Proof of Concept: réparation automatique de régression à l'aide du versionning

Yassine Badache et Romain Sommerard

January 17, 2016

Contents

1 Introduction

Avec plus de 2 millions de dépôts actifs[1], GitHub[2], basé sur son système de versionning Git[3], est la plateforme de partage de code *(open-sourcing)* la plus populaire au monde. Elle a notamment su acquérir sa notoriété grâce à la facilité de partage, de contribution et de vérification via une interface simplifiée et efficace de *pull requesting*[4].

Cependant, malgré cette facilité d'usage, il arrive que des erreurs passent à travers les mailles du filet, d'une contribution extérieure ou même au sein d'une équipe reconnue. C'est alors un travail laborieux, malgré l'usage des tests, que de revérifier son code afin de trouver l'erreur laissée par mégarde dans la version actuelle.

Ces erreurs qui parviennent d'une version à une autre sont rarement dépendantes de l'architecture ou du contexte, et nous sommes capables de déduire où se trouve l'erreur à l'aide des batteries de tests fournies avec un logiciel.

Prenant certaines hypothèses en compte, nous nous proposons de réaliser un Proof of Concept de ce principe: en détectant une erreur sur la version actuelle d'un logiciel, nous essayons de la fixer en utilisant le corps de la méthode en faute à sa version n-1.

Afin de valider cette expérience, nous partirons de la granularité la plus simple: un programme contenant un bug à sa version actuelle, qui n'apparaissait pas dans sa version antérieure. Le POC sera validé si, après passage de notre travail, le programme en question passe automatiquement la compilation et tous les tests concernés par ce type d'erreurs passent vert.

2 Travail technique

2.1 But

Le but de ce POC est de réparer les bugs de régression à l'aide de logiciel de versionning en prenant la version précédente (non buggée) du programme. Il s'agit d'un POC qui concerne les bugs de régression uniquement, et vise à les réparer de manière certes simple, mais automatique.

2.2 Overview

Nous partons du principe qu'un push sur le répertoire distant est vérifié par un serveur d'intégration continue. Celui-ci n'accepte et ne valide le push que pour les programmes qui passent les tests. Sinon il les refuse. Notre approche se place avant la validation, lorsque les tests sont exécutés. Si les tests passent, nous n'intervenons pas. Nous n'intervenons que dans le cas où certains tests passent rouge.

[1] GitHub statistics - http://githut.info/
[2] GitHub, Where softare is built - https://github.com/
[3] Git, versionning system - https://git-scm.com/
[4] Pull requesting help - https://help.github.com/articles/using-pull-requests/

Dans ce POC, nous émettons l'hypothèse que la version n-1 du programme que nous voulons réparer est toujours fonctionnelle; les tests passent tous.

Le POC suppose également que les noms des tests du programme contiennent le nom de la méthode testée. Chaque test ne contient qu'un seul assert et donc ne teste qu'une méthode. De ce fait, notre approche ne prend donc pas en compte la surcharge de méthode.

2.3 Algorithme

Il y a deux algorithmes à prendre en compte: le premier est un algorithme d'ordre général, tandis que le second concerne le traitement du programme buggé.

Le premier algorithme est le suivant:

Algorithm 1 Traitement général du POC

```
Cloner le programme concerné
Compiler et exécuter les tests
Pour tous les tests qui ne passent pas
    Utiliser l'algorithme de réparation automatique
Fin Pour
Relancer la compilation et exécuter les tests
```

L'algorithme est un succès si la compilation et les tests sont tous deux verts à la fin de cet algorithme. Il se contente d'appliquer le résultat de nos travaux sur le programme en question, récupéré directement sur un dépôt Git distant. Concernant l'algorithme de réparation énoncé ci-dessus, le voici décrit:

Algorithm 2 Algorithme de réparation automatique de bug

```
Pour tous les tests qui ne passent pas
    Récupérer le test
    Récupérer la méthode correspondant au test rouge
    Récupérer le corps de cette méthode
    Se rendre à la version  1 du programme buggé
    Récupérer le corps de la méthode à la version  1
    Remplacer le corps de la méthode actuelle par celui de la méthode  1
Fin Pour
```

Cet algorithme laisse transparaître le fait que l'intégralité de la méthode soit transposée d'une version ancienne à une autre.

2.4 Implémentation

L'implémentation utilise le language Java ainsi qu'un script bash. Nous utilisons le moteur de transformation de code source Spoon[5] pour effectuer les changements.

Le script bash permet de lancer le processus de réparation. Il récupère le projet à réparer et en fait 3 copies:

- *Previous*: version N-1 du programme.
- *Current*: version courante du programme.
- *After*: version réparée du programme.

Le script lance les tests sur les copies *Previous* et *Current* en enregistrant les résultats dans des fichiers. Ensuite, il lance la réparation avec le programme Java qui exécute 2 processeurs Spoon:

- *ProcessorDiff*: récupère le contenu de la méthode buggée de la copie *Previous*.
- *ProcessorApplyDiff*: applique le contenu récupéré par *ProcessorDiff* à la copie *After*.

