Université Lille 1

Rapport OPL

Pull Request Engineering

Optimisation de l'affichage des Pull Requests

Auteurs :
Franck Warlouzet
Hedi Mokhtar

Responsable :
Martin Monperrus

10 octobre 2016

Table des matières

Introduction

Nous avons aujourd'hui l'opportunité d'effectuer auprès d'un auteur de projet open source une pull request, et ainsi lui proposer des modifications sur son code. Cet auteur décidera ensuite d'accepter ou non ces modifications après les avoir prise en compte.

Lorsque des modifications parviennent jusqu'à l'auteur, l'ordre des éléments qui ont été modifiés s'affiche de manière alphabétique. Cela ne nous semble clairement pas le plus adapté à une relecture efficace de code source, celui ci n'étant que très rarement organisé sous forme alphabétique. Nous pouvons alors nous demander s'il existe un meilleur moyen d'afficher ces modifications aux yeux du créateur pour optimiser ses observations et accélérer le processus de validation ou rejet de pull request [1].

Ce que nous souhaitons transmettre avec ce rapport est de montrer qu'il y a des moyens pour afficher les fichiers modifiés au sein d'un pull request de manière plus intéressante et réfléchie que la façon originelle proposée par GitHub [2].

Pour atteindre cet objectif nous avons mis en place un système qui récupère les pull requests d'un projet et qui réorganise l'ordre des fichiers qui ont été altérés en se basant pour cela sur leur importance. C'est à dire que le relecteur aura à la découverte de la pull request, directement sous les yeux les fichiers qui nécessitent davantage d'attention et dont les modifications peuvent avoir un impact plus important sur le reste du code. Les fichiers à l'importance moindre seront quant à eux placés plus en retrait car leur relecture n'est pas capitale et demande moins d'attention.

Nous utiliserons des repositories [3] GitHub. témoins sur lesquels nous avons réalisé des pull requests afin de contrôler leur contenu et d'être conscient de l'importance ou non des classes modifiées pour pouvoir évaluer notre travail.

1. Action de proposer des modifications de code à l'auteur d'un projet
2. Service web d'hébergement et de gestion de développement de logiciels
3. Dépot sûr et protégé où l'on peut stocker n'importe quel type de fichier

Chapitre 1

Travail technique

1.1 Principes

Le principe est d'évaluer la valeur d'une classe en Java d'un projet et ensuite afficher ces différentes classes par ordre de valeurs. Evaluer une classe en Java se fera par plusieurs approches comme évaluer la taille du code source, rechercher des mots clés ou encore déterminer une dépendance de classe. Nous avons décidé de limiter le scope de l'application au langage Java étant donné les délais dans lesquels celle-ci devait être réalisée. Il est envisageable dans le futur d'étendre notre application à tous les langages, à commencer par les langages orientés objets, car les patterns [1] d'analyse sont sensiblement les mêmes que l'on soit en Java, Smalltalk ou encore C#. En effet, notre analyse portant sur l'héritage, la taille des classes, les références d'une classe dans le reste du code source, nous ne sommes pas bloqués à un langage précis. En revanche le fait d'utiliser Spoon pour réaliser notre analyse de code et notre implémentation en général nous limite à l'étude du code Java.

Ensuite pour afficher notre travail aux yeux du relecteur, nous avons choisi de conserver le code couleur actuel de GitHub, à savoir, fond de code source vert pour un ajout et rouge pour une suppression. Ce code couleur est connu de tous et il aurait été déroutant de le modifier. Nous avons également fait le choix de grossir le texte des lignes modifiées dans le cas d'une modification partielle d'un fichier afin de faire rapidement ressortir le contenu changé. Le but est de faire gagner du temps à la personne qui va valider ou non la pull request. En effet, un code couleur clair et des changements habilement mis en avant facilitent la lecture et l'approbation ou non de ces changements.

