V&V - Projet

Sujet 2 : Dynamic testing

22 Décembre 2017

Pierre Gaudichon

Timothée Merlet-Thomazeau

Ahmed Nomane

Introduction

Ce projet a été réalisé par **Pierre Gaudichon**, **Timothée Merlet-Thomazeau** et **Ahmed Nomane**. Il a été supervisé par **Nicolas Harrand** dans le cadre du cours de **V&V** de **Benoît Baudry**.

Le but de ce projet est d'implémenter un outil d'analyse dynamique de projet Java (sujet 2[1]). Pour cela nous utilisons principalement l'outil Javassist pour l'analyse et la modification du bytecode[2] Java. L'idée est de modifier le projet cible en lui injectant des sondes pour logger différents événements. Les tests du projet cible sont ensuite exécutés et nous pouvons générer des rapports pour vérifier différentes métriques sur la couverture de tests.

Notre projet doit être capable des analyses suivantes :

- Compter le nombre d'exécution de chaque ligne de code du projet cible. Cette analyse est fonctionnelle même si le rapport n'est pas très clair.
- Déterminer si toutes les branches d'une conditionnelle ont été exécutées. Cette analyse n'est pas fonctionnelle, mais elle existe au stade embryonnaire.
- Obtenir la séquence d'appels des méthodes. Cette analyse est complètement fonctionnelle. De la même façon, le rapport serait à améliorer.
- Enregistrer les arguments des appels des méthodes. Cette analyse est presque fonctionnelle.

Le code source du projet est accessible sur github[3]. Ce document est disponible depuis Google Document[4].

Solution

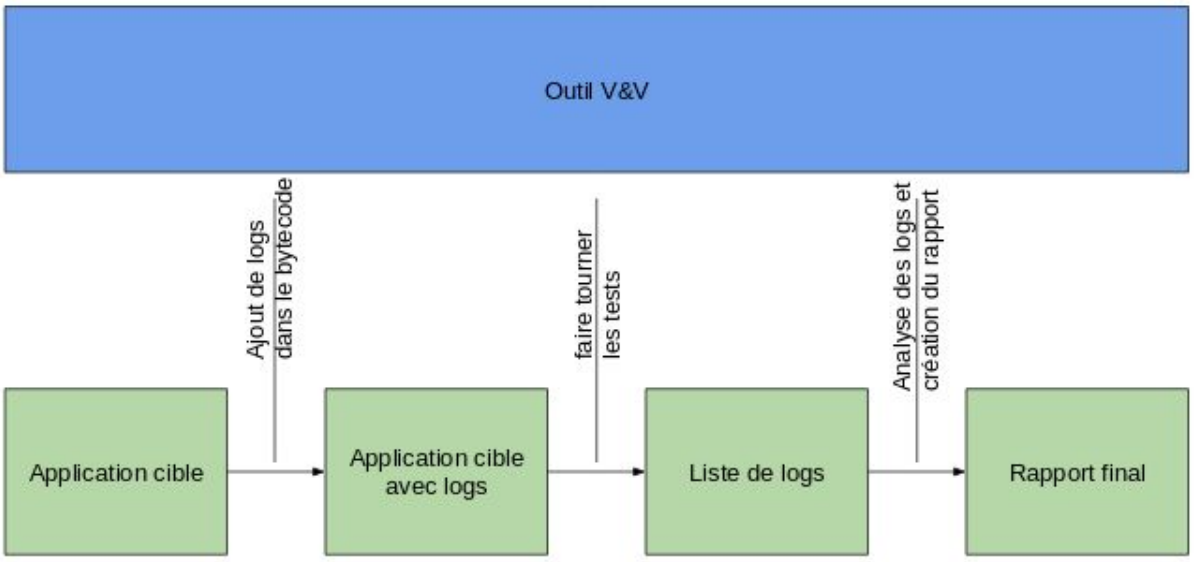
Pour réaliser notre outil de couverture de test, l'approche a été de rajouter dans le programme cible des sondes à chaque début de fonction et de bloc et ensuite d'observer les différents logs produits afin de les analyser. Notre outil se découpe donc en deux parties bien distinctes.

- La première ajoute directement un système de sondes sur le bytecode avec la technologie *Javassist*. Quand cette étape est réalisée, nous pouvons lancer les tests fournis par le programme. Nous récupérons alors une liste de logs.

