Validation et vérification
Rapport de projet : Mutation testing

Julien Laurent et Nicolas Fortun

December 2017

1 Introduction

1.1 Contexte

Le développement d'un programme informatique est principalement constitué de deux phases : le développement d'une fonctionnalité et l'élaboration de suites de tests afin de vérifier la bonne réaction de celle-ci. Ces tests, assurent une qualité minimale ainsi qu'un comportement connu au programme développé. Notre projet s'inscrit dans la vérification de la qualité de ces suites de test.

1.2 Description du projet

Notre projet est une implémentation de la technique du *Mutation Testing*. Afin de vérifier la bonne qualité d'une suite de tests, cette technique suggère d'effectuer des modification au sein du programme. Ces programmes modifiés sont appelés *mutants*. Ensuite, les suites de tests sont lancées sur ces programmes modifiés. Une suite de tests de bonne qualité détectera les modifications effectuées et en conséquence, ces tests échoueront.

1.3 Solution

Notre projet modifie chaque classe du projet passé en paramètre. La principale particularité de notre projet est l'utilisation des lambdas expressions ainsi que les streams de Java 8. Cela permet d'effectuer des traitements de manière plus simple et efficace. Les streams permettent d'effectuer des traitements à la volée, contrairement aux iterators.

1.4 Évaluation

Notre projet comprend quatre opérateurs de mutation :

- Suppression du corps d'une fonction void;
- Remplacement du corps d'une fonction booléenne par **return true** puis par **return false**;
- Modification des opérateurs arithmétiques;
- Modification des opérateurs booléens.

2 Solution

Notre programme fonctionne de la façon suivante :

- Suppression des répertoires générés si existants;
- Récupération du projet cible;
- Initialisation du projet;
- Récupération des tests;
- Application des générateurs de mutant pour chaque classe du projet cible;
- Lancement des tests unitaires du projet cible;
- Génération d'un rapport.

Les algorithmes utilisés pour la génération des mutants sont similaires entre eux :

- Récupération de la classe à muter;
- Sauvegarde de l'état de la classe avant mutation;
- Récupération des méthodes;
- Pour chaque méthode, récupération de son corps;
- En fonction du mutant, application d'un algorithme de modification (par exemple : Remplacement du corps de fonction par un corps vide si l'on est dans le cas d'une méthode *void*);
- Lancement des tests et récupération des tests échoués;
- Ajout de la classe modifiée dans une liste;
- Retour de la classe à son état de départ.

Tous nos générateurs de mutant étendent la classe abstraite *MyProcess* qui elle même étend la classe abstraite *AbstractProcessor*. Cette classe met en place le patron de conception *Template Method*, qui facilite l'ajout d'un générateur de mutant dans notre outil.

Lors de l'implémentation des tests, la classe *TestUnitHandler* a généré des problèmes. En effet, étant une classe statique faisant appel au classLoader (qui est aussi une classe statique), la gestion des retours des appels de fonction étaient difficiles à mocker.

3 Évaluation

Nous ne pouvons pas réellement évaluer la complexité de notre outil, qui est trop dépendant de programmes extérieurs. Au niveau des performances, de nombreuses améliorations sont possibles. Cela est principalement dû à la compilation du projet cible, qui peut prendre du temps. Ces modifications étaient difficiles à implémenter, au vu du temps imparti.

Notre outil est capable de supporter des programmes de toutes tailles sans trop de difficultés. Là encore cette partie est très dépendante de l'exécution du projet cible.

4 Discussion

Afin de vérifier le bon fonctionnement de notre programme, nous avons créé un *DummyProject* qui nous a permis de tester des fonctionnalités de notre outil sur un projet basique.

Lors du passage à l'échelle, nous avons testé notre outil sur les différents projets Apache, tels que *commons-cli* et *commons-lang*. Les premiers tests n'étaient pas concluants. Le compilateur Spoon n'arrivait pas à charger les classes pour les compiler. Ce problème se manifestait pour tous les projets excepté *commons-cli*. Nous avons donc modifié notre manière de compiler le programme cible. Actuellement, un *ProcessBuilder* lance la commande Maven **mvn compile**. Pour que notre outil fonctionne, le projet cible doit respecter certaines conditions, basées sur une structure générale des projets Maven :

- Les sources doivent se situer à *src/main/java*;
- Le projet doit supporter la commande **mvn install** et **mvn compile** dans leurs objectifs de base (install doit tout compiler, source et tests, et compile seulement les sources);
- La compilation doit avoir lieu dans target, et les tests compilées dans le répertoire *target/test-classes*.

La première approche consistait à supprimer les fichiers sources et demander à *Spoon* de générer le modèle pour chaque mutant. Cette méthode n'était pas efficace, car, pour un changement donné, Spoon recréait le modèle. Nous avons donc mis en place une solution permettant de remplacer le fichier modifié sans regénérer le modèle. Lors du développement de l'outil, un problème est apparu. Nous sommes revenus à une version fonctionnelle, n'implémentant pas ces changements. Il semblerait que l'annotation **@CoverageIgnore** de cobertura soit à l'origine du problème.

5 Conclusion

Nous avons donc créé un outil permettant de faire du Mutation Testing. Cet outil gère actuellement quatre types de mutation et génère un rapport lorsque les mutations ont toutes été générées et testées. Cet outil utilise l'analyseur et transformateur de code *Spoon* qui nous permet de manipuler les classes des projets cibles. Le compilateur intégré à *Spoon* a généré des problèmes lors de la génération des **.class** des projets cibles. Pour la suite, il serait intéressant d'ajouter des opérations de mutations au sein de ce projet afin d'avoir une granularité plus fine dans la vérification de la qualité des suites de tests.