Injection de code source et couverture de code en temps réel

Auteurs :
Maxime CLEMENT
Jordan PIORUN

16 Décembre 2015

Table des matières

Introduction

La couverture de code est une métrique représentant le pourcentage de nombre de lignes de code exécutés. Cette métrique est utilisée lors de l'exécution de suites de tests afin de mesurer le nombre de lignes de code couvertes par ces suites.

Certaines méthodes de développement comme le TDD[1] vont garantir une bonne couverture de par le fait que les tests sont écrit avant le code. Une question peut alors se poser, la totalité du code couvert est il bien exécuté en production ? Il est probable qu'une ligne de code exécuté lors d'une suite de tests, ne soit jamais exécutée dans un environnement de production.

Le but est de montrer que calculer la couverture de code sur un programme exécuté dans un environnement de production est possible, de plus, ce calcul pourrait être réalisé en temps réel pour ne pas avoir à stopper l'exécution du code en production afin d'obtenir cette fameuse métrique.

Pour atteindre notre objectif, nous avons utilisé Spoon, une librairie Java permettant de faire de la transformation de code source Java. L'idée est que le programme transformé puisse s'auto-instrumenter afin de notifier à l'utilisateur sa couverture à un moment t en temps réel.

Nous avons évalué notre approche sur plusieurs critères. Tous d'abord la couverture de code calculée par notre outils comparé à d'autres déjà existant, ensuite le coût supplémentaire nécessaire en mémoire afin de pouvoir réaliser ce calcul en temps réel. Et pour finir l'impact sur le temps d'exécution du programme instrumenté par rapport à l'original.

[1] Test-Driven Development

Travail technique

1.1 Principes

Calculer la couverture de code, quelle soit de branches, de ligne ou encore de méthodes peut se faire de deux manières[1], soit en modifiant le code source, soit en modifiant le byte code. Ces deux approches ont un but commun, stocker de l'information concernant l'exécution du programme. Ces informations permettrons par le suite d'évaluer le code exécuter, et de calculer la métrique voulu.

1.2 Réalisation

Pour notre approche nous avons choisi la première technique appelée "Source To Source". Ce choix s'explique par plusieurs raisons, il est plus facile de debugger du code source plutôt que du byte code. Nous maîtrisions déjà un outils pour transformer du code source. Cependant, les techniques expliquées ci-dessous pourrait être reproduite sur le byte code.

La différence avec les techniques expliquées en 1.1 est que notre but n'est pas de stocker uniquement de l'information. En effet, afin de pouvoir réaliser le calcul de couverture de code en temps réel, nous devons également interpréter ces informations durant l'exécution du programme.

1.3 Architecture et design

1.3.1 Instrumentation

Instrumenter les lignes de code exécuté n'est pas quelque chose de difficile. Il suffit simplement d'insérer une nouvelle ligne après chaque ligne existante comme représenté dans l'exemple 1.1. Cette ligne permettra de stocker l'information relative à l'exécution de la ligne instrumenté.

```
1 maMethod :
2         instruction
3         instruction
```

```
1 maMethod :
2         instruction
3         stocker l'execution de la ligne 1
4         instruction
5         stocker l'execution de la ligne 2
```

Figure 1.1: Transformation du code original vers le code instrumenté

1.3.2 Interprétation

La couverture de ligne se calcule à l'aide de mathématique basique. Appelons L_{total} le nombre de lignes instrumentées et L_{exec} le nombre de lignes exécutées, le taux de couverture de ligne C est alors le suivant :

$$C = \frac{L_{exec}}{L_{total}} \tag{1.1}$$

Cette interprétation peut être effectuée sur les classes, packages ou encore l'intégralité du projet instrumenté. Notre implémentation permet de visualiser la couverture de ligne pour les différents scopes cités.

C'est lors de l'instrumentation que nous allons calculer L_{total}. Ensuite, lors de l'exécution nous pourront calculer dynamiquement L_{exec} et donc C en fonction de scope désiré.

1.3.3 Design

Notre implémentation est conçu sur le design MVC[1]. Les sections suivantes ont pour but de détailler chaque parti du design.

Model

Le model de l'application contient les données nécéssaires au calcule du taux de couverture. Comme expliquer dans les sections 1.3.1 et 1.3.2, nous avons besoin de l'ensemble des lignes de code instrumenté, ainsi que de l'ensemble des lignes de code exécuter. Le premier ensemble construit lors de l'instrumentation, le second sera construit dynamiquement à l'exécution du programme instrumenté.

[1]Model View Controller

View

La vue notifie l'utilisateur de la couverture actuel du programme en cours d'exécution. Cette vue est dynamique, est évolue avec le temps en fonction des modifications sur le model grâce au patron de conception. *Observateur/Observable*

Nous avons mis au point deux vues : La première est une vue textuellement qui ne permet pas la navigation entre les différents packages, mais qui pour un surcout moindre nous donne le détail de l'un d'entre eux fixé au début de l'exécution. La seconde est une vue graphique qui débute à la racine du projet. Il est ensuite possible de naviguer de package en package en cliquant sur ceux ci.

Controller

Le controller capture les actions utilisateurs sur la vue afin de changer le scope désiré. L'utilisateur peut donc naviguer entre les différent packages ou classe du projet.

1.3.4 Injection

1.4 Technologie

1.4.1 Java Fx

Pour mettre au point notre vue graphique, nous avons utilisé une librairie graphique java, JavaFx. Notre choix s'est porté sur celle ci car elle permet de lier facilement une vue à son modèle, en gérant la mise a jour de données, ce qui était particulièrement adapté à notre utilisation.

Évaluation

2.1 Pertinence

2.2 Temps

2.3 Mémoire

2.4 Complexité

Bilan personnel

Conclusion

Bibliography

[1] Ira. D. Baxter. Branch coverage for arbitrary languages made easy. 2002.