Gestion des bibliothèques tierces dans un contexte de maintenance logicielle

Cédric Teyton

▶ **To cite this version:**

Cédric Teyton. Gestion des bibliothèques tierces dans un contexte de maintenance logicielle. Génie logiciel [cs.SE]. Université de Bordeaux, 2014. Français. NNT : 2014BORD0123 . tel-01152561

HAL Id: tel-01152561
https://theses.hal.science/tel-01152561

Submitted on 18 May 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ACADÉMIE DE BORDEAUX
UNIVERSITÉ DE **B**ORDEAUX
Sciences et **T**echnologies

THÈSE

présentée au Laboratoire Bordelais de Recherche en Informatique pour obtenir le grade de Docteur de l'Université de Bordeaux

Spécialité	:	**Informatique**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Mathématiques et Informatique**

Gestion des bibliothèques tierces dans un contexte de maintenance logicielle

par

Cédric TEYTON

Soutenue le 26 Septembre 2014, devant le jury composé de :

Président du jury
François PELLEGRINI, professeur ..... Université de Bordeaux, France

Directeur de thèse
Xavier BLANC, professeur ..... Université de Bordeaux, France

Co-Directeur de thèse
Jean-Rémy FALLERI, maître de conférences ..... Institut Polytechnique de Bordeaux, France

Rapporteurs
Laurence DUCHIEN, professeur ..... Université Lille 1, France
Roberto DI COSMO, professeur ..... Université de Paris Diderot VII, France

Examinateurs
Tom MENS, professeur ..... Université de Mons, Belgique

Remerciements

Cette partie demeure certainement la moins scientifique du manuscrit. Néanmoins, si j'ai bien effectué mon travail de rédaction, le contenu des chapitres suivants devrait être tout aussi accessible (il faudra s'y pencher pour s'en convaincre !). Je vais donc en profiter pour remercier toutes les personnes qui m'ont permis d'arriver à bout de cette thèse, chacune à leur manière.

Tout d'abord, j'adresse mes plus profonds remerciements à Xavier Blanc et Jean-Rémy Falleri pour toute la confiance et la liberté qu'ils m'ont accordées durant ces 3 années. J'ai pris beaucoup de plaisir à travailler dans de telles conditions, et je n'oublierai pas toute la dévotion et l'écoute dont ils ont fait preuve à mon égard. J'espère désormais avoir l'occasion de travailler et d'échanger avec eux le plus longtemps possible.

Je remercie Laurence Duchien, Roberto Di Cosmo, Tom Mens et François Pellegrini de s'être intéressés à mon manuscrit et d'avoir assisté à ma soutenance de thèse. C'est probablement le dernier souvenir que je garderai de toute cette expérience, et je suis très honoré de vous avoir eu parmi mon jury.

J'adresse également une infinie gratitude à mes parents sans qui je n'aurais certainement pas pu accomplir ce long parcours d'études. Je vous suis entièrement reconnaissant pour tout ce que vous avez fait pour moi et je ne vous remercierai jamais assez pour tout ce que vous continuez de m'apporter. J'embrasse également le reste de ma famille pour leur présence et pour toute l'affection qu'ils me portent.

Il me faut évidemment remercier la personne formidable qui m'apporte tant de bonheur et d'amour au quotidien, et qui a su accepter et tolérer les aléas de cette expérience. C'est aussi grâce à toi que j'ai trouvé la force et la joie nécessaires à accomplir ce travail. Je salue également ta petite famille, avec qui je continue de passer de très agréables moments. Cela m'a fait plaisir qu'ils viennent assister à ma soutenance.

Je souhaiterais aussi remercier mes collègues doctorants Matthieu et Alan à qui je sou-

haite une bonne continuation dans leurs thèses respectives. Je remercie également Floréal pour m'avoir expliqué comment codent les grands hommes sur cette planète, ainsi que Laurent et David pour leurs conseils tout au long de ma thèse. Je salue aussi Marc, Xavier, Amine, Jannik, Eloi et David (*2) avec qui c'était un plaisir de travailler et d'échanger. Je remercie aussi les personnes du LaBRI qui m'ont aidé dans mes différentes démarches. Je salue enfin toutes les personnes qui m'ont tenu compagnie et avec qui j'ai pris et prends toujours plaisir à discuter lorsque je les croise.

Je remercie également Julien avec qui nous avons partagé toutes ces années d'étude, je garde un bon souvenir de ta compagnie et merci d'avoir partagé autant de pauses café avec moi. Je suis certain que tes qualités te permettront de faire de belles choses dans le monde universitaire. Sache néanmoins que je ne te rendrai visite que si ton futur laboratoire possède une table de ping-pong.

Bien entendu, j'ai des remerciements tout particuliers à adresser à toutes mes amies et tous mes amis qui ont été présents autour de moi pendant ces 3 années (et même avant), et avec qui j'ai partagé de si bons moments. Je sais que j'ai parfois des difficultés à expliquer ce que je fais dans mon travail, mais sachez que votre soutien et votre intérêt m'ont été énormément précieux. J'ai de la chance d'être entouré par des personnes comme vous, et je ne vous remercierai jamais assez de toute l'affection que vous me portez. J'espère pouvoir être un aussi bon ami que vous l'êtes pour moi. Une pensée particulière à Romain et William pour ces chouettes moments de vie en colocation, ainsi qu'à Sébastien (qui, selon la légende, est la personne qui m'a convaincue de me lancer dans une thèse... :)).

Mon doctorat m'a permis de m'initier aux joies de l'enseignement et du partage de connaissance. Je salue donc Eric Grivel et toutes les personnes du département Télécommunications de l'ENSEIRB-MATMECA pour m'avoir accueilli au sein de l'équipe pédagogique et pour ces moments très agréables en leur compagnie. Je remercie à nouveau Jean-Rémy pour avoir facilité mon intégration et pour toute l'aide apportée dans cet exercice. Je salue également tous les étudiants que j'ai côtoyés et j'espère que vous garderez un bon souvenir de ces séances que nous avons partagées.

J'ai une pensée pour les personnes que j'ai pu rencontrer durant les conférences et autres manifestations scientifiques auxquelles j'ai eu la chance d'assister. Ces voyages ont été vraiment été enrichissants à plusieurs niveaux et j'en garde d'excellents souvenirs.

Je salue aussi tous mes anciens camarades étudiants et les enseignants que j'ai pu rencontrer pendant ces années à l'université. J'ai également une pensée pour toutes les personnes rencontrées lors pendant mon séjour à Aalborg, qui a constitué une des plus belles expériences qu'il m'a été donné de vivre.

Je salue également les personnes de l'US Galgon qui m'ont permis de me changer les idées et de me défouler sur les terrains de football de Gironde (non, je ne suis pas assez bon pour jouer plus loin).

Comment ne pourrais-je pas remercier mon petit chat adoré, pour ses nombreuses relectures, commentaires et suggestions, et ainsi pour toute l'attention qu'il a su porter à ce tapuscrit.

Enfin, je m'excuse envers toutes les personnes que j'ai oubliées dans ces remerciements, et qui ont compté pour moi pendant cette aventure.

Table des matières

CHAPITRE

1 Introduction

Nous introduisons dans ce chapitre notre contexte général relatif aux bibliothèques tierces, ainsi que les raisons pour lesquelles ces bibliothèques sont omniprésentes dans le développement logiciel. Nous précisons ensuite ce contexte en présentant le parallèle entre la maintenance logicielle et l'utilisation des bibliothèques tierces. Nous détaillons après cela les problématiques de recherche abordées dans cette thèse. Les solutions que nous proposons pour répondre à ces problèmes sont ensuite présentées et nos contributions mises en avant. Enfin, nous concluons par la structure générale du manuscrit.

Sommaire

1.1 Bibliothèques tierces et maintenance logicielle

La majorité des logiciels dépendent d'un ensemble de logiciels externes réutilisables couramment appelés bibliothèques tierces. Nous exposons dans cette section la position de ces bibliothèques dans le développement logiciel et clarifions en quoi elles doivent être considérées comme des entités de premier plan. Nous soulevons par la suite leur implication dans un contexte de maintenance logicielle pour justifier la nécessité de prendre en compte ces bibliothèques durant cette phase. Cette section a donc pour objectif de délimiter notre contexte de travail.

1.1.1 Réutilisation de code et bibliothèques

Concepts fondamentaux

La réutilisation logicielle consiste à tirer profit d'artefacts existants pour la conception d'un nouveau logiciel [Mcilroy, 1969]. Les communautés industrielles et de recherche ont admis que ce principe améliore la qualité d'un logiciel tout en diminuant les coûts liés à sa maintenance et à son développement [Griss, 1993; Lim, 1994; Joos, 1994]. Il peut s'appliquer lors de différentes étapes de l'évolution d'un logiciel. Ainsi, Basili et al. ont affirmé que la réutilisation est prospère lorsqu'elle est accomplie durant la phase de développement du logiciel [Basili, 1990]. À l'inverse, Baldassarre et al. soutiennent que ce principe est bénéfique lors de chaque phase du cycle de vie du logiciel [Baldassarre *et al.*, 2005].

La réutilisation logicielle concerne tous les types d'artefacts logiciels, tels qu'un fichier de code source, un fichier de configuration ou encore une suite de tests. Nous nous intéressons à la réutilisation de code, qui a d'ailleurs conduit au concept de *design to interface* visant à séparer l'interface d'un composant de son implémentation [Frakes et Kang, 2005]. Stroustrup a d'ailleurs mis en pratique ce concept en implémentant dans le langage C++ des composants abstraits et génériques pouvant être réutilisés [Stroustrup, 1996]. Ainsi, réutiliser du code source est prédominant dans le développement logiciel, et ce indépendamment du langage de programmation adopté [Frakes et Fox, 1995]. Même si les solutions que nous apportons dans cette thèse se veulent génériques à tout langage, nous nous restreignons au langage Java à typage statique, principalement pour des raisons techniques.

La réutilisation de code conduit à envelopper un ensemble fini de code dans une « boîte » qui fait office de paquetage. Celui-ci peut ensuite être distribué pour être mis à disposition des développeurs. Un tel paquetage est couramment appelé bibliothèque logicielle, et nous en proposons la définition suivante à partir de la terminologie suggérée par Mili et al. [Mili *et al.*, 1998] :

Définition 1.1 (Bibliothèque) *Une bibliothèque est une collection de fonctions implémentées et prêtes à être réutilisées. Elle fournit une interface de programmation qui constitue sa façade visible et rassemble l'ensemble des fonctions qu'elle propose. Cette interface est également appelée* ***API*** *(Application Programming Interface).*

Une bibliothèque Java propose ainsi un ensemble de classes, méthodes de classes, champs de classes, interfaces ou encore annotations. Nous qualifierons également ces entités de symboles ou éléments d'une API. L'implémentation de ces symboles est généralement cachée et interne à la bibliothèque.

De plus, nous introduisons la notion de versions pour une bibliothèque logicielle :

Définition 1.2 (Version de bibliothèque) *Une bibliothèque est un projet logiciel dont le cycle de vie entraîne la production de nouvelles versions à une fréquence variable selon*

chaque bibliothèque. Chaque version est une mise à jour de la bibliothèque qui apporte des corrections d'erreurs ou encore de nouvelles fonctionnalités. L'ensemble des versions d'une bibliothèque est donc ordonné chronologiquement.

Nous proposons également une définition de dépendance à une bibliothèque, bien qu'il en existe d'autres proposées dans la littérature comme celle de Nagappan et Ball [Nagappan et Ball, 2007] :

Définition 1.3 (Dépendance à une bibliothèque) *Un logiciel client C dépend d'une bibliothèque L s'il existe un traitement au sein de C qui dépend de l'exécution d'un traitement défini dans L.*

Les bibliothèques sont dites externes ou tierces lorsqu'elles sont maintenues par une organisation extérieure et indépendante du logiciel client, et ce sont précisément celles-ci qui nous intéressent. Par exemple, certaines ont pour objectif de faciliter la manipulation de données dans des formats comme XML ou JSON, ou même de proposer une conversion entre ces deux formats. D'autres permettent l'accès à des ressources extérieures comme des *web services* ou encore une base de données. La bibliothèque la plus populaire pour un logiciel Java est *JUnit* qui permet d'écrire et d'exécuter des tests unitaires, et est présente dans presque un projet Java *open source* sur deux (voir Chapitre 3).

Omniprésence des bibliothèques dans le développement logiciel

L'utilisation des bibliothèques est répandue dans le développement *open source* de logiciels Java. Thung et al. ont à ce sujet montré que 93,3 % des projets d'au moins 10 000 lignes de code dépendent d'au moins une bibliothèque, et que la moyenne du nombre de bibliothèques utilisées est de 28, ce qui représente une valeur importante [Thung *et al.*, 2013]. Schwittek et Eicker ont relevé une tendance similaire sur 36 logiciels J2EE *open source* utilisant en moyenne 70 bibliothèques tierces [Schwittek et Eicker, 2013]. Une étude de Heinemann et al. a montré que pour la moitié d'un corpus de logiciels Java *open source*, la quantité de code réutilisé excède la quantité de code nouvellement développé (en excluant les bibliothèques natives de Java) [Heinemann *et al.*, 2011]. De plus, les études que nous présentons dans cette thèse montrent qu'en très large majorité les projets *open source* Java utilisent au moins une bibliothèque tierce (autour de 90 %).

Ce phénomène s'observe également par l'existence de larges infrastructures à travers Internet faisant office de dépôts de bibliothèques tierces Java. Ainsi, le dépôt central de MAVEN recensait environ 67 000 bibliothèques distinctes en novembre 2013 [1]. Cette source de données est exploitée en abondance comme en témoignent les 70 millions de téléchargements hebdomadaires de bibliothèques (en décembre 2010) [2]. Ces chiffres confirment

1. http://maven.org
2. http://blog.sonatype.com/2010/12/now-available-central-download-statistics-for-oss-projects/

d'une part la quantité importante de bibliothèques disponibles pour les développeurs et d'autre part que ceux-ci tirent grandement partie de ces ressources.

En conclusion, la réutilisation de code source par l'utilisation des bibliothèques n'est pas qu'une théorie mais bien une pratique concrétisée et observée. Leur importance au sein de logiciels clients est donc indéniable et il est clair que leur absence rendrait non fonctionnels et non économiques une grande majorité de logiciels.

1.1.2 Implications dans la maintenance logicielle

La phase de maintenance est considérée comme le travail le plus coûteux au cours du cycle de vie d'un projet logiciel [Boehm et Papaccio, 1988; Abran et Nguyenkim, 1991]. Ce processus est défini comme un ensemble d'activités visant à corriger, adapter et faire évoluer le logiciel existant [Arthur, 1988]. La maintenance corrective a pour but de corriger les erreurs connues, alors que la maintenance adaptative œuvre pour garantir l'intégrité d'un logiciel lorsque son environnement est mis à jour, ou lorsque ses besoins évoluent. La maintenance perfective recherche à améliorer l'efficacité et la maintenabilité du logiciel.

Notre objectif est de justifier la position des bibliothèques tierces durant cette phase du cycle de vie du logiciel. Pour nous en convaincre, nous illustrons pour chaque type de maintenance mentionné précédemment des scénarios impliquant ces bibliothèques :

— Maintenance corrective : une erreur apparaît dans le logiciel et provient d'une de ses bibliothèques tierces. Or, les développeurs de la bibliothèque ont cessé de la maintenir et il n'y aura pas de nouvelle mise à jour. Cette bibliothèque ne répond plus aux exigences du projet et il faut envisager de la remplacer par une nouvelle possédant des fonctionnalités équivalentes.

— Maintenance adaptative : une des bibliothèques utilisées par le logiciel ne permet pas de satisfaire une nouvelle fonctionnalité que les développeurs veulent intégrer au logiciel. Cependant, les mainteneurs de la bibliothèque ne souhaitent pas intégrer cette fonctionnalité dans la prochaine version. Il faut à nouveau envisager de remplacer cette bibliothèque.

— Maintenance perfective : les développeurs souhaitent rendre le logiciel compatible avec plusieurs plateformes d'exécution telles que des terminaux mobiles. Or, il s'avère qu'une de leurs bibliothèques connaît des incompatibilités sur ces nouveaux environnements. Comme précédemment, si la bibliothèque n'est plus maintenue ou si ses développeurs ne sont pas intéressés pour corriger ces incompatibilités, elle devra certainement être remplacée.

Nous voyons ainsi que la phase de maintenance logicielle peut obliger à reconsidérer la pertinence des bibliothèques tierces et à procéder à certaines évolutions. Nous distinguons d'ailleurs deux types d'évolution : une mise à niveau de la bibliothèque ou un changement de bibliothèque.

Quand une erreur apparue dans un logiciel provient d'une bibliothèque, il est possible que des versions plus récentes de cette bibliothèque intègrent les corrections nécessaires. La mise à niveau de bibliothèque consiste ainsi à mettre à jour au sein d'un logiciel client une bibliothèque vers une version plus récente (Ex : mise à jour de *JUnit 3.8.1* vers *JUnit 4.8.1*). À partir d'une version source et version cible d'une bibliothèque, il se peut que des changements apportés à la version cible soient rétro-incompatibles. Le problème consiste à déterminer comment mettre à jour le code client pour pallier de telles modifications.

Cependant, les illustrations précédentes suggèrent que la mise à niveau de bibliothèque ne peut résoudre tous les problèmes. Dans une telle situation, il est nécessaire de remplacer la bibliothèque par une nouvelle qui permet de répondre aux besoins du logiciel. Celle-ci est donc qualifiée de similaire à l'actuelle et nous nommons cette évolution une migration de bibliothèque. Il est par exemple possible en Java de migrer de *JUnit* vers *TestNG*.

C'est précisément la migration de bibliothèque qui constitue le contexte des travaux de cette thèse. Nous dévoilons les trois points du processus d'une migration de bibliothèque sur lesquels nous nous focalisons :

— Le « Quoi ? » : vers quelles bibliothèques tierces un développeur peut-il migrer ?

— Le « Comment ? » : de quelle manière doit-il mettre à jour son logiciel pour devenir compatible avec la nouvelle bibliothèque ?

— Le « Qui ? » : quel expert de la nouvelle bibliothèque peut lui apporter une aide ?

1.2 Problématiques et objectifs de recherche

Après avoir précisé notre contexte de recherche et introduit nos problématiques, nous les présentons plus en détail dans cette section. Pour chacune d'elles, nous proposons une illustration et mettons en avant les défis qu'elle soulève et quelles sont ses perspectives.

1.2.1 Déterminer des bibliothèques candidates à la migration

Au sein d'un logiciel, nous appelons la pratique de remplacer une bibliothèque par une autre qui lui est similaire migration de bibliothèque.

Illustration. Un développeur utilise une bibliothèque pour traiter des documents XML et souhaite la remplacer. Il commence par utiliser des moteurs de recherche comme Google avec une requête telle que « Java XML library ». Les résultats obtenus sont difficiles à synthétiser pour déterminer les points forts et points faibles de chaque bibliothèque. De plus, ces informations ne sont pas nécessairement à jour et peuvent également s'avérer incomplètes. Ainsi, une fois les données rassemblées, le développeur n'est pas efficacement orienté vers le choix le plus pertinent.

Problèmes soulevés. Un développeur doit identifier des bibliothèques candidates vers lesquelles il peut migrer, et qui peuvent se substituer à celle qu'il souhaite remplacer. Puis, en supposant un ensemble de bibliothèques candidates établi, il doit enfin faire un choix pour ne conserver qu'une seule bibliothèque afin de migrer vers celle-ci. Or, il est probable qu'il n'ait pas connaissance de ces bibliothèques candidates et il devient délicat pour lui de prendre sa décision. Sans données supplémentaires, il prend le risque de migrer vers une bibliothèque encore moins pertinente que celle dont il souhaite se séparer.

Perspective. Lorsqu'un développeur souhaite migrer, il devient nécessaire de lui fournir à la fois une liste de candidats et des indicateurs de comparaison objectifs pour l'aider dans sa décision. En d'autres termes, le défi consiste à générer des catégories de bibliothèques qui peuvent être remplacées les unes par les autres. De plus, nous devons mettre à la disposition du développeur des indicateurs pour l'orienter vers des bibliothèques pertinentes. Actuellement, aucune étude n'a encore été réalisée dans ce sens.

1.2.2 Application d'une migration de bibliothèque

Appliquer une migration signifie transformer le code source d'un logiciel pour éliminer les références à la bibliothèque actuelle, que nous nommerons aussi bibliothèque source, et d'ajouter celles vers la nouvelle bibliothèque dite cible. Cette opération est possible si les bibliothèques source et cible proposent un ensemble de fonctionnalités similaires.

Illustration. Un développeur client de la bibliothèque *Commons-lang* utilise la méthode *Validate.isTrue(boolean)* qui retourne vrai si la condition passée en argument est vraie. Souhaitant migrer vers la bibliothèque *Guava*, le développeur doit parcourir cette nouvelle API pour finalement trouver une solution candidate, *Preconditions.checkArgument(boolean)*. Cette opération fut d'autant plus difficile que les noms des deux méthodes n'ont aucun point commun. De plus, rien ne lui garantit que ce remplacement est la meilleure solution. Il lui faut donc soit accepter cette fonction, soit continuer à parcourir l'API de la bibliothèque *Guava*.

Problèmes soulevés. Un développeur doit se familiariser avec une nouvelle interface de programmation, ce qui est connu comme une tâche complexe et coûteuse en temps [Robillard et Deline, 2011]. Il doit identifier les références de la bibliothèque source utilisées par son logiciel, et s'assurer de leur sémantique et de leur rôle. Puis, pour chacune d'elles, il doit extraire une ou plusieurs références correspondantes dans la bibliothèque cible. Cette opération est d'autant plus difficile que les structures des deux interfaces de programmation diffèrent intégralement, car elles ont été développées de façon indépendante.

Perspective. Il existe un besoin pour un développeur d'identifier automatiquement des correspondances entre des références similaires de deux interfaces de programmation. Nous devons lui proposer une solution en ce sens pour diminuer les coûts en temps et en énergie à mettre à jour son logiciel lors d'une migration de bibliothèque.

1.2.3 Expertise des développeurs en bibliothèques

La recherche d'experts en bibliothèques vise à identifier parmi un ensemble de développeurs, lesquels ont le plus de connaissances relatives à une ou plusieurs bibliothèques données. Il s'agit donc d'évaluer le niveau d'expertise de ces développeurs sur un ensemble de bibliothèques, pour pouvoir ensuite classer ces experts en fonction de leur niveau respectif.

Illustration. Aussi bien dans des équipes de développement de plusieurs dizaines de développeurs ou dans des communautés telles que celle d'ECLIPSE [3], il est difficile de déterminer le niveau des connaissances des développeurs concernant chacune des potentielles dizaines de bibliothèques qu'un logiciel utilise. Des canaux de communication tels que des forums ou listes de diffusion permettent d'interroger ses collaborateurs mais aucune réponse rapide et de qualité n'est garantie. Si une réponse est apportée, elle intervient souvent plusieurs heures ou jours après la découverte du problème. Le développeur a probablement déjà réussi à le résoudre, mais a dû en contrepartie y consacrer du temps. Celui-ci aurait pu être réduit si une personne compétente avait pu être directement contactée.

Problèmes soulevés. Parmi une population de développeurs, il faut déterminer lesquels ont une expertise sur une bibliothèque donnée. Nous devons tout d'abord définir le concept d'expertise de bibliothèque et comment cela se traduit en pratique. Il faut ensuite mettre au point le procédé de mesure des niveaux d'expertise, et établir si celui-ci doit être manuel (le développeur s'auto-évalue) ou automatique. Il y a aussi un besoin de comparer les experts pour pouvoir les ordonner selon leur niveau de compétence.

Perspective. Il n'existe actuellement pas de solution qui permet d'identifier des développeurs ayant une expertise dans des bibliothèques tierces. La recherche d'experts est une opération longue dont l'issue est incertaine si le développeur ne possède pas de données et d'outils supplémentaires. Achevée dans un temps efficace, elle améliore la qualité du travail des développeurs, notamment lors des tâches de maintenance.

3. http ://eclipse.org/

1.3 Solutions proposées et contributions

Nous présentons nos contributions qui répondent aux trois problématiques que nous avons détaillées précédemment. Les solutions qu'elles apportent sont motivées et détaillées. La méthodologie mise en place pour ces trois parties est similaire, puisque nous sommes à chaque reprise amenés à analyser des grandes populations de logiciels pour identifier certaines tendances, maximiser la quantité des résultats produits et évaluer la viabilité de nos solutions. Nous concluons cette section en présentant cette méthodologie, basée sur l'observation et l'analyse des changements des logiciels.

1.3.1 Déterminer des bibliothèques candidates à la migration

Notre solution repose sur l'observation des tendances de migrations de bibliothèque effectuées par une grande population de logiciels. Nous pensons qu'un développeur prend en considération les actions de ses pairs dans ses pratiques de développement. Ce phénomène plus général appelé tendance (ou effet de mode) étudié par Leibenstein stipule qu'une personne adoptera un comportement donné si une grande population de personnes s'y conforme [Leibenstein, 1950].

Par conséquent, un développeur identifie plus efficacement la bibliothèque vers laquelle migrer s'il dispose des tendances de migrations associées à sa bibliothèque actuelle. Les tendances permettent en effet de répondre aux questions suivantes : vers quelles bibliothèques cibles se sont orientés des logiciels existants ayant remplacé la bibliothèque source au cours de leur histoire ? Quelles sont les tendances de migration parmi les bibliothèques cibles ? Existe-t-il des bibliothèques dont le taux d'adoption est sensiblement supérieur à son taux d'abandon ? Les réponses à ces interrogations peuvent potentiellement permettre d'une part de faire apparaître un ensemble de bibliothèques similaires qui peuvent se substituer, et d'autre part de mettre en avant les bibliothèques populaires lors des migrations et à l'inverse celles fréquemment remplacées. Dans un tel cas, nous affirmons que ces tendances de migration offrent des indicateurs de recommandations de bibliothèques.

Nos contributions sur ce problème sont les suivantes :

— nous présentons un algorithme d'extraction des migrations de bibliothèque ;

— nous caractérisons le phénomène de migrations de bibliothèque à partir de deux cas d'études ;

— nous proposons des visualisations pour aider le développeur à la compréhension des tendances de migration ;

— nous étudions des perspectives d'automatisation afin d'améliorer la qualité des données produites.

1.3.2 Application d'une migration de bibliothèque

Pour aider un développeur à migrer d'une bibliothèque source vers une bibliothèque cible, nous proposons une approche basée sur l'observation de l'évolution du code source de logiciels ayant déjà accompli cette migration. L'idée est ainsi d'observer comment un code source client s'est adapté au remplacement d'une bibliothèque par une nouvelle. Cette analyse repose sur des calculs de différences à un fin niveau de granularité entre des versions successives d'un code source durant une période de migration. Notre hypothèse est qu'un développeur met à jour l'utilisation de l'ancienne bibliothèque par la nouvelle à des positions similaires dans un fichier de code source. En Java, la grande majorité des éléments mis à jour durant une migration sont les fonctions proposées par les classes de l'API de la bibliothèque cible. Nos contributions sur ce problème sont les suivantes :

— nous présentons un algorithme d'identification de logiciels ayant effectué une migration de bibliothèque (différent de celui présenté en Section 1.3.1) ;

— nous proposons une analyse fine de l'évolution du code source de tels logiciels pour extraire automatiquement des correspondances de fonctions ;

— nous discutons des résultats obtenus suite à la mise en application de notre approche ;

— nous introduisons et évaluons un filtre pour améliorer la précision des résultats.

1.3.3 Expertise des développeurs en bibliothèques

Pour déterminer l'expertise de développeur en bibliothèques tierces, nous proposons une approche automatique basée sur l'extraction de l'utilisation faite des bibliothèques par les développeurs. Cette approche repose sur le principe de Saint Thomas (*« Je ne crois que ce que je vois »*), en évaluant les développeurs uniquement sur leurs actions. Notre solution consiste à analyser les contributions des développeurs sur le code source d'un logiciel pour y détecter des utilisations de bibliothèque, témoignant d'une connaissance de ces bibliothèques.

Nos contributions sur ce problème sont les suivantes :

— nous proposons un modèle de représentation des expertises de développeurs sur les bibliothèques permettant de rechercher et de classer des experts selon plusieurs critères ;

— nous décrivons une implémentation basée sur l'analyse fine des contributions à partir des systèmes de contrôle de versions ;

— nous proposons un prototype nommé LIBTIC ainsi qu'un langage de requête pour manipuler le modèle de données obtenu ;

— nous déployons LIBTIC et soulignons l'intérêt de notre approche en présentant des cas d'utilisation dans un contexte de maintenance logicielle.

1.3.4 Méthodologie par l'observation et l'analyse des changements

Les solutions que nous venons de présenter ont pour point commun d'observer les actions effectuées par une population de logiciels et donc de développeurs. Cette méthodologie repose sur l'analyse de l'évolution des logiciels pour comprendre les différentes transformations qu'ils ont subies, mais également d'en extraire des tendances. L'analyse de dépôts logiciels ouvre l'accès à l'historique d'un logiciel et permet de comprendre ses changements ainsi que leurs causes et leurs effets [Kagdi *et al.*, 2007; Mens, 2008]. Cette pratique s'inscrit dans un domaine de recherche connu sous le terme de *MSR* (*Mining Software Repositories*) [Kagdi *et al.*, 2007]. Mettre en place une étude MSR implique des analyses de dépôts logiciels de différentes natures pour extraire et découvrir des informations relatives à l'évolution logicielle. Les trois principaux types de dépôts logiciels sujets à des études MSR sont :

— les systèmes de contrôle de versions. Exemple : détection de couplages entre composants d'un projet [Gall *et al.*, 1998], prédictions de *refactoring* [Ratzinger *et al.*, 2007] ;

— les systèmes de gestion et suivi d'erreurs (*bug trackers*). Exemple : corrélation entre cycle de développement et apparition d'erreurs [Khomh *et al.*, 2012], affectation de résolutions d'erreurs [Matter *et al.*, 2009] ;

— les archives de listes de diffusion. Exemple : comprendre le langage et le vocabulaire utilisés selon différents scénarios du cycle de développement [Rigby et Hassan, 2007], recommander des sujets pour les développeurs [Ibrahim *et al.*, 2010].

Il est bien entendu permis de croiser les analyses entre dépôts, afin d'étudier des relations entre artefacts contenus dans chaque dépôt distinct. Par exemple, des travaux ont abordé la prédiction d'erreurs à partir de l'historique des changements du code source [Giger *et al.*, 2011; Sisman et Kak, 2012].

Nous étudions dans nos travaux uniquement les systèmes de contrôle de versions (VCS), c'est-à-dire l'historique du code source d'un logiciel et de ses changements.

Remarque. Nous invitons le lecteur à prendre connaissance dans l'Annexe A de la taxonomie et des termes relatifs aux VCS utilisés tout au long de cette thèse.

Hemmati et al. ont parcouru 10 années de travaux dans le domaine du MSR et ont synthétisé qu'un processus typique de MSR se décompose en 4 étapes [Hemmati *et al.*, 2013] :

1. Collecte et préparation des données : comment extraire des données à partir des dépôts logiciels, quelles sont les relations inter-dépôts, comment fusionner les identités de développeurs sur plusieurs dépôts.
2. Synthèse : élaborer des modèles de prédiction, des classificateurs ou encore des arbres de décision à partir des données obtenues.
3. Analyse : interpréter les résultats, les inspecter manuellement, effectuer des tests statistiques, analyser des corrélations.

4. Réplication : partager des artefacts (dépôts, données ou encore outils) relatifs à l'expérimentation et permettre ainsi à d'autres chercheurs de répliquer l'étude.

Pour faciliter le travail des chercheurs, des outils existent pour chacune ou plusieurs de ces étapes. Cependant, pour capitaliser notre travail, nous avons conçu un *framework* pour nous assister dans la partie *Collecte et préparation des données*. Ce *framework* nommé HARMONY est développé depuis 2012 au sein de notre équipe de recherche au LaBRI [Falleri *et al.*, 2013]. Son objectif est de faciliter l'analyse de systèmes de contrôle de versions et d'abstraire l'extraction des données des dépôts. Il permet de définir et d'orchestrer des analyses qui seront exécutées sur un dépôt en utilisant le modèle et l'API fournis par HARMONY. Ce *framework* se situe donc dans la première étape du processus MSR que nous avons explicité. Notons cependant qu'il ne fait pas partie des contributions de cette thèse, et que des outils proposés récemment dans la littérature tels que Boa pourraient être utilisés en complément dans un travail futur [Dyer *et al.*, 2013].

L'intégralité des expérimentations conduites est donc menée avec cet outil. L'argument qui nous pousse à avoir utilisé notre propre *framework* est qu'il permet plus de liberté et de flexibilité dans l'écriture d'analyses sur des dépôts de code source, ce qui répond à nos besoins. Les outils alternatifs sont conçus plutôt pour répondre à un domaine de questions prédéfinies sur l'évolution d'un logiciel.

1.4 Structure

Nous présentons tout d'abord dans le Chapitre 2 un état de l'art afin de mieux familiariser le lecteur avec les travaux existants dans le contexte de cette thèse, et lui permettre de mieux appréhender le positionnement de nos contributions. Puis, dans le Chapitre 3 nous détaillons nos travaux sur l'extraction des tendances de migrations de bibliothèque. Nous présentons ensuite dans le Chapitre 4 nos contributions sur la génération de correspondances entre bibliothèques similaires. Dans le Chapitre 5, nous présentons le travail réalisé sur l'identification de développeurs experts dans le domaine des bibliothèques tierces. Finalement, nous concluons dans le Chapitre 6 en résumant les contributions et les principales perspectives.

CHAPITRE

2 État de l'Art

Ce chapitre présente l'état actuel du savoir sur nos trois problématiques liées à la migration de bibliothèques. Notre premier problème est : vers quoi migrer ? Nous présentons ici les travaux liés à l'évolution de l'utilisation des bibliothèques et à la catégorisation de logiciels. Vient ensuite le second problème : comment migrer ? Nous détaillons les approches consacrées à l'évolution des bibliothèques dans un logiciel client, aussi bien par la mise à niveau que par la migration. Pour finir, nous étudions l'existant autour de la question : avec qui migrer ? Les travaux présentés identifient des développeurs experts au sein d'un projet logiciel selon différents domaines d'expertise visés.

Sommaire

2.1 Déterminer des bibliothèques candidates à la migration

Nous présentons dans cette section un état de l'art sur l'évolution des bibliothèques tierces au sein de logiciels clients, et en particulier comment leur utilisation est extraite. Puis, étant donné que notre problématique est connexe au problème de la catégorisation de logiciels, nous présentons également un état de l'art relatif à ce sujet.

2.1.1 Évolution de l'utilisation des bibliothèques

Analyser les migrations de bibliothèques consiste à observer quelles bibliothèques ont été utilisées par un logiciel au cours de son cycle de vie. Les travaux que nous présentons ont, pour d'autres raisons, étudié l'évolution de cette utilisation.

Mileva et al. ont observé l'évolution des bibliothèques utilisées par 250 projets gérés sous Apache MAVEN sur deux années [Mileva *et al.*, 2009]. La popularité de ces bibliothèques a été examinée dans un premier temps avant de se focaliser ensuite sur la très populaire bibliothèque *JUnit* et l'évolution de son adoption à travers ses versions. Des situations de mises à niveau à des versions antérieures ont d'ailleurs pu être observées. Même si nos travaux étudient plutôt les évolutions inter-bibliothèques plutôt qu'entre les versions, cette étude nous montre qu'il est possible d'analyser l'évolution des bibliothèques utilisées par un corpus de logiciels.

Robbes et al. ont pour leur part mesuré comment réagissent un ensemble de logiciels clients d'une bibliothèque lorsque des symboles de son API deviennent obsolètes [Robbes *et al.*, 2012]. Ils ont notamment pu observer un effet cascade de répercussions sur ces clients qui mettaient à jour leur code régulièrement. Dans un esprit similaire, McDonnell et al. ont analysé l'évolution des changements de l'API Android et le temps nécessaire à une population cliente pour s'adapter à ces changements [McDonnell *et al.*, 2013]. Il ressort notamment que la plupart des clients sont réticents à s'adapter aux changements les plus récents, puisque ceux-ci sont sujets à faire apparaître de nouvelles erreurs.

Businge et al. ont observé un corpus de 467 greffons tierces de l'outil ECLIPSE et ont séparé ceux utilisant exclusivement des API stables et supportées par ECLIPSE (S-API) de ceux qui à l'inverse dépendent d'au moins une API instable et non supportée (NS-API) [Businge, 2013]. L'extraction des sections `import` des fichiers Java du code client permet d'identifier les API utilisées. Les compatibilités des S-API et NS-API avec les différentes versions du SDK de ECLIPSE y ont notamment été étudiées. Il a donc été possible d'analyser l'évolution des dépendances de ces greffons.

