Les pratiques de code : de la documentation à la détection

Corentin Latappy

LABORATOIRE BORDELAIS DE RECHERCHE EN INFORMATIQUE LaBRI Packmind université de BORDEAUX

THÈSE

Présentée pour obtenir le grade de docteur délivré par

L'université de Bordeaux

École doctorale : Mathématiques et Informatique

Spécialité : Informatique

présentée et soutenue publiquement par

Corentin LATAPPY

le 19 Juin 2024

Les pratiques de code : de la documentation à la détection

Directeur de thèse : **Jean-Rémy FALLERI**

Membres du Jury

M. FALLERI, Jean-Rémy	Professeur des universités	Bordeaux INP	Directeur de thèse
Mme ETIEN, Anne	Professeure des universités	Université de Lille	Rapporteur
M. ACHER, Mathieu	Professeur des universités	INSA Rennes	Rapporteur
M. ZEMMARI, Akka	Professeur des universités	Université de Bordeaux	Président
M. ZIMMERMANN, Théo	Maître de Conférences	Télécom Paris	Examinateur

Membres invités

M. DEGUEULE, Thomas	Chargé de recherche	CNRS	Co-encadrant
M. TEYTON, Cédric	CTO	Packmind	Invité

Université de Bordeaux, LaBRI
351, cours de la Libération F-33405 Talence cedex
France

THESE

Titre

Les pratiques de code : de la documentation à la détection

Résumé

Les pratiques de code sont de plus en plus utilisées dans le domaine du développement logiciel. Leur mise en place permet d'assurer la maintenabilité, la lisibilité et la consistance du code, ce qui contribue fortement à la qualité logicielle. La majorité de ces pratiques est implémentée dans des outils d'analyse statique, ou linters, qui permettent d'alerter automatiquement les développeurs lorsqu'une pratique n'est pas respectée.

Toutefois, de plus en plus d'organisations, ayant tendance à créer leurs propres pratiques internes, rencontrent des problèmes sur leur compréhension et leur adoption par les développeurs. Premièrement, afin d'être appliquée, une pratique doit d'abord être comprise par les développeurs, impliquant donc d'avoir une documentation correctement rédigée. Or, ce sujet de la documentation n'a été que peu étudié dans la littérature scientifique. Ensuite, pour favoriser leur adoption, il faudrait pouvoir étendre les outils d'analyse existants pour y intégrer de nouvelles pratiques, ce qui est difficile compte tenu de l'expertise nécessaire pour apporter ces modifications. Packmind, société bordelaise, développe une solution pour accompagner les développeurs à faire émerger ces pratiques internes à l'aide d'ateliers. Cependant, elle souffre des mêmes problématiques citées précédemment.

Dans cette thèse, nous nous sommes d'abord intéressés à fournir des recommandations aux auteurs de la documentation des pratiques. Pour cela, nous avons analysé la documentation de plus de 100 règles provenant de 16 linters différents afin d'en extraire une taxonomie des objectifs de documentation et des types de contenu présents. Nous avons ensuite réalisé une enquête auprès de développeurs afin d'évaluer leurs attentes en termes de documentation. Cela nous a notamment permis d'observer que les raisons pour lesquelles une pratique doit être appliquée étaient très peu documentées, alors qu'elles sont perçues comme essentielles par les développeurs. Dans un second temps, nous avons étudié la faisabilité de l'identification automatique de violations de pratiques à partir d'exemples. Notre contexte, nous contraignant à détecter des pratiques internes pour lesquelles nous avons peu d'exemples pour apprendre, nous a poussé à mettre en place du transfert d'apprentissage sur le modèle de machine learning CodeBERT. Nous montrons que les modèles ainsi entraînés obtiennent de bonnes performances dans un contexte expérimental, mais que la précision s'écroule lorsque nous les appliquons à des bases de code réelles.

Mots clés

— qualité logicielle
— software craftsmanship
— pratiques de code
— outils d'analyse statique

Title

Coding practices : from documentation to detection

Abstract

Coding practices are increasingly used in the field of software development. Their implementation ensures maintainability, readability, and consistency of the code, which greatly contributes to software quality. Most of these practices are implemented in static analysis tools, or linters, which automatically alert developers when a practice is not followed.

However, more and more organizations, tending to create their own internal practices, encounter problems with their understanding and adoption by developers. First, for a practice to be applied, it must first be understood by developers, thus requiring properly written documentation. Yet, this topic of documentation has been little studied in the scientific literature. Then, to promote their adoption, it would be necessary to be able to extend existing analysis tools to integrate new practices, which is difficult given the expertise required to make these modifications. Packmind, a company based in Bordeaux, develops a solution to support developers in bringing out these internal practices through workshops. However, it suffers from the same issues mentioned above.

In this thesis, we first focused on providing recommendations to the authors of practice documentation. To do this, we analyzed the documentation of more than 100 rules from 16 different linters to extract a taxonomy of documentation objectives and types of content present. We then conducted a survey among developers to assess their expectations in terms of documentation. This notably allowed us to observe that the reasons why a practice should be applied were very poorly documented, while they are perceived as essential by developers. Secondly, we studied the feasibility of automatically identifying violations of practices from examples. Our context, forcing us to detect internal practices for which we have few examples to learn from, pushed us to implement transfer learning on the machine learning model CodeBERT. We show that the models thus trained achieve good performance in an experimental context, but that accuracy collapses when we apply them to real code bases.

Keywords

— software quality
— software craftsmanship
— coding practices
— static analysis tools

Remerciements

Salut à toi ô Promyze Je tiens tout d'abord à te remercier, ou devrais-je dire Packmind maintenant! En effet depuis ce stage en 2020, beaucoup de choses ont évolué. Ton nom en fait partie et ce n'est pas prêt de s'arrêter. Je tiens particulièrement à vous remercier, Cédric et Arthur, de m'avoir accordé votre confiance dès le début. Cela m'a permis de m'épanouir pleinement et de prendre plaisir à travailler. Je vous remercie également pour le cadre de ces 3 dernières années dans lequel j'ai pu aisément naviguer entre Packmind et le LaBRI. Merci à tous les collègues avec qui j'ai pu travailler pendant ces 4 années. Je vous souhaite à tous une bonne continuation, et que Packmind puisse atteindre sa North Star.

Salut à vous Jean-Rémy et Thomas Je vous remercie de m'avoir fait découvrir ce monde, complètement inconnu à l'origine, de la recherche. Merci d'avoir pris le temps de m'inculquer cette rigueur scientifique. Merci pour ces quelques réunions où je suis sorti en ayant compris le quart des choses, mais pendant lesquelles j'étais content d'entendre des esprits brillants échanger. Merci pour votre totale disponibilité et votre présence pour m'accompagner dans mes différents travaux. Je mesure la chance d'avoir réalisé cette thèse avec vous en tant qu'encadrants. D'ailleurs, je poursuis, à la surprise générale, pour au moins une année supplémentaire (Merci Thomas pour cette confiance), c'est donc que votre rôle de transmission a été efficace. Je pense, au passage, que vous formez un duo parfaitement équilibré et que vous vous complétez à merveille. Et enfin, comme dirait l'autre, je résumerai ces 3 années en : "C'était pas pire!".

Salut à toi Xavier Je te remercie d'être l'élément catalyseur de ces quatre dernières années. Merci de m'avoir fait découvrir le Génie Logiciel. Merci d'avoir fait intervenir Cédric dans le cadre d'un cours. Merci pour l'opportunité de réaliser une thèse. Et enfin, merci pour ton aide sur les différentes publications.

Salut à vous Papa et Maman Je vous remercie pour votre éducation rigoureuse qui a, sans aucun doute, contribué à la réussite de ce travail et de tant d'autres. Merci pour votre confiance permanente dans tout ce que j'ai pu entreprendre. Merci d'être là pour la Tribu. Enfin, merci de m'avoir vivement "conseillé" de continuer mes études à l'issue de mon DUT. Moi qui comptais m'arrêter après 2 ans d'étude, je me retrouve maintenant 8 années plus tard avec un doctorat.

Salut à vous Papou et Mamou Je vous remercie pour votre soutien de tous les instants. Merci pour les billets que chacun d'entre vous me glissait discrètement dans le creux de la main et que je devais cacher à l'autre. Et merci de m'avoir empêché de mourir de faim en remplissant mes valises quand je prenais la direction du LMA. Enfin, merci Mamou de te battre pour nous malgré la dure absence de Papou.

Salut à toi le Labo Je te remercie de m'avoir ouvert tes portes, alors qu'il y a encore 4 ans je te contemplais de loin pendant que mes différents professeurs y rentraient. Merci à tous les membres de l'équipe Progress dans laquelle j'ai été rapidement intégré, et en particulier à deux personnes. Merci Romain pour ton aide précieuse et ton analyse sur notre publication commune. Merci Christophe d'avoir pu partager nos galères et réussites ensemble, et je suis très content que l'on vienne à bout de nos thèses au même moment. J'espère que notre futur papier en commun aura plus de succès que nos tentatives de monter le Mont-Dore. Merci ensuite à l'IMT Mines Alès pour cette superbe collaboration. Merci à l'AFoDIB pour ces quelques sorties autour d'un jeu ou d'une bière, c'est important ce que vous faîtes. Enfin, merci à tous les doctorants, post-doctorants, chercheurs et autres avec qui j'ai eu l'occasion de découvrir l'immensité de la recherche en informatique à travers nos échanges.

Salut à toi l'Ovalie Je te remercie de m'avoir amené à l'arbitrage. J'ai découvert une seconde famille avec des gens sur lesquels je peux compter. Merci de me faire vivre des expériences de fou et d'avoir l'occasion de saisir de belles opportunités. Merci pour la confiance acquise afin d'être capable de me tenir devant une assemblée de personnes et d'animer des présentations. Enfin, merci à tous mes potes pour tous ces bons moments que l'on partage ensemble.

Salut à toi le Métal Je te remercie d'avoir tous ces merveilleux artistes et d'avoir rythmé, souvent de manière forte, mes journées de travail. Merci également d'être à l'origine d'un groupe de potes qui se retrouve régulièrement autour de tes hymnes.

Salut à toi Blandine Je crois qu'on dit qu'on garde toujours le meilleur pour la fin, ou la meilleure dans ce cas. Je te remercie pour ces, bientôt, 5 dernières années. Merci d'avoir été là dans ces moments importants. Merci de me supporter comme je suis, tout en m'ayant rendu plus tolérant et ouvert (un poil plus) à l'imprévu. Merci pour ta précieuse relecture et correction de ce long manuscrit. On ne sait pas de quoi sera fait notre avenir, mais je suis fier de t'avoir à mes côtés et de nous avoir vu évoluer ensemble dans nos domaines respectifs et avec réussite.

Table des matières

Chapitre 1

Introduction

Ce premier chapitre permet de présenter le contexte et les objectifs de cette thèse CIFRE en collaboration avec la société Packmind.

Sommaire

1.1 Contexte

La qualité logicielle fait maintenant partie intégrante du processus de création d'une solution logicielle. Le but principal de la qualité logicielle est d'assurer que les produits logiciels répondent aux attentes des utilisateurs, tout en étant fiables et maintenables. En 1991, dans le but d'uniformiser ces différents enjeux, la qualité logicielle a été normalisée sous la norme ISO/IEC 9126 [1], avant d'être révisée en 2011 à travers la norme ISO/IEC 25010. [2] Ces normes ont introduit six puis huit indicateurs à respecter : la capacité fonctionnelle, la facilité d'utilisation, la fiabilité, la performance, la maintenabilité et la portabilité, puis la sécurité et la compatibilité. Ces multiples dimensions de la qualité logicielle représentent un objectif majeur de tout projet de développement logiciel. Pour atteindre ces standards élevés de qualité, il est impératif de s'attarder sur ce qui constitue avant tout un logiciel : le code. En effet, la manière dont le code est conçu, écrit, testé, et maintenu joue un rôle crucial dans la réalisation des objectifs de qualité. Chaque ligne de code, chaque décision architecturale et chaque test effectué contribuent de manière significative à la stabilité, la performance et la sécurité du produit final. C'est le résultat de ce mélange, alliant connaissances théoriques, expériences pratiques et évolution constante des technologies, qui a mené à l'émergence des pratiques de code.

Les pratiques de code sont ainsi un pan majeur de la qualité logicielle. Elles prennent différentes formes et peuvent porter sur divers aspects inhérents à l'écriture du code. Cela peut être simplement des conventions visant à maintenir la cohérence du style du code à travers un projet [1], tout comme la gestion des dépendances externes pour éviter les conflits [2], ou bien encore la mise en place de tests automatisés garantissant que le code fonctionne comme attendu [3]. Elles influent directement sur les aspects principaux de

1. https://www.iso.org/fr/standard/16722.html
2. https://www.iso.org/fr/standard/78176.html

la qualité logicielle, tels que la maintenabilité, la performance et la sécurité du logiciel. Adopter et maintenir de bonnes pratiques de code est donc un investissement stratégique pour la qualité globale du logiciel.

À l'origine, ces bonnes pratiques de développement étaient présentes dans les livres techniques ou les articles scientifiques, mais elles sont maintenant plus largement accessibles en ligne, à travers les blogs et les forums. "Clean Code" de Robert C. Martin [4] regroupe un ensemble de pratiques sur la lisibilité et la maintenabilité du code. Il tient d'ailleurs un blog sur ce même thème. [3] Il existe des ouvrages orientés sur le domaine de l'architecture logicielle, comme "Design Patterns" de Erich Gamma *et al.* [5] ou sur le DDD d'Eric Evans [6]. Des sujets comme la sécurité [7, 8] ou la performance [9] sont moins couverts, car souvent plus liés spécifiquement à un langage ou à une utilisation spécifique. Nous observons la présence et l'importance de ces pratiques dans l'industrie du développement logiciel, et ce même chez les géants de cette industrie. Nous pouvons citer le travail de Winters *et al.* [10] qui décrit tout le processus autour du code mis en place par Google de par sa longue expérience. Google met même à disposition de manière publique ses conventions d'écriture pour le code. [4]

Les concepteurs de langage de programmation et de bibliothèques participent également à l'émergence de ces pratiques [11]. Ces concepteurs sont avant tout des développeurs qui mettent eux aussi en place un ensemble de pratiques internes pendant la phase de développement. Lorsque leur projet est utilisé par les utilisateurs finaux, ils souhaitent généralement que ces conventions soient également adoptées. Ils les mettent donc à disposition dans leur documentation officielle. Par exemple, *Angular*[5], framework populaire permettant de réaliser des applications web, partage un guide sur le style à adopter pour développer une application en Angular. [6] Le fait d'appliquer ces conventions facilite également l'arrivée d'un nouveau développeur dans un projet existant. Si celui-ci possède déjà une expérience dans le langage ou une bibliothèque utilisés dans le projet, son intégration sera plus facile si ce sont les conventions classiques qui avaient été adoptées. Les équipes de développement font donc régulièrement le choix de mettre en place ces conventions pour des raisons de cohérence et de maintenabilité.

Les outils destinés aux développeurs sont également une source importante de pratiques. Nous y retrouvons les environnements de développement intégré (IDE en anglais), mais surtout les outils d'analyse statique (aussi connus sous l'appellation de *linters*). Ces outils, de plus en plus utilisés [12], fournissent un ensemble de pratiques, appelées dans ce contexte *règles*, et détectent automatiquement les fragments de code qui ne les respectent pas, facilitant ainsi leur adoption par les développeurs. Ils permettent également de couvrir plusieurs aspects de la qualité et ce, dans de multiples langages de programmation. *Brakeman*[7] alerte en cas de présence de vulnérabilités dans des applications Ruby on Rails, tandis qu'un linter comme *ESLint*[8] possède des règles pour JavaScript sur un spectre beaucoup plus large : sécurité, performance, mauvaise utilisation du langage ou encore stylistique.

Enfin, de plus en plus de pratiques émergent au sein des équipes de développement [13], appelées pratiques *internes*. Ce sont des pratiques directement créées par les équipes de développement. Elles correspondent ainsi réellement à la façon interne de développer et prennent dès lors en compte le contexte unique de chaque projet. Chez Packmind,

3. https://blog.cleancoder.com
4. https://google.github.io/styleguide
5. https://angular.io
6. https://angular.io/guide/styleguide
7. https://brakemanscanner.org
8. https://eslint.org

par exemple, nous avons une pratique intitulée "Always use PMModal when creating modals", qui documente la façon d'utiliser un composant interne qui affiche une boîte de dialogue personnalisée. Les équipes apprennent souvent de leurs erreurs passées, et l'analyse de ces erreurs peut conduire à l'adoption de nouvelles pratiques pour éviter de les répéter. Ces pratiques internes vont également être amenées à évoluer en même temps que les développeurs et que le projet gagne en maturité. La difficulté majeure de ces pratiques est leur mise en commun et leur adoption par les développeurs d'une même équipe. Elles sont régulièrement documentées au sein de wikis [14] qui souffrent du manque de mise à jour [15], mais elles manquent surtout d'outils, à l'image des linters, permettant d'assurer de manière automatique qu'elles sont bien appliquées par les développeurs.

Packmind[9] (ex-Promyze), entreprise dans laquelle j'effectue ma thèse CIFRE, est une société bordelaise créée en 2016 sur la base des travaux de thèse de Cédric Teyton au sein du LaBRI. Depuis sa création, l'objectif est d'aider les différents acteurs du développement logiciel (managers, développeurs, CTOs, ...) à mieux appréhender la qualité de leur code. Nous sommes maintenant une équipe de 9 personnes qui travaillons sur un projet commencé il y a 4 ans, les Ateliers Craft. Le but de ces ateliers est de créer un temps d'échange entre les développeurs d'une entreprise afin de mettre en commun leurs pratiques de code. Le contenu de ces ateliers est alimenté par les développeurs eux-mêmes pendant leur processus de développement. À l'aide des plugins directement présents dans leurs outils, comme dans les IDE ou les Systèmes de Contrôle de Versions (VCS), ils peuvent remonter des pratiques sur du code existant sur lequel ils souhaitent discuter. Ces suggestions de pratique peuvent être orientées de manière positive ou négative; cela peut être une pratique à généraliser ou au contraire à éviter ou à améliorer. Ensuite, lors des ateliers craft, les acteurs du code se réunissent afin de parcourir les pratiques remontées. L'idée est de passer en revue chaque pratique et de débattre de son adoption. Au fur et à mesure de ces ateliers, les pratiques validées contribuent à alimenter la base de connaissances internes. Nous obtenons ainsi une documentation vivante et fortement liée au code interne sur la façon de développer. L'intention principale de Packmind est donc de diffuser et d'homogénéiser les pratiques internes afin d'améliorer de manière globale la qualité logicielle au sein d'un projet.

1.2 Problématiques et Objectifs

Avec le recul et les différentes expériences des utilisateurs, notre solution comporte un certain nombre de problématiques liées à un manque d'opérationnalisation.

Améliorer la qualité de la documentation des pratiques La problématique principale concerne la bonne transmission de la connaissance à un développeur qui découvre une nouvelle pratique. Avec Packmind, cette connaissance est transmise en théorie pendant le déroulement des Ateliers Craft. Cependant, si nous considérons le cas de nouveaux arrivants dans une équipe, n'ayant pas pu participer aux précédents ateliers, ceux-ci n'auront pas d'autre choix que d'aller consulter la documentation des pratiques déjà créées s'ils veulent être informés de manière autonome. Nous rencontrons le même cas lorsqu'un développeur utilisant un linter dans son IDE est alerté qu'il a écrit du code ne respectant pas une pratique. Lorsqu'il est notifié d'une telle violation, le développeur souhaitera corriger son erreur pour obtenir un code conforme. S'il a déjà rencontré cette règle, cette

9. https://www.packmind.com

opération ne lui posera aucun problème; mais si ce n'est pas le cas, il va là aussi parcourir la documentation de la pratique associée. Cette documentation devient donc un point d'entrée capital pour apporter la connaissance aux développeurs, ainsi que pour promouvoir le respect de ces pratiques. Le premier défi réside dans le fait de documenter une pratique sans aucun contexte particulier, de manière à ce que les développeurs soient en mesure de les appliquer plus tard dans leur propre contexte. La subjectivité de certaines pratiques est un autre point délicat, comme par exemple le débat sur l'utilisation de l'instruction goto [16]. Ici, la documentation de la pratique joue un rôle important car elle apporte toutes les justifications afin de comprendre pourquoi et dans quels cas il est important d'adopter une telle pratique. La littérature scientifique de ce domaine ne s'est intéressée qu'aux messages de notifications [17, 18, 19] et a montré que leur qualité n'était pas idéale. À notre connaissance, il n'existe pas encore d'étude réalisée sur la documentation des pratiques de code. Nous pensons que la qualité actuelle de rédaction est équivalente à la qualité du contenu affiché dans le message de notification. C'est la raison pour laquelle nous souhaitons fournir dans cette thèse des recommandations à destination des rédacteurs de documentation. Nous souhaitons d'abord établir un état des lieux sur la façon dont sont documentées les pratiques de code, puis ensuite évaluer les attentes des développeurs vis-à-vis de ce type de documentation.

Au delà du cas d'utilisation interne à Packmind, cela bénéficierait également à d'autres outils qui documentent du code. Les éditeurs de linters, qui exposent des règles à leurs utilisateurs, doivent les détailler à travers une documentation. L'outil Semgrep [10] offre la possibilité aux développeurs d'écrire leurs propres patterns de détection, puis de les partager publiquement. Cependant, il ne fournit aucune recommandation afin que ces patterns soient ensuite correctement compris par tous. À plus petite échelle, de nombreuses entreprises mettent également en place des solutions internes afin de documenter leurs propres pratiques. Répondre à cette question permettra donc de guider plusieurs acteurs, contribuant à la qualité logicielle, à documenter de manière plus efficiente leurs pratiques de code.

Apprendre automatiquement les pratiques internes La seconde problématique est le suivi de la mise en application des pratiques par les développeurs. En effet, les Ateliers Craft sont des outils idéaux pour faire émerger les pratiques, mais pas pour les faire appliquer à l'échelle d'un projet. Certes, les pratiques sont validées en accord pendant les ateliers, mais leur nombre grandissant fait qu'il devient difficile de toutes s'en souvenir. De plus, le turnover important dans le domaine de l'ingénierie logicielle implique un changement régulier de développeurs au sein d'une équipe à qui il faut inculquer ces pratiques. Pour cette étape d'onboarding, qui permet entre autres de présenter les processus de développement interne, la base de connaissances alimentée dans Packmind semble être une solution idéale. Cependant, dans des équipes qui réalisent des ateliers régulièrement et depuis longtemps, l'intégration de cette masse importante d'informations peut être compliquée.

À l'image d'un linter, nous souhaitons fournir un outil qui alertera un développeur lorsqu'une pratique ne sera pas respectée. Il ne devra nécessiter aucune intervention humaine lorsqu'une nouvelle pratique est ajoutée à la base de connaissances, et idéalement être indépendant du langage de programmation. Un atout de Packmind est le fait de stocker les fragments de code lorsqu'une pratique est soumise. Nous imaginons donc une solution qui, basée sur ces exemples en nombre restreint, infère automatiquement un pattern qui permettra de détecter lorsqu'une pratique n'est pas correctement appliquée

10. https://semgrep.dev

par un développeur. Un tel outil directement intégré dans les IDE, et par la suite dans la chaîne d'intégration continue, permettrait ainsi de pouvoir réagir immédiatement lors de l'utilisation d'une mauvaise pratique de code. Avoir cette information aussi tôt permettra de corriger le problème directement à la source, et économisera du temps lors d'une code review ou lors d'une phase de correction.

À nouveau, cette problématique n'est pas unique à l'outil Packmind. En effet, les pratiques internes ne sont pas une nouveauté, elles existent déjà au sein des entreprises et sont disponibles à travers des documentations internes. Ainsi, pouvoir automatiquement identifier ces pratiques en se basant sur cette documentation interne, quelle qu'en soit sa forme, permettra de les diffuser de manière plus efficace au sein des équipes de développement.

1.3 Contributions

Pour répondre à la problématique sur la documentation d'une pratique de code, nous avons souhaité étudier ce qui est fait dans des outils existants et déjà utilisés par les développeurs. Comme détaillé précédemment, la majorité des pratiques proviennent des linters. Ces outils sont très largement utilisés par les développeurs pour diffuser et faire appliquer les pratiques de code. Ils sont présents au sein des chaînes d'intégration continue. Cependant, les linters représentent un atout majeur lorsqu'ils sont directement intégrés dans les IDE puisque la détection a lieu en simultané avec l'écriture du code. Lorsqu'une violation est détectée, le linter alerte le développeur à travers une notification. Pour ne pas surcharger l'environnement des développeurs, cette notification affiche le minimum d'informations, ce qui nécessite de consulter la version en ligne afin d'obtenir plus de détails. Nous pouvons donc utiliser ces documentations en ligne afin de réaliser un état des lieux sur leur composition. Nous avons ainsi analysé la documentation de plus de 100 règles de 16 linters pour 7 langages, et nous en avons extrait une taxonomie sur les objectifs (*Quoi*, *Pourquoi*, *Correction*) et les types de contenu (*Texte*, *Code*, *Lien*) dans une documentation. Nous avons ensuite confronté ces premiers résultats en menant une enquête auprès des développeurs. Cela nous a permis de mettre en avant des problèmes avec la documentation du *Pourquoi*, considéré essentiel par les développeurs, mais majoritairement absent ou de mauvaise qualité en pratique. Nous avons aussi trouvé que le *Code* et le *Texte* sont intéressants pour documenter le *Quoi* et la *Correction*. Ce travail a contribué, en plus de son package de réplication [20], à la publication suivante :

> C. Latappy, T. Degueule, J.-R. Falleri, R. Robbes, X. Blanc, C. Teyton, "What the Fix? A Study of ASATs Rule Documentation", 46th International Conference on Program Comprehension (ICPC), Lisbon, Portugal, 2024, doi : 10.1145/3643916.3644404

Pour accompagner notre publication, nous avons également créé un outil web [11] qui nous a permis de fournir plus de données, et notamment de consulter les résultats de l'enquête de manière interactive.

Concernant la seconde problématique, nous souhaitions répondre à la question de l'apprentissage automatique des pratiques de code à partir d'exemples soumis par les développeurs. Nous avons ainsi réalisé une étude de faisabilité à l'aide de CodeBERT, modèle de Machine Learning (ML) à l'état de l'art au moment de la contribution, pour tenter

11. https://icpc2024-asats.github.io

d'apprendre des pratiques et de les identifier par la suite. La spécificité de ce travail a été de se placer dans le contexte que nous avons chez Packmind. En effet, comme les pratiques sont identifiées manuellement par les développeurs pendant leur processus habituel de travail, elles ne contiennent que très peu d'exemples disponibles. Ce contexte, inhabituel pour le ML, nous a notamment permis d'étudier l'impact du nombre d'exemples sur la qualité de l'apprentissage, et de déterminer si l'apprentissage par transfert permettait de se contenter de très peu d'exemples. Pour procéder à cet apprentissage, nous avons généré un dataset [21] contenant quasiment 13 millions de violations pour 38 règles différentes du linter ESLint. Cela nous a permis de réaliser diverses expériences mixant les tailles d'apprentissage et les ratios entre du code conforme, du code non-conforme et du code aléatoire. Bien que les résultats dans le cadre expérimental soient prometteurs, nous obtenons des résultats de précision trop faible pour que notre solution soit utilisable dans un cas industriel. Cette étude a mené à cette seconde publication :

C. Latappy, Q. Perez, T. Degueule, J.-R. Falleri, C. Urtado, S. Vauttier, X. Blanc, C. Teyton, "MLinter : Learning Coding Practices from Examples—Dream or Reality?", 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Taipa, Macao, 2023, pp. 795-804, doi : 10.1109/SANER56733.2023.00092

1.4 Structure de la Thèse

Le contenu de la thèse est organisé comme suit :

Le Chapitre 2 explique toutes les notions essentielles sur les pratiques de code, et notamment sur leur partage. Le Chapitre 3 présente l'analyse effectuée sur la documentation des pratiques présente dans les outils d'analyse statique. Le Chapitre 4 détaille une étude effectuée pour apprendre à identifier une pratique en se basant sur des exemples de code. Enfin, le Chapitre 5 présente nos perspectives de recherche et la conclusion.

Chapitre 2

Diffusion et Adoption des Pratiques de Code

Ce chapitre a pour objectif de présenter les moyens actuels permettant la diffusion et l'adoption des pratiques de code au sein des équipes de développement, et d'en exposer les principales problématiques dans le but de motiver nos différents travaux de recherche.

Sommaire

Comme introduit dans la Section 1.1, l'objectif des pratiques de code est d'améliorer la qualité du logiciel et l'efficacité du processus de développement. En adoptant ces pratiques, les équipes s'assurent que leur code reste maintenable, performant, sécurisé et compréhensible pour les développeurs actuels et futurs. Cela permet ainsi de produire un logiciel qui soit non seulement fonctionnel mais aussi fiable, maintenable et évolutif. Cependant, pour que ces pratiques apportent réellement leurs bénéfices, elles doivent être utilisées par les développeurs, ce qui implique qu'elles soient correctement diffusées au sein des équipes, puis adoptées. Ces pratiques de code représentent la manière interne de développer. Il s'agit donc d'une connaissance essentielle pour une organisation qu'il faut partager.

La Gestion des Connaissances (*Knowledge Management*, en anglais) est devenue une composante cruciale de l'ingénierie logicielle, émergeant au milieu des années 1980 pour faire face au "déluge d'informations" [22]. Bjørnson *et al.* [23] la définissent comme "une méthode qui simplifie le processus de partage, de distribution, de création, de capture et de compréhension de la connaissance d'une entreprise". Selon Lindvall *et al.* [24], les organisations sont confrontées à la plus grosse perte de connaissances lorsque les employés partent. Cette perte représente non seulement un manque à gagner en termes d'expertise mais également un coût potentiellement élevé pour l'organisation qui doit former de nouveaux employés ou récupérer cette connaissance perdue. La capacité de

savoir qui détient quelles connaissances devient ainsi essentielle pour optimiser la collaboration et l'efficacité au sein de l'équipe. Elle contribue également à maintenir un niveau d'expertise constant malgré le turnover des employés, en assurant une transmission efficace des connaissances cruciales pour l'organisation. Rus *et al.* [22] mettent en lumière d'autres avantages de la gestion des connaissances, notamment la réduction des coûts et du temps de développement. En évitant de répéter les erreurs passées, les organisations peuvent économiser des ressources significatives. Par ailleurs, en disposant d'un accès facile à des informations précises et à jour, les développeurs peuvent prendre de meilleures décisions, ce qui se traduit par une amélioration de la qualité du logiciel produit.

En plus de mettre en place des pratiques de code, il est également essentiel que les organisations entretiennent une atmosphère propice au partage de cette connaissance. Il existe divers moyens afin de s'assurer de la diffusion de ces pratiques. Dans ce chapitre, nous nous concentrons sur 3 solutions : la revue de code, l'utilisation des linters, et Packmind.

2.1 Les Revues de Code

La revue de code est une pratique importante du génie logiciel dans laquelle le code écrit par un développeur est examiné par une ou plusieurs autres personnes [25]. L'objectif est de détecter de potentielles erreurs, de garantir la conformité aux normes de codage, et de s'assurer que le code est bien compris par les autres membres de l'équipe. Michael Fagan, ingénieur chez IBM dans les années 70, est à l'origine de cette pratique suite à la publication de son papier "Design and Code Inspections to Reduce Errors in Program Development" [26]. Il présente l'*inspection* qui est un processus structuré pour examiner le code et la conception, dans le but de détecter et d'éliminer les erreurs le plus tôt possible dans le cycle de développement. Cette approche s'est adaptée aux changements dans les méthodologies de développement logiciel ainsi qu'à l'évolution des technologies.

De nos jours, la revue de code, dite moderne, est associée à l'utilisation d'un gestionnaire de versions, comme *git*[1], d'une plate-forme de développement collaboratif, comme *GitHub*[2] ou *GitLab*[3] et des *Pull Requests* (PR). Bosu et Carver [27] ont observé que les développeurs pouvaient passer plus de six heures par semaine à réaliser des revues de code. Dans le processus de développement logiciel, l'utilisation d'un gestionnaire de versions facilite le maintien de la qualité du code déployé en production. Les modifications, qu'il s'agisse de corrections de bugs ou d'ajouts de nouvelles fonctionnalités, sont généralement effectuées sur des branches séparées de la branche principale, pour préserver la stabilité de la version en production. Ces branches permettent une isolation des changements, facilitant ainsi les tests et les validations avant leur intégration. La consolidation de ces modifications dans la branche principale s'effectue via une Pull Request, permettant une validation collaborative, cruciale pour maintenir l'intégrité et la qualité du code dans un environnement de production. En général, les projets possèdent un ensemble de bonnes pratiques à appliquer pour réaliser une PR. Par exemple, il est possible de consulter celles pour le projet *VSCode*[4]. Un exemple de PR ouverte pour ce projet est disponible sur la Figure 2.1a. Ici, son auteur décrit de manière succincte quel problème il a corrigé en le référençant, et ajoute également des captures d'écran sur le résultat final obtenu. Ensuite, une autre personne, le *reviewer*, va s'assurer que le code produit réalise bien ce qui

1. https://git-scm.com
2. https://github.com
3. https://about.gitlab.com
4. https://github.com/microsoft/vscode/wiki/How-to-Contribute#pull-requests

est attendu, n'introduit pas de bugs, et respecte les conventions de code du projet. Selon la taille des équipes disponibles, il est aussi possible que plusieurs personnes réalisent cette revue. Pour réaliser ces différentes tâches, les outils actuels permettent de parcourir facilement l'ensemble des modifications apportées dans le code (*cf.*, Figure 2.1b). Ils permettent également d'ajouter des commentaires sur le code modifié, et d'échanger facilement avec l'auteur et les autres reviewers. Une fois les éventuels retours intégrés par l'auteur de la PR, elle est acceptée et le nouveau code est ainsi inséré dans la branche cible.

