Loïc Beaulieu
Antoine Charpentier

Rapport de Projet

Validation et Vérification

Introduction

Le projet que nous avons choisi est le projet d'analyse dynamique.
Ce projet à pour but de générer un rapport en fonction d'un projet donné en entrée de notre outil. Ce fameux "projet en entrée" sera nommé "projet distant" dans la suite de ce rapport.

L'outil que l'on a développé permet principalement de se rendre compte de la couverture de tests d'un projet distant. Il accumule divers données par rapport au bytecode exécuté par la suite de tests unitaires du projet distant.

Afin de réaliser cette outil, nous avons utilisé Javassist un outil permettant d'éditer du bytecode avant que celui-ci ne soit exécuté. Nous avons aussi utiliser Junit afin de démarrer programmatiquement les tests unitaires du projet distant.

Vous trouverez à cette adresse https://github.com/Tanaht/VNVProject le code source de notre projet. Globalement le projet est fonctionnel cependant il n'est pas fait pour être utilisé contre des projet de trop grandes tailles.

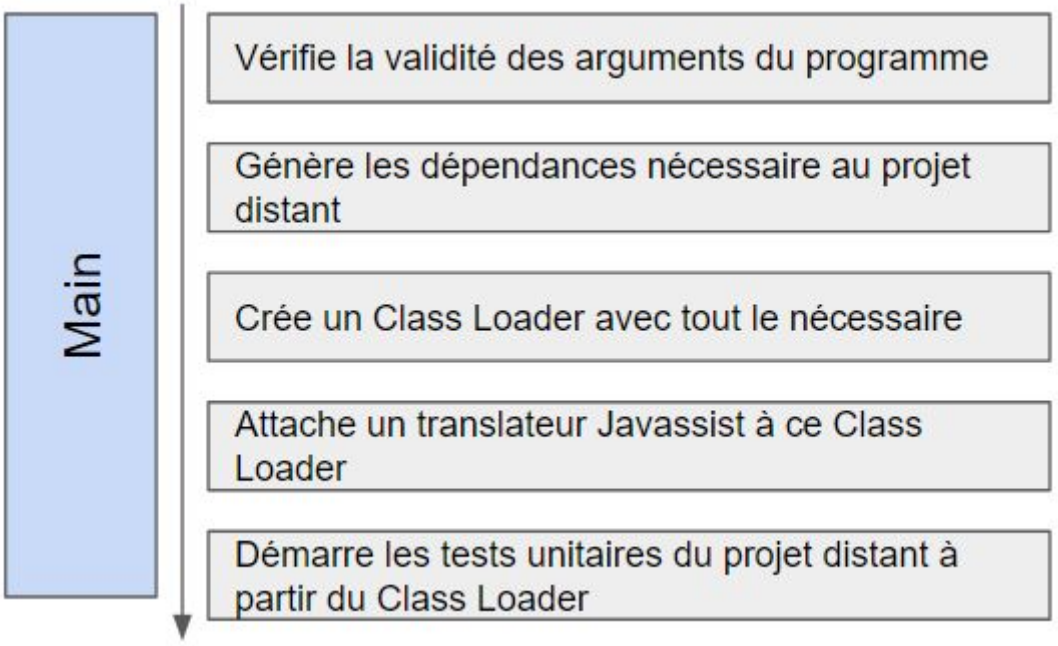
Solution

L'outil ne fonctionne actuellement qu'avec des projets maven respectants les conventions d'usages à savoir:

