TD Mutateurs

Équipe Sous Entendu (n°3)

Berger Matthieu

Chevalier Mathias

Mônier Marhold

Palumbo Adrian

Zago Arnaud

POLYTECH
NICE-SOPHIA

Introduction

L'objectif de ce TD est de mettre en oeuvre un processus permettant d'acquérir des informations et un retour sur la qualité des tests d'un projet. Cependant, notre solution se doit d'être automatisée, ou, tout du moins, être déclenchable facilement par l'utilisateur *(à défaut d'être intégrée dans un cycle de développement continu)*. Ce document décrit à la fois l'implémentation, les choix qui ont été faits pour mettre en place notre processus ainsi que les décisions que nous avons réalisées sur les mutateurs eux-mêmes.

Le projet test sur lequel nous avons travaillé est le code d'un automate décisionnel, créé pour le projet ***Island*** du cours de "Qualité et Génie Logiciel" de SI3. Notons que ce projet disposait d'un grand panel de tests unitaires *(une couverture de tests de 80% au minimum était demandée par le client)*, mais ne disposait pas de tests d'intégration ni de tests fonctionnels.

Cela convient néanmoins parfaitement à notre objectif, qui est de mettre en oeuvre un processus de vérification de la qualité de tests, quels qu'ils soient. Le code du projet est assez varié, certaines analyses sont mathématiques, d'autres textuelles. La notion d'héritage est aussi très utilisée, et tout ceci en fait un projet hétérogène et intéressant à manipuler.

Mécanisme d'automatisation

Choix structurel

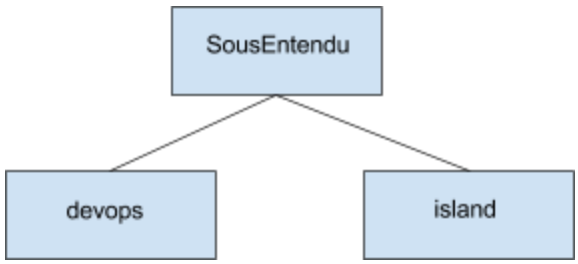
Nous avions deux pistes pour l'automatisation et l'exécution des processeurs ***Spoon*** : la première était d'utiliser des commandes de la librairie *Spoon* dans un script qui gère lui-même la compilation et l'application des processeurs pour ensuite déclencher les tests et la génération d'un rapport. La deuxième étant de modifier *à la volée* le contenu du fichier *pom.xml* qui définit les processeurs que *Spoon* utilisera ; ce choix est défini dans la balise ***<processors>*** :

```
<processors>
    <processor>spoon.processors.LTtoGT</processor>
    <processor>spoon.processors.LTtoGTNotInLoop</processor>
    ...
</processors>
```

Balise "processors" gérant les processeurs à utiliser

Nous avons choisi la deuxième solution pour différentes raisons : la première étant que l'on délègue à *Maven* la compilation et l'enchaînement des phases, et il n'y a de plus aucun changement à effectuer au niveau de l'architecture du projet. *Maven* aurait de toute façon été nécessaire au moins à un moment de l'exécution puisque ce même projet avait déjà des dépendances.

Notre structure est donc la suivante:



Nous avons ainsi un projet ***parent*** *(**SousEntendu**)* de 2 *modules*, l'un gérant la partie *Spoon* et la modification du comportement *(**devops**)*, l'autre étant la partie gérant le projet ***Island*** *(le pom modifié à la volée étant celui d'Island)*.

Génération de rapports

Afin de permettre la génération de rapports plus facilement exploitables, un plugin a été ajouté : il s'agit du plugin ***Surefire*** et son complément ***surefire-report***. On peut ainsi, d'une simple ligne de commande, exécuter les phases du *lifecycle* souhaitées, à savoir le *build*, l'application des processeurs, les tests et enfin la génération des rapports.

Il est en effet inutile d'aller plus loin : créer les *packages* n'aurait pas de valeur. Il est même nécessaire de ne pas aller plus loin, car il y a un gain de temps non négligeable à chaque exécution de la commande *Maven*. Notons que puisque notre script exécute cette commande de manière itérative sur chacun des processeurs, on gagne ainsi un temps proportionnel au nombre de processeurs.

La commande injectée dans notre script est la suivante: ***mvn surefire-report:report***. Cette commande déclenche l'exécution de toutes les phases jusqu'aux tests, puis génère le rapport, qui n'est pas stylisé. Afin d'avoir un rapport plus "beau" *(où le style CSS s'appliquerait)*, il faudrait utiliser ***mvn site***, mais l'utilisation d'une telle commande rend le temps d'exécution beaucoup plus important, pour un bénéfice minimum ; on a alors choisi d'utiliser simplement le rapport de base, puisque les données restent lisibles. Chaque rapport est généré à la racine de notre projet, dans le dossier ***/output***, et est de la forme "*NomDuProcesseur_report.html*".

