Dev-Ops
Rapport Rendu n°1

Groupe 9 :

Julien Tocci (G3)
David Borry (G1)
Justin Grivon (G2)
Baissangour Akhmadov (G2)
Thomas Gillot (G1)

Introduction

Afin de valider les tests d'un projet, nous pouvons créer des mutants du code en modifiant intelligemment celui-ci, puis les tester, ce qui va nous permettre de voir l'impact d'une modification sur le code au niveau des tests. La difficulté réside dans le fait de créer des bons mutants afin de mettre en avant les failles non traitées par les tests.

Travail réalisé

Nous avons commencé par créer un environnement de travail totalement automatisé ce qui nous a permis en outre de revoir les bases de maven. Cela fait nous avons commencé à créer des mutants, d'abord basique puis plus élaboré.

Plomberie

Nous possédons deux projets: un avec les mutants et un avec le code Island.

Nous appliquons les mutants un à un au code métier, nous récupérons ensuite le code résultat ainsi que les résultats des tests après l'application de chaque mutant.
Pour automatiser ce processus nous utilisons un script:

- On lance la commande maven package dans le projet contenant les mutants
- On copie le fichier .jar résultat dans un dossier du projet contenant le code Island
- On entre dans le répertoire du projet contenant les fichiers sources des mutants
- Pour chaque fichier source mutant présent dans ce dossier on récupère le nom du mutant (sans extension de fichier) et on lancer la commande maven package du projet contenant le code Island en donnant en paramètre le nom du fichier comme valeur de variable
- On récupère le résultat d'exécution des tests dans un fichier texte

Nous avons fait le choix de ne pas utiliser surefire-log pour voir le nombre d'erreurs lors des tests sur un mutant mais plutôt un script shell afin de pouvoir personnaliser entièrement l'affichage.

Mutants

Nous n'avons pas décrit dans cette partie la totalité des mutants que nous avons créé (nous étions limité par la taille du rapport). En voici 3 parmis les 6 disponibles :

IntComparaisonProcessor

Ce mutant assez simple permet néanmoins de mettre en avant plusieurs techniques utilisées dans le code métier et test. On utilise un élément de type **CtLitteral** pour récupérer directement des expression comme les entiers, strings, booléens...

Chaque variable de type **int** est récupérée, incrémentée et enregistrée dans le nouveau code. Ce qui a pour conséquence que sur les 103 tests unitaires, **42** ont échoué dont **13 à cause d'erreurs**. On peut rapidement en déduire que la présence de nombreuses comparaisons d'entiers dans les tests est en partie due à ce résultat.

Quant aux erreurs, elles s'expliquent entre autres par la présence de **tableaux** dans le code. En effet, l'application ayant comme fonctionnalité de traiter des objets **JSON**, dont des tableaux ne peut pas fonctionner correctement si elle tente d'accéder à un indice dépassant la taille d'un tableau, ce qui est causé par ce mutant.

Avec un taux de réussite de **59%** pour les tests unitaires, **IntComparaisonProcessor** montre l'importance des manipulations d'entiers et de tableaux dans l'application testée.
Cette grande différence dans les taux de réussites montre également que les comparaisons d'entiers et parcours de tableaux sont efficacement gérées par les tests unitaires.

SwitchProcessor

Ce mutant a deux objectifs:

- Supprimer le premier case dans un bloc switch
- Si le bloc switch ne contient pas de case default, ajouter un commentaire d'avertissement dans le code juste avant ce switch pour indiquer l'absence de default

Dans le repository de rendu, la partie qui supprime le premier case a été commentée, la raison pour cela est que bien que le code ait été correctement modifié certains tests conduisaient à un blocage (boucle infinie).

ContractProcessor

Ce mutant permet de modifier le constructeur de la classe Contracts, est ainsi de falsifier les données enregistrées à partir d'un objet JSON. Son action est de remplacer les valeurs des matières premières par 0 et ainsi voir comment réagissent les tests qui utilisent le contrat.
7 tests ne passent pas, 5 failures et 2 errors, les failures sont "normales" dans le sens qu'on test la quantité restante à partir du contrat qui a été modifié, mais les errors sont plus intéressantes, il y a ici un problème assez important : la méthode Task.getDecision().getAction() renvoie un NullPointer alors qu'elle est sensé retourner une action. le cas où toutes les ressources sont à 0 n'a pas été pris en charge ici.
Ce scénario est peu probable mais ça reste un cas qui n'a pas été traité et qui a été repéré par ce mutant.