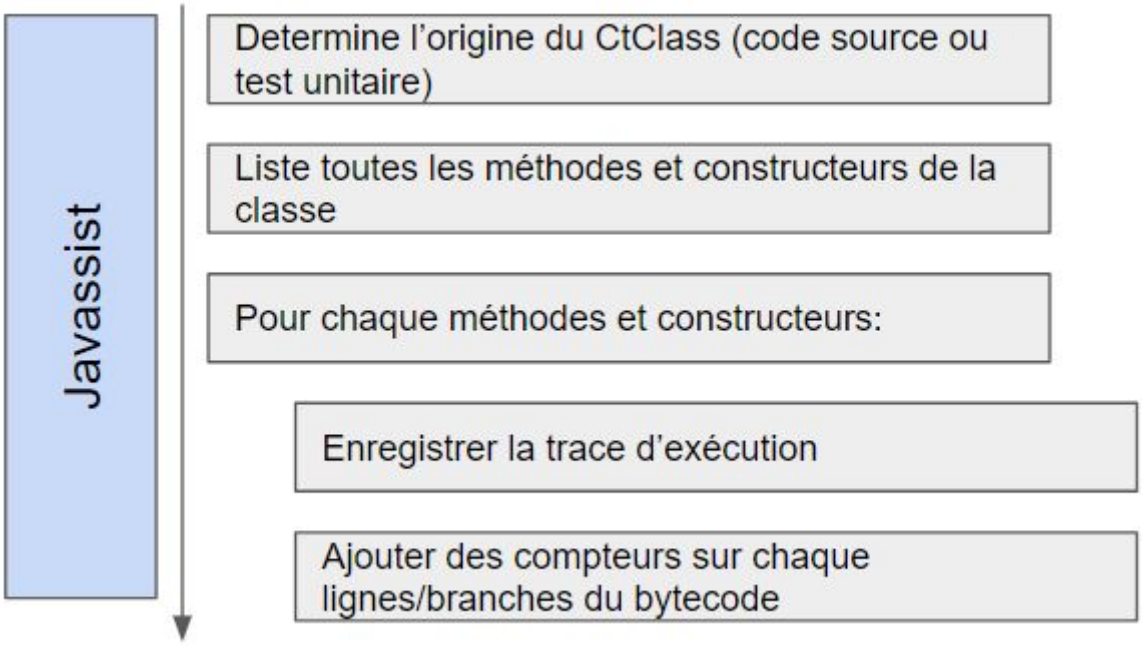
- le code source se situent dans src/main/java
- les tests unitaires se situent dans src/test/java
- le bytecode source se situent dans target/classes
- le bytecode des tests unitaires se situent dans target/test-classes

Le fonctionnement de l'outil en interne peut être séparée en deux, il y a un bloc qui est chargée de vérifier la cohérence du projet distant et de lancer les tests unitaires et un autre bloc qui est chargée de modifier le bytecode du projet distant.

Voici un schéma qui résume l'exécution du premier bloc, le bloc principal qui initialise tout ce qui est nécessaire pour effectuer la manipulation du bytecode:



Le second bloc de notre projet permet donc de modifier le bytecode exécuté par la dernière étape du schéma ci-dessus. En faite lorsque les tests unitaires démarrent ils sont tous chargé grâce au classloader qui à été créer de toutes pièces un peu plus tôt dans ce flux d'exécution. A ce classloader, Javassist à attaché un translateur, qui permet d'intercepter le bytecode chargée par le classloader avant de le restituer. Donc le second bloc ci-dessous n'est pas appelé une seule fois, il est appelé pour chaque classes qui est chargé par le classloader.



Malgré les mécanisme de javassist qui permet de déléguer certains namespaces java pour éviter de les "javassister". Ce n'est pas possible de différencier les tests unitaires du vrai code source de cette manière, car très souvent ils ont le même namespace. C'est pour cela qu'avant d'effectuer toutes modification de bytecode on vérifie l'origine de la classe

intercepté par javassist si il s'agit d'une classe de test on effectue aucune instrumentations.

Cet outil permet de modifier le bytecode de chaque méthodes/constructeurs d'une classe de deux manière différentes:

- Il peut ajouter un appel avant l'exécution du reste de l'opération pour garder en mémoire la trace d'exécution.
- Il peut modifier complètement le bytecode de l'opération en apposant des compteurs à certains point-clé du bytecode permettant ainsi de récupérer des statistiques sur le branch coverage et le line coverage du projet distant.

