OLS-Test : Code Synthesis Applied To Test-Driven Development

Auteur :
Maxime CLEMENT

9 Février 2016

Table des matières

Introduction

Il existe plusieurs méthodologies pour développer une application, dont le TDD[1]. Cette pratique préconise d'écrire en premier lieux les spécifications sous forme de tests unitaires. C'est uniquement après avoir vérifier que ces spécifications ne sont pas respectées, que le développeur va écrire le code nécessaire à les valider. Cette méthode apporte ne nombreux avantages, comme la garantie d'avoir un code testé et moins sujet à la régression, le respect de la spécification, apporte de la confiance aux développeurs lors de *refactoring*.

Il est vrai que si dans la théorie, le développement dirigé par les tests ne possède que des avantages, ce n'est pas pour autant que cette technique est utilisée systématiquement dans le monde professionnel. Il est courant dans le monde professionnel de développer des applications sans aucun tests. Réaliser de bons tests est un coût pour l'entreprise, et ces dernières se focalisent généralement sur le code métier dans un premier temps. Si jamais des tests sont écrits, ils le sont très souvent qu'une fois l'application terminé, ne respectant pas le principe du TDD. Une question peut alors se poser : Est-il possible de mettre au point un outil permettant aux développeurs d'écrire uniquement une spécification, et ensuite synthétiser le code métier ?

L'objectif est de montrer qu'un tel outils existe. La solution proposée ici est appelée OLS-Test, elle est grandement inspirée par un outils appelé Nopol[1], et plus particulièrement DynaMoth[**?**], la dernière version de son moteur de synthèse. Cette approche utilise la notion d'oracle ainsi que des valeurs collectées de manières dynamiques dans le but de synthétiser une ligne de code valide.

L'outil a est évaluer sur un projet réel. Ce projet est disponible ici [LIEN GITHUB TEST MODIFIE]. L'évaluation porte sur plusieurs critères, donc le nombre de méthodes synthétisables par OLS-Test.

[1]Développement dirigé par les tests / Test driven development

Travail technique

1.1 Principe

Le principe est d'utiliser les suites de tests d'un projet comme spécifications de celui-ci. Chacune d'entre elles est liée à une méthode. Lors de l'exécution des tests, toutes les variables accessibles depuis la dernière ligne de la méthode référencée seront collectées. De plus, les assertions contenu dans le test permettent de récupérer les valeurs attendu par l'exécution de la méthode ciblée, ce sont les oracles. Il est ensuite possible de combiner de nombreuse manières les valeurs collectées avec des opérateurs afin d'essayer de faire correspondre une expression avec les oracles. Si une tel combinaison existe, la dernière ligne de la méthode est donc synthétisable, sinon il faudra réitérer après avoir synthétiser d'autres lignes, modifiant ou augmentant les valeurs collectées.

1.2 Hypothèses de travail

OLS-Test se limite uniquement aux méthodes ayant un type de retour, les méthodes de type *void* ne peuvent pas être synthétisées. De plus, la phase de collecte des oracles ne permet pas de récupérer une sortie de type exception lorsqu'elle n'est pas spécifiée dans une assertion (*@Test en java*). Lors de la synthèse, uniquement la dernière ligne de la méthode sera synthétisée, le reste de la méthode ne sera pas modifié. De part le fait que la collecte des oracles est dynamique, il n'y pas de limite sur la complexité ou l'écriture d'un test.