Wermelinger et Yu ont quant à eux étudié l'évolution de l'architecture du projet ECLIPSE à travers ses versions [Wermelinger et Yu, 2008]. Ils ont ainsi pu identifier une composante cœur de l'architecture, qui reste stable au cours du temps, et dont les dépendances n'évoluent que très peu, sauf lors de phases de maintenance marquées par d'importants *refactoring*. Les dépendances des composants ECLIPSE peuvent s'extraire à partir de fichiers spécifiques nommés `Manifest.mf`.

Bavota et al. ont étudié l'évolution sur 14 années de l'ensemble des projets Apache ainsi que l'évolution des inter-dépendances entre ces projets [Bavota *et al.*, 2013]. Ils ont observé que le nombre de projets au sein de cet écosystème augmente de façon linéaire, tandis que le nombre d'inter-dépendances évolue pour sa part de façon exponentielle. Encore une fois, certains projets clients n'effectuent pas de mise à niveau de version des dépendances lorsque de nouvelles versions apparaissent. Les inter-dépendances sont extraites par l'analyse de fichiers de configuration uniformément utilisés par les projets Apache. Dans un

esprit similaire, German et al. ont étudié l'évolution de l'écosystème du projet R afin de caractériser les différences entre le code des paquetages cœurs du projet et ceux étant le produit de contributions par des utilisateurs [German *et al.*, 2013]. Il en ressort notamment que les paquetages cœurs sont ceux dont les autres paquetages dépendent le plus.

2.1.2 Catégorisation de logiciels

Nous considérons qu'une bibliothèque peut être remplacée par une autre si celle-ci lui est similaire. Les travaux abordant la catégorisation de logiciels consistent à collecter et classifier un large corpus de logiciels selon leur domaine, avec pour objectif principal d'améliorer la maintenance de large dépôts de logiciels. Ils peuvent potentiellement être utilisés pour regrouper des bibliothèques. Nous résumons brièvement les principes adoptés par ces travaux, puis présentons les particularités de chacune des approches recensées.

L'idée générale est basée sur l'analyse sémantique latente (*Latent Semantic Analysis*, ou *LSA*). Un corpus de logiciels est représenté sous forme matricielle, où les lignes sont des termes, les colonnes des documents (ici des logiciels) et les cases complétées avec le nombre d'occurrences des termes dans le document. La nature de ces derniers dépend du choix de l'approche, comme nous le verrons par la suite. Une fois cette première étape accomplie, des algorithmes d'apprentissage et de classification (tels que les machines à vecteurs de support) sont utilisés pour répartir les logiciels en plusieurs catégories.

L'état de l'art est divisé dans la nature des termes utilisés pour représenter un logiciel. Une première solution consiste à analyser le code source des logiciels pour l'extraction de termes. MUDABlue adopte les identificateurs du code source (nom de variables, nom de méthodes, etc...) pour faire office de termes [Kawaguchi *et al.*, 2004]. À l'inverse, McMillan et al. ont choisi d'utiliser les appels de méthodes vers les bibliothèques natives de Java (JDK) comme termes [Mcmillan *et al.*, 2011]. L'idée sous-jacente est que deux logiciels similaires sont sujets à utiliser les mêmes bibliothèque.

Une seconde approche considère que les termes sont plutôt les méta-données décrivant un logiciel sur des plateformes de dépôts logiciels. Sur SourceForge ou Ohloh par exemple, les utilisateurs peuvent ajouter librement des mots-clés pour décrire un projet, en collaboration avec les autres utilisateurs. Les travaux de Thung et al. et Wang et al. utilisent cette source d'information [Thung *et al.*, 2012; Wang *et al.*, 2012]. Une extension de ce dernier travail y ajoute la notion de hiérarchies de catégories qui améliore la qualité de la classification produite [Wang *et al.*, 2013]. Il démontre d'ailleurs que l'utilisation des méta-données d'un logiciel et les appels vers des API génèrent des résultats similaires en termes de précision.

2.1.3 Synthèse

Premièrement, nous observons que l'étude de l'évolution des dépendances d'un logiciel est réaliste et a déjà été exploitée. De plus, elle n'est pas mise en péril par la taille de la

population observée ni par la taille elle-même des logiciels qui sont analysés. Par ailleurs, nous notons que l'extraction des bibliothèques utilisées par un logiciel peut s'effectuer par la lecture de fichiers de configuration ou de code source. Les objectifs des travaux existants ne correspondent cependant pas aux nôtres qui visent à étudier les migrations de bibliothèque.

Deuxièmement, la catégorisation logicielle permet de grouper des logiciels similaires mais s'avère incomplète pour atteindre nos objectifs. En effet, il n'y a d'une part aucune garantie que deux logiciels d'une même catégorie puissent en pratique être substitués l'un à l'autre. Il y a donc un premier problème de précision. L'autre inconvénient est qu'aucune attention n'est portée sur le classement des logiciels au sein d'une catégorie pour aider le développeur à identifier les plus pertinents. En revanche, il se peut que dans des travaux futurs nous ayons recours à ces techniques pour optimiser la précision des migrations que nous détectons.

2.2 Appliquer la migration de bibliothèque

Nous introduisons dans cette section les travaux existants dans le domaine de la migration de bibliothèque mais aussi de la mise à niveau de bibliothèque. Ce dernier problème est orthogonal au nôtre, comme discuté en Section 1.1.2. Cependant, l'état de l'art dans ce domaine est fourni et nous avons pu exploiter certains résultats obtenus, d'où l'importance de présenter ce problème. Nous débutons donc par l'étude de la mise à niveau de bibliothèque, puis nous discutons des travaux sur la migration de bibliothèque. Cette section est ponctuée par une synthèse.

Remarque. Nous nommerons *capture* d'une bibliothèque une version de la bibliothèque contenant uniquement son code source et aucune information sur son évolution. De façon analogue, une *capture* d'un client est une représentation instantanée du code source d'un logiciel client, sans considérer son historique.

2.2.1 Correspondances entre deux versions de bibliothèque

À notre connaissance, il existe trois catégories de techniques d'extraction des changements entre deux versions d'une bibliothèque et donc de son API : l'analyse basée alignement, client et enfin opération. Ceci fait office de structure pour la présentation de ces travaux.

Alignement

L'approche basée alignement compare deux versions d'une bibliothèque pour générer des règles d'évolution à partir des différences entre celles-ci. Un pré-traitement est généralement effectué pour déterminer l'ensemble des symboles de l'API source non présents

dans la version cible, et vice-versa. Il permet donc de construire l'ensemble des éléments ajoutés à la nouvelle version ainsi que ceux supprimés. Ces deux ensembles sont ensuite analysés pour en extraire des correspondances. Les comparaisons sont effectuées en incluant quatre catégories d'heuristiques qui peuvent être utilisées séparément ou en combinaison :

1. Similarité textuelle : les éléments de deux API sont représentés par des chaînes de caractères. Il s'agit soit du nom de l'élément (par exemple, le nom d'une méthode) ou alors la signature complète (le nom ainsi que les paramètres et le type de retour). Ensuite, la distance syntaxique entre paires d'éléments est calculée. L'idée est qu'une faible distance entre un élément supprimé et un élément ajouté indique qu'il y a probablement une relation entre ceux-ci (par exemple, renommage d'une méthode).
2. Similarité structurelle : les dépendances structurelles d'un élément d'une API sont considérées. Ainsi, l'héritage de classe, l'implémentation d'interface, la surcharge, la redéfinition de méthode, ou encore le type de retour d'une méthode sont des propriétés prises en compte. Une forte similarité structurelle indique probablement un renommage ou un déplacement d'une classe ou d'une méthode. Elle permet d'être plus précis que la similarité textuelle.
3. Similarité de dépendance : pour un élément d'une API, les autres éléments qui dépendent directement voire transitivement de celui-ci sont considérés. Ainsi, pour une méthode, les ensembles de méthodes appelantes et appelées sont extraits. L'idée est qu'une méthode supprimée puis remplacée est utilisée par un ensemble similaire de méthodes, et à l'inverse fait appel aux mêmes éléments d'un code source.
4. Similarité de métrique : généralement utilisées en complément, des métriques sont calculées sur chaque élément pour appuyer la comparaison entre deux éléments. Des exemples de métriques peuvent être le nombre de lignes de code, un pourcentage de *code clone*, ou encore la complexité cyclomatique. Il existe probablement une relation entre une méthode supprimée et une autre ajoutée si elles possèdent un même corps en termes de code.

Diehl et al. utilisent l'analyse syntaxique pour détecter des correspondances candidates [Weissgerber et Diehl, 2006]. Celles-ci sont ensuite classées en utilisant des outils de détection de *code clone* pour raffiner le score de similarité. Taneja et al. combinent la similarité textuelle à des métriques liées à la taille des classes et méthodes Java, notamment pour générer des correspondances [Taneja *et al.*, 2007].

À l'inverse DIFF-CATCHUP transforme les deux versions d'une bibliothèque en deux modèles écrits en langage UML (*Unified Modeling Language*), et utilise les similarités textuelles, structurelles et de dépendances [Xing et Stroulia, 2007]. Kim et al. produisent quant à eux des motifs de transformations textuelles uniquement [Kim *et al.*, 2007]. L'approche AURA implique les similarités textuelles et de dépendances ainsi que des itérations multiples [Wu *et al.*, 2010].

L'approche HIMA proposée par Meng et al. analyse à un niveau de granularité plus fin les évolutions apportées entre deux versions d'une bibliothèque [Meng *et al.*, 2012]. Pour cela, chaque révision du système de contrôle de versions associé à la bibliothèque est analysée, sur la période de temps comprise entre les deux versions. Ainsi, les règles de transformation sont inférées pour chaque révision. La principale caractéristique est le traitement du langage naturel sur les messages des *commits* pour identifier les transformations (ex : « *Rename Method* A.x() *into* A.y() »). Ceci constitue une première étape qui valide un ensemble de règles, et par la suite les similarités textuelles et de dépendances sont utilisées pour les éléments supprimés et ajoutés entre deux révisions (en excluant ceux détectés lors de la première étape). Pour terminer, toutes les règles générées par révision sont agrégées pour obtenir des règles finales résumant l'évolution entre les deux versions de bibliothèque. Le principal inconvénient de cette dernière approche est qu'elle est fortement dépendante de la qualité des messages de *commits*, qui est un facteur variable selon les projets logiciels.

Notons que seuls les algorithmes de AURA et HIMA ne nécessitent pas de fixation de seuils spécifiques pour fonctionner. La fixation obligatoire de tels seuils signifie que ces derniers impactent la qualité des résultats obtenus [Wu *et al.*, 2010].

Client

Une technique basée client vise à examiner comment un code client d'une bibliothèque s'adapte à l'évolution de la bibliothèque. Ainsi, Schäfer et al. infèrent des règles de transformation extraites en analysant un code client avant et après la mise à niveau vers une nouvelle version d'une bibliothèque [Schäfer *et al.*, 2008]. Cette analyse extrait des faits liés à l'utilisation de la bibliothèque, et les attache à des contextes d'utilisation (par exemple, une méthode de classe). Encore une fois, il est justement supposé que la mise à niveau d'une bibliothèque n'entraîne pas la mise à jour de tous les symboles d'une API utilisés dans de tels contextes. Des algorithmes de recherche de règles d'association permettent ensuite d'extraire des correspondances, et un filtrage par similarité textuelle est également appliqué.

Dans un esprit similaire, SEMDIFF étudie l'historique des changements de la bibliothèque et examine comment celle-ci s'adapte à son évolution [Dagenais et Robillard, 2009]. Ainsi, le code client se situe au sein-même de la bibliothèque. Il est à noter que ces deux approches nécessitent la fixation de seuils particuliers dans l'algorithme proposé.

La technique basée client a pour avantage de pouvoir détecter des changements conceptuels de la bibliothèque liés à des évolutions dans la manière de l'utiliser (et non à du *refactoring*). Une illustration de cela se situe dans les notes de changements de la bibliothèque *JDom* entre les versions 1.0.b7 et 1.0.b8 : *« Deprecated elt.addAttribute(String name, String prefix, String value). Instead, setAttribute() should be used »*. Nous constatons que la méthode dépréciée ici n'a pas été supprimée, et ne fera donc pas partie des éléments

supprimés dont on recherchera les remplaçants. Les résultats de Schäfer et al. ont d'ailleurs montré que 25 % des règles détectées s'expliquent par des changements conceptuels.

Opération

L'idée d'une approche basée opération est d'enregistrer au sein d'un IDE (*Integrated Development Environment)* les changements appliqués à une API lors de son évolution vers une nouvelle version. Ces changements sont synthétisés sous forme d'opérations de type *refactoring*. Ces opérations sont ensuite rejouées et reproduites directement dans un logiciel client de la bibliothèque. C'est le cas de CATCHUP ! proposé par Henkel et al. qui ont implémenté leur approche sous forme de greffon pour l'IDE ECLIPSE [Henkel et Diwan, 2005]. Dig et al. ont proposé un outil de configuration logicielle nommé MOLHADOREF, capable d'enregistrer les *refactorings* appliqués au code dans le but d'améliorer la qualité des fusions de code source [Dig *et al.*, 2007]. Même si la mise à niveau de bibliothèque n'est pas ciblée de façon explicite par les auteurs, ce travail peut être adapté pour cet objectif.

Synthèse

Nous présentons dans le Tableau 2.1 un résumé des travaux existants dans la mise à niveau de bibliothèque. Notons d'ailleurs que ces travaux emploient la même méthodologie pour évaluer leur performance, à savoir calculer la précision et le rappel sur l'ensemble des correspondances détectées. La précision évalue la part de correspondances vraies par rapport au nombre total de correspondances générées. Le rappel évalue le nombre de correspondances correctes par rapport au nombre total de correspondances connues.

Nous observons que la majorité de l'état de l'art est composée d'approches basée alignement. Celles-ci tendent d'ailleurs vers une absence de seuils dans les algorithmes proposés pour mettre en avant leur généralisation sur différents projets. Ce type de technique considère tous les éléments des bibliothèques et permet de garantir un score raisonnable de rappel sur les correspondances trouvées. L'inconvénient de la technique par alignement est qu'elle n'est pas robuste au code *préservé*. Un code préservé consiste en des éléments d'une API remplacés par de nouveaux éléments dans la nouvelle version de la bibliothèque, mais qui ne sont pas supprimés de cette nouvelle version afin de garantir des rétro-compatibilités. De tels éléments sont donc souvent exclus des comparaisons. À l'inverse, l'approche basée client garantit une meilleure précision et est robuste à la présence de code préservé. Elle a l'avantage de détecter des changements conceptuels hors *refactorings*, à l'inverse de l'approche basée alignement. En revanche, elle ne prend pas en compte les symboles d'une bibliothèque qui sont très peu voire non utilisés en interne, bien que ceux-ci aient subi des transformations entre les deux versions étudiées. Cette remarque est d'autant plus vraie si le code client est extérieur à la bibliothèque, comme nous le soulignons plus loin. Enfin, l'approche basée opération a pour avantage d'obtenir des taux de précision et de rappel significatifs. De plus, elle semble adaptée pour détecter des chan-

	Approche		Bibliothèque			Technique			Heuristique			
	Automatique	Seuils requis	Capture	Evolution	Message de *commits*	Interaction	Client	Alignement	Textuelle	Structurelle	Dépendances	Métriques
[Henkel et Diwan, 2005]	✓					✓						
[Weissgerber et Diehl, 2006]	✓	✓	✓					✓	✓			✓
[Kim *et al.*, 2007]	✓	✓	✓					✓	✓			
[Taneja *et al.*, 2007]	✓	✓	✓					✓	✓			✓
[Dig *et al.*, 2007]	✓			✓		✓			✓			
[Xing et Stroulia, 2007]	✓	✓	✓					✓	✓	✓	✓	
[Schäfer *et al.*, 2008]	✓	✓					✓					
[Dagenais et Robillard, 2009]	✓	✓		✓			✓					
[Wu *et al.*, 2010]	✓		✓					✓	✓		✓	
[Meng *et al.*, 2012]	✓			✓	✓			✓	✓		✓	

Tableau 2.1 : Tableau de synthèse des différentes approches traitant du problème de la mise à niveau de bibliothèque.

gements impliquant des éléments très peu voire pas du tout utilisés par l'API elle-même dans son implémentation. En revanche, elle est difficilement applicable et généralisable en pratique, étant donné qu'elle contraint les développeurs à maintenir et faire évoluer leur bibliothèque au sein d'un IDE ou d'un environnement très spécialisé.

Dans un récent travail proposé par Cossette et al., une étude rétroactive des changements à appliquer dans un contexte de mise à niveau de la bibliothèque a été menée [Cossette et Walker, 2012]. Toutes les adaptations à effectuer du côté client ont été relevées et catégorisées. Il s'avère que les approches existantes omettent certaines transformations complexes et nécessaires à la mise à niveau. Ces changements ne peuvent se résumer à une suite d'opérations de *refactoring*. La conclusion est qu'une intervention humaine est indispensable dans ce contexte, notamment de la part des personnes qui maintiennent une API. Également, il est difficile de prévoir la complexité pour un client d'appliquer une mise à niveau de bibliothèque.

2.2.2 Correspondances entre deux bibliothèques

Les travaux réalisés dans le but de faciliter la migration entre deux bibliothèques indépendantes sont peu nombreux. Nous en dégageons trois catégories de solutions.

Premièrement, Nita et Notkin ont proposé une technique nommée *jumelage* pour rendre un code client compatible avec deux API similaires [Nita et Notkin, 2010]. Un format de spécification de correspondances entre fonctions et classes des deux API Java est proposé, et il est à la charge du développeur de les indiquer. À partir de cela, une troisième API est générée pour abstraire le comportement commun aux deux API, et c'est précisément cette API qui sera utilisée par le client. L'objectif final est de diminuer la maintenance et la duplication du code côté client. Les correspondances sont cependant des relations directes de cardinalité 1 :1. Pour pallier ces limitations, Bartolomei et al. proposent une architecture dont l'objectif est d'intercepter les appels vers une bibliothèque source pour adapter le traitement en utilisant des symboles de la bibliothèque cible [Tonelli Bartolomei *et al.*, 2009, 2010]. Cette technique appelée *Wrapper* (inspirée directement du patron de conception *Adaptateur* proposée par Gamma et al. [Gamma *et al.*, 1995]), permet à un logiciel client de ne pas modifier son code source et de déléguer aux adaptateurs les appels vers les fonctionnalités équivalentes de la bibliothèque ciblée. Cette solution apporte ainsi plus de flexibilité.

Cependant, l'inconvénient de ces deux techniques est qu'elles nécessitent un travail manuel non négligeable pour déterminer l'ensemble des correspondances entre les fonctions de deux bibliothèques. Elles sont donc difficilement généralisables et automatisables, et c'est pourquoi une assistance est judicieuse dans ce contexte.

Dans un esprit différent, Gokhale et al. ont proposé ROSETTA, un prototype conçu pour générer des correspondances entre fonctions de deux bibliothèques [Gokhale *et al.*, 2013]. Cette approche nécessite un ensemble de paires d'applications similaires mais développées chacune en dépendant d'une bibliothèque différente. Si l'on suppose par exemple une application développée sur une plateforme Java2 Platform Micro Edition (JavaME) mais aussi sous Android, l'objectif est d'identifier les correspondances entre les API fournies par JavaME et Android. Le prototype analyse ensuite les traces d'exécution de l'application sur les deux systèmes et extrait les séquences d'appels aux API respectives de JavaME et Android. La comparaison des séquences des fonctions invoquées permet d'extraire des correspondances. Des facteurs comme la similarité textuelle et les dépendances aident à raffiner les règles. Cependant, la mise en application de cette approche est contraignante et difficilement généralisable, car cela nécessite des applications portées pour plusieurs bibliothèques, ce qui est rare (occasionnel pour les interfaces graphiques, mais beaucoup moins fréquent pour d'autres domaines).

Un récent travail de Bazelli et al. étudie la migration entre deux API REST (*REpresentational State Transfer*) du domaine du *web* [Bazelli *et al.*, 2013]. L'invocation de services REST produit des réponses dans des formats comme XML ou JSON. Les types de données et les valeurs des réponses sont utilisés pour déterminer des correspondances entre services. Il n'y a donc pas besoin de code client, mais uniquement de deux captures des API. Le contexte d'une API REST est cependant bien différent de Java, principalement du fait que la taille des API est bien inférieure dans le cas des bibliothèques REST, mais aussi parce que les types de données sont réduits à des types primitifs (nombre, chaîne de caractères

ou date), tandis qu'en Java les types de données peuvent être n'importe quels objets. Nous pensons qu'il n'est pas pertinent de comparer des méthodes de classe Java à des services REST.

2.2.3 Synthèse

Au vu de l'état de l'art sur la mise à niveau de bibliothèque, l'approche basée alignement ne semble pas adaptée à notre problème étant donné que deux API indépendantes diffèrent dans la totalité de leurs symboles. De plus, nous n'avons aucune garantie concernant la similarité textuelle, à savoir que deux méthodes similaires dans chaque bibliothèque respective auront un nom semblable. Cette supposition est même dangereuse et inadaptée comme nous le prouverons dans le Chapitre 4. À l'inverse, la technique basée sur l'analyse de code client semble plus judicieuse, puisqu'elle observe comment un client a effectué une mise à niveau de bibliothèque. Enfin, dans un souci d'être le plus généralisable possible, nous écartons définitivement les techniques basées opérations pour ne pas contraindre le déploiement de la solution proposée à des environnements de développement.

Aucun travail existant sur les migrations de bibliothèques ne se base sur l'observation et l'apprentissage de logiciels clients ayant déjà effectué une migration de bibliothèque. Pour autant, les conclusions tirées précédemment nous confortent dans notre choix d'adopter ce type de solution. Notre objectif n'est donc pas de modifier un code client mais plutôt de guider le développeur vers les correspondances de fonctions existantes.

Il nous paraît d'ailleurs pertinent de mentionner des travaux caractérisant l'utilisation générale des bibliothèques par des logiciels clients. Ceux-ci ont démontré qu'une API de bibliothèque possède en moyenne 20 % de « points chauds » qui sont des symboles utilisés par une majorité des logiciels clients, et qu'à l'inverse il existe 80 % de « points froids » qui ne sont que très rarement voire jamais utilisés [Holmes et Walker, 2007; Thummalapenta et Xie, 2008; Lämmel *et al.*, 2011; Okur et Dig, 2012; Raemaekers *et al.*, 2012]. Ces chiffres indiquent que notre approche ne sera pas en mesure de détecter une part importante d'éventuelles correspondances, car une majorité des méthodes de chacune des bibliothèques ne sont pas référencées dans les projets clients. En revanche, il est plus probable que nous puissions déterminer des correspondances sur des éléments régulièrement utilisés. Il sera donc plus intéressant de fournir ce type de correspondances à un développeur puisqu'elles ciblent les parties centrales des bibliothèques.

2.3 Expertise des développeurs en bibliothèques

La recherche de développeurs experts soulève un premier problème concernant la collecte des niveaux d'expertise des développeurs. Une auto-évaluation ou un système d'évaluation par les pairs posent des problèmes liés à la subjectivité d'opinion ainsi qu'à la

maintenance des informations. Il est en effet coûteux de demander à chaque personne de mettre à jour un profil personnel de compétences, puisque celui-ci est amené à évoluer en permanence. Ainsi, maintenir le niveau des compétences de développeurs au sein d'une équipe est un processus qui devient rapidement complexe s'il n'est pas entièrement automatisé. Vivacqua et Lieberman ont par le passé souligné l'importance de proposer des solutions automatiques pour éviter ces contraintes [Vivacqua et Lieberman, 2000]. C'est pourquoi nous pensons également adopter une solution de ce type.

Intuitivement, l'expertise de bibliothèques tierces s'exprime dans le code source d'un logiciel. Pour cette raison, nous présentons dans cette section un ensemble de travaux portant sur la notion d'expertise relative au code d'un logiciel. Nous les classifions selon la nature de la technique proposée pour mesurer l'expertise : l'implémentation et l'utilisation.

2.3.1 Expertise par implémentation

L'expertise par implémentation suppose que les développeurs ont des compétences sur un code source s'ils ont contribué sur ce code. McDonald et Ackerman ont proposé le concept de fraîcheur en mémoire, en définissant une personne experte d'un fichier comme celle ayant édité pour la dernière fois ce fichier [McDonald et Ackerman, 2000]. Une idée alternative proposée par Mockus et Herbsleb suggère que l'expert sera la personne ayant le plus souvent édité un fichier [Mockus et Herbsleb, 2002].

Girba et al. ont dans un esprit similaire proposé une approche qui définit l'expert d'un fichier selon deux critères : le pourcentage de lignes ajoutées et supprimées (extrait depuis les journaux d'un système de contrôle de versions), et la date à laquelle les modifications ont été effectuées [Girba *et al.*, 2005].

L'approche proposée par Kagdi et al. analyse des systèmes de contrôle de versions pour déterminer des experts sur un fichier de code source [Kagdi *et al.*, 2008]. Pour un développeur et un fichier donnés, le nombre de *commits*, de jours d'activité impliquant des *commits* le fichier et le nombre récent (selon une fenêtre de temps définie) de jours d'activité impliquant des *commits* sur le fichier sont combinés pour générer une liste ordonnée de développeurs experts.

Linares-Vásquez et al. utilisent les méta-données fournies par les commentaires du code source pour déterminer l'expert d'une classe ou d'une méthode Java [Linares-Vasquez *et al.*, 2012]. La technique suppose que le code source est commenté selon les conventions de l'outil JavaDoc, indiquant l'identité d'un auteur pour chacune des méthodes des classes. C'est précisément cet auteur qui devient l'expert dans ce contexte. Cette approche n'a donc pas recours à l'historique du logiciel client.

Pour finir, Servant et al. ont proposé une technique pour déterminer quel développeur est le plus à même de corriger un test unitaire défectueux en analysant l'historique du changement du code source [Servant et Jones, 2012]. Cette technique construit un graphe d'histoire des instructions du code (les lignes de code) pour déterminer les développeurs experts des parties responsables du test défectueux. Une analyse dynamique de l'exécution

des tests est nécessaire pour identifier quelles instructions sont appelées par chacun des tests.

D'autres approches visent à évaluer plus finement le niveau d'expertise réel des développeurs sur une entité de code source. Pour cela, elles mesurent grâce à des IDEs spécialisés les interactions entre un développeur et un code source donné. Leur principale motivation est de masquer les biais induits par l'utilisation des systèmes de contrôle de versions. Il se peut en effet qu'un *commit* ne soit pas représentatif de l'effort et du temps dévolus à modifier un fichier de code source. De plus, selon les habitudes des développeurs, certains seront amenés à *commiter* de façon plus régulière leur travail que d'autres. Il a été montré que ce comportement est parfois expliqué par la crainte des conflits, c'est-à-dire que d'autres développeurs travaillent en même temps sur le même fichier, comme indiqué par Grinter et al. [Grinter, 1996]. Cependant, cette fois encore, ce type de technique nécessite le déploiement d'outils spécifiques pour enregistrer les interactions des développeurs avec un code source.

Un exemple d'approche dans ce sens a été proposé par Hattori et Lanza qui déterminent l'expert d'un fichier comme celui ayant effectué le plus de changements sur ce fichier [Hattori et Lanza, 2009]. Chaque nouvel enregistrement de fichier dans un IDE est équivalent à un *commit* mais à une échelle plus fine.

Les travaux de Fritz et al. ont montré que la fréquence et le facteur récent des interactions d'un développeur avec un code source sont des bons indicateurs pour mesurer son niveau de connaissance [Fritz *et al.*, 2007]. Leur étude a consisté à suivre et questionner des développeurs Java sur leur niveau de connaissance de certaines parties d'un code source, en considérant celles avec lesquelles ils interagissaient le plus régulièrement. Une mesure de DOI (*Degree of interest*) a été proposée pour représenter la connaissance qu'a un développeur d'un code source donné.

Robbes et al. ont récemment proposé une approche combinant l'expertise sur une entité de code source ainsi que l'analyse des interactions des développeurs avec le code pour accomplir une tâche donnée [Robbes et Röthlisberger, 2013]. Il s'agit d'utiliser des outils comme MYLYN [1] qui enregistrent le temps et les opérations nécessaires à une personne pour achever une tâche de développement. La métrique d'expertise proposée suggère qu'un développeur expert accomplira plus rapidement une tâche qu'une personne moins experte, et ce pour un travail équivalent.

2.3.2 Expertise par utilisation

L'expertise par utilisation privilégie les développeurs ayant directement utilisé une entité logicielle donnée.

Schüler et al. ont proposé une approche qui prend en considération l'utilisation de méthodes Java comme indicateurs d'expertise [Schuler et Zimmermann, 2008], en considé-

1. http://www.eclipse.org/mylyn/

rant la fréquence d'utilisation de ces méthodes. Ma et al. ont par la suite démontré que l'expertise par l'utilisation génère une précision semblable à celle par l'implémentation [Ma *et al.*, 2009]. Dans ces deux travaux, l'expertise est calculée à partir d'analyses sur les systèmes de contrôle de versions. A chaque reprise, l'outil APFEL proposé par Zimmermann a été utilisé pour analyser du code Java [Zimmermann, 2006]. La portée de l'analyse est cependant restreinte à un seul fichier, c'est pourquoi les méthodes sont identifiées par leur nom et leur nombre de paramètres, ce qui peut engendrer des problèmes de précision puisqu'une telle identité peut être partagée par plusieurs méthodes au sein de plusieurs classes. APFEL représente chacune des méthodes de classe (appelées aussi contexte) par une séquence de *tokens*. Des différences ensemblistes sont ensuite calculées entre les contextes présents dans les deux versions d'un fichier.

Une alternative proposée par Ye et al. est un outil à la recherche de développeurs ayant des connaissances sur des éléments d'une API donnée [Ye *et al.*, 2007]. L'idée est la suivante : un développeur importe dans l'outil un programme binaire Java. Le code étant compilé, il est possible d'extraire les signatures complètes de tous les appels de méthodes aussi bien internes qu'externes. Par la suite, le développeur se voit proposer une liste de ces méthodes, dans laquelle il doit indiquer si oui ou non il admet avoir connaissance de cette méthode. Si oui, il sera considéré comme étant un expert de cette méthode. Cet outil n'est donc pas compatible avec les systèmes de contrôle de versions, car un seul développeur importe dans le système un programme entier, quel que soit le nombre total de contributeurs sur le projet. De plus, le procédé étant semi-automatique, il est difficilement applicable en pratique car il est clair que l'étape de filtrage est coûteuse en temps. Il faudrait dans l'idéal que chaque développeur indique les méthodes qu'il connaît, ce qui s'avérerait relativement long. Ceci nous conforte dans notre intuition que la recherche d'experts se doit d'être un procédé automatisable pour pouvoir être exploitable en pratique.

2.3.3 Synthèse

Un récapitulatif des approches présentées dans cette section est disponible dans le Tableau 2.2. Le critère *temps* indique que l'approche tient compte du moment où les développeurs ont agi sur un code source et privilégiera les actions récentes. Le critère *fréquence* indique que le nombre d'actions est pris en compte pour comparer deux développeurs ayant agi sur un même code.

Nous observons que la majorité des travaux utilisent l'évolution d'un code source pour en extraire les contributions de chaque développeur sur celui-ci. Nous voyons également que la fréquence est considérée à chaque fois et que l'aspect temps revient régulièrement. Les auteurs des actions sont identifiés à partir des données contenues dans les systèmes de contrôle de versions ou dans des outils comme MYLYN. Seule la technique proposée par Linares-Vasquez et al. suggère une alternative en analysant les méta-données contenues dans le code source pour identifier l'expert d'une entité donnée.

	Client		Unité ciblée			Technique		Option			Auteur			
	Capture	Evolution	Fichier	Entité Java	Ligne d'instruction	Implémentation	Utilisation	Interaction (IDE, MYLYN)	Temps considéré	Fréquence considérée	Fournisseur du code	Auteur du *commit*	Meta-données du code	Acteur de l'interaction
[Mockus et Herbsleb, 2002]		✓	✓			✓				✓		✓		
[Girba *et al.*, 2005]		✓	✓			✓			✓	✓		✓		
[Fritz *et al.*, 2007]		✓		✓			✓	✓	✓	✓				✓
[Ye *et al.*, 2007]	✓			✓			✓			✓	✓			
[Kagdi *et al.*, 2008]		✓	✓			✓			✓	✓		✓		
[Schuler et Zimmermann, 2008]		✓		✓			✓	✓		✓		✓		
[Hattori et Lanza, 2009]		✓	✓			✓		✓	✓	✓				✓
[Servant et Jones, 2012]		✓			✓	✓			✓	✓		✓		
[Linares-Vasquez *et al.*, 2012]	✓			✓		✓				✓			✓	
[Robbes et Röthlisberger, 2013]		✓	✓			✓		✓	✓	✓				✓

Tableau 2.2 : Synthèse des approches sur l'identification d'experts dans différents domaines liés au code source d'un logiciel.

En ce qui nous concerne, l'approche par implémentation n'est pas applicable dans le contexte des bibliothèques tierces car ce sont par nature des parties externes d'un logiciel. Toutefois, nous pourrions considérer l'expert d'une bibliothèque en utilisant l'expertise par implémentation. L'idée serait d'identifier les méthodes qui utilisent cette bibliothèque et de déterminer le développeur ayant le plus contribué au développement de ces méthodes. Cette idée n'a cependant pas été retenue, car nous estimons que l'utilisation de bibliothèques tierces dans un fichier représente une très faible proportion de ses lignes de code, et qu'une analyse très fine de l'utilisation semble ainsi plus judicieuse.

2.4 Conclusion

Les trois principaux enseignements à tirer de notre présentation de l'état de l'art sont les suivants :

— Parmi tous les travaux sur l'évolution de l'utilisation des bibliothèques, aucun d'autre eux n'a été effectué pour observer des migrations de bibliothèque. Ceci motive notre première contribution sur la définition d'une approche d'extraction de ces migrations.

— Les travaux existants dans la génération de correspondances entre deux bibliothèques sont difficilement généralisables. Nous souhaitons traiter ce problème via une technique nouvelle inspirée de l'état de l'art sur la mise à niveau de bibliothèque.

— Les bibliothèques tierces n'ont jamais été considérées comme des domaines d'expertise pour les développeurs. Pourtant, elles ont un rôle clé dans un projet logiciel, et cela renforce notre motivation à traiter ce problème.

CHAPITRE

3 Déterminer des bibliothèques candidates à la migration

Ce chapitre aborde nos contributions sur la migrations de bibliothèque en rappelant tout d'abord le problème étudié. Nous présentons ensuite le modèle de représentation des bibliothèques tierces et décrivons notre approche d'extraction des migrations de bibliothèque. Nous étudions sa mise en application dans deux cas d'étude sur deux sources de données différentes. Nous discutons ensuite des résultats obtenus nous permettant d'affirmer que notre approche est réaliste en pratique, tout en ayant certaines limites, notamment concernant la précision. Par la suite, nous exploitons ces tendances de migrations au moyen de techniques de visualisation démontrant l'intérêt pour un développeur d'avoir à disposition de telles informations. Dans une dernière partie, nous fournissons des recommandations quant à la réplication de nos expérimentations. Nous y proposons notamment une identification automatique des règles de migration afin de pallier le problème de précision rencontré précédemment. Nous détaillons pour terminer les obstacles à la validité de ce travail et ouvrons différentes perspectives futures.

Sommaire

3.1 Introduction

Nous abordons dans ce chapitre la problématique introduite en Section 1.2.1. Pour rappel, elle fait référence au scénario suivant : un développeur souhaite migrer une de ses bibliothèques actuelles vers une nouvelle proposant des fonctionnalités équivalentes. Il doit donc tout d'abord identifier des bibliothèques similaires pouvant se substituer à cette bibliothèque source qu'il souhaite remplacer. Il doit ensuite décider quelle bibliothèque cible retenir parmi les candidates identifiées.

Notre proposition pour aider un développeur dans ce contexte est basée sur l'observation des tendances de migrations de bibliothèques. La solution consiste à extraire et analyser les remplacements de bibliothèques effectués par d'autres logiciels. Il y a deux raisons pour lesquelles cette solution nous semble pertinente. Premièrement, si nous observons qu'une bibliothèque a été remplacée par une autre pour une même utilisation, alors ces deux bibliothèques sont similaires et peuvent être substituées l'une à l'autre. L'analyse des migrations de bibliothèque permet alors d'obtenir des groupements de bibliothèques similaires. Deuxièmement, ces tendances de migrations véhiculent des informations ayant un intérêt pour un développeur. Ces informations sont générées sous formes d'indicateurs d'aide à la décision. Ainsi, nous mettons en avant les bibliothèques fortement adoptées dans un contexte de migration et donc implicitement à recommander. À l'inverse, nous tâchons de faire apparaître les bibliothèques abandonnées à de nombreuses reprises et qui doivent être écartées de toute recommandation.

