Ajout d'opérateurs de mutation orientés-objet à Pitest

Dieulin MAMBOUANA[1] and Romain SOMMERARD[2]

Université de Lille 1

1 Introduction

La technique de mutation appliquée à un projet permet de mesurer la qualité et la pertinence des suites de test. Dans le cadre du module OPL (Outils pour la programmation des Logiciels), nous avons réalisé un projet qui porte sur l'ingénierie des suites de tests. Nos travaux portent sur l'ajout d'opérateurs de mutation orientés-objet au framework de mutation Pitest [1].

Pitest est un système de mutation pour Java qui regroupe plusieurs types de mutation de base. Il inclut un certain nombre d'opérateurs de mutation comme les mutateurs arithmétiques, qui modifient les signes tels que l'addition, la multiplication et soustraction dans les instructions de calcul, ou encore les mutations permettant de supprimer les blocs switch et if-else dans les programmes. D'autres mutations plus complexes sont également disponibles comme les mutations de try/catch.

Toutefois Pitest ne fournit pas d'opérateurs de mutation orienté-objet. C'est pour cela que nous avons décidé d'implémenter et de voir si ce genre de mutations sont pertinantes. Ils existent plusieurs types d'opérateurs de mutations orienté-objet. En fonction de leur domaine d'application elles portent sur :

- Le polymorphisme
- Le surcharge de méthode
- La redéfinition de méthode/masquage dans l'héritage
- Le masquage de champ de variable en héritage
- Les états d'objets statiques/dynamiques

Pour notre projet, nous avons implémenté deux opérateurs de mutations qui touchent aux appels de méthode. Le premier, AOC (Argument Order Change), change l'ordre des arguments dans les appels de méthodes et le deuxième, AND (Argument Number Decrease), réduit progressivement le nombre d'arguments s'il y en a plus d'un.

Ces deux opérateurs implémentés ont été appliqués à deux projets Java open source.

1. http ://pitest.org/

2 Travail technique

2.1 But

L'objectif principal de notre projet est d'intégrer de nouveaux opérateurs de mutation objet à Pitest [2]. Notre scope se limite à deux opérateurs de mutation orienté-objet liés au concept d'appel de méthode : la mutation Argument Number Decrease et la mutation Argument Order Change.

2.2 Overview

En programmation orienté-objet, et notamment dans le langage Java, une classe peut avoir une ou plusieurs méthodes qui portent le même nom à condition que celles-ci aient des signatures différentes. Ce mécanisme est appelé surcharge de méthodes. Ce comportement intéressant et très utile peut très vite devenir source d'erreurs lors des appels de méthode avec parfois des oublis de paramètres par exemple. On peut être amené à appeler une méthode à la place d'une autre.

Plusieurs mutations touchent aux fonctionnalités de surcharge de méthodes :

1. AND (Argument Number Decrease) ;
2. AOC (Argument Order Change) ;
3. POC (Parameter Order Change) ;
4. VMR (oVerloading Method declaration Removal).

Pour notre projet nous avons choisi d'intégrer les opérateurs AOC et AND à Pitest.

Pitest est un framework Java de mutation particulier dans le sens où il applique les mutations au Runtime. Cela veut dire que les modifications sont apportées au niveau du bytecode. Les mutants utilisent la librairie ASM Java pour faire les transformations.

2.3 Architecture

L'architecture de Pitest est complexe et contient plusieurs packages qui implémentent diverses fonctionnalités de l'application. Parmi lesquelles on retrouve le package *org.pitest.mutationtest.engine.gregor.mutators* qui contient l'ensemble des mutations. C'est dans ce package où nous avons placé les deux classes Java *ArgumentDecreaseMutator* et *ArgumentOrderChangeMutator* qui implémentent respectivement les opérateurs de mutations AND et AOC.

