Université de Montréal

Étude empirique des commentaires et application des techniques de résumé par extraction pour la redocumentation

par
Dorsaf Haouari

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences - Université de Montréal

Mémoire présenté à la Faculté des arts et des sciences
en vue de l'obtention du grade de Maître ès sciences (M.Sc)
en informatique

Août, 2011

Université de Montréal
Faculté des arts et des sciences

Ce mémoire intitulé :

Étude empirique des commentaires et application des techniques de résumé par extraction pour la redocumentation

Présentée par :
Dorsaf Haouari

a été évaluée par un jury composé des personnes suivantes :

Guy Lapalme, président-rapporteur
Houari Sahraoui, directeur de recherche
Philippe Langlais, co-directeur
Bruno Dufour, membre du jury

RÉSUMÉ

La documentation des programmes aide les développeurs à mieux comprendre le code source pendant les tâches de maintenance. Toutefois, la documentation n'est pas toujours disponible ou elle peut être de mauvaise qualité. Le recours à la redocumentation s'avère ainsi nécessaire.

Dans ce contexte, nous proposons de faire la redocumentation en générant des commentaires par application de techniques de résumé par extraction.

Pour mener à bien cette tâche, nous avons commencé par faire une étude empirique pour étudier les aspects quantitatifs et qualitatifs des commentaires. En particulier, nous nous sommes intéressés à l'étude de la distribution des commentaires par rapport aux différents types d'instructions et à la fréquence de documentation de chaque type. Aussi, nous avons proposé une taxonomie de commentaires pour classer les commentaires selon leur contenu et leur qualité.

Suite aux résultats de l'étude empirique, nous avons décidé de résumer les classes Java par extraction des commentaires des méthodes/constructeurs. Nous avons défini plusieurs heuristiques pour déterminer les commentaires les plus pertinents à l'extraction. Ensuite, nous avons appliqué ces heuristiques sur les classes Java de trois projets pour en générer les résumés. Enfin, nous avons comparé les résumés produits (les commentaires produits) à des résumés références (les commentaires originaux) en utilisant la métrique ROUGE.

Mots-clés : **Redocumentation, pertinence des commentaires, résumé automatique.**

ABSTRACT

Programs documentation is very useful to programmers during maintenance tasks, especially for program comprehension. However, the documentation is not always available or it may be badly written. In such cases, redocumentation becomes so necessary.

In this work, we propose a redocumentation technique that consists in generating comments by using extraction summary techniques.

For this purpose, we conducted an empirical study to analyze the quantitave and qualitative aspects of comments. Particularly, we were interested in studying comment distribution over different types of construct and studying the frequency of documentation for each construct type. We propose a comment taxonomy to classify them according to their content and quality.

Given the results of the empirical study, we decided to summarize Java classes by extracting the comments of methods and constructors. We defined several heuristics in order to determine the most relevant comments to be extracted. After that, we applied these heuristics to Java classes from three projects in order to generate summaries. Finally, we compared the generated summaries (generated comments) to the reference ones (original comments) by using the metric ROUGE.

Keywords : **Redocumentation, comments relevance, automatic summary.**

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

À mes très chers parents qui ont toujours été là pour moi, et qui m'ont donné un magnifique modèle de labeur et de persévérance.

À ma sœur Rym et à sa petite famille, pour leur soutien et leur encouragement à aller toujours en avant.

À mon frère Walid, pour son encouragement et son soutien moral.

À Raja, pour les bons moments que nous avons passé ensemble.

À Tous mes ami(es) et collègues qui m'ont soutenu et m'ont comblé par leurs conseils pour la réalisation de ce travail.

Et enfin à tous ceux qui me sont chers,

Je dis ***Merci***

REMERCIEMENTS

Je tiens à remercier tout d'abord mon directeur de recherche Dr Houari Sahraoui de m'avoir enseigné les bases de mes connaissances dans le domaine du génie logiciel et de m'avoir ensuite dirigée dans ma recherche. Je tiens également à lui exprimer ma profonde reconnaissance pour sa disponibilité, sa confiance et son assistance technique et morale.

Je remercie également mon co-directeur, Dr Philippe Langlais, pour la pertinence de ses orientations ainsi que pour la grande disponibilité dont il a fait preuve tout au long du déroulement de ce mémoire. Qu'il trouve dans ces quelques lignes l'expression de mon profond respect et de ma réelle gratitude.

J'adresse pareillement mes sincères remerciements au Dr Guy Lapalme, pour l'honneur qu'il nous a fait en présidant notre jury de mémoire de maîtrise.

Je tiens aussi à remercier vivement Dr Bruno Dufour d'avoir bien voulu apporter ses conseils et son jugement pour ce modeste travail.

Je remercie également les membres du laboratoire GÉODES, Hajar, Martin, Arbi, Aymen, Jamel ainsi que tous les autres avec lesquels j'ai passé d'agréables moments.

J'exprime également ma gratitude à tous mes enseignants qui ont contribué chacun dans son domaine à ma formation universitaire, sans laquelle je ne serais jamais arrivée à réaliser ce travail.

Finalement, je remercie tous ceux ou celles qui ont contribué de près ou de loin à la réalisation de ce travail.

Chapitre 1

INTRODUCTION

Dans ce chapitre, nous présentons le contexte de notre étude en mettant l'accent sur l'importance de la documentation pour les développeurs et en expliquant la cause de l'émergence du besoin en redocumentation. Ensuite, nous discutons des problèmes auxquels fait face la redocumentation. Nous présentons notre solution pour résoudre ces problèmes et nous mettons en évidence les principales contributions. Enfin, nous donnons un aperçu de la structure du mémoire.

1.1 Contexte

Un effort majeur est demandé aux développeurs face aux tâches de maintenance de logiciels. La maintenance consiste dans la modification de logiciels pour différentes raisons, comme par exemple la correction des défauts ou l'amélioration de la qualité. Elle nécessite souvent des altérations au niveau du code source du logiciel en question.

Par ailleurs, les logiciels sont de plus en plus complexes. Ils peuvent contenir des centaines de milliers de classes et des millions de lignes de code source. Le système d'exploitation Debian contient, dans sa version 5.0 par exemple, 1,446 millions de fichiers et 323 millions de lignes de code[1].

Plus la complexité des logiciels augmente, plus le coût de la maintenance croît [1].
De nos jours, le coût de la maintenance coûte entre 85-90% du coût de développement de logiciels [6]. La majeure partie du coût est due à l'effort que met le développeur dans la compréhension du code. D'après [30], le coût de la compréhension couvre entre 50-90% du coût de la maintenance. En effet, trouver les parties du code qui sont susceptibles aux changements est certainement une tâche fastidieuse qui demande une étude du code pour en comprendre les fonctionnalités et en délimiter les bouts à modifier.

---

[1] http: //libresoft.dat.escet.urjc.es/debian-counting/

Les développeurs ont souvent recours à la documentation comme support à la compréhension du code. La documentation est une description du logiciel qui peut être présentée sous différents formats. Par exemple, elle peut exister sous la forme de modèle tels que les modèles de conception UML (diagramme de classes, diagramme de séquence, etc.) ou sous forme textuelle, notamment dans les commentaires présents dans le code source. Grâce à la documentation, le temps mis pour la recherche des parties du code à changer est réduit et la compréhension du code par les développeurs est meilleure, ce qui se reflète positivement sur la qualité du travail à faire ainsi que sur la productivité [35].

Cependant, un problème majeur de la documentation est qu'elle peut être manquante ou bien non mise à jour [27]. Elle peut être aussi d'une mauvaise qualité dans le sens qu'elle ne révèle pas toute l'information utile à la compréhension du fonctionnement des bouts de code commentés.

Ainsi, le besoin de redocumentation s'avère nécessaire soit pour combler le manque de documentation ou pour générer une nouvelle documentation plus claire et plus précise.

Dans ce mémoire, nous nous intéressons à la redocumentation et nous proposons une approche pour la génération automatique de la documentation pour les programmes développés en Java. En particulier, nous nous restreindrons aux codes sources et aux commentaires comme source de documentation.

1.2 Problématique

Comme mentionné ci-haut, la documentation est d'une aide principale pour les développeurs et son absence influe négativement sur leur rendement. Une approche possible pour la génération de la documentation consiste à inciter les développeurs à écrire les commentaires au fur et à mesure qu'ils programment. Il s'agit d'une approche manuelle et purement préventive. En effet, les développeurs sont les meilleurs connaisseurs du code et ils sont les plus aptes à fournir une description correcte et claire des bouts de code qu'ils sont en train de programmer. À cette fin, des outils sont disponibles pour les encourager à commenter le code tel que Commenting by voice [2].

Un premier inconvénient de cette approche est que, malgré les facilités mises à l'usage des développeurs pour les motiver à documenter le code, ces derniers ne sont pas toujours persévérants à le faire. Ceci pourrait être expliqué par le fait que le développeur est pressé de terminer la tâche de développement, ainsi la cause revient principalement à un manque de temps. Le deuxième inconvénient de cette approche est qu'elle ne permet pas de résoudre le problème pour les projets déjà développés qui manquent de la documentation.

Une autre façon de résoudre le problème est de générer la documentation de manière automatique. Pour réussir l'approche automatique, nous pensons qu'il faut contourner trois problèmes de base.

Le premier problème touche à l'aspect technique et il peut être formulé comme suit : en absence du développeur et tout en ayant seulement le code source écrit dans un langage de programmation spécifique, comment peut-on générer une description en langue naturelle à l'aide d'un système de génération de documentation automatique ? Il s'agit ici d'une question sur la technique à suivre pour produire de manière automatique des descriptions écrites en langue naturelle.

Le deuxième problème s'intéresse à la qualité des descriptions produites et il peut être formulé comme suit : comment générer une documentation qui a un niveau de qualité acceptable ? En effet, il ne faut pas oublier que l'objectif est de générer automatiquement de la documentation afin d'aider les développeurs lors des tâches de maintenance. Il ne suffit pas donc de se limiter à la production automatique des descriptions mais il faudrait aussi que ces dernières soient, d'une part, compréhensibles et d'autre part, qu'elles contiennent de l'information utile sur le fonctionnement des bouts de code documentés.

Enfin, le troisième problème peut être énoncé comme suit : quel est le niveau de granularité adapté à la génération d'une documentation utile au développeur ? Nous pouvons aussi reformuler la question de la manière suivante : la documentation doit-elle être générée pour décrire les packages, les classes, les méthodes ou même les instructions élémentaires ? Il s'agit ainsi de voir quels sont les éléments qui intéressent le plus les développeurs et qui seront l'objet du système de génération de la documentation. Par

ailleurs, il est important de mentionner que quelques travaux se sont intéressés à la redocumentation automatique. Ces travaux seront présentés dans le chapitre *État l'art.*

1.3 Solution proposée

Nous proposons de considérer le problème de la génération de la documentation comme un problème de génération de résumé de code. En effet, d'après Sparck Jones [10], résumer un texte peut être défini comme suit :

> Une transformation réductrice d'un texte source vers un résumé par compression du contenu à l'aide d'une sélection et/ou généralisation de ce qui est important dans le texte source.

Le résumé du texte est donc une représentation réduite du texte original qui rend compte l'essentiel. Suite à cette définition, nous pouvons voir la documentation ou plus spécifiquement les commentaires, comme analogue au résumé : les commentaires sont en général des descriptions brèves du code source écrites par les développeurs dans le but de rendre la compréhension du code plus facile et plus rapide. Aussi, nous considérons que le texte original à résumer est analogue au code source et que finalement, la génération des commentaires consiste à résumer le code source.

Par ailleurs, plusieurs travaux s'intéressent à la génération automatique de résumé (voir chapitre *État de l'art).* Il est évident que l'automatisation de ce processus permet de réduire le temps et l'effort liés au déploiement d'une approche manuelle. Ainsi, nous proposons d'appliquer les techniques de résumé automatique de la langue naturelle pour la génération de la documentation de code.

Plus spécifiquement, dans notre mémoire, nous nous intéressons à résumer du code écrit dans le langage Java. Pour déterminer le niveau adéquat de génération du résumé, nous avons fait une étude empirique des commentaires. Cette étude est composée de deux sous-études. La première sous-étude est purement quantitative. Elle révèle certaines des habitudes de documentation des développeurs et elle montre que les développeurs s'intéressent particulièrement à commenter les classes. Nous avons par conséquent choisi

de nous limiter à la documentation de classes. Une classe encapsule une structure et un comportement communs à un ensemble d'objets. Elle représente donc un élément central pour les tâches de maintenance. Nous pensons alors qu'elle représente un bon niveau de granularité pour la redocumentation et nous considérons qu'une classe Java est le texte source à résumer.

Il existe deux techniques pour faire du résumé automatique : résumé par extraction ou résumé par abstraction [7]. Notre approche se base sur la technique du résumé par extraction. Le résumé par extraction consiste à sélectionner les phrases pertinentes d'un document source et à les concaténer pour obtenir un texte court [7]. Comme stipulé ci-haut, le document source est la classe Java à résumer. Chaque classe est formée d'instructions écrites en Java et éventuellement de commentaires. Sachant que les commentaires sont écrits en langue naturelle et qu'ils peuvent apporter des informations sur les fonctionnalités, nous les considérons comme objet d'extraction. Pour cela, nous nous sommes intéressés lors de la deuxième partie de l'étude empirique des commentaires à analyser leur contenu et leur qualité. Cette étude a permis de déterminer quels sont les commentaires les plus pertinents. Elle montre que la majorité des commentaires des méthodes sont de bons descripteurs de fonctionnalité du code. Par conséquent, nous proposons d'extraire les commentaires des méthodes pour résumer chaque classe.

Afin de procéder à l'extraction des commentaires des méthodes, nous avons défini plusieurs heuristiques. Chaque heuristique définit des règles d'extraction qui permettent de sélectionner parmi l'ensemble des commentaires des méthodes de la classe en question, ceux qui seront extrait.

Une fois que les commentaires à extraire spécifiés, la dernière étape consiste à les concaténer pour former le résumé de la classe.

Plus de détails sur l'approche seront présentés tout au long de ce mémoire.

1.4 Contributions

Les contributions principales du mémoire se résument en deux points :

- la réalisation d'une étude empirique des commentaires dans les programmes Java : l'étude s'intéresse à l'analyse de la distribution des commentaires et leur fréquence ainsi qu'à leur pertinence. Elle révèle des habitudes de documentation et elle détermine quels sont les commentaires les plus importants pour la compréhension des programmes. Cette étude a fait l'objet d'une publication acceptée à *ESEM 2011* [9].
- l'exploration d'heuristiques d'extraction pour la redocumentation : la redocumentation est faite en appliquant les techniques de résumé par extraction des commentaires de méthodes. Des heuristiques d'extraction ont été explorées à cette fin.

1.5 Structure du mémoire

Le reste du mémoire est organisé comme suit. Le chapitre 2 présente des travaux qui traitent des commentaires. Il dresse un état de l'art du résumé automatique de la langue naturelle. La méthode suivie lors de l'étude empirique des commentaires ainsi que les résultats obtenus sont décrits dans le chapitre 3. Le chapitre 4 explore les différentes heuristiques proposées pour la génération du résumé. Enfin, le dernier chapitre récapitule les idées principales et propose des perspectives au travail réalisé.

Chapitre 2

ÉTAT DE L'ART

2.1 Introduction

Dans ce chapitre, nous consacrons une première partie aux commentaires. Nous présentons brièvement les différents formats des commentaires et nous passons à la présentation des différents travaux qui se sont intéressés à leur étude. Ensuite, nous consacrons une deuxième partie à la présentation de certains travaux marquants dans le domaine du résumé automatique de texte naturel. Nous mettons l'accent sur les différentes techniques employées pour faire du résumé par extraction.

2.2 Les commentaires

2.2.1 Commentaires et formats

Les programmes Java peuvent avoir deux types de commentaires : les commentaires d'implémentation et les commentaires de documentation[2]. Les commentaires d'implémentation servent à décrire le code et ils peuvent se placer partout. Ils peuvent commencer par //ou bien ils peuvent être entourés par /*…..*/. Quant aux commentaires de documentation, ou encore appelés les commentaires *Javadoc*, ils servent à présenter les spécifications du code indépendamment de l'aspect implémentation. Ils sont faits pour documenter les classes, les interfaces, les attributs, les méthodes et les constructeurs. Les commentaires de documentation[3] suivent un format spécifique. Ils sont entourés par les délimiteurs /**…..*/. Un commentaire Javadoc est formé par une partie description du

[2] http://www.oracle.com/technetwork/java/codeconvtoc136057html
[3] http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

code et par un ensemble de tags. Les tags Javadoc permettent de préciser la valeur de certains éléments d'une façon standard. Par exemple, les tags @author et @version permettent de noter le nom de l'auteur et la version d'une classe. Les tags @param et @return permettent de décrire les paramètres ainsi que la valeur retournée d'une méthode.

2.2.2 Commentaires et travaux existants

Les commentaires sont très utiles aux développeurs principalement lors de tâches de maintenance. Souza et al. [28] ont fait une enquête pour connaître les artefacts les plus importants aux développeurs lors de tâches de maintenance. Les auteurs ont demandé à des professionnels expérimentés en maintenance d'évaluer l'importance des artefacts de deux approches de développement à savoir l'analyse structurée et le processus unifié. Les sujets répondaient en spécifiant le degré d'importance de chaque document sur une échelle allant de 1 à 4 (1 = pas important, 4 = très important). Pour les deux paradigmes de développement, l'analyse des réponses a montré que le code source et les commentaires sont les artefacts les plus importants pour la compréhension d'un système.

Malgré l'importance des commentaires, les développeurs ne sont pas toujours très enthousiastes à les écrire ou à les mettre à jour. Les commentaires non mis à jour ou incorrects peuvent malheureusement conduire les développeurs à effectuer des erreurs de programmation.

Dans ce contexte, Tan et al. [32] ont étudié manuellement les rapports de bugs de Mozilla. Ils ont trouvé que des bugs ont été introduits parce que les programmeurs ont suivi les indications incorrectes des commentaires. De plus, ils ont étudié les rapports de bugs de FreeBSD. Ils ont trouvé que 68 rapports concernent la non correctitude et l'ambiguïté des commentaires. Ceci montre bien que les programmeurs réalisent l'importance de la mise à jour des commentaires.

Malik et al. [18] ont effectué une étude empirique pour comprendre le raisonnement derrière la mise à jour des commentaires des fonctions quand ces dernières subissent un changement. Ils ont évalué l'effet de trois dimensions qui sont: les caractéristiques de la fonction modifiée, les caractéristiques du changement, la date du changement et le

propriétaire du code. Pour chaque dimension, ils ont examiné plusieurs attributs en utilisant la technique de forêt d'arbres décisionnels. Ils ont fait une étude de cas sur 4 projets dont la durée moyenne de l'historique s'étend sur 10 ans. Ils ont trouvé que plus le changement effectué est important, plus les développeurs ont tendance à actualiser les commentaires. Les auteurs en ont conclu que la dimension qui correspond aux caractéristiques du changement est le facteur qui influe le plus sur la mise à jour des commentaires.

Quelques travaux se sont intéressés à l'étude du contenu des commentaires. Ying et al. [36] ont fait une analyse manuelle des commentaires de tâche TODO. La première contribution de ce travail consiste dans la catégorisation des commentaires selon le contenu. Les auteurs ont trouvé que les commentaires sont un moyen de communication entre les développeurs. La communication peut être de type point-à-point (un développeur s'adresse à un autre en indiquant le nom de ce dernier dans le commentaire) ou bien elle peut être une communication par diffusion (un développeur s'adresse aux membres de l'équipe). De plus, ils ont trouvé que certains commentaires font référence aux identifiants des requêtes de changements enregistrées dans les systèmes de suivi de changement. À l'aide des identifiants, les développeurs peuvent consulter les détails d'un changement spécifique s'ils veulent avoir plus d'informations. De plus, les auteurs ont remarqué la présence de commentaires qui décrivent les tâches déjà effectuées, les tâches actuelles à faire et les tâches futures dont la réalisation dépend de traitements/ressources disponibles dans l'avenir. Aussi, ils ont trouvé que certains commentaires traitent d'un même concept et permettent par conséquent de détecter les parties reliées d'un code. Enfin, les auteurs ont affirmé que les commentaires TODO sont tous des marqueurs d'emplacement. Ils sont utilisés par le navigateur de tâches d'Eclipse pour diriger les développeurs vers les bouts de code contenant des commentaires de tâches. La deuxième contribution du travail consiste dans l'étude de la possibilité d'analyser automatiquement les commentaires. Les auteurs pensent qu'il est utile d'extraire automatiquement les noms des auteurs des commentaires et les identifiants des requêtes de changement. Ils ont cependant trouvé qu'il est difficile de traiter tous les commentaires automatiquement. En effet, les commentaires peuvent être

informels et ils sont généralement liés au contexte du code qui les entoure. Ceci rend leur traitement par un outil très délicat.

Padioleau et al. [22] ont étudié manuellement les informations contenues dans les commentaires dans le but d'en dégager des nouvelles directions de recherche pour améliorer la fiabilité des logiciels. Les auteurs se sont intéressés particulièrement aux commentaires qui expriment les besoins et les hypothèses des développeurs et ce, dans 3 systèmes d'exploitation écrits en C. Ils ont traduit les besoins exprimés en des directions de recherche. Par exemple, le commentaire de l'instruction suivante :

```
int mem; /* memory in 128 MB units */
```

explique que l'unité de la variable mem est MB. Par conséquent, les auteurs disent qu'il serait intéressant si les langages de programmation définissaient de nouveaux types comme seconds, milliseconds, Kilobyte, MegaByte, etc. De cette manière, les développeurs utiliseront le type le plus approprié au lieu d'utiliser le type général `int`. En général, les auteurs ont suggéré des idées pour étendre les langages de programmation et les langages d'annotation existants. Aussi, ils ont suggéré d'ajouter des fonctionnalités aux éditeurs de code pour simplifier la tâche de développement. Enfin, ils ont proposé le développement et l'extension d'outils capables d'analyser les commentaires pour détecter la présence de bugs potentiels dans le code.

