Validation & Vérification

Master 2 Ingénierie Logicielle - 2017/2018

Sujet 3 : Mutation Testing

Etudiants
June Benvegnu-Sallou, Antoine Ferey, Emmanuel Loisance

Enseignants
Nicolas Harrand, Oscar Vera-Perez, Benoit Baudry

Introduction

Ce projet s'inscrit dans le module V&V du master 2 Ingénierie Logicielle de l'université de Rennes 1 et de l'Istic. Le but est de pouvoir s'assurer qu'une suite de tests d'un programme cible soit fonctionnelle, c'est-à-dire qu'elle permette de détecter les bugs éventuels.

Pour cela, nous avons recours au principe du mutation testing. Globalement, il s'agit de générer des bugs à certains endroits du code (chaque changement de code = un mutant) et de lancer la suite de tests du projet cible pour constater si cette dernière détecte le changement de code ou non. Si les tests du projet cible échouent, alors le mutant a été "tué" et la situation correspond à un comportement attendu. Plus il y a des mutants tués, plus les tests du projet cible sont pertinents. Nous avons implémenté ce principe dans notre projet.

Solution

Description

Notre projet se découpe en plusieurs étapes : chargement du projet cible, création des mutants, lancement des tests, évaluation des résultats de tests et génération d'un rapport.

La première étape consiste à compiler le projet cible. Le projet cible doit néanmoins remplir quelques conditions :

- être un projet Maven
- pouvoir compiler
- ses tests doivent passer

Ensuite on récupère la liste des méthodes du projet cible. En parcourant ces méthodes, une liste de mutants va être produire en fonction des opérateurs de mutation (les différents types de changement du code, ie. mutators) choisis.

Maintenant, pour chaque mutant créé précédemment, nous allons :

1. Modifier le code de la méthode
2. Enregistrer cette modification
3. Lancer les tests du projet cible
4. Si il y a des erreurs (ie. les tests échouent), le mutant est "killed"
5. On revient à la méthode initiale, avant la modification
6. On passe au mutant suivant

Pour finir, on génère un rapport qui contient toutes les informations intéressantes : le pourcentage de mutant tuée et quel test a tué quel mutant.

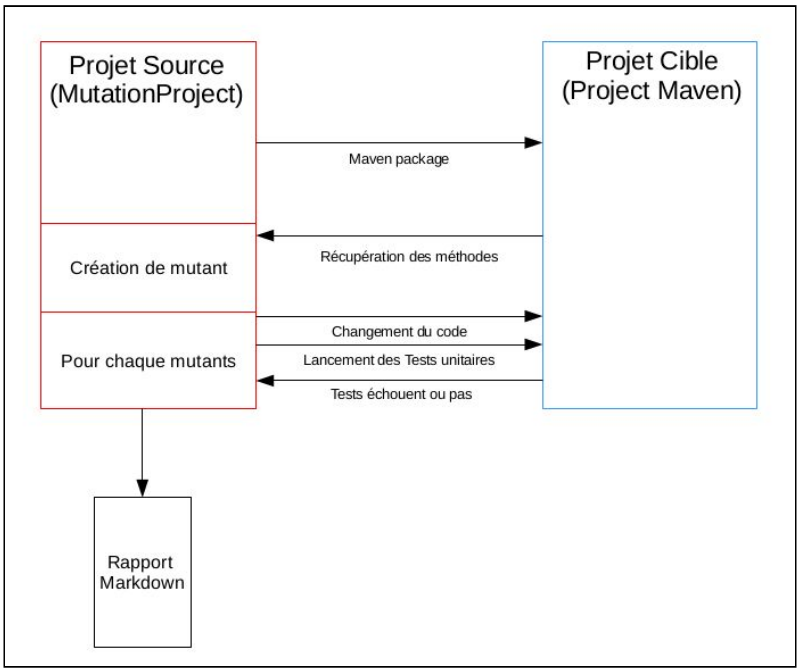

Figure 1 - Schéma du fonctionnement global de notre programme

De façon plus détaillée, nous avons conçu une architecture permettant de réaliser tous ces processus. Le diagramme de classes correspondant est en annexe (Annexe 3).