1. Modèle spécifique représentant d'une façon schématique la structure d'un comportement individuel ou collectif

1.2 Réalisation

Afin d'implémenter notre solution nous avons eu plusieurs idées. Tout d'abord pour démarquer les classes entre elles, trouver une notion de dépendance semble être intéressant, si une classe est héritée par beaucoup d'autres alors il semble naturel qu'elle soit plus importante. Ensuite plus une classe a d'ampleur en terme de taille plus elle a de chance d'être importante aussi. Tous ces critères nous permettent de calculer un score pour chaque classe, qui détermine son importance dans le projet. Par exemple une interface dont hérite la moitié du code du projet a une importance énorme par rapport à une implémentation en bout de chaîne qui n'est utilisée que localement dans le projet.

Concernant l'affichage, nous avons opté pour une simple page web dans laquelle l'utilisateur entre le nom du repository GitHub pour lequel il y a des pull requests et qui contient du code Java. S'en suit le traitement puis un affichage sous forme de liste des pull requests ouvertes pour ce repository dont les classes modifiées dans la pull request sont triées selon leur score.

Les repositories que nous avons utilisés pour les tests sont les suivants :

— `https://github.com/sallareznov/snake`
— `https://github.com/FranckW/OPLTest`

Nous avons crée des pull requests dont nous connaissions le contenu afin de pouvoir juger de la pertinence des scores calculés. Plus tard, une fois l'application terminée, nous avons testé notre application sur des repositories inconnus plus gros.

1.3 Architecture et design

1.3.1 Langages

Pour le traitement back-end le langage Java a été utilisé, ce qui est en effet plus simple pour analyser du code Java grâce à la librairie Spoon. Cette librairie nous permet d'avoir facilement accès à toutes les références d'une classe dans le projet, ainsi que ses sous classes, super classes, interfaces implémentées, mais aussi le nombre de ses méthodes. De plus, nous maîtrisons Java, cela nous évitait de nous mettre une barrière technologique.

Côté front-end, nous avons opté pour AngularJS. Maîtrisant cette technologie, nous pouvions vite arriver à une page web pour afficher nos résultats de manière claire, nous laissant ainsi le champ libre pour nous concentrer sur la partie analyse et calcul des scores.

Ces deux parties communiquent via une API REST écrite en Java. Nous utilisons Ajax coté JavaScript pour communiquer avec l'API GitHub de manière asynchrone, cela permet ainsi d'accélérer le traitement des requêtes qui peuvent être lourdes lorsqu'il s'agit d'un projet volumineux qui comporte énormément de pull requests.

1.3.2 Librairies

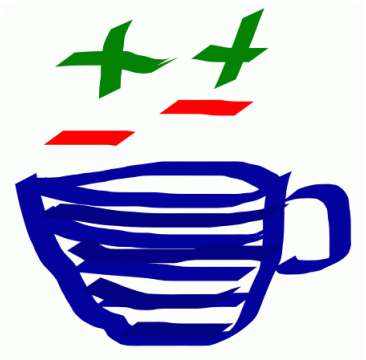
Voici la liste des différentes librairies utilisées prendant notre projet :

La librairie d'analyse et de transformation de code source Java Spoon. Cette librairie permet de transformer et d'analyser le code Java avec la granularité que l'on choisit, cela peut être très large comme la définition de la classe elle-même, jusqu'à très précis comme un bloc try catch au sein d'une méthode.

API GitHub pour récupérer les pull requests d'un repository, la liste et le contenu des fichiers modifiés, ainsi que les modifications faites par la Pull Request.

JGit, librairie Java permettant de communiquer avec Git depuis Java. Nous utilisons cette librairie uniquement pour cloner [2] les repositories avant de procéder à l'analyse du code Java de ceux-ci.

2. Action de copier entièrement un repository



La librairie rdash-angular a été utilisée afin de ne pas perdre de temps sur le style de la page web, il s'agit d'un template html, dont nous n'avons pas gardé grand chose.

google-code-prettify a été utilisée pour réaliser la colorisation du code source sur la page web. `https://github.com/google/code-prettify`

1.3.3 Design

Nous avons implémenté le calcul du score des classes à l'aide d'une map statique qui contient la CtClass (voir doc de Spoon pour le détail) en clé et son score en valeur. Nous interrogeons ensuite cette map avec les appels REST pour connaître le score de chaque classe concernée dans la pull request et ainsi l'afficher à sa place dans la liste.

