CoCoSpoon : Couverture de code en temps réel avec Spoon

Auteurs :
Maxime CLEMENT
Jordan PIORUN

16 Décembre 2015

Table des matières

Introduction

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

Certaines méthodes de développement comme le TDD[1] vont garantir une bonne couverture de par le fait que les tests sont écris avant le code. Une question peut alors se poser : le code couvert est-il bien exécuté en production ? Il est probable qu'une ligne de code exécutée 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.

Pour atteindre cet objectif, nous proposons CoCoSpoon[2], un outils utilisant une librairie de transformation de code source. L'idée est que le programme transformé puisse s'auto-instrumenter pour notifier l'utilisateur de sa couverture à n'importe quel moment de son exécution.

Nous avons évalué CoCoSpoon sur plusieurs critères. Tous d'abord la couverture de code calculée par notre outil comparé à d'autres déjà existants. Ensuite l'impact sur le temps d'exécution du programme instrumenté par rapport à l'original. Pour finir le coût supplémentaire nécessaire en mémoire afin de pouvoir réaliser ce calcul en temps réel.

[1] Test-Driven Development

[2] `https://github.com/maxcleme/CoCoSpoon`

Travail technique

1.1 Principes

Calculer la couverture de code, qu'elle soit de branches, de lignes 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 exécuté, et de calculer la métrique voulue.

1.2 Réalisation

Pour implémenter CoCoSpoon nous avons choisi la première technique appelée "Source To Source"[1]. Ce choix s'explique de plusieurs manières : il est plus facile de debugger du code source transformé plutôt que le byte code lors du développement. Cependant, les techniques expliquées ci-dessous pourraient être reproduites 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.

```
1 maMethod :
2     instruction a
3     instruction b
```

```
1 maMethod :
2     instruction a
3     stocker l'execution de l'instruction a
4     instruction b
5     stocker l'execution de l'instruction b
```

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 mathématiques basiques. 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 pourrons calculer dynamiquement L_{exec} et donc C en fonction du scope désiré.

1.3.3 Design

Notre implémentation est conçue sur le design MVC[1]. Les sections suivantes ont pour but de détailler chaque partie de celui-ci.

Modèle

Le modèle de l'application contient les données nécessaires au calcul du taux de couverture. Comme expliqué dans les sections 1.3.1 et 1.3.2, nous avons besoin de l'ensemble des lignes de code instrumentées, ainsi que de l'ensemble des lignes de code exécutées. Le premier ensemble est construit lors de l'instrumentation, le second sera construit dynamiquement lors de l'exécution du programme instrumenté.

[1] Modèle Vue Controlleur

Vue

La vue notifie l'utilisateur de la couverture actuelle du programme en cours d'exécution. Cette vue est dynamique, et évolue avec le temps en fonction des modifications sur le modèle.

Nous avons mis au point deux vues : La première est une vue textuellement qui ne permet pas la visualisation de la couverture d'une classe en particulier. La seconde est une vue graphique qui débute à la racine du projet. Il est ensuite possible de naviguer dans le projet pour visualiser la couverture des packages ou des classes. De plus, cette vue permet pour une classe choisi d'ouvrir le fichier original en distinguant les lignes exécutées, non exécutées et non instrumentées.

Controlleur

Le contrôleur capture les actions de utilisateur sur la vue afin de changer le scope désiré. L'utilisateur peut donc naviguer entre les différents packages ou classes 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 injecté dans le projet pour que ce dernier puisse s'auto-instrumenter et notifier sa couverture à l'utilisateur. CoCoSpoon est uniquement utilisé pour instrumenter 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.