2.1.1 Bénéfices

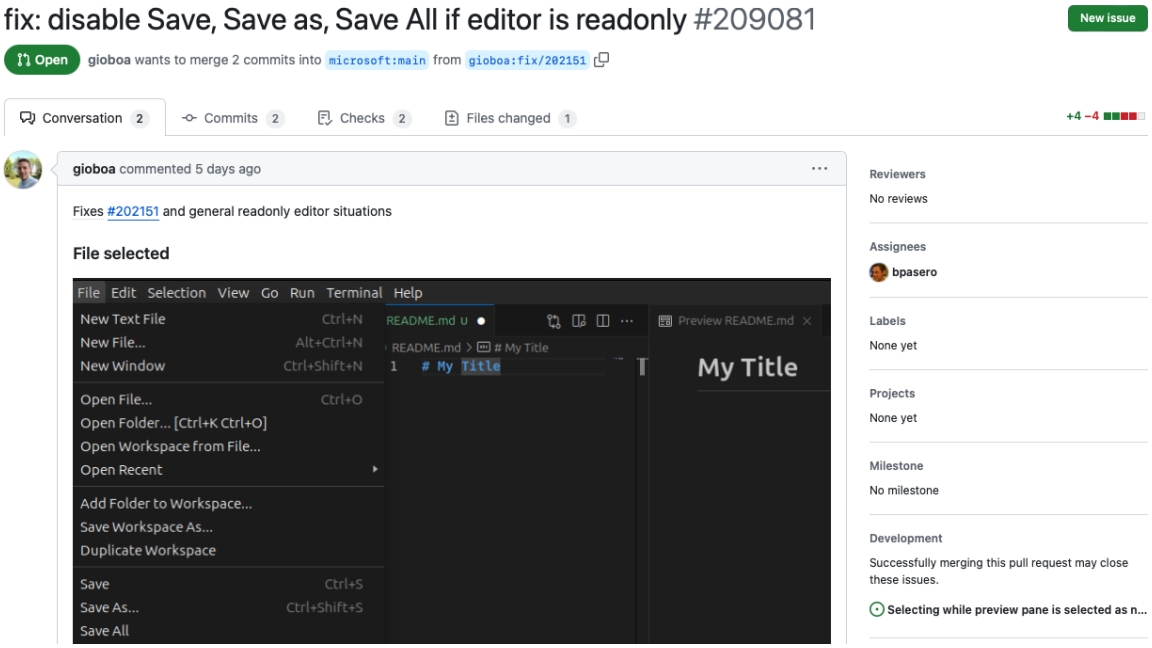
La mise en place des revues de code participe à l'amélioration de la qualité logicielle, et ce pour plusieurs raisons. Bacchelli et Bird [28] soulignent que la revue de code moderne est principalement motivée par la détection et la correction des bugs. Bien que cette étude soit réalisée au sein des équipes de développement de Microsoft, cette tendance est également observée par Rigby *et al.* [29] et Beller *et al.* [30] qui se concentrent sur les systèmes open source. McIntosh *et al.* [31] ont montré la corrélation entre la participation des développeurs aux revues de code et la réduction du nombre de bugs introduits dans la base de code. Cela souligne donc l'importance d'impliquer activement les développeurs dans ce processus. Bavota et Russo [32] ont confirmé cette efficacité de la revue de code dans la détection précoce des bugs. En effet, ils ont montré que le code non soumis à une revue était plus susceptible d'amener à une modification ultérieure que le code soumis à une revue. Cette dernière étude montre également que le code révisé pendant une revue présente une meilleure lisibilité. Ainsi, la revue de code contribue non seulement à la maintenabilité du code mais aussi à sa compréhension générale par les développeurs.

L'apprentissage par transfert représente un autre avantage crucial de la revue de code [29, 28]. Ont notamment été mis en lumière les bénéfices apportés par la formation de nouveaux développeurs. La revue permet également d'apporter de nouvelles approches dans la résolution d'un problème [28]. Des échanges, observés entre les auteurs et reviewers d'une PR, ont amené à la mise en place de meilleures implémentations en termes de lisibilité ou de performance. Morales *et al.* [33] ont aussi constaté les effets positifs de la revue de code en ce qui concerne la diminution de l'introduction d'anti-patterns. Ces anti-patterns sont décrits par Coplien et Harrison [34] comme *"something that looks like a good idea, but which back-fires badly when applied"*. Ils citent notamment l'exemple de la *God Class*, qui est l'utilisation d'une unique classe qui contient l'implémentation de tous les comportements internes d'une application, ce qui la rend lourde et complexe à maintenir.

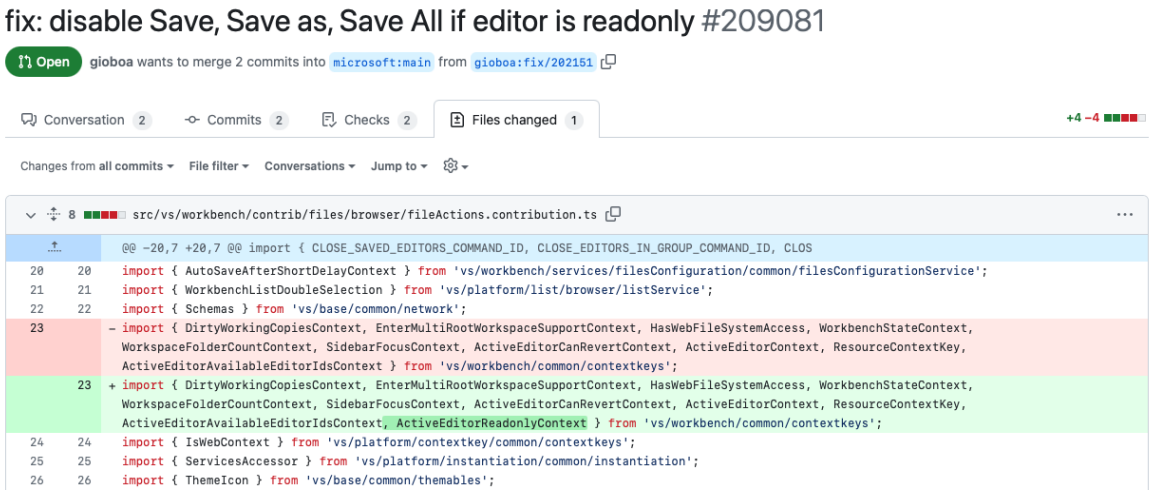
En somme, la revue de code se révèle être un vecteur essentiel pour améliorer la qualité globale du produit logiciel. Elle permet non seulement de réduire les bugs et d'améliorer la lisibilité et la maintenabilité du code, mais aussi de favoriser le partage de connaissances et l'innovation parmi les développeurs, contribuant ainsi à la construction d'un environnement de développement plus robuste et collaboratif.

2.1.2 Freins à l'Adoption

Un problème majeur rencontré lors des revues de code est la compréhension des changements effectués [28]. Ce challenge a un effet direct sur la qualité des commentaires de la revue, puisqu'il a été observé que les PR fournissant des informations sur le contexte avec des commentaires sur les modifications avaient des retours plus rapides tout en ap-



(a) Commentaires associés



(b) Changements réalisés

FIGURE 2.1 – Exemple de PR sur le projet *VSCode* dans l'interface de *GitHub*
Accessible à : https://github.com/microsoft/vscode/pull/209081

portant plus de valeur. D'ailleurs, si l'auteur d'un changement est le seul possédant l'ensemble de la connaissance nécessaire et qu'il ne la partage pas, il n'existe potentiellement aucune autre personne capable de faire la revue de manière efficace.

Ce manque de contexte mène parfois à un délai important pour obtenir le retour de la part du reviewer, ce qui est un second problème. Rigby et Bird [35] ont mesuré que le temps médian pour la complétion d'une revue était de l'ordre de la journée au sein d'entreprises comme Google et Microsoft. Ce temps est similaire dans le monde des logiciels open source [36]. Comme souligné par Czerwonka *et al.* [37] et Kononenko *et al.* [38], l'effort nécessaire pour une revue n'est pas seulement un coût en terme de temps, mais demande aussi aux développeurs de changer régulièrement de contexte entre leurs différentes tâches. Il est commun pour les développeurs de travailler sur plusieurs tâches, telles que la correction de bugs, l'intégration de nouvelles fonctionnalités ou la réalisation de revue de code. Ce changement régulier peut devenir complexe lorsque le contexte n'est pas toujours le même, allongeant le temps nécessaire pour faire une revue. Ainsi, plus le temps de retour de la part du reviewer est long, plus il sera dur pour l'auteur à l'origine de se remémorer tout le contexte et d'intégrer les retours du reviewer sans introduire de nouveaux problèmes. Ce temps important entre les échanges peut également impacter le reste de l'équipe de développement qui dépend de ce nouveau code pour poursuivre leur travail.

Un autre inconvénient majeur de la revue de code réside dans la superficialité des retours fournis par les reviewers [28]. En effet, certaines critiques soulignent que les reviewers ont tendance à se concentrer sur les erreurs faciles à identifier, telles que les fautes de formatage, au lieu de s'attaquer aux problèmes plus substantiels. Cette tendance est d'autant plus forte lorsque le code examiné n'appartient pas à la base de code habituelle du reviewer. Comme le révèlent les témoignages recueillis, il est parfois gênant pour un développeur de recevoir des commentaires qui ne soulignent que des erreurs de formatage mineures, laissant passer des erreurs plus critiques.

Rigby et Storey [39], qui se sont concentrés sur la revue dans l'open source et ont analysé les interactions entre les différents acteurs impliqués, ont notamment observé deux comportements notables : certaines revues sont repoussées tant que les changements apportés ne sont pas perçus comme essentiels par les développeurs principaux, et deuxièmement, ces développeurs principaux ont tendance à ignorer les revues qui peuvent amener à des discussions jugées improductives. Ces deux problématiques sont notamment présentes lorsqu'il n'y a aucune forme de contrôle sur qui peut soumettre une PR, et quand la révision des modifications effectuées est fortement dirigée par les besoins exprimés par les utilisateurs.

Pour répondre à ces obstacles, Tufano *et al.* [40] ont exploré, à travers des méthodes de Deep Learning, l'automatisation de deux activités liées à la revue de code : fournir des recommandations à l'auteur du code avant la soumission de la PR, et résumer au reviewer les modifications effectuées à travers des commentaires. Bien que les résultats soient prometteurs, les modèles construits nécessitent d'être encore améliorés afin d'être utilisables par les développeurs.

2.2 Les Linters

Après avoir examiné la revue de code, une méthode manuelle et collaborative d'assurance qualité, nous nous tournons désormais vers les linters. Ces outils automatisent la détection des erreurs et l'adhésion aux conventions de code, offrant une approche complémentaire qui optimise le processus de vérification du code sans remplacer l'expertise

humaine essentielle des revues de code.

Les linters sont des outils d'analyse statique qui analysent automatiquement le code source pour alerter les développeurs lorsqu'ils ne respectent pas certaines pratiques de code, appelées règles dans ce contexte. Lorsqu'une règle est enfreinte, une notification, détaillant succinctement la règle, est affichée au développeur et amène celui-ci à réagir pour corriger le problème. Ces règles peuvent porter sur divers sujets tels que la sécurité, la performance, les conventions de codage ou encore des règles de style. Ils jouissent d'une grande popularité. À ce titre, Tómasdóttir *et al.* [12] ont découvert qu'un quart des dépôts JavaScript qu'ils ont analysés sur GitHub utilisent au moins un linter.

Les linters sont des outils d'analyse, dite **statique**, puisqu'ils nécessitent simplement le code source du programme afin de pouvoir fonctionner. Ils se complémentent aux outils d'analyse **dynamique** qui requièrent l'exécution du programme. Le fait d'analyser un programme pendant son exécution permet de récupérer des informations liées à son utilisation. Nous pouvons par exemple détecter des failles mémoires [5], ou encore calculer la couverture de code [6], qui sont deux tâches plus difficiles à obtenir de façon statique que de façon dynamique. Cependant, un linter est plus facile à mettre en place puisqu'il ne requiert pas d'avoir un environnement fonctionnel pour exécuter puis analyser le code. En pratique, ces linters analysent le code source en construisant d'abord un Arbre Syntaxique Abstrait (*AST*, en anglais), qui représente la structure logique du code ; puis ils appliquent un ensemble de règles, rédigées sous forme de patterns ou de programmes, pour identifier les erreurs.

Le mot *lint* est utilisé pour la première fois en 1978 par Stephen C. Johnson lors de la présentation de son outil éponyme [41] qui permet de détecter les problèmes de portabilité pour le langage de programmation C. En terme d'étymologie, *lint* signifie une peluche de vêtements en anglais ; le *linter* est donc l'outil qui permet de capturer ces mauvaises peluches et de rendre ainsi l'habit plus propre. Le terme est ensuite utilisé l'année suivante pour la première fois de manière publique lors de la sortie de la septième version d'Unix. Aujourd'hui, les linters visent toujours le même objectif : réduire les erreurs de programmation ; mais ils couvrent également un domaine d'analyse plus large. Selon le site web https://analysis-tools.dev, il existe plus de 600 linters disponibles pour environ 70 langages de programmation différents. Même si de manière générale un linter est spécialisé dans un unique langage, il est régulier qu'un linter supporte deux langages proches, par exemple JavaScript et TypeScript, ou C et C++. Cependant, il est plus rare de trouver un linter qui serait compatible avec du Java, du C# et du Python simultanément. Les linters sont pour la plupart des projets en open-source, mais il existe également des solutions payantes, comme *Klocwork* [7] [8] par exemple.

Chaque linter met à disposition un ensemble de règles qui sont déjà implémentées dans l'outil et utilisables immédiatement. À titre d'exemple, *ESLint* [9], un linter pour JavaScript, en comporte quasiment 200 dans sa configuration par défaut, qu'il est ensuite possible d'étendre à l'aide de plugins externes. Si un développeur a besoin de comprendre une règle, l'éditeur du linter met à disposition une documentation pour chacune des règles. Nous pouvons par exemple consulter la règle *no-var* présente dans *ESLint*, accessible à https://eslint.org/docs/latest/rules/no-var, qui proscrit l'utilisation du mot clé *var* pour déclarer une variable. Il n'est pas rare qu'un linter comporte des règles

5. https://valgrind.org
6. https://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html
7. https://www.perforce.com/products/klocwork
8. Petite anecdote sur le nom qui se décompose en **K** = 1000, **loc** = lines of code et **work**, ce qui donne "Des milliers de lignes qui fonctionnent"
9. https://eslint.org

FIGURE 2.2 – Notification de la règle *no-var* de *ESLint* dans l'IDE *WebStorm*

```
  60:157644  error    Strings must use singlequote                                  quotes
  60:157651  error    Missing semicolon                                             semi

/Users/clatappy/Documents/Thèse/linter-taxonomy/survey/results-viewer/src/composables/results/getCompleteResults.ts
   69:5   error    Unexpected var, use let or const instead  no-var
  166:13  warning  'P' is assigned a value but never used    @typescript-eslint/no-unused-vars

/Users/clatappy/Documents/Thèse/linter-taxonomy/survey/results-viewer/src/domain/Result.ts
  353:65  warning  '_' is defined but never used  @typescript-eslint/no-unused-vars

/Users/clatappy/Documents/Thèse/linter-taxonomy/survey/results-viewer/src/env.d.ts
  4:40  error  Missing semicolon
```

FIGURE 2.3 – Exemple de rapport généré par *ESLint* en utilisant la ligne de commande

qui ne correspondent pas à la culture interne d'une équipe de développeurs. Ils offrent donc la possibilité, à l'aide d'un fichier de configuration, de choisir quelles règles doivent être activées sur le projet. Plus finement, certaines règles mettent elles-mêmes à disposition des options que nous pouvons paramétrer. Par exemple, la règle *quotes*[10] de *ESLint*, assurant l'utilisation constante du même type de guillemets, permet de configurer si nous souhaitons utiliser les simples ou les doubles. Cela permet ainsi d'adapter une règle afin qu'elle corresponde aux habitudes de l'équipe de développement.

L'atout principal du linter est d'être directement intégré au sein des outils des développeurs, notamment de leur éditeur de code. Nous pouvons donc analyser en temps réel à chaque modification du code si des erreurs viennent d'être introduites. Cette analyse à *t-0* permet d'éviter d'introduire des erreurs dans la base de code et ce, avant l'exécution du code. Lorsqu'un problème est détecté dans le code, un avertissement est émis au développeur sur la ligne correspondante afin qu'il puisse inspecter le code incriminé et agir en conséquence. Sur la Figure 2.2, il est présenté un exemple d'intégration de *ESLint* dans un éditeur. Nous observons au début de la fonction l'utilisation du mot-clé *var*, qui est pourtant interdit par la règle précédemment présentée *no-var*. Ainsi, le mot-clé est souligné en rouge dans l'éditeur, et en passant la souris dessus, nous obtenons plus de détails sur l'erreur afin d'apprendre quelle en est l'origine et comment la résoudre.

Certains linters vont même au-delà de la simple détection d'erreurs et permettent de corriger automatiquement le code afin de le rendre conforme. Sur la figure précédente (*cf.*, Figure 2.2), nous pouvons voir que *ESLint* propose de convertir *var rawResultRes = ...;* en *const rawResultRes = ...;*. Cela permet de gagner du temps pour obtenir un code conforme. Par ailleurs, cela a également un but éducatif puisqu'en ayant vu comment corriger l'erreur, le développeur utilisera peut-être directement la bonne syntaxe la prochaine fois sans l'intervention du linter.

La plupart des linters offrent la possibilité de lancer une analyse d'un projet à travers une interface en ligne de commande. Comme visible sur la Figure 2.3, le retour de base de l'analyse n'est pas aussi convivial qu'une intégration dans un éditeur. La ligne de commande est donc rarement invoquée telle quelle, mais elle présente surtout un fort intérêt quand elle est utilisée dans les chaînes d'intégration continue et de livraison (CI/CD). Ainsi, à chaque mise à jour sur le dépôt de code, une analyse peut être exécutée automatiquement afin de générer un rapport sur les erreurs introduites lors des modifications.

10. https://eslint.org/docs/latest/rules/quotes

2.2.1 Bénéfices

Le premier bénéfice lié à l'utilisation des linters est l'augmentation de la qualité logicielle en réduisant le nombre de potentielles erreurs [12, 42, 43, 44, 45, 46]. Ce bénéfice, bien que remonté par les développeurs lors d'entretiens, n'est pas clairement mesuré dans les données empiriques lorsque nous observons les bugs évités par l'utilisation des linters [47, 48]. Cependant, le fait important de la détection de violation est d'avoir lieu tôt dans le processus de développement. Le développeur est averti de manière presque immédiate lorsqu'il écrit du code qui enfreint une règle. Cela permet de corriger immédiatement un problème qui aurait pu coûter des ressources importantes de débogage pour trouver son origine.

En plus de limiter la présence d'erreurs, les linters permettent également d'imposer un style de code consistant au sein d'une équipe de développement [46]. Ceci est crucial dans les environnements où plusieurs développeurs travaillent sur le même projet, car cela facilite la lecture et la maintenance du code. Les règles de style peuvent inclure la mise en forme du code, le nommage des variables, l'utilisation de constructions spécifiques du langage, etc.

En automatisant la détection de problèmes courants et en imposant des normes de codage, les linters diminuent ainsi le temps nécessaire pour les revues de code [46]. Cela permet aux développeurs de se concentrer sur des aspects plus critiques du code lors des revues, tels que la conception et la logique métier, plutôt que sur des erreurs triviales ou des débats sur le style de codage.

Enfin, un autre avantage à l'utilisation des linters est d'être informé lors de l'évolution d'un langage [46]. Par exemple, JavaScript évolue régulièrement avec de nouvelles fonctionnalités ajoutées à travers les mises à jour d'ECMAScript. Les linters aident les développeurs à adopter ces nouvelles constructions de langage en les identifiant et en fournissant des suggestions ou des corrections automatiques. Cela aide ainsi les équipes à rester à jour avec les meilleures pratiques et à exploiter pleinement les capacités du langage.

2.2.2 Freins à l'Adoption

Une première problématique à laquelle les linters sont confrontés concerne les faux-positifs [49, 42, 44, 50]. Un faux-positif se produit lorsqu'un linter identifie une violation concernant une règle sur du code qui, après vérification, n'enfreint en réalité pas la règle. L'écriture du pattern étant réalisée par un développeur, il se peut, par exemple, qu'une exception ait été omise lors du développement de la règle et signale un cas qui ne le devrait pas. Nous parlons également de faux-négatif lorsqu'à l'inverse une règle est enfreinte par le code, mais non signalée par le linter. Cependant, en pratique, les faux-positifs ont un impact négatif plus important que les faux-négatifs. En effet, Christakis *et al.* [42] ont découvert que les développeurs préféraient disposer d'outils détectant moins d'erreurs mais réelles, plutôt que ceux en générant beaucoup à tort. Ils ont également relevé que 90% des développeurs étaient prêts à accepter un taux de faux-positifs de maximum 5%, et seulement 24% d'entre eux toléraient un taux de faux-positifs supérieur à 20%. Un développeur qui perd du temps à vérifier les informations retournées par un outil, qui est censé l'aider à travailler plus efficacement, finira par le désactiver, ce qui freinera fortement son adoption. Les outils imposant des pratiques de code doivent donc s'efforcer d'atteindre une haute précision ($\geq 80\%$) plutôt qu'un haut rappel. Pour rappel, la précision calcule la proportion d'éléments pertinents et sélectionnés par rapport à l'ensemble des éléments sélectionnés, et comme précision, le rappel calcule le nombre d'éléments pertinents et

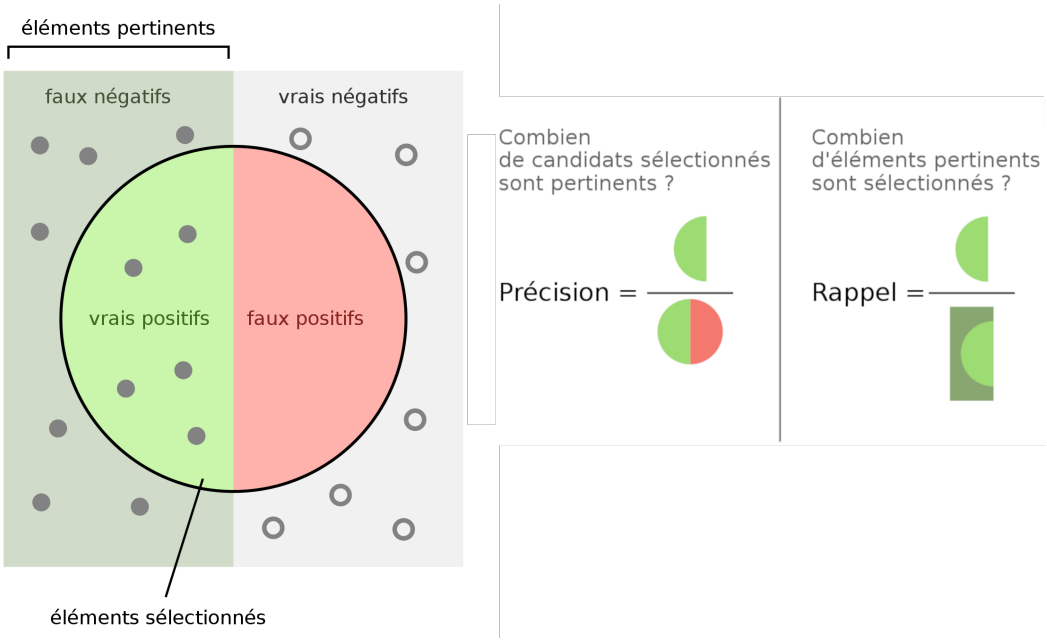
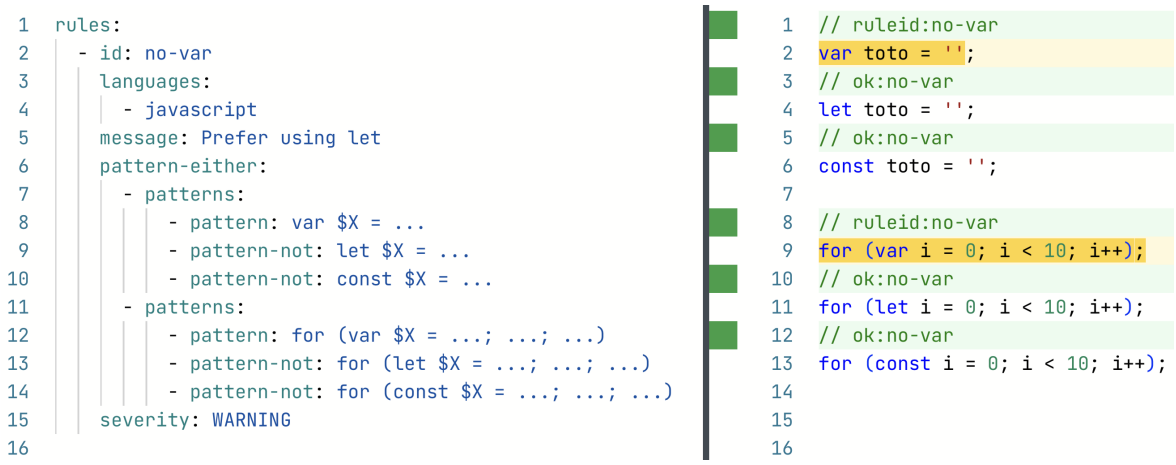


FIGURE 2.4 – Diagramme de Venn illustrant le calcul de la précision et du rappel
Crédits : Datamok / User :Walber, CC BY-SA 4.0, via Wikimedia Commons

sélectionnés par rapport à l'ensemble des éléments pertinents (*cf.* Figure 2.4).

Tómasdóttir *et al.* [46] ont constaté des difficultés liées à la quantité de règles fournies par les linters. Comme expliqué plus haut, les linters possèdent souvent plus d'une centaine de règles et qui sont, pour la plupart, configurables à travers des options, ce qui ajoute de la complexité. Lorsque nous souhaitons intégrer un linter, il y a trois options : laisser la configuration par défaut, utiliser une liste de configuration fournie directement par les éditeurs ou par la communauté, ou parcourir l'ensemble des règles une par une pour savoir lesquelles activer. Les deux premières options permettent un démarrage rapide de cette intégration, mais il y a de fortes chances que certaines règles ne correspondent pas à la convention habituelle de l'équipe. La troisième option nécessite de passer un temps considérable à examiner et à comprendre chaque règle pour s'assurer qu'elles s'alignent avec les pratiques de code de l'équipe. Dans tous les cas, il faudra y accorder du temps pour affiner la configuration qui convient pour le projet [44]. Beller *et al.* [51] ont d'ailleurs constaté que, dans plus de la moitié des cas (56%), les linters étaient configurés uniquement au démarrage du projet, mais n'étaient plus jamais modifiés. Un autre frein concerne la mise en accord des règles à activer dans une équipe de développeurs. Chaque développeur a des habitudes de développement qui lui sont propres, et trouver un consensus peut s'avérer complexe. Tous ces problèmes sont accentués lors de l'intégration d'un nouveau linter sur un projet qui existe déjà [46]. Il y a de fortes chances, si nous activons a posteriori un linter, que des conventions différentes aient été appliquées pendant la phase de développement. Cet effet aura pour risque de provoquer un nombre élevé d'erreurs à corriger, rendant le processus d'intégration du linter fastidieux et potentiellement décourageant pour l'équipe.

Une autre problématique importante concerne la limitation des règles implémentées par le linter. En effet, comme expliqué précédemment, de plus en plus d'organisations tendent à créer leurs propres pratiques internes qu'elles souhaiteraient ensuite détecter. La plupart des linters offrent la possibilité d'implémenter ses propres pratiques et donc de pouvoir les intégrer dans le processus de qualité d'une organisation. Cependant, cette opération a un coût non négligeable puisque cela implique d'allouer de la ressource, en



```yaml
rules:
  - id: no-var
    languages:
      - javascript
    message: Prefer using let
    pattern-either:
      - patterns:
          - pattern: var $X = ...
          - pattern-not: let $X = ...
          - pattern-not: const $X = ...
      - patterns:
          - pattern: for (var $X = ...; ...; ...)
          - pattern-not: for (let $X = ...; ...; ...)
          - pattern-not: for (const $X = ...; ...; ...)
    severity: WARNING
```

```javascript
// ruleid:no-var
var toto = '';
// ok:no-var
let toto = '';
// ok:no-var
const toto = '';

// ruleid:no-var
for (var i = 0; i < 10; i++);
// ok:no-var
for (let i = 0; i < 10; i++);
// ok:no-var
for (const i = 0; i < 10; i++);
```

FIGURE 2.5 – Exemple de pattern *Semgrep* pour reproduire la règle *no-var* d'*ESLint*

temps et en expertise, pour les mettre en place. Il sera nécessaire dans un premier temps d'apprendre le fonctionnement et la syntaxe pour créer la nouvelle pratique. Si nous regardons le fichier source développé pour la règle *no-var* présente dans ESLint [11], le fichier final fait quasiment 150 lignes de code. Alors que la règle de départ est triviale, en ce sens qu'elle consiste simplement à interdire l'utilisation du mot-clé *var* pour déclarer une variable, nous pouvons imaginer la difficulté de l'opération pour des règles plus complexes. Il ne faut pas non plus négliger la maintenance de ces règles a posteriori. Les langages et les pratiques évoluant constamment, il sera nécessaire d'apporter des correctifs à certains patterns pour qu'ils suivent cette évolution. Également, pour lutter contre les faux-positifs, il est indispensable d'implémenter une suite de tests pour vérifier que les patterns répondent bien à ce qui est attendu. Cela représente donc un effort considérable et un travail d'expertise que peu d'organisations sont prêtes à investir. Enfin, le dernier frein est la structure inhérente aux linters qui utilisent majoritairement des arbres syntaxiques abstraits (AST en anglais) pour détecter les patterns. L'écriture d'une nouvelle règle nécessite donc que son auteur maîtrise ce type de structure, ce qui est une compétence rare [52]. Il existe cependant des solutions qui abstraient l'utilisation de ces arbres, comme Semgrep [12]. Il s'agit d'un outil qui offre une syntaxe très proche de la structure réelle du code (*cf.*, Figure 2.5) et permet un onboarding beaucoup plus rapide pour créer des premières règles. Mais là encore, Semgrep nécessite un travail profond afin d'en saisir toutes ses subtilités, et les phases de test et de maintenance des règles restent aussi lourdes.

Une dernière problématique concerne l'utilisation en soi du linter. Pour une des raisons citées précédemment, un développeur peut prendre la décision de désactiver son linter dans son éditeur ou d'ignorer simplement les erreurs remontées [46]. Il a d'ailleurs été observé que ce phénomène était accentué lorsque le niveau de sévérité de la règle enfreinte n'était pas critique. Une solution a été d'intégrer l'analyse du linter dans la CI/CD et de bloquer la phase de construction si des erreurs étaient remontées. Cela permet de s'assurer qu'aucune erreur n'est introduite, mais rend l'écriture du code plus fastidieuse en ajoutant des aller-retours entre l'analyse de la chaîne d'intégration et le développeur qui doit corriger les erreurs détectées.

Ainsi, les linters sont régulièrement utilisés au sein des projets pour améliorer la cohérence, la lisibilité, la performance, la sécurité et la maintenabilité du code. Cependant,

11. https://github.com/eslint/eslint/blob/main/lib/rules/no-var.js
12. https://semgrep.dev

il existe encore des challenges, tels que les faux-positifs ou la mise en place difficile de règles internes, qui bloquent l'adoption des linters par les équipes de développement. La solution que nous développons chez Packmind tente de répondre à ces différentes problématiques, en favorisant l'émergence des pratiques, leur mise en commun, puis leur accès.

2.3 L'Approche Packmind du Partage des Pratiques

Packmind est une entreprise française spécialisée dans le partage de connaissances pour les développeurs. Son outil éponyme permet aux développeurs d'identifier et de partager des pratiques de code au sein d'une équipe de développement. Les développeurs créent des pratiques pendant leur processus de développement habituel en sélectionnant directement du code depuis leurs outils. Ensuite, ils se réunissent autour d'un atelier afin de discuter des pratiques remontées depuis le précédent. Packmind est une solution disponible en hébergement interne pour chaque organisation, mais aussi accessible facilement en version SaaS [13]. Depuis le lancement de cette dernière fin-juin 2021 et à date de début-mars 2024, 4765 pratiques ont été créées par 228 organisations différentes qui se sont retrouvées autour de 1315 ateliers. En complément à Packmind, il existe un hub communautaire [14], ouvert aux contributions, qui regroupe des catalogues de pratiques. Actuellement, il compte 413 pratiques dans une large gamme de langages de programmation, stockées dans 30 catalogues.

Dans le but d'appréhender plus facilement les problématiques auxquelles nous sommes exposés, cette section présente les principales fonctionnalités de Packmind.

2.3.1 L'Écosystème Packmind

Les utilisateurs de Packmind se divisent en deux catégories : ceux qui alimentent les pratiques et ceux qui doivent les appliquer. Nous appelons ces deux acteurs les *Drivers* et les *Contributors*. Le premier acteur correspond à un profil possédant une forte expertise qui sera un moteur de la qualité et identifiera des pratiques régulièrement. Il sera également sollicité afin d'animer les revues de pratiques ; cela correspond donc plutôt à un développeur senior, un Tech Lead ou un Coach Craft. Le second acteur, quant à lui, est un développeur pour qui le partage et la création de pratiques n'est pas une priorité ; il se laissera guider pendant les ateliers.

Les pratiques sont créées et discutées par les développeurs eux-mêmes. Il n'y a pas de limite fixée en termes de catégories et de langages. Lors d'un atelier, il est donc possible d'échanger sur des pratiques classiques d'architecture ou de lisibilité, mais aussi sur des pratiques internes, des frameworks spécifiques ou sur la manière de tester, et ce pour des pratiques spécifiques à un langage, ou globales et indépendantes d'un langage. À titre d'exemple, la Figure 2.6 montre une pratique créée et présente sur notre instance. La pratique recommande l'ajout d'un identifiant unique, souvent le timestamp courant, en paramètre lors d'une requête à une URL. Cela force le navigateur à ne pas utiliser son cache afin de récupérer la réponse, et permet donc de récupérer la dernière version à jour de la ressource.

Une pratique chez Packmind est constituée au minimum d'un nom et d'un exemple de code, et peut aussi être agrémentée d'une description et de catégories. Dans le cadre

13. https://subscribe.promyze.app
14. https://bestcodingpractices.dev

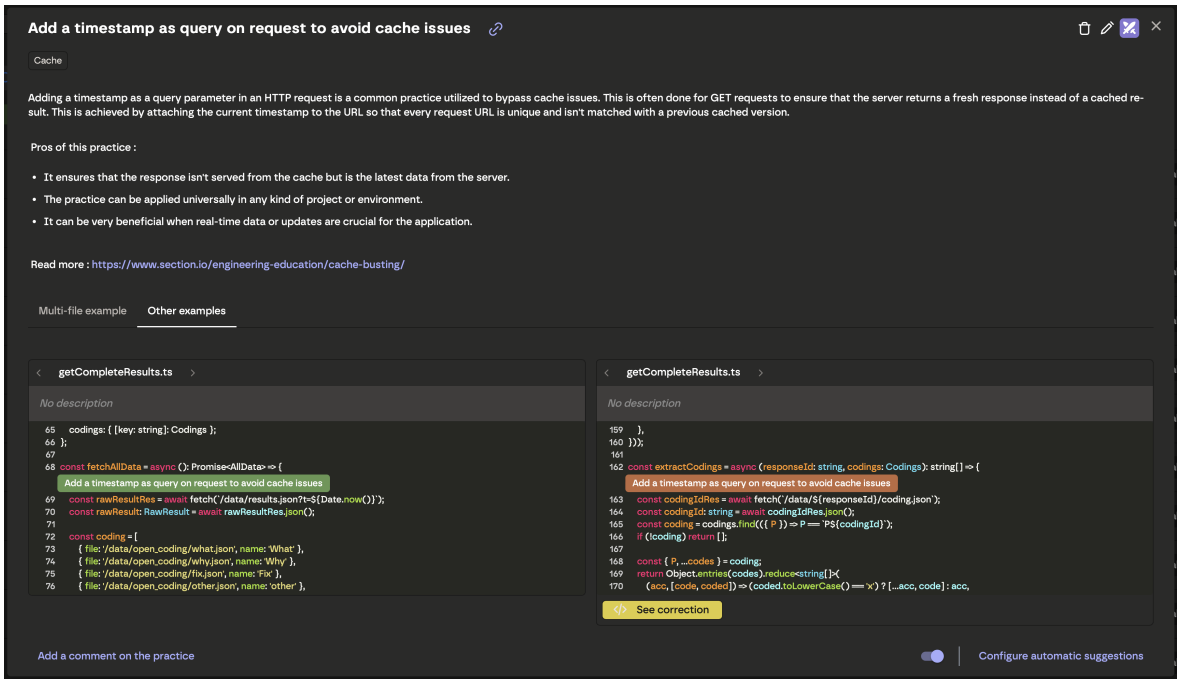


FIGURE 2.6 – Interface Packmind présentant une pratique

de leur travail, les développeurs sont régulièrement amenés à évaluer la qualité du code produit, à travers l'écriture ou la révision de code. Ce processus d'évaluation conduit à l'identification de deux catégories distinctes de pratiques : celles considérées comme bénéfiques, qui devraient être encouragées et adoptées au sein de l'équipe, et celles jugées préjudiciables, qui devraient être évitées. Pour une même pratique, nous pouvons donc la documenter avec du code conforme, dit positif chez Packmind, et du code non-conforme, dit négatif. Lorsqu'une pratique est documentée avec ces deux types d'exemple, il est ainsi plus aisé de savoir quel est le code à ne pas reproduire et comment le remplacer. De plus, pour un exemple négatif, il est également possible d'y associer un fragment de code représentant sa correction, facilitant ainsi le fait de corriger du code non-conforme. Un aspect intéressant de ces exemples de code est la proximité avec le code réel de l'application. Comme les développeurs créent les pratiques pendant leur processus de développement, le code soumis est du code de production. Cela permet d'avoir une documentation de pratique fortement liée à l'existant, évitant ainsi l'utilisation de code, dit "foo-bar", qui peut être difficile à contextualiser. Ce code réel peut aider un développeur qui découvre une pratique à mieux appréhender les différents cas d'usage et faciliter sa mise en application.