1.4 Screenshots

Voici un screenshot montrant la manière dont on a conservé le code couleur de GitHub.

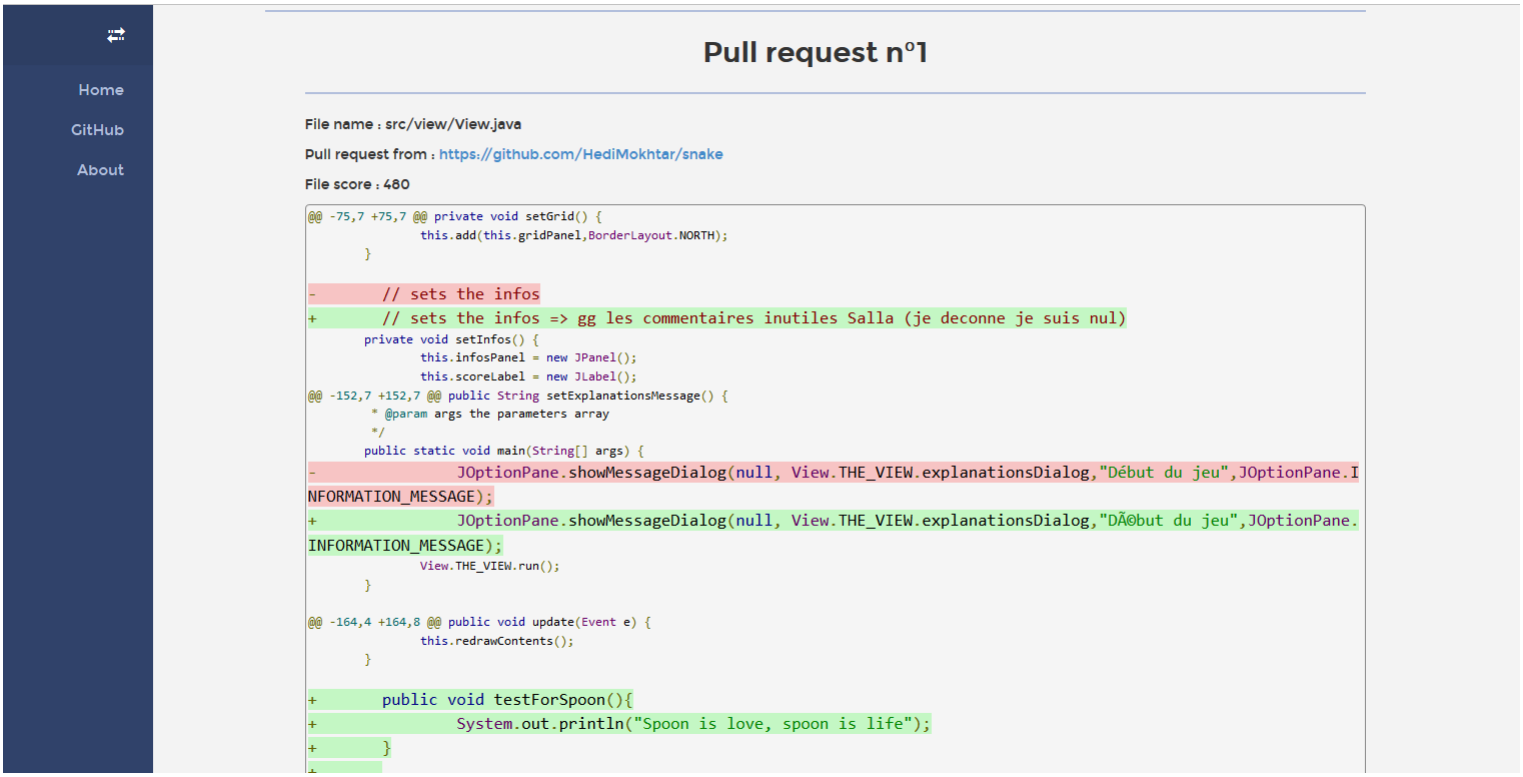


FIGURE 1.1 – Exemple de modification de code

Et un autre screenshot à la page suivante montrant que les classes ne sont plus classées par ordre alphabétique mais par ordre de pertinence que l'on a nous même calculée, la première classe qui apparaitra sera donc celle qui aura la plus grosse valeur selon notre algorithme.