D'une manière similaire à ces deux derniers travaux, nous nous intéressons à l'étude du contenu des commentaires dans une première partie de notre travail. Cependant, nous ne nous restreindrons pas à l'étude de certains types de commentaires spécifiques. Mais plutôt, nous faisons en sorte que l'étude soit généralisable et ce, en effectuant l'analyse d'un échantillon représentatif des différents commentaires qui existent.

Les travaux qui suivent se sont intéressés à l'analyse automatique des commentaires. Tan et al. [31] ont proposé un outil pour faire l'analyse automatique des commentaires dans le but de détecter la présence de bugs ou de commentaires incorrects. Ils se sont intéressés particulièrement aux commentaires communiquant des hypothèses de programmation. À partir de ces commentaires, ils ont extrait des règles. Les règles produites sont vérifiées par exploration du graphe d'appel du code. Chaque violation de règle traduit une inconsistance

entre le code et les commentaires. Toute inconsistance correspond soit à la présence d'un bug, soit à la présence d'un commentaire incorrect.

Shreck et al. [26] se sont intéressés à l'aspect qualité des commentaires. Dans leur travail, ils ont développé un outil pour évaluer automatiquement la qualité des commentaires. Ils ont défini la qualité selon trois dimensions : la complétude, la quantité et la lisibilité. Pour chaque dimension, plusieurs métriques ont été proposées. Par exemple, la métrique *DIR* (Documented Item Ratio) est définie pour mesurer la complétude des commentaires Javadoc. D'après les auteurs, un commentaire Javadoc est dit complet, si chaque méthode paramétrée est documentée par `@param`, chaque valeur retournée est documentée par `@return` et chaque exception est documentée par `@exception` ou `@throws`. DIR calcule la moyenne des éléments documentés. La métrique *WJPD* (Words in the Javadoc Per Declaration) est proposée pour évaluer la dimension quantité. Elle permet de mesurer la verbosité des commentaires en calculant la moyenne du nombre des mots qui forment les commentaires. Un petit nombre de mots indique que les commentaires sont peu verbeux alors qu'un nombre élevé indique que les commentaires présentent trop d'information. Enfin, la dimension lisibilité permet de mesurer si les commentaires sont facilement compréhensibles. Une manière de calculer cette dimension est d'utiliser la formule de *Kincaid* [13] qui est une fonction de la moyenne du nombre des mots et de syllabes présents dans la documentation. Une limite principale de cette approche vient au fait que les métriques proposées ont une nature plutôt quantitative et ne peuvent pas par conséquent prendre en considération le contexte qui entoure un commentaire ou encore sa sémantique. Par exemple, en utilisant la métrique WJPD, conclure qu'une documentation contient trop de mots ne veut pas dire forcément qu'elle contient de l'information inutile. Au contraire, ceci pourrait être indicateur de la complexité des méthodes commentées, et dans ce cas, une documentation importante en volume serait nécessaire au développeur chargé de la maintenance. Une deuxième limite est que les métriques proposées risquent de donner une bonne mesure de qualité à un commentaire bien écrit mais qui est en réalité un commentaire non consistant avec le code.

Le travail de Khamis et al. [12] pallie cette dernière limite par la définition de métriques qui mesurent la consistance code/commentaire. Ils ont proposé par exemple la métrique *RSYNC* qui permet de s'assurer que chaque tag `@return` commence par le type de la variable réellement retournée. De même, ils ont défini des métriques similaires pour vérifier le contenu des tags `@param`, `@throws` et `@exception`. Néanmoins, les métriques suggérées se restreignent à juger la consistance entre le contenu des tags et le code, vu que l'écriture des tags suit un format standard. Par contre, elles sont incapables de qualifier la consistance de la partie qui décrit les fonctionnalités avec la méthode en question. En effet, il n'y a pas une convention qui spécifie comment écrire la partie description du commentaire et évaluer son équivalence avec la méthode revient à un problème d'évaluation de la sémantique.

Par ailleurs, les deux travaux susmentionnés se sont intéressés à l'évaluation des commentaires à des niveaux hauts, i.e., projet, package ou module et ils comparent la qualité de la documentation globalement. Le deuxième travail trouve, par exemple, que le module PDE d'éclipse a une documentation de meilleure qualité que celles du reste des modules. En particulier, l'évaluation manque de mesurer la qualité des commentaires des concepts de bas niveaux et de les comparer entre eux comme comparer les commentaires des déclarations de variables et ceux des attributs. Ceci revient au fait que certaines métriques ne peuvent pas mesurer la qualité des commentaires à différents niveaux. La métrique DIR, par exemple, peut traiter les commentaires Javadoc seulement et elle ne peut pas traiter les commentaires au niveau des instructions.

Dans notre travail, nous nous intéressons de même à l'étude de la qualité des commentaires. À contrario des travaux précédents, nous faisons une étude manuelle et non pas automatique. En effet, nous sommes convaincus que la prise en compte du contexte est nécessaire pour une évaluation juste et que l'humain est le meilleur candidat pour accomplir cette tâche. D'autre part, nous étudions la qualité des commentaires des différents types d'instructions et nous faisons une comparaison entre eux.

Plusieurs outils ont été développés pour faciliter la génération de documentation de code. Javadoc[4] et Doxygen[5] sont parmi les outils les plus populaires. Ces outils génèrent une documentation par extraction des commentaires présents dans le code. Les commentaires doivent être écrits dans un format précis pour qu'ils puissent être exploités pour la documentation. Javadoc se limite à la génération de la documentation sous format HTML alors que Doxygen est capable de la générer sous d'autres formats (HTML, RTF, PDF, etc.). L'utilisation de ces outils est typique à la documentation par approche postérieure [21] qui se base sur l'extraction de l'information à partir de programmes existants.

Les *doclets* sont des programmes qui fonctionnent avec Javadoc. Ces programmes permettent de sélectionner le contenu à inclure dans la documentation et de générer la documentation dans différents formats. Par exemple, le doclet standard de Javadoc génère la documentation en HTML, PDFDoclet[6] génère la documentation dans des fichiers PDF, TexDoclet[7] génère la documentation dans un format Latex, etc.

Le format XML est particulièrement intéressant. En effet, la structure des fichiers XML rend l'analyse de l'information plus facile surtout pour les applications du traitement automatiques des langues naturelles (*TALN*).

SSLDoclet[8] par exemple est un doclet qui est proposé dans [11]. Ce doclet permet de représenter certains éléments du code source (tel que les classes, les méthodes, les attributs, etc.) et les commentaires Javadoc par un arbre syntaxique abstrait présenté dans un format XML. Il sépare entre la partie description du commentaire Javadoc et ses tags en les mettant dans des balises différentes. Par exemple, la balise <Author> spécifie le nom de l'auteur indiqué par le tag @author, la balise <Return_Block> contient le commentaire relatif au tag @return, etc.

---

[4] http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html
[5] http://www.stack.nl/ dimitri/doxygen/
[6] http://sourceforge.net/projects/pdfdoclet/
[7] http://java.net/projects/texdoclet
[8] http://www.semanticsoftware.info/javadoclet

Cependant SSLDoclet ne permet pas de représenter les différents types d'instructions tel que les appels de méthodes, les déclarations de variables, les affectations, etc. Aussi, il ne permet d'extraire que les commentaires Javadoc et il ne tient pas compte des autres commentaires, i.e, les commentaires d'implémentation qui commencent par // ou /*.

Contrairement à l'approche postérieure, l'approche [21] réfère aux scénarios où le développeur documente le code avant ou en même temps qu'il programme. Des outils comme Verbal source code descriptor [33] ou Commenting by voice [2] aident le développeur à commenter le code au fur et à mesure qu'il est en train de l'écrire. Ces outils permettent aux développeurs de verbaliser leurs pensées pendant qu'ils programment. Ils sont dotés d'un système de reconnaissance de la voix. Ce système permet de reconnaître les différents mots prononcés par le programmeur. Une fois les mots reconnus, ils seront insérés soit dans le code à l'emplacement du curseur (cas de Commenting by voice) ou dans une fenêtre de description dédiée à l'affichage de la documentation (cas de Verbal source code descriptor). Cependant, ces outils ne permettent pas de pallier le manque de la documentation dans les systèmes déjà développés.

Malgré les facilités mises à l'usage des développeurs pour les motiver à documenter le code, ces derniers ne sont pas toujours persévérants à le faire. Le manque d'intérêt à documenter pourrait être une cause majeure, i.e., les développeurs peuvent être inconscients de l'utilité des descriptions qu'ils étaient censés écrire pour les tâches de maintenance futures, qui en général, sont faites par une équipe de maintenance. De plus, dans une compagnie, la tâche assignée aux développeurs est de réaliser un logiciel qui s'exécute sans faute, suivie souvent par une phase de test afin de s'assurer du bon fonctionnement du logiciel. Contrairement à ce processus rigide, il n'y a pas d'obligation sur le contrôle de la présence de la documentation ou de sa qualité. Les développeurs se trouvent libres de mettre les commentaires ou non.

Pour pallier le manque de documentation, des travaux, le notre parmi eux, se sont intéressés à la génération de commentaires. Sridhara et al. [29] ont proposé une approche automatique pour la génération de la documentation à partir du code des méthodes Java. Étant donnés la signature et le corps d'une méthode, ils commencent par extraire les

instructions importantes au résumé. Ensuite, ils produisent pour chaque instruction sélectionnée une description en langue naturelle. Enfin, ils concatènent les descriptions générées pour former le résumé de la méthode en question. Les auteurs notent que 5 types d'instructions sont nécessaires au résumé. Les instructions sont :

- la dernière instruction qui termine une méthode
- les appels de méthode sans valeur de retour ou dont la valeur de retour n'est pas affectée à une variable
- les appels de méthode dont l'action est la même que celle de la méthode. Par exemple, si une méthode au nom de `compile` fait appel à une autre méthode nommée `compileRE`, alors, la méthode appelante et la méthode appelée possèdent la même action `compile`
- les instructions d'affectation qui assignent des valeurs aux variables utilisées dans les instructions précédentes
- les instructions qui contrôlent l'exécution des instructions sélectionnées (par exemple, la condition d'une instruction `if`)

D'après les auteurs, ces choix émergent après étude des commentaires et des besoins exprimés dans les résumés de plusieurs projets Java et après avoir pris l'avis de programmeurs sur les instructions qu'ils considèrent nécessaires. Une fois les instructions importantes extraites de la méthode, l'étape suivante consiste à transformer chaque instruction en une phrase formulée en langue naturelle. Pour ce faire, les auteurs ont défini des modèles de génération de texte pour chaque type d'instruction. Par exemple, le modèle des appels de méthode qu'ils proposent est le suivant :

```
action thème arguments_secondaires
and get type_retour, [si retour de valeur]
```

où action représente le rôle principal de la méthode, le thème est l'objet de l'action et arguments secondaires réfèrent aux paramètres qui sont utilisés pour réaliser l'action principale. Ces éléments sont extraits des instructions suite à l'analyse syntaxique de ces dernières. Une fois les résumés générés, les auteurs ont évalué les résultats en demandant à des sujets de donner leur avis sur la qualité des commentaires produits. Ils ont trouvé que les résumés sont précis, ne manquent pas d'information importante et qu'ils sont assez concis. L'inconvénient de cette approche est qu'elle ne peut s'appliquer que pour la génération d'un résumé de méthodes. Dans notre travail, l'approche que nous proposons est applicable pour différents niveaux de granularité. Néanmoins, à cause de contraintes de temps, nous nous limiterons dans ce mémoire à la génération des résumés pour les classes.

Haiduc et al. [8] ont proposé l'application des techniques de résumé de la langue naturelle pour la génération de la documentation au niveau des méthodes et des classes. Ils se sont basés principalement sur une approche extractive dont le résultat est un résumé formé par les mots les plus importants. Tout d'abord, ils ont commencé par former un corpus de mots à partir des commentaires et des identifiants du code. Ensuite, ils ont essayé plusieurs techniques pour déterminer les K premiers mots à inclure dans le résumé. Ils ont appliqué 4 techniques: la technique *Lead* basée sur la position des mots, les techniques *VSM* (Vector Space Model) et *LSI* (Latent Semantic Indexing) qui sont des techniques de la recherche d'information et une technique étalon (*baseline*) qui sélectionne des mots au hasard (technique *random*). Pour faire l'évaluation, les auteurs ont demandé à des sujets de juger la qualité des résumés générés. Ils ont trouvé que les résumés produits par Lead sont les meilleurs alors que les résumés générés par la technique random sont les moins appréciés. Ensuite, ils ont évalué les résumés produits par combinaison des techniques précédentes. Ils ont constaté que la combinaison des résumés générés par Lead et VSM est très intéressante et qu'elle donne de meilleurs résultats qu'en utilisant Lead seulement.

D'autre part, les auteurs ont évalué l'effet de la longueur des résumés sur la qualité. Ils ont remarqué que la majorité des sujets préféraient les résumés formés de 10 mots à ceux contenant 5 mots. Enfin, les auteurs ont remarqué que les résumés des méthodes sont mieux notés que ceux des classes.

D'une manière similaire à ce travail, nous appliquons la technique du résumé par extraction. Cependant, nous générons des résumés formés par des phrases et non pas par des mots. Nous pensons que les phrases portent plus de sens et communiquent mieux l'information.

2.3 Résumé automatique de la langue naturelle

Un résumé est "*une transformation réductrice d'un texte source en résumé par condensation du contenu par sélection et/ou généralisation de ce qui est important dans la source"* [10]. Ou encore, d'après le dictionnaire du Trésor de la Langue Française informatisé, le résumé est "*une présentation abrégée, orale ou écrite, qui rend compte de l'essentiel*". La notion d'important ou d'essentiel est la notion de base pour un résumé. Justement, les travaux des résumés automatiques qui existent se focalisent sur comment choisir l'information importante ou essentielle à partir d'un document pour en construire le résumé. Il existe deux grandes classes de résumés : les résumés par extraction et les résumés par abstraction. L'extraction consiste à extraire les phrases importantes qui vont former le résumé à partir du document alors que l'abstraction consiste en une phase d'extraction des phrases importantes, puis une phase de reformulation des phrases extraites. Dans ce qui suit, nous présentons les travaux influents dans le domaine du résumé automatique par extraction.

Luhn [17] fut le premier à proposer une méthode statistique pour la génération de résumés par extraction. Il avance que la fréquence des mots est un indicateur de l'importance des phrases. Luhn explique que l'importance d'une phrase vient en réalité de l'importance des mots qu'elle contient. En effet, lorsqu'un écrivain développe un aspect d'un sujet ou argumente une idée, il a tendance à mettre l'accent sur des mots significatifs et ce, en les répétant tout au long de son texte. L'auteur exclut les mots vides du traitement, tels que les pronoms et les prépositions, en les comparant à une liste de mots vides (*stop-list*) déjà préparée. Ensuite, les termes similaires qui ont un même préfixe et qui ont moins de six lettres différentes sont regroupés et ils sont considérés comme représentatifs d'une même notion. Enfin, seul les termes les plus fréquents sont conservés et sont considérés

comme des mots significatifs. La dernière étape consiste à attribuer un score à chaque phrase. Le score est une fonction du nombre des mots importants contenus par une phrase ainsi que la distance qui les séparent entre eux. Les phrases ayant les scores les plus élevés sont sélectionnées pour former le résumé.

Edmundson [4] a étendu le travail de Luhn en ajoutant d'autres critères de sélection de phrases du résumé. Il s'est intéressé à la présence de mots indices, à la présence de mots provenant des titres et des entêtes et à la position des phrases. En effet, Edmundson suggère que l'importance des phrases est affectée par la présence de mots indices tels que *impossible, difficilement ou signifiant*. Aussi, la présence de mots provenant des titres du document et des paragraphes augmentent la probabilité de sélection des phrases qui les contiennent puisque ces mots sont généralement bien choisis par l'écrivain. Enfin, les phrases qui se trouvent au début et à la fin du texte et des paragraphes sont des phrases pertinentes au résumé. Pour évaluer son approche, Edmundson a comparé les résumés générés avec des résumés de référence en calculant le pourcentage des phrases qui apparaissent dans les deux types de résumé. Il a trouvé que le critère de sélection des phrases selon leur position donne les meilleurs résultats, meilleurs même que ceux obtenus en appliquant le critère de fréquence des mots. De plus, il a trouvé que la combinaison des critères (présence de mots indices, présence des mots des titres et position des phrases) donne aussi de bons résultats.

Pollock et al. [23] se sont intéressés à résumer les articles scientifiques de chimie. Ils ont proposé d'appliquer le critère de présence de mots indices pour la sélection des phrases qui formeront le résumé. Leur hypothèse est que l'utilisation d'une liste de mots spécifiques au sujet traité donne de meilleurs résultats qu'en utilisant une liste de mots dédiés à l'anglais général ou à un autre sujet que celui des articles à résumer. La liste des mots indices contient à priori une catégorisation des mots en mots positifs et mots négatifs. Un mot positif augmente la probabilité d'extraction d'une phrase alors qu'un mot négatif diminue la probabilité d'extraction. Les auteurs appliquent le critère de la fréquence de mots pour adapter la liste des mots indices à chaque document à résumer et ce, en réajustant la positivité/négativité des mots. D'autre part, Pollock et al. ont proposé de supprimer les

parties inutiles d'une phrase comme supprimer les propositions introductives qui commencent par *in* ou celles qui se terminent par *that*. Aussi, ils ont suggéré de supprimer les phrases qui sont liées à des phrases précédentes. Enfin, dans le but d'alléger le texte, les auteurs ont proposé de remplacer les mots par leurs abréviations et les noms des substances chimiques par leurs formules.

Les trois travaux que nous venons de citer sont au fondement de tous les travaux sur le résumé automatique de la langue naturelle. Dans notre travail, nous nous intéressons à un type particulier de documents qui sont les classes Java. Nous exploitons le critère de position en l'adaptant à la structure spécifique des classes Java pour la génération de résumés de classes.

Des travaux sur le résumé automatique se sont basés sur la *théorie de la structure rhétorique* (RST : Rhetorical Structure Theory). D'après Mann et al. [19], la RST s'intéresse à l'organisation de textes naturels cohérents. La RST postule que les différentes parties d'un texte sont reliées entre elles par des relations. Pour représenter les relations, les auteurs ont proposé d'utiliser une structure noyau/satellite. Par exemple, pour le cas d'un texte formé d'une affirmation suivie d'une démonstration l'étayant[9], la RST pose une relation de démonstration entre les différents segments textuels. L'affirmation est l'information centrale et elle représente le noyau alors que la démonstration est le satellite.

Dans cette optique, Marcu [20] postule que le principe de nucléarité de la RST peut être utilisé pour déterminer les unités importantes d'un texte et qu'il serait intéressant d'extraire ces unités pour former un résumé. Marcu a effectué des expériences pour démontrer son hypothèse. Il a demandé à 13 juges d'évaluer le degré d'importance des différentes unités textuelles de 5 textes bien formés. En parallèle, il a construit pour chaque texte un arbre rhétorique pour en déterminer les noyaux. Les arbres sont construits manuellement. Ensuite, il a comparé les noyaux avec les unités qui ont été jugées comme importantes par les 13 juges. Il a trouvé que la correspondance entre les deux ensembles est significative avec un rappel et une précision de 70 %. Il conclut ainsi que la RST permet bel et bien de

[9] Exemple tiré de: http://www.sfu.ca/rst/

déterminer les parties importantes d'un texte. À partir de cette conclusion, il a proposé d'extraire les unités intéressantes pour former un résumé pour chacun des textes. Pour pouvoir évaluer son système de génération de résumé basé sur l'application de la RST, il a comparé les résumés générés avec des résumés générés par un système commercial de Microsoft Office97 et aussi avec des résumés générés d'une manière aléatoire. En comparant les résultats, il a trouvé que son système donnait les meilleurs résumés.

Teufel et al. [34] se sont intéressés particulièrement au résumé des articles scientifiques. Leur but est de générer des résumés capables de reproduire les différents contextes mentionnés dans l'article à résumer. Ils postulent que le contexte est représenté en réalité par le rôle rhétorique que les phrases possèdent et ils ont spécifié les statuts rhétoriques qu'ils pensent nécessaires à inclure dans le résumé. Par exemple, les auteurs proposent d'extraire les phrases qui décrivent l'objectif de l'article à résumer, les phrases qui expliquent la différence avec les autres travaux, les phrases qui citent les solutions importées de travaux connexes, etc. Pour définir les statuts rhétoriques, ils ont eu recours à des humains pour annoter les textes. Étant donnés les textes annotés, ils ont proposé ensuite de faire l'annotation d'une manière automatique par apprentissage machine.

Nous pensons qu'il est intéressant d'appliquer la RST pour faire l'extraction des commentaires pertinents. Cependant, l'emploi d'une telle technique nécessite une base de commentaires annotés par les relations rhétoriques qu'ils portent.

Kupiec et al. [14] ont proposé l'application des techniques d'apprentissage automatique pour la génération de résumé. Les auteurs ont définit une fonction de classification bayésienne qui calcule la probabilité qu'une phrase soit incluse dans le résumé en tenant compte de cinq critères qui sont: la longueur de la phrase, la présence de mots indices, la position des phrases au niveau des paragraphes, la présence de mots thématiques et la présence de noms propres. La fonction de classification suit la formule suivante :

$$P(s \in S|F_1, F_2, \dots F_k) = \frac{P(F_1, F_2, \dots F_k|s \in S)P(s \in S)}{P(F_1, F_2, \dots F_k)}$$

où *s* est une phrase, *S* est le résumé, F1;F2, … FK représentent les critères. L'apprentissage est effectué sur un corpus d'articles scientifiques formé de 188 paires (document/résumé).

