

OPL – CODE REWINDER

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

Abstract : Code Rewinder is an automatic Java program bug fixer that uses unit tests, as well as Git version history. Its main objective is to offload the tedious manual debugging on developers. In its beta-beta version, it suffers from several bugs and limitations, making its use not trivial and not adapted to software of real size. This project aims to correct, improve and operate Code Rewinder on large size applications.

Résumé : Code Rewinder est un réparateur automatique de bugs de programmes Java qui utilise les tests unitaires, ainsi que l'historique des versions Git. Son objectif est principalement de décharger les développeurs de la tâche fastidieuse de débogage manuel. Dans sa version bêta-bêta, il souffre de plusieurs bugs et limitations rendant son utilisation non triviale et non adaptée aux logiciels de taille réelle. Ce projet vise à corriger, améliorer et faire fonctionner Code Rewinder sur des applications de grande taille.

WATTEBLED Étienne

03/01/2017

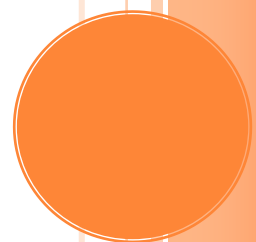


Table des matières

Introduction.....	2
Travail technique.....	4
Améliorations	4
Analyses et corrections.....	6
Exécution sur demoproject	6
Exécution sur jsoup	8
Évaluation.....	17
Tests unitaires	20
Tests du téléchargement.....	20
Tests des transformations.....	21
Tests du clean.....	22
Tests du déplacement des fichiers « ressources »	23
Critiques et limites	24
Conclusions	26
Lien GitHub	27

INTRODUCTION

Les logiciels développés de nos jours, notamment en Java, sont de plus en plus complexes et de tailles importantes. Ils s'inscrivent généralement dans un processus de développement collaboratif regroupant plusieurs développeurs, voire même plusieurs équipes.

Ainsi, pour gérer efficacement les diverses modifications et versions de développements, les développeurs modernes ont recours de façon quasi systématique à des plateformes de gestion de versions telles que GITHUB.

Cependant, même avec l'utilisation d'une telle plateforme, il reste à la charge des développeurs un certain nombre de tâches difficiles et consommatrice de temps afin de produire des logiciels fonctionnels et fiables. La correction de bugs est l'une des plus fastidieuses parmi toutes ces tâches, et son automatisation déchargerait les développeurs d'importants efforts. C'est d'ailleurs pour cela que les correcteurs automatiques de bug commencent à faire leurs apparitions ces dernières années dans le monde du développement logiciel.

Code Rewinder est un réparateur automatique de bugs de programmes Java qui utilise les tests unitaires (JUnit), ainsi que l'historique des versions git (n derniers commits) d'un projet pour parvenir à ses fins. Il a été développé dans sa version bêta-bêta, fonctionnant principalement en trois temps :

- Avant tout, il doit exécuter un batch permettant de télécharger les n dernières versions du projet à réparer.
- À l'aide de ces versions, il construit, en utilisant une librairie appelée Spoon, une nouvelle version du projet qui contient toutes les versions des méthodes. Pour chaque méthode, un switch sur un nouvel attribut static de la classe permet de passer une méthode d'une version à l'autre et un autre nouvel attribut permet d'obtenir la version maximale de la méthode. 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.

Cette version préliminaire de Code Rewinder présente un certain nombre d'erreurs et de bugs l'empêchant de fonctionner correctement, même sur des logiciels de tailles modestes.

Le but de ce projet est d'apporter, suite à un effort d'analyse, les modifications et améliorations nécessaires à Code Rewinder afin de lui permettre de fonctionner sur un véritable projet, et de taille réelle (Jsoup).

Avant tout, ce document présente quelques améliorations qui ont été apportées à Code Rewinder. Il détaille par la suite les corrections qui ont été apportées lors de l'exécution de Code Rewinder sur Jsoup.

La troisième partie présente la phase d'évaluation qui a eu lieu sur Jsoup afin de tester l'efficacité, les performances... etc. de Code Rewinder.

S'en suit, une section dédiée aux tests unitaires. Le chapitre suivant présente les limitations et éventuelles futures évolutions, avant de conclure le document.

