Rapport du TD n°1 de DevOps

Mutants avec Spoon

Antoine Aubé
Maxime Carlier
Thibaut Terris
Xiaohan Huang

I. Fonctionnement

A. Description du dépôt

Le dépôt Git est constitué de la façon suivante :

- Un projet Maven “spoon-mutators”.
- Plusieurs scripts Bash.
- Un, plusieurs, projet(s) Maven (par exemple “kata”).

1. spoon-mutators

Il s’agit d’un projet que l’on construit avec Maven et qui contient les mutateurs Spoon que nous avons défini. Le pom.xml contient à cet effet une dépendance vers Spoon (pour pouvoir décrire les mutateurs).
Les mutateurs que nous avons défini sont :

- Ne rien changer au code (mutateur d’essai pour s’assurer que le build fonctionne) ;
- Remplacer les opérateurs “-” et “--” par “+” et “++” ;
- Remplacer les “==” par des “!=” et les “!=” par des “==” ;
- Inverser le sens des opérateurs de comparaison ;
- Remplacer les opérateurs “+” par des opérateur “*”.

Ce dernier mutateur ne fonctionne que si le code ne contient pas de concaténation de chaînes de caractères (il n’existe pas d’opérateurs “*” entre chaînes de caractères en Java).

Nous avons implémenté la possibilité de modifier uniquement un pourcentage des éléments cibles. Il faut pour cela modifier le fichier “percentages.properties” situé dans le dossier “resources” du code (toujours de spoon-mutators). On peut de cette manière spécifier pour chaque mutateur le pourcentage de modification.

2. Les scripts Bash

Les scripts Bash sont notre “tuyauterie” dans ce projet. Le déroulé est décrit dans la partie qui suit. Le rôle général de chaque fichier est défini ci-dessous :

- `execution.sh`

C’est le script principal, il exécute toute la série d’actions en utilisant les autres scripts.

- `clean_repertory.sh`

Équivalent d'un make clean ; il supprime les dossiers des projets mutés.

- `build_mutants.sh`

Il construit les projets mutés pour chacun des mutateurs. Il ne s'agit que de génération de code, pas de compilation.

- `launch_tests.sh`

Il exécute chaque mutant et génère les rapports sous forme de page HTML.

3. Les autres projets

Les autres projets sont de simples projets Maven.
Ils n'ont aucune particularité, sinon d'être présent pour pouvoir exécuter le script et de ne fournir une base de code et de tests sur lesquels, venir appliquer nos mutations.

B. Déroulement d'une exécution

L'utilisateur exécute les mutations en tapant dans un terminal possédant un interpréteur Bash la commande suivante :

```
./execution.sh <DossierÀMuter>
```

Par exemple :

```
./execution.sh kata
```

Si l'utilisateur utilise une machine Windows, et qu'il utilise un interpréteur tel que MinGW reposant sur l'implémentation MSys, nous offrons à celui-ci la possibilité d'exécuter nos scripts avec la commande :

```
./execution.sh <DossierÀMuter> --windows
```

Par exemple :

```
./execution.sh kata --windows
```

Les étapes de l'exécution sont :

- Les dossiers produits par d'éventuellement mutations précédentes sont nettoyés.

Il s'agit simplement de l'exécution du script ./clean-repertory.sh.

- On construit spoon-mutators avec la commande mvn clean install.

De cette manière, spoon-mutators est disponible dans le dépôt local et utilisable par la suite.

- Pour chaque mutateur, on produit un dossier mutant.

Un dossier mutant contient une copie du code source (src/main/java) dont le contenu est modifié par un mutateur.
On le produit de la façon suivante :

- Une copie temporaire du pom.xml du projet à muter est créée.
- Le pom.xml est modifié par l'ajout d'une dépendance vers spoon-mutators.

- Le but `mvn clean generate-sources` est exécuté.
- Le code généré est déplacé dans un nouveau dossier “mutant_NomDuMutateur” à la racine du répertoire, avec le chemin approprié.
- Le pom.xml du projet à muter est rétabli, puis copié vers le projet mutant.

- Un fichier HTML est préparé pour accueillir les résultats des tests.

Ce fichier contiendra un minimum d’informations sur les résultats de tests et pointera vers une page détaillée. Il donne notamment les pourcentages de mutation choisi pour chaque mutateur.