1.3 Overview

1.3.1 Référencement des méthodes à synthétiser

Les méthodes testées doivent être référencés de manières explicites dans les tests associés. Une convention a été mise en place, le développeur doit ensuite la respecter lors de l'écriture des spécifications. La figure 1.1 représente un exemple de référencement. À la différences des

annotations, cette convention ne demande pas d'ajouter de nouvelle dépendances à un projet car elle utilise simplement la documentation.

```
/**
 * @see mon.package.MaClass#MaMethod(type.param1, type.param2)
 */
```

Figure 1.1: Exemple de référencement utilisé à l'aide de la Javadoc

1.3.2 Collecte des données nécessaires

Collecte des variables accessibles depuis une position

La collecte des variables accessibles depuis une position est effectuée par DynaMoth. Cette position correspond à la dernière ligne de la méthode ciblée. DynaMoth utilise ensuite une API[1] de débogage pour placer un point d'arrêt à cette position. Il collecte ensuite toutes les valeurs accessibles : les variables, les attributs et même les appels de méthodes. Des constantes peuvent également être ajoutées. La figure **??** est un exemple de collecte à partir d'une position donnée.

[1] https://fr.wikipedia.org/wiki/Interface_de_programmation

```
 1 public class Exemple {
 2
 3       private int zero = 0;
 4
 5       public int addOne(int a){
 6               return a + 1;
 7       }
 8       public int max(int a, int b){
 9               if ( a > b )
10                       return a;
11               return b;
12       }
13 }
```

Constante	Variable	Attribut	Méthode
-1	a	zero	addOne
0	b		
1			

Figure 1.2: Exemple de valeurs collectées à partir de la ligne 11

Collecte des oracles lors de l'exécution des tests

La collecte des oracles s'effectue en modifiant les tests et en les exécutants. La transformation représentée dans la figure 1.3 a pour but de récupérer les valeurs attendues par les assertions pour ensuite les envoyer à DynaMoth. Les oracles sont collectés et évaluées de manières dynamiques.

```
methodTest :
+    collect(methodeCible.signature, methodTest.signature, sortie1)
     assertion(sortie1, methodeCible(param1))
+    collect(methodeCible.signature, methodTest.signature, sortie1)
     assertion(sortie2, methodeCible(param1))
```

Figure 1.3: Exemple de transformation pour récupérer les oracles

1.3.3 Combinaison des différents opérateurs de synthèse

Une fois les variables accessibles depuis une position donnée collectées, il faut les faire correspondre avec les oracles, pour chaque exécution de la position. Si cette correspondance n'est pas direct, DynaMoth essaye de combiner les variables avec des opérateurs pour agrandir l'ensemble des possibilités. Il recommence récursivement jusqu'à trouver une relation, ou atteindre un *timeout* paramètrable. La table 1.1 représente les opérateurs utilisées par DynaMoth.

Opérateurs	! ++ −− && | | == != <= < + − * / % [appel de méthode]

Table 1.1: Les opérateurs utilisés par DynaMoth

1.4 Implementation

OLS-Test est implementé en Java, pour les différentes transformation et instrumentation de code source, il utilise Spoon[2]. Les spécifications sous forme de tests sont également écrite en Java, à l'aide du framework JUnit[3]. Comme énoncé en 1.3.1, le référencement de méthode utilise la Javadoc et *@see*. OLS-Test utilise également Maven Invoker API[4], afin de démarrer différent *goals* sur le projets en cours d'analyse.

1.4.1 Injection du harnais de test

La phase de collecte des oracles expliquée en 1.3.2 s'effectue de la manière suivante. Les tests sont modifiées à l'aide Spoon pour récupérer les valeurs attendues par les assertions. Or, puisque ces tests seront exécuter à l'aide de Maven Invoker API, le context de OLS-Test ne sera pas accessible. Pour résoudre se problème, OLS-Test injecte également un harnais de test dans le projet contenant les méthodes à synthétiser. Ce harnais est composé d'un *Runner* personnalisé, ainsi que d'une structure contenant les différents oracles durant l'exécution. La collecte se fera donc dans le context du projet passé en paramètre. À la fin de l'exécution des tests, le *Runner* sérialise les oracles dans un fichier. Il ne restera plus qu'à OLS-Test de les récupérer une fois le *goal* maven terminé. La figure 1.4 représente le workflow lors de la phase de collecte des sorties.

