OPL – Code Rewinder

*Le réparateur automatique de bugs par agrégation de code multi-version*

Code Rewinder est un réparateur automatique de bugs découpé en trois principales parties. Dans un premier temps, il télécharge les n dernières versions du projet à réparer sur GitHub. Puis, il construit, grâce à ces versions, une nouvelle version qui contient toutes les versions des fonctions. Il exécute par la suite, les tests unitaires tout en faisant varier les versions des méthodes jusqu’à trouver le plus de tests « successful » possibles.

WATTEBLED Étienne

03/01/2017

OPL – Code Rewinder

Le réparateur automatique de bugs par agrégation de code multi-version

Table des matières

[Introduction 2](#_Toc469944841)

[Travail technique 3](#_Toc469944842)

[Améliorations préalables 3](#_Toc469944843)

[Analyses et corrections 4](#_Toc469944844)

[Exécution de Code Rewinder sur demoproject 4](#_Toc469944845)

[Exécution de Code Rewinder sur jsoup 5](#_Toc469944846)

[Évaluation 12](#_Toc469944847)

# Introduction

Code Rewinder est un réparateur automatique de bugs utilisant les n derniers commits (donc n dernières versions) de SVN pour parvenir à ses fins.

Il fonctionne principalement en trois temps :

* Avant tout, il doit exécuter un batch permettant de télécharger les n dernières versions du projet à réparer sur GitHub.
* À l’aide de ces versions, il construit en utilisant une librairie appelée Spoon, une nouvelle version comprenant toutes les versions des méthodes. Un switch sur un attribut static de la classe permet de passer une méthode d’une version à l’autre et un autre attribut permet d’obtenir la version maximale. Pour chaque méthode, il y a donc deux attributs, un qui contient le numéro de version en cours, le deuxième qui contient le numéro de version maximale.
* Code Rewinder exécute ensuite les tests unitaires tout en faisant varier les versions des méthodes jusqu’à obtenir le maximum de tests « successful » possibles.

Avant tout, quelques améliorations ont été apportées à Code Rewinder.   
Par la suite, Code Rewinder a été exécuté pour la première fois sur un projet concret et des corrections ont été apportées. Des tests unitaires ont alors été mis en place afin d’éviter qu’il y ait une quelconque régression dans un futur plus ou moins éloigné.

# Travail technique

## Améliorations préalables

Des améliorations ont été apportées à Code Rewinder, même si celles-ci n’étaient pas bloquantes :

1 – La plupart des utilisateurs dispose d’un OS Windows. Malheureusement, le batch permettant de télécharger les n dernières versions d’un projet github était conçu pour Linux. Un batch Windows a donc été mis en place afin de satisfaire un maximum d’utilisateurs.

2 – Le réparateur était divisé en deux parties, une partie concernant le téléchargement des versions et la transformation, et une autre partie permettant d’exécuter les tests unitaires et donc, de réparer. Les deux parties ont donc été liées afin de simplifier l’exécution de Code Rewinder.

3 – Une compilation et un envoi automatique des classes dans le dossier target du projet a été mis en place afin que Reflections puisse les retrouver et les charger. Autrefois, on devait les déplacer manuellement entre les deux parties de Code Rewinder.

4 – Lors du chargement des classes avec Reflections, le nom du package était autrefois écrit en dur dans l’application. Un argument -packages a donc été ajouté et celui-ci permet de lister les packages du projet à réparer dans Code Rewinder (chaque package doit être séparé par un point-virgule, comme le classpath). Il est même possible même de préciser le nom de la classe avec le package et, indiquer le nom du package en entier n’est pas obligatoire (si le package des classes contient un de vos mots alors la classe est prise en compte, qu’il s’agisse d’une classe de test ou d’une classe du projet).

5 – L’utilisation de StringBuilder au-delà de la concaténation de deux chaînes de caractères a été favorisée afin de gagner en performances.

## Analyses et corrections

### Exécution de Code Rewinder sur demoproject

Afin de faire évoluer Code Rewinder Progressivement, ce dernier n’a pas été exécuté de suite sur un gros projet. Il a tout d’abord été exécuté sur un petit « projet » appelé demoproject (avec trois classes applicatives et une seule classe de tests qui ne contient qu’un seul test toujours vrai) qui était fourni dans la version bêta-bêta. Le but de demoproject était vraiment d’avoir une base un minimum fonctionnelle avant de se lancer dans l’exécution de Code Rewinder sur un véritable projet.

Code Rewinder utilise une librairie Java appelée Reflections qui permet entre-autres d’accéder et de modifier la valeur des attributs static des classes (et donc, de changer la version d’une méthode). Cependant, la dépendance n’était pas présente dans le POM Maven. Elle a, par conséquent, été ajoutée.