Script d'exécution

Le script lançant l'exécution et le rapport se trouve dans le dossier ***/scripting***, à la racine du projet. Premièrement ce dernier va récupérer tous les processeurs *(qui se trouvent dans le module **devops**, et qui doivent forcément avoir comme package **spoon.processors**)*. Ensuite, pour chaque processeur, on va modifier le fichier *pom.xml* du module *Island* en supprimant tous les processeurs présents dans la balise "processors" *(cf. plus haut)* et en ajoutant que le processeur courant.

Ainsi, nos fichiers sources seront modifiés avec le bon processeur, et nous pourrons générer par la suite le rapport avec la commande *Maven* décrite ci-dessus, puis le déplacer où il faut *(voir ci-dessus)*.

Mutateurs

Prise en main de l'outil Spoon

Dans un premier temps nous avons choisi d'élaborer un mutateur brutal et irréfléchi pour comprendre le fonctionnement de *Spoon*. Ce premier mutateur permet de changer tous symboles de comparaison inférieurs *(ou égal : </<=)* en supérieurs *(>)*. Il est évident que ce mutateur n'est pas très pertinent, sachant qu'il va modifier ces symboles dans les boucles et donc empêcher la majorité du code de fonctionner correctement, en plus d'entraîner de potentielles boucles infinies et de réellement détruire le comportement du code qui serait toujours fonctionnel.

Une fois compilé et testé on observe que nous avons, sur **97** tests, **20** echecs et **23** qui sont en erreurs *(ces derniers sont évidemment provoqués par les modifications du mutateur)*. En revanche, avec **55%** de tests qui passent toujours, nous pouvons conclure que les tests de ce projet ne sont pas forcément pertinents ; en effet, avoir plus de la moitié des tests qui passent alors que le projet a subi de grosses modifications prouve qu'il y a un problème.

Création de mutateurs pertinents

Dans la continuité du mutateur cité ci-dessus, le mutateur que nous avons implémenté par la suite effectue les mêmes changements, mais plus intelligemment : il ne sélectionne cette fois **pas** les signes de comparaison dans nos boucles ***for*** *(dans notre cas, aucun* ***while*** *n'a été implémenté, mais il suffirait d'ajouter une condition dans le code du mutateur pour toucher aussi les* ***while****)*.

On évite ainsi de perdre tout le comportement du code, mais on garde des modifications importantes. Et effectivement, on obtient alors, sur nos **97** tests, **6** échecs et **15** tests en erreurs. Le problème est que certaines actions du code sont basées sur une étude d'un objet JSON, dont on ne connaît pas la longueur avant de le recevoir ; de ce fait, on agit tout de même sur le comportement du code tout en minimisant les dégâts.

Pour continuer, il nous semblait pertinent d'essayer d'ajouter un caractère aléatoire à nos modifications ; nous avons donc implémenté un mutateur remplaçant les signes de comparaison, mais ne s'appliquant qu'à une partie de ces derniers. La partie aléatoire de processus n'est pas complexe : ici, il s'agit simplement d'un ***Math.random*** entre 0 et 1, qui est inférieur ou non à un seuil.

Conclusion & Résultats

Rendons-nous à l'évidence : les tests écrits sur le projet *Island* ne sont pas d'une très bonne qualitée.

En effet, c'était les premiers tests unitaires que nous écrivions, et le comportement de ces derniers est très basique et extrêmement naïf. Ajoutons à cela que le code fonctionnel du projet n'est pas non plus excellent, on arrive à un comportement global très hétérogène et par conséquent complexe à analyser.

PROCESSEUR	SUCCÈS	ECHECS	ERREURS	% DE RÉUSSITE
Aucun	97	0	0	100 %
GTtoLT	68	15	14	70 %
LTtoGT	54	20	23	55 %
GTtoLT - *sans les boucles*	68	15	14	70 %
LTtoGT - *sans les boucles*	76	6	15	78 %
GTtoLT - *aléatoire, seuil de 30%**	82	11	4	84 %

"Les échecs sont prévus avec des assertions, les erreurs sont imprévues"
GT : Greater Equal or Greater Than > / >=
LT : Lower Equal or Lower than < / <=
** ce test repose sur un facteur aléatoire.*

Encore une fois, l'analyse de ces résultats nous montre simplement que les tests ne sont pas réellement pertinent ; pour chacun de nos processeurs, nous avons quasiment 80% des tests qui passent toujours, ce qui est un réel problème !

En effet, à de nombreuses reprises, nous modifions clairement le comportement de notre code, allant même jusqu'à échanger les opérateurs de comparaison ; cependant, nous avons toujours un pourcentage de test élevé qui continuent "d'être au vert".

Ainsi, grâce au processus mis en place pour appliquer et analyser nos mutants, nous avons a bien souligné les limites du système précédemment implémenté pour *Island*.

Néanmoins, notre processus global, du script au rapport, est fonctionnel et compréhensible, et est donc adaptable à différents projets, la structure *Maven* étant un réel avantage. Il peut être vraiment très intéressant d'avoir un retour sur la qualité de nos tests, et si l'on souhaite modifier/améliorer ces tests, nous générons même aujourd'hui un rapport détaillé avec un retour test par test, ou même par classe de test, ce qui permet évidemment ensuite d'identifier les défaillances et procéder aux ajustements adéquats.