



Injection de code source et couverture de code en temps réel

Auteurs :

Maxime CLEMENT

Jordan PIORUN

16 DÉCEMBRE 2015

Table des matières

Introduction	4
1 Travail technique	5
1.1 Principes	5
1.2 Réalisation	5
1.3 Architecture et design	5
1.3.1 Instrumentation	5
1.3.2 Interprétation	6
1.3.3 Design	6
1.3.4 Injection	7
1.4 Technologie	7
1.5 Screenshots	7
2 Évaluation	8
2.1 Pertinence	8
2.2 Temps	8
2.3 Mémoire	8
Bilan personnel	10
Conclusion	11
Références	12

Introduction

La couverture de code est une métrique représentant le pourcentage de code exécuté. Cette métrique est utilisée lors l'exécution de suites de tests afin de mesurer le code couvert par ces suites.

Certaines méthodes de développement comme le TDD¹ 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 exécuté dans un environnement de production.

Le but est 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 une librairie permettant de faire de la transformation de code source. L'idée est que le programme transformé puisse d'auto-instrumenter afin de notifier l'utilisateur de sa couverture à n'importe quel moment de son exécution.

Nous avons évalué notre approche sur plusieurs critères. Tout d'abord la couverture de code calculer par notre outils comparé à d'autres déjà existant, ensuite le coût supplémentaire nécessaires 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.

¹Test-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 pendant l'exécution du programme. Ces informations permettront par la 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 transformé plutôt que le byte code. Nous maîtrisons déjà l'outil 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 exécutées 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. Ces lignes permettront de stocker les informations relatives à l'exécution des lignes instrumentées.

TODO: aligner les codes figure 1.1

1	maMethod :
2	instruction
3	instruction

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

$$C = \frac{L_{exec}}{L_{total}} \quad (1.1)$$

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 pourrions calculer dynamiquement L_{exec} et donc C en fonction du scope désiré.

1.3.3 Design

Notre implémentation est conçu sur le design MVC¹. Les sections suivantes ont pour but de détailler chaque parti du design.

Model

Le model de l'application contient les données nécessaires au calcul 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 est construit lors de l'instrumentation, le second sera construit dynamiquement lors de l'exécution du programme instrumenté.

¹Model 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*

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

Lors de la phase d'instrumentation du programme, les lignes permettant de calculer L_{exec} ne sont pas les seules à être insérées. Notre MVC est également injecter dans le projet pour que ce dernier puisse d'auto-instrumenter et notifier sa couverture à l'utilisateur. Notre programme est uniquement utilisé pour instrumenté un programme. Il suffira ensuite de démarrer le programme exécuté pour avoir le comportement voulu. La figure 1.2 représente le workflow de notre approche.

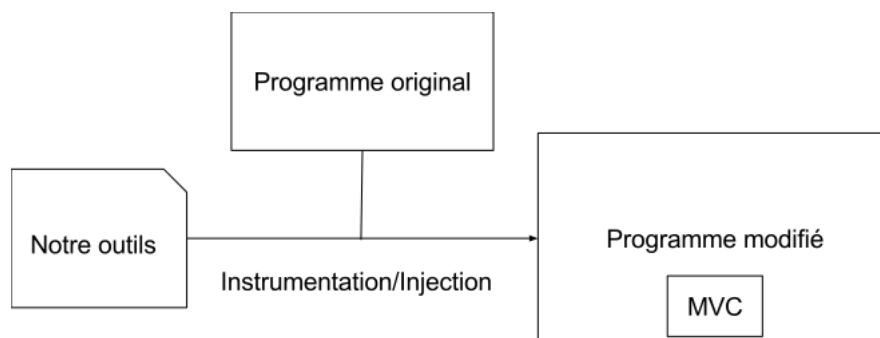


Figure 1.2: Workflow

1.4 Technologie

L'outil proposé est implémenté en Java à l'aide de Spoon, une librairie permettant la transformation de code source Java. La vue de notre MVC utilise JavaFX 8.

1.5 Screenshots

TODO: inserer screenshot graphic vu projet, packages, vu classe

Évaluation

Les différentes parties ci-dessous représente l'évaluation effectué sur notre approches. Nous avons évaluer celle-ci sur 3 critères, la pertinence du taux de couverture calculé par rapport à d'autre outils. Le temps d'exécution du programme instrumenté ainsi que la quantité maximale de mémoire nécessaire comparé à l'original. Pour ces différentes évaluation nous avons choisi Apache Commons Lang comme projet de référence. Pour simuler un environnement de production, les tests du projet jouent le rôle des actions utilisateurs.

2.1 Pertinence

La pertinence de notre outils est évalué sur le pourcentage de couverture calculé comparé à d'autre outils. Les autres outils de références sont JaCoCo et JCoverage, respectivement utilisé à l'aide de EclEmma et Cobertura. La table 2.1 représente les résultats obtenus.

JaCoCo	JCoverage	Notre outils
94.1	93.0	83.0

Figure 2.1: Comparatif du taux de couverture calculé en %

2.2 Temps

L'évaluation sur le temps d'exécution a été réalisé sur dix exécution successif du programme. La mesure du temps d'exécution est celle retourné par le framework JUnit lors de l'exécution des tests. La table 2.1 représente les résultats obtenus.

2.3 Mémoire

L'évaluation sur la quantité de mémoire maximale utilisée été réalisée sur dix exécution successif du programme. La mesure est celle retourné par VisualVM lors de l'exécution des tests. La

	Commons Lang	Commons Lang' + Vue Text	Commons Lang + JavaFX
Run 1	20.038	26.756	38.772
Run 2	19.605	27.020	38.450
Run 3	19.059	26.800	38.282
Run 4	19.417	28.094	40.412
Run 5	18.810	26.016	38.834
Run 6	20.399	27.881	39.228
Run 7	19.905	26.739	38.958
Run 8	18.737	27.293	39.102
Run 9	18.963	26.199	39.056
Run 10	19.500	27.882	38.637
Moyenne	19.486	27.062	38.973
Overhead (%)	-	38.88	100.01

Table 2.1: Comparatif sur le temps d'exécution en seconde

table 2.2 représente les résultats obtenus.

	Commons Lang	Commons Lang' + Vue Text	Commons Lang + JavaFX
Run 1	459	526	469
Run 2	426	495	430
Run 3	416	453	432
Run 4	485	596	419
Run 5	455	419	422
Run 6	443	499	453
Run 7	541	459	333
Run 8	403	526	470
Run 9	450	482	396
Run 10	504	513	473
Moyenne	458.2	496.8	429.7
Overhead (%)	-	8.42	6.22

Table 2.2: Comparatif sur la quantité de mémoire maximale utilisée lors de l'exécution en Mo

Bilan personnel

Conclusion

Bibliography

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