2.4 Implémentation

Pour atteindre notre objectif, nous avons ajouté deux programmes Java dans le package *org.pitest.mutationtest.engine.gregor.mutators* :

1. *ArgumentDecreaseMutator.java* ;

2. http ://citeseerx.ist.psu.edu/viewdoc/download ?doi=10.1.1.31.7963&rep=rep1&type=pdf

2. *ArgumentOrderChangeMutator.java.*

Le premier met en oeuvre l'opérateur de mutation orienté-objet *AND* (Argument Number Decrease). Pour ce faire elle définit une énumération *ArgumentDecreaseMutator* qui implémente les méthodes *getGloballyUniqueId()* et *getName()* de l'interface *MethodMutatorFactory* du package *org.pitest.mutationtest.engine.gregor* afin de définir son identité dans le système de mutation, notamment son nom *ARGUMENT_NUMBER_DECREASE_MUTATOR*.

Le programme requiert également la classe *ArgumentDecreaseVisitor* qui étend la classe abstraite *MethodVisitor* du package *package org.objectweb.asm* de la librairie ASM. Elle décrit le comportement du mutant lorsqu'une classe est visitée. Pour ce faire notre classe redéfinit la méthode *visitMethodInsn* qui se déclenche lorqu'une instruction est rencontrée dans la méthode visitée.

L'algorithme se déroule de la manière suivante :

- Vérification du nombre d'arguments de la méthode (¿ 1)
- Suppression du dernier argument de l'appel de méthode
- Enlèvement de l'argument stocké sur la pile d'exécution
- Création du nouveau descripteur de la méthode sans le type du dernier arguments

Le deuxième programme *ArgumentOrderChangeMutator.java* met en oeuvre l'opérateur de mutation orientée-objet *AOC* (Overloading Method declaration Removal).

Celui-ci met en oeuvre la même technique que le mutant AND hormis le fait qu'il échange les deux derniers arguments au lieu de les supprimer comme avant.

2.5 Utilisation

Le projet se lance à l'aide de scipts. Nous avons un script par projet. Ces scripts se contentent de build pitest avec Maven et de l'installer localement afin que le projet testé ait la version de Pitest modifiée contenant nos mutants.

Ensuite, la commande qui lance Pitest-Maven est exécutée et le processus de test se lance. Pitest nous fournit un jeu de résultats en sortie qui permettent de voir précisément les portions de code couvertes par les mutants et leur état à la fin du run.

Les resultats des lancements sont disponibles dans le dossier results/ du projet.

3 Evaluation

Pour évaluer les mutations implémentées, nous avons choisi 2 projets open source : Spark Java [3] et Joda Time [4].

Spark Java est un petit projet qui contient 76 tests. Joda Time, quant à lui, est un projet plus conséquent qui contient 4172 tests. Sur ces projets que

3. http ://sparkjava.com/
4. http ://www.joda.org/joda-time/

nous avons lancé Pitest avec les mutations. Chaque mutation (AND et AOC) est lancée indépendamment.

3.1 AND

Lors du lancement de Pitest avec une mutation, celui-ci ne va lancer que les tests qui sont concernés par la mutation. Les résultats du mutant ANC sont présentés dans le Tableau 1 ci-dessous.

Project	General	Details						
	Generated	Killed	Killed	Survived	Timed out	Non viable	Run error	No coverage
Spark Java	399	186	136	2	44	0	6	211
Joda Time	2922	2567	2438	9	0	129	0	346

Table 1. Détails de la transformation AND par projet.

Au total, ce ne sont pas moins de 2801 tests qui sont lancés sur Joda Time et 115 sur Spark Java. Pitest génère 399 mutants pour le projet Spark Java. 186 sont tués par les tests, ce qui représente 47% des mutants générés. Ceci nous montre que les tests du projet ne sont pas tellement efficaces face à notre mutant.

Pour le projet Joda, nous avons 2922 mutants générés avec 2567 tués. Ceci représente 88% de tués. On en conclu que le projet Joda contient des tests pertinents et efficaces.

Le tableau nous montre que des mutants survivent aux tests. Le nombre de mutants qui survivent reste cependant bas. Cependant, le nombre de mutant de type NO_COVERAGE est très élevé, il représente 52% des mutants pour le projet Spark Java et 22% pour le projet Joda Time. Les mutants qui termine avec l'état NO_COVERAGE nous disent que le projet ne teste pas du tout les cas de notre mutation. Ce qui représente des portes ouvertes pour les bugs.

