Rapport Mutateurs

Rendu 1 - Groupe 2

Architecture

Architecture et Technologies utilisées

Afin de gérer facilement les dépendances vis à vis des bibliothèques nécessaires au projet, mais surtout afin d'automatiser certaines tâches, nous avons utilisé maven.
Cela nous aura permis de lier plusieurs projets maven qui sont séparé (en les identifiants comme modules): nous avons d'un côté le code source initial de Island, et d'un autre côté le code des mutants.
Voici ci-dessous la structure de notre projet :

Dossier racine

- **Island**
  - src (code originel du projet à muter)
  - mutation.EQToNEQProcessor (code muté par le processeur EQToNEQ)
    - fr (contenant les classes mutées)
    - report (contenant le rapport des passages des tests)
  - mutation.LGInverterProcessor (code muté par le processeur LGInverter)
    - fr (contenant les classes mutées)
    - report (contenant le rapport des passages des tests)
  - ...
  - pom.xml
- **Mutator**
  - src
    - mutation (contenant tout les mutateurs)
      - EQToNEQProcessor
      - …
  - pom.xml
- **Logs**
  - log[<nom_processeur>]
  - ...
- pom.xml
- README.md
- compile.sh
- report.sh
- report.txt (résultat de report.sh)

Concernant les mutants, nous utilisons Spoon, une librairie fourni par l'INRIA, qui nous permet une étape de transformation sur notre code.
Enfin, pour générer des rapports et pouvoir suivre l'impact des mutants sur nos tests, nous utilisons surefire-report.

Fonctionnement de nos scripts

Afin d'automatiser l'ensemble des tâches nécessaires, nous avons créé deux scripts à la racine du projet : compile.sh et report.sh.

Script compile.sh :

- Récupère le nom de tous les mutants créés dans le dossier /mutator/src/main/java/mutation
- Pour chaque mutant récupéré :
    - Modifie le pom.xml, en précisant dans les balises de configuration de spoon, le mutants à utiliser (balise **<processor>**)
    - Modifie le pom.xml, en précisant le répertoire de sortie (nom du mutant) du code source muté (balise **<outFolder>**)
    - Lance la commande **mvn surefire-report:report.** Cette commande s'occupera, de compiler le code source muté, de lancer les tests dessus, ainsi que de générer un rapport surefire propre à ce mutant.
- Lance le script report.sh (décrit plus bas)

Pour éviter de parcourir tous les dossiers de chaque mutant, de regarder les résultats, etc… nous avons choisi d'écrire un petit script qui s'occupera de parcourir tous ces rapports à notre place et d'en récupérer les éléments les plus intéressant (taux de réussite des tests global ainsi que le temps d'exécution global).

Script report.sh :

- Pour chaque rapport surefire des mutants :
    - on récupère le taux de réussite des tests
    - on récupère le temps d'exécution global des tests
    - on écrit ces résultat dans un fichier report.txt

Mutants utilisés

Nous avons créé en tout 4 mutants différents :

- EQToNEQProcessor : Change les == en != et inversement
- LGInverterProcessor : Change les < en >, <= en >=, >= en <= et > en < (hors boucle for et while)
- MinusToPlusProcessor : Change les + en -
- OperandInveterProcessor : Change l'ordre dans une expression (Exemple : x + y devient y + x).

Nous avons choisis d'utiliser ces mutateurs car ils concernent les opérations de tests les plus basiques. De plus, le code concerné par les tests est Island, où l'élément principal est une Carte. En modifiant les opérations entre entiers (+ en -, <, >, …) nous changeons radicalement le comportement du programme sur cette Carte. De ce fait, ces mutateurs font passer des tests dans le rouge .