La première étape, qui consiste donc à identifier des pratiques, est réalisable depuis l'interface web de Packmind, mais prend tout son sens lorsqu'elle est réalisée sans interrompre le flux de travail des développeurs, directement depuis leurs outils.

Plugins IDE et navigateurs Afin d'encourager la mise en avant de pratiques en ne perturbant pas le processus habituel des développeurs, nous avons créé des plugins pour les éditeurs de code ainsi que pour les navigateurs. L'objectif principal est de pouvoir facilement sélectionner du code depuis n'importe où et d'en créer une pratique. Ils offrent également la possibilité d'avoir accès aux pratiques déjà créées. Cela permet en cas de doute sur une pratique, de la retrouver, de la consulter et de l'appliquer directement dans le code, sans avoir à quitter son outil de travail. Un dernier atout important de cette intégration est de pouvoir facilement apporter une correction lorsqu'un exemple négatif a été

envoyé. Lorsque cela se présente, il y a une forte possibilité que le développeur concerné en profite pour corriger le code enfreint. Dans ce cas, il peut simplement utiliser le plugin pour envoyer la correction, qui sera automatiquement associée au dernier exemple.

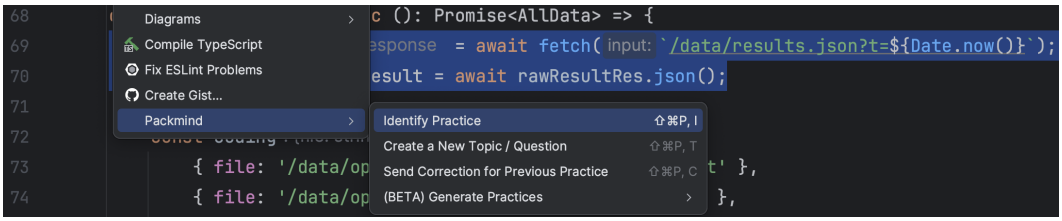
Bien qu'ils présentent des fonctionnalités similaires, les deux types de plugins n'ont cependant pas les mêmes cibles. Le plugin intégré à l'éditeur de code est destiné aux développeurs qui passent la plus grande partie de leur temps dans cet environnement. De leur côté, les plugins navigateurs permettent plutôt aux personnes parcourant le code à travers des outils web d'avoir accès à Packmind. Ces plugins navigateurs ont été spécialement conçus pour s'interfacer avec les outils de revue de code, présents par exemple sur GitHub ou GitLab, et en particulier sur les fonctionnalités de pull request. Souvent une revue de code est un échange entre un auteur et un reviewer, ce qui implique que les commentaires effectués n'impactent pas tous les développeurs d'une équipe. Ainsi, créer une pratique Packmind directement lors d'une revue de code permet de communiquer une unique fois en impliquant tous les développeurs qui seront présents lors de l'atelier suivant.

La création d'une pratique depuis les plugins est relativement simple. Comme indiqué sur la Figure 2.7a, il faut d'abord sélectionner les lignes de code qui sont concernées par la pratique que nous souhaitons mettre en avant, puis accéder au sous-menu de Packmind. La fenêtre de dialogue qui s'ouvre, voir la Figure 2.7b, expose l'ensemble des champs qu'il est possible de remplir pour décrire une pratique, le nom étant le seul champ obligatoire. La dernière étape est de valider la création en sélectionnant si l'exemple représente un exemple positif ou négatif de la pratique. Ainsi, en très peu de temps un développeur a la possibilité de partager une pratique sur laquelle il souhaite échanger, sans avoir eu besoin de quitter son environnement de travail.

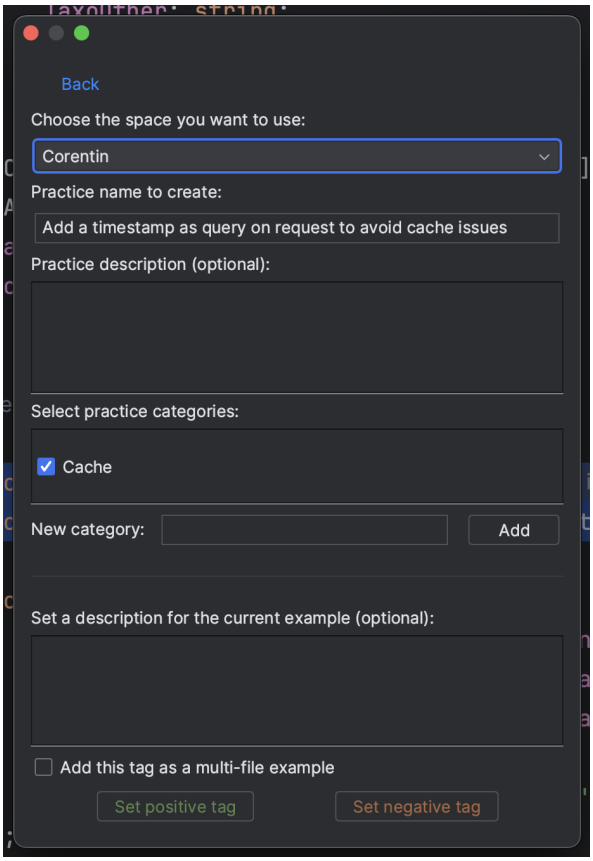
Lorsque suffisamment de pratiques ont été soumises, il est temps de procéder à un atelier de revue de pratiques.

Atelier de Revue des Pratiques L'atelier de revue des pratiques est le rendez-vous, en présentiel ou en distanciel, pendant lequel tous les acteurs du code d'une équipe, d'un projet ou d'une organisation se retrouvent afin de discuter des dernières pratiques remontées. À titre personnel, je pense que le cœur de notre approche se trouve dans ce moment d'échanges. Lors d'un atelier, l'interface de Packmind est disponible sur la Figure 2.8. La pratique en cours de revue est affichée dans un éditeur affichant l'ensemble du fichier dans lequel elle a été posée, et un widget est affiché sur les lignes concernées par la pratique. Ce widget permet notamment de la consulter, de la modifier, de la supprimer, ou si c'est une nouvelle pratique, de la valider pour qu'elle intègre la base commune. Nous pouvons ensuite naviguer entre les différentes pratiques qui sont en attente dans l'atelier.

En général, à chaque revue, une personne différente est désignée afin d'animer l'atelier. Cet animateur parcourt les pratiques une par une, et demande à la personne ayant posé la pratique courante de la présenter. Cette personne expose les raisons pour lesquelles elle pense que l'exemple est une bonne ou une mauvaise pratique, et le cas échéant comment il aurait fait. L'animateur s'occupe ensuite d'animer le débat, si besoin, pour savoir si la pratique doit être adoptée. Il est essentiel que tout le monde puisse être entendu, du développeur junior à l'expert technique. Comme déjà soulevé par les linters (*cf.*, Section 2.2), les développeurs peuvent parfois avoir du mal à trouver un consensus pour s'accorder sur l'application d'une pratique. Lorsque ce cas de figure se présente, il est possible de démarrer une *Battle* sur la pratique. Cela permet de la mettre de côté pour le moment et d'attendre la prochaine revue pour en débattre avec plus de recul. Pour ce



(a) Sélection du code à envoyer



(b) Saisie des informations de la nouvelle pratique

FIGURE 2.7 – Étapes pour créer une nouvelle pratique depuis l'IDE *WebStorm*

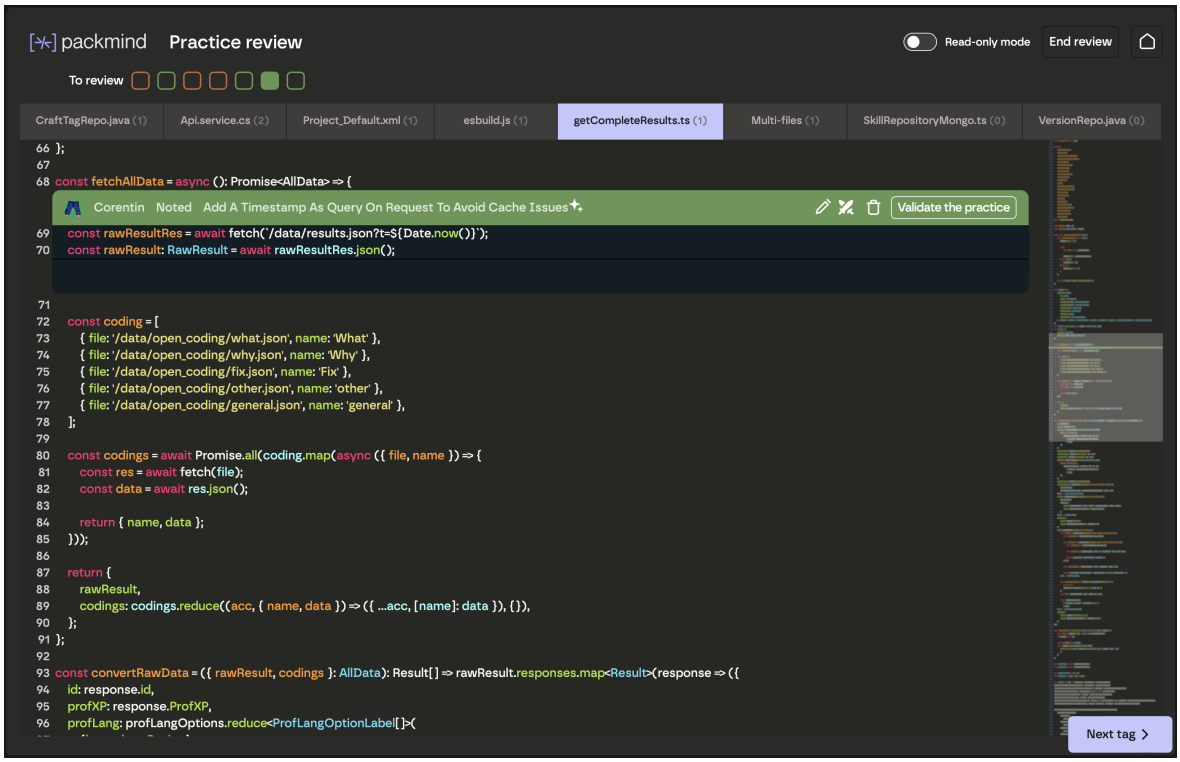


FIGURE 2.8 – Interface Packmind pendant le déroulement d'un Atelier Craft

faire, chacun peut déposer entre temps des arguments qu'ils soient pour ou contre et également voter pour son adoption. Ainsi à la revue suivante, il est possible de refaire un état des lieux en prenant en compte les avis de chacun, et finalement aboutir à une décision.

La fréquence de ces ateliers est propre à chaque équipe; certaines fixent une revue à intervalle de temps régulier (toutes les deux semaines par exemple) alors que d'autres attendent d'avoir un nombre suffisant de pratiques à parcourir. Il est important que l'atelier ne dure pas trop longtemps pour éviter une certaine saturation en apportant trop de changements, et de bien laisser le temps à chacun d'intégrer chaque pratique. Nous recommandons d'effectuer des ateliers d'une heure. Il peut parfois arriver de passer trente minutes à échanger sur une seule pratique et ne pas avoir le temps de parcourir toutes les pratiques soumises, mais cela ne doit pas être un problème. Si un tel échange a émergé, cela signifiait qu'il y avait un besoin particulier sur ce sujet et qu'il était important qu'il soit traité.

Au fur et à mesure du déroulement de ces ateliers, les pratiques s'accumulent constituant ainsi la base de connaissances internes à l'organisation.

La Base de Connaissances La Figure 2.9 présente l'affichage des pratiques disponibles au sein d'une instance Packmind. Nous y retrouvons l'ensemble des pratiques qui ont déjà été validées lors des ateliers, mais aussi celles qui sont en attente. Nous pouvons y rechercher une pratique, et la consulter afin d'obtenir toutes ses informations. C'est cette page qui regroupe l'ensemble des connaissances internes de l'organisation sur sa façon de développer. Cette documentation présente l'atout d'avoir été construite par les développeurs, qui se sont accordés tous ensemble, et donc d'avoir pour chaque pratique des exemples issus de la base de code existante.

Ainsi, l'atout de Packmind est de fournir un support aux développeurs pour qu'ils puissent échanger sur leurs propres pratiques internes. Cependant, à l'image des linters,

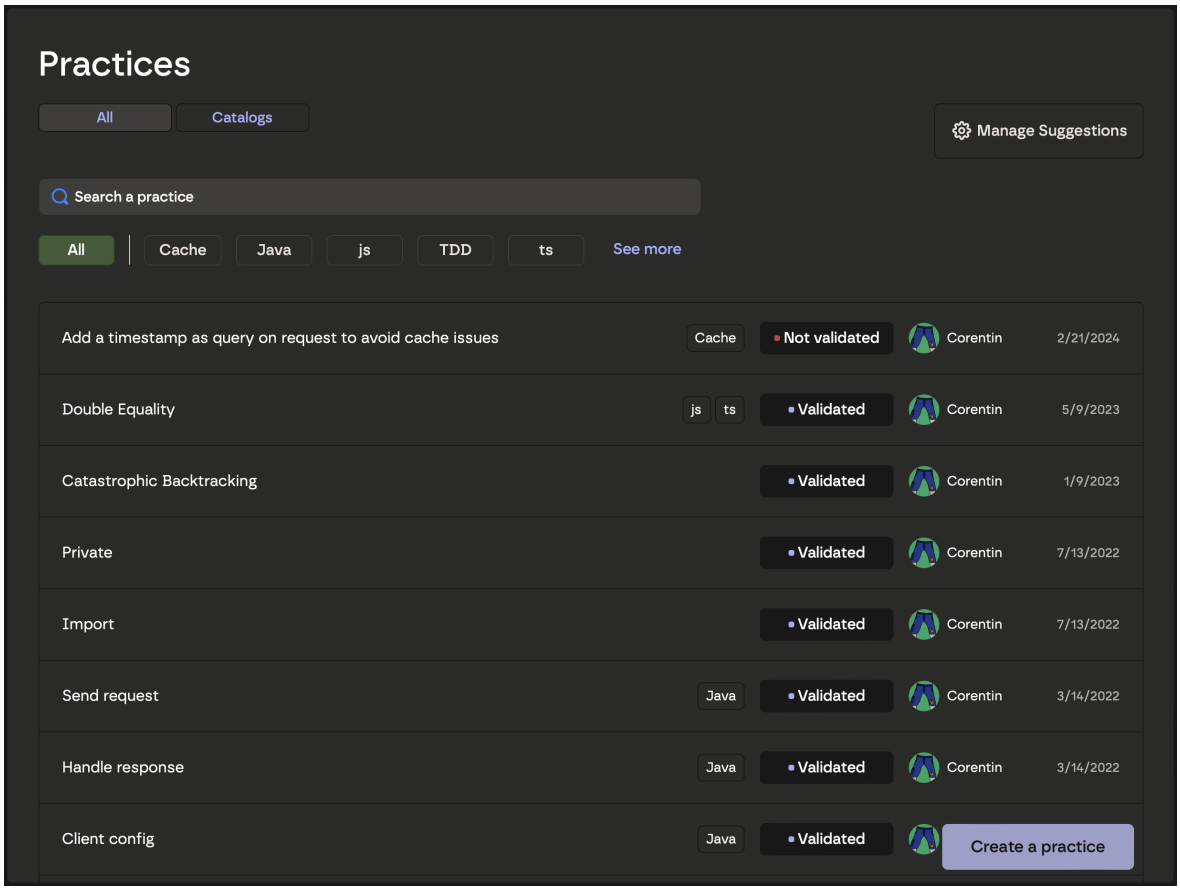


FIGURE 2.9 – Interface Packmind présentant la liste des pratiques

bien que la plate-forme soit correctement intégrée dans le processus de développement au début, avec le temps les ateliers se font moins réguliers jusqu'à devenir délaissés. Nous avons identifié quelques problèmes qui expliquent cette situation.

2.3.2 Freins à l'Adoption

Packmind est un outil qui permet de facilement faire émerger les pratiques, mais le manque d'opérationnalisation rend l'application systématique de ces pratiques beaucoup plus difficile.

Une problématique importante est la prise en compte des nouveaux développeurs au sein d'une organisation. C'est une étape primordiale, que nous appelons l'onboarding, qui a pour but de transmettre aux nouvelles personnes la culture et la manière de développer en interne. Actuellement, sur notre instance interne Packmind que nous utilisons pour la réalisation de nos propres ateliers de revue, nous avons quasiment 450 pratiques créées par notre équipe. Il est impossible pour une nouvelle personne nous rejoignant de toutes les consulter et de les intégrer. C'est un processus qui doit se faire petit à petit au fil du développement. Lorsqu'un nouvel arrivant consulte une pratique, pour laquelle il n'était pas présent lors de l'atelier, la manière dont elle est documentée influera fortement sur sa prise en compte. Également, agrémenter nos plugins avec la détection automatique permettra de faciliter grandement ce besoin.

Documentation des pratiques Le premier point à prendre en compte pour s'assurer qu'une pratique est appliquée par tous les développeurs, est d'être comprise par tous.

Comme pour les linters, si un développeur rencontre une nouvelle pratique qu'il ne connaît pas, il ira consulter sa documentation en espérant y trouver toutes les informations nécessaires. Cette documentation représente donc un point d'entrée important, et la façon dont la pratique est décrite va avoir un fort impact sur son adoption. À titre d'exemple, actuellement sur l'ensemble des pratiques créées sur la version SaaS de Packmind, 33% ne possèdent pas de description. Nous ne savons pas si c'est un élément indispensable, surtout qu'il n'est pas rare qu'une pratique ne nécessite pas de détails très poussés afin d'être comprise, un simple titre peut suffire. Mais il est déjà intéressant de noter une différence importante sur un aspect très simple des règles créées. Lorsque nous rédigeons la documentation d'une pratique, il faut être capable de se projeter à la place d'une personne qui va la découvrir et va devoir l'appliquer. Cependant, au début de cette thèse, nous n'avons trouvé aucune méthode permettant d'aider à réaliser cette tâche. Il nous semble donc primordial, afin notamment d'aider nos *Drivers* à documenter des pratiques, d'évaluer les attentes et les besoins des développeurs pour en inférer des recommandations.

Nous avons répondu à cette problématique en réalisant un état de l'art sur la documentation actuelle des linters, présenté dans le Chapitre 3.

Détection des pratiques La seconde problématique repose sur l'absence de retour de la bonne application des pratiques par les développeurs. Les *Drivers*, qui sont les responsables de la qualité, peuvent avoir besoin d'obtenir un suivi précis sur la diffusion des pratiques, afin de pouvoir organiser des ateliers spécifiques selon les besoins. Il est également important pour les *Contributors* d'être assurés en permanence qu'ils appliquent ce qui est attendu. Bien que les revues de code pourraient permettre de s'assurer de ces points, il serait préférable de consacrer ce temps à discuter d'autres problématiques que celles déjà présentées lors des ateliers. Afin de répondre à cela, il serait intéressant d'être capable de reproduire le comportement des linters. Dans cette optique, nous avons déjà tenté de mettre en place diverses solutions, telles que les regex ou Semgrep [15], pour automatiser cette détection. Cependant, à la suite d'échanges avec nos clients, il a été conclu que leur utilisation était trop complexe, car cela nécessitait une expertise avancée. L'idéal serait donc de disposer d'un outil qui ne nécessiterait aucune intervention humaine supplémentaire à partir du moment où une pratique est validée.

Pour cela, nous avons réalisé une étude, détaillée dans le Chapitre 4, dans laquelle nous avons utilisé un modèle d'apprentissage automatique pour apprendre et détecter automatiquement des pratiques.

15. https://semgrep.dev

Chapitre 3

Documenter les Pratiques de Code

Ce chapitre présente les travaux effectués pour la publication suivante [53] :

> C. Latappy, T. Degueule, J.-R. Falleri, R. Robbes, X. Blanc, C. Teyton, "What the Fix? A Study of ASATs Rule Documentation", 32th International Conference on Program Comprehension (ICPC), Lisbon, Portugal, 2024, doi : 10.1145/3643916.3644404

Sommaire

3.1 Introduction

Comme introduit dans la Section 1.2, dans ce chapitre, nous nous concentrons sur le scénario dans lequel un développeur rencontre pour la première fois une notification

d'un linter et souhaite connaître les détails sur la règle enfreinte. Habituellement, les avertissements générés par les linters dans l'IDE ou dans le terminal, à la suite de l'utilisation de son interface en ligne de commande, ne contiennent que des informations limitées [54, 17]. Pour expliquer plus en détail une règle donnée, les linters fournissent donc une documentation plus extensive et disponible en ligne.

Nous estimons que cette documentation est essentielle pour améliorer les connaissances et les compétences des développeurs, ainsi que pour promouvoir le respect des règles. Les notifications affichées par les linters ont déjà été étudiées [54, 17], et des analyses empiriques montrent que leur qualité n'est pas idéale [17, 18, 19]. De plus, à notre connaissance, il n'existe pas d'étude complète sur la documentation des règles des linters, que les développeurs consultent après la notification initiale. La documentation des règles de linters n'est pas une documentation logicielle classique, telle que peut l'être une documentation d'API [55]. En effet, la documentation des règles des linters est difficile car elle doit expliquer les problèmes indépendamment de tout contexte, de telle manière que les développeurs puissent les comprendre dans leur propre contexte. Un autre point difficile réside dans la subjectivité de certaines règles des linters, par exemple celles liées à l'utilisation des instructions goto [16] ou au dilemme classique de choisir entre snake et camel case [56]. Ainsi, nous savons peu de choses sur la meilleure façon de documenter les règles des linters. Ce manque de connaissance est préjudiciable lorsque des outils comme Semgrep permettent aux développeurs du monde entier de créer et partager facilement des centaines de règles.

Dans ce chapitre, nous réalisons d'abord une analyse empirique sur la manière dont les règles de linters sont documentées en pratique, puis nous confrontons cette analyse aux attentes des développeurs afin d'en extraire des recommandations, dans le but d'augmenter la qualité de cette documentation.

Nous commençons par réaliser l'état de l'art dans la Section 3.2. La Section 3.3 détaille notre étude de plus de 100 règles à travers 16 linters, couvrant 7 langages de programmation. Grâce à une analyse itérative de ces règles, nous décrivons une nomenclature couvrant 15 concepts de documentation dans 3 thèmes. Dans la Section 3.4, nous affinons notre nomenclature en une taxonomie sur les objectifs de documentation (*Quoi* : ce qui déclenche la règle, *Pourquoi* : pourquoi elle est importante, *Correction* : comment corriger le problème) et sur les types de contenu (*Texte*, *Code*, et *Liens*). Seulement la moitié des règles que nous analysons définissent l'objectif *Pourquoi*, alors que le *Quoi* et le *Correction* sont souvent présents, sous la forme de texte et de code. Ensuite, nous utilisons notre taxonomie pour confronter la documentation de 12 règles existantes avec les attentes des développeurs via une enquête. La Section 3.5 présente l'enquête dans laquelle nous avons collecté 298 évaluations de règles produites par 85 participants. Parmi les découvertes, les développeurs soulignent les problèmes de qualité avec la documentation de l'objectif *Pourquoi*, et mettent en évidence à la fois les aspects pédagogiques et le besoin de concision dans la documentation des linters. Pour clore ce chapitre, nous discutons des limites de cette étude dans la Section 3.6, avant de conclure dans la Section 3.7 et de présenter dans la Section 3.8 l'impact de cette étude pour Packmind.

Un package de réplication incluant les données brutes, la nomenclature et la taxonomie, ainsi que les résultats de l'enquête, est disponible en ligne [1] et sur Zenodo [20].

1. https://icpc2024-asats.github.io

3.2 État de l'Art

Cet état de l'art est divisé en deux parties : une première partie qui présente des travaux effectués sur différents types de contenu liés aux outils d'analyse statique (*Static Analysis Tools* ou *SAT* en anglais) et une seconde partie qui regroupe des travaux sur la documentation logicielle.

3.2.1 Analyse des Contenus liés aux SAT

À notre connaissance, il n'existe aucune étude aussi approfondie sur le contenu des documentations de pratiques de code que le travail que nous souhaitons réaliser. La majorité des contributions actuelles ont analysé les avertissements (également appelés notifications) affichés par les SAT lors de la détection d'une erreur.

Étude Générale sur les SAT Il existe une étude, réalisée par Novak *et al.* [57], qui propose une taxonomie des caractéristiques et atouts des outils d'analyse statique. Ils ont analysé 4 outils différents, à savoir *Gendarme* et *StyleCop* pour le *C#*, et *CheckStyle* et *FindBugs* pour le *Java*. L'analyse de ces outils a mené à la production d'une taxonomie à travers 10 catégories différentes :

- Input : le type de fichier qui est chargé (code source, bytecode)
- Publication : la fréquence des mises à jour
- Langages supportés : les langages pris en compte
- Technologie : les technologies utilisées pour l'analyse du code (syntaxe, flux de données, ...)
- Règles : le type de règles
- Configuration : la manière de configurer l'outil
- Extension : la possibilité de créer ses propres règles
- Accès : l'accès à l'outil (open source, gratuit, payant)
- Expérience utilisateur : les options offertes pour utiliser l'outil (IDE, CLI, ...)
- Sortie : la mise en forme des résultats (HTML, XML, ...)

La catégorie sur le type de règles existantes semble intéressante. En effet, cela s'apparente à une sorte de catégorie à laquelle la règle pourrait être rattachée et qui pourrait par la suite être intégrée dans sa documentation. Elle est composée de 9 sujets : Style, Naming, General, Concurrency, Exceptions, Performance, Interoperability, Security et Maintainability. Cette liste a ensuite été légèrement modifiée par les travaux de Vassallo *et al.* [44] : Style, Naming, Concurrency, Exceptions, Performance, Security, SQL, Maintainability et Correctness. Cependant, nous pensons que cette liste pourrait être encore réduite, notamment sur l'aspect SQL qui est très lié à une technologie. Associer une technologie à une catégorie de règles impliquerait, par exemple, une création pour chaque type de base de données existante. Ce travail permet de définir un cadre sur ce que doit fournir un outil d'analyse statique ; cependant la documentation des règles de ces outils n'est pas du tout abordée.

Do *et al.* [58] ont également réalisé une étude sur les SAT en analysant de multiples aspects. Ils se sont intéressés à comprendre les différents facteurs qui expliquent la large utilisation de ces outils par les développeurs. Ce travail a été réalisé en collaboration avec

l'entreprise *Software AG*. Ils ont divisé leur recherche en deux parties : d'abord une évaluation interrogeant les développeurs, puis une analyse des rapports générés par *Checkmarx*[2] pour compléter leurs résultats. Checkmarx fait partie des outils d'analyse statique utilisés chez *Software AG*. Il s'agit d'un outil classique qui supporte de nombreux langages. Il peut être intégré dans les IDE et permet de générer des rapports alertant les développeurs lors du non-respect des règles. Ces rapports d'analyse contiennent notamment des informations précieuses sur la façon dont les développeurs ont fait face aux différentes erreurs remontées. Leur questionnaire comporte plus 40 questions et a été divisé en 6 parties pour le participant : profil, expérience avec les SAT, qualité des avertissements des SAT, contexte actuel d'utilisation des SAT, importance des fonctionnalités des SAT et stratégies pour corriger une erreur. Ce questionnaire a ensuite été partagé aux 120 développeurs de l'entreprise *Software AG* et a reçu 87 réponses. La partie de leurs résultats la plus pertinente pour nos travaux est l'évaluation réalisée sur les fonctionnalités offertes par les SAT. Ils ont identifié 19 fonctionnalités différentes. Parmi celles-ci, 3 ont été jugées comme étant les plus importantes par les développeurs : *i)* Expliquer le fonctionnement du bug ; *ii)* Expliquer comment le bug affecte le code ; *iii)* Expliquer comment corriger le bug. Les 3 fonctionnalités les plus importantes sont toutes des fonctionnalités détaillant la nature et la manière de résoudre le bug. Nous retrouvons souvent ces explications dans la documentation des règles ou dans un message d'avertissement.

Étude des messages d'avertissement des SAT Les études sur le contenu des avertissements des SAT sont étroitement liées à la nôtre. En effet, l'avertissement lors de la détection d'une erreur est le premier élément qu'un développeur rencontre.

Johnson *et al.* [49] ont réalisé une première étude sur les outils d'analyse statique. Ils ont conduit des entretiens oraux avec 20 développeurs, dont l'expérience dans le domaine variait entre 3 et 25 années. Leur analyse était divisée en trois questions, dont deux qui avaient pour but *i)* d'analyser les raisons pour lesquelles les outils d'analyse statique n'étaient pas utilisés plus largement par les développeurs et *ii)* de savoir les améliorations à apporter à ces outils. En réponse à ces deux questions, le thème des notifications émises par ces outils a émergé. Ces notifications sont perçues comme un frein à l'adoption des outils d'analyse. La majorité des participants ont estimé que ces outils ne présentaient pas leurs résultats d'analyse avec suffisamment d'informations. Il est impossible pour eux d'évaluer la nature du problème, ses raisons et les actions à mettre en place pour y remédier, ce qui va jusqu'à provoquer la non-prise en compte de l'erreur. Nous observons, ici, que les développeurs expriment certains besoins et s'attendent à une documentation plus détaillée et de meilleure qualité, qui va au-delà des simples messages affichés dans les notifications. Concernant les améliorations à apporter, est revenue plusieurs fois l'idée de pouvoir mettre de côté l'erreur temporairement afin de pouvoir en discuter plus tard en équipe. Par ailleurs, un retour régulier est le besoin que ces notifications n'interrompent pas le processus de travail des développeurs et fournissent l'information principale de manière rapide et concise.

Suite à cette première étude, ils ont réalisé une nouvelle étude spécifiquement sur les notifications des outils réalisant de l'analyse de programme [18]. Ils ont fait le choix de s'intéresser aux notifications produites par un outil d'analyse statique (FindBugs), un outil de couverture (EclEmma) et un compilateur (Eclipse Java Compiler). Le but était d'évaluer les problèmes rencontrés par les développeurs face aux notifications de ces outils. Pour cette étude, ils ont réalisé des entretiens avec 26 participants qui avaient des tâches

2. https://checkmarx.com

de code à réaliser. Durant ces tâches, les participants étaient confrontés à de réelles notifications provenant des trois outils et devaient les interpréter. Parmi leurs résultats, deux enseignements sont importants. Ils suggèrent d'abord que les outils devraient constamment fournir des informations sur la manière de corriger une notification en détaillant le raisonnement. Ensuite, il faudrait adapter ces informations affichées aux développeurs de manière personnalisée. Pour cela, ils recommandent d'estimer l'expérience d'un développeur en prenant, par exemple, en compte les fonctionnalités du langage, les outils et les bibliothèques utilisés, ainsi que les interactions avec les notifications précédemment rencontrées.

Tahaei *et al.* [17] ont réalisé une enquête en ligne pour évaluer la capacité des notifications des outils d'analyse statique à accompagner les développeurs pour corriger des erreurs de sécurité. Ils ont choisi 4 erreurs récurrentes de sécurité : une injection SQL, l'utilisation de mots de passe en clair, un mauvais chiffrement et une exposition de données sensibles. Ils ont créé des fichiers de code contenant à chaque fois une des 4 vulnérabilités et ont utilisé SonarQube et SpotBugs pour générer des messages de notification. Le but était de comparer ces messages réels à une notification simple, affichant seulement le numéro de la ligne en cause. Cette dernière sert de groupe de contrôle pour mesurer la pertinence des notifications générées par de vrais outils. Ils ont ensuite mis en place un questionnaire dans lequel un développeur était confronté à du code vulnérable, dont la ligne non conforme était mise en avant, et la notification, qui pouvait provenir de SonarQube, de SpotBugs ou du groupe de contrôle. À chaque fois, le développeur devait choisir la correction à appliquer parmi un ensemble de propositions, et répondre à des questions pour évaluer le niveau de compréhension. Cette enquête, publiée en ligne, a obtenu 132 réponses. À titre informatif, les notifications de SpotBugs ont obtenu de meilleurs résultats que le groupe de contrôle, mais ce n'est pas concluant pour SonarQube. Le facteur qui a le plus influencé l'application de la bonne correction est l'expérience du développeur. Un développeur expérimenté a quatre fois plus de chance de corriger correctement l'erreur qu'un développeur junior. En outre, les résultats vraiment intéressants pour notre étude concernent les réponses à la question de savoir quels éléments dans les notifications ont été utiles aux développeurs. Il apparaît que les exemples de code ont été évalués comme éléments les plus utiles. Cependant, les liens et les métadonnées ont reçu des avis mitigés.

Pour résumer, le principal challenge des notifications des outils d'analyse statique est de fournir les informations principales de façon rapide et compréhensible. Un résultat important concerne le contenu des notifications d'erreur qui n'est pas perçu comme suffisamment utile par les développeurs [17, 18, 19]. Une autre constatation est la nécessité d'avoir des avertissements clairs et concis, en particulier lorsque les développeurs utilisent les SAT en ligne de commande [59]. Le contenu des notifications n'est donc pas satisfaisant et manque clairement d'informations importantes pour les développeurs. Si les notifications ne répondent pas aux besoins des développeurs, nous nous attendons à trouver ces informations dans la documentation complète des règles. Cependant, à notre connaissance, il n'existe aucune étude qui a analysé la qualité de la documentation des règles de SAT. Dans ce chapitre, nous souhaitons donc répondre à cette question en analysant la qualité actuelle de la documentation des règles et en confrontant cette analyse avec les attentes des développeurs.

3.2.2 Documentation Logicielle

La documentation logicielle apparaît comme un aspect crucial et important dans l'environnement du développement logiciel [60]. C'est un type de documentation qui a largement été étudié dans la littérature scientifique. De ce fait, il nous paraît utile de regarder ce pan de l'état de l'art pour informer notre propre étude sur la documentation très particulière des règles de SAT.