ProjectTarget : Il représente le projet cible. C'est avec cette classe que l'on peut le compiler ou lancer ses tests unitaires.

Scanner : C'est l'objet qui va scanner toutes les méthodes du projet cible. Il va renvoyer une liste de mutants créés à partir de plusieurs Mutators (cf paragraphe suivant).

Mutator et ses classes héritées : Chaque mutator contient le code pour créer une liste de mutants à partir d'une méthode, faire la mutation et le retour à la méthode initiale sans mutation. Comme chaque mutator est différent, il faut définir l'algorithme pour ces 3 opérations (createListMutant, doMutate, revert).

Notre projet comporte 4 mutators :

- remplacer le corps des méthodes boolean par return true
- supprimer le corps des méthodes void
- remplacer les opérations arithmétiques

Original	Remplacé par
+	-
-	+
*	/
/	*

- remplacer les opérations de comparaison

Original	Remplacé par
>	>=
>=	>
<	<=
<=	<

Mutant : Créé par les mutators. Un mutant comporte plusieurs informations :
- La méthode que l'on va modifier.
- Le mutator qui l'a créé. (Cela va servir lors du changement de code et du retour à la méthode initiale).
- L'index du bytecode à changer ou une copie de la méthode que l'on va modifier.

Executor : Exécute l'algorithme principal et génère le rapport.

Technologies employées

Nous préférions manipuler du code compilé que du code source au niveau du projet source. Ainsi, nous avons choisi d'utiliser Javassist (version 3.12.1) plutôt que Spoon. Par ailleurs, nous pensions trouver davantage de documentation avec Javassist que Spoon. En effet, après avoir réalisé le tutoriel en cours/td sur Spoon et quelques recherches, nous sommes parvenus à la conclusion que peu de documentation sur cet outil était disponible. Cet élément a également eu un impact dans notre choix.

Pour le développement du projet en équipe nous avons utilisé différents outils pour la gestion de projet. L'utilisation de Git et de son système de branche pour pouvoir travailler à trois en même temps sur le projet et parfois sur la même classe via des fonctionnalités différentes ainsi que le système de release / tag pour les différentes livraisons attendues.

La création d'issues sur Github a permis de répertorier les erreurs connues sur le projet et les évolutions à apporter. L'utilisation du Kanban nous a donné une vue synthétique et globale de l'état du projet.

Nous avons également mis en place l'outil d'intégration continue Travis sur notre projet. Travis nous permettait de détecter rapidement un problème sur notre code en vérifiant après chaque commit ou merge que le projet se construisait toujours correctement.

Usage

Nous mettons à disposition un fichier executable (jar) qui prend deux paramètres :
- le chemin d'accès (path) du dossier du projet cible
- le chemin d'accès du rapport qui va être généré par notre programme.

Exemple:

```
aferey@aferey-istic-ubun:~/Documents/VV/MutationTestingProject$ java -jar MutationTestingProject.jar
"/home/aferey/Documents/VV/MockMathSoftware" "/home/aferey/Documents/VV/"
[INFO] - Target project is building ...
[INFO] - Target project is scanning ...
[INFO] - Execute All mutants
Mutant{Class=fr.istic.vv.afe.MathAfe, Method=ladd, index=6, Mutator=ArithMutator}
Result : not killed
1 on 39
kill : 0
0.0% killed
Mutant{Class=fr.istic.vv.elo.SampleELO, Method=isGreaterOrEqual, index=2, Mutator=ComparisonMutator}
Result : killed
2 on 39
kill : 1
50.0% killed
Mutant{Class=fr.istic.vv.afe.MathAfe, Method=fdiv, index=2, Mutator=ArithMutator}
Result : not killed
3 on 39
kill : 1
33.333336% killed
Mutant{Class=fr.istic.vv.afe.MathAfe, Method=dmul, index=2, Mutator=ArithMutator}
```

Figure 2 - exemple d'utilisation du programme

Evaluation

Dans un premier temps, nous avons créé un projet minimaliste "MockMathSoftware" avec une structure et des méthodes simples pour tous les opérateurs de mutation choisis et 8 tests. Cette version minimaliste nous a permis de pouvoir rapidement tester nos mutants sans se préoccuper de la complexité des différents cas à gérer.