Les phrases des documents sources sont alignées avec celles du résumé. Ensuite, le système détermine les caractéristiques des phrases alignées selon les 5 critères. Chaque phrase reçoit un score pour chacun des critères résultant ainsi en une estimation de la probabilité que la phrase soit incluse dans le résumé. Kupiec et al. ont trouvé des résultats conformes avec ceux de Edmundson dans la mesure où le critère de position donne les meilleurs résultats. Dans le cas de combinaison, ils ont trouvé que la combinaison (position dans les paragraphes + mots indices + longueur des phrases) est la meilleure des combinaisons.

Nous croyons que l'emploi des techniques d'apprentissage est très intéressant, surtout que les commentaires références sont disponibles dans les projets documentés. Il est possible ainsi d'envisager l'emploi de cette technique pour la génération de résumé dans le futur.

Salton et al. [25] ont proposé l'extraction de paragraphes pour former un résumé. Dans leur approche, chaque document est représenté par un graphe. Les sommets du graphe sont les paragraphes et les liens représentent le degré de similarité entre les différents paragraphes. Les paragraphes les plus importants sont ceux dont les nœuds ont plus de connexions, c'est à dire, ce sont les paragraphes qui ont plusieurs liens de similarité avec les autres paragraphes. Ces paragraphes discutent de sujets couverts dans beaucoup d'autres paragraphes. Les auteurs ont proposé trois façons de faire l'extraction. La première façon, appelée *Bushy path*, consiste à extraire les nœuds qui ont le plus de connexions (*bushy node*). La deuxième façon, appelée *Depth-first path*, commence par extraire un nœud important et continue ensuite l'extraction du voisin le plus similaire au nœud extrait et ainsi de suite. La troisième façon, appelée *Segmented bushy path*, propose d'extraire au moins un paragraphe par segment pour assurer plus de cohérence au résumé. Un segment est une suite de paragraphes qui forme une unité fonctionnelle. Par exemple, une unité qui introduit le texte, une autre unité qui le conclut, etc. En comparant les résumés générés par les trois méthodes avec ceux générés manuellement, les auteurs trouvent que l'application de la méthode *Bushy path* produit les meilleurs résumés.

Contrairement à Salton, Erkan et al. [5] se sont intéressés aux résumés multi-documents. L'ensemble des documents est représenté par un graphe où les sommets représentent les

phrases et les liens représentent le degré de similarité entre les différentes phrases. Comme Salton, les auteurs ont proposé de considérer comme centrales les phrases dont les nœuds possèdent beaucoup de liens de similarité. De plus, ils ont proposé deux autres techniques basées sur l'algorithme *PageRank*. PageRank [3] est un algorithme utilisé par le moteur de recherche *Google*. L'algorithme classe les pages selon leur importance dans le web. Il se base sur le principe de propagation de l'importance. Une page a un PageRank élevé si elle possède plusieurs pages pointant vers elle ou bien si elle en possède quelques unes ayant des PageRanks élevés. D'une façon similaire, les auteurs ont proposé d'appliquer le principe de distribution de centralité aux phrases par application de PageRank. Après évaluation des résultats, les auteurs ont trouvé que les heuristiques proposées donnent de bons résultats et qu'il n'y a pas une différence significative entre elles.

Pour la génération de résumé de classes Java, nous proposons plusieurs heuristiques qui se basent principalement sur le critère de position. Dans l'une des heuristiques proposées, nous utilisons les graphes pour déterminer les commentaires candidats à l'extraction. Les nœuds du graphe sont les méthodes et les liens représentent les liens d'appels entre les méthodes. Nous donnerons plus de détails sur les heuristiques dans le dernier chapitre.

2.4 Conclusion

Dans ce chapitre, nous avons présenté des travaux qui se sont intéressés aux commentaires. De plus, nous avons présenté des techniques de résumé par extraction, et ce, à travers la présentation des travaux qui les ont appliquées. Nous avons mis en évidence les aspects particuliers à notre étude par rapport aux autres travaux (tels que l'étude des commentaires de différents types d'instructions, l'étude de la qualité des commentaires avec la prise en compte du contexte, etc.). Aussi, nous avons présenté les techniques des autres approches que nous avons adoptées dans la notre (tel que l'adaptation du critère de position pour l'extraction des commentaires, représentation des classes Java par un graphe, etc.).

Chapitre 3

ÉTUDE EMPIRIQUE DES COMMENTAIRES

3.1 Introduction

Durant les tâches de maintenance, les commentaires apportent une aide précieuse aux développeurs. Ils communiquent des informations utiles sur les fonctionnalités d'un système. Dans ce chapitre, nous présentons une étude empirique qui a permis d'analyser particulièrement la dimension quantitative et la dimension qualitative des commentaires. À cette fin, nous avons effectué trois études. La première étude analyse la distribution et la fréquence des commentaires dans du code. La deuxième étude s'intéresse au contenu des commentaires et à leur qualité. Enfin, la troisième étude est menée à partir d'un questionnaire sur les habitudes de documentation d'un nombre de sujets

3.2 Étude 1 : Étude de la distribution des commentaires et de leur fréquence

3.2.1 Méthodologie

La première étude s'intéresse au calcul de la distribution des commentaires et de leur fréquence et ceci en tenant compte des différents types d'instructions. Pour effectuer cette étude, nous avons suivi les étapes suivantes :

1. Sélection des projets à étudier
2. Génération d'arbres syntaxiques
3. Calcul de la distribution et de la fréquence des commentaires

Les étapes sont illustrées par le schéma de la figure 1. Elles seront explicitées en détail dans la suite.

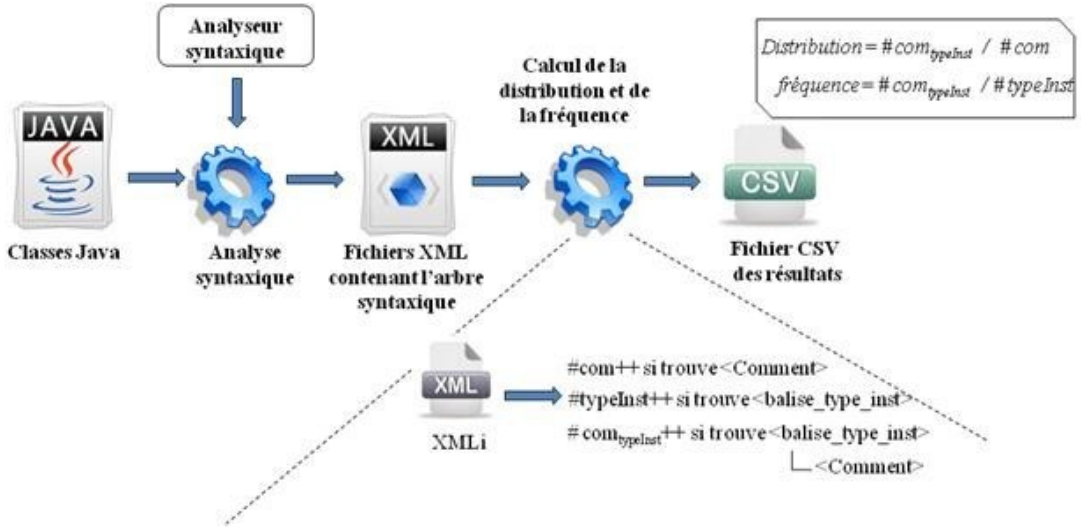


Figure 1- Méthode suivie dans l'étude 1

3.2.1.1 Sélection des projets et justification

Nous avons fait l'étude de la distribution des commentaires et de leur fréquence dans trois projets libres développés en Java :

- DrJava[10] : est un environnement de développement de programmes Java dont la réalisation a commencé depuis 2001 au sein du groupe JavaPLT[11] de l'université Rice. Ce projet est toujours en cours de développement. Il est développé par plus de 50 programmeurs au fil des années.
- SweetHome3D[12] : est un logiciel d'aménagement d'intérieur qui aide les utilisateurs à placer leurs meubles sur le plan d'un logement en 2D, avec une prévisualisation en 3D. Il a été développé entre 2005 et 2010 par un développeur Java senior.
- jPlayMan[13] : est un logiciel pour la création et la gestion de listes de lecture de musique. Il est développé par un unique programmeur depuis 2009.

[10] http://www.drjava.org/
[11] http://www.cs.rice.edu/ javaplt/
[12] http://www.sweethome3d.com/fr/index.jsp
[13] http://jplayman.sourceforge.net/

Les trois projets ont été sélectionnés pour différentes raisons. Tout d'abord, ils traitent de différents domaines. Aussi, ils sont développés à des années différentes et pendant des périodes variées (d'une année à dix ans). De plus, le nombre de développeurs des différents projets varie. Ce sont des projets ordinaires développés par des étudiants et par des professionnels. Ils représentent des projets de la vie de tous les jours où les pratiques concernant les commentaires varient (de bien à mal écrit). Nous n'avons pas pris des projets connus pour leur bonne documentation par ce qu'ils ne représentent pas forcément un standard des projets qui existent. Enfin, les projets sont de tailles différentes. Nous résumons leurs caractéristiques principales dans le tableau 1.

Tableau 1- Caractéristiques principales des projets étudiés

	DrJava	SHome3D	JPlayMan
|packages|	14	9	9
|classes + interfaces|	591	180	40
|méthodes|	9 200	3 600	664
|lignes de code (LOC)|	145 511	66 484	20 869
|lignes de commentaires (CLOC)|	50 666	16 509	5 814
|nb de commentaires (NCOM)|	14 954	5 636	1 357
CLOC/LOC	34,8%	24,8%	27,9%
NCOM/LOC	10,3%	8,5%	6,5%

Les projets comportent de 40 à 591 classes et leur taille varie de 20 à 145 KLOC (*Kilo Line Of Code*). Les commentaires représentent plus ou moins 30% du nombre des lignes de code ce qui indique que la présence des commentaires n'est pas anecdotique dans ces projets. Il est à noter que ce chiffre inclut les lignes de commentaires vides. Le nombre des commentaires vides n'est pas important. Il est égal à 208 pour les trois projets. Le programme DrJava est le plus commenté. Ce projet comporte en moyenne un commentaire pour chaque dix lignes de code. Cette moyenne est calculée en divisant le nombre des commentaires du projet DrJava par le nombre de ses lignes de code (NCOM/LOC=10,3%). Le fait que DrJava comporte le plus de commentaires n'est pas

surprenant vu qu'il s'agit du plus grand système avec un nombre de développeurs important. Les deux autres projets ont relativement moins de commentaires.

3.2.1.2 Génération de l'arbre syntaxique

Pour pouvoir calculer automatiquement la distribution et la fréquence des commentaires, il est important d'avoir une représentation du code qui permet de délimiter les commentaires et les différents types d'instructions. Il s'agit ainsi de représenter le code sous la forme d'un *arbre syntaxique abstrait* (ASA) afin d'en mettre en évidence la structure. La figure 2 montre un exemple d'un ASA simplifié d'une classe qui comporte une méthode nommée m_1 possédant un paramètre `i` de type `int`. La méthode m_1 fait appel à une méthode m_2.

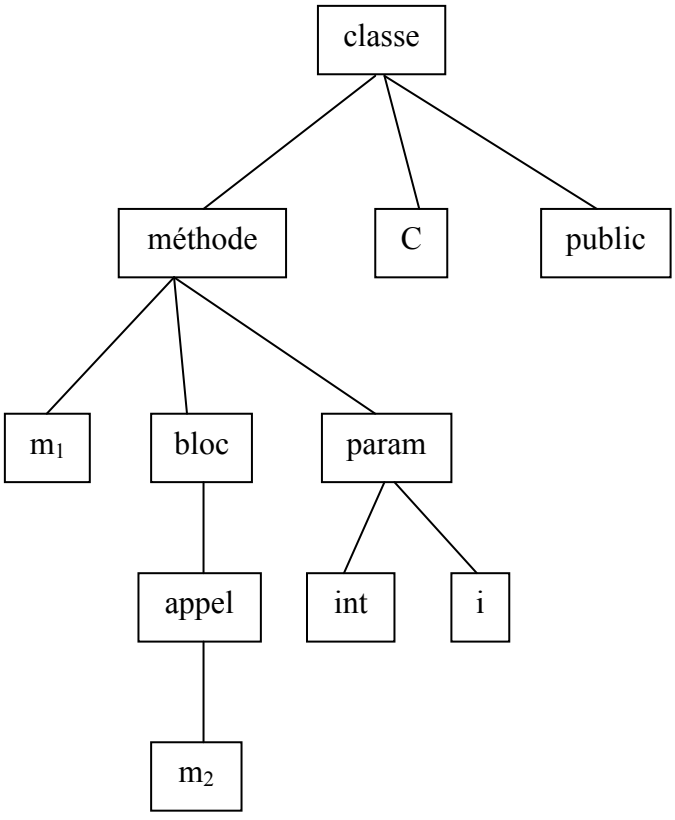


Figure 2 - Exemple d'un arbre syntaxique simplifié d'une classe

Pour générer l'ASA, nous avons utilisé l'outil SableCC[14] afin de produire un analyseur syntaxique pour la grammaire Java. Ensuite, nous avons appliqué cet analyseur sur les trois projets étudiés. Nous obtenons ainsi, pour chaque classe Java, un fichier XML qui contient l'ASA correspondant. Cependant, l'ASA généré par défaut ignore les commentaires. Nous avons ainsi modifié le code de SableCC pour que les commentaires soient présentés dans des balises <Comment> dans l'ASA. Aussi, à des fins de vérification, nous avons ajouté des attributs pour spécifier le numéro de la première et de la dernière ligne de chaque commentaire.

La figure 3 montre une instruction de type déclaration d'attribut et la partie de l'ASA qui lui correspond. Chaque commentaire est contenu par les balises <Comment> et </Comment>. L'analyseur syntaxique associe par défaut le commentaire à l'instruction qui le suit. Dans la figure 3 par exemple, le commentaire est entouré par la balise <AMemberClassBodyDeclaration> ce qui signifie qu'il est associé à un membre de la classe (soit attribut ou méthode). Le membre est spécifié grâce à la balise <AFiledClassMemberDeclaration> qui entoure les instructions de type déclaration d'attribut. Ainsi, après lecture de cette partie de l'arbre, nous pouvons dire que le commentaire est associé à une instruction de déclaration d'attribut.

En partant de l'intuition que les développeurs écrivent les commentaires juste avant l'instruction dont ils veulent communiquer des informations, nous croyons que l'association d'un commentaire à l'instruction qui le suit est un choix raisonnable. Nous avons adopté ce choix lors de l'étude 1. Dans l'étude 2, nous vérifierons cette hypothèse et nous montrerons à quel degré elle est juste.

[14] http://sablecc.org/

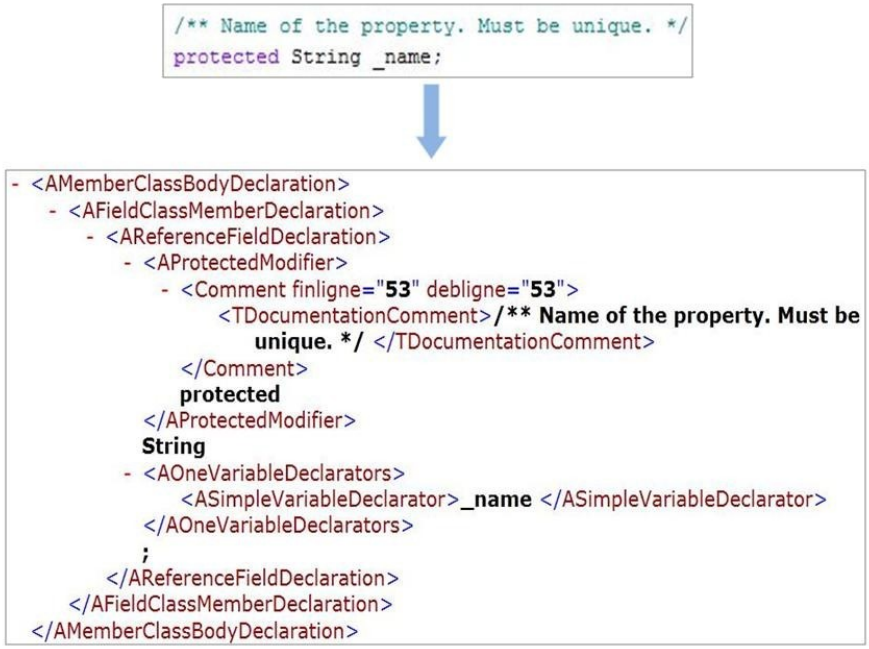


Figure 3 - Extrait d'un arbre syntaxique en format XML et le code correspondant

3.2.1.3 Calcul de la distribution et de la fréquence

Nous avons développé un outil pour calculer automatiquement la distribution et la fréquence des Commentaires. La distribution représente le pourcentage des commentaires de chaque type d'instruction par rapport au nombre total des commentaires. Elle est calculée selon la formule suivante :

$$distribution = \frac{\#\mathrm{com}_{\mathrm{typeInst}}}{\#\mathrm{com}}$$

où $\#com_{typeInst}$ est le nombre de commentaires associés aux instructions de type *typeInst* et *#com* représente le nombre total de commentaires.

La fréquence représente le pourcentage de commentaires d'un type d'instruction par rapport au nombre total des instructions du même type. Elle est calculée selon la formule suivante :

$$\text{fréquence} = \frac{\#\mathrm{com}_{\mathrm{typeInst}}}{\#\mathrm{typeInst}}$$

où *#typeInst* est le nombre d'instructions de type *typeInst*.

Dans l'arbre syntaxique, les commentaires et les différents types d'instructions sont entourés par des balises appropriées. Aussi, les commentaires sont associés aux instructions qui les suivent. L'outil développé exploite ces informations pour parcourir l'arbre syntaxique, déterminer le nombre total des commentaires et calculer le nombre de chaque type d'instructions ainsi que le nombre des commentaires associés. Dans ce qui suit, nous expliquerons comment l'outil fonctionne.

L'outil utilise L'API JDOM[15] pour parcourir l'arbre syntaxique et pour localiser les différentes balises. Il calcule le nombre total des commentaires en dénombrant le nombre des balises <Comment>. Le compte est fait avec l'hypothèse que chaque suite de commentaires forme un seul commentaire et donc chaque suite de balises <Comment> est calculée comme une seule balise. En effet, les développeurs communiquent parfois de l'information détaillée, et en plus de la possibilité de l'écrire sur une seule ligne, ils peuvent aussi l'écrire sur plusieurs lignes pour en faciliter la lecture. La figure 4 en montre un exemple.

Pour calculer le nombre d'instructions de chaque type, l'outil calcule le nombre des balises correspondantes.

Par exemple, la balise <ALocalVariableDeclarationStatement> correspond aux déclarations de variable et la balise <AForLoopStatement> correspond aux boucles for.

Calculer le nombre de commentaires de chaque type d'instruction revient à déterminer pour chaque balise <Comment> la balise correspondante au type d'instruction qui l'enveloppe et d'en incrémenter le nombre. Toutefois, les commentaires ne sont pas tous suivis d'instructions. Il y'a ceux qui se placent au début d'un bloc de code et ils sont suivis de { ou ceux qui se trouvent à la fin d'un bloc et sont suivis par }. Ces commentaires sont affectés à la catégorie non suivi. Il est possible de les détecter automatiquement vu qu'ils sont des descendants directs de la balise <ABlock> qui sert à entourer tout bloc de code.

---

[15] www.jdom.org

Dans la figure 5, le premier commentaire :

// Detach the 3 canvases from their view

est enveloppé par la balise <AStatementBlockStatement> et il n'est pas fils direct de la balise <ABlock> puisqu'il est suivi d'une instruction d'appel de méthode.

Le deuxième commentaire :

// Super class did the remaining of clean up

est mis à la fin du bloc et la balise qui l'entoure est fille de la balise <ABlock>. Par conséquent, il est affecté à la catégorie non suivi.

À la fin des calculs, l'outil génère un fichier CSV qui contient les résultats que nous présentons dans ce qui suit.