TRAVAIL TECHNIQUE

AMÉLIORATIONS

Un certain nombre d'améliorations ont été apportées à Code Rewinder. Même s'il restait utilisable sans ces améliorations, elles le rendent notamment plus facile à utiliser, sa prise en main moins fastidieuse et élargissent son utilisation à d'autres systèmes d'exploitation notamment. Les améliorations apportées sont les suivantes :

1 – Un nombre significatif d'utilisateurs dispose d'un OS Windows. Malheureusement, le batch permettant de télécharger les n dernières versions d'un projet git était conçu, exclusivement, pour Linux. Un batch Windows a donc été mis en place afin de satisfaire un maximum d'utilisateurs. Tout comme le batch Linux, celui-ci récupère dans un premier temps la liste de tous les commits et la sauvegarde dans un fichier. Par la suite, des phases de « clone » ainsi que de « checkout » sur l'id du commit ont eu lieu. La difficulté principale a été le fait que la documentation pour créer un batch Windows est beaucoup plus limitée.

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 – 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 File.pathseparator, 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 des mots alors la classe est prise en compte, qu'il s'agisse d'une classe de test ou d'une classe du projet).

4 – L'utilisation de la classe StringBuilder au-delà de la concaténation de deux chaînes de caractères a été favorisée afin de gagner en performances. Le gain de performances est dû au fait que chaque concaténation de deux chaînes de caractères via le caractère « + » engendre un appel à un

constructeur de la classe `String`. La construction d'une instance prend un temps non négligeable dans le développement des applications.

5 – Mise en place d'un message de sortie pour indiquer à l'utilisateur le changement de méthode qui a permis le plus de tests « successful ». Aucune sortie n'était mise en place dans Code Rewinder, il était donc impossible de connaître la solution trouvée par Code Rewinder sans analyser dans la console toutes les tentatives effectuées par Code Rewinder.

6 – L'argument `-sourcePath` est désormais devenu `-sourceMainPath` et l'argument `-sourceTestPath` a été ajouté. Ces deux arguments sont désormais valorisés par défaut si aucune valeur n'a été spécifiée.

7 – Des constantes `targetMainPath` ainsi que `targetTestPath` (et non des arguments) ont été ajoutées.

8 – L'utilisation de la constante `File.separator` a été préconisée contrairement à celle du caractère « / » ou du « \ » afin que Code Rewinder fonctionne correctement quel que soit le système d'exploitation.

ANALYSES ET CORRECTIONS

EXÉCUTION 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.

L'un des premiers problèmes a été que, Reflections ne parvenait pas à « loader » les classes. Par conséquent, les tests unitaires ne pouvaient pas se lancer. Après avoir effectué plusieurs recherches, il s'est avéré que Reflections avait besoin d'avoir les classes compilées dans le dossier « target ». Une compilation des classes puis leur envoi automatique dans les dossiers « target/classes » et « target/test-classes » ont été mis en place. Les classes dans le dossier « target/test-classes » n'étaient, pour une raison inconnue, pas trouvées par Reflections. Des tentatives de modifications du classpath et des urls dans les ClassLoader ont été effectuées afin que Reflections puisse trouver les classes de test mais les classes étaient toujours invisibles dans le dossier « target/test-classes ». Les classes de tests ont, au final, été envoyées dans le dossier « target/classes » ce qui a permis de corriger le problème. Il est possible que Reflections ne puisse pas « loader » des classes qui sont en dehors du dossier « target/classes » afin de garantir une certaine sécurité (il pourrait être dangereux de pouvoir charger n'importe quelle classe de n'importe quel projet et de pouvoir modifier les valeurs de certains attributs).

Un autre problème rencontré par la suite est que 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 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 corrigée en créant à chaque fois un nouvel Integer et en passant le type primitif au constructeur de la classe Integer.

Grâce à l'ensemble de ces modifications, le projet demoprojet est devenu fonctionnel pour la première fois et fonctionnait correctement sur Code Rewinder.

EXÉCUTION SUR JSOUP

Après l'exécution réussie de Code Rewinder sur un projet de taille relativement modeste dans la section précédente, l'étape suivante est le passage à l'échelle à travers un projet de taille réelle et assez importante.

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 peut être ignoré sans répercussions par Code Rewinder lors de la compilation. Une méthode a été mise en place permettant en traitement préliminaire d'effacer tous les fichiers portant ce nom dans le projet (en utilisant un parcours en largeur). Le fichier est supprimé dans le projet déposé dans le dossier « ressources » par l'utilisateur mais le « repository git » reste inchangé. Le fichier n'est donc, pas « réellement » supprimé.