Forward *et al.* [61] ont réalisé une enquête auprès de professionnels dans le secteur du logiciel. Leur but était de mesurer la pertinence perçue de la documentation logicielle et d'identifier les solutions mises en place pour gérer cette documentation. L'enquête comporte 48 questions et couvre diverses thématiques de la documentation. Parmi ces dernières, nous retrouvons l'implication du participant dans la mise en place de la documentation dans son entreprise, la perception de l'état actuel de cette documentation ou encore l'évaluation de son utilité et de son efficacité. Cette enquête a récolté 48 réponses et a abouti à 8 grands enseignements, dont 2 qui pourraient également s'appliquer aux règles des SAT : la maintenance et l'évolution de la documentation. Un défaut majeur dans la documentation logicielle est la difficulté de la maintenir à jour tout au long de sa durée de vie. À ce titre, près de 70% des participants ont répondu que leur documentation était très rarement mise à jour. Cela contraste fortement avec le fait qu'une grande partie d'une documentation logicielle possède une durée de vie limitée et qu'elle devrait donc être supprimée. D'ailleurs, 80% des participants s'accordent sur le besoin d'adapter et de faire évoluer la documentation pendant le cycle de vie d'un projet. Nous pensons que si les auteurs de telles documentations disposaient de recommandations claires, l'écriture et la maintenance de la documentation serait plus aisée. Nous avons constaté récemment, à travers nos clients chez Packmind, que la documentation des pratiques de code souffre également du même problème de maintenance. C'est pourquoi nous voulons fournir des indications précises sur comment documenter les pratiques pour faciliter leur évolution dans le temps.

Aghajani *et al.* [62] ont réalisé une étude empirique sur la documentation logicielle afin d'explorer les difficultés rencontrées par les développeurs. Ce travail a permis de mettre en avant que les développeurs avaient besoin de documentations complètes et à jour. Mais il a surtout abouti à une taxonomie complète sur les différents problèmes de cette documentation en analysant diverses sources, comme des mails, des projets open source et des échanges sur StackOverflow. Cette taxonomie est divisée en 4 grandes catégories de problème : *i)* lié au contenu qui est écrit dans la documentation ; *ii)* lié à la façon dont est écrit le contenu dans la documentation ; *iii)* lié aux processus pour documenter ; *iv)* lié aux outils pour documenter. Les 2 premières catégories citées montrent l'importance du contenu et de la manière dont ce contenu est écrit dans les documentations. Cela nous conforte dans la volonté de réaliser une étude analysant le contenu actuel des documentations des règles de SAT, puis d'en extraire des recommandations.

Comme nous pouvons le constater, la documentation propre aux règles des outils d'analyse statique n'a pas encore été étudiée. Toutes les études se sont majoritairement concentrées sur les avertissements. Cependant, nous pensons que la documentation des SAT est aussi un aspect important qui mérite une analyse détaillée. Nous souhaitons être capable de mettre en avant ce qui est présent ou manquant dans la documentation existante et les comparer avec les besoins des développeurs. Nous aimerions également prendre en compte plus d'outils et de langages de programmation différents. Cela nous permettra de confirmer si les différents challenges observés dans les études sur les avertissements et la documentation logicielle sont applicables aussi à la documentation des pratiques.

Pour notre étude, nous souhaitons dans un premier temps analyser l'état actuel de la documentation des pratiques de code, puis, dans un second temps, confronter cette analyse avec les attentes des développeurs. Cela nous permettra de fournir des recommandations précises sur les éléments importants à documenter et comment le faire. Ces recommandations accompagneront les auteurs de documentation de pratiques dans la rédaction d'une documentation correctement structurée en termes de contenu et de forme. Cela facilitera notamment l'écriture initiale de la documentation, tout en simplifiant son évolution et sa maintenance dans le temps.

3.3 Nomenclature sur la Documentation des Pratiques

La première étape de notre étude consiste à créer une nomenclature basée sur la documentation des règles fournies par les linters. Une nomenclature est un outil de classification qui fournit une manière structurée et systématique de nommer et de se référer à une large gamme d'objets. Cette nomenclature nous aidera à définir les types d'informations que nous trouvons dans la documentation d'une règle et à les regrouper par thème. Pour la construire, nous utilisons un processus en trois étapes (*cf.* Figure 3.1) : ❶ nous sélectionnons un total de 16 linters divers qui fournissent une documentation pour leurs règles ; ❷ nous construisons itérativement un corpus de 119 règles, pour lesquelles nous codons les concepts de documentation que nous rencontrons ; ❸ nous comparons et analysons les documentations des règles pour fournir la nomenclature attendue, qui couvre 15 concepts différents.

3.3.1 Sélection des Linters

Un projet GitHub répertoriant les linters [3] référence plus de 600 outils différents ; en choisir un sous-ensemble raisonnable est à la fois nécessaire et difficile. Une première exigence stricte est que nous n'incluons que les linters se concentrant sur l'analyse et la détection des pratiques de code. Nous excluons donc, par exemple, les outils spécialisés dans le formatage comme Prettier [4]. Une seconde exigence stricte est que nous incluons uniquement les linters qui fournissent un site web présentant la documentation des règles. Enfin, nous sélectionnons des linters en mettant l'accent à la fois sur la *diversité* et la *popularité*. Puisque nous voulons que notre nomenclature soit utilisée au-delà de notre étude, indépendamment des linters, nous incluons un ensemble diversifié de

3. https://github.com/analysis-tools-dev/static-analysis, 12K étoiles
4. https://prettier.io

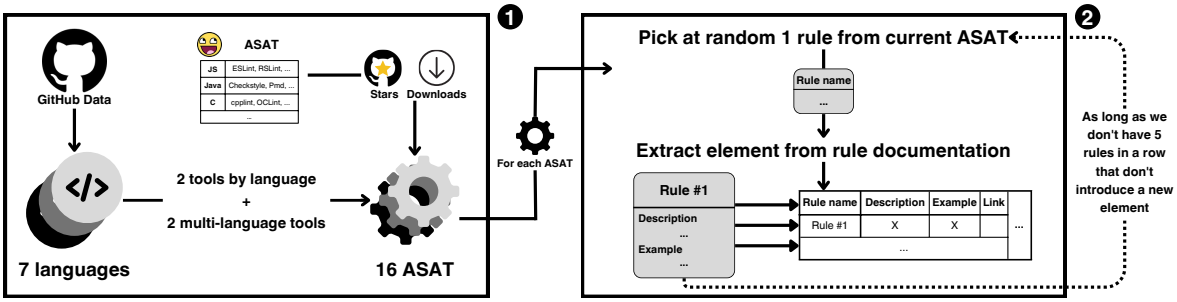


FIGURE 3.1 – Processus pour sélectionner et extraire les informations depuis les descriptions de règles

TABLEAU 3.1 – Langages et Outils sélectionnés

Langages	Outils	Langages	Outils
C / C++	OCLint Cppcheck	PHP	PHP CS Fixer Psalm
C#	Gendarme Roslynator	Python	Pylint Flake8
Java	Checkstyle SpotBugs	Ruby	RuboCop Brakeman
JS / TS	ESLint RSLint	Multi	Semgrep SonarLint

linters ciblant différents langages de programmation. Enfin, nous voulons orienter notre corpus vers les linters populaires car nous estimons que les linters populaires ont plus de chances d'inclure une documentation bien pensée et complète.

Nous définissons deux critères pour juger de la diversité et un critère pour juger de la popularité. Les deux critères de diversité que nous proposons sont : d'une part la couverture de la plupart des langages de programmation populaires par les linters [5], et d'autre part la possibilité d'avoir au moins deux linters par langage de programmation choisi. Le critère de popularité que nous proposons est la nécessité pour les linters d'avoir un dépôt GitHub avec au moins 1000 étoiles.

À partir de la liste des linters, nous sélectionnons les 16 linters, listés dans le Tableau 3.1, en utilisant nos critères et notre expertise Packmind. La seule exception au critère de popularité est Gendarme, que nous choisissons pour augmenter la diversité : c'est le linter officiel pour l'implémentation de Mono C#. Nous sommes conscients que cette sélection est quelque peu arbitraire et qu'une autre sélection aurait pu être faite avec les mêmes critères ; nous discutons de cette question dans la Section 3.6.

En plus des linters conçus pour un langage de programmation spécifique, nous trouvons intéressant d'examiner la documentation des linters supportant plusieurs langages de programmation. Dans ce cas, nous exigeons que ces linters aient leur propre moteur d'analyse et documentation des règles. Nous filtrons donc des outils tels que MegaLinter [6] car il combine seulement les résultats de plusieurs linters. En utilisant les critères décrits précédemment, nous sélectionnons finalement SonarLint et Semgrep.

Au total, nous obtenons 14 linters pour nos 7 langages de programmation et 2 linters prenant en charge plusieurs langages.

3.3.2 Codage des Éléments de Documentation

Notre sélection de linters offre une documentation de règles qui diffère en structure et en contenu. Cette différence peut également apparaître entre deux documentations de règles pour un même linter. Pour comparer les documentations des linters, nous identifions et alignons tous les concepts utilisés dans la documentation.

Par exemple, la Figure 3.2 affiche la documentation pour la règle *pointless-statement* fournie par `Pylint`. En utilisant des indices visuels et graphiques sur cette documentation (titres, sauts de ligne, cadres, images, etc.), nous extrayons et codons quatre concepts :

5. https://madnight.github.io/githut/#/pull_requests/2021/4
6. https://megalinter.io

pointless-statement / W0104

Message emitted:

`Statement seems to have no effect`

Description:

Used when a statement doesn't have (or at least seems to) any effect.

Problematic code:

```
[1, 2, 3]  # [pointless-statement]
```

Correct code:

```
NUMBERS = [1, 2, 3]

print(NUMBERS)
```

FIGURE 3.2 – Documentation de la règle *pointless-statement* de `Pylint`

un *Message emitted* (message d'erreur lorsque la règle est enfreinte), une *Description*, un *Problematic code* et un *Correct code*. Si nous voulons maintenant comparer cette documentation avec une autre règle, nous devons aligner leurs concepts de documentation. Cet alignement révélera, par exemple, quels concepts sont présents dans les deux documentations, ou quels sont ceux présents dans une seule règle.

Nous utilisons un processus en deux étapes pour d'abord coder puis uniformiser les concepts de documentation. Plus précisément, nous construisons un codage indépendant pour chaque linter de manière itérative, puis nous les uniformisons en un seul codage global pour la documentation des règles.

Notre objectif est de remplir un tableau qui contiendra sur les lignes l'ensemble des règles analysées, et sur les colonnes tous les éléments que nous pouvons trouver dans une documentation. Si une règle r contient l'élément E dans sa documentation, nous indiquerons un marqueur X au croisement de r et E, sinon nous laisserons la case vide. Initialement, le tableau est complètement vide et ne contient encore aucune colonne. Pour chaque linter, nous choisissons ensuite des règles au hasard et nous examinons la documentation à la recherche des différents éléments qui la constituent. Dès que nécessaire, nous ajoutons une nouvelle colonne pour chaque élément de documentation qui n'était pas présent dans les règles précédentes. Au fur et à mesure du parcours des règles, le nombre de colonnes commence ainsi à s'étoffer.

Chaque linter fournit de nombreuses règles ; dès lors, nous ne pouvons pas analyser l'ensemble de toutes les règles de tous les linters choisis. Pour pallier à ce problème, nous répétons ces étapes jusqu'à atteindre saturation, c'est-à-dire analyser 5 règles successives sans découvrir de nouveau concept pour un même linter. En utilisant ce processus, nous analysons au minimum 6 règles par linter ; dans le Tableau 3.2, la colonne 3 compte les

règles inspectées par linter.

L'uniformisation des concepts de documentation identifiés dans chaque linter a été effectuée manuellement par les membres de l'équipe lors d'une session d'harmonisation. Par exemple, nous considérons que le concept de **Message emitted** dans la règle de `Pylint` de la Figure 3.2 est très similaire au concept de **Message output** identifié dans les règles de `Cppcheck`. Nous uniformisons donc ces deux concepts et les codons en **Error Output**.

Au final, nous obtenons un corpus de 119 règles, dans lequel nous identifions 15 concepts de documentation pour les linters (*cf.*, le Tableau 3.2). La cohérence de la documentation varie : selon l'outil, nous avons besoin de 6 à 10 règles pour atteindre la saturation.

Voici chacun des concepts de documentation que nous avons identifié, associé avec sa définition si nécessaire :

— **Description**

— **Code Example**

— **Further Information** : ressources ou documentation supplémentaires expliquant la règle ou le concept plus en détail, souvent en lien vers des références externes.

— **When Not To Use It** : situations ou contextes où l'application de la règle n'est pas recommandée, souvent en raison d'exigences spécifiques de projet ou de styles de codage.

— **Auto Fix** : indique si le linter peut corriger automatiquement les violations de cette règle dans le code source.

— **Compatibility** : information sur la manière dont cette règle interagit avec différents langages de programmation, cadres de travail ou versions du logiciel.

— **Configurations** : options disponibles pour personnaliser la manière dont la règle est appliquée, telles que le réglage de seuils, exceptions ou comportements spécifiques.

— **Error Output** : le message ou le format de sortie que le linter fournira lorsque cette règle est violée.

— **IDE Fix** : détails sur le support pour la correction automatique des violations au sein d'un IDE.

— **Since** : la version du linter depuis laquelle la règle est disponible.

— **Usage Example** : exemples pratiques montrant comment appliquer ou configurer la règle dans du code réel.

— **Related Rules** : autres règles qui sont similaires ou liées à la règle actuelle, souvent utilisées pour des références croisées dans la documentation du linter.

— **Rule Definition** : détails sur la logique ou l'algorithme utilisé pour détecter les violations.

— **Rule Set** : la collection de règles qui contient celle en cours, souvent organisée par normes de codage ou objectifs spécifiques.

— **Severity** : le niveau d'importance ou d'impact de la règle, pouvant varier de simples suggestions à des erreurs qui doivent être traitées.

TABLEAU 3.2 – Pourcentage de présence de chaque élément pour chaque linter

			Comprehension				**Usage**							**Metadata**			
Langage	**Linter**	**# Règles**	*Code Example*	*Description*	*Further Information*	*When Not To Use It*	*Auto Fix*	*Compatibility*	*Configurations*	*Error Output*	*IDE Fix*	*Since*	*Usage Example*	*Related Rules*	*Rule Definition*	*Rule Set*	*Severity*
C++	OCLint	6	100	100								100			100		
	Cppcheck	6	100	100	83					67				100			100
C#	Gendarme	9	89	100				22	11			33					
	Roslynator	10	100		10				10								100
Java	Checkstyle	6	100	100	67				83	100		100	100			100	
	SpotBugs	6		100	33												
JS	ESLint	10	70	100	40	50	40	20	80		20	100		30	100		
	RSLint	8	100	100					38						100		
PHP	PHP CS Fixer	7	100	100		29	100		29							100	29
	Psalm	7	100	100	14				14								
Python	Pylint	10	100	100	10					100							100
	Flake8	6	100	100	83												
Ruby	RuboCop	9	89	100	56		78		56			100					
	Brakeman	7	71	100	57					43							
Multi	Semgrep	6	100	100	50										100		100
	SonarLint	6	100	100	17												100
	Total	119	104	109	36	7	18	4	26	23	2	34	6	9	30	13	40
	Pourcentage		89	94	33	5	14	3	20	19	1	29	6	8	25	13	33

3.3.3 La Nomenclature

Après avoir identifié les concepts de documentation dans la phase précédente et pour finaliser notre nomenclature, nous avons regroupé ces concepts dans des thèmes pour mieux encadrer leur rôle. Cette classification a été réalisée par les auteurs en suivant une approche similaire à l'*open card sorting* [63]. Nous avons obtenu les trois thèmes suivants :

— **Comprehension**, qui contient tous les concepts identifiant les parties de la documentation qui aident à comprendre la règle.

— **Usage**, qui contient tous les concepts identifiant les parties expliquant comment utiliser et configurer correctement la règle pour un projet donné.

— **Metadata**, qui contient tous les concepts identifiant les informations spécifiques aux linters (telles que le schéma organisationnel ou le code source des règles).

Notre nomenclature est présentée dans le Tableau 3.2 avec 15 concepts de documentation regroupés en trois thèmes. Pour des raisons de lisibilité, nous présentons les résultats par linter, même s'il existe des différences entre les règles au sein d'un même linter. Pour chaque linter et concept, le Tableau 3.2 montre le pourcentage de présence du concept à travers toutes les règles analysées (la version complète par règle est disponible dans notre kit de réplication [7]).

Le premier enseignement est qu'*il n'existe aucun concept de documentation qui se retrouve dans tous les linters*. Les deux concepts les plus courants sont *description* (100% dans tous sauf un linter, `Roslynator`) et *code example* (70–100% dans tous sauf un linter, `SpotBugs`). Cette présence se reflète dans les règles : 109 sur 119 règles ont des descriptions, et 104 sur 119 ont des exemples de code. Nous avons été surpris par l'absence de description pour `Roslynator`, d'autant plus que tous les autres linters ont une description ; pour ce linter, seul le titre de la règle agit comme une description [8]. D'autres concepts sont plus rares, avec 40 règles sur 119 (6 linters) qui incluent une *severity*, et 36 sur 119 qui incluent *further information* (12 linters). Certains concepts sont très rares, comme *IDE Fix* (2 règles sur 119) et *compatibility* (4 sur 119).

Le second résultat est le suivant : à part la description, l'exemple de code, et les concepts d'informations supplémentaires, *peu de linters partagent les mêmes concepts*. Nous notons que le nombre moyen de concepts par linter est de 5,1, avec une médiane de 4,5. `ESLint` est l'exception, avec 11 concepts utilisés, bien que seulement 3 concepts soient utilisés de manière cohérente (*description*, *since*, et *rule definition*).

Le dernier point intéressant est que peu de linters sont cohérents dans leur utilisation des éléments de documentation. Seul `OCLint` est complètement cohérent, tandis que `ESLint` est le plus incohérent.

Résultat #1 : La projection de notre nomenclature sur les règles des 16 linters que nous avons sélectionnés révèle clairement les différences en termes de documentation des règles. Cela renforce le besoin de définir une taxonomie plus abstraite et de mener une enquête auprès des développeurs pour mieux comprendre leurs attentes, ce qui est l'objectif des sections suivantes.

7. https://icpc2024-asats.github.io?page=analysis&tab=nomenclature
8. par exemple, https://josefpihrt.github.io/docs/roslynator/analyzers/RCS0033

3.4 Taxonomie sur les Objectifs et Types de Contenu

Dans cette section, nous souhaitons analyser de manière plus précise le contenu de la documentation des règles. Nous nous concentrons essentiellement sur le thème **Comprehension** de notre nomenclature, car nous nous intéressons en priorité à la compréhension des règles des linters par les développeurs et aux meilleures pratiques qu'elles décrivent. De plus, comme visible sur le Tableau 3.2, ce thème est celui qui est le plus homogène dans les concepts qui le composent.

Le thème **Comprehension** se divise en 4 concepts : *Code Example*, *Description*, *Further Information*, et *When Not To Use It* (*cf.* Tableau 3.2) ; nous nous concentrons donc uniquement sur ces concepts dans la suite. Lors de l'analyse des données, nous nous rendons rapidement compte que ces concepts généraux dissimulent une riche diversité de buts et de types de contenu. Par exemple, le contenu de *Description* met parfois en évidence la raison d'être d'une règle particulière ; dans d'autres occasions, il décrit comment identifier le code qui enfreint la règle. De même, les extraits de *Code Example* peuvent être sous la forme de code conforme, non conforme, ou un mélange des deux. Nous observons également que la documentation est souvent une combinaison entre du texte, des liens et du code source.

La Section 3.4.1 présente la méthodologie que nous suivons pour extraire et consolider ces informations. La Section 3.4.2 décrit la façon dont nous validons en interne la taxonomie résultante (l'enquête dans la Section 3.5 fournit une validation supplémentaire). Enfin, la Section 3.4.3 présente les résultats de l'application de notre taxonomie sur les règles du Tableau 3.2.

3.4.1 Extraction

Pour mieux comprendre quels sont les buts et les types de contenu utilisés dans la documentation des règles des linters, nous employons une méthodologie informelle d'*open card sorting* [63] couplée à un processus de saturation similaire à celui utilisé dans la Section 3.3. L'objectif est double : identifier *comment* le contenu est matérialisé dans la documentation (par exemple, texte, code source, images, etc.) et quel est le *but* du contenu.

Le *card sorting* traditionnel nécessite d'imprimer des fragments du contenu de la documentation sur des cartes et de les regrouper physiquement par thèmes communs. Malheureusement, la grande quantité de contenu présent dans notre échantillon (plus d'une centaine de règles) rend cette méthodologie impraticable.

Pour simplifier le processus, nous passons en revue les règles de manière incrémentale dans un ordre arbitraire, en limitant le processus au contenu pertinent pour le thème **Comprehension**. Nous terminons le processus lorsqu'aucun nouveau thème n'émerge après cinq règles successives.

Nous avons réalisé ce processus lors d'une session collaborative en binôme, au cours de laquelle plusieurs dizaines de règles ont été examinées. À la fin de la session, nous avons obtenu la taxonomie suivante :

- **Objectif**
  - *Quoi* : qu'est-ce qui déclenche l'activation de cette règle et comment reconnaître le code non-conforme ?
  - *Pourquoi* : pourquoi cette règle est-elle importante et pourquoi devrait-elle être appliquée ?
  - *Correction* : comment le code non-conforme devrait-il être corrigé pour se conformer à la règle ?

FIGURE 3.3 – Taxonomie appliquée à la règle *MultipleVariableDeclarations* de `Checkstyle`

— **Type de contenu**
  — *Texte* : texte en prose libre
  — *Code* : code source écrit dans un (des) langage(s) de programmation ciblé(s) par le linter, pouvant inclure de la prose sous forme de commentaires
  — *Lien* : liens vers d'autres documentations, pages web, PDFs, etc.

La Figure 3.3 montre un exemple de règle de `Checkstyle` que j'ai analysée manuellement pour identifier ses buts et types de contenu. Sa description utilise un mélange de *Texte* et de *Lien* pour documenter les buts *Quoi* et *Pourquoi*. Le *Code* documente les buts *Quoi* et *Correction* via des exemples de code conforme et non conforme.

3.4.2 Validation

Nous validons l'exhaustivité et l'objectivité de notre taxonomie en calculant l'accord entre des annotations indépendantes d'un ensemble de règles. Notons qu'il s'agit d'une validation interne de notre taxonomie ; notre enquête (*cf.* Section 3.5) la valide avec des participants externes.

Le premier auteur sélectionne 12 règles parmi les linters précédemment choisis, en veillant à choisir des règles non triviales [9] couvrant les principales catégories de règles de linters identifiées par Vassolo *et al.* [44] : nommage et style, correction, performance et sécurité. Le premier auteur retire ensuite manuellement de leur documentation le contenu lié aux thèmes **Usage** et **Metadata** (lorsque présent), ne conservant que le contenu lié au thème de la **Comprehension**. Enfin, le premier auteur annote les règles en soulignant les éléments qui abordent les objectifs *Quoi*, *Pourquoi* et *Correction*, ainsi que les types de contenu utilisés, comme illustré dans la Figure 3.3.

Les douze règles sont ensuite réparties aléatoirement entre trois des autres auteurs, qui suivent le même processus d'évaluation sur quatre règles chacun. Chaque règle est

9. Un exemple de règle triviale est https://rslint.org/no-await-in-loop/

ainsi annotée par deux évaluateurs indépendants. Nous évaluons notre taxonomie sur ces règles en évaluant *i)* dans quelle mesure les objectifs *Quoi*, *Pourquoi* et *Correction* sont nécessaires et suffisants pour évaluer tout le contenu de la documentation (exhaustivité), et *ii)* dans quelle mesure les évaluateurs indépendants s'accordent de manière fiable sur les objectifs du contenu de la documentation (objectivité).

Exhaustivité. Tous les évaluateurs ont utilisé tous les objectifs sur toutes les règles, et au moins un objectif sur tout le contenu *Texte*, la plupart des *Liens* et la majorité des extraits de *Code*. Certaines parties des extraits de *Code* n'ont pas été évaluées car elles ne se rapportent à aucun objectif mais servent plutôt de squelettes de code pour garantir que les extraits sont syntaxiquement valides (par exemple la déclaration de classe dans la Figure 3.3). Un évaluateur a trouvé que certains *Liens* étaient trop généraux pour être actionnables et que leur intérêt à être dans la documentation de la règle n'était pas suffisamment perceptible (par exemple, un lien vers une liste des dix vulnérabilités les plus critiques dans les applications web). [10]

Objectivité. Nous mesurons l'accord entre les évaluateurs sur le codage des objectifs en utilisant le kappa de Cohen. Cependant, nous ne mesurons pas l'accord sur le type de contenu (texte, code ou lien) qui n'est pas subjectif. Le kappa de Cohen convient à cette situation car il y a deux évaluateurs par mot, et chaque mot est évalué avec un seul objectif. Cependant, pour le *Code* et les *Liens*, le même lien ou extrait de code peut être associé à plusieurs étiquettes. Nous utilisons donc le kappa de Fleiss pondéré, mieux adapté, avec un poids calculé en utilisant la distance de MASI entre les ensembles d'étiquettes [64].

— *Texte* : nous obtenons des valeurs de kappa de 0,3, 1 et 0,741 entre le premier auteur et les trois autres évaluateurs, indiquant un accord plutôt fort sur le contenu textuel, sauf avec un évaluateur. L'évaluateur avec la valeur d'accord la plus basse a effectué son évaluation au niveau de la phrase, tandis que les autres évaluateurs ont codé au niveau plus détaillé des mots individuels, ce qui explique l'accord plus faible.

— *Code* : les valeurs de kappa sont de 1, 0,71 et 1, indiquant un accord très fort sur ce type de contenu.

— *Liens* : nous obtenons des valeurs de kappa de 0,04, 0,33 et 0,14, suggérant un faible accord entre les évaluateurs pour ce type de contenu. Les désaccords sont dus à : *i)* certains évaluateurs n'ont évalué qu'en utilisant le contexte dans lequel le lien était utilisé, sans regarder son contenu, pour effectuer l'évaluation ; *ii)* certains évaluateurs n'ont attribué aucune évaluation à des liens très généraux, tandis que d'autres ont attribué toutes les évaluations possibles.

Puisque l'accord est globalement élevé, avec des raisons claires pour les désaccords que nous observons, nos résultats indiquent que notre taxonomie convient pour classer le contenu de la documentation des linters.

3.4.3 Résultats

Le premier auteur applique la taxonomie aux 119 règles du Tableau 3.2 pour mettre en évidence la présence des différents types de contenu et objectifs de la documentation. Compte tenu des problèmes d'objectivité identifiés dans la Section 3.4.2, le premier auteur applique une stratégie fine pour évaluer le contenu *Texte* (évaluant au niveau du mot)

10. https://owasp.org/www-project-top-ten/

et une stratégie optimiste pour évaluer les *Liens* (pouvant attribuer plusieurs objectifs). Le Tableau 3.3 montre le pourcentage de règles, pour chaque linter, qui documentent les objectifs *Quoi*/ *Pourquoi*/ *Correction* pour *Texte*, *Code* et *Liens*.

Objectifs. Sur les 119 règles analysées, 119 (100%) documentent l'objectif *Quoi*, 60 (50%) documentent l'objectif *Pourquoi* et 92 (77%) documentent l'objectif *Correction*, quel que soit le type de contenu. En détaillant par type de contenu, nous voyons :

— *Quoi* : 110 règles le documentent avec *Texte* (92%), 108 avec *Code* (91%), et 69 avec *Liens* (58%).

— *Pourquoi* : 60 règles le documentent avec *Texte* (50% global, 100% lorsqu'il est présent), 6 avec *Code* (5% global, 10% lorsqu'il est présent), et 14 avec *Liens* (12% global, 23% lorsqu'il est présent).

— *Correction* : 59 règles le documentent avec *Texte* (50% global, 64% lorsqu'il est présent), 82 avec *Code* (69% global, 90% lorsqu'il est présent), et 8 avec *Liens* (7% global, 9% lorsqu'il est présent).

Types de contenu. *Texte* est présent dans 110 des 119 règles (92%), *Code* dans 108 (91%), et *Liens* dans 70 (59%). En détaillant par objectif, nous voyons :

— *Texte* documente l'objectif *Quoi* dans 100% des cas, le *Pourquoi* dans 55% des cas, et le *Correction* dans 54% des cas.

— *Code* documente l'objectif *Quoi* dans 100% des cas, le *Pourquoi* dans 6% des cas, et l'objectif *Correction* dans 76% des cas.

— *Liens* documente l'objectif *Quoi* dans 99% des cas, le *Pourquoi* dans 20% des cas, et le *Correction* dans 11% des cas.

Par outil. Comme vu dans la nomenclature, il y a une certaine variabilité. Alors que tous les linters documentent l'objectif *Quoi*, un linter ne documente pas l'objectif *Correction* (OCLint) et certains le documentent rarement (Cppcheck, Psalm). Enfin, certains linters ne documentent pas l'objectif *Pourquoi* (Roslynator, PHP CS Fixer) et certains le documentent rarement (Psalm, Pylint, Flake8, OCLint).

Résultat #2 : La documentation des linters a trois objectifs principaux : tandis que le *Quoi* est systématiquement documenté, la *Correction* est souvent documentée (77%), et le *Pourquoi* est documenté seulement la moitié du temps (50%). Le *Quoi* est documenté avec *Texte* (92%) et *Code* (91%), le *Pourquoi* avec *Texte* (100%), et la *Correction* avec *Code* (90%) et *Texte* (64%).

TABLEAU 3.3 – Pourcentage de présence de chaque objectif selon le type de contenu pour chaque linter

			Texte			**Code**			**Lien**		
Langage	**Linter**	**# Règles**	*Quoi* (%)	*Pq* (%)	*Corr* (%)	*Quoi* (%)	*Pq* (%)	*Corr* (%)	*Quoi* (%)	*Pq* (%)	*Corr* (%)
C / C++	OCLint	6	100	33		100			100		
	Cppcheck	6	100	83		100		17	83	83	
C#	Gendarme	9	100	67	78	89	22	89			
	Roslynator	10	10			100		90	10		
Java	Checkstyle	6	100	50	50	100		100	100		
	SpotBugs	6	100	83	100				33	17	
JS / TS	ESLint	10	100	80	90	100	10	100	100		
	RSLint	8	100	100	50	100	36	75	100		
PHP	PHP CS Fixer	7	100		57	100		100			
	Psalm	7	100	14	14	100		29			14
Python	Pylint	10	100	10	40	80		80	100		
	Flake8	6	100	17	83	100		100	83		
Ruby	RuboCop	9	100	44	33	100		89	44		
	Brakeman	7	100	86	57	71		14	71	57	43
Multi	Semgrep	6	100	67	67	100		67	100	50	50
	SonarLint	6	100	100	83	100		100	17	17	17
Total		119	110	60	59	108	6	82	69	14	8

3.5 Enquête

Dans cette section, nous souhaitons évaluer si la documentation des règles d'un linter répond aux attentes des développeurs les utilisant. Plus précisément, nous voulons observer si les objectifs documentés dans les règles et leur incarnation en *Texte*, *Code* et *Liens* satisfont les développeurs qui les rencontrent. Également, nous souhaitons valider auprès des développeurs la pertinence de notre taxonomie.

Pour répondre à ces questions, nous concevons une enquête sous forme de questionnaire anonyme comprenant un mélange de questions ouvertes et fermées, partagée avec des partenaires industriels et des collègues chercheurs. La Section 3.5.1 présente la conception de notre enquête et la Section 3.5.2 donne un aperçu sur les participants et sur la méthodologie d'analyse. La Section 3.5.3 détaille les résultats quantitatifs obtenus pour les questions fermées et la Section 3.5.4 les résultats qualitatifs émergeant des questions ouvertes. L'enquête, les réponses et les graphiques que nous discutons dans cette section sont disponibles sur une page web interactive. [11]

3.5.1 Conception de l'Enquête

L'enquête commence par un message de bienvenue détaillant ses objectifs, auteurs, temps de réalisation estimé et politique de données. Le reste de l'enquête s'articule autour de quatre parties : profil du développeur, évaluation de la taxonomie, analyse des règles et retour global. Le Tableau 3.4 montre les questions de l'enquête. Le questionnaire étant posé en anglais aux participants, nous avons laissé les questions originales.

11. https://icpc2024-asats.github.io?page=survey

TABLEAU 3.4 – Les questions de l'enquête.

	Question	**Type**	**Obligatoire**
Profil du développeur	What is your experience as a developer?	Choix unique	✓
	Which of the following languages do you use regularly?	Choix multiples	✗
	Do you know what a linter is?	Oui/Non	✓
	Do you use a linter on some of your projects?	Oui/Non	✗
	Which of the following linters were used in those projects?	Choix multiples	✗
Évaluation de la taxonomie	Rate the usefulness of each purpose in the documentation of a linter	Choix unique pour chaque objectif	✓
	For the *What* purpose, why do you think it is (not) important to be present in the documentation?	Réponse ouverte	✗
	For the *Why* purpose, why do you think it is (not) important to be present in the documentation?	Réponse ouverte	✗
	For the *Fix* purpose, why do you think it is (not) important to be present in the documentation?	Réponse ouverte	✗
	Do you think that there are other purposes that a linter documentation should have?	Réponse ouverte	✗
Analyse des règles	Have you ever seen this rule?	Oui/Non	✓
	For the rule and taxonomy provided, evaluate for each type of content its importance to explain the *What* purpose	Choix unique pour chaque type	✓
	For the rule and taxonomy provided, evaluate for each type of content its importance to explain the *Why* purpose	Choix unique pour chaque type	✓
	For the rule and taxonomy provided, evaluate for each type of content its importance to explain the *Fix* purpose	Choix unique pour chaque type	✓
	For the rule and taxonomy provided, indicate your satisfaction level on the quality of the documentation for each purpose	Choix unique pour chaque objectif	✓
Retour global	Please comment freely on the linters documentation you saw : what you appreciated, disliked, and how it compared with your expectations.	Réponse ouverte	✗

Profil du développeur Pour établir le profil de nos participants, nous demandons leur expérience en tant que développeurs et les langages de programmation qu'ils utilisent régulièrement. Les participants peuvent choisir entre quatre groupes : *Novice* (0 à 4 ans d'expérience), *Junior* (5–9), *Confirmé* (10–19) et *Senior* (20+). Ils sélectionnent ou saisissent leur(s) langage(s) de programmation préféré(s) à partir d'une liste ouverte. Comme notre étude se concentre explicitement sur les linters, nous interrogeons les participants sur leur expérience avec les linters. La première question demande s'ils savent ce qu'est un linter, la deuxième s'ils utilisent des linters dans certains de leurs projets, et la dernière quels linters ils utilisent (choisis parmi les 16 linters précédemment présentés dans cette étude ou saisis manuellement).

Évaluation de la taxonomie Dans cette partie, nous présentons aux participants une capture d'écran de la règle `FetchEnvVar` de RuboCop [12] qui sert d'illustration à la terminologie que nous utilisons dans l'enquête (linter, règle, code conforme, code non conforme). Ensuite, nous introduisons les participants aux termes de notre taxonomie et leur définition : objectifs (*Quoi*/*Pourquoi*/*Correction*) et types de contenu (*Texte*, *Code*, *Liens*). Nous demandons ensuite aux participants d'évaluer l'importance de chaque objectif dans la documentation des règles de linters. Nous utilisons une échelle de réponse asymétrique inspirée par Kano *et al.* [65] et adaptée par Begel *et al.* pour le génie logiciel [66] : *Essential*, *Worthwhile*, *Unimportant*, *Unwise*, *I don't understand*. Pour chaque objectif, nous incluons une question ouverte supplémentaire demandant pourquoi sa présence dans la documentation est ou n'est pas importante. La dernière question ouverte demande aux participants si les linters devraient documenter des objectifs supplémentaires, que nous aurions pu manquer dans notre taxonomie.