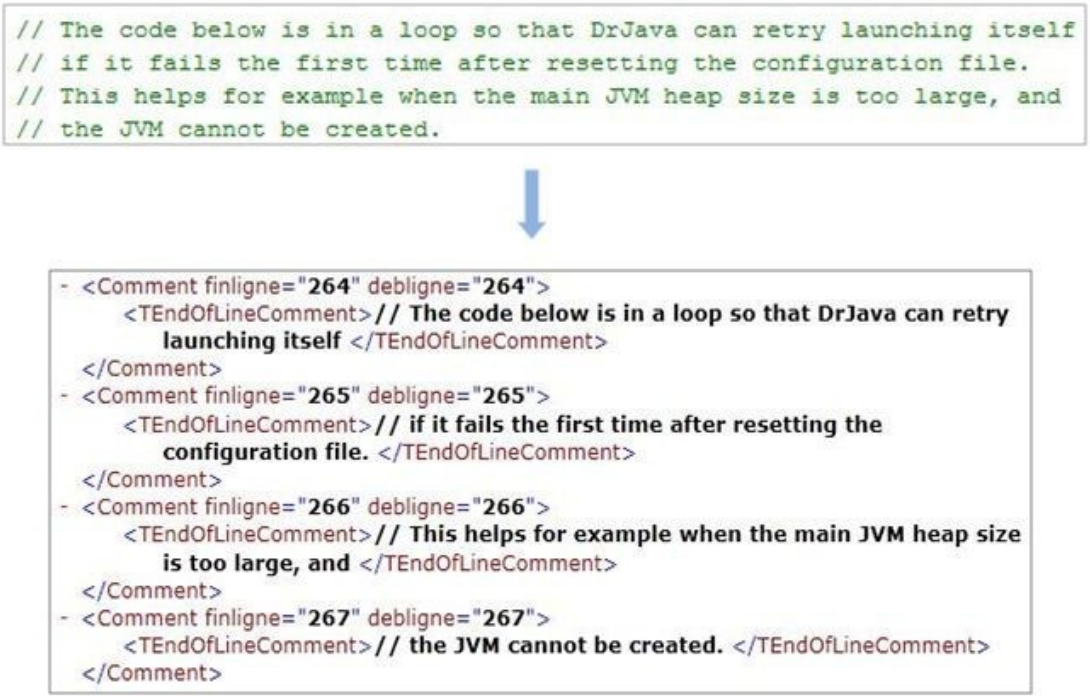


Figure 4 - Extrait d'un ASA correspondant à un commentaire écrit sur plusieurs lignes

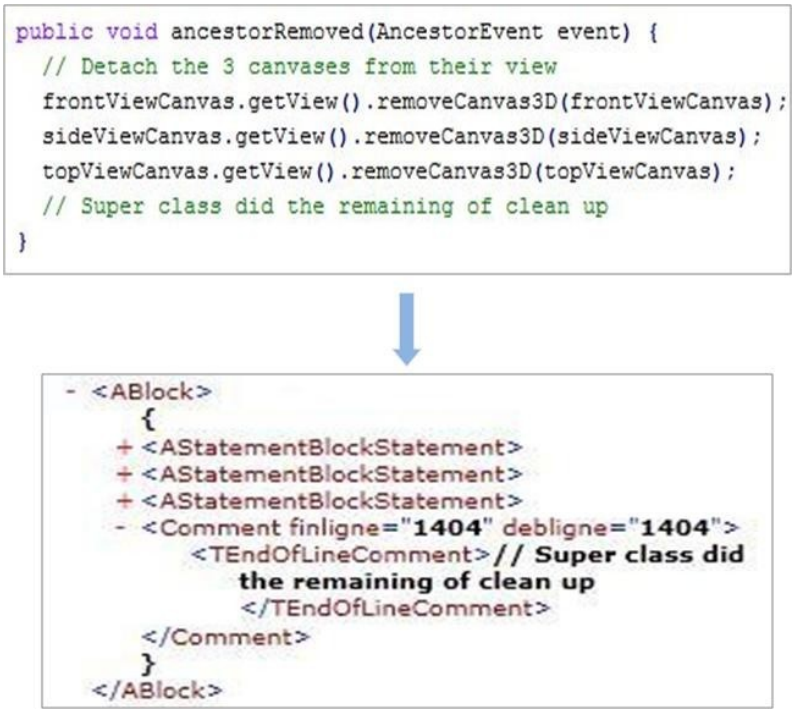


Figure 5 - Exemple de commentaire de la catégorie non suivi

3.2.2 Résultats et discussion

Le tableau 2 présente les résultats de la distribution des commentaires sur les types d'instructions que nous jugeons intéressants.

Nous dégageons trois tendances du tableau 2 :

- La plus grande partie des commentaires (33,4%) précède les méthodes. Cette observation est attendue puisque les méthodes intéressent les développeurs. Elles servent à implémenter les fonctionnalités des classes et elles ont généralement un niveau de granularité assez important.
- Les commentaires précédant les déclarations de variables locales et les appels de méthodes représentent respectivement 12,5% et 15,7% du total des commentaires. Les déclarations de variables et les appels de méthodes sont des types qui sont présents souvent. Ceci explique leur deuxième et leur troisième position respectivement.

Tableau 2 - Distribution des commentaires sur les types d'instructions

Type	Dr Java (%)	SHome3D (%)	jPlayMan (%)	moy.
déclaration de package	3,9	3,2	3,0	3,4
déclaration d'import	0,1	0,0	0,0	0,0
déclaration de classe	4,6	5,2	5,2	5,0
déclaration d'interface	0,6	0,4	0,2	0,4
déclaration d'attribut	10,3	1,1	3,5	5,0
déclaration de constante	2,2	0,0	0,0	0,7
méthode	27,2	36,3	36,6	33,4
méthode abstraite	4,7	2,2	0,7	2,5
constructeur	3,1	3,3	4,7	3,7
variable locale	8,8	13,8	14,9	12,5
affectation	3,4	4,4	4,6	4,1
appel de méthode	17,5	19,5	10,1	15,7
for	0,6	1,7	1,4	1,2
while	0,4	0,0	0,2	0,2
if	3,5	4,0	6,1	4,5
return	1,3	1,0	2,2	1,5
tryCatch	0,8	0,0	0,3	0,4
énumération	0,1	1,4	0,8	0,8
autre	6,9	2,5	5,5	5,0

- Le quatrième type d'instructions qui possède le plus de commentaires sont les déclarations de classe (5%) bien qu'elles soient moins fréquentes que les importations de package dont 0,03 % seulement sont commentées. Les classes sont des types importants aux développeurs puisqu'elles permettent d'implémenter les fonctionnalités du système.

La distribution des commentaires donne une idée de leur emplacement dans le code. Elle est liée à l'importance des types commentés. Il est cependant à noter que ces figures sont clairement biaisées par la fréquence des types d'instructions considérés, i.e., les commentaires ont davantage tendance à précéder les éléments fréquents comme il est le cas des variables locales et des appels de méthode. Tout de même, ils permettent de refléter certaines habitudes de documentation. Par exemple, bien que les méthodes soient moins nombreuses (12501) que les appels de méthode (34365), elles sont précédées par les

commentaires plus souvent que ces derniers. D'autre part, la distribution des commentaires montre quelles sont les parties du code qui ne sont pas souvent commentées.

Le calcul de la fréquence des commentaires pour les différents types d'instructions permet d'équilibrer ces observations. Les résultats sont présentés dans le tableau 3.

Tableau 3 - Fréquence des commentaires par type d'instruction

Type	Dr Java (%)	SHome3D (%)	jPlayMan (%)	moy.
déclaration de package	98,8	100,0	100,0	99,6
déclaration d'import	0,4	0,0	0,0	0,1
déclaration de classe	84,6	98,3	100,0	94,3
déclaration d'interface	97,9	100,0	100,0	99,3
déclaration d'attribut	54,1	3,5	13,3	23,6
déclaration de constante	87,2	0,0	0,0	29,1
méthode	48,6	58,9	75,9	61,1
méthode abstraite	85,6	95,4	100,0	93,7
constructeur	61,3	58,8	80,0	66,7
variable locale	18,0	18,0	15,4	17,1
affectation	8,3	8,5	9,0	8,6
appel de méthode	11,1	12,1	7,6	10,3
for	11,3	16,4	11,4	13,0
while	21,8	2,3	12,0	12,0
if	9,7	6,0	6,6	7,4
return	4,9	3,3	4,9	4,4
tryCatch	5,0	0,0	1,3	2,1
énumération	25,0	14,6	26,2	21,9

Nous distinguons trois groupes de type d'instruction :

- Les méthodes abstraites, les déclarations de classe, les déclarations d'interface et les déclarations de package sont presque toujours précédées par des commentaires (en moyenne, 93,7% à 99,6% des cas sont commentés).
- Les méthodes et les constructeurs sont précédés par les commentaires dans 61,1% et 66,7% des cas respectivement.
- Les déclarations d'importation et les instructions `try-catch` sont les éléments les moins commentés avec une fréquence de documentation de 0,1% et 2,1% respectivement.

Nous observons aussi dans le tableau que la fréquence de documentation de plusieurs éléments varie énormément d'un projet à l'autre. Nous citons comme exemple le cas de déclarations d'attribut qui sont commentés la moitié du temps dans le projet DrJava mais dans de rares occasions dans les deux autres projets. Ceci indique que malgré qu'il y ait un consensus général pour commenter certains types d'instructions, il y a une certaine différence "culturelle" pour d'autres. Cependant, il est à noter que le cas où les déclarations de constante sont commentées souvent dans le projet DrJava (87,2% des cas) alors qu'elles ne sont pas documentées du tout dans les deux autres projets revient en réalité à l'absence de ces types dans les 27 interfaces de ces deux derniers projets.

Bien que ces figures permettent de donner une image absolue et relative sur les habitudes de documentation des développeurs, elles ne permettent pas de caractériser la qualité des commentaires, ni ne permettent de spécifier si un commentaire est véritablement lié à une instruction. Pour comprendre ces aspects, nous avons effectué une deuxième étude que nous expliciterons dans la section qui suit.

3.3 Étude 2 : Étude du contenu des commentaires et de la pertinence

3.3.1 Méthodologie

Contrairement à l'étude de la distribution des commentaires et de leur fréquence, l'étude du contenu et de la pertinence s'intéresse à la sémantique des commentaires et ne peut pas par conséquent être automatisée. Pour cette raison, nous avons effectué une expérience durant laquelle nous avons demandé à des sujets programmeurs d'analyser des fragments de code.

Nous avons utilisé l'échantillonnage par strates pour générer un ensemble représentatif de fragments de code avec leurs commentaires. L'échantillonnage par strate consiste à diviser une population en sous populations homogènes appelées strates et à appliquer ensuite l'échantillonnage au niveau de chaque strate. Il est utilisé quand la population est très variée. Dans notre cas, la population correspond à l'ensemble des commentaires

extraits des trois projets présentés lors de l'étude 1 ainsi que les lignes du code qui leurs sont adjacentes. Nous divisons ainsi la population en strates qui correspondent chacune à un type d'instruction. Nous trouvons dans chaque strate les commentaires qui précèdent un type d'instruction particulier. Parmi les strates générées, nous avons par exemple, la strate des commentaires qui précèdent les déclarations de classe, la strate des commentaires qui précèdent les boucles while, etc. Ensuite, nous sélectionnons d'une manière aléatoire de chaque sous population un ensemble de commentaires et les lignes de code qui l'entourent. Le nombre des commentaires tirés est proportionnel à la moyenne de la taille de la sous population observée durant l'étude de la distribution des commentaires (voir étude 1, tableau 2).

Chaque fragment de code tiré est transformé en un formulaire web (figure 6) où chaque participant inspecte le code et répond à des questions sur le contenu et la pertinence du commentaire. Pour chaque fragment de code, nous avons décidé de présenter le commentaire choisi en couleur rouge avec les 10 lignes du code qui le précèdent et les 20 lignes qui le suivent. Ce choix émerge de l'examen de plusieurs commentaires afin de décider de la taille de la fenêtre nécessaire pour communiquer l'information appropriée aux participants. De plus, ce choix limite la taille du fragment du code et permet aux participants d'effectuer la tâche d'annotation sans devoir à priori faire défiler le code dans la fenêtre.

Concernant les sujets, nous avons invité par email des programmeurs (académiques, industriels et étudiants gradués) pour participer à l'expérience de l'analyse des commentaires. Parmi ces personnes, 49 ont accepté d'y participer. Tous les sujets sont expérimentés en programmation orientée objet et plus particulièrement en Java. Pour valider les réponses des participants, nous avons créé aléatoirement des groupes de trois personnes de manière à ce que chaque groupe annote 30 fragments de code. Une personne est donc a été exclue et au final, nous avons obtenu 16 groupes de 3 participants. Au total, 480 commentaires sont inclus dans l'échantillon à analyser.

Une session d'une heure a été planifiée pour chaque participant. Elle commençait par un mini-tutoriel qui expliquait la tâche d'annotation avec des exemples et présentait

l'application web à utiliser. Après, les participants étaient laissés à eux même jusqu'à ce qu'ils terminent leur tâche. À la fin, les sujets remplissaient un dernier questionnaire sur leur profil et leurs habitudes de documentation. Pour chaque fragment de code, nous avons collecté les réponses des trois sujets qui l'ont analysé et nous utilisons un système de vote pour décider des valeurs finales des réponses à retenir. Dans les cas rares où aucun accord n'a été trouvé, c'est à dire, que chaque sujet a choisi une réponse différente des autres, nous excluons le commentaire de l'échantillon. Nous avons aussi exclu les réponses des participants qui n'ont pas terminé leur tâche d'annotation. Après la collecte et la validation des données, nous avons obtenu des réponses complètes pour 407 commentaires.

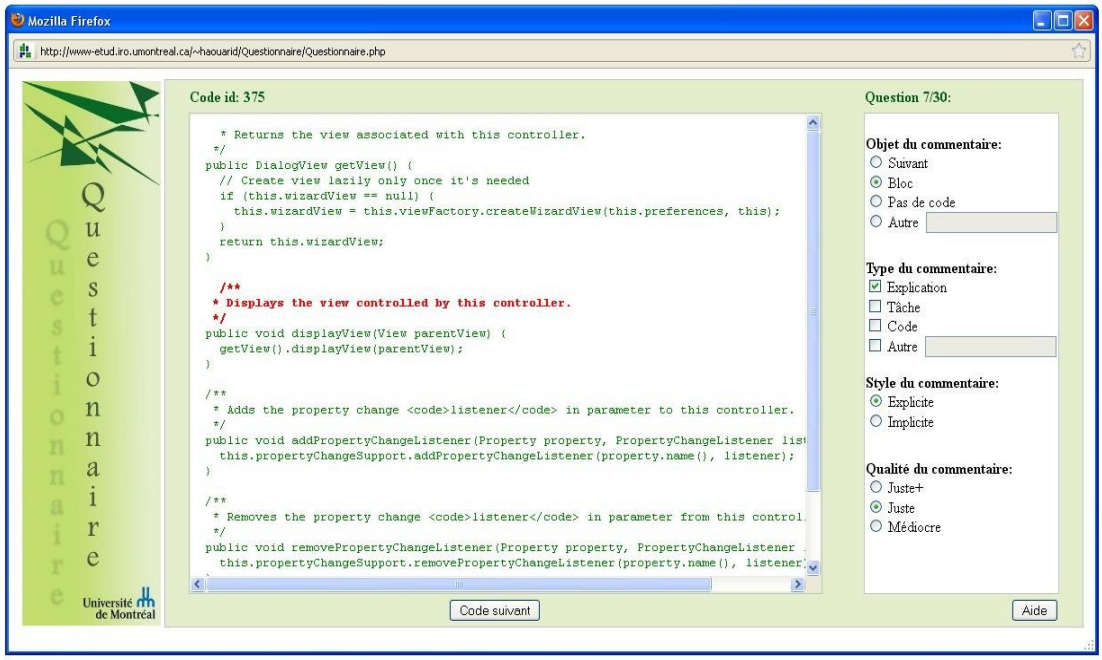


Figure 6 - Le formulaire web

3.3.2 Taxonomie des commentaires

Avant de commencer l'étude, il était important de définir une taxonomie pour guider les sujets lors de la tâche d'annotation. Le rôle de la taxonomie est de caractériser le contenu et la pertinence des commentaires. Nous avons ainsi examiné un échantillon aléatoire de

commentaires et nous avons relevé certaines dimensions que nous jugeons importantes pour l'analyse du contenu et la pertinence des commentaires. La taxonomie résultante est la suivante :

- **Objet du commentaire** (*objet*): cette dimension distingue l'instruction qui est en rapport avec le commentaire. Intuitivement, les développeurs ont tendance à mettre les commentaires avant les instructions qu'ils veulent décrire. Par conséquent, nous proposons les catégories suivantes pour caractériser l'objet du commentaire :
  - `suivant` indique que le commentaire concerne l'instruction suivante.
  - `bloc` représente le cas où le commentaire est en rapport avec le bloc d'instructions suivantes.
  - `pas de code` correspond au cas où le commentaire ne concerne aucune instruction.
  - `autre` correspond aux autres situations telles que le code commenté précède le commentaire. Il est à noter cependant qu'il aurait été intéressant de définir une catégorie «même ligne» pour caractériser les commentaires qui se placent sur la même ligne que le code commenté.
- **Type du commentaire** (*type*) : les types de commentaires possibles sont :
  - `explication` correspond au cas où le commentaire décrit les fonctionnalités du code en rapport.
  - `tâche`, si le commentaire décrit les tâches futures à effectuer (tels que les commentaires TODO) ou bien encore si le commentaire est en rapport avec du code mais ne fournit pas d'information sur ses fonctionnalités (telles que les pré-conditions des boucles).
  - code commenté (`code`) correspond au vieux code qui est souvent commenté au lieu d'être supprimé.
  - tout autre type (`autre`) telle que la licence et les commentaires de crédit.

  Il est important de noter qu'un commentaire peut avoir plusieurs types. Tel est le cas par exemple du commentaire dont une partie explique l'instruction suivante (`explication`) et une autre partie décrit les tâches futures à effectuer (`tâche`).

- **Style du commentaire** (*style*) : cette dimension est spécifique aux commentaires explicatifs (type explication). Nous distinguons les catégories explicite et implicite qui caractérisent respectivement les situations où le commentaire est écrit en terme de mots clés et d'identifiants de l'instruction ou en terme de mots plus abstraits (voir les exemples dans le tableau 4).
- **Qualité du commentaire** (*qualité*) : cette dimension est aussi spécifique aux commentaires explicatifs et elle inclut trois catégories :
  - juste indique si les fonctionnalités du code sont décrites adéquatement.
  - juste+ désigne un commentaire qui présente d'autres informations en plus des fonctionnalités.
  - médiocre caractérise les situations où quelques ou aucune des fonctionnalités du code sont décrites.

Le tableau 4 présente quelques commentaires et leur classification selon la taxonomie proposée. Par exemple, le premier commentaire est annoté comme un commentaire d'explication de style explicite et qui commente convenablement l'instruction qui le suit. Le quatrième exemple est un commentaire de tâche dont l'objet correspond à l'instruction qui le précède. Finalement, le dernier exemple montre un cas de code commenté.

3.3.3 Analyse

Le tableau 5 rapporte la distribution des commentaires selon les catégories de la taxonomie proposée. Nous observons que 73% (22% suivant + 51% bloc) des commentaires sont dédiés aux instructions qui les suivent. Ceci confirme l'intuition que nous avons mentionnée précédemment sur le fait que les développeurs ont tendance à mettre les commentaires juste avant les instructions qu'ils visent décrire. Aussi, nous observons que la majorité des commentaires sont de type explication (71%). Moins de 20% des commentaires sont annotés comme commentaires de tâche. Il est aussi remarquable que la plus part du temps les commentaires explicatifs sont écrits d'une manière explicite (80%). Finalement, plus que les deux tiers (56% + 15%) sont de bonne qualité.

Tableau 4 - Exemples de classification de commentaires

// if the document was an auxiliary file,
// remove it from the list
if (doc.isAuxiliaryFile())
removeAuxiliaryFile(doc);
	suivant
	explication
	explicite
	juste

/** A state variable indicating whether the class path has changed. Reset to false by resetInteractions. */ private volatile boolean classPathChanged = false;	suivant explication explicite juste+

_tokens = new TokenList();
$cursor = _tokens.getIterator();
// we should be pointing to the head of the list
_cursor.setBlockOffset(0);
	suivant
	explication
	implicite
	médiocre

ServiceManager.setServiceManagerStub(
new StandaloneServiceManager(
applet.getAppletContext(), codeBase, this.name));
// Caution: setting a new service manager stub
// won't replace the existing one
	autre: précédent
	tâche

// _mainFrame.hourglassOff();
disableChangeListeners();
_mainFrame.toFront();
	pas de code
	code

Tableau 5 – Distribution des commentaires selon les catégories de la taxonomie (%)

Dimensions	Catégories	%
Objet	suivant	22,0
	bloc	51,0
	pas de code	17,0
	autre	10,0
type	explication	71,0
	tâche	17,9
	code	9,3
	autre	8,8
style	explicite	80,0
	implicite	20,0
qualité	juste	56,0
	juste+	15,0
	médiocre	29,0

La distribution de chaque catégorie de la taxonomie pour les différents types est rapportée dans le tableau 6. D'une manière globale, nous observons que les pratiques de la documentation varient selon le type de l'instruction commentée. Dans ce qui suit, nous présentons les résultats relatifs à chaque dimension de la taxonomie.

Tableau 6 - Distribution des commentaires selon les catégories de la taxonomie en tenant en compte des types d'instructions commentées

	Objet (%)				Type (%)				Style (%)		Qualité (%)		
	suivant	bloc	pas de code	autre	explication	tâche	code	autre	explicite	implicite	juste	Juste+	médiocre
décl. package/import	0,0	5,7	83,3	11,0	0,0	27,8	5,5	66,7					
décl. interface/classe	0,0	88,9	7,4	3,7	81,5	25,9	3,7	14,8	63,6	36,4	50,0	27,3	22,7
décl. constante/attribut	52,6	34,2	7,9	5,3	76,3	21,1	0,0	5,3	62,1	37,9	51,7	6,9	41,4
méthode abstraite/de classe	16,8	80,5	2,7	0,0	93,8	8,0	0,0	4,4	87,7	12,3	62,3	17,9	19,8
constructeur	0,0	100,0	0,0	0,0	100,0	0,0	0,0	0,0	83,3	16,7	75,0	16,7	8,3
structure de contrôle	22,5	45,0	15,0	17,5	67,5	20,0	15,0	0,0	74,1	25,9	44,4	11,2	44,4
variable locale	23,5	55,9	14,7	5,9	76,5	20,6	14,7	2,9	76,9	23,1	61,5	3,9	34,6
affectation	18,2	18,2	54,5	9,1	18,2	45,4	36,4	0,0	100,0	0,0	50,0	0,0	50,0
appel de méthode	32,6	19,6	26,1	21,7	52,2	28,3	21,7	6,5	83,3	16,7	50,0	12,5	37,5
return	44,4	0,0	44,4	11,2	44,4	11,2	33,3	11,1	0,5	0,5	0,5	0,25	0,25

3.3.3.1 Objet des commentaires

Le premier aspect que nous avons étudié est l'objet de commentaires. Les commentaires se plaçant avant les déclarations de classes/interfaces et les déclarations de membres concernent en général les instructions qu'ils précèdent. Ceci est particulièrement le cas des constructeurs où 100% des commentaires ont pour objet le bloc des instructions suivantes.

Dans plusieurs cas, un seul commentaire est écrit pour un ensemble de déclarations d'attributs et de constantes au lieu d'un commentaire par déclaration. Ceci explique la grande portion dédiée au bloc suivant pour ces instructions (34,2%). Le commentaire 1 représente un exemple pour ce cas :

Commentaire 1 (bloc+explication+explicite+juste)

```
/** Color for highlighting find results. */
public static final ColorOption FIND_RESULTS_COLOR1 =
new ColorOption("find.results.color1", new Color(0xFF, 0x99, 0x33));
public static final ColorOption FIND_RESULTS_COLOR2 =
new ColorOption ("find.results.color2", new Color (0x30, 0xC9, 0x96));
```

Contrairement à ce qui précède, les commentaires qui se trouvent avant les affectations, les appels de méthodes et les instructions `return` sont moins liés à ces instructions. Plusieurs d'entre eux sont de l'ancien code transformé en commentaire ou bien des commentaires de tâche comme c'est le cas dans le commentaire 2.