Une autre anomalie est survenue et concerne le changement de nom des paramètres d'une version d'une méthode à une autre.

Code Rewinder reprenait toujours la même signature d'une méthode pour toutes les versions. 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 en tenant compte des modifications de la signature des méthodes au fur et à mesure des versions.

C'est alors qu'un problème encore plus important 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 A {  
    public void m() {  
        ...  
    }  
}
```

```
public class B extends A {  
    public void m() {  
        ...  
        super.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 static Integer m_version = 0 ;  
    public static 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 static Integer m_version = 0 ;  
    public static 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 :

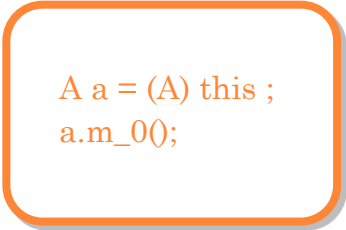
- `B.m`
 - `B.m_0`
 - `A.m`
 - `B.m_0` – Java appelle par défaut la méthode de la classe fille.
 - `A.m`
 - `B.m_0`
- etc.

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 même type que la classe sur laquelle la JVM est, puis, d'appeler la méthode sur cette variable.

Ainsi, dans la classe `A`, le code serait alors :



```
A a = (A) this ;
a.m_0();
```

Le comportement reste encore inchangé, car dans tous les cas, l'objet reste de type `B`. Java persiste à appeler 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. Ainsi il peut emmener à nous poser un certain nombre de questions importantes et avec des conséquences considérables sur le projet.

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, le changement total de la structure du code n'était pas indispensable et ce, grâce à une solution de contournement. Cette dernière 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 ayant visiblement déjà été traitée est apparue. Quelques attributs contenant la version et la version max, étaient à nouveau inaccessibles par Reflections. Cette anomalie a été traitée en utilisant Reflections pour les rendre accessibles avant toute manipulation 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).

```
public static Integer m_version = 0 ;  
public static Integer m_version_max = 2 ;
```

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

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).

Une autre anomalie, cette fois plus problématique, était que les numéros de version des méthodes, une fois incrémentés, n'étaient jamais remis à 0 avant de passer aux incréments des méthodes suivantes.