- La deuxième partie de l'outil est l'analyse de ces différents logs afin de pouvoir réaliser différents rapports sur le code. Nous pouvons notamment calculer la couverture de code de l'application cible.

[1] Sujet : https://github.com/Software-Testing/Project-2017-2018
[2] Bytecode : Code intermédiaire ou code octet
[3] Github du projet : https://github.com/PierreGaudichon/vev-dynamic-testing
[4] Compte rendu : http://bit.ly/2DyPs44

Schéma global du fonctionnement de l'application



Création des logs

Nous avons réalisé la première partie de l'application avec Javassist. Le bytecode de l'application est modifié pour rajouter des logs que nous utiliserons par la suite.

Avant l'insertion (bytecode décompilé)

```
public boolean isOrigin() {
  return this.getX() == 0 && this.getY() == 0;
}
```

Après l'insertion (bytecode décompilé)

```
public boolean isOrigin() {
  Logs.getInstance().addLogs("CALLING", "METHOD", "istic.fr.prog_test.Point.isOrigin()");
  Logs.getInstance().addLogs("BEGIN", "METHOD", "istic.fr.prog_test.Point.isOrigin()");
  boolean var10000;
  if (this.getX() == 0 && this.getY() == 0) {
    Logs.getInstance().addLogs("BEGIN", "BLOCK",
"istic.fr.prog_test.Point.isOrigin(),14,2");
    var10000 = true;
    Logs.getInstance().addLogs("END", "BLOCK",
"istic.fr.prog_test.Point.isOrigin(),14,2");
  } else {
    Logs.getInstance().addLogs("BEGIN", "BLOCK",
"istic.fr.prog_test.Point.isOrigin(),40,2");
    var10000 = false;
    Logs.getInstance().addLogs("END", "BLOCK",
"istic.fr.prog_test.Point.isOrigin(),40,2");
  }
  boolean var2 = var10000;
  Logs.getInstance().addLogs("END", "METHOD", "istic.fr.prog_test.Point.isOrigin()");
  return var2;
}
```

Exemple de bytecode d'une méthode avant et après l'insertion des sondes de log.

Afin d'ajouter ces logs, nous avons découpé l'architecture en 3 classes principales :

- *ClassLogger* : permet de gérer la classe en entière, elle utilise les deux autres classes pour fonctionner.
- *MethodLogger* : permet de gérer les différentes méthodes de la classe cible, elle ajoute des logs au début pour le nom de la méthode et à chaque début et fin de bloc qu'elle croise.
- *ConstructorLogger* : permet d'ajouter les logs sur le ou les constructeurs de la classe.

Notre programme insère des logs au début et a la fin de chaque méthode. Cela nous permet de suivre les appels de méthode imbriqués et ainsi de connaître la suite exacte d'appels de méthode.

De la même façon, notre programme insert des logs au début et à la fin des *blocks* du bytecode. Un *block* est une suite d'instructions ne contenant pas d'embranchements. Avec ces logs, on peut vérifier si chaque partie d'une branche conditionnelle est bien exécuté.

Création des rapports

Une fois les sondes insérées dans le code cible et les tests lancés, il faut gérer la liste de logs que nous récupérons.

Pour cela les *streams* de Java, qui permettent de gérer de longues listes efficacement, ont été utilisés. Les rapports consistent essentiellement à comprendre l'architecture émergente des logs et de l'afficher à l'utilisateur.

Number of execution of each block.	**Sequence of method calls.**
istic.fr.prog_test.Point.notTestedYet(),36,2 0 istic.fr.prog_test.Point.getX(),0,5 6 istic.fr.prog_test.Point.isOrigin(),14,2 1 **…** istic.fr.prog_test.Point.distance(istic.fr.prog_test.Point),0,47 1	istic.fr.prog_test.Point.setX(int) istic.fr.prog_test.Point.setY(int) istic.fr.prog_test.Point.setX(int) istic.fr.prog_test.Point.setY(int) istic.fr.prog_test.Point.distance(istic.fr.prog_test.Point) istic.fr.prog_test.Point.setX(int) **…** istic.fr.prog_test.Point.setX(int) istic.fr.prog_test.Point.getY()

Exemple de rapport généré

Outils de gestion de code

Pour notre projet, nous avons utilisé les outils offerts par Github pour la gestion du code.