3.2 Modèle abstrait

Les définitions et notations que nous présentons sont nécessaires à la compréhension de ce chapitre ainsi que de celle du manuscrit. Nous reprenons et formalisons les définitions préliminaires exposées au début de ce manuscrit en Section 1.1.1.

Soit L un ensemble de bibliothèques. Chaque bibliothèque $l \in L$ est une entité individuelle qui possède un nom et qui exporte un ensemble de symboles accessibles via son interface de programmation.

Définition 3.1 (Symboles d'une API) *Soit $l \in L$ une bibliothèque. Nous notons S_l l'ensemble des symboles permettant en Java d'identifier les classes, interfaces, annotations, méthodes et champs de classes proposés dans l'interface de programmation de l. Chaque symbole possède un nom unique pour chaque bibliothèque $l \in L$. Nous définissons la fonction* library_L *qui à partir d'un symbole s, identifie la bibliothèque l à laquelle il appartient. Ainsi, nous avons* $\text{library}_L : \text{String} \rightarrow L \cup \emptyset$ *tel que* $\text{library}_L(s) = l$ *si et seulement si* $s \in S_l$. *Aussi,* $\text{library}_L(s) = \emptyset$ *si est seulement s n'est défini dans aucune des bibliothèques de L.*

Par exemple, le Tableau 3.1 propose pour les bibliothèque `junit` et `testng` un extrait de trois symboles de leurs API respectives.

Tableau 3.1 : Extrait des symboles des bibliothèques `junit` et `testng`.

`junit`	`testng`
org.junit.Assert.assertTrue(boolean)	org.testng.Assert
org.junit.Test	org.testng.annotations.BeforeClass
org.junit.runner.Description.isEmpty()	org.testng.xml.XmlSuite.getTest()

Tableau 3.2 : Exemple d'utilisation de bibliothèques par deux projets *Foo* et *Bar*.

Version i	$dep_{Foo}(i)$	$dep_{Bar}(i)$
1	{`junit`}	{`testng`}
2	{`junit`, `log4j`}	{`testng`}
3	{`junit`, `log4j`}	{`testng`, `slf4j`}
4	{`log4j`}	{`junit`, `slf4j`}
5		{`junit`, `slf4j`}
6		{`junit`, `slf4j`}

Définition 3.2 (Versions de bibliothèque) *Pour toute bibliothèque $l \in L$, il existe un ensemble V_l de versions ordonnées chronologiquement. Pour une version $v \in V_l$ de bibliothèque, nous définissons son ensemble de symboles associés S_l^v. Ainsi, lorsque nous ne considérerons pas les versions des bibliothèques, nous obtenons :*

$$S_l = \bigcup_{v \in V_l} S_l^v \,.$$

Pour des raisons de simplicité, nous supposons que l'ensemble des versions d'une bibliothèque possède un ordre total.

Définition 3.3 (Dépendances aux bibliothèques) *Soit P un ensemble de projets logiciels. Nous considérons que chaque projet $p \in P$ possède un ensemble ordonné de versions $V_p \subset \mathbb{N}$. Ces versions sont triées par ordre chronologique. Nous considérons qu'un projet peut dépendre d'une bibliothèque à une de ses versions s'il utilise au moins un symbole de la bibliothèque à cette version. Ainsi, pour un projet $p \in P$ à la version $i \in V_p$, nous définissons $dep_p(i) : V_p \to \mathcal{P}(L)$ l'ensemble des bibliothèques utilisées par le projet, $\mathcal{P}(L)$ étant l'ensemble des parties de L.*

Pour mieux comprendre, soit $L = \{$`junit`, `testng`, `log4j`, `slf4j`$\}$. Les dépendances aux bibliothèques de deux projets fictifs *Foo* et *Bar* composés de respectivement 4 et 6 versions sont indiquées dans le Tableau 3.2. Le projet *Foo* a adopté la bibliothèque `log4j` à partir de sa version 2 mais a également abandonné `junit` à sa version 4. À l'inverse, le projet *Bar* a inclus `slf4j` à partir de la version 3, puis a remplacé `testng` par `junit` à la version 4.

3.3 Extraction des migrations de bibliothèque

L'objectif de l'approche présentée ici est d'extraire des tendances de migrations en observant une large population de projets logiciels. Nous supposons deux ensembles définis de projets P et de bibliothèques L. Il est nécessaire de déterminer pour un projet donné les migrations qu'il a effectuées par le passé. Nous devons définir ce qu'est une migration de bibliothèque et comment elle est représentée dans l'historique d'un projet. Cependant, il est difficile de distinguer une migration d'une suppression et ajout de bibliothèques indépendantes. Ce scénario est envisageable au vu de l'utilisation massive des bibliothèques par les projets. C'est pour cela que notre algorithme génère en fait une sur-approximation des migrations à partir d'un projet. Un premier ensemble de migrations, qualifiées de candidates, est ainsi formé, et une analyse manuelle a posteriori permet finalement de filtrer les migrations valides.

Nous débutons cette section en présentant notre modèle de migration de bibliothèque. Nous décrivons ensuite l'algorithme d'identification des migrations candidates et comment celles-ci sont filtrées. Enfin, nous introduisons la notion de catégorie de bibliothèques, qui permet de regrouper des bibliothèques similaires.

3.3.1 Migration de bibliothèque candidate

Une migration de bibliothèque a lieu lorsqu'un projet remplace une bibliothèque par une autre qui lui est similaire. La définition de similarité entre bibliothèques est cependant floue, et il se peut qu'une bibliothèque soit remplacée par une autre qui ne lui est pas similaire. Nous introduisons pour cette raison la notion de migration candidate. Une migration candidate est observée lorsqu'un projet cesse d'utiliser une de ses bibliothèques et dans le même temps commence à en utiliser une autre.

Définition 3.4 (Migration candidate) *Un projet p migre une bibliothèque $s \in L$ vers une bibliothèque $t \in L$ pendant une période définie entre deux versions $i, j \in V_p$, $i < j$, si et seulement si (1) il dépend de s et non de t à la version i (c'est-à-dire $s \in \text{dep}_p(i)$ et $t \notin \text{dep}_p(i)$), et (2) il dépend de t et non de s à la version j (c.à.d $s \notin \text{dep}_p(j)$ et $t \in \text{dep}_p(j)$), et (3) il dépend des deux bibliothèques pendant la migration (c.à.d. $\forall v \in]i, j[, s \in \text{dep}_p(v)$ and $t \in \text{dep}_p(v)$. Une migration candidate est un 5-uplet $(p, i, j, s, t) \in P \times \mathbb{N} \times \mathbb{N} \times L \times L$. Nous notons M toutes les migrations candidates observées dans chacun des projets de P.*

La Figure 3.1 expose un exemple d'évolution de dépendances de bibliothèques pour un projet $p \in P$. Ce dernier a utilisé deux bibliothèques au cours de son histoire, `junit` et `testng`. Au regard de cet exemple, nous pouvons identifier une migration candidate : $(p, 18, 23, \textit{junit}, \textit{testng})$.

Algorithme 1 Algorithme d'extraction des migrations de bibliothèque candidates

Requiert: p ▷un projet
Requiert: V_p ▷les versions de p
Requiert: dep_p ▷une fonction qui calcule les bibliothèques utilisées
Requiert: $tick$ ▷mémorise les versions auxquelles les bibliothèques sont introduites
  Cand $\leftarrow \emptyset$ ▷les migrations candidates
  Dep $\leftarrow \emptyset$ ▷l'ensemble des bibliothèque actives
  pour tout $i \in V_p$ **faire** ▷ordre croissant
    $L \leftarrow dep_p(i)$ ▷inclure les bibliothèques à la version i dans L
    pour tout $l \in L \setminus Dep$ **faire** ▷pour chaque bibliothèque ajoutée à i
      $append(Dep, l)$ ▷les ajouter à la liste des bibliothèques actives
      $tick(l) \leftarrow i$
    fin pour
    pour tout $s \in Dep \setminus L$ **faire** ▷pour chaque bibliothèque s supprimée à i
      pour tout $t \in tail(Dep, s) \cap L$ **faire** ▷pour les bibliothèques actives hormis s
        si $tick(s) < tick(t)$ **alors** ▷si s a été introduite avant t
          $Cand \leftarrow Cand + (s, t)$ ▷mettre à jour les migrations candidates
        fin si
      fin pour
      remove(Dep,s) ▷supprimer s des bibliothèques actives
    fin pour
    retourner Cand
  fin pour

3.3.2 Identification semi-automatique des migrations

Notre algorithme d'identification des migrations candidates est capable de détecter si deux bibliothèques, s et t, ont été respectivement ajoutée et supprimée pendant une même période de temps. Cet algorithme (voir Algorithme 1) itère à travers les versions d'un projet et maintient l'ensemble des bibliothèques actives à chaque version. Puis, lorsqu'une bibliothèque s est supprimée, une migration candidate est créée pour chaque bibliothèque cible t introduite strictement après l'ajout et avant ou pendant la suppression définitive de s.

L'identification automatique des migrations candidates produit une large quantité de migrations contenant assurément une grande part de faux positifs. Un faux positif est une migration candidate (s, t) où s et t ne sont pas considérées comme similaires. Cette situation fréquente apparaît simplement lorsqu'une bibliothèque est supprimée d'un projet, puis une autre est ajoutée dans la même période de temps, mais aucun lien particulier n'existe entre elles.

3.3.3 Règle de migration et filtrage

Pour mettre en avant quelles bibliothèques sont les cibles ou sources des migrations candidates, nous introduisons le concept de règle de migration qui porte uniquement sur la source et la cible d'une migration sans tenir compte du projet ou de la date.

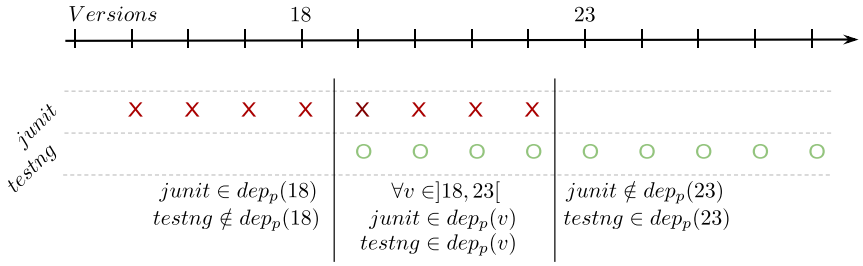


FIGURE 3.1 : Exemple de migration au sein d'un projet $p \in P$.

Définition 3.5 (Règle de migration candidate) *Une règle de migration est un couple* $(s,t) \in L^2$ *tel qu'il existe au moins un projet* $p \in P$ *ayant migré de* s *vers* t *au cours de son évolution. Cette règle sera également notée* $s \rightarrow t$*. Nous appelons* R *l'ensemble de toutes les règles de migration candidates.*

Si nous reprenons l'exemple de la Figure 3.1, nous obtenons une unique règle de migration, $R = \{(junit, testng)\}$. Également, à chaque règle de migration est associé un score :

Définition 3.6 (Score d'une règle de migration) *Soit une règle de migration* $(s,t) \in L^2$*. Nous notons* $sc(s,t)$ *le score de la règle correspondant au nombre d'observations de cette migration parmi la population de projets* P*.*

Une fois l'ensemble R des règles constitué, nous effectuons une analyse manuelle a posteriori pour évaluer la valeur de vérité de chaque règle candidate. Une règle est marquée comme correcte si nous estimons que les deux bibliothèques offrent des services similaires et peuvent être employées dans un même but. Dans le cas inverse, la règle est marquée comme fausse. Pour valider ou non une règle, nous utilisons les ressources disponibles sur Internet, telles que des moteurs de recherche et les pages web des bibliothèques, afin d'évaluer si les deux bibliothèques concernées peuvent être qualifiées de similaires.

3.3.4 Catégories de bibliothèques

Une catégorie regroupe des bibliothèques parmi lesquelles des migrations ont été effectuées, et indique ainsi un ensemble de bibliothèques similaires. Nous appelons taille de la catégorie, le nombre de bibliothèques qu'elle contient.

Pour constituer efficacement ces catégories, nous construisons des graphes de migrations. Il s'agit de graphe orientés dont les sommets représentent des bibliothèques, et les arcs orientés lient deux bibliothèques, respectivement la source et la cible d'une règle de migration. L'arc pointe vers la cible de la migration. Ces arcs sont étiquetés par le score de la règle de migration, correspondant à son nombre d'observations. À partir de ce graphe, les composantes connexes de la version non orientée du graphe sont extraites pour générer des catégories de bibliothèques. Chaque catégorie rassemble ainsi des bibliothèques connectées par des règles de migration (directes ou transitives).

3.4 Mise en application

Nous présentons dans cette section deux applications de notre approche sous la forme de deux cas d'étude. L'objectif est d'évaluer sa capacité à extraire des migrations de bibliothèque en pratique. Nous souhaitons obtenir un aperçu du nombre de migrations identifiées et de la précision de notre approche. Dans le premier cas d'étude, nous étudions les migrations de bibliothèque sur un corpus de projets *open source* collectés sur la plateforme GITHUB. Nous déployons ensuite notre approche sur le principal dépôt de logiciels proposé par Apache MAVEN. Nous suivons un procédé en au plus 5 étapes pour la description de chaque étude :

1. nous présentons la source de données analysée ;
2. nous décrivons comment l'ensemble des bibliothèques L est construit ;
3. nous expliquons comment sont extraites les bibliothèques utilisées par les projets clients (le dep_p) ;
4. nous adaptons si nécessaire l'algorithme de génération des migrations candidates (cas MAVEN uniquement) ;
5. nous précisons comment l'ensemble des projets P est construit.

Les résultats obtenus nous permettent de caractériser le phénomène de migration de bibliothèque et d'évaluer les performances de notre solution. De plus, ils mettent en lumière les catégories de bibliothèques les plus sujettes aux migrations.

3.4.1 Cas d'étude n°1 : Projets *open source* GITHUB

Source de données

Nous débutons par l'étude des migrations de bibliothèque sur un corpus de logiciels *open source* hébergés sur la plateforme GITHUB [1]. GITHUB est le principal hébergeur de logiciels *open source* et contenait plus de 10 millions de dépôts en janvier 2014. Il est possible d'accéder librement et facilement à des données sur l'activité des projets GITHUB par

1. http://www.github.com

le biais d'une API REST [2], du projet GITHUB Archive [3], voire de l'infrastructure GHTorrent proposée par Gousios et al. [Gousios, 2013]. Ainsi, notre source de données est ici l'historique d'un projet via son système de contrôle de versions Git. Le fait que Git soit un outil performant et rapide a influencé notre décision d'étudier des projets sur GITHUB.

Ensemble des bibliothèques

Nous avons décidé de réutiliser un ensemble de bibliothèques Java extraites dans une étude qui sera présentée dans le Chapitre 5. Cet ensemble a été construit en analysant un corpus de 6 330 projets configurés avec Apache MAVEN, puis en utilisant des fonctionnalités de MAVEN pour télécharger les fichiers JAR associés aux dépendances de ces projets. Un fichier JAR (Java ARchive) est utilisé pour distribuer un ensemble de classes Java. Nous justifions cette approche par le fait que l'ensemble des bibliothèques collectées couvre une majorité des bibliothèques couramment utilisées en Java.

Nous avons ainsi obtenu un ensemble de 8 795 fichiers JAR, qui ont été groupés selon leur similarité de nom pour écarter les informations sur les versions des bibliothèques. Par exemple, nous considérons que `junit-4.8.1.jar` et `junit-4.8.2.jar` font partie de la même bibliothèque `junit`. Après cette opération un ensemble de 3 326 bibliothèques a été obtenu. Nous avons ensuite procédé à l'extraction des symboles depuis les fichiers JAR associés. Nous décidons ici de retenir uniquement les noms des paquetages Java pour obtenir une représentation simplifiée des symboles et nommons cet ensemble *symboles agrégés* d'une bibliothèque. Nous devons ensuite contrôler la présence de conflits de symboles agrégés entre paires de bibliothèques. Un tel conflit a lieu lorsqu'un même symbole agrégé est produit par l'analyse de deux bibliothèques. Cette situation apparaît dans les deux cas particuliers suivants :

1. *Inclusion de bibliothèque* : une bibliothèque est distribuée sous plusieurs fichiers JAR et un de ces fichiers correspond à la version complète de la bibliothèque (l_{full}) tandis que les autres correspondent à des composants réduits (l_{comp}). De ce fait, tous les symboles définis par les composants sont inclus dans la version l_{full} de la bibliothèque. Nous affirmons que seule cette version complète doit être considérée pour la bibliothèque, et que l'ensemble des composants doit être écarté. En d'autres termes, lorsque tous les symboles agrégés inclus dans une bibliothèque l_{comp} sont inclus dans une autre bibliothèque l_{full} (c.à.d. $S_{l_{comp}} \subset S_{l_{full}}$), nous supprimons l_{comp} de l'ensemble des bibliothèques L à considérer. C'est le cas par exemple de la bibliothèque `batik` qui propose le fichier JAR *batik-all* contenant une distribution complète du code de la bibliothèque, mais qui propose également d'autres fichiers JAR représentant des parties réduites de la bibliothèque `batik` (*batik-dom*, *batik-css*, ...). Par conséquent, nous ne considérons qu'une seule bibliothèque `batik` avec tous ses symboles.

2. http://developer.github.com/v3/
3. http://www.githubarchive.org/

2. *Réutilisation de bibliothèque* : une bibliothèque l_a dépend d'une autre bibliothèque l_b et est amenée à la modifier. Elle est ainsi contrainte à être déployée avec du code provenant de l_b. De ce fait, le fichier JAR de l_a exporte des symboles qui sont initialement définis par l_b. Pour des raisons de simplicité, nous avons choisi de supprimer dans une bibliothèque les symboles qui sont redéfinis ou importés à partir d'autres bibliothèques. C'est le cas par exemple de la bibliothèque `junit` qui inclut des symboles originellement définis dans la bibliothèque `hamcrest`. Nous décidons donc de supprimer ces symboles de la bibliothèque `junit`.

Ces conflits de symboles doivent être résolus manuellement car il est difficile de déterminer de façon sûre et automatique à quelle bibliothèque appartient réellement un symbole donné. Une personne experte en Java s'est consacrée pendant 12 heures à la résolution des conflits de symboles. Le résultat final propose un ensemble L de 1 189 bibliothèques avec pour chacune son ensemble de symboles agrégés.

Extraction des bibliothèques utilisées

Ce cas d'étude nous a conduit à concevoir un outil nommé SCANLIB, qui permet d'extraire les bibliothèques utilisées par un projet au travers exclusivement d'une analyse statique de code source. L'objectif que nous nous sommes fixés est d'être générique à tout type de projet, indépendamment du système de configuration logicielle utilisé. La technique que nous présentons n'est pas spécifique à des projets GITHUB mais plutôt à n'importe quel code source versionné. Le postulat utilisé est qu'un logiciel utilise une bibliothèque l si au moins un des symboles de S_l apparaît dans un des fichiers de code source du logiciel.

Dans cet objectif, nous avons développé un moteur de recherche textuelle capable d'analyser les fichiers de code source d'un projet pour y détecter la présence de symboles agrégés définis par une bibliothèque. Ce moteur s'applique en séquence sur tous les fichiers en question (tous les fichiers avec une extension *.java* dans le cas du langage Java), et recherche tous les symboles connus dans le contenu de ces fichiers. Lorsqu'une correspondance est détectée, la bibliothèque associée est ajoutée à l'ensemble $dep_p(i)$. À la fin du processus, tous les $dep_p(i)$ de tous les projets ont été construits en utilisant exclusivement leur code source.

Le choix volontaire de rechercher textuellement un symbole agrégé dans le fichier au lieu de se restreindre aux sections `import` des fichiers Java vise à améliorer la précision finale. En effet, un symbole peut être utilisé en indiquant son nom qualifié complet à tout moment dans le code. De plus, le mécanisme de réflexion permet par exemple d'écrire le code suivant pour se connecter à une base de données `mysql` via JDBC :

```
Class.forName("com.mysql.jdbc.Driver").newInstance();
```

Notre analyse de ce code permet d'identifier la dépendance vers la bibliothèque `mysql`.

Remarque. Pour information, l'algorithme de recherche utilisé est celui d'Aho-Corasick qui opère en complexité linéaire selon la taille du texte [Aho et Corasick, 1975]. Il fut initialement utilisé dans l'utilitaire *grep* disponible sous Unix.

Ensemble de projets

Pour construire le corpus P de projets, nous avons utilisé les données ouvertes de GITHUB Archive [4] en plus du service Google BIGQUERY. Le projet GITHUB Archive a pour vocation d'enregistrer et d'archiver toutes les activités ayant lieu sur GITHUB. L'interaction avec Google BIGQUERY permet d'écrire et d'exécuter des requêtes sur des masses importantes d'informations dans une optique *Big Data*, et d'obtenir des temps de réponse de l'ordre de la seconde. À l'inverse, l'utilisation de l'API REST de GITHUB est beaucoup plus coûteuse.

Nous avons ainsi rassemblé un ensemble de 20 000 projets logiciels, ce qui nous paraît être une taille raisonnable pour observer des migrations de bibliothèques. La Figure 3.2 présente la requête exécutée pour récupérer cet ensemble, écrite pour se restreindre aux projets ayant au moins 10 *commits* et imposer un minimum d'activité, évitant ainsi les projets vides. De plus, nous écartons les projets qui sont des *forks* ou des dépôts privés. Cette requête a été exécutée en octobre 2013.

Remarque. Un *fork* est un dépôt cloné à partir d'un autre dépôt, et donc contenant le même historique que le dépôt source. L'intérêt d'éviter ces dépôts est de ne pas biaiser l'observation des migrations de bibliothèque en augmentant injustement le score d'une migration.

```
SELECT repository_url, count(repository_url) as pushes
FROM [githubarchive:github.timeline]
WHERE type="PushEvent"
AND repository_language="Java"
AND repository_fork="false"
AND repository_private="false"
GROUP BY repository_url
HAVING COUNT(*) >= 10
```

FIGURE 3.2 : Requête Google BIGQUERY pour former notre corpus de projets.

Nous avons par la suite écarté les projets comprenant moins de 100 lignes de code (LOC), moins de 30 jours d'activité et qui n'utilisent pas de bibliothèques (ces filtres ne sont pas applicables avec BIGQUERY). Nous supposons que ces projets ont une faible activité et qu'il est peu probable qu'ils aient migré de bibliothèques. Durant cette étape, nous avons également supprimé les projets qui n'existaient plus (dont l'URL ne répondait plus).

4. http://www.githubarchive.org/

Après ce filtrage, nous avons obtenu un ensemble de 15 168 projets. Ceci signifie que 24 % des projets retournés par notre requête Google BIGQUERY étaient vides, contenaient très peu d'activité, n'utilisaient pas de bibliothèque, ou étaient supprimés de GITHUB. Enfin, les ensembles de versions V_p associés aux projets $p \in P$ sont constitués des révisions disponibles sur leur dépôt Git respectif.

3.4.2 Cas d'étude n°2 : Dépôt central de MAVEN

Source de données

Le MAVEN *Central Repository*[5] (MCR) est une infrastructure de stockage et de mise à disposition de logiciels Java versionnés. Un tel dépôt est similaire à ceux contenant les paquetages pour les distributions Linux telles que Fedora[6]. Ce système centralisé bénéficie d'une API REST permettant de faciliter la recherche de données.

Les logiciels contenus sur ce dépôt sous configurés avec Apache MAVEN. MAVEN est un outil de configuration logicielle pour les projets Java. Chaque projet configuré sous MAVEN se doit de définir un fichier nommé POM (*Project Object Model*) au format XML qui contient trois éléments obligatoires : le *groupId*, qui définit l'organisation qui maintient le projet, l'*artifactId*, représentant le nom unique du projet au sein du *groupId*, et enfin un numéro de *version*. Ainsi, chaque artefact (selon la terminologie MAVEN) disponible sur le dépôt possède ces trois informations dans son fichier POM. Les logiciels disponibles sur le MCR sont potentiellement des bibliothèques ou *frameworks* qui cherchent à être distribués pour favoriser leur intégration dans d'autres logiciels. Il n'en demeure pas moins qu'ils dépendent eux-mêmes de bibliothèques pour pouvoir fonctionner.

Ensemble des bibliothèques

La particularité du fichier *pom.xml* est qu'il contient une section *dependencies* dédiée à la spécification des dépendances du projet. Chacune d'entre elles devra être exprimée au sein d'un élément *dependency*, en indiquant au minimum le *groupId* et l'*artifactId*. La Figure 3.3 montre un extrait du fichier POM pour la version 4.10 de la bibliothèque `junit`, où nous retrouvons la dépendance vers `hamcrest`. Par la suite, MAVEN est en mesure de télécharger, via des dépôts tels que le MCR, l'ensemble des dépendances nécessaires à la compilation et à l'exécution d'un projet. L'union des éléments (*groupId* :*artifactId*) des dépendances de chaque version des projets analysés représente l'ensemble des bibliothèques L. Notons qu'il n'est pas nécessaire d'extraire les symboles dans ce contexte. Nous avons ainsi pu construire un ensemble L de 10 224 bibliothèques.

5. http://search.maven.org
6. https://admin.fedoraproject.org/pkgdb/

```
<project>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>4.10</version>
   ...
   <dependencies>
      <dependency>
         <groupId>org.harmcrest</groupId>
         <artifactId>hamcrest-core</artifactId>
         <version>1.1</version>
      </dependency>
   </dependencies>
   ...
</project>
```

FIGURE 3.3 : Extrait du fichier POM de `junit` 4.10.

Extraction des bibliothèques utilisées

Comme décrit précédemment, les dépendances d'un projet sont identifiables en analysant des sections dédiées dans un fichier POM. L'ensemble $dep_p(i)$ d'un projet p correspond à des éléments *(groupId :artifactId)* selon les données spécifiées dans le fichier POM du projet p à la version i.

Génération des migrations candidates

Nous avons décidé d'apporter deux adaptations pour ce cas d'étude, compte tenu de la nature des données contenus dans le MCR.

Premièrement, ce dépôt contient des versions de projets sous forme de *releases*, et non de révisions. Bien qu'il soit difficile d'évaluer le temps consacré à produire une nouvelle version d'un projet, nous supposons qu'une migration de bibliothèque s'effectuera uniquement entre deux versions successives d'un projet. Par exemple, si pour un projet p nous avons $V_p = \{1.0, 2.0, 3.0\}$, alors les migrations de bibliothèque seront recherchées uniquement entre $[1.0, 2.0]$ et $[2.0, 3.0]$ (alors que par défaut, notre algorithme permet d'identifier des migrations entre $[1.0, 3.0]$).

De plus, il est possible sous MAVEN que les dépendances d'un projet partagent la même organisation que le projet, et donc le même *groupId*. Nous pensons que ces dépendances ne sont pas externes mais qu'il s'agit plutôt de bibliothèques internes ou proches du projet, dont l'utilisation peut être contrainte par des directives de cette organisation. De plus, il est probable qu'elles introduisent une quantité importante de bruit lors de l'extraction des migrations. Pour cette raison, nous excluons les dépendances portant le même *groupId* que celui du projet en cours d'analyse.

Il se peut également qu'un projet existe sous plusieurs identités alors qu'il devrait être perçu comme un seul et unique projet. Par exemple, un projet peut dépendre de `junit` ou de `junit-dep`, toutes deux faisant référence à la bibliothèque `junit`. La différence est que

`junit-dep` est délivrée sans aucune dépendance, mais que pour fonctionner il sera nécessaire d'en spécifier des supplémentaires au sein d'un projet (`hamcrest` dans ce cas précis). Ainsi, si nous observons qu'un projet migre de `junit` à `junit-dep`, il n'est pas pertinent de considérer cela comme une migration. De plus, nous observons que certains projets comme la solution de base de données `hsqldb` est disponible sous les *groupId* `org.hsqldb` et `hsqldb`. Si un projet décide de changer uniquement le *groupId* d'une dépendance, nous ne devrions pas considérer cela comme une migration. Ainsi, dans le cas de migration de `hsqldb` vers la bibliothèque `h2`, les deux règles de migration (`org.hsqldb:hsqldb`, `com.h2database:h2`) et (`hsqldb:hsqldb`, `com.h2database:h2`) devraient être fusionnées en une seule règle. Pour atténuer ces problèmes, nous proposons de considérer uniquement l'élément *artifactId* des dépendances et de mettre de côté les *groupId*. Puis, lorsqu'une migration (s,t) candidate est générée, nous calculons une distance de similarité syntaxique entre s et t. Les ensembles de *tokens* respectifs T_s et T_t sont donc produits en fonction des *artifactId* de s et de t. La segmentation est effectuée par les caractères non alpha-numériques, puis la migration est validée si et seulement si $T_s \cap T_t = \emptyset$.

Ensemble des projets

Nous considérons ici un projet logiciel comme un couple (*groupId*,*artifactId*) dont l'ensemble V_p correspond aux différentes versions sous lesquelles il est disponible. Nous avons réalisé l'extraction des données durant le mois d'avril 2012. Via l'API REST mise à notre disposition, nous avons pu collecter un total de 38 588 projets, et précisément 310 571 versions de ces projets. Il aura fallu 37 heures pour obtenir l'intégralité des fichiers POM associés. Afin d'améliorer la qualité du corpus, nous avons choisi d'écarter les versions ne respectant pas la convention de nommage basée sur l'expression régulière suivante :

```
^[0-9]{1,3}(\.[0-9]{1,3})*$
```

Ceci nous permet de ne conserver que les numéros de versions tels que `1.0` ou `1.0.1`, et de ne pas considérer des versions comme `1.1-M1`, `1.1.RC1` ou encore `1.1-alpha.2`. En écartant des versions bêta ou issues de branches de développement particulières, nous évitons d'introduire du bruit lors du classement chronologique des versions d'un projet. Ce tri a été établi selon le schéma utilisé par MAVEN [7]. L'intérêt est de se restreindre aux *releases* de la branche principale des projets. Nous avons également mis de côté les projets n'ayant qu'une seule version à disposition, soit 9 071 projets.

Après ce filtrage, nous avons obtenu un corpus de 20 987 projets comprenant un total de 173 405 versions. L'étape d'extraction des migrations de bibliothèque nous a permis d'observer que 10 224 dépendances distinctes étaient utilisées par les projets de notre corpus, ce qui représente un ensemble L neuf fois plus important que celui constitué lors du cas d'étude précédent.

7. http://docs.codehaus.org/display/MAVEN/Versioning

3.4.3 Résultats obtenus

Nous détaillons les résultats obtenus grâce aux expérimentations menées dans les deux cas d'études. Nous présentons ensuite des données sur l'utilisation des bibliothèques ainsi que des mesures quantitatives des migrations et des catégories que nous avons pu extraire.

Statistiques générales

Pour le premier cas d'étude, nos expérimentations nous ont permis d'obtenir un total de 28 052 migrations groupées en un ensemble de 17 113 règles de migration. Comme supposé précédemment, cet ensemble contient une large quantité de faux positifs que nous avons dû manuellement identifier. Ensuite, nous avons, durant 12 heures, vérifié chacune de ces règles pour déterminer si elles étaient vraies ou fausses. À la fin de ce processus de validation, nous avons obtenu un total de 1 198 migrations effectuées par 866 logiciels. Ces 1 198 migrations ont été groupées en 329 règles de migration. Les résultats révèlent que 164 bibliothèques y ont été impliquées, ce qui signifie que 16,1 % des bibliothèques de l'ensemble des 1 018 utilisées par les projets sont concernées par au moins une migration. Notre analyse montre également que 866 logiciels ont effectué une migration, soit 5,57 % du corpus initial de 15 168 projets. Une première conclusion est donc que la pratique de migration de bibliothèque peut être qualifiée d'occasionnelle mais existe bien néanmoins. Cette première étape révèle une précision brute de notre approche de 1,92 %. Ceci nous confirme qu'identifier automatiquement des migrations de bibliothèque est une opération qui est loin d'être triviale. Nous discuterons de cet aspect en Section 3.6.

Tableau 3.3 : Résumé des données obtenues à partir des projets GitHub et du MCR.

		GitHub	**Maven**
	Projets	15 168	20 987
	Bibliothèques	1 018	10 224
Brutes	Migrations	28 052	15 867
	Règles de migration	17 113	12 600
Corrigées	Migrations	1 198	880
	Règles de migration	**329**	**180**
	Précision	1,92 %	1,44 %
	Bibliothèques impliquées	164 (16,1 %)	137 (2 %)
	Catégories	32	39
	Projets migrants	866 (5,57 %)	811 (3,9 %)

Pour le deuxième cas d'étude, un total de 15 867 migrations regroupées en 12 600 règles a été généré. Après 6 heures d'analyse manuelle des règles, 254 règles de migration ont été retenues. Nous avons ensuite appliqué une opération similaire à celle appliquée en

Section 3.4.1, à savoir que nous avons fusionné certaines bibliothèques lorsqu'elles correspondaient à une même et unique entité (tel que `junit` et `junit-dep` par exemple). Après cela nous disposions de 180 règles englobant un total de 880 migrations. Celles-ci impliquent 137 bibliothèques réparties dans 39 catégories. De plus, nous avons observé que 839 projets de notre corpus ont effectué une migration, soit une part de 3,9 % de la population de départ. Un dernier constat frappant sur le cas d'étude n°2 est que, malgré un ensemble initial de 10 224 bibliothèques, seulement 137 bibliothèques (1,3 %) sont impliquées dans des migrations. Nous relevons également une précision de 1,44 % par rapport aux règles de migration. Un résumé de ces résultats obtenus est disponible dans le Tableau 3.3.

Utilisation des bibliothèques

À titre d'information, nous proposons dans le Tableau 3.4 les 10 bibliothèques les plus utilisées par les projets de nos deux cas d'étude. Nous observons que la bibliothèque `junit` est très populaire, et ce pour les deux sources de données analysées. Le classement pour MAVEN diffère légèrement de GITHUB où l'on note l'absence de la bibliothèque `android` et la présence de bibliothèques spécifiques au déploiement de logiciels gérés avec MAVEN.

Tableau 3.4 : Classement des 10 bibliothèques utilisées par les corpus des deux cas d'étude. Les bibliothèques notées en gras sont présentes dans les deux cas.

	GITHUB			MAVEN	
Bibliothèque	Clients	Description	Bibliothèque	Clients	Description
junit	7176	Tests Unitaires	**junit**	7360	Tests Unitaires
android	3672	Mobile	**slf4j**	2795	Logging
xmlParserAPIs	3088	XML	servlet	1904	Servlet Java
commons-lang	2558	Java Util.	**log4j**	1615	Logging
slf4j	2248	Logging	commons-logging	1317	Logging
spring	2064	J2EE	**commons-lang**	1189	Java Util.
guava	1998	Java Util.	commons-io	1039	Entrées/Sorties
httpclient	1940	Http	maven-plugin-api	1029	MAVEN
httpcore	1905	Http	maven-project	752	MAVEN
log4j	1843	Logging	plexus-utils	684	MAVEN

Détails des règles de migration

Nous proposons dans le Tableau 3.5 classement des 10 règles de migration les plus courantes pour le cas d'étude n°1. Il est intéressant d'observer que la plupart des migrations impliquent des bibliothèques qui sont parmi les plus populaires comme vu précédemment. Il apparaît clairement que les migrations les plus fréquentes concernent les bibliothèques de journaux de bord et ciblent très particulièrement `slf4j`. Par ailleurs,

Tableau 3.5 : Top 10 des règles de migration pour les projets GITHUB.

Source	Cible	Score	Description
log4j	slf4j	95	Logging
commons-logging	slf4j	61	Logging
junit	testng	45	Tests Unitaires
commons-httpclient	httpclient	33	Ressource Http
commons-httpclient	httpcore	32	Ressources Http
org.json	gson	28	Traitement JSON
testng	junit	27	Tests Unitaires
org.json	jackson	26	Traitement JSON
hamcrest	fest	25	Écriture de Tests
easymock	mockito	23	Écriture de Tests
gson	jackson	22	Traitement JSON

Tableau 3.6 : Top 10 des règles de migration pour l'étude du dépôt MAVEN.

Source	Cible	Score	Description
commons-logging	slf4j	208	Logging
log4j	slf4j	60	Logging
slf4j	logback	57	Logging
google-collections	guava	31	Utilitaires Java
jsch	jclouds-sshj	25	SSH
log4j	logback	25	Logging
junit	testng	20	Tests Unitaires
testng	junit	15	Tests Unitaires
commons-logging	log4j	14	Logging
jmock	mockito	12	Écriture de Tests

nous constatons qu'une alternative à la très populaire bibliothèque `junit` est `testng`, dont l'adoption à partir de `junit` a été observée 45 fois, ce qui représente la 3^e migration la plus observée. Les bibliothèques permettant de manipuler des données au format JSON sont également présentes 3 fois dans ce classement.

