Académie de Bordeaux
Université De Bordeaux
Sciences et Technologies

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**

Contributions à l'usage des détecteurs de clones pour des tâches de maintenance logicielle

par

Alan Charpentier

Soutenue le 17 Octobre 2016, devant le jury composé de :

Président du jury
Guillaume Blin, professeur .... Université de Bordeaux, France

Directeur de thèse
Laurent Réveillère, maître de conférences HDR .... Bordeaux INP, France

Co-Directeur de thèse
Jean-Rémy Falleri, maître de conférences HDR .... Bordeaux INP, France

Rapporteurs
Julia Lawall, directeur de recherche .... INRIA Paris-Rocquencourt, France
Tom Mens, professeur .... Université de Mons, Belgique

Examinateur
Martin Monperrus, maître de conférences HDR .... Université de Lille 1, France

Remerciements

L'écriture de ces remerciements marque la fin d'une extraordinaire aventure qui a débuté en Octobre 2013 et s'est écoulée sur trois années. Je souhaite remercier dans cette section toutes les personnes qui ont contribué à la richesse de cette expérience autant sur le plan professionnel que personnel.

Je remercie tout d'abord, Laurent Réveillère et Jean-Rémy Falleri, mes deux directeurs de thèse. Leur engagement et leur qualité d'accompagnement ont conduit à la réussite de cette thèse. Merci à eux deux pour leur rigueur ainsi que pour les conseils et enseignements qu'ils m'ont prodigués.

Je remercie Julia Lawall, Tom Mens, Martin Monperrus et Guillaume Blin qui ont accepté de participer à mon jury de thèse. J'ai été honoré d'échanger avec des chercheurs aussi brillants à l'occasion de ma dernière journée au laboratoire.

J'adresse un remerciement tout particulier à Xavier Blanc qui a suivi mon parcours depuis le Master. Depuis lors, ses nombreux conseils toujours avisés m'ont été d'une grande aide.

Je remercie Floréal pour toute l'aide qu'il m'a apportée et ses discussions toujours passionnantes et riches en enseignement.

Je remercie David Lo avec qui j'ai eu l'occasion de collaborer pour ma première publication scientifique. Je remercie également Cédric, Arthur, Mohamed, Elyas et Hanyang pour leur participation et leur aide dans mes recherches. Je remercie plus généralement tous mes collègues du LaBRI : les membres du thème Génie Logiciel, mes collègues du bureau 125 et le personnel administratif.

Je remercie à nouveau Jean-Rémy ainsi que Bruno Pinaud et le personnel de l'ENSEIRB-MATMECA qui m'ont donné l'opportunité d'enseigner l'informatique.

Je remercie Matthieu et Xavier, que je compte maintenant parmi mes amis, qui ont fait

de chaque journée au bureau une expérience unique. Je n'oublierai jamais nos pauses café et nos parties de ping-pong. Merci à eux deux pour leur aide, leurs conseils et leur soutien.

J'adresse mes plus profonds remerciements à mes parents qui m'ont donné la chance de faire des études. Je vous serai à jamais reconnaissant pour tout le soutien que vous m'avez apporté. Je tiens également à remercier Elyane ma grand-mère.

J'adresse également mes plus profonds remerciements à ma chérie, Marianne, qui m'a soutenu et épaulé durant cette aventure, et bien plus encore.

Je remercie enfin mes amis : Joris, Marina, Yann, Jeanne, Christophe, Milan, Véronica, Emilie, Sébastien, Laura, Christine, Charlie et Eric. Merci à eux d'avoir été présents. Je voudrais également remercier Pédro et Dominique qui ne sont plus là aujourd'hui mais que je n'oublierai jamais.

Pour conclure ces remerciements, je présente mes excuses à toutes les personnes que j'aurais involontairement oublié de mentionner.

Table des matières

Remerciements **i**

1 Introduction **1**
1.1 Duplication de code logiciel 2
1.2 Détection de clones 2
1.3 Problématiques de recherche et contributions 3
1.3.1 Principale méthodologie d'évaluation et de comparaison 4
1.3.2 Évaluation contextuelle des détecteurs de clones 5
1.3.3 Adaptation des techniques de détection 6
1.4 Structure 7

2 Contexte, état de l'art **9**
2.1 Contexte 10
2.1.1 Définitions de clones dans la littérature 10
2.1.2 Processus de détection de clones 15
2.1.3 Techniques et outils de détection de clones 17
2.2 Évaluation des techniques de détection de clones 22
2.2.1 Construction de *benchmarks* de clone 24
2.2.2 Opinion des développeurs sur les clones 27
2.3 Détecteurs de clones et maintenance logicielle 30
2.4 Synthèse de l'état de l'art 33

3 Évaluation empirique d'un *benchmark* de clones **35**
3.1 Introduction 36
3.2 Contexte 37

3.2.1 *Benchmark* de Bellon . . . 37
3.2.2 Questions de recherche . . . 41
3.3 Méthodologie . . . 42
3.3.1 Participants . . . 42
3.3.2 Sélection des clones et examen . . . 42
3.3.3 Récupération de l'opinion des participants . . . 43
3.3.4 Niveau de confiance des clones . . . 44
3.4 Résultats et discussion . . . 44
3.4.1 Niveau de confiance des clones du corpus de référence . . . 45
3.4.2 Niveaux de confiance et mesures d'efficacité . . . 47
3.4.3 Niveau de confiance et caractéristiques des clones . . . 48
3.5 Validité de l'étude . . . 53
3.5.1 Validité de construction . . . 53
3.5.2 Validité interne . . . 53
3.5.3 Validité externe . . . 53
3.6 Conclusion . . . 53

4 Fiabilité des juges dans la construction de *benchmarks* 55
4.1 Introduction . . . 56
4.2 Questions de recherche . . . 56
4.3 Méthodologie . . . 58
4.3.1 Approche globale . . . 58
4.3.2 Sujets et objets . . . 59
4.3.3 Sélection des clones . . . 60
4.3.4 Collecte des données . . . 63
4.4 Résultats et discussion . . . 66
4.4.1 Reproductibilité des réponses des juges . . . 66
4.4.2 Fiabilité inter-juges . . . 67
4.4.3 Remarques des participants . . . 72
4.5 Validité de l'étude . . . 74
4.5.1 Validité de construction . . . 74
4.5.2 Validité interne . . . 74
4.5.3 Validité externe . . . 75
4.6 Conclusion . . . 76

5 Technique de détection de clones spécialisée 77
5.1 Introduction . . . 78
5.2 Contexte . . . 80
5.2.1 Le langage CSS . . . 80
5.2.2 Gestion de la duplication en CSS . . . 81
5.2.3 Travaux connexes . . . 83

5.3 Notre approche 85
5.3.1 Génération de la sous-hiérarchie de Galois 85
5.3.2 Extraction de mixins à partir d'une sous-hiérarchie de Galois 87
5.3.3 Préservation du rendu CSS initial 93
5.3.4 Implémentation de l'outil 94
5.4 Évaluation 94
5.4.1 Expérimentations 95
5.4.2 Études de cas 98
5.5 Conclusion 102

6 Conclusion **103**
6.1 Résumé des contributions 103
6.2 Perspectives 106

Bibliographie **109**

Table des figures **117**

Liste des tableaux **119**

Liste des listings **121**

CHAPITRE 1

Introduction

Ce chapitre introduit au lecteur le contexte, les motivations et les objectifs de notre thèse. Celle-ci repose sur la nécessité pour les développeurs de préserver la qualité des logiciels au cours de leur évolution et maintenance. La duplication de code logiciel (ou simplement clones) peut s'avérer problématique lors de ces phases : plusieurs instances d'un même bogue peuvent par exemple exister. Nous présentons alors l'intérêt de la détection de clones comme support à la qualité logicielle.

Nous avons choisi d'étudier la faible utilisation des outils de détection de clones dans les tâches de maintenance logicielle. Nos contributions sont liées à l'évaluation empirique des outils de détection de clones.

Sommaire

1.1 Duplication de code logiciel **2**
1.2 Détection de clones **2**
1.3 Problématiques de recherche et contributions **3**
1.4 Structure **7**

1.1 Duplication de code logiciel

Le code source des logiciels peut contenir des fragments identiques ou très similaires qui résultent principalement de l'utilisation de copier-coller. Les copies d'un même fragment de code sont nommées des *clones* dans la littérature. Des études empiriques montrent que ce phénomène est important : certains logiciels contiennent en effet entre 5 et 20% de code source dupliqué, c'est-à-dire de clones [Baker, 1995; Baxter *et al.*, 1998].

La création de duplication de code logiciel par les développeurs a plusieurs origines. Elle peut être volontaire lorsqu'il s'agit du résultat d'une stratégie de développement ou de maintenance. En effet, le copier-coller de code existant avec ou sans modification est une manière rapide de produire du nouveau code fiable car éprouvé. Cordy rapporte à ce propos une pratique de développement dans l'industrie de la finance [Cordy, 2003] où les nouvelles fonctionnalités sont développées en adaptant le code actuel afin de profiter des tests existants. Le recours à la duplication de code peut également être contraint lorsque les langages informatiques utilisés ne disposent pas de mécanismes d'abstractions suffisants [Patenaude *et al.*, 1999; Basit *et al.*, 2005]. À l'inverse, l'apparition de duplication de code peut être simplement accidentelle. L'utilisation d'une bibliothèque logicielle particulière peut amener les développeurs à inconsciemment dupliquer plusieurs fois la même série d'appels de fonction, séquences de commandes, etc. [Kapser et Godfrey, 2006].

La duplication de code peut s'avérer problématique lors de la maintenance et de l'évolution logicielle [Juergens *et al.*, 2009]. Par exemple, en présence de code dupliqué dans un logiciel, la résolution complète d'un bogue requiert une étape supplémentaire qui consiste à vérifier s'il existe des fragments similaires à l'original eux aussi affectés par ledit bogue [Li *et al.*, 2006]. Ainsi, pour fiabiliser et faciliter la propagation de correction de bogues, les développeurs et mainteneurs ont besoin de savoir si leurs logiciels contiennent des clones et le cas échéant de connaître leur localisation. Par conséquent, la détection de clones est un enjeu important pour améliorer la qualité logicielle, propriété primordiale pour le succès d'un logiciel [Ralph et Kelly, 2014].

1.2 Détection de clones

La détection de clones a pour objectif d'identifier des fragments de code logiciel identiques ou similaires. Une étape préalable à la recherche de clones dans les logiciels consiste à définir quels fragments de code sont à considérer (notion de granularité) et quelles différences entre ceux-ci sont autorisées (notion de similarité). Ces deux notions centrales sont adaptées et précisées selon la tâche pour laquelle les clones sont recherchés. En effet, différents clones sont à considérer selon que leur identification a pour objectif de réaliser une opération de *refactoring* (« réingénierie logicielle »), appliquer une co-évolution, etc. Par exemple, pour l'extraction de procédures, des clones à la fois identiques et similaires sont recherchés : les différences entre les fragments de code deviennent des paramètres

des nouvelles procédures. À l'inverse, pour une détection de plagiat, seuls des clones identiques sont pertinents. Ainsi, l'utilité d'un clone dépend de la tâche qu'il doit servir.

Les outils de détection de clones (ou plus simplement détecteurs de clones) existants implémentent des techniques très variées, offrent de nombreux paramètres de configuration [Roy *et al.*, 2009] et ont la particularité d'être généralistes dans le sens où les résultats qu'ils produisent ne sont liés à aucune tâche particulière. De plus, leurs résultats dépendent à la fois de la technique de détection sous-jacente et de la configuration utilisée [Wang *et al.*, 2013]. La comparaison des détecteurs de clones devient alors une nécessité pour leurs utilisateurs. Ces derniers ont en effet besoin de déterminer l'outil qui identifiera les clones les plus utiles pour la tâche à accomplir. Comme les outils de détection de clones sont généralistes, ils peuvent produire des résultats non pertinents pour une tâche donnée. Les chercheurs, qui sont les principaux développeurs des outils de détection de clones, ont donc proposé une méthodologie pour leur comparaison et évaluation.

1.3 Problématiques de recherche et contributions

L'objectif général de cette thèse est de contribuer à une meilleure utilisation des outils de détection de clones dans des tâches de maintenance logicielle. Pour cela, nous avons centré nos contributions sur deux axes différents. Tout d'abord, nous évaluons si les méthodologies existantes pour comparer les détecteurs de clones sont pertinentes pour déterminer l'outil le plus adapté à une tâche donnée. Très peu de recherches ont été menées sur ces méthodologies et aucune sur leur validation, alors que l'étude de la duplication de code logiciel est un domaine de recherche actif depuis plus de vingt ans dans la communauté du génie logiciel : Roy *et al.* ont recensé 353 articles sur le sujet publiés entre 1994 et 2013 [Roy *et al.*, 2014]. Ensuite, nous examinons si le caractère généraliste des détecteurs de clones actuels n'est pas un frein à leur adoption dans des tâches de maintenance logicielle. La grande majorité des environnements de développement intégré ne proposent par exemple aucun mécanisme pour la détection de duplication de code. Nous proposons alors le recours à des détecteurs de clones spécialisés dans une tâche pour aider les développeurs à gérer les clones présents dans leurs logiciels.

Les problématiques de recherche abordées dans cette thèse portent sur deux aspects de l'utilisation des outils de détection de clones pour des tâches de maintenance logicielle :

— le choix du détecteur de clones et de la configuration les plus adaptés à une tâche donnée est un défi pour les potentiels utilisateurs. Lorsque nous avons commencé cette thèse, une méthodologie pour la comparaison d'outils de détection de clones était disponible dans la littérature [Bellon *et al.*, 2007]. Notre première problématique est donc d'évaluer cette méthodologie et de confirmer ou d'infirmer les résultats dérivés de celle-ci. Notre deuxième problématique est de déterminer si cette méthodologie peut être raffinée afin de comparer des détecteurs de clones pour

une tâche donnée, avec pour objectif de proposer à chaque utilisateur l'outil le plus adapté à ses propres besoins ;

— la majorité des outils de détection de clones actuels sont généralistes [Roy et Cordy, 2008; Göde et Koschke, 2009] et identifient des clones qui ne sont destinés à aucune tâche de maintenance particulière. Notre troisième problématique est donc d'examiner si l'usage de détecteur de clones spécialisés dans une tâche peut aider les développeurs à gérer les clones présents dans leurs logiciels.

1.3.1 Principale méthodologie d'évaluation et de comparaison

L'un des plus grands défis pour une application efficace de détection de clones est de produire des résultats pertinents pour l'utilisateur de l'outil. Cependant, comme la définition de clones s'applique de manière indifférenciée à de nombreux contextes, certains des clones identifiés par un outil de détection de clones pourraient être considérés comme des faux positifs. Détecter ces faux positifs est toutefois difficile à cause de la très grande masse de données qu'une analyse de clones génère, notamment pour les gros logiciels (exemple du code du noyau Linux).

Une approche traditionnelle pour réduire le nombre de faux positifs est d'améliorer l'algorithme de détection de clones et régler avec précision l'outil de détection. Plusieurs *benchmarks* de vrais clones ont été proposés permettant aux développeurs de détecteurs de clones de comparer et évaluer les résultats de leurs outils. La construction de ces *benchmarks* est généralement assurée par une seule personne décidant elle-même quel clone est ou n'est pas un vrai positif. Par conséquent, nous pouvons nous demander dans quelle mesure cette méthodologie de construction de *benchmarks* de clones est valide.

Le *benchmark* de Bellon [2007] est apparu comme le principal *benchmark* pour comparer et évaluer les détecteurs de clones en utilisant les traditionnelles mesures de précision et rappel [Ducasse *et al.*, 2006; Koschke *et al.*, 2006; Nguyen *et al.*, 2012; Wang *et al.*, 2013]. Cependant, un obstacle préoccupant à la validité de ce corpus de référence est qu'il a été construit et validé par une seule personne, à savoir Bellon. Or, comme ce dernier n'est pas un expert des logiciels dont sont issus les clones et que la définition de clones peut être subjective, il est envisageable que d'autres personnes ne considèrent pas certains des clones de ce corpus de référence comme étant des vrais positifs. Ce phénomène pourrait alors biaiser les mesures de précision et rappel calculées à partir de ce *benchmark*.

Notre problématique est alors de déterminer si la construction et la validation de *benchmarks* par une seule personne produisent des résultats fiables. Pour étudier cette méthodologie, nous menons une évaluation empirique du corpus de référence construit par Bellon en sollicitant l'opinion de dix-huit nouvelles personnes sur un sous-ensemble de ce dernier.

Nos contributions à cette problématique sont les suivantes [Charpentier *et al.*, 2015] :

— nous conduisons une expérience contrôlée avec neuf groupes de deux participants. Pour chaque groupe, nous sélectionnons aléatoirement 120 clones du corpus de

référence de Bellon et demandons aux deux participants s'ils considèrent ces clones comme des vrais positifs ou non;
— nous montrons qu'une part significative des clones de ce corpus de référence sont contestables;
— nous observons que les mesures de précision et de rappel calculées à partir de ce *benchmark* peuvent être significativement affectées par de nouveaux avis;
— finalement, nous montrons que la construction d'un *benchmark* de clones par une seule personne ne permet pas de garantir des résultats fiables.

1.3.2 Évaluation contextuelle des détecteurs de clones

Évaluer des clones pour une tâche donnée consiste à déterminer si ces derniers sont utiles dans cette tâche pour un utilisateur en particulier. En effet, la pertinence d'un clone dépend à la fois de la tâche et de l'utilisateur qu'il doit servir. Les développeurs n'ont par exemple pas tous les mêmes pratiques quand il s'agit de réaliser des opérations de réingénierie logicielle. Or, les *benchmarks* de clones existants (dont celui de Bellon) sont construits sans objectif particulier; il n'existe pas de relation entre leur usage et la manière dont ils sont définis. Pour cette raison, nous les appelons *benchmarks* sans contexte.

De notre première contribution, nous avons observé que la construction et la validation de *benchmarks* de clones réalisées par une seule personne ne permet pas de garantir des résultats fiables. Nous nous demandons donc si pour des clones liés à une tâche, plusieurs personnes peuvent aboutir à la construction de *benchmarks* fiables. Ainsi, notre deuxième problématique est de déterminer dans quelle mesure la méthodologie actuelle de construction de *benchmarks* peut être adaptée pour concevoir des *benchmarks* permettant une évaluation fiable des détecteurs de clones pour une tâche donnée. Notre but est de rechercher des recommandations pour faciliter la construction de *benchmarks* de clones liés à une tâche. Pour répondre à cette question de recherche, nous sollicitons quatre juges pour analyser les résultats produits par un détecteur de clones sur deux logiciels. Pour chacun des deux logiciels, trois des juges n'ont pas ou peu de connaissances sur le code du logiciel et le dernier en est l'un des principaux développeurs; il y a donc trois juges dits « externes » et un expert. L'avis de l'expert est primordial pour un *benchmark* de clones liés à une tâche car les clones identifiés par un détecteur de clones doivent être pertinents pour lui. Ainsi, les deux experts sont les oracles décidant si un clone est un vrai ou un faux positif selon la tâche considérée (dans notre cas co-évolution et réingénierie, les deux principales activités de maintenance logicielle). Les réponses des juges sont ensuite utilisées pour débattre de la fiabilité de leur avis. Finalement, nous avons soulevé plusieurs points par rapport à cette problématique [Charpentier *et al.*, 2016a] :
— l'existence de différences significatives entre l'avis des juges externes et des experts;
— une forte influence de la pratique des développeurs sur l'utilité d'un clone;
— la nécessité de disposer d'un expert du projet dont sont issus les clones pour construire un *benchmark* fiable.

1.3.3 Adaptation des techniques de détection

En l'état actuel des choses, le recours à un corpus de référence pour comparer et évaluer les résultats de détecteurs de clones n'est pas satisfaisant pour aider les utilisateurs à choisir l'outil le plus adapté à leurs besoins. Le problème est en fait que cette méthodologie d'évaluation et de comparaison n'est pas adaptée pour déterminer l'outil à utiliser pour une tâche donnée. Il faudrait en effet que les résultats dérivés d'un *benchmark* de clones puissent refléter l'intention des utilisateurs, ce qui est impossible avec des corpus de référence construits avec une définition de clones trop générale.

La difficulté à choisir l'outil le plus adapté pour une tâche de maintenance logicielle n'est peut-être pas uniquement due aux faiblesses des techniques d'évaluation et de comparaison de détecteurs de clones. Les outils de détection de clones pourraient faire partie du problème. Ces derniers pourraient en effet être considérés comme trop généralistes car ils ne disposent pas d'option de configuration pour adapter leurs résultats aux attentes des utilisateurs. Notre troisième problématique est donc de déterminer si l'utilisation de détecteurs de clones spécialisés dans une tâche peut aider les développeurs à gérer les clones présents dans leurs logiciels. Cette information supplémentaire lors de la détection de clones aurait deux effets notables. Premièrement, un outil avec de telles informations en entrée identifierait davantage de clones pertinents qu'un détecteur classique car l'utilisateur aurait spécifié ses attentes. De plus, l'évaluation et la comparaison de ces outils seraient facilitées car la classification des clones serait moins subjective du fait d'une définition plus raffinée. Le dernier volet de nos contributions se focalise donc sur les avantages à prendre en considération l'intention des utilisateurs pour améliorer l'usage des détecteurs de clones et ainsi accroître leur adoption dans les tâches de maintenance logicielle. Finalement, nos contributions à cette problématique sont les suivantes [Charpentier *et al.*, 2016b] :

— nous proposons une nouvelle approche pour automatiquement détecter des clones dans du code CSS [1] qui sont des candidats valides pour une factorisation via des mixins;
— un prototype permettant un contrôle fin sur le code généré s'adaptant aux besoins des utilisateurs;
— une évaluation de notre approche avec une méthodologie différente basée sur des études de cas. Les résultats de ces études de cas peuvent ensuite être utilisés pour construire des ensembles de vrais et faux clones dans le but de comparer les outils spécialisés dans une tâche et ainsi mieux répondre aux besoins des utilisateurs.

1. CSS pour *Cascading Style Sheet* en anglais est un langage de balisage notamment utilisé pour décrire le style visuel de documents web.

1.4 Structure

Le deuxième chapitre de ce document présente un aperçu de l'état de l'art de ce domaine, afin de positionner nos contributions vis-à-vis des travaux existants. Nous consacrons ensuite un chapitre à chacune de nos problématiques. Enfin, nous apportons une conclusion générale aux recherches menées pour cette thèse et abordons les perspectives de celles-ci.

CHAPITRE

2 Contexte, état de l'art

Nous débutons ce chapitre en définissant le contexte de cette thèse. Nous abordons la notion générale de clone et discutons des différentes définitions proposées dans la littérature. Nous présentons également les techniques de détection de clones, leurs implémentations et les domaines d'application associés.

Ensuite, nous poursuivons par la présentation de travaux liés à l'évaluation et la comparaison des outils de détection de clones. Nous détaillons notamment la construction des benchmarks *de clones, pierre angulaire de l'automatisation de l'évaluation des détecteurs de clones. Enfin, nous discutons les limites inhérentes et actuelles de ces ensembles de clones de référence.*

Ensuite, nous abordons un ensemble de travaux traitant de la manière dont les clones sont perçus par les développeurs, et dans quelle mesure leur jugement sur les clones peut affecter la construction fiable de benchmarks *de clones.*

Le dernier ensemble de travaux aborde les problématiques d'adaptation des techniques de détection de clones pour des activités de maintenance logicielle.

Sommaire

2.1 Contexte **10**

2.2 Évaluation des techniques de détection de clones **22**

2.3 Détecteurs de clones et maintenance logicielle **30**

2.4 Synthèse de l'état de l'art **33**

2.1 Contexte

Dans cette section, nous définissons le contexte de cette thèse nécessaire à la bonne appréciation de nos travaux. Nous débutons par la notion centrale de clone et ses nombreuses définitions dans la littérature. Nous poursuivons par la description du processus général d'identification de clones ainsi que la présentation des différentes techniques de détection qui ont été proposées par la communauté.

2.1.1 Définitions de clones dans la littérature

L'identification de la duplication de code logiciel est un problème de recherche qui ne dispose pas de définition précise de son sujet d'étude : les clones. D'importantes recherches sur la détection de clones sont menées sans savoir exactement ce qu'est un clone ni pour quelle tâche le détecter. La définition de clones dépend généralement de la technique de détection sous-jacente et est adaptée à l'étude menée.

Baxter *et al.* [1998] définissent des clones comme des fragments de code qui sont similaires selon une certaine définition de similarité. Leur calcul de similarité entre deux fragments est basé sur des seuils. Kamiya *et al.* [2002] utilisent une définition plus vague. Ils définissent des clones comme des portions de code qui sont identiques ou similaires les uns aux autres. Alors que le terme identique représente des copies exactes, ils ne donnent pas de définition formelle du terme similaire. Une autre définition imprécise est celle utilisée par Burd *et al.* dans leur étude sur l'évaluation d'outils de détection de clones [Burd et Bailey, 2002] : un fragment de code représente un clone s'il existe une ou plusieurs occurrences de ce fragment avec peu ou pas de modifications. Ils ne spécifient pas le niveau de modifications toléré.

Des taxonomies de clones ont été proposées pour tenter de lever les ambiguïtés soulignées ci-dessus. Mayrand *et al.* [1996] définissent une échelle ordinale de huit différents types de clone. Leur taxonomie ne suffit néanmoins pas à fournir une définition précise de clone car certaines de ses catégories sont vagues. Par exemple, une catégorie désigne des clones représentant des expressions différentes mais similaires. Une autre taxonomie est proposée par Balazinska *et al.* [1999] et comporte dix-huit catégories différentes. Elle souffre toutefois des mêmes maux que la précédente avec des catégories imprécises : par exemple, « *One long difference* », « *Two long differences* », etc.

La taille minimale des fragments de code est une autre imprécision dans la définition de clones et de fait un autre sujet à débat. Selon la technique de détection utilisée, la manière de calculer la taille d'un fragment varie. Certaines utilisent le nombre de lexèmes [Kamiya *et al.*, 2002] alors que d'autres se basent sur le nombre de lignes [Baker, 1995] ou de nœuds dans l'arbre de syntaxe abstraite [Li *et al.*, 2006] ou le graphe de dépendances [Komondoor et Horwitz, 2003]. D'autres encore ne s'intéressent qu'à des fonctions complètes quelle que soit leur taille [Mayrand *et al.*, 1996]. De plus, pour une technique de détection donnée, la taille minimale d'un fragment diffère. Par exemple, Bellon *et al.* [2007] utilisent

un nombre minimal de lignes de 6 alors que Baker [1995] fixe ce même seuil à 15 et Johnson [1994] à 50.

Ces exemples soulignent bien la difficulté de définir ce que représente un clone. Finalement, une catégorisation des clones en quatre types a été proposée [Bellon *et al.*, 2007] et s'est imposée comme une référence, malgré la subsistance d'imprécisions dans la définition de certains d'entre eux. Elle est basée sur le principe que la similarité entre fragments de code peut être exprimée soit au niveau syntaxique soit au niveau sémantique. Trois types sont dédiés à la représentation de clones syntaxiquement similaires et chacun autorise à plus ou moins de différences entre les fragments de code. Le dernier type définit des clones sémantiques où les fragments de code implémentent une même fonctionnalité écrite avec des syntaxes différentes. Cette classification des clones en quatre types définit une échelle ordinale de la complexité de détection des clones. Il est bien plus difficile d'identifier des clones de type-4 que de type-1 et les techniques de détection associées sont également bien plus complexes. Nous donnons les définitions et des exemples de chacun des quatre types de clone ci-après.

Clone de type-1

Définition 2.1 : Clone de type-1
Des clones de type-1 représentent des fragments de code identiques à l'exception des espaces, de la mise en page et des commentaires.

Les deux fragments de code du listing 2.1 illustrent des clones de type-1. Ils sont identiques à l'exception des commentaires, de la position des accolades et des espaces dans la déclaration de la variable `obsoleteForm`. Cette définition est exempte de toute interprétation et peut s'appliquer à n'importe quel langage de programmation. Notons qu'une approche de détection de clones basée uniquement sur les lignes n'identifierait pas les deux fragments du listing 2.1 comme étant des clones de type-1 à cause des différences au niveau de la mise en page.

Clone de type-2

Définition 2.2 : Clone de type-2
Des clones de type-2 représentent des fragments de code syntaxiquement identiques à l'exception des identificateurs, types, mise en page et commentaires.

Les deux fragments du listing 2.2 illustrent des clones de type-2 : la structure syntaxique reste similaire entre les deux fragments malgré le renommage de la variable `obsoleteForm` en `outofdateForm`. Cette définition est à première vue précise mais conduit à l'identification de très nombreux clones dont certains douteux. Avec une telle définition, tous

LISTING 2.1 – Exemple d'un clone de type-1.

Fragment 1 :

```
1 boolean obsoleteForm = (new Random()).nextBoolean();
2 if (obsoleteForm) {
3     System.out.println("Zenzizenzizenzic"); // Comment1
4 } else {
5     System.out.println("x^8"); // Comment2
6 }
```

Fragment 2 :

```
1 boolean obsoleteForm=(new Random()).nextBoolean();
2 if (obsoleteForm)
3 {
4     System.out.println("Zenzizenzizenzic"); // Comment1'
5 }
6 else
7 {
8     System.out.println("x^8"); // Comment2'
9 }
```

les appels d'une méthode sur un objet sont des clones : les instructions `obj1.foo();` et `obj2.bar();` forment un clone de type-2 indépendamment des objets `obj1` et `obj2` et des méthodes `foo()` et `bar()`.

Clone de type-3

Définition 2.3 : Clone de type-3
Des clones de type-3 représentent des fragments de code comportant certaines modifications : des instructions peuvent être ajoutées, supprimées ou modifiées en plus des variations au niveau des identificateurs, types, mise en page et commentaires.

Le second fragment du listing 2.3 contient deux instructions additionnelles (lignes 1 et 5). La définition de clones de type-3 possède les imperfections de celle de type-2 et en ajoute de nouvelles. Il est en effet à la charge des concepteurs d'outils de détection de clones de déterminer le nombre d'instructions pouvant être ajoutées et supprimées.

LISTING 2.2 – Exemple d'un clone de type-2.

Fragment 1 :

```
1 boolean obsoleteForm = (new Random()).nextBoolean();
2 if (obsoleteForm) {
3     System.out.println("Zenzizenzizenzic"); // Comment1
4 } else {
5     System.out.println("x^8"); // Comment2
6 }
```

Fragment 2 :

```
1 boolean outofdateForm=(new Random()).nextBoolean();
2 if (outofdateForm)
3 {
4     System.out.println("Zenzizenzizenzic"); // Comment1'
5 }
6 else
7 {
8     System.out.println("x^8"); // Comment2'
9 }
```

Clone de type-4

Définition 2.4 : Clone de type-4
Des clones de type-4 représentent des fragments de code réalisant le même calcul mais implémentés avec des syntaxes différentes.

Dans le premier fragment du listing 2.4, la fonction `fact` calcule la factorielle de son argument `n` et dans le second la variable `j` contient *in fine* la factorielle de la variable `n`. D'un point de vue sémantique, ces deux fragments de code réalisent donc le même traitement malgré qu'un soit une fonction complète et l'autre une simple suite d'instructions. Ils représentent bien des clones de type-4.

Les définitions de clones de type-4 et type-1 sont les seules exemptes de toute interprétation. Cependant en pratique la détection de clones de type-4 est autrement plus complexe que celle de type-1 et requiert des techniques bien plus élaborées.

LISTING 2.3 – Exemple d'un clone de type-3.

Fragment 1 :

```
boolean obsoleteForm = (new Random()).nextBoolean();
if (obsoleteForm) {
    System.out.println("Zenzizenzizenzic"); // Comment1
} else {
    System.out.println("x^8"); // Comment2
}
```

Fragment 2 :

```
Random random = new Random();
boolean obsoleteForm = random.nextBoolean();
if (obsoleteForm)
{
    System.out.println("Obsolete form:");
    System.out.println("Zenzizenzizenzic"); // Comment1'
}
else
{
    System.out.println("x^8"); // Comment2'
}
```

Autres dénominations

D'autres dénominations que les quatre types définis précédemment apparaissent dans la littérature pour décrire les clones. Pour la plupart d'entre elles, il est toutefois possible de trouver une correspondance avec l'un des quatre types de clone. Nous ne citons ici que les deux plus usitées pour remplacer les types, à savoir les termes anglais *exact clones* et *near-miss clones*. Le terme *exact clones* correspond aux clones de type-1 et le terme *near-miss clones* recouvre les clones de type-2 et type-3. Dans la suite de cette thèse, nous utiliserons uniquement les dénominations de types de clone.

Terminologie

Définition 2.5 : Paire de clones
Une paire de clones est une paire de fragments de code qui sont identiques ou similaires selon la définition de similarité considérée.

LISTING 2.4 – Exemple d'un clone de type-4.

Fragment 1 :

```
1 int fact (int n) {
2     if (n == 0) return 1;
3     else return n * fact(n - 1);
4 }
```

Fragment 2 :

```
1 int j = 1;
2 for (int i = 1; i <= n; i++)
3     j = j * i;
```

Une paire de clones définit une relation de similarité entre fragments de code dupliqués qui est une relation d'équivalence : réflexive, transitive et symétrique.

Synthèse

Nous avons montré dans cette section qu'il n'existe aucune définition universelle de clone et que les recherches sur le sujet adaptent la notion selon leurs besoins. La tentative la plus aboutie de précision de la notion de clone est la catégorisation en quatre types permettant une distinction entre les similarités syntaxique et sémantique. Cette définition fondée sur des types est la plus utilisée dans la littérature bien que la spécification de certains de ces types comporte toujours des imprécisions. Une conséquence directe à l'absence de définition formelle de clone est l'apparition de faux positifs et faux négatifs lors de la recherche de duplication de code logiciel. Les faux positifs correspondent à des résultats incorrects et dénotent la précision d'une recherche de clones. À l'inverse, les faux négatifs sont de vrais clones qui n'ont pas été détectés et dénotent le rappel d'une recherche de clones.

2.1.2 Processus de détection de clones

La détection de clones a pour objectif de trouver des portions de code logiciel identiques ou similaires. Cette recherche de fragments de code dupliqués est complexe d'une part à cause de l'absence d'une définition formelle comme nous l'avons souligné dans les sections précédentes et d'autre part à cause de la taille du domaine de comparaison. Les fragments identiques ou similaires d'un logiciel ne sont pas connus à l'avance car la du-

plication de code logiciel est un processus non documenté qui a des origines diverses. Par conséquent, un outil de détection de clones a la charge de contrôler toutes les paires de fragments possibles. Une telle comparaison est très coûteuse en temps de calcul quand la taille du logiciel analysé est très importante et que la technique de détection utilisée est complexe. Nous verrons dans la section suivante que certaines techniques de détection permettant notamment l'identification de clones sémantiques ont de grandes complexités. Concentrons-nous tout d'abord sur le processus général de détection de clones. Nous présentons ici les phases principales suivies par à peu près toutes les techniques de détection.

Pré-traitement. La première phase consiste à déterminer le domaine de comparaison. Le code source pour lequel il n'est pas pertinent de rechercher des clones est supprimé. Par exemple, le code généré est le plus souvent éliminé avant de passer à la phase suivante. Ensuite, le code source est partitionné en un ensemble d'unités de comparaison qui représentent les plus grands fragments de code qui peuvent intervenir dans un clone : fichiers, classes, fonctions, instruction, etc. Ces unités de comparaison définissent la granularité des clones qui devront être recherchés.

Transformation. Les unités de comparaison sont transformées dans une représentation intermédiaire adaptée à un algorithme de comparaison donné. Une normalisation du code source peut éventuellement précéder cette transformation. Par exemple, presque toutes les techniques de détection de clones suppriment les commentaires et ignorent les différences au niveau des espaces. D'autres remplacent l'ensemble des identificateurs par un seul et même identificateur afin de faciliter la détection de clones de type-2.

Comparaison. Les unités de comparaison transformées sont données en entrée de l'algorithme de détection de clones et comparées les unes aux autres pour trouver des correspondances, c'est-à-dire des clones. Les unités de comparaison similaires adjacentes sont agrégées afin d'identifier des correspondances de longueur maximales. Le résultat produit est une liste de paires de clones contenant chacune les informations de localisation des fragments de code. Un clone est en règle générale représenté par un couple de la forme : $\{(fichier1, ligneDebut1, ligneFin1), (fichier2, ligneDebut2, ligneFin2)\}$ où $fichier1$ représente le nom du fichier contenant un des deux fragments et $ligneDebut1$ et $ligneFin1$ identifient respectivement les lignes de début et de fin de ce fragment dans le fichier. Les informations de l'autre fragment sont stockées de manière analogue dans le second élément du couple.

Post-traitement. Au cours de cette phase, un filtrage des résultats peut être opéré afin de mettre de côté les faux positifs, c'est-à-dire les clones non pertinents. Ce filtrage est un processus manuel qui peut être accéléré par l'utilisation d'outils de visualisation. Finalement,

les paires de clones peuvent être agrégées en classes afin de réduire la quantité de données à traiter.

Un point intéressant à noter ici est que le filtrage des faux positifs dans les résultats produits repose sur le jugement d'une ou plusieurs personnes. Ainsi, il est susceptible de varier selon les individus qui le réalisent. Nous portons une attention toute particulière à ce point dans la suite de ce chapitre.

