de nos analyses ne sont accessibles que par les test Java, mais pourrait être enregistrés dans un rapport (fichier texte, ...) pour une meilleure lecture des résultats. Le programme pourrait aussi s'exécuter directement dans un terminal pour une utilisation plus rapide.

Conclusion :

Ce projet nous a permis de porter une attention précise sur l'analyse des NPE. Nous avons appris que cette analyse n'est pas trivial et que certaines détection de NPE sont facile à décélér mais dans notre cas plus complexe avec des structures de contrôles ce qui rend la tâche plus difficile.

De plus par ces travaux, nous avons pu nous intéresser à la librairie Spoon qui permet un grand nombre de traitement sur l'analyse de code. Il serait très intéressant de pouvoir réaliser un outil complet sur la détection des NPE pour le développement de projet conséquent afin d'obtenir une qualité de code indéniable.