Réalisation De callGraph À L'aide De Spoon/GraphStream

Yassine Badache Ouamar Sais

17 novembre 2015
M2 IAGL

Sommaire

I - Introduction

L'exécution d'un programme peut être parfois difficile à cerner lorsque celui-ci devient complexe, notamment lorsque celle-ci ne tourne pas comme prévu. Cependant, peu de moyens sont mis en oeuvre pour faciliter la tâche des développeurs, que ce soit lors de la réalisation d'un programme conséquent ou d'un *debug* sur une application de grande ampleur: les Stacks Traces fournies par les différents IDE sont souvent précises mais peu lisibles tandis que les outils de *debug* permettent une avancée pas à pas au travers du programme, cependant fastidieuse.
Notre sujet concerne une solution qui pourrait concilier la lisibilité et la précision des deux approches. En effet, l'approche envisagée a été de constituer le *Call Graph*(1), aussi dit *graphe d'appels*, d'un projet, afin d'en avoir une visualisation à la fois graphique et suffisamment précise en termes d'informations pour pouvoir en tirer des conclusions, tant dans le *design*(2) de l'application que dans le *debugging* de celle-ci.
Cependant vient un point important à mentionner: parcourir l'intégralité d'un projet peut s'avérer très long, et très coûteux, rendant le *process* relativement inutile, car plus long que la démarche humaine de recherche d'une quelconque information.

Ce projet s'attelle donc dans un premier temps à la réalisation d'un *Call Graph* d'un projet complet à l'aide des outils offerts par Spoon(3) (voir partie 2) ainsi qu'une librarie nommée GraphStream(4), avant d'attaquer l'analyse du temps et de la consommation lors de la création, via une méthode simpliste de seuillage par nombre aléatoire.

II - Technologies

Afin de mener à bien la réalisation de ce projet, nous avons décider d'utiliser la technologie Java, pour une raison majeure: la librairie Spoon, écrite en Java, qui nous a grandement servi dans le parcours d'un projet et de l'analyse du code correspondant pour la construction du Call Graph.

2.1 - Spoon

La première librairie utilisée est Spoon, une solution *open-source* qui permet de transformer et analyser du code source Java. Ici, c'est la partie analyse qui va nous intéresser. A l'aide d'un processeur de code, il sera possible de récupérer tous les appels de méthode lors de l'exécution d'un programme afin de créer le *Call Graph* au fur et à mesure à l'aide de la seconde librairie, dénommée GraphStream.

2.2 - GraphStream

La seconde librairie utilisée est GraphStream, une librairie de construction et de visualisation de graphes. Nous l'utiliserons pour construire le graphe à partir des appels de méthode conférés par Spoon, afin de reproduire le plus fidèlement possible le *Call Graph* d'un projet conséquent.

Afin de tester notre solution en situation réelle, nous avons choisi un projet existant, contenant de nombreuses classes et un nombre conséquent d'appels, afin de pouvoir constater deux choses: l'utilité de notre démarche et les conclusions que nous pouvons tirer, tout d'abord directement, puis via une analyse rapide du résultat obtenu.

Le projet choisi est un projet nommé sobrement *"java"*(5), qui est en fait une application nommée *MarkShark*.

III - Generation du Call Graph

Pour la réalisation du graphe, nous avons implémenté un unique processeur Spoon s'appliquant aux méthodes appelées par le programme passé en paramètre. Pour récupérer les éléments correspondants à des appels de méthode, nous avons utilisé la méthode suivante :

arg0.getElements(new TypeFilter(CtInvocation.class))