Pour le cas d'étude n°2, les règles sont affichées dans le Tableau 3.6. Le constat marquant est une nouvelle fois la part importante des migrations pour les bibliothèques de journaux de bord, puisque les trois règles les plus observées font partie de cette catégorie. Il est intéressant de retrouver certaines tendances constatées plus tôt dans le Tableau 3.5. Ainsi, les migrations entre `junit` et `testng` sont à nouveau présentes, tandis que les nombreux abandons de `commons-logging` et de `log4j` au profit de `slf4j` sont encore observés. En revanche, il n'apparaît pas ici de bibliothèques de la catégorie JSON.

L'intérêt de proposer des tendances de migration de bibliothèques est ici parfaitement

Tableau 3.7 : Classement des 10 catégories en termes de migrations pour les deux cas d'étude.

GITHUB			MAVEN		
Description	Migrations	Taille	Description	Migrations	Taille
Logging	238	4	Logging	387	4
JSON	210	13	XML	143	20
Collections	128	14	Écriture de Tests	54	12
Écriture de Tests	106	8	Java Util.	39	5
Http	100	5	Base de données	36	6
XML	90	17	Tests Unitaires	35	2
Base de données	83	15	SSH	25	2
Tests Unitaires	72	2	Bytecode	20	6
Réflexion	43	8	XML Beans	19	4
GUI	24	11	Java Server Faces	13	4

illustré. En effet, nous constatons que les bibliothèques `commons-logging` et `log4j` sont très populaires, comme le laisse apparaître le Tableau 3.4. Ainsi, il est difficile d'imaginer que ces bibliothèques ont été massivement abandonnées et qu'elles sont à exclure des recommandations. À l'inverse, la bibliothèque `logback`, absente du classement par popularité des bibliothèques, intervient dans 2 des 6 règles de migration les plus observées sur le MCR, comme l'indique le Tableau 3.6.

Catégories de bibliothèques

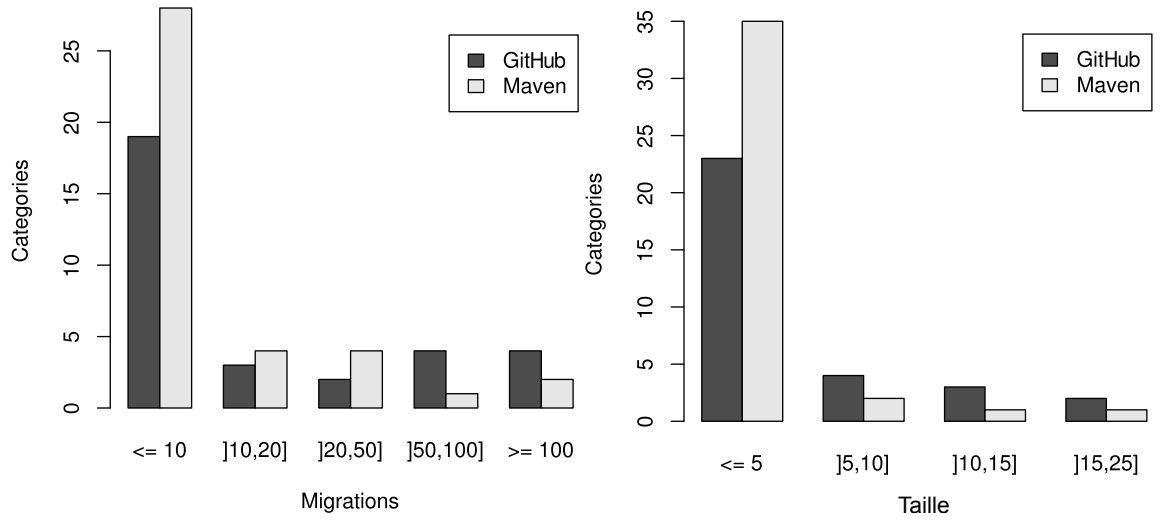
Nous avons pu obtenir respectivement 32 et 39 catégories de bibliothèques à partir des projets GITHUB et MAVEN. Le classement des 10 catégories en termes de migrations pour les deux cas d'étude est disponible dans le Tableau 3.7. Il en ressort que la catégorie *Logging* apparaît à chaque fois en première position. Pour les projets MAVEN en particulier, cette catégorie regroupe 387 des 883 des migrations, soit une part de 43 %. Il est également surprenant d'observer seulement 4 migrations pour les bibliothèques JSON, alors que cette catégorie était fortement représentée dans notre cas d'étude sur GITHUB.

Les distributions par taille et nombre de migrations (appelée aussi *activité*) des catégories sont détaillées dans la Figure 3.4. Nous pensons que ces deux mesures permettent d'évaluer la qualité des catégories produites. En effet, une catégorie avec une faible activité de migration aura peu de valeur pour un développeur, car il sera difficile pour lui de prendre une décision grâce aux tendances proposées. La taille des catégories reflète quant à elle la diversité des bibliothèques disponibles au sein d'une catégorie.

Pour les projets GITHUB et MAVEN, nous observons que respectivement 19 et 28 catégories contiennent moins de 10 migrations, ce qui représente un faible taux d'activité. À l'inverse, les 13 et 10 catégories restantes pour ces cas d'étude ont plus de 10 migrations.

De plus, il apparaît que 9 catégories pour les projets GITHUB sont formées par plus de 5 bibliothèques, contrairement à 4 seulement pour les projets MAVEN. Ainsi, 35 sur 39 des catégories formées à partir du cas d'étude sur MAVEN fournissent moins de 5 bibliothèques.

Nous avons identifié 25 catégories dans le cas d'étude MAVEN dont les bibliothèques n'apparaissent dans aucune des catégories générées par le cas d'étude GITHUB. À l'inverse, 22 catégories de bibliothèques du deuxième cas d'étude ne font intervenir aucune bibliothèque recensée dans le premier cas. Il existe en revanche seulement 2 catégories similaires identifiées dans les résultats des deux expériences. Enfin, nous avons extrait 11 paires de catégories ayant au moins une bibliothèque en commun. Parmi elles, il s'avère qu'à 8 reprises la taille des catégories fournies par GITHUB est supérieure, représentant un total de 96 bibliothèques contre 35 pour MAVEN sur ces 11 paires. Ces observations nous laissent penser que la qualité des résultats obtenus par le cas d'étude sur GITHUB est plus intéressante par rapport à celle du cas d'étude sur MAVEN.



(a) Selon le nombre de migrations.

(b) Selon la taille.

FIGURE 3.4 : Distribution du nombre de catégories selon le nombre de migrations et la taille.

3.5 Exploitation des résultats

Jusqu'à présent, nous avons démontré que notre approche est réaliste et permet en pratique de générer des règles de migration et des catégories de bibliothèques. La seconde partie de notre travail consiste à exploiter les données obtenues pour démontrer qu'un développeur peut tirer profit de ces résultats. L'objectif de cette section est de proposer des indicateurs pour un développeur via des visualisations des tendances de migrations.

Dans ce contexte, nous supposons que les questions qu'un développeur se pose et auxquelles nous cherchons à répondre sont : existe-t-il une bibliothèque massivement adoptée ? Existe-t-il des bibliothèques émergentes pour lesquelles on observe un taux d'adoption significatif ? Peut-on identifier des bibliothèques qui sont fréquemment abandonnées ?

Remarque. Les concepts d'adoption et d'abandon sont ici liés exclusivement à un contexte de migration, et non d'utilisation générale.

Pour cela, nous présentons dans un premier temps les graphes de migration de bibliothèques. Nous considérons ensuite l'évolution temporelle des tendances de migrations et affirmons qu'elle peut être utilisée en complément des graphes de migration. Pour chacune des visualisations proposées, nous définissons des motifs visuels servant d'indicateurs pour caractériser les bibliothèques.

3.5.1 Graphe de migrations

Motivations. Nous proposons une première aide au développeur : le graphe de migrations de bibliothèques. Celui-ci a pour vocation d'exhiber les tendances de migrations au sein d'une catégorie de bibliothèques. Nous définissons pour cela une mesure nommée *degré* de migration. Il s'agit de la différence entre le degré entrant et le degré sortant d'une bibliothèque dans une catégorie. Un degré positif indique que la bibliothèque a subi plus d'adoptions que d'abandons, alors qu'un degré négatif évoque une tendance inverse. Pour aider le développeur à comprendre les informations proposées dans ce graphe, nous proposons les caractérisations suivantes :

1. *Best challenger* est la bibliothèque ayant le plus haut degré de migration de la catégorie.
2. *Challenger* est une bibliothèque montrant un degré de migration positif.
3. *Breaking bad* est une bibliothèque montrant un degré de migration négatif.
4. *Worst breaking bad* est la bibliothèque avec le plus faible degré de migration.

Les bibliothèques *challenger* devraient être recommandées comme cibles de migrations, avec une attention particulière sur la bibliothèque *best challenger*. À l'opposé, nous déconseillons de migrer vers les bibliothèques *breaking bad* et surtout de ne pas retenir celle étant *worst breaking bad*. Nous offrons ainsi une unique visualisation mettant en valeur toutes les migrations ayant eu lieu dans une catégorie et permettant de classer les bibliothèques selon les quatre caractérisations proposées.

Méthodologie. Nous avons présenté en Section 3.3.4 les concepts de base des graphes de migrations. Les sommets du graphe sont des bibliothèques, le graphe est orienté et les arcs représentent une migration d'une bibliothèque vers une autre. De plus, les arcs sont étiquetés par le nombre de fois qu'une migration a été observée, c'est-à-dire son score.

L'épaisseur des arcs met également en valeur cette donnée. Les sommets avec un degré positif sont représentés par un cercle, et ceux négatifs par un carré. Un triangle indique un degré égal à zéro.

Les sommets sont étiquetés avec le nom de la bibliothèque ainsi que leur degré de migration. De plus, ils sont colorés pour représenter l'intensité de cette valeur. Un rouge marqué fait référence à la bibliothèque *worst breaking bad* d'une catégorie, tandis qu'un vert marqué pointera vers le *best challenger*. La taille des sommets reflète également le degré de migration.

Pour illustrer nos propos, considérons une catégorie *Sample* contenant 3 bibliothèques, *X, Y* et *Z*. Le graphe de migration de bibliothèque associé est exposé dans la Figure 3.5. Ici, la bibliothèque *X* apparaît comme l'unique *challenger* tandis que *Y* fait office de *breaking bad* et que *Z* est neutre avec un degré de 0.

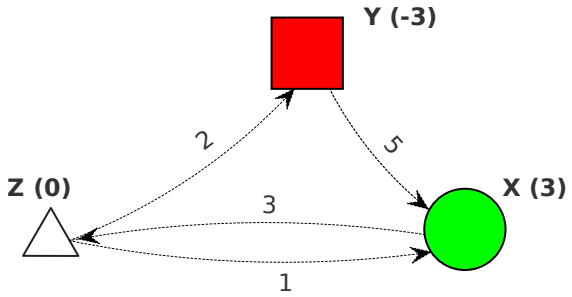


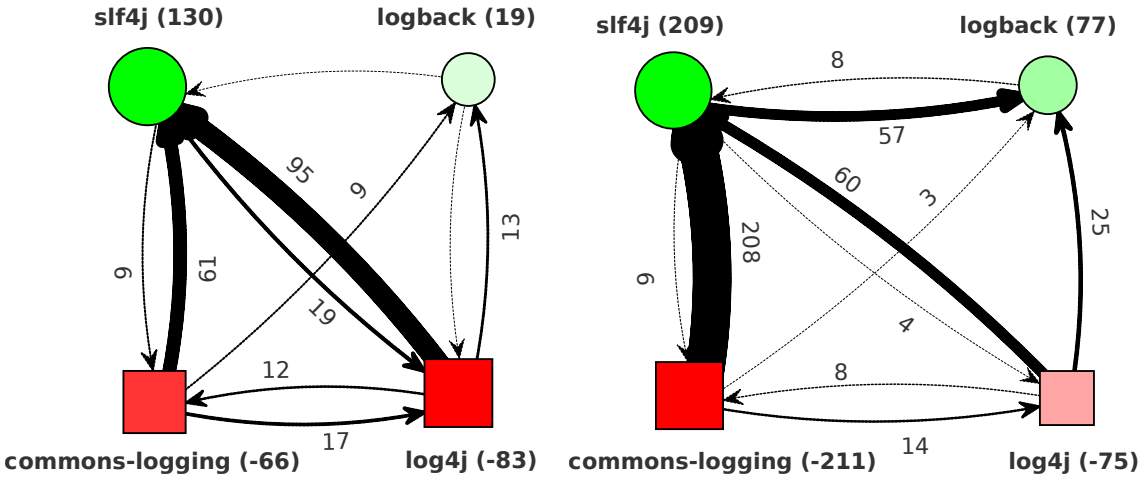
FIGURE 3.5 : Graphe de migrations pour la catégorie *Sample*.

Exemples concrets. Nous illustrons ces graphes de migrations grâce aux résultats obtenus durant nos expérimentations. Notons que par souci de clarté, nous n'affichons pas les scores des migrations strictement inférieurs à 3.

Pour commencer, nous proposons dans la Figure 3.6 les deux graphes de migrations associés à la catégorie *Logging* pour chaque cas d'étude. Nous retrouvons à chaque fois le même ensemble de 4 bibliothèques. Il est intéressant de constater que les tendances de migrations sont similaires pour les deux graphes. En effet, nous retrouvons `slf4j` comme étant *best challenger* pour la catégorie. De plus, `logback` s'avère être un *challenger* solide avec un degré de migration de 19 et 77 respectivement pour le cas d'étude GITHUB et MAVEN. Pourtant, cette bibliothèque ne faisait pas partie des plus populaires de cette catégorie (voir Section 3.4.3) et cela illustre à nouveau l'intérêt d'étudier les tendances de migrations. La description de `logback` indique que cette bibliothèque est le « *successeur de* `log4j` »[8]. Nous qualifions ce type de bibliothèque d'émergente. À l'inverse, les deux bibliothèques `commons-logging` et `log4j` sont clairement des *breaking bad* de par leur

8. http://logback.qos.ch/

fort degré de migration négatif. Nous ne recommanderions donc pas d'utiliser ces bibliothèques mais plutôt de considérer `logback` avec `slf4j`.



(a) Cas d'étude GITHUB. (b) Cas d'étude MAVEN.

FIGURE 3.6 : Graphes de migrations pour la catégorie *Logging*.

Le second exemple présente les bibliothèques de la catégorie JSON utilisées pour traiter des ressources au format JSON. Ces données ont été générées à partir du cas d'étude sur les projets GITHUB. Nous constatons à partir de la Figure 3.7 que les bibliothèques `jackson` et `org.json` sont respectivement le *best challenger* et le *worst breaking bad* de cette catégorie. En effet, leur degré de migration respectif est de 53 et de -50 ce qui constitue des valeurs marquantes. Nous remarquons ensuite que `gson`, `fastjson` et `flexjson` sont les trois principaux *challengers*, mais aussi que leur écart avec `jackson` est assez prononcé. En conclusion, nous suggérons 4 bibliothèques pour cette catégorie de bibliothèques, où parmi elles `jackson` semble se détacher.

Ainsi, cette visualisation se veut simple mais efficace pour permettre d'interpréter les tendances de migrations de bibliothèque.

3.5.2 Graphe d'évolution de migrations

Motivations. Les graphes de migration de bibliothèques peuvent potentiellement présenter des informations biaisées sur l'état actuel d'une bibliothèque, en termes de migrations. Ce biais est dû à la période de temps au sein de laquelle l'analyse des migrations a été effectuée. Dans notre cas, nous n'avons pas contraint la date initiale de notre analyse. Pour comprendre l'impact du temps, supposons une bibliothèque X possédant un degré de migration de 5. Si nous observons qu'elle fut la cible de 15 migrations jusqu'en 2011, et la source de 10 migrations depuis, la recommandation de X est remise en question. Mais la

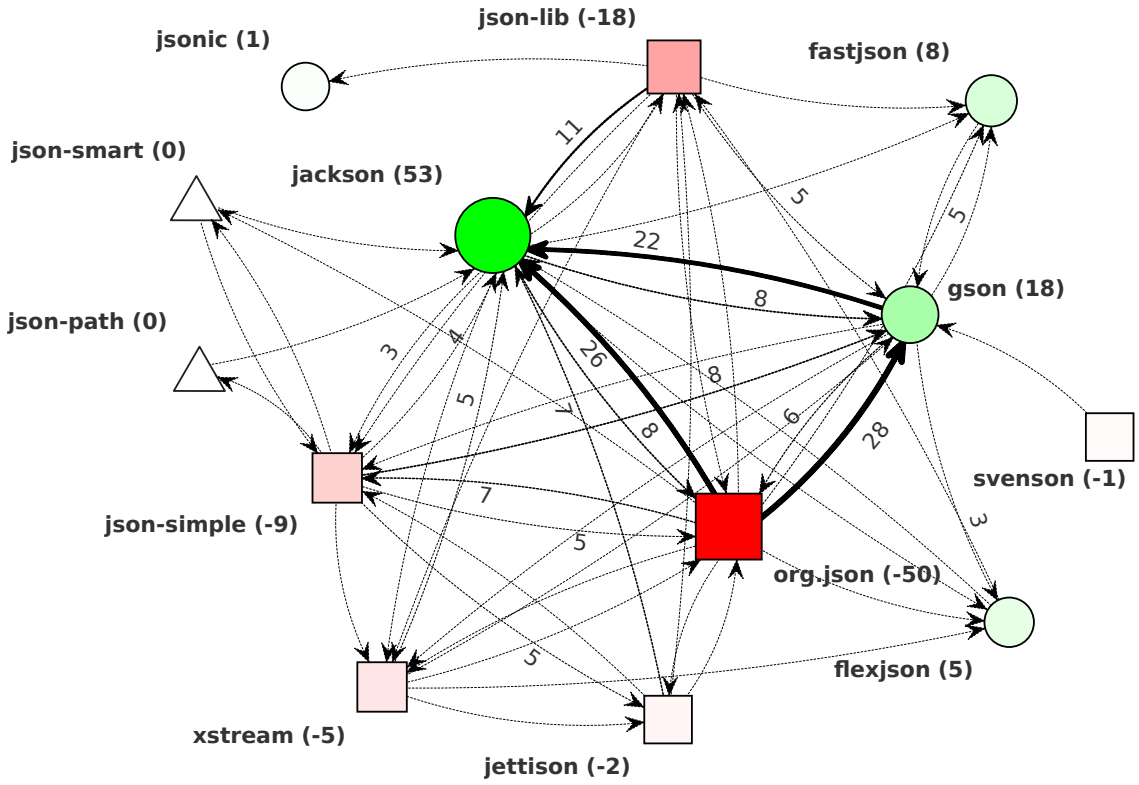


FIGURE 3.7 : Graphe de migrations pour la catégorie JSON.

situation opposée peut tout aussi bien se produire. Considérons une bibliothèque Y avec un degré de migration de -5. Si Y fut la source de 10 migrations avant 2011, mais que par la suite elle fut adoptée à 5 reprises, ses tendances de migrations récentes sont à son avantage. Ce revirement de situation peut par exemple s'expliquer par l'ajout de fonctionnalités majeures dans des versions récentes de la bibliothèque. En conséquence, nous pensons qu'il y a un intérêt à considérer les dates des migrations afin d'affiner les indicateurs de recommandations.

Les deux questions auxquelles nous nous intéressons ici sont :

— Peut-on identifier des bibliothèques *challenger* qui sont devenues *breaking bad* sur la période de temps récente ? Nous les qualifierons de *collapsing*.

— Peut-on identifier des bibliothèques *breaking bad* qui sont devenues *challenger* sur la période de temps récente ? Nous les qualifierons de *hopeful*.

Méthodologie. Les données du cas d'étude sur les projets GITHUB ont été exclusivement utilisées ici. En effet, nous n'avons pu accéder de façon précise aux données temporelles

des *releases* présentes sur le dépôt central de MAVEN. La date d'une migration est définie par la position intermédiaire entre les dates des versions qui la délimitent.

Pour construire des graphes d'évolution de migrations, la période de temps entre 2010 et octobre 2013 est tout d'abord divisée en 4 années. Nous écartons ainsi seulement 6 % des migrations de cette étude. Pour chaque année, nous générons le degré de migration des bibliothèques pour une catégorie donnée en incluant uniquement les migrations sur l'année en cours. Un triangle noir orienté vers le haut indique un degré positif alors qu'un triangle blanc orienté vers le bas évoque un degré négatif. Rien n'est affiché lorsque le degré équivaut à zéro.

Une illustration d'un tel graphe pour notre catégorie d'exemple *Sample* est montrée en Figure 3.8. Grâce à ce graphe, nous observons que la bibliothèque *X* est *collapsing* puisqu'elle présente un degré de -3 pour l'année 2013, malgré son statut de *challenger* sur la globalité de la période. À l'inverse, *Y* et *Z* sont des bibliothèques *hopeful* de par leur degré respectif de 2 et 1 sur cette période.

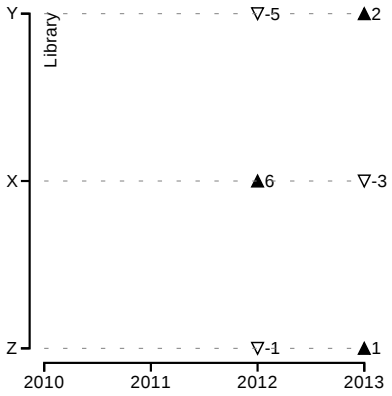
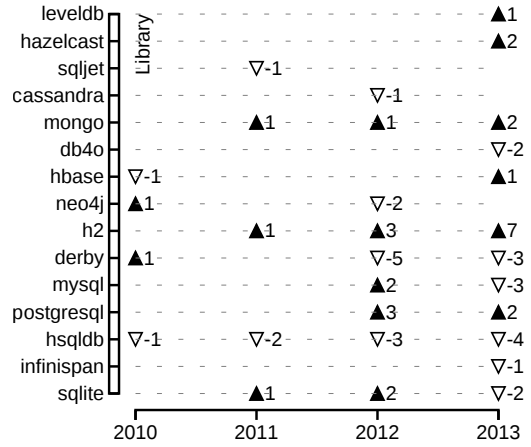


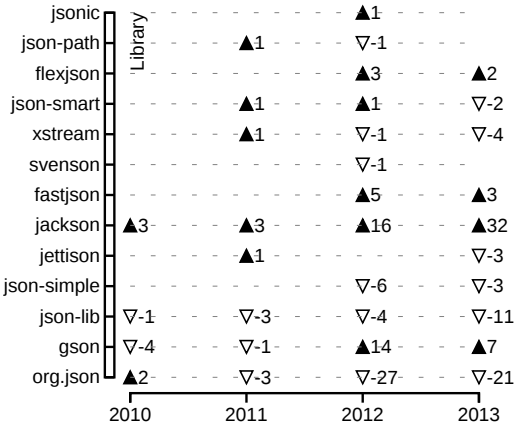
FIGURE 3.8 : Graphe d'évolution de migrations pour la catégorie *Sample*.

Exemples concrets. Nous avons pu extraire 4 catégories permettant d'illustrer nos propos. Pour commencer , nous observons en Figure 3.9(a) que `sqlite` est une bibliothèque *collapsing*, tout comme `json-smart` et `jsoup` le sont pour leur catégorie respective dans les Figures 3.9(b) et 3.9(c). En effet, `sqlite` est une bibliothèque *challenger* dans sa catégorie, mais sa récente tendance en 2013 n'est pas à son avantage puisque son degré de migration est de -2 en 2013. De façon similaire, `json-smart` possède un degré de -2 en 2013 alors que son degré de migration total est de zéro. Pour terminer, `jsoup` qui possède un degré de 4 sur l'année 2012 dégage un degré de -1 sur 2013. La chute est donc légère mais malgré tout présente. À l'inverse, la Figure 3.9(d) permet d'observer que la bibliothèque `reflections` possède un degré de 3 sur 2013 tandis que son degré sur la période totale est de -5. C'est donc une bibliothèque *hopeful* pour cette catégorie, puisque sa tendance est positive.

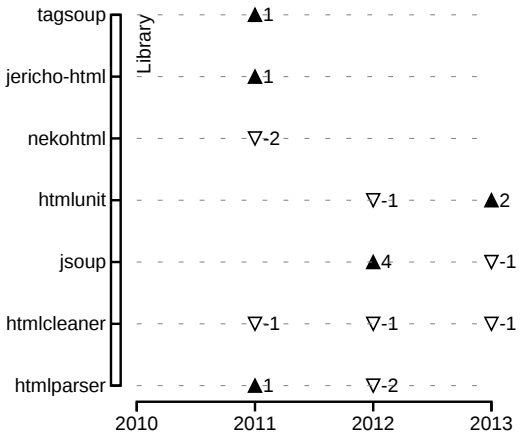
Nous pensons ainsi que les bibliothèques *hopeful* et *collapsing* fournissent des indicateurs pertinents pour les développeurs ; c'est pourquoi nous avons exploité l'aspect temporel des tendances de migrations.



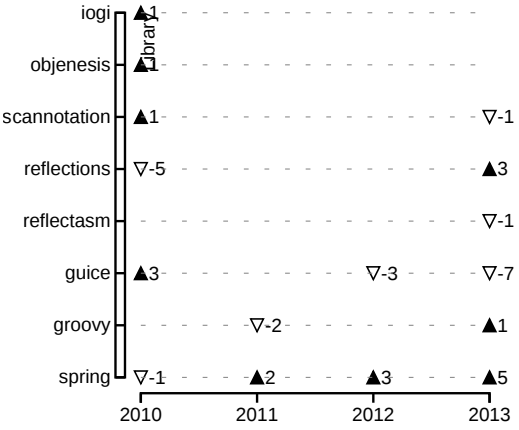
(a) Catégorie *Base de données.*



(b) Catégorie *JSON.*



(c) Catégorie *Html.*



(d) Catégorie *Reflexion.*

FIGURE 3.9 : Graphe d'évolution de migrations pour identifier les bibliothèques *collapsing* and *hopeful.*

3.5.3 Conclusion

Nous avons proposé deux exploitations possibles des tendances de migrations par le biais de visualisations. Notre objectif premier ici est de faciliter l'interprétation des données observées. D'autre part, si nous sommes en mesure de fournir des informations ayant un intérêt pour les développeurs, cela conforte et appuie nos motivations pour étudier ces

tendances de migrations. Notons que nous n'avons volontairement pas inclus les graphes de toutes les catégories, afin de ne pas surcharger le manuscrit.

3.6 Aides à la réplication

Puisque étudier les migrations permet de générer des indicateurs pertinents, il est naturel de vouloir rafraîchir régulièrement les données et donc de répliquer nos expérimentations. Pour cela, nous proposons des recommandations sur la méthodologie à adopter pour optimiser la qualité des règles de migration obtenues. Pour commencer, nous discutons des différences entre les sources de données GITHUB (cas d'étude n°1) et MAVEN (cas d'étude n°2). Puis, nous tirons parti du cas d'étude n°1 pour déterminer si certains profils de projets sont plus enclins que d'autres à avoir effectué une migration de bibliothèque. Dans un dernier temps, nous tentons d'automatiser la détermination des valeurs de vérité des règles de migration.

3.6.1 Source de données

À partir des résultats obtenus, nous pensons qu'analyser des dépôts *open source* GITHUB fournit des données plus riches que celles produites en analysant le MCR. L'avantage de GITHUB est qu'il y réside une très large population de projets ayant eu une activité récente. Les tendances de migrations sont clairement plus pertinentes si elles ont été observées récemment. De plus, la population du MCR augmente de façon beaucoup moins rapide que celle de GITHUB, et il est plus délicat de filtrer les projets sur un tel dépôt puisque très peu d'informations y sont attachées.

L'inconvénient du dépôt MAVEN est que l'ajout de nouvelles *releases* suit bien souvent une évolution discontinue et aléatoire. À l'inverse, les projets GITHUB sont plus diversifiés car il ne s'agit pas uniquement de projets gérés sous MAVEN, et il n'y est pas nécessaire d'en considérer les *releases*. De plus, nous avons observé dans le cas d'étude n°2 que pour une part non négligeable des règles de migration il fut parfois impossible de trouver des informations sur les bibliothèques concernées. Ne parvenant pas à déterminer le domaine d'utilisation d'une bibliothèque donnée, nous n'avons pas validé les règles où elle était présente. Il serait judicieux de pré-traiter les projets analysés et les dépendances qu'ils utilisent pour améliorer la qualité des règles produites. Par exemple, nous pourrions considérer uniquement les dépendances situées sur le même dépôt que les projets étudiés. Il est en effet possible sous MAVEN de télécharger des dépendances situées sur des dépôts tiers.

De plus, l'analyse des fichiers POM présente certaines limitations. La principale est une perte d'informations due aux dépendances transitives. Pour mieux comprendre ce phénomène, considérons un projet Java fictif *Foo* géré avec MAVEN, où nous spécifions dans le fichier de configuration la bibliothèque `android`. La Figure 3.10(a) illustre cette situation. Or, la bibliothèque `android` a elle aussi besoin de bibliothèques pour fonctionner, mais

celles-ci ne sont pas directement incluses dans cette bibliothèque. Au contraire, sa configuration fait apparaître un ensemble de 6 bibliothèques dans la section *dependencies*. L'une d'entre elles, `httpclient`, dépend également de 3 bibliothèques dont `commons-logging` qui fait partie des dépendances d'`android`. Finalement, le projet *Foo* dépend indirectement de 8 bibliothèques, qui sont donc utilisables via leur interface de programmation. L'aperçu de la Figure 3.10(b) soutient cette affirmation.

```
<dependencies>
    <dependency>
        <groupId>com.google.android</groupId>
        <artifactId>android</artifactId>
        <version>4.1.1.4</version>
    </dependency>
</dependencies>
```

(a) Extrait du fichier pom.xml du projet *Foo*.

(b) Bibliothèques réelles utilisées par le projet *Foo* (vue sous l'IDE ECLIPSE).

FIGURE 3.10 : Limites de l'analyse des fichiers POM de MAVEN.

Par conséquent, une analyse du fichier de configuration indique qu'une seule bibliothèque est utilisée, alors qu'en pratique il est permis au développeur de *Foo* d'utiliser jusqu'à 9 bibliothèques différentes au sein de son projet. C'est pour cette raison que MAVEN induit une perte d'informations dans notre contexte. Il est donc tout à fait envisageable qu'un projet migre son code source d'une bibliothèque vers une autre sans que cela ne soit retranscrit au sein du fichier POM. Pour terminer, il est à noter que chaque projet peut hériter d'un projet *parent*, et donc de ses dépendances. Si ces données ne sont pas accessibles, nous souffrirons à nouveau d'un problème de manque d'information.

Ainsi, nous conseillons plutôt de déployer notre approche sur des plateformes comme GITHUB à l'aide de SCANLIB. Nous voyons deux principaux avantages à utiliser cette dernière technique. D'une part, elle se veut générique à tout type de projet et indépendamment de leur système de configuration logicielle et même de leur langage de programmation. D'autre part, la précision est améliorée puisque nous observons quelles bibliothèques sont utilisées en pratique par les logiciels. Elle permet aussi de restreindre la taille du corpus de bibliothèques et de ne pas diminuer le nombre de règles obtenues, bien au contraire. À l'inverse, notre technique nécessite de devoir construire et maintenir une base de données de bibliothèques tierces et de symboles associés.

Tableau 3.8 : Description des 30 déciles de la population des projets GITHUB calculés selon 3 métriques distinctes.

	Déciles									
Métrique	10 %	20 %	30 %	40 %	50 %	60 %	70 %	80 %	90 %	100 %
Versions	19	27	37	49	65	87	123	188	360	13 130
Java KLOC	0,7	1,2	1,8	2,6	3,8	5,6	8,6	15,2	35,9	31 611
Développeurs	1	1	1	2	2	3	3	5	8	855

3.6.2 Profils des projets migrants

À partir des 15 168 projets collectés dans le cas d'étude n°1, nous avons décidé d'effectuer des mesures statistiques pour mieux comprendre ce corpus. Pour cela, le nombre de versions, de kilo-lignes de code Java ainsi que de développeurs ont été calculés pour chaque projet. Le Tableau 3.8 montre les déciles associés à ces trois mesures. Le premier constat est que notre corpus suit une loi de puissance au regard de ces mesures. En effet, nous observons que 80 % des projets sont composés de 1 à 5 développeurs, et que les 20 % restants impliquent des équipes d'au moins 5 à 855 développeurs. De façon similaire, 80 % des projets contiennent moins de 188 versions, tandis que les 20 % restants possèdent entre 188 et 13 130 versions.

Notre objectif est de déterminer quels profils de projets ont de fortes chances de contenir des migrations de bibliothèque. Nous avons décidé de former deux groupes, appelés respectivement `Fort` et `Faible`. La partition `Fort` est composée de l'union des projets les 20 % plus importants en nombre de versions, en nombre de développeurs et en kilo lignes de code Java. 5 959 projets font partie de cette partition, soit 39,3 % du corpus initial. Ceci signifie que le groupe `Faible` contient les 9 209 projets restants, soit 60,7 % du corpus.

Au final, nous avons constaté que 593 des projets migrants appartiennent au groupe `Fort` (68,5 %). Cela signifie que 9,95 % de cette catégorie de projets ont accompli une migration de bibliothèque. À l'inverse, 273 logiciels au sein de la catégorie `Faible` ont migré, ce qui représente 2,96 % de cette population. Des analyses plus approfondies n'ont pas permis d'identifier des profils de projets plus fins ayant plus de chances de contenir une migration. En revanche, deux enseignements sont à tirer de ces chiffres. Tout d'abord, tout type de projet peut effectuer une migration indépendamment de sa taille. Deuxièmement, la probabilité d'extraire des règles de migration sera plus élevée au sein de projets de dimension moyenne à grande. Ce constat n'est pas surprenant, puisque les projets ayant une certaine durée de vie ont investi plus de temps dans la gestion de leurs bibliothèques tierces par rapport à des projets peu voire très peu matures.

Remarque. Ces données pourraient être utilisées dans un travail futur pour déterminer s'il

existe des périodes dans le cycle de vie des projets où les migrations de bibliothèque sont plus fréquentes que dans d'autres.

3.6.3 Identification automatique des règles de migration

Les résultats obtenus durant nos expériences montrent une précision brute de notre approche de 1,92 % pour le cas d'étude n°1 et de 1,44 % pour le cas n°2, ce qui est très faible. Notre objectif est d'éviter au maximum le travail manuel de vérification des règles de migration, et pour cela nous souhaitons améliorer nos scores de précision. Dans l'idéal, la finalité est d'atteindre un procédé entièrement automatisé permettant d'extraire des vraies migrations de bibliothèque. Nous avons pour cela étudié différentes caractéristiques des migrations de bibliothèque dans le but de distinguer les fausses migrations candidates des vraies. Nous avons retenu trois critères pouvant être mesurés de façon automatique : la durée de la migration, la similarité des fichiers et le contenu des messages de *commits*. Nous détaillons dans un premier temps les intuitions derrière ces trois métriques et évaluons ensuite leur impact sur la qualité des résultats obtenus.

Métriques proposées

Métrique de longueur de migration (LEN). Nous nous intéressons au nombre de versions que représente une migration candidate à partir des deux versions qui la délimitent. Nous pensons que les vraies migrations ont lieu dans un court intervalle de versions, afin de ne pas laisser cohabiter les deux bibliothèques pour une durée trop importante. Ainsi, si une migration a lieu entre deux *commits* successifs, la longueur sera de 1. Nous nommons ce paramètre LEN.

Métrique de similarité des fichiers (SIM). L'idée de la métrique de similarité de fichiers est que lors d'une migration de bibliothèque (s, t), l'ensemble des fichiers de code source utilisant anciennement des symboles de S_s a été mis à jour avec des symboles de S_t. Notre hypothèse est que l'ensemble des fichiers contenant anciennement les symboles de S_s et l'ensemble de ceux contenant nouvellement les symboles de S_t, se recouvrent très fortement. Cette métrique, nommée SIM, est donc un coefficient mesurant la divergence entre ces deux ensembles.