La Figure 1 présente un exemple de code qui survit dans le projet Spark Java.

```
 @Override
 public File getFile() throws IOException {
     URL url = getURL();
-    return ResourceUtils.getFile(url, getDescription());
+    return ResourceUtils.getFile(url); // SURVIVED
 }
```

Figure 1. spark.resource.AbstractFileResolvingResource.java

Un autre exemple Figure 2, cette fois sur le projet Joda Time.

```
 public long roundCeiling(long instant) {
     if (iTimeField) {
         int offset = getOffsetToAdd(instant);
         instant = iField.roundCeiling(instant + offset);
         return instant - offset;
     } else {
         long localInstant = iZone.convertUTCToLocal(instant);
         localInstant = iField.roundCeiling(localInstant);
-        return iZone.convertLocalToUTC(localInstant, false, instant);
+        return iZone.convertLocalToUTC(localInstant, false); // SURVIVED
     }
 }
```

Figure 2. org.joda.time.chrono.ZonedChronology.java

Les résultats de la mutation de type AND sont donc concluants car ils nous montrent les lacunes qu'a le petit projet Spark Java. Pour le projet Joda Time, bien que celui-ci soit bien plus solide au niveau des tests, notre mutation produit 9 mutants qui parviennent à survivre.

3.2 AOC

Les tests du mutant AOC ont été lancés dans les mêmes circonstances que la mutation ANC. Le Tableau 2 ci-dessous présente les résultats des tests.

Pour ce mutant, ce sont 5499 tests lancés sur Joda Time et 159 sur Spark Java. Nous avons 2922 mutants qui sont générés par Pitest pour le projet Spark Java avec 2538 tués. Comme précédemment avec le mutant ANC, 87% des mutants générés sont tués. Pour le projet Spark Java, ce sont les mêmes métriques que pour l'ANC avec 399 mutants générés et 186 tués, ce qui représente 47% de mutants tués.

Project	General		Details					
	Generated	Killed	Killed	Survived	Timed out	Non viable	Run error	No coverage
Spark Java	399	186	134	2	44	0	8	211
Joda Time	2922	2538	2409	38	0	129	0	346

Table 2. Détails de la transformation AOC par projet.

Les mêmes observations s'appliquent pour ce mutant. Celui-ci obtient de bon résultats.

Les mutants de type NO_COVERAGE sont toujours les mêmes puisque les mutations s'appliquent sur des appels de méthode. Il est donc normal d'avoir des résultats quasi similaires. On remarque cependant un nombre plus important

de mutants vivant pour le projet Joda Time. Ces résultats sont très concluant puisque l'on vient de mettre le doigt sur un possible manque de tests concernant les appels de méthode.

Deux nouveaux exemples de mutants type AOC pour le projet Spark Java (Figure 3) et Joda Time (Figure 4.

```
 private static void registerCommonClasses(Class<?>... commonClasses) {
     for (Class<?> clazz : commonClasses) {
-        commonClassCache.put(clazz.getName(), clazz);
+        commonClassCache.put(clazz, clazz.getName()); // SURVIVED
     }
 }
```

Figure 3. spark.utils.ClassUtils.java

```
 private synchronized String[] getNameSet(Locale locale, String id, String nameKey) {

     ...

     Map<String, Map<String, Object>> byIdCache = iByLocaleCache.get(locale);
     if (byIdCache == null) {
-        iByLocaleCache.put(locale, byIdCache = createCache());
+        iByLocaleCache.put(byIdCache = createCache(), locale); // SURVIVED
     }

     ...

 }
```

Figure 4. org.joda.time.tz.DefaultNameProvider.java

4 Conclusion

Le code de notre expérimentation est disponible sur Github [5].

Nous avons, grâce à cette étude, démontré que les opérateurs de mutation de type AOC et AND sont utiles. Ces mutations permettent bien de détecter de potentiels bugs, non triviaux, mais qui pourraient se produires.

En travaillant sur Pitest, qui effectue les modifications de code au runtime, nous nous sommes rendu compte de la complexité à mettre en place des mutations de type objet. Nous pensons qu'il serait plus commode et plus efficace de

5. https ://github.com/rsommerard/pitest-object-mutator

faire ces mutations directement sur le code source. Cela permettrai d'aller plus loin dans l'implémentation de mutants de type objet et de faire des choses plus complexes qui sont difficiles à faire au niveau du bytecode.