Afin que les tests et les classes principales fonctionnent correctement, les fichiers « propriétés » et le contenu des dossiers « ressources » 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 et dossiers devaient être placés. Des méthodes ont alors été mises en place afin de déplacer convenablement dans le target les dossiers et 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 semaine de persévérance, rien n'a fonctionné pour cette classe, qui était toujours introuvable pour JUnit quel que soient 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 ;  
}
```

Ce problème ne concernait que les méthodes qui ne renvoyaient aucune valeur, car en Java, le « return » est une opération bloquante.

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 apportées à Code Rewinder, les tests unitaires de Jsoup passent maintenant tous parfaitement et sans aucune erreur.

Un dernier bug a été détecté même s'il n'était pas bloquant et s'il n'empêchait pas Code Rewinder de fonctionner. En fait, il est possible d'avoir plusieurs

déclarations d'un même nom de méthode dans une même classe (mais avec des paramètres différents). Le problème était que, chaque attribut était créé par rapport au nom de méthode. Par conséquent, si un nom de méthode était présent plusieurs fois, un seul attribut était créé. Il était alors impossible de détecter réellement la méthode qui bogue dans ces cas-là.

Une façon simple de corriger l'anomalie aurait été de concaténer les noms des paramètres après le nom de la méthode, et de, par exemple, les séparer par un « _ ». Malheureusement, si le nombre de paramètres est le même, si les noms des paramètres sont les mêmes, et si les types sont différents (du point de vue de Java, cela est correct) alors le problème serait toujours d'actualité.

Par conséquent, au lieu de se baser sur les noms des paramètres, la correction apportée à Code Rewinder a été de se baser sur les types des paramètres. Cependant, certains caractères sont interdits dans les noms d'attributs mais sont présents dans le type (exemples : [] . , < > ? etc.) donc, un remplacement de ces caractères par le caractère « 0 » a été effectué. Il est alors ainsi possible par exemple de différencier « int[] » qui donnera « int00 » au type « int... » qui donnera « int000 » ou encore « char » qui donnera « char » et « char[] » qui deviendra « char00 ».

Une activation des transformations sur les classes « protected » ainsi que « private » a été faite.

Pour finir, des tentatives d'activation des transformations sur les méthodes « static » ont échoué. L'erreur « NoClassDefFound » est réapparue lors du « load » des classes et aucune solution n'a pu être trouvée à cause d'un manque de temps. Cependant :

- Un appel à la méthode « intValue() » a été testé sur les attributs afin d'être sûr que cela ne vient pas d'un problème de comparaison d'un « Integer » avec un « int » dans les switch.
- Une désactivation des transformations sur les fonctions « main » a été testée.
- Un passage de « private » à « public » a été testé uniquement pour les méthodes « static ».
- Un contrôle visuel et manuel de la classe qui pose problème (Tag.java) a été effectué afin de trouver la moindre anomalie ou le moindre cas particulier que possèderait cette classe (Qu'a-t-elle, de si spécial, que

les autres classes n'ont pas ?)

- Une correction afin que les méthodes « static » soient gérées a été effectuée sur Code Rewinder. Celle-ci a consisté à utiliser deux setters sur les objets Spoon CtInvocation et CtExecutableReference, afin de valoriser deux données supplémentaires pour que Spoon comprenne que la méthode est « static ».

Malheureusement la recherche n'a pas pu aboutir à un résultat à cause d'un manque de temps. Cependant, la transformation fonctionne tout de même pour les méthodes « static » et le code compile sans erreur. Il s'agit donc là, que d'une erreur de « loading ».

Les transformations des méthodes « static » ont donc été désactivées mais la correction a été conservée.

ÉVALUATION

Tous les tests unitaires de Jsoup sont « successful » ce qui laisse présager une certaine fiabilité. Néanmoins, dans un souci de performances une attention particulière a été accordée à l'évaluation dans ce projet. Afin d'effectuer cette l'évaluation, des bugs ont été injectés et une analyse du comportement de Code Rewinder a été effectué.

Des codes couleur ont été choisis pour représenter les différentes issues possibles de l'exécution de Code Rewinder :

- Vert : Code Rewinder a bien trouvé la méthode boguée et la bonne version de la méthode à utiliser.
- Jaune : Code Rewinder n'a pas trouvé la solution à cause du fait que le bug est présent dans un angle mort (une zone non couverte par les tests unitaires).
- Orange : L'anomalie est hors-périmètre de Code Rewinder (dans un cas non géré par Code Rewinder) et le bug est présent dans un angle mort.
- Rouge : L'anomalie est hors-périmètre de Code Rewinder.

L'idéal est donc que toutes les cases « Solution proposée par CodeRewinder » soient toutes vertes. Si quelques cases sont jaunes alors cela n'est pas très alarmant. En revanche, le rouge est vraiment à éviter et doit être géré en priorité dans de futures évolutions.

