POLYTECH NICE-SOPHIA

Rapport de DevOps

Mutateurs et tests

DALIE Basil, LAFAURIE Paul, LEQUIENT Steven, MERINO Mathieu

I. Introduction

Ce projet consiste à automatiser une suite d'opérations chargées de jauger l'efficacité des tests de notre projet, au moyen de la librairie Spoon et de scripts d'exécution.

Les tests constituent une étape importante du cycle de vie du code, et ajouter une étape automatisée au processus de livraison continue chargée de les vérifier augmente l'assurance en la qualité du code.

II. Arborescence du projet

L'arborescence du projet est divisée entre les dossiers sources et les dossiers générés.

Les dossiers sources sont:

- *island*: contient les sources et tests unitaires du projet island
- *spoon_rewriter*: qui contient les sources des mutants utilisant spoon

Il existe un *pom.xml* à la racine du projet et un dans chaque dossier source. *Island* et *spoon_rewriter* sont définis comme des modules du projet.

A la racine du projet se trouvent les différents fichiers rendant possible la pipeline de tests automatisée. La boucle principale du système se trouve dans le fichier *script.sh* (*script.bat* pour windows). Pour chacun des mutateurs existant, s'exécute la boucle suivante:

- On ajoute dans le dossier *generated*, qui contiendra tous les codes sources altérés, un sous-dossier portant le nom du mutateur. On recopie dans ce sous-dossier une grande partie des fichiers du dossier *island*: les tests, les maps (informations nécessaires aux tests), et les ressources du programme. On ajoute également à ce sous-dossier un fichier *pom.xml* différent de celui de island (celui-ci ne contient pas d'appel à *maven-spoon-plugin)*.
- Le *pom.xml* de *island* contient un appel à *spoon-maven-plugin*, dont le rôle est de générer des sources modifiées par un mutateur spoon. On modifie la ligne du pom qui spécifie quel mutateur utiliser afin d'utiliser le mutateur de l'itération actuelle de la boucle. On appelle ensuite *mvn compile* à la racine, ce qui va exécuter les instructions du *pom.xml* de island et créer des sources modifiées par mutateur dans *island/target/generated-sources/spoon*. On bouge alors ces sources dans *generated/{currentMutator}/src/main/java*.
- Il ne reste plus qu'à effectuer les tests et récupérer le rapport. On se déplace dans *generated/{currentMutator}* et on appelle *mvn surefire-report:report* qui va générer un fichier html dans le dossier *results* à la racine, contenant les données de réussite des tests.

A la fin de ce script, on a donc effectivement créé autant de sous-dossiers contenant des sources modifiées qu'il existe de mutateurs, et on a appliqué les tests originels à chacune de ces sources mutées.

Nous avons également mis en place une solution docker, pouvant être executée en appelant le script *docker_linux.sh*. L'avantage de cette solution est qu'elle fonctionne a 100% sur tout système Linux ayant Docker (nous n'avons pas encore implémenté de script de lancement pour Windows à cause de difficulté avec la gestion des volumes docker sur Windows) et qu'elle ne touche pas le système hôte.

III. Les mutants et leurs impacts

Plusieurs choses sont à garder en tête lors du choix d'un mutateur. Tout d'abord certaines mutations trop importantes vont simplement faire des erreurs de partout et n'apporter aucune information.

Par exemple, ajouter des caractères à la fin de toutes les strings va forcément causer des tonnes de problèmes. Nous nous sommes retrouvés avec énormément d'erreurs après avoir essayer un mutateur de ce genre.

Il faut trouver un changement ayant un impact pertinent sans qu'il ne casse tout le programme. Selon les résultas obtenus, on peut obtenir des données intéressantes sur

l'utilité de nos tests. Par exemple si on utilise un mutateur modifiant les formules mathématiques au hasard dans un programme contenant beaucoup de calculs, et que les tests passent tous, ce mutateur nous aura appris que nos tests ne vérifient aucun calcul ni formule.

Voici les quatre mutants que nous avons mis en place dans le cadre d'island.

1. Condition inverser

Un mutateur repérant les bloc if possédant un else, et inversant le contenu de ces deux blocs.

Nous avons pensé qu'il serait utile de tester si les tests passaient toujours en inversé une partie du contrôle de flot du programme, car island est un programme de décision (on décide dans quel direction le drone et l'expédition vont bouger). C'est donc par définition un programme utilsiant beaucoup de tests dans ses classes de comportement. On s'attend donc ici à beaucoup d'erreurs dans les tests des Behavior (classes qui choisissent les actions à entreprendre.

Les résultats des tests montrent que le seul test des Behavior ne passe pas. Ils nous aurait fallu plus de tests de Behavior pour s'assurer vraiment de l'utilité de ce mutateur, mais c'est un résultat attendu.

2. Add to variables

Un mutateur ajoutant une valeur aléatoire entre 0 et 4 aux int, double et float du programme.