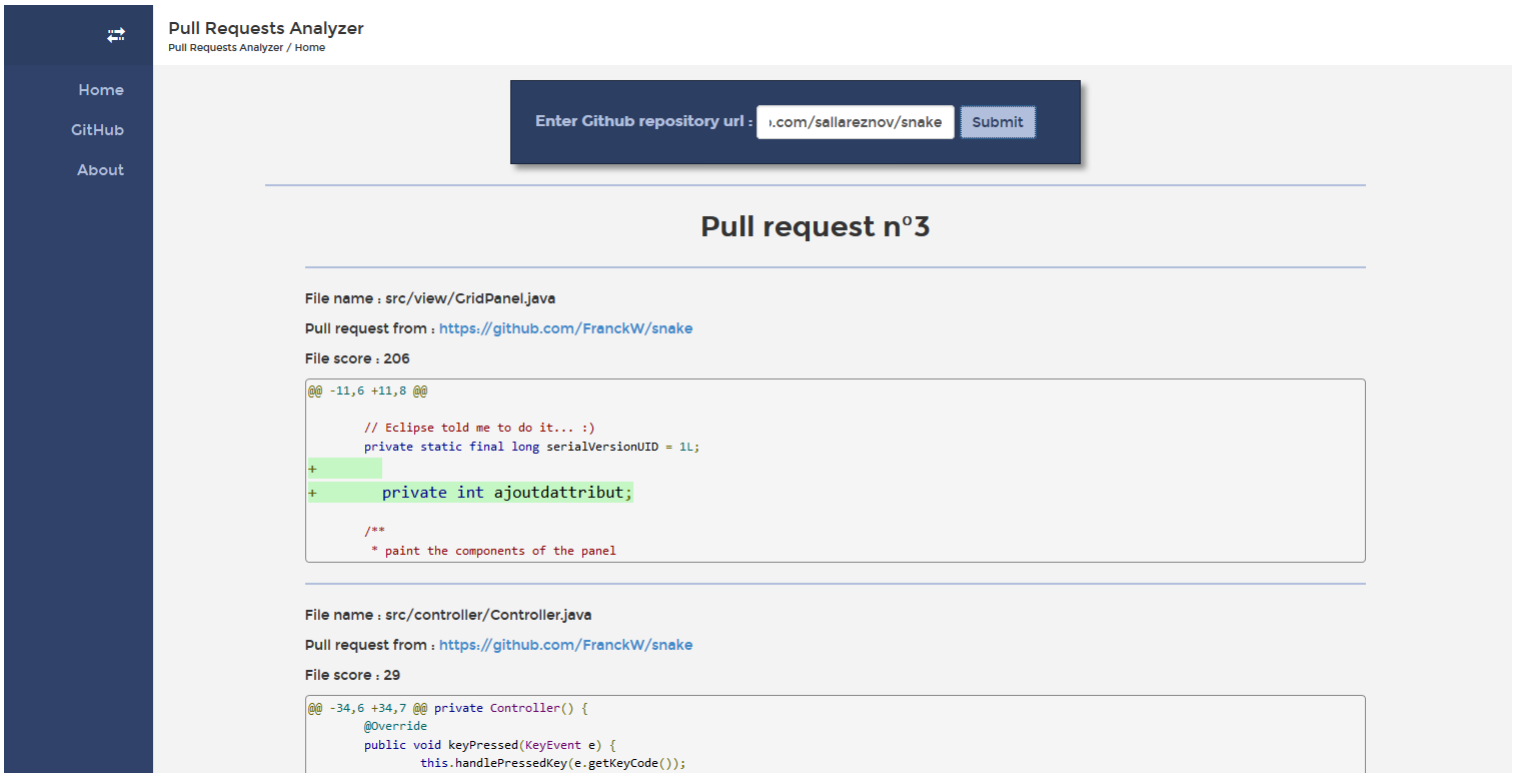


FIGURE 1.2 – Exemple de classement de classes par leur score

Chapitre 2

Evaluation

2.1 Pertinence

Après avoir réalisé notre analyse sur plusieurs repositories connus, nous avons pu juger de la pertinence de nos résultats. Il en ressort que le classement obtenu est cohérent. Nous avons les interfaces fortement utilisées via héritage ou référence directe avec un score élevé, les longues classes référencées à plusieurs endroits dans le projet avec un score moyen et les classes ne servant pas à grand chose avec un petit score, voire un score minimal pour les classes vraiment basiques.

2.2 Temps

Le bémol dans notre analyse est le temps d'exécution. En effet, notre façon de faire impose de cloner tous les forks [1] du repository GitHub qui ont soumis une pull request pour analyser le code Java à l'intérieur et scorer chaque classe. L'analyse en tant que telle avec Spoon est rapide même sur un gros projet, en revanche, selon la connexion dont dispose l'utilisateur, les différents clonages peuvent être longs.

On présente dans le tableau suivant le temps d'execution de l'analyseur Java sur quelques repositories, chaque temps du tableau est la moyenne de cent essais en millisecondes.

1. Copie entière du repository d'un auteur qui permet par la suite d'effectuer des pull requests

Élément testé	Temps d'execution
Projet petit (549 octets)	1718 ms
Projet moyen (1,03 Mo)	4230 ms

TABLE 2.1 – Temps d'execution de l'analyseur Java sur différents projets.

2.3 Mémoire

L'impact mémoire de notre application est faible, nous ne gardons rien en cache. Par contre nous utilisons de l'espace disque lors des clonages des différents forks, puis avec l'analyse de Spoon, qui elle aussi écrit sur le disque. La mémoire utilisée s'exprime par la relation suivant :

