|  |
| --- |
|  |
| OPL - PathImpact |
|  |
| Analyseur automatique d’impacts de Code Source |

**Université Lille 1 – Sciences et Technologies**

13 février 2017

Créé par : Étienne WATTEBLED

OPL - PathImpact

Analyseur automatique d’impacts de Code Source

Table des matières

[Introduction 3](#_Toc473995121)

# Introduction

De jour en jour, afin d’améliorer la productivité ou l’efficacité, les métiers évoluent provoquant des besoins fonctionnels qui ne cessent d’évoluer même lorsque les applications sont opérationnelles et utilisées par les utilisateurs. Les applications possèdent, par conséquent, de plus en plus de fonctionnalités et deviennent de plus en plus complexes. Très vite, la maintenance devient alors très onéreuse, difficile à gérer et préoccupante. Il n’est pas rare qu’une ou plusieurs fonctionnalités soient modifiées, ou même supprimées durant le développement d’un logiciel, ce qui peut provoquer de multiples modifications qui se répercutent.

On appelle ces modifications des « impacts ».

Ces derniers peuvent devenir rapidement catastrophiques, de par l’apparition de régressions ou de code simplifiable voire mort qui peut persister pendant des années. De plus, à cause de la complexité et de la taille de certaines applications, certaines modifications peuvent s’avérer extrêmement coûteuses à mettre en place, si bien que, le client pourrait refuser de procéder à une évolution.

Il est alors indispensable et pouvoir évaluer avec précision, les impacts provoqués par une modification donnée d’une méthode, ne serait-ce que pour estimer le coup ou conserver le code sans parasites. Cependant, l’analyse manuelle peut s’avérer complexe et longue, et si l’évolution est annulée, ce temps utilisé aura été inutilement perdu.

Une solution existe : il existe des algorithmes capables de détecter automatiquement les impacts, plus précisément, de lister toutes les méthodes susceptibles d’être « touchées » par la modification d’une méthode. Ils sont utilisés pour faire des chiffrages, pour gagner du temps à mettre à jour une STD (Spécifications Techniques Détaillées), ou encore, lister avec précision les modifications à apporter avant de procéder au développement pour ne rien oublier.

Certains analyseurs d’impacts sont plus précis que d’autres mais les algorithmes permettant de parvenir à des résultats extrêmement précis sont très consommateurs en termes de puissance de calculs.

Ce rapport présente une grande partie de l’implémentation d’un analyseur d’impacts appelé « PathImpact » qui fait partie des analyseurs qui consomment le plus de ressources, mais il dispose d’une excellente précision. Seule la dernière étape permettant de déterminer les impacts n’est pas été traitée dans ce projet.

Avant tout, ce document explique comment fonctionne PathImpact et détaille par la suite son implémentation. La troisième partie concerne les tests unitaires JUnit, suivie d’un chapitre traitant l’évaluation. Et enfin, une phase de critiques et de limitations précède la conclusion.

# PathImpact

PathImpact est un analyseur d’impacts fonctionnant en trois étapes principales :

1. **Détermination et concaténation des stacktraces**
2. **Compression des stacktraces via un algorithme appelé Sequitur**
3. **Construction d’un arbre appelé DAG, et détermination des impacts.**

Ces étapes sont expliquées dans les parties ci-dessous.

## Stacktraces

L’algorithme PathImpact a besoin des différentes stacktraces en entrée pour déterminer les impacts.

Afin de les déterminer, il est nécessaire d’exécuter le programme dans tous ses états possibles, autrement dit, tester tous les cas de paramètres en entrée. Ainsi, si le programme a juste besoin d’un entier « n » pour fonctionner, il faut tester toutes les valeurs de « n » représentables (de Integer.MIN\_VALUE jusque Integer.MAX\_VALUE en Java).

La stacktrace est représentée par une collection ordonnée d’éléments représentant les méthodes.  
Ainsi, « A B » signifie « la méthode A appelle la méthode B ». Cependant, pour pouvoir faire la différence entre : « A appelle B, B retourne une valeur, puis A appelle C » et « A appelle B, puis B appelle C », il est nécessaire d’intégrer un nouvel élément que l’on appelle « r » et qui signifie « return ».  
Ainsi, « ABrCDrrr » signifie : « A appelle B, B retourne une valeur, A appelle C, C appelle D, D retourne une valeur, C retourne une valeur, puis A retourne une valeur ».

Toutes les stacktraces sont par la suite concaténées les unes après les autres. Par conséquent, un nouvel élément « x » est utilisé et signifie « fin du programme ».

En considérant l’exemple ci-dessous, la stacktrace générée doit alors être :

« main, m, r, r, x, main, m2, m3, r, m4, r, r, r, x »

public class Test {

public static void main(String args[]) {

int i = 0;

i = Integer.parseInt(args[i]) ;

if (i == 0) {

m();

} else {

m2();

}

}

public static void m() {

...

}

public static void m2() {

m3();

m4();

}

public static void m3() {

...

}

public static void m4() {

...

}

}

## Sequitur

Sequitur est un algorithme de compression utilisant la théorie des langages, plus précisément, la grammaire pour parvenir à ses fins.

Une grammaire est un ensemble de règles de production qui permet de générer les mots d’un langage.  
Toute grammaire possède une règle de départ (axiome) que l’on appelle la plupart du temps « S ».

Voici par exemple, la grammaire des expressions arithmétiques, dans laquelle l’axiome est « exp » :

exp → exp + exp | exp × exp | (exp) | num  
num → chiffre num | chiffre  
chiffre → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

**Wikipédia (Grammaire formelle)**

L’algorithme de Sequitur est basé sur deux propriétés fondamentales :

1. Il ne doit pas y avoir de doublons de digramme.  
   *Définition de digramme : assemblage de deux signes (généralement deux lettres de l’alphabet).*
2. Si une règle de production n’est plus utilisée ou n’est utilisée qu’une seule fois, elle doit être « désappliquée » puis supprimée.

Ainsi, la grammaire G définit par les règles :

S → aaaabb1p  
1 → cde  
2 → fgh

déroge les deux propriétés (1 et 2) car :

1. Le digramme « aa » est présent deux fois.  
   Solution : créer une nouvelle règle 3 → aa
2. La règle 1 n’est utilisée qu’une seule fois.  
   La règle 2 n’est pas du tout utilisée.  
   Solution : supprimer la règle 2 et « désappliquer » la règle 1.

La même grammaire respectant les deux propriétés est alors :

S → 33bbcdep  
3 → aa

Étant donné que la grammaire vide respecte les deux propriétés, celle-ci devient le point de départ de Sequitur et la règle de production « S » va grandir petit à petit.

Dès qu’une règle est dérogée, Sequitur va utiliser les solutions associées aux propriétés (ci-dessus) pour rectifier la grammaire.

Le pseudo algorithme de Sequitur est donc :

En paramètre, une simple liste l d’éléments

Créer une grammaire g vide.  
Ajouter le 1er élément de la liste l à la règle S de la grammaire g.

Pour chaque élément e de la liste l (à partir du 2ème élément)

Ajouter l’élément e dans la règle S de la grammaire (à la fin)

Tant que les deux derniers éléments forment un bigramme déjà rencontré

S’il existe déjà une règle r qui produit ces deux éléments

La récupérer

Sinon

Créer la règle r qui produit les deux derniers éléments de S

Fin si  
 Appliquer la règle r sur toute la grammaire

Supprimer toutes les règles qui ne sont pas utilisées au moins deux fois

Fin tant que

Fin pour