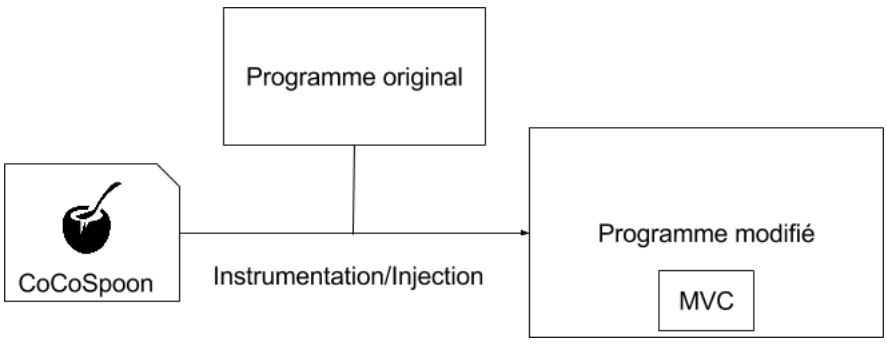

Figure 1.2: Workflow

1.4 Technologie

CoCoSpoon est implémenté en Java à l'aide de Spoon[2], une librairie que nous maîtrisions déjà permettant la transformation de code source Java. Pour la vue il utilise JavaFX[3] car elle permet de lier facilement une vue à son modèle, en gérant la mise à jour de données, ce qui était particulièrement adapté à l'utilisation désirée.

1.5 Screenshots

Figure 1.3: Représentation du projet

[2] http://spoon.gforge.inria.fr
[3] http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm

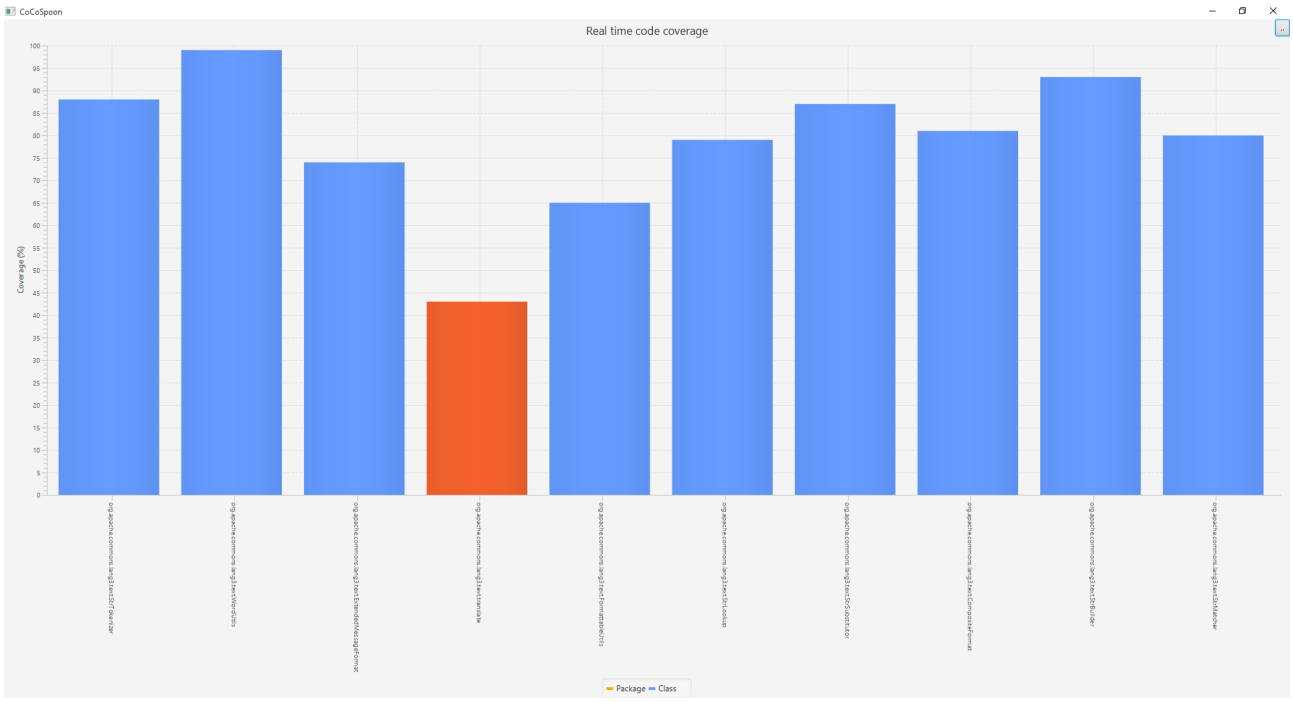

Figure 1.4: Représentation d'un package, de son sous-package et de ses classes

Figure 1.5: Représentation d'une classe

Évaluation

Les différentes parties ci-dessous représentent l'évaluation effectuée sur CoCoSpoon. Nous avons évalué celui-ci sur trois 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 évaluations nous avons choisi Apache Commons Lang[1] comme projet de référence. Pour simuler un environnement de production, les tests du projet jouent le rôle des actions utilisateur. Les évaluations ont été réalisées sur un ordinateur tournant sur Windows 10, Intel i5-2430M 2.40Ghz, 6Go Ram, JRE 8u60. Dans la suite de cette partie, Commons Lang' répresente la version instrumenté.