Nous avons vérifié que notre projet source réalisait bien ce que nous voulions, c'est-à-dire répondre à la problématique en générant des mutants et évaluant la pertinence de la suite de tests du projet cible. Des tests ont été produits dans cette optique. Pour faire cela, nous avons simulé des classes dénommées "Mock" au sein de notre projet source. Les classes ont été insérées dans notre projet pour que nous puissions lancer nos tests sur ces classes sans être dépendant d'une autre structure ou projet. Ainsi, nous pouvons lancer nos tests à partir de notre projet uniquement.

Nous avons ensuite utilisé les projets "commons-math" et "commons-cli" en tant que projets cibles de plus grande taille. En utilisant le projet "commons-math" en tant que projet cible, nous sommes notamment tombés sur des cas auxquels nous n'avions pas pensés / rencontrés avec notre projet cible minimaliste :

- Les opérations (~ méthodes sans "body") contenues au sein d'interfaces étaient considérées comme des méthodes. Cela posait problème lorsque que notre programme essayait de récupérer les informations et le bytecode de ces méthodes.

- Prise en compte seulement des fichiers ".class" du projet pour récupérer les méthodes.

Pour évaluer notre programme, nous avons également cherché à estimer la qualité de la suite de tests et la couverture du code grâce à l'utilisation de l'outil Cobertura.

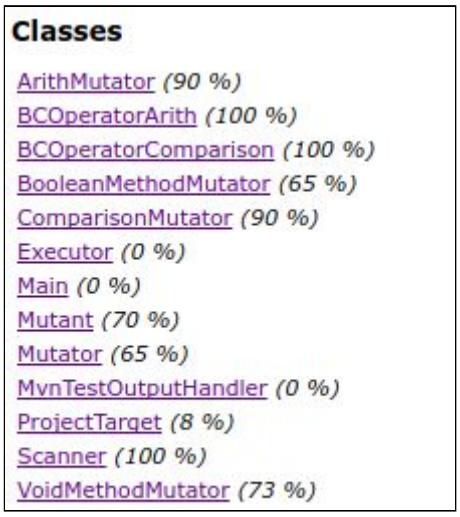

Figure 3 - Couverture de toutes les classes mesurée par cobertura

Discussion

Nous avons rencontré plusieurs problèmes tout au long du développement du projet, pour les résoudre nous avons dû faire des choix. En fonction des cas, nous avons dû opter pour une modification de notre façon de penser et adapter l'architecture globale du projet ou réaliser des changements d'outils qui ne correspondaient pas à nos besoins.

Nous avons remarqué que notre code était mal structuré et mal divisé. Nous avons décidé de séparer les tâches de création des mutants (classe Scanner), des changements de code (classe Mutator) et le lancement des tests du projet cible sur ces mutants (classe Executor).

JUnitCore n'était finalement pas la meilleure solution pour charger un projet cible complexe et lancer les tests de ce même projet, nous l'avons donc remplacé par Maven Invoker.

De plus, certains comportements de Javassist étaient inattendus. En effet, la manipulation du byte code par Javassist n'était pas celle que l'on aurait pu imaginer. Nous avons d'ailleurs ouvert une issue sur le projet Javassist pour avoir plus d'informations sur le problème rencontré : https://github.com/jboss-javassist/javassist/issues/173.

Nous avons détecté ce bug important lors de l'exécution de nos mutants sur des projets conséquents comme commons-math. Cela concerne uniquement les mutators qui touchent directement le bytecode (Comparaison et arithmétique). Le bytecode entre le moment du scan réalisé par l'objet Scanner (pour récupérer les index) et celui de la mutation avait changé.