Analyse des règles Dans cette étape, les participants doivent évaluer la documentation de règles issues de linters. Pour chacune des 12 règles utilisées pour valider notre taxonomie dans la Section 3.4.2, nous créons une page spécifique dans l'enquête. La page comprend une capture d'écran de la documentation de la règle, un lien vers sa documentation officielle et une série de questions concernant sa qualité. Comme notre objectif n'est pas d'évaluer la capacité des participants à annoter la règle avec notre taxonomie, la capture d'écran inclut les informations concernant ses objectifs et types de contenu (comme indiqué dans la Figure 3.3). Ces informations ont été établies par quatre des auteurs de cette contribution.

La première question demande si le participant connaît déjà la règle. Ensuite, pour chaque objectif et chaque type de contenu, une question demande d'évaluer l'importance de ce type de contenu pour documenter cet objectif spécifique en utilisant l'échelle asymétrique introduite ci-dessus. Enfin, la dernière série de questions demande au participant de juger la qualité globale de la documentation pour chaque objectif, en utilisant une échelle symétrique pour mesurer la satisfaction : *Very satisfied*, *Satisfied*, *Neither satisfied nor dissatisfied*, *Dissatisfied*, *Very dissatisfied*. Les participants peuvent répondre *Not present* à toute question, indiquant qu'il n'y a pas de contenu du type approprié ou que la règle ne documente pas l'objectif. La réponse *Not present* sert également de contrôle de qualité, comme montré plus tard dans la Section 3.5.2.

Lorsque les participants ont terminé les deux premières parties de l'enquête, la page d'une règle parmi les 12 est tirée au sort et affichée. Ils peuvent ensuite choisir d'analyser une autre règle, tirée au sort parmi les restantes, jusqu'à ce qu'il n'y ait plus de règle à

12. https://docs.rubocop.org/rubocop/cops_style.html#stylefetchenvvar

examiner. Lorsqu'un participant a évalué toutes les règles ou choisi de ne pas continuer, il est redirigé vers la dernière partie de l'enquête.

Retour global Cette dernière partie consiste en une unique question ouverte demandant aux participants de commenter la documentation des règles qu'ils ont évaluées. Cette question est conçue pour mettre de côté notre taxonomie et inviter les participants à partager plus librement leurs opinions : ce qu'ils ont aimé et ce qu'ils n'ont pas aimé concernant les règles et leur documentation, et comment cela se compare à leurs attentes.

3.5.2 Participants et Méthodologie

Nous avons principalement partagé l'enquête avec des partenaires industriels et des collègues chercheurs par contact direct et via des listes de diffusion. Nous l'avons également partagé sur les réseaux sociaux et professionnels (Twitter, LinkedIn, Slack). L'enquête était disponible en ligne sur une instance LimeSurvey auto-hébergée du 4 juillet au 19 octobre 2023. Au total, nous avons reçu 179 réponses anonymes. Les participants ont quitté à différents stades : 179 ont renseigné leur profil de développeur, 119 ont évalué notre taxonomie, 85 ont évalué au moins une règle (pour un total de 289 évaluations de règles) et 26 ont répondu à la dernière question ouverte.

Notre méthodologie pour nettoyer les données et analyser les réponses est la suivante. D'abord, nous nettoyons les réponses et tentons de retirer le bruit. Comme mentionné plus tôt, les participants peuvent répondre *Not present* lorsqu'un type de contenu ou un objectif donné manque dans la documentation de la règle qu'ils évaluent. Nous observons que, dans certains cas, des participants ont marqué certains types de contenu ou objectifs comme *Not present* alors qu'ils étaient présents et marqués comme tels dans la capture d'écran. Par exemple, certains participants ont répondu que l'objectif *Correction* pour le *Code* n'était pas présent dans la capture d'écran de Figure 3.3. Dans ce cas, nous écartons les réponses du participant liées à cet objectif pour la règle donnée, pour tous les types de contenu. Nous appliquons le même filtre lorsqu'un participant fournit une évaluation pour un objectif qui n'est pas présent dans la règle qui lui est présentée. Nous avons obtenu finalement, après application des filtres, 225 évaluations pour l'objectif *Quoi*, 91 pour l'objectif *Pourquoi*, et 161 pour l'objectif *Correction*.

Ensuite, nous utilisons l'analyse thématique [67] pour extraire des codes à partir des réponses aux questions ouvertes. Deux auteurs lisent ces réponses et attribuent des codes. Ensuite, les quatre premiers auteurs se réunissent pour harmoniser les codes sous des thèmes de niveau supérieur, discutés dans la Section 3.5.4. Globalement, nous avons obtenu 33 réponses à la question demandant si les règles des linters devraient documenter d'autres objectifs, 56 réponses à la question demandant d'évaluer l'importance de chaque objectif dans la documentation (168 au total), et 26 réponses à la question de *Retour global*.

3.5.3 Analyse Quantitative

Dans cette section, nous examinons les réponses aux questions fermées. Nous incluons seulement les réponses des 85 participants qui ont évalué au moins une règle.

Profil du développeur La distribution des participants en termes d'expérience est assez équilibrée. Parmi les 85 participants, 37 ont moins de 5 ans d'expérience en tant que développeur (novices), 22 ont entre 5 et 9 ans (juniors) et 26 ont plus de 10 ans (seniors).

Le langage de programmation le plus utilisé est le C/C++ (55%), suivi de près par Python (54%), JavaScript et TypeScript (44%) et Java (42%). Les autres langages sont utilisés par moins de 15% des répondants.

Une grande majorité (81%) des participants sait ce qu'est un linter ; la proportion augmente avec l'expérience (73% des novices, 82% des juniors et 92% des seniors). La même tendance est observée pour l'utilisation des linters : 65% des participants utilisent ou ont utilisé des linters dans leurs projets (46% pour les novices, 68% pour les juniors et 88% pour les seniors). Il y a également un déséquilibre notable dans l'utilisation des linters en fonction des langages de programmation utilisés : 53% des développeurs C/C++ utilisent un linter, 61% des développeurs Python, 64% des développeurs Java et 81% des développeurs JavaScript et TypeScript. Une explication plausible est que ESLint est souvent inclus par défaut lors de l'initialisation des projets JavaScript et TypeScript. Lorsque les participants utilisent des linters dans leurs projets, le plus populaire est en effet ESLint (41%), suivi de Pylint (21%) et SonarLint (19%). Les autres outils sont utilisés par moins de 10% des participants. Par ailleurs, trois répondants mentionnent des linters supplémentaires qu'ils utilisent — clang tidy, Fortify et OCaml platform — indiquant que notre sélection de linters reflète ceux majoritairement utilisés dans la pratique. L'utilisation globale des linters reflète l'utilisation des langages, à l'exception du C/C++ : comme les développeurs utilisent plus d'un langage, les utilisateurs de C/C++ tendent à utiliser des linters avec d'autres langages de programmation.

Évaluation de la taxonomie Dans cette partie, nous avions demandé aux participants de juger l'utilité des objectifs *Quoi*, *Pourquoi* et *Correction* dans la documentation des linters, indépendamment de toute règle particulière. La Figure 3.4a montre les résultats : les participants s'attendent fortement à ce que chacun des objectifs soit présent et documenté. Alors que les objectifs *Quoi* et *Pourquoi* sont jugés essentiels par une majorité, l'objectif *Correction* semble légèrement moins essentiel pour les participants, mais reste néanmoins utile. En particulier, nous notons l'importance de l'objectif *Pourquoi* pour les participants, ce qui indique que la documentation des règles devrait non seulement documenter le problème et sa solution, mais aussi la raison motivant la règle ; en contraste, *seulement la moitié des règles* que nous avons analysées avaient une raison documentée (*cf.* Tableau 3.3). Lorsque nous regroupons les réponses par expérience de développeur, langage de programmation ou autres critères de profil, nous n'observons pas de différences majeures.

Analyse des règles Les 85 participants ont analysé un total de 289 règles, avec une moyenne de 3,4 règles par participant. Chacune des 12 règles a été évaluée par 19 à 29 participants différents. La Figure 3.4b montre comment les participants évaluent l'importance de chaque type de contenu pour documenter chaque objectif dans les règles qu'ils ont examinées. Si une règle n'inclut pas un type de contenu particulier pour documenter un objectif donné, et que le participant le marque comme *Not present*, nous omettons ce point de données car il ne transmet aucun jugement positif ou négatif. Ceci explique pourquoi il n'y a pas d'évaluation du *Code* pour documenter l'objectif *Pourquoi*, puisqu'aucune des 12 règles ne le documente avec du code.

Les participants évaluent positivement l'importance de l'objectif *Quoi* (*Essential* ou *Worthwhile*), quel que soit son type (94% pour *Texte*, 88% pour *Code* et 68% pour *Liens*). Alors que le *Texte* et le *Code* sont considérés comme essentiels, les *Liens* sont principalement jugés comme utiles. Pour l'objectif *Pourquoi*, les participants évaluent principalement *Texte* comme essentiel (69%) et *Liens* comme utile (68%). Pour l'objectif *Correction*,

les participants évaluent principalement *Texte* (59%) et *Code* (60%) comme essentiels, et *Liens* comme utiles (48%).

Les participants évaluent très positivement *Texte* et *Code* pour expliquer les trois objectifs. Cela suggère qu'une combinaison de texte et de code source pourrait être le meilleur choix pour documenter les règles des linters. Les participants évaluent également positivement *Liens*, avec un minimum de 68% d'évaluations positives pour chaque objectif. Cependant, la plupart des participants les évaluent seulement comme *Worthwhile* plutôt que *Essential*. De plus, une portion significative les juge comme *Unimportant* : trois fois ou plus que *Texte* ou *Code*. Une raison possible est que l'ouverture de ressources externes interrompt le flux de lecture et que des informations importantes peuvent être noyées parmi les autres informations, en particulier si le lien renvoie à des documents plus larges. Une solution pourrait être d'extraire et d'afficher les informations importantes des ressources externes dans la documentation, et de les citer comme source.

Résultat #3 : Le texte et le code source sont les mieux adaptés pour documenter les objectifs *Quoi* et *Correction*, et une bonne documentation pour une règle d'un linter devrait inclure les deux. *Texte* est le support le mieux adapté pour documenter l'objectif *Pourquoi*. Les *Liens* sont rarement considérés comme essentiels et devraient être utilisés avec parcimonie.

Enfin, la Figure 3.4c affiche la satisfaction des participants concernant la qualité de la documentation des règles qu'ils ont évaluées, pour chaque objectif. Nous observons que les participants sont principalement satisfaits de la qualité de la documentation pour les objectifs *Quoi* (84%) et *Correction* (70%). Cependant, nous observons que près d'un quart des participants ne sont pas satisfaits de la qualité de la documentation pour l'objectif *Pourquoi*. Pire, dans 13% des cas, les participants sont très insatisfaits de sa qualité. Cela contraste fortement avec l'évaluation de l'utilité émise par les participants dans la Figure 3.4a.

Résultat #4 : Les participants expriment un fort intérêt à comprendre la raison derrière les règles d'un linter lors de la lecture de leur documentation (*Pourquoi*). Pourtant, le **Résultat #1** indique que seulement 50% des règles d'un linter documentent l'objectif *Pourquoi*. De plus, lorsqu'il est présent, les participants sont insatisfaits de la manière dont il est documenté dans près de 25% des cas. Clairement, *les documentations devraient inclure et expliquer de manière plus claire l'objectif* Pourquoi.

3.5.4 Analyse Qualitative

Dans ce qui suit, nous résumons les commentaires libres basés sur notre codage. Les codes mentionnés pour la première fois incluent leur fréquence **comme ceci (0)**. Les codes référencés après avoir été introduits une fois sont *comme ceci*. Nous ne rapportons pas les codes « évidents » (par exemple, **comprendre la raison (29)** pour le but de comprendre le *Pourquoi*). Les citations issues des réponses apportées par les participants sont *"sous cette forme"*. Nous laissons d'ailleurs ces citations dans leur langue originale.

Thème transversal : apprentissage Un des thèmes les plus importants, à travers les trois objectifs (**apprendre-*Quoi* (11)**, **apprendre-*Pourquoi* (21)** et **apprendre-*Correction* (13)**), est *l'apprentissage* : 45 commentaires ont abordé ce thème d'une manière ou d'une

autre. Cela est parfois exprimé comme l'auto-amélioration des compétences du développeur, notamment pour les débutants ou (plus occasionnellement) pour l'intégration de nouveaux membres dans l'équipe. Les commentaires sur l'aspect *Quoi* mentionnent le besoin de comprendre l'erreur pour ne pas la répéter (améliorant ainsi les compétences), ainsi que les aspects liés à l'équipe (*"because it facilitates the integration of new members"*). Les commentaires sur le *Pourquoi* sont les plus réguliers. Il est essentiel d'expliquer la raison d'une erreur pour comprendre son but et son importance, et également pour la retenir : *"If I don't know why, then I don't know why it's a bad thing and I cannot improve as a developer"*. Enfin, concernant la *Correction*, une fois le problème connu et compris, l'apprentissage des solutions possibles est précieux : *"It is likely the coder introduced a bad pattern/error due to lack of expertise; as such, it would be wrong to assume he/she will know how to fix it"*.

Thème transversal : gain de temps Les répondants ont souligné l'efficacité dans les trois aspects (**gagner du temps-*Quoi* (2)**, **gagner du temps-*Pourquoi* (3)**, particulièrement **gagner du temps-*Correction* (13)**, **corrections automatiques (4)**). Par exemple, un répondant souhaite *"understand in seconds what a lint is about"*, ce qui nécessite des explications claires et concises. L'absence d'information dans le *Pourquoi* empêche de décider s'il faut agir sur un avertissement (*"I will probably lose time looking it up on the internet. Also I will be less motivated to fix it"*). Puisque les corrections sont les plus actionnables, il y a une demande plus forte pour avoir des solutions standards disponibles pour résoudre rapidement le problème, plutôt que de chercher des informations sur le web ou de demander à des collègues. Devoir chercher l'information est perçu comme augmentant la probabilité qu'un avertissement soit ignoré. Allant plus loin, l'objectif logique est l'automatisation : *"ideal thing is to just click a button to 'autofix' the issue, when available"*.

Aspects spécifiques au *Quoi* Le *Quoi* permet de comprendre ce qui déclenche une règle. Il devrait être rédigé de manière claire et concise pour rendre cela aussi facile que possible (économisant ainsi du temps). Le *Quoi* aide à trouver **où l'avertissement se situe (8)**, ce qui n'est pas toujours évident car *"many different things may be discussed/considered on a single piece of code"*. Une étape clé pour trouver où l'erreur se situe, est de **relier l'erreur à son propre code (12)** : les règles sont soit décrites de manière abstraite, ou, au mieux, avec un **exemple (3)**. Les exemples sont préférés pour cela : *"simpler the exemple [sic], the easier it is to relate to one's code"*. Un autre thème concerne **les faux positifs et négatifs (5)**. Les règles sont mises en œuvre par des heuristiques qui peuvent être imparfaites, surtout dans les cas complexes (par exemple, les expressions régulières). Décrire ce qui déclenche une règle en détail est utile pour lever l'ambiguïté entre les vrais positifs et les faux positifs, ainsi que pour connaître les cas que la règle peut manquer (faux négatifs).

Aspects spécifiques au *Pourquoi* Le *Pourquoi* est crucial pour décider d'agir ou non sur l'avertissement. Les développeurs **analysent les compromis et les risques impliqués (11)** : le *Pourquoi* devrait notamment expliquer la gravité de l'avertissement : *"I can judge whether the criticality justify modifying this piece of code. I may choose to disregard the rule if I judge it not worthwhile"*. Dans certains cas, la **pertinence (9)** de la règle sera remise en question (comme lorsque c'est un *false positives and negatives*, ou lorsqu'il s'agit d'une question de **préférences (5)** personnelles ou d'équipe, plutôt que d'un véritable problème. Ainsi, le *Pourquoi* devrait **motiver et justifier l'effort (10)** qui sera investi dans la correction de l'avertissement. Savoir que l'effort nécessaire pour respecter la règle est

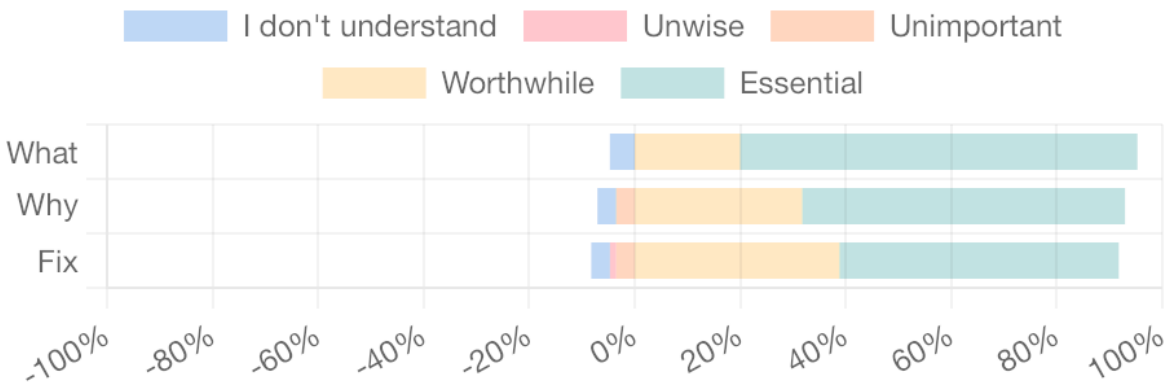
faible (*saving time* via de bons *examples* ou *automated fixes*), fera que celle-ci sera plus considérée.

Aspects spécifiques à la *Correction* Il y a plus de débats quant à l'importance de la *Correction*, comparé au *Quoi* et au *Pourquoi*. Certains répondants affirment que les **corrections sont moins importantes ou pas importantes (7)** pour plusieurs raisons : soit parce que les règles sont simples, les corrections seraient trop basiques, ou parce que le *Pourquoi* et le *Quoi* sont suffisants (*"in most cases previous information should be enough to infer how to fix"*. Pour d'autres répondants, les **corrections sont essentielles ou très importantes (10)**. Pour eux, une documentation sans correction n'est pas actionnable : *"If the Fix is not here, understanding the what and the why lead us to nowhere"*. Les corrections **augmentent la convivialité des linters (5)**, et aident à *saving time*. Les corrections sont particulièrement **utiles quand elles ne sont pas évidentes (4)**, soit en raison d'un manque de connaissance (*learning*) soit en raison de leur difficulté. Fournir des **exemples (10)** est utile pour clarifier les règles complexes. Certains répondants mentionnent que les **corrections peuvent ne pas être optimales (7)** compte tenu du contexte, dans le pire des cas elles peuvent *"lead beginners to apply a cascade of bad decisions made to satisfy the linter"*.

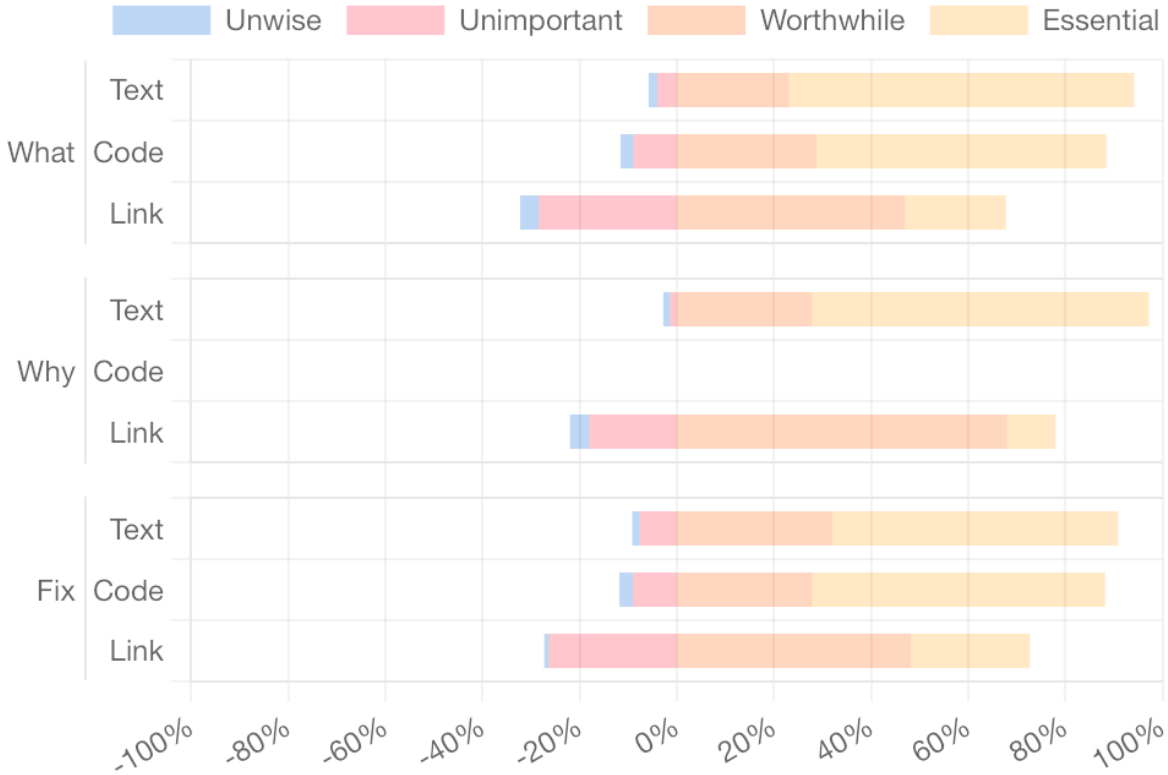
Autres objectifs de la documentation La plupart des répondants ont mentionné **aucun objectif supplémentaire (14)** autre que le *Quoi*, le *Pourquoi*, et la *Correction*. Quelques répondants ont toutefois mentionné d'autres buts de la documentation, tels que **exceptions, alternatives, et limitations (3)**, **risques (3)**, ou **configuration (2)** des avertissements du linter. En effet, nous avons rencontré ces éléments et les avons inclus dans notre nomenclature, mais avons décidé de les exclure du reste de notre analyse car ils concernaient des aspects plus opérationnels de la documentation.

Commentaires supplémentaires sur la documentation Plusieurs répondants ont souligné dans leurs commentaires libres certains sujets qui avaient émergé plus tôt. Plusieurs répondants ont mis en avant le **besoin d'un résumé (7)** (*"Sometimes too much text which does not encourage taking the time to read and therefore deal with the error"*) ou un **modèle structuré (6)** où *"the what, the why and the fix are clearly separated"*. D'autres ont réitéré la nécessité d'**utiliser des exemples de code (4)**, y compris en ajoutant des exemples de code conforme et non conforme, ou d'**éviter les liens (2)** (*"external links are almost never useful"*).

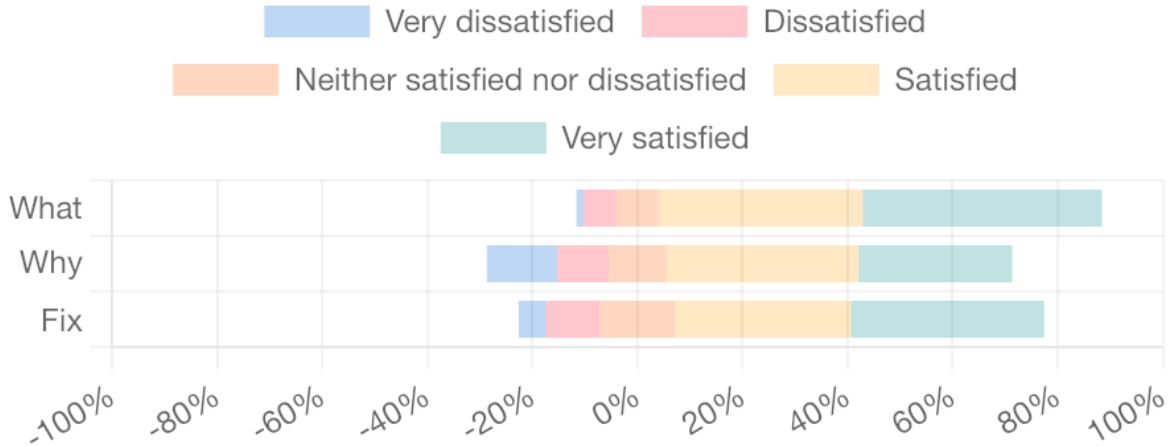
Résultat #5 : La documentation d'un linter a des objectifs d'apprentissage importants (particulièrement le *Pourquoi*), et devrait être rédigée en gardant l'efficacité à l'esprit. Les répondants lisent le *Quoi* pour comprendre l'erreur, et le *Pourquoi* pour décider s'ils doivent appliquer la *Correction*. Les éléments manquants ralentissent le processus, rendant les corrections moins régulières. Les répondants recommandent d'utiliser des résumés, des exemples et d'éviter les liens.



(a) Utilité de chaque objectif dans la documentation



(b) Importance à documenter chaque type de contenu pour chaque objectif



(c) Qualité de la documentation pour chaque objectif

FIGURE 3.4 – Évaluation des participants sur l'utilité, l'importance, et la qualité des types de contenu et objectifs

3.6 Obstacles à la Validité

Validité Externe Notre étude possède plusieurs obstacles en ce qui concerne la validité externe. Le premier obstacle est que notre corpus de linters et de documentation des règles de linters n'est pas représentatif, et est même biaisé en faveur des linters populaires. Par conséquent, nous n'avons aucune garantie que nos résultats se généraliseraient à la population générale, en particulier les pourcentages affichés dans le Tableau 3.2 et le Tableau 3.3. Une autre menace est que les répondants de notre enquête ne constituent pas un échantillon aléatoire de la population des utilisateurs de linter. En conséquence, les résultats que nous avons obtenus avec nos participants pourraient ne pas se généraliser, en particulier les pourcentages affichés dans les Figures 3.4a à 3.4c.

Validité Interne Nous utilisons largement des méthodes qualitatives pour construire notre nomenclature et notre taxonomie (*open card sorting* [63]) et analyser les réponses ouvertes de l'enquête (analyse thématique [67]). Il est bien connu que ces méthodes peuvent être affectées par la subjectivité [68]. En conséquence, différents chercheurs auraient pu obtenir une nomenclature, une taxonomie et d'autres thèmes différents à partir des réponses de l'enquête. Comme mesure d'atténuation pour la taxonomie, nous avons effectué une validation interne et un contrôle de réalité dans l'enquête où nous avons observé qu'elle était bien comprise par les répondants. Pour les autres résultats obtenus à partir de l'analyse qualitative, nous avons systématiquement utilisé des sessions d'harmonisation pour réduire la subjectivité de nos résultats.

En ce qui concerne les réponses des participants à l'enquête, il y a une chance qu'ils n'aient pas pleinement compris les questions que nous avons posées. Cet obstacle affecte principalement les questions fermées où nous ne pouvons pas faire de vérification de la validité en regardant la réponse. Les questions les plus difficiles concernaient l'analyse de l'utilité de chaque type de contenu pour documenter chaque objectif, car les participants devaient analyser attentivement comment nous avions évalué le contenu des règles. Comme vérification de la validité, nous avons inclus un élément d'échelle spécifique (Not present) pour vérifier deux fois que les participants avaient bien compris notre évaluation. Nous avons utilisé cette vérification pour filtrer les réponses incohérentes. Cependant, cette vérification n'est pas parfaite et il est possible que les participants aient répondu à ces questions sans comprendre notre évaluation.

3.7 Conclusion

Dans ce chapitre, nous avons exploré comment les règles de linters sont documentées et les avons confrontées avec les attentes des développeurs via une étude en deux phases. Nous avons d'abord étudié comment plus de 100 règles — couvrant 16 linters dans plusieurs langages de programmation — sont documentées. Cela a conduit à une nomenclature sur les concepts de documentation, raffinée en une taxonomie des objectifs et des types de contenu de la documentation. Nous utilisons ensuite cette taxonomie pour contraster la documentation de 12 règles du monde réel avec les attentes des développeurs via une enquête impliquant 85 répondants qui ont évalué les règles 289 fois. Parmi d'autres résultats, nous soulignons les problèmes avec l'objectif *Pourquoi* : bien qu'il soit considéré comme essentiel par les développeurs pour décider d'agir sur un avertissement, la moitié des règles l'omettent ; de plus, un quart des réponses à l'enquête pointe des problèmes de qualité. Nous découvrons également que des exemples

de code en plus du texte sont attrayants pour documenter les objectifs *Quoi* et *Correction*. Enfin, certains développeurs ont exprimé des préoccupations concernant le gain de temps, menant à la recommandation d'inclure des résumés et de réduire l'utilisation de liens externes qui perturbent le flux de lecture.

3.8 Impact pour Packmind

Cette section permet de discuter l'impact de cette publication pour la compagnie Packmind. Elle est donc rédigée depuis le point de vue de Packmind.

Cette étude a été présentée à l'occasion d'une conférence internationale (ICPC 2024, mi-avril 2024) qui avait lieu seulement quelques semaines avant le rendu de ce manuscrit. Les résultats de cette étude n'ont donc pas encore pu être pleinement analysés afin d'apporter des modifications concrètes à notre solution Packmind. Cependant, ces résultats ont été présentés lors d'une réunion à l'ensemble des collaborateurs Packmind et nous avons pu discuter des possibles évolutions que nous pourrions implémenter.

Le premier changement majeur envisagé concerne la structure de nos pratiques, que ce soit pour l'affichage ou pour la création. La première modification serait de mettre plus en avant la correction d'une pratique. Actuellement, lorsqu'un utilisateur soumet un exemple négatif d'une pratique, il peut également y lier une correction. Cependant, cette correction n'est pas directement accessible depuis la fenêtre d'une pratique. Il est nécessaire de cliquer sur un bouton supplémentaire qui présente un défaut de visibilité. Comme c'est un élément perçu comme crucial par les développeurs, nous envisageons d'afficher cette correction directement dans la fenêtre sans nécessiter une nouvelle interaction. Il nous parait également essentiel de fournir un travail spécifique et important sur la description d'une pratique. Actuellement, il s'agit simplement d'un champ texte dans lequel nous pouvons saisir la description, mais rien n'indique comment bien décrire une pratique. Grâce à notre taxonomie, nous pouvons maintenant guider les auteurs de pratique sur comment rédiger une pratique pour s'assurer de sa bonne compréhension. Pour cela, nous imaginons la division du champ unique de la description en 4 nouveaux champs : un champ permettant de saisir un court résumé de la pratique, et un champ pour chaque objectif de notre taxonomie (*Quoi*, *Pourquoi* et *Correction*). Chaque champ pourra être composé d'un texte fantôme (*placeholder* en anglais) afin de décrire quelles sont les attentes du champs en question. Enfin, il faudra faire évoluer l'affichage de cette description, pour faire apparaître distinctement ce court résumé et les 3 objectifs.

Une seconde amélioration possible porte sur une fonctionnalité, récemment implémentée, qui permet de générer de manière automatique une description pour les pratiques. Cette fonctionnalité repose sur l'utilisation d'un *prompt* envoyé à l'API d'OpenAI [13]. Pour des raisons de confidentialité, le prompt utilisé ne peut pas être communiqué, mais son idée générale est d'envoyer le nom de la pratique et de demander un format spécifique pour la réponse. Notre taxonomie obtenue sur les objectifs de la documentation d'une pratique pourrait être intégrée à ce prompt afin que la description générée soit plus précise et conforme aux attentes des développeurs. Nous pourrions, par exemple, demander à ce que la description résultante soit divisée en trois parties pour le *Quoi*, *Pourquoi* et *Correction* en détaillant chacun des objectifs.

Enfin, une dernière suggestion serait de pousser les équipes à écrire des pratiques de bonne qualité en les accompagnant de manière plus précise sur les attentes. Nous pourrions mettre en place un système de paliers à atteindre pour qu'une pratique soit considé-

13. https://openai.com/product

rée comme correctement rédigée. Le premier palier minimal serait d'avoir juste un titre et un exemple, puis un palier suivant serait d'ajouter une courte description, jusqu'à obtenir une pratique qui soit composée de tous les éléments possibles. Nous aurions la possibilité de calculer un score global sur la qualité de toutes les pratiques présentes, et ainsi encourager les auteurs de pratique à maintenir un haut degré de qualité. Afin de juger de la qualité du contenu d'une pratique, en plus de sa structure, nous pourrions également imaginer une revue manuelle à laquelle participeraient les développeurs une fois par semaine. Cela permettrait d'avoir un retour rapide sur la qualité perçue des pratiques, et ainsi affiner le niveau de détail nécessaire selon la difficulté d'une pratique, tout en prenant en compte l'expérience globale de l'équipe de développement. Suite à un scénario réalisé en interne chez Packmind, nous pensons même qu'un LLM pourrait être en capacité de réaliser cette revue et de fournir directement les modifications à effectuer. Nous avons fourni une documentation de pratiques à un LLM et nous lui avons demandé de juger la capacité qu'aurait un autre LLM à détecter cette pratique dans du code juste avec cette documentation. Cela permet de construire un indice de confiance sur notre capacité à détecter une pratique de code donnée, mais nous pourrions détourner ce scénario en demandant un indice de qualité sur la rédaction de la pratique.

Cette récente contribution ne nous a pas encore permis de mettre à jour notre solution, mais plusieurs idées sont apparues et devraient participer prochainement à améliorer notre outil Packmind.

Chapitre 4

Apprendre et Détecter les Pratiques de Code

Ce chapitre présente les travaux effectués pour la publication suivante [69] :

> C. Latappy, Q. Perez, T. Degueule, J.-R. Falleri, C. Urtado, S. Vauttier, X. Blanc, C. Teyton, "MLinter : Learning Coding Practices from Examples—Dream or Reality?", 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Taipa, Macao, 2023, pp. 795-804, doi : 10.1109/SANER56733.2023.00092

Sommaire

4.1 Introduction

Comme détaillé plus tôt (*cf.* Section 1.1), les pratiques de code sont essentielles à la qualité logicielle. Certaines d'entre elles sont bien connues et largement partagées par la communauté des développeurs. Elles sont même régulièrement intégrées dans les projets de développement grâce à l'utilisation des linters [12]. Bien que les pratiques internes à un projet gagnent en popularité, elles ne sont cependant pas prises en charge directement par les linters. Régulièrement décrites en langage naturel avec des exemples de code, elles sont destinées à être appliquées par les développeurs, mais finissent souvent par disparaître dans des wikis non maintenus. Avec Packmind, nous visons à démocratiser l'utilisation de ces pratiques de code internes. Pour rendre ces dernières détectables, des outils spécialisés, tels que CodeQL [1] ou Semgrep [2], fournissent, à travers une syntaxe simplifiée, la possibilité de créer des patterns personnalisés. Cependant, la création des patterns nécessite une expertise avancée, rendant leur écriture trop complexe pour être largement utilisés dans les projets logiciels.

Ce manque d'opérationnalisation des pratiques de code internes nuit souvent à leur mise en application par les développeurs. Le code non conforme est ainsi susceptible de se retrouver lors des revues de code et d'être discuté à plusieurs reprises. Cela est particulièrement vrai pour les contributions des développeurs juniors qui doivent encore se familiariser avec les pratiques du projet. Notre rêve est d'apprendre automatiquement les pratiques à partir d'exemples de code conforme et non conforme fournis par les développeurs. Nous faisons l'hypothèse que cette apprentissage automatique permettra aux équipes de développement de concevoir et d'adopter des pratiques de code personnalisées plus efficacement.