2.1.3 Techniques et outils de détection de clones

De nombreuses approches ont été proposées dans la littérature afin d'identifier des clones dans les logiciels. Ces approches peuvent être classées en quatre catégories selon le niveau d'analyse appliqué sur le code source : textuelle, lexicale, syntaxique et sémantique. Dans cette section, nous présentons un état de l'art des différentes techniques de détection en fonction de ces quatre approches et montrons que les clones identifiés varient d'une technique à l'autre.

Approche textuelle

Dans une approche textuelle, le code source est considéré comme une séquence de lignes ou de chaînes de caractères. Une détection de clones avec une telle approche consiste alors à rechercher des sous-séquences similaires dans la séquence initiale. Le listing 2.5 donne un exemple d'une paire de clones qu'une approche textuelle est en mesure de détecter. Cette approche identifie aussi bien des clones de type-1, type-2 et type-3 selon que la comparaison entre les sous-séquences est exacte ou approchée (à l'aide de seuils par exemple). Les clones de type-3 peuvent également être obtenus en concaténant des clones de type-1 ou type-2 qui sont proches dans la séquence initiale; la proximité maximale autorisée entre deux clones pour former un clone de type-3 peut généralement être paramétrée par un seuil défini par l'utilisateur. Les clones identifiés sont nécessairement des séquences similaires de taille maximale. Dans l'exemple du listing 2.5, il y a bien une seule paire de clones détectée correspondant aux deux fragments de trois lignes chacun.

Avant la recherche effective de clones, les techniques textuelles peuvent opérer des transformations sur le code source. Les commentaires et les espaces sont le plus souvent supprimés et selon le niveau de connaissance du langage de programmation utilisé par le code source à analyser, des normalisations élémentaires peuvent également être réalisées telles que l'uniformisation des chaînes de caractères, nombres, identificateurs ou types primitifs. Le listing 2.6 illustre un exemple de normalisation : les espaces et sauts de ligne ont été enlevés, les identificateurs ont été remplacés par `i`, le type primitif `int` par `num` et les nombres et chaînes de caractères par une valeur arbitraire. Cet exemple de normalisation requiert un minimum d'informations sur le langage de programmation du code source à analyser (mots-clés et types primitifs notamment).

LISTING 2.5 – Exemple d'une paire de clones identiques détectés à l'aide d'une approche textuelle.

Fragment 1 :

```
1 int i = 1;
2 float pi = 3.14;
3 String question = "Question";
```

Fragment 2 :

```
1 int i = 1;
2 float pi = 3.14;
3 String question = "Question";
```

LISTING 2.6 – Normalisation basique d'un fragment de code.

Fragment avant normalisation :

```
1 int i = 1;
2 int j = 2;
3 String question = "Question";
```

Fragment après normalisation :

```
1 num i=1;num i=1;String i="...";
```

Le principal avantage des techniques textuelles est la simplicité : aucun analyseur syntaxique n'est par exemple requis pour détecter des clones. Tout langage de programmation peut ainsi être supporté aisément. De plus, les algorithmes de comparaison sont simples et efficaces permettant l'analyse de millions de lignes de code. Les limitations tiennent aux faits que les techniques textuelles disposent d'un ensemble de normalisations restreint et qu'elles ne peuvent garantir d'identifier des clones qui sont alignés sur la structure du code. Toutefois, si les délimiteurs de blocs du langage de programmation considéré sont connus, un pré-traitement ou un post-traitement peut permettre de filtrer les résultats pour n'identifier que des clones correspondant à des blocs syntaxiques.

Nous présentons ici quelques techniques notables basées sur le texte qui ont été proposées dans la littérature. Ducasse *et al.* considèrent le code d'un programme comme une

séquence de lignes et utilisent une fonction de hachage pour accélérer leur comparaison [Ducasse *et al.*, 1999]. Le résultat de ces comparaisons est alors visualisé dans un graphique : un point à la coordonnée (x, y) signifie que la $x^{ème}$ ligne est identique à la $y^{ème}$ ligne. Les clones maximaux correspondent alors aux diagonales dans un tel graphique. De plus, les diagonales contenant des trous représentent quant à elles de potentiels clones de type-3. Ducasse *et al.* utilisent finalement le filtrage par motif afin d'automatiser la détection de clones dans ce type de graphique. Une extension des recherches de Ducasse *et al.* est proposée par Wettel et Marinescu [2005] afin d'identifier des clones approchants (*near-miss clones*). Leur algorithme rattache ensemble les lignes voisines ayant la même valeur de hachage détectant ainsi des clones de type-3.

Approche lexicale

Une approche lexicale est identique à une approche textuelle à la différence que le code source est considéré comme une suite de lexèmes et non comme une séquence de lignes ou de chaînes de caractères. Une détection de clones basée sur une approche lexicale consiste à rechercher des sous-séquences dupliquées dans la suite initiale de lexèmes et à rapporter pour chaque ensemble de sous-séquences similaires le code source correspondant comme un clone. Une approche lexicale identifie les mêmes types de clone qu'une approche textuelle et peut opérer les mêmes transformations et normalisations sur le code source original. Elle possède donc les mêmes avantages et limitations. Notons toutefois qu'une approche lexicale est généralement plus robuste à des changements mineurs dans le code qu'une approche textuelle.

Baker fut l'un des premiers à proposer une technique efficiente de détection de clones basée sur les lexèmes [Baker, 1995]. Son outil, nommé *Dup*, utilise un analyseur lexical pour produire les lexèmes correspondant au code source à analyser puis opère des normalisations sur ces derniers afin de permettre la détection de clones de type-2. Chaque ligne ainsi transformée est comparée à l'aide d'un algorithme reposant sur les arbres de suffixes. La technique de Baker est en mesure d'identifier des clones de type-1 et type-2. Des clones de type-3 sont trouvés en concaténant des clones de type-1 ou type-2 qui sont proches ; la distance maximale autorisée entre deux clones pour une concaténation est fixée par l'utilisateur. *CCFinderX* [Kamiya *et al.*, 2002], développé par Kamiya *et al.*, est le plus connu des outils de détection de clones fondés sur les lexèmes. Les lexèmes de tous les fichiers du code source à analyser sont concaténés en une seule séquence de lexèmes. Cette dernière est ensuite transformée selon des règles dépendant du langage de programmation considérée afin de normaliser les identificateurs et faciliter l'identification des différentes structures syntaxiques. Comme pour l'outil *Dup*, un algorithme basé sur les arbres de suffixes est utilisé pour la recherche effective des clones.

Approche syntaxique

Avec les approches syntaxiques, le code source est transformé en arbre syntaxique ou en arbre syntaxique abstrait. Ces représentations du code source autorisent des normalisations plus avancées avec notamment la possibilité de distinguer fonctions, types et variables. De plus, l'utilisation de l'une de ces deux représentations pour la recherche de clones garantit de produire des résultats alignés sur la structure du code. Les algorithmes pour mener une détection de clones avec l'une de ces représentations sont basés soit sur les arbres soit sur des métriques de code calculées sur les logiciels à analyser.

Les techniques basées sur les arbres identifient des clones comme des sous-arbres similaires de taille maximale. Les feuilles des arbres qui correspondent en fait à des lexèmes peuvent être abstraites afin de permettre une détection de clones plus avancée. Baxter *et al.* font partie des pionniers dans les techniques de détection de clones fondées sur les arbres avec leur outil *CloneDR* [Baxter *et al.*, 1998]. Ils génèrent un arbre syntaxique abstrait annoté du code source à analyser et classent les sous-arbres en différents ensembles à l'aide d'une fonction de hachage dans le but de minimiser le nombre des comparaisons de sous-arbres à réaliser. Ils définissent également une fonction de similarité pour la comparaison des sous-arbres permettant la détection de clones similaires et non uniquement identiques. Le code source des sous-arbres similaires est finalement retourné comme clones. Jiang *et al.* ont proposé un autre algorithme pour l'identification de sous-arbres similaires et l'ont implémenté dans un outil appelé *Deckard* [Jiang *et al.*, 2007]. Ils définissent des vecteurs caractéristiques pour capturer des informations structurelles des arbres. Le vecteur caractéristique d'un sous-arbre est un point $< c_1, \ldots, c_n >$ dans l'espace euclidien $\mathbb{R}^n$. Ils utilisent ensuite un algorithme pour grouper ces vecteurs par rapport à la métrique de distance euclidienne. Finalement, les sous-arbres avec des vecteurs dans le même groupe sont considérés similaires.

Le calcul de métriques sur le code source peut également servir à la détection de clones. Une technique populaire reposant sur les métriques consiste à calculer une empreinte pour chaque unité syntaxique telle qu'une classe, fonction et instruction. Les unités qui possèdent des empreintes identiques sont rapportées comme clones. Dans la plupart des cas, le code source est d'abord transformé en arbre syntaxique abstrait ou en graphe de flot de contrôle puis les métriques sont calculées sur ces représentations. Mayrand *et al.* proposent une technique fondée sur les métriques pour détecter des clones correspondant à des corps de fonction entiers [Mayrand *et al.*, 1996]. Afin de déterminer si deux corps de fonction sont similaires, ils utilisent leur nom, mise en page, flot de contrôle, nombre d'expressions qu'elles contiennent ainsi que leur nature et leur complexité.

Approche sémantique

Des approches sémantiques ont été proposées afin de considérer davantage que la seule similarité syntaxique. Elles reposent sur l'analyse statique de programmes pour obte-

nir des informations encore plus précises sur le code source à analyser. Généralement dans ces approches le code source est transformé en un graphe de dépendances. Les nœuds de ce graphe sont les expressions et les instructions alors que les arêtes représentent les dépendances entre les données et entre les structures de contrôle. La recherche de clones revient alors à trouver des sous-graphes isomorphes (pour lesquels seuls des algorithmes approximatifs efficaces existent). Le principal inconvénient d'une approche sémantique est le passage à l'échelle.

Krinke utilise une approche itérative pour détecter des sous-graphes maximaux similaires dans le graphe de dépendances d'un programme [Krinke, 2001]. Gabel *et al.* proposent une technique efficiente de détection de clones sémantiques en réduisant le problème de similarité de graphes à un problème de similarité d'arbres [Gabel *et al.*, 2008]. Komondoor et Horwitz ont un autre outil de détection de clones basé sur la recherche de sous-graphes isomorphes dans les graphes de dépendances de programmes.

Approche hybride

Des approches hybrides combinent plusieurs des approches que nous avons présentées ci-dessus pour tenter de tirer le meilleur parti de chacune d'elles. NiCad [Cordy et Roy, 2011], développé par Cordy et Roy, est probablement le détecteur de clones le plus connu reposant sur une approche hybride. Il utilise une analyse syntaxique simple afin de normaliser et formater le code source considéré puis réalise une comparaison textuelle pour la recherche effective de clones. Leitão fournit une approche hybride qui combine les techniques syntaxiques et sémantiques à travers une combinaison de fonctions de comparaison spécialisées [Leitão, 2004]. L'outil R^2D^2 implémente cette approche. Chaque fonction de comparaison explore différents aspects (sous-graphes similaires, opérateurs commutatifs, équivalences prédéfinies par l'utilisateur, transformations du code dans des formes syntaxiques canoniques, etc.) et donne une probabilité que deux fragments de code forment un clone. La combinaison de plusieurs fonctions de comparaison accroît la confiance dans les clones identifiés. Koschke *et al.* proposent une amélioration des complexités en temps et en espace pour les techniques reposant sur les arbres [Koschke *et al.*, 2006]. Leur approche consiste à sérialiser les nœuds de l'arbre de syntaxe abstraite généré à partir du code source à analyser afin de permettre une recherche de clones à l'aide d'arbres de suffixe (approche lexicale). Cette idée leur permet une détection de clones syntaxiques aussi rapide qu'une approche lexicale.

Détecteurs de clones

Nous avons montré dans cette section qu'il existe à la fois un grand nombre de techniques de détection de clones et d'implémentations. La table 2.1 liste les principaux outils de détection de clones ainsi que l'approche utilisée par chacun d'eux. Les outils de détection ont tous plusieurs options de configuration chacune disposant de nombreuses valeurs

TABLEAU 2.1 – Liste des principaux outils de détection de duplication de code.

Outil	**Approche**	**Langage(s) Supporté(s)**	**Référence**
CCFinderX	Lexicale	C, C++, COBOL, Java	[Kamiya *et al.*, 2002]
CLAN	Syntaxique	C	[Mayrand *et al.*, 1996]
CloneDR	Syntaxique	C, C++, COBOL, Java	[Baxter *et al.*, 1998]
D-CCFinder	Lexicale	C, C++, COBOL, Java	[Livieri *et al.*, 2007]
Deckard	Syntaxique	C, Java	[Jiang *et al.*, 2007]
Dup	Lexicale	C, C++, Java	[Baker, 1992]
Duplix	Sémantique	C	[Krinke, 2001]
DupLoc	Textuelle	*langage indépendant*	[Ducasse *et al.*, 1999]
iClones	Lexicale	ADA, C, C++ Java	[Göde et Koschke, 2009]
NiCad	Hybride	C, C#, Java, Python	[Cordy et Roy, 2011]
SourcererCC	Lexicale	C, C#, Java	[Sajnani *et al.*, 2015]

possibles. Pour une performance optimale de l'outil, ces options doivent être ajustées avec précision selon le logiciel considéré. Ce problème de configuration des outils de détection de clones est bien connu dans la littérature : selon une évaluation menée par Wang *et al.* [2013], sur 185 études empiriques sur les clones, 113 (61%) reconnaissent que la configuration des outils de détection de clones qu'ils ont utilisés pourrait impacter leurs résultats. Ainsi, pour les utilisateurs d'outils de détection de clones, une difficulté est de déterminer l'outil et la configuration à utiliser pour une tâche particulière. Ceci suppose alors de pouvoir comparer et évaluer les résultats des différents outils de détection de clones. Nous détaillons dans la section suivante les défis que représente l'évaluation de détecteurs de clones.

2.2 Évaluation des techniques de détection de clones

Plusieurs expérimentations sur l'évaluation et la comparaison d'outils de détection de clones sont disponibles dans la littérature. Différents aspects peuvent être évalués : portabilité, précision, rappel, passage à l'échelle, etc. La validation des techniques et outils de détection de clones est primordiale pour déterminer le ou les outils les plus adaptés à une tâche donnée. Privé de ce type d'information, le choix d'un détecteur de clones par un utilisateur relèverait du défi tant les outils sont nombreux et leurs résultats différents. Le gain de l'utilisation d'un détecteur de clones serait alors fortement limité.

L'une des premières études sur l'évaluation d'outils de détection de clones a été menée par Burd et Bailey [2002]. Ils ont évalué les résultats de deux outils de détection de plagiat et trois outils de détection de clones : *CCFinder* [Kamiya *et al.*, 2002], *CloneDR* [Baxter *et al.*, 1998] et *Covet* (une ré-implémentation d'un outil développé par Bailey et Mayrand [1996]).

Le sujet d'étude pour leur expérimentation est le code source de GraphTool, un outil Java de taille moyenne (63 fichiers pour un total de 16 335 lignes de code) développé au sein de leur université. Afin de faciliter la comparaison des résultats produits par les cinq détecteurs étudiés, ceux-ci sont traduits en un nouveau format où chaque clone est représenté par une paire de fragments similaires, eux-mêmes décrits par le nom du fichier dont ils sont issus et les numéros de début et fin de ligne. Ensuite, l'ensemble des clones identifiés par les cinq détecteurs (1 463) est contrôlé manuellement pour déterminer les vrais et les faux positifs. Dans cette étude, Bailey et Burd considèrent un clone comme étant un vrai positif si celui-ci est approprié à une tâche de maintenance. Pour rendre la phase de classification manuelle moins subjective, ils définissent quatre critères permettant de juger si un clone est un vrai ou un faux positif. Ainsi, un clone est approprié à une tâche de maintenance si ses deux fragments ont soit des commentaires identiques ou similaires, soit des variables identiques ou similaires, soit un nom de méthode identique ou similaire, soit un flot de contrôle et une mise en page identique ou similaire. Une fois cette phase réalisée, les mesures de précision et de rappel sont calculées pour chacun des cinq détecteurs par rapport aux nombres de vrais et faux positifs relevés. La précision d'un détecteur de clones donne le ratio de clones identifiés qui correspondent effectivement à des vrais positifs. Le rappel donne la proportion de tous les clones présents dans le logiciel considéré qui ont été effectivement détectés. La table 2.2 rapporte les résultats des mesures de précision et de rappel obtenues par Burd et Bailey. L'outil *CloneDR* montre une précision de 100% indiquant qu'il ne produit pas de faux positifs. Toutefois, cet outil a identifié uniquement 9% de l'ensemble des vrais clones qui ont été considérés pour l'évaluation. Ce faible taux de rappel indique que ce dernier ne semble pas adapté lorsque l'objectif est de trouver une majorité des clones présents dans un logiciel. L'outil *Covet* quant à lui propose un meilleur rappel mais a une précision moindre. Les deux outils de détection de plagiat, *JPlag* et *Moss*, ont sensiblement les mêmes mesures de précision et de rappel. Finalement, l'outil *CCFinder*, basé sur une approche lexicale, semble être le plus approprié pour la recherche de clones en proposant à la fois une précision et un rappel raisonnables. Les résultats obtenus par Bailey et Burd doivent toutefois être considérés avec précaution car ils se basent sur un seul projet Java. Bien que la taille de ce dernier leur ait permis de vérifier manuellement tous les clones détectés par les cinq outils, il n'est pas représentatif de l'ensemble des logiciels. De plus, les clones ont été jugés pour une certaine tâche (aide à la maintenance), ce qui a pu influencer leur validation.

Comme nous l'avons montré avec l'étude de Burd et Bailey [2002], la classification des clones en vrais et faux positifs est une étape clef dans l'évaluation d'outils de détection de clones car elle impacte directement les mesures de précision et de rappel. Des études ont donc été menées sur l'automatisation de la comparaison et de l'évaluation de détecteurs de clones. Le but recherché est de disposer de méthodes fiables et efficaces permettant aux développeurs d'outils de détection de clones de se comparer à la concurrence et aux utilisateurs de choisir le détecteur le plus adapté à leur besoin. Cette automatisation se base sur des ensembles de clones de référence (*benchmarks*). Les ensembles de référence

TABLEAU 2.2 – Résultats de l'étude menée par Burd et Bailey.

	CCFinder	**CloneDR**	**Covet**	**JPlag**	**Moss**
Nombre de clones identifiés	1 128	84	278	131	120
Rappel	72%	9%	19%	12%	10%
Précision	72%	100%	63%	82%	73%

générés par des humains sont une solution pour établir des standards de performance en l'absence d'une définition précise d'exactitude. Cette idée a notamment été utilisée en ingénierie inverse et en maintenance ([Girard *et al.*, 1997]) mais les corpus de référence se retrouvent partout où se posent des problèmes définis vaguement ou subjectivement : médecine, linguistique, analyse d'images, recherche d'informations, etc. Pour l'évaluation et la comparaison d'outils de détection de clones, les résultats sont comparés aux *benchmarks* qui contiennent uniquement des clones dont l'état est connu : vrai ou faux positif. Ensuite, les mesures de précision et de rappel sont calculées. Ces deux mesures sont essentielles pour un outil de détection de clones car elles déterminent la qualité de ses résultats. Elles ne peuvent être calculées sans le support d'un ensemble de clones de référence. Dans la suite de cette section, nous détaillons tout d'abord les différentes manières de construire des *benchmarks* de clones dans la littérature puis nous discutons du problème de l'opinion des développeurs sur les clones.

2.2.1 Construction de *benchmarks* de clone

Bellon *et al.* ont proposé le premier *benchmark* de clones [Bellon, 2002; Bellon *et al.*, 2007] qui reste à ce jour le plus utilisé comme support à l'évaluation d'outils de détection de clones [Ducasse *et al.*, 2006; Koschke *et al.*, 2006; Wang *et al.*, 2013; Murakami *et al.*, 2014]. Nous ne donnons ici que les informations nécessaires et suffisantes sur ce *benchmark* pour le comparer aux autres proposés dans la littérature. En effet, nous détaillons la construction et l'utilisation de ce *benchmark* dans la section 3.2.1. Pour la construction de ce *benchmark*, Bellon a classifié seul 2% des 325 935 clones identifiés par six outils de détection de clones sur huit gros logiciels C et Java. À partir des clones qu'il a jugés comme étant des vrais positifs, il a construit un corpus de référence. Ce dernier est utilisé pour le calcul des mesures de précision et de rappel des détecteurs de clones et ainsi pour leur évaluation et comparaison.

Krutz et Le décrivent une méthodologie pour la construction de *benchmarks* de clones correspondant à des méthodes complètes ainsi que les données qu'ils ont obtenues avec celle-ci [Krutz et Le, 2014]. Ils considèrent trois logiciels *open-source* pour la recherche de vrais clones à inclure dans leur *benchmark* : Apache 2.2.14, Python 2.5.1 et PostgreSQL 8.5. Pour chacun de ces logiciels, ils sélectionnent aléatoirement entre trois et six classes et énumèrent toutes les paires de fonctions possibles pour celles-ci. Ils appliquent ensuite

plusieurs outils de détection de clones (Simcad [Uddin *et al.*, 2013], NiCad [Roy et Cordy, 2008], MeCC [Kim *et al.*, 2011] et CCCD [Krutz et Shihab, 2013]) sur les 45 109 paires créées et sélectionnent 1 536 clones ainsi générés pour une évaluation manuelle. Ils font appel à trois chercheurs ayant une expérience sur les clones et à quatre étudiants pour la classification des clones. Le groupe des chercheurs et celui des étudiants ont réalisé l'expérimentation séparément et personne avaient connaissance de la provenance des clones. Chaque clone est discuté au sein des deux groupes, résultant soit en un accord (vrai ou faux positif) soit en un état dit *incertain*. La méthodologie de construction de *benchmarks* de clones proposée par Krutz et Le est plus rigoureuse que celle de Bellon *et al.* [2007] mais comporte des limitations fortes. Tout d'abord, la taille de leur *benchmark* est nettement inférieure à celui de Bellon, ce qui pourrait avoir un impact sur son utilité. De plus, leur *benchmark* ne contient que des clones correspondant à des méthodes complètes et à du code écrit avec le langage de programmation C. Il ne peut donc servir que pour évaluer des détecteurs de clones capables d'analyser du code C.

Svajlenko *et al.* proposent *BigCloneBench*, un *benchmark* de clones inter-projets [Svajlenko *et al.*, 2014] dans le dépôt logiciel IJaDataset contenant des projets Java. Leur *benchmark* a la particularité de capter à la fois des clones syntaxiques et sémantiques et d'être construit indépendamment de tout outil de détection de clones. Il n'est donc pas limité aux clones que des détecteurs de clones sont capables d'identifier et est donc adapté pour relever les faiblesses des techniques de détection de clones. En fait, leur *benchmark* contient des clones implémentant des fonctionnalités similaires dans le dépôt logiciel IJaDataset. Pour ajouter une nouvelle fonctionnalité (et les clones correspondant) à leur *benchmark*, il est nécessaire d'identifier les différentes manières de l'implémenter avec le langage de programmation Java. Pour cela, ils inspectent les discussions sur internet (*Stack Overflow* par exemple) et la documentation des interfaces de programmation (*JavaDoc* par exemple). Ainsi, ils construisent une spécification précise de la fonctionnalité étudiée. À partir de cette spécification, ils définissent une heuristique de recherche afin de trouver des fragments de code dans IJaDataset qui devraient implémenter la fonctionnalité considérée. Comme IJaDataset contient vingt-quatre millions de méthodes, une recherche manuelle est en effet impossible. Tous les fragments de code ainsi identifiés sont manuellement classés par des juges en vrais et faux clones pour la fonctionnalité donnée à l'aide de la spécification. Tous les clones ainsi classés sont injectés dans le *benchmark* et ce dernier évolue donc à mesure que de nouvelles fonctionnalités sont considérées. Finalement, les mesures de rappel et de précision peuvent être calculées de manière globale ou pour une fonctionnalité donnée. *BigCloneBench* est le seul benchmark qui considère des clones relatifs à un certain contexte qui est en fait une fonctionnalité. Il contient un peu plus de six millions de vrais clones et les quatre types de clone sont représentés. Deux cents seize heures ont été nécessaires à trois juges pour valider l'ensemble de ces clones.

Roy et Cordy proposent une méthode automatique pour évaluer empiriquement les outils de détection de clones [Chanchal K. Roy et James R. Cordy, 2009] qui utilise la mutation de fragments de code pour artificiellement créer des clones et les injecter ensuite dans

un ensemble de référence afin de mesurer efficacement le rappel et la précision de détecteurs de clones. Leur cadriciel (*framework* en anglais) se base sur une taxonomie d'édition qui synthétise des clones artificiels correspondant à des clones que des développeurs auraient pu introduire. Leur cadriciel est capable d'évaluer et de comparer le rappel de détecteurs de clones pour de nombreux langages et types de clone sans intervention manuelle. Il peut mesurer la précision automatiquement à partir de règles de validation dépendantes des langages de programmation des logiciels étudiés. Roy et Cordy ont défini quatorze opérateurs de mutation qui couvrent les quatre types de clone (par exemple : insertion d'une ou plusieurs lignes). Leur cadriciel pour l'évaluation a deux phases principales : la génération et l'évaluation. La première phase débute par la création de versions mutantes de fragments de code, c'est-à-dire obtenues à partir de la taxonomie d'édition. Des fragments dans le code source des sujets de l'étude sont aléatoirement sélectionnés pour subir une mutation ; leur nombre est prédéterminé. Ces fragments peuvent être de n'importe quelle granularité et peuvent être sélectionnés manuellement. Ensuite, une combinaison fixée ou aléatoire d'opérateurs de mutation est appliquée à chacun des fragments sélectionnés pour créer plusieurs versions mutantes. Finalement, chaque fragment mutant est injecté aléatoirement dans le code source initial. La phase d'évaluation consiste à vérifier que le ou les outils de détection de clones en cours d'étude sont en mesure d'associer chaque fragment mutant à sa version originale. Les mesures de précision et de rappel sont ensuite calculées et utilisées pour l'évaluation et la comparaison des détecteurs de clones. Le principal avantage de l'approche de Roy et Cordy est qu'elle ne nécessite pas une validation des clones car tous les clones générés par leur processus de mutation sont par hypothèse de vrais positifs. Ils s'affranchissent ainsi des problèmes de subjectivité dans la classification de clones. Toutefois, comme ils le soulignent dans leur étude, leur taxonomie d'édition ne garantit pas de créer des clones correspondant à une tâche donnée, telle que la réingénierie logicielle ou la co-évolution.

Les caractéristiques des quatre *benchmarks* discutés ci-dessus sont synthétisées dans la table 2.3. La première observation que nous pouvons faire est que le nombre de clones varie fortement d'un *benchmark* à l'autre. Celui construit par Roy et Cordy est à mettre de côté sur ce point précis car les clones servant de référence sont générés pour chaque logiciel analysé. De plus, trois *benchmarks* sur quatre considèrent uniquement des clones généraux, dans le sens où les clones qu'ils contiennent ne sont liés à aucune tâche particulière, c'est-à-dire que leur validation ne dépend pas d'un contexte tel que la réingénierie logicielle et la co-évolution. Notons que le *benchmark* de Svajlenko *et al.* ne répond pas complètement à ce critère ; leurs clones sont liés à une fonctionnalité (tri à bulles par exemple) plutôt qu'à une tâche de maintenance ou d'évolution logicielle. La disponibilité de *benchmarks* de clones contextuels (valides par rapport à une tâche donnée) est importante car elle garantit de pouvoir évaluer les détecteurs de clones par rapport à une tâche et donc de déterminer quel outil est le plus adapté à cette dernière. De même, les *benchmarks* semblent se focaliser sur les vrai positifs et de fait se désintéresser des faux positifs. Les deux informations sont pourtant intéressantes pour mesurer la qualité des résultats

produits par un outil de détection de clones. Un autre point de divergence pour ces *benchmarks* est la provenance des clones. Le *benchmark* de Bellon contient uniquement des clones qui ont été identifiés par des détecteurs de clones. Ceci est une forte limitation pour mesurer le rappel des détecteurs de clones car l'ensemble de clones de référence correspond à ce que les outils à évaluer sont capables d'identifier. Les trois autres *benchmarks* qui ont été construits après celui de Bellon *et al.* sont passés outre cette limitation. À l'inverse, le *benchmark* proposé par Roy et Cordy contient des clones générés qui ne correspondent pas nécessairement à des clones existants dans de vrais logiciels. Ce fait est inhérent à leur méthodologie mais est également une limitation. Enfin, à l'exception de Roy et Cordy qui considèrent par hypothèse que tous les clones produits par leur taxonomie sont de vrais positifs, la validation des clones est similaire. Elle est manuelle et est réalisée généralement par une seule personne. Toutefois, la validation des clones pour ces trois *benchmarks* n'est jamais faite avec l'aide d'un expert des logiciels dont sont issus les clones. Pourtant, les clones détectés dans un logiciel doivent être pertinents pour les développeurs et mainteneurs de ce logiciel. Nous avons montré précédemment que la pertinence d'un clone dépend de l'utilisateur et de la tâche à réaliser. Dans un *benchmark* de clones contextuels, l'avis d'un expert est primordial car les résultats d'un outil de détection de clones doivent être pertinents pour lui. Il a de fait le rôle d'oracle décidant si un clone est un vrai ou un faux positif selon la tâche considérée.

2.2.2 Opinion des développeurs sur les clones

Nous avons montré précédemment que la validation des clones était une opération manuelle réalisée le plus souvent par une seule personne. Si la validation d'un clone diverge d'une personne à l'autre et est donc subjective, la fiabilité des *benchmarks* pourrait être remise en question notamment pour ceux où la validation a été réalisée par une seule personne. Il est connu depuis longtemps qu'il existe plusieurs écueils à l'utilisation d'ensembles de référence générés par des humains [Landis et Koch, 1977; Rust et Cooil, 1994]. Tout d'abord, plusieurs juges peuvent ne pas être d'accord sur la même réponse ; le niveau d'accord entre les juges est généralement appelé la fiabilité des jugements. Des problèmes de fiabilité peuvent remettre en cause la signification et la validité des ensembles de référence générés par des humains. Ensuite, une définition particulière donnée aux juges peut influencer leurs résultats ou la fiabilité de leurs résultats [Koschke et Eisenbarth, 2000]. Plusieurs études ont ainsi été menées pour évaluer le problème de la subjectivité comme le potentiel obstacle à la validité et à la qualité des *benchmarks* de clones existants. Nous présentons dans cette section ces études empiriques.

L'étude menée par Walenstein *et al.* [2003] est précurseuse dans le domaine de l'évaluation de la fiabilité des développeurs dans la construction de *benchmarks* de clones. Les auteurs présentent les facteurs qui peuvent influer sur la construction d'ensembles de clones de référence fiables liés ou non à une tâche. Malgré l'aspect exploratoire de leur étude qui empêche la généralisation de leurs observations, ils soulèvent plusieurs points

TABLEAU 2.3 – Résumé des caractéristiques des quatre *benchmarks* de clones discutés.

	Bellon[1]	**Krutz**[2]	**Svajlenko**[3]	**Roy**[4]
Général				
Nombre de clones	4 319	1 536	6 164 953	∞
Vrais et faux positifs	non	non	oui	non
Représentatif d'une tâche donnée	non	non	oui	non
Provenance des clones				
Clones issus d'un logiciel	oui	oui	oui	non
Nombre de logiciels utilisés	8	3	25 000	-
Clones détectés par un outil	oui	non	non	non
Validation				
Validation manuelle	oui	oui	oui	non
Nombre de personnes impliquées	1	7	1	-
Auteurs de l'étude	oui	non	oui	-
Validation avec un expert	non	non	non	non

[1] [Bellon *et al.*, 2007].
[2] [Krutz et Le, 2014].
[3] [Svajlenko *et al.*, 2014].
[4] [Chanchal K. Roy et James R. Cordy, 2009].

qui indiquent que la manière actuelle de construire des *benchmarks* pourrait conduire à des résultats qui ne sont pas fiables. Ils examinent deux questions : 1) quels critères de classification des clones seraient appropriés pour l'analyse comparative des détecteurs de clones ? et 2) la fiabilité est-elle un problème dans le cadre de la construction de *benchmark* de clones ? Afin de répondre à ces questions, Walenstein *et al.* ont récupéré les clones d'une étude menée par Bellon [2002]. Au cours de quatre phases distinctes, les auteurs de l'étude conduisent des sessions individuelles et collectives pour déterminer si les clones étudiés (qui correspondent tous à des fonctions complètes) sont utiles, c'est-à-dire valides, pour une tâche de réingénierie logicielle. Pour chaque phase, les conditions d'acceptation d'un clone sont raffinées et l'accord entre les auteurs est enregistré. Finalement, ils observent que selon la tâche considérée la construction d'un ensemble de clones de référence est plus ou moins aisée. Ils soulignent l'importance de mener des recherches sur l'identification des tâches qui posent le plus de difficulté lors de la création de *benchmarks* de clones contextuels. De plus, selon leur expérimentation les réponses d'un juge unique sont nécessairement biaisées. Ils suggèrent alors que les précédentes évaluations de la précision et du rappel d'outils de détection de clones doivent être interprétées avec précaution. Ils soutiennent que la fiabilité inter-juges devrait être calculée pour les ensembles de clones de référence construits par des humains. En marge des deux questions de recherche ini-

tiales, leur étude montre également que la fiabilité inter-juges dépend du logiciel dont sont issus les clones à analyser.

Kapser *et al.* ont mené une étude pour mesurer le degré d'accord entre experts sur les clones quand il s'agit de classer des fragments de code comme des vrais clones ou non [Kapser *et al.*, 2007]. Vingt clones ont été sélectionnés comme base à cette expérimentation; ils ont été identifiés par l'outil CCFinder [Kamiya *et al.*, 2002] sur le code source du logiciel PostgreSQL. Chacun des huit participants a évalué individuellement ces vingt clones en utilisant sa propre définition de ce qu'il considère être un clone. Les participants ont veillé à ne pas discuter des définitions de clone avant le début de la phase de classification. Une fois cette première phase terminée par les huit participants, les vingt clones ont été réévalués conjointement par le groupe de participants. En conséquence, sur les vingt clones analysés, seulement la moitié a été classée de la même manière par au moins 87% (correspondant à sept personnes) des participants. Quand le standard d'accord est diminué à 75% (c'est-à-dire six personnes sur huit), les participants sont d'accord pour 65% des clones. L'étude révèle également une liste des raisons pour lesquelles deux fragments de code devraient être considérés comme des clones ou non. Cette liste a été établie lors d'une session de discussion et est en fait une simple concaténation des raisons de chacun des participants. Un point intéressant à noter est que les participants ont des avis divergents sur ces raisons. Considérons, par exemple, l'une d'entre elles : « le code est idiomatique ». Pour certains participants, cette raison implique que deux fragments forment un clone alors que pour d'autres c'est l'inverse. Finalement, les auteurs soulignent l'importance de rapporter le ou les critères utilisés pour juger les clones dans les papiers de recherche afin de permettre la comparaison avec les travaux précédents.

À l'inverse des précédentes études indiquant la difficulté de classer des clones et la subjectivité de cette tâche, Mende *et al.* observe un accord raisonnable entre les juges au cours de l'évaluation de leur technique de détection de clones [Mende *et al.*, 2009] qui est basée à la fois sur une approche lexicale et un calcul de métriques avec la distance de Levenshtein. Mende *et al.* font appel à neuf juges pour évaluer leur approche : cinq sont des chercheurs et quatre sont des étudiants diplômés d'une licence. Les juges évaluent les mêmes paires de clones et un scénario concret leur est donné pour décider si oui ou non deux fonctions forment un clone. La tâche fictive était de décider si deux fonctions devraient être combinées dans une ligne de produits logiciels. Finalement, les auteurs mesurent l'accord entre les juges et trouvent un accord raisonnable entre les différents participants dans leur jugement si deux fonctions sont suffisamment similaires pour être combinées. Cette observation sur l'accord entre juges est différente des précédentes et pourrait s'expliquer par le fait que dans l'évaluation de Mende *et al.* seuls des clones correspondant à des fonctions complètes sont considérés.

Synthèse

Dans cette section, nous avons abordé les différents aspects des *benchmarks* de clones dont le but est l'évaluation et la comparaison de détecteurs de clones. Leur construction diverge mais possède toujours des limitations. La principale est la validation des clones qu'ils contiennent. Nous avons en effet présenté plusieurs études qui indiquent que la validation de clones est une tâche subjective. La plupart des *benchmarks* ayant été construits par généralement une seule personne qui n'est pas un expert des logiciels dont sont issus les clones, la fiabilité des résultats dérivés de leur utilisation peut être remise en question. D'autre part, aucune étude empirique n'a été menée sur l'évaluation des *benchmarks* actuels. Notons néanmoins à ce propos les travaux de Svajlenko et Roy [2014] qui mettent en lumière le fait que différents *benchmarks* produisent différentes évaluations de détecteurs de clones.

