istic Informatique Électronique

UNIVERSITÉ DE RENNES 1

Projet V&V : Mutation Testing

M2 ILA - 2017/2018

Étudiants:

Mullier Antoine

Sadok Romain

Introduction

Le sujet choisi pour ce projet est le sujet sur l'approche de test par mutation. Le principe est de prendre un projet, avec comme prérequis qu'il soit construit avec une architecture Maven et écrit en Java, puis générer différentes variantes contenant une modification sur le code. Pour chacune des variantes, on exécute les tests du projet cible. Le comportement normal devrait être un échec lors de l'exécution des tests : on parle de *mutant tué*, dans le cas contraire le *mutant est en vie*.

Les mutations apportées aux projets peuvent être diverses, dans notre projet nous avons établis **6 mutations possibles** :

Opération source	Opération cible
+	**-**
-	**+**
*****	**/**
/	*****
ifeq	**ifneq**
ifneq	**ifeq**

A l'issue des tests pour chacun des mutants générés, on récupère les résultats des tests pour savoir si les mutants ont été tués. Le résultat est ensuite présenté sous forme de rapports CSV et HTML.

Vous retrouverez l'ensemble des ressources de ce projet sur le repository Github[1]. Le readme présente les étapes pour construire et lancer le projet. Un projet cible[2] contenant quelques classes et tests unitaires permet de lancer rapidement le projet pour avoir un aperçu des possibilités de notre outil.

[1] https://github.com/amullier/M2INFO-VV-PROJET/

[2] https://github.com/amullier/M2INFO-VV-DUMMY-PROJET la procédure pour importer et lancer le projet est détaillée dans le readme du projet principal.

Solution

Main

Notre solution se base sur des projets déjà compilés, notre projet demande en entrée du Main un argument spécifiant le chemin du dossier racine du projet. Ainsi, comme le projet est supposé être dans une architecture Maven, on va chercher les fichiers .class c'est le travail du ClassLoader.

Mutator

Ensuite, le Mutator va démarrer l'algorithme du mutation qui peut se décrire de la façon suivante :

```
Pour chaque classe c:
   Pour chaque opérateur bytecode o de c:
      Si o fait parti des opérations sources des mutations possibles:
         Il faut changer l'opérateur de bytecode selon la mutation
         Le TestRunner lance les tests sur le projet cible
         Remettre la classe dans son état original
```

La manipulation du code est faite avec Javassist sur les fichiers .class et donc directement sur le bytecode.

TestRunner

Le *TestRunner* possède une opération `execute()` qui va se charger de lancer l'exécution des tests. Pour cela on utilise la commande `mvn surefire:test` qui va prendre les classes déjà compilées dont fait partie la classe modifiée. Le résultat de cette commande nous permet de savoir si les tests sont passés ou pas à l'aide du retour de commande linux. De plus, la mutation de code peut entrainer des boucles infinies dans le code et donc stopper l'algorithme. Pour pallier à ce problème, nous avons mis en place un délai maximal pour la commande mvn de 5 minutes, si la commande est plus longue que le timeout défini on ne prend pas en compte le mutant créé dans la rapport final.

Pour le reporting, le *Mutator* transmet un *MutantContainer* qui contient toutes les informations utiles pour le reporting : méthode et classe mutées, type de mutation. Avec le retour de la commande ces informations sont transmises au *ReportService* via un objet *Report*.