Commentaire 2 (pas de code+tâche)

```
// TODO, maybe: remove playlist config file
// from file system as well. Maybe provide
// this as a user option (pop-up asking for
// confirmation or something). But for now,
// not.
nonFatalConfigError = true;
```

3.3.3.2 Type des commentaires

Le deuxième aspect que nous avons étudié est le type de commentaires. Le pourcentage cumulatif de cet aspect excède 100% à cause des choix multiples possibles. Les commentaires explicatifs sont plus fréquents que les commentaires de tâches à l'exception des déclarations de package, les imports et les affectations. La plupart des commentaires précédant les déclarations de package sont des copyrights ou des informations de crédits. Les instructions `import` sont généralement précédées par des commentaires de tâche, comme c'est illustré dans le commentaire 3.

Commentaire 3 (pas de code+tâche)

```
import java.awt.*;
// TODO: Check synchronization.
import java.util.Vector;
```

Il y a très peu de commentaires explicatifs avant les instructions d'affectation. La majorité est de type commentaire de tâche. Au contraire, tous les commentaires précédant les constructeurs sont de type explication. Cependant, il est important de mentionner que l'échantillon des commentaires étudié inclut 3,7% seulement de commentaires liés aux constructeurs mais qui sont tous annotés comme descriptifs de fonctionnalités. Finalement, comme prévu, les commentaires contenant du vieux code se trouvent essentiellement dans le corps des méthodes (affectation 36%, return 33%, appels de méthode 22%, etc.). Les anciennes déclarations de classe (3%) et de membres (0%) sont d'habitude enlevées au lieu d'être commentées.

3.3.3.3 Style de commentaires

Comme mentionné précédemment, les sujets ont analysé le style (explicite vs implicite) pour les commentaires de type explication seulement.

Un résultat intéressant est que les déclarations de méthode incluant les méthodes abstraites ainsi que les constructeurs sont généralement expliquées explicitement. Quand une méthode retourne une valeur, le commentaire associé décrit souvent la valeur retournée comme c'est le cas du commentaire 4.

Commentaire 4 (bloc+explication+explicite+juste)

```
/** Returns the current build directory in the project profile. */
private File _getBuildDir() {...}
```

Concernant les méthodes sans type de retour (void), le commentaire explique typiquement la tâche effectuée par la méthode, tel qu'illustré par le commentaire 5.

Commentaire 5 (bloc+explication+explicite+juste)

```
/** Displays this panel in a dialog box. */
public void displayView(View parentView) {...}
```

Il y a une grande portion de commentaires précédant les classes, les attributs et les déclarations de constantes qui sont écrits d'une manière implicite (plus qu'un tiers). Les commentaires implicites se limitent à expliquer où les attributs sont utilisés sans mentionner comment. Le commentaire 6 en montre un exemple.

Commentaire 6 (suivant+explication+implicite+médiocre)

```
// used for playlist searches
ConditionalPlaylist ownerPlaylist = null;
```

Dans d'autres cas, les classes, les attributs et les constantes sont commentés d'une manière explicite, tel que dans le cas du commentaire 7.

Commentaire 7 (suivant+explication+explicite+juste)

```
/** Edit mode if true. */
protected boolean _editMode = false;
```

Le dernier résultat trouvé montre qu'un quart des commentaires de structures de contrôle sont implicites. Lors de l'examen de ces commentaires, nous remarquons que seule la condition de test est expliquée et que les tâches faites par les boucles et les blocs conditionnels ne le sont pas, voir le commentaire 8 pour un exemple.

Commentaire 8 (autre+explication+implicite+médiocre)

```
// keys() may return null!
if (im.keys()!= null) {...}
```

3.3.3.4 Qualité des commentaires

La quatrième et la dernière dimension que nous avons étudiée est la qualité des commentaires explicatifs. La qualité des instructions d'affectation a attiré notre attention. Nous avons mentionné précédemment que 100% des commentaires d'affectation sont explicites. Toutefois, une inspection plus précise montre que la moitié d'entre eux sont d'une qualité médiocre. Les structures de contrôle aussi ont beaucoup de commentaires médiocres (44%). En parallèle, les méthodes et les constructeurs sont souvent bien expliqués. D'une manière générale, il y a plus de commentaires d'une qualité médiocre dans le corps des méthodes que dans les déclarations de classes et des membres, à l'exception des déclarations de constantes et d'attributs. Pour ces derniers, le commentaire est très bref et se limite à mentionner le nom ou le type de l'attribut/constante. Les commentaires 9 et 10 illustrent des exemples de ces cas.

Commentaire 9 (suivant+explication+explicite+médiocre)

```
/** Extra class path. */
public static final VectorOption<File>
EXTRA_CLASSPATH = new ClassPathOption()
                                .evaluate("extra.classpath");
```

Commentaire 10 (suivant+explication+implicite+médiocre)

```
/** Frame state. */
protected FrameState _lastState = null;
```

La dernière observation à mentionner est qu'un nombre très limité de commentaires est trop descriptif. Bien qu'il soit difficile de mettre une séparation claire entre les commentaires descriptifs et les commentaires trop descriptifs, les commentaires longs sont généralement considérés comme trop descriptifs. Un exemple de ces commentaires est illustré par le commentaire 11.

Commentaire 11 (bloc+explication+explicite+juste+)

```
/**
* A Standard Playlist may have multiple
* instances of the same song, and the Wrapper
* class allows these to be placed in the
* model.
* This method locates all of these wrapped
* instances of a particular song
* and returns all of them in a Vector
* @param song
* @return a Vector containing all the wrapped
* instances of the supplied song, sorted in
* order from least to greates index in the
* Playlist
*/
private Vector<Wrapper>
getAllWrappedInstancesOfSong (AbstractSongInfo song){...}
```

3.3.4 Étude subjective

En plus de la tâche de classification, les sujets ont répondu à un questionnaire. Le questionnaire contient trois types de questions concernant le sujet : (1) expérience en programmation, (2) habitudes de documentation et (3) réaction envers leur participation. Nous discuterons du premier et du troisième type de questions dans la section Étude de la validité. Dans cette section, nous discuterons seulement des données collectées pour les questions du deuxième type. Nous étudions ces données afin de comparer et contraster certains des résultats observés lors de l'étude 2 avec les habitudes de documentation exprimées par les sujets. Les deux premières questions concernent la fréquence et l'emplacement des commentaires dans le code. Comme il est montré dans la figure 7-gauche, seulement un tiers des sujets affirment qu'ils commentent systématiquement le

code. Les autres (presque les deux tiers) ne le font qu'occasionnellement. Quand ils écrivent des commentaires, presque chaque sujet les place avant l'instruction à commenter (82% comme indiqué dans la figure 7-droite). Ce résultat confirme notre choix d'affecter les commentaires à l'instruction suivante. Il est aussi conforme aux résultats observés dans l'étude 2 (voir tableau 5).

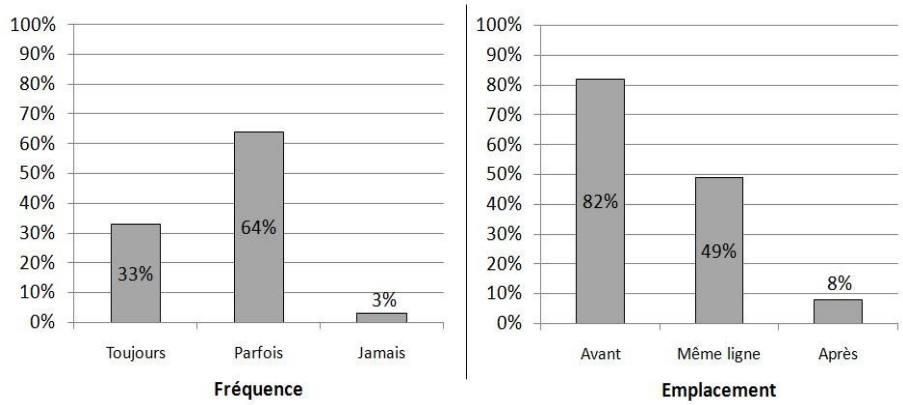


Figure 7 - Fréquence et emplacement des commentaires

En demandant aux sujets de donner les fréquences avec lesquelles ils commentent les différents types d'instructions, certaines des réponses ont confirmé les résultats du tableau 6, d'autres étaient différentes. En effet, comme il est montré dans la figure 8, nous trouvons que les méthodes (incluant les constructeurs) sont fréquemment commentées (plus que 80% pour toujours + souvent). Ce résultat est conforme à ce que nous avons trouvé lors de l'étude 2. Aussi, les instructions les moins commentées sont les même dans l'évaluation subjective et dans l'étude 2 (affectation, return, etc.). Cependant, nous étions surpris par le taux bas de documentation pour les packages et à un degré moindre pour les classes. Dans notre échantillon de commentaires analysés, presque tous les packages sont commentés (99% en moyenne). Dans l'évaluation subjective, seulement 20% des sujets ont choisi toujours ou souvent. Puisque les commentaires précédant les packages ne sont pas explicatifs, il est possible que le nombre de 20% correspond au cas où les sujets mettaient les copyrights comme commentaires de package.

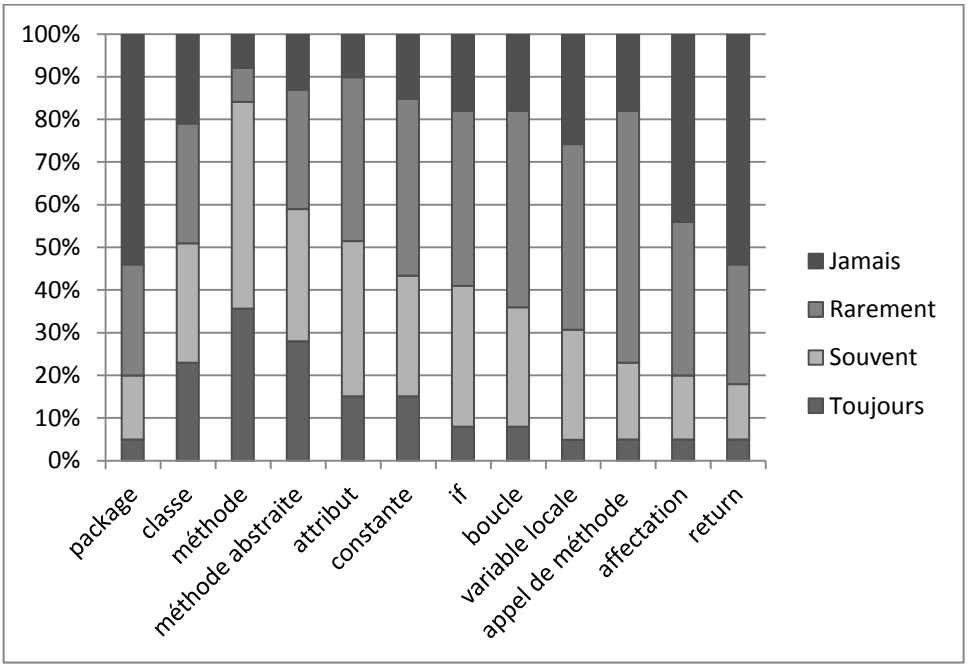


Figure 8 - Fréquence de documentation des types d'instruction

La dernière question concerne le contenu des commentaires. Nous avons demandé aux sujets de donner le type de leurs commentaires en sélectionnant un ou plusieurs des choix suivants : explication, tâche ou autres. Les résultats sont représentés dans la figure 9. Les résultats étaient similaires à ceux présentés dans le tableau 5 et le tableau 6. Comme nous pouvons le constater d'après la figure 9, les commentaires d'explication sont les plus sélectionnés (90%). Les commentaires de tâche sont mentionnés par le tiers des sujets. Cette valeur est légèrement plus grande que celle indiquée dans le tableau 5, mais plus proche des valeurs obtenues pour les différents types d'instructions du tableau 6. D'une manière globale, les résultats de l'évaluation subjective confirment les observations produites à partir de l'étude de l'échantillon de commentaires.

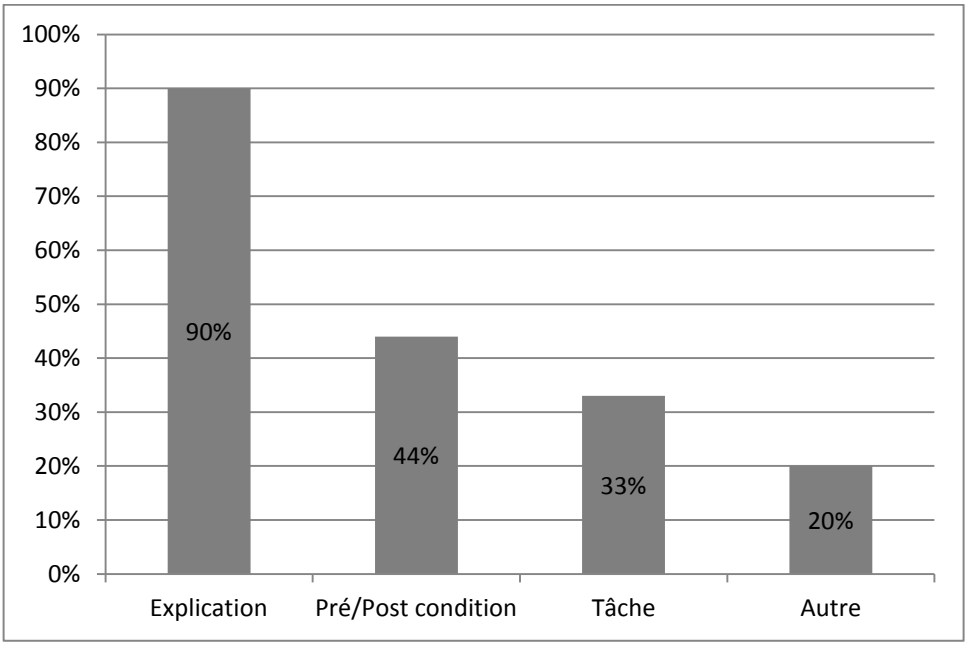


Figure 9 - Type des commentaires

3.3.5 Étude de la validité

La validité réfère à la préoccupation d'un chercheur de produire des résultats qui contribuent à mieux comprendre une réalité, un phénomène [24]. Dans ce qui suit nous traitons les différents problèmes qui peuvent affecter la validité de l'étude effectuée.

3.3.5.1 Validité interne

La validité interne est un indicateur qui permet de vérifier que les résultats obtenus sont l'effet du phénomène étudié et non pas d'autres biais. Nous avons identifié deux menaces possibles à la validité interne: la maturation et la diffusion des traitements. La maturation correspond aux effets qui peuvent surgir à cause du passage du temps. Dans notre cas, nous avons traité particulièrement l'effet de fatigue et d'apprentissage en présentant les commentaires aux sujets dans un ordre aléatoire et différent. Nous avons aussi essayé de réduire l'effet de fatigue en limitant le nombre de fragments de code présentés aux sujets à trente. Nous sommes conscients que ce nombre reste élevé. Plusieurs sujets ont mentionné

en répondant à la partie " réaction envers la participation " du questionnaire que la durée de la tâche était un peu élevée.

La menace de la diffusion des traitements peut survenir quand un groupe de personnes est soumis à un traitement commun. Si ces personnes communiquent les informations du traitement entre elles, elles risquent de reproduire certains de leurs comportements. Par conséquent, leurs réponses seront influencées. Pour prévenir cette menace, nous avons donné des instructions aux sujets pour qu'ils ne parlent pas de l'expérience avant la fin de la collecte des données.

3.3.5.2 Validité du construit

La validité du construit vise l'établissement de mesures opérationnelles correctes pour les concepts théoriques étudiés [24]. Dans notre cas, afin de nous assurer de la précision de la classification de commentaires effectuée par les sujets, nous avons considéré trois opinions par commentaire et nous les avons agrégées grâce à un mécanisme de vote. Pour les cas où aucun accord entre les réponses n'a été trouvé, une solution était de décider des valeurs finales à l'aide de jugement personnel mais nous avons préféré plutôt que de trancher de supprimer les fragments de code ayant trois opinions divergentes.

3.3.5.3 Validité externe

La validité externe est un indicateur de la généralisation des résultats. La sélection des sujets et des fragments de code peut présenter des menaces. Concernant les sujets, quelques personnes sont des étudiants. Bien qu'ils ne soient pas des programmeurs professionnels, la plupart d'entre eux ont une expérience et des connaissances industrielles comparables à des professionnels juniors. En effet, les réponses à la partie " expérience professionnelle " du questionnaire confirme que 59% des sujets sont de bons programmeurs, i.e., une expérience de deux à cinq années de programmation, et que 41% des sujets sont des experts. Les sujets ont particulièrement de l'expérience en Java (69% bons en Java et 31% experts). La sélection des fragments de code est faite en deux étapes. En premier lieu, nous avons choisi de faire l'étude sur trois projets en considérant la variabilité de leurs tailles (20 KLOC à

145 KLOC), leurs domaines et le nombre de développeurs impliqués (de 1 à 55 développeurs). Ensuite, nous avons utilisé un échantillonnage par strates à la place d'un échantillonnage aléatoire pur. Ce type d'échantillonnage utilisé permet d'obtenir un échantillon représentatif de la distribution des commentaires observée lors de l'étude 1. Toutefois, le fait que nous avons éliminé les fragments de code qui ont des opinions divergentes et ceux des sujets qui n'ont pas terminé leur tâche peut compromettre la représentativité de la distribution. Nous avons vérifié la distribution finale et nous n'avons pas trouvé une différence significative avec la distribution initiale.

3.4 Conclusion

Dans ce travail, nous avons fait une étude empirique de l'emplacement des commentaires, de leur contenu et de leur pertinence. Nous avons proposé une taxonomie pour guider les sujets durant la tâche de classification. L'étude couvre trois parties : quantitative, qualitative et subjective. Dans la partie quantitative, nous nous sommes intéressés à la distribution des commentaires par rapport aux différents types d'instructions et à la fréquence de documentation de chaque type. La partie qualitative traite de l'objet du commentaire, le type, le style et la qualité. Finalement, la partie subjective concerne les habitudes de documentation des sujets. Nous avons trouvé des résultats consistants entre les différentes études. Les résultats montrent que certains éléments tels que les méthodes sont commentés régulièrement pour des raisons explicatives. Aussi, nous avons trouvé qu'une portion importante des commentaires est dédiée à la communication entre programmeurs et à des notes de futurs changements à effectuer. Cette étude donne une image intéressante sur l'emplacement et le contenu des commentaires dans le code.

Cependant, d'autres réplications avec plus de sujets et de commentaires dans d'autres projets tels qu'eclipse ou apache sont nécessaires pour confirmer et compléter cette image. Aussi, il serait intéressant d'ajouter la catégorie « *même ligne* » à la dimension objet de la taxonomie des commentaires proposée. Cette catégorie permettra de caractériser les commentaires qui sont écrits sur la même ligne du type commenté.

Chapitre 4

Exploration d'heuristiques d'extraction pour la redocumentation

4.1 Introduction

Dans ce chapitre, nous explorons l'application de techniques de résumé par extraction pour la génération de la documentation. En effet, nous définissons des heuristiques d'extraction pour documenter des classes écrites dans le langage Java. Ensuite, nous explorons l'application de ces heuristiques sur les commentaires des méthodes pour résumer les classes. Ce chapitre est organisé en deux sections. La première section présente l'idée générale de l'approche ainsi que les heuristiques d'extraction proposées. La deuxième section présente l'évaluation des résumés générés par les heuristiques et la discussion des résultats obtenus.

4.2 Approche proposée

4.2.1 Idée générale et justification des choix

Dans la partie dédiée au calcul de la fréquence des commentaires par type d'instructions de l'étude empirique du chapitre 3, nous avons observé que les classes sont commentées dans 94,3% des cas (voir le tableau 3 du chapitre précédent). Ceci montre l'intérêt particulier que portent les programmeurs à commenter ces entités.

Toutefois nous croyons que la production de résumés de classes non documentées est néanmoins utile. Elle apporte, en effet, une aide précieuse aux développeurs principalement pendant les tâches de maintenance. Par ailleurs, une classe déclare les propriétés communes à un ensemble d'objets. Elle possède une structure interne et un comportement. Par

conséquent, elle présente un niveau de granularité assez consistant. Pour ces raisons, nous nous intéressons dans ce chapitre à la redocumentation des classes.

Pour ce faire, nous nous basons sur l'extraction des commentaires des méthodes et des constructeurs. En effet, dans la partie qui s'intéresse à l'étude du contenu et de la pertinence des commentaires de l'étude empirique, nous avons remarqué que la majorité des commentaires des méthodes et des constructeurs sont des commentaires pertinents (voir le tableau 6 du chapitre précédent). Plus de 90% des commentaires des méthodes et des constructeurs sont explicatifs. De plus, ces commentaires sont bien écrits : puisque 80% d'entre eux sont écrits avec un style explicite et possèdent une bonne qualité.

Cependant, il est à noter que l'approche proposée est généralisable. Son application ne se limite pas à la documentation de classes. Elle peut être appliquée à d'autres niveaux en respectant le principe que les commentaires des concepts de bas niveaux peuvent aider dans la documentation des concepts de plus haut niveau. Par exemple, il est possible de générer des résumés de méthodes en faisant l'extraction des commentaires présents dans leurs corps.

4.2.2 Heuristiques d'extraction

Les commentaires des méthodes et des constructeurs présentent une quantité importante d'information en langue naturelle. En particulier, les commentaires des méthodes sont très nombreux. L'étude empirique a montré que les commentaires des méthodes représentent un tiers du nombre total des commentaires des trois projets étudiés (voir le tableau 2 du chapitre précédent). Cependant, sont-ils tous pertinents pour former le résumé de la classe? Quels choix doit-on faire lors de l'extraction des commentaires pour la génération d'une documentation de bonne qualité?