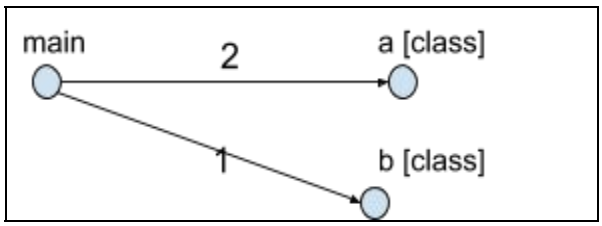
Cette méthode nous renvoie une liste d'objets CtInvocation correspondant à tous les appels de fonction de l'exécution du programme. A partir de celle-ci, il sera possible d'obtenir le parent, si il existe, de chaque méthode, c'est-à-dire la méthode à partir de laquelle la fonction courante à été appelée. Au parcours de chaque élément de la liste d'invocation, il sera donc possible de créer un nouveau lien *[fonction_parente]->[fonction_courante]* dans le graphe en créant les noeud ainsi que l'arête, ou alors d'incrémenter le compteur d'appel si ce cas d'appel s'est déjà produit auparavant.

La récupération de la méthode parente se fait comme suit :

inv.getParent(CtMethod.class)

En ce qui concerne le graphe, nous avons choisi d'appeler les noeuds par le nom simple de chaque méthode, et de fournir un compteur aux arêtes qui représentent le nombre de fois que le cas qu'elles représentent s'est produit.

L'image ci-dessous avec son code associé donne un exemple très simple de l'apparence d'un *Call Graph* :



Code correspondant:

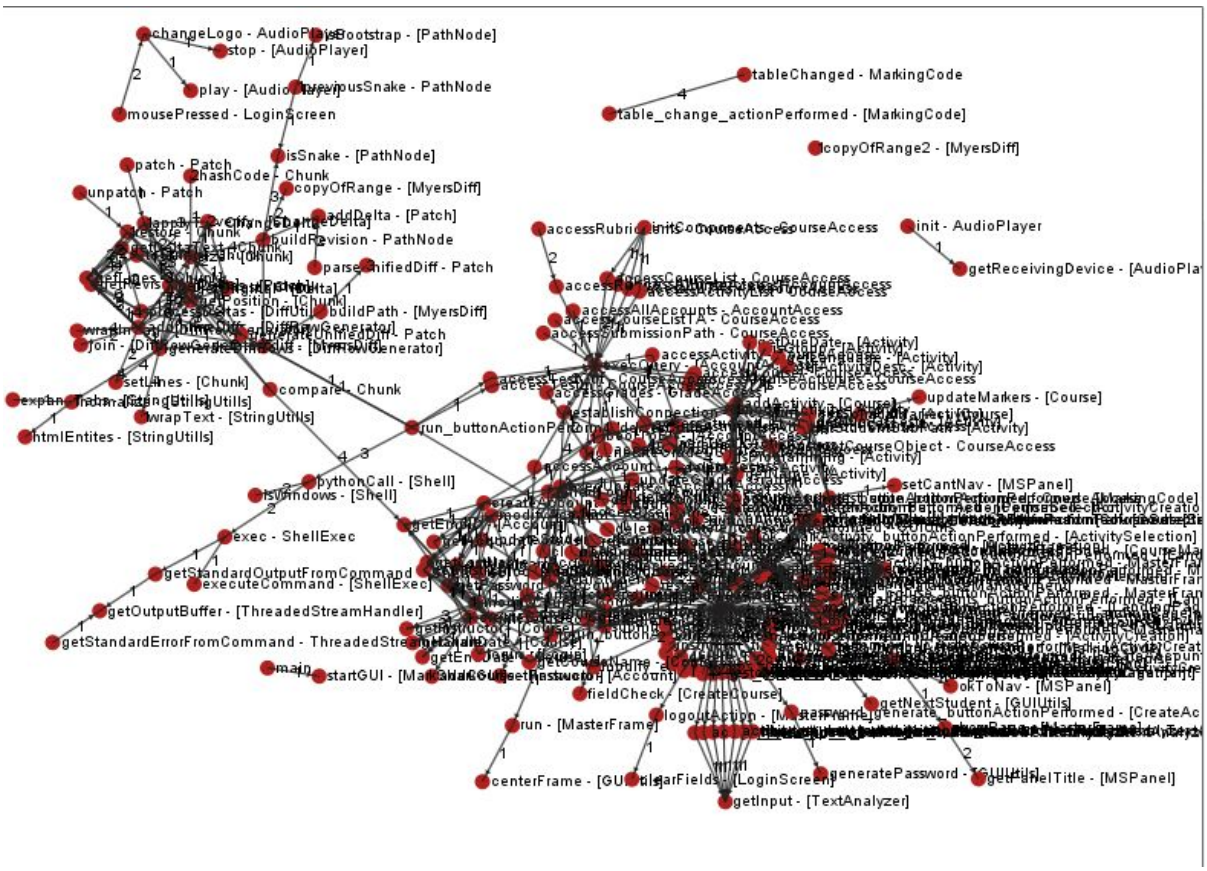
```
public static void main(){
  a();
  a();
  b();
}
```