2.3 Détecteurs de clones et maintenance logicielle

La maintenance logicielle est connue pour être une phase importante et coûteuse dans le développement logiciel. Des techniques existent pour assister les développeurs lors de cette phase. L'une d'entre elles, la réingénierie, est efficace pour améliorer la maintenabilité des logiciels. Or, de par leur nature, les clones sont des candidats de choix pour une réingénierie. Par conséquent, les détecteurs de clones et plus généralement la détection de clones ont un rôle à jouer lors de la maintenance logicielle. De plus, même quand l'élimination des clones par réingénierie est impossible, leur identification a toujours un intérêt. Un exemple de cet intérêt est la propagation d'une correction d'un bogue à un ensemble de fragments de code dupliqués. Dans la suite de cette section, nous présentons des travaux sur l'utilisation des outils de détection de clones pour réaliser des tâches de maintenance logicielle. Nous soulignons notamment leurs limites dans la réalisation de telles tâches et ainsi dans l'aide qu'ils peuvent apporter aux développeurs.

Van Rysselberghe et Demeyer ont mené une étude sur l'adéquation des techniques de détection de clones aux tâches du processus de maintenance logicielle [Van Rysselberghe et Demeyer, 2004]. Ils ont examiné quatre questions de recherche, chacune soulignant des fonctionnalités importantes au cours du processus de maintenance logicielle. Leur première question examine la quantité de travail nécessaire pour adapter une technique donnée à un contexte de programmation particulier. La deuxième évalue le type des résultats produits par une technique donnée ; selon la tâche de maintenance le type de duplication recherché varie. Par exemple, durant une phase d'évaluation des problèmes, les mainteneurs souhaitent plutôt obtenir une vue d'ensemble de la quantité de duplication existant dans les fichiers sources. D'un autre côté, au cours d'une phase de réingénierie logicielle, les mainteneurs sont davantage intéressés par identifier des constructions qui peuvent être factorisées par l'outil de réingénierie à disposition. Ainsi, pour un outil de réingénierie spé-

cialisé dans le déplacement de méthodes dans une hiérarchie de classes, seuls les corps de méthodes dupliqués devraient être identifiés. La troisième question concerne la justesse des résultats produits par chaque technique et la dernière mesure leur temps d'exécution. Afin de répondre à ces questions, Van Rysselberghe et Demeyer ont développé un prototype en Java pour chacune des quatre techniques de détection de clones qu'ils souhaitaient évaluer et les ont ensuite appliquées sur cinq logiciels Java de tailles moyennes (moins de dix mille lignes de code). Leurs conclusions sur l'utilisation de ces techniques dans le cadre du processus de maintenance logicielle sont les suivantes. L'approche textuelle est utile pour donner un aperçu du code dupliqué dans un logiciel. Elle est donc appropriée lors des phases de détection et d'évaluation des problèmes. Les techniques lexicales et celles basées sur les métriques sont adaptées à certaines opérations de réingénierie : *extract method* et *pull-up method* notamment. Van Rysselberghe et Demeyer concluent que les différentes techniques de détection de clones décrites dans la littérature présentent chacune des avantages spécifiques par rapport aux autres. Chaque technique semble en effet plus appropriée pour une certaine tâche de maintenance.

L'outil de détection de clones CCFinder [Kamiya *et al.*, 2002] sert de base à de nombreux outils de réingénierie logicielle. Higo *et al.* proposent une méthode [Higo *et al.*, 2004a] qui élimine les clones présents dans des logiciels orientés objet en utilisant les techniques de réingénierie existantes, notamment *extract method* et *pull up method*. Leur méthodologie repose sur l'usage de CCFinder pour l'identification des clones à éliminer. Comme les clones détectés par CCFinder sont des séquences de lexèmes, ils ne sont pas nécessairement tous de bons candidats pour une fusion en un seul module (sous-programme, fonction, etc.), c'est-à-dire qu'ils ne sont pas tous adaptés à une réingénierie. Pour palier ce problème, les auteurs proposent une méthode qui extrait les clones candidats à une réingénierie de l'ensemble des résultats produits par CCFinder. Leur méthode est implémentée comme un filtre sur la sortie de l'outil CCFinder et se nomme CCShaper. Le point important à noter au sujet de cette étude de Higo *et al.* est que les résultats du détecteur de clones CCFinder ne sont pas directement utilisables pour réaliser des opérations de réingénierie dans des logiciels orientés objet ; un filtrage des clones détectés est en effet nécessaire. Le caractère généraliste de CCFinder implique qu'une partie de ses résultats sont non pertinents pour certaines tâches. Higo *et al.* proposent un autre outil, nommé Aries, pour l'aide à la réingénierie dans des logiciels orientés objet [Higo *et al.*, 2004b]. Leur outil est toujours fondé sur l'analyse de clones et utilise CCShaper en interne pour la phase de détection. Il offre une interface graphique affichant de nombreuses informations sur les clones détectés. Les utilisateurs utilisent alors ces informations pour choisir les clones qu'ils souhaitent fusionner. Pour chaque clone à supprimer, Aries indique l'ensemble des techniques de réingénierie applicables. Ainsi, Aries est un outil qui s'adapte aux besoins de chaque utilisateur. L'ensemble des clones éliminés par un utilisateur grâce à l'outil Aries diffère plus ou moins fortement de l'ensemble initial de clones identifiés par CCFinder. Des travaux similaires ont été menés par Choi *et al.* [2011]. Ces derniers proposent une méthode pour extraire de la sortie de l'outil CCFinder un ensemble de clones candidats à une réingé-

nierie. Une fois encore, les résultats de CCFinder ne sont pas directement utilisables pour proposer des suppressions de clones pertinentes dans le cadre d'une maintenance logicielle.

CCFinder n'est pas le seul détecteur de clones utilisé pour identifier des opportunités de réingénierie. NiCad [2011] est utilisé dans les travaux de Mondal *et al.* [2014] et iClones [Göde et Koschke, 2009] dans ceux de Wang et Godfrey [2014]. Dans ces deux exemples, le détecteur de clones sert de support à un autre outil qui lui identifie les vrais possibilités de réingénierie. Ainsi, comme pour CCFinder les résultats des outils NiCad et iClones ne sont pas directement exploitables par les développeurs pour réaliser des tâches de réingénierie logicielle. Une autre difficulté à l'usage de détecteurs de clones généralistes tels que CCFinder, Nicad et iClones pour des tâches spécifiques telles que la réingénierie est la configuration de l'outil. Wang *et al.* ont montré que les résultats d'un détecteur de clones dépendent fortement de la configuration utilisée [Wang *et al.*, 2013]. Le problème est d'autant plus important que les paramètres de configuration de ces outils ne sont pas adaptés à des tâches spécifiques, ils sont trop bas niveau. Par exemple pour iClones, quel devrait être le nombre minimal de lexèmes des deux fragments d'un clone ? La valeur est probablement à adapter au cas par cas. À ce propos, Mondal *et al.* note dans leur étude [Mondal *et al.*, 2014] que leurs observations pourraient être modifiées par une configuration différente de l'outil NiCad.

Göde présente une étude de cas sur l'élimination délibérée de clones par les développeurs [Göde, 2010]. Son objectif est d'aborder la duplication du point de vue d'un mainteneur logiciel afin d'acquérir des connaissances pour améliorer les outils de réingénierie reposant sur l'analyse de clones. Pour atteindre ce but, Göde a étudié l'évolution des clones de quatre projets Java et C++ grâce à une version améliorée de son algorithme de détection incrémental [Göde et Koschke, 2009]. Pour ces quatre projets, il a manuellement contrôlé 977 fragments de code pour rechercher des suppressions volontaires de clones. Finalement, il a observé de nombreuses situations où la duplication de code a été volontairement éliminée. L'analyse de ces suppressions de clones a révélé que l'extraction de méthode est la technique de réingénierie la plus fréquemment utilisée pour éliminer la duplication. De plus, Göde a observé que les clones ne sont presque jamais supprimés dans leur entièreté. Selon lui, cela pourrait indiquer soit que les développeurs ne sont pas au courant de toute l'étendue de la duplication, soit que les résultats des détections de clones sont trop imprécises. Il conclut son étude de cas en soulignant l'existence d'un écart entre les clones identifiés par un détecteur de clones classique et la manière dont les mainteneurs abordent la duplication.

Les détecteurs de clones peuvent être utilisés dans le cadre de la maintenance logicielle pour des tâches autres que l'élimination de la duplication. Celle-ci n'est en effet pas toujours possible. L'identification de fragments de code dupliqués s'avère utile pour des tâches de co-évolution (propagation d'une correction de bogue par exemple). Cependant, les détecteurs de clones actuels ne sont pas adaptés à de telle tâches. Leur exécution produit en effet trop de résultats même sur des logiciels de moyenne taille. Déterminer

le sous-ensemble de clones pertinents parmi l'ensemble des clones détectés par un outil serait bien trop long. Dans la construction de son *benchmark* [Bellon *et al.*, 2007], Bellon a passé 44 heures pour contrôler 3 260 clones. Une fois encore, les possibilités de configuration des détecteurs de clones actuels se révèlent insuffisantes. Ainsi, pour des tâches de co-évolution un post-traitement des résultats produits par un détecteur est également nécessaire pour ne proposer aux utilisateurs que des clones pertinents.

Synthèse
Dans cette section, nous avons montré que les résultats des outils de détection de clones ne sont pas directement utilisables par les développeurs pour réaliser des tâches de maintenance logicielle. Au minimum, un filtrage des résultats est nécessaire pour proposer aux utilisateurs un ensemble pertinent de clones sur lequel ils peuvent alors travailler. Nous avons également souligné que les détecteurs de clones classiques ont des paramètres de configuration trop bas niveau et sont par conséquent inadaptés pour des tâches de maintenance logicielle. Leur aspect trop généraliste est donc un frein à leur adoption par les développeurs.

2.4 Synthèse de l'état de l'art

L'état de l'art que nous avons présenté dans ce chapitre apporte de nombreuses informations concernant les techniques de détection de clones et les méthodologies permettant l'évaluation et la comparaison des détecteurs de clones.

Au vu de cet état de l'art, il apparaît que :

- la correction des *benchmarks* actuels n'a pas été évaluée à notre connaissance dans la littérature. Cependant, l'évaluation de la fiabilité et de la validité de ces *benchmarks* est nécessaire pour confirmer les résultats des nombreuses études qui en dépendent pour évaluer ou comparer des outils de détection de clones ;
- la fiabilité de l'opinion des développeurs sur les clones a été étudiée dans le cadre de clones non contextuels (valides quelle que soit la tâche considérée). Or, dans la construction d'ensembles de référence par des humains, l'utilisation d'une définition particulière par des juges peut avoir un impact sur les résultats [Koschke et Eisenbarth, 2000]. Une évaluation de la fiabilité des développeurs face à des clones contextuels (liés à une tâche particulière) est donc nécessaire pour définir une méthodologie fiable de construction de *benchmarks* de clones contextuels ;
- les détecteurs de clones actuels ne produisent pas des résultats directement utilisables pour des tâches de maintenance logicielle. Le caractère généraliste de ces outils en est la raison et explique leur faible adoption par les développeurs. La spécialisation des détecteurs de clones semble donc nécessaire pour accroître l'utilisation de ceux-ci dans des tâches de maintenance logicielle. De tels outils spécialisés

ont par ailleurs déjà été proposés dans la littérature. Nous pouvons par exemple citer les travaux de Mazinanian *et al.* sur une technique pour éliminer la duplication de code dans des fichiers CSS [Mazinanian *et al.*, 2014].

CHAPITRE

3 Évaluation empirique d'un *benchmark* de clones

La construction d'un benchmark *de clones est généralement assurée par une seule personne, décidant elle-même quel clone est ou n'est pas un vrai positif. Cependant, cette méthodologie ne tient pas compte de la subjectivité de la notion de clone. D'autres personnes pourraient être en désaccord sur la validité de certains des clones inclus dans un* benchmark *et construire de fait d'autres* benchmarks *avec des résultats possiblement différents. Notre objectif dans ce chapitre est de déterminer si la construction et la validation de* benchmarks *par une seule personne garantissent des résultats fiables. Pour cela, nous menons une évaluation empirique d'un* benchmark *largement utilisé par la communauté de chercheurs. Nous sollicitons l'avis de dix-huit participants sur un sous-ensemble de ce* benchmark *qui a été construit et validé par une seule personne afin d'évaluer sa correction. Notre expérimentation montre qu'une part significative des clones qu'il contient sont contestables, et que ce phénomène peut introduire du bruit dans les mesures faites à partir de ce* benchmark.

Sommaire

3.1 Introduction **36**
3.2 Contexte **37**
3.3 Méthodologie **42**
3.4 Résultats et discussion **44**
3.5 Validité de l'étude **53**
3.6 Conclusion **53**

3.1 Introduction

Le *benchmark* de Bellon [2007] est apparu comme le principal *benchmark* pour comparer et évaluer les détecteurs de clones [Ducasse *et al.*, 2006; Koschke *et al.*, 2006; Nguyen *et al.*, 2012; Wang *et al.*, 2013]. Bellon a construit un corpus de référence de 4 319 vrais clones, sélectionnés manuellement en regardant deux pour cent des 325 935 clones reportés par six outils de détection de clones sur huit logiciels C et Java. Les résultats des outils de détection de clones sont comparés à ce corpus de référence et évalués en utilisant les traditionnelles mesures de précision et de rappel.

Cependant, un obstacle préoccupant à la validité de ce corpus de référence est qu'il a été construit et validé par une seule personne, à savoir Bellon. Or, comme ce dernier n'est pas un expert des logiciels contenant les clones et que la définition de clones est subjective, il est envisageable que d'autres personnes ne considèrent pas certains des clones du corpus de référence comme étant des vrais positifs. Ce phénomène pourrait alors biaiser les mesures de précision et de rappel calculées à l'aide de ce *benchmark*.

Dans ce chapitre, nous examinons si la construction et la validation de *benchmarks* par une seule personne peuvent produire des résultats fiables. Pour cela, nous sollicitons l'avis de dix-huit nouvelles personnes sur un sous-ensemble du *benchmark* de Bellon et examinons les trois questions de recherche suivantes :

1. *Les chercheurs peuvent-ils avoir confiance dans le corpus de référence de Bellon ?*
2. *Les mesures de précision et de rappel calculées à partir du* benchmark *de Bellon sont-elles fiables ?*
3. *Certaines caractéristiques des clones rendent-elles plus facile l'accord entre les juges ?*

Les contributions de l'étude décrite dans ce chapitre sont donc les suivantes :

1. Nous conduisons une expérience contrôlée avec neuf groupes de deux participants. Pour chaque groupe, nous sélectionnons aléatoirement 120 clones du corpus de référence de Bellon et demandons aux deux participants s'ils considèrent ces clones comme des vrais positifs ou non.
2. Nous analysons s'il existe des clones du corpus de référence pour lesquels les participants et Bellon ont des avis divergents.
3. Nous examinons si les clones pour lesquels il n'y a pas de consensus entre les participants et Bellon peuvent avoir un impact sur les valeurs des mesures de précision et de rappel calculées à partir de ce *benchmark*.
4. Nous évaluons si certaines caractéristiques des clones sont corrélées à un niveau de confiance plus élevé.

La suite de ce chapitre est organisée comme suit. La section 3.2 détaille le *benchmark* de Bellon et notre problème de recherche. Ensuite, nous décrivons la méthodologie de notre

TABLEAU 3.1 – Participants à la construction du *benchmark*.

Participant	Outil	Approche
Brenda S. Baker	Dup	Lexicale
Ira D. Baxter	CloneDR	Syntaxique
Toshihiro Kamiya	CCFinder	Lexicale
Jens Krinke	Duplix	Sémantique
Ettore Merlo	CLAN	Syntaxique
Matthias Rieger	Duploc	Textuelle

étude empirique dans la section 3.3. La section 3.4 présente les résultats de notre expérience contrôlée. Nous discutons de la validité de cette étude dans la section 3.5. Enfin, nous concluons dans la 3.6.

3.2 Contexte

Dans cette section, nous débutons par une présentation détaillée du *benchmark* de Bellon, de sa construction à son utilisation. Nous terminons par les questions de recherche examinées dans cette étude.

3.2.1 *Benchmark* de Bellon

Le *benchmark* de Bellon a été construit comme un support à la comparaison et l'évaluation des outils de détection de clones. Six chercheurs, concepteurs d'outils de détection de clones, ont aidé Bellon dans la construction de ce *benchmark*. La table 3.1 présente ces six participants.

Définition de clones

Les participants se sont tous accordés à ne considérer que des paires de clones syntaxiquement similaires. Ils ont distingué trois types de clones (type-1, type-2 et type-3) qui sont ceux que nous avons définis dans le chapitre 2. À l'inverse des clones de type-1 et type-2, la définition de clones de type-3 est vague. Certains outils considèrent par exemple que deux clones de type-1 ou type-2 consécutifs forment un clone de type-3. Les participants ont donc décidé de considérer comme étant de type-3 tous les clones identifiés par un outil qui ne seraient ni de type-1 ni de type-2. Il incomberait alors à un ou plusieurs humains de déterminer si les candidats de type-3 sont de vrais clones ou non. Les participants ont ajouté une contrainte à la définition de clones : les clones doivent pouvoir être remplacés par des appels de fonction, c'est-à-dire qu'ils doivent être syntaxiquement complets (par

TABLEAU 3.2 – Logiciels utilisés pour la construction du *benchmark*.

Logiciel	**Langage de programmation**	**Taille du logiciel** [1]
weltab	C	11K SLOC
cook	C	80K SLOC
snns	C	115K SLOC
postgresql	C	235K SLOC
netbeans-javadoc	Java	19K SLOC
eclipse-ant	Java	35K SLOC
eclipse-jdtcore	Java	148K SLOC
j2sdk1.4.0-javax-swing	Java	204K SLOC

exemple un fragment qui débute dans le corps d'une fonction ne peut terminer dans celui d'une autre).

Construction

Les six participants ont chacun appliqué leur propre outil de détection de clones sur le même ensemble de logiciels. Le code source de ces logiciels a été préalablement normalisé (par exemple suppression des lignes vides) afin de faciliter la comparaison des résultats produits par les différents outils. Les participants ont tous approuvé cette normalisation ainsi que le fait de ne reporter que des clones d'au moins six lignes de code. La table 3.1 montre le détecteur de clones utilisé par chaque participant et la table 3.2 liste les huit logiciels C et Java considérés. Finalement, l'application de ces six outils sur ces huit projets a généré 325 935 clones. Leur distribution par projet est donnée dans la deuxième colonne de la table 3.3.

Bellon a examiné seul deux pour cent des 325 935 clones et a construit seul un corpus de référence en ne conservant que les clones qu'il a jugés comme étant des vrais positifs. Il s'est autorisé à étendre les fragments de code de certains clones avant de les insérer dans le corpus. Lors de l'examen d'un clone, il ne savait pas quel outil l'avait détecté. La classification de tous les clones lui a pris soixante dix-sept heures. Le nombre de clones qu'il a conservés par projet est reporté dans la dernière colonne de la table 3.3. Ainsi, selon le jugement de Bellon, le corpus de référence contient 4 319 vrais clones. Dans la suite de ce chapitre, chaque clone identifié par un outil de détection de clones est nommé un candidat et chaque clone du corpus de référence une référence.

1. La taille des logiciels est donnée en termes de ligne de code (SLOC en anglais).

TABLEAU 3.3 – Nombre de clones proposés, examinés et conservés pour les huit logiciels.

Logiciel	Clones proposés	Clones examinés	Clones conservés
weltab	13 901	280	252
cook	27 122	544	402
snns	66 331	1 329	903
postgresql	59 114	1 182	555
netbeans-javadoc	7 860	159	55
eclipse-ant	2 440	51	30
eclipse-jdtcore	92 905	1 856	1 345
j2sdk1.4.0-javax-swing	56 262	1 127	777
Total	325 935	6 528	4 319

Utilisation

Le *benchmark* de Bellon fournit un ensemble de vrais clones — le corpus de référence — pour évaluer et comparer les résultats des outils de détection de clones. Cette évaluation est un processus en deux étapes. Premièrement, il est nécessaire de vérifier que les candidats (l'ensemble *Cands*) ont une correspondance dans le corpus de référence (l'ensemble *Refs*). Deux ensembles émergent alors : *DetectedRefs* contenant les candidats ayant une correspondance dans le corpus de référence et *RejectedCands* contenant ceux qui n'en ont pas. Pour calculer ces correspondances, Bellon *et al.* ont défini une méthodologie basée sur le nombre de lignes communes entre un candidat et une référence. Deuxièmement, les mesures de précision et de rappel sont calculées. Nous donnons ci-dessous les définitions de rappel et de précision utilisées par Bellon *et al.* Le rappel est alors le nombre de candidats de type τ détectés par l'outil T dans le projet P ayant une correspondance divisé par le nombre de clones du corpus de référence pour le projet P avec le type τ. Et, la précision est le nombre de candidats de type τ détecté par l'outil T dans le projet P ayant une correspondance divisé par le nombre total de candidats identifiés dans le projet P par l'outil T avec le type τ.

Définition 3.1 :

$$Recall(P,T,\tau) = \frac{|DetectedRefs(P,T,\tau)|}{|Refs(P,\tau)|}$$

Définition 3.2 :

$$Precision(P,T,\tau) = \frac{|DetectedRefs(P,T,\tau)|}{|Cands(P,T,\tau)|}$$

Puisque pour un projet donné le nombre de clones examinés est faible par rapport au nombre de clones proposés (voir deuxième et troisième colonnes de la table 3.3), la précision calculée avec la définition précédente n'est qu'une borne inférieure de la vraie précision. Par conséquent, Bellon *et al.* ont proposé une autre mesure pour évaluer et comparer les outils de détection de clones : la proportion de clones rejetés. Cette mesure est obtenue en divisant le nombre de candidats de type τ détecté par l'outil T dans le projet P n'ayant pas de correspondance par le nombre total de candidats de l'outil T dans le projet P avec le type τ qui ont été examinés par Bellon (l'ensemble *OracledCands*).

Définition 3.3 :

$$Rejected(P,T,\tau) = \frac{|RejectedCands(P,T,\tau)|}{|OracledCands(P,T,\tau)|}$$

Le *benchmark* de Bellon dans la littérature

Nous présentons ici quelques recherches notables sur les clones faisant usage du *benchmark* construit par Bellon *et al.* De nombreuses études utilisent en effet le *benchmark* de Bellon et font de celui-ci une référence pour l'évaluation et la comparaison d'outils de détection de clones.

Ducasse *et al.* présentent une approche textuelle légère afin d'identifier des portions de code source dupliquées [Ducasse *et al.*, 2006]. Ils utilisent les clones du corpus de référence de Bellon pour comparer le rappel de leur outil à ceux de Baker [1995] et Kamiya [2002]. Ils trouvent que leur technique de détection de clones atteint généralement un haut rappel et une précision acceptable. Murakami *et al.* proposent une approche lexicale de détection de clones [Murakami *et al.*, 2012], et l'ont implémentée dans un outil nommé FRISC. Ils utilisent le *benchmark* de Bellon comme un ensemble de vrais clones pour évaluer cet outil. Ils trouvent que leur outil détecte davantage de vrais clones que n'importe quel autre outil évalué dans le *benchmark* de Bellon. Nguyen *et al.* proposent JSync, un outil de gestion de clones [Nguyen *et al.*, 2012]. Ils utilisent le *benchmark* de Bellon pour mener plusieurs tests empiriques. Comme le corpus de référence de Bellon contient uniquement 2% des 325 935 candidats soumis, ils l'utilisent seulement pour comparer le rappel entre les outils et non pour mesurer le rappel absolu de chacun des outils. Wang *et al.* examinent le problème du choix de configuration des outils de détection de clones [Wang *et al.*, 2013], et proposent une approche pour trouver automatiquement des configurations pour des outils de détection de clones qui maximisent un critère donné. Ils évaluent leur approche en menant

une étude empirique à large échelle en s'appuyant sur le *benchmark* de Bellon. Selim *et al.* présentent une technique de détection de clones hybride basée sur la transformation du code source [Selim *et al.*, 2010]. Ils examinent les performances de leur technique en utilisant le *benchmark* de Bellon. Selim *et al.* ont manuellement classé durant huit jours les clones produits par des outils non présents dans le *benchmark* de Bellon. En ce sens, ils l'ont étendu. Koschke *et al.* introduisent une approche de détection de clones basée sur les arbres de suffixes [Koschke *et al.*, 2006]. Ils comparent leur approche avec celles présentes dans le *benchmark* de Bellon. Comme Selim *et al.* [2010], ils ont étendu le corpus de référence de Bellon avec des clones produits par de nouveaux outils. Murakami *et al.* proposent une méthode pour détecter une certaine forme de clones de type-3 [Murakami *et al.*, 2013]. Ils se basent sur le corpus de référence de Bellon pour leur évaluation et ajoutent des informations à chacun des clones afin de connaître les lignes ajoutées, supprimées ou modifiées. Ils observent que la précision des détecteurs de clones est améliorée lorsque ces nouvelles informations sont utilisées.

3.2.2 Questions de recherche

Dans les formules de précision et de rappel que nous avons reportées ci-dessus, l'ensemble *DetectedRefs* est calculé à partir du contenu du corpus de référence. Ce dernier a donc un impact fort sur les valeurs de précision et de rappel calculées à partir du *benchmark* de Bellon et par conséquent sur l'évaluation et la comparaison des outils de détection de clones.

Cependant, un obstacle à la validité de ce corpus de référence est qu'il a été construit par une seule personne. D'autres personnes auraient pu construire un corpus de référence différent en s'appuyant sur les mêmes définitions. Bellon *et al.* ont d'ailleurs souligné dans leur étude [Bellon *et al.*, 2007] que leurs résultats dépendent du jugement de Bellon. Par conséquent, nous nous interrogeons sur la confiance à accorder à ce corpus de référence et de fait dans quelle mesure d'autres personnes seraient d'accord que les clones qu'il contient sont effectivement de vrais clones. Dans cette étude, nous examinons donc les questions de recherche suivantes afin d'évaluer le corpus de référence de Bellon.

Question de recherche 1. *Les chercheurs peuvent-ils avoir confiance dans le corpus de référence de Bellon ?* Pour répondre à cette question, nous sollicitons l'avis de dix-huit nouveaux juges sur un sous-ensemble du corpus de référence et évaluons si tous les considèrent eux-aussi comme de vrais clones.

Question de recherche 2. *Les mesures de précision et de rappel calculées à partir du* benchmark *de Bellon sont-elles fiables ?* Pour répondre à cette question, nous distinguons les clones pour lesquels il y a un consensus entre Bellon et les nouveaux juges des autres et

évaluons l'effet que cette distinction pourrait avoir sur les mesures de précision et de rappel.

Question de recherche 3. *Certaines caractéristiques des clones rendent-elles plus facile l'accord entre les juges?* Pour répondre à cette question, nous étudions trois caractéristiques des clones et recherchons si les clones associés à l'une d'entre elles sont toujours évalués comme de vrais positifs par Bellon et les nouveaux juges.

3.3 Méthodologie

Dans cette section, nous décrivons l'expérience que nous avons menée pour évaluer le *benchmark* de Bellon. Nous postulons qu'un clone est fiable et par conséquent peut être intégré dans un corpus de référence si quiconque à qui il est présenté ne doute pas qu'il s'agit d'un vrai clone. Par conséquent, nous soumettons un sous-ensemble du corpus de référence de Bellon à de nouveaux juges et collectons leur opinion sur ces clones. Dans la suite de cette section, nous présentons les participants à cette étude, la manière dont nous avons sélectionné un sous-ensemble du corpus de référence de Bellon, la manière dont nous avons recueilli l'opinion des juges et enfin la manière dont nous avons affecté un niveau de confiance à chacun des clones examinés.

3.3.1 Participants

Nous demandons à dix-huit étudiants d'examiner les clones du corpus de référence. Quatre sont des étudiants en doctorat et quatorze sont des étudiants en école d'ingénieur. Trois des étudiants en doctorat sont de Singapour (*Singapore Management University*) et le dernier de Bordeaux (université). Les quatorze étudiants sont dans leur dernière année d'étude au département télécommunications de l'institut polytechnique de Bordeaux. Tous ont une formation en informatique et maîtrisent les langages de programmation C et Java.

3.3.2 Sélection des clones et examen

La première étape consiste à déterminer le nombre de clones que chaque participant doit analyser. La principale contrainte est que les quatorze étudiants en école d'ingénieur doivent pouvoir finir l'expérimentation en un seul créneau horaire, afin de garantir qu'ils ne communiquent pas et ne s'influencent pas les uns les autres. Afin d'évaluer ce seuil maximal de clones à analyser, nous avons mené une expérience préliminaire avec un membre de notre équipe de recherche. À partir des observations faites lors de cette expérience préliminaire, nous avons décidé de fixer à 120 le nombre de clones par partici-

pant. Environ deux heures sont nécessaires pour évaluer 120 clones sans avoir besoin de se presser et sans être lassé de l'expérience.

L'objectif de notre expérience étant d'évaluer si plusieurs personnes ont le même avis sur un ensemble de clones, nous devons assigner au moins deux participants à chacun des clones à analyser. Nous avons donc sélectionné de manière aléatoire 1 080 clones ($18 \times 120 \div 2$) parmi les 4 319 contenus dans le corpus de référence de Bellon. Ainsi, les participants évaluent approximativement un quart de ce corpus de référence. Les 1 080 clones sont ensuite divisés en neuf groupes de 120 chacun évalué par un groupe de deux étudiants choisis aléatoirement.

Les quatorze étudiants en école d'ingénieur ont réalisé l'expérience dans une salle de l'institut polytechnique de Bordeaux sous l'encadrement de deux des auteurs de l'étude, afin notamment de garantir qu'ils ne communiquent pas entre eux. Un créneau horaire de trois heures leur a été consacré pour assurer que tous aient le temps d'analyser la totalité des clones. À l'inverse, les étudiants en doctorat ont réalisé l'expérimentation directement sur internet. Nous avons échangé par courrier électronique avec eux et leur avons demandé de s'accorder au plus un créneau de trois heures et de ne pas communiquer les uns avec les autres.

3.3.3 Récupération de l'opinion des participants

Dans cette étude, nous demandons à des participants de classer un ensemble de clones issus du corpus de référence de Bellon, puis de répondre anonymement à un questionnaire sur les clones qu'ils ont examinés. Une étape clef de cette expérience est de collecter l'opinion des participants. Pour cet objectif, nous avons développé une interface web, illustrée dans la figure 3.1, permettant de réaliser des sondages sur des clones. Il est demandé aux participants de répondre *oui* pour les clones qu'ils jugent comme étant des vrais positifs et *non* pour les autres.

Pour chaque clone, l'interface web affiche les deux fichiers impliqués dans le clone et surligne en jaune le fragment de code correspondant au clone dans chaque fichier. Les différences entre les deux fragments sont montrées en orange. L'interface web est capable de sauvegarder et de restaurer une session de travail et autorise la navigation dans l'ensemble des clones à juger pour la mise à jour des réponses.

Nous avons mené une expérience contrôlée avec les étudiants en école d'ingénieur. Au début de la session de classification des clones, nous avons présenté les définitions de clones utilisées par Bellon [2007] et nous avons indiqué aux étudiants qu'ils pouvaient s'y référer à tout moment au cours de l'expérience. Nous n'avons toutefois pas mentionné que les clones que nous leur proposions avaient été jugés par Bellon comme de vrais positifs. Nous leur avons dit à la place que ces clones avaient été aléatoirement sélectionnés à partir des résultats d'un outil de détection de clones. Ensuite, nous avons expliqué en détail comment fonctionne l'interface web. Pour les étudiants en doctorat, nous avons donné toutes ces consignes par courrier électronique et leur avons fait confiance pour veiller à

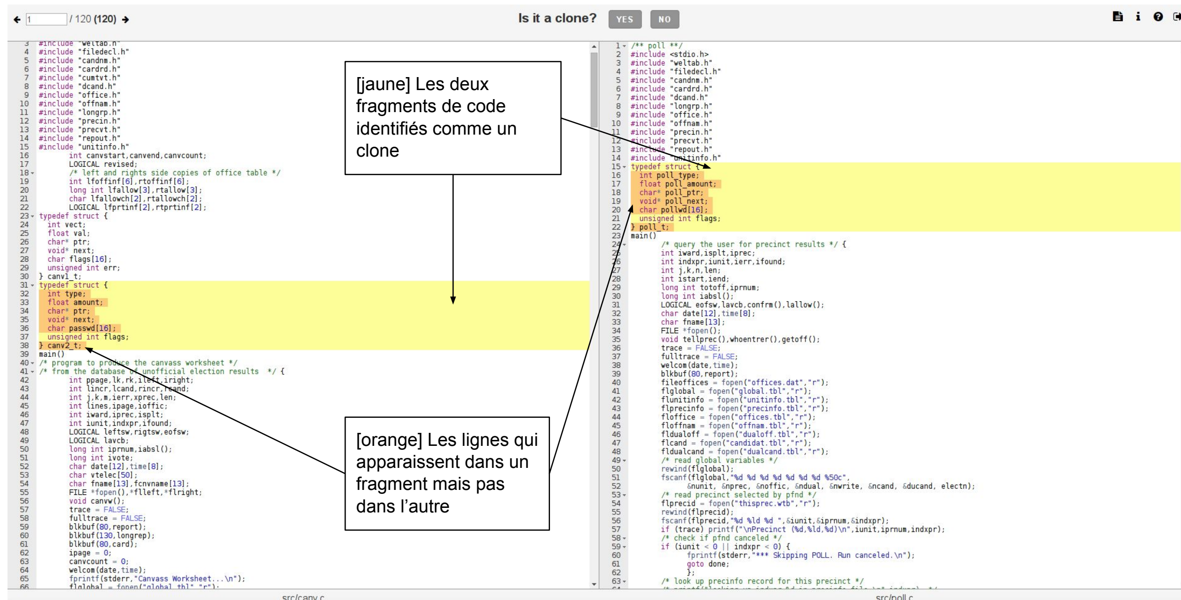


FIGURE 3.1 – Interface web pour la collecte d'opinion sur des clones.

bien les respecter. Une fois l'ensemble des clones analysé, chaque participant a répondu à un questionnaire afin d'exprimer son sentiment sur l'expérimentation à laquelle il avait participé.

3.3.4 Niveau de confiance des clones

À la suite de cette expérience, nous obtenons un ensemble de 1 080 clones avec trois opinions chacun. La première opinion est celle de Bellon et est positive car les clones proviennent de son corpus de référence. Les deux autres sont celles données par les participants et peuvent être aussi bien positives que négatives. À partir de ces trois opinions, nous affectons un niveau de confiance à chacun des 1 080 clones. Un clone avec trois opinions positives est associé à un niveau de confiance *élevé*, un clone avec deux opinions positives à un niveau de confiance *moyen* et enfin un clone avec uniquement une opinion positive (qui est nécessairement celle de Bellon) à un niveau de confiance *faible*. Ces trois niveaux définissent l'échelle ordinale suivante : $faible < moyen < élevé$.

3.4 Résultats et discussion

Dans cette section, nous répondons aux questions de recherche à partir des données collectées au cours de l'expérience. Ensuite, nous discutons les résultats des questionnaires remplis par les participants.

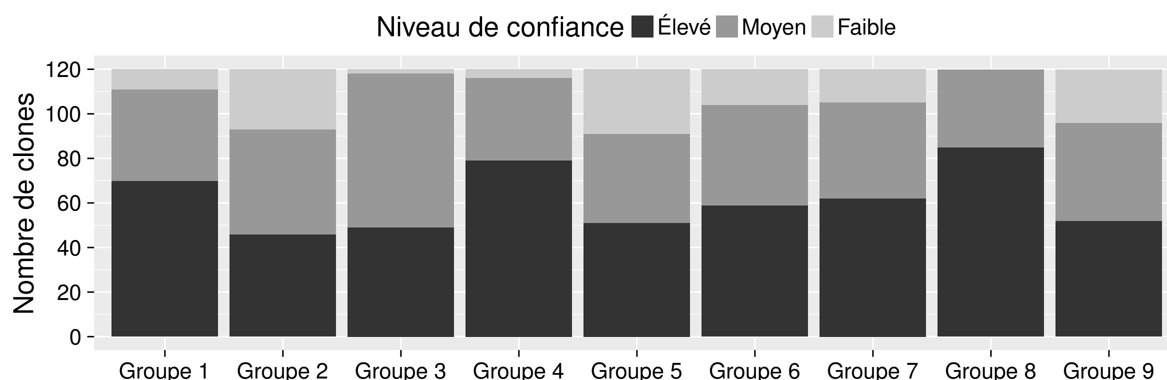


FIGURE 3.2 – Niveau de confiance des clones selon les opinions de Bellon et des deux participants de chaque groupe.

3.4.1 Niveau de confiance des clones du corpus de référence

Notre première question de recherche consiste à évaluer le niveau de confiance des clones du corpus de référence de Bellon. Les résultats bruts de notre expérimentation sont présentés dans la figure 3.2. Pour chacun des neuf groupes, le nombre de clones avec un niveau de confiance *faible*, *moyen* et *élevé* est reporté.

La première observation que nous pouvons faire est que dans la plupart des groupes plus de la moitié des clones n'ont pas un niveau de confiance *élevé*, ce qui représente une proportion significative. Concrètement, cela signifie qu'environ seulement la moitié des clones issus du corpus de référence de Bellon ont un niveau de confiance *élevé*. La seconde observation est qu'à part pour les groupes 3, 4 et 8, il y a entre 10 et 20% des clones avec uniquement un niveau de confiance *faible*, ce qui est également une proportion importante.