Soit $Files_p(i)$ l'ensemble des fichiers de code source contenus dans le projet p à la version i. Pour une bibliothèque l donnée, nous définissons $U(l, i, p) \rightarrow \mathcal{P}(Files_p(i))$ comme la fonction déterminant l'ensemble des fichiers de code source de p utilisant les symboles de l à la version i. Soit $m = (s, t)$ une migration de bibliothèque ayant eu lieu entre les ver-

sions i et j. La valeur SIM est calculée de la façon suivante en utilisant le coefficient de Dice [Dice, 1945] :

$$SIM = \frac{2 * |U(s,i,p) \cap U(t,j,p)|}{|U(s,i,p)| + |U(t,j,p)|} . \tag{3.1}$$

Métrique des messages de *commits* (MSG). Lorsqu'un développeur migre une de ses bibliothèques, il est probable qu'il mentionne cette évolution dans les messages de *commits* du système de contrôle de versions. Ainsi, supposons une migration (s,t) observée sur un projet p entre deux versions i et j. Nous définissons l'ensemble $Msg_p(i,j)$ comme l'ensemble des messages de *commits* sur le projet p entre les versions i et j sur son système de contrôle de versions. Pour rappel, un seul message peut être indiqué par l'auteur du *commit*. Ainsi, nous calculons l'ensemble $Msg_p(i,j)$ pour toute migration de bibliothèque.

Chacun des messages $ms \in Msg_p(i,j)$ est ensuite séparé en phrases pour former un ensemble S_m de telles phrases. Nous avons utilisé la bibliothèque `opennlp` [9] qui emploie des marqueurs de fin de phrase pour ce traitement. Notons que la très grande majorité des messages de *commits* sont écrits en langue anglaise. Finalement, pour chacune des phrases $ph \in S_m$, la métrique de messages de *commits* MSG produit la valeur vraie si les trois conditions suivantes sont respectées :

1. ph contient le nom de la bibliothèque s ;
2. ph contient le nom de la bibliothèque t ;
3. ph contient au moins un des termes parmi l'ensemble des termes ayant une sémantique de migration (la liste est exposée en Annexe B).

Dans le cas inverse, MSG prend la valeur faux, et l'opération se répète pour toutes les autres phrases et messages restants. Lorsque MSG obtient la valeur vraie, le processus s'arrête et cette valeur est retournée.

Évaluation

Nous avons évalué ces trois paramètres sur le corpus des 866 projets GitHub ayant migré au moins une fois. Pour cela, nous avons à nouveau appliqué notre algorithme de détection des migrations de bibliothèque. Puis nous avons mesuré le nombre de règles détectées selon plusieurs valeurs de seuils pour les trois métriques. Pour rappel, le nombre total de règles initialement identifiées était de 329.

Remarque. Il nous paraît plus logique de travailler sur les règles de migration plutôt que sur les migrations elles-mêmes. Dans l'idéal, si le procédé automatique valide une règle de migration, elle le sera de façon définitive. Si la même règle est à nouveau rencontrée sur un

9. http ://opennlp.apache.org/

autre projet, celle-ci ne sera pas traitée de nouveau et sera donc comptabilisée (son score sera incrémenté).

Nous allons ainsi mesurer la précision et le rappel de notre approche selon plusieurs valeurs des trois métriques utilisées, ainsi que le F-score associé. Pour rappel, le F-score se calcule de la façon suivante :

$$F-score = \frac{2 * precision * rappel}{precision + rappel} . \tag{3.2}$$

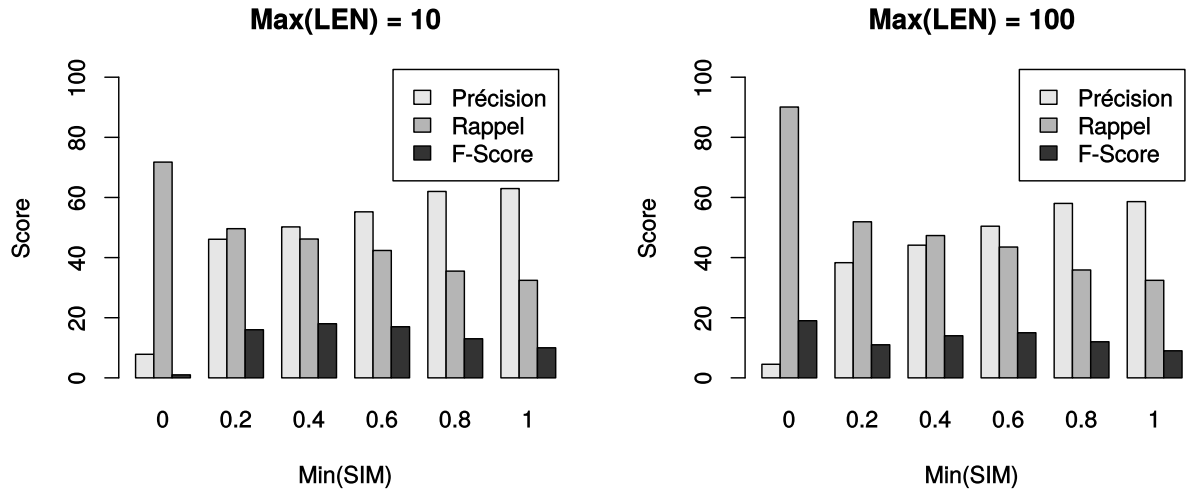
Il est à noter que, sur le corpus restreint de 866 projets, un total de 7 764 règles de migration a été généré. Ce qui signifie que notre corpus n'est pas biaisé puisqu'il contient 7764-329 = 7435 règles fausses. Les résultats des calculs effectués sont détaillés dans la Figure 3.11.

Observations

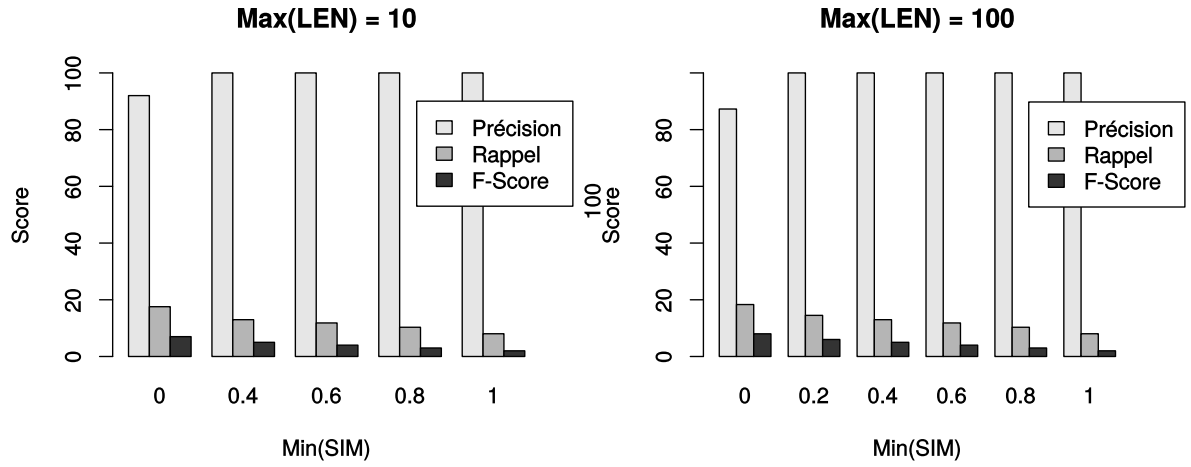
Premièrement, la métrique de longueur LEN apporte une faible valeur ajoutée sur la qualité des résultats. En effet, nous pouvons voir dans la Figure 3.11(a) (partie gauche) que, lorsqu'elle est utilisée sans les deux autres métriques, elle apporte une précision de 7,84 % avec un rappel de 71,76 %, ce qui signifie que peu de règles sont filtrées. Réduire ce paramètre améliore très légèrement la précision brute. L'impact entre les valeurs 10 et 100 entraîne une variation très faible en comparant les parties gauche et droite respectives des Figures 3.11(a) et 3.11(b). Cependant, il est intéressant pour la suite de nos travaux de constater que 71,76 % des règles de migration peuvent être observées dans des migrations de moins de 10 versions.

En revanche, la métrique de similarité de fichiers SIM démontre une influence plus élevée sur les résultats. Utilisée avec Max(LEN)=100 et MSG=FAUX, nous observons qu'un seuil minimal de 0,2 produit une précision de 38,3 % et un rappel de 51,9 %. Cette métrique est donc plus pertinente que LEN. De plus, l'augmentation de ce seuil provoque des effets attendus avec une amélioration de la précision en contrepartie d'une baisse du rappel. Ainsi, ce paramètre permet d'atteindre une précision de 30 % et un rappel de 33,2 % lorsqu'il est fixé à 1 (pour Max(LEN) = 10), soit un F-score de 42 %. De plus, il permet d'atteindre le F-score maximal de nos résultats (48,11 %).

Pour conclure, la métrique MSG est clairement celle ayant l'influence la plus marquée sur les résultats, comme nous pouvons le voir en Figure 3.11(b). En effet, elle génère une précision de 73.5 % et un rappel de 19,08 % lorsqu'elle est appliquée seule. Il est intéressant d'observer qu'une combinaison avec la métrique SIM permet d'atteindre 100 % de précision, avec SIM = 0,2. En revanche, le rappel est clairement diminué, puisque nous retombons à un score de 15,3 %.



(a) MSG=FAUX.



(b) MSG=VRAI.

FIGURE 3.11 : Précision, rappel et F-score des résultats produits pour plusieurs valeurs de LEN, MSG et SIM.

Conclusion

Notre objectif était d'améliorer la précision de notre approche en écartant un maximum de migrations candidates non valides. Si nous souhaitons répliquer nos expérimentations, une des priorités est d'alléger autant que possible le travail manuel de revue des règles. Les métriques proposées visent à filtrer les vraies migrations candidates des fausses.

Nous recommandons d'utiliser au minimum la métrique SIM pour garantir une précision moyenne mais acceptable, compte tenu des performances brutes obtenues sans filtrage. Utilisée seule, elle permet de couvrir une quantité non négligeable de règles de migration et atteint un compromis satisfaisant. Cependant, la métrique MSG peut être utilisée en complément pour améliorer considérablement la précision, mais en contrepartie la quantité de migrations observées est amoindrie.

Il est à noter que la métrique MSG demeure un paramètre variable et dépendant des pratiques des développeurs sur les systèmes de contrôle de versions sur lesquels ils contribuent. À l'inverse, nous pensons que la métrique SIM est moins sujette aux variations suivant les logiciels clients analysés.

Les métriques que nous proposons sont une première contribution vers l'identification automatique de vraies migrations de bibliothèque. Celles-ci demeurent encore perfectibles et nous pensons que d'autres analyses plus fines peuvent être effectuées. Par exemple, il arrive que la période de migration encapsule plusieurs *commits* sur le système de contrôle de versions. Nous pourrions donc effectuer des calculs pour chaque paire de *commits* successifs, puis agréger le résultat obtenu. Nous réservons néanmoins cette idée pour un travail futur.

3.7 Incertitudes sur la validité

Nous identifions plusieurs facteurs pouvant influencer les résultats et conclusions obtenus durant cette étude. Tout d'abord, une seule personne a manuellement déterminé si les règles de migration extraites lors des deux cas d'étude étaient vraies ou fausses. Cependant, il est permis d'envisager que cette personne se soit trompée dans son évaluation. Nous pensons malgré tout que l'impact sur les résultats finaux est négligeable.

Ensuite, même si les ensembles de bibliothèques L pour chaque cas d'étude s'avèrent être de taille raisonnable, ils ne sont constitués que de bibliothèques configurées avec MAVEN. Il se peut que cet ensemble ne soit pas entièrement représentatif de la totalité des bibliothèques Java disponibles.

De plus, la taille de L pour le cas d'étude n°2 est initialement de 10 224. Par manque de temps, nous n'avons pas fusionné les bibliothèques qui correspondent en réalité à une même bibliothèque. Cette fusion n'a été opérée que sur les bibliothèques impliquées dans les migrations candidates valides de cas d'étude. Ainsi, la taille réelle de cet ensemble est certainement inférieure à celle que nous mentionnons.

L'ensemble de projets *P* produit pour le cas d'étude n°1 a été exclusivement construit à partir de la plateforme GITHUB. Même si nous avons abordé plus en détail les caractéristiques de ce corpus, nous n'avons pas utilisé de méthodes d'échantillonnage rigoureuses pour l'établir. Nous ne pouvons donc généraliser nos résultats à tous les projets Java *open source*.

Enfin, les deux cas d'étude ont été réalisés à des dates différentes, puisque l'analyse du dépôt MAVEN a été effectuée en mars 2012 et que les projets GITHUB ont été traités en octobre 2013. Néanmoins, nous pensons que cela a une faible influence sur les conclusions que nous tirons puisqu'il est difficile de croire que la pratique de migration se soit accentuée d'une manière générale entre ces deux dates.

3.8 Limites et travaux futurs

Actuellement, l'intérêt d'observer les tendances de migrations de bibliothèque demeure au stade de la supposition et de nos expériences personnelles de développeur. Cependant, nous sommes persuadés que cet intérêt est réel et n'est pas uniquement théorique. Pour cette raison, nous envisageons de mettre en place une étude contrôlée avec des développeurs pour déterminer dans quelles mesures ces indicateurs ont une influence sur leurs décisions. Cela nous permettrait de perfectionner les visualisations existantes à partir des besoins concrets des développeurs dans ce contexte. Nous pourrions également vérifier si un effet de mode a lieu parmi une population de développeurs.

Une alternative à cette étude est d'étudier l'impact des fonctionnalités sociales proposées par GITHUB. La transparence et la visibilité des informations permettent aux utilisateurs d'être notifiés des activités d'autres projets et d'autres utilisateurs. Ainsi, nous pourrions observer s'il y a un effet de propagation des migrations de bibliothèque à travers le réseau social, lorsqu'un projet applique cette migration. Nous pourrions regarder plus en détail si les projets jouissant d'une forte popularité ont plus d'impact que les autres. Ce concept de dissémination a été abordée par Jiang et al. pour observer comment les développeurs recrutent des utilisateurs présents dans leur réseau [Jiang *et al.*, 2013].

Nous avons discuté des perspectives d'automatisation de notre approche. Des solutions supplémentaires peuvent être apportées dans ce sens pour améliorer la précision des résultats. Une première étape serait de caractériser plus en détail les migrations de bibliothèques en mesurant l'effort des développeurs pour accomplir une migration. Nous devons chercher à mieux comprendre les effets de bords d'une migration de bibliothèque et étudier par exemple si cette opération implique du *refactoring*. Ces résultats nous permettraient d'identifier plus efficacement une vraie migration de bibliothèque d'une fausse. Une autre piste consiste à améliorer la construction des catégories de bibliothèques en utilisant d'autres sources d'information. Si, dans l'absolu, toutes les catégories sont définies à l'avance, nous ne pouvons pas générer de migrations candidates fausses.

De plus, nous souhaiterions mettre en place une infrastructure d'observation communautaire des migrations de bibliothèques. L'idée serait d'extraire en permanence de nouvelles tendances de migrations à partir de corpus de projets régulièrement mis à jour, de façon similaire à un organisme de sondage des citoyens par exemple. Les utilisateurs seraient invités à ajouter de nouvelles bibliothèques (et donc des symboles) à la base de données de SCANLIB, pour prendre en compte ces bibliothèques dans nos analyses.

Nous envisageons d'analyser les messages des *commits* associés aux migrations de bibliothèques, ainsi que d'autres dépôts logiciels, pour extraire les raisons qui ont poussé les développeurs à migrer. L'objectif est d'améliorer la qualité des indicateurs proposés en permettant au développeur de mieux comprendre les tendances. Nous pensons notamment identifier des défauts dans les performances ou des fonctionnalités limitées chez certaines bibliothèques.

Pour terminer, nous aimerions également surmonter deux limites de notre approche. La première concerne l'existence de migrations de bibliothèques à cardinalité multiple n :m, au lieu du seul cas 1 :1 uniquement considéré ici. Par exemple, si une bibliothèque subit des transformations importantes, il se peut qu'elle soit divisée en deux bibliothèques distinctes. C'est le cas de la bibliothèque `commons-httpclient` qui a été séparée en deux bibliothèques `httpcore` et `httpclient`. Le deuxième point concerne la formation des catégories. Il existe des bibliothèques offrant des services diversifiés, comme la bibliothèque `guava` proposée par Google. Le problème est qu'il est possible de migrer depuis les bibliothèques Apache `commons-lang` et `commons-collections` vers `guava`, mais pas de `commons-lang` vers `commons-collections`. Cependant, selon notre définition, ces trois bibliothèques appartiennent à la même catégorie, tout en ne pouvant être substituées les unes aux autres. C'est pour cette raison que notre concept de catégorie nécessite des raffinements, pour prendre en compte ces aspects particuliers.

CHAPITRE

4 Application d'une migration de bibliothèque

Ce chapitre aborde notre deuxième contribution, dont le but est d'aider un développeur à accomplir une migration. Nous commençons par rappeler rapidement le contexte, les difficultés du problème et la solution proposée. Nous présentons dans un deuxième temps notre approche pour détecter automatiquement des correspondances de fonctions entre deux bibliothèques. Notre solution est ensuite implémentée et évaluée lors d'expérimentations qui nous permettent de discuter de ses performances. Nous concluons en ouvrant des perspectives futures, pour notamment améliorer la qualité des résultats produits.

Sommaire

4.1 Introduction

Ce chapitre cible la problématique détaillée en Section 1.2.2 et plus précisément l'application d'une migration de bibliothèque. Nous avons discuté dans le chapitre précédent que la première étape d'une migration consiste à déterminer vers quelles bibliothèques un développeur peut s'orienter. Une fois sa décision prise, le développeur doit procéder à la migration d'une bibliothèque source vers une bibliothèque cible. Cette opération exige

de lui qu'il mette à jour son code source afin qu'il devienne compatible avec la bibliothèque cible. Cela soulève plusieurs défis comme discuté en Section 2.2. Lors d'une migration de bibliothèque, le développeur souhaite préserver le fonctionnement de son logiciel. Il recherche donc les fonctions de l'API de la bibliothèque cible qui peuvent remplacer celles qu'il utilise et qui appartiennent à l'API source. Or, deux bibliothèques similaires mais indépendantes proposent des interfaces de programmation qui diffèrent. Considérons l'exemple du Listing 4.1, où un logiciel a effectué une migration de `commons-lang` vers `guava` [1]. Nous observons que la fonction *Validate.notNull* a été remplacée par un appel de *Preconditions.checkArgument*. Néanmoins, la lecture textuelle des signatures des deux fonctions laisse difficilement paraître une similarité sémantique.

LISTING 4.1 : Exemple de migration de `commons-lang` vers `guava`.

```
1 -import org.apache.commons.lang.Validate;
2 +import com.google.common.base.Preconditions;
3
4 public long getProblemVersion(String id) {
5 - Validate.notNull(id);
6 + Preconditions.checkArgument(id != null);
7 }
```

Notre solution tire parti des travaux menés dans le Chapitre 3. Puisque nous savons observer les migrations de bibliothèque, nous pouvons désormais étudier les changements des logiciels ayant accompli une migration donnée. L'idée est donc tout d'abord d'identifier un ensemble de tels logiciels, que nous appelons *projets sujets*. Par la suite, nous analysons les transformations du code source effectuées par chaque projet sujet pendant la migration. La finalité de ce procédé est d'obtenir un ensemble de correspondances entre fonctions de deux API indépendantes.

4.2 Approche

Notre approche comprend trois parties. La première a pour objectif de déterminer rapidement si un projet a effectué une migration donnée. Plus cette opération sera efficace, plus le temps de recherche de projets sujets sur une grande population de projets sera diminué, et plus la quantité finale de sujets sera accrue. Si une migration a eu lieu, nous identifions avec précision le plus petit segment de migration permettant de la délimiter :

Définition 4.1 (Segment de migration) *Pour une migration candidate* $(p, i, j, s, t) \in P \times \mathbb{N} \times \mathbb{N} \times L \times L$ *donné, le segment de migration* $[i, j]$ *correspond à l'intervalle de la migration de la bibliothèque* s *vers* t *sur le projet* p*. En reprenant l'exemple de la Figure 3.1 (voir Section 3.3.2), le segment obtenu pour la migration de* `junit` *vers* `testng` *est [18,23].*

1. https://bitbucket.org/anjensan/jaj/commits/c9a32d2

Un segment est donc le plus petit intervalle de versions englobant une migration. Dans un deuxième temps, nous effectuons à l'intérieur de chaque segment une analyse par *delta* des versions successives des fichiers de code source, afin d'identifier avec précision les parties éditées. Nous y recherchons en particulier les remplacements d'appels de fonctions vers les API des bibliothèques source et cible. A partir de cela, nous pouvons générer des correspondances de fonctions candidates. Cette approche par l'apprentissage est d'ailleurs inspirée du travail de Schäfer et al. qui se situe dans un contexte de mise à niveau de bibliothèque [Schäfer *et al.*, 2008]. Nous discuterons à ce sujet des différences obtenues avec notre technique, qui utilise un niveau d'analyse plus fin, en Section 4.3.3.

Dans un troisième temps, nous justifions que certaines des correspondances détectées peuvent s'avérer incorrectes, réduisant ainsi la précision de notre approche. Nous introduisons pour cela une technique de filtrage pour tenter d'écarter les fausses correspondances.

Les notations et termes utilisés dans ce chapitre sont repris de la Section 3.2. Pour ce travail, nous avons décidé de restreindre l'ensemble des symboles d'une bibliothèque à ses fonctions. Ce sont les symboles les plus utilisés dans un langage orienté objet tel que Java car ils permettent aux différents objets de communiquer. Nous nommons F_s un tel ensemble de fonctions, qui est un sous-ensemble des symboles S_l pour toute bibliothèque l. Notre technique peut néanmoins tout à fait s'appliquer à d'autres symboles, tels que les champs de classe ou les annotations.

4.2.1 Identification des segments de migration

Pour identifier un corpus de logiciels sujets et donc de segments, nous réutilisons l'algorithme d'extraction des migrations introduit en Section 3.3.2. Nous relevons cependant une différence avec le contexte actuel de ce travail. Il s'agissait précédemment de déterminer des migrations de bibliothèques candidates en supposant que nous n'avions pas connaissance de règles vraies existantes. Or, nous ne recherchons pas ici toutes les migrations candidates possibles mais bien une migration précise. Nous avons néanmoins observé qu'il est peu fréquent d'observer des migrations. Cela accentue la difficulté pour nous d'identifier un ensemble de projets sujets. Ainsi, nous proposons d'adapter cet algorithme afin d'améliorer sa performance et son temps d'exécution. Plus le nombre de logiciels sujets identifiés est important, et plus nous sommes susceptibles d'extraire des correspondances de fonctions.

Le point de départ de notre approche nécessite un ensemble de règles de migration vraies. Nous définissons M comme l'ensemble des règles de migration recherchées et notons $lib(M)$ l'ensemble de toutes les bibliothèques cibles des migrations de M.

L'opération la plus coûteuse dans la recherche de segments de migrations est d'extraire les bibliothèques utilisées par un projet logiciel pour chacune de ses versions. Nous confirmerons, grâce aux expériences menées en Section 4.3.3, l'aspect coûteux de cette opération. Pour surmonter ce point, nous proposons d'utiliser une recherche dichotomique

pour révéler l'existence d'un segment de migration tout en épargnant le calcul des bibliothèques utilisées à chaque version. Pour nous aider en complément, nous introduisons quatre heuristiques qui reposent respectivement sur les hypothèses suivantes :

1. les bibliothèques peuvent être remplacées mais jamais supprimées du projet. Si une migration a lieu, la bibliothèque cible doit être présente à la dernière version du projet ;
2. il est peu probable d'observer plus d'une migration (ex : $s \rightarrow t \rightarrow u$) au sein d'un court intervalle de versions ;
3. les situations de *rollbacks* (ex : $s \rightarrow t \rightarrow s$) sont très rares ;
4. il existe peu de segments de migration dans l'historique d'un projet, et ceux-ci sont de courte durée (en termes de nombre de versions).

L'Algorithme 2 décrit la recherche dichotomique des segments de migration basée sur ces hypothèses. Il prend en entrée deux versions i et j d'un projet p avec $i < j$. Puis, si $dep_p(i) = dep_p(j)$ ou $dep_p(j) \cap lib(M) = \emptyset$, la recherche s'arrête entre i et j. Sinon, nous traitons les deux cas suivants en introduisant t_{min}, qui est un seuil prédéfini de taille d'intervalle de versions :

1. $j - i \leq t_{min}$: nous vérifions qu'une règle valide de migration peut être inférée à partir des bibliothèques utilisées aux versions i et j (resp. $dep_p(i)$ et $dep_p(j)$). Si c'est le cas, nous utilisons un algorithme de recherche binaire pour déterminer les limites exactes du segment, et nous l'ajoutons à l'ensemble de segments.
2. $j - i > t_{min}$: l'algorithme est exécuté récursivement sur les intervalles $(i, \frac{i+j}{2})$ et $(\frac{i+j}{2}, j)$. La version équivalente à $\frac{i+j}{2}$ représente ici le point de séparation déterminé par l'algorithme.

Le point faible de cet algorithme est inhérent à sa nature dichotomique et concerne le choix des points de séparation. Il se peut que la séparation s'effectue au milieu d'un segment, auquel cas celui-ci ne peut être détecté. De ce fait, la valeur choisie pour t_{min} ne doit donc pas être trop restrictive.

Pour mieux comprendre cet algorithme, observons l'illustration de la Figure 4.1 où apparaît un projet p composé de 7 versions. Soit un ensemble de 2 règles de migration M = {`junit`→`testng`, `log4j`→`slf4j` }. Nous utilisons ici un seuil t_{min} égal à 3. Nous pouvons voir que les bibliothèques associées aux versions v_0 et v_6 sont différentes, et que `slf4j`, présente à la version v_6, est une cible de migration. Ainsi, nous divisons l'intervalle en deux et appliquons la recherche sur les tronçons (v_0, v_3) et (v_3, v_6). Puisque $dep(v_0) \neq dep(v_3)$ et que $3 - 0 \leq t_{min}$, nous calculons pour ce premier intervalle `rem` = {`log4j`} et `add` = {`slf4j`}. Étant donné que `slf4j`→`log4j` $\in M$, nous appelons la procédure `extract` avec les paramètres v_0, v_3, `log4j` et `slf4j`. Cette procédure effectue une recherche binaire pour déterminer les limites exactes du segment. À l'inverse, comme $dep(v_3) = dep(v_6)$, aucune action supplémentaire n'est effectuée dans cet intervalle.

Algorithme 2 Extraction de segments de migration

Requiert: p : un projet
Requiert: V_p : l'ensemble des versions de p
Requiert: t_{min} : une longueur minimale
Requiert: M : un ensemble de règles de migration vraies

```
Seg ← ∅
FIND_SEGMENT(v_0, v_head, Seg)
retourner Seg
fonction FIND_SEGMENT(v_i, v_j, Seg)
    si dep_p(v_j) ∩ lib(M) = ∅ ∨ dep_p(v_i) = dep_p(v_j) alors
        retourner
    sinon si (j − i) ≤ t_min alors
        rem = dep_p(v_i) \ dep_p(v_j)
        add = dep_p(v_j) \ dep_p(v_i)
        pour tout s ∈ rem faire
            pour tout t ∈ add faire
                si s → t ∈ M alors
                    Seg ← Seg ∪ EXTRACT(v_i, v_j, s, t)
                    retourner
                fin si
            fin pour
        fin pour
    sinon
        v_m ← (v_i + v_j) / 2
        FIND_SEGMENT(v_i, v_m, Seg)
        FIND_SEGMENT(v_m, v_j, Seg)
    fin si
fin fonction
```

Finalement, le segment [2,3] est extrait et 4 versions ont été analysées au total dans cet exemple (en incluant celles analysées par la procédure `extract`). L'approche décrite auparavant en Section 3.3.2, qualifiée d'*exact* car ne manquant aucun segment de migration, aurait analysé l'intégralité des 7 versions du projet.

4.2.2 Génération des correspondances de fonctions

La deuxième étape consiste à examiner chaque segment de migration identifié. Notre technique suppose que, durant une migration de bibliothèque $s \rightarrow t$, un développeur remplace des symboles de S_s par des symboles de S_t à des positions similaires dans le fichier de code source. Par conséquent, nous utilisons une analyse fine des changements du code

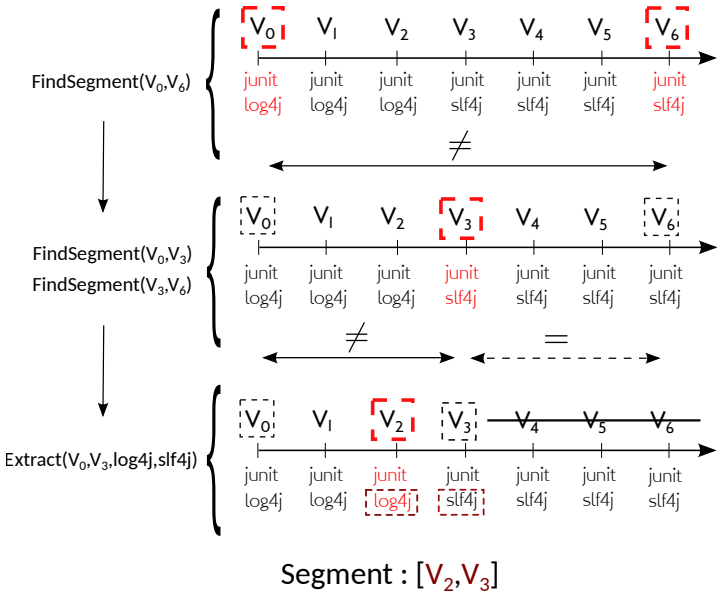


FIGURE 4.1 : Illustration de notre algorithme sur un projet d'exemple. Ici, $t_{min} = 3$.

entre chaque paire de versions successives de l'intervalle $[i, j]$. Pour chacune d'elles, nous retenons les fichiers de code source marqués comme modifiés dans le *changeset* de la version la plus récente. Nous supposons qu'il est possible d'accéder à ces fichiers ainsi qu'à leur version précédente, et ce pour toute paire de versions $(k, k+1)$, avec $i \leq k < j$. Nous calculons, pour chacun de ces fichiers, les différences textuelles à partir de leur révision parente, au moyen de l'outil *diff* [Myers, 1986]. Cette opération génère une collection de *hunks* qui résument les modifications produisant le fichier à la version $k+1$ à partir de la version k.

Un hunk est une séquence de lignes de code supprimées, ajoutées et supprimées ou ajoutées. Il contient un en-tête avec la position des lignes supprimées dans le fichier à la version k (*old*), et la position des lignes ajoutées dans le fichier à la version $k+1$ (*new*). Notons que soit une ligne, soit une plage de n lignes peuvent être indiquées dans cet en-tête. Le Listing 4.4 montre les hunks calculés à partir des codes sources visibles dans les Listings 4.2 et 4.3. Notre intérêt est de conserver uniquement les hunks contenant à la fois des lignes supprimées et ajoutées, et d'enregistrer quelles sont les positions des lignes supprimées (resp. ajoutées) dans la version *old* (resp. *new*) du fichier.

LISTING 4.2 : Bar.java - version 1.

```
1  public class Bar {
2
3    public void test() {
4      Log.getLog("MyLogger");
5      something(3);
6    }
7
8    public void something(int i) {
9      Log.getLog("MyLogger");
10     if (i > 0) {
11       Log.fatal("Error");
12     }
13   }
14 }
```

LISTING 4.3 : Bar.java - version 2.

```
1  public class Bar {
2
3    public void test() {
4      Logger.getLogger("MyLogger");
5      other(3);
6    }
7
8    public void other(int i) {
9      Logger.getLogger("MyLogger");
10     if (i > 0) {
11       Logger.error("Error");
12     }
13   }
14 }
```

LISTING 4.4 : *Diff* entre les versions 1 et 2 de Bar.java.

```
1  @@ -4,2 +4,2 @@
2  -  Log.getLog("MyLogger");
3  -  something(3);
4  +  Logger.getLogger("MyLogger");
5  +  other(3);
6  @@ -8,2 +8,2 @@
7  - public void something(int i) {
8  -  Log.getLog("MyLogger");
9  + public void other(int i) {
10 +  Logger.getLogger("MyLogger");
11 @@ -11 +11 @@
12 -   Log.fatal("Error");
13 +   Logger.error("Error");
```

Nous analysons ensuite les deux versions du fichier pour collecter la position des lignes contenant des appels de fonctions à F_s et F_t respectivement dans les fichiers *old* et *new*. Nous ne retenons que les hunks h pour lesquels (1) les lignes supprimées contiennent au moins un appel de fonction de F_s ;(2) les lignes ajoutées contiennent au moins un appel de fonction de F_t. Nous notons $rem(h)$ (resp. $add(h)$) les fonctions de F_s (resp. F_t) supprimées (resp. ajoutées) au sein de h. Pour un tel hunk, la génération de correspondances de fonctions candidates s'effectue par le produit cartésien $rem(h) \times add(h)$.

Définition 4.2 (Correspondance de fonctions) *Étant donné une règle de migration $s \rightarrow t \in M$, une correspondance de fonctions $x \leftrightarrow y$, avec $x \in F_s$ et $y \in F_t$, indique qu'une similarité existe entre x et y et qu'il est possible de remplacer x par y, et vice-versa. Le score $sc(x,y)$ de la correspondance est le nombre de hunks dans lesquels elle a été observée. Nous la nommerons également règle de correspondance.*

Pour illustrer nos propos, supposons les versions 1 et 2 du fichier *Bar.java* décrites dans les Listings 4.2 et 4.3. Le *diff* associé est composé de trois hunks visibles dans le Listing 4.4, et indiqués par des symboles @@. Ceux-ci sont retenus car ils contiennent des suppressions d'appels de fonctions de F_s et des ajouts d'appels fonctions de F_t. Ils permettent de générer les deux correspondances suivantes :

— Log.getLog(String) ↔ Logger.getLogger(String) (hunks n°1 et n°2) ;

— Log.fatal(String) ↔ Logger.error(String) (hunk n°3)

Notons pour terminer que cet exemple démontre en quoi les hunks permettent de détecter finement les modifications du code. La méthode `something(int)` a ici été renommée en `other(int)`, mais nous sommes malgré tout en mesure d'extraire les correspondances sans générer de faux positifs. En effet, l'instruction `if` aux lignes 10 des deux versions permet de séparer les transformations du code qui se sont produites ici.

4.2.3 Filtrage des correspondances

Des produits cartésiens sont appliqués pour extraire des correspondances de fonctions. Cependant, si le nombre de lignes de code d'un hunk s'avère conséquent, il est probable qu'il englobe plusieurs appels de fonctions. Dans une telle situation, le produit cartésien a de fortes chances de produire des faux positifs. C'est une des limites de la technique du *diff* sur les lignes de code. Pour réduire ce nombre de faux positifs impactant la précision de notre approche, nous proposons une technique légère de filtrage des correspondances. Nous reprenons une idée introduite par Melnik et al. basée sur les similarités relatives [Melnik *et al.*, 2002].

L'objectif est de mettre en avant les éléments de deux ensembles ayant une forte similarité mutuelle. Nous supposons que les correspondances de fonctions candidates sont formées à partir de deux ensembles C_s et C_t, incluant respectivement des fonctions de S_s et S_t. À chaque règle candidate x↔y, $x \in S_s$ et $y \in S_t$, est associé le score $sc(x,y)$ aussi appelé *similarité absolue*. Un graphe G non orienté est ensuite construit, chaque sommet représente une fonction, et il existe un arc entre deux sommets x et y pour chaque règle candidate x↔y connue. Puis, pour chaque sommet n du graphe, nous enregistrons sa similarité absolue maximale max_n avec ses sommets voisins. Un nouveau graphe G' est ensuite construit en étiquetant chaque arc (x,y) aux deux extrémités avec deux scores relatifs, $\frac{sc(x,y)}{\max_x}$ (côté x) et $\frac{sc(x,y)}{\max_y}$ (côté y), notés respectivement $\alpha(x,y)$ et $\alpha(y,x)$. Le graphe final est ainsi composé de similarités relatives sous la forme de valeurs comprises entre 0 et 1.

Pour mieux clarifier cette opération, considérons l'exemple exposé en Figure 4.2. Il y a quatre règles de correspondance candidates associées, avec leur similarité absolue. Le graphe G de base apparaît sur la gauche. La partie centrale représente G', avec les similarités relatives calculées par fraction pour chaque nœud n et avec leur valeur $\max_n$. Les résultats finaux apparaissent dans le graphe de la partie droite. À cette étape, nous avons associé à chaque règle x↔y deux scores de similarité relatives.

