


Rapport Projet Mutant Team H

Florian Bourniquel
Alexandre Clément
Quentin Duret
Florian Lehmann

Arborescence et langage

Pour installer le projet, veuillez suivre les étapes du Readme.md contenu dans le dépôt oldtcf. L'arborescence obtenue devrait ressembler à celle-ci:



```

C:\Windows\System32\cmd.exe

C:\Users\user\Desktop\isa-devops-H>tree
Structure du dossier
Le numéro de série du volume est 5A0C-E140
C:.
├── mutation
│   └── src.main.java
└── oldtcf
    ├── doc
    └── src.main.java

C:\Users\user\Desktop\isa-devops-H>
  
```

Pour le langage, nous avons décidé d'utiliser Python3.6 pour son côté pratique et sa simplicité d'utilisation.

Compilation et application des mutateurs

Pour pouvoir compiler de manière automatique nos mutateurs, nous utilisons un projet maven. Cela nous permet également de pouvoir facilement les importer pour pouvoir les appliquer au projet Cookie Factory.

Pour chaque mutateur, nous créons un dossier temporaire dans le dossier "mutants".

On copie puis on renomme le fichier "mutant_pom.xml" dans le dossier du mutant pour pouvoir l'exécuter en tant que projet maven.

On renomme l'arborescence générée par spoon pour qu'elle soit conforme à un projet maven. Enfin, on copie les tests de Cookie Factory dans le dossier du mutant.

On termine en exécutant la commande `$ mvn test` dont on redirige la sortie vers un fichier correspondant au mutant dans le dossier "result" puis on supprime le dossier contenant tous les mutants.

Pour avoir un résultat lisible, on récupère les informations les plus importantes de l'exécution que l'on ajoute aux fichiers "result/resume.txt".

Processeurs écrit :

- **ProcessorIf**: Ce processeur affiche un "hello" à chaque fois qu'on rencontre un "if", cela nous a permis de prendre en main spoon.

- **ProcessorClass**: Ce processeur affiche le nom des classes du projet, cela nous a permis de prendre en main spoon.

- **ProcessorAndToOr:** Ce processeur remplace les “&&” par des “||”, cela permet de modifier le comportement des boucles “while” et des conditions comme “if” qui utilise “&&”.
- **ProcessorEqual:** Ce processeur remplace les “==” par des “!=”, cela permet de modifier les comportements des égalités.
- **ProcessorIncToDec:** Ce processeur remplace les “++” par des “--”, cela change le comportement des incrémentations mais principalement le comportement des boucles for.
- **ProcessorMoreToLessThan:** Ce processeur remplace les “>” par des “<”, cela permet de changer le comportement des boucles et des conditions.
- **ProcessorLessToMoreThanPackage:** Ce processeur remplace les “<” par des “>” dans le package “customer”, cela permet de changer le comportement des boucles et des conditions seulement dans le package customer.
- **ProcessorPositiveToNegativeValue:** Ce processeur remplace les valeurs positives par des valeurs négatives, cela permet de changer le comportement du calcul du prix à payer.
- **ProcessorRandomNullAssignment:** Ce processeur modifie 50% des affectations de type non primitif sur null. Les affectations à modifier sont choisis aléatoirement. Ce type d'affectation permet de vérifier que la validité des objets sont prises en compte.

Caractéristique d'un bon mutateur ?

Dans un premier temps, nous allons définir le but d'un mutateur. Un mutateur a pour objectif d'être appliqué sur une partie ou l'ensemble du code afin d'en modifier le comportement. Puisque le comportement du code a été modifié lors de l'exécution des tests, certains d'entre eux ne doivent plus fonctionner. Ce qui nous indiquera que les méthodes sont bien testées, dans le cas contraire il faut écrire des tests supplémentaires pour couvrir les comportements modifiés par le mutateur.

Plusieurs critères rentrent en oeuvre dans la création d'un bon mutateur:

- **Granularité:** Il faut trouver le bon périmètre sur lequel appliquer le mutateur. Si la granularité est trop importante par exemple sur l'ensemble du code beaucoup de tests vont échouer mais nous n'aurons aucune indication sur quelles méthodes sont ou ne sont pas testées. A l'inverse faire un mutateur pour une seule classe, voir une seule méthode, nous indique clairement si la méthode ou les méthodes de la classe sont testées. Cependant cela nous oblige à réaliser un mutateur pour chaque méthode ou classe ce qui à terme dans de gros projets n'est pas viable. On préférera donc utiliser cette granularité fine uniquement sur les méthodes ou classes vraiment critiques. De façon générale une bonne granularité est au niveau du package puisqu'elle permet d'appliquer la mutation sur un ensemble de classes qui sont censées avoir plus ou moins le même comportement.