À titre d'exemple, le listing 3.1 reporte un clone avec un niveau de confiance *élevé*, c'est-à-dire qu'il a été jugé comme un vrai positif par les deux participants qui l'ont analysé. À l'inverse, le listing 3.2 présente un clone avec un niveau de confiance *faible*, c'est-à-dire qu'aucun des deux participants ne l'a considéré comme un vrai positif.

Maintenant que nous disposons d'informations sur le niveau de confiance d'un sous-ensemble du corpus de référence de Bellon, nous utilisons une méthode statistique pour estimer les proportions de clones avec un niveau de confiance *faible*, *moyen* et *élevé* dans la totalité du corpus de référence. Nous utilisons la méthode *bootstrap* [Efron, 1979], qui permet de calculer des intervalles de confiance, sur notre échantillon de 1 080 clones. Les résultats de cette méthode statistique sont donnés dans la table 3.4. Nous pouvons noter qu'il n'y a qu'entre 48 et 54% des clones du corpus de référence avec un niveau de confiance *élevé*, ce qui est relativement faible. De plus, il y a une proportion significative de clones qui

LISTING 3.1 – Exemple d'un clone détecté dans *eclipse-ant* avec un niveau de confiance *élevé*.

Fragment 1 : src/ant/taskdefs/compilers/Jikes.java

```
119 if (debug) {
120     cmd.createArgument().setValue("-g"); }
121 if (optimize) {
122     cmd.createArgument().setValue("-O"); }
123 if (verbose) {
124     cmd.createArgument().setValue("-verbose"); }
```

Fragment 2 : src/ant/taskdefs/compilers/Jvc.java

```
105 if (debug) {
106     cmd.createArgument().setValue("/g"); }
107 if (optimize) {
108     cmd.createArgument().setValue("/O"); }
109 if (verbose) {
110     cmd.createArgument().setValue("/verbose"); }
```

TABLEAU 3.4 – Intervalle de confiance à 95% des proportions de clones avec un niveau de confiance *faible*, *moyen* et *élevé* dans le corpus de référence de Bellon.

	Intervalle de confiance	
Niveau de confiance	**Borne inférieure**	**Borne supérieure**
faible	0, 10	0, 14
moyen	0, 34	0, 40
élevé	0, 48	0, 54

ne font pas l'unanimité : entre 34 et 40% des clones ont un niveau de confiance *moyen*, et entre 10 et 14% ont un niveau de confiance *faible*.

Synthèse

Le corpus de référence de Bellon contient un nombre significatif de clones qui sont discutables. En effet, environ la moitié de ces clones n'ont pas un niveau de confiance *élevé* et entre 10 et 20% d'entre eux ont seulement un niveau de confiance *faible*.

LISTING 3.2 – Exemple d'un clone détecté dans *postgresql* avec un niveau de confiance *faible*.

Fragment 1 : src/backend/nodes/copyfuncs.c

```
1170 static ResTarget *
1171 _copyResTarget(ResTarget *from) {
1172     ResTarget  *newnode = makeNode(ResTarget);
1173     if (from->name)
1174         newnode->name = pstrdup(from->name);
1175     Node_Copy(from, newnode, indirection);
1176     Node_Copy(from, newnode, val);
1177     return newnode; }
```

Fragment 2 : src/backend/nodes/copyfuncs.c

```
1707 static AlterGroupStmt *
1708 _copyAlterGroupStmt(AlterGroupStmt *from) {
1709     AlterGroupStmt *newnode = makeNode(AlterGroupStmt);
1710     if (from->name)
1711         newnode->name = pstrdup(from->name);
1712     newnode->action = from->action;
1713     Node_Copy(from, newnode, listUsers);
1714     return newnode; }
```

3.4.2 Niveaux de confiance et mesures d'efficacité

La deuxième question de recherche vise à déterminer si les mesures de précision et de rappel calculées à partir du *benchmark* de Bellon sont fiables. Nous cherchons à évaluer si le niveau de confiance des clones a un impact sur les valeurs de ces mesures. Ainsi, pour répondre à cette question, nous comparons les valeurs de ces deux mesures obtenues à l'aide du *benchmark* de Bellon à celles obtenues avec d'autres ensembles de référence. Comme nous disposons d'un ensemble de clones jugé par trois personnes, nous pouvons construire aisément deux nouveaux ensembles de référence. Le premier, nommé F, contient uniquement des clones qui ont un niveau de confiance *moyen* ou *élevé* et le second, nommé G, n'est composé que de clones avec un niveau de confiance *élevé*. Dans tous les cas, nous considérons que les clones avec un niveau de confiance *faible* ne peuvent faire partie d'un ensemble de référence. Finalement, d'après les résultats de notre expérimentation, F contient 954 clones et G en contient 553. De plus, B représente l'ensemble

des 1 080 clones issus du *benchmark* de Bellon qui ont été analysés lors de l'expérimentation. Par construction, nous avons $G \subset F \subset B$.

Nous utilisons une analyse pire cas afin d'évaluer si le niveau de confiance affecté aux clones du corpus de référence de Bellon peut modifier les valeurs des mesures de précision et de rappel. Le principe est le suivant : nous supposons disposer d'un détecteur de clones fictif t qui trouve exactement les clones du corpus de référence construit par Bellon. Il a donc une précision et un rappel de 1 vis à vis de cet ensemble de référence. Nous supposons ensuite que le corpus de référence devrait en fait être F ou G (car nous voulons prendre en considération le niveau de confiance), et nous calculons la diminution de la précision pour chacun de ces deux nouveaux ensembles de référence. Comme F et G sont des sous-ensembles de B, la valeur du rappel ne changera pas. Réciproquement, nous mesurons la diminution du rappel. Tout d'abord, nous supposons que t identifie exactement les clones de F ou G et qu'il a par conséquent une précision et un rappel de 1 par rapport à ceux-ci. Ensuite, nous supposons que le corpus de référence devrait être B et nous calculons alors la diminution de la valeur du rappel. De même, comme F et G sont des sous-ensembles de B, la précision restera inchangée.

Les résultats de cette analyse pire cas sont les suivants. Si un détecteur de clones identifie exactement les clones du corpus de référence de Bellon alors les valeurs des mesures de précision et de rappel ne sont plus de 1. Elles sont égales à $0,88$ dans le cas où l'ensemble de référence contient des clones avec un niveau de confiance *moyen* ou *élevé* et elles valent $0,51$ quand uniquement des clones avec un niveau de confiance *élevé* sont considérés. Ainsi, la diminution de la précision et du rappel est respectivement de $0,12$ et $0,49$.

Synthèse
Nous avons montré que les mesures de précision et de rappel peuvent être significativement modifiées par le niveau de confiance affecté aux clones d'un ensemble de référence. Concernant le corpus de référence de Bellon, nous avons observé qu'avec un niveau de confiance *élevé*, une baisse de $0,49$ des valeurs de ces deux mesures est possible, et qu'avec un niveau de confiance au moins *moyen*, cette possible baisse est de $0,12$. Par conséquent, les comparaisons de détecteurs de clones à l'aide du *benchmark* de Bellon doivent être interprétées avec précaution lorsque les différences des valeurs de précision et de rappel sont dans ces intervalles.

3.4.3 Niveau de confiance et caractéristiques des clones

Dans la troisième question de recherche, nous examinons si certaines caractéristiques des clones sont corrélées à un niveau de confiance *élevé*. Autrement dit, nous cherchons à déterminer s'il existe des caractéristiques des clones qui facilitent le consensus. Nous

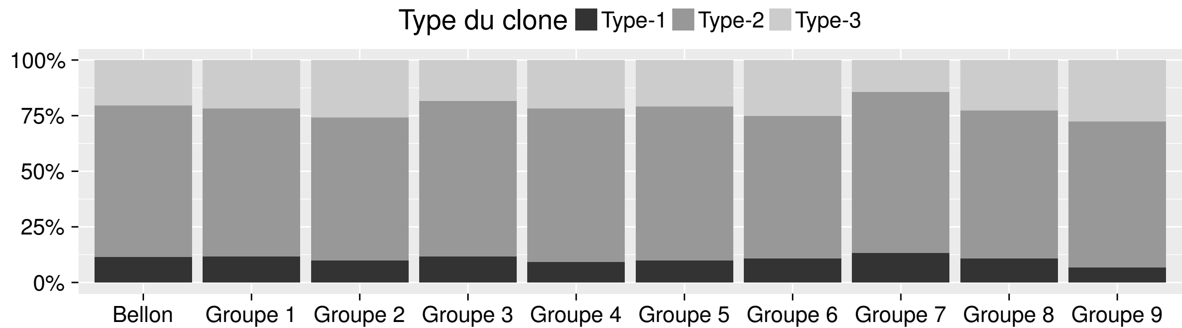


FIGURE 3.3 – Distribution des clones pour la caractéristique *type*.

étudions trois caractéristiques des clones : le *type*, la *taille* et le *langage*. Les types de clone sont définis dans la section 3.2. La taille d'un clone est la somme des nombres de lignes de ces deux fragments. Et *langage* représente le langage de programmation du logiciel dans lequel un clone a été identifié. Les logiciels utilisés dans le *benchmark* de Bellon sont écrits en C ou en Java, comme reporté dans la table 3.2.

Les figures 3.3, 3.4 [2] et 3.5 présentent les distributions des clones respectivement selon leur *type*, *taille* et *langage*. Dans ces trois figures, la première colonne représente les 4 319 clones du corpus de référence de Bellon. Les autres colonnes représentent quant à elles les neuf ensembles de 120 clones chacun évalué par un groupe de deux participants. Trois points sont à relever. Tout d'abord, la proportion des clones dans chacun des neuf groupes est plutôt similaire à celle du corpus de référence de Bellon. Cette observation est valide pour les trois caractéristiques considérées. Ensuite, le corpus de référence de Bellon contient une grande majorité de clones de type-2. Enfin, la plupart des clones ont une taille inférieure à cinquante lignes et une très faible proportion a une taille de plusieurs centaines de lignes.

Afin de répondre à cette question de recherche, nous évaluons l'impact de ces trois caractéristiques de clones sur le niveau de confiance comme défini dans la section précédente.

Caractéristique taille. Au sujet de l'association entre le niveau de confiance et la taille d'un clone, nous formulons l'hypothèse nulle H_0 suivante ainsi que son alternative H_a.

Hypothèse H_0^1
Il n'y a pas de corrélation entre la taille d'un clone et son niveau de confiance.

2. Notons que sur cette figure l'échelle de la taille des clones est logarithmique.

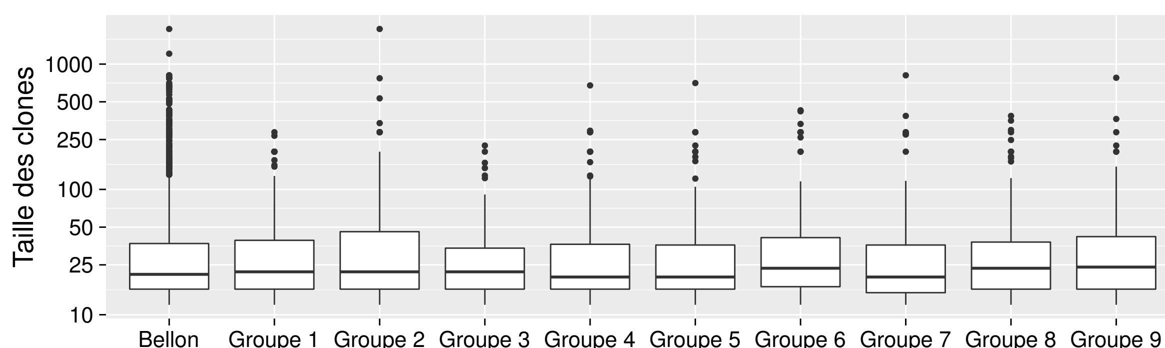


FIGURE 3.4 – Distribution des clones pour la caractéristique *taille*.

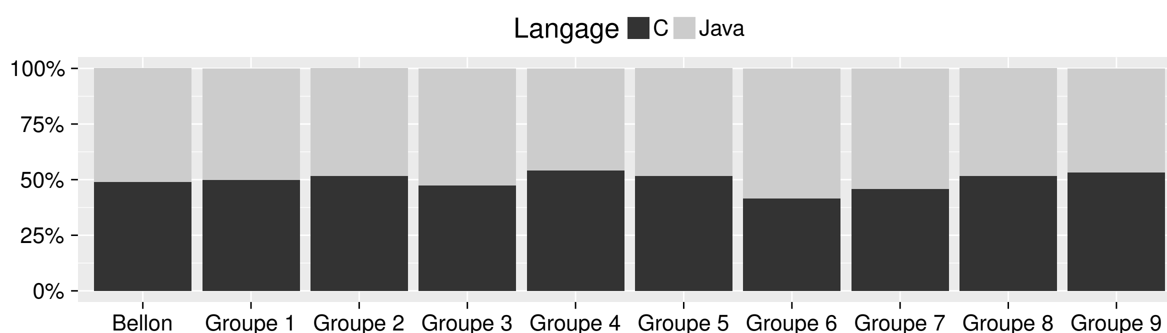


FIGURE 3.5 – Distribution des clones pour la caractéristique *langage*.

Hypothèse H_a^1
Plus la taille d'un clone est élevée plus son niveau de confiance est grand.

L'hypothèse alternative suit l'intuition que les clones très grands sont davantage susceptibles d'être de vrais clones. Le niveau de confiance par rapport à la taille des clones est tracé dans la figure 3.6. Dans cette figure, il semble que les clones avec une grande taille ont un niveau de confiance légèrement meilleur.

Puisque la taille des clones est mesurée sur une échelle d'intervalles et le niveau de confiance sur une échelle ordinale, nous utilisons un test de corrélation de Spearman, dont le résultat est significatif : $\rho = 0,06$, $p = 0,03$ (test unilatéral). Nous utilisons ensuite la méthode *bootstrap* pour calculer un intervalle de confiance de 95% pour le ρ de Spearman. Le résultat est le suivant : $0 \leq \rho \leq 0,12$. La taille de l'effet est donc très faible, ce qui signifie que la taille des clones et le niveau de confiance sont très peu associés.

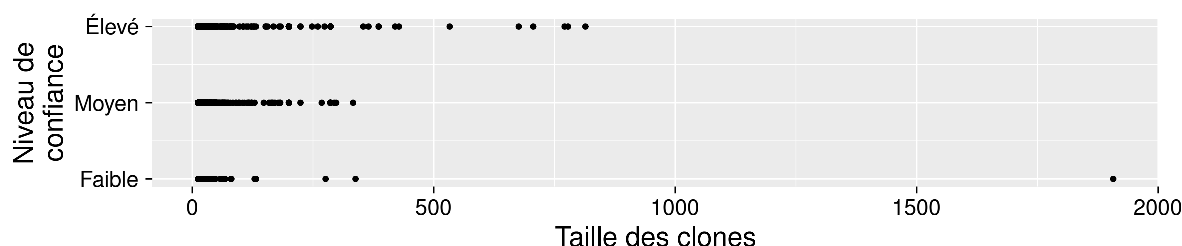


FIGURE 3.6 – Niveau de confiance en fonction de la taille des clones.

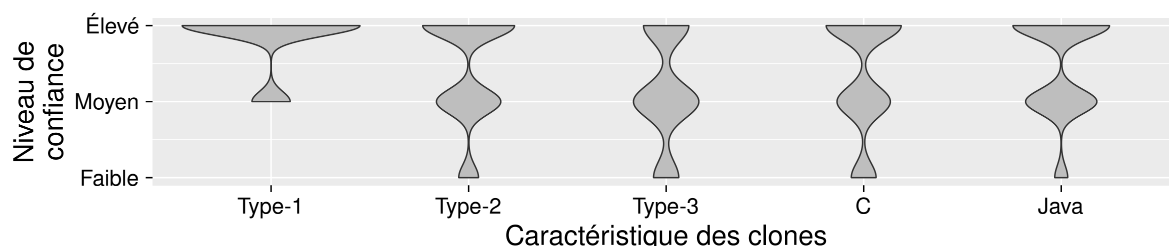


FIGURE 3.7 – Niveau de confiance en fonction du type et langage des clones.

Caractéristique type. Concernant l'association entre le niveau de confiance et le type des clones, nous formulons l'hypothèse nulle H_0 suivante ainsi que son alternative H_a.

Hypothèse H_0^2
Il n'y a pas de corrélation entre le type d'un clone et son niveau de confiance.

Hypothèse H_a^2
Plus le type d'un clone est grand (moins ses deux fragments sont similaires) moins son niveau de confiance est élevé.

Le niveau de confiance en fonction du type des clones est tracé dans la figure 3.7. Dans cette figure, il semble que le type d'un clone est associé à son niveau de confiance.

Nous considérons que le type d'un clone est mesuré sur l'échelle ordinale suivante : type 1 ≤ type 2 ≤ type 3. Cette échelle fait sens car quand le type d'un clone augmente, sa définition devient moins restrictive, comme expliqué dans la section 3.2. L'hypothèse alternative suit l'intuition que les clones avec des fragments identiques (type-1) ont un niveau de confiance plus élevé que les autres clones.

Comme à la fois le type d'un clone et le niveau de confiance sont mesurés sur une échelle ordinale, nous utilisons un test de corrélation de Spearman dont le résultat est si-

gnificatif : $\rho = -0,26$, $p = 0$ (test unilatéral). Nous utilisons ensuite la méthode *bootstrap* pour calculer un intervalle de confiance de 95% pour le ρ de Spearman. Le résultat est le suivant : $-0,31 \le \rho \le -0,21$. La taille de l'effet est donc modérée et négative, ce qui signifie que comme postulé plus les deux fragments d'un clone sont similaires plus le niveau de confiance associé est élevé. Nous pouvons noter que sur la figure 3.7 les clones de type-1 ont au pire un niveau de confiance *moyen* et ont le plus souvent un niveau de confiance *élevé.*

Caractéristique langage. Concernant l'association entre le niveau de confiance et le langage des clones, nous formulons l'hypothèse nulle H_0 suivante ainsi que son alternative H_a.

Hypothèse H_0^3
Le langage de programmation n'a pas d'impact sur le niveau de confiance.

Hypothèse H_a^3
Le langage de programmation a un impact sur le niveau de confiance.

Nous n'avons pas d'intuition particulière sur la direction de l'effet de l'association entre le langage de programmation et le niveau de confiance car les participants à notre étude sont qualifiés à la fois en C et en Java. Les deux graphiques de droite sur la figure 3.7 montrent le niveau de confiance par rapport au langage des clones. Les clones en Java semblent être associés à un niveau de confiance légèrement meilleur que ceux en C.

Puisque nous avons deux groupes (Java et C) et une variable mesurée sur une échelle ordinale, nous utilisons le test de Wilcoxon-Mann-Whitney dont le résultat produit une différence significative : $W = 133672,5$, $p = 0,008709$ (test bilatéral). Nous calculons ensuite le delta de Cliff [1996] pour évaluer la taille de l'effet. Le delta de Cliff, nommé d, mesure la quantité de différences entre deux variables et varie entre -1 et 1. La taille de l'effet est interprétée en utilisant les seuils fournis par Romano *et al.* [2006] : $d < 0,147$ l'effet est « négligeable », $d < 0,33$ « faible », $d < 0,474$ « moyen » et sinon « fort ». Finalement, nous avons $d = -0,08$ et l'intervalle de confiance est $-0,14 \le d \le -0,02$. L'effet est donc négligeable.

Synthèse
L'unique caractéristique qui a un effet significatif sur le niveau de confiance est le type des clones. De plus, seuls les clones de type-1 peuvent être considérés comme ayant un niveau de confiance *élevé* dès qu'ils ont une opinion positive d'un juge. Les autres clones requièrent davantage d'opinions.

3.5 Validité de l'étude

Nous avons identifié les obstacles suivants à la validité de notre étude.

3.5.1 Validité de construction

Le principal obstacle à la validité de construction est lié au processus de sélection des clones pour chacun des groupes de participants pour lesquels nous avons utilisé un échantillonnage aléatoire. Des résultats différents pourraient être produits en donnant d'autres clones aux participants. Cependant, comme nous l'avons noté dans la section 3.4, les caractéristiques des clones sont distribuées de façon similaire dans chacun des groupes. Par conséquent, nous considérons que cet obstacle a un impact faible.

3.5.2 Validité interne

Nous avons créé les groupes d'étudiants sans prendre en considération leur expertise sur les clones. Il est alors possible que les groupes ne soient pas équilibrés. Les niveaux de confiance calculés à partir des réponses des étudiants auraient pu être différents si nous avions créé d'autres groupes. Néanmoins, comme nous l'avons noté dans la section 3.4, le niveau de confiance des clones est distribué de la même façon dans chacun des groupes. Par conséquent, nous considérons que cet obstacle a une faible influence.

3.5.3 Validité externe

Notre étude comporte deux principaux obstacles à la validité externe. Tout d'abord, les participants tout comme Bellon étant des étudiants leur jugement pourrait ne pas correspondre à ceux de développeurs. Toutefois, ils ont tous été formés dans les langages de programmation C et Java. De plus, leur participation à cette étude était sur la base du volontariat. Deuxièmement, les participants ont analysé seulement un sous-ensemble du *benchmark* de Bellon. Les résultats sur la totalité du *benchmark* pourraient donc différer. Toutefois, une proportion acceptable de l'ensemble original de Bellon a été évaluée (25%), et les clones examinés par les participants ont été sélectionnés aléatoirement.

3.6 Conclusion

Dans cette étude, nous avons mené une évaluation empirique du *benchmark* de clones de Bellon. Nous avons sollicité l'opinion de dix-huit personnes sur un sous-ensemble du corpus de référence construit par Bellon et exploré trois questions de recherche. Premièrement, nous avons montré qu'une part significative des clones de ce corpus de référence sont contestables : une majorité des clones évalués par les participants ont un niveau de

confiance *faible* ou *moyen*. Deuxièmement, nous avons découvert que les mesures de précision et de rappel pouvaient être significativement affectées par le niveau de confiance des clones du corpus de référence. Les résultats dérivés de ce *benchmark* doivent donc être interprétés avec précaution. Enfin, nous avons montré que parmi les trois caractéristiques de clones étudiées seul le type est corrélé au niveau de confiance. En effet, seuls les clones de type-1 peuvent être considérés comme ayant un niveau de confiance *élevé* dès lors qu'ils reçoivent au moins une opinion positive. La leçon que nous pouvons tirer de cette étude est qu'un *benchmark* de clones construit par une seule personne ne suffit pas à garantir des résultats fiables.

CHAPITRE

4

Fiabilité des juges dans la construction de *benchmarks*

Nous avons montré dans le chapitre précédent que la construction et la validation de benchmarks *de clones réalisées par une seule personne ne permet pas de garantir des résultats fiables. Toutefois, nous ne disposons d'aucune information concernant les* benchmarks *de clones liés à une tâche, c'est-à-dire valides pour un objectif particulier. Nous nous demandons donc quelle mesure la méthodologie actuelle de construction de* benchmarks *peut être adaptée pour concevoir des* benchmarks *permettant une évaluation fiable des détecteurs de clones pour une tâche donnée. Notre objectif dans ce chapitre est d'examiner la fiabilité des opinions des juges à propos des clones dépendant d'une tâche. Pour cela, nous avons sélectionné aléatoirement six cents clones de deux logiciels et avons demandé à plusieurs juges, incluant des experts de ces deux logiciels, de manuellement vérifier ces clones. Nous avons observé qu'ils sont rarement d'accord les uns avec les autres ainsi qu'avec l'expert. Par conséquent, l'utilisation de juges non experts pour construire des* benchmarks *de clones pourrait ne pas être fiable.*

Sommaire

4.1 Introduction **56**
4.2 Questions de recherche **56**
4.3 Méthodologie **58**
4.4 Résultats et discussion **66**
4.5 Validité de l'étude **74**
4.6 Conclusion **76**

4.1 Introduction

Les *benchmarks* de clones existants ont été construits sans objectif particulier. Nous les appelons *benchmarks* sans contexte car il n'existe pas de relation entre leur usage et la manière dont ils ont été définis. Nous nous demandons donc dans quelle mesure la manière actuelle de construire des *benchmarks* de clones est fiable quand des clones dépendant d'une tâche donnée sont considérés (dans un tel cas, un clone est pertinent par rapport à un objectif particulier).

Dans cette étude, nous examinons la fiabilité de l'avis des juges sur des clones liés à une tâche. Notre but est de faire ressortir plusieurs recommandations pour faciliter la construction de *benchmarks* de clones liés à une tâche. Dans cette étude, nous considérons à la fois les activités de co-évolution et de réingénierie logicielle. Pour mener cette étude, nous sollicitons quatre juges pour analyser les résultats produits par un détecteur de clones sur deux logiciels. Pour chacun des deux logiciels, trois des juges n'ont pas ou peu de connaissances sur le code du logiciel et le dernier en est l'un des principaux développeurs ; il y a donc trois juges dits « externes » et un expert. L'avis de l'expert est primordial pour un *benchmark* de clones liés à une tâche car les clones identifiés par un détecteur de clones doivent être pertinents pour lui. Ainsi, les deux experts de notre étude sont les oracles décidant si un clone est un vrai ou un faux positif, selon la tâche considérée. Finalement, nous utilisons les réponses des juges pour débattre de la fiabilité de leur avis.

Nous montrons dans cette étude que l'utilisation de juges externes pour la construction de *benchmarks* de clones liés à une tâche pourrait ne pas être fiable. Nous montrons aussi que le logiciel étudié peut avoir un impact significatif sur le niveau d'accord entre juges externes et experts. De plus, nous observons que les vrais clones (selon l'avis des experts) semblent considérablement plus difficiles à analyser par les juges externes. À partir de nos résultats, nous recommandons d'utiliser plusieurs juges externes afin de construire des *benchmarks* de clones liés à une tâche et d'y impliquer des experts pour valider les vrais positifs.

La suite de ce chapitre est organisée comme suit. La section 4.2 introduit le problème de la précision de la détection. Ensuite, nous décrivons la méthodologie de notre étude empirique dans la section 4.3. La section 4.4 présente les résultats de cette étude empirique ; elle expose plusieurs questions de recherche ainsi que leurs réponses. Nous discutons les obstacles à la validité de cette étude dans la section 4.5. Nous concluons dans la section 4.6.

4.2 Questions de recherche

L'examen manuel des clones candidats dans les *benchmarks* existants est généralement assuré par des étudiants ou des chercheurs qui ne sont pas des experts des logiciels dont sont issus les clones. De plus, ces juges ne suivent pour la plupart pas d'entraînement parti-

culier avant d'analyser des clones. Cependant, comme ces *benchmarks* sont sans contexte, nous nous demandons dans quelle mesure cette manière de construire des *benchmarks* de clones est fiable quand des clones liés à une tâche sont considérés.

Dans cette étude, nous explorons la fiabilité de l'avis des juges sur des clones liés à une tâche donnée. Nous concentrons notre recherche sur deux points principaux. Premièrement, nous examinons la reproductibilité des réponses des juges. Nous nous demandons si un juge analyse toujours de façon consistante un clone qui lui est présenté plusieurs fois. Ensuite, nous étudions la fiabilité entre juges. Plus précisément, nous évaluons si plusieurs juges analysent les clones de manière identique. Nous incluons un expert des logiciels dont sont issus les clones dans l'ensemble des juges. L'avis d'un expert est important pour un *benchmark* de clones liés à une tâche car les clones identifiés doivent être pertinents pour lui. Par conséquent, nous évaluons si les juges externes peuvent analyser les clones comme les experts. Enfin, nous évaluons également s'il existe des clones pour lesquels il est plus facile de parvenir à un accord entre juges externes et experts. Pour résumer, nous examinons les questions de recherche suivantes.

Reproductibilité des réponses des juges

Question de recherche 1. *Les réponses des juges sont-elles cohérentes dans le temps?* Comme expliqué précédemment, les juges ne reçoivent généralement pas d'entraînement avant de classer des clones. Ceci pourrait être un obstacle à la reproductibilité de leurs réponses car ils peuvent être sujets à un effet d'apprentissage lors de la classification des clones. Dans cette question de recherche, nous voulons évaluer si un juge analyserait toujours de la même manière un clone. Nous souhaitons également raffiner cette question en distinguant les experts des juges externes. L'avis des juges non reproductible pourrait être le signe d'un effet d'apprentissage.

Fiabilité entre juges

Question de recherche 2. *Est-ce que les juges externes sont d'accord les uns avec les autres et également avec l'expert d'un projet?* Dans cette question de recherche, nous voulons évaluer si plusieurs juges analysent des clones de façon identique. Tout d'abord, nous examinons dans quelle mesure plusieurs juges sont d'accord les uns avec les autres. Ensuite, nous évaluons si les juges externes sont d'accord avec les experts. En répondant à cette question de recherche, nous savons s'il est fiable de construire des *benchmarks* de clones liés à une tâche en utilisant des juges externes à la place des experts.

Question de recherche 3. *Quelles sont les caractéristiques des clones qui influencent l'accord entre les juges externes et les experts?* Dans cette question de recherche, nous voulons déterminer si certaines caractéristiques des clones aident les juges externes à avoir le même avis que les experts. Nous examinons cette question car s'il s'avère qu'il existe

des caractéristiques pour lesquelles les juges externes ont le même avis que les experts, il est alors possible de construire des *benchmarks* de clones liés à une tâche en incluant des clones avec ces caractéristiques sans avoir besoin de recourir à des experts.

Nous examinons des caractéristiques relatives directement aux clones et d'autres relatives aux projets et aux experts.

Les caractéristiques relatives aux clones sont les suivantes :

— *Type :* est-il plus facile de juger des clones de type-2 ou de type-3 (comme définis par Bellon *et al.* [2007]) ? Nous ne considérons pas les clones de type-1 car ils sont moins contestables.

— *Taille :* est-il plus facile de juger des clones avec des gros fragments plutôt que ceux avec des petits ?

— *Distance :* est-il plus facile de juger des clones dans le même fichier ou dans des fichiers complètement différents ?

Les caractéristiques relatives aux projets et aux experts sont les suivantes :

— *Projet :* est-il plus facile de juger des clones d'un projet plutôt que d'un autre ?

— *Vrais/faux positifs :* est-il plus facile de juger de vrais clones que de faux clones, comme défini par l'expert ?

Dans la suite de cette étude, le terme clones fait référence à des clones liés à une tâche particulière.

4.3 Méthodologie

Dans cette section, nous décrivons l'étude empirique que nous avons menée pour répondre aux questions de recherche et pour examiner la précision des jugements des utilisateurs sur les clones.

4.3.1 Approche globale

Notre expérimentation consiste à extraire aléatoirement de deux logiciels un ensemble de clones et à présenter ceux-ci à quatre juges incluant un expert de chacun des deux logiciels. Les juges doivent alors classer les clones en vrais et faux positifs via une interface web que nous avons développée. Cette dernière affiche les deux fragments de chacun des clones à juger. Un juge dispose de trois réponses possibles pour la classification de chaque clone : *oui* dans le cas où il considère qu'il s'agit d'un vrai clone, *non* dans le cas d'un faux positif et *indéterminé* autrement ; il n'y a pas de réponse évidente. Enfin, nous utilisons une analyse statistique sur les données produites afin de répondre à nos questions de recherche. Cette analyse est discutée dans la section 4.4.

TABLEAU 4.1 – Description des logiciels.

Logiciel	Fichiers Java	Lignes de code Java	Clones
FastR	343	54 511	49 911
GumTree	77	4 750	216

4.3.2 Sujets et objets

Nous décrivons dans cette section les sujets et objets de notre expérimentation. Tout d'abord, nous présentons les logiciels desquels nous extrayons les clones. Puis, nous justifions l'outil de détection de clones utilisé ainsi que sa configuration. Enfin, nous présentons les juges qui ont participé à cette étude.

Logiciels

Notre expérimentation requiert un expert pour chaque logiciel étudié. Ainsi, nous avons sélectionné deux projets développés dans notre équipe de recherche : FastR [1] et GumTree [2]. Ces deux logiciels utilisent Java comme principal langage de programmation. FastR [Kalibera *et al.*, 2014] est une implémentation libre et efficiente du langage statistique R [Ihaka et Gentleman, 1996]. GumTree [Falleri *et al.*, 2014] est un outil libre d'analyse de différences entre arbres de syntaxe abstraite.

Un pré-traitement a été appliqué à ces deux logiciels afin d'éviter que le détecteur de clones analyse des fragments de code non pertinents. Premièrement, les fichiers automatiquement générés ou jamais modifiés manuellement sont mis à l'écart. Deuxièmement, les fichiers correspondant à des tests ou des exemples sont également mis de côté car ce type de code est généralement maintenu d'une manière différente que le code principal du projet. Ce pré-traitement a été réalisé en collaboration avec l'expert de chacun des deux logiciels. À la fin, nous supprimons les lignes vides et les commentaires de tous les fichiers restants. La table 4.1 synthétise le nombre de fichiers et de lignes de code Java que nous obtenons pour chaque projet. GumTree est un projet de petite taille alors que FastR est de taille moyenne. La petite taille de GumTree permet de vérifier manuellement tous les clones qu'un outil de détection de clones pourrait y détecter.

Détecteur de clones

De nombreux outils de détection de clones ont été introduits pour identifier des clones dans les logiciels. Tous possèdent un grand nombre de paramètres pour varier le type des clones reportés. Ainsi, les choix d'un détecteur de clones particulier et de sa configuration

1. https://github.com/allr/fastr/tree/v0.168
2. https://github.com/jrfaller/gumtree/tree/v1.0.0

peuvent avoir un fort impact sur la liste des clones identifiés. Dans cette étude, nous nous basons sur les travaux de Wang *et al.* [2013] pour minimiser l'impact de ce choix. Ils ont introduit une approche pour trouver des configurations de détecteurs de clones adaptées pour mener des études empiriques. Nous avons choisi une configuration qui maximise le rappel car nous voulions une vue d'ensemble des clones présents dans les projets sélectionnés.

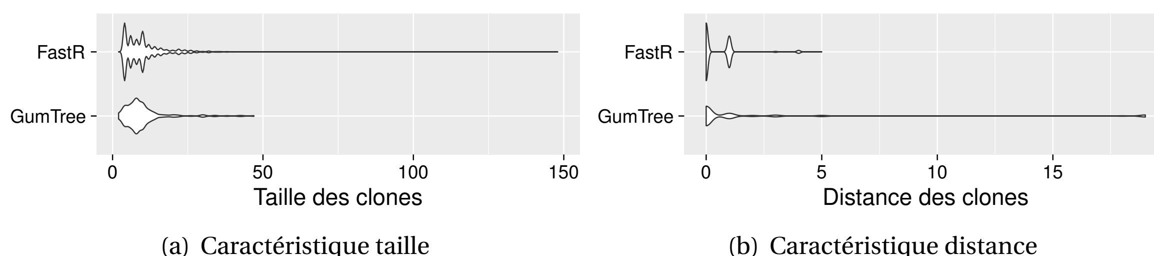
Nous avons opté pour iClones [Göde et Koschke, 2009] comme outil de détection de clones pour deux raisons. Premièrement, il est facilement disponible à des fins de réplication. Deuxièmement, il est le seul détecteur de clones réputé dans la littérature qui soit compatible avec les génériques de Java ; nos deux projets sélectionnés utilisent énormément cette fonctionnalité. iClones a deux paramètres influant sur le type des clones identifiés : *minblock*, la taille minimale des séquences de lexèmes identiques qui sont utilisés pour fusionner les clones proches et *minclone*, la taille minimale des clones exprimée en nombre de lexèmes. Nous utilisons les recommandations de Wang *et al.* pour maximiser le rappel pour des projets Java : *minblock* à 6 et *minclone* à 26.

Participants

Cinq personnes ont contribué à cette étude. La première, Alan Charpentier, était responsable du bon déroulement de l'expérimentation et n'a donc pas participé à la classification des clones. Il était le seul à connaître les questions de recherche lors de la phase de classification des clones. Les quatre autres ont été les juges qui ont classé les clones. Ils ont tous une forte expérience en programmation Java et sont tous plus que familiers avec la notion de clone. Pour chaque projet, nous avons un expert et trois juges externes. Pour FastR, l'expert est Floréal Morandat qui est l'un des principaux développeurs de ce projet. Pour GumTree, l'expert est Jean-Rémy Falleri, le mainteneur officiel. Les juges externes sont désignés par *juge1*, *juge2* et *juge3* dans la suite de cette étude.