Voici le tableau regroupant les tests effectués afin de vérifier que Code Rewinder détecte bien les bugs (avec 3 commits sur Jsoup) :

Le bug				Code Rewinder
Classe	Méthode	Type de méthode	Version(s)	Solution trouvée
Cleaner	head(Node source, int depth)	Objet	0	
TreeBuilder	processStartTag(String name, Attributes attrs)	Objet	0	
Entities	codepointsForName(final String name, final int[] codepoints)	Static	0	
Document	outputSettings(OutputSettings outputSettings)	Objet	0	
NodeTraversor	traverse(Node root)	Objet	0	
QueryParser	parse()	Objet	0	
W3CDom	convert(org.jsoup.nodes.Document in, Document out)	Objet	0 et 1	
CharacterReader	advance()	Objet	0	
Tag	isInline()	Objet	0	
Tag	isInline()	Objet	0 et 1	
Cleaner	clean(Document dirtyDocument)	Objet	0	
Cleaner	clean(Document dirtyDocument)	Objet	0 et 1	
TokenQueue	matchesAny(String... seq)	Objet	0	
TokenQueue	matchesAny(char...seq)	Objet	0	
QueryParser	combinator(char combinator)	Objet	0	
Selector	select(String query, Element root)	Static	0	
Tokeniser	emit(Token token)	Objet	0	
Tokeniser	emit(final String str)	Objet	0	

Comme on peut constater sur le tableau précédent, il n'y pas de rouge sur la dernière colonne, ce qui est tout à fait rassurant sur les performances de Code Rewinder.

Voilà le tableau regroupant les temps d'exécution :

Nombre de commits	Temps
2	1m28
3	2m02
4	2m42

On constate sur ce tableau que le temps d'exécution se situe dans une fourchette raisonnable et assez intéressante sachant qu'il s'agit là de réparation automatique de bugs sur une application de taille réelle. En effet des temps situés entre 1 et 3 minutes seraient infiniment petits comparés aux temps que mettraient des développeurs lors d'un processus de débogage manuel.

À partir de 5 commits, Code Rewinder a cessé de fonctionner.

Le problème ne vient pas des transformations Spoon, mais d'un blocage au niveau du batch, donc, du téléchargement.

TESTS UNITAIRES

Les tests unitaires sont primordiaux et constituent une étape impérative dans le processus de vérification et validation de tout logiciel. Ils permettent d'éviter, principalement, qu'il y ait une quelconque régression.

Dans le but de vérifier Code Rewinder de la façon la plus large et complète possible, un maximum de tests unitaires (JUnit) a donc été mis en place.

Ces tests unitaires remplissent deux critères principaux :

- Ils sont le plus simples possibles.
- Ils doivent détecter un maximum d'anomalies tout en ayant un maximum de couverture.

L'importance de cette phase de test a fait qu'une importance particulière lui ait été accordée dans ce projet. Les tests ainsi développés couvrent l'ensemble des étapes principales de Code Rewinder.

TESTS DU TÉLÉCHARGEMENT

Une nouvelle classe de test « DownloadVersionsTest » a été créée afin de s'assurer que les différentes versions du projet se téléchargent bien.

Pour effectuer ces tests, les tests de cette classe utilisent le projet « demoproject » sur 2, puis 3 commits.

Afin de s'assurer que le téléchargement a bien eu lieu, il faut que les tests ne soient pas trop compliqués mais qu'ils couvrent un maximum d'anomalies.

Après réflexion, l'idée suivante a été retenue :

- On vérifie que les dossiers, qui contiennent chacun une version, ont bien été créés ou non après avoir lancé le batch.
Ainsi, si une version est téléchargée alors qu'elle ne devrait pas l'être ou inversement, les tests unitaires détecteront l'anomalie sans aucune difficulté.
- Pour ce qui est du contenu, une vérification de la taille des répertoires a été mise en place. Ainsi, si tous les répertoires sont tous identiques, ou si un seul possède une taille différente ou anormale, les tests unitaires détecteront l'anomalie.

Ainsi, le moindre petit octet de différence, ou le moindre changement d'un nom d'un dossier entraînera l'échec des tests unitaires. Ces tests sont simples à mettre en place, et surtout, efficaces.

TESTS DES TRANSFORMATIONS

La classe « TransformationsTest » permet de tester les transformations de code, c'est-à-dire, la création de la version du projet à réparer qui possède toutes les versions des méthodes, les tests unitaires sont plus complexes.

Une phase de réflexion a dû avoir lieu afin de trouver une solution pour que les tests soient performants, simples, mais très efficaces.

C'est alors que, l'utilisation de Reflections semblait la meilleure solution.

Grâce à Reflections, il a été alors possible de vérifier :

- La présence des attributs générés grâce à leur nom.
- La présence des méthodes générées grâce à leur nom et leur type de retour ainsi que leur visibilité qui doit être égale à « private ».
- Les valeurs des attributs qui contiennent la version max.
- La présence du corps des méthodes (contenu) : ce test consiste à changer la valeur de l'attribut qui contient la version d'une méthode et à vérifier les retours en utilisant la méthode « invoke » de

Reflections.

- Que le changement d'un nom de variable n'influe pas sur le fonctionnement de Code Rewinder.

Les tests sont alors très efficaces, et assez simples à mettre en place. Ils sont effectués sur un petit projet d'une seule classe, dédié uniquement pour ça et s'appelant « demotransformations ».