Pour finir, les tests sont lancés sur la copie *After* pour vérifier le fix.

Un répertoire distant contient le projet de test de notre POC. Le dernier commit de ce projet contient plusieurs modifications pour vérifier que le POC modifie bien les éléments au bon endroit. De plus, le dernier commit contient un bug de régression, qui sera l'objet de notre étude.

2.5 Utilisation

L'utilisation du logiciel se fait via un script bash. Il ne convient de l'utiliser, comme dit précédemment, que si la version précédente du programme à réparer est opérationnelle, et si il respecte certaines conventions de nommage, notamment celles des tests.

[5]Spoon, Source Code Analysis and Transformation for Java - http://spoon.gforge.inria.fr/

3 Evaluation

L'évalution du POC se fait sur un projet de test contenant plusieurs modifications entre deux versions. La version N-1 de ce projet passe tous les tests alors que la version N ne les passe pas.

Les données intéressantes tirées du projet de test sont disponibles sur le Github du projet[6].

Le listing 1 est un exemple de test. On retrouve facilement le nom de la méthode testée grâce au nom du test. Il n'y a qu'une seule assertion dans chaque test, comme indiqué précédemment.

Listing 1: Test for A.getPrivateInteger() method.

```
@Test
public void testGetPrivateInteger() {
        A a = new A();
        assertEquals(42, a.getPrivateInteger());
}
```

Le listing 2 correspond à la méthode buggée testée par le test du listing 1. Le bug provient du retour. En effet, le test attend la valeur 42 alors que c'est la valeur 24 qui est retournée.

Listing 2: A.getPrivateInteger() method before repair.

```
public int getPrivateInteger() {
        return 24;
}
```

Listing 3: A.getPrivateInteger() method after repair.

```
public int getPrivateInteger() {
        // mPrivateInteger = 42
        return mPrivateInteger;
}
```

Comme le montre le listing 3, notre programme de réparation récupère le contenu de la méthode de la version N-1 pour l'appliquer à la version courante.

Cette version passe les tests avec succès. La correction est donc *valide*.

4 Discussion / Limitations

Nous sommes conscients que cette méthode contient quelques faiblesses. Elles sont décrites ci-dessous.

[6]Projet de test - https://github.com/rsommerard/program-repair-test

4.1 Hypothèses

Dans un premier temps, la réussite de ce POC réside dans l'étroiteté de ses hypothèses. En effet, elle réclame de nombreuses contraintes pour sa validation. En voici une liste exhaustive:

- Les tests doivent contenir le nom des méthodes qu'ils traitent.
- Les tests ne contiennent qu'un seul *assert*.
- Les tests servent de contrat pour une méthode unique.
- Les changements entre deux versions d'une méthode n'impliquent pas de changements architecturaux dans le logiciel.
- Les méthodes ont un nom unique.
- Les méthodes ne sont jamais surchargées.
- ...

Elles sont nombreuses, et restreignent le champ des programmes *"réels"* possiblement traités. Elles entravent donc la validation de notre POC.

4.2 Validation sur projets réels

Comme énoncé précédemment, notre POC n'a pas été validé sur de grands projets, ou même des projets réels utilisés à l'heure actuelle. C'est un second point bloquant, dans la mesure où notre POC s'applique sur un espace très faible de projets valides.

4.3 Axes d'amélioration

Il est possible d'améliorer notre POC en augmentant son espace de projets utilisables. Par exemple, prendre en compte les conventions de nommage différentes. Le plus grand axe d'amélioration serait la prise en compte de différents types de contrats. En effet, les contrats des tests sont des contrats par méthode unique, et non par ensemble de contraintes. Rendre possible la non-régression par contrat de contraintes serait un grand pas en avant dans l'espace des programmes valides.

De même, si jamais un bug perdure sur plusieurs versions, notre programme ne permet pas de les traiter. Prendre en compte N commits pour la résolution des bugs serait également un grand progrès vers l'achèvement d'une méthode plus globale de réparation automatique.

5 Conclusion

Notre approche permet de réparer les bugs de régression en s'appuyant sur l'historique des commits. Il s'appuie sur plusieurs hypothèses qui, bien qu'entravant la validation du POC, sont nécessaires à sa bonne complétion.

Avec cette méthode, on se prémunit des erreurs d'inattention du développeur (e.g. commit d'une faute de frappe, d'un changement de valeur) qui retarderaient les mises en production ou les debugs.

Des améliorations sont possibles, notamment, en prenant en compte tout l'historique de commit ou les différents types de contrat représentés par les tests.

Malgré le scope de notre POC, nous avons produit quelque chose de fonctionnel et efficace, qui permet aux développeurs les plus maladroits d'éviter de se faire réprimander pour autre chose que leurs talents de codeur. Nous sommes tous égaux face à un compilateur, et il faut aimer même les plus étourdis d'entre nous, car comme le dit si bien Gauthier Galand, *"La réponse est l'amour"*.