4.3.3 Sélection des clones

Pour chaque projet, nous utilisons iClones pour calculer la liste des clones. La table 4.1 rapporte le nombre de clones (uniquement de type-2 et de type-3) détectés par iClones dans les deux projets considérés avec la configuration décrite ci-dessus. Nous rappelons que notre troisième question de recherche vise à explorer s'il existe des facteurs qui ont un effet sur l'opinion des juges. Certains de ces facteurs sont des informations calculées directement à partir des clones : le type, la taille et la distance. Nous utilisons la même définition de type que Bellon *et al.* [2007]. La taille d'un clone est définie par le nombre de lignes de ses deux fragments de code mis bout à bout. La distance d'un clone détermine à quel point les deux fragments d'un clone sont proches dans le système de fichiers. Si les deux fragments sont dans le même fichier, la distance vaut 0. S'ils appartiennent à des



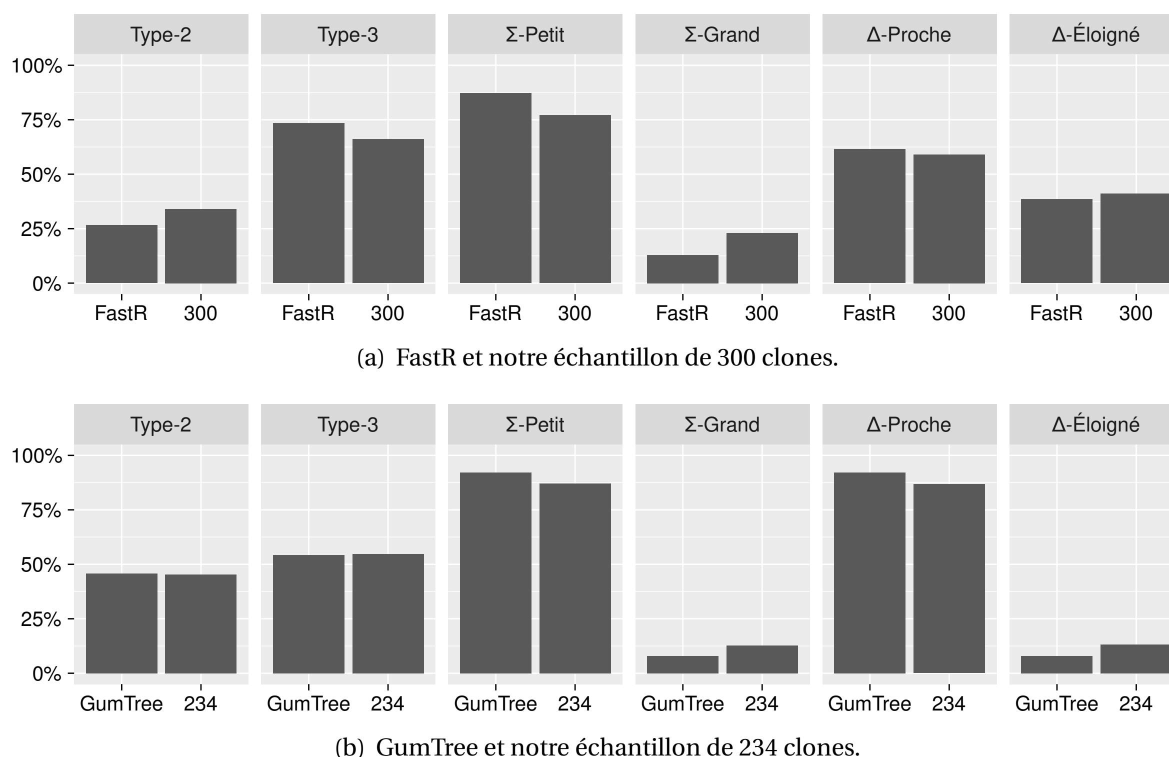
(a) Caractéristique taille (b) Caractéristique distance

FIGURE 4.1 – Distributions de la taille et de la distance des clones dans FastR et GumTree.

fichiers différents dans le même répertoire, la distance est 1. Dans les autres cas, la distance est égale à 1 plus le nombre minimum de sauts pour atteindre un fichier à partir de l'autre.

Afin de s'assurer de disposer de suffisamment de représentants pour chacun de ces facteurs, les clones sont tirés aléatoirement selon une caractéristique donnée. Pour la caractéristique du type, une fois les clones de type-1 mis à l'écart (voir la troisième question de recherche), deux groupes naturels s'imposent : les clones de type-2 et ceux de type-3. Cependant, il n'y a pas de définition naturelle de groupes pour les deux autres caractéristiques : la taille et la distance. En menant une analyse exploratoire des distributions de la taille et de la distance des clones des deux projets sélectionnés (figure 4.1), nous avons observé que ces distributions sont décalées à droite de la médiane avec une très longue queue. Par conséquent, nous choisissons de séparer les clones en deux groupes : ceux de la tête de la distribution et ceux de la queue. Pour éviter le biais de sélectionner un seuil, nous utilisons une technique automatique pour créer des groupes pour les deux caractéristiques taille et distance. Nous souhaitons cette technique déterministe afin que d'autres chercheurs puissent reproduire nos résultats. Pour répondre à ces exigences, nous utilisons l'algorithme d'extraction de données *neural-gas* [T. Martinetz, 1991] qui est une généralisation du partitionnement en k-moyennes qui produit des groupes stables. Sur notre corpus, des dizaines d'exécutions de cet algorithme produisent des groupes sans différence significative. Ainsi, nous obtenons quatre groupes supplémentaires; deux de la caractéristique taille : Σ-*Grand* et Σ-*Petit* et deux de la caractéristique distance : Δ-*Proche* et Δ-*Éloigné*. Les seuils obtenus pour séparer les groupes de la caractéristique taille sont 17 pour FastR et 18 pour GumTree. Autrement dit, les clones de FastR ayant une taille inférieure ou égale à dix sept sont dans le groupe Σ-*Petit* et les autres dans Σ-*Grand*. Les seuils pour la caractéristique distance sont 0 pour FastR et 5 pour GumTree.

Sur la base de notre expérience antérieure de classification de clones, le nombre de clones à analyser pour chaque participant a été limité à six cents. Puisque nous avons six groupes, nous sélectionnons de manière aléatoire cinquante clones dans chacun des groupes pour les deux projets. Dans le cas où un groupe a moins de cinquante éléments, nous sélectionnons tous les clones du groupe. Ce cas se produit pour deux groupes du projet GumTree : Σ-*Grand* et Δ-*Éloigné* n'ont tous les deux que dix-sept clones. Par consé-



(a) FastR et notre échantillon de 300 clones.

(b) GumTree et notre échantillon de 234 clones.

FIGURE 4.2 – Distributions des clones dans les six groupes pour chaque projet et son échantillon associé.

quent, nous obtenons un ensemble de 300 clones pour FastR et un autre de 234 pour Gumtree pour un total de 534 clones à analyser.

En plus des facteurs type, taille et distance, nous évaluons l'influence de l'avis de l'expert sur les clones. Ainsi, nous calculons également l'union des six groupes pour chaque projet. Nous nommons cette union *générale* et l'utilisons pour évaluer l'effet des deux facteurs mentionnés ci-dessus. Comme l'échantillonnage est susceptible de produire des échantillons biaisés lorsque les groupes sont liés, nous traçons la distribution des clones de nos échantillons par rapport aux clones de l'ensemble du projet dans la figure 4.2. FastR contient 49 911 clones et GumTree 216 comme reporté dans la table 4.1 Pour GumTree, nous pouvons noter que les groupes Σ-*Grand* et Δ-*Éloigné* ont très peu d'éléments (moins de 17%, correspondant à 50/300). De plus, nous pouvons relever que chaque projet et son échantillon associé ont une distribution similaire des clones pour les six groupes.

Comme les clones de chaque groupe sont sélectionnés aléatoirement à partir de l'ensemble total des clones identifiés par iClones, des doublons peuvent apparaître. En particulier, puisque le nombre de clones tirés dans GumTree est supérieur au nombre total de clones, les doublons sont obligatoires. Cela a été fait dans le but d'évaluer la répétabi-

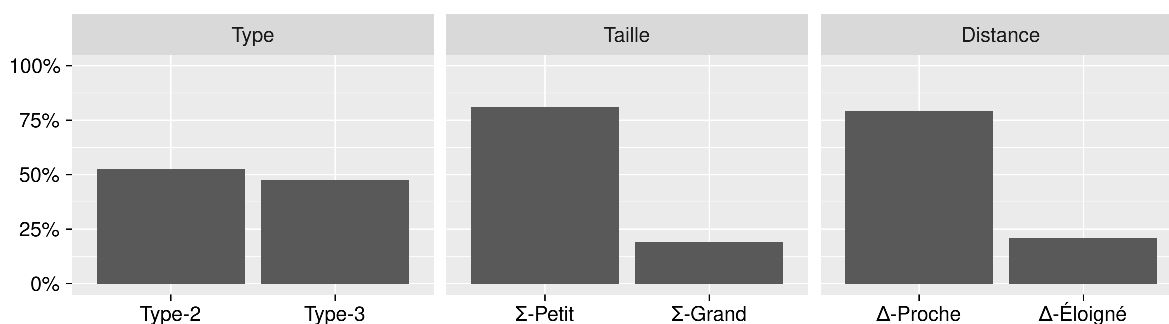


FIGURE 4.3 – Distributions dans chaque groupe des 65 doublons dans l'échantillon de GumTree.

lité de l'opinion des juges comme indiqué dans la deuxième question de recherche. Alan Charpentier a recherché la présence de doublons dans l'ensemble des 534 clones. Finalement, il y a 65 doublons dans les 234 clones de GumTree et un seul dans les 300 de FastR. Autrement dit, il n'y a en fait que 163 clones uniques pour GumTree et 299 pour FastR. Comme expliqué précédemment, Alan Charpentier n'a pas averti les juges de la présence de doublons dans l'ensemble des clones à analyser ; il était le seul à savoir qu'il y avait des doublons. La figure 4.3 présente les distributions des doublons de GumTree dans nos six groupes. Nous pouvons noter que ces distributions sont similaires à celles de l'ensemble des clones de GumTree reportées à la figure 4.2.

4.3.4 Collecte des données

Dans cette section, nous décrivons d'abord la procédure utilisée pour collecter l'opinion des juges puis nous présentons les résultats bruts que nous avons obtenus.

Procédure

Les réponses des juges sont collectées via une interface web que nous avons développée. Celle-ci affiche les deux fichiers impliqués dans le clone, souligne en jaune les fragments appartenant au clone et fait apparaître en orange la différence textuelle entre les deux fragments. Pour éviter la fatigue des juges, l'interface web est capable de sauvegarder et de restaurer la session de travail et de revenir à des clones déjà traités afin de modifier la réponse. Les juges peuvent donc classer les clones en prenant autant de temps que nécessaire. Les juges ont pu remplir le questionnaire en plusieurs jours fractionnant leur travail comme ils le souhaitaient. L'ordre des clones a été randomisé et ils ont été présentés à chaque juge dans le même ordre. Pour chaque clone la question posée aux juges est la suivante : « *Ce clone est-il utile pour une activité de co-évolution ou de réingénierie logicielle ?* ». Afin de ne pas forcer une réponse plutôt qu'une autre, les juges externes ont été autorisés

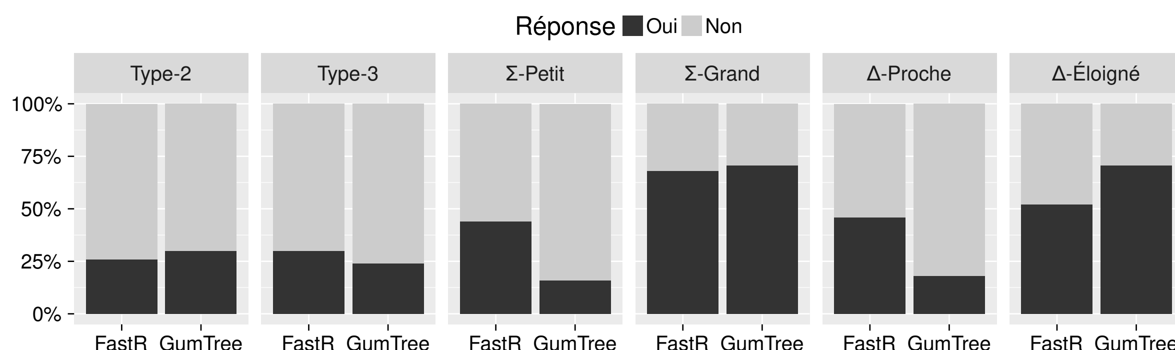


FIGURE 4.4 – Proportions des réponses *oui* et *non* données par l'expert sur les clones de chaque projet.

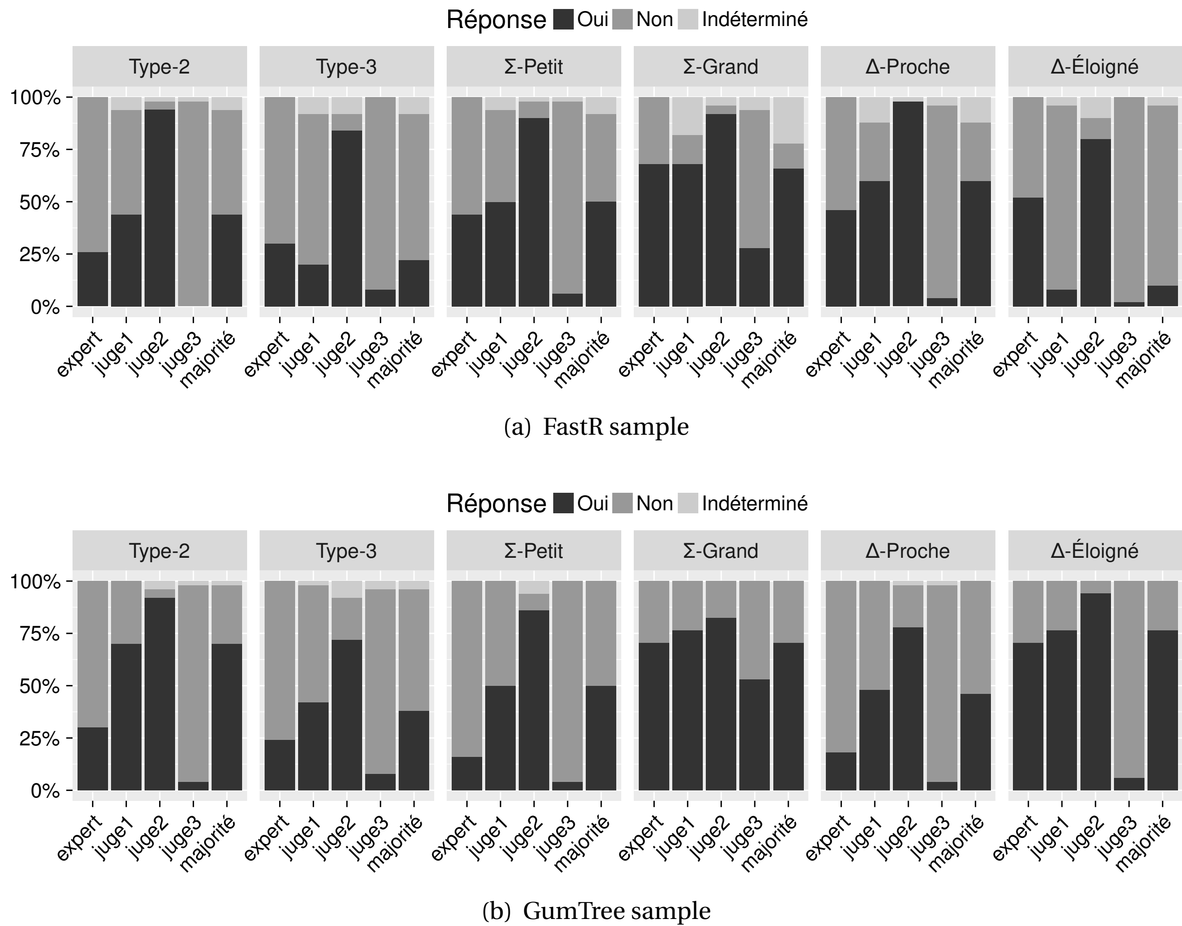
à répondre *indéterminé* à la question lorsqu'ils n'avaient pas d'avis sur un clone. Il a été formellement interdit aux participants d'échanger sur l'expérimentation tant que celle-ci n'était pas complétée par tout le monde. Au cours de la phase de classification des clones, seul Alan Charpentier avait connaissance des questions de recherche et de la manière dont les clones avaient été sélectionnés.

Résultats

Chaque participant est arrivé au bout des 534 clones. Pour chaque projet, en plus des réponses de l'expert et des juges externes, nous calculons aussi le vote de la majorité qui regroupe les votes des évaluateurs externes. Pour calculer ce vote de la majorité nous utilisons les règles suivantes. Si deux juges externes ont répondu *oui* (respectivement *non*), le vote de la majorité est *oui* (respectivement *non*). Dans les autres cas, le vote de la majorité est *indéterminé*.

Les résultats de cette expérimentation pour les experts de chaque projet sont reportés dans la figure 4.4. Nous pouvons observer les proportions de vrais et faux positifs identifiés par l'expert de chacun des projets pour les six groupes que nous avons définis. Globalement, l'expert de FastR trouve un nombre de vrais positifs légèrement plus élevé que l'expert de GumTree mais le rapport moyen des vrais clones se trouve dans la même plage pour les deux projets.

Les deux experts semblent convenir que le groupe Σ-*Grand* contient principalement de véritables clones positifs. De plus, pour les deux experts le groupe Δ-*Éloigné* semble contenir un taux plus élevé de véritables clones que les autres groupes. Tous les autres groupes contiennent plus de faux positifs que de vrais positifs. Dans cette étude, nous considérons les réponses des experts comme des références ; c'est-à-dire quand un expert répond *oui* (respectivement *non*) nous avons un vrai positif (respectivement un faux négatif).



(a) FastR sample

(b) GumTree sample

FIGURE 4.5 – Proportions des réponses *oui*, *non* et *indéterminé* données par les experts, les juges externes et la majorité pour les clones de chaque projet.

Les juges externes ont des comportements différents. Leurs réponses sont données dans la figure 4.5. Pour le projet FastR, *juge1* et *juge3* sont pessimistes à propos des clones alors que *juge2* est optimiste. Les groupes Σ-*Grand* et Δ-*Proche* sont les seuls dans lesquels la majorité des juges externes trouve plus de vrais positifs que de faux positifs. Pour le projet GumTree, *juge1* et *juge2* sont bien plus optimistes que *juge3*. Le groupe Σ-*Grand* est le seul dans lequel tous les juges trouvent une majorité de vrais clones. De plus, le vote de la majorité a un rapport significatif de vrais positifs dans les groupes *Type-2* et Δ-*Éloigné*.

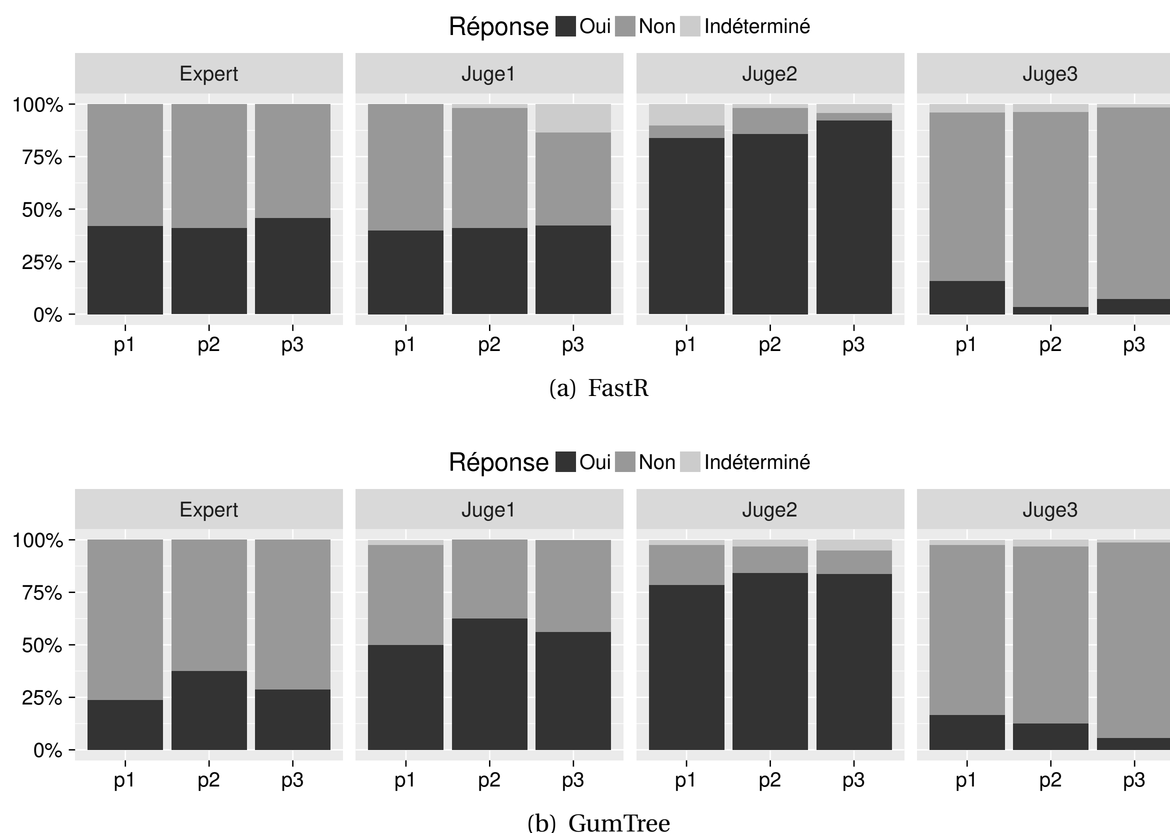


FIGURE 4.6 – Réponses des juges selon trois périodes de temps (*p1* : premier tiers des clones, *p2* : deuxième tiers et *p3* : dernier tiers).

4.4 Résultats et discussion

Dans cette section, nous présentons d'abord les résultats de notre étude en fonction de chaque question de recherche et donnons des lignes directrices pour rendre la construction de *benchmarks* de clones liés à une tâche plus fiable. Nous retranscrivons et discutons ensuite les remarques des juges recueillies lors d'une séance de discussion organisée après l'expérience.

4.4.1 Reproductibilité des réponses des juges

La première question de recherche est liée à la régularité des réponses des juges. Par régularité nous entendons que le même clone présenté plusieurs fois au même juge est toujours classé de la même manière. Afin d'évaluer ce phénomène, nous examinons s'il y a un changement dans le comportement des juges au fil du temps. Nous postulons que,

TABLEAU 4.2 – Réponses inconsistantes pour les 65 doublons dans GumTree.

Juge	Nombre	Pourcentage
expert	1	1,7
juge1	11	18,6
juge2	3	5,1
juge3	7	11,9

comme l'ordre dans lequel les clones sont présentés aux juges est aléatoire, la proportion de réponses *oui*, *non* et *indéterminé* devrait rester stable pendant toute l'expérience. Dans le cas contraire, cela signifie qu'un effet d'apprentissage ou de fatigue biaise les réponses des juges. Pour quantifier ce phénomène, nous divisons les clones de chaque projet en trois ensembles : *p1* est le premier tiers des clones analysés par les juges, *p2* est le second tiers et *p3* le dernier. Comme illustré dans la figure 4.6, il n'y a pas de variation significative entre les périodes, ce qui signifie que le comportement des juges reste stable dans le temps. Toutefois, cela ne signifie pas pour autant qu'un juge évaluera toujours le même clone de manière consistante.

Pour évaluer la régularité des réponses des juges, nous utilisons les 65 doublons contenus dans les clones du projet GumTree. Tout d'abord, nous examinons si pour un même clone, un juge peut changer sa réponse (entre *oui*, *non* et *indéterminé*). Pour chaque juge, nous calculons donc le nombre de réponses inconsistantes qu'il a donné pour ces 65 clones. Les résultats, donnés dans la table 4.2, montrent que le comportement de l'expert à ce propos diffère de celui des juges externes. Les réponses de l'expert de GumTree sont en effet très consistantes : seulement un des soixante-cinq doublons a été évalué différemment, ce qui est négligeable. Alors que pour les juges externes, le nombre de clones avec des réponses inconsistantes varie d'environ cinq à vingt pour cent, ce qui est significatif. Nous avons cherché des tendances dans la façon dont les réponses inconsistantes évoluent dans le temps mais il n'y a pas assez de doublons inconsistants pour tirer des conclusions. Les réponses inconsistantes peuvent donc être attribuées à une sorte d'effet d'apprentissage. Pour conclure, une session préalable d'entraînement serait bénéfique aux juges externes.

4.4.2 Fiabilité inter-juges

Dans cette section, nous étudions la fiabilité inter-juges des jugements sur les clones. Tout d'abord, nous évaluons si plusieurs juges externes donnent ou non les mêmes avis sur les mêmes clones. Ensuite, nous examinons si les juges externes ont les mêmes avis que les experts des projets. Enfin, nous étudions s'il existe des facteurs qui permettent aux juges externes de plus facilement approcher l'avis des experts.

TABLEAU 4.3 – Accord entre les juges calculé avec les tests du Kappa de Fleiss et de Cohen.

	FastR		GumTree	
Juges	κ	**Accord**	κ	**Accord**
juges externes	$-0,12$	aucun	$-0,05$	aucun
expert et juge1	$0,24$	faible	$0,42$	modéré
expert et juge2	$0,09$	très faible	$0,13$	très faible
expert et juge3	$0,16$	très faible	$0,28$	faible
expert et la majorité	$\mathbf{0,25}$	faible	$\mathbf{0,44}$	modéré

Accord entre juges

Nous commençons par évaluer l'accord entre les juges externes en utilisant le Kappa de Fleiss. Le résultat de cette mesure statistique de l'accord dans un groupe de personnes est donné dans la première ligne de la table 4.3. Cette statistique s'interprète avec les seuils définis par Landis et Koch [1977] reportés dans la table 4.4. Nous pouvons observer que dans les deux projets il n'y a pas d'accord entre les juges externes, ce qui signifie que leurs jugements divergent fortement sur les mêmes clones. Il semble donc que juger des clones est une tâche très subjective. Deuxièmement, nous évaluons l'accord entre chaque juge externe et l'expert des deux projets avec le test du Kappa de Cohen. Les résultats sont donnés dans les lignes 2 à 4 de la table 4.3. Enfin, nous examinons l'accord entre le vote de la majorité des juges externes (comme défini dans la section 4.3.4) et l'expert des deux projets. La dernière ligne de la table 4.3 donne la valeur de cet accord. Nous observons un comportement différent dans les deux projets. Il semble en effet plus facile pour les juges externes d'être d'accord avec l'expert de GumTree qu'avec celui du projet FastR. Nous observons également que le choix d'un juge externe particulier a un fort impact sur l'accord avec l'expert. Par exemple, dans FastR l'accord est au pire très faible et au mieux faible alors que pour GumTree il est au pire faible et au mieux modéré. Par conséquent, utiliser un seul juge externe pour évaluer des clones n'est pas fiable. Enfin, nous pouvons noter dans les deux projets que l'utilisation du vote de la majorité des juges externes est toujours une meilleure stratégie que le vote d'un juge particulier pour approcher l'avis de l'expert. En effet, la valeur maximale de la statistique Kappa (en gras dans la table 4.3) est atteinte pour les deux projets avec le vote de la majorité.

Facteurs impactant l'accord entre experts et juges externes

Dans la troisième question de recherche, nous étudions s'il existe des facteurs qui facilitent l'accord entre les juges externes et les experts. Comme le vote de la majorité des juges externes s'est avéré être toujours la meilleure stratégie dans la section précédente, nous considérons uniquement dans cette question de recherche les avis correspondant

TABLEAU 4.4 – Interprétation du test du Kappa.

κ	**Interprétation**
≤ 0	Aucun accord
$]0, 0.2]$	Accord très faible
$]0.2, 0.4]$	Accord faible
$]0.4, 0.6]$	Accord modéré
$]0.6, 0.8]$	Accord fort
$]0.8, 1]$	Accord presque parfait

au vote de la majorité. La figure 4.7 rapporte l'accord brut entre l'expert et la majorité pour chacune des caractéristiques étudiées. Les premières observations tendent à indiquer que certaines caractéristiques ont un impact sur l'accord entre la majorité et l'expert (par exemple la *taille* ou le *projet*). Dans la suite de cette section, nous évaluons statistiquement l'influence des caractéristiques sur l'accord entre l'expert et la majorité. Nous présentons d'abord l'effet des caractéristiques de clones puis l'effet des facteurs liés à l'avis de l'expert sur les clones.

Caractéristiques des clones. Pour évaluer l'effet des caractéristiques des clones, nous utilisons les deux groupes définis pour chaque caractéristique (comme expliqué dans la section 4.3.3). Pour chaque groupe, nous nous intéressons à deux informations : le nombre de réponses identiques et différentes avec l'expert. Ainsi, pour chaque caractéristique, nous construisons une table de contingence de 2×2 comme illustré dans la table 4.5. Notre hypothèse nulle est que le nombre de réponses identiques et différentes avec l'expert est indépendant de chaque caractéristique. L'hypothèse alternative est que ce nombre n'est pas indépendant. Nous utilisons le test statistique du χ^2 pour évaluer notre hypothèse. Nous appliquons la correction de Yates au calcul du χ^2 quand la valeur d'une cellule est trop faible, c'est-à-dire inférieure à 5. De plus, nous calculons la taille de l'effet de chaque caractéristique en utilisant la mesure du V de Cramer interprétée avec les seuils reportés dans la table 4.6.

Pour la caractéristique type, la valeur-p est $p = 0,07$ ($V = 0,19$) pour FastR et $p = 0,01$ ($V = 0,25$) pour GumTree. La valeur-p pour FastR n'est pas significative avec un seuil de $0,05$. Cependant, celle pour GumTree est significative et la taille de l'effet est modérée. Ceci indique que cette caractéristique pourrait avoir une influence dans le projet GumTree. Pour la caractéristique taille, la valeur-p est $p = 0,04$ ($V = 0,20$) pour FastR et $p = 7,1e^{-3}$ ($V = 0,37$) pour GumTree. La valeur-p est significative pour FastR et la taille de l'effet est modérée. La valeur-p est également significative pour GumTree et la taille de l'effet est forte. Ceci indique que cette caractéristique semble avoir une influence sur les deux projets. Notons quand même que la correction de Yates a été utilisée à cause du faible nombre de clones dans le groupe Σ-*Grand* pour GumTree. Pour la caractéristique distance,

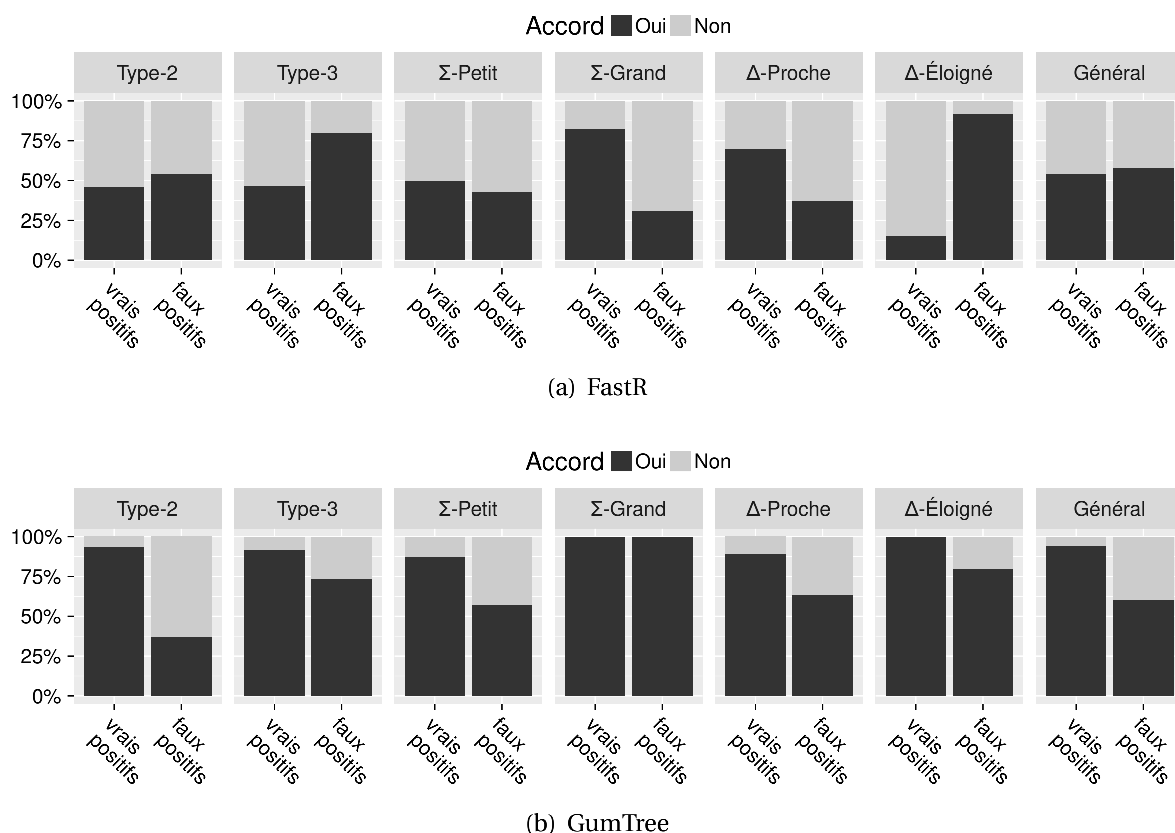


FIGURE 4.7 – Accord entre l'expert et la majorité pour les six caractéristiques étudiées.

TABLEAU 4.5 – Nombre de réponses identiques et différentes entre l'expert et la majorité pour l'ensemble des clones de chaque projet.

	FastR		**GumTree**	
Caractéristique	**Identiques**	**Différentes**	**Identiques**	**Différentes**
Type-2	26	24	27	23
Type-3	35	15	39	11
Σ-*Petit*	23	27	31	19
Σ-*Grand*	33	17	17	0
Δ-*Proche*	26	24	34	16
Δ-*Éloigné*	26	24	16	1
Général	169	131	164	70

TABLEAU 4.6 – Interprétation du V de Cramer.

V de Cramer	**Taille de l'effet**
≤ 0.10	Très faible
$]0.10, 0.19]$	faible
$]0.20, 0.29]$	Modéré
≥ 0.30	Fort

TABLEAU 4.7 – Nombre de réponses identiques et différentes entre l'expert et la majorité.

Projet	**Identiques**	**Différentes**
FastR	199	101
GumTree	191	43

la valeur-p est $p = 1$ ($V = 0$) pour FastR et $p = 0,07$ ($V = 0,26$) pour GumTree. Les deux valeur-p ne sont pas significatives. En conclusion, la caractéristique taille est la seule à avoir une influence sur les deux projets. Il est beaucoup plus difficile pour les juges externes d'être d'accord avec l'expert sur les petits clones. Pour ces clones, utiliser des juges externes pourrait être peu fiable ; l'utilisation d'experts des projets considérés semble donc obligatoire.

Projet et avis de l'expert. Pour évaluer l'influence du projet, nous construisons une table de contingence de 2×2 (reportée table 4.7) avec en lignes les projets, et en colonnes le nombre de réponses identiques et différentes dans le projet. Nous utilisons un test du χ^2 qui donne $p = 1,1e^{-3}$ et $V = 0,14$. Cette valeur-p est significative et la taille de l'effet est faible. Ceci signifie qu'il semble plus difficile pour les juges d'approcher l'avis de l'expert pour les clones de FastR plutôt que pour ceux de GumTree. Par conséquent, les clones de certains projets ne peuvent pas être jugés par des juges externes.

Pour évaluer si le jugement de l'expert sur les clones a un effet sur le nombre de réponses identiques avec la majorité des juges, nous utilisons l'union des clones de tous les groupes désignée par *général* comme expliqué dans la section 4.3.3. Nous commençons par examiner s'il est plus facile de se mettre d'accord sur des vrais ou des faux positifs comme jugés par l'expert. Pour cela, nous divisons les clones en deux ensembles : ceux jugés positivement par l'expert (l'ensemble des oui) et ceux jugés négativement par l'expert (l'ensemble des non). Pour chaque ensemble, nous comptons le nombre de réponses identiques et différentes entre le vote de la majorité et l'expert de chaque projet. Nous utilisons ces informations pour construire deux tables de contingence de 2×2 reportées dans la table 4.8. De façon similaire à la section précédente, nous utilisons un test du χ^2 qui donne $p = 0,49$ avec $V = 0,04$) pour FastR et $p = 2,8e^{-7}$ avec $V = 0,34$ pour GumTree.

TABLEAU 4.8 – Nombre de réponses identiques et différentes entre l'expert et la majorité pour les vrais et faux positifs tels que jugés par l'expert.

	FastR		**GumTree**	
Avis de l'expert	**Identiques**	**Différentes**	**Identiques**	**Différentes**
Oui	72	61	64	4
Non	97	70	100	66

La valeur-p est significative pour GumTree. Cela signifie qu'il est beaucoup plus difficile pour les juges externes d'évaluer les vrais positifs que les faux positifs dans GumTree. En effet pour juger de vrais positifs, les juges externes ont davantage de réponses différentes que de réponses identiques avec l'expert. À l'inverse pour juger les faux positifs, les juges externes donnent plus de réponses identiques que différentes. Ceci indique que l'utilisation de juges externes pour évaluer des vrais positifs pourrait ne pas être fiable. Pour ces clones, utiliser un expert du projet semble obligatoire. Toutefois, ces observations doivent être considérées prudemment car nous n'avons pas relevé les mêmes effets sur le projet FastR. Nous reconnaissons le fait que davantage de projets doivent être examinés pour tirer des conclusions plus générales.

4.4.3 Remarques des participants

Après avoir mené l'expérience, tous les participants ont effectué une réunion où les questions importantes concernant la difficulté de la classification des clones ont été discutées. Ces questions sont classées et discutées dans le reste de cette section.

Classification des clones

Chaque participant a constaté qu'il était très difficile d'évaluer les clones d'une manière homogène, ce qui corrobore les observations faites sur l'inconsistance de leurs réponses (table 4.2)

Chaque participant a estimé qu'il aurait été plus facile d'évaluer les clones s'ils avaient été ordonnés par classes. Au moins, cela aurait amélioré l'homogénéité de leurs réponses.