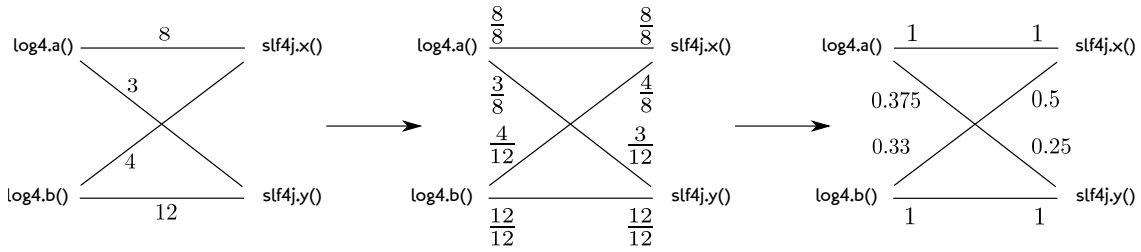


FIGURE 4.2 : Calcul des similarités relatives pour des règles de correspondance candidates.

Pour terminer, nous retenons les règles de correspondance x↔y dont les similarités relatives sont au minimum égales à un seuil t_{rel}, avec $0 < t_{rel} \leq 1$, ce qui signifie que $\alpha(x,y)$ et $\alpha(y,x)$ doivent être égaux ou supérieurs à t_{rel} pour valider cette règle.

Dans notre exemple, fixer t_{rel} = 0,6 nous permet de conserver les règles `log4j.a()`↔`slf4j.x()` et `log4j.b()`↔`slf4j.y()`. En revanche, le fixer à 0,25 valide toutes les règles candidates tandis que seulement la correspondance `log4j.a()`↔`slf4j.y()` est écartée si le seuil est égal 0,3.

4.3 Évaluation empirique

Dans cette section, nous appliquons notre approche à un corpus de projets *open source*. Puis, nous analysons quantitativement les résultats obtenus pour juger de la mise en pratique de notre technique. Dans un deuxième temps, nous proposons une analyse qualitative des résultats, afin d'évaluer les performances de notre approche et discuter de sa généralisation.

4.3.1 Constitution du corpus et du jeu de données

Après avoir examiné les résultats de notre étude précédente (voir Chapitre 3), nous avons sélectionné quatre couples de bibliothèques, où chacun représente une migration de bibliothèque et son opposée. Chaque couple forme ici une catégorie. Deux d'entre eux incluent des bibliothèques proposées par les projets Apache Commons et Guava. Apache Commons est « *un projet focalisé sur tous les aspects des composants Java réutilisables* »[2], tandis que Guava « *contient plusieurs bibliothèques centrales à Google qui reposent sur des projets Java* »[3]. Ainsi, ces deux projets étendent ou ré-implémentent des fonctionnalités disponibles dans les bibliothèques natives de Java et du JDK[4]. Pour chaque projet, nous

2. http://commons.apache.org/
3. https://code.google.com/p/guava-libraries/
4. http://docs.oracle.com/javase/7/docs/api/

avons sélectionné deux API pour former les deux premières catégories : `guava.io` et `commons.io` (nommé *I/O*) ainsi que `guava.lang` et `commons.lang` (nommé *Lang*).

La troisième catégorie concerne deux bibliothèques proposant des fonctionnalités pour traiter des données au format JSON : la bibliothèque standard `org.json`[5] et la bibliothèque Google `gson`[6]. Nous nommerons *Json* ce couple de règles de migration. Pour terminer, la dernière catégorie nommée *Mock* implique deux bibliothèques aidant à l'écriture de tests contenant des objets mock[7], `jmock`[8] et `mockito`[9].

Le résumé des catégories sélectionnées ainsi que les API associées est récapitulé dans le Tableau 4.1.

Tableau 4.1 : Corpus des catégories de bibliothèques avec la taille des API respectives en nombre de fonctions.

Catégorie	**Paquetage**	**Taille**
I/O	`org.apache.commons.io`	944
	`com.google.common.io`	201
Lang	`org.apache.commons.lang`	2005
	`com.google.common.base`	224
	`com.google.common.primitives`	225
Json	`org.json`	184
	`com.google.gson`	270
Mock	`org.jmock`	118
	`org.mockito`	733

Pour effectuer cette étude, nous avons collecté un ensemble de projets depuis les plateformes *open source* SOURCEFORGE, GOOGLECODE et GITHUB. Nous avons ainsi aléatoirement tiré 14 000 projets, desquels nous avons supprimé les projets très peu actifs (moins de 10 *commits*) et ceux ayant été désactivés (l'adresse du dépôt étant devenue inaccessible), produisant un corpus total de 11 598 projets Java. Ce corpus a été constitué en mars 2013.

4.3.2 Données quantitatives sur les résultats

Extraction des segments de migration

Nous avons fixé une distance minimale $t_{min} = 25$ pour la recherche de segments. Nous avons appliqué l'algorithme décrit en Section 4.2.1 sur notre corpus de projets. Cette opé-

5. http://json.org/java/
6. http://code.google.com/p/google-gson/
7. http://en.wikipedia.org/wiki/Mock_object
8. http://jmock.org/
9. https://code.google.com/p/mockito/

Tableau 4.2 : Segments de migration et hunks.

(a) Statistiques sur les données obtenues.

Catég.	**#Segments**	**#Hunks**	**#Fonctions**
I/O	12	33	31
Lang	14	66	55
Json	4	99	48
Mock	6	87	35
Total	36	285	169

(b) Nombre de fonctions dans les hunks.

Catég.	2 Fonc.	3 Fonc.	4-6 Fonc.	7-10 Fonc.
I/O	31	2	0	0
Lang	39	8	19	0
Json	51	7	32	9
Mock	9	6	12	60
Total	130	23	63	69

ration a nécessité plus de trois jours de calcul. Les nombres de segments de migration qui ont été extraits sont exposés dans le Tableau 4.2(a). Nous avons ainsi rassemblé un total de 36 segments de migration, ce qui est peu par rapport à la taille du corpus analysé. Néanmoins, ces chiffres ne sont pas surprenants, étant donné que la migration de bibliothèque demeure une pratique peu fréquente, comme nous l'avons vu en Section 3.4.3. Au total, 169 fonctions apparaissent dans les résultats, les catégories *Lang* et *Json* étant les plus représentées.

Correspondances obtenues

Nous avons pu collecter 285 hunks de migrations. La répartition du nombre de fonctions par hunk est exposée dans le Tableau 4.2(b). Nous observons que la majorité des hunks contiennent uniquement 2 fonctions, ce qui constitue le minimum pour une correspondance de fonctions. La quasi-totalité des hunks de migrations pour la catégories *I/O* est d'ailleurs de ce type, ce qui suppose une bonne précision des correspondances générées puisqu'il y a de fortes chances qu'un hunk de 2 fonctions produise une correspondance valide. À l'inverse, nous observons des hunks plus fournis en termes de fonctions pour les catégories *Json* et *Mock*, pour qui 60 hunks sur 87 possèdent plus de 7 fonctions. Nous nous attendons ainsi à une précision affaiblie sur ces catégories.

Un total de 228 correspondances de fonctions a finalement pu être extrait. Deux personnes ont ensuite manuellement examiné chacune des règles durant 3 heures chacune. Ces deux personnes pratiquent toutes les deux le développement Java depuis respectivement 4 et 8 années, mais n'avaient pas de connaissance spécifique des bibliothèques impliquées dans l'étude. La documentation des bibliothèques (sous forme de *Javadoc*) ainsi que les *diff* textuels obtenus sur les projets clients ont servi d'aide pour l'opération. Tous les cas épineux ont été discutés et décidés d'un commun accord. Les experts ont ainsi validé 115 correspondances et écarté 113 autres. La précision brute de l'approche sur cette évaluation est donc d'environ 50 %. Nous nommons C l'ensemble des correspondances valides de fonctions.

L'ensemble des règles retenues est disponible sur notre page web [10]. Pour chaque règle, nous proposons les extraits de code source associés. Un exemple de règle validée pour la catégorie *I/O* est affiché en Figure 4.3.

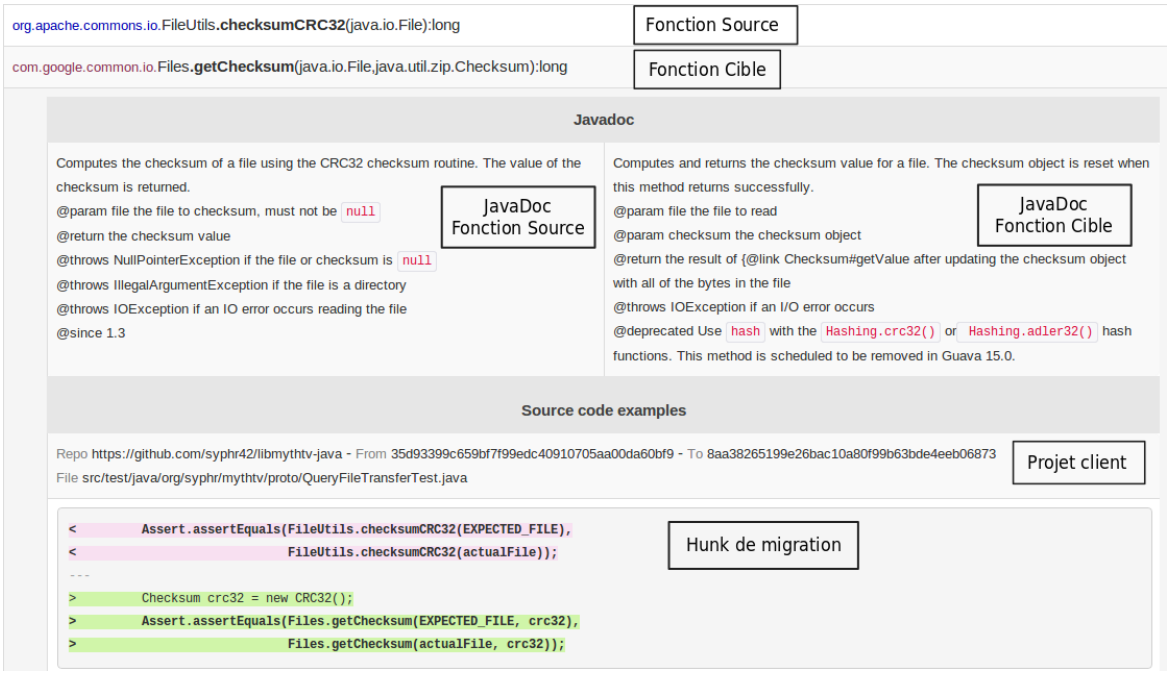


FIGURE 4.3 : Exemple de rapport produit pour les règles de correspondance, ici avec une règle validée pour la catégorie *I/O*.

4.3.3 Mesures qualitatives de l'approche

Nous examinons plus en détail les résultats afin de répondre aux trois questions suivantes :

— Notre algorithme d'extraction de segments est-il aussi efficace que l'algorithme exact (voir Section 3.3.2) et est-il plus rapide en temps de calcul ? (Q1)

— Notre technique d'extraction basée sur les hunks est-elle plus précise qu'une technique alternative basée sur la structure du code source proposée par Schäfer et al. [Schäfer *et al.*, 2008] ? (Q2)

— Notre technique de filtrage améliore-t-elle la qualité des résultats ? (Q3)

10. https://se.labri.fr/Matching/

Tableau 4.3 : Performance de notre technique d'extraction de segments de migration. P est le numéro du projet dans notre corpus. #KLOC est le nombre de kilos LOC Java à la dernière version du projet. V_{exact} et V_{ours} correspondent au nombre de versions analysées respectivement par l'algorithme exact et par notre approche. Γ_V est le gain de temps en pourcentage de V_{ours} par rapport à V_{exact}. T_{exact} et T_{ours} sont les temps d'extraction en secondes pour respectivement l'algorithme exact (voir Section 3.3.2) et notre approche. Γ_T est le gain de temps en pourcentage de T_{ours} par rapport à T_{exact}, et Δ_S est le nombre de segments de migration manqués par notre algorithme.

P	#KLOC	V_{exact}	V_{ours}	Γ_V	T_{exact}	T_{our}	Γ_T	Δ_S
1	0,8	56	8	-85,7 %	1,2	1,5	+25,7 %	0
2	4,5	199	12	-93,9 %	6,8	1,2	-82,4 %	0
3	39	528	14	-97,4 %	88	2,9	-96,7 %	0
4	53	1095	13	-98,8 %	442	7,5	-98,3 %	0
5	1,1	106	11	-89,6 %	2	0,2	-90,0 %	0
6	2,9	76	10	-86,8 %	3	0,5	-83,3 %	0
7	2,9	56	8	-85,7 %	2	0,3	-85,0 %	0
8	3,4	116	12	-89,6 %	10	0,8	-92,0 %	0
9	29	4411	36	-99,1 %	1643	13	-99,2 %	0
10	9,1	453	17	-96,2 %	30	3	-90,0 %	0

Extraction des segments (Q1)

Nous avons comparé notre technique d'extraction de segments de migration avec l'algorithme exact qui capture tous les segments au sein d'un projet p. Pour cela, pour chacune des versions $i \in V_p$, nous déterminons l'ensemble des bibliothèques utilisées. Puis, l'intégralité des versions est parcourue afin d'identifier des segments de migrations. Nous voulons démontrer que cette technique naïve a pour inconvénient d'être beaucoup plus coûteuse que la technique que nous proposons, rendant ainsi son exploitation à grande échelle moins convaincante.

Les deux techniques ont été appliquées sur un corpus similaire de 10 projets au sein desquels nous avons observé des migrations durant nos expérimentations. Nous avons mesuré le nombre de versions analysées, le nombre de segments extraits et le temps d'exécution. Notons que ce temps n'inclut pas le clonage initial des dépôts ni le temps d'extraction des correspondances, mais bien uniquement le temps d'extraction des segments. Les résultats de cette comparaison sont exposés dans le Tableau 4.3.

Deux principales observations ressortent de ces résultats. Premièrement, les deux algorithmes ont détecté le même nombre de segments de migration. Après inspection manuelle, les segments détectés sont identiques au niveau de leurs bornes et correspondent aux mêmes migrations. Deuxièmement, le temps d'exécution et le nombre de versions analysées sont réduits de façon significative, ce qui montre bien le gain en temps apporté

par notre technique. En effet, le temps d'exécution ne dépasse pas 13 secondes et est en moyenne 80 % plus rapide.

Remarque. Le temps d'exécution de notre algorithme sur le projet n°1 s'avère plus long que l'algorithme exact, bien que le nombre de versions traitées soit moindre. Nous n'avons pu expliquer ce phénomène, dont l'ampleur doit être tempérée par la faible échelle de valeur (dixième de seconde) des temps d'exécution sur ce projet.

Ainsi, notre technique est tout à fait capable d'extraire des segments de migration, et même si nous avons potentiellement manqué des migrations durant nos expérimentations, il y a fort à penser que cette proportion demeure très faible. Cependant, il est important de souligner que les performances de notre algorithme sont dépendantes de la taille de la base de données M de migrations valides à rechercher. En effet, plus le nombre de bibliothèques impliquées dans cet ensemble est conséquent, plus l'algorithme effectuera des itérations supplémentaires. Nous pensons néanmoins que notre solution présente un intérêt lorsque les migrations de bibliothèques recherchées sont prédéfinies.

Qualité des hunks de migration (Q2)

Résultats bruts. Les données quantitatives sur les règles identifiées par notre approche sont présentées dans le Tableau 4.4. Comme nous l'attendions, notre approche produit de bons résultats en termes de précision pour la catégorie *I/O* avec un seul faux positif parmi 22 correspondances générées. La catégorie *Lang* possède également une bonne précision, avec 40 règles valides sur 47, tandis que les précisions sont passables pour *Json* et *Mock* (resp. 31,2 % et 37,9 %).

Nous justifions les écarts de précision par les caractéristiques d'utilisation des bibliothèques. En effet, une majorité des correspondances trouvées au sein des catégories *Lang* et *I/O* sont des fonctions Java *static* avec des objectifs très spécifiques et une sémantique aisée à comprendre. En revanche, les bibliothèques `jmock` et `mockito` sont très différentes et reflètent bien la complexité du problème de migration de bibliothèque. Les structures des bibliothèques des catégories *Lang* et *I/O* sont donc plus similaires que celles des bibliothèques des deux autres catégories. L'inspection manuelle des correspondances pour cette catégorie fut par moment délicate, car l'emploi de ces deux bibliothèques diffère fortement. C'est donc l'opposé des catégories *Lang* et *I/O*, dont les utilisations sont plus triviales. Ainsi, si l'utilisation typique d'une bibliothèque découle d'une combinaison de plusieurs appels de méthodes, la précision sera influencée négativement.

Comparaison avec une technique alternative. Nous confrontons l'utilisation des hunks de migration avec la solution proposée par Schäfer et al. [Schäfer *et al.*, 2008], qui s'applique sur le problème de mise à niveau de bibliothèque. Nous avons repris l'idée de cette technique et l'avons adaptée plus finement à notre contexte. Cette approche (nommée Schäfer par la suite) considère les méthodes des classes Java comme code client des bibliothèques et donc comme contexte d'utilisation. Elle analyse deux versions du code client

Tableau 4.4 : Précision et rappel des approches Hunk et Schäfer. #Vrai et #Faux indiquent le nombre de règles vraies et fausses resp.. Le rappel est calculé à partir de l'union des 135 correspondances détectées par les deux méthodes.

	Hunk		**Schäfer**	
Catégorie	#Vrai	#Faux	#Vrai	#Faux
I/O	21	1	18	10
Lang	40	7	38	30
Json	29	64	25	163
Mock	25	41	11	42
Total	**115**	**113**	**92**	**245**
Précision	**50 %**		**27 %**	
Rappel	**85 %**		**68 %**	

avant et après la mise à niveau de bibliothèque. Pour chaque méthode de classe présente dans ces deux versions, les appels de fonctions vers la bibliothèque sont calculés pour chacune des deux versions. Un produit cartésien est ensuite effectué afin de générer les correspondances candidates entre fonctions appelées supprimées et ajoutées. Tout comme notre procédé, cette opération est appliquée pour chaque paire de versions successives incluse dans un segment de migration. Notre objectif est de nous assurer que l'utilisation des hunks garantit une meilleure précision que la technique d'extraction par contexte de méthode.

En appliquant la technique Schäfer sur notre corpus de projets migrants, nous avons obtenu un total 337 correspondances de fonctions, dont 92 ont été validées et 245 mises de côté (selon le même processus de vérification). L'union des correspondances valides identifiées par les deux approches fait augmenter le total de C à 135 règles, signifiant que 20 nouvelles règles ont pu être identifiées. La distribution des correspondances extraites par catégorie de migration est disponible dans le Tableau 4.4. Notons que le rappel est calculé sur l'ensemble des 135 règles comprises dans C. Il s'avère que la précision de cette approche est de 27 %, ce qui est bien inférieur à la technique que nous proposons. De plus, nous obtenons un rappel de 68 %, puisque 92 règles ont été générées contre 115 avec nos résultats.

La précision obtenue par la méthode Schäfer pour la catégorie *I/O* est moins bonne que la nôtre au vu des 10 faux positifs obtenus parmi 28 règles. De plus, même si la précision obtenue pour la catégorie *Json* est très passable nous concernant, elle est particulièrement faible pour la méthode Schäfer (25 règles validées et 163 règles fausses). Il apparaît donc que notre approche permet d'atteindre une meilleure précision dans les résultats obtenus, sans pour autant diminuer significativement le nombre de règles détectées.

En définitif, la détection de correspondances de fonctions entre bibliothèques par

Tableau 4.5 : Impact du filtre sur la précision et le rappel pour plusieurs valeurs de t_{rel}.

	I/O		**Lang**		**Json**		**Mock**	
t_{rel}	Prec.	Rap.	Prec.	Rap.	Prec.	Rap.	Prec.	Rap.
0,0	*95,5%*	*100%*	*85,1%*	*100%*	*31,2%*	*100%*	*37,9%*	*100%*
0,2	95,5 %	100 %	84,8 %	97,5%	39,1%	93,1%	46,4%	52%
0,4	94,7 %	85,7 %	83,7 %	90 %	61,9 %	89,7 %	64,7 %	44%
0,6	93,3 %	66,7 %	82,4 %	70 %	81,8 %	62,1 %	60 %	24 %
0,8	92,9 %	61,9 %	81,8 %	67,5 %	81,8 %	62,1 %	50 %	16%
1	92,9 %	61,9 %	79,3 %	57,5 %	81 %	58,6 %	50 %	8 %

l'analyse de code client nécessite une analyse très fine des changements. De plus, en reprenant l'exemple du Listing 4.4 vu précédemment, la méthode Schäfer détecterait que seule la méthode `test()` est présente au sein des versions 1 et 2, et omettrait les méthodes `something(int)/other(int)`. Cependant, l'utilisation de techniques de détection de renommage d'entités de code source ne suffit pas à surmonter ce problème, puisqu'un produit cartésien effectué à partir des méthodes `something(int)/other(int)` produit 2 faux positifs et 2 vrais positifs. À l'inverse, notre technique est certes plus robuste au renommage des méthodes, mais elle est plus sensible aux déplacements de tronçons de code au sein d'un fichier. En effet, l'opération de *diff* utilisée peut perdre la trace des positions d'origine lorsque de nombreuses transformations ont lieu.

Évaluation du filtre (Q3)

Le filtre présenté en Section 4.2.3 vise à améliorer la précision de notre approche. Nous allons voir si toutefois il permet d'atteindre un compromis satisfaisant entre précision et rappel des résultats, et nous mesurons pour cela l'impact du seuil t_{rel} sur les règles obtenues. Il est ici uniquement appliqué sur les 115 règles identifiées par notre technique d'extraction. Les données obtenues pour plusieurs valeurs de t_{rel} entre 0 et 1 sont exposées dans le Tableau 4.5.

L'impact du filtre n'est pas constant selon les catégories de migrations. En effet, pour les catégories *I/O* et *Lang*, la précision des résultats n'est pas améliorée. Cela entraîne même une diminution du rappel et de la précision. Ceci peut s'expliquer par une qualité brute satisfaisante des règles pour ces deux catégories. À l'inverse, l'impact est positif pour les catégories *Json* et *Mock* puisque la précision est améliorée. Pour une valeur de $t_{rel} = 0,6$, la précision et le rappel atteignent respectivement 81,8 % et 62,1 %. Dans tous les cas, le filtre a pour inconvénient attendu de diminuer la quantité de règles générées. Cependant, il est difficile de prédire une valeur générique de ce seuil. Nous pensons toutefois que ce filtre peut être utilisé dans le cas où la précision brute est faible.

Tableau 4.6 : Similarité syntaxique entre correspondances de fonctions.

t_n	I/O	Lang	Json	Mock
0.5	82,6 %	27,1 %	43,2 %	22,2 %
0.6	73,9 %	16,7 %	37,8 %	18,5 %
0.7	73,9 %	12,5 %	29,7 %	18,5 %
0.8	73,9 %	12,5 %	27 %	18,5 %
0.9	73,9 %	12,5 %	27 %	18,5 %
1.0	73,9 %	12,5 %	27 %	18,5 %

4.3.4 Discussion sur les correspondances de fonctions

Nous concluons en analysant plus finement les correspondances de fonctions détectées par notre approche, afin de dégager des perspectives futures.

Similarité textuelle

Une de nos hypothèses de travail est que la similarité textuelle entre les noms de fonctions n'est pas pertinente dans un contexte de bibliothèques indépendantes. Cela implique que les signatures des fonctions pour chaque bibliothèque n'ont aucun point commun. Nous souhaitons étudier ici la véracité de ce propos. Nous avons pour cela repris l'ensemble C de 135 règles et avons calculé la similarité n-gram entre les noms (et non pas les signatures complètes) de chaque paire de fonctions [Shannon, 2001]. Cette mesure produit une valeur entre 0 et 1, où 1 indique que deux chaînes de caractères sont similaires. Nous avons simulé, pour plusieurs valeurs de seuil t_{txt}, le nombre de règles filtrées par une telle opération. Les résultats indiqués dans le Tableau 4.6 confirment notre hypothèse, puisqu'une minorité des règles possèdent une forte similarité syntaxique, excepté pour la catégorie *I/O* où 73,3 % des correspondances sont composées de fonctions ayant le même nom. Cependant, pour les autres catégories de migration, les résultats sont beaucoup plus faibles et ne dépassent pas 30% de correspondances détectées.

Correspondances à cardinalité multiple

Nous examinons les règles incluses dans C pour mesurer combien de fonctions sont associées à plus d'une fonction. Le but est de mieux comprendre si un tel scénario existe en pratique. Si ce n'est pas le cas, nous pourrions alors appliquer un algorithme glouton assurant la cardinalité 1 :1 des règles. Pour obtenir ce résultat, nous avons simplement compté combien de fonctions sont associées à une seule fonction (mono-associations) et combien le sont à plusieurs fonctions (multi-associations). Le Tableau 4.7 indique que le nombre de fonctions multi-associées est stable autour de 33 % pour toutes les catégories.

Tableau 4.7 : Distribution des fonctions mono-associées et multi-associées.

Catégorie	#Fonctions mono-associées	#Fonctions multi-associées
I/O	23 (69,7 %)	10 (30,3 %)
Lang	42 (66,7 %)	21 (33,3 %)
Json	33 (66 %)	17 (34 %)
Mock	20 (66 %)	10 (34 %)

Par conséquent, toute technique restrictive sur les correspondances 1 :1 diminuerait de façon non négligeable le rappel final.

4.4 Incertitudes sur la validité

Nous identifions deux risques pouvant affecter la validité de l'étude que nous avons menée. Tout d'abord, nous n'avons aucune donnée sur le rappel réel de notre approche pour les quatre couples de bibliothèques étudiées. Pour obtenir une telle mesure, il aurait été nécessaire de déterminer manuellement l'ensemble des correspondances pour chaque couple. Nous n'avons pas considéré cette tâche pour des raisons de complexité et de fort coût en temps. Ainsi, nous ne pouvons pas nous prononcer sur la quantité réelle de correspondances existant entre ces paires de bibliothèques.

Par la suite, le jugement des deux experts peut s'avérer incorrect. Pour remédier à ce problème, nous aurions pu envisager d'écrire du code client pour chaque bibliothèque impliquée dans les règles de migration sélectionnées. Puis, nous aurions nous-mêmes tenté d'effectuer une migration du code en utilisant les correspondances que nous avons extraites. Les correspondances validées seraient alors celles qui permettent d'atteindre les objectifs des fonctions précédemment utilisées.

4.5 Limites et travaux futurs

Malgré les résultats encourageants présentés dans ce chapitre, il est important de souligner les limites que nous avons relevées et qui permettent néanmoins d'ouvrir plusieurs perspectives.

Tout d'abord, notre technique analyse chacune des versions comprises dans le segment de migration pour générer des correspondances. Cependant, ce niveau d'analyse est potentiellement trop fin. Il se peut que des erreurs ou du code défectueux soient introduits par les développeurs entre deux versions, et par la suite corrigés. Dans une telle situation, il serait préférable de ne pas analyser les modifications du code source. Nous souhaiterions ainsi mesurer la qualité des règles obtenues en ne considérant que les deux versions délimitant un segment de migration. De plus, la technique de *diff* utilisée ici s'applique

à n'importe quel contenu textuel. Nous pourrions améliorer la qualité des hunks de migration produits en calculant la différence entre deux arbres de syntaxe abstraite avec une sémantique propre à chaque langage de programmation [Fluri *et al.*, 2007].

Nous aimerions également répliquer notre étude avec un échantillon plus large de bibliothèques. Les résultats de notre approche ne peuvent être généralisés pour le moment, puisque nous avons seulement étudié 4 couples de bibliothèques. Il serait donc judicieux de considérer un corpus plus étoffé pour un travail futur. Nous souhaiterions, dans la lignée des perspectives du chapitre précédent, déterminer une solution efficace pour identifier des projets ayant de fortes chances d'avoir migré. Un exemple de piste est d'utiliser des moteurs de recherche de code source tels que Ohloh Code [11] pour construire un corpus de projets clients d'une des deux bibliothèques impliquées dans une migration. Ceci augmenterait selon nous efficacement l'identification des projets sujets.

Nous profiterions d'une réplication pour mener une étude en collaboration avec des développeurs pour mesurer l'effort d'une migration de bibliothèque, avec ou sans nos correspondances. En plus de renforcer l'intérêt de nos travaux, cela nous permettrait d'adapter le format des rapports que nous produisons selon les besoins des développeurs.

Enfin, nous souhaiterions compléter notre approche en analysant des données supplémentaires telles que les documentations des deux bibliothèques impliquées dans une migration. En analysant textuellement les commentaires et descriptions associées à chaque fonction des API, nous pensons pouvoir raffiner les correspondances en y détectant des similitudes. Des techniques de traitement du langage naturel et de recherche d'informations peuvent être envisagées dans ce contexte.

11. http://code.ohloh.net/

CHAPITRE

5 Expertise des développeurs en bibliothèques

Ce dernier chapitre présente nos contributions sur l'identification de développeurs experts en bibliothèques tierces. Nous rappelons tout d'abord le problème étudié et synthétisons la solution proposée. Puis, nous introduisons notre modèle d'utilisation des symboles d'une bibliothèque par des développeurs, et détaillons comment les niveaux d'expertise sont mesurés, manipulés et comparés. Nous décrivons un procédé complet pour déployer notre approche pour le langage Java sur un système de contrôle de versions. Nous collectons par la suite une large base de données d'informations à partir d'un corpus de projets open source *de* GITHUB. *Nous proposons un prototype expérimental nommé* LIBTIC *permettant de manipuler les données générées, et nous illustrons ensuite son intérêt par plusieurs cas d'études. Nous présentons enfin les limites et les perspectives futures de ce travail.*

Sommaire

5.1 Introduction

Nous abordons dans cette section la problématique introduite en Section 1.2.3, relative à la recherche d'experts en bibliothèques tierces. Nous présentons une solution auto-

matique pour s'affranchir des contraintes d'une solution manuelle ou semi-automatique. Cela implique que les développeurs doivent être évalués uniquement sur leurs actions (principe de l'apôtre Saint-Thomas). Dans notre contexte, ces actions sont les contributions sur les systèmes de contrôle de versions qui impliquent l'utilisation de bibliothèques. Le concept d'expertise est ici rattaché à celui d'expérience, en supposant que l'expérience soit corrélée avec l'expertise réelle des développeurs. Notre hypothèse de travail stipule que plus un développeur utilise une bibliothèque, plus son niveau d'expertise sur cette bibliothèque est important.

Les contributions doivent être extraites à un niveau fin de granularité pour garantir la précision des résultats. De plus, l'utilisation d'une bibliothèque représente généralement une partie mineure du code source d'un logiciel client. Considérons par exemple le logiciel Apache HBase [1] qui utilise la bibliothèque Google *Guava*. HBase contenait, en mars 2013, 2 083 fichiers Java, parmi lesquels 229 contiennent des références à *Guava*. Il s'avère que l'utilisation de *Guava* représente 718 lignes de code sur un total de 113 408 lignes (soit 0,04 %) que contiennent ces 229 fichiers. Cet exemple conforte notre décision d'analyser finement les contributions des développeurs.

Nous proposons un procédé d'extraction des expériences de développeurs sur les bibliothèques, ainsi qu'un langage de requête dédié à la recherche d'experts selon plusieurs critères et différents niveaux d'informations. Un prototype nommé LibTic a été développé dans ce but.

5.2 Expérience des développeurs sur les bibliothèques

Nous débutons cette section par notre modèle d'utilisation des bibliothèques par les développeurs. Puis, nous décrivons comment rechercher des experts en présentant le concept de dimension de bibliothèques. Nous détaillons ensuite comment les experts peuvent être comparés pour produire un classement.

5.2.1 Utilisation des symboles des bibliothèques

Les termes et définitions employés dans ce chapitre sont à nouveau repris de la Section 3.2. Pour rappel, les développeurs contribuent au code source d'un projet par le biais de *commits* sur un système de contrôle de versions. Nous considérons qu'un développeur possède une expérience sur une bibliothèque à partir du moment où il introduit un des symboles de cette bibliothèque. Cette introduction est observable par l'analyse des *commits* de chaque développeur d'un logiciel.

Définition 5.1 (Introduction de symbole) *Soit D un ensemble de développeurs. Un développeur $d \in D$ utilise un symbole $s \in S_l$ pour une bibliothèque $l \in L$, si l'un des* commits

1. https://github.com/apache/hbase

dont il est l'auteur introduit s dans le code source. La version de la bibliothèque utilisée au moment du commit *est notée v. Nous notons ainsi (d, l, s, v) le 4-uplet définissant l'utilisation d'une bibliothèque par un développeur. Nous notons U l'union de toutes les utilisations de bibliothèques effectuées par tous les développeurs contribuant sur un ensemble P de projets logiciels.*

En guise d'exemple, considérons $L = \{\texttt{h2}\}$, `h2` étant une bibliothèque de base de données embarquée pour Java dont on suppose qu'elle contient trois symboles *load()*, *start()* et *stop()*. Alice est une développeuse ayant par deux fois contribué à un projet. Lors du premier *commit*, elle a introduit le symbole *load()* et la version 1 de `h2` était utilisée ce moment-là. Lors du second, elle a utilisé le symbole *start()* et la version 2 de `h2` était cette fois-ci présente. L'utilisation qui résulte de ce scénario équivaut à $U = \{(Alice, h2, load(), 1), (Alice, h2, start(), 2)\}$.

5.2.2 Mesure d'expérience et dimension de bibliothèque

Les développeurs les plus experts sur une bibliothèque donnée sont ceux ayant le plus haut niveau d'expérience sur cette bibliothèque. Néanmoins, nous pensons que la recherche d'experts ne doit pas se limiter à une seule bibliothèque et donc à un seul niveau d'information. Il est par exemple pertinent d'identifier un expert dans les bibliothèques `junit` et `testng` dans un contexte de migration. Nous pouvons également souhaiter faire appel à des experts de certaines versions d'une bibliothèque, voire d'un sous-ensemble de son API. En effet, une bibliothèque comme *Hadoop* est connue pour avoir subi des changements fondamentaux entre ses versions 1.x et 2.x. Un expert sur une version ne l'est donc pas nécessairement sur l'autre. Également, la bibliothèque `guava` propose une pluralité de services (entrées/sorties, calculs mathématiques, ...) et c'est pourquoi un expert de `guava` peut n'avoir aucune connaissance de certains de ses sous-domaines. Il s'agit donc de trouver quels développeurs ont une expérience sur un ensemble de bibliothèques tout en respectant pour chacune certaines contraintes.

Pour cette raison, nous proposons le concept de dimension de bibliothèque afin de pouvoir définir de telles contraintes sur les expériences recherchées. Une dimension de bibliothèque doit impérativement cibler une bibliothèque et éventuellement un sous-ensemble de symboles ou de versions.

Définition 5.2 (Dimension de bibliothèque) *Une dimension d'expérience de bibliothèque $dim = (l, dim_s, dim_v)$ vise une bibliothèque $l \in L$ et définit de façon facultative une fonction de filtre pour les symboles* $\dim_s : S_l \rightarrow \mathbb{B}$ *et une fonction de filtre pour les versions $dim_v : V_l \rightarrow \mathbb{B}$ de la bibliothèque. Nous notons Dim l'ensemble des dimensions exprimées dans une recherche d'experts.*

L'identification d'experts s'effectue selon au moins une dimension de bibliothèque, et cible des experts ayant utilisé les symboles engendrés par les dimensions. Sur l'exemple

précédent, nous pouvons définir une dimension qui vise la bibliothèque `h2` et qui considère tous les symboles et toutes les versions de cette bibliothèque. Grâce aux actions qu'elle a effectuées, Alice est retenue dans l'ensemble des experts généré à partir de cette dimension. Si nous ajoutons à cette dimension un filtre pour se restreindre à la version 2 de `h2`, seulement la deuxième contribution d'Alice est prise en compte.

Définition 5.3 (Utilisation des symboles) *Étant donnée une dimension de bibliothèque dim=(l,dim_s,dim_v) ∈ Dim, l'ensemble des symboles $s \in S_l$ utilisés par un développeur $d \in D$ correspond à :*

$$u(d, dim) = \{s \in S_l \mid \exists\, (d, l, s, v) \in U \wedge dim_s(s) \wedge dim_v(v)\}\ .$$

De plus, à chaque dimension de bibliothèque est associé un ensemble de symboles, dit *fournis*. Il s'agit de l'ensemble maximal de symboles couverts par une dimension de bibliothèque.

Définition 5.4 (Symboles fournis) *Pour une dimension de bibliothèque dim=(l,dim_s,dim_v) ∈ Dim, l'ensemble des symboles fournis correspond à :*

$$p(dim) = \{s \in S_l \mid dim_s(s) \wedge dim_v(v)\}\ .$$

Pour mesurer l'expérience d'un développeur d en regard d'une dimension de bibliothèque *dim*, nous calculons le ratio entre le nombre des symboles qu'il utilise, $u(d, dim)$, et le nombre de total de symboles fournis par la dimension $p(dim)$. Ceci produit une valeur comprise entre 0 et 1 qui représente le pourcentage d'utilisation des symboles utilisés parmi les symboles fournis, où 1 indique une utilisation complète et 0 une connaissance nulle.