L'Apprentissage Automatique (*Machine Learning* en anglais) a donné des résultats prometteurs pour automatiser les tâches de codage, telles que la complétion de code [70, 71]. Une étude précédente a également montré que les pratiques de code pouvaient être apprises avec environ 700 exemples en utilisant des arbres de décision [72]. Cependant, cette quantité représente presque déjà une limite supérieure dans notre contexte industriel. En effet, notre cas d'utilisation contraste avec l'utilisation traditionnelle du ML car l'objectif est d'apprendre des pratiques à partir d'un petit ensemble d'exemples. Dans notre vision, les développeurs doivent fournir eux-mêmes des exemples de code conforme et non conforme. Obtenir un grand ensemble de données d'entraînement représente un énorme défi dans notre contexte puisque nous essayons d'apprendre des pratiques personnalisées qui ne sont généralement pas partagées par une large communauté. À titre d'exemple, sur l'instance SaaS de Packmind et pour toutes les organisations créées, le nombre maximal d'exemples créés pour une seule pratique est de 34, avec la particularité de n'avoir que des exemples négatifs. Cependant, la moyenne est de 1,06 exemple disponible par pratique. Nous estimons pouvoir obtenir, dans le meilleur des cas et nécessitant une campagne coordonnée dans une grosse communauté de développeurs, un millier d'exemples pour une seule règle donnée. Par conséquent, la question de savoir si le ML est une bonne solution pour automatiser la détection de pratiques de code internes avec le moins d'exemples possible, est actuellement une question ouverte. Pour cette raison, ce travail utilisera l'apprentissage par transfert (*fine-tuning* en anglais), une solution permettant de limiter le nombre d'exemples requis. Notre objectif dans ce chapitre est de réaliser une étude de faisabilité afin d'évaluer s'il est possible de créer des classifieurs efficaces avec un ensemble d'apprentissage très restreint. Notre idée est d'abord d'ex-

1. https://codeql.github.com/
2. https://semgrep.dev/

ploiter un linter populaire avec des dizaines de pratiques de code (ESLint) et un vaste ensemble de projets open source avec du code conforme et non conforme. Par la suite, il s'agit d'évaluer dans quelle mesure une technique ML de pointe (CodeBERT [73]), compatible avec l'apprentissage par transfert, peut apprendre les règles du linter avec un budget d'exemples limité.

Nous visons à répondre aux questions de recherche suivantes :

RQ1 **Combien d'exemples sont nécessaires pour apprendre une pratique ?** Notre objectif global est de pouvoir apprendre une pratique, idéalement en utilisant le moins d'exemples possible.

RQ2 **Quels sont les meilleurs exemples de code pour apprendre une pratique ?** Nous pouvons fournir plusieurs types d'exemples de code pour entraîner notre classifieur : des exemples qui ne respectent pas la pratique, des exemples résultant de l'application de la pratique sur eux (code corrigé), et des exemples tiers à la pratique, sans lien avec la pratique. Est-il utile de fournir du code corrigé et/ou du code tiers pour entraîner les classifieurs ?

Nos résultats montrent que les classifieurs générés atteignent une haute précision et un rappel élevé lorsqu'ils sont évalués sur un ensemble de données synthétique équilibré. Cependant, leur application sur des données réalistes, bien que conservant un bon rappel, souffre d'une chute sévère de précision qui entrave leur usabilité.

Ce chapitre est organisé comme suit :

Nous commençons par réaliser l'état de l'art dans la Section 4.2. Ensuite, la Section 4.3 présente la conception globale de notre approche et la façon dont nous comptons réaliser la tâche d'apprentissage avec CodeBERT. La Section 4.4 détaille notre processus pour créer notre jeu de données. La Section 4.5 présente notre protocole expérimental, et la Section 4.6 discute de nos résultats obtenus. Enfin, la Section 4.7 conclut et la Section 4.8 présente l'impact de cette étude pour Packmind.

4.2 État de l'Art

4.2.1 CCFlex

À notre connaissance, seul le travail de Ochodek *et al.* [72] a pour but d'identifier automatiquement les pratiques internes d'entreprises. C'est l'étude qui est la plus proche de ce que nous souhaitons atteindre, et nous choisissons donc de la présenter en détail. L'intérêt de détecter les pratiques internes automatiquement est notamment d'éviter d'allouer des ressources pour écrire manuellement des patterns de détection. Les auteurs ont développé un outil permettant d'identifier automatiquement les violations de code, par l'utilisation de méthodes de ML nécessitant un ensemble d'apprentissage contenant peu d'exemples, tout en étant indépendant du langage et ne nécessitant pas que le code analysé compile. Pour cela, ils se sont basés sur un outil, nommé CCFlex, qu'ils avaient développé dans le but d'effectuer des mesures statistiques sur les logiciels [52]. Ils ont donc adapté cet outil pour détecter les pratiques de code internes.

La première étape consiste à préparer le dataset pour la phase d'apprentissage. Chaque ligne de code est extraite du fichier de base pour être insérée dans un fichier CSV. Si cette ligne correspond à une ligne de code non conforme, elle est préfixée avec un caractère spécial. Ensuite, toutes les lignes sont envoyées dans un ensemble d'extracteurs de caractéristiques. Chaque extracteur calcule une caractéristique de la ligne qui est ajoutée dans

une nouvelle colonne du fichier CSV. Une caractéristique est seulement une information sur la ligne : sa taille, sa position, ou encore le nombre de fois où un token[3] est présent. Enfin, ce fichier CSV final est donné en entrée au classifieur pour que ce dernier puisse s'entraîner. Ce processus de transformation est également appliqué au code qui sera par la suite évalué par le classifieur.

Cette étude étant réalisée dans le cadre d'un partenariat industriel, deux critères imposés ont dicté le choix pour les classifieurs. Le premier critère était de se contenter d'une taille pour l'ensemble d'apprentissage très petite. Cette limite est due au nombre réduit d'exemples disponibles pour une unique pratique de code. La seconde contrainte était que les résultats obtenus par les classifieurs devaient être explicables. Les techniques utilisant les arbres décisionnels étant notamment reconnues pour faciliter l'interprétation des résultats [74], ils ont donc utilisé différentes versions, telles que CART [75] et les Random Forest [76].

Ils ont également rajouté une étape supplémentaire à leur outil, appelée l'Active Learning. L'AL a pour but d'entraîner un modèle en minimisant l'effort nécessaire pour fournir un ensemble d'entraînement de départ, en étiquetant par la suite des instances pertinentes [77]. Il existe plusieurs stratégies pour réaliser la sélection de ces instances, mais ici le choix s'est porté sur la stratégie Query By Committee [78]. Ainsi, lorsque nous souhaitons faire apprendre une pratique de code interne, il suffit de fournir un jeu de données créé à partir d'exemples positifs et négatifs issus de la documentation de cette pratique. Ensuite, nous pouvons profiter des bénéfices de l'AL pour se faire proposer des lignes de code tirées de la base de code, qu'il faut annoter à la main.

L'évaluation de leur outil s'est fait en trois temps avec des objectifs différents à chaque fois. Dans un premier temps, et dans le but de vérifier la validité de leur approche sur un grand ensemble de données, ils ont commencé par apprendre des règles liées aux standards Sun [4] et Google [5] sur trois projets open-source en Java. Ils ont utilisé l'outil d'analyse statique Checkstyle comme un oracle pour pouvoir étiqueter les lignes de code de chaque projet et pour chaque règle disponible. Ces premiers résultats leur ont permis de confirmer que leur approche était capable d'apprendre et de reconnaître des conventions de code, avec des scores de précision et de rappel entre 0,99 et 1,00.

Dans un deuxième temps, ils ont collaboré avec une première entreprise dans le but d'évaluer si l'évolution de la base de code nécessitait l'évolution des exemples utilisés pour entraîner le modèle. Ils ont choisi d'apprendre trois pratiques internes à l'entreprise et d'utiliser différentes versions de la base de code pour mesurer l'impact de l'évolution du code. Dans ce cas, ils n'avaient pas d'oracle pour vérifier la pertinence des résultats de CCFlex. Ils ont donc utilisé une revue manuelle pour évaluer chaque détection. Cette seconde analyse a permis d'apporter quelques nouveaux résultats, parmi lesquels nous retrouvons la nécessité d'utiliser du code ayant les mêmes conventions pour l'ensemble d'entraînement et de validation. Également, les conventions qui évoluent au sein d'une entreprise ont un impact négatif sur l'*accuracy*[6] du classifieur s'il n'est pas mis à jour. En outre, l'ensemble d'entraînement peut nécessiter seulement 300 lignes de code pour détecter de manière satisfaisante des pratiques unilignes. Enfin, cela a permis d'identifier qu'il manquait certains types d'extracteurs de caractéristiques à l'outil CCFlex, et de les implémenter pour la troisième et dernière étape.

Dans un dernier temps, ils ont collaboré avec une seconde entreprise dans le but de

3. Courte chaîne de caractères
4. https://www.oracle.com/technetwork/java/codeconventions-150003,pdf
5. https://checkstyle.sourceforge.io/reports/google-java-style-20170228.html
6. Nous conservons le terme *accuracy* en anglais pour éviter la confusion avec *precision* en français

mesurer quelle était la quantité minimale de données nécessaires pour apprendre 7 pratiques internes. Cette fois, ils avaient un oracle pour pouvoir calculer la précision et le rappel de leurs différents classifieurs. Ils ont utilisé un processus itératif par lequel ils tentaient d'apprendre une pratique avec seulement 100 exemples à l'origine. Tant que la précision ou le rappel étaient inférieurs à 0,90 ou que le nombre d'exemples pour apprendre était inférieur ou égal à 700, ils rajoutaient 100 nouveaux exemples à l'ensemble d'apprentissage.

Bien que les rappels obtenus soient très bons, avec des scores supérieurs à 0,97 pour toutes les pratiques, cela s'est fait au dépend de la précision qui varie de 0,35 à 1,00. Cependant, les trois différentes expériences, réalisées en Java, en C puis en C++, permettent d'affirmer que les outils utilisant des approches de ML peuvent reconnaître des violations de code pour différents langages de programmation.

Leurs résultats permettent également d'anticiper les performances que nous sommes susceptibles d'obtenir selon les caractéristiques nécessaires pour détecter la règle. Cela établit ainsi des stratégies d'apprentissage différentes selon le type de règle rencontrée. La première observation est qu'il est possible d'obtenir un haut rappel avec un ensemble d'entraînement contenant environ 300 lignes de code. La mise en place de l'Active Learning a permis d'atteindre une meilleure accuracy que la fourniture d'exemples de code provenant de référentiels de pratiques ou de la base de code. Les règles qui sont basées sur des propriétés textuelles ou sur des mots clés spécifiques ont tendance à être facile à apprendre, et ce même avec un ensemble d'entraînement ne contenant que 100 lignes. Cependant, les règles qui nécessitent d'avoir un contexte étendu sur plusieurs lignes sont plus difficiles à apprendre.

Enfin, il est important de prendre en considération un dernier enjeu lors de l'utilisation d'un analyseur basé sur le ML : celui de conserver l'outil à jour. Les pratiques peuvent être amenées à évoluer dans le temps. Certes, cela peut poser des problèmes de précision sur l'analyse du code ; en revanche, il est toujours plus rapide de ré-entraîner un modèle pour une règle donnée que de devoir écrire un pattern de détection pour celle-ci.

Ce travail, bien qu'il soit très proche de nos objectifs et de nos méthodes envisagés, possède un obstacle que nous voulons franchir. Nous souhaitons pousser notre expérimentation plus loin en étudiant la performance des classifieurs entraînés sur des ensembles d'entraînement encore plus petits, afin de permettre à des équipes de développement de taille réduite de pouvoir automatiquement identifier leurs pratiques internes.

Dans le reste de cette section, nous discutons d'autres travaux de Machine Learning pour détecter le non-respect des pratiques de code. Nous faisons le choix de séparer les travaux qui se concentrent uniquement sur les *code smells* (ou anti-patterns), des travaux généraux sur la qualité de code.

4.2.2 Détection de Code Smells par le Machine Learning

Un *code smell* représente un marqueur dans le code qui peut indiquer un mauvais choix de design ou d'implémentation [79]. Nous parlons également d'anti-pattern. Un exemple courant est la présence d'une méthode trop longue ; cela peut indiquer qu'elle possède trop de responsabilités et qu'elle peut être difficile à maintenir. La différence majeure qui les différencie des autres pratiques est la méthode utilisée pour les détecter. Dans le cas spécifique des smells, il s'agit simplement d'un ensemble de métriques qui sont calculées sur le code source, puis ces métriques sont comparées à des seuils prédéfinis pour chaque smell à détecter.

Khomh *et al.* [80] ont introduit une approche bayésienne pour répondre à la question de la détection des code smells. Cette méthode est notamment basée sur les *Bayesian Belief Networks*. Dans un travail précédent, Moha *et al.* [81] avaient développé DECOR, un langage dédié pour décrire le code et les smells. Ce langage a permis d'identifier avec succès 4 des smells les plus connus, dont le *Blob* (plus connu sous le nom de *God Class*). Khomh *et al.* se sont basés sur ce travail pour transformer l'approche utilisée en une représentation probabiliste. Pour montrer la faisabilité de leur processus, ils se sont concentrés sur la détection du smell *Blob*. En plus du BNN, ils ont utilisé d'autres méthodes de ML, permettant notamment d'adapter la détection au contexte actuel à l'aide de la conservation des résultats précédents. Les résultats, testés sur deux projets open source, montrent une grande performance pour identifier les classes présentant des signes de *Blob*.

Maiga *et al.* [82] ont quant à eux privilégié une approche utilisant les *Support Vector Machines*. Leur outil, SVMDetect, a été spécialement conçu pour identifier 4 smells différents : *Blob*, *Functional Decomposition*, *Spaghetti Code*, et *Swiss Army Knife*. Ils ont comparé leur étude à DETEX qui est la suite des travaux réalisés par Moha *et al.* [83], en mesurant leurs performances sur 3 projets Java open source. Les résultats ont démontré que SVMDetect atteint une plus grande précision que DETEX dans l'identification des smells, et montrent que les approches SVM semblent surpasser les performances des méthodologies précédentes.

Fontana *et al.* [84, 85] ont réalisé une série de 2 études dans le but d'évaluer l'efficacité de 16 classifieurs différents pour identifier les code smells. Les performances des classifieurs ont encore été mesurées sur 4 smells : *Data Class*, *Blob*, *Feature Envy*, et *Long Method*; et ils obtiennent des scores d'accuracy jusqu'à 95%. Cette étude a permis de mettre en avant que les classifieurs *J48*, *JRip*, *Naive Bayes* et *Random Forest* ont donné les meilleurs résultats, tandis que les méthodes basées SVM semblent être moins performantes.

Cependant, ces résultats ont été remis en question par Di Nucci *et al.* [86] dans une étude de reproductibilité. Cette étude a pointé des problèmes sur les ensembles de données choisis, notamment sur la proportion entre les violations et les non-violations, qui auraient faussé les précisions obtenues jusqu'à 90%. Cet écart est en partie imputable à l'ensemble de données utilisé, qui contenait un tiers d'éléments avec des code smells, soit une proportion nettement plus élevée que celle généralement observée dans les systèmes logiciels. Par exemple, Palomba *et al.* [87] ont identifié que le smell *Blob* était présent dans moins d'un 1% des classes dans le code source de projets logiciels. Un autre problème, soulevé par cette étude, est lié à l'utilisation de la *10-fold cross-validation* sur un ensemble de données équilibré entre code conforme et non conforme. Cela ne reflète ainsi pas la proportion réelle de présence des smells, ce qui peut avoir fausser les phases d'entraînement et de test. Pour cette raison, nous nous fixons pour objectif d'évaluer notre méthode sur des ensembles de données équilibrés et déséquilibrés afin d'éviter ces écueils.

La revue systématique sur les approches de détection de smells réalisée par Azeem *et al.* [88] permet de résumer parfaitement les raisons pour lesquelles nous souhaitons réaliser notre propre étude. Sur les 15 études sélectionnées, seulement 20 smells différents ont été étudiés, dont le smell *God Class* qui est présent dans 73%. Même si cela a du sens puisque ces études étaient concentrées sur les *code smells*, aucune pratique interne d'équipe n'a été étudiée. Également, la détection de ces smells reposant principalement sur des métriques, il n'y a aucune certitude que les méthodes de ML utilisées (*JRip*, *SVM*, *Decision Trees*) se généraliseront à des pratiques diverses. Nous constatons donc, malgré les résultats obtenus par ces approches, qu'elles n'ont pas été évaluées dans les conditions

que nous souhaitons adopter.

4.2.3 Mesure de la Qualité Logicielle par le Machine Learning

Chappelly *et al.* [89] ont exploré l'utilisation de méthodes de machine learning pour détecter des erreurs en C. Pour cela, ils ont exploré trois techniques différentes : *Random Forest*, *Convolutional Neural Networks* et *Recurrent Neural Networks*. Ils ont comparé leurs différents résultats obtenus à ceux de l'outil Parfait, largement utilisé par les développeurs d'Oracle. Malgré des premiers résultats prometteurs, il est apparu que les méthodes de ML utilisées ne permettaient pas encore de remplacer les outils d'analyse statique. En effet, il est apparu que la précision obtenue était trop faible, et ce pour divers facteurs : la non-prise en compte de la sémantique du langage dans le processus d'apprentissage, et la qualité de l'ensemble de données utilisé pour l'entraînement. Cependant, une approche intéressante de leur travail est d'entraîner un modèle de ML pour chaque règle à détecter. En effet, les algorithmes utilisés permettent de classer une donnée dans une classe ou une autre, alors qu'un même morceau de code peut présenter plusieurs bugs.

Mi *et al.* [90] ont étudié le potentiel d'évaluer la lisibilité du code source à l'aide de *Convolutional Neural Networks*. Les travaux précédents, bien que présentant des résultats satisfaisants, nécessitaient la création de caractéristiques très coûteuses en travail manuel. Ils ont donc proposé une nouvelle approche, basée sur des techniques de Deep Learning, permettant d'identifier automatiquement ces caractéristiques directement à partir du code source. Ils se sont notamment appuyés sur 3 *CNN* opérant à différents niveaux de granularité : au niveau des caractères, au niveau des tokens et au niveau des nœuds de l'AST. L'utilisation de réseaux profonds a cependant nécessité un corpus conséquent de code open source déjà labellisé. Pour valider l'efficacité de leur outil, ils ont réalisé une évaluation avec 5 autres modèles classifiant la lisibilité du code. Les résultats révèlent que DeepCRM a surpassé l'ensemble des précédents modèles, avec une accuracy de plus de 83%. Cette étude a également permis de mettre en avant pour la première fois la supériorité du DL dans ce champ d'étude.

À notre connaissance, une seule approche, par Kovacevic *et al.* [91], utilise des techniques qui encodent le code (à savoir code2vec, code2seq, et CuBERT), en plus des métriques classiques, pour détecter les violations de code. Dans les résultats rapportés, CuBERT surpasse toutes les autres combinaisons testées. Ce résultat est cohérent avec notre intuition que les approches basées sur BERT sont bien adaptées pour aborder ce type de tâche d'apprentissage.

Il existe de nombreuses études utilisant des algorithmes de ML pour évaluer la qualité du code. Cependant, aucun travail, à part celui présenté précédemment par Ochodek *et al.* [72], ne se concentre sur la détection de pratiques internes à une équipe de développement. Comme notre objectif est de se concentrer sur ces pratiques, nous souhaitons réaliser une nouvelle étude identifiant un ensemble de règles plus large. Également, comme déjà énoncé, nous souhaitons pouvoir apprendre les pratiques avec un ensemble d'apprentissage très réduit, ce qui n'a encore jamais été fait.

4.3 Notre Outil MLinter

Nous allons maintenant détailler le fonctionnement que nous souhaitons obtenir pour notre outil MLinter. Son but est d'être capable d'apprendre automatiquement les pra-

tiques de code à partir d'exemples fournis par les développeurs.

4.3.1 Fonctionnement Global

Notre problème d'apprentissage des pratiques de code est modélisé comme un problème de classification binaire. Comme Chappelly *et al.* et Ochodek *et al.* l'ont déjà montré, il s'agit d'une approche qui fonctionne [89, 72]. Étant donné une pratique de code, notre objectif est de former un classifieur binaire qui prend une ligne de code comme entrée et produit une réponse *conforme* ou *non conforme* comme sortie. Cette modélisation du problème s'adapte bien à notre contexte puisque nous pouvons ajouter de nouvelles pratiques en formant et en déployant un nouveau classifieur sans affecter ceux existants. Si le besoin se présente, le processus de classification peut être parallélisé pour s'adapter à un plus grand nombre de pratiques de code.

Lorsque les linters détectent une violation dans du code, l'erreur est le plus souvent affichée au niveau d'une ligne de code. Cependant, de nombreuses pratiques de code ne peuvent pas être détectées avec une seule ligne de code comme entrée. Par exemple, la pratique `indent` de `ESLint`, qui assure que le code est correctement indenté, nécessite des informations contextuelles des lignes environnantes. Pour les besoins de notre étude de faisabilité, nous nous concentrons uniquement sur les pratiques de code qui peuvent être identifiées à partir d'une seule ligne de code. La détection des pratiques nécessitant plus de contexte pourra faire l'objet de travaux futurs.

Dans notre contexte, il existe deux grands types de lignes de code qui peuvent être fournies comme exemples d'apprentissage pour une pratique donnée :

— *Code non conforme* : une ligne de code source qui viole la pratique ;

— *Code conforme* : une ligne de code source qui ne viole pas la pratique.

Si nous regardons en détail une ligne de code considérée conforme par un linter relativement à une règle donnée, elle est soit une ligne initialement non conforme qui a été manuellement ou automatiquement corrigée pour se conformer à la pratique - c'est ce que nous appelons une *ligne de code corrigée*; soit il s'agit d'une ligne qui fait partie du reste de l'ensemble de la base de code qui n'est ni corrigée, ni non conforme - ce que nous appelons une *ligne de code tierce*. Du point de vue du modèle que nous entraînerons, celui-ci recevra des lignes de code qui seront non conformes ou conformes. Cependant, dans le contenu, elles auront trois origines possibles : non conformes, corrigées ou tierces. En anglais, ces trois possibilitées sont nommées respectivement *violated*, *fixed* et *extant*.

4.3.2 Le Choix CodeBERT

L'apprentissage par transfert est une méthode d'apprentissage spécifique qui peut permettre de palier au problème du manque de données d'entraînement. Le principe de l'apprentissage par transfert est d'abord de créer un modèle avec un ensemble de données large, dans le but de lui permettre d'acquérir certaines connaissances de base. Par la suite, le modèle est ensuite ré-entraîné sur des données spécialisées afin de le rendre capable de réaliser des tâches plus spécifiques. Plusieurs modèles permettant cette spécialisation existent. Pour en citer quelques-uns, nous pouvons mentionner Global Vectors for Word Representation (GloVe) [92], Word2Vec [93] ou Bidirectional Encoder Representations from Transformers (BERT) [94].

CodeBERT est un modèle d'apprentissage profond basé sur les Transformers [95], créé par Feng *et al.* [73] et spécifiquement conçu pour le code source. CodeBERT repose sur le modèle de langage naturel BERT. Le modèle BERT a surpassé les techniques de pointe

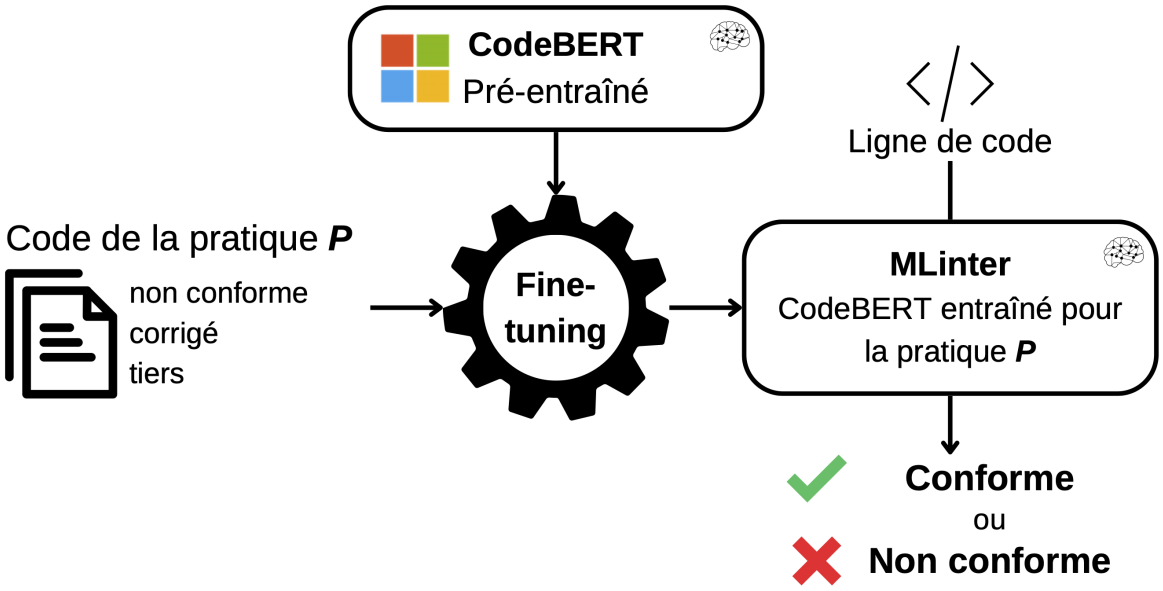


FIGURE 4.1 – Processus pour fine-tuner CodeBERT pour une pratique *P* donnée

dans de nombreuses tâches de traitement du langage naturel (NLP), telles que la prédiction des mots/phrases [94], l'analyse des sentiments [96] et la classification de texte [97]. BERT est un modèle pré-entraîné : son entraînement est effectué sur de grands corpus de documents en langage naturel. Ainsi, BERT peut être utilisé tel quel, ou ajusté en le ré-entraînant sur un nouveau corpus pour créer un modèle spécifique pour un contexte ou une tâche donnée. CodeBERT bénéficie donc de ce même atout pour être adapté comme classifieur via un ré-entraînement.

Dans notre contexte, nous choisissons CodeBERT pour plusieurs raisons. Premièrement, CodeBERT est un modèle travaillant purement avec des informations textuelles brutes et qui n'utilise aucune autre information telle que des arbres syntaxiques abstraits (AST) comme Code2Seq [98] ou Code2Vec [99] qui doivent transformer le code d'entrée en AST. Ainsi, nous pouvons utiliser ce modèle sur des lignes de code source uniques, et cela libère la contrainte de parsabilité sur la ligne. Deuxièmement, CodeBERT est publiquement disponible via le dépôt Hugging Face [7] et peut donc être facilement exploité. Troisièmement, CodeBERT a été pré-entraîné sur plus de six millions de lignes de code écrites dans six langages populaires (Go, Java, **JavaScript**, PHP, Python et Ruby). Ce pré-entraînement permet notamment de requérir moins de données lors de la spécialisation sur nos tâches de classification, par rapport à la quantité nécessaire lors de son entraînement initial. L'utilisation de six langages dans l'entraînement initial de CodeBERT le rend aussi plus généralisé que d'autres modèles entraînés en utilisant un seul langage [100]. Enfin, CodeBERT a montré de bonnes performances pour de nombreuses tâches de génie logiciel, comme par exemple la réparation de programmes [101], la prédiction de tests instables [102], et la prédiction de défauts [103].

4.3.3 Construire MLinter pour une Pratique

La Figure 4.1 illustre le processus que nous suivons pour spécialiser le modèle CodeBERT à détecter les violations d'une pratique de code donnée *P*. Ce processus utilise le modèle CodeBERT de base, qui est facilement téléchargeable. Nous récupérons un en-

7. https://huggingface.co/microsoft/codebert-base

semble de lignes de code *non conformes*, *corrigées* ou *tierces* par rapport à la pratique *P*, que nous annotons finalement en *non conformes* ou *conformes*. Nous fournissons ces différentes lignes de code annotées au modèle CodeBERT, dont il en résulte notre nouveau modèle, MLinter, spécialisé pour détecter les violations de code de la pratique *P*. Ce modèle obtenu est ainsi prêt pour classifier de nouvelles lignes de code source en deux classes : *non conforme* ou *conforme* vis à vis de cette pratique *P*. Ce processus doit être répété pour chacune des pratiques de code qui doivent être apprises.

Nous allons maintenant détailler le processus utilisé afin de créer notre dataset permettant l'apprentissage du modèle CodeBERT.

4.4 Création du Dataset

Afin d'utiliser la méthode d'apprentissage par transfert sur un modèle pré-entraîné et de l'appliquer à notre contexte, nous avons besoin que notre jeu de données contienne des exemples de code non conforme et conforme pour une pratique de code donnée, y compris du code conforme obtenu en corrigeant du code non conforme (*i.e.*, *code corrigé*). Avant de créer notre propre jeu de données, nous avons examiné les jeux de données existants [104, 72]. Cependant, notre besoin de disposer d'exemples de code corrigé n'était jamais satisfait dans les jeux de données analysés. Nous avons donc construit notre propre jeu de données en utilisant un linter sur des projets populaires de GitHub.

Pour cette étude, nous avons choisi d'utiliser le langage JavaScript. Parmi les linters disponibles pour ce langage, nous avons utilisé ESLint car il est le linter JavaScript le plus populaire [12]. C'est également un outil bien documenté et qui a la capacité de corriger automatiquement le code non conforme de certaines règles.

Nous avons procédé en trois étapes successives, détaillées ci-dessous et résumées sur la Figure 4.2 : récupérer les projets JavaScript (*e.g.*, `11ty/eleventy`), configurer ESLint, puis extraire et stocker les résultats. [8]

4.4.1 Sélection des Projets

Nous utilisons les dépôts publics de GitHub pour créer la base de code sur laquelle nous allons exécuter ESLint. En utilisant l'API GitHub, nous récupérons tous les projets publics dont le langage est JavaScript et qui ont au moins 10000 étoiles pour obtenir un jeu de données de taille raisonnable et contenant du code de bonne qualité. Nous obtenons un total de 550 dépôts que nous clonons ❶. Une fois tous les projets clonés, nous filtrons tous les fichiers possédant l'extension `.min.js`. En effet, il est courant pour les développeurs JavaScript de minifier leurs fichiers JavaScript pour accélérer les temps de transfert entre les serveurs et les clients. Cependant, cela réduit considérablement la lisibilité du code car cela implique généralement de raccourcir les noms des fonctions et des variables, et de supprimer la plupart des espaces blancs. De plus, le code minifié n'est pas censé être conforme aux pratiques de code, car il est uniquement utilisé pour le déploiement. Par conséquent, nous excluons un tel code de notre ensemble d'entraînement, car il introduirait du bruit.

8. Tous les scripts utilisés sont documentés et disponibles pour la reproductibilité : https://github.com/labri-progress/MLinter

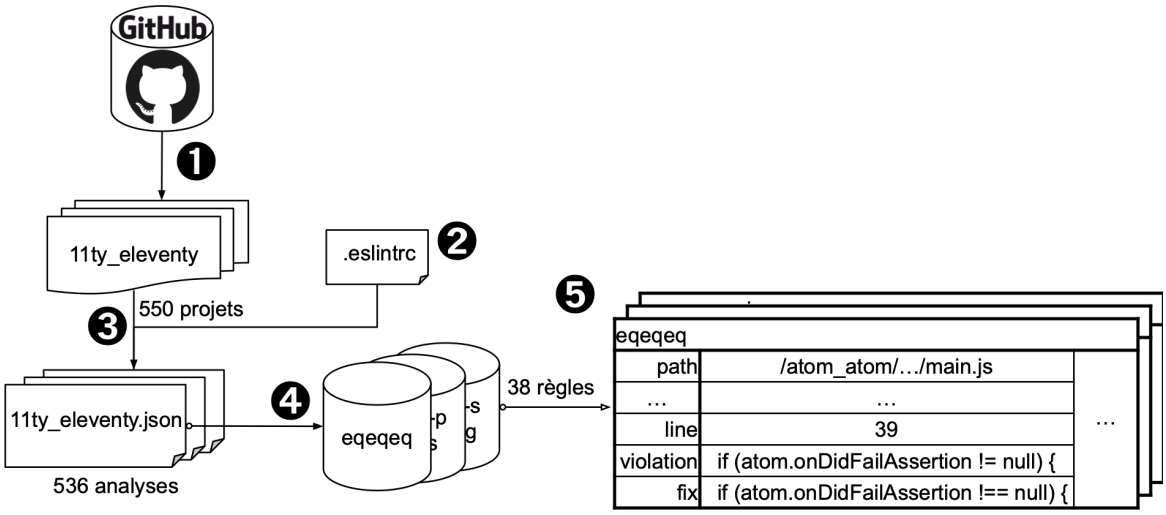


FIGURE 4.2 – Processus pour créer notre dataset

4.4.2 Sélection des Règles

Avant de commencer l'analyse ESLint, nous devons d'abord sélectionner les règles à activer. Puisque nous avons besoin de code non conforme avec le code conforme correspondant corrigé pour chaque pratique, nous ne considérons que les règles corrigibles. Elles représentent précisément 100 règles. Pour cette première étude, nous nous concentrons uniquement sur les règles identifiables sur une seule ligne de code, car ce sera la seule donnée que nous fournirons en entrée au classifieur (*cf.* Section 4.3). Pour se faire, en regardant sa documentation, nous déterminons pour chaque règle si toutes les informations nécessaires à sa détection se trouvent sur la même ligne. Notre configuration ESLint finale a 54 règles activées. Certaines règles nous permettent de configurer des options, résultant en un comportement d'analyse différent. Pour simplifier, nous laissons chaque règle avec les options par défaut ❷.

Nous modifions la configuration du *package.json*[9] de chaque projet en ajoutant (si non présent) une dépendance à ESLint avec notre configuration activée. Enfin, nous exécutons ESLint sur chaque projet en demandant la génération d'un rapport d'analyse au format JSON. Avant d'exécuter le linter, nous avions 550 projets disponibles, mais nous avons obtenu des résultats pour 536 d'entre eux (environ 2,5% de perte). Pour deux projets, l'analyse a échoué en raison d'erreurs du linter. Quant aux autres résultats manquants, ESLint n'a trouvé aucun code non conforme avec la configuration spécifiée. À cette étape, nous obtenons donc 536 fichiers JSON, un par projet, contenant les résultats de l'analyse ESLint ❸.

4.4.3 Extraction du Code Non Conforme et Corrigé

Pour chaque analyse d'un projet, ESLint génère un fichier JSON contenant les résultats de l'analyse sur chacun des fichiers sources présents. Pour chacun de ces fichiers sources, nous avons le nombre d'erreurs trouvées dans son contenu par l'analyse. Si un fichier ne contient aucune erreur trouvée, nous avons à disposition uniquement son nom. Sinon, lorsqu'un fichier comporte au moins une erreur, nous avons le contenu complet du fichier ainsi que les détails sur les violations. Dans presque tous les cas, chaque violation est

9. Ce fichier, présent pour les projets JavaScript et Node.js, définit les métadonnées d'un projet et ses dépendances.

composée du nom de la règle enfreinte, des détails sur son emplacement dans le fichier et de sa correction.