Taux de réussite des mutateurs :

- EQToNEQProcessor : 75%
- LGInverterProcessor : 67%
- MinusToPlusProcessor : 85%
- OperandInverterProcessor : 55%

Analyse

Limite des tests et de l'analyse de couverture de code

Jusqu'à présent, afin de vérifier la fiabilité de nos tests, nous nous basons uniquement sur le taux de réussite des tests ainsi que le taux de couverture des tests.
Grâce à l'analyse de couverture, nous pouvons vérifier lors de l'exécution des tests précisément quelles lignes ou non ont été parcourues. Cet indicateur est intéressant, mais possède ses limites. En effet, il prend en compte les instructions qui ont été parcourues pendant l'exécution des tests, mais ne nous garantit pas que leur bon fonctionnement a été vérifié.
Prenons un exemple simple :

```
public class IntegerOp{
  public int sum(a, b) {
    return a;
  }
}
```

```
@Test
public void testSum(){
  int x = new IntegerOp().sum(8, 0);
  assertEquals(x, 8);
}
```

On remarque clairement que la fonction sum est fausse. Cependant, l'exécution des tests ici ne relève aucune anomalie et nous avons bien une couverture de 100%.

Ainsi, l'analyse de la couverture de code atteint vite ses limites lorsqu'il s'agit de mesurer la fiabilité des tests.

Utilité des mutants

Comme dit plus haut, une couverture de code de 100% n'est pas synonyme de fiabilité à 100%, elle indique seulement que l'ensemble du code possède des tests spécifiques à chaque méthode.

Les mutants vont servir à remettre en question l'utilité de nos tests unitaires, en les mettant à l'épreuve de changements d'opérateurs, de booléen et autres conditions. En effet, c'est après avoir généré ces mutations que l'on va vérifier la pertinence de nos tests unitaires. Il faut d'abord s'assurer que l'ensemble des tests passent sur le code source, les mutants sont ensuite censés les faire échouer. Cela atteste de la capacité du test à bloquer les changements de code.
Les mutants vont permettre de rajouter du sens aux tests.
En adoptant cette approche, la qualité et notre confiance envers notre code ne peuvent qu'être améliorées

Avantages / Inconvénients

L'utilisation d'un tel mécanisme apporte forcément des avantages mais aussi, comme toujours, quelques inconvénients.
Les avantages ont plus ou moins déjà été évoqués plus haut dans le rapport. En effet, le principal avantage est que cela apporte encore plus de fiabilité à nos tests. De plus, cela requiert assez peu de développement spécifique, puisque les mutants sont en règle générale des mutations simples (comme par exemple changer les == en !=).
Malgré un principe simple, sa mise en œuvre peut apporter quelques inconvénients/difficultés. En effet, trouver de vrais mutants pertinents reste compliqué. De plus, pour un projet qui commence à être imposant, le nombre de mutants nécessaire pour assurer une fiabilité sur l'ensemble des tests doit être important. Cela implique un temps d'exécution conséquent.

Comment écrire des mutants pertinents ?

Écrire des mutants pertinents semble être la tâche la plus dure et fastidieuse lorsqu'on souhaite utiliser cette technique. En effet, il ne s'agit pas de modifier des opérateurs ou des bouts de code de façon aléatoire/générale mais il s'agit de cibler des zones potentiellement critiques dans notre code et d'y modifier légèrement le comportement. Cette tâche s'avère d'autant plus complexe puisque pour pouvoir exploiter les résultats des mutants, il faut modifier le code tout en s'assurant qu'il puisse toujours compiler par la suite.

Une première solution pour écrire des mutants pertinents est de parcourir toutes les instructions du code métier, et pour chacune d'elle, déterminer si des mutations sont applicables ou non. Cela amène à un des désavantages relevés plus haut : le nombre de mutants devient conséquent pour un gros projet.
Ainsi, comme deuxième solution pour écrire des mutants pertinents, il faut cibler nos mutants sur des zones spécifiques de notre code. En effet, par exemple, la modification d'opérateur mathématiques devrait être privilégiée sur des classes effectuant des calculs.