Un autre problème fit, par la suite, son apparition : Reflections ne parvenait pas à accéder aux attributs des classes car ces derniers étaient « private ». Ils ont donc été changés en « public » mais étant donné qu’ils étaient aussi déclarés « final », alors le changement de valeur n’était donc pas autorisé par Java. Le mot clé « final » a donc dû être supprimé afin que Code Rewinder puisse changer la version des méthodes.

Malgré ces corrections sur les attributs, une autre exception s’est déclenchée durant l’analyse. Les attributs étaient de type Integer, alors que, lors de la modification des valeurs, il était question du type primitif « int ». Cette erreur a été corrigé en créant à chaque fois un nouvel Integer et en passant le type primitif au constructeur de la classe Integer.

Le projet demoproject a permis la correction d’un autre point. Les classes n’étaient pas automatiquement compilées et envoyées dans le target du projet Code Rewinder. Par conséquent, Reflections ne parvenaient pas à retrouver (charger) les classes. La compilation et le déplacement des fichiers « .class » ont été effectués grâce à l’argument « --compile » de Spoon.

Grâce à ces modifications, le projet demoprojet a par la suite été fonctionnel pour la première fois et fonctionnait correctement sur Code Rewinder.

### Exécution de Code Rewinder sur jsoup

Code Rewinder a par la suite été testé sur Jsoup, qui est, quant à lui, un projet de taille réelle avec lequel de multiples erreurs sont survenus durant l’analyse.

La première anomalie a été le fait qu’un fichier appelé « package-info.java » empêchait la compilation. Ce fichier Java n’est en fait utilisé que pour la javadoc, il n’est donc pas important et peut être ignoré sans répercutions par Code Rewinder. Une méthode a été mis en place permettant en traitement préliminaire d’effacer tous les fichiers portant ce nom dans le projet.

Mais le travail était loin d’être terminé et ce n’était que le début.   
Une autre anomalie survenue a été le fait qu’un paramètre d’une méthode a changé de nom. Par conséquent, Code Rewinder ne parvenait pas à faire le lien entre l’ancien nom de variable et le nouveau : ce problème a été corrigé.

C’est alors qu’un problème beaucoup plus grave est apparu. En réalité, dans un cas très particulier, l’héritage provoquait une boucle infinie qui générait un StackOverflowError. Le cas particulier est le suivant :   
- Deux classes : A et B   
- B hérite de A :   
- Une méthode m commune à A et à B   
- La méthode m de la classe fille B fait un appel à super pour appeler la méthode m de la classe mère A.

public class B extends A {

public void m() {

…

super.m() ;

…

}

}

public class A {

public void m() {

…

}

}

Ces classes, après les transformations faites par Code Rewinder sur deux « commits » deviennent alors :

public class A {

public void m() {

switch (m\_version) {

case 0 :

m\_0() ;

break ;

case 1 :

m\_1() ;

break ;

}

}

public void m\_0() {

…

}

public void m\_1() {

..

}

public Integer m\_version = 0 ;

public Integer m\_version\_max = 1 ;

}

public class B extends A {

public void m() {

switch (m\_version) {

case 0 :

m\_0() ;

break ;

case 1 :

m\_1() ;

break ;

}

}

public void m\_0() {

…

super.m() ;

…

}

public void m\_1() {

..

super.m() ;  
 …

}

public Integer m\_version = 0 ;

public Integer m\_version\_max = 1 ;

}

Par conséquent, le comportement attendu lorsque la méthode m de l’objet B est appelée, si la version est égale à 0 est :   
- B.m   
- B.m\_0   
- A.m   
- A.m\_0

Cependant, cela n’était pas le cas car, par défaut, Java appelle la méthode de la classe mère. Le comportement obtenu était par conséquent :  
- B.m   
- B.m\_0   
- A.m   
- B.m\_0 – Java appelle par défaut la méthode de la classe mère.   
- A.m   
- B.m\_0   
etc.

Lors de l’exécution, la correction consistait alors, lors du switch, à forcer Java à utiliser la méthode de la classe en question.

Une première tentative a été menée. Celle-ci consistait à utiliser le mot clé java « this » devant l’appel des méthodes mais cela n’a eu aucune conséquence sur le comportement.

Une autre tentative a consisté à déclarer une variable du type de la classe dans laquelle le code est présent, et de « caster this » en une instance de cette classe. Par la suite, appeler la méthode sur cette variable.  
Ainsi, si nous sommes présents dans la classe A, le code est alors :

A a = (A) this ;  
a.m\_0();

Hélas, l’objet dans tous les cas reste de type B. Java appelle tout de même la méthode m\_0 de l’objet fille B et la boucle infinie est toujours présente.