2.1 Pertinence

La pertinence de CoCoSpoon est évaluée sur le pourcentage de couverture calculé comparé à d'autres outils : JaCoCo[2] et JCov[3], respectivement utilisé à l'aide de EclEmma[4] et Cobertura[5]. La table 2.1 représente les résultats obtenus. Les différences se trouvent essentiellement sur les éléments statiques du code ainsi que les interfaces, considérés exécutés par les deux autres frameworks.

JaCoCo	JCov	CoCoSpoon
94.1	93.0	83.0

Table 2.1: Comparatif du taux de couverture calculé en %

[1] `https://github.com/apache/commons-lang`
[2] `https://github.com/jacoco/jacoco`
[3] `https://wiki.openjdk.java.net/display/CodeTools/jcov`
[4] `http://eclemma.org/`
[5] `http://cobertura.github.io/cobertura/`

2.2 Temps

L'évaluation sur le temps d'exécution a été réalisée sur dix exécutions successives du programme. La mesure du temps d'exécution est celle retournée par le framework JUnit[6] lors de l'exécution des tests. La table 2.2 représente les résultats obtenus. L'overhead s'explique par le fait que dans le pire des cas, le nombre de lignes exécutées est doublé. Le calcul de la vue est également effectué à chaque modification du modèle.

	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 2.2: Comparatif sur le temps d'exécution en seconde

2.3 Mémoire

L'évaluation sur la quantité de mémoire maximale utilisée a été réalisée sur dix exécutions successives du programme. La mesure est celle retournée par VisualVM[7] lors de l'exécution des tests. La table 2.3 représente les résultats obtenus. On peut constater un léger overhead dû au stockage du modèle.

[6] `http://junit.org/`
[7] `https://visualvm.java.net/`

	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	537
Run 5	455	419	422
Run 6	443	499	511
Run 7	541	459	433
Run 8	403	526	470
Run 9	450	482	487
Run 10	504	513	502
Moyenne	458.2	496.8	469.3
Overhead (%)	-	8.42	2.42

Table 2.3: Comparatif sur la quantité de mémoire maximale utilisée lors de l'exécution en Mo

2.4 Limitations

Lors de l'évaluation de CoCoSpoon, plusieurs problèmes ont été soulevés.

Premièrement, comme décrit dans la section 2.2 l'overhead en temps est assez conséquent. L'utilisateur doit donc pouvoir se permettre de perdre en temps d'exécution dans le but de visualiser sa couverture de code.

L'utilisation de JavaFX 8 permet de réaliser une vue dynamique facilement mais cela impose au projet d'être compilé en utilisant Java 8. Dans le cas où l'utilisateur ne désire pas utiliser Java 8, CoCoSpoon propose une vue texte.

Conclusion

CoCoSpoon constitue une solution élégante afin de répondre à la question initiale : peut-on connaître en temps réel la couverture de code exécuté en production ? Cependant, nous soulignons que la mise en place de notre outils introduit un overhead non-négligeable sur le temps d'exécution. Il pourrait être utilisé sur des applications en phase bêta pour détecter le code mort. En effet la monté en charge de ce type d'application est plus faible et les attentes des utilisateurs sont généralement moins exigeantes.

Notre instrumentation permet actuellement de mesurer la couverture de ligne. Il pourrait être intéressant d'étendre l'outil pour mesurer d'autre type de couverture, tel que couverture de branches ou de méthodes.

Une nouvelle métrique pourrait être également ajoutée : le nombre de fois qu'une ligne est exécutée. Celle-ci permettrait de cibler les parties intéressantes à optimiser dues à leurs nombreuses utilisations.

On pourrait également imaginer un plugin Maven permettant d'automatiser le workflow de CoCoSpoon.

Bibliography

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