[2] `http://spoon.gforge.inria.fr/`
[3] `http://junit.org/`
[4] `https://maven.apache.org/shared/maven-invoker/`

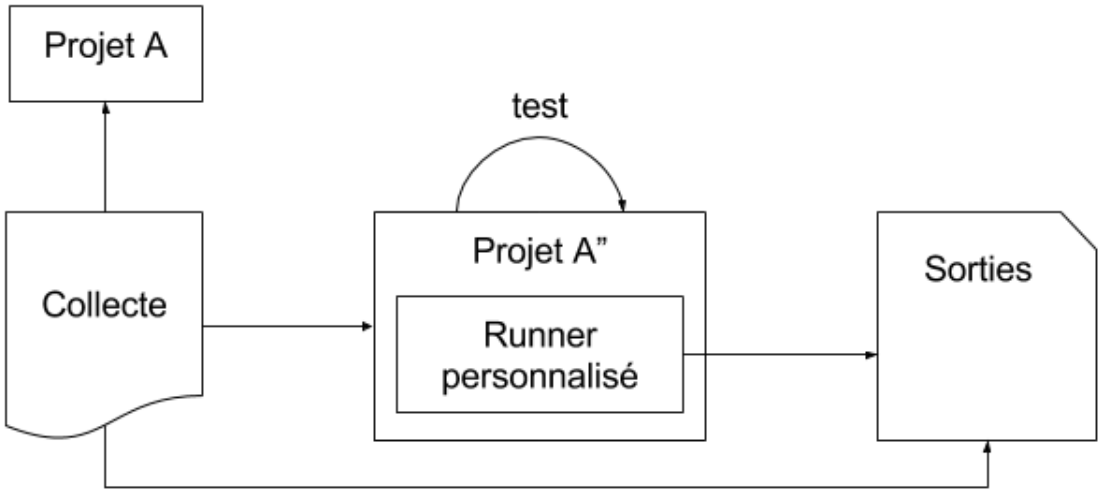

Figure 1.4: Workflow d'exécution lors de la collecte des sorties

1.4.2 Conservation du comportement original des tests

Le framework JUnit fonctionne le manière suivante, lors qu'une assertion échoue à l'intérieur d'un test, elle soulève une exception. Cependant, lors d'un tel comportement, les assertions suivantes ne seront pas exécutées et donc pas collectées. Afin collecter toutes les sorties, la transformation appliquée insert également des *try/catch* pour ne pas stopper l'exécution du test. Une exception est soulevée à la fin du test, uniquement si l'une des assertions a échoué. La figure 1.5 reprend l'exemple énoncé en 1.3.2 en appliquant cette transformation supplémentaire.

```
public void test() :
+ hasException=false
+ try
+    collect(methodeCible.signature, methodTest.signature, sortie1)
     assertion(sortie1, methodeCible(param1))
+ catch AssertionError
+    hasException=true
+ try
+    collect(methodeCible.signature, methodTest.signature, sortie2)
     assertion(sortie2, methodeCible(param1))
+ catch AssertionError
+    hasException=true
+ assertion(hasException, false)
```

Figure 1.5: Exemple de transformation spécifique à JUnit pour récupérer les oracles

1.5 Utilisation

```
Usage: OLS_Test
    -s, --source-path           path_program
    -m, --maven-home-path       path_maven_home
   [-c, --constant              one_constant_to_add]*
   [-t, --time-out-collection   time in second]
   [-d, --time-out-dynamoth     time in second]
   [-o, --override]
   [-v, --verbose]
```

Évaluation

L'évaluation de OLS-Test a été réalisé sur le projet java-util[1]. Ce projet a pour avantage de ne pas être trop complexe, et possède de une très bonne couverture de code ($> 98\%$). Parmis les différentes classes utilitaires de ce projet, l'attention s'est porté sur les classes *StringUtilities* et *TestStringUtilities*.