Définition 5.5 (Expérience de bibliothèque) *Pour une dimension donnée dim=(l,dim_s,dim_v) ∈ Dim, un score d'expérience $e(d, dim)$ est attribué $\forall d \in D$, dont la valeur correspond à :*

$$e(d, dim) = \frac{|u(d, dim)|}{|p(dim)|}\ .$$

Pour déterminer le niveau de connaissance d'un développeur sur une bibliothèque, nous privilégions le nombre de symboles distincts utilisés plutôt que le nombre de fois où ceux-ci ont été utilisés. Nous supposons en effet qu'un développeur connaît un symbole à partir du moment où il l'introduit, indépendamment de la fréquence à laquelle il l'utilise. Nous mettons ainsi volontairement en avant la diversité des symboles et la couverture d'une bibliothèque. Les symboles d'une bibliothèque sont tous placés au même niveau, quelle que soit la nature du symbole ou de son rôle dans l'API.

Remarque. Contrairement aux travaux présentés dans l'état de l'art de la Section 2.3.2, nous n'utilisons donc ni le facteur temporel ni le facteur de fréquence.

En reprenant notre exemple, nous avons ainsi $e(Alice, h2) = 2/3$, puisque Alice utilise 2 des 3 symboles de `h2`. De façon similaire, si l'on contraint le filtre des versions à la version 2 de `h2`, nous obtenons $e(Alice, (h2, 2)) = 1/3$ car Alice n'a utilisé qu'un seul symbole de `h2` à la version 2, bien que le symbole utilisé à la version 1 soit également présent dans la version 2.

5.2.3 Distance entre expériences

Nous décrivons la procédure employée pour comparer des expériences de développeurs pour un ensemble de dimensions *Dim* donné. Nous définissons pour cela un vecteur de référence $\top$ correspondant à une expérience complète pour chacune des dimensions de bibliothèque requises ($\forall$ *dim* $\in$ *Dim*, *e*($\top$,*dim)*=1). Un vecteur de scores d'expérience est ainsi calculé pour chaque développeur impliqué dans la comparaison.

Pour reprendre notre exemple en fil rouge, considérons une seconde bibliothèque `slf4j` avec *e(Alice,slf4j)* = 1/2 et un autre développeur Bob tel que *e(Bob,h2)* = 1/3 et *e(Bob,slf4j)* = 1/4. En supposant deux dimensions ciblant les bibliothèques `h2` et `slf4j` sans aucun filtre, nous obtenons les trois vecteurs suivants :

$$v_\top = (1, 1) .$$

$$v_{Alice} = (2/3, 1/2) .$$

$$v_{Bob} = (1/3, 1/4) .$$

Nous calculons ensuite une distance euclidienne entre chaque vecteur de développeur et celui du vecteur de référence $\top$, chaque vecteur représentant les coordonnées d'un point dans un espace à n dimensions. La distance entre deux points p et q de coordonnées respectives $(p_1, p_2, \ldots, p_n)$ et $(q_1, q_2, \ldots, q_n)$ est déterminée par la formule suivante :

$$d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \ldots + (p_n - q_n)^2} .$$

Ainsi, cette formule appliquée aux deux vecteurs v_{Alice} et v_{Bob} permet d'obtenir les distances euclidiennes suivantes :

$$d(v_{Alice}, v_\top) = \sqrt{(2/3 - 1)^2 + (1/2 - 1)^2} = \sqrt{0.11 + 0.25} = 0.6 .$$

$$d(v_{Bob}, v_\top) = \sqrt{(1/3 - 1)^2 + (1/4 - 1)^2} = \sqrt{0.44 + 0.56} = 1 .$$

Finalement, ces distances sont classées par ordre croissant, puisqu'une distance proche de zéro révèle une proximité avec le point de référence $\top$ et donc une expérience significative. Ainsi, Alice possède un niveau d'expérience supérieur à celui de Bob dans le domaine des dimensions spécifiées.

5.3 Implémentation avec LIBTIC

Nous proposons dans cette section une implémentation de notre approche sous forme d'un prototype nommé LIBTIC. Nous identifions deux sous-composants distincts à mettre en place pour y parvenir, LIBTIC-DUD (*developer usage database*) et LIBTIC-QUERY. Le LIBTIC-DUD a pour objectif de construire la base de données des utilisations des bibliothèques faites par les développeurs (tous les $u \in U$, comme défini en Section 5.2.1). Le LIBTIC-QUERY agit pour sa part comme moteur de requête pour l'identification de développeurs experts à partir de spécifications de dimensions de bibliothèque.

5.3.1 Extraction des utilisations de bibliothèques

La construction du LIBTIC-DUD nécessite deux sous-parties : un index de symboles qui recense les symboles des bibliothèques pour chacune de leurs versions et associe un logiciel aux versions des bibliothèques utilisées, puis un extracteur d'utilisation qui collecte les utilisations des bibliothèques.

Index des symboles

Le but de cet index est d'identifier, à partir d'un symbole donné, quelle version de quelle bibliothèque. Il est donc perçu comme une fonction qui retourne soit un couple formé de la bibliothèque et de la version associé, soit l'ensemble vide si un tel couple n'a pu être identifié. Pour construire un tel index, nous avons cherché à former une population de bibliothèques versionnées sous forme de fichiers JAR, afin d'analyser a posteriori leur contenu.

Dans cet objectif, nous avons recherché des projets hébergés sur GITHUB écrits en Java et configurés avec Apache MAVEN. Nous avons choisi MAVEN dans un souci de simplicité, mais le procédé que nous présentons s'applique à des systèmes similaires comme GRADLE ou Apache IVY. Nous avons ainsi constitué un ensemble de 6 330 projets. Pour chacun d'eux, nous avons utilisé certaines fonctionnalités proposées par MAVEN pour récupérer l'ensemble des bibliothèques dont ils dépendent (comprenant le fichier JAR physique et le numéro de version). 50 heures ont été nécessaires pour accomplir cette opération et obtenir 8 795 bibliothèques versionnées.

Remarque. Contrairement au cas d'étude de la Section 3.4.1, les données sur les versions ne sont pas effacées. Les bibliothèques *junit-3.8.1* et *junit-4.8.1* sont donc perçues ici comme deux entités distinctes.

Ces fichiers JAR ont ensuite été analysés pour en extraire les symboles ayant une visibilité `public` indiquant leur présence dans l'API de la bibliothèque. Nous avons utilisé la bibliothèque `javassist` pour cette tâche [Chiba, 1998], qui produit un ensemble de triplets (s, l, v) indiquant que le symbole s est fourni par la bibliothèque l à la version v. La

seconde étape vise à transformer cet ensemble en un index où les symboles sont les clés et les valeurs sont des couples (l, v).

Cependant, un symbole est souvent présent dans plusieurs versions successives d'une bibliothèque puisque celles-ci introduisent généralement des modifications mineures. Il est donc difficile en pratique d'affecter un symbole à un seul couple de bibliothèque et de version. Pour faire face à ce problème, nous utilisons l'information $deps_p$ qui contient tous les couples (l, v) qui sont utilisés par un projet p. Cet ensemble est donc une extension du dep_p que nous utilisions jusqu'à présent. Par conséquent, les clés de l'index de symboles sont désormais des couples de la forme $(s,deps_p)$. Cet index ne retient ensuite que les triplets (s, l, v) où $(l, v) \in deps_p$. Au final, un unique couple (l, v) est retourné si et seulement si il n'y a qu'un seul couple identifié par l'index. Dans le cas où aucun ou plusieurs couples sont identifiés, alors l'ensemble vide est retourné. Dans notre cas, l'information sur les versions des bibliothèques utilisées est donnée par des fonctionnalités de MAVEN.

Remarque. Dans un travail futur, nous pourrions reprendre des approches de résolution automatique des dépendances pour les projets *open source* comme celles proposées récemment par Ossher et al. [Ossher *et al.*, 2010]. De plus, contrairement au cas d'étude de la Section 3.4.1, nous ne prêtons pas attention à la résolution de conflits entre symboles appartenant à plusieurs bibliothèques.

Extracteur d'utilisation

Le processus d'extraction de LIBTIC-DUD suppose tout d'abord que pour une quelconque révision d'un projet, la liste des bibliothèques qu'il utilise est accessible. Notons que dans les cas où les versions sont indisponibles, nous utilisons un *wildcard*. Nous avons défini un prototype basé sur HARMONY afin de mettre en place cet extracteur. Pour un logiciel donné, nous analysons chacune de ses versions à partir de son dépôt Git associé. Sur chaque *changeset*, nous conservons les fichiers indiqués comme étant modifiés ou ajoutés pour qu'ils soient ensuite analysés. Les fichiers supprimés sont mis de côté, puisqu'ils ne peuvent par définition pas introduire de symboles.

Remarque. Nous avons volontairement fait le choix de ne pas considérer les suppressions de symboles comme des marqueurs d'expérience. Ce choix peut être discuté, puisque nous pouvons supposer qu'un développeur ne supprimera des symboles que s'il en a connaissance.

Une utilisation de symbole n'est ensuite validée que si elle fait partie des contributions d'un développeur lors de l'édition d'un fichier. Chaque fichier de code source modifié ou ajouté est analysé dans le but de produire un arbre de syntaxe abstraite dont chaque nœud possède un type, une valeur et une position dans le fichier. Pour cette opération, nous avons fait appel à l'outil JDT proposé par ECLIPSE. Nous collectons ainsi chaque nœud correspondant à l'utilisation d'un symbole quelconque. Un ensemble SP de couples (s, lp)

extraits à partir du fichier courant contient la liste des symboles s et des numéros de lignes lp où ils sont utilisés.

Dans un deuxième temps, un *diff* textuel est calculé entre le fichier courant et sa version antérieure. Puisqu'un fichier ajouté ne possède pas de version parente, un *diff* virtuel est effectué avec un fichier vide. Le résultat produit la nouvelle version du fichier où chaque ligne ajoutée est repérée par un '+' ainsi que par sa position dans le fichier. L'étape finale consiste à filtrer le contenu de SP pour ne conserver que les symboles situés dans ces lignes nouvellement ajoutées.

Pour mieux comprendre notre méthode, considérons l'exemple exposé en Figure 5.1, dans lequel un développeur a modifié le fichier Foo.java pour y ajouter une référence vers la méthode *assertFalse(boolean)* de l'API de `junit`. À partir du *diff*, nous observons que seule la ligne 7 est nouvelle dans la version v1 du fichier. L'extracteur d'utilisation détecte ici que deux symboles sont utilisés, dont un à la ligne 7. Ainsi, l'auteur de cette nouvelle version augmente son expérience sur `junit` grâce à l'introduction de la méthode *assertFalse(boolean)* .

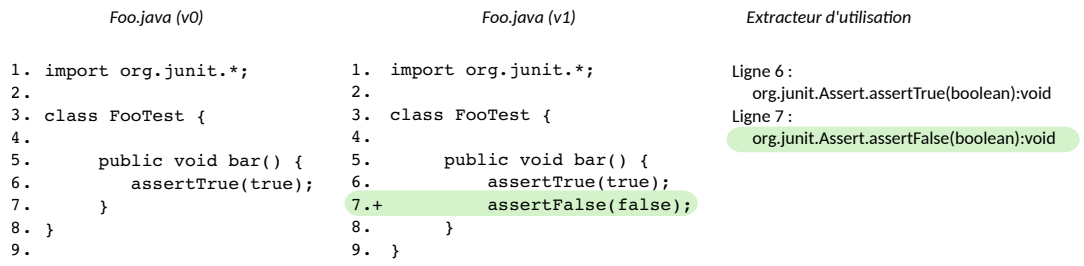


FIGURE 5.1 : Extracteur d'utilisation des bibliothèques en action. Ici, un nouveau symbole de la bibliothèque `junit` a été ajouté.

Pour terminer, l'index des symboles est utilisé pour déterminer la bibliothèque et la version associées à chacun de ces nouveaux symboles. L'auteur de la révision du projet en cours d'analyse est également extrait. La base de connaissance de LIBTIC-DUD est ensuite mise à jour à chaque nouvelle correspondance identifiée. Notre procédé a l'avantage d'être incrémental et peut s'appliquer sur un ensemble de révisions récentes d'un projet pour en mettre à jour les informations.

Déploiement

La construction du LIBTIC-DUD a été réalisée à partir du corpus de 6 330 projets collectés précédemment. Comme les développeurs sont libres de valider des *commits* sous l'identité qu'ils souhaitent, il est possible qu'un même développeur apparaisse en fait sous plusieurs noms. Pour réduire cette fragmentation des développeurs, nous avons utilisé une technique simple de fusion des identités. Nous avons recherché le nom de l'auteur

du *commit* sur GITHUB via l'API REST disponible [2] et, dans le cas où un unique utilisateur est retourné nous extrayons son login. Dans la situation opposée, le nom de l'auteur est conservé. Nous discuterons de cet aspect en Section 5.6.

150 heures de calcul ont été nécessaires pour peupler le LIBTIC-DUD qui est finalement composé de 3 705 développeurs, 1 026 bibliothèques, 51 585 symboles et 161 917 utilisations de symboles.

Remarque. L'index initial des symboles n'apparaît pas dans ces chiffres, qui ne concernent que les bibliothèques utilisées au moins une fois.

Un aperçu des résultats nous indique que 77 % des développeurs connaissent entre 1 et 5 bibliothèques, et 2 % d'entre eux utilisent plus de 20 bibliothèques. Nous évaluerons plus en détail ce jeu de données en Section 5.4.

5.3.2 Recherche d'experts avec LIBTIC

Analyser efficacement et facilement les données collectées représente notre prochain défi. LIBTIC-DUD stocke ses informations dans une base de données relationnelle, mais nous pensons que l'utilisation du langage SQL conduit à des écritures de requêtes lourdes, difficiles à rédiger et où des erreurs surviennent rapidement. Afin de manipuler nos données, nous avons conçu un prototype LIBTIC-QUERY ainsi qu'un langage de domaine spécifique (*Domain Specific Language, DSL*) très léger au-dessus de SQL. Ce DSL possède peu de constructions de langage mais est suffisant pour couvrir les applications de LIBTIC-QUERY décrites en Section 5.4. Nous présentons un aperçu de ce langage dans cette sous-section.

who, who*, how et how*. Notre principal opérateur est le `who`, qui requiert impérativement un ensemble de contraintes de bibliothèques représentant chacune une dimension de bibliothèque. Cet opérateur ne considère que les développeurs avec une expérience strictement supérieure à 0 sur chacune des dimensions. Une contrainte de bibliothèque est exprimée par un nom de bibliothèque, et éventuellement un filtre sur les versions et les symboles. Un filtre sur les symboles est un ensemble d'expressions régulières simplifiées où nous retenons chaque symbole identifié par au moins une expression. Un filtre sur les versions représente quant à lui des contraintes arithmétiques. Par exemple, la requête :

```
who guava{*.html.* *.io.*} guava{*.math.*} guava >4
```

identifie les développeurs utilisant au moins la version 4 de `guava`, des symboles dans ses paquetages `io` ou `html`, et des symboles du paquetage `math`.

L'opérateur `who` peut recevoir en complément des contraintes sur les développeurs. Ainsi, la requête :

2. http://developer.github.com/v3/

```
who {alice bob*} guava{*.math.*}
```

retourne les développeurs dont le nom est *alice* ou commence par *bob*, et connaissant au moins un symbole du paquetage `math` de la bibliothèque `guava`.

Étant donné que l'opérateur `who` est restrictif, nous proposons en complément `who*`, qui permet de relâcher certaines contraintes. Cet opérateur va sélectionner un développeur s'il possède une expérience strictement supérieure à 0 dans au moins une des dimensions exprimées par la requête accompagnant le `who*` (à l'inverse de toutes pour l'opérateur `who`).

Enfin, il est intéressant de connaître la distance des développeurs en vertu de chacune des dimensions plutôt que la distance globale vis à vis de l'ensemble des domaines *Dim* de la requête. C'est pour cette raison que nous avons défini les opérateurs `how` et `how*` pour considérer chaque dimension de bibliothèque de façon indépendante. La syntaxe de ces opérateurs est similaire aux deux précédentes.

desc et find. Pour permettre de rechercher des détails sur les bibliothèques, symboles, ou développeurs, nous mettons à disposition les opérateurs `desc` et `find`. `desc` est utilisé pour consulter les détails d'un développeur particulier. L'ensemble des symboles utilisés ainsi que les bibliothèques et versions associées sont alors calculés. Les résultats peuvent être restreints en spécifiant des contraintes de bibliothèques, avec une syntaxe similaire aux opérations `who` et `how`. Le rôle de `find` est quant à lui de déterminer le nom exact de bibliothèques, de symboles ou de développeurs à partir d'une chaîne de caractère.

set. Pour terminer, nous proposons un mécanisme simple de variables pour manipuler des contraintes complexes ou répétitives. Une variable peut représenter une contrainte (`@var`) ou un ensemble de symboles ou de développeurs (`$var`). Ces deux types sont stockés dans des espaces de nommage différents et peuvent contenir des *wildcards* qui seront étendus dans leur contexte respectif. Par exemple, l'expression :

```
set $o {*o*} ; set @g guava ; who $o @g $o
```

retourne uniquement les développeurs dont le nom contient le caractère 'o' et connaissant un symbole de `guava` contenant également un caractère 'o'.

Pour terminer, nous présentons en Figure 5.2 un aperçu de l'interface utilisateur de LIBTIC-QUERY.

5.4 Évaluation et cas d'utilisation

Nous étudions dans cette section la validité des résultats que nous produisons, en montrant que nous sommes capables d'identifier de vrais experts. Par la suite, nous proposons

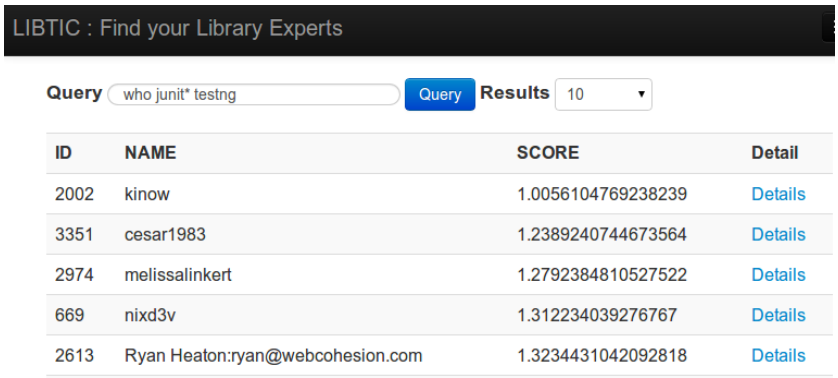


FIGURE 5.2 : Recherche d'experts sur les bibliothèques `junit` et `testng` à partir de LIBTIC.

des cas d'utilisation de LIBTIC dans le cadre d'un unique projet et dans un contexte de maintenance logicielle.

Contexte n°1 : Recherche de développeurs experts

Nous voulons ici montrer que LIBTIC identifie des experts en bibliothèques à partir d'une grande population de développeurs. Nous avons focalisé notre attention sur deux situations : (1) trouver des experts d'une bibliothèque ; (2) trouver des experts de plusieurs bibliothèques.

Pour le premier cas, nous avons sélectionné trois bibliothèques (`guava`, `servlet-api` et `jackson-core-asl`) utilisées par au moins 20 projets de notre corpus. Pour chacune, les deux meilleurs experts ont été identifiés grâce à LIBTIC. Nous avons ensuite enquêté sur leur profil pour vérifier nos informations. Nous nous sommes adressés à chacun d'eux par courriel. Nous leur avons demandé de s'auto-évaluer sur 4 bibliothèques choisies au hasard ainsi que sur celle qui nous intéresse, en s'attribuant une note allant de 0 (aucune connaissance de la bibliothèque) à 5 (très forte connaissance de la bibliothèque).

Pour identifier des experts de plusieurs bibliothèques, nous avons appliqué le même processus, si ce n'est que nous avons arbitrairement inclus les bibliothèques `junit` et `testng`.

Nous avons obtenu un taux de réponse de 50 % à partir des questionnaires envoyés. Nous masquons volontairement les noms des développeurs, par respect du secret des données personnelles. L'expert le mieux classé pour la bibliothèque `guava` est *guava-expert-1*, avec 55 symboles utilisés, le second étant *guava-expert-2* avec 41 symboles utilisés. Nous avons analysé le profil de *guava-expert-1* sur GITHUB, et il s'avère que ce développeur contribue seul sur 3 projets Java utilisant la bibliothèque `guava`. Ceci est donc de bon augure pour nos résultats. Le profil de *guava-expert-2* nous a amené à constater que cette personne utilisait également la bibliothèque dans ses projets personnels. Seul le premier

développeur a répondu à notre courriel et s'est attribué une note de 4/5, la note maximale de la liste que nous lui avons fournie.

Pour la bibliothèque `servlet-api`, nous avons identifié *servlet-expert-1* et *servlet-expert-2* ayant utilisé respectivement 106 et 72 symboles de cette bibliothèque. En regardant le profil de *servlet-expert-1*, nous avons remarqué qu'il est le créateur de *Jenkins*, un serveur d'intégration continue écrit en Java et définissant à de nombreuses reprises des servlets. Pour le second expert, sa page personnelle mentionne qu'il est « *Web Architecture Consultant specialized in open source frameworks* ». Il est de plus actif dans de nombreux projets Java orientés web, et les servlets sont justement des composants classiques pour des programmes Web en Java. Seul *servlet-expert-2* a répondu à notre questionnaire, et s'est attribué la note maximale de 5/5.

Pour la bibliothèque `jackson-core-asl` manipulant des données JSON, nous avons identifié *jackson-expert-1* et *jackson-expert-2* ayant utilisé respectivement 68 symboles et 31 symboles de cette bibliothèque. Contrairement au premier, le second utilise des versions plus récentes de la bibliothèque allant de 1.82 à 1.9 au lieu de 1.1.1 à 1.5 pour *jackson-expert-1*. La page personnelle de *jackson-expert-1* indique qu'il est un développeur actif de Jersey, une plateforme Java utilisée pour la construction de services REST. JSON étant le principal format de sortie des services REST, il est probable que ce développeur ait une bonne connaissance de cette bibliothèque. Nous avons pu constater que *jackson-expert-2* maintient un outil en ligne de validation de documents JSON, qui repose justement sur la bibliothèque `jackson`. *jackson-expert-1* n'a pas répondu à notre courriel, alors que *jackson-expert-2* s'est attribué une note de 4/5, la note maximale de la liste que nous lui avons fournie.

Nous avons identifié 25 développeurs ayant connaissance à la fois des bibliothèques `junit` et `testng`. *test-expert-1* est le mieux classé et utilise respectivement 26 et 85 symboles de `junit` et `testng`. Le second développeur, *test-expert-2*, a pour sa part connaissance de 14 et 31 symboles parmi ces deux bibliothèques. La page personnelle du premier développeur indique qu'il a exercé en tant que "*software quality engineer (writing tests, executing, creating automated tests, preparing CI environments, analyzing source code and writing tools to assist developers and testers)*". Il s'est attribué 3/5 et 3/5 pour ces bibliothèques, les scores les plus hauts par rapport à la liste fournie. En revanche, *test-expert-2* n'a pas répondu à notre courriel, et aucune information supplémentaire n'a pu être trouvée le concernant.

Nous pouvons donc constater que les données collectées par LIBTIC semblent valides et permettent d'identifier de vrais experts en bibliothèques. Cela soutient également l'idée que l'analyse par delta des fichiers de code source est une approximation fiable des contributions réelles des développeurs.

LISTING 5.1 : Variables utilisées par le cas d'étude n°2 de la Section 5.4.

```
1 set $hbase_devs {hbase-dev-1 ... hbase-dev-21} ; # HBase developers
2
3 set @hbase_libs jaxb-api hadoop-core high-scale-lib jersey-core mockito-all libthrift
     jersey-server jackson-mapper-asl commons-lang stax-api avro protobuf-java
    jersey-json jetty commons-cli junit jsr311-api slf4j-api metrics-core
    servlet-api-2.5 jsp-2.1 commons-logging guava htrace thrift hadoop-test zookeeper
     log4j json ; # HBase libraries
```

Contexte n°2 : Utilisation de LIBTIC dans un projet logiciel

Nous proposons ici deux cas d'études pour affirmer l'utilité de LIBTIC dans le cadre d'un projet logiciel. Nous avons sélectionné le projet Apache HBase [3], car il fait partie des projets les plus actifs de notre corpus initial. Nous y avons recensé 33 développeurs ayant effectué au moins un *commit* sur le dépôt de code source. Nous avons totalisé un ensemble de 29 bibliothèques utilisées par ce projet. Dans le premier cas, nous utilisons LIBTIC pour produire un bilan de connaissance des bibliothèques pour chaque développeur. Dans le second cas, nous avons utilisé LIBTIC comme système de recommandation de tâches pour la résolution d'erreurs impliquant la bibliothèque `guava` et extraites depuis le gestionnaire d'erreurs de HBase. Les deux requêtes disponibles dans le Listing 5.1 définissent les deux variables contenant respectivement les développeurs de HBase et les bibliothèques utilisées.

Bilan des connaissances des bibliothèques. Nous pensons qu'une vue globale des expertises de bibliothèque de tous les développeurs est bénéfique pour la gestion d'un projet. Un développeur expérimenté sur de nombreuses bibliothèques est une ressource vitale pour le projet et son départ pénalisera fortement l'équipe de développement. Aussi, nous pensons qu'il est dangereux d'avoir des bibliothèques pour lesquelles il y a très peu d'experts dans l'équipe. Dans ce cas, il serait judicieux de former des développeurs supplémentaires sur cette bibliothèque.

À partir de LIBTIC nous avons pu calculer ce que nous appelons une matrice d'expertise de bibliothèque, qui indique quels développeurs connaissent chaque bibliothèque. Ces résultats sont obtenus en exécutant la requête suivante :

```
how* $hbase_devs @hbase_libs.
```

Cette matrice, présentée dans le Tableau 5.1, révèle en premier lieu que 21 développeurs parmi les 33 recensés ont connaissance d'au moins une bibliothèque. Nous observons que chaque bibliothèque possède un nombre différent d'experts. Par exemple, 8 bibliothèques (marquées en gras) sont seulement connues par un développeur. Dans un tel

3. https://github.com/apache/hbase

Tableau 5.1 : Expertises de bibliothèques pour les développeurs de `hbase`. Les lignes représentent les bibliothèques et les colonnes les développeurs. Les bibliothèques en gras ne sont connues que par un seul développeur. Les développeurs en gras utilisent plus de la moitié des bibliothèques.

	hbase-dev-1	hbase-dev-2	hbase-dev-3	hbase-dev-4	hbase-dev-5	hbase-dev-6	hbase-dev-7	hbase-dev-8	hbase-dev-9	hbase-dev-10	hbase-dev-11	hbase-dev-12	hbase-dev-13	hbase-dev-14	hbase-dev-15	hbase-dev-16	hbase-dev-17	hbase-dev-18	**hbase-dev-19**	hbase-dev-20	**hbase-dev-21**
avro																			•		
cms-cli	•																		•	•	•
cms-lang														•							
cms-logging	•	•		•	•	•	•		•	•	•		•	•	•		•		•	•	•
guava	•				•			•					•	•			•	•	•	•	•
hadoop-core	•		•	•	•	•		•			•	•	•	•					•	•	•
hadoop-test	•																				
high-scale-lib											•										
htrace																					•
jackson-mapper																			•		•
jaxb-api	•																		•		•
jersey-core																			•		•
jersey-json	•																		•		•
jersey-server	•																		•		•
jetty	•																				
json	•																				
jsp-2.1																					•
jsr311-api																			•		•
junit	•				•			•		•	•	•	•	•		•	•	•	•		•
libthrift																			•		•
log4j						•													•		
metrics-core																			•		•
mockito-all							•												•		•
protobuf-java	•										•								•		•
servlet-api-2.5	•																		•		•
slf4j-api																			•		•
stax-api	•																		•		•
thrift																			•		•
zookeeper	•																				

scénario, nous qualifions ces personnes de critiques. Nous observons que 3 développeurs (marqués en gras) sont experts de plus la moitié des bibliothèques, et sont donc des personnes clés pour ces projets, du point de vue des bibliothèques.

Remarque. Il est intéressant, au vu de cette matrice, de retrouver la tendance des *Heroes* dans les projets *open source*, puisqu'une minorité de développeurs clés maîtrise la majorité des bibliothèques [Ricca et Marchetto, 2010]. Également, cette matrice peut être améliorée pour un travail futur, pour indiquer les scores d'expérience plutôt que des valeurs binaires, comme c'est le cas dans cette version.

Affectation de tâches. Nous pensons que LIBTIC peut servir à affecter des développeurs experts pour la résolution d'une tâche liée à une bibliothèque. Dans cet objectif, nous

avons d'abord recensé dans le système de gestion d'erreurs de HBase les entrées dont le titre, la description ou les commentaires contenaient le mot-clé `guava`. Nous avons choisi cette bibliothèque car elle est fortement utilisée dans ce projet. Nous nous sommes assurés, après vérification manuelle, que ces erreurs étaient effectivement en lien avec la bibliothèque. Ce procédé nous a permis de rassembler un total de 11 rapports d'erreurs. Nous avons sélectionné la date de chaque erreur et avons exécuté la requête suivante sur la base de connaissances de LIBTIC-DUD entre la date de création du projet et la date de création de chaque erreur :

```
how* $hbase_devs guava.
```

Cela nous permet de simuler un système de recommandation en respectant la cohérence des données liées à la position dans le temps de chaque erreur. Par conséquent, le nombre d'experts de la bibliothèque `guava` augmente avec le temps, atteignant 10 experts à la date de la dernière erreur recensée.

Ensuite, nous avons collecté manuellement la liste des personnes impliquées dans le processus de résolution de chaque erreur. Les développeurs ayant simplement rapporté l'erreur ou bien modifié son statut n'ont pas été retenus. Ceux qui ne figuraient pas parmi notre liste initiale de développeurs ont été également écartés (ce fut le cas pour *HBase-QA* et *Hudson*). Au final, nous avons partitionné chaque ensemble de développeurs en deux parties : les experts EX (ceux identifiés comme ayant utilisé `guava`), et les non-experts NEX (ceux n'ayant aucune connaissance de cette bibliothèque). Nous avons ainsi pu calculer la précision et le rappel d'un système de recommendation qui aurait suggéré des candidats à la résolution de ces erreurs au moment où elles ont été rapportées. Une précision de 1 signifie que tous les experts identifiés ont participé, tandis qu'une valeur de 0,5 indique que seulement la moitié des experts ont participé. De façon similaire, un rappel de 1 signifie que nous avons au moins trouvé tous les participants, alors qu'une valeur de 0,5 indique que nous avons détecté seulement la moitié des participants. Les résultats sont disponibles dans le Tableau 5.2.

Nous observons que le rappel d'un tel système augmente avec le temps. Pour les premières erreurs, la précision est satisfaisante mais le rappel est faible. La situation s'inverse pour les erreurs plus récentes, ce qui est intéressant dans un tel contexte : il est préférable de notifier le plus de vrais experts pour une erreur donnée. Nous obtenons un rappel de 1 sur les 4 erreurs les plus récentes, avec néanmoins une précision inférieure ou égale à 0,5. De plus, il est possible que d'autres experts auraient participé à la résolution de l'erreur s'ils en avaient été notifiés, ce qui aurait amélioré la précision de nos résultats. En effet, parmi les experts de `guava` connus au moment de l'erreur, nous n'avons pas vérifié si ceux-ci avaient pour habitude d'intervenir dans le système de gestion d'erreurs du projet. Ainsi, les résultats obtenus sont encourageants dans la perspective d'approfondir cet aspect affectation de tâches liées aux bibliothèques.

Tableau 5.2 : Classification des développeurs impliqués dans la résolution des erreurs liées à la bibliothèque `guava` et précision/rappel d'un système de recommendation qui aurait notifié les erreurs à tous les experts de `guava`. #Experts indique le nombre d'experts que LIBTIC a identifié pour l'erreur. #EX et #NEX indiquent respectivement le nombre d'experts et de non-experts qui ont participé à la résolution de l'erreur.

N° erreur	Date	#Experts	#EX	#NEX	Précision	Rappel
2714	11/06/10	1	1	2	1	0,33
2724	14/07/10	1	1	2	1	0,33
3264	23/11/10	4	2	2	0,5	0,5
3609	07/03/11	6	1	1	0,17	0,5
3952	04/06/11	6	2	1	0,33	0,67
4012	21/06/11	6	2	2	0,33	0,5
4385	13/09/11	7	1	1	0,14	0,5
4569	10/10/11	7	2	0	0,29	1
5739	06/04/12	9	3	0	0,33	1
5955	08/05/12	9	4	0	0,44	1
6368	10/07/12	10	5	0	0,5	1

5.5 Incertitudes sur la validité

Nous identifions trois principaux risques pesant sur la validité de nos résultats. Premièrement, l'opération de *diff* utilisée pour déterminer les contributions des développeurs n'est pas robuste dans certaines situations, comme évoqué dans le chapitre précédant. En effet, les déplacements, les « copiés/collés » de code ou encore les renommages de fichiers sont perçus comme des éléments nouvellement ajoutés. Dans de telles situations, le LIBTIC-DUD est peuplé par des introductions erronées de symboles.

Deuxièmement, la limite de notre analyse par observation est qu'elle ne fournit aucune donnée qualitative de l'utilisation de la bibliothèque. Il existe potentiellement des experts selon notre définition qui, pour autant, utilisent de façon incorrecte une API et dont les contributions ne devraient pas servir d'exemple.

Enfin, lorsqu'un développeur valide un *commit* sur un système de contrôle de version, libre à lui d'indiquer l'identité qu'il souhaite. Ainsi, il n'est pas rare qu'un développeur contribue sur le même projet sous plusieurs noms. Ce phénomène peut avoir une forte influence sur la qualité des résultats obtenus. Dans le cas d'étude sur LIBTIC, nous avons ici utilisé une technique sommaire pour fusionner des profils d'experts. Nous recommandons donc d'utiliser des techniques plus avancées sur ce problème, comme celles rassemblées par Goeminne et al. [Goeminne et Mens, 2011].

5.6 Limites et travaux futurs

Nous avons présenté dans ce chapitre nos contributions sur l'identification de développeurs experts sur les bibliothèques tierces. Ce travail ouvre des perspectives futures permettant notamment de pallier les limites actuelles que nous avons rencontrées.

Nous identifions tout d'abord plusieurs points qui permettraient d'améliorer la version actuelle de LIBTIC. Premièrement, notre langage pourrait être étendu afin d'offrir des fonctionnalités supplémentaires, comme de pouvoir comparer l'expérience de deux développeurs sur une même bibliothèque en termes de recouvrement. En effet, il se peut que ces deux personnes aient le même niveau d'expérience mais que les deux ensembles de symboles connus se recouvrent, ou à l'inverse soient disjoints.

Ensuite, concernant l'extraction des expériences (et donc les ajouts de symboles), nous souhaitons considérer des techniques plus avancées que celle du *diff*, comme la différence structurelle d'arbres de syntaxe abstraite [Fluri *et al.*, 2007].

Troisièmement, il arrive que certains logiciels, pour dépendre de bibliothèques, incluent celles-ci non pas sous leur format de distribution binaire, mais comme un répertoire contenant le code source complet de la bibliothèque. Dans un tel scénario nous détecterions que l'auteur du *commit* est expert de tous les symboles de la bibliothèque utilisée dans le code interne de la bibliothèque. Des heuristiques pourraient être mises en place afin de déterminer l'origine extérieure du code [Davies *et al.*, 2011] ou détecter si ce code n'est pas cloné d'un code existant [Hummel *et al.*, 2010].

Ensuite, une limite de notre approche est que nous ne considérons pas l'aspect récent de l'utilisation des symboles. Ainsi, nous sommes potentiellement capables d'identifier des experts d'une bibliothèque qui en fait ne l'ont pas utilisée pendant plusieurs mois, voire des années. Il serait donc judicieux de s'inspirer de travaux existants pour raffiner notre définition d'expertise en prenant en compte le facteur temporel.

Pour finir, il serait judicieux d'attribuer des poids aux symboles. En effet, l'existence de points froids et de points chauds dans les API de bibliothèques (voir Section 2.2.1) suggère que nous pourrions donner une importance supérieure aux symboles les moins fréquemment utilisés, indiquant une connaissance plus approfondie de la bibliothèque.

De plus, la gestion des versions dans la mesure des expériences peut être améliorée. Considérons un développeur qui a utilisé 20 symboles de la version 1 d'une bibliothèque puis que celle-ci est mise à jour à la version 2. Si aucun symbole n'est ajouté par la suite, le développeur ne possède pas d'expérience sur la version 2 de la bibliothèque, ce qui s'avère erroné en pratique. En effet, la mise à niveau d'une bibliothèque n'entraîne pas nécessairement des changements dans le code source du logiciel client.