Nous passons en revue tous les fichiers de résultats et vérifions deux paramètres pour chaque erreur ❹. Premièrement, en lien avec notre besoin d'analyser uniquement les règles sur une ligne, nous nous assurons que les lignes de début et de fin de la violation sont les mêmes. Deuxièmement, nous contrôlons l'usage de la minification. Malgré le filtre précédemment appliqué sur les noms de fichiers, nous avons observé que le code minifié était encore présent dans notre jeu de données puisque tous les développeurs n'utilisent pas la convention `min.js`. Nous supprimons les lignes de plus de 115 caractères pour éviter le code minifié. Pour calculer ce seuil, nous avons parcouru 385 fichiers aléatoires dans notre jeu de données, récupéré la longueur de toutes les lignes et utilisé le quantile à 99% comme seuil de filtrage. Nous choisissons seulement 385 fichiers pour éviter de parcourir tous les fichiers et gagner du temps. Nous utilisons la formule de taille d'échantillon de Cochran avec un niveau de confiance à 95% et une précision de 5%.

Lorsque les deux conditions sont remplies, nous enregistrons le numéro de ligne, le contenu de la violation, la correction si fournie par ESLint, et le chemin vers le fichier contenant l'erreur ❺. Nous enregistrons également le projet GitHub original avec le SHA de commit associé au moment du clonage à des fins de reproductibilité.

Enfin, pour les besoins de notre protocole, nous avons besoin d'un minimum de 1000 exemples (violations et corrections) pour chaque règle. L'application de ce seuil élimine 16 règles, résultant en un total final de 38 règles.

4.4.4 Statistiques Descriptives

Notre base de code de 550 projets clonés de GitHub contient 218530 fichiers avec plus de 33 millions de lignes de code. Notre jeu de données final contient presque 13 millions de violations de 38 règles différentes. Il existe un écart important entre les règles les plus et les moins enfreintes. Nous avons presque 4 millions d'exemples pour la règle `quotes` et à peine 1480 exemples pour la règle `no-floating-decimal` (*cf.* Figure 4.3). Une observation importante est le rapport entre le nombre d'exemples non conformes et le nombre de lignes de notre corpus pour chaque règle. Il y a 35 règles ayant un ratio inférieur à 1%. De cette observation, nous pouvons conclure que la plupart des pratiques de code dans notre jeu de données résultent en des problèmes de classification extrêmement déséquilibrés [105]. D'ailleurs, ces ratios sont cohérents avec ceux observés dans les travaux précédents [72]. Par conséquent, nous conjecturons que cette distribution des ratios est inhérente au problème de détection du code non conforme.

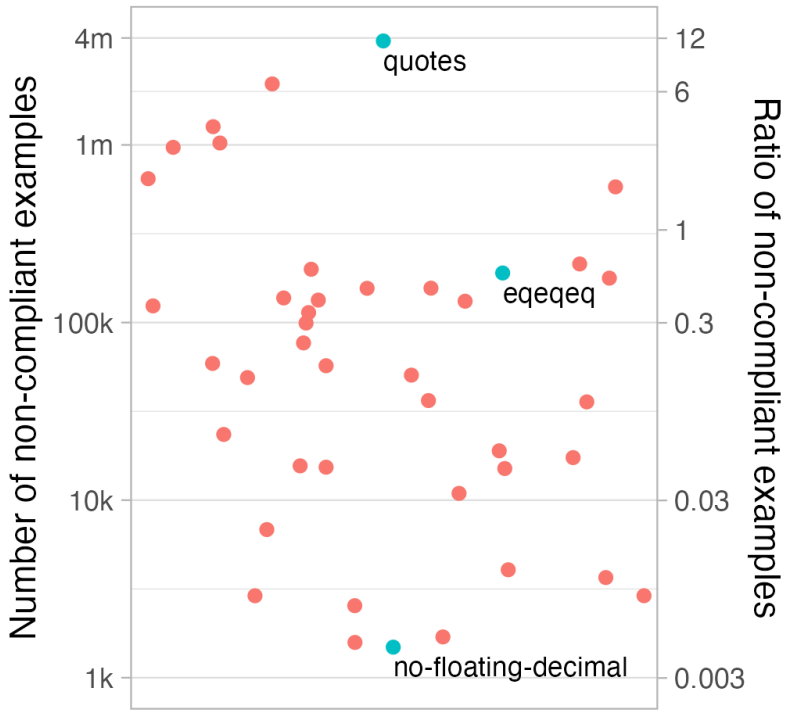


FIGURE 4.3 – Nombre d'exemples non conformes, avec le ratio, par règle

4.5 Protocole Expérimental

Répondre à nos deux questions de recherche nécessite la construction de plusieurs classifieurs pour différentes règles de linter, avec différents nombres d'exemples utilisés pour apprendre une règle et différentes proportions de code conforme et non conforme. Tous ces classifieurs sont entraînés sur un ensemble d'entraînement et sont validés sur un ensemble de test en vérifiant leur capacité à détecter les mêmes violations de règles que le linter.

4.5.1 Configurations d'Apprentissage

Chaque classifieur cible une pratique de code définie par une règle ESLint et vérifie si une ligne de code donnée est conforme à la pratique. Il est entraîné sur un ensemble d'entraînement qui consiste en des lignes de code, certaines non conformes et d'autres conformes.

Concernant le nombre de lignes dans l'ensemble d'entraînement, nous rappelons que notre motivation principale est de vérifier s'il est possible d'entraîner un classifieur en l'entraînant avec un nombre relativement petit d'exemples. Suite à des discussions avec Packmind, nous considérons qu'un ensemble de 10 exemples est assez petit (facilement réalisable par un seul développeur), 100 est moyen (à la portée d'une équipe de développement) et 1000 est grand (nécessiterait un travail coordonné de plusieurs équipes). Nous considérons donc trois tailles (S=10, M=100, L=1000) pour les ensembles d'entraînement.

Pour la plupart des pratiques, nous sommes confrontés à des données extrêmement déséquilibrées pour lesquelles la présence du code conforme domine largement le code non conforme (*cf.* Section 4.4). Pour aborder ce problème, nous employons une méthode, souvent utilisée pour éviter le sur-échantillonage d'une instance, reposant sur le sous-échantillonnage des instances de code conforme pour construire des ensembles d'entraînement équilibrés avec 50% d'instances de code conforme et 50% d'instances de code

non conforme [105]. Les 50% de code conforme dans les ensembles d'entraînement peuvent être du code corrigé ou du code tiers, comme décrit dans la Section 4.3. Notre hypothèse est que le classifieur sera plus performant lorsqu'il est entraîné avec une certaine quantité de code corrigé, qui représente une sorte de «frontière» entre les instances de code conforme et non conforme. Suivant cette idée, nous optons pour trois ratios pour construire nos ensembles d'entraînement : 50% de code non conforme et 50% de code corrigé (le ratio VF) ; 50% de code non conforme et 50% de code tiers (le ratio VE) ; et 50% de code non conforme, 25% de code corrigé, et 25% de code tiers (le ratio VFE). Ainsi, les trois ratios respectent toujours une répartition égale entre code conforme et non conforme.

Nous obtenons 9 configurations d'apprentissage pour chaque classifieur en combinant les trois tailles (S, M, L) et les trois ratios (VF, VE, VFE). Par exemple, un classifieur M/VF est entraîné sur un ensemble de 100 lignes, réparties en 50 lignes de code non conforme et 50 lignes de code corrigé. Notons que le cas spécial S/VFE est entraîné sur 10 lignes composées de 5 lignes de code non conformes, 3 lignes de code corrigées et 2 lignes de code tierces.

4.5.2 Protocole de Validation

Étant donné que le nombre de lignes de code source utilisées pour nos ensembles d'entraînement (10, 100 ou 1000) est très petit par rapport au nombre total de lignes de code source dans notre jeu de données (33 millions de lignes), il est probable que la précision et le rappel obtenus par un classifieur particulier ne soient pas suffisamment représentatifs. Pour adresser ce problème, nous utilisons une validation inspirée de l'approche de *out-of-sample bootstrap* [106], qui est l'approche de validation la plus performante selon Tantithamthavorn *et al.* [107]. Nous entraînons 100 classifieurs à chaque fois pour chacune de nos 9 configurations et chacune des 38 règles de linter. Les lignes incluses dans un ensemble d'entraînement donné sont tirées au hasard sans remise depuis l'ensemble du jeu de données jusqu'à ce que la taille et les ratios attendus soient obtenus. Cependant, entre chacun des 100 jeux de données créés pour réaliser l'entraînement d'un classifieur, les lignes sont replacées. Par exemple, pour un ensemble d'entraînement de classifieur S/VFE, nous tirons au hasard 5 lignes de code non conformes, 3 lignes de code corrigées et 2 lignes de code tierces. Puis, ces lignes sont replacées pour créer l'ensemble d'entraînement suivant. Contrairement à l'*out-of-sample bootstrap* classique, nous ne tirons pas un ensemble d'entraînement de la même taille que l'ensemble du jeu de données car, dans notre étude, nous voulons évaluer l'effet de la taille du test d'entraînement. Cet ensemble d'entraînement est ensuite utilisé pour fine-tuner CodeBERT, tel que décrit dans la Section 4.3. Comme recommandé par Devlin *et al.* [94], nous utilisons les hyperparamètres suivants pour le processus de fine-tuning : 4 époques, une taille de batch de 16 et un taux d'apprentissage de 5e−5.

Nous validons nos classifieurs sur un ensemble de lignes de code source (l'ensemble de test), et demandons, pour chaque ligne, si elle est conforme ou non. Nous comparons ensuite les résultats du classifieur avec la vérité établie par ESLint sur les mêmes lignes et calculons sa précision, son rappel, son accuracy et son F-score. Pour construire un ensemble de test, nous devons définir sa taille et son ratio. Nous définissons deux approches pour construire l'ensemble de test : les approches *équilibrée* et *réaliste*. Dans l'approche équilibrée, nous construisons un ensemble de test de la même taille et ratio que l'ensemble d'entraînement, en tirant des instances au hasard sans remise parmi les instances non incluses dans l'ensemble d'entraînement, pour imiter l'approche *out-*

of-sample bootstrap. En revanche, contrairement à l'*out-of-sample bootstrap*, nous n'utilisons pas toutes les instances non incluses dans l'ensemble d'entraînement comme ensemble de test car notre jeu de données contient des millions de lignes de code, et son utilisation serait trop coûteuse en termes de calcul. Dans l'approche réaliste, nous construisons un ensemble de test avec un équilibre similaire aux fichiers de code source réels. Nous construisons un ensemble de test composé de toutes les lignes de moins de 115 caractères contenues dans 5 fichiers de code source tirés. Ces 5 fichiers sont tirés au hasard sans remise parmi les fichiers qui n'ont aucune ligne commune avec l'ensemble d'entraînement, et qui possèdent au moins une ligne de code non conforme par rapport à la règle apprise. Cet ensemble de test vise à approximer l'équilibre des violations qui existent dans les fichiers de code réels, tout en ayant une quantité minimale de code non conforme pour réaliser nos mesures.

4.5.3 Exécution du Protocole

Au total, notre protocole a construit 34200 classifieurs (38 règles × 9 types × 100). Chaque classifieur apprend sur son propre ensemble d'entraînement, construit aléatoirement à partir d'un ensemble global de lignes de code source contenant des lignes de code non conforme, corrigé et existant. Chaque classifieur est ensuite validé deux fois selon nos deux approches de validation. Le script permettant d'exécuter notre protocole a été rédigé dans un notebook Python. Nous avons ensuite utilisé la plate-forme Google Colab [10] afin d'exécuter tout notre protocole en parallèle et sur des machines performantes. Cela a nécessité au total plus de 48 heures d'exécution pour entraîner et valider nos classifieurs.

4.6 Résultats

Pour répondre à nos questions de recherche, nous analysons maintenant les résultats obtenus en appliquant le protocole introduit dans la Section 4.5 au jeu de données conçu dans la Section 4.4. Plus précisément, nous étudions l'influence de nos deux principaux paramètres sur la performance des classifieurs résultants : la taille de l'ensemble d'apprentissage et le ratio de lignes non conformes et conformes.

Nous divisons cette section selon nos deux méthodes de validation. Nous présentons d'abord les résultats obtenus avec la validation équilibrée dans la Section 4.6.1, puis les résultats obtenus avec la validation réaliste dans la Section 4.6.2. Nous discutons des résultats dans la Section 4.6.3 et enfin, nous concluons avec les obstacles à la validité dans la Section 4.6.4.

Pour chaque classifieur, nous calculons leurs scores de précision, d'accuracy et de rappel. Nous agrégeons les résultats obtenus pour chaque règle par taille et par ratio. Cela signifie que, pour une configuration donnée (par exemple M/VF) et une méthode de validation (par exemple réaliste), nous agrégeons les 3 800 scores obtenus par les classifieurs correspondants (38 règles × 100 classifieurs).

4.6.1 Validation Équilibrée

La Figure 4.4 illustre les résultats obtenus avec la validation équilibrée. Tout d'abord, indépendamment de la taille et du ratio, l'ensemble complet des 34200 classifieurs (38

10. https://colab.google

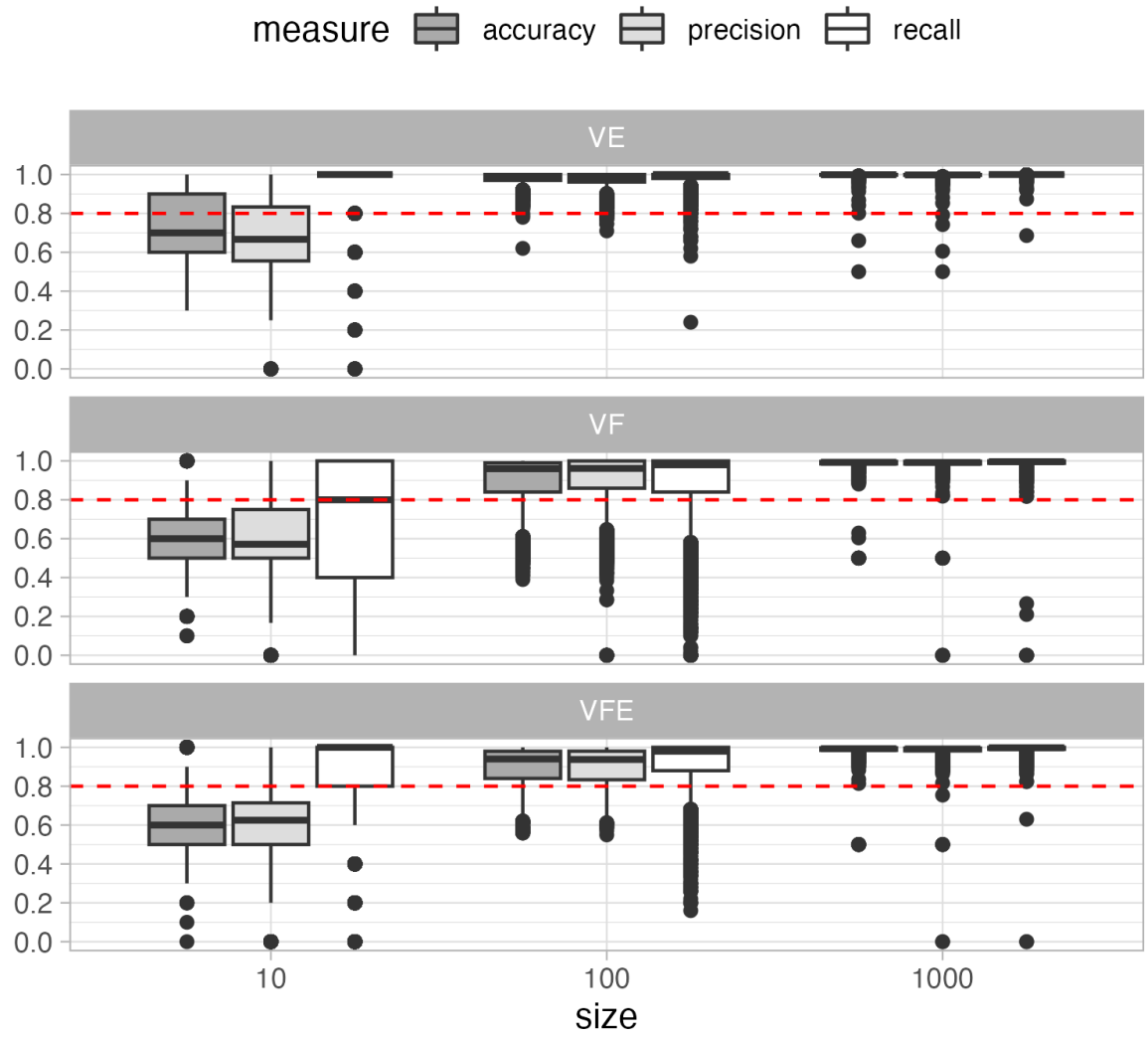


FIGURE 4.4 – Scores obtenus par les classifieurs selon le nombre de lignes utilisées pour l'entraînement, groupés par ratio, pour la validation équilibrée

règles × 3 tailles × 3 ratios × 100 mesures) obtient une précision médiane de 0,979 et un rappel médian de 1. Ces premiers résultats sont très prometteurs puisque nous visons une précision minimale de 0,8 et une précision idéale ≥ 0,95.

Influence de la taille

En examinant les distributions, nous remarquons que tous les scores augmentent avec la taille de l'ensemble d'entraînement, indépendamment du ratio. Tout d'abord, nous regroupons toutes les mesures obtenues pour chaque taille et comparons leur médiane. En ce qui concerne la précision, les tailles S, M et L obtiennent respectivement 0,625, 0,977 et 0,995. L'accuracy suit la même tendance avec 0,700, 0,970 et 0,996. Le rappel est stable quelle que soit la taille, avec des scores de 1, 0,980 et 0,998. Nous observons la même tendance pour chaque ratio individuel, comme indiqué dans la Figure 4.4.

Pour confirmer la différence entre chaque taille d'apprentissage, nous calculons ensuite les *non-parametric Mann-Whitney U tests* [108] et les *effect sizes* (en utilisant la *rank-biserial correlation*—RBC [109]) entre les groupes (*cf.* Tableau 4.1). Le RBC est une valeur comprise entre −1 et 1. Une valeur RBC de 1 indique que toutes les valeurs du premier

TABLEAU 4.1 – p-values de Mann-Whitney et rank-biserial correlation obtenues pour l'analyse de la précision dans la validation équilibrée

Size			
Size 1	Size 2	p-value	RBC
S	M	0	−0,70
S	L	0	−0,76
M	L	0	−0,34
Ratio			
Ratio 1	Ratio 2	p-value	RBC
VE	VFE	$1,8e^{-301}$	0,28
VE	VF	$3,2e^{-112}$	0,17
VFE	VF	$2,5e^{-38}$	−0,10

TABLEAU 4.2 – Nombre de règles pour lesquelles la précision médiane des classifieurs est supérieure à P pour chaque taille et ratio dans la validation équilibrée

	P = 0,8			P = 0,95		
Size	VE	VFE	VF	VE	VFE	VF
S	1	0	3	0	0	1
M	38	32	30	38	17	25
L	38	38	38	38	38	38

groupe sont supérieures à toutes les valeurs du second groupe. Une valeur de 0 indique une quantité égale de valeurs dans chaque groupe supérieure à l'autre groupe. Nous comparons les groupes deux à deux : S vs. M, S vs. L et M vs. L. Nous ajustons les p-values obtenues en utilisant une correction de Bonferroni. Nous obtenons des p-values égales à 0 pour les trois comparaisons, rejetant les hypothèses nulles. Concernant l'*effect size*, le RBC indique que S < M < L. La différence entre S et M, ainsi qu'entre S et L, est grande avec un RBC ≤ −0,7. La différence entre M et L est moins marquée, avec un RBC de −0,34.

Les développeurs privilégient les outils qui minimisent les faux positifs. Ainsi, nous examinons maintenant combien des 38 règles produisent des classifieurs qui obtiennent une précision médiane au-dessus de notre objectif minimum de 0,8 et de notre objectif idéal de 0,95 (*cf.* Tableau 4.2). Nous observons que chaque règle atteint les objectifs minimum et idéal pour la taille L, que certaines n'atteignent pas les objectifs pour la taille M, et que la taille S ne peut pas produire de classifieurs satisfaisants.

En résumé, nous observons que le nombre de lignes utilisées pour l'apprentissage affecte positivement la performance des classifieurs résultants. Il est important de relever que pour une taille moyenne de 100 lignes, nous obtenons de nombreux résultats qui répondent à nos exigences, et que ces résultats sont proches de ceux obtenus avec la grande taille de 1000 lignes.

Influence du ratio

Nous appliquons la même méthodologie que celle utilisée pour étudier l'influence de la taille afin d'étudier l'influence du ratio. Dès lors, nous voyons immédiatement que

l'impact est moins évident.

Nous analysons les mesures médianes en les regroupant par ratio. Pour les ratios VE, VFE et VF, nous obtenons des médianes de précision de 0,992, 0,943 et 0,978, des médianes d'accuracy de 0,990, 0,940 et 0,960, et des médianes de rappel de 1, 0,998 et 0,988. Nous observons que le ratio VE donne les meilleurs résultats, suivi de VF et VFE. Lors du regroupement des mesures par taille (*cf.* Figure 4.4), nous observons que VE donne les meilleurs résultats pour les tailles S et M, les résultats pour la taille L étant très similaires pour les trois ratios. Il n'y a pas de différence visible entre les ratios VFE et VF pour les tailles S et L.

Les tests statistiques confirment notre première observation : la différence entre les trois configurations n'est pas aussi marquée qu'avec la taille (*cf.* Tableau 4.1). Les trois comparaisons par paires sont désormais les suivantes : VE vs. VFE, VE vs. VF et VFE vs. VF. Les résultats obtenus par les classifieurs avec différents ratios sont différents, comme l'indiquent les p-values. Cependant, les scores RBC indiquent que leur impact est beaucoup plus faible que la taille.

Calculer le nombre de règles avec une précision médiane supérieure à nos seuils n'aide pas à discriminer les ratios. Pour la taille S, VF semble mieux performer ; pour la taille M, VE performe mieux ; et pour la taille L, toutes les règles atteignent l'objectif idéal de 0,95.

En résumé, le ratio de code non conforme et conforme (corrigé ou tiers) utilisé pour entraîner nos classifieurs a un effet limité : le meilleur ratio pour toutes les tailles d'ensemble d'entraînement est VE, mais l'amélioration est mineure. Pour cette validation équilibrée, nous concluons que la configuration d'apprentissage la plus performante possède une taille L et un ratio VE. Cependant, les résultats obtenus avec la taille M atteignent également nos objectifs, ce qui les rend utilisables et plus faciles à appliquer dans notre domaine d'application.

4.6.2 Validation Réaliste

La Figure 4.5 illustre les résultats obtenus avec la validation réaliste. La première observation est claire : les scores de précision chutent, quelle que soit la taille et le ratio utilisés pour l'entraînement. Nous obtenons de bons résultats pour la médiane globale de l'accuracy (0,887) et du rappel (0,929), mais la médiane de la précision tombe à 0,043, comparée à 0,979 avec la validation équilibrée.

Influence de la taille

Concernant la taille, nous observons la même tendance qu'avec la validation équilibrée : des tailles plus grandes pour l'ensemble d'entraînement donnent de meilleurs résultats. Par souci de concision, nous n'effectuons pas une analyse statistique aussi détaillée que pour la validation équilibrée car les scores de précision obtenus sont faibles dans l'ensemble : 0,014 pour la taille S, 0,091 pour la taille M, et 0,133 pour la taille L. Cependant, les tests statistiques confirment deux points (*cf.* Tableau 4.3). Premièrement, il existe effectivement une différence entre les distributions des précisions obtenues pour chaque taille, et les *rank-biserial correlation* indiquent que S < M et S < L. Deuxièmement, le score RBC obtenu entre M et L est seulement de −0,02, indiquant un effet très faible. Bien que les scores de précision soient beaucoup plus faibles que ceux observés dans la validation équilibrée, la tendance est similaire.

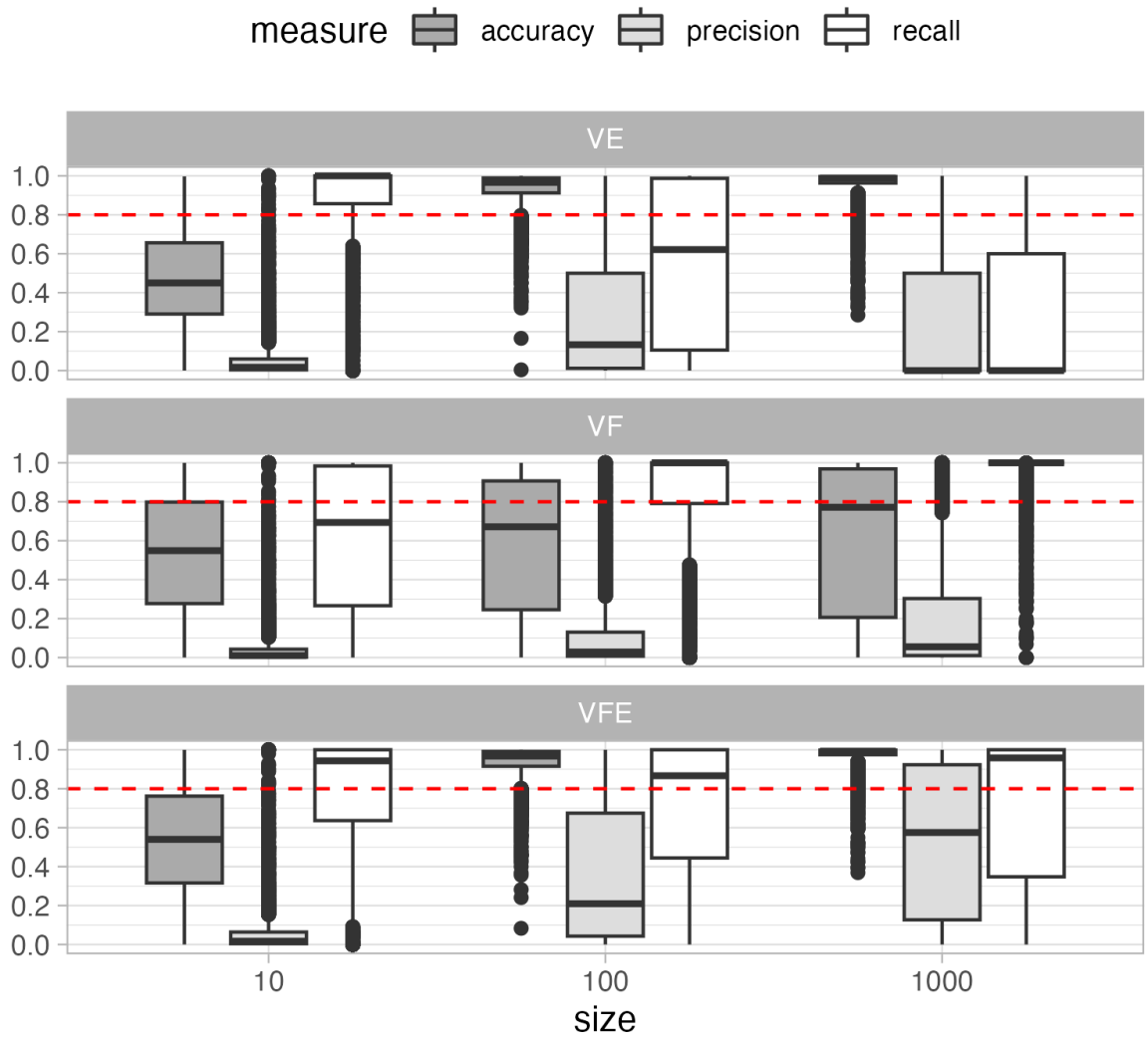


FIGURE 4.5 – Scores obtenus par les classifieurs selon le nombre de lignes utilisées pour l'entraînement, groupés par ratio, pour la validation réaliste

TABLEAU 4.3 – p-values de Mann-Whitney et rank-biserial correlation obtenues pour l'analyse de la précision dans la validation réaliste

Size			
Size 1	Size 2	p-value	RBC
S	M	0	−0,39
S	L	0	−0,30
M	L	0,0005	−0,02
Ratio			
Ratio 1	Ratio 2	p-value	RBC
VE	VFE	$5,0e^{-219}$	−0,2408
VE	VF	0,31	0,0006
VFE	VF	0	0,2909

Influence du ratio

Concernant le ratio, avec la validation équilibrée, aucune configuration ne s'est démarquée. Ici, une configuration produit des résultats légèrement meilleurs. Entre VF et VE, les tests indiquent deux populations similaires, mais elles sont toutes deux dominées par VFE. Dans notre validation réaliste, le ratio a un impact réel sur la précision résultante des classifieurs.

4.6.3 Discussions

Comme conclusion générale de nos analyses, nous observons que la méthode de validation a un impact considérable sur la précision résultante des classifieurs. En effet, de nombreuses règles et classifieurs qui ont obtenu de bons résultats de précision dans la validation équilibrée ont chuté dans la validation réaliste. Comme notre scénario industriel implique l'analyse de fichiers du monde réel, où le ratio de code non conforme sur le code conforme est généralement très faible, les classifieurs ne se comporteraient pas bien par rapport aux exigences des développeurs. Nous pouvons expliquer ce manque de succès par deux points.

Premièrement, bien que CodeBERT s'adapte bien à l'apprentissage par transfert, il est généralement utilisé pour apprendre sur des ensembles d'entraînement plus grands que ce que nous exigeons pour notre scénario. Notre objectif est d'utiliser le moins d'exemples possible pour rendre l'approche utilisable en pratique, nous nous concentrons donc sur des tailles d'ensembles d'entraînement allant jusqu'à 1 000 lignes de code. Il est probable que nous obtiendrions de bien meilleurs résultats en utilisant des tailles d'ensembles d'entraînement plus importantes, comme cela a déjà été observé par d'autres auteurs [72]. Cependant, cela impliquerait de ne pas pouvoir appliquer l'approche à notre scénario cible. Dans d'éventuels travaux futurs, il pourra être envisageable d'étudier comment l'approche de Ochodek *et al.* [72] se comporte avec des tailles d'ensembles d'entraînement plus petites, et de comparer sa méthode de validation (k-fold stratifié) à la nôtre.

Deuxièmement, la validation réaliste obtient de faibles résultats de précision parce qu'il y a seulement quelques lignes de code non conforme dans l'ensemble de validation donné aux classifieurs. Pour rappel, 35 des 38 règles ont un ratio de lignes non conformes sur des lignes conformes inférieur à 1%. Les classifieurs sont confrontés à un problème

de données extrêmement déséquilibrées. De fait, même si les classifieurs atteignent une très haute accuracy, ils classeront inévitablement une petite portion des lignes conformes comme non conformes. Cela nuira donc à la précision obtenue lorsque les lignes conformes sont majoritaires, en raison du base rate fallacy [110, 111].

En résumé, nous répondons à nos questions de recherche comme suit :

RQ1 : Avec les deux méthodes de validation, nous observons que des tailles plus grandes d'ensembles d'entraînement donnent de meilleurs résultats. Une petite taille S de 10 lignes est clairement insuffisante pour apprendre efficacement une pratique de code. Par ailleurs, dans la validation équilibrée, nous observons que la différence entre les tailles M et L est petite et que les deux peuvent être utilisées pour apprendre efficacement les pratiques avec une précision $\geq 0,8$. Bien que la taille M soit capable d'apprendre certaines pratiques de code avec une précision $\geq 0,95$, la taille L se comporte mieux dans ce cas.

RQ2 : Dans la validation équilibrée, nous n'observons pas qu'un ratio a un avantage clair sur les autres. Dans la validation réaliste, le ratio VFE obtient de meilleurs scores, mais la précision atteinte dans chaque cas est trop faible indépendamment.

4.6.4 Obstacles à la Validité

En ce qui concerne la validité interne, nous avons extrait des projets populaires de GitHub et analysé tout le code JavaScript qu'ils contiennent. Cependant, une partie de ce code, comme le code obscurci, le code minifié ou le code généré, n'est pas destinée à être soumise à des pratiques de code, et n'est donc pas une bonne base pour apprendre et tester. Nous avons atténué cet obstacle en faisant de notre mieux pour exclure le code minifié de notre jeu de données, mais nous ne pouvons pas garantir que notre jeu de données contient uniquement du code qui a été manuellement écrit par des développeurs.

En ce qui concerne la validité externe, notre étude utilise uniquement un sous-ensemble de règles d'ESLint qui peuvent être décidées sur une ligne. Par conséquent, nous ne pouvons pas garantir que les résultats obtenus sur un ensemble arbitraire de règles spécifiques à une entreprise seraient similaires. Cependant, nous nous sommes assurés de sélectionner des règles qui peuvent être apprises. Par conséquent, nos résultats pourraient être vus comme une sorte de limite supérieure.

4.7 Conclusion

Dans ce chapitre, nous avons présenté une étude de faisabilité sur la capacité de CodeBERT à apprendre des pratiques de code par transfert d'apprentissage en utilisant peu d'exemples. Bien que notre approche obtienne une assez bonne accuracy et un bon rappel, sa précision est trop faible pour être utilisable, probablement en raison du déséquilibre entre la proportion de code conforme et non conforme présents dans les bases de code. Une découverte intéressante est l'obtention de résultats de précision plus faible que dans une étude précédente [72] utilisant une approche d'extraction des caractéristiques personnalisée et des arbres de décision classiques sur des ensembles d'entraînement de tailles comparables. Cela a été surprenant car CodeBERT surpasse généralement les classifieurs classiques dans les tâches de génie logiciel.

4.8 Impact pour Packmind

Comme dans le chapitre précédent, cette dernière section discute l'impact de cette publication pour la compagnie Packmind.

Comme présenté dans la Section 4.6.3, notre outil MLinter n'est pas assez performant pour être utilisé dans un contexte industriel. Cela implique donc qu'il ne peut pas être mis en production sous cette forme au sein de notre outil Packmind. Cependant, cette contribution nous a permis de nous intéresser de manière plus proche à l'Intelligence Artificielle pour Packmind.

Depuis que cette étude a été réalisée, l'essor des *Large Language Model* nous a notamment offert la possibilité de fournir à nos clients de nouvelles fonctionnalités basées sur l'IA. L'arrivée de l'API d'OpenAI [11] nous a permis d'avoir accès facilement à des modèles très performants, interrogeables à l'aide des *prompts*, dans le but d'atteindre nos scénarios souhaités. La première fonctionnalité que nous avons implémentée date de septembre 2023 et permet de suggérer automatiquement de nouvelles pratiques. Un développeur peut soumettre un fichier de code depuis son IDE, et notre système s'occupe de lui suggérer des améliorations à apporter, en créant de nouvelles pratiques avec un nom, une description complète et le code concerné. Une autre fonctionnalité a permis de corriger le problème de la création régulière de pratiques sans description. En fournissant simplement le titre d'une pratique, nous avons ainsi pu faire générer automatiquement des descriptions qui améliorent la qualité de la documentation des pratiques. Enfin, depuis mars 2024, nous avons finalement atteint notre but d'automatiquement détecter les pratiques de code internes à une équipe directement dans leurs outils.

Même si MLinter n'est finalement resté qu'un rêve, il nous a permis une ouverture sur le monde de l'IA en nous introduisant les différents concepts et le fonctionnement des modèles de ML, ce qui a par la suite facilité notre prise en main des LLM. Cela a ainsi contribué à améliorer notre solution Packmind.

11. https://openai.com/product

Chapitre 5

Conclusion et Perspectives

Sommaire

5.1 Conclusion

Au début de cette thèse, nous avions identifié deux problématiques au cœur de l'outil Packmind, et plus généralement de la gestion des pratiques de code : l'amélioration de leur documentation et leur apprentissage automatique à partir d'exemples existants. Chacune de ces problématiques a finalement mené à la réalisation d'une contribution scientifique.