IV - Méthode de seuillage simplifiée

Afin de mesurer l'impact de la création d'un *Call Graph* sur un projet de bonne taille, nous avons décider d'opter pour une approche de seuilage simpliste par nombre aléatoire.

Tout d'abord, un seuil entre 0 et 1 est choisi. Ensuite, la démarche consiste à tirer un nombre aléatoire pour chaque méthode récupérée par le processeur Spoon, et de le comparer: si il est inférieur -ou supérieur, arbitrairement-, nous effectuons tout le traitement inhérent à la création d'un noeud ou d'une arête dans le graphe. Autrement, nous passons simplement la méthode concernée. Ceci nous permet d'observer le comportement des deux extrêmes, et d'en déduire la pertinence d'une telle approche.

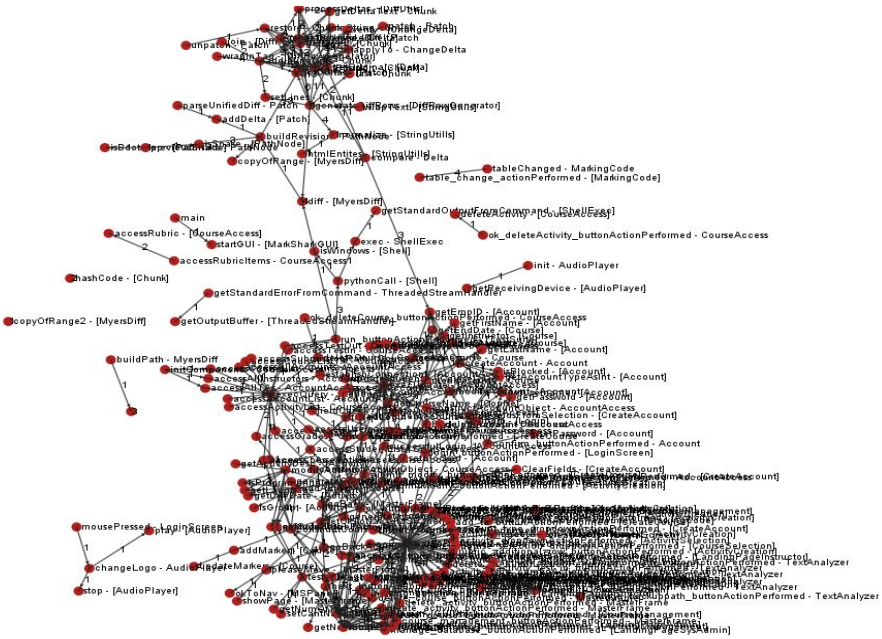
Pour rappel, avec un seuil inexistant, le *Call Graph* obtenu à l'exécution de notre programme donnait le résultat suivant:



Call Graph complet de l'application MarkShark.