Prise de recul

Au niveau de la qualité logicielle, l'utilisation de mutants présente plusieurs avantages.
En premier lieu, l'utilisation de mutants permet de vérifier la qualité du code et/ou des tests. Si on prend même un mutant simple qui remplace des symboles inférieur / supérieur par des symboles inférieur ou égal / supérieur ou égal, on pourra vérifier comment les tests réagissent à l'addition d'un nouveau cas « extrême », extérieur à l'intervalle initial mais proche de celui-ci.

Dans le même principe, nous avons créé un processeur qui supprime un des **case** dans un bloc **switch**, l'objectif étant de voir comment le code et le test réagissent au fait que dans un cas où avant on avait du code particulier prévu, on n'en ait plus. Cela permettait de vérifier qu'au niveau code ou test on avait bien pris en compte les situations où on entre dans un **switch** avec une valeur qui n'a aucun **case** correspondant, par exemple par le biais de **default**.

Un autre bénéfice des mutants est qu'ils permettent d'automatiser des vérifications au sein du code. Par exemple, reprenons l'exemple du switch, nous avons créé un mutant qui vérifie si un bloc **switch** contient un case **default**. Si ce n'est pas le cas, le mutant ajoute un commentaire avec un avertissement qui indique que ce switch ne gère pas les cas default.

On pourrait aussi appliquer ce type de vérifications aux blocs try catch par exemple, afin de nous assurer qu'on gère des Exceptions précises (comme NullPointerException) et non pas un type générique (Exception).

À plus grande échelle (i-e: appliqués à des travaux à plus longs termes), les mutants ont d'autres avantages.

En effet, ils permettent également de vérifier bien entendu la qualité des tests, mais aussi la couverture de ces derniers. En effet, si l'on crée un mutant, et que les tests déjà écrits passent toujours il y a deux possibilités à envisager :

- les tests et/ou le code sont incorrects : dans ce cas le mutant ainsi créé aura permis de mettre en relief une erreur de code ou de tests, non relevée par les tests unitaires.
- les tests n'accèdent jamais au code modifié : dans ce cas le mutant aura permis de mettre en relief un manque de couverture sur une partie de code en particulier.

Les mutants sont donc une alternative aux frameworks de vérification de couverture de code habituels (tels que ceux intégrés dans les IDE ou encore SonarQube).

De plus, le Mutation Testing a l'avantage de pouvoir fournir une donnée chiffrée pour mesurer la qualité des tests unitaires effectués et ainsi permettre d'apprécier la qualité globale du projet en lui-même. Cette donnée se nomme le "Mutation score" et se calcule de cette façon :

$$Mutation\ Score\ = \frac{Killed\ Mutants}{Total\ number\ of\ mutants} * 100$$

Les "killed Mutants" étant les mutants qui ont été les morceaux de codes rejetés par les tests unitaires et le "Total number of mutants", le nombre total de mutants écrits et soumis aux tests
unitaires écrits. Un Mutation Score de 100% indique donc que tous les mutants écrits et soumis aux tests, ont été rejetés. Ce qui indique l'optimalité du code ainsi écrit.

D'un autre côté les mutateurs ont de multiples désavantages pour les travaux à grande échelle.

Premièrement, lors de la création des mutateurs, même si un seul morceau de code a été modifié, à chaque fois le code source est entièrement recréé ce qui est inconvénient tant au niveau de la consommation de temps et de ressources. Appliqué à un code source contenant des milliers de lignes de code (voire des millions) , la génération de mutants peut-être coûteux.

Comme nous l'avons découvert au cours de la réalisation de ce travail, nous avons dû reprendre le code déjà effectué l'année passée par un de nos camarade, et, comme le Mutation Testing s'appuie sur la modification de code, il faut, pour mettre en place des mutants, bien connaître le code source au préalable pour pouvoir faire les bons choix dans la création de mutants (d'autant plus que la création de mutants peut-être coûteuse). Il est

donc difficile pour la maintenabilité du code source, que les mutants soient effectués par d'autres personnes que celles qui ont codé le code-source.