- **Git** lui même, pour centraliser le code en suivre ses évolutions.
- L'**issue tracker** de Github, qui a servi plus de *todo list* que de gestionnaire d'issue en lui même.

De plus, nous avons mis en place **cobertura** pour vérifier la couverture de nos tests unitaires et nous permettre de la compléter.

Enfin, nous avons mis en place **Travis**[5] pour l'intégration continue. Celui çi nous permet de voir si le projet est dans un état stable.

Usage

- Pour compiler le projet, lancer les tests et calculer la couverture de tests, il faut utiliser le script *build.sh*.
- Pour lancer le projet, il faut utiliser un IDE.

Evaluation

Pour tester notre projet, nous avons utilisé un programme jouet pendant la majeure partie du développement. Celui ci consistait en une classe unique *Point* et sa classe de test *PointTest*. Ce projet nous a permis de travailler sur un ensemble simple de données.

Tout l'ajout de sondes et les reports ont été testé avec ce projet. Ensuite, la création de tests unitaires ont permis de vérifier le bon fonctionnement du programme. Grâce à l'intégration de *cobertura*, avons pu vérifier que ces tests unitaires ont une couverture de 100%. Il est intéressant de voir que cobertura est capable de gérer les opérations *filter* des *streams* Java.

Par la suite, nous avons utilisé le package *commons-cli* pour voir si notre projet fonctionne toujours. Ce n'est pas le cas. Dans le cas d'un programme plus compliqué, nos algorithmes de manipulation de bytecode ne sont plus adaptés. Malheureusement, nous n'avons pas eu le temps de régler les bugs.

Le système de gestion de différent projet cibles est fonctionnel. Notre programme peut donc être utilisé pour générer des rapports sur différentes bases de code.

De plus, notre projet est scalable. En effet, il faut très peu de temps pour insérer les sondes dans le code cible. L'insertion se fait en *O(n)* avec *n* le nombre de ligne de code du projet cible. De la même façon, les rapports sont générés très efficacement.

[5] Travis : https://travis-ci.org/PierreGaudichon/vev-dynamic-testing

Discussion

Une grande partie de ce projet consiste a utiliser Javassist pour manipuler le bytecode du projet cible. Cet outil s'est avéré ne pas être adapté pour ce projet.

Javassist est un outil trop bas niveau. Il opère au niveau du bytecode. Cela rend difficile beaucoup d'opération, notamment la gestion des branchements conditionnels. Globalement, pouvoir interagir avec le code source du projet cible au lieu du bytecode aurait été plus simple et aurait permis une plus grande granularité dans nos opérations.

Avec le recul et l'expérience acquise pendant ce projet, nous avons pu nous rendre compte, maintenant, que Spoon ou l'API de Réflexion de Java sont plus appropriés dans notre cas. En effet, beaucoup d'outils pour ajouter des logs dans un code Java utilisent cette API de Réflexion.

De plus, nous avons eu des difficulté pour trouver une solution pour tester la bonne intégration des logs dans le bytecode. En effet le bytecode n'est pas facile à manipuler et à inspecter, ce qui rajoute une difficulté supplémentaire dans la construction de nos tests.

Aussi, les IDE[6] que nous utilisons ajoutent des packages, modules et classes au *classpath* ce qui nous a empêché d'utiliser notre projet en ligne de commande. Nous n'avons pas réussi à résoudre ce bug dans le temps imparti et notre programme ne peut pas s'exécuter en ligne de commande en l'état.

Néanmoins, la création de rapports avec les logs obtenus est très intéressante. Elle a permis de se rendre compte des informations nécessaires pour créer des outils tels que le calcul de la couverture des fonctions, conditions et instructions en Java.

Une des améliorations possible de notre projet est l'ajout de plus de rapports sur la couverture des tests. Actuellement, seul deux rapports sont générés. En créer d'autres nécessite de modifier les logs insérés pour obtenir plus d'information pendant l'exécution des tests.

Conclusion

Ce projet nous a permis de nous rendre compte de la difficulté de manipuler du bytecode Java. Il nous a aussi permis de pratiquer et d'améliorer nos compétences dans l'écosystème Java et la gestion de code. Nous avons notamment utilisé des outils comme **Cobertura**, **Maven**, **Git** et **Travis**.

[6] IDE : Integrated Development Environment (Fr : environnement de développement)