Ce problème a remis en cause toute la structure du code généré.  
Faut-il changer la structure du code généré ?   
Tout ce qui a déjà été fait, doit-il être recommencé ?   
Comment faire en sorte de passer une méthode d’une version à l’autre désormais ?

Heureusement, une solution de contournement a tout de même été trouvée.  
Elle consiste à jouer sur la visibilité des méthodes en les déclarant « private ». Ainsi, à partir de la classe fille, la méthode de la classe mère ne peut pas être appelée. Java est alors obligé d’appeler la méthode de la classe en question et ne peut plus appeler la méthode de la classe fille.

Après avoir effectué ces modifications, les tests unitaires commençaient petit à petit à démarrer. Une autre exception Java fit son apparition, plus précisément, une exception qui avait visiblement déjà été traité. Celle-ci venait à nouveau d’un problème sur l’accessibilité des attributs via Reflections. Cette anomalie a été traitée en utilisant la réflexion afin d’autoriser l’accès aux attributs avant tout traitement de ceux-ci.

Les corrections apportées étaient toujours insuffisantes afin de faire fonctionner Code Rewinder sur Jsoup de façon satisfaisante.

Une autre anomalie a été détectée lors d’une relecture du code, celle-ci concernait le parcours des versions des méthodes. En effet, si Code Rewinder a transformé un projet pour 3 « commits », la version maximale de chaque méthode sera 2 (puisque cela générera les versions 0,1 et 2, ce qui correspond bien à 3 versions).

Exemple d’une génération d’attributs d’une méthode avec 3 « commits »

public Integer m\_version = 0 ;  
public Integer m\_version\_max = 2 ;

Cependant, lorsque Code Rewinder change les versions des méthodes, la condition du parcours était que la version devait être inférieure strictement à la version maximale, ce qui était erroné. Une mise à jour a donc bien été effectuée même si cela n’était pas bloquant (Code Rewinder ne se concentrait juste que sur un « commit » de moins que ce qu’il devait lors de l’exécution des tests unitaires et donc, du débogage).

Afin que les tests et les classes principales fonctionnent correctement, les fichiers « properties » et le contenu des dossiers « resources » devaient être déposés dans le dossier target automatiquement. Une analyse a donc été effectuée sur le projet Jsoup afin de voir et comprendre où et comment les fichiers devaient être placés. Des méthodes ont alors été mises en place afin de déplacer convenablement dans le target les fichiers.

Grâce aux corrections, Code Rewinder fonctionnait beaucoup mieux.  
Dans les logs JUnit, une des classes générait une exception de type « NoClassDefFoundError : could not initialize … »   
Cette exception semblait venir du fait que JUnit ne trouvait pas une classe dans le target (ou éventuellement, les dépendances de Jsoup). Plusieurs essais ont été réalisés afin de corriger ce problème :

* Vérification que la classe est bien présente dans le target.
* Récupération de l’URLClassLoader du thread en cours et ajout du classpath ainsi que du chemin du dossier target de force via la réflexion.
* Récupération de l’URLClassLoader de Reflections et même tentative.
* Récupération de l’URLClassLoader de JUnit, même tentative.
* Ajout des jars dans le classpath d’Eclipse.
* Ajout des jars dans le dossier target
* Analyse du code généré afin de déceler le moindre problème.
* Analyse des méthodes parcourues lors de l’exécution avant la transformation, et après la transformation afin de vérifier le chemin emprunté.

Après une petite semaine de persévérance, rien n’a fonctionné pour cette classe, qui était toujours introuvable pour JUnit quel que soit les tentatives de correction.

C’est alors qu’un cas dans l’application semblait ne pas être géré. Pour en être certain, une vérification dans le code généré a été effectuée.  
Pour ce qui était du switch, dans le cas où la méthode ne renvoyait aucune valeur, aucun break n’était mis en place ce qui provoquait pour plusieurs méthodes des appels non souhaités. Ainsi :

switch(m\_version) {

case 0 : m\_0() ;  
 case 1 : m\_1() ;  
 case 2 : m\_2() ;

}

Le code généré devait donc être :

switch(m\_version) {

case 0 :

m\_0() ;  
break ;

case 1 :

m\_1() ;

break ;

case 2 :

m\_2() ;

break ;

}

Uniquement dans le cas où la méthode ne renvoyait aucune valeur.

En réalité, la correction de ce cas non géré sur le switch a corrigé l’anomalie de la classe non trouvable par JUnit (qui ne pouvait pas être initialisée).   
Sur le projet Jsoup et sur la dernière version, tous les tests unitaires passent sans problème. Grâce à toutes ces corrections effectuées petit à petit sur Code Rewinder, les tests unitaires de Jsoup passent maintenant tous sans problème.

# Évaluation