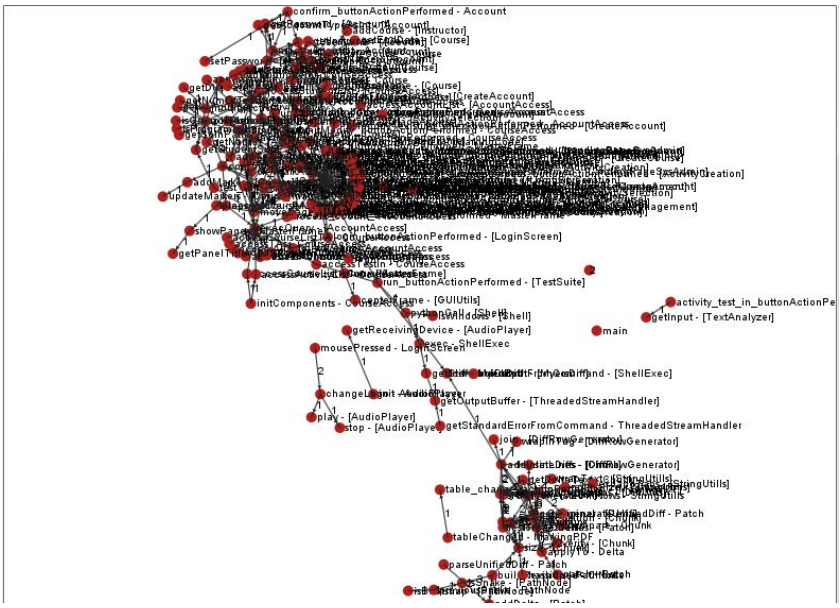
De tels résultats nous éclairent directement sur la nécessité d'un seuillage, bien que quelques informations soient déjà repérables à l'oeil nu dans ce premier graphe.
Voici les résultats collectés avec différents seuils choisis par nos soins.

Seuil: 0.2.
Temps de création: 05,39sec.



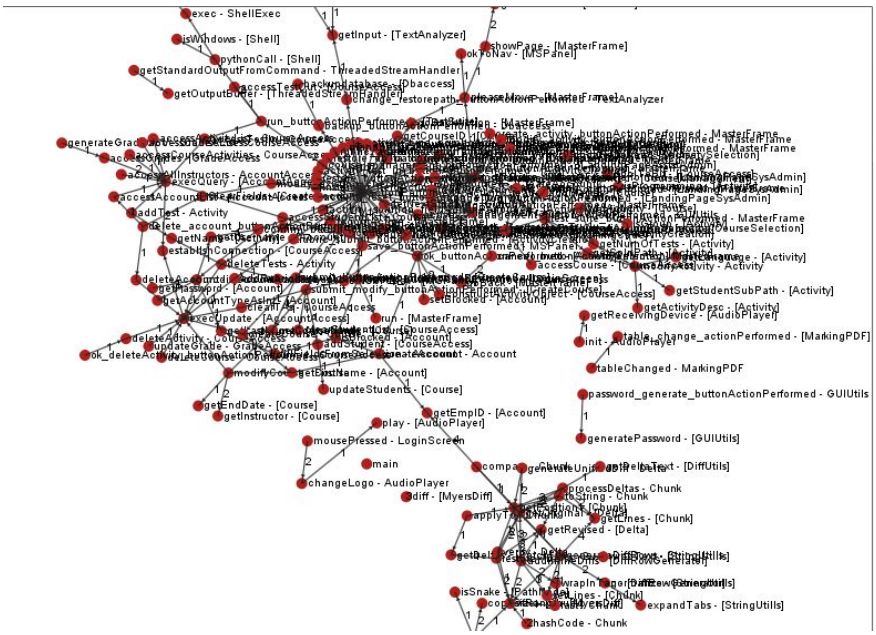
Le graphe est pratiquement complet

Seuil: 0.4.
Temps de création: 05,28s.