Deux participants ont constaté qu'il aurait été plus facile d'évaluer les clones s'ils avaient été ordonnés par leur contexte (clones au sein de la même fonction ou classe Java). Ils ont expliqué que cela prend parfois du temps pour comprendre le code entourant un clone et qu'il serait donc préférable de ne faire cet effort qu'une seule fois.

Chaque participant a remarqué qu'il était plus facile de décider si un clone est pertinent pour une tâche de réingénierie logicielle plutôt que pour une activité de co-évolution. Ils ont justifié leur remarque en expliquant que la co-évolution exige une connaissance beaucoup plus approfondie du code. Cependant, ils n'étaient pas toujours d'accord sur les pos-

sibilités de réingénierie logicielle. Par exemple, certains d'entre eux appliquent des opérations de réingénierie logicielle pour une seule ligne de code tandis que pour d'autres une quantité importante de code est nécessaire pour effectuer une opération de réingénierie logicielle.

Outil de détection de clones

Alignement. Chaque participant a eu des problèmes avec certains des clones détectés par iClones qui n'étaient pas alignés sur la structure du code (par exemple un clone qui commence dans une fonction et se termine dans une autre). Cela est dû au fait que iClones utilise une détection de clones lexicale. Tous les participants ont mis de côté plusieurs clones qui auraient pu être utiles s'ils avaient été alignés sur la structure du code.

Extension. Chaque participant a également remarqué que iClones reporte certains clones qui pourraient être facilement étendus. Les experts et non-experts ont eu du mal à évaluer ces clones. Chaque participant a jugé ces clones de manière consistance en utilisant sa propre stratégie. Certains ont décidé de les garder tandis que plusieurs autres ont décidé de les considérer comme des faux positifs.

Les participants ont identifié tous ensemble 43 clones qui auraient pu être étendus : 32 pour FastR et 11 pour GumTree pour un total de 8% des analysés. Les résultats sur la fiabilité inter-juges (section 4.4.2) ont été recalculés à partir de l'ensemble des 491 clones (534 − 43) correspondant à ceux qui ne nécessitent pas d'être étendus. Aucun changement significatif n'a été observé. Ainsi, nos résultats ne sont pas affectés par ces clones. En conséquence, lors de la construction d'un *benchmark* de clones, les juges doivent être en mesure de modifier les limites d'un clone candidat avant de décider s'il s'agit d'un vrai ou faux positif. Il est à noter que Bellon a utilisé cette bonne pratique lors de la construction de son *benchmark* [Bellon *et al.*, 2007].

Pour conclure, il semble qu'un détecteur de clones doit à la fois augmenter la taille d'un clone au maximum par rapport à la structure du code et ne reporter que des clones syntaxiquement complets. Toutefois, cela nécessite soit un post-traitement soit un outil de détection de clones disposant d'un analyseur syntaxique pour tous les langages de programmation qu'il supporte.

Langage de programmation

Deux participants ont noté que les caractéristiques du langage de programmation ont parfois un impact important sur la pertinence d'un clone. Par exemple, le typage statique du langage Java rend certains clones sans importance alors qu'ils auraient été d'une grande aide dans un langage typé dynamiquement. Cette observation a été particulièrement vérifiée dans FastR qui requiert de l'*inlining* de type à cause de l'utilisation de l'*autoboxing* en Java.

Deux participants ont constaté que le patron de conception visiteur induit la présence d'un grand nombre de clones qui sont difficiles à évaluer.

4.5 Validité de l'étude

Nous avons identifié les obstacles suivants à la validité de notre étude.

4.5.1 Validité de construction

Le principal obstacle à la validité de construction est lié au processus de construction des groupes de clones pour lesquels nous avons fait de l'échantillonnage. Il pourrait y avoir une certaine influence entre les groupes : un grand clone pourrait avoir plus de chances d'être un clone de type-3. Cela pourrait fausser le résultat de l'influence des caractéristiques du type, de la taille et de la distance. Toutefois, nous n'avons constaté aucune influence de ces caractéristiques. De même, l'effet du jugement de l'expert est basé sur l'ensemble général des clones comme expliqué dans la section 4.3.3. Comme cet ensemble est l'union de tous les groupes, et non un échantillonnage aléatoire de tous les clones, il pourrait être biaisé. Cependant, nous avons montré que les distributions des caractéristiques de ces clones sont semblables à celles de l'ensemble total des clones.

4.5.2 Validité interne

Notre évaluation empirique possède plusieurs obstacles à la validité interne. Tout d'abord, les juges ont pu communiquer entre eux au cours du processus d'évaluation et donc influencer leurs jugements. Toutefois, tous ont convenu à l'avance de ne pas parler de l'étude tant que celle-ci n'était pas complétée par tous les participants. Un deuxième obstacle à la validité interne tient au fait que tous les juges font partie de la même équipe de recherche, ce qui a pu influencer leurs réponses. Pour minimiser cet obstacle, toutes les données collectées au cours de cette étude sont disponibles pour examen et réplication. Un autre obstacle est lié au processus de construction des groupes. Nous l'avons minimisé en utilisant un algorithme de groupement réputé (*neural gas*) pour extraire les groupes de chacune des caractéristiques étudiées. Cet algorithme de groupement a été sélectionné car il a la réputation d'être très stable (il trouve toujours les mêmes groupes d'une exécution à l'autre). Le nombre d'échantillons que nous avons sélectionnés au sein de chaque groupe peut également ne pas être suffisant pour être représentatif dans certains groupes (notamment ceux avec une forte population tels que Σ-*Petit* et Δ-*Proche*). Malheureusement, inspecter manuellement des clones est très coûteux et nous avons été limités par le nombre total de clones qu'un juge peut analyser. L'un des juges externes de GumTree a déjà soumis plusieurs correctifs sur ce projet, ce qui aurait pu influencer son accord avec l'expert. Toutefois, lui-même ne se considère pas comme un expert de ce projet car son interven-

tion sur le code de GumTree a été très limitée. Un autre obstacle est lié à la suppression des commentaires dans les projets étudiés. Les juges externes ont pu avoir davantage de difficulté à classer les clones sans ces informations. Cet obstacle devrait être examiné dans une nouvelle étude afin de compléter nos résultats. Ainsi, l'impact des commentaires dans une telle étude pourrait être quantifié.

4.5.3 Validité externe

Les obstacles à la validité externe concernent la généralisation de nos résultats. Tout d'abord, nous avons concentré cette étude sur deux projets qui ne peuvent certainement pas représenter tous les projets du monde réel. Cependant, le nombre de projets qui peuvent être étudiés est limité par la nécessité de disposer d'un expert et par le nombre de clones qu'un juge peut évaluer. De plus, les deux projets sélectionnés utilisent Java comme principal langage de programmation. Par conséquent, les résultats pourraient être différents avec des projets utilisant d'autres langages de programmation. Néanmoins, Java est reconnu comme l'un des langages de programmation les plus populaires utilisés dans les projets logiciels [Bissyande *et al.*, 2013]. De plus, Walenstein *et al.* ont également trouvé que le choix d'un projet semble avoir un impact sur la fiabilité inter-juges [Walenstein *et al.*, 2003]. Davantage de validations sur de nouveaux projets écrits dans d'autres langages de programmation doivent être effectuées. Dans cette optique, nous fournissons toutes les données nécessaires pour reproduire cette étude et compléter nos résultats. Un deuxième obstacle est que nos résultats sont basés uniquement sur l'avis de quatre juges ; ils pourraient être modifiés avec d'autres juges. Bien que chaque projet étudié a un expert propre, nos résultats peuvent ne pas être généralisables à d'autres projets libres. Toutefois, tous les juges ayant participé à cette étude ont une forte expérience en programmation Java et aucun d'eux n'était étudiant. Un autre obstacle est lié à l'utilisation d'un seul outil de détection de clones dans notre étude. Une fois encore, le nombre de clones qu'un juge peut évaluer limite le nombre de détecteurs de clones qui peuvent être considérés. Cependant pour minimiser cet obstacle, nous nous basons sur les travaux de Wang *et al.* [2013] pour déterminer les paramètres de configuration de l'outil. La configuration recommandée par Wang *et al.* maximise l'ensemble des clones qui auraient été détectés par tous les principaux outils de détection de clones limitant ainsi cet obstacle à la validité externe. De plus, nous fournissons toutes les données nécessaires pour reproduire cette expérience avec d'autres outils de détection de clones qui seraient compatibles avec les génériques de Java. Nous avons étudié une seule question sur les clones dans cette étude. Nos résultats pourraient changer avec une autre question. Cependant, la question que nous avons posée est très représentative de l'utilisation des clones par les développeurs. Un dernier obstacle à la validité externe tient au fait que nous avons ignoré les clones de type-1 dans cette étude. Dans un précédent travail [Charpentier *et al.*, 2015], nous avons constaté que les développeurs ont un meilleur accord sur les clones de type-1 que sur ceux d'autres types. Sur la base de ce résultat, nous nous sommes concentrés sur d'autres caractéristiques malgré

l'absence d'experts dans ce précédent travail. Finalement, bien que les clones de type-1 auraient dû être inclus pour compléter cette étude, nos résultats ne sont pas affectés.

4.6 Conclusion

Dans ce chapitre, nous avons étudié la fiabilité de l'opinion des juges sur des clones liés à une tâche. Nous avons distingué deux types de juges : les experts et les juges externes. Nous avons observé que l'opinion des juges externes peut ne pas être fiable. Premièrement, la reproductibilité des réponses des juges externes semble ne pas être satisfaisante. Ceci indique qu'une session d'entraînement pourrait être bénéfique pour améliorer la reproductibilité des réponses. Ensuite, la fiabilité inter-juges semble médiocre pour les juges externes ainsi qu'entre juges externes et experts. Ceci pourrait aboutir à des *benchmarks* qui ne reflètent pas le jugement des experts du projet et ainsi conduire à des *benchmarks* de clones liés à une tâche qui seraient peu fiables. Nous avons montré que l'utilisation du vote de la majorité est une manière d'atténuer ce problème. Enfin, nous avons vu qu'il y a plusieurs caractéristiques qui semblent avoir un effet significatif sur le nombre de réponses identiques et différentes entre juges externes et experts. Tout d'abord, le projet choisi ainsi que l'expert ont une influence faible à modérée à ce propos. Cela signifie que certains projets produisent des clones qui ne peuvent pas être jugés par des juges externes. Ensuite, les vrais positifs semblent plus difficiles à analyser par des juges externes que les faux positifs. En effet, cette caractéristique a une forte influence sur le nombre de réponses identiques et différentes. Enfin, il semble plus difficile pour les juges externes d'être d'accord avec l'expert sur les petits clones. Par conséquent, il semble obligatoire de compter sur un expert du projet pour valider à la fois les vrais positifs et les petits clones.

CHAPITRE

5

Technique de détection de clones spécialisée

La détection de clones est importante lors de la maintenance logicielle aussi bien pour identifier que pour supprimer la duplication de code. Toutefois, les résultats des outils de détection de clones existants ne sont pas directement utilisables par les développeurs pour réaliser des tâches de maintenance logicielle. Le caractère généraliste de ces outils en est la raison et explique leur faible adoption par les développeurs. Notre objectif dans ce chapitre est donc de déterminer si l'utilisation de détecteurs de clones spécialisés dans une tâche peut aider les développeurs à gérer les clones présents dans leurs logiciels. Pour cela, nous avons choisi d'étudier la détection de clones pour de la réingénierie de code CSS (un langage de balisage). Nous proposons une technique automatisée pour extraire des mixins à partir de code CSS et éliminer ainsi la duplication. Notre technique permet un contrôle fin sur le code généré s'adaptant aux besoins des développeurs.

Sommaire

5.1 Introduction **78**
5.2 Contexte **80**
5.3 Notre approche **85**
5.4 Évaluation **94**
5.5 Conclusion **102**

5.1 Introduction

La détection de clones a un rôle important lors de la maintenance logicielle pour deux raisons principales. Premièrement, de par leur nature, les clones sont des candidats de choix pour une réingénierie. Deuxièmement, même quand leur élimination par réingénierie est impossible, leur identification a toujours un intérêt. Un exemple de cet intérêt est la propagation d'une correction d'un bogue à un ensemble de fragments de code dupliqués. Cependant, nous avons noté dans le chapitre 2 que les détecteurs de clones actuels ne produisent pas des résultats directement utilisables pour des tâches de maintenance logicielle. La spécialisation des détecteurs de clones pourrait alors être une solution pour accroître leur usage dans des tâches de maintenance logicielle. Dans ce chapitre, nous considérons l'exemple du langage de balisage CSS pour déterminer si l'utilisation de détecteurs de clones spécialisés dans une tâche peut aider les développeurs à gérer les clones présents dans leurs logiciels.

CSS est un langage utilisé pour décrire la présentation d'un document écrit dans un langage de balisage. Bien qu'il ait été conçu principalement pour définir le style visuel des pages web écrites en HTML, le langage est maintenant couramment utilisé pour créer des interfaces utilisateur visuellement attrayantes pour les applications web ainsi que des interfaces utilisateur pour de nombreuses applications de bureau et mobiles. Avec JavaScript, CSS est une technologie clef utilisée pour fournir une visualisation et navigation optimales avec un minimum de redimensionnement et de défilement pour une large gamme d'appareils allant des écrans d'ordinateurs à ceux des téléphones mobiles. Cette approche de sites web adaptatifs est rendue possible grâce à l'utilisation de CSS qui permet la séparation du contenu et de la présentation d'un document.

L'adoption massive du langage a conduit à une augmentation significative de la taille moyenne du code CSS. En effet, de nos jours, les applications web ont souvent plusieurs milliers de lignes de code CSS et par conséquent une grande quantité de duplication [Mazinanian *et al.*, 2014]. La présence de code dupliqué dans les logiciels est connue pour compliquer la maintenance et l'évolution du code [Monden *et al.*, 2002; Lozano et Wermelinger, 2008] car les corrections de bogues et modifications doivent être propagées à plusieurs endroits. Par conséquent, les développeurs de logiciels ont tendance à garder la quantité de code dupliqué aussi basse que possible.

Cependant, contrairement à la plupart des langages de programmation, CSS ne fournit pas de mécanismes avancés tels que des variables ou des fonctions afin de permettre la réutilisation du code. Pour pallier ce manque, plusieurs préprocesseurs CSS tels que Sass [1] et Less [2] ont récemment émergé comme extensions de CSS. Ils fournissent des mécanismes avancés tels que les variables et les mixins pour faciliter le développement et la maintenance de code CSS. Les développeurs qui utilisent ces langages invoquent un

1. http://sass-lang.com/
2. http://lesscss.org/

compilateur pour générer le code CSS de bas niveau correspondant qui peut ensuite être intégré dans leurs applications.

Parce qu'ils facilitent la réutilisation du code et évitent la duplication, les préprocesseurs CSS sont de plus en plus utilisés par les développeurs de grands projets bien connus tels que *Bootstrap*[3] et *Foundation*[4] (les deux cadriciels CSS les plus populaires). Cependant, de nombreux sites dépendent encore de code CSS de bas niveau. En effet, une étude récente[5] avec plus de 13 000 réponses de développeurs web a montré que près de la moitié d'entre eux n'utilise pas de préprocesseurs CSS.

Dans cette étude, nous soutenons que ce faible niveau d'adoption des préprocesseurs CSS en pratique est en partie dû à la complexité de migrer du code CSS existant et à l'absence d'outillage pour supporter ces migrations. Pour évaluer cette affirmation, nous avons mené un sondage informel. Nous avons tout d'abord récupéré une liste de projets utilisant le langage CSS à partir de la plate-forme GitHub[6]. Nous nous sommes basés sur les projets les plus consultés de septembre 2015 pour construire cette liste. Finalement, nous avons identifié cinq cents développeurs à partir des quarante-trois projets de notre liste. Nous avons envoyé un e-mail à chaque développeur demandant leur avis sur l'utilité d'un outil d'extraction automatique de mixins dans du code CSS. Soixante-quatre développeurs ont répondu anonymement à notre sondage pour un taux de réponse de 12,8%. Nous avons observé qu'environ la moitié des participants étaient intéressés par un tel outil. Pour les développeurs qui utilisent à la fois CSS et ses préprocesseurs, 65% étaient intéressés.

Globalement, cette étude apporte les contributions suivantes :

1. Nous introduisons une approche automatisée qui identifie des mixins à partir de code CSS. Nous avons implémenté la technique dans un logiciel *open source* appelé *Mocss*. Notre technique permet un contrôle fin sur le code généré s'adaptant aux besoins des développeurs.
2. Nous évaluons l'efficacité de notre approche avec quatre études de cas démontrant que notre outil peut aider les développeurs à éliminer la duplication de code en extrayant des mixins à partir de code CSS existant.

La structure de cette étude est la suivante. La section 5.2 introduit le langage CSS et ses extensions. Notre approche pour l'extraction de mixins à partir de code CSS est décrite dans la section 5.3 et une évaluation de son efficience est reportée dans la section 5.4. Nous concluons dans la section 5.5.

3. http://getbootstrap.coms/
4. http://foundation.zurb.com/
5. https://css-tricks.com/poll-results-popularity-of-css-preprocessors
6. https://github.com/trending?l=css&since=monthly

5.2 Contexte

CSS est un langage utilisé pour paramétrer le style visuel des documents web. Il permet la séparation du contenu et de la présentation des documents. Cette séparation des rôles facilite la réutilisation du code de présentation dans différents documents. Dans le reste de cette section, nous décrivons d'abord les mécanismes offerts par CSS pour décrire la présentation d'un document puis nous examinons les raisons de la présence de duplication en CSS.

5.2.1 Le langage CSS

Un code CSS se compose d'une liste de règles de style et chaque règle se compose d'un ou plusieurs sélecteurs ainsi que d'une liste de déclarations. Le listing 5.1 illustre la syntaxe d'une règle CSS.

LISTING 5.1 – Syntaxe d'une règle CSS.

```
1 selector_1, ..., selector_n {
2     property_1: value_1;
3     ...
4     property_n: value_n;
5 }
```

Les sélecteurs spécifient l'ensemble des éléments d'un document sur lequel un style s'applique. Il existe trois façons d'identifier un élément dans un document. Premièrement, un élément peut être sélectionné par son nom. Par exemple, `h1` sélectionne tous les éléments `h1` dans un document HTML. Deuxièmement, les éléments peuvent être sélectionnés en fonction des attributs qu'ils définissent tel qu'un identifiant unique (de la forme `#id1`) ou une classe qui regroupe plusieurs éléments (de la forme `.class1`). Troisièmement, les éléments peuvent être sélectionnés en fonction de leur position relative dans l'arborescence du document. Par exemple, `div > p` sélectionne tous les éléments `p` qui sont des enfants directs d'un élément `div`. En outre, les sélecteurs peuvent être combinés pour obtenir des sélections plus spécifiques. Dans ce cas, l'ordre des sélecteurs est important. Par exemple, `p .class1` sélectionne tous les éléments de la classe `class1` qui sont à l'intérieur d'éléments `p`, alors que `.class1 p` sélectionne tous les éléments `p` qui sont dans des éléments de classe `class1`.

Les déclarations d'une règle CSS définissent le style des éléments sélectionnés par cette règle. Une déclaration se compose d'une propriété et d'une valeur. Chaque propriété a un ensemble fini de valeurs possibles définies dans la spécification CSS.

Des règles de style différentes peuvent sélectionner les mêmes éléments et appliquer des valeurs aux mêmes propriétés. Lorsque plusieurs règles définissent des propriétés contradictoires pour un même élément, le sélecteur avec la plus grande spécificité [7] gagne (c'est-à-dire applique ses valeurs aux éléments sélectionnés). Enfin, si plusieurs sélecteurs ont la même spécificité, le sélecteur défini en dernier dans le fichier CSS gagne.

L'ordre des règles de style dans un fichier CSS a donc un impact sur le rendu des documents HTML et fait donc partie de la sémantique du fichier CSS. Considérons le document HTML présenté dans le listing 5.2 qui contient une balise `div` utilisant les règles de style à la fois des classes `class1` et `class2`. Considérons également les deux listings 5.3 et 5.4 qui reportent deux fichiers CSS identiques à l'exception de l'ordre des règles. L'utilisation de l'un ou l'autre des fichiers CSS aura un impact sur le rendu du document HTML. Dans le cas où le premier (respectivement le second) fichier CSS est lié au document CSS, ~~`content`~~ (respectivement `content`) sera affiché.

LISTING 5.2 – Une balise `div` utilisant les règles de style de deux classes.

```
1 <div class="class1 class2">content</div>
```

LISTING 5.3 – Un exemple de fichier CSS.

```
1 .class1 { text-decoration: underline; }
2 .class2 { text-decoration: line-through; }
```

LISTING 5.4 – Un autre exemple de fichier CSS.

```
1 .class2 { text-decoration: line-through; }
2 .class1 { text-decoration: underline; }
```

5.2.2 Gestion de la duplication en CSS

La duplication peut survenir à différents niveaux dans le code CSS. Le listing 5.5 montre des exemples de duplication au niveau des propriétés (`font-size`), au niveau des valeurs (`red`) et enfin au niveau des déclarations (`margin:0`). CSS n'offre aucun mécanisme pour

7. http://www.w3.org/TR/selectors/#specificity

éviter la duplication au niveau des propriétés et des valeurs. L'élimination de la duplication au niveau des déclarations se fait habituellement par regroupement des sélecteurs, comme illustré dans le listing 5.6. Cependant, ce mécanisme peut réduire la lisibilité du code. En effet, les déclarations d'un même sélecteur pourraient être placées dans plusieurs règles et le nombre de sélecteurs pour une même règle pourrait être très élevé. Une autre façon d'éliminer la duplication au niveau des déclarations est d'introduire une nouvelle classe au lieu de regrouper les sélecteurs. Cette solution implique néanmoins de modifier les documents HTML pour ajouter la référence à la nouvelle classe. Ainsi, l'absence de mécanismes avancés pour gérer la duplication a conduit à l'émergence de langages d'extension de CSS.

LISTING 5.5 – Duplication de propriété, valeur et déclaration en CSS.

```
1  #id1 {
2      font-size: 1.1em;
3      background: red;
4      margin: 0;
5  }
6
7  #id2 {
8      font-size: 1.2em;
9      color: red;
10     margin: 0;
11 }
```

LISTING 5.6 – Élimination de la duplication par regroupement des sélecteurs.

```
1  #id1, #id2 {
2      margin: 0;
3  }
4  #id1 {
5      font-size: 1.1em;
6      background: red;
7  }
8  #id2 {
9      font-size: 1.2em;
10     color: red;
11 }
```

Plusieurs langages sont apparus pour étendre CSS en fournissant des mécanismes avancés qui ne sont pas disponibles dans la dernière spécification de CSS (CSS3). Les langages d'extension les plus connus sont Sass, Less et Stylus [8]. Les programmes écrits dans ces langages sont compilés en code CSS de bas niveau. Ils prennent en charge les variables afin d'éviter la duplication au niveau des valeurs. Dans cette étude, nous nous concentrons sur la duplication aux niveaux des propriétés et des déclarations. Les langages d'extension proposent la notion de mixins pour éviter ce genre de duplication. Un mixin est similaire à une fonction macro et peut être paramétré par des variables. Un exemple de l'utilisation d'un mixin en Sass est présenté dans le listing 5.7 et le listing 5.8 montre le code CSS généré correspondant.

LISTING 5.7 – Un exemple de mixin en Sass qui factorise deux déclarations.

```
1  @mixin config($s) {
2      font-size: $s;
3      margin: 0;
4  }
5  #id1 {
6      @include config(1.1em);
7      background: red;
8  }
9  #id2 {
10     @include config(1.1em);
11     color: red;
12 }
```

Identifier des mixins dans du code CSS existant est difficile à cause du nombre de combinaisons possibles de toutes les déclarations. Ce nombre peut en effet exploser du fait que les mixins peuvent être combinés. Par conséquent, l'extraction de mixins à partir de fichiers CSS devient une tâche fastidieuse.

5.2.3 Travaux connexes

Mazinanian *et al.* proposent une approche pour éliminer la duplication de code dans des fichiers CSS [Mazinanian *et al.*, 2014]. Leur approche consiste à factoriser ensemble les déclarations CSS (de la forme `color:red`) communes. Leur approche contrôle également que les factorisations ainsi produites préservent la sémantique des fichiers CSS, c'est-à-dire que le rendu dans le navigateur doit être strictement identique avant et après la phase de réingénierie. Pour atteindre leur objectif, Mazinanian *et al.* ont tout d'abord

8. https://learnboost.github.io/stylus

LISTING 5.8 – Le code CSS généré à partir du code Sass du listing 5.7.

```
#id1 {
    font-size: 1.1em;
    margin: 0;
    background: red;
}

#id2 {
    font-size: 1.1em;
    margin: 0;
    color: red;
}
```

défini trois types de duplication de déclarations CSS. Un clone de type-1 correspond à des déclarations ayant des valeurs identiques. Un clone de type-2 représente des déclarations qui ont des valeurs équivalentes; la couleur *violet* est par exemple équivalente à sa notation hexadécimale *#EE82EE*. Enfin, un clone de type-3 représente l'équivalence entre des déclarations; la déclaration `font:italic .8em` est en fait équivalente aux deux déclarations `font-size:.8em` et `font-style:italic`. La première étape de leur approche pour la détection de duplication dans du code CSS consiste à construire un modèle du code CSS à analyser en le normalisant au préalable. Puis, les clones sont identifiés en comparant toutes les paires possibles de déclarations dans le modèle CSS et en vérifiant si elles sont égales (type-1) ou équivalentes (type-2 et type-3). La recherche des opportunités de factorisation est ensuite réalisée à l'aide de l'algorithme d'extraction de règles d'association *FP-Growth* [Han *et al.*, 2000] qui identifie les items fréquents en se basant sur une structure de données où les sélecteurs CSS représentent les transactions et les propriétés CSS représentent les items. Une fois les opportunités de factorisation de code CSS détectées elles sont classées selon leur impact sur la taille code. La dernière étape consiste à déterminer quelles sont parmi les opportunités de factorisation celles qui préservent la sémantique du code CSS initial. Pour cela, le code CSS après factorisation doit produire le même rendu que le code initial. En analysant les fichiers HTML sur lesquels s'appliquent le code CSS, ils vérifient que les règles CSS sont appliquées sur les éléments HTML dans le bon ordre. Cette vérification revient à résoudre un problème de satisfaction de contraintes pour déterminer une position appropriée pour chaque règle CSS dans le code généré. Finalement, à partir de leur nouvelle technique de détection, ils ont empiriquement constaté sur trente-huit sites web que le degré de duplication dans le code CSS est important allant de 40 à 90%.

TABLEAU 5.1 – Le contexte formel correspondant au code CSS du listing 5.9.

	font-size	font-size : 1em	font-size : 1.2em	font-size : 1.3em	padding	padding : 0	color	color : black	font-weight	font-weight : 100	font-weight : 200	margin	margin : 5px	margin : 10px
body	×	×			×	×								
a							×	×						
.info	×		×				×	×	×	×		×	×	
.error	×			×			×	×	×		×	×		×
#content	×	×			×	×	×	×	×	×				

5.3 Notre approche

Notre approche applique l'analyse formelle de concepts pour automatiser l'extraction de mixins dans des fichiers CSS. Les mixins sont identifiés à partir d'une sous-hiérarchie de Galois qui est un graphe orienté acyclique. Dans cette section, nous décrivons d'abord comment générer une sous-hiérarchie de Galois à partir d'un fichier CSS puis nous détaillons comment extraire des mixins de cette structure.

5.3.1 Génération de la sous-hiérarchie de Galois

L'analyse formelle de concepts [Ganter et Wille, 1997] (AFC) est une branche de la théorie des treillis [Birkhoff, 1948; Barbut et Monjardet, 1970] qui vise à automatiquement trouver des groupes d'*objets* qui partagent en commun un groupe d'*attributs*. AFC travaille sur une structure de données spécifique, nommée un *contexte formel*, dans lequel des objets sont décrits par plusieurs attributs. Un contexte formel est un triplet $K = (O, A, I)$, où O et A sont des ensembles finis d'objets et d'attributs respectivement et $I \subseteq O \times A$ est une relation binaire associant des objets avec des attributs : $(o, a) \in I$ si l'objet o possède l'attribut a.

Lors de la construction d'un contexte formel à partir d'un fichier CSS, les objets (O) sont des sélecteurs CSS (e.g. `h1`) et les attributs (A) sont à la fois des propriétés CSS (e.g. `color`) et des déclarations CSS (e.g. `color:black`). Ainsi, la duplication aux niveaux des propriétés et des déclarations peut être facilement détectée. Considérons le code CSS représenté dans le listing 5.9 pour illustrer la notion de contexte formel. Cet exemple définit cinq règles qui contiennent de une à quatre déclarations. La table 5.1 montre le contexte formel correspondant $|O| \times |A|$.

Soit $X \subseteq O$, $Y \subseteq A$, $f(X) = \{a \in A \mid \forall o \in X, (o, a) \in I\}$ et $g(Y) = \{o \in O \mid \forall a \in Y, (o, a) \in I\}$, alors $f(X)$ donne tous les attributs partagés par les objets contenus dans X et $g(Y)$ donne

LISTING 5.9 – Un exemple de code CSS.

```
1 body {
2         font-size: 1em;
3         padding: 0;
4 }
5 a {
6         color: black;
7 }
8 .info {
9     font-size: 1.2em;
10    color: black;
11    font-weight: 100;
12    margin: 5px;
13 }
14 .error {
15    font-size: 1.3em;
16    color: black;
17    font-weight: 200;
18    margin: 10px;
19 }
20 #content {
21    font-size: 1em;
22    padding: 0;
23    color: black;
24    font-weight: 100;
25 }
```

tous les objets partageant les attributs contenus dans Y. Par exemple, dans le contexte formel de la table 5.1, $f(\{body, .info\}) = \{font\text{-}size\}$ et $g(\{padding:0\}) = \{body, \#content\}$.

Un *concept* est une paire d'ensembles (X, Y) telle que $X = g(Y)$ et $Y = f(X)$. En d'autres termes, un concept est une collection maximale d'objets partageant des attributs communs. Dans le contexte formel de la table 5.1, les objets $\{.info, .error, \#content\}$ et les attributs $\{font\text{-}size, color, color:black, font\text{-}weight\}$ constituent un concept car ces attributs ensemble décrivent seulement ces objets et ces objets ensemble ne partagent pas d'autres attributs. À l'inverse, les objets $\{a, .error, \#content\}$ et les attributs $\{color, color:black\}$ ne forment pas un concept car ces attributs décrivent aussi l'objet *.info*. Pour un concept (X, Y), l'ensemble des objets X est nommé l'*extension* et l'ensemble des attributs Y est nommé l'*intension*.

L'ensemble de tous les concepts possibles extraits d'un contexte formel constitue un

ordre partiel complet et peut être ordonné dans un treillis. Les sous-concepts (respectivement super-concepts) d'un concept $c_i = (X_i, Y_i)$ sont les concepts $c_j = (X_j, Y_j)$ tels que $X_j \subseteq X_i$, ou de manière équivalente $Y_i \subseteq Y_j$ (respectivement $X_i \subseteq X_j$ ou $Y_j \subseteq Y_i$). Le résultat de l'application d'AFC sur un contexte formel est un *treillis de concepts*. La figure 5.1 montre le treillis de concepts correspondant au contexte formel de la table 5.1. La source d'un arc est le super-concept et la destination est le sous-concept. Le concept *top* représente les attributs partagés par tous les objets et le concept *bottom* représente les objets possédant tous les attributs. Dans cet exemple, le concept *top* a une intension vide et le concept *bottom* a une extension vide.

L'extension simplifiée (respectivement l'intension simplifiée) d'un concept (X, Y) est l'ensemble $X' \subseteq X$ (respectivement $Y' \subseteq Y$) des objets (respectivement des attributs) de ce concept qui ne sont pas dans l'extension de ses sous-concepts (respectivement dans l'intension de ses super-concepts). Les extensions et intensions simplifiées sont montrées en gras dans la figure 5.1. Un treillis simplifié est un treillis où les concepts sont représentés avec leur extension et intension simplifiées.

La sous-hiérarchie de Galois [Barbut et Monjardet, 1970] est une simplification d'un treillis de concepts. Les concepts avec une extension simplifiée ou intension simplifiée vides sont abandonnés. La figure 5.2 montre la sous-hiérarchie de Galois correspondant au treillis de concepts de la figure 5.1. Dans cet exemple, les concepts *top* et *bottom* ont été supprimés. Alors que le nombre de concepts dans un treillis peut être exponentiel, il est au plus égal au nombre d'objets et d'attributs ($|O| + |A|$) dans une sous-hiérarchie de Galois [Berry *et al.*, 2012]. Ainsi, les sous-hiérarchies de Galois — qui sont des graphes orientés acycliques — sont la pierre angulaire pour l'identification des mixins.

5.3.2 Extraction de mixins à partir d'une sous-hiérarchie de Galois

Notre algorithme pour l'extraction de mixins prend en entrée une liste topologiquement triée des concepts extraits d'une sous-hiérarchie de Galois et génère en sortie à la fois un ensemble de mixins et un ensemble de règles CSS. Nous définissons un mécanisme de filtrage pour rejeter les mixins non pertinents avant la phase de génération du code.

Filtrage des concepts non pertinents

Pour supprimer des mixins non pertinents, nous enlevons des concepts de la sous-hiérarchie de Galois. Supprimer un concept de la sous-hiérarchie de Galois modifie l'intension simplifiée de ses sous-concepts. Le contenu de l'intension simplifiée du concept supprimé est donc dupliqué dans l'intension simplifiée de chacun de ses sous-concepts. À l'inverse, l'extension simplifiée n'est pas modifiée.

Quatre informations sont disponibles pour décider si un concept doit être filtré ou non : son extension, son intension, ses super-concepts et ses sous-concepts. N'importe quelle fonction de filtrage peut être développée à partir de ces informations. De plus, plu-

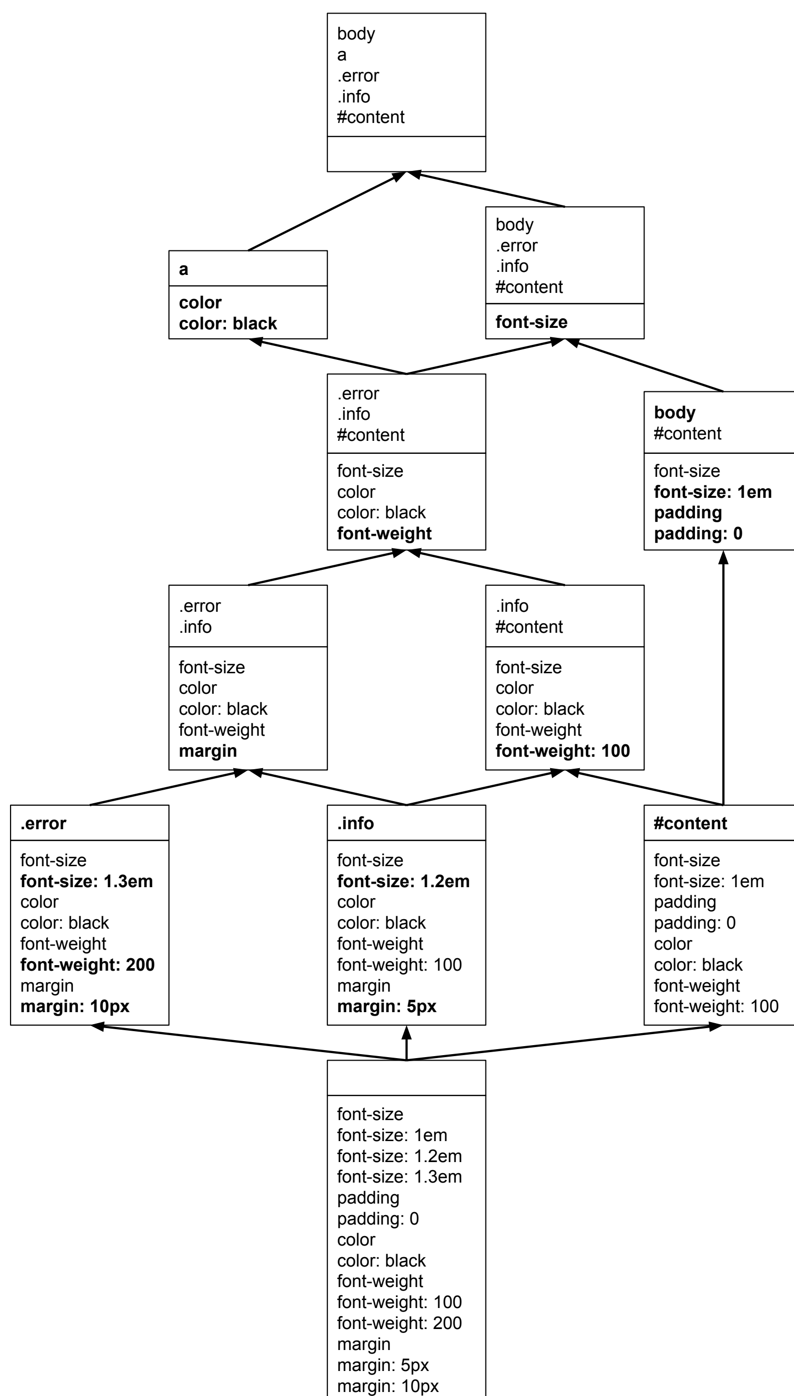


FIGURE 5.1 – Treillis de concepts correspondant au contexte formel de la table 5.1. Les extensions et intensions simplifiées sont en gras.

