TITRE

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. En effet, 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, pour ensuite synthétiser le code métier ?

L'objectif est de montrer qu'un tel outils existe. La solution proposée ici est appelée ????, 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 les entrées / sorties des tests pour définir des contraintes, et essaye de les résoudre.

EVAL ????

[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 pas encore implementée. Lors de l'exécution des tests, toutes les variables accessibles depuis la méthode ciblée seront collectées, ce sont les sorties. De plus, les assertions contenu dans le test permettent de définir les valeurs attendu par l'exécution de la méthode ciblée, ce sont les sorties. Il est ensuite possible de combiner de nombreuse manières les entrées avec des opérateurs afin d'essayer d'arriver à la sortie. Si une tel combinaison existe, la méthode est donc synthétisable, sinon il faudra réitérer après avoir synthétiser d'autres méthodes, augmentant le nombre d'entrées.

1.2 Overview

1.2.1 Ciblage

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

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

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

1.2.2 Entrées

La collecte des entrées est effectuée par DynaMoth. Une position dans le code source doit être définie lorsque le synthétiseur est appelé. 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. Il collecte ensuite toutes les valeurs accessibles ou possible lors de chaque exécution de la position : les variables, les attributs et même les appels de méthodes. Des constantes peuvent également être ajoutées.

1.2.3 Sorties

La collecte des sorties s'effectue en modifiant les tests et en les exécutants. La transformation représentée dans la figure 1.2 a pour but de récupérer les valeurs attendues par les assertions pour ensuite les envoyer à DynaMoth. Les sorties sont évalué et collecté 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 1.2: Exemple de transformation pour récupérer les sorties

1.2.4 Combinaisons

Une fois les entrées et sorties collectées, il faut les faire correspondre. Pour chaque exécution de la position ciblée, il faut trouver une relation entre toutes les entrées et la sortie. Si cette relation n'est pas direct, DynaMoth essaye de combiner les entrées avec des opérateurs pour agrandir l'ensemble des possibilités. Si aucune relation existe, il recommence récursivement jusqu'à trouver une relation, ou atteindre un *timeout* paramètrable. La table 1.1 représente les opérateurs utilisées par DynaMoth.

Unaire	! ++ −−
Binaire	&& || == != <= < + − * / %

Table 1.1: Les opérateurs utilisés par DynaMoth

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

1.3 Implementation

??? 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 1.2.1, le ciblage de méthode utilise la Javadoc et *@link*. ??? utilise également Maven Invoker API[4], afin de démarrer différent *goals* sur le projets en cours d'analyse.

1.3.1 Injection

La phase de collecte des sorties expliquée en 1.2.3 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 ??? ne sera pas accessible. Pour résoudre se problème, ??? injecte également du code dans le projet en cours de modification, ainsi qu'un *Runner* personnalisé. La collecte se fera donc dans le context du projet en cours d'intrumentation. À la fin de l'exécution des tests, le *Runner* sérialisera les sorties dans un fichier. Il ne restera plus qu'à ??? de les récupérer une fois le *goal* maven terminé. La figure 1.3 représente le workflow lors de la phase de collecte des sorties.

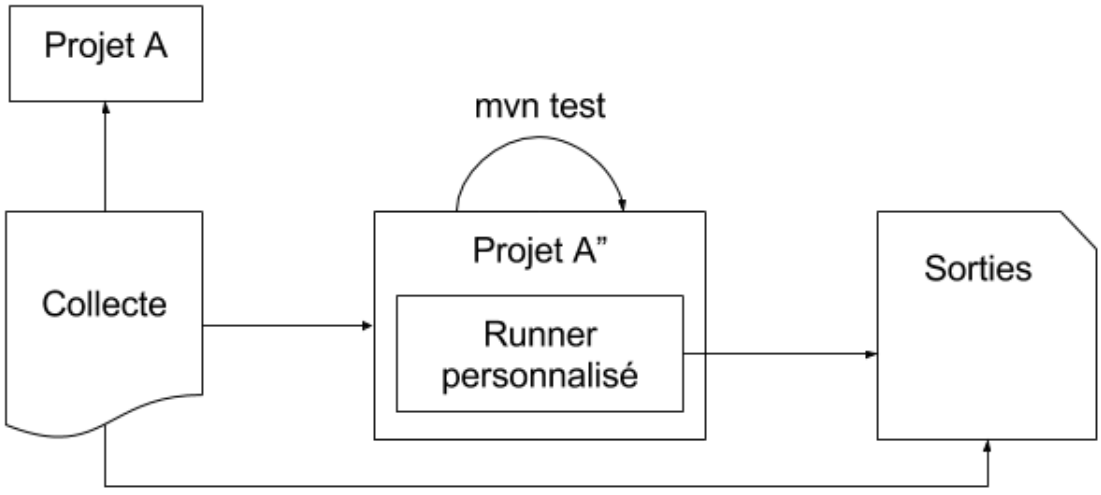

Figure 1.3: Workflow d'exécution lors de la collecte des sorties

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

1.3.2 Test

TRY/CATCH TO PRESERVED ORIGINAL BEHAVIOUR

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 1.4 reprend l'exemple énoncé en 1.2.3 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 1.4: Exemple de transformation spécifique à JUnit pour récupérer les sorties

1.4 Scope

??? 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 sorties expliquée en 1.2.3 ne permet pas de récupérer une sortie de type exception lorsqu'elle est spécifiée dans l'annotation *@Test*.

1.5 Utilisation

```
Usage: OLS_Repair
    -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

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. ??? 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 ??? dans les cycles d'intégration continue. Il suffirait uniquement de faire évoluer les tests, et le code s'adapterait tout seul.

??? 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 prend énormément de temps. 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.