- **Intelligence:** Un bon mutateur doit aussi modifier le comportement du code qu'il mute et non pas juste changer quelques variables ou conditions. Si un mutant ne modifie pas le comportement du code ce n'est pas un bon mutant.
- **Ciblé:** Un mutant appliqué par exemple à une classe ne doit pas faire échouer tous les tests de cette classe puisque dans ce cas nous n'aurons aucune indication sur quel ou quel comportement est testé ou non. Un mutant doit se limiter à un scope de comportement réduit. Par exemple dans une classe qui calcule les réductions à appliquer au panier, nous ne devons pas en même temps modifier le calcul des réductions et quand ces réductions sont appliquées.

Problèmes rencontrés et solution?

Le statique est le premier problème que nous avons rencontré, en effet lorsque nous avons une variable static dans une classe, spoon nous ajoutait un import vers la classe dans la liste des imports ce qui provoquait des erreurs de compilation. Une solution simple a donc été de retirer les variables statiques des classes et de créer une classe où nous avons mis toutes les variables statiques.

Ensuite nous avons souhaité pouvoir générer le code muté en dehors d'une phase maven (test, clean, validate,...) et dans un dossier défini par l'utilisateur. Pour résoudre ce problème nous avons défini une propriété dir. Cette propriété a pour but de définir le dossier dans lequel le mutant va être généré. Le dossier par défaut est le dossier target mais il peut être défini lors de l'exécution de la commande maven: `$ mvn spoon:generate -Ddir="Test"`.

Au départ, il n'y avait qu'un seul processeur. Pour permettre l'exécution de plusieurs processeurs nous avons ajouté une autre propriété nommée processor. Cette propriété est définie par l'intermédiaire de la commande maven suivante : `$ mvn spoon:generate -Dprocessor="processors.ProcessorsName"`. Cette commande est valide uniquement si le processeur a été installé au préalable à l'aide de la commande: `$ mvn install`. L'écriture d'un script python nous a permis d'automatiser tout cela et nous permet d'exécuter tous les mutateurs avec la commande `python mutants.py`.

Pour obtenir un rapport lisible de l'exécution de chacun des mutateurs, il nous a fallu analyser les rapports d'exécution de maven pour en extraire les informations les plus pertinentes. Pour cela, nous récupérons et nous listons dans un même fichier le nombre de test qui sont exécutés, qui échouent, qui engendrent une erreur à l'exécution ou qui sont ignorés car un test dont ils dépendent a échoué.

Nous avons également rencontré un problème de portabilité entre plate-forme. En effet, la gestion des appels système en Python est différente entre Windows et les systèmes Unix. Nous avons donc dû adapter les commandes permettant les appels maven en fonction de la plate-forme.

Qu'avez vous appris ?

Lorsque nous livrons un projet nous devons être sûr qu'il fonctionne correctement. Même si dans la réalité il est impossible de créer de gros projets dépourvus de bugs, il faut par le biais de tests, de la documentation, ect en identifier le plus d'erreurs possibles afin de les corriger. Cette couverture doit être assurée par plusieurs types de tests différents (unitaires, intégrations...), de plus ce système de tests doit être le plus automatique possible.

Il existe de très nombreux types de tests différents, beaucoup plus que ce que nous avons étudié pour le moment. Chacun de ces types a ces caractéristiques et objectifs propres, mais tous important dans leurs domaines d'applications. Nos tests doivent être intelligents et tester le comportement de nos fonctions. Un projet ayant 80% de couverture de tests mais qui teste uniquement ses getter et setter, est en réalité un projet non testé. Pour résumer, il existe un grand nombre de tests différents ayant chacun leurs importances. Ces tests avec toutes les méthodes misent en place pour identifier, corriger et documenter les bugs, permettent d'assurer une bonne livraison d'un produit et d'obtenir la confiance du client.