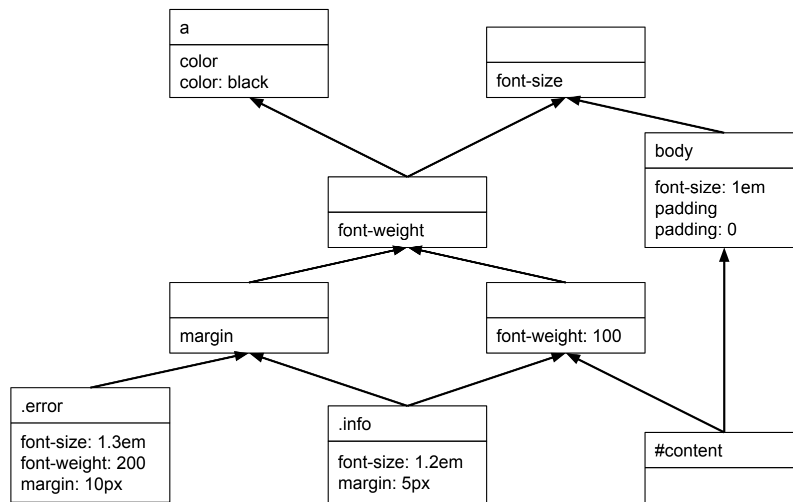


FIGURE 5.2 – Sous-hiérarchie de Galois obtenue à partir du treillis de concepts de la figure 5.1.

sieurs méthodes de filtrage peuvent être appliquées successivement. Dans cette étude, nous considérons un mixin inutile si : (1) il n'est pas suffisamment utilisé, (2) il ne factorise pas assez de déclarations ou (3) il prend trop de paramètres. Pour chaque concept, ces critères sont vérifiés en utilisant : (1) le nombre de sous-concepts directs, (2) le nombre de déclarations et propriétés dans l'intension simplifiée et (3) le nombre de propriétés dans l'intension. Des seuils sont associés à chacun de ces critères et les concepts qui ne satisfont pas l'un d'entre eux sont supprimés. Toutefois, les concepts avec une extension simplifiée non vide ne peuvent pas être rejetés car les sélecteurs qu'ils introduisent seraient perdus. Par conséquent, quand un concept avec une extension simplifiée non vide doit être filtré, seuls les arcs avec ses sous-concepts sont supprimés s'ils en existent.

La figure 5.3 montre le filtrage de concepts sur la sous-hiérarchie de Galois de la figure 5.2. La numérotation des concepts sur cette figure a pour unique but de faciliter leur identification dans la discussion ci-dessous. Dans cet exemple, nous avons fixé le nombre minimum d'enfants à 2, le nombre minimum de déclarations à 1, et le nombre maximum de paramètres du mixin à 3. *Concept#1* et *Concept#4* ont un seul sous-concept chacun. Par conséquent, l'arc avec leur sous-concept est enlevé. Leur intension simplifiée est ainsi dupliquée dans l'intension simplifiée de leur sous-concept. Les autres concepts avec une extension simplifiée non vide ne possèdent pas plus de trois propriétés et introduisent au moins une déclaration ou une propriété : ils sont donc conservés.

Nous proposons une autre méthode de filtrage pour adapter les mixins identifiés aux besoins des développeurs. Celle-ci peut être appliquée en combinaison de la méthode des seuils présentée ci-dessus. Les concepts factorisant différents types de propriétés sont considérés non pertinents et sont donc supprimés. Ainsi, les mixins ne contiennent que

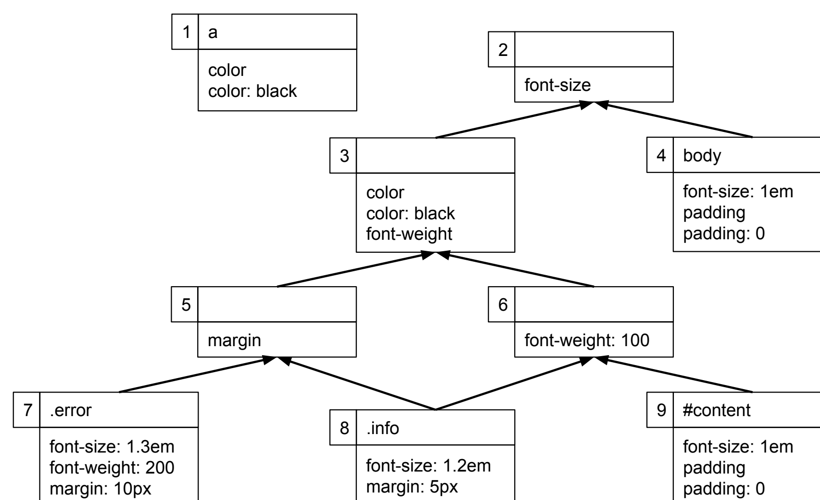


FIGURE 5.3 – Filtrage de concepts sur la sous-hiérarchie de Galois de la figure 5.2.

des propriétés connexes, telles que celles dédiées à la description de l'arrière-plan (propriété CSS *background-**), aux polices de caractères (propriété CSS *font-**) ou encore aux animations (propriété CSS *animation-**).

Génération des mixins et des règles CSS

Avant de générer les mixins et les règles CSS, les propriétés qui ont une déclaration correspondante dans l'intension simplifiée d'un concept sont supprimées. Par exemple, dans la figure 5.3, *Concept#1* a à la fois une propriété `color` et une déclaration `color:black`. La propriété `color` est donc supprimée.

Un mixin est généré lorsqu'un concept a au moins un sous-concept et une règle CSS est générée lorsqu'un concept a une extension simplifiée non vide. Par exemple, dans la figure 5.3, une règle CSS est générée à partir de *Concept#8* et un mixin à partir de *Concept#6*. Le listing 5.10 reporte le code Sass généré avec notre approche pour la sous-hiérarchie de Galois de la figure 5.3. Quatre mixins sont générés correspondant aux concepts possédant des sous-concepts. Quand un concept a à la fois une extension simplifiée non vide et un ou plusieurs sous-concepts, un mixin et une règle CSS sont générés. Les préprocesseurs CSS tels que Sass et Less ont un support limité de l'héritage de règles CSS : les sélecteurs imbriqués ne peuvent pas être hérités. Il est par exemple impossible avec ces langages d'hériter du sélecteur `.a #b`. Ceci est la raison pour laquelle nous générons à la fois un mixin et une règle CSS pour ces concepts.

Un appel de mixin est généré quand il y a un arc entre deux concepts. Appeler un mixin nécessite de fournir une valeur à chacun de ses paramètres. Une trace entre le paramètre et la propriété associée est donc conservée afin de garantir le bon ordre des paramètres dans

l'appel de mixin. Si un paramètre a une déclaration correspondante, c'est-à-dire une déclaration avec la même propriété, la valeur correspondante est utilisée dans l'appel du mixin. Les déclarations qui correspondent à un paramètre sont supprimées si elles appartiennent à l'intension simplifiée. Si un paramètre n'a pas de déclaration correspondante, une valeur abstraite est utilisée dans l'appel du mixin. Les valeurs abstraites seront fixées ultérieurement par les sous-concepts. Considérons l'appel de mixin entre *Concept#2* et *Concept#3* sur la figure 5.3. Comme *Concept#3* n'a pas une déclaration qui correspond à la propriété `font-size`, une valeur abstraite est utilisée dans l'appel de mixin (ligne 5 du listing 5.10). À l'inverse, la valeur `1em` est utilisée pour l'appel de mixin entre *Concept#2* et *Concept#4* (ligne 31 du listing 5.10).

Générer un mixin à partir d'un concept est un processus en trois étapes. En premier lieu, un nom générique est généré pour le mixin et la liste de tous les sélecteurs dont les propriétés et déclarations ont été factorisées est fournie en en-tête de la déclaration du mixin. Cette liste a pour objectif d'aider les utilisateurs à nommer les mixins. En second lieu, les appels de mixin sont générés pour chacun des super-concepts et les déclarations restantes dans l'intension simplifiée sont ajoutées au contenu du mixin. En troisième lieu, la liste des paramètres du mixin est calculée. Les propriétés de l'intension simplifiée et les valeurs abstraites dans les appels de mixins constituent la liste des paramètres. Par exemple, dans la figure 5.3, *Concept#2* génère un mixin avec un paramètre correspondant à sa propriété `font-size` (ligne 1 du listing 5.10). Il n'y a aucun autre paramètre car il n'a pas de super-concept. À l'inverse, *Concept#5* génère un mixin avec trois paramètres (ligne 12 du listing 5.10) : un correspond à sa propriété `margin` et les deux autres (`font-weight` et `font-size`) proviennent de ses super-concepts. Aucun ordre particulier n'est utilisé pour la liste des paramètres.

Générer une règle CSS à partir d'un concept est un processus en deux étapes. Tout d'abord, les sélecteurs sont obtenus à partir de tous les éléments contenus dans l'extension simplifiée joints par des virgules. Deuxièmement, comme pour la génération de mixins, les appels de mixins sont générés et les déclarations restantes dans l'intension simplifiée sont ajoutées au contenu de la règle CSS. La génération est alors terminée puisque les concepts avec une extension simplifiée non vide ont seulement des déclarations dans leur intension simplifiée. Par exemple, *Concept#9* dans la figure 5.3 génère une règle CSS avec une déclaration et un appel de mixin car il a un seul super-concept (ligne 23 du listing 5.10).

Il doit être mentionné que la compilation d'un code Sass en code CSS ne gère pas l'inclusion multiple d'une même déclaration. Pour illustrer ceci, considérons le code Sass du listing 5.10. La règle CSS avec le sélecteur `.info` (ligne 19) a deux appels de mixin et chacun d'eux inclut le mixin $m2$. Par conséquent, toutes les déclarations introduites dans $m2$ apparaîtront deux fois dans `.info`. Sass ne vérifie pas si les déclarations existent déjà avant de les insérer. Le seul problème vis à vis de ce comportement est l'augmentation de la taille du code CSS généré. Le listing 5.11 reporte le code CSS compilé à partir du code Sass du listing 5.10 pour la règle `.info`.

En fait, le problème survient quand un concept a au moins deux chemins vers le même

LISTING 5.10 – Code Sass généré à partir de la sous-hiérarchie de Galois de la figure 5.3.

```
1 @mixin m1($s) {
2     font-size: $s;
3 }
4 @mixin m2($w, $s) {
5     @include m1($s);
6     font-weight: $w;
7     color: black;
8 }
9 @mixin m3($s) {
10     @include m2(100, $s);
11 }
12 @mixin m4($m, $w, $s) {
13     @include m2($w, $s);
14     margin: $m;
15 }
16 a {
17     color: black;
18 }
19 .info {
20     @include m3(1.2em);
21     @include m4(5px, 100, 1.2em);
22 }
23 #content {
24     @include m3(1em);
25     padding: 0;
26 }
27 .error {
28     @include m4(10px, 200, 1.3em);
29 }
30 body {
31     @include m1(1em);
32     padding: 0;
33 }
```

super-concept; ce problème est connu sous le nom de « problème des diamants » dans le domaine des langages orientés objet. Pour le surmonter, nous calculons une arborescence couvrante sur la sous-hiérarchie de Galois pour le transformer en arbre. Comme pour l'étape de filtrage, l'intension simplifiée de certains concepts est dupliquée en raison

de la suppression d'arcs dans la sous-hiérarchie de Galois. Cette transformation est donc facultative car moins de possibilités de factorisation sont identifiées lorsqu'elle est activée.

LISTING 5.11 – Extrait du code CSS compilé à partir du code Sass du listing 5.10.

```
1 .info {
2     color: black; font-size: 1.2em; font-weight: 100;
3     color: black; font-size: 1.2em; font-weight: 100;
4     margin: 5px;
5 }
```

5.3.3 Préservation du rendu CSS initial

Dans la section 5.2.1, nous soulignons l'importance de l'ordre des règles de style dans un fichier CSS sur le rendu final d'un document web. Afin de préserver la sémantique d'un fichier CSS, nous raffinons les objets (les sélecteurs CSS) à inclure dans le contexte formel pour l'analyse formelle de concepts en ajoutant les numéros de ligne dans le fichier CSS initial. Ainsi, les objets avec le même nom peuvent être distingués par AFC.

Considérons l'extrait de fichier CSS du listing 5.12 qui contient deux règles portant sur le même sélecteur. Lorsque les objets ne sont que des sélecteurs CSS, le contexte formel contient un objet (*.info*) avec 4 attributs (*color*, *color :black*, *margin*, *margin :5px*) et génère seulement un concept : ({*.info*}, {*color, color :black, margin, margin :5px*}). À cette étape, nous perdons l'information que le sélecteur a été défini dans deux endroits différents. Enfin, l'unique concept ne génère qu'une seule règle CSS `.info` qui contient deux déclarations : `color:black` et `margin:5px`. Par conséquent, nous ne pouvons pas garantir que la sémantique du fichier CSS initial est préservée. A l'inverse, lors de l'ajout de l'information du numéro de ligne pour les sélecteurs CSS, le contexte formel contient deux objets : *.info1* et *.info4*. Ces identificateurs signifient que deux règles situées aux lignes 1 et 4 partagent le même sélecteur `.info`. Ce contexte formel génère deux concepts : ({*.info1*}, {*color, color :black*}) et ({*.info4*}, {*margin, margin :5px*}). Par conséquent, deux règles différentes sont générées. Elles sont ensuite triées pour être dans le même ordre que dans le fichier CSS initial. Dans le cas général, les sélecteurs regroupés dans une même règle par notre approche sont scindés en différentes règles à moins qu'ils n'étaient déjà groupés dans le fichier original. La duplication introduite lors de la séparation des sélecteurs en différentes règles peut être évitée à l'aide de mixins ; ces mixins sont nommés différemment pour les différencier de ceux générés à partir de la sous-hiérarchie de Galois.

La préservation de la sémantique du fichier CSS original est effectuée par défaut mais peut être désactivée. Nous appliquons la procédure suivante pour contrôler la validité de

LISTING 5.12 – Un exemple de fichier CSS.

```
1 .info { color: black; }
2 .info { margin: 5px; }
```

la sémantique des fichiers générés par notre approche. Une fois le fichier contenant les mixins généré nous le compilons et comparons le résultat au fichier CSS original. Pour cela, nous vérifions qu'exactement toutes les règles CSS du fichier original sont présentes dans le même ordre et qu'elles contiennent exactement le même ensemble de déclarations.

5.3.4 Implémentation de l'outil

Nous avons implémenté notre approche pour identifier automatiquement des mixins à partir de code CSS dans un outil *open source* Java, nommé *Mocss*, que nous mettons à la disposition du public sur la plate-forme GitHub [9]. *Mocss* prend en entrée un fichier CSS et génère le code Sass équivalent qui utilise des mixins pour réduire la duplication de code. L'outil prend en charge tous les types de sélecteurs et déclarations y compris les pseudo-éléments (tels que `::first-line` utilisé pour ajouter un style spécial à la première ligne d'un texte) et les pseudo-classes (comme `:hover` utilisé pour sélectionner des liens survolés par la souris). Toutefois, il ne supporte pas pour le moment les *media queries*. Le code non supporté est reproduit à la fin du fichier généré. Il est actuellement distribué comme un programme en ligne de commande qui accepte plusieurs paramètres pour contrôler le code généré, comme décrit dans la table 5.2. Nous nous appuyons sur les travaux de Mazinanian et Tsantalis [2016] pour définir les valeurs par défaut des trois premières options. Ils ont mené une étude empirique sur cent cinquante sites web développés avec des préprocesseurs CSS et ont examiné les usages de quatre fonctionnalités de ces langages d'extension CSS incluant les mixins. Ils ont découvert que 63% des mixins sont appelés deux fois ou plus (*min-children* = 2) et 68% des mixins ont un ou zéro paramètre (*max-parameters* = 1). Nous avons empiriquement évalué que 3 semble une valeur raisonnable pour l'option *min-declarations*.

5.4 Évaluation

Pour évaluer notre approche, nous effectuons deux expériences et quatre études de cas. Tout d'abord, nous évaluons les performances de notre outil en termes de temps de calcul, pour assurer qu'il est utilisable par les développeurs sur des projets réels. Deuxièmement, nous évaluons l'effet des seuils décrits dans la section précédente sur le nombre de mixins

9. https://github.com/acharpen/mocss

TABLEAU 5.2 – Options de l'outil *Mocss*.

Option	**Description**
max-parameters *n*	Évite les mixins avec plus de *n* paramètres. (valeur par défaut : 1)
min-children *n*	Évite les mixins utilisés moins de *n* fois. (valeur par défaut : 2)
min-declarations *n*	Évite les mixins factorisant moins de *n* déclarations. (valeur par défaut : 3)
groups-filter	Génère que des mixins factorisant des propriétés liées. (valeur par défaut : `false`)
no-duplicates-into-rule	Évite la duplication telle que présentée dans le listing 5.11. (valeur par défaut : `false`)
keep-semantic	Préserve la sémantique du fichier initial : • `full` : la sémantique est préservée et des mixins additionnels sont générés pour éviter la duplication; • `slight` : la sémantique est préservée sans créer de nouveaux mixins; • `none` : la sémantique n'est pas préservée. (valeur par défaut : `full`)

générés. Enfin, nous décrivons quatre études de cas dans lesquelles notre approche a été utilisée par des développeurs professionnels sur du code CSS provenant de projets sur lesquels ils travaillent.

5.4.1 Expérimentations

Dans les expériences décrites dans le reste de cette section, nous appliquons notre approche sur 108 fichiers CSS. Nous utilisons les données de l'étude menée par Mazinanian *et al.* [2014] qui contient les fichiers CSS de trente-huit applications web du monde réel. Ils ont extrait cent cinquante-cinq fichiers CSS à partir de ces applications web. Cependant, certains d'entre eux contiennent du code CSS invalide, et peuvent par conséquent provoquer des erreurs d'analyse. La table 5.3 synthétise, pour chacune des trente-huit applications web, le nombre de fichiers CSS analysés par Mazinanian *et al.* et le nombre de fichiers CSS analysés avec succès par l'analyseur syntaxique utilisé dans notre outil [10]. En conséquence, nous analysons 108 fichiers CSS (69% du total) tandis que Mazinanian *et al.* n'en analysent que 90 (58%).

10. https://github.com/phax/ph-css/releases/tag/ph-css-4.0.1

TABLEAU 5.3 – Sujets sélectionnés.

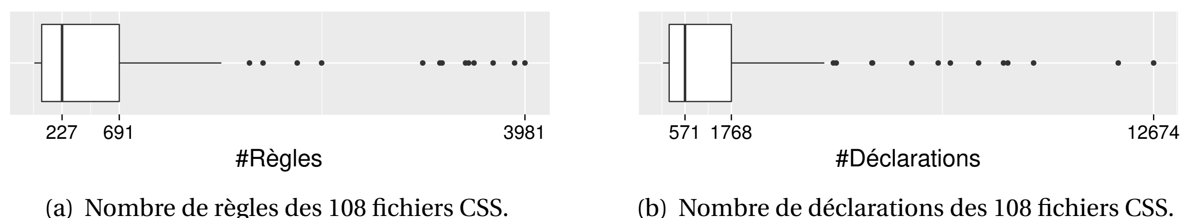
Application web	#Fichiers[1]	#Fichiers[2]	Application web	#Fichiers[1]	#Fichiers[2]
Facebook	6	7	Pinterest	2	3
YouTube	4	5	Reddit	1	2
Twitter	2	1	Tumblr.com	2	3
YahooMail	3	9	Wordpress.org	1	3
Outlook.com	6	11	Vimeo.com	3	2
Gmail	5	5	Igloo	2	2
Github	2	1	Phormer	1	1
Amazon.ca	3	3	BeckerElectric	1	0
Ebay	2	2	Equus	1	1
About.com	1	1	ProToolsExpress	1	3
Alibaba	3	0	UniqueVanities	3	3
Apple.ca	3	4	ICSE12	3	4
BBC	3	1	EmployeeSolutions	3	5
CNN	1	0	SyncCreative	3	3
Craiglist	1	3	GlobalTVBC	5	6
Imgur	2	1	Lenovo	1	1
Microsoft	1	1	MountainEquip	2	3
MSN	1	1	Staples	2	1
Paypal	1	3	MSNWeather	3	3

[1] Nombre de fichiers CSS utilisés dans l'étude de Mazinanian *et al.* [2014].
[2] Nombre de fichiers CSS utilisés dans notre étude.

Évaluation des performances

Nous évaluons notre outil selon deux caractéristiques d'un fichier CSS : le nombre de règles et le nombre de déclarations. La figure 5.4 montre les distributions de ces deux caractéristiques pour les 108 fichiers CSS utilisés dans notre étude. Nous pouvons noter que 75% de ces fichiers CSS ont moins de 691 règles (respectivement 1 768 déclarations) et que la médiane est de 227 règles (respectivement 571 déclarations). Le nombre maximum de règles (respectivement de déclarations) dans cet ensemble de fichiers CSS est 3 981 (respectivement 12 674).

Nous avons empiriquement évalué que *keep-semantic* est la seule option ayant un impact sur le temps d'exécution. En effet, en conservant l'ordre des sélecteurs, davantage d'objets sont inclus dans le contexte formel et donc plus de concepts sont insérés dans la sous-hiérarchie de Galois générée. Par souci de lisibilité, nous présentons uniquement les performances du temps d'exécution d'une seule configuration : celle par défaut car elle est la plus susceptible d'être utilisée et elle préserve la sémantique. En outre, nous fournis-



(a) Nombre de règles des 108 fichiers CSS.

(b) Nombre de déclarations des 108 fichiers CSS.

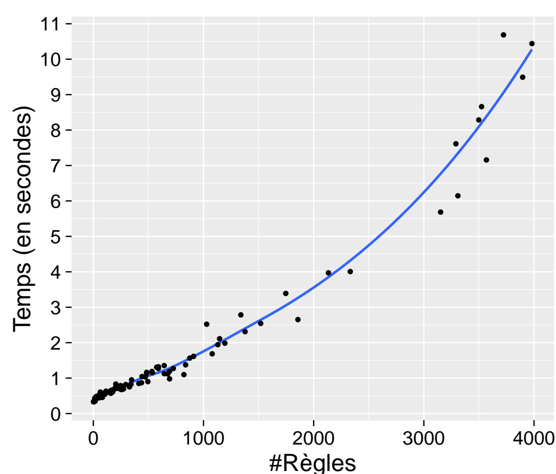
FIGURE 5.4 – Caractéristiques des 108 fichiers CSS utilisés dans notre expérience.

sons toutes les données nécessaires pour analyser les performances du temps d'exécution d'autres configurations. Nous exécutons notre outil dix fois sur chacun des fichiers 108 CSS et traçons la médiane de ces dix exécutions sur la figure 5.5. Les courbes sur les deux figures représentent l'interpolation des temps médians. Notre machine de test possède un processeur Core i7-4600U cadencé à 2,10Ghz avec 16GB de mémoire. Dans l'ensemble, *Mocss* traite 63% des fichiers CSS de notre corpus en moins d'une seconde, et 83% en moins de deux secondes. Ainsi, dans la plupart des cas, les résultats de *Mocss* sont quasi immédiats. Plus précisément, 85% des fichiers CSS qui ont un nombre de règles inférieur à 691 ou un nombre de déclarations inférieur à 1 768 sont traités par *Mocss* en moins d'une seconde, et la totalité en moins de deux secondes. Pour les plus gros fichiers de notre corpus, le temps d'exécution de notre outil est d'environ onze secondes, ce qui est encore acceptable. Pour conclure, cette expérience montre que notre outil est utilisable par les développeurs sur des projets réels.

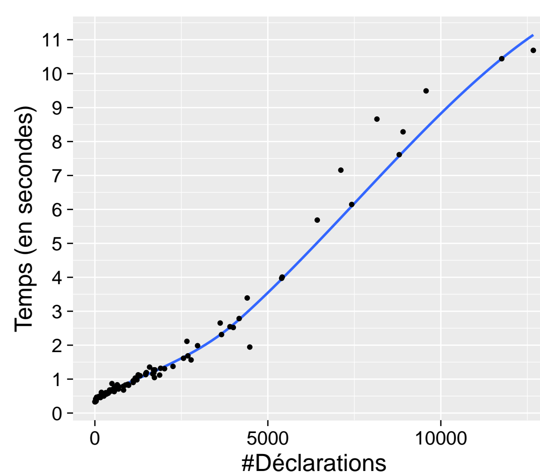
Évaluation des seuils

Notre outil a trois seuils qui permettent un contrôle fin des mixins générés : *min-children*, *min-declarations* et *max-parameters*. Nous évaluons chacun d'eux individuellement sur l'ensemble des 108 fichiers CSS avec des valeurs allant de 0 à 10. Lorsqu'un seuil est évalué les autres sont désactivés afin de garantir la validité des mesures. Par conséquent, pour chaque fichier CSS et chaque seuil, nous obtenons onze valeurs correspondant chacune au nombre de mixins générés par *Moccs* pour une configuration spécifique sur le fichier considéré. Pour des raisons de lisibilité, nous reportons uniquement les résultats pour deux fichiers CSS dans la figure 5.6 : le fichier ayant le nombre maximal de règles CSS et celui avec un nombre de règles proche de la médiane.

Dans l'ensemble, nous pouvons noter que les seuils permettent de réduire considérablement le nombre de mixins générés. Les seuils *min-children* et *min-declarations* ont un plus grand impact sur le nombre de mixins générés que *max-parameters*. Le nombre de mixins générés diminue très rapidement quand une valeur supérieure ou égale à deux est fixée pour les seuils *min-children* et *min-declarations*. En outre, il n'y a que quelques



(a) Temps d'exécution en fonction du nombre de règles.



(b) Temps d'exécution en fonction du nombre de déclarations.

FIGURE 5.5 – Temps d'exécution de notre outil sur les 108 fichiers CSS en fonction du nombre de règles (à gauche) et du nombre de déclarations (à droite).

mixins avec plus de trois paramètres : seuls quelques mixins supplémentaires sont générés quand une valeur supérieure à trois est fixée pour ce seuil.

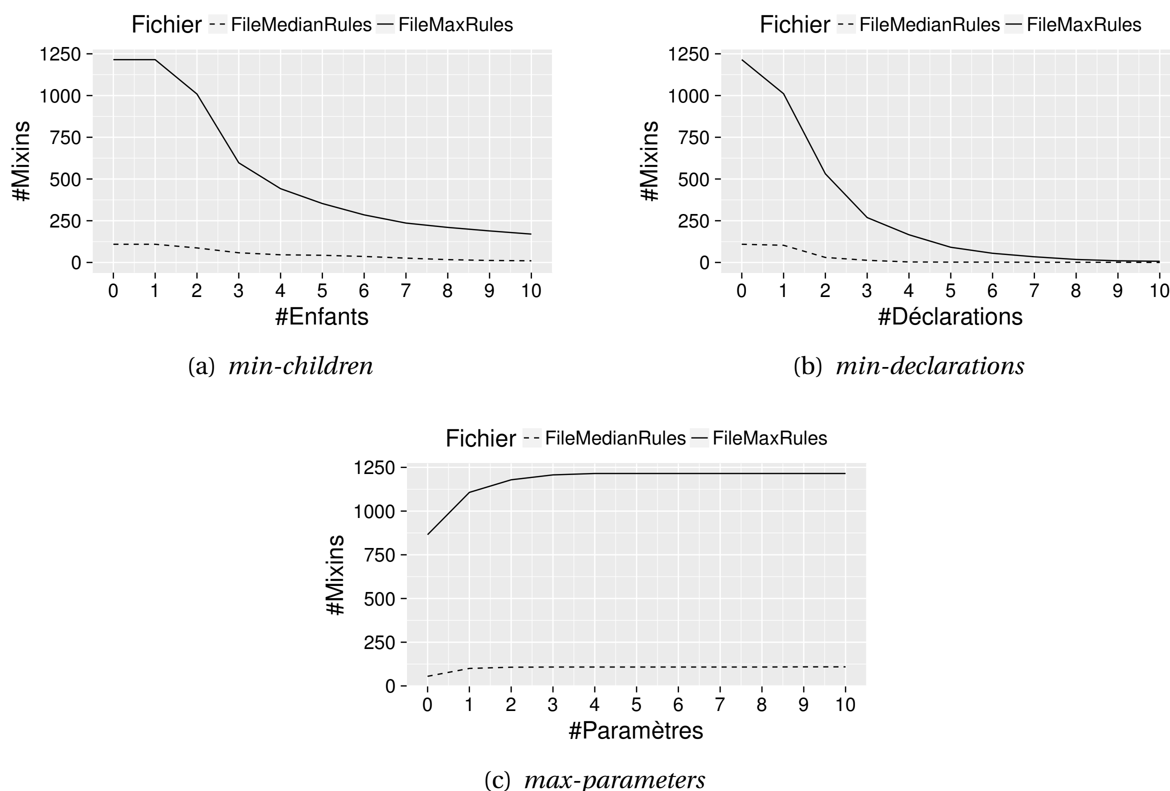
5.4.2 Études de cas

Les études de cas que nous menons visent à obtenir un aperçu de la pertinence réelle de notre approche. Par conséquent, nous sollicitons l'opinion de développeurs qui sont plus que familiers avec CSS. Nous étudions la pertinence des seuils et des options de configuration de notre approche ainsi que la sortie générée.

Méthodologie de recherche

Quatre développeurs ont participé à l'évaluation de notre approche. Trois sont développeurs web dans des entreprises locales et travaillent quotidiennement avec CSS ; ils sont experts sur CSS et ses langages d'extension, y compris Sass. Le quatrième est un membre du corps professoral de notre université. Il donne des cours sur les technologies du web à l'université, dont CSS et ses préprocesseurs.

Nous menons une enquête par le biais d'un formulaire Google qui comprend des questions ouvertes. Les participants ont ainsi un temps illimité pour remplir le questionnaire. Ils sont invités à évaluer notre outil sur les sites web qu'ils ont conçus. Chaque participant est autorisé à effectuer l'évaluation sur plusieurs projets. Pour chacun d'eux, un procédé en deux étapes doit être effectué. D'abord, le participant exécute l'outil avec sa configura-



(a) *min-children*

(b) *min-declarations*

(c) *max-parameters*

FIGURE 5.6 – Évolution du nombre de mixins générés en fonction de chacun des trois seuils : (a) *min-children*, (b) *min-declarations* et (c) *max-parameters*.

tion par défaut sur un fichier CSS et doit déterminer le nombre de mixins pertinents dans l'ensemble des résultats. Dans le cas où il trouve quelques mixins non pertinents, il lui est demandé d'en expliquer si possible les raisons. Cette étape vise à étudier les valeurs par défaut des paramètres de configuration de l'outil. De manière analogue, la deuxième étape vise à évaluer l'impact des seuils sur le nombre de mixins pertinents générés. Cette étape peut être ignorée si la configuration par défaut n'identifie que des mixins pertinents. Les participants sont invités à tester différentes valeurs pour les paramètres et de déterminer la configuration qui produit les meilleurs résultats, c'est-à-dire le ratio le plus élevé de mixins pertinents. Une fois le sondage terminé, les participants sont invités à donner leurs sentiments sur l'outil.

La table 5.4 présente les caractéristiques des sites web utilisés dans les études de cas, et la table 5.5 reporte les résultats bruts. *Participant#1* a évalué notre outil sur trois fichiers CSS. Nous pouvons noter que les deux premiers sont de taille moyenne et le dernier de grande taille. Les deux premiers fichiers sont en CSS tandis que le dernier est en Sass. Pour ce dernier, la version compilée est utilisée comme entrée de notre approche. En évaluant

TABLEAU 5.4 – Caractéristiques des sites web utilisés dans les études de cas.

Participant	Origine	Fichier généré	#Règles	#Déclarations
1	professionnel	oui (avec Sass)	2 819	6 133
1	personnel	non	194	721
1	personnel	non	104	327
2	professionnel	non	327	1 014
3	personnel	non	258	821
4	personnel	non	97	255

TABLEAU 5.5 – Caractéristiques des mixins identifiés par la configuration par défaut ainsi que la meilleure pour chaque étude de cas.

	Configuration par défaut		Meilleure configuration[1]	
Participant	#Total mixins	%Mixins pertinents	#Total mixins	%Mixins pertinents
1	203	97%	118	100%
1	15	100%	-	-
1	11	100%	-	-
2	37	32%	-	-
3	25	80%	19	100%
4	5	100%	-	-

[1] Les deux dernières colonnes ne sont pas remplies quand la configuration par défaut est identique à la meilleure.

notre outil sur du code CSS généré par Sass, nous pouvons évaluer dans quelle mesure notre approche trouve des mixins présents initialement dans le fichier Sass. *Participant#2* et *Participant#3* ont mené l'étude sur des fichiers CSS de taille moyenne, alors que le *Participant#4* — le membre du corps professoral de notre université — a utilisé un fichier CSS de petite taille.

Résultats

La configuration par défaut de notre outil rapporte 203 mixins sur le premier fichier CSS considéré par *Participant#1*. Selon lui, 198 (97%) étaient pertinents et 5 n'étaient pas assez appelés pour être utiles. Il a modifié la valeur du seuil *min-declarations* de 3 à 4 pour obtenir la meilleure configuration. Cette dernière a identifié 118 mixins qui sont tous pertinents. Pour les deux autres fichiers CSS qu'il a analysés avec notre outil, la configuration par défaut a identifié respectivement 15 et 11 mixins. Dans les deux cas, tous les mixins

étaient pertinents. Dans l'ensemble, il a trouvé l'outil facile à utiliser et très intéressante la possibilité de migrer un site web de CSS vers Sass.

Pour *participant#2*, 12 mixins des 37 (32%) identifiés par l'outil étaient pertinents. Les mixins non pertinents avaient trop de paramètres et factorisaient des propriétés qui n'étaient pas liées. Il n'a pas trouvé une meilleure configuration que celle par défaut. Selon lui, l'outil devrait proposer de factoriser uniquement des propriétés liées. Nous avons trouvé sa suggestion vraiment intéressante et avons discuté à ce sujet par e-mails. Nous avons finalement implémenté cette nouvelle manière de filtrer des mixins (voir section 5.3.2). Néanmoins, il n'a pas eu le temps de tester la nouvelle version de notre outil.

Pour *participant#3*, la configuration par défaut a identifié 25 mixins ; 20 (80%) étaient pertinents. Comme pour *participant#1*, certains mixins étaient non pertinents car pas assez utilisés. Afin d'atteindre la meilleure configuration, *participant#3* a augmenté la valeur du seuil *min-children* de 2 à 3. Cette configuration personnalisée a conduit à 19 mixins qui sont tous pertinents. Finalement, elle a trouvé que la détection de mixins à partir de code CSS existant est utile et que notre outil est capable d'identifier des mixins intéressants.

Participant#4 a constaté que tous les mixins rapportés par la configuration par défaut étaient pertinents. Il a trouvé la détection de mixins dans du code CSS intéressante et a constaté que notre approche répondait correctement à ses besoins.

Tous les participants ont constaté que le temps d'exécution de l'outil est plus que satisfaisant. En outre, la plupart des participants ont constaté que le nommage des mixins a posteriori n'est pas une tâche difficile. Ils ont indiqué que le commentaire inséré par l'outil devant la définition de chaque mixin aide grandement à exécuter cette tâche de nommage. Ce commentaire comprend le nom des sélecteurs à partir desquels les déclarations ont été factorisées.

Validité des études de cas

Nos études de cas ont un obstacle à la validité interne : les participants sont des contacts de notre groupe de recherche et donc leurs réponses peuvent avoir été influencées. Néanmoins, ce sont tous des professionnels et nous sommes donc confiants qu'ils ont donné des réponses impartiales.

Les obstacles à la validité externe réfèrent à la généralisation de nos résultats. Tout d'abord, le nombre de fichiers CSS évalués par chaque participant pourrait ne pas être suffisant. Des études de cas menées par ces mêmes participants sur d'autres fichiers CSS pourraient conduire à des résultats différents. Pour atténuer cet obstacle, les participants ont choisi seuls les fichiers CSS sur lesquels ils ont évalué l'outil et ils avaient également la possibilité de mener autant d'évaluations qu'ils le souhaitaient. Deuxièmement, seulement quatre développeurs ont évalué notre outil. Cependant, ils sont tous des experts sur CSS et ses préprocesseurs, et leur participation à l'enquête était volontaire. Davantage de validations avec de nouveaux participants doivent être effectuées pour augmenter la gé-

néralisation des résultats. Néanmoins, notre outil semble avoir une pertinence dans le monde réel.

5.5 Conclusion

Le langage CSS est largement adopté dans le développement web et il est maintenant commun pour des projets web d'avoir plusieurs milliers de lignes de code CSS. Bien que son utilisation soit presque obligatoire, le langage ne dispose pas de fonctionnalités avancées pour supporter la réutilisation et la structuration du code. Ainsi, des langages d'extension tels que Sass ont récemment vu le jour pour étendre les capacités de CSS. Les programmes écrits dans ces langages sont ensuite compilés en code CSS de bas niveau. Cependant, la migration de code CSS existant vers du Sass est difficile pour les développeurs web en raison de l'absence d'outil adapté.