Taille du code de notre analyseur + taille du repository à tester

2.4 Limitations

Lors de nos manipulations de tests pour s'assurer de l'efficacité de notre application, nous avons été confronté à des soucis concernant la prise en main de Spoon. En effet, plusieurs fois, sans que nous l'expliquions, l'analyse lancée par Spoon plantait en cours de route sans que l'on trouve pourquoi. C'est pourquoi nous n'avons pas pu tester notre application sur des plus gros projets, comme Spoon lui même étant écrit en Java et ayant plusieurs pull requests sur son repository. De plus, comme décrit dans la section précédente, si l'utilisateur dispose d'une connexion très limitée, le clonage de tous les forks risque d'être extrêmement long, voire il sera face à des timeouts.

Conclusion

Notre projet présente l'ébauche d'une solution qui peut être très largement et facilement étendue afin de répondre à la question initiale : Peut on ordonner les modifications apportées par une pull request de manière à classer son contenu par importance ? Cependant il y a quelques contraintes comme le temps d'execution non negligeable pour copier les repositories et le fait que certains projets déclenchent des exceptions quand Spoon les parcourt.

Notre analyseur permet de bien identifier l'importance des relations entre les classes en leur donnant une valeur, et notre affichage permet de bien montrer à l'utilisateur les modifications apportées par les pull requests. Nous proposons donc une solution qui permet d'accélérer la validation ou non des pull requests par l'intégrateur d'un projet open source.

D'autres fonctionnalités pourraient être ajoutées, comme s'attaquer à d'autres types de langages ou bien afiner l'analyse et le calcul de l'importance d'un fichier.