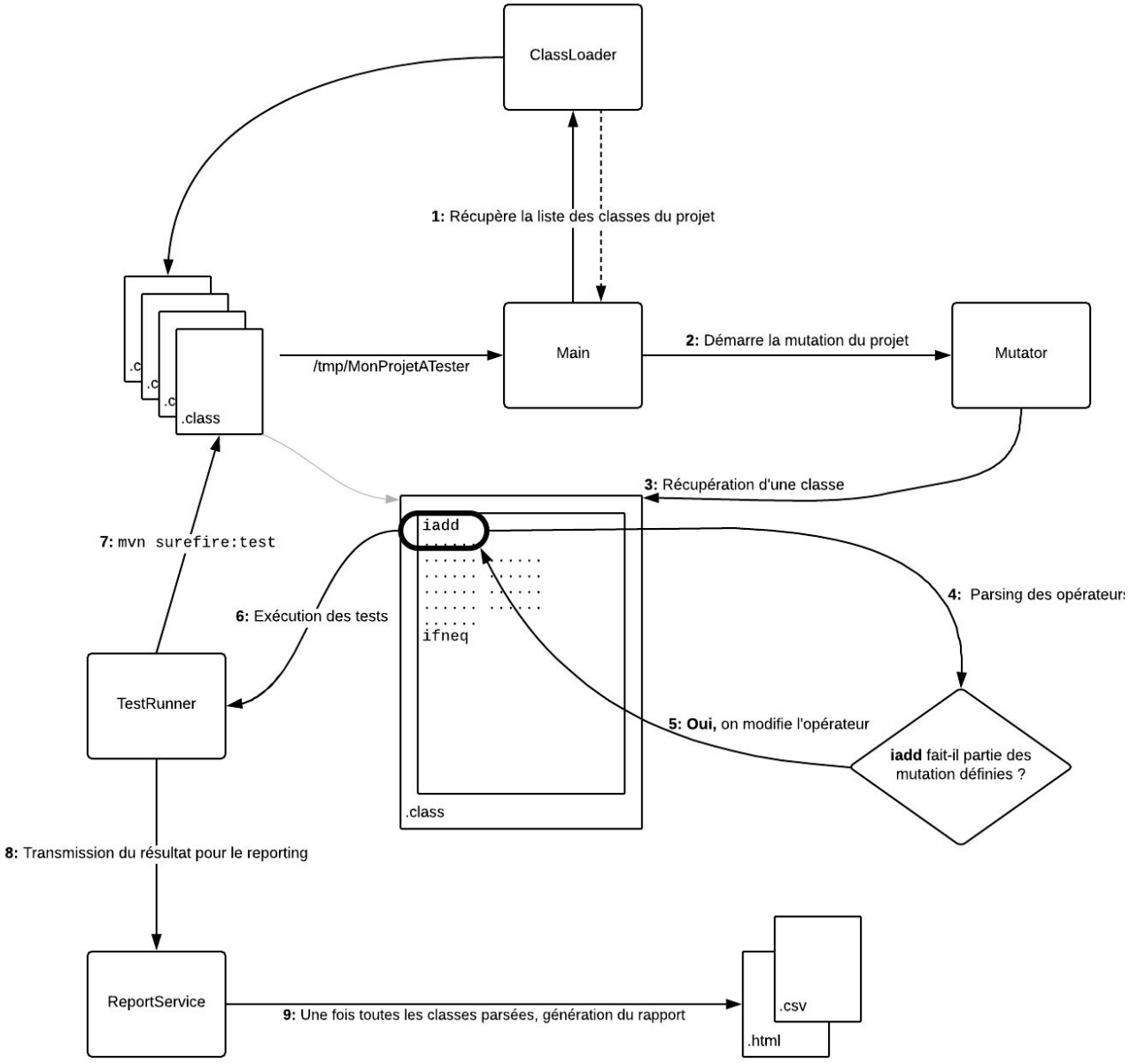
ReportService

Lors de l'exécution le *ReportService* permet de collecter, toutes les informations de l'exécution de l'algorithme de mutation. A la fin de celle-ci, on génère grâce aux informations collectées deux fichiers :

- Un fichier CSV regroupant les informations du *MutantContainer* ainsi que le résultat des tests avec un booléen. Le résultat est pratique pour être potentiellement réinterprété par un programme mais peu lisible pour un humain dès qu'on dépasse la centaine de lignes.

- Le fichier HTML est un version amélioré du reporting classique en CSV. Il présente les informations de base avec une présentation des résultats montrant d'abord les mutants non tués pendant les tests (potentiellement sources d'erreurs dans la suite de tests). Mais aussi des statistiques comme la durée totale d'exécution, le pourcentage de mutants tués.

Voici le principe général de l'algorithme de mutation sous forme de schéma :



Evaluation

Projets cibles

Pour tester notre programme en cours de développement nous avons utilisé notre projet cible, créer ce projet fut d'ailleurs une des premières tâches faites pour travailler sur le projet. Ensuite, nous avons conçu notre programme pour qu'il puisse travailler avec d'autres projets en ajoutant un paramètre en entrée du Main précisant le chemin du projet cible.

Lorsque nous avons étendu notre programme pour qu'il puisse accepter d'autres projets, nous avons commencé par exécuter notre projet sur commons-cli[3]. Cela nous a permis de remettre en cause notre politique de logs en effet il nous fait montrer l'avancement du programme et donner une indication sur le temps restant à l'utilisateur du projet, notre solution à été d'afficher des pourcentage de classes analysées et testées. De plus, nous avons réduit le nombre d'informations affichées à l'écran pour que les logs soient le plus clair possible.

Le temps que prenait les autres projets tels que commons-collection a s'exécuter ne nous a pas permis d'arriver à la fin de l'exécution de la totalité des mutations. Cependant, cela nous a permis de soulever la question des performances de notre algorithme de mutation.

Couverture et qualité de code

La qualité de code a été analysée avec Sonar ce qui nous a permis de rectifier facilement les éventuels problèmes détectés par Sonar.

La couverture de code a été analysée avec l'outil Jacoco qui intégré à Sonar permet de bien identifier les points à améliorer sur la suite de tests. L'outil a été choisi pour son intégration déjà établie dans Sonar. Actuellement, la couverture de code est de 93%.

Discussion

Le problème des performances est un facteur qui peut être dû à plusieurs choses :

- Les tests ont été effectués sur les machines délivrées par l'université, il aurait été intéressant de pouvoir tester notre programme sur des machines plus performantes pour comparer les performances et connaître précisément l'impact que le matériel a sur cette problématique.
- Peut être que la parallélisation des traitements d'exécution des tests avec des Threads aurait pu nous permettre d'améliorer les performances. Mais cela n'est pas certain car lors des exécutions la machine était quasiment à 100% sur les quatres processeurs pendant toute la durée de l'exécution d'où l'intérêt du premier point.

Le reporting pourrait être amélioré en faisant référence au code existant et muté pour les cas où le mutant est resté en vie.

[3] https://github.com/apache/commons-cli