Dans cette étude, nous avons présenté une nouvelle approche pour automatiquement identifier et supprimer la duplication dans du code CSS. Notre approche repose sur l'analyse formelle de concepts pour automatiser l'identification et l'extraction de mixins. Nous avons implémenté notre approche dans un outil nommé *Mocss* qui génère automatiquement du code Sass à partir de code CSS existant. La seule autre étude sur la recherche de factorisation dans du code CSS a été menée par Mazinanian *et al.* [2014]. Notre étude apporte une réelle contribution par rapport à cette dernière car elle est en mesure d'identifier et de supprimer la duplication aux niveaux propriétés et déclarations.

Nous avons mené une expérimentation sur 108 fichiers CSS extraits d'applications web du monde réel et avons constaté que notre outil permet un contrôle fin du code généré et a également des performances plus que satisfaisantes. Par conséquent, il semble que l'utilisation d'un détecteur de clones spécialisé peut concrètement aider les développeurs à gérer les clones présents dans leurs logiciels. De plus, nous avons proposé une nouvelle méthodologie pour l'évaluation de détecteurs de clones spécialisés reposant sur des études de cas. Celles-ci ont permis d'évaluer la pertinence réelle de notre approche et ont démontré que notre outil aide les développeurs à supprimer la duplication de code via l'extraction de mixins à partir de code CSS existant. Finalement, les développeurs qui ont testé notre outil ont apprécié la possibilité qu'il offre de migrer automatiquement du code CSS existant vers du Sass.

Chapitre 6

Conclusion

La détection de clones est un enjeu important pour améliorer la qualité logicielle et contribuer ainsi au succès des logiciels [Ralph et Kelly, 2014]. Plusieurs techniques ont par conséquent été proposées [Roy *et al.*, 2009] pour l'identification de différents types de clones. Toutefois, les outils de détection de clones ne sont pas ou peu utilisés par les développeurs et mainteneurs logiciels. Une première raison est que déterminer l'outil de détection de clones le plus adapté à une tâche donnée est difficile. En effet, comme les résultats des détecteurs de clones varient fortement selon la technique de détection sous-jacente et la configuration utilisée [Wang *et al.*, 2013], leur comparaison et évaluation sont nécessaires. Toutefois, la méthodologie existante [Bellon *et al.*, 2007] pour comparer et évaluer les détecteurs de clones peut être insuffisante ou inadaptée car pas suffisamment fiable. Une seconde raison est que le caractère trop généraliste des détecteurs de clones actuels rend leurs résultats difficilement utilisables. En effet, les outils existants ont la particularité de rechercher des clones qui ne sont destinés à aucune tâche particulière. Or, l'utilité d'un clone dépend de l'utilisateur et de la tâche qu'il doit servir.

Dans la suite de ce chapitre, nous dressons un bilan des trois contributions de notre thèse et présentons quelques perspectives de recherche.

6.1 Résumé des contributions

Les contributions de cette thèse sont doubles. Dans un premier temps, nous évaluons si la méthodologie pour comparer et évaluer les détecteurs de clones est pertinente pour déterminer l'outil le plus adapté à une tâche fixée. Dans un second temps, nous examinons si le recours à des détecteurs de clones spécialisés dans une tâche peut aider les développeurs à gérer les clones présents dans leurs logiciels.

Évaluation empirique du *benchmark* de clones de Bellon

Notre première contribution, présentée dans le chapitre 3, a pour but de déterminer si la construction et la validation de *benchmarks* par une unique personne garantissent des résultats fiables. L'étude menée dans ce chapitre est une évaluation empirique du *benchmark* de clones proposé par Bellon [2007] qui s'est imposé comme une référence pour la comparaison et l'évaluation de détecteurs de clones. Pour mener cette étude, nous avons sollicité l'opinion de dix-huit personnes sur un sous-ensemble du *benchmark* construit par Bellon.

Notre étude révèle premièrement que le corpus de référence de Bellon contient une part significative de clones contestables : pour plus de la moitié des clones examinés, les participants ont un avis différent de celui de Bellon. Deuxièmement, nous montrons que les mesures de précision et de rappel peuvent être significativement affectées par le niveau de confiance des clones du corpus de référence et donc que les résultats dérivés du *benchmark* de Bellon doivent être interprétés avec précaution. Ainsi, la participation de plusieurs personnes à la construction d'un *benchmark* de clones a un fort impact sur ses résultats. Enfin, notre examen de trois caractéristiques de clones montre que seul le type est corrélé au niveau de confiance : les clones de type-1 ont un niveau de confiance élevé dès lors qu'ils reçoivent au moins une opinion positive. La principale leçon que nous tirons de cette étude est qu'un *benchmark* de clones ne peut être construit par une unique personne.

Ces travaux ont fait l'objet d'une publication dans la conférence *Evaluation and Assessment in Software Engineering (2015)* [Charpentier *et al.*, 2015].

Fiabilité des juges dans la construction de *benchmarks*

Notre deuxième contribution, présentée dans le chapitre 4, est une extension de la première. Notre objectif est de déterminer si la méthodologie actuelle de construction de *benchmarks* est adaptée pour concevoir des *benchmarks* permettant une évaluation fiable des détecteurs de clones pour une tâche particulière. Cette problématique se pose car il n'existe pas de relation entre la manière dont sont définis les *benchmarks* existants et leur usage. Or, ces derniers doivent identifier l'outil le plus adapté pour chaque utilisateur et chaque tâche. Pour compléter notre première étude, nous sollicitons quatre juges pour analyser les résultats produits par un détecteur de clones sur deux logiciels. Pour chacun des deux logiciels, trois des juges n'ont pas ou peu de connaissances sur le code du logiciel et le dernier en est l'un des principaux développeurs ; il y a donc trois juges dits « externes » et un expert. Les deux experts sont les oracles décidant si un clone est un vrai ou faux positif selon deux tâches classiques de maintenance logicielle : co-évolution et réingénierie.

Premièrement, notre étude montre que la reproductibilité des réponses des juges externes semble ne pas être satisfaisante. Une session d'entraînement pourrait être bénéfique pour améliorer la reproductibilité des réponses. Deuxièmement, la fiabilité inter-

juges semble médiocre pour les juges externes ainsi qu'entre juges externes et experts. Ceci pourrait aboutir à des *benchmarks* qui ne reflètent pas le jugement des experts du projet et ainsi conduire à des *benchmarks* de clones liés à une tâche qui seraient peu fiables. Nous montrons que l'utilisation du vote de la majorité est une manière d'atténuer ce problème. Enfin, notre étude révèle que plusieurs caractéristiques semblent avoir un effet significatif sur le nombre de réponses identiques et différentes entre juges externes et experts. Tout d'abord, certains projets produisent des clones qui ne peuvent pas être examinés par des juges externes. Ensuite, les vrais positifs semblent plus difficiles à analyser par des juges externes que les faux positifs. Enfin, il semble plus difficile pour des juges externes d'être d'accord avec des experts sur les petits clones. Par conséquent, il semble obligatoire de compter sur un expert du projet pour valider à la fois les vrais positifs et les petits clones.

Ces travaux ont fait l'objet d'une publication dans le journal *Empirical Software Engineering (2016)* [Charpentier *et al.*, 2016a].

Extraction automatique de mixins dans du code CSS

Notre troisième contribution, présentée dans le chapitre 5, s'appuie sur les résultats de nos deux premières contributions et a pour objectif de déterminer si l'usage de détecteurs de clones spécialisés dans une tâche peut aider les développeurs à gérer les clones présents dans leurs logiciels. Notre motivation pour cette étude est que la difficulté à trouver l'outil le plus adapté pour un utilisateur et une tâche de maintenance logicielle n'est peut-être pas uniquement due aux faiblesses des techniques d'évaluation de détecteurs de clones. Les outils de détection de clones actuels pourraient être considérés comme trop généralistes car ils n'offrent pas des options de configuration permettant d'adapter leurs résultats aux attentes des utilisateurs.

Nous présentons dans ce chapitre une nouvelle approche pour l'identification et la suppression automatique de la duplication dans du code CSS. Notre approche se base sur l'analyse formelle de concepts pour l'extraction de mixins. Son implémentation actuelle génère automatiquement du code Sass à partir de code CSS existant et est librement accessible. De plus, nous proposons une méthodologie différente pour l'évaluation et la comparaison de détecteurs de clones qui est basée sur des études de cas. Les résultats des études de cas que nous avons menées montrent que notre approche pour la migration automatique de code CSS existant vers du Sass est appréciée des développeurs. Enfin, notre approche est une extension des travaux de Mazinanian *et al.* [2014]. Ces derniers proposent une technique automatique pour éliminer la duplication au niveau des déclarations CSS. Notre approche va plus loin en supprimant en plus la duplication au niveau des propriétés. Elle est ainsi un atout supplémentaire pour faciliter la maintenance de code CSS.

Ces travaux ont fait l'objet d'une publication dans la conférence *International Conference on Software Maintenance and Evolution (2016)* [Charpentier *et al.*, 2016b].

6.2 Perspectives

Les travaux qui sont présentés dans cette thèse contribuent à l'usage des détecteurs de clones pour des tâches de maintenance logicielle. Différentes perspectives permettraient d'étendre ces travaux pour une utilisation encore plus avancée de la détection de clones dans les activités de maintenance logicielle.

Notre troisième contribution, détaillée dans le chapitre 5, propose l'usage de détecteurs de clones spécialisés dans une tâche pour une meilleure adaptation de leurs résultats aux besoins de chaque utilisateur. Cette adaptation est rendue possible par des options de configuration de l'outil de détection plus fines. L'utilisateur est alors en mesure de spécifier ses attentes pour que l'outil ne produise que des résultats pertinents pour lui. La spécialisation des détecteurs de clones est la réponse pour rendre ces outils utiles au quotidien pour les développeurs et mainteneurs logiciels. L'approche proposée lors de notre étude sur la suppression de la duplication dans du code CSS a des limites notamment liées à son aspect automatique. Une approche uniquement automatique peut conduire à la production de résultats non pertinents pour l'utilisateur. Une approche semi-automatique devrait donc être envisagée pour prendre en considération encore davantage les attentes de chaque utilisateur. Une problématique liée à cette nouvelle approche est de trouver un bon compromis entre production automatique et intervention humaine afin que les résultats produits restent intéressants au regard du temps consacré à les obtenir. Deux axes peuvent alors être considérés pour contribuer à cette problématique de recherche. Premièrement, déterminer si pour une tâche donnée il existe des types [1] de clone qui doivent nécessairement être détectés par un outil et d'autres au contraire qui ne le doivent pas. L'identification de ces types de clone permettrait de restreindre le nombre de clones à contrôler par chaque utilisateur et ainsi augmenter le gain de productivité des outils. L'objectif de ce premier axe de recherche est de déterminer s'il existe des constantes entre tâche de maintenance logicielle et type de duplication de code. Deuxièmement, utiliser l'apprentissage automatique pour ne faire valider aux utilisateurs que les clones les plus pertinents pour eux. Comme les détecteurs de clones ont pour vocation à être utilisés à de nombreuses reprises au cours de la maintenance logicielle, les exécutions précédentes devraient pouvoir améliorer les suivantes. Le recours à l'apprentissage automatique pour aider à la classification de clones a déjà été utilisé par Yang *et al.* [2014]. L'objectif de ce second axe de recherche est d'ancrer la détection de clones dans le processus de maintenance logicielle en améliorant au fil du temps son usage et en le personnalisant pour chaque utilisateur.

Pour les deux axes de recherche décrits ci-dessus, l'évaluation des détecteurs de clones est nécessaire pour valider les résultats. Nous avons proposé dans notre troisième contribution de recourir à des études de cas pour évaluer les détecteurs de clones spécifiques à une tâche. Celles-ci permettent en effet de contrôler que les résultats d'un outil donné cor-

1. Le terme type est utilisé au sens large ; il ne fait pas référence aux quatre types de clone définis dans le chapitre 2.

respondent effectivement aux attentes de chaque utilisateur. Toutefois, elles ne sont pas adaptées pour la comparaison d'outils. Or, cette comparaison est nécessaire pour choisir entre plusieurs outils dédiés à une même tâche. Une solution pour mener ces comparaisons serait de créer des *benchmarks* de clones à partir des études de cas. En effet, lors d'une étude de cas, chaque participant classe un ensemble de clones en vrais ou faux positifs qui peuvent servir de base à la création de *benchmarks*. Ainsi, pour toutes les études de cas relatives à des outils spécifiques à une même tâche, les ensembles de vrais et faux positifs identifiés comme tels par chaque participant peuvent être agrégés pour construire des corpus de référence fiables spécifiques à une tâche. Toutefois, comme la pertinence d'un clone dépend à la fois de la tâche et de l'utilisateur, les ensembles de vrais et faux positifs vont diverger d'un participant à l'autre. Des règles sont alors nécessaires pour déterminer si un clone est un vrai ou faux positif et de fait s'il doit être inclus ou non dans un corpus de référence. Par exemple, un clone peut être considéré comme un vrai positif si et seulement si au moins la majorité des participants l'ont jugé comme tel. D'autres critères sont évidemment envisageables et donnent la possibilité de spécialiser la comparaison de détecteurs de clones dédiés à une tâche. La pratique des développeurs pourrait être un autre critère : pour une tâche donnée, un développeur choisit un outil car un autre développeur avec les mêmes pratiques pour cette tâche précise l'a jugé pertinent. Finalement, la principale contrainte de cette approche est de disposer d'un nombre suffisant d'études de cas pour chacun des outils à comparer.

Bibliographie

Baker, B. S. (1992). A program for identifying duplicated code. *Computing Science and Statistics : Proceedings of the 24th Symposium on the Interface*, 24(Mar):49–57. Cité page 22.

Baker, B. S. (1995). On finding duplication and near-duplication in large software systems. *In Proceedings of the Second Working Conference on Reverse Engineering*, WCRE '95, pages 86–95, Washington, DC, USA. IEEE Computer Society. Cité pages 2, 10, 11, 19, et 40.

Balazinska, M., Merlo, E., Dagenais, M., Lague, B. et Kontogiannis, K. (1999). Measuring clone based reengineering opportunities. *In Software Metrics Symposium, 1999. Proceedings. Sixth International*, pages 292–303. Cité page 10.

Barbut, M. et Monjardet, B. (1970). *Ordre et classification algèbre et combinatoire*. Hachette. Cité pages 85 et 87.

Basit, H., Rajapakse, D. C. et Jarzabek, S. (2005). Beyond templates : a study of clones in the STL and some general implications. *In 27th International Conference on Software Engineering, 2005. ICSE 2005. Proceedings*, pages 451–459. Cité page 2.

Baxter, I., Yahin, A., Moura, L., Sant'Anna, M. et Bier, L. (1998). Clone detection using abstract syntax trees. *In , International Conference on Software Maintenance, 1998. Proceedings*, pages 368–377. Cité pages 2, 10, 20, et 22.

Bellon, S. (2002). Vergleich von techniken zur erkennung duplizierten quellcodes. Master's thesis, Institut fur Softwaretechnologie, Universitat Stuttgart, Stuttgart, Germany. Cité pages 24 et 28.

Bellon, S., Koschke, R., Antoniol, G., Krinke, J. et Merlo, E. (2007). Comparison and Evaluation of Clone Detection Tools. *IEEE TSE*, 33(9):577–591. Cité pages 3, 4, 10, 11, 24, 25, 28, 33, 36, 41, 43, 58, 60, 73, 103, et 104.

Berry, A., Huchard, M., Napoli, A. et Sigayret, A. (2012). Hermes : an efficient algorithm for building Galois sub-hierarchies. *In CLA'2012 : 9th International Conference on Concept Lattices and Applications*, pages 21–32. Universidad de Malaga. Cité page 87.

Birkhoff, G. (1948). *Lattice theory*, volume 25. American Mathematical Society New York. Cité page 85.

Bissyande, T. F., Thung, F., Wang, S., Lo, D., Jiang, L. et Reveillere, L. (2013). Empirical Evaluation of Bug Linking. *In Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, CSMR '13, pages 89–98, Washington, DC, USA. IEEE Computer Society. Cité page 75.

Burd, E. et Bailey, J. (2002). Evaluating clone detection tools for use during preventative maintenance. *In Second IEEE International Workshop on Source Code Analysis and Manipulation, 2002. Proceedings*, pages 36–43. Cité pages 10, 22, et 23.

Chanchal K. Roy et James R. Cordy (2009). A Mutation / Injection-based Automatic Framework for Evaluating Code Clone Detection Tools. *In Proceedings of the ICST 4th International Workshop on Mutation Analysis (Mutation 2009)*, pages 157 – 166. IEEE Press. Cité pages 25 et 28.

Charpentier, A., Falleri, J.-R., Lo, D. et Réveillère, L. (2015). An Empirical Assessment of Bellon's Clone Benchmark. *In Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, EASE '15, pages 20 :1–20 :10, Nanjing, China. ACM. Cité pages 4, 75, et 104.

Charpentier, A., Falleri, J.-R., Morandat, F., Yahia, E. B. H. et Réveillère, L. (2016a). Raters' reliability in clone benchmarks construction. *Empirical Software Engineering*, pages 1–24. Cité pages 5 et 105.

Charpentier, A., Falleri, J.-R. et Réveillère, L. (2016b). Automated Extraction of Mixins in Cascading Style Sheets. *In Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution*, Raleigh, North Carolina, USA. Cité pages 6 et 105.

Choi, E., Yoshida, N., Ishio, T., Inoue, K. et Sano, T. (2011). Extracting Code Clones for Refactoring Using Combinations of Clone Metrics. *In Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 7–13, New York, NY, USA. ACM. Cité page 31.

Cliff, N. (1996). *Ordinal methods for behavioral data analysis.* Psychology Press. Cité page 52.

Cordy, J. (2003). Comprehending reality - practical barriers to industrial adoption of software maintenance automation. *In 11th IEEE International Workshop on Program Comprehension, 2003*, pages 196–205. Cité page 2.

Cordy, J. R. et Roy, C. K. (2011). The NiCad Clone Detector. *In Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 219–220, Washington, DC, USA. IEEE Computer Society. Cité pages 21, 22, et 32.

Ducasse, S., Nierstrasz, O. et Rieger, M. (2006). On the Effectiveness of Clone Detection by String Matching : Research Articles. *J. Softw. Maint. Evol.*, 18(1):37–58. Cité pages 4, 24, 36, et 40.

Ducasse, S., Rieger, M. et Demeyer, S. (1999). A language independent approach for detecting duplicated code. *In IEEE International Conference on Software Maintenance, 1999. (ICSM '99) Proceedings*, pages 109–118. Cité pages 19 et 22.

Efron, B. (1979). Bootstrap Methods : Another Look at the Jackknife. *The Annals of Statistics*, 7(1):1–26. Cité page 45.

Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M. et Monperrus, M. (2014). Fine-grained and Accurate Source Code Differencing. *In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA. ACM. Cité page 59.

Gabel, M., Jiang, L. et Su, Z. (2008). Scalable detection of semantic clones. *In ACM/IEEE 30th International Conference on Software Engineering, 2008. ICSE '08*, pages 321–330. Cité page 21.

Ganter, B. et Wille, R. (1997). *Formal Concept Analysis : Mathematical Foundations.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st édition. Cité page 85.

Girard, J. F., Koschke, R. et Schied, G. (1997). A metric-based approach to detect abstract data types and state encapsulations. *In Automated Software Engineering, 1997. Proceedings., 12th IEEE International Conference*, pages 82–89. Cité page 24.

Göde, N. (2010). Clone Removal : Fact or Fiction ? *In Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 33–40, New York, NY, USA. ACM. Cité page 32.

Göde, N. et Koschke, R. (2009). Incremental Clone Detection. *In Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, CSMR '09, pages 219–228, Washington, DC, USA. IEEE Computer Society. Cité pages 4, 22, 32, et 60.

Han, J., Pei, J. et Yin, Y. (2000). Mining Frequent Patterns Without Candidate Generation. *In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 1–12, New York, NY, USA. ACM. Cité page 84.

Higo, Y., Kamiya, T., Kusumoto, S. et Inoue, K. (2004a). Refactoring Support Based on Code Clone Analysis. *In Kansai Science City*, pages 220–233. Springer. Cité page 31.

Higo, Y., Kamiya, T., Kusumoto, S., Inoue, K. et Words, K. (2004b). Aries : Refactoring support environment based on code clone analysis. *In In The 8th IASTED International Conference on Software Engineering and Applications(SEA 2004*, pages 222–229. ACTA Press. Cité page 31.

Ihaka, R. et Gentleman, R. (1996). R : A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314. Cité page 59.

Jiang, L., Misherghi, G., Su, Z. et Glondu, S. (2007). DECKARD : Scalable and Accurate Tree-Based Detection of Code Clones. *In 29th International Conference on Software Engineering, 2007. ICSE 2007*, pages 96–105. Cité pages 20 et 22.

Johnson, J. H. (1994). Substring Matching for Clone Detection and Change Tracking. *In Proceedings of the International Conference on Software Maintenance*, ICSM '94, pages 120–126, Washington, DC, USA. IEEE Computer Society. Cité page 11.

Juergens, E., Deissenboeck, F., Hummel, B. et Wagner, S. (2009). Do code clones matter? *In IEEE 31st International Conference on Software Engineering, 2009. ICSE 2009*, pages 485–495. Cité page 2.

Kalibera, T., Maj, P., Morandat, F. et Vitek, J. (2014). A Fast Abstract Syntax Tree Interpreter for R. *In Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 89–102, New York, NY, USA. ACM. Cité page 59.

Kamiya, T., Kusumoto, S. et Inoue, K. (2002). CCFinder : A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Softw. Eng.*, 28(7): 654–670. Cité pages 10, 19, 22, 29, 31, et 40.

Kapser, C., Anderson, P., Godfrey, M., Koschke, R., Rieger, M., van Rysselberghe, F. et Weißgerber, P. (2007). Subjectivity in Clone Judgment : Can We Ever Agree? *Dagstuhl Seminar Proceedings*. Cité page 29.

Kapser, C. et Godfrey, M. W. (2006). "Cloning considered harmful" considered harmful. *In Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 19–28. IEEE. Cité page 2.

Kim, H., Jung, Y., Kim, S. et Yi, K. (2011). MeCC : memory comparison-based clone detector. *In 2011 33rd International Conference on Software Engineering (ICSE)*, pages 301–310. Cité page 25.

Komondoor, R. et Horwitz, S. (2003). Effective, Automatic Procedure Extraction. *In Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, pages 33–, Washington, DC, USA. IEEE Computer Society. Cité page 10.

Koschke, R. et Eisenbarth, T. (2000). A framework for experimental evaluation of clustering techniques. *In 8th International Workshop on Program Comprehension, 2000. Proceedings. IWPC 2000*, pages 201–210. Cité pages 27 et 33.

Koschke, R., Falke, R. et Frenzel, P. (2006). Clone Detection Using Abstract Syntax Suffix Trees. *In 13th Working Conference on Reverse Engineering, 2006. WCRE '06*, pages 253–262. Cité pages 4, 21, 24, 36, et 41.

Krinke, J. (2001). Identifying similar code with program dependence graphs. *In Eighth Working Conference on Reverse Engineering, 2001. Proceedings*, pages 301–309. Cité pages 21 et 22.

Krutz, D. E. et Le, W. (2014). A Code Clone Oracle. *In Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 388–391, New York, NY, USA. ACM. Cité pages 24 et 28.

Krutz, D. E. et Shihab, E. (2013). CCCD : Concolic code clone detection. *In 2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 489–490. Cité page 25.

Landis, J. R. et Koch, G. G. (1977). The Measurement of Observer Agreement for Categorical Data. *Biometrics*, 33(1):159–174. Cité pages 27 et 68.

Leitão, A. M. (2004). Detection of Redundant Code Using R 2 D 2. *Software Quality Journal*, 12(4):361–382. Cité page 21.

Li, Z., Lu, S., Myagmar, S. et Zhou, Y. (2006). CP-Miner : finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192. Cité pages 2 et 10.

Livieri, S., Higo, Y., Matushita, M. et Inoue, K. (2007). Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder : D-CCFinder. *In Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 106–115. Cité page 22.

Lozano, A. et Wermelinger, M. (2008). Assessing the effect of clones on changeability. *In IEEE International Conference on Software Maintenance, 2008. ICSM 2008*, pages 227–236. Cité page 78.

Mayrand, J., Leblanc, C. et Merlo, E. (1996). Experiment on the automatic detection of function clones in a software system using metrics. *In , International Conference on Software Maintenance 1996, Proceedings*, pages 244–253. Cité pages 10, 20, et 22.

Mazinanian, D. et Tsantalis, N. (2016). An empirical study on the use of CSS preprocessors. *In Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, SANER 2016, Osaka, Japan. Cité page 94.

Mazinanian, D., Tsantalis, N. et Mesbah, A. (2014). Discovering Refactoring Opportunities in Cascading Style Sheets. *In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 496–506, New York, NY, USA. ACM. Cité pages 34, 78, 83, 95, 96, 102, et 105.

Mende, T., Koschke, R. et Beckwermert, F. (2009). An Evaluation of Code Similarity Identification for the Grow-and-prune Model. *J. Softw. Maint. Evol.*, 21(2):143–169. Cité page 29.

Mondal, M., Roy, C. K. et Schneider, K. A. (2014). Automatic Identification of Important Clones for Refactoring and Tracking. *In 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 11–20. Cité page 32.

Monden, A., Nakae, D., Kamiya, T., Sato, S. et Matsumoto, K.-i. (2002). Software quality analysis by code clones in industrial legacy software. *In Eighth IEEE Symposium on Software Metrics, 2002. Proceedings*, pages 87–94. Cité page 78.

Murakami, H., Higo, Y. et Kusumoto, S. (2014). A Dataset of Clone References with Gaps. *In Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 412–415, New York, NY, USA. ACM. Cité page 24.

Murakami, H., Hotta, K., Higo, Y., Igaki, H. et Kusumoto, S. (2012). Folding Repeated Instructions for Improving Token-Based Code Clone Detection. *In 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 64–73. Cité page 40.

Murakami, H., Hotta, K., Higo, Y., Igaki, H. et Kusumoto, S. (2013). Gapped code clone detection with lightweight source code analysis. *In 2013 IEEE 21st International Conference on Program Comprehension (ICPC)*, pages 93–102. Cité page 41.

Nguyen, H. A., Nguyen, T. T., Pham, N., Al-Kofahi, J. et Nguyen, T. (2012). Clone Management for Evolving Software. *IEEE Transactions on Software Engineering*, 38(5):1008–1026. Cité pages 4, 36, et 40.

Patenaude, J.-F., Merlo, E., Dagenais, M. et Lague, B. (1999). Extending software quality assessment techniques to Java systems. *In Seventh International Workshop on Program Comprehension, 1999. Proceedings*, pages 49–56. Cité page 2.

Ralph, P. et Kelly, P. (2014). The Dimensions of Software Engineering Success. *In Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 24–35, New York, NY, USA. ACM. Cité pages 2 et 103.

Romano, J., Kromrey, J., Coraggio, J. et Skowronek, J. (2006). Appropriate statistics for ordinal level data : Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys ? *In annual meeting of the Florida Association of Institutional Research*, pages 1–3. Cité page 52.

Roy, C. et Cordy, J. (2008). NICAD : Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. *In The 16th IEEE International Conference on Program Comprehension, 2008. ICPC 2008*, pages 172–181. Cité pages 4 et 25.

Roy, C., Zibran, M. et Koschke, R. (2014). The vision of software clone management : Past, present, and future (Keynote paper). *In 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 18–33. Cité page 3.

Roy, C. K., Cordy, J. R. et Koschke, R. (2009). Comparison and Evaluation of Code Clone Detection Techniques and Tools : A Qualitative Approach. *Sci. Comput. Program.*, 74(7): 470–495. Cité pages 3 et 103.

Rust, R. T. et Cooil, B. (1994). Reliability Measures for Qualitative Data : Theory and Implications. *Journal of Marketing Research*, 31(1):1–14. Cité page 27.

Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K. et Lopes, C. V. (2015). SourcererCC : Scaling Code Clone Detection to Big Code. *arXiv :1512.06448 [cs]*. Cité page 22.

Selim, G., Foo, K. et Zou, Y. (2010). Enhancing Source-Based Clone Detection Using Intermediate Representation. *In 2010 17th Working Conference on Reverse Engineering (WCRE)*, pages 227–236. Cité page 41.

Svajlenko, J., Islam, J. F., Keivanloo, I., Roy, C. K. et Mia, M. M. (2014). Towards a Big Data Curated Benchmark of Inter-project Code Clones. *In Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 476–480, Washington, DC, USA. IEEE Computer Society. Cité pages 25 et 28.

Svajlenko, J. et Roy, C. K. (2014). Evaluating Modern Clone Detection Tools. *In 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 321–330. IEEE Computer Society. Cité page 30.

T. Martinetz, K. S. (1991). A Neural-Gas Network Learns Topologies. *Artificial neural networks*, 1:397–402. Cité page 61.

Uddin, M. S., Roy, C. K. et Schneider, K. A. (2013). SimCad : An extensible and faster clone detection tool for large scale software systems. *In 2013 21st International Conference on Program Comprehension (ICPC)*, pages 236–238. Cité page 25.

Van Rysselberghe, F. et Demeyer, S. (2004). Evaluating clone detection techniques from a refactoring perspective. *In 19th International Conference on Automated Software Engineering, 2004. Proceedings*, pages 336–339. Cité page 30.

Walenstein, A., Jyoti, N., Li, J., Yang, Y. et Lakhotia, A. (2003). Problems creating task-relevant clone detection reference data. *In 10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings*, pages 285–294. Cité pages 27 et 75.

Wang, T., Harman, M., Jia, Y. et Krinke, J. (2013). Searching for Better Configurations : A Rigorous Approach to Clone Evaluation. *In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 455–465, New York, NY, USA. ACM. Cité pages 3, 4, 22, 24, 32, 36, 40, 60, 75, et 103.

Wang, W. et Godfrey, M. W. (2014). Recommending Clones for Refactoring Using Design, Context, and History. *In 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 331–340. Cité page 32.

Wettel, R. et Marinescu, R. (2005). Archeology of code duplication : recovering duplication chains from small duplication fragments. *In Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2005. SYNASC 2005*, pages 8 pp.–. Cité page 19.

Yang, J., Hotta, K., Higo, Y., Igaki, H. et Kusumoto, S. (2014). Classification model for code clones based on machine learning. *Empirical Software Engineering*, 20(4):1095–1125. Cité page 106.

Table des figures

3.1 Interface web pour la collecte d'opinion sur des clones. 44
3.2 Niveau de confiance des clones selon les opinions de Bellon et des deux participants de chaque groupe. 45
3.3 Distribution des clones pour la caractéristique *type*. 49
3.4 Distribution des clones pour la caractéristique *taille*. 50
3.5 Distribution des clones pour la caractéristique *langage*. 50
3.6 Niveau de confiance en fonction de la taille des clones. 51
3.7 Niveau de confiance en fonction du type et langage des clones. 51

4.1 Distributions de la taille et de la distance des clones dans FastR et GumTree. . 61
4.2 Distributions des clones dans les six groupes pour chaque projet et son échantillon associé. 62
4.3 Distributions dans chaque groupe des 65 doublons dans l'échantillon de GumTree. 63
4.4 Proportions des réponses *oui* et *non* données par l'expert sur les clones de chaque projet. 64
4.5 Proportions des réponses *oui*, *non* et *indéterminé* données par les experts, les juges externes et la majorité pour les clones de chaque projet. 65
4.6 Réponses des juges selon trois périodes de temps (*p1* : premier tiers des clones, *p2* : deuxième tiers et *p3* : dernier tiers). 66
4.7 Accord entre l'expert et la majorité pour les six caractéristiques étudiées. . . . 70

5.1 Treillis de concepts correspondant au contexte formel de la table 5.1. Les extensions et intensions simplifiées sont en gras. 88

5.2 Sous-hiérarchie de Galois obtenue à partir du treillis de concepts de la figure 5.1. 89
5.3 Filtrage de concepts sur la sous-hiérarchie de Galois de la figure 5.2. 90
5.4 Caractéristiques des 108 fichiers CSS utilisés dans notre expérience. 97
5.5 Temps d'exécution de notre outil sur les 108 fichiers CSS en fonction du nombre de règles (à gauche) et du nombre de déclarations (à droite). 98
5.6 Évolution du nombre de mixins générés en fonction de chacun des trois seuils : (a) *min-children*, (b) *min-declarations* et (c) *max-parameters*. 99

Liste des tableaux

2.1 Liste des principaux outils de détection de duplication de code. 22
2.2 Résultats de l'étude menée par Burd et Bailey. 24
2.3 Résumé des caractéristiques des quatre *benchmarks* de clones discutés. 28

3.1 Participants à la construction du *benchmark*. 37
3.2 Logiciels utilisés pour la construction du *benchmark*. 38
3.3 Nombre de clones proposés, examinés et conservés pour les huit logiciels. . . 39
3.4 Intervalle de confiance à 95% des proportions de clones avec un niveau de confiance *faible*, *moyen* et *élevé* dans le corpus de référence de Bellon. 46

4.1 Description des logiciels. 59
4.2 Réponses inconsistantes pour les 65 doublons dans GumTree. 67
4.3 Accord entre les juges calculé avec les tests du Kappa de Fleiss et de Cohen. . . 68
4.4 Interprétation du test du Kappa. 69
4.5 Nombre de réponses identiques et différentes entre l'expert et la majorité pour l'ensemble des clones de chaque projet. 70
4.6 Interprétation du V de Cramer. 71
4.7 Nombre de réponses identiques et différentes entre l'expert et la majorité. . . . 71
4.8 Nombre de réponses identiques et différentes entre l'expert et la majorité pour les vrais et faux positifs tels que jugés par l'expert. 72

5.1 Le contexte formel correspondant au code CSS du listing 5.9. 85
5.2 Options de l'outil *Mocss*. 95
5.3 Sujets sélectionnés. 96
5.4 Caractéristiques des sites web utilisés dans les études de cas. 100

5.5 Caractéristiques des mixins identifiés par la configuration par défaut ainsi que la meilleure pour chaque étude de cas. 100

Liste des listings

2.1	Exemple d'un clone de type-1.	12
2.2	Exemple d'un clone de type-2.	13
2.3	Exemple d'un clone de type-3.	14
2.4	Exemple d'un clone de type-4.	15
2.5	Exemple d'une paire de clones identiques détectés à l'aide d'une approche textuelle.	18
2.6	Normalisation basique d'un fragment de code.	18
3.1	Exemple d'un clone détecté dans *eclipse-ant* avec un niveau de confiance *élevé*.	46
3.2	Exemple d'un clone détecté dans *postgresql* avec un niveau de confiance *faible*.	47
5.1	Syntaxe d'une règle CSS.	80
5.2	Une balise `div` utilisant les règles de style de deux classes.	81
5.3	Un exemple de fichier CSS.	81
5.4	Un autre exemple de fichier CSS.	81
5.5	Duplication de propriété, valeur et déclaration en CSS.	82
5.6	Élimination de la duplication par regroupement des sélecteurs.	82
5.7	Un exemple de mixin en Sass qui factorise deux déclarations.	83
5.8	Le code CSS généré à partir du code Sass du listing 5.7.	84
5.9	Un exemple de code CSS.	86
5.10	Code Sass généré à partir de la sous-hiérarchie de Galois de la figure 5.3.	92
5.11	Extrait du code CSS compilé à partir du code Sass du listing 5.10.	93
5.12	Un exemple de fichier CSS.	94

Abstract

The existence of several copies of a same code fragment — called code clones in the literature — in a software can complicate its maintenance and evolution. Code duplication can lead to consistency problems, especially during bug fixes propagation. Code clone detection is therefore a major concern to maintain and improve software quality, which is an essential property for a software's success.

The general objective of this thesis is to contribute to the use of code clone detection in software maintenance tasks. We chose to focus our contributions on two research topics. Firstly, the methodology to compare and assess code clone detectors, i.e. clone benchmarks. We perform an empirical assessment of a clone benchmark and we found that results derived from this latter are not reliable. We also identified recommendations to construct more reliable clone benchmarks. Secondly, the adaptation of code clone detectors in software maintenance tasks. We developed a specialized approach in one language and one task — refactoring — allowing developers to identify and remove code duplication in their softwares. We conducted case studies with domain experts to evaluate our approach.

Keywords: *software maintenance, code duplication, empirical studies.*

Résumé

L'existence de plusieurs copies d'un même fragment de code (nommées des clones dans la littérature) dans un logiciel peut compliquer sa maintenance et son évolution. La duplication de code peut poser des problèmes de consistance, notamment lors de la propagation de correction de bogues. La détection de clones est par conséquent un enjeu important pour préserver et améliorer la qualité logicielle, propriété primordiale pour le succès d'un logiciel.

L'objectif général de cette thèse est de contribuer à l'usage des détecteurs de clones dans des tâches de maintenance logicielle. Nous avons centré nos contributions sur deux axes de recherche. Premièrement, la méthodologie pour comparer et évaluer les détecteurs de clones, i.e. les *benchmarks* de clones. Nous avons empiriquement évalué un *benchmark* de clones et avons montré que les résultats dérivés de ce dernier n'étaient pas fiables. Nous avons également identifié des recommandations pour fiabiliser la construction de *benchmarks* de clones. Deuxièmement, la spécialisation des détecteurs de clones dans des tâches de maintenance logicielle. Nous avons développé une approche spécialisée dans un langage et une tâche (la réingénierie) qui permet aux développeurs d'identifier et de supprimer la duplication de code de leurs logiciels. Nous avons mené des études de cas avec des experts du domaine pour évaluer notre approche.

Mots clefs : *maintenance logicielle, duplication de code, études empiriques.*

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