Nous lions la pertinence d'un commentaire à l'importance de la méthode qu'il commente. En effet, une classe possède plusieurs méthodes et il est évident que certaines méthodes sont plus abstraites que d'autres. Plus la méthode est abstraite, plus elle décrit le comportement principal de la classe et plus elle est importante pour la classe. En

conséquence, nous proposons de faire l'extraction des commentaires des méthodes les plus importantes à la classe. Pour ce faire, nous suggérons les heuristiques suivantes:

- *All methods* (AM): extrait les commentaires de toutes les méthodes et de tous les constructeurs de chaque classe à documenter. Comme nous l'avons déjà vu dans l'étude empirique, la majorité des commentaires qui précèdent les méthodes/constructeurs sont pertinents. Ils sont par conséquent de bons candidats à l'extraction. Contrairement aux autres heuristiques, AM est une heuristique naïve qui ne fait pas de choix d'extraction. Nous nous attendons à ce que ses résultats soient inférieurs à ceux des autres heuristiques. Nous la considérons comme une heuristique référence (baseline).
- *Public Methods* (PM): extrait les commentaires des méthodes et des constructeurs à visibilité publique (ceux qui portent le modificateur `public`). Les méthodes/constructeurs publiques d'une classe sont accessibles par les autres classes. Ils offrent des fonctionnalités aux classes qui les utilisent. Nous faisons l'hypothèse que ces membres publics sont importants puisque les opérations et les services rendus sont utiles pour les autres classes. Par conséquent, nous proposons l'extraction des commentaires qui les précèdent.
- *N First Methods* (NFM): extrait les commentaires des N méthodes et constructeurs définis au début de la classe. Cette heuristique se base sur l'hypothèse que les programmeurs écrivent les membres les plus importants en haut des classes.
- *N Last Methods* (NLM): extrait les commentaires des N méthodes et constructeurs définis à la fin de la classe. Cette heuristique part de l'hypothèse que les programmeurs écrivent les membres les plus importants à la fin des classes.
- *N Lines* (NL): extrait les N premières lignes de commentaires. Cette heuristique part de l'hypothèse que les premières lignes de commentaires sont pertinentes aux résumés des classes.
- *Distance to Main* (DMainN): extrait les commentaires des N plus proches méthodes et constructeurs de la méthode principale `main` du programme à résumer. Nous faisons l'hypothèse que ces méthodes/constructeurs sont importants et par

conséquent leurs commentaires le sont aussi. Nous commençons d'abord par la génération d'un graphe d'appels statique où les nœuds représentent les méthodes/constructeurs et les arcs représentent les appels entre ces derniers. Ensuite, nous calculons la distance séparant chaque nœud du nœud racine représentant la méthode principale du projet (méthode `main` qui s'exécute en premier).

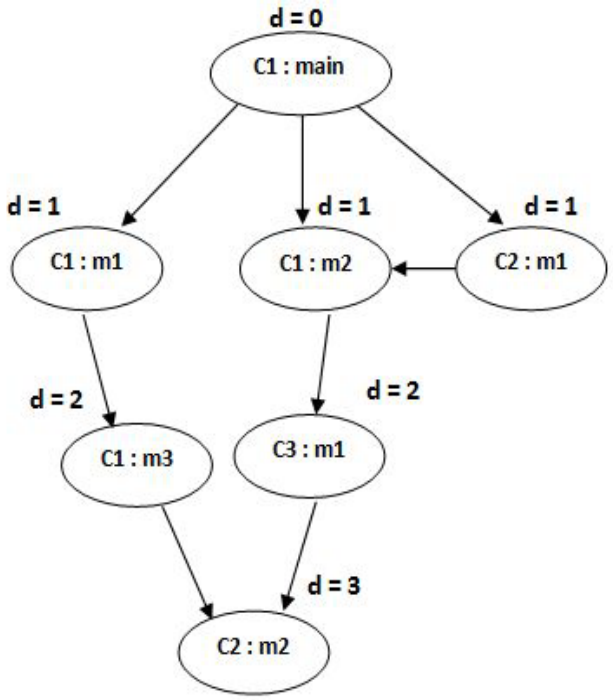


Figure 10 - Exemple d'un graphe d'appels

La figure 10 montre un exemple d'un graphe d'appels simplifié d'un programme qui comporte trois classes C1, C2 et C3. C1 possède quatre méthodes m1, m2, m3 et la méthode principale du programme `main`. C2 possède deux méthodes m1 et m2. C3 possède une seule méthode m1. Le calcul de la distance par rapport au `main` est fait tout au long du parcours du graphe. Par exemple, la méthode m1 de C1 est distante de 1 par rapport au `main`, la méthode m2 de C2 est distante de 3 par rapport au `main`, etc. Il est à noter que le graphe d'appels représente les liens

d'appel des méthodes/constructeurs de l'ensemble des classes d'un projet. Ainsi, pour résumer une classe, nous sélectionnons les commentaires des méthodes de la classe en question. Par exemple, pour résumer la classe C1 par application de DMain2, l'heuristique sélectionne les commentaires de (C1 : main + C1 : m1) ou bien (C1 : main + C1 : m2). Nous conservons un seul choix pour le résumé.

Par ailleurs, nous proposons de supprimer les tags javadoc et leur contenu des commentaires extraits par toutes les heuristiques. Les tags permettent aux développeurs de préciser les valeurs de certains éléments de l'unité décrite et ce, de façon standard. Les méthodes possèdent des tags qui permettent de définir des propriétés qui leur sont spécifiques. Par exemple, le tag @param décrit le nom d'un paramètre passé en argument à une méthode et en présente une description brève, le tag @throws documente une exception lancée par une méthode, le tag @return documente la valeur de retour d'une méthode. Ces tags sont propres aux méthodes et ils ne peuvent pas être utilisés dans les commentaires des classes. D'autres tags sont définis pour décrire les propriétés des classes tels que @author et @version qui permettent de préciser respectivement le nom du développeur et la version de la classe. Par ailleurs, d'autres tags sont communs à tous les commentaires Javadoc comme le tag @since et le tag @todo. Ces tags peuvent être utilisés dans les commentaires des interfaces, des classes, des méthodes, des constructeurs et des attributs. Le tag @since permet de préciser depuis quelle version Java l'élément commenté a été ajouté, par exemple @since 1.2 indique que l'élément est défini dans la version Java 1.2. Le tag @todo permet aux développeurs de prendre notes des tâches à effectuer dans l'avenir. Dans tous les cas, que les tags soient propres à un type particulier ou communs aux commentaires Javadoc, ils sont utilisés pour décrire des "propriétés" spécifiques à l'élément commenté. Vu cette spécificité, il n'est pas intéressant d'inclure les tags des méthodes/constructeurs et leurs contenus dans les résumés générés pour documenter les classes. Aussi, ils sont supprimés des résumés et ce pour toutes les heuristiques appliquées.

4.3 Évaluation

Dans cette section, nous évaluons les heuristiques d'extraction proposées. Pour ce faire, nous utilisons un corpus formé par un ensemble de résumés candidats et de résumés de référence. Les résumés candidats sont générés par application des heuristiques d'extraction sur les classes Java à résumer. Les résumés de référence sont des résumés écrits manuellement. L'évaluation consiste à comparer les résumés candidats aux résumés de référence. Plus un résumé candidat ressemble au résumé de référence, plus il est apprécié.

Il est possible de faire cette comparaison manuellement ou automatiquement. L'évaluation manuelle est coûteuse. Elle nécessite des humains pour comparer les résumés candidats et les résumés de référence. Par contre, l'évaluation automatique est moins coûteuse et plus rapide.

Dans ce travail, nous utilisons la métrique ROUGE[16] (*Recall-Oriented Understudy for Gisting Evaluation*) pour faire une évaluation automatique des résumés produits par les heuristiques.

Dans la section qui suit, nous détaillons les éléments nécessaires au processus de l'évaluation et nous présentons les résultats obtenus.

4.3.1 Métrique ROUGE

ROUGE est proposé par Lin [16]. Il s'agit d'un package utilisé pour l'évaluation automatique des résumés de documents. Il est utilisé dans la conférence *Document Understanding Conference* depuis 2004. ROUGE procède à l'évaluation d'un résumé généré automatiquement en le comparant à un ou plusieurs résumés de référence produits manuellement. ROUGE comporte plusieurs variantes. Dans ce qui suit, nous nous baserons sur les articles [16] [15] de Lin pour présenter les différentes variantes de ROUGE.

[16] http://berouge.com/default.aspx

4.3.1.1 ROUGE-N

ROUGE-N se base sur le calcul de co-occurrence de N-grammes (séquence de N mots) dans les résumés de référence et dans le résumé à évaluer. Le rappel est calculé par ROUGE-N selon la formule suivante:

$$ROUGE - N = \frac{\sum_{S \in \{\text{résumés références}\}} \sum_{\text{gram}_n \in S} \text{Count}_{\text{match}}\ (\text{gram}_n)}{\sum_{S \in \{\text{résumés références}\}} \sum_{\text{gram}_n \in S} \text{Count}\ (\text{gram}_n)}$$

ROUGE-N divise le nombre de N-grammes communs aux résumés de référence et au résumé à évaluer par le nombre des N-grammes des résumés de référence. ROUGE-N permet d'estimer la fluidité des résumés pour N supérieur à 1. L'inconvénient de ROUGE-N est qu'il peut attribuer le même score à deux phrases qui ont des significations différentes. Par exemple, considérons l'exemple suivant:

S1 (*Référence*): police killed the gunman

S2 (*Candidate*): police kill the gunman

S3 (*Candidate*): the gunman kill police

où S1 est le résumé de référence et S2 et S3 sont les résumés candidats à évaluer. L'application de ROUGE-2 aux résumés candidats résulte dans l'attribution du même score à S2 et à S3 bien qu'elles soient sémantiquement différentes. Les deux phrases possèdent un seul bigramme commun avec S1 qui est the gunman et elles auront un score égal à 1/3.

4.3.1.2 ROUGE-L

ROUGE-L se base sur le calcul de la plus longue sous séquence commune à deux résumés. La plus longue sous séquence commune (LCS: Longest Common Subsequence) à deux séquences X et Y est une séquence commune ayant une longueur maximale. ROUGE-L attribue un score à deux séquences candidates en partant de l'intuition que plus la LCS est longue, plus les deux séquences sont similaires. ROUGE-L calcule le ratio de la longueur de la LCS du résumé candidat et du résumé de référence sur la longueur du résumé de référence. La longueur est mesurée en nombre de mots. Contrairement à ROUGE-N, ROUGE-L prend en considération les correspondances non consécutives de

mots. Par application de ROUGE-L sur l'exemple précédent, S2 obtient un score égal à 0,75. Le score (Rappel) est calculé de la façon suivante:

$$ROUGE - L(S2) = \frac{\text{longueur}\big(\text{LCS}(\text{S1}, \text{S2})\big)}{\text{longueur}(\text{S1})}$$
$$= \frac{\text{longueur}(\ police\ the\ gunman)}{\text{longueur}(\ police\ killed\ the\ gunman)}$$
$$= \frac{3}{4}$$
$$= 0{,}75$$

S3 obtient un score égal à 0,5 calculé selon la formule suivante:

$$ROUGE - L(S3) = \frac{\text{longueur}\big(\text{LCS}(\text{S1}, \text{S3})\big)}{\text{longueur}(\text{S1})}$$
$$= \frac{\text{longueur}(the\ gunman)}{\text{longueur}(\ police\ killed\ the\ gunman)}$$
$$= \frac{2}{4}$$
$$= 0{,}5$$

Contrairement à ROUGE-2, ROUGE-L favorise S2 qui est plus proche du sens de S1.

Néanmoins, ROUGE-L ne considère que la plus longue sous séquence commune et néglige les sous séquences communes courtes ainsi que les autres alternatives de LCS. Considérons par exemple le résumé candidat suivant:

S4 (*Candidat*): the gunman police killed

ROUGE-L considère soit la LCS the gunman ou bien police killed. Par conséquent, par application de ROUGE-L, S4 a le même score que S3. À noter que contrairement à ROUGE-L, ROUGE-2 attribue à S4 un score plus élevé que celui de S3.

$$\text{ROUGE} - 2(\text{S4}) = \frac{\text{Count}_{\text{match}}(\ the\ gunman, police\ killed)}{\text{Count}(\ police\ killed, killed\ the, the\ gunman)}$$
$$= \frac{2}{3}$$
$$= 0{,}75$$

4.3.1.3 ROUGE-W

Contrairement à ROUGE-L, *ROUGE-W* (W: Weighted Longest Common Subsequence) tient compte de la position des LCS dans les séquences. Par exemple, étant donné X une référence et Y1 et Y2 deux séquences candidates:

X (*Référence*): [A B C D E F G]

Y1 (C*andidate*): [A B C D H I K]

Y2 (*Candidate*): [A H B K C I D]

Par application de ROUGE-L, Y1 et Y2 obtiennent le même score. Pourtant, Y1 devrait être meilleure qu'Y2 puisqu'il existe une correspondance consécutive entre ses mots et ceux de la référence X. ROUGE-W favorise les résumés ayant des correspondances consécutives de LCS et attribut à Y1 un score plus élevé que celui d'Y2.

4.3.1.4 ROUGE-S

ROUGE-S se base sur le calcul du nombre de co-occurrence de "skip-bigrams" dans les résumés de référence et le résumé à évaluer. Un skip-bigram est toute paire de mots respectant son ordre d'apparition dans une phrase. Par exemple, la phrase "police killed the gunman" possède 6 skip-bigrams qui sont (police killed, police the, police gunman, killed the, killed gunman et the gunman).

4.3.1.5 ROUGE-SU

ROUGE-SU est une extension de ROUGE-S. En plus des skip-bigrams, ROUGE-SU tient compte aussi des unigrammes. En effet, ROUGE-S attribut un score nul à toute phrase ne possédant pas de skip-bigrams en communs avec la référence, tel est le cas de la phrase suivante:

S5 (*Candidate*): gunman the killed police

S5 n'a pas de skip-bigrams partagés avec la référence, mais par contre, elle possède des unigrammes en commun avec cette dernière. ROUGE-SU permet d'attribuer un score non nul à de telles phrases.

4.3.2 Données de l'évaluation

Pour faire l'évaluation, nous appliquons ROUGE sur un corpus formé de résumés de référence et de résumés candidats. Les résumés de référence sont les commentaires originaux des classes des projets DrJava, SweetHome3D et jPlayMan pris en exemple dans le chapitre précédent. Nous avons vu dans l'étude empirique des commentaires que la majorité de ces commentaires sont pertinents. Ils sont écrits manuellement par les développeurs originaux du code. Ils représentent ainsi une base importante de résumés de référence auxquels nous suggérons de comparer les résumés candidats. Les exemples 1 et 2 sont des exemples de résumés de référence. Il est à noter que la taille des résumés de référence de notre corpus varie d'une ligne à 35 lignes. Le tableau 7 présente le nombre des classes en fonction du nombre de leurs lignes. Nous constatons que les programmeurs ont tendance à mettre des commentaires assez courts pour décrire les classes.

#lignes	1	2	3	4	5-9	10-14	35
#classes	268	87	34	10	17	10	1

Tableau 7 - Nombre de classes en fonction du nombre de lignes

Exemple 1 :

Résumé de référence :

Utility class which can tokenize a String into a list of String arguments, with behavior similar to parsing command line arguments to a program.
Quoted Strings are treated as single arguments, and escaped characters are translated so that the tokenized arguments have the same meaning.
Since all methods are static, the class is declared abstract to prevent instantiation.

Exemple 2 :

Résumé de référence :

Indents current line to the indent level of the previous line augmented by a suffix.

Quant aux résumés candidats, ils sont générés suite à l'application des heuristiques d'extraction sur les classes des 3 projets sus mentionnés. Au total, nous avons 667 classes. Chaque classe résumée obtient à la fin de cette opération six résumés candidats (un résumé par heuristique).

Étant donné que les heuristiques proposées extraient les commentaires des méthodes, il est inutile de traiter les classes qui n'ont aucune méthode commentée. Ces classes sont supprimées. Nous avons supprimé aussi les classes qui ne possèdent pas de commentaires ainsi que celles dont la qualité des commentaires n'est pas bonne. De plus, nous avons supprimé les classes dont aucune méthode n'est présente dans le graphe d'appels car il n'est pas possible d'appliquer l'heuristique DMain sur ce type de classes. Nous avons supprimé également les tags et leurs contenus des résumés de référence, comme @author, @version, etc.). Au total, nous avons supprimé 240 classes.

Par conséquent, nous obtenons 427 résumés de référence par extraction des commentaires originaux des classes et 427 résumés candidats par heuristique d'extraction.

Nous avons choisi d'appliquer NL avec N=10 (10L) parce qu'en moyenne chaque classe comporte 16 lignes de commentaires (nombre de lignes de commentaires / nombre de classes = 7222 / 427 = 16,91). Nous pensons ainsi que N=10 serait une valeur adéquate pour tester NL. Quant aux heuristiques NFM, NLM et DMainN, elles seront utilisées avec N=5 (5FM, 5LM et DMain5). En effet, les 427 classes traitées contiennent 5176 méthodes et constructeurs commentés. Chaque classe comporte donc 12 méthodes/constructeurs commentés en moyenne. Nous pensons que la spécification de N à la valeur 5 permet de distinguer l'importance des 5 premières méthodes choisies par l'heuristique DMainN des

autres méthodes. Ce choix permet aussi d'empêcher l'extraction des mêmes commentaires par NFM et NLM qui risquent de se chevaucher si N>6.

4.3.3 Mots vides

Lors de l'évaluation, il est important de ne pas considérer les mots vides (stopwords). Les mots vides sont des mots non significatifs. Chaque langue possède une liste de mots vides. Par exemple, *a*, *the* et *that* sont parmi les mots vides de la langue anglaise.

Dans notre cas, les résumés à évaluer sont en anglais. Par conséquent, nous appliquons ROUGE en ignorant la présence de mots vides spécifiques à l'anglais afin de ne pas biaiser les résultats de l'évaluation.

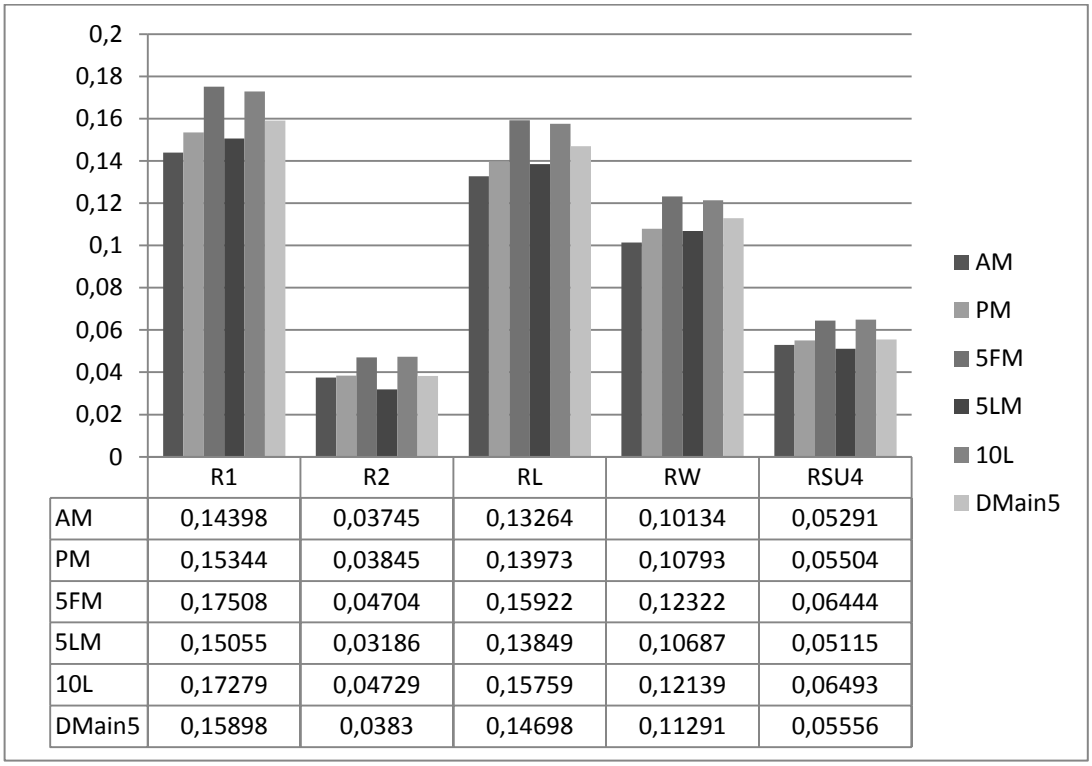
De plus, nous ajoutons à la liste de mots vides, une liste de balises HTML comme <code>, <i>, <b>, etc. En effet, une partie des commentaires extraits sont de type Javadoc. Les commentaires Javadoc peuvent être présentés sous format HTML et les développeurs peuvent personnaliser l'affichage des commentaires grâce à l'utilisation des balises HTML. ROUGE ignore les balises lors de l'évaluation.

4.3.4 Résultats et discussion

4.3.4.1 Résultats

La figure 11 présente l'histogramme ainsi que la table des valeurs des scores ROUGE des différentes heuristiques d'extraction. Les scores présentés sont des F-mesure calculés par ROUGE. Les variantes de ROUGE calculées sont ROUGE-N avec N=1 (R1), ROUGE-N avec N=2 (R2), ROUGE-L (RL), ROUGE-W (RW) et ROUGE-SU4 (RSU4). La distance maximale séparant les deux mots de chaque skip-bigram considéré par ROUGE-SU4 est de 4 mots.

L'observation de la figure 11 montre que les scores ROUGE obtenus par les heuristiques sont très proches. Étant donné que la différence entre les scores n'est pas importante, nous n'avons pas tenu compte des intervalles de confiance calculés par ROUGE durant l'analyse des résultats.