TESTS DU CLEAN

Certains fichiers sont bloquants pour la compilation mais ne sont pas indispensables. C'est le cas par exemple du fichier « package-info.java ».

Des tests unitaires ont été effectués afin de s'assurer que le fichier « package-info.java » est bien supprimé, qu'il soit présent à la racine, ou dans n'importe quel dossier du projet.

La vérification consiste à :

- Créer un dossier de test
- Le remplir de fichiers Java dont quelques-uns qui portent le nom de « package-info ».
- Lancer la phase de nettoyage.
- S'assurer que seuls les fichiers « package-info » ont été supprimés.

Le dossier est par la suite supprimé.

La classe possédant les tests de nettoyage s'appelle « ProcessCleanFilesTest ».

TESTS DU DÉPLACEMENT DES FICHIERS « RESSOURCES »

Une nouvelle classe de tests « ProcessResourcesFoldersTest » a été aussi créée afin de s'assurer que la méthode permettant de déplacer les données des dossiers « ressources » dans les dossiers « target/classes » et target/test-classes » fonctionne correctement.

Le test consiste donc à créer des dossiers et des fichiers dans des dossiers « ressources », et de vérifier que les données de ces derniers se retrouvent bien dans les dossiers « target/classes » et « target/test-classes » une fois la méthode de déplacement appelée.

CRITIQUES ET LIMITES

Code Rewinder a été validé sur le projet Jsoup et fonctionne correctement. Cependant Code Rewinder, dans sa version actuelle demeure encore perfectible. Il est en effet possible de lui apporter encore quelques modifications et améliorations qui lui permettraient de gérer encore plus de cas. Ces perspectives d'évolution de Code Rewinder devraient élargir le spectre des logiciels sur lesquels il pourrait réparer automatiquement les bugs.

Les prochaines étapes consisteront donc à :

- Faire en sorte que le « loading » des classes fonctionne aussi dans le cas où les transformations s'effectuent aussi sur des méthodes « static ».
- Faire en sorte que Code Rewinder gère les classes internes.
- Faire en sorte que Code Rewinder gère les constructeurs.
- Améliorer la sortie et éventuellement, faire en sorte que la modification se fasse directement dans le projet en entrée.
Il ne faudra pas oublier de remettre les fichiers qui ont été supprimé
- Il est possible qu'il n'y ait pas qu'une seule méthode qui bogue.
Au lieu de ne traiter qu'un seul attribut à la fois, il faudrait essayer de tester toutes les combinaisons possibles.
Cela risque cependant de mettre trop de temps, et si c'est le cas, il pourrait être possible par exemple de « verrouiller » un attribut, et de tester de modifier la valeur des autres. Si Code Rewinder ne parvient plus à avoir plus de tests unitaires « successful », il pourra s'arrêter.
- Essayer de corriger le fait Reflections ne trouve pas les classes présentes dans le « target/test-classes ».
Si cela n'est pas possible, une phase de simplification pourra être effectuée (il sera alors inutile de conserver le chemin vers le dossier « target/test-classes »)

- Il pourrait aussi être possible d'ajouter un argument afin de préciser le système utilisé ou avoir une détection automatique du système afin de choisir de façon appropriée le batch à utiliser.

Cette liste d'améliorations est bien entendu non exhaustive. Elle est constituée d'un ensemble de réflexions et de pistes qui ont semblé potentiellement pertinente, au fur et à mesure de l'avancement de ce projet.

CONCLUSIONS

Code Rewinder dans sa version bêta-bêta a été développé pour répondre à l'une des problématiques majeures des développeurs Java utilisant la plateforme de versioning GitHub. Il s'agit en effet d'une application de réparation de bugs basée sur les versions successives du logiciel en cours de développement.

Cependant, dans sa version bêta-bêta, Code Rewinder souffre d'une multitude d'erreurs et de bugs l'empêchant de s'exécuter correctement, même sur un logiciel de taille modeste. Ces erreurs ont toutes été corrigées dans le cadre de ce projet, pour aboutir à une version bêta fonctionnant parfaitement sur des exemples d'application de taille peu importante, mais aussi sur un logiciel de taille réelle et relativement grande comme JSOUP.

Dans le souci de rendre Code Rewinder plus facile d'utilisation, à la fois pour les utilisateurs et les développeurs, et pour étendre le périmètre de son utilisation au-delà du système Linux, plusieurs modifications et améliorations lui ont été apportées dans ce projet. Dans la version actuelle de Code Rewinder, toutes ces améliorations sont fonctionnelles et dépourvues de bugs et d'erreurs.

L'analyse des performances a montré que le nombre de commits influe sur le temps d'exécution. Cependant, les temps d'exécutions mesurés, de l'ordre de 2 minutes, restent très raisonnable au vu de l'apport précieux de la correction automatique de bugs par rapport à un débogage manuel.

La version de Code Rewinder à laquelle a abouti ce projet est tout à fait fonctionnelle sur des logiciels réels. Néanmoins, il est encore possible de lui apporter quelques améliorations afin de la rendre encore plus performante.

LIEN GITHUB

<https://github.com/Etienne-Wattebled/OPL3.git>