2.1 Transformations apportées

Afin de pouvoir évaluer OLS-Test sur ce projet, des petites transformation ont été appliquées. Tous d'abord, le référencement dans les méthodes de tests vers les implémentations comme expliqué en 1.1. Ensuite, la dernière ligne de chaque méthode référencée est remplacée par une exception. Suite à cette transformation, tous les tests utilisant ces méthodes échoues. Les sources utilisés pour l'évaluation sur disponiblen sur le GitHub[2] de OLS-Test.

2.2 Résultats

Lors des différentes exécutions de OLS-Test sur java-util, des métriques ont pu être obtenus, ces dernières sont présentées dans la table 2.1. Pour des contraintes de temps d'exécution, les méthodes collectés sur le type String ont été drastiquement réduite. Uniquement les méthodes *equals*, *equalsIgnoreCases* et *length* ont pu être utilisé. Lors des synthèses des méthodes *isEmpty* et *hasContent*, aucune solution n'a été trouvée en premier lieu. Cependant, c'est en ajoutant la constante *0*, une synthèse est possible. Pour les quatres dernières méthodes de table 2.1, des erreurs internes ont empêché le bon fonctionnement de OLS-Test. La table 2.2 montre les différentes synthèses réalisées.

[1] `https://github.com/jdereg/java-util`
[2] `https://github.com/maxcleme/OLS-Test`

Méthode	Constantes	Variables	Expressions	Temps(ms)	Synthèse
isEmpty	0	1	3230	11229	oui
hasContent	0	1	5125	12096	non
equalsIgnoreCaseWithTrim	∅	2	182	11924	oui
equals	∅	2	11	12592	oui
equalsIgnoringCase	∅	2	11	12325	oui
lastIndexOf	∅	/	/	/	non
length	∅	/	/	/	non
levenshteinDistance	∅	/	/	/	non
damerauLevenshtein	∅	/	/	/	non

Table 2.1: Résultats des différentes exécutions sur java-util

Méthode	Ligne synthétisée
isEmpty	(s == null) || (0 == s.length())
hasContent	(s != null) && ((s == null) || (0 != s.length()))
equalsIgnoreCaseWithTrim	(s2.length() - s1.length()) <= s1.length()
equals	! str1.equalsIgnoreCase(str2)
equalsIgnoringCase	! str1.equalsIgnoreCase(str2)

Table 2.2: Synthèses trouvées lors des exécutions sur java-util

Limitations

Le nombre de tests est determinant car il représente une spécification pour la méthode à synthétiser. Si ils sont peu nombreux, le code synthétisé s'éloignera du comportement nominal. À l'inverse, si ils sont trop nombreux, il est envisageable que DynaMoth n'arrive pas à trouver une relation entre les entrées et les sorties dans le temps qui le lui est accordé.

Conclusion

Grâce à la synthèse de code, un développeur peut se concentrer uniquement sur les tests. Le développement de ces derniers n'est donc plus une perte de temps car c'est l'unique chose qu'il doit faire. OLS-Test pourrait permettre au monde professionnel ne pas voir les tests comme une perte de temps, mais plutôt comme une alternative au classique cahier des charges. On peut également imaginer intégrer OLS-Test dans les cycles d'intégration continue. Il suffirait uniquement de faire évoluer les tests, et le code s'adapterait tout seul.

OLS-Test est encore jeune et il pourrait être grandement améliorer. Remplacer le lancement en ligne de commande par un plugin pour environnement de développement permettrait d'être plus *user-friendly*. Il subsiste encore également de nombreuses optimisations possibles, comme par exemple compiler les fichiers sans utiliser maven car se dernier est chronophage. Pour finir, il pourrait être intéressant de lancer la synthèses de plusieurs méthodes en parallèle.

Bibliography

[1] Daniel Le Berre Favio DeMarco Martin Monperrus, Jifeng Xuan. Automatic repair of buggy if conditions and missing preconditions with smt. 2014.