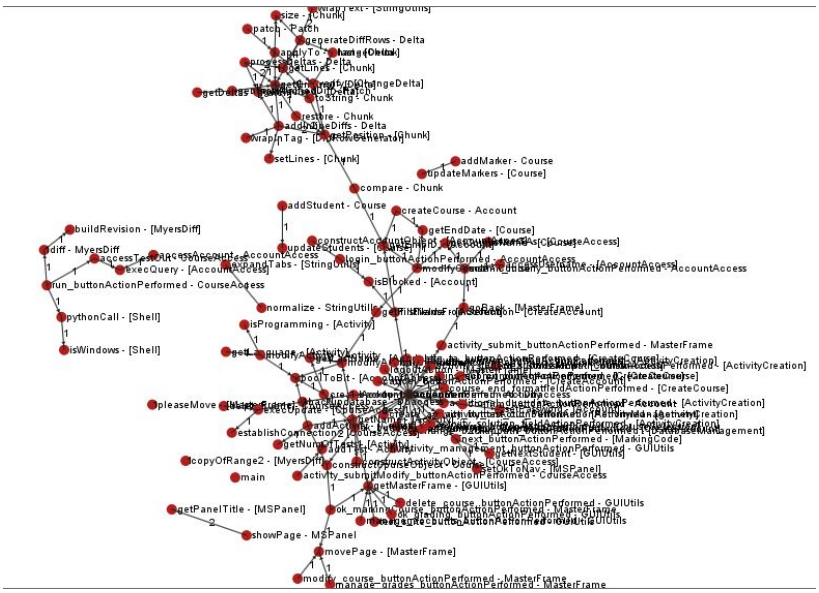
Le graphe ne perd pas en contenance.

Seuil: 0.6.
Temps de création: 05,31s.



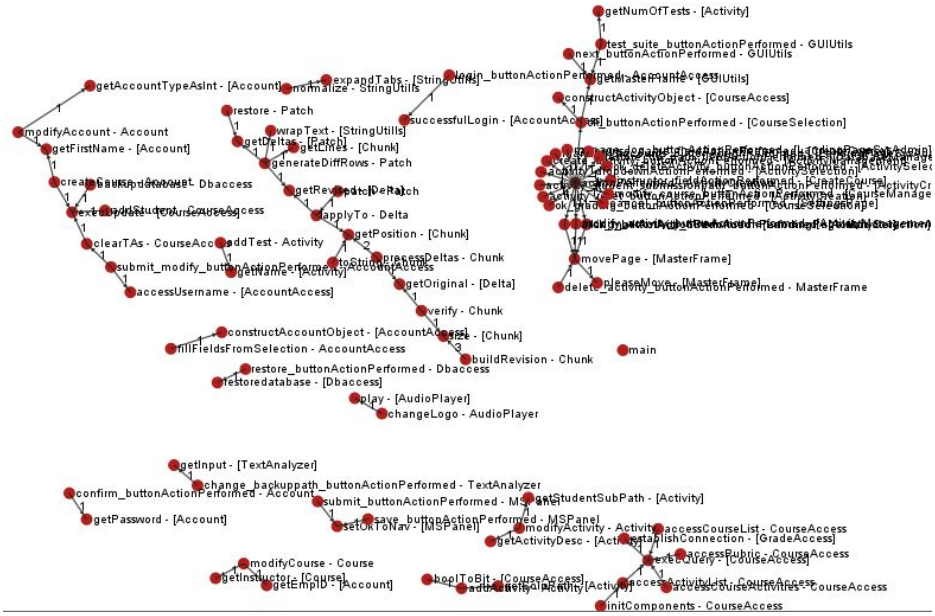
Le graphe se désemplit un petit peu, mais pas autant qu'escompté.

Seuil: 0.8.
Temps de création: 5,83s.