	R1	R2	RL	RW	RSU4
AM	0,14398	0,03745	0,13264	0,10134	0,05291
PM	0,15344	0,03845	0,13973	0,10793	0,05504
5FM	0,17508	0,04704	0,15922	0,12322	0,06444
5LM	0,15055	0,03186	0,13849	0,10687	0,05115
10L	0,17279	0,04729	0,15759	0,12139	0,06493
DMain5	0,15898	0,0383	0,14698	0,11291	0,05556

Figure 11 - Scores ROUGE des heuristiques d'extraction obtenus par extraction des commentaires des méthodes/constructeurs

5FM et 10L sont les meilleures heuristiques. Pour toutes les variantes de ROUGE, ces deux heuristiques obtiennent les scores les plus élevés. DMain5 occupe la troisième position à l'exception de son évaluation avec ROUGE-2 où son score devient légèrement inférieur à celui de PM. PM est classée quatrième meilleure heuristique sauf lors de son évaluation avec ROUGE-2 où elle est considérée la troisième meilleure heuristique. AM et 5LM sont les heuristiques ayant les scores les plus bas.

Par ailleurs, toutes les heuristiques proposées se basent sur l'hypothèse que les tags et leurs contenus ne sont pas pertinents aux résumés. Afin de vérifier cette hypothèse, nous avons généré de nouveau les résumés en appliquant deux nouvelles versions des heuristiques. Une version élimine les tags et conserve leurs contenus dans les résumés générés (version *sans tags*). Une autre version fait l'extraction des commentaires sans

suppression des tags et de leurs contenus (version *brute*). Nous avons évalué ces deux versions avec ROUGE. La figure 12 présente les scores ROUGE des différentes versions des heuristiques.

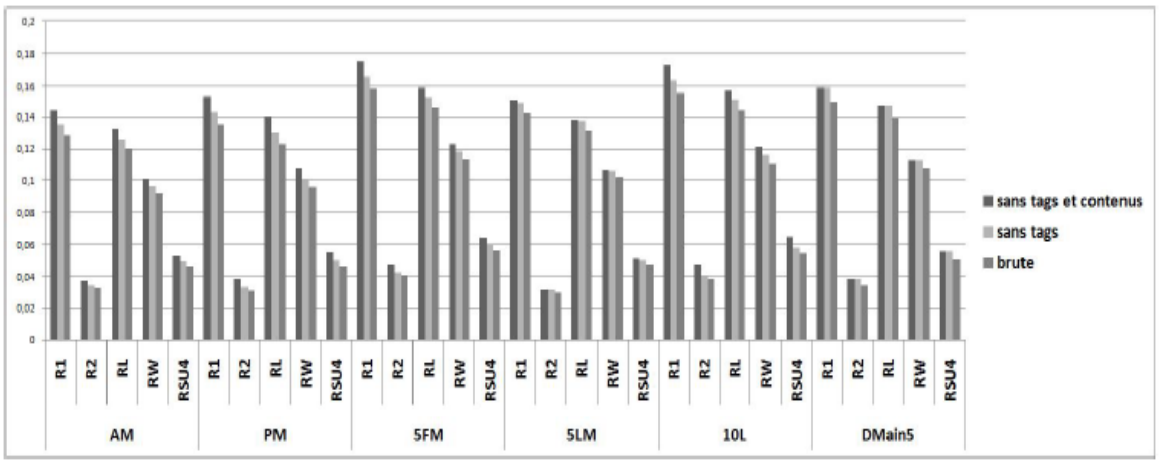


Figure 12 - Scores ROUGE des heuristiques d'extraction (version: sans tags et contenus, sans tags et brute) obtenus par extraction des commentaires des méthodes/constructeurs

L'analyse de la figure 12 montre que, pour les différentes versions des heuristiques, les scores de la version sans tags et contenus sont meilleurs que ceux de la version éliminant les tags (version *sans tags*) qui de même, sont meilleurs de ceux de la version *brute*.

Dans le tableau 8, nous présentons des exemples de résumés générés par les différentes heuristiques et ce, pour la même classe `EventHandlerThread`. Cet exemple appuie le fait que les résumés générés par `5FM` et `10L` sont plus proches du résumé de référence que les autres résumés.

Tableau 8 - Résumé de référence et les résumés candidats de la classe EventHandlerThread générés par application des heuristiques d'extraction

Résumé de référence: A thread that listens and responds to events from JPDA when the debugger has attached to another JVM.
Résumé candidat (PM): **[R1=0,133 - R2=0 - RL=0,133 - RW=0,105 - RSU4=0,031]** Continually consumes events from the VM's event queue until it is disconnected.
Résumé candidat (DMain5): **[R1=0,136 - R2=0,048 - RL=0,136 - RW=0,125 - RSU4=0,156]** Continually consumes events from the VM's event queue until it is disconnected. Responds when a VMDisconnectedException occurs while dealing with another event. We need to flush the event queue, dealing only with exit events (VMDeath, VMDisconnect) so that we terminate correctly. Processes a given event from JPDA. A visitor approach would be much better for this, but Sun's Event class doesn't have an appropriate visit() method. Responds to a breakpoint event.
Résumé candidat (5FM): **[R1=0,196 - R2=0,041 - RL=0,196 - RW=0,162 - RSU4=0,281]** Creates a new EventHandlerThread to listen to events from the given debugger and virtual machine. Calling this Thread's start() method causes it to begin listenting. Logs any unexpected behavior that occurs (but which should not cause DrJava to abort). Logs any unexpected behavior that occurs (but which should not cause DrJava to abort). Continually consumes events from the VM's event queue until it is disconnected. Processes a given event from JPDA. A visitor approach would be much better for this, but Sun's Event class doesn't have an appropriate visit() method.
Résumé candidat (10L): **[R1=0,2 - R2=0 - RL=0,2 - RW=0,153 - RSU4=0,219]** Creates a new EventHandlerThread to listen to events from the given debugger and virtual machine. Calling this Thread's start() method causes it to begin listenting. Logs any unexpected behavior that occurs (but which should not cause DrJava to abort). Logs any unexpected behavior that occurs (but which should not cause DrJava to abort). Continually consumes events from the VM's event queue until it is disconnected.

Résumé candidat (5LM): [R1=0,134 - R2=0 - RL=0,136 - RW=0,112 - RSU4=0,186]
Reponds to a thread death event.
Responds if the virtual machine being debugged dies.
Responds if the virtual machine being debugged disconnects.
Cleans up the state after the virtual machine being debugged dies or disconnects.
Responds when a VMDisconnectedException occurs while dealing with another event.
We need to flush the event queue, dealing only with exit events (VMDeath, VMDisconnect) so that we terminate correctly.

4.3.4.2 Discussion

Dans cette section, nous commencerons par discuter les résultats observés dans la figure 11 et ensuite ceux de la figure 12. La comparaison des heuristiques avec l'heuristique référence AM (baseline) depuis la figure 11 montre que les heuristiques PM, DMain5, 5FM et 10L sont meilleures que AM, alors que 5LM montre des résultats similaires ou inférieurs à AM. Nous discuterons des différentes heuristiques dans la suite.

Comme il est prévu, AM a des scores très bas. AM est une heuristique naïve. Contrairement aux autres heuristiques, elle fait l'extraction des commentaires de tous les méthodes/constructeurs sans exception. Elle a ainsi l'avantage d'avoir le nombre maximum de mots qui peuvent coïncider avec ceux des résumés de référence. Elle obtient un rappel égal à 0,4264. Il est évident qu'il est le plus élevé par rapport aux autres heuristiques. En contre partie, AM perd en précision puisque le nombre de mots formant les résumés candidats d'AM est plus grand que celui des autres heuristiques.

Par ailleurs, la valeur du rappel de AM traduit qu'au plus 42% des unigrammes des résumés de référence apparaissent dans les commentaires des méthodes/constructeurs des classes à résumer. Par conséquent, le rappel des autres heuristiques ne pourra pas dépasser cette valeur ce qui montre la difficulté de la tâche.

PM extrait les commentaires des méthodes/constructeurs publics. Cette heuristique obtient des scores meilleurs que ceux de la baseline AM. Les constructeurs publics permettent l'instanciation de la classe et son utilisation dans les autres classes. Les

méthodes publiques rendent des services aux autres classes du système. À priori, nous pouvons conjecturer que l'attribution de la visibilité publique aux membres (méthodes et constructeurs) d'une classe rend ces derniers plus représentatifs de la classe. Ils reflètent plus son comportement, plus que les membres ayant d'autres visibilités (la visibilité private, la visibilité protected et la visibilité par défaut). Cependant, il est important d'analyser l'impact des constructeurs vs méthodes sur les résumés générés afin de confirmer ou nier cette conjecture. Cette analyse sera présentée plus tard dans cette section.

L'exemple 3 montre un résumé de référence et un résumé candidat généré par application de PM. Le résumé candidat couvre tous les mots du résumé de référence sauf le mot *localized*. Les mots qui coïncident sont soulignés dans le résumé candidat. Aussi, il est équivalent sémantiquement au résumé de référence. Il est à noter cependant que nous n'avons pas traité le problème de la redondance et par conséquent, des phrases redondantes peuvent apparaître dans le résumé candidat. Aussi, il est remarquable que le résumé généré est long par rapport au résumé référence.

Exemple 3 :

Résumé de référence :

Textures default catalog read from localized resources.

Résumé candidat(PM): [**R1=0,1754 - R2=0,0363 - RL=0,1403 - RW=0,1131 - RSU4=0,7**]

Creates a default textures catalog read from resources.
Creates a default textures catalog read from resources and textures plugin folder if texturesPluginFolder isn't null.
Creates a default textures catalog read from resources and textures plugin folder if texturesPluginFolder isn't null.
Creates a default textures catalog read only from resources in the given URLs.
Creates a default textures catalog read only from resources in the given URLs.
Texture image URLs will built from texturesResourcesUrlBase if it isn't null.

DMain5 donne des résultats meilleurs que ceux de la baseline AM. Cette heuristique priorise les méthodes/constructeurs qui sont plus proches du main dans le graphe d'appels statique. Ces méthodes décrivent le comportement de la classe. Cependant, nous croyons que les résultats pourraient être meilleurs si nous utilisons un graphe dynamique au lieu d'un graphe statique. En effet, un graphe statique représente toutes les possibilités d'exécutions d'un programme. Par conséquent certaines méthodes apparaissent dans le graphe alors qu'en réalité, elles ne sont jamais exécutées. Aussi, si une méthode est plus proche du main qu'une autre méthode, cela n'implique pas forcément qu'elle sera exécutée en premier. Contrairement au graphe statique, le graphe dynamique est plus exact et trace les appels de méthodes qui sont réellement exécutées. Il permet de garder traces d'une seule exécution à la fois. Il est possible de réaliser plusieurs exécutions et d'utiliser les différents graphes dynamiques correspondant pour déterminer la liste des méthodes qui sont exécutées en premier.

L'exemple 4 montre un résumé de référence et un résumé candidat généré par DMain5.

Exemple 4 :

Résumé de référence :

A tokenizer that splits a stream into string tokens while
balancing quoting characters.

Résumé candidat(DMain5): **[R1=0,4 - R2=0 - RL=0,32 - RW=0,218 - RSU4=0,2812]**

Returns the type of the current token.
Return the next token, or null if the end of the stream has been reached.
Specify a new keyword.
Specify a pair of quotes.
Create a new balancing stream tokenizer.

5FM est la meilleure heuristique. 5FM extrait les commentaires des cinq premières méthodes écrites dans la classe. Les résultats obtenus consolident l'hypothèse que les méthodes/constructeurs les plus pertinents se positionnent au début des classes. Nous conjecturons que les développeurs commencent par écrire les méthodes/constructeurs les plus importants. Ensuite, ils raffinent l'implémentation de plus en plus.

Par ailleurs, en inspectant le code manuellement, nous avons remarqué que les constructeurs sont la plupart du temps positionnés en haut des classes. L'heuristique 5FM par exemple fait l'extraction de 496 commentaires de constructeurs ce qui montre qu'en moyenne chaque classe possède un constructeur positionné parmi les 5 premières méthodes/constructeurs (496/427=1,16). L'extraction des commentaires des constructeurs pourrait expliquer les résultats avantageux de 5FM. Nous rappelons que nous présenterons plus tard dans cette section une analyse de l'impact des constructeurs vs méthodes sur les résumés générés.

L'exemple 5 montre un résumé de référence et un résumé candidat généré par 5FM.

Exemple 5 :

Résumé de référence :

A check box that accepts null values. Thus this check box is
able to display 3 states : null, false and true.

Résumé candidat(5FM): [**R1=0,4444 - R2=0,1176 - RL=0,3333 - RW=0,2291 - RSU4=0,2741**]

Creates a nullable check box.
Returns null, Boolean.TRUE or Boolean.FALSE.
Sets displayed value in check box.
Returns true if this check box is nullable.
Sets whether this check box is nullable.

10L obtient des résultats similaires à ceux de 5FM. En effet, 10L extrait les 10 premières lignes. Or, la taille des commentaires des méthodes/constructeurs écrits par les développeurs n'est pas grande. La moyenne du nombre de lignes des commentaires par méthodes/constructeurs est égale à une ligne (nombre de lignes des commentaires / nombre de méthodes et constructeurs commentés = 7222 / 5178 = 1,32). Par conséquent, les résumés générés par 10L possèdent du contenu en commun avec 5FM ainsi que du contenu supplémentaire qui provient, en moyenne, des méthodes/constructeurs dont l'ordre d'écriture dans la classe est supérieur à 5. Le fait que les résultats de 10L soient parfois inférieurs à 5FM montre que plus les méthodes/constructeurs qui sont écrits plus bas dans la classe sont considérés par l'heuristique, moins les scores sont élevés. L'exemple 6 montre un résumé de référence, un résumé candidat généré par 10L et un résumé candidat généré par 5FM. Il est à noter qu'un résumé généré par 10L pourrait avoir moins de 10 lignes après suppression des tags et de leurs contenus.

Exemple 6 :

Résumé de référence:

Class for keeping track of watched fields and variables.

Résumé candidat (10L): [R1=0,2051 - R2=0,1621 - RL=0,2051 - RW=0,1923 - RSU4=0,45]

L1: Object to keep track of a watched field or variable.
l2: Returns the name of this field or variable.
L3: Returns the most recently determined value for this field or variable.
L4: Returns the type of this field or variable in the current context.
L5: Sets a new name for this field or variable.
L6: Sets the most recently determined value for this field or variable.
L7: Hides the value for this watch (when no thread is suspended).

Résumé candidat (5FM): **[R1=0,2666 - R2=0,2142 - RL=0,2666 - RW=0,2455 - RSU4=0,45]**

Object to keep track of a watched field or variable.

Returns the name of this field or variable.
Returns the most recently determined value for this field or variable.
Returns the type of this field or variable in the current context.
Sets a new name for this field or variable.

5LM extrait les commentaires des 5 dernières méthodes d'une classe. Cette heuristique montre des performances similaires à la baseline AM. Elle a même obtenu des scores inférieurs à ceux d'AM. L'écart significatif entre les résultats obtenus par 5FM et les résultats obtenus par 5LM affaiblit l'hypothèse que les méthodes les plus importantes se trouvent à la fin des classes. Par contre, ils consolident l'hypothèse que les méthodes les plus pertinentes au résumé se positionnent au début des classes. L'exemple 7 montre un résumé de référence, un résumé candidat généré par 5LM et un résumé candidat généré par 5FM.

Exemple 7 :

Résumé de référence:

A Swing specific model for the DrJava InteractionsPane.
It glues together an InteractionsDocument, an InteractionsPane and a JavaInterpreter.
This abstract class provides common functionality for all such models.
The methods in this class generally can be executed only in the event thread once the model has been constructed.

Résumé candidat (5LM): **[R1=0,0714 - R2=0 - RL=0,0714 - RW=0,0546 - RSU4=0,0172]**

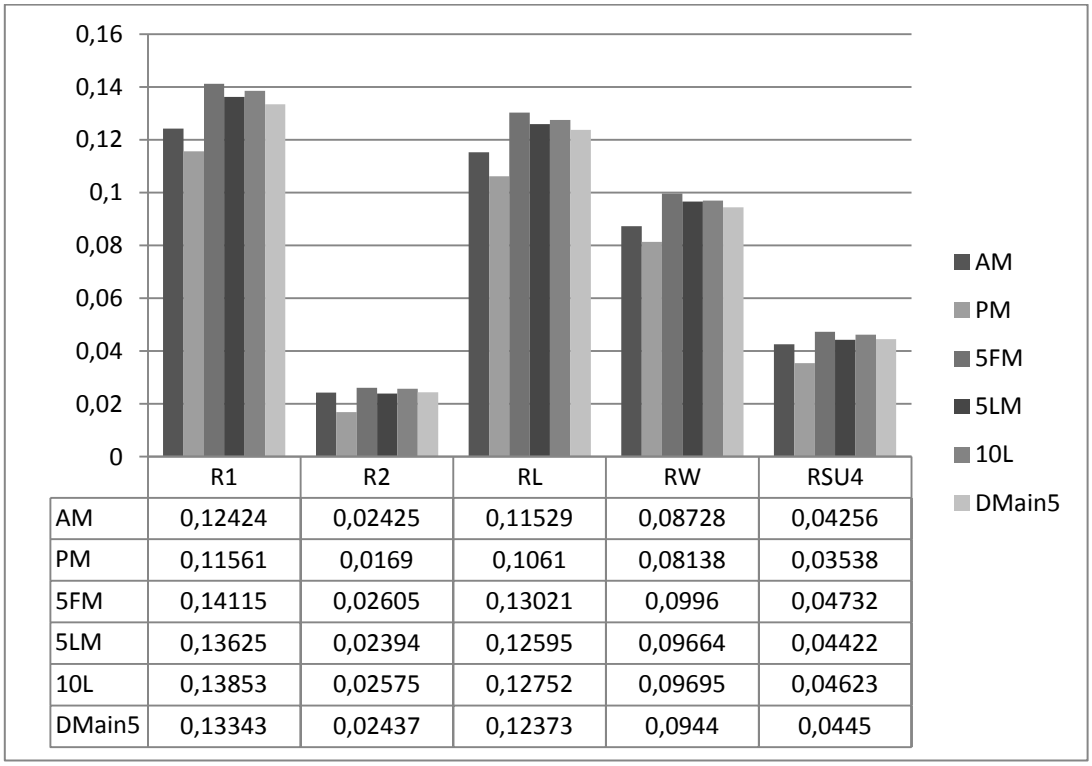
Perform the default imports of the classes and packages
listed in the INTERACTIONS_AUTO_IMPORT_CLASSES.
Return the last error, or null if successful.
Return the second to last error, or null if successful.
Reset the information about the last and second to last error.
Returns the last history item and then removes it, or returns null if the history is empty.

Résumé candidat (5FM): [R1=0,2413 - R2=0,0357 - RL=0,2413 - RW=0,1379 - RSU4=0,0603]
Constructs an InteractionsModel.
The InteractionsPane is created later by the InteractionsController.
As a result, the posting of a banner at the top of InteractionsDocument must be deferred until after the InteracationsPane has been set up.
Sets the _pane field and initializes the caret position in the pane.
Called in the InteractionsController.
Adds an InteractionsListener to the model.
Removea an InteractionsListener from the model.
If the listener is not currently listening to this model, this method has no effect.
Removes all InteractionsListeners from this model.

Pour vérifier l'importance des commentaires des constructeurs et vérifier l'impact des méthodes positionnées en haut des classes sur les résumés des classes, nous avons appliqué de nouveau les heuristiques d'extraction en ignorant les commentaires des constructeurs et en extrayant les commentaires des méthodes uniquement. Les résultats sont présentés dans la figure 13.

En comparant les scores de la figure 13 à ceux de la figure 11, nous constatons que les scores des heuristiques de la première sont inférieurs à ceux de la deuxième figure et ce, pour toutes les variantes de ROUGE. Ceci montre que toutes les heuristiques sont affectées par l'absence des commentaires des constructeurs. Nous pouvons confirmer ainsi que les commentaires des constructeurs rapprochent les résumés générés aux résumés de référence et qu'ils sont par conséquent importants aux résumés des classes.

D'un autre côté, nous remarquons que même en absence des commentaires des constructeurs, 5FM obtient toujours les meilleurs scores. Nous pouvons confirmer ainsi que les méthodes positionnées au début des classes sont importantes et que leurs commentaires influencent les résumés générés. Cette conjecture est conforme avec les travaux en résumé de texte qui ont montré que les mots des titres et les phrases qui se positionnent au début des textes sont pertinents (voir chapitre *État de l'art*).



	R1	R2	RL	RW	RSU4
AM	0,12424	0,02425	0,11529	0,08728	0,04256
PM	0,11561	0,0169	0,1061	0,08138	0,03538
5FM	0,14115	0,02605	0,13021	0,0996	0,04732
5LM	0,13625	0,02394	0,12595	0,09664	0,04422
10L	0,13853	0,02575	0,12752	0,09695	0,04623
DMain5	0,13343	0,02437	0,12373	0,0944	0,0445

Figure 13 - Scores ROUGE des heuristiques d'extraction obtenus par extraction des commentaires des méthodes

À notre surprise, nous remarquons que le classement de 5LM par rapport aux autres heuristiques s'est nettement amélioré. Les scores de 5LM se sont rapprochés de ceux de 5FM. Une explication possible est que la possibilité d'extraction des commentaires des mêmes méthodes par 5FM et 5LM devient plus probable dès l'ignorance des constructeurs. Nous avons calculé le nombre des classes possédant chacune moins de 10 méthodes. Nous avons trouvé que 211 classes possèdent moins de 5 méthodes et que 84 classes possèdent entre 6 et 9 méthodes. Pour presque la moitié des classes (211 classes parmi 427 classes), les résumés générés par 5FM et 5LM sont totalement similaires. Pour 84 classes, les résumés générés possèdent des commentaires en commun.