Au moment du retour à la méthode initiale (sans la mutation apportée), après le lancement des tests, la méthode est remplacée par une copie faite avant la mutation. On se retrouve donc avec le code initial sans mutation. Le problème ici est que la copie n'est pas conforme.

CtNewMethod.copy ne renvoie pas le même bytecode. Il y a exactement le même problème avec le setCodeAttribute et getCodeAttribute de MethodInfo.

On peut observer la sortie d'une méthode avant copie *(voir annexe 1)* et la sortie après avoir remplacé par la copie *(voir annexe 2)*. On constate une différence à l'instruction 71. On se retrouve avec un décalage d'index. Nos mutants après 71 ne comporte plus le bon index.

Pour résoudre le problème, nous n'appliquons pas la copie lors du retour à la méthode initiale. Nous appliquons une deuxième fois le mutant et nous nous retrouvons avec le bytecode inchangé. Cela est uniquement valable pour les mutators qui touchent au bytecode. Les mutators Void et Boolean gardent l'ancienne stratégie de remplacement (utilisation de la copie).

Les fonctionnalités non terminées / implémentées :

- Un plugin Maven permettant ainsi d'ajouter directement dans le projet à tester (projet cible) le nom du plugin de notre projet dans le "*pom.xml*". Une fois le plugin en place sur le projet, une commande permettrait de lancer la génération des mutants directement sur le projet et de générer le rapport. Grâce au paramètre qu'il est possible de renseigner dans le pom.xml pour un plugin, le développeur pourra choisir la destination pour le rapport. Pour pouvoir utiliser un paramètre dans le plugin, il faut utiliser l'annotation @Parameter dans la classe Mojo.

- Test de la classe ProjectTarget : Pour tester cette classe, il nous faut créer un projet test au sein même de notre projet source. Nous avons pensé mettre ce projet Mock au sein d'un dossier "*src.test.java.com.istic.tp.mockproject*".

- Mutation coverage : Comme à la manière de Pit, nous voulions savoir si le programme passait dans nos mutations. On aurait eu un pourcentage de mutant "testé".

La dernière fonctionnalité n'a pas été implémentée au sein de notre projet mais nous avons cherché les éléments de réponses pour le faire. Nous énonçons ces indications ci-dessous.

Problème :

On ne peut pas juste mettre un print après (ou avant) la ligne ou le mutant est exécuté.

```
System.out.println("Je passe dans le mutant");
if ( x || y < z) {
    System.out.println("Je passe dans le mutant");
}
```

Ici, si x est égale à true, jamais on passera dans y < z. Malgré cela on verra bien les print.

Première solution :

Insérer directement dans le Bytecode un appel à une fonction qui fait un *System.out.println*.
Fonction print :

```
public void printMutateCoverage();
    Code:
       0: getstatic     #83 // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #85 // String nameClass:nameMethod:index:ArithMutator
       5: invokevirtual #88 // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
```

Appel à la fonction sur une méthode comprenant un mutant (index 2 : iadd -> isub)

```
 public int Add(int, int);
    Code:
       0: iload_1
       1: iload_2
       2: isub
       3: aload_0
       4: invokevirtual #90 // Method printMutateCoverage:()V
       7: ireturn
```

On peut voir qu'à l'index 4 on fait un appel de la fonction générée *printMutateCoverage* juste après le mutant isub.

Problème de la première solution :

Dans notre exemple, l'appel à la fonction à l'index 4 modifie la pile. Le *ireturn* retourne ce que renvoie l'appel de fonction *printMutateCoverage* et non plus le résultat de *isub*.

Deuxième solution :

Pour résoudre le problème de la pile dans le cas d'un mutant comparaison, on peut remplacer *valeur operator valeur* par un appel à une fonction renvoyant un *boolean*. Dans la méthode générée, on aura un print et un return de *valeur operator valeur*.