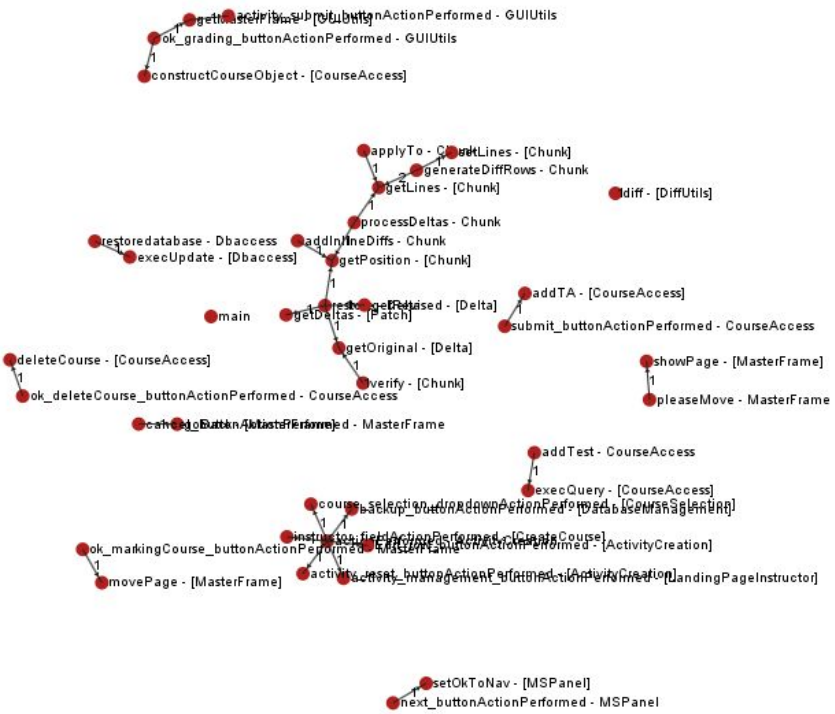
Le Call Graph commence seulement à se désemplir.

Seuil: 0.9.
Temps de création: 5,49s.



Le graphe s'est bien désempli et nous permet une identification des fonctions principales.

Seuil: 0.95.
Temps de création: 5,31s.



Clair, mais avec très peu de représentation sur la vraie ampleur du projet.

Ces résultats nous permettent de constater plusieurs choses. Premièrement, et c'est une erreur de notre part, notre système ne nous permet pas d'évaluer de manière significative les temps de performance. Après analyse, nous nous rendons compte de l'erreur, qui se trouve dans le code même: la récupération dans un premier temps de toutes les méthodes dans une liste nullifie l'impact que pourrait avoir la création d'un graphe, dans la mesure où de toute manière il traite la même quantité de données.

La seconde analyse nous permet en comparant les seuils d'identifier les méthodes dites *godly*, ou les *superméthodes*. En effet, la plus voyante est la node nommée *ActionPerformed* (ça n'est pas très lisible mais c'est bien elle). Elle apparaît quel que soit le seuil et est toujours abondamment reliée par d'autres. On en conclut que c'est une méthode très importante, et que son maintien et sa solidité sont indispensables. Cette démarche nous permet, comme nous pouvons le voir, d'étudier un programme afin d'en identifier les points-clés et, au contraire, les points auxiliaires ou très peu utilisés. C'est également un bon moyen d'étudier le *design* d'une application, afin d'observer les cas de dépendances trop nombreuses à une classe, de *superméthodes* ou encore de méthodes inutilisées, de dépendances redondantes, etc …

La troisième analyse concerne le seuil choisi. En effet, on constate qu'un seuil compris entre 0 et 0.5 n'offre pas de grands changements visuellement, ils n'apparaissent même qu'à partir d'un seuil de 0.8. Ceci est dû au fait qu'une application passe bien souvent plus d'uns fois par le même chemin, et que celui-ci peut être ré-emprunté lors d'une itération future. On peut en conclure qu'un seuil bas peut être pratique pour évaluer la dimension globale d'une application, tandis qu'un seuil plus élevé peut être pratique pour observer le comportement global d'une application et son fonctionnement général.

V - Conclusion

Ce projet nous a permis de manipuler deux libraires aux usages très différents, et de les combiner pour obtenir un résultat des plus intéressants. Nous avons pu mettre en avant l'utilité d'un tel procédé et en tirer des conclusions sur ses capacités en terme d'interprétation de code. Comme *take-away*, nous dirions que cette démarche a de l'avenir dans le monde du *debug* et de l'analyse de projet de grande ampleur, mais méritent d'être creusées plus en avant et d'être améliorées davantage.

VI - Références

- *Call Graph:* https://en.wikipedia.org/wiki/Call_graph

- *Design:* https://en.wikipedia.org/wiki/Software_design_pattern

- *Spoon*: https://github.com/INRIA/spoon

- *GraphStream*: graphstream-project.org

- *Projet de test 'java'*: https://github.com/Donnervoegel/java