D'autre part, nous observons que les résultats obtenus par 10L sont homogènes avec ceux qu'elle a obtenus précédemment (avec méthodes/constructeurs). Elle donne des

résultats similaires à ceux de 5FM. Comme nous l'avons déjà expliqué, les résumés générés par 10L ressemblent à ceux de 5FM.

L'heuristique DMain5 précède toujours la baseline AM. Ceci confirme l'hypothèse que les méthodes les plus proches du main sont importantes pour la classe, même en absence des constructeurs.

Toutefois, nous remarquons que les scores de PM deviennent plus faibles que ceux de l'heuristique naïve AM. Les résumés générés par l'heuristique PM sont donc fortement biaisés par la présence des commentaires des constructeurs à visibilité publique. Par exemple, le résumé candidat de l'exemple 3 donné précédemment est formé entièrement par des commentaires de constructeurs. Par ailleurs, nous en déduisons que les méthodes publiques ne sont pas forcément plus importantes que le reste des méthodes.
Il serait cependant important de faire d'autres expériences pour vérifier cette conjecture, comme la comparaison séparée entre les résumés générés par extraction des méthodes à différentes visibilités.

L'analyse de la figure 12 montre que la suppression des tags et de leurs contenus est avantageuse et ce, pour toutes les heuristiques. Ceci prouve que les tags et les contenus des méthodes sont plutôt spécifiques aux méthodes mêmes. L'information incluse n'est pas importante pour la classe à résumer. L'exemple 8 montre un résumé de référence et les résumés candidats générés par les trois versions de 5FM. D'après l'exemple, il est clair qu'il est inutile d'inclure les tags dans le résumé. Le contenu des tags est aussi non nécessaire puisqu'il est rare de trouver des mots qui coïncident avec le résumé de référence (par exemple, @param word the new word to seek n'a aucun mot en commun avec le résumé de référence et en plus, cette ligne est différente sémantiquement de la référence).

Exemple 8 :

Résumé de référence:
Implementation of logic of find/replace over a document.

Résumé candidat (5FM: sans tags et contenus): [**R1=0,162 - R2=0,057 - RL=0,162 - RW=0,154 - RSU4=0,357**]
Standard Constructor.
Creates new machine to perform find/replace operations on a particular document starting from a given position.
Called when the current position is updated in the document implying _skipText should not be set if the user toggles _searchBackwards.
Gets the character offset to which this machine is currently pointing.
Change the word being sought.
Change the replacing word.

Résumé candidat (5FM: sans tags): **[R1=0,12 - R2=0,042 - RL=0,12 - RW=0,116 - RSU4=0,357]**
Standard Constructor.
Creates new machine to perform find/replace operations on a particular document starting from a given position.
docIterator an object that allows navigation through open Swing documents (it is DefaultGlobalModel).
Called when the current position is updated in the document implying _skipText should not be set if the user toggles _searchBackwards.
Gets the character offset to which this machine is currently pointing.
Change the word being sought.
word the new word to seek.
Change the replacing word.
word the new replacing word.

Résumé candidat (5FM: brute): **[R1=0,113 - R2=0,039 - RL=0,113 - RW=0,109 - RSU4=0,357]**
Standard Constructor.
Creates new machine to perform find/replace operations on a particular document starting from a given position.
@param docIterator an object that allows navigation through open Swing documents (it is DefaultGlobalModel).
Called when the current position is updated in the document implying _skipText should not be set if the user toggles _searchBackwards.

Gets the character offset to which this machine is currently pointing.
Change the word being sought.
@param word the new word to seek.
Change the replacing word.
@param word the new replacing word.

4.3.4 Limites de l'évaluation automatique

Il est vrai que l'évaluation automatique est avantageuse puisqu'elle est moins coûteuse qu'une évaluation manuelle. Mais malheureusement, l'évaluation automatique ne tient pas compte de l'aspect sémantique. Dans cette section, nous présentons des paires de (résumé de référence/résumé candidat) que nous avons étudiés manuellement et nous montrons des cas où la sémantique des commentaires n'a pas été considérée lors de l'évaluation.

ROUGE ne trouve de correspondance au mot 0 présent dans la séquence the following: ';', '{', '}', bien qu'il correspond en réalité au mot nothing dans la séquence the following: ';', '{', '}' or nothing. La détection de ce type de correspondance n'est pas possible d'une manière automatique. Seul l'humain en est capable. Aussi, les mots checks et disregarded ne sont pas présents dans les résumés candidats mais leurs synonymes le sont (determine et ignore). Lors de l'évaluation, les synonymes ne sont pas pris en considération.

Exemple 9 :

Résumé de référence:
Determines if the current line is starting a new statement by searching backwards to see if the previous line was the end of a statement.
Specifically, checks if the previous non-whitespace character not on this line is one of the following: ';', '{', '}', or 0.
Note that characters in comments and quotes are disregarded.

Résumé candidat(NFM): [**R1=0,6153 - R2=0,3783 - RL=0,5641 - RW=0,2864 - RSU4=0,336**]
Constructs a new rule to determine if the current line is the start of a new statement.
Determines if the previous non-whitespace character not on this line was one of the ollowing: ';', '{', '}' or nothing.
Ignores characters in quotes and comments.

L'exemple 10 montre le cas où ROUGE ne trouve pas les mots correspondants aux noms des identifiants. ROUGE ne trouve pas dans le résumé candidat les mots correspondant aux mots tag, editor et dialog du résumé de référence, malgré que le mot TagEditorDialog qui correspond au nom de la classe à résumer est utilisé dans le résumé candidat. Les développeurs utilisent les noms de classes tel quel au lieu de les expliciter parce que les noms sont déjà significatifs par eux même. Nous suggérons de diviser les noms des identifiants en leurs sous noms comme travail futur.

Exemple 10 :

Résumé de référence:
Dialog to provide a simple tag editor for song files.
Unlike the other dialog boxes in this package, this one is self-contained in that if the OK button is clicked it updates the configured song file directly and no other action is needed by the client code.

Résumé candidat(NFM): [**R1=0,2127 - R2=0 - RL=0,1702 - RW=0,1083 - RSU4=0,0655**]
Create a new TagEditorDialog associated with the supplied MainWindow and SongInfo object.
Handle the user clicking the Cancel button.
Handle the user clicking the OK button – verify the values the user has input and if they are valid then update the Song object and file.

Les résumés générés peuvent être très brefs. L'exemple 11 présente un résumé candidat qui est en partie sémantiquement équivalent au résumé de référence et qui résume en deux mots la fonctionnalité principale de la classe. ROUGE donne automatiquement une valeur basse à ce résumé. Cependant, les mots utilisés sont très significatifs. Il serait intéressant, comme travail futur, de déterminer les mots les plus importants dans le résumé et de donner par la suite un poids élevé à ces mots.

Exemple 11 :

Résumé de référence:

An exception thrown by DefinitionsDocument.getPackageName()
when the document contains an invalid package statement.
This can happen if there is nothing between "package" and ";", or if there is no terminating semicolon ever.

Résumé candidat(DMain5): [**R1=0,1428 - R2=0 - RL=0,1428 - RW=0,103 - RSU4=0**]

Constructs a exception.

Nous avons constaté que les résumés de référence expliquent parfois les fonctionnalités des classes d'une manière générale alors que les résumés candidats en donnent plus de détails. Par exemple, la phrase Configuration class that accesses to Mac OS X specifics du résumé de référence de l'exemple 12 explique qu'il s'agit d'une classe de configuration qui a accès aux propriétés de MAC OS. Par contre, les deux phrases du résumé candidat expliquent en termes d'actions (Binds homeApplication to menu et adds window to frame) en quoi consiste la configuration. Il serait intéressant d'exploiter, dans l'avenir, les relations rhétoriques pour détecter les relations qui peuvent exister entre les phrases références et les phrases candidates (relation d'explication dans notre cas).

Exemple 12 :

Résumé de référence:

Configuration class that accesses to Mac OS X specifics.
Do not invoke methods of this class without checking first if os.name System property is Mac OS X.
This class requires some classes of com.apple.eawt package to compile.

Résumé candidat(NFM): [**R1=0,2162 - R2=0,1142 - RL=0,2162 - RW=0,1434 -RSU4=0,0625**]

Binds homeApplication to Mac OS X application menu.
Adds Mac OS X standard Window menu to frame.

4.4 Conclusion

Dans ce travail, nous avons fait une première exploration de l'application des techniques de résumé par extraction pour la génération de la documentation de classes. Nous avons défini des heuristiques d'extraction et nous avons procédé à l'évaluation automatique des résumés candidats en utilisant la métrique ROUGE sur un corpus de paires de (résumé de référence/résumé candidat). Nous avons trouvé que la différence entre les scores obtenus par les différentes heuristiques n'est pas importante. En analysant de près les scores, nous avons trouvé que la majorité des heuristiques sont meilleures que la baseline AM. En particulier 5FM produit les meilleurs résumés en la comparant aux autres heuristiques.

Par ailleurs, il est important de noter que les commentaires générés automatiquement ne pourront pas remplacer ceux écrits manuellement. Ils permettent plutôt d'aider le développeur à comprendre les fonctionnalités d'une classe au cas où son commentaire original est absent.

De plus, nous mentionnons que dans ce travail, nous avons considéré le problème de documentation de classes comme analogique à celui de résumés de textes naturels. Mais, nous avons trouvé des spécificités liées à la tâche effectuée. En effet, résumer un texte

naturel consiste à générer une description plus brève que le texte à résumer en relevant les points importants du texte. Dans notre cas, les commentaires des classes des projets que nous avons étudiés sont souvent courts et les commentaires générés sont généralement longs. Toutefois, nous pensons que les commentaires générés aideront à un certain degré les développeurs à connaître les fonctionnalités d'une classe en cas d'absence du commentaire original. Il serait cependant important de traiter le problème de la redondance des phrases au niveau des commentaires générés. Ceci permettra d'éliminer l'information redondante et de rendre les commentaires générés plus lisibles et plus compréhensibles.

Conclusion

5.1 Rétrospective

Dans ce mémoire, nous proposons d'explorer des techniques de résumé par extraction pour la redocumentation de programmes. En particulier, nous proposons de produire des commentaires pour les classes Java en faisant l'extraction des commentaires des méthodes selon des heuristiques d'extraction. Les commentaires produits sont appelés des résumés de classes. Pour arriver à notre but, nous avons divisé ce travail en deux grandes parties.

Dans la première partie, nous avons fait une étude empirique des commentaires dans trois projets Java. Cette étude a pour but de déterminer la répartition et la pertinence des commentaires dans la perspective de les utiliser dans la génération d'autres commentaires pour le résumé. Elle traite ainsi de l'aspect quantitatif et qualitatif des commentaires.

En termes de quantité, nous nous sommes intéressés à l'étude de la distribution des commentaires par rapport aux différents types d'instructions et de la fréquence de documentation de chaque type. Nous avons trouvé, entre autres, que les classes sont souvent commentées.

En termes de qualité, nous nous sommes intéressés à l'étude du contenu et de la qualité des commentaires. Nous avons fait une expérience où des sujets classifient les commentaires selon une taxonomie proposée. Nous avons trouvé que la majorité des commentaires des méthodes/constructeurs sont pertinents puisqu'ils sont la plupart du temps explicatifs et de bonne qualité.

L'intérêt porté par les programmeurs à commenter les classes montre que les classes et leur documentation leur sont nécessaires. Nous avons proposé ainsi de documenter les classes pour combler au manque de la documentation au cas où elle n'est pas disponible.

Aussi, comme la majorité des commentaires des méthodes/constructeurs sont pertinents et sachant que les fonctionnalités des classes sont implémentées par les méthodes, nous

avons décidé d'extraire les commentaires des méthodes/constructeurs pour résumer les classes.

Ainsi, dans la deuxième partie de notre projet, nous avons défini des heuristiques pour déterminer quels sont les commentaires à considérer, parmi l'ensemble des commentaires des méthodes/constructeurs possibles pour la génération de résumés de classe.

Nous avons appliqué nos heuristiques sur 427 classes de trois projets. Ensuite, nous avons évalué d'une manière automatique les résumés générés en comparant les commentaires obtenus avec les commentaires originaux par le biais de la métrique ROUGE. Les scores obtenus par l'heuristique qui considère les commentaires des cinq premiers méthodes/constructeurs dans l'ordre d'écriture du programme sont les meilleurs. Nous avons trouvé que les méthodes/constructeurs écrits en premier sont plus pertinents que les autres. Nous avons trouvé que les commentaires des méthodes/constructeurs qui sont les plus proches de la méthode principale `main` sont également pertinents. Par contre, nous avons trouvé que les commentaires des méthodes/constructeurs écrits à la fin de la classe ne sont pas de bons candidats à l'extraction.

D'un autre côté, nous avons fait un premier pas dans le résumé par abstraction en supprimant les tags et leurs contenus des résumés générés. Nous avons trouvé que les résumés ainsi générés sont meilleurs que les résumés bruts et les résumés dont les tags sont supprimés.

5.2 Perspectives futures

De nombreuses pistes d'amélioration sont possibles pour ce travail.

Nous avons généré des résumés par extraction des commentaires des méthodes. Nous avons remarqué que du contenu redondant apparaît dans le résumé généré. Ce contenu diminue la qualité du résumé et réduit sa lisibilité. Nous pourrons éviter la redondance en calculant la similarité entre chaque commentaire candidat à l'extraction et les commentaires déjà extraits. Si le commentaire candidat obtient un score de similarité élevé alors il peut être ignoré.

Par ailleurs, nous avons défini un paramètre N pour les heuristiques. Ce paramètre permet de spécifier le nombre de méthodes/constructeurs ou le nombre de lignes à considérer pour l'extraction. Lors de l'évaluation, nous avons fixé la valeur de N. Il serait important cependant de faire d'autres évaluations en faisant varier N et de voir l'impact de cette variation sur les résumés.

Il serait aussi intéressant de voir s'il est possible de définir une fonction qui calcule la valeur adéquate de N pour chaque classe à résumer, et ce, en tenant compte de la spécificité de cette dernière. Par exemple, nous pourrons regarder si ce paramètre serait une fonction du nombre des méthodes et des constructeurs ou bien s'il serait une fonction du nombre des lignes des commentaires de la classe à résumer.

De plus, il serait intéressant d'essayer de nouvelles heuristiques d'extraction. Par exemple, nous pourrons définir une heuristique qui extrait les commentaires des méthodes ayant un nombre important de lignes de code ou qui sont les plus appelées et ce, à partir d'un graphe d'appel. Nous pourrons utiliser les graphes d'appels dynamiques au lieu des graphes statiques. En effet, les graphes dynamiques apportent un peu plus de précision en considérant les exécutions réellement passées.

Finalement, nous envisageons l'application de l'approche proposée à d'autres niveaux de granularité que celui de la classe. En effet, le principe de l'approche est d'extraire les commentaires des concepts de bas niveau pour documenter les concepts de plus haut niveau. En particulier, dans ce travail, les commentaires des méthodes ont été extraits pour documenter les classes. Il serait intéressant d'appliquer le même principe à d'autres éléments de granularité variée. Par exemple, nous pourrons résumer les méthodes à partir des instructions élémentaires qui se trouvent dans leurs corps. Pour ce faire, nous pourrons définir une heuristique qui extrait les commentaires des instructions écrites en premier, ou bien, nous pourrons extraire les commentaires des blocs d'instructions qui sont toujours exécutés et ce, en utilisant un graphe de flux de contrôle.

Bibliographie

[1] Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig, "Software complexity and maintenance costs," *Communications of the ACM*, pp. 81-94, 1993.

[2] Andrew Brian Begel, "Spoken language support for software development," University of California, Berkley, mémoire de maîtrise 2005.

[3] Sergey Brin and Lawrence Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks and ISDN Systems*, vol. 30, pp. 107-117, April 1998.

[4] H. P. Edmundson, "New Methods in Automatic Extracting," *Journal of the ACM*, vol. 16, pp. 264-285, April 1969.

[5] Günes Erkan and Dragomir R. Radev, "LexRank: graph-based lexical centrality as salience in text summarization," *Journal of Artificial Intelligence Research*, vol. 22, pp. 457-479, December 2004.

[6] Len Erlikh, "Leveraging legacy system dollars for e-business," *IT Professional*, vol. 2, pp. 17-23, May 2000.

[7] Udo Hahn and Inderjeet Mani, "The Challenges of Automatic Summarization," *Computer*, vol. 33, pp. 29-36, November 2000.

[8] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus, "On the Use of Automated Text Summarization Techniques for Summarizing Source Code," *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, pp. 35-44, 2010.

[9] Dorsaf Haouari, Sahraoui Houari, and Philippe Langlais, "How good is your comment? a study of comments in Java programs," *Empirical Software Engineering and Measurement*, Septembre 2011.

[10] Karen Sparck Jones, "Automatic Summarising: Factors and Directions," *Advances in automatic text summarization*, pp. 1-33, 1999.

[11] Ninus Khamis, Juergen Rilling, and René Witte, "Generating an NLP Corpus from

Java Source Code: The SSL Javadoc Doclet," *Proceedings of the Natural language processing and information systems, and 15th international conference on Applications of natural language to information systems, NLDB'10*, pp. 68-79, 2010.

[12] Ninus Khamis, René Witte, and Juergen Rilling, "Automatic quality assessment of source code comments: the JavadocMiner," *Proceedings of the Natural language processing and information systems, and 15th international conference on Applications of natural language to information systems*, pp. 68-79, 2010.

[13] J. Peter Kincaid, Robert P. Fishburne, Richard L. Rogers, and Brad S. Chissom, "Derivation of New Readability Formulas (Automated Readability Index, Fog Count and Flesch Reading Ease Formula) for Navy Enlisted Personnel," rapport technique 1975.

[14] Julian Kupiec, Jan Pedersen, and Francine Chen, "A trainable document summarizer," *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 68-73, juillet 1995.

[15] Chin-Yew Lin, "Looking for a Few Good Metrics: Automatic Summarization Evaluation - How Many Samples are Enough?," *Proceedings of the NTCIR Workshop 4*, 2004.

[16] Chin-Yew Lin, "ROUGE: A Package for Automatic Evaluation of Summaries," *In Proceedings of the Workshop on Text Summarization Branches Out, post-conference workshop of ACL*, pp. 74-81, 2004.

[17] H. P. Luhn, "The automatic creation of literature abstracts," *IBM Journal of Research and Development*, vol. 2, pp. 159-165, April 1958.

[18] H. Malik, I. Chowdhury, Hsiao-Ming Tsou, Zhen Ming Jiang, and A.E. Hassan, "Understanding the rationale for updating a function's comment," *24th IEEE International Conference on Software Maintenance*, pp. 167-176, 2008.

[19] William C. Mann and Sandra A. Thompson, "Rhetorical Structure Theory: Toward a functional theory of text organisation," *Text*, vol. 3, pp. 234-281, 1988.

[20] Daniel Marcu, "From discourse structures to text summaries," *Proceedings of the*

ACL/EACL '97 Workshop on Intelligent Scalable Text Summarization, pp. 82-88, 1997.

[21] Kurt Nørmark, "Requirements for an Elucidative Programming Environment," *Proceedings of the 8th International Workshop on Program Comprehension, IWPC '00*, pp. 119-128, 2000.

[22] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou, "Listening to programmers Taxonomies and characteristics of comments in operating system code," *Proceedings of the 31st International Conference on Software Engineering, ICSE '09*, pp. 331-341, 2009.

[23] J. Pollock and Antonio Zamora, "Automatic Abstracting Research at Chemical Abstracts Service," *Journal of Chemical Information and Computer Sciences*, vol. 15, pp. 226-232, 1975.

[24] M.H. Rispal, *La méthode des cas: Application à la recherche en gestion*, De Boeck, Ed. Bruxelles, 2002.

[25] Gerard Salton, Amit Singhal, Mandar Mitra, and Chris Buckley, "Automatic text structuring and summarization," *Information Processing and Management: an International Journal - Special issue: methods and tools for the automatic construction of hypertext*, vol. 33, pp. 193-207, March 1997.

[26] Daniel Schreck, Valentin Dallmeier, and Thomas Zimmermann, "How documentation evolves over time," *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting, IWPSE '07*, pp. 4-10, 2007.

[27] Maria Joao Sousa, "A Survey on the Software Maintenance Process," *Proceedings of the International Conference on Software Maintenance, ICSM '98*, pp. 265-274, 1998.

[28] Sergio Cozzetti B. Souza, Nicolas Anquetil, and Káthia Marçal Oliveira, "Which documentation for software maintenance?," *Journal of the Brazilian Computer Society*, vol. 12, no. 3, pp. 31-44, 2006.

[29] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods,"

Proceedings of the IEEE/ACM international conference on Automated software engineering, ASE '10, pp. 43-52, 2010.

[30] Thomas A. Standish, "An Essay on Software Reuse," *IEEE Transactions on Software Engineering*, vol. 10, pp. 494-497, Septembre 1984.

[31] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou, "/*iComment: Bugs or Bad Comments?*/," *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pp. 145-158, 2007.

[32] Lin Tan, Ding Yuan, and Yuanyuan Zhou, "Hotcomments: how to make program comments more useful?," *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pp. 19:1-19:6, 2007.

[33] Sima Soudian Tehrani, "Verbal source code descriptor," The University of British Columbia, Mémoire de maîtrise 2003.

[34] Simone Teufel and Marc Moens, "Summarizing scientific articles: experiments with relevance and rhetorical status," *Computational Linguistics - Summarization*, vol. 28, pp. 409-445, december 2002.

[35] Eirik Tryggeseth, "Report from an Experiment: Impact of Documentation on Maintenance," *Journal of Empirical Software Engineering*, vol. 2, pp. 201-207, 1997.

[36] Annie T. T. Ying, James L. Wright, and Steven Abrams, "Source code that talks: an exploration of Eclipse task comments and their implication to repository mining," *Proceedings of the 2005 international workshop on Mining software repositories, MSR '05*, pp. 1-5, 2005.