Pour ce second point qui est un peu plus complexe que le premier, nous avons utilisé un outil mis à disposition par javassist permettant de divisé le bytecode d'une opération en blocs d'instruction contigües c'est à dire sans instruction de saut (GOTO, IF) il s'agit de la méthode **ControlFlow.basicBlocks()** qui retourne donc une liste de **ControlFlow.Block.**

Avec cela il était très facile de connaître les différents blocs d'instructions conditionnels, mais il n'était pas possible de clairement faire le rapprochement entre une instruction de saut et son équivalent dans le code source lisible. Alors notre outil à pris exemple sur cobertura: *"Une ligne du code source est considérée comme exécuté si tout les blocs d'instructions conditionnels ont été exécuté au moins une fois."*
Le dernier défis à été de mettre en relation une instruction bytecode et la ligne du code source correspondante, mais Java lorsqu'il génère les fichiers bytecode, il génère aussi une table qui mappent les lignes du source code et un index représentant le début de l'instruction bytecode associé. Javassist met d'ailleurs à disposition un outil pour effectuer ce travail de traduction **index bytecode <=> ligne sourcecode**, cet outil prend la forme de la classe **LineNumberAttribute.**

notre analyseur dynamique fonctionne à merveille sur commons-cli, mais nous avons observé quelques soucis sur les autres projets recommandés pour tester notre outil d'analyse dynamique. Aussi nous avons mis en place un système d'options pour choisir comment modifier le bytecode:

- Activer ou désactiver la génération de la trace d'exécution.
- Choisir la profondeur de la trace d'exécution à enregistrer.
- Activer ou désactiver la génération du branch coverage.

La plupart des problèmes rencontrés sont causé par la génération de la trace d'exécutions qui ralentit énormément l'outil et qui génère des quantités astronomique de données, dans la plupart des cas il suffit de réduire la profondeur à 3, ou désactiver la trace d'exécution pour exécuter correctement l'outil.

Le projet à été développé en utilisant la méthodologie git flow.

Toutes les informations pour utiliser correctement l'outil sont indiqués sur le fichier readme du repository.

Evaluation

Les 5 programmes proposés dans le sujet ont été utilisé pour tester et améliorer notre outil.

- commons-cli est le projet le plus petit parmis ces cinq là, et il se termine sans problème.
- Commons-collections, commons-lang fonctionne avec le branch coverage et la trace d'exécution de profondeur trois, augmenter la profondeur augmentera la durée d'exécution et la taille du rapport générées.

Discussion

Il avait été planifié de générer une sorte de rapport html/css à la manière de cobertura pour afficher concrètement ou le code était couvert ou non, en soulignant en vert les lignes du code source complètement couvert par les tests unitaires. Cependant beaucoup de problèmes de performances ont été trouvée lors de l'exécution des projets autres que commons-cli. Actuellement ces problèmes de performances n'ont pas été résolue et sont pour la plupart causé par la génération de la trace d'exécution. Nous avons donc décidé de faire de la génération de la trace d'exécution une fonctionnalité optionnelle activable en connaissance de cause. Et pour ajouter plus de contrôle à cette fonctionnalité nous offrons la possibilité de n'enregistrer qu'un nombre prédéterminé d'instructions, si l'utilisateur le désire.

Le gros problème de la trace d'exécution est la mémoire qu'il demande, pendant une grosse partie de la vie de notre projet, toutes les informations étaient stocké dans la mémoire jusqu'a la fin de l'exécution du programme, cependant, lorsque nous avons commencé à tester l'outil contre commons-collections nous avons décidé d'écrire au fur et à mesure dans des fichiers afin de libérer de la mémoire.

Conclusion

En conclusion ce projet à été très instructif, nous avons appris à utiliser un outil d'instrumentation de bytecode au sein d'un projet intéressant. Nous avons été confronté à certains problème au fil du développement, que l'on à pu résoudre ou non. Le projet ne s'est pas finis tout à fait comme prévu, on est resté trop longtemps à n'utiliser que commons-cli pour tester le bon fonctionnement de notre outil, lorsque l'on est passé sur commons-collections certaines partie du code source ont du être réécrite. Cependant l'outil issue de notre projet est utilisable dans une certaine mesure.