Exemple :

```
...
if( x || printMutateCoverage(y,z)){
    return i + j;
}

public boolean printMutateCoverage(int y, int z){
```

```
    System.out.println("nameClass:nameMethod:index:ComparisonMutator");
    return y < z;
}
```

Il faudrait prévoir tout les cas des paramètres entrants de la fonction *printMutateCoverage* (int, Integer, float, ...).

Voici le même raisonnement dans le cas des opérateurs arithmétique :

```
if( x || y < z ) {
    return printMutateCoverage(i , j);
}
```

```
public int printMutateCoverage(int i, int j) {
    System.out.println("nameClass:nameMethod:index:ComparisonMutator");
    return i - j;
}
```

Dans le cas des deux autres mutators (Boolean et Void), un simple print à l'entrée de la fonction modifié aurait suffit pour vérifier le mutant.

Conclusion

La mise en pratique du Mutation Testing nous a permis de constater que bien que le concept soit relativement simple, l'implémenter ne l'est pas autant. En effet, il est nécessaire de manipuler des projets cibles dont on ne connait pas les spécificités à l'avance. Il faut donc prévoir les différents situations et produire un programme générique mais tout de même complet. Nous avons pu également subir les désagréments d'un passage d'un projet cible minimaliste et de petite taille à un projet réel et complexe.

Notre projet source estime la fiabilité et pertinence de la suite de tests d'un projet cible en évaluant si cette dernière repère des mutants. Et pour faire cela, nous utilisons nous-même une suite de tests. Cependant, peu d'éléments (couverture de code par Cobertura) nous assure que cette suite de tests est elle-même pertinente. Il faudrait à nouveau réaliser une phase de mutation testing mais cette fois-ci avec notre projet source comme étant le projet cible.

Annexes

Annexe 1 : Code d'une méthode avant la méthode *copy*

```
0: iload_3
1: i2d
2: invokestatic #53 = Method org.apache.commons.math4.util.FastMath.sqrt((D)D)
5: dstore 4
7: dload_1
8: dload 4
10: dmul
11: dstore 6
13: dload_1
14: dload_1
15: dmul
16: iload_3
17: i2d
18: dmul
19: dstore 8
21: dload 8
23: dload 8
25: dmul
26: dstore 10
28: dload 10
30: dload 8
32: dmul
33: dstore 12
35: dload 10
37: dload 10
39: dmul
40: dstore 14
42: dconst_0
43: dstore 16
45: dconst_0
46: dstore 18
48: dconst_0
49: dstore 20
51: dconst_0
52: dstore 22
54: ldc2_w #55 = int 9.869604401089358
57: ldc2_w #57 = int 8.0
60: dload 8
62: dmul
63: ddiv
64: dstore 24
66: iconst_1
67: istore 26
69: iload 26
71: ldc #59 = int 100000
73: if_icmpge 127
76: iconst_2
77: iload 26
79: imul
```

Annexe 2 : Code d'une méthode après la méthode *copy*

```
0: iload_3
1: i2d
2: invokestatic  #306                // Method
org/apache/commons/math4/util/FastMath.sqrt:(D)D
5: dstore        4
7: dload_1
8: dload         4
10: dmul
11: dstore        6
13: dload_1
14: dload_1
15: dmul
16: iload_3
17: i2d
18: dmul
19: dstore        8
21: dload         8
23: dload         8
25: dmul
26: dstore        10
28: dload         10
30: dload         8
32: dmul
33: dstore        12
35: dload         10
37: dload         10
39: dmul
40: dstore        14
42: dconst_0
43: dstore        16
45: dconst_0
46: dstore        18
48: dconst_0
49: dstore        20
51: dconst_0
52: dstore        22
54: ldc2_w        #307               // double 9.869604401089358d
57: ldc2_w        #309               // double 8.0d
60: dload         8
62: dmul
63: ddiv
64: dstore        24
66: iconst_1
67: istore        26
69: iload         26
71: ldc_w         #311               // int 100000
74: if_icmpge     128
77: iconst_2
78: iload         26
80: imul
```

Annexe 3 : Diagramme de classe