Concernant les perspectives plus lointaines, nous identifions trois principaux points. Tout d'abord, nous souhaiterions étendre la définition d'expertise de bibliothèque. À l'heure actuelle, nous sommes restreints à l'utilisation de son interface de programmation. Nous pourrions aller plus loin en proposant des motifs d'utilisation d'une bibliothèque caractérisant certaines de ses fonctionnalités. Cela pourrait se traduire par un enchaînement

ordonné d'utilisation des symboles. Le défi serait donc de pouvoir spécifier de tels motifs qui seraient ensuite recherchés dans les contributions des développeurs.

Notre second objectif prioritaire pour un travail futur est d'étendre LIBTIC à des domaines d'expertise autres que les bibliothèques tierces. Nous pourrions considérer les constructions des langages de programmation et déterminer quels développeurs les utilisent (par exemple, les *Generics* en Java). Notre but est d'augmenter la couverture des compétences de développeurs logiciels pouvant être mesurées automatiquement. Il n'existe pas selon nous de solution offrant une généricité dans la recherche d'expertise de développeurs.

Pour terminer, notre hypothèse de travail est que l'expertise d'un développeur est liée à son niveau d'expérience sur une bibliothèque. Cependant, nous ne pouvons garantir ni le réel niveau d'expertise d'un développeur ni la qualité du travail qu'il produit avec la bibliothèque. Nous aimerions donc mettre en place une étude empirique permettant de valider l'hypothèse qu'un fort niveau d'expérience d'un développeur sur une compétence donnée (non nécessairement liée à une bibliothèque) est révélateur d'un réel niveau d'expertise sur celle-ci.

CHAPITRE

6 Conclusion

6.1 Résumé des contributions

La réutilisation de code est une pratique courante dans le développement logiciel. Les bibliothèques logicielles permettent d'appliquer ce principe en proposant à un développeur une interface de programmation encapsulant une implémentation robuste de fonctionnalités. Une bibliothèque est qualifiée de tierce lorsqu'elle provient d'une organisation extérieure au logiciel client qui l'utilise. Nous affirmons dans cette thèse que la majorité des logiciels dépendent aujourd'hui de telles bibliothèques pour tirer parti de code existant. Cette forte dépendance oblige les logiciels clients à considérer les bibliothèques comme des entités majeures.

Notre contexte de recherche se focalise sur le sort des bibliothèques durant la phase de maintenance logicielle. Nous illustrons par différents scénarios leur implication pendant cette phase. Nous soulignons en particulier que les bibliothèques sont amenées à évoluer, soit via une mise à niveau, soit via une migration lorsqu'un remplacement est nécessaire. Les problématiques que nous abordons dans cette thèse sont précisément extraites de ce contexte de migration de bibliothèque. Nos contributions portent sur les trois points suivants :

— Le « Quoi ? » : vers quelles bibliothèques tierces un développeur peut-il migrer ?

— Le « Comment ? » : de quelle manière doit-il mettre à jour son logiciel pour devenir compatible avec la nouvelle bibliothèque ?

— Le « Qui ? » : quel expert de la nouvelle bibliothèque peut lui apporter une aide ?

La méthodologie commune pour répondre à ces trois problèmes repose sur l'analyse des changements d'un logiciel. Des études à large échelle sur plusieurs milliers de projets ont été nécessaires pour chacun des trois travaux. Ceci nous a conduit à utiliser plusieurs

outils pour capitaliser les expériences acquises dans ces études du domaine *Mining Software Repository*. Nous avons déployé les 3 prototypes suivants dans ces opérations :

— HARMONY [1] : un *framework* pour orchestrer et spécifier des analyses sur des systèmes de contrôle de version hétérogènes (Git, SVN et Mercurial). Cet outil est régulièrement utilisé par les membres du thème Génie Logiciel du LaBRI. L'auteur de cette thèse fait partie des contributeurs réguliers sur ce projet.

— SCANLIB [2] : un programme léger permettant de rechercher textuellement des symboles de bibliothèques dans un logiciel Java.

— LIBTIC [3] : un prototype pour identifier des experts en bibliothèques tierces Java. Seule une interface web permettant d'utiliser le langage de requête est disponible.

Dans notre première contribution, détaillée dans le Chapitre 3, nous aidons le développeur à déterminer une bibliothèque candidate à la migration par l'observation et l'exploitation des tendances de migrations de bibliothèques. Il s'agit d'analyser le comportement d'une grande population de logiciels afin d'identifier quels remplacements de bibliothèques y ont été effectués. La première étape consiste en un algorithme permettant d'extraire des migrations de bibliothèques candidates en étudiant les versions successives d'un projet. Nous montrons lors de deux cas d'étude que cette opération est réaliste et permet de recenser des migrations de bibliothèque candidates. Chaque cas d'étude se distingue par la source de données étudiée (GITHUB et dépôt MAVEN) ainsi que par le mode d'extraction des bibliothèques utilisées (analyse de code source et de fichier de configuration). Pour chacun d'eux, nous validons ou écartons manuellement les différentes migrations candidates. Deux constats ressortent des résultats de ce travail. Tout d'abord, notre approche est efficace pour identifier des migrations même si elle nécessite encore des raffinements pour pouvoir être entièrement automatisée. Ensuite, les tendances de migrations peuvent être exploitées grâce à des visualisations qui mettent en avant les bibliothèques pertinentes et, à l'inverse, celles à ne pas considérer.

Dans notre deuxième contribution, détaillée dans le Chapitre 4, nous assistons le développeur voulant appliquer une migration de bibliothèque. Nous proposons pour cela une approche et un procédé pour extraire automatiquement des correspondances de fonctions entre bibliothèques similaires. L'idée consiste, d'une part, à identifier un ensemble de logiciels ayant effectué une migration donnée, et, par la suite, à analyser quelles modifications ont été nécessaires dans leur code source respectif pour accomplir la migration. Cette observation par l'apprentissage permet de comprendre comment un logiciel s'est adapté à la nouvelle bibliothèque lors d'une migration. La liste des correspondances que nous produisons fournit selon nous une aide pour les développeurs. Nous déployons notre approche sur un corpus de 11 598 projets logiciels *open source* et pour un ensemble défini de migrations de bibliothèques. Nous faisons ressortir deux constats de ce travail. Tout d'abord,

1. https://code.google.com/p/harmony/
2. https://code.google.com/p/scanlib-java/
3. https://code.google.com/p/labri-se/source/checkout?repo=libtic

notre approche est réaliste et viable, puisqu'elle permet d'identifier des correspondances. Cependant, ces premiers résultats nécessitent des travaux futurs pour évaluer son efficacité, notamment en ce qui concerne l'ensemble des correspondances existantes à identifier.

Notre dernière contribution, détaillée dans le Chapitre 5, concerne la recherche de développeurs experts en bibliothèques tierces. Elle consiste en une solution automatique permettant de s'affranchir des problèmes liés à l'intervention des développeurs dans un tel procédé. Cette décision implique que ces développeurs doivent être évalués objectivement selon leurs contributions, et nous proposons en ce sens de mesurer quantitativement l'utilisation qu'ils font des bibliothèques tierces. Ainsi, la définition d'expertise est dans ce contexte fortement liée à celle d'expérience, et notre hypothèse de travail suppose que l'expérience est corrélée à l'expertise. Nous proposons un prototype nommé LIBTIC, permettant d'extraire à partir des systèmes de contrôle de version les expériences des développeurs sur les bibliothèques. Ce prototype définit également un langage de requêtes pour manipuler le modèle de données et les résultats générés. Nous collectons l'utilisation des bibliothèques effectuée par les contributeurs de 6 330 projets *open source*. Nous étudions la validité de nos résultats en interrogeant directement des personnes identifiées comme des experts selon nos données. Notre observons que notre approche permet d'identifier de vrais experts parmi une large population de développeurs. Nous illustrons l'intérêt de LIBTIC via deux scénarios d'utilisation lors de la maintenance d'un projet logiciel.

6.2 Perspectives

Nous identifions trois perspectives que nous souhaiterions aborder en priorité dans le cadre de la poursuite de nos travaux.

6.2.1 Automatiser la détection de migration de bibliothèque

L'objectif est ici de pouvoir efficacement répliquer notre approche de façon automatique, pour mettre à jour régulièrement les tendances de migrations, et offrir aux développeurs des informations qui ne sont pas obsolètes. Pour cela, nous devons travailler sur les deux axes suivants :

1. La précision des résultats : nous devons éliminer le plus possible les migrations candidates non valides, tout en excluant une intervention manuelle.
2. La couverture des bibliothèques : nous devons identifier le plus de bibliothèques possibles afin de proposer aux développeurs le plus d'alternatives possibles.

La piste principale pour y arriver est de renforcer la construction des catégories de bibliothèques à partir de nouvelles sources d'information. Le problème de précision est surmonté si ces catégories sont prédéfinies à l'avance. Tout d'abord, nous envisageons de reprendre les travaux de Wang et al. sur la catégorisation de projets *open source* [Wang *et al.*,

2013]. Une plateforme est notamment disponible en ligne [4], qui permet de tester leur approche. Cela permettrait d'agrandir notre base de données de bibliothèques, mais également celles des catégories. Il sera néanmoins nécessaire d'étudier la qualité des catégories proposées dans un contexte d'utilisation de bibliothèques. Une autre piste serait d'analyser les plateformes d'échanges telles que StackOverflow [5] pour extraire des bibliothèques similaires. Il est fréquent que les développeurs interrogent leurs pairs pour chercher la réponse au problème que nous considérons dans nos travaux (comme ici par exemple [6]). Le contenu de ces fils de discussions fournit probablement des noms de bibliothèques candidates à la migration. Il faudrait donc analyser ces messages pour extraire de telles informations.

6.2.2 Améliorer la détection de correspondances entre fonctions de bibliothèques similaires

Les résultats exposés dans le Chapitre 4 indiquent que des efforts supplémentaires sont nécessaires pour parvenir à générer automatiquement des informations utiles aux développeurs souhaitant migrer. L'axe principal d'amélioration concerne l'identification des logiciels sujets. Pour augmenter nos chances d'en trouver, nous envisageons d'utiliser des moteurs de recherche de code source comme celui proposé par Ohloh [7]. L'idée est de rechercher les symboles d'un couple de bibliothèques pour déterminer un ensemble de projets clients. Ces projets sont les plus enclins à avoir effectué une migration de bibliothèque au cours de leur histoire.

Notre deuxième objectif est d'utiliser des techniques supplémentaires pour raffiner les correspondances voire en détecter de nouvelles. Puisque l'on sait que peu de fonctions sont généralement utilisées par les projets clients, notre approche ne peut en théorie pas détecter toutes les correspondances existantes. Notre solution consiste à analyser les documentations des bibliothèques et les commentaires permettant de décrire les fonctions (par exemple, la *Javadoc* pour des fonctions Java). En effet, il est possible que deux fonctions similaires de deux bibliothèques aient une documentation dont le contenu et la signification convergent. Par exemple, si l'on reprend l'exemple exposé en Section 4.1, les documentations des deux fonctions concernées indiquent :

— Validate.isTrue(boolean) : « *Validate that the argument condition is true ;[...]* » ;

— Preconditions.checkArgument(boolean) : « *Ensures the truth of an expression involving one or more parameters to the calling method.[...]* ».

4. http://influx.trustie.net/icsm.jsp
5. http://stackoverflow.com
6. http://stackoverflow.com/questions/1668862/good-json-java-library
7. https://code.ohloh.net/

Nous voyons ici que le rapprochement peut être effectué. Des techniques d'analyse du langage naturel peuvent selon nous permettre de détecter automatiquement de nouvelles correspondances de fonctions.

6.2.3 Généraliser l'expertise des développeurs

Les bibliothèques tierces ne représentent pas les seules ressources exploitées lors du développement d'un logiciel. Les développeurs utilisent des langages de programmation ainsi que d'autres technologies dont il peut s'avérer nécessaire d'identifier des experts. Un meilleur suivi des compétences des développeurs a un impact positif sur la qualité de la gestion des équipes de développement.

Notre objectif principal est de proposer une approche qui spécifie différentes compétences génériques pour un développeur. Cela peut concerner un ensemble de fichiers ou un répertoire (haut-niveau d'information) mais aussi la sémantique des fichiers et leur contenu (bas-niveau). Nous pourrions ainsi identifier quels développeurs utilisent un langage de programmation ou une bibliothèque donnée, et lesquels éditent le plus un module ou écrivent des tests unitaires. De plus, la spécification des compétences doit être simple et être permise par un langage dédié. Un outil d'analyse de dépôts logiciels doit ensuite extraire automatiquement chaque niveau de compétence pour les développeurs (ou niveau d'expérience, comme introduit par nos travaux avec LIBTIC).

À l'heure actuelle, nous avons abordé cette perspective et mis en place un outil complet nommé XTIC, dont la description a été publiée récemment [Teyton *et al.*, 2014]. Une partie de ce travail inclut notamment une expérience conduite en collaboration avec un partenaire industriel (AKKA[8]), où nous comparons les niveaux d'expérience de développeurs déterminés par notre outil avec ceux fournis par deux responsables d'un projet. Les résultats indiquent que les données que nous produisons sont plus fidèles à la réalité que celles proposées par ces deux personnes.

8. http://www.akka.fr/

ANNEXE

A Taxonomie des systèmes de contrôle de versions

Nous donnons ici une liste de termes relatifs aux systèmes de contrôle de versions. Ces termes sont régulièrement repris tout au long de ce manuscrit :

— *VCS* : *Version Control System*, ou système de contrôle de versions. Ils permettent d'enregistrer chaque nouvelle version des fichiers d'un logiciel. Des outils comme SVN, Mercurial ou Git sont des exemples de tels systèmes.

— *commit* ou *révision* : une référence à une révision unique du projet sur le *VCS*, résultant d'une validation par un développeur d'un ensemble de changements.

— *auteur d'un commit* : l'identité de la personne ayant validé les changements entraînant une nouvelle version.

— *changeset d'un commit* : ensemble des fichiers affectés par les éditions d'un développeur lors d'un *commit*.

— *commit parent* : la référence pour un *commit* donné du *commit* prédécesseur. Seul le premier *commit* de l'historique n'a pas de parent.

— *méta-données d'un commit* : les *VCS* proposent des facilités pour indiquer des informations comme la date, l'auteur, le message de commit ainsi que le *changeset* de la transaction.

— *synchroniser un dépôt à une version* : la copie de travail du dépôt est synchronisée physiquement à une version donnée d'un projet.

Remarque. Nous tenons à clarifier les différences sur la notion de *version* pour une bibliothèque et pour un projet logiciel sous *VCS*. Une *version* d'une bibliothèque (ex : *JUnit 3.8.1*) correspondra à une *release* de la bibliothèque à un instant donné. Sur un système de contrôle de versions, une version sera caractérisée par un numéro d'identifiant unique correspondant à un *commit* donné. Ainsi, lorsque nous parlerons de

JUnit 3.8.1 et de *JUnit 4.8.1*, il s'agira de deux *releases* ne donnant aucune information sur le temps et le nombre de *commits* écoulées entre les deux versions de la bibliothèque. Les versions d'un projet client dans notre contexte seront toujours associées à des numéros de *commit* sur le *VCS* de ce projet.

ANNEXE

B

Termes avec une sémantique de migration

Voici la liste des termes recherchés dans les messages de *commits* :

```
"migrate","migrating","migrated","migration",
"change","changed","changing",
"replace","replaced","replacing","replacement",
"switch","switched","switching",
"swap","swapped",
"move","moved","moving",
"convert","converted","converting","conversion",
"alter","altered","altering",
"substitute","substituted","substituting",
"refactor","refactored","refactoring",
"modify","modified","modifying","modification",
"integrate","integration","integrating","integrated",
"evolution","evolve","evolved","evolving",
"configure","configuration","configured","configuring",
"use","using","usage",
"fix","fixing","fixed",
"update","updated","updating",
"->","instead of","transition"
```

Bibliographie

A. Abran et H. Nguyenkim : Analysis of maintenance work categories through measurement. *In Conference on Software Maintenance, 1991., Proceedings*, pages 104–113, 1991. Cité page 4.

A. V. Aho et M. J. Corasick : Efficient string matching : An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, juin 1975. ISSN 0001-0782. Cité page 38.

L. J. Arthur : *Software Evolution : The Software Maintenance Challenge*. Wiley-Interscience, New York, NY, USA, 1988. ISBN 0-471-62871-9. Cité page 4.

M. T. Baldassarre, A. Bianchi, D. Caivano et G. Visaggio : An industrial case study on reuse oriented development. *In Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05*, pages 283–292, 2005. Cité page 2.

V.R. Basili : Viewing maintenance as reuse-oriented software development. *IEEE Software*, 7(1):19–25, 1990. ISSN 0740-7459. Cité page 2.

G. Bavota, G. Canfora, M. Di Penta, R. Oliveto et S. Panichella : The evolution of project inter-dependencies in a software ecosystem : The case of apache. *In 2013 29th IEEE International Conference on Software Maintenance (ICSM)*, pages 280–289, septembre 2013. Cité page 14.

B. Bazelli, M. Fokaefs et E. Stroulia : Mapping the responses of RESTful services based on their values. *In 2013 15th IEEE International Symposium on Web Systems Evolution (WSE)*, pages 15–24, 2013. Cité page 21.

B. W. Boehm et P. N. Papaccio : Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477, 1988. ISSN 0098-5589. Cité page 4.

J. Businge : Co-evolution of the eclipse SDK framework and its third-party plug-ins. *In 2013 17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 427–430, 2013. Cité page 14.

S. Chiba : Javassist – a reflection-based programming wizard for java. *In Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, octobre 1998. Cité page 88.

B. E. Cossette et R. J. Walker : Seeking the ground truth : a retroactive study on the evolution and migration of software libraries. *In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, page 55 :1–55 :11, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1614-9. Cité page 20.

B. Dagenais et M. P. Robillard : SemDiff : analysis and recommendation support for API evolution. *In IEEE 31st International Conference on Software Engineering, 2009. ICSE 2009*, pages 599–602, 2009. Cité pages 18 et 20.

J. Davies, D. M. Germán, M. W. Godfrey et A. Hindle : Software bertillonage : finding the provenance of an entity. *In Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*, pages 183–192. IEEE, 2011. ISBN 978-1-4503-0574-7. Cité page 99.

L. R. Dice : Measures of the amount of ecologic association between species. *Ecology*, 26 (3):297, juillet 1945. ISSN 00129658. Cité page 57.

D. Dig, K. Manzoor, R. Johnson et T. N. Nguyen : Refactoring-aware configuration management for object-oriented programs. *In 29th International Conference on Software Engineering, 2007. ICSE 2007*, pages 427–436, 2007. Cité pages 19 et 20.

R. Dyer, H. A. Nguyen, H. Rajan et T.N. Nguyen : Boa : A language and infrastructure for analyzing ultra-large-scale software repositories. *In Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 422–431, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. Cité page 11.

J.-R. Falleri, C. Teyton, M. Foucault, M. Palyart, F. Morandat et X. Blanc : The harmony platform. Rapport technique, Univ. Bordeaux, LaBRI, UMR 5800, septembre 2013. Cité page 11.

B. Fluri, M. Würsch, M. Pinzger et H. Gall : Change distilling : Tree differencing for fine-grained source code change extraction. *IEEE Trans. Software Eng.*, 33(11):725–743, 2007. Cité pages 81 et 99.

W. B. Frakes et C. J. Fox : Sixteen questions about software reuse. *Commun. ACM*, 38(6): 75–ff., juin 1995. ISSN 0001-0782. Cité page 2.

W. B. Frakes et K. Kang : Software reuse research : Status and future. *IEEE Trans. Softw. Eng.*, 31(7):529–536, juillet 2005. ISSN 0098-5589. Cité page 2.

T. Fritz, G. C. Murphy et E. Hill : Does a programmer's activity indicate knowledge of code ? *In Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, page 341–350, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-811-4. Cité pages 24 et 26.

H. Gall, K. Hajek et M. Jazayeri : Detection of logical coupling based on product release history. *In , International Conference on Software Maintenance, 1998. Proceedings*, pages 190–198, 1998. Cité page 10.

E. Gamma, R. Helm, R. Johnson et J. Vlissides : *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, Amsterdam, 1 édition, 1995. ISBN 0201633612. 37. Reprint (2009). Cité page 21.

D.M. German, B. Adams et A.E. Hassan : The evolution of the r software ecosystem. *In 2013 17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 243–252, mars 2013. Cité page 15.

E. Giger, M. Pinzger et H. C. Gall : Comparing fine-grained source code changes and code churn for bug prediction. *In Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, page 83–92, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0574-7. Cité page 10.

T. Girba, A. Kuhn, M. Seeberger et S. Ducasse : How developers drive software evolution. *In Eighth International Workshop on Principles of Software Evolution*, pages 113–122, 2005. Cité pages 23 et 26.

M. Goeminne et T. Mens : A comparison of identity merge algorithms for software repositories. *Science of Computer Programming*, (0):–, 2011. ISSN 0167-6423. Cité page 98.

A. Gokhale, V. Ganapathy et Y. Padmanaban : Inferring likely mappings between APIs. *In Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 82–91, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. Cité page 21.

G. Gousios : The GHTorrent dataset and tool suite. *In Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, page 233–236, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-2936-1. Cité page 36.

R. E. Grinter : Supporting articulation work using software configuration management systems. *Computer Supported Cooperative Work (CSCW)*, 5(4):447–465, 1996. ISSN 0925-9724. Cité page 24.

M. L. Griss : Software reuse : From library to factory. *IBM Systems Journal*, 32(4):548–566, 1993. ISSN 0018-8670. Cité page 2.

L. Hattori et M. Lanza : Mining the history of synchronous changes to refine code ownership. *In 6th IEEE International Working Conference on Mining Software Repositories, 2009. MSR '09*, pages 141–150, 2009. Cité pages 24 et 26.

L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel et M. Irlbeck : On the extent and nature of software reuse in open source java projects. *In Proceedings of the 12th international conference on Top productivity through software reuse*, ICSR'11, page 207–222, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-21346-5. Cité page 3.

H. Hemmati, S. Nadi, O. Baysal, O. Kononenko, W. Wang, R. Holmes et M. W. Godfrey : The MSR cookbook : Mining a decade of research. *In Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, page 343–352, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-2936-1. Cité page 10.

J. Henkel et A. Diwan : CatchUp ! capturing and replaying refactorings to support API evolution. *In 27th International Conference on Software Engineering, 2005. ICSE 2005. Proceedings*, pages 274–283, 2005. Cité pages 19 et 20.

R. Holmes et R. J. Walker : Informing eclipse API production and consumption. *In Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '07, page 70–74, New York, NY, USA, 2007. ACM. ISBN 978-1-60558-015-9. Cité page 22.

B. Hummel, E. Juergens, L. Heinemann et M. Conradt : Index-based code clone detection : incremental, distributed, scalable. *In 2010 IEEE International Conference on Software Maintenance (ICSM)*, pages 1–9, septembre 2010. Cité page 99.

W. M. Ibrahim, Nicolas Bettenburg, E. Shihab, B. Adams et A. E. Hassan : Should i contribute to this discussion ? *In 2010 7th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 181–190, 2010. Cité page 10.

J. Jiang, L. Zhang et L. Li : Understanding project dissemination on a social coding site. *In 2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 132–141, octobre 2013. Cité page 61.

R. Joos : Software reuse at motorola. *IEEE Software*, 11(5):42–47, 1994. ISSN 0740-7459. Cité page 2.

H. Kagdi, M. L. Collard et J. I. Maletic : A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution : Research and Practice*, 19(2):77–131, 2007. ISSN 1532-0618. Cité page 10.

H. Kagdi, M. Hammad et J. I. Maletic : Who can help me with this source code change ? *In Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 157–166, 2008. Cité pages 23 et 26.

S. Kawaguchi, P. K. Garg, M. Matsushita et K. Inoue : MUDABlue : an automatic categorization system for open source repositories. *In Software Engineering Conference, 2004. 11th Asia-Pacific*, pages 184–193, 2004. Cité page 15.

F. Khomh, T. Dhaliwal, Y. Zou et B. Adams : Do faster releases improve software quality ? an empirical case study of mozilla firefox. *In 2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 179–188, 2012. Cité page 10.

M. Kim, D. Notkin et D. Grossman : Automatic inference of structural changes for matching across program versions. *In Proceedings of the 29th international conference on Software Engineering*, ICSE '07, page 333–343, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. Cité pages 17 et 20.

R. Lämmel, E. Pek et J. Starek : Large-scale, AST-based API-usage analysis of open-source java projects. *In Proceedings of the 2011 ACM Symposium on Applied Computing - SAC '11*, page 1317, 2011. Cité page 22.

H. Leibenstein : Bandwagon, snob, and veblen effects in the theory of consumers' demand. *The Quarterly Journal of Economics*, 64(2):183–207, mai 1950. ISSN 0033-5533, 1531-4650. Cité page 8.

W. C. Lim : Effects of reuse on quality, productivity, and economics. *IEEE Software*, 11 (5):23–30, 1994. ISSN 0740-7459. Cité page 2.

M. Linares-Vasquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers et D. Poshyvanyk : Triaging incoming change requests : Bug or commit history, or code authorship ? *In 2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 451–460, 2012. Cité pages 23 et 26.

D. Ma, D. Schuler, T. Zimmermann et J. Sillito : Expert recommendation with usage expertise. *In Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 535–538, 2009. Cité page 25.

D. Matter, A. Kuhn et O. Nierstrasz : Assigning bug reports using a vocabulary-based expertise model of developers. *In Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, page 131–140, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3493-0. Cité page 10.

D. W. McDonald et M. S. Ackerman : Expertise recommender : a flexible recommendation system and architecture. *In Proceedings of the 2000 ACM conference on Computer supported cooperative work*, CSCW '00, page 231–240, New York, NY, USA, 2000. ACM. ISBN 1-58113-222-0. Cité page 23.

T. McDonnell, B. Ray et M. Kim : An empirical study of API stability and adoption in the android ecosystem. *In Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, page 70–79, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4981-1. Cité page 14.

D. Mcilroy : Mass-produced software components. pages 138–155, janvier 1969. Cité page 2.

C. Mcmillan, M. Linares-vásquez, D. Poshyvanyk et M. Grechanik : Categorizing software applications for maintenance. 2011. Cité page 15.

S. Melnik, H. Garcia-Molina et E. Rahm : Similarity flooding : A versatile graph matching algorithm and its application to schema matching. *In ICDE*, pages 117–128, 2002. Cité page 70.

S. Meng, X. Wang, L. Zhang et H. Mei : A history-based matching approach to identification of framework evolution. *In 2012 34th International Conference on Software Engineering (ICSE)*, pages 353–363, 2012. Cité pages 18 et 20.

T. Mens : Introduction and roadmap : History and challenges of software evolution. *In Software Evolution*, pages 1–11. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-76439-7. Cité page 10.

Y. M. Mileva, V. Dallmeier, M. Burger et A. Zeller : Mining trends of library usage. *In Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, IWPSE-Evol '09, page 57–62, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-678-6. Cité page 14.

A. Mili, R. Mili et R. T. Mittermeir : A survey of software reuse libraries. *Ann. Softw. Eng.*, 5:349–414, janvier 1998. ISSN 1022-7091. Cité page 2.

A. Mockus et J. D. Herbsleb : Expertise browser : a quantitative approach to identifying expertise. *In Proceedings of the 24rd International Conference on Software Engineering, 2002. ICSE 2002*, pages 503–512, 2002. Cité pages 23 et 26.

E. W. Myers : An o(ND) difference algorithm and its variations. *In Algorithmica*, pages 251–266, 1986. Cité page 68.

N. Nagappan et T. Ball : Using software dependencies and churn metrics to predict field failures : An empirical case study. *In First International Symposium on Empirical Software Engineering and Measurement, 2007. ESEM 2007*, pages 364–373, septembre 2007. Cité page 3.

M. Nita et D. Notkin : Using twinning to adapt programs to alternative APIs. *In 2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 205–214, mai 2010. Cité page 21.

S. Okur et D. Dig : How do developers use parallel libraries ? *In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, page 54 :1–54 :11, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1614-9. Cité page 22.

J. Ossher, S. Bajracharya et C. Lopes : Automated dependency resolution for open source software. *In 2010 7th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 130–140, mai 2010. Cité page 89.

S. Raemaekers, A. van Deursen et J. Visser : Measuring software library stability through historical version analysis. *In 2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 378–387, 2012. Cité page 22.

J. Ratzinger, T. Sigmund, P. Vorburger et H. Gall : Mining software evolution to predict refactoring. *In First International Symposium on Empirical Software Engineering and Measurement, 2007. ESEM 2007*, pages 354–363, 2007. Cité page 10.

F. Ricca et A. Marchetto : Heroes in FLOSS projects : An explorative study. *In 2010 17th Working Conference on Reverse Engineering (WCRE)*, pages 155–159, octobre 2010. Cité page 96.

P. C. Rigby et A. E. Hassan : What can OSS mailing lists tell us ? a preliminary psychometric text analysis of the apache developer mailing list. *In Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, page 23–, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2950-X. Cité page 10.

R. Robbes, M. Lungu et D. Röthlisberger : How do developers react to API deprecation ? : The case of a smalltalk ecosystem. *In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, page 56 :1–56 :11, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1614-9. Cité page 14.

R. Robbes et D. Röthlisberger : Using developer interaction data to compare expertise metrics. *In Proceedings of the Tenth International Workshop on Mining Software Repositories*, page 297–300, 2013. Cité pages 24 et 26.

M. P. Robillard et R. Deline : A field study of API learning obstacles. *Empirical Softw. Engg.*, 16(6):703–732, décembre 2011. ISSN 1382-3256. Cité page 6.

T. Schäfer, J. Jonas et M. Mezini : Mining framework usage changes from instantiation code. *In Proceedings of the 13th international conference on Software engineering - ICSE '08*, page 471, 2008. Cité pages 18, 20, 65, 74 et 76.

D. Schuler et T. Zimmermann : Mining usage expertise from version archives. *In Proceedings of the 2008 international working conference on Mining software repositories*, MSR '08, page 121–124, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-024-1. Cité pages 24 et 26.

W. Schwittek et S. Eicker : A study on third party component reuse in java enterprise open source software. *In Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '13, page 75–80, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2122-8. Cité page 3.

F. Servant et J. A. Jones : WhoseFault : automatic developer-to-fault assignment through fault localization. *In Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, page 36–46, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. Cité pages 23 et 26.

C. E. Shannon : A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1):3–55, janvier 2001. ISSN 1559-1662. Cité page 79.

B. Sisman et A. C. Kak : Incorporating version histories in information retrieval based bug localization. *In 2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 50–59, 2012. Cité page 10.

B. Stroustrup : Language-technical aspects of reuse. *In , Proceedings Fourth International Conference on Software Reuse, 1996*, pages 11–19, 1996. Cité page 2.

K. Taneja, D. Dig et T. Xie : Automated detection of api refactorings in libraries. *In Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, page 377–380, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. Cité pages 17 et 20.

C. Teyton, M. Palyart, J.R. Falleri, F. Morandat et X. Blanc : Automatic extraction of developer expertise. *In* IEEE, éditeur : *18th International Conference on Evaluation and Assessment in Software Engineering*, London, UK, mai 2014. Cité page 105.

S. Thummalapenta et T. Xie : SpotWeb : detecting framework hotspots and coldspots via mining open source code on the web. *In 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008. ASE 2008*, pages 327–336, 2008. Cité page 22.

F. Thung, D. Lo et L. Jiang : Detecting similar applications with collaborative tagging. *In 2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 600–603, septembre 2012. Cité page 15.

F. Thung, D. Lo et J. Lawall : Automated library recommendation. *In Proc.\ WCRE*, page 182–191. IEEE, 2013. Cité page 3.

T. Tonelli Bartolomei, K. Czarnecki et R. Lämmel : Swing to SWT and back : Patterns for API migration by wrapping. *In 26th IEEE International Conference on Software Maintenance (ICSM)*, Timisoara, Romania, septembre 2010. Cité page 21.

T. Tonelli Bartolomei, K. Czarnecki, R. Lämmel et T. v. d. Storm : Study of an API migration for two XML APIs. *In 2nd International Conference on Software Language Engineering (SLE)*, volume 5969/2010, pages 42–61, Denver, USA, octobre 2009. ISBN 978-3-642-12106-7. Cité page 21.

A. Vivacqua et H. Lieberman : Agents to assist in finding help. *In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '00, page 65–72, New York, NY, USA, 2000. ACM. ISBN 1-58113-216-6. Cité page 23.

T. Wang, H. Wang, G. Yin, C. X. Ling, X. Li et P. Zou : Mining software profile across multiple repositories for hierarchical categorization. *In 2013 29th IEEE International Conference on Software Maintenance (ICSM)*, pages 240–249, 2013. Cité pages 15 et 103.

T. Wang, G. Yin, X. Li et H. Wang : Labeled topic detection of open source software from mining mass textual project profiles. *In Proceedings of the First International Workshop on Software Mining*, SoftwareMining '12, page 17–24, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1560-9. Cité page 15.

P. Weissgerber et S. Diehl : Identifying refactorings from source-code changes. *In Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, page 231–240, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2579-2. Cité pages 17 et 20.

M. Wermelinger et Y. Yu : Analyzing the evolution of eclipse plugins. *In Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, page 133–136, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-024-1. Cité page 14.

W. Wu, Y. -G. Guéhéneuc, G. Antoniol et M. Kim : AURA : a hybrid approach to identify framework evolution. *In 2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 325–334, 2010. Cité pages 17, 18 et 20.

Z. Xing et E. Stroulia : API-Evolution support with diff-CatchUp. *IEEE Transactions on Software Engineering*, 33(12):818–836, 2007. ISSN 0098-5589. Cité pages 17 et 20.

Y. Ye, Y. Yamamoto, K. Nakakoji, Y. Nishinaka et M. Asada : Searching the library and asking the peers : learning to use java APIs on demand. *In Proceedings of the 5th international symposium on Principles and practice of programming in Java*, PPPJ '07, page 41–50, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-672-1. Cité pages 25 et 26.

T. Zimmermann : Fine-grained processing of CVS archives with APFEL. *In Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '06, page 16–20, New York, NY, USA, 2006. ACM. ISBN 1-59593-621-1. Cité page 25.

Table des figures

Liste des tableaux

Listings

Abstract

Software depend on third-party libraries to reduce development and maintenance costs. Developers have access to robust functionalities through an application programming interface designed by these libraries. However, due to the strong relationship with these libraries, developers have to reconsider their position when the software evolves. In this thesis, we identify several research problems involving these third-party libraries in a context of software maintenance. More specifically, a library may not satisfy the software new requirements and has to be replaced by a new one. We call this operation a library migration.

We leverage three points that characterize the impediments met by developers in this situation. To which library should they migrate ? How to migrate their software ? Who can help them in this case ? This thesis suggests answers and exposes several contributions to these problems. We define three approaches that are evaluated through several case studies. To achieve this work, we use a methodology based on software evolution analysis to observe and understand how software change. We describe numerous perspectives to overcome the current limitations of our solutions.

Keywords: *Software maintenance, Software evolution, Experts identification, Software library, Version control system, Code Reuse*

Résumé

Les logiciels dépendent de bibliothèques tierces pour réduire les coûts liés à leur développement et à leur maintenance. Elles proposent un ensemble de fonctionnalités robustes dont les développeurs peuvent tirer parti depuis une interface de programmation. Cependant, cette forte dépendance entre un logiciel et ses bibliothèques oblige les développeurs à reconsidérer leur rôle lorsque le logiciel évolue. Dans cette thèse, nous identifions plusieurs problématiques impliquant les bibliothèques tierces dans un contexte de maintenance logicielle. Plus particulièrement, une bibliothèque peut ne plus répondre aux besoins d'un logiciel et doit être remplacée par une nouvelle. Nous nommons cette opération une migration de bibliothèque.

Nous soulevons dans ce contexte trois points qui caractérisent les difficultés rencontrées par les développeurs. Vers quelle bibliothèque migrer ? Comment appliquer la migration ? Avec l'aide de quels développeurs ? Cette thèse discute de solutions et apporte des contributions autour de ces problèmes. Nous présentons plusieurs approches et les évaluons lors de différents cas d'étude. L'analyse de l'évolution logicielle sera notre support de travail, dont la méthodologie est basée sur l'observation des changements de logiciels. Nous décrivons les limites actuelles de nos contributions et ouvrons des perspectives futures pour enrichir l'état de l'art dans ce domaine.

Mots clefs : *Maintenance logicielle, Évolution logicielle, Identification d'experts, Bibliothèque logicielle, Système de contrôle de versions, Réutilisation de code*

LaBRI — 351, Avenue de la Libération— 33400 Talence — France