C'est un mutateur un peu général qui va impacter une grande partie du code. Pour commencer, on s'attend déjà à avoir des erreurs de tableau out of bound, puisque l'indice peut risquer d'être trop grand par rapport à sa taille après le passage du mutateur. Mais en ignorant les passages du code générant des erreurs de ce genre, on peut se concentrer sur le reste, et repérer les endroits du code où les tests passent alors qu'il s'agit d'une partie utilisant beaucoup de nombres.

Les résultats des tests après ce mutateur sont assez révélateurs. On ne compte pas énormément de failure, le success rate est de 75%. On s'attendait à plus, ce qui peut signifier un manque de rigueur dans les tests. Cependant nous n'avons pas de couverture de test pour la partie la plus interessante du code avec ce mutateur, c'est à dire le Behavior de collecte de ressources. On aurait bien voulu savoir si les tests de ce behavior, devant vérifier des valeurs numériques exact (on a ramassé tant de cette ressources, fabriqué tant de celle là…).

3. Minus to plus

Un mutateur échangeant les opérateurs binaires de soustraction (pas les indicateurs de nombres négatifs) en "+" (pas l'inverse sinon on se retrouve avec des soustractions de strings).

En inversant l'ensemble des signes négatifs en signes positifs pour chaque opération binaires, nous observons un echec sur uniquement trois échecs (failure) lors de l'exécution des tests. Deux d'entre eux concernent la classe Direction, qui permet de stocker une direction empruntable par le Drone dans le projet Island. En effet, la méthode getRight permet d'obtenir la direction ordinale correspondant à une rotation à 90° vers la droite par rapport à la direction courante. En modifiant les signes on aboutit ainsi à une rotation inverse, ce qui provoque une erreur. La même erreur est répercutée sur la classe DroneEchoBehaviour qui utilise des opérations sur les directions.

Ensuite, en ce qui concerne l'échangeur négatif/positif dans le cadre des opérations binaires. On peut ainsi mettre en doute la qualité des tests unitaires car on s'attendrait à ce que plus d'erreurs soient mises en évidence par ce mutant étant donné la fréquence d'utilisation des opérations binaires de soustractions utilisés dans le cadre de ce projet. Cependant, on peut également critiquer la pertinence de ce mutant : en effet étant donné que l'on change la sémantique de la méthode getRight(), il est naturel qu'au moins une petite proportion de test échoue.

4. Operators swapper

Un mutateur changeant les divisions en multiplication et vice versa, et les && (and) en || (or) et vice versa.

L'inversion des and en or et vice versa est intéressant, encore une fois, pour la partie comportementale de notre application. On s'attend à ce qu'un tel mutateur impacte fortement les tests de Behavior, puisque le drone devrait se mettre à agir de façon inattendue.

Cependant on voit dans les résultats des tests, que l'unique test de comportement passe sans failure. C'est surprenant et surement le signe de tests mal pensés. Cela nous fournit donc une piste d'amélioration pour nos tests. Il est important de bien y réfléchir et d'étudier la situation pour savoir s'il s'agit d'un cas exceptionnel où les tests sont bien fait et cette opération particulière n'a eu, contre toute attente, aucun impacte sur la fonctionnement du code.

Encore une fois on regrette de ne pas avoir mis en place plus de tests pour les behavior, sachant que l'application dispose de 5 classes Behavior.

IV. Conclusion

Tester son programme requiert d'être imaginatif quant à la façon dont on essaye de couvrir le plus de cas de fonctionnement possibles. Tester la pertinence des tests par des mutateurs requiert également un certain doigté, car il est très simple de faire un mutateur qui change beaucoup de choses sans rien nous apprendre.

Différents types d'applications rendent plus pertinents certains mutateurs et moins d'autres. Une application de modélisation mathématiques bénéficiera beaucoup de mutateurs intervenant dans les formules mathématiques, en permettant d'identifier les zones de code passant toujours les tests après mutation, ce qui nous indique quels tests sont à affiner. Une autre application utilisant des automates finis pourra elle utiliser un mutateur échangeant le code interne de deux états d'un automate, pour changer son comportement et repérer les tests trops faibles qui ne remarqueront pas ce changement.

Les résultats obtenus sont une indication des tests potentiellement trop imprécis ou erronés. Il est important de bien interpréter le résultat des tests après mutation, puisque par exemple beaucoup d'erreurs peuvent signifier que le mutateur a cassé le code d'une manière non informative, ou bien simplement indiquer que le test est bien fait et qu'il rate après une bonne modification du mutateur.

Il n'est donc pas nécessaire, lorsqu'on regarde les résultats des tests du code mutés, de regarder l'ensemble des résultats. Certains mutateurs sont adaptés pour tester une certaine partie du code. Peu importe si d'autres parties sans rapports ne passent plus au test, c'est n'est pas sur ces parties là que le mutateur nous apprend quoi que ce soit.