- Les tests sont exécutés pour chaque projet muté.

Pour chaque dossier mutant :

- La commande mvn test est exécutée.
- Les fichiers produits dans le dossier target/surefire-reports sont exploités pour enrichir la page HTML. On récupère le nombre de tests exécutés, le nombre de tests en erreur/en failure et le nombre de tests ignorés.
- Le pourcentage de réussite des tests est calculé et inséré dans la page HTML.
- La page HTML de test surefire est générée et un lien vers elle est fait dans la page située à la racine.

II. Prise de recul

A. Utilité des mutateurs

L’objectif des tests unitaires est de valider les différents usages de chaque fonction du code : chaque test décrit un comportement attendu du programme. Lorsque le code est muté, il est attendu que ce comportement soit modifié, et que les bons tests unitaires qui passaient avant la mutation ne passent plus.
On en déduit que l’intérêt premier des mutateurs est de valider la bonne facture des tests d’un projet : au plus ils échouent après mutation, au mieux ils sont.

Ce qui peut être détecté grâce à la mutation :

- Les tests sans assertion (passeront quelque soit la mutation).
- Les séries de tests qui traversent un seul chemin d’une fonction (passeront tous ensemble ou échoueront tous ensemble si ce chemin est muté).

De là, on peut déduire l’intérêt de ne pas muter l’intégralité du code : si on le fait, on s’attend à ce que tous les tests échouent. Avec une fraction du code muté seulement, les tests qui passent par le même chemin échoueront tous en même temps, ce qui nous aide à les repérer.

B. Qu'est-ce qu'un bon mutateur ?

Compte tenu de ce qui a été dit précédemment, la notion de "bon mutateur" est dépendante du projet duquel on va évaluer les tests. Il s'agit de mettre en erreur les tests sur lesquels reposent la confiance en le programme :

- S'il s'agit d'un projet qui se repose beaucoup sur du calcul (par exemple le kata mis sur le dépôt : calcul du prix d'achat en tenant compte de réductions, …), changer les opérateurs de calcul devrait significativement impacter le comportement.
- S'il s'agit de décrire un comportement (comme dans island), changer les valeurs booléennes peut influer sur les décisions et changer le comportement.
- …

Si les mutations ne sont pas liées au projet, il est vraisemblable que la plupart des tests ne soient pas mis en échec et on ne pourra rien en déduire (car effectivement ils ne doivent pas être mis en échec).
Comme dit plus haut, il n'est pas indispensable d'effectuer la mutation sur 100% des candidats à cette mutation car on ne pourrait pas déterminer si un échec est dû à un unique morceau de code (mauvais) ou pas. L'idée est plutôt de muter par exemple 50% des candidats et d'effectuer plusieurs fois la mutation.

C. "Tuyauterie"

Comme on pouvait s'y attendre, la mise en oeuvre de toutes les étapes menant à la production finale de notre rapport de test, nécessite de "bricoler" plus que ce qui est raisonnable pour un projet. Nous avons fait le choix d'utiliser le scripting shell pour résoudre les problématiques du projet. Ce choix s'est avéré raisonnable pour toute la partie qui consiste à compiler les mutateurs à l'aide d'un appel Maven, de créer des répertoires et d'y générer les codes mutés. Mais pour les manipulations plus fines, nous nous sommes vite retrouvé à faire du "bricolage".

On peut notamment citer, l'enchainement de 3 sed successifs pour traduire le path de l'interpréteur Bash MSys de la forme '/C/foo/bar/kata/' vers un path windows 'C:\foo\bar\kata'. Bien que ça marche, c'est la preuve que notre choix technique n'est pas parfait.

Ou encore la génération de notre fichier de rapport html obtenu en écrivant dans des fichiers temporaires a partir de l'output de la commande précédente.

On peut se demander quelle serait effectivement la solution à mettre en oeuvre sur un projet d'une taille supérieure à celui-ci, qui permettrait de ne pas s'appuyer sur du scripting.

III. Conclusion

Nous avons pu mesurer l'intérêt des mutations de code dans le cadre de la qualité de nos programmes. Tout comme la couverture de tests assure que nous traversons tous les chemins possibles, les mutations nous permettent d'être sûr de la valeur des autres tests.

Ce concept appliqué nous permet d’avoir une plus grande confiance dans notre code.