La première partie de cette thèse a mis en lumière l'importance cruciale de la documentation des pratiques de code. En analysant plus de 100 règles issues de 16 linters différents pour 7 langages de programmation, nous avons développé une taxonomie détaillée des éléments présents dans les documentations de pratique actuelles. Les résultats de notre enquête auprès des développeurs ont ensuite souligné une lacune notable concernant l'explication du *Pourquoi* des pratiques de code. Cette composante a pourtant été jugée déterminante pour l'adoption et la compréhension de ces pratiques. Nous avons aussi constaté que le mélange de *Texte* et de *Code* est efficace pour documenter les objectifs *Quoi* et *Correction*. Enfin, il est apparu un conflit, lors de la découverte d'une pratique, entre le besoin d'apprendre et le besoin de gagner du temps. Pour cela, l'utilisation d'une structure cohérente composée, en partie, d'un court résumé et d'un minimum de liens externes a été exprimée.

Dans la seconde partie, nous avons étudié la possibilité d'utiliser des techniques d'apprentissage automatique (*Machine Learning*) pour apprendre et identifier des pratiques de code à partir d'un nombre limité d'exemples. Bien que notre approche a démontré des résultats prometteurs dans un scénario contrôlé, nous avons cependant été confrontés à des problèmes de précision dans le contexte d'une application à du code réel. Le fort déséquilibre entre la proportion de code conforme et de code non conforme présents dans

les bases de code est une des raisons majeures de ce résultat. Ce travail a cependant permis de mettre en lumière le potentiel de l'apprentissage par transfert (*fine-tuning*) lorsque nous possédons un faible jeu de données d'entraînement.

Cette thèse a ainsi répondu à nos deux problématiques initiales en contribuant scientifiquement à la connaissance sur les pratiques de code, tout en ayant un impact positif direct ou indirect sur la solution Packmind. L'intégration de l'intelligence artificielle a permis de simplifier l'expérience utilisateur, et de nouvelles applications sont en cours d'étude. Nous travaillons également pour améliorer la mise en forme et la structure de notre documentation de pratiques.

5.2 Perspectives

Les travaux précédemment présentés ont laissé encore quelques questions ouvertes concernant nos deux problématiques originales, et de nouvelles questions ont fait leur apparition. Cette section a pour but de les présenter et de proposer des pistes pour de nouvelles contributions.

5.2.1 Améliorer la Qualité de la Documentation des Pratiques

Maintenant que nous avons analysé le contenu actuel des documentations de pratiques de code, nous souhaiterions pouvoir aller plus loin dans les recommandations. Notre contribution a permis d'avancer sur ce qu'il fallait faire, mais sans vraiment détailler comment le réaliser. C'est d'ailleurs une question qui nous a régulièrement été posée lors de la présentation de nos travaux : quelles sont les actions à mettre en place pour améliorer des documentations existantes ? Pour cela, nous souhaiterions pouvoir fournir une sorte de patron à appliquer, ce qui, à la manière des pratiques de code, rendrait leur documentation cohérente entre tous les outils. D'ailleurs, nous nous sommes récemment aperçus que les règles de Sonar avaient été mises à jour depuis notre extraction de données et intégraient maintenant des sections commentant explicitement le *Pourquoi* et le *Correction* de leurs règles. [1] Il pourrait donc être intéressant de collaborer avec la société Sonar [2] afin d'obtenir un retour d'expérience sur les impacts internes et externes de ces changements.

Après avoir interrogé les développeurs dans notre contribution, nous pensons qu'il serait également intéressant de consulter les auteurs de documentations de pratiques. Le but serait d'avoir leur point de vue sur la pertinence de nos résultats, et quelles actions pourraient être mises en place. Nous avons d'ailleurs commencé à partager les résultats de notre recherche en créant une pull request sur le dépôt GitHub de PHP CS Fixer. [3] Un des contributeurs a pris le temps de nous répondre et de commenter nos résultats. Nous avons notamment eu un retour assez intéressant sur l'inclusion du *Pourquoi* dans leurs règles qui lui paraissait difficile à réaliser, puisque je cite : *"Really often it's just an opinionated logic and the "why" can be subjective."* Cela pourrait être une première raison pour laquelle la qualité perçue du *Pourquoi* n'est pas optimale. Cette réponse pose également la question de la différence entre pratique et convention. Il existe de nombreuses pratiques de code, qui peuvent d'ailleurs se contredire. Lorsque nous cadrons une pratique à un contexte donné, elle devient une convention, validée selon les convictions de l'équipe,

1. Un exemple est disponible ici : https://rules.sonarsource.com/javascript/RSPEC-6105/
2. https://www.sonarsource.com
3. https://github.com/PHP-CS-Fixer/PHP-CS-Fixer/discussions/7934

qui doit être appliquée par tous. Une pratique validée à l'aide de Packmind devrait être basée sur une opinion, ce qui la transforme en convention et la raison du *Pourquoi* devrait émerger.

Enfin, lors de l'analyse des règles des linters, nous avons remarqué que l'organisation interne des pratiques était également très différente entre les différents outils. Par exemple, certains linters regroupent les règles par catégorie, certains les affichent simplement par ordre alphabétique, alors qu'à Packmind nous les affichons par ordre de création. Nous avons noté d'autres différences sur les catégories utilisées ou encore sur le systèmes de tri et de recherche. Toujours dans le but d'améliorer notre solution Packmind, nous pourrions envisager une étude sur les catalogues de pratiques en consultant une nouvelle fois les développeurs.

5.2.2 Apprendre Automatiquement les Pratiques Internes

Même si notre outil MLinter a apporté des enseignements intéressants sur l'apprentissage par transfert, il reste, cependant, à notre échelle un échec qui n'a pas abouti directement à une amélioration pour Packmind. Cela a été une surprise pour nous. Nous avions trouvé une étude similaire qui obtenait déjà des résultats satisfaisants avec l'utilisation d'arbres décisionnels [72], et nous supposions que cette approche serait moins performante que CodeBERT. En effet, CodeBERT semblait être une meilleure option car il a obtenu des résultats intéressants dans des tâches similaires de génie logiciel. Une première étape que nous souhaiterions réaliser sur ce domaine de la détection automatique de pratiques, serait de réaliser une étude de réplication du travail d'Ochodek *et al.* [72]. Nous voudrions voir si leur méthode se généralise bien à d'autres langages comme JavaScript, et à un linter comme ESLint. Selon les résultats obtenus, et notamment le nombre minimal d'exemples nécessaires, nous pourrions analyser son éventuelle intégration au sein de Packmind en complément de notre solution actuelle.

Cependant, un point important sur notre étude réalisée concerne la date à laquelle nous l'avons soumise à SANER 2023. Nous avons envoyé notre papier dans la track REproducibility Studies and NEgative Results (RENE) le 20 novembre 2022, et quelques jours plus tard le premier prototype ouvert de ChatGPT était disponible. Si nous avions réalisé notre étude quelques mois plus tard, il y a de fortes chances que nous aurions adopté une stratégie complètement différente. La démocratisation des LLM a grandement transformé la façon dont nous pouvons atteindre des résultats intéressants avec une quantité d'information initiale relativement réduite. Comme expliqué précédemment dans la Section 4.8, nous sommes maintenant capables chez Packmind d'analyser un fichier complet et d'y identifier les pratiques internes d'une équipe, et ce directement dans l'IDE d'un développeur. Tout cela est désormais réalisable grâce à un prompt. Certes, il a fallu itérer sur celui-ci pour obtenir des résultats satisfaisants, mais cette tâche n'a pas représenté la même charge de travail que pour les méthodes jusqu'à maintenant utilisées. Bien que cette solution soit maintenant utilisée par nos clients, nous pensons qu'il reste tout de même utile de réaliser une étude sur sa mise en place. Nous souhaiterions notamment comparer les performances de l'analyse en faisant évoluer le contenu envoyé dans le prompt (intégrer la description de la pratique, un exemple de correction ou encore utiliser du *chain of thought*) ou en réalisant de l'apprentissage par transfert pour chaque pratique.

5.2.3 Inférer Automatiquement une Correction

Une problématique qui a émergé au début de cette thèse était de pouvoir corriger automatiquement les erreurs identifiées. À ce moment-là, nous n'avions pas encore la possibilité d'analyser le code à la recherche de pratiques internes. Il nous a donc paru moins prioritaire de corriger du code que nous ne pouvions pas encore identifier. Cependant, maintenant que nous commençons à fournir une identification des pratiques, il nous paraît primordial de nous pencher sur ce sujet. En outre, Packmind stockant des exemples positifs et négatifs pour une même pratique, cela pourrait nous permettre d'avoir des données pertinentes pour pouvoir inférer une correction automatique lors de la détection d'une erreur.

Cette question a déjà été soulevée de nombreuses fois pour réparer automatiquement les erreurs identifiées par les linters [112, 113, 114, 115, 116] et corriger l'utilisation d'API obsolètes, avec de bons résultats mesurés [117, 118, 119, 120, 121, 122, 123]. L'idée est d'inférer automatiquement une transformation d'arbre basée sur l'AST à partir d'un ensemble d'exemples. Certaines approches vont même au-delà et peuvent traiter les dépendances de contrôle et de flux de données [121]. Un autre avantage est qu'elles semblent nécessiter moins d'exemples que les méthodes d'apprentissage automatique traditionnelles.

Cependant, même si ce sujet a largement été étudié dans la littérature scientifique, il reste néanmoins un nombre important de défis à relever. Ces différentes études opérant au niveau de l'AST, elles nécessitent des outils avancés et spécifiques à chaque langage de programmation utilisé. Par conséquent, adapter ces approches à une large gamme de langages nécessite un effort significatif. De plus, même dans de récentes études [124], il a été montré que choisir le bon contexte nécessaire pour inférer automatiquement des corrections reste relativement non prédictible. Là encore, l'utilisation des LLM pourrait répondre à ces différentes problématiques, notamment grâce à leur large entraînement permettant de se soustraire à la création d'un AST. La correction d'une erreur pourrait donc fonctionner de manière plus universelle, peu importe le langage de programmation. L'évolution rapide de ces outils a également permis l'augmentation de la capacité de contexte qu'il est possible de fournir en entrée. Cela pourrait résoudre les problèmes de choix à faire concernant le contexte pour corriger une erreur, en envoyant un fichier complet.

5.2.4 Maintenir à Jour la Base de Connaissances

Une dernière problématique qui est apparue est la gestion des pratiques au fil de l'évolution d'un projet. En effet, à mesure que la base de code se transforme avec le temps, les pratiques évoluent en parallèle. Ainsi, une pratique créée à un moment donné peut ne plus correspondre à la façon de développer quelques mois plus tard. Or, nous nous apercevons que cette pratique n'est jamais supprimée, puisque noyée dans le flot grandissant des pratiques. Sur le long terme, ces pratiques s'accumulent, polluent la base de connaissances et donnent de mauvaises indications à des développeurs non informés. Nous pouvons associer ce problème à une dette sur la documentation des pratiques. Nous avons observé que des linters, comme ESLint, conservaient des pratiques marquées comme "dépréciées" [4] ou "supprimées" [5] dans leur documentation. Nous pensons donc qu'il pourrait être pertinent de réaliser, dans un premier temps, une étude dans la litté-

4. https://eslint.org/docs/latest/rules/#deprecated
5. https://eslint.org/docs/latest/rules/#removed

rature scientifique sur la gestion de la connaissance au travers du temps ; puis, dans un second temps, de poursuivre avec des entretiens au sein des entreprises mettant en place ce genre de politique. Cela permettrait d'analyser les processus utilisés, notamment la manière dont sont organisées les révisions et dont sont prises les décisions, et le rythme des mises à jour.

Par ailleurs, cette gestion de la dette de la documentation des pratiques semble être fortement liée à l'interface entre les pratiques et les utilisateurs. Cette interface est en réalité incarnée par les catalogues précédemment rencontrés. Nous nous apercevons que ces derniers semblent être un autre pilier important pour la qualité d'une documentation de pratiques. En ce sens, ils pourraient être intéressants de les étudier.

Bibliographie

[1] M. Smit, B. Gergel, H. J. Hoover, and E. Stroulia, “Code convention adherence in evolving software,” in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2011, pp. 504–507, iSSN : 1063-6773. [Online]. Available : https://ieeexplore.ieee.org/abstract/document/6080819 2

[2] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, “Do the dependency conflicts in my project matter?” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA : Association for Computing Machinery, Oct. 2018, pp. 319–330. [Online]. Available : https://doi.org/10.1145/3236024.3236056 2

[3] V. Garousi and M. V. Mäntylä, “When and what to automate in software testing? A multi-vocal literature review,” *Information and Software Technology*, vol. 76, pp. 92–117, Aug. 2016. [Online]. Available : https://www.sciencedirect.com/science/article/pii/S0950584916300702 2

[4] R. C. Martin, *Clean Code : A Handbook of Agile Software Craftsmanship*, 1st ed. USA : Prentice Hall PTR, Jul. 2008. 3

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns : Elements of Reusable Object-Oriented Software*, 1st ed. Boston, Mass. Munich : Addison Wesley, Oct. 1994. 3

[6] E. Evans, *Domain-driven Design : Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004, google-Books-ID : xColAAPGubgC. 3

[7] R. C. Seacord, *Secure Coding in C and C++*. Addison-Wesley, Mar. 2013, google-Books-ID : Z9aNTafcb3IC. 3

[8] B. S. Das and V. Chu, *Security as Code*. "O’Reilly Media, Inc.", Jan. 2023, google-Books-ID : UAqmEAAAQBAJ. 3

[9] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, “A Systematic Literature Review on Fault Prediction Performance in Software Engineering,” *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276–1304, Nov. 2012, conference Name : IEEE Transactions on Software Engineering. [Online]. Available : https://ieeexplore.ieee.org/abstract/document/6035727 3

[10] T. Winters, T. Manshreck, and H. Wright, *Software Engineering at Google : Lessons Learned from Programming Over Time*. "O’Reilly Media, Inc.", Feb. 2020, google-Books-ID : V3TTDwAAQBAJ. 3

[11] J. Bloch, *Effective Java*. Addison-Wesley Professional, Dec. 2017, google-Books-ID : BIpDDwAAQBAJ. 3

[12] K. F. Tómasdóttir, M. Aniche, and A. Van Deursen, "The Adoption of JavaScript Linters in Practice : A Case Study on ESLint," *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 863–891, Aug. 2020. 3, 13, 15, 55, 63

[13] E. Torunski, M. O. Shafiq, and A. Whitehead, "Code style analytics for the automatic setting of formatting rules in IDEs : A solution to the Tabs vs. Spaces Debate," in *2017 Twelfth International Conference on Digital Information Management (ICDIM)*, Sep. 2017, pp. 6–14. [Online]. Available : https://ieeexplore.ieee.org/abstract/document/8244675 3

[14] P. Louridas, "Using wikis in software development," *IEEE Software*, vol. 23, no. 2, pp. 88–91, Mar. 2006, conference Name : IEEE Software. [Online]. Available : https://ieeexplore.ieee.org/abstract/document/1605183 4

[15] J. A. Meloche, H. Hasan, D. Willis, C. C. Pfaff, and Y. Qi, "Cocreating Corporate Knowledge with a Wiki," *International Journal of Knowledge Management (IJKM)*, vol. 5, no. 2, pp. 33–50, Apr. 2009, publisher : IGI Global. [Online]. Available : https://www.igi-global.com/article/cocreating-corporate-knowledge-wiki/www.igi-global.com/article/cocreating-corporate-knowledge-wiki/2750 4

[16] M. Nagappan, R. Robbes, Y. Kamei, E. Tanter, S. McIntosh, A. Mockus, and A. E. Hassan, "An empirical study of goto in C code from GitHub repositories," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. Bergamo Italy : ACM, Aug. 2015, pp. 404–414. [Online]. Available : https://dl.acm.org/doi/10.1145/2786805.2786834 5, 26

[17] M. Tahaei, K. Vaniea, K. K. Beznosov, and M. K. Wolters, "Security Notifications in Static Analysis Tools : Developers' Attitudes, Comprehension, and Ability to Act on Them," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. Yokohama Japan : ACM, May 2021, pp. 1–17. [Online]. Available : https://dl.acm.org/doi/10.1145/3411764.3445616 5, 26, 29

[18] B. Johnson, R. Pandita, J. Smith, D. Ford, S. Elder, E. Murphy-Hill, S. Heckman, and C. Sadowski, "A cross-tool communication study on program analysis tool notifications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Seattle WA USA : ACM, Nov. 2016, pp. 73–84. [Online]. Available : https://dl.acm.org/doi/10.1145/2950290.2950304 5, 26, 28, 29

[19] M. Nachtigall, M. Schlichtig, and E. Bodden, "A large-scale study of usability criteria addressed by static analysis tools," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual South Korea : ACM, Jul. 2022, pp. 532–543. [Online]. Available : https://dl.acm.org/doi/10.1145/3533767.3534374 5, 26, 29

[20] C. Latappy, T. Degueule, J.-R. Falleri, R. Robbes, X. Blanc, and C. Teyton, "Replication Kit - What the Fix? A Study of ASATs Rule Documentation," Jan. 2024. [Online]. Available : https://zenodo.org/records/10522473 6, 26

[21] C. Latappy, Q. Perez, T. Degueule, J.-R. Falleri, C. Urtado, S. Vauttier, X. Blanc, and C. Teyton, “Dataset - MLinter : Learning Coding Practices from Examples—Dream or Reality?” Nov. 2022. [Online]. Available : https://zenodo.org/records/7341456 7

[22] I. Rus and M. Lindvall, “Knowledge management in software engineering,” *IEEE Software*, vol. 19, no. 3, pp. 26–38, May 2002, conference Name : IEEE Software. 8, 9

[23] F. O. Bjørnson and T. Dingsøyr, “Knowledge management in software engineering : A systematic review of studied concepts, findings and research methods used,” *Information and Software Technology*, vol. 50, no. 11, pp. 1055–1068, Oct. 2008. [Online]. Available : https://www.sciencedirect.com/science/article/pii/S0950584908000487 8

[24] M. Lindvall, I. Rus, R. Jammalamadaka, and R. Thakker, “Software Tools for Knowledge Management,” Dec. 2001. 8

[25] B. Boehm and V. Basili, *Software defect reduction top 10 list. Foundations of empirical software engineering : the legacy of Victor R. Basili.* Springer, 2005, vol. 426. 9

[26] M. E. Fagan, “Design and code inspections to reduce errors in program development,” 1976. 9

[27] A. Bosu and J. C. Carver, “Impact of Peer Code Review on Peer Impression Formation : A Survey,” in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, Oct. 2013, pp. 133–142, iSSN : 1949-3789. [Online]. Available : https://ieeexplore.ieee.org/abstract/document/6681346 9

[28] A. Bacchelli and C. Bird, “Expectations, outcomes, and challenges of modern code review,” in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 712–721, iSSN : 1558-1225. 10, 12

[29] P. C. Rigby, D. M. German, and M.-A. Storey, “Open source software peer review practices : a case study of the Apache server,” in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE ’08. New York, NY, USA : Association for Computing Machinery, May 2008, pp. 541–550. [Online]. Available : https://doi.org/10.1145/1368088.1368162 10

[30] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, “Modern code reviews in open-source projects : which problems do they fix?” in *Proceedings of the 11th Working Conference on Mining Software Repositories*. Hyderabad India : ACM, May 2014, pp. 202–211. [Online]. Available : https://dl.acm.org/doi/10.1145/2597073.2597082 10

[31] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, “The impact of code review coverage and code review participation on software quality : a case study of the qt, VTK, and ITK projects,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA : Association for Computing Machinery, May 2014, pp. 192–201. [Online]. Available : https://doi.org/10.1145/2597073.2597076 10

[32] G. Bavota and B. Russo, “Four eyes are better than two : On the impact of code reviews on software quality,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Bremen, Germany : IEEE, Sep. 2015, pp. 81–90. [Online]. Available : http://ieeexplore.ieee.org/document/7332454/ 10

[33] R. Morales, S. McIntosh, and F. Khomh, “Do code review practices impact design quality? A case study of the Qt, VTK, and ITK projects,” in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Montreal, QC, Canada : IEEE, Mar. 2015, pp. 171–180. [Online]. Available : http://ieeexplore.ieee.org/document/7081827/ 10

[34] J. O. Coplien and N. B. Harrison, *Organizational Patterns of Agile Software Development*. Prentice-Hall, Inc., Jul. 2004. 10

[35] P. C. Rigby and C. Bird, “Convergent contemporary software peer review practices,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. Saint Petersburg Russia : ACM, Aug. 2013, pp. 202–212. [Online]. Available : https://dl.acm.org/doi/10.1145/2491411.2491444 12

[36] P. C. Rigby, “Understanding Open Source Software Peer Review : Review Processes, Parameters and Statistical Models, and Underlying Behaviours and Mechanisms,” Ph.D. dissertation, University of Victoria, 2004. 12

[37] J. Czerwonka, M. Greiler, and J. Tilford, “Code Reviews Do Not Find Bugs. How the Current Code Review Best Practice Slows Us Down,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, May 2015, pp. 27–28, iSSN : 1558-1225. [Online]. Available : https://ieeexplore.ieee.org/abstract/document/7202946 12

[38] O. Kononenko, O. Baysal, and M. W. Godfrey, “Code review quality : how developers see it,” in *Proceedings of the 38th International Conference on Software Engineering*. Austin Texas : ACM, May 2016, pp. 1028–1038. [Online]. Available : https://dl.acm.org/doi/10.1145/2884781.2884840 12

[39] P. C. Rigby and M.-A. Storey, “Understanding broadcast based peer review on open source software projects,” in *Proceedings of the 33rd International Conference on Software Engineering*. Waikiki, Honolulu HI USA : ACM, May 2011, pp. 541–550. [Online]. Available : https://dl.acm.org/doi/10.1145/1985793.1985867 12

[40] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, “Towards Automating Code Review Activities,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, May 2021, pp. 163–174, iSSN : 1558-1225. [Online]. Available : https://ieeexplore.ieee.org/abstract/document/9402025 12

[41] S. C. Johnson, “Lint, a C Program Checker,” Jul. 1978. 13

[42] M. Christakis and C. Bird, “What Developers Want and Need from Program Analysis : An Empirical Study,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA : ACM, 2016, pp. 332–343. [Online]. Available : http://doi.acm.org/10.1145/2970276.2970347 15

[43] P. Devanbu, T. Zimmermann, and C. Bird, “Belief & evidence in empirical software engineering,” in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE ’16. New York, NY, USA : Association for Computing Machinery, May 2016, pp. 108–119. [Online]. Available : https://dl.acm.org/doi/10.1145/2884781.2884812 15

[44] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, “How developers engage with static analysis tools in different contexts,” *Empirical Software Engineering*, vol. 25, no. 2, pp. 1419–1457, Mar. 2020. [Online]. Available : https://doi.org/10.1007/s10664-019-09750-5 15, 16, 27, 38

[45] L.-P. Querel and P. C. Rigby, “Warning-Introducing Commits vs Bug-Introducing Commits : A tool, statistical models, and a preliminary user study,” in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, May 2021, pp. 433–443, iSSN : 2643-7171. [Online]. Available : https://ieeexplore.ieee.org/abstract/document/9463031 15

[46] K. F. Tomasdottir, M. Aniche, and A. van Deursen, “Why and how JavaScript developers use linters,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Urbana, IL : IEEE, Oct. 2017, pp. 578–589. [Online]. Available : http://ieeexplore.ieee.org/document/8115668/ 15, 16, 17

[47] D. A. Tomassi, “Bugs in the wild : examining the effectiveness of static analyzers at finding real-world bugs,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA : Association for Computing Machinery, Oct. 2018, pp. 980–982. [Online]. Available : https://dl.acm.org/doi/10.1145/3236024.3275439 15

[48] D. A. Tomassi and C. Rubio-González, “On the Real-World Effectiveness of Static Bug Detectors at Finding Null Pointer Exceptions,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2021, pp. 292–303, iSSN : 2643-1572. 15

[49] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, “Why Don’t Software Developers Use Static Analysis Tools to Find Bugs?” in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE ’13. Piscataway, NJ, USA : IEEE Press, 2013, pp. 672–681. [Online]. Available : http://dl.acm.org/citation.cfm?id=2486788.2486877 15, 28

[50] A. Trautsch, S. Herbold, and J. Grabowski, “Are automated static analysis tools worth it? An investigation into relative warning density and external software quality on the example of Apache open source projects,” *Empirical Software Engineering*, vol. 28, no. 3, p. 66, Apr. 2023. [Online]. Available : https://doi.org/10.1007/s10664-023-10301-2 15

[51] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, “Analyzing the State of Static Analysis : A Large-Scale Evaluation in Open Source Software,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Suita : IEEE, Mar. 2016, pp. 470–481. [Online]. Available : http://ieeexplore.ieee.org/document/7476667/ 16

[52] M. Ochodek, M. Staron, D. Bargowski, W. Meding, and R. Hebig, "Using machine learning to design a flexible LOC counter," in *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, Feb. 2017, pp. 14–20. [Online]. Available : https://ieeexplore.ieee.org/abstract/document/7882011 17, 56

[53] C. Latappy, T. Degueule, J.-R. Falleri, R. Robbes, X. Blanc, and C. Teyton, "What the Fix? A Study of ASAT Rules Documentation," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension, ICPC 2024, Lisbon, Portugal, April 15-16, 2024*. ACM, 2024. [Online]. Available : http://arxiv.org/abs/2402.08270 25

[54] J. Smith, L. N. Q. Do, and E. Murphy-Hill, "Why Can't Johnny Fix Vulnerabilities : A Usability Evaluation of Static Analysis Tools for Security," in *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*, 2020, pp. 221–238. [Online]. Available : https://www.usenix.org/conference/soups2020/presentation/smith 26

[55] R. Watson, M. Stamnes, J. Jeannot-Schroeder, and J. H. Spyridakis, "API documentation and software community values : a survey of open-source API documentation," in *Proceedings of the 31st ACM international conference on Design of communication*. Greenville North Carolina USA : ACM, Sep. 2013, pp. 165–174. [Online]. Available : https://dl.acm.org/doi/10.1145/2507065.2507076 26

[56] D. Binkley, M. Davis, D. Lawrie, and C. Morrell, "To camelcase or under_score," in *2009 IEEE 17th International Conference on Program Comprehension*, May 2009, pp. 158–167, iSSN : 1092-8138. 26

[57] J. Novak, A. Krajnc, and R. Žontar, "Taxonomy of static code analysis tools," in *The 33rd International Convention MIPRO*, May 2010, pp. 418–422. 27

[58] L. N. Q. Do, J. R. Wright, and K. Ali, "Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 835–847, Jun. 2020, conference Name : IEEE Transactions on Software Engineering. 27

[59] P. L. Gorski, Y. Acar, L. Lo Iacono, and S. Fahl, "Listen to Developers! A Participatory Design Study on Security Warnings for Cryptographic APIs," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. Honolulu HI USA : ACM, Apr. 2020, pp. 1–13. [Online]. Available : https://dl.acm.org/doi/10.1145/3313831.3376142 29

[60] N. J. Kipyegen and W. P. Korir, "Importance of software documentation," *International Journal of Computer Science Issues (IJCSI)*, vol. 10, no. 5, p. 223, 2013, publisher : International Journal of Computer Science Issues (IJCSI). 30

[61] A. Forward and T. C. Lethbridge, "The Relevance of Software Documentation, Tools and Technologies : A Survey," in *Proceedings of the 2002 ACM Symposium on Document Engineering*, ser. DocEng '02. New York, NY, USA : ACM, 2002, pp. 26–33. [Online]. Available : http://doi.acm.org/10.1145/585058.585065 30

[62] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza, "Software Documentation Issues Unveiled," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, May 2019,

pp. 1199–1210, iSSN : 1558-1225. [Online]. Available : https://ieeexplore.ieee.org/abstract/document/8811931 30

[63] T. Zimmermann, "Card-sorting : From text to themes," in *Perspectives on Data Science for Software Engineering*, T. Menzies, L. Williams, and T. Zimmermann, Eds. Boston : Morgan Kaufmann, Jan. 2016, pp. 137–141. [Online]. Available : https://www.sciencedirect.com/science/article/pii/B9780128042069000271 36, 37, 51

[64] R. Passonneau, "Measuring Agreement on Set-valued Items (MASI) for Semantic and Pragmatic Annotation," in *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*. Genoa, Italy : European Language Resources Association (ELRA), May 2006. [Online]. Available : http://www.lrec-conf.org/proceedings/lrec2006/pdf/636_pdf.pdf 39

[65] K. N, "Attractive Quality and Must-Be Quality," *Journal of the Japanese Society for Quality Control*, vol. 31, no. 4, pp. 147–156, 1984. [Online]. Available : https://cir.nii.ac.jp/crid/1572261550744179968 44

[66] A. Begel and T. Zimmermann, "Analyze this! 145 questions for data scientists in software engineering," in *Proceedings of the 36th International Conference on Software Engineering*. Hyderabad India : ACM, May 2014, pp. 12–23. [Online]. Available : https://dl.acm.org/doi/10.1145/2568225.2568233 44

[67] G. Guest, K. M. MacQueen, and E. E. Namey, "Applied Thematic Analysis," Oct. 2023. [Online]. Available : https://uk.sagepub.com/en-gb/eur/applied-thematic-analysis/book233379 45, 51

[68] R. Dowling, "Power, subjectivity and ethics in qualitative research," in *Qualitative research methods in human geography*, I. Hay, Ed. South Melbourne, Vic. : Oxford University Press, 2005, pp. 19–29. 51

[69] C. Latappy, Q. Perez, T. Degueule, J.-R. Falleri, C. Urtado, S. Vauttier, X. Blanc, and C. Teyton, "MLinter : Learning Coding Practices from Examples—Dream or Reality?" in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2023, pp. 795–804, iSSN : 2640-7574. [Online]. Available : https://ieeexplore.ieee.org/abstract/document/10123453 54

[70] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan, "Pythia : AI-assisted Code Completion System," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '19. New York, NY, USA : Association for Computing Machinery, Jul. 2019, pp. 2727–2735. [Online]. Available : https://doi.org/10.1145/3292500.3330699 55

[71] A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J. V. Franco, and M. Allamanis, "Fast and Memory-Efficient Neural Code Completion," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, May 2021, pp. 329–340, iSSN : 2574-3864. 55

[72] M. Ochodek, R. Hebig, W. Meding, G. Frost, and M. Staron, "Recognizing lines of code violating company-specific coding guidelines using machine learning : A Method and Its Evaluation," *Empirical Software Engineering*, vol. 25, no. 1, pp. 220–265, Jan. 2020. [Online]. Available : https://doi.org/10.1007/s10664-019-09769-8 55, 56, 60, 61, 63, 65, 73, 74, 78

[73] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT : A Pre-Trained Model for Programming and Natural Languages," Sep. 2020, arXiv :2002.08155 [cs]. [Online]. Available : http://arxiv.org/abs/2002.08155 56, 61

[74] A. A. Freitas, "Comprehensible classification models : a position paper," *ACM SIGKDD Explorations Newsletter*, vol. 15, no. 1, pp. 1–10, Mar. 2014. [Online]. Available : https://doi.org/10.1145/2594473.2594475 57

[75] L. Breiman, J. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. New York : Chapman and Hall/CRC, Oct. 2017. 57

[76] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001. [Online]. Available : https://doi.org/10.1023/A:1010933404324 57

[77] Y. Fu, X. Zhu, and B. Li, "A survey on instance selection for active learning," *Knowledge and Information Systems*, vol. 35, no. 2, pp. 249–283, May 2013. [Online]. Available : https://doi.org/10.1007/s10115-012-0507-8 57

[78] H. S. Seung, M. Opper, and H. Sompolinsky, "Query by committee," in *Proceedings of the fifth annual workshop on Computational learning theory*, ser. COLT '92. New York, NY, USA : Association for Computing Machinery, Jul. 1992, pp. 287–294. [Online]. Available : https://dl.acm.org/doi/10.1145/130385.130417 57

[79] M. Fowler, *Refactoring : Improving the Design of Existing Code*. Addison-Wesley Professional, Nov. 2018, google-Books-ID : 2H1_DwAAQBAJ. 58

[80] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A Bayesian Approach for the Detection of Code and Design Smells," in *2009 Ninth International Conference on Quality Software*. Jeju, Korea (South) : IEEE, Aug. 2009, pp. 305–314. [Online]. Available : http://ieeexplore.ieee.org/document/5381430/ 59

[81] N. Moha, Y.-g. Gueheneuc, and P. Leduc, "Automatic Generation of Detection Algorithms for Design Defects," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. Tokyo : IEEE, 2006, pp. 297–300. [Online]. Available : http://ieeexplore.ieee.org/document/4019591/ 59

[82] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aïmeur, "Support vector machines for anti-pattern detection," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. Essen Germany : ACM, Sep. 2012, pp. 278–281. [Online]. Available : https://dl.acm.org/doi/10.1145/2351676.2351723 59

[83] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Meur, "DECOR : A Method for the Specification and Detection of Code and Design Smells," *IEEE Transactions on Software Engineering*, vol. 36, pp. 20–36, Jan. 2010. 59

[84] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mäntylä, "Code Smell Detection : Towards a Machine Learning-Based Approach," in *2013 IEEE International Conference on Software Maintenance*, Sep. 2013, pp. 396–399, iSSN : 1063-6773. [Online]. Available : https://ieeexplore.ieee.org/abstract/document/6676916 59

[85] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, Jun. 2016. [Online]. Available : http://link.springer.com/10.1007/s10664-015-9378-4 59

[86] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques : Are we there yet?" in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Campobasso : IEEE, Mar. 2018, pp. 612–621. [Online]. Available : http://ieeexplore.ieee.org/document/8330266/ 59

[87] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, "On the diffuseness and the impact on maintainability of code smells : a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, Jun. 2018. [Online]. Available : http://link.springer.com/10.1007/s10664-017-9535-z 59

[88] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection : A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–138, Apr. 2019. [Online]. Available : https://www.sciencedirect.com/science/article/pii/S0950584918302623 59

[89] T. Chappelly, C. Cifuentes, P. Krishnan, and S. Gevay, "Machine learning for finding bugs : An initial report," in *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. Klagenfurt, Austria : IEEE, Feb. 2017, pp. 21–26. [Online]. Available : http://ieeexplore.ieee.org/document/7882012/ 60, 61

[90] Q. Mi, J. Keung, Y. Xiao, S. Mensah, and Y. Gao, "Improving code readability classification using convolutional neural networks," *Information and Software Technology*, vol. 104, pp. 60–71, Dec. 2018. [Online]. Available : https://linkinghub.elsevier.com/retrieve/pii/S0950584918301496 60

[91] A. Kovačević, J. Slivka, D. Vidaković, K.-G. Grujić, N. Luburić, S. Prokić, and G. Sladić, "Automatic detection of Long Method and God Class code smells through neural source code embeddings," *Expert Systems with Applications*, vol. 204, p. 117607, Oct. 2022. [Online]. Available : https://www.sciencedirect.com/science/article/pii/S0957417422009186 60

[92] J. Pennington, R. Socher, and C. Manning, "GloVe : Global Vectors for Word Representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar : Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. [Online]. Available : https://aclanthology.org/D14-1162 61

[93] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'13. Red Hook, NY, USA : Curran Associates Inc., Dec. 2013, pp. 3111–3119. 61

[94] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT : Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proceedings*

of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics : Human Language Technologies, Volume 1 (Long and Short Papers). Minneapolis, Minnesota : Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available : https://aclanthology.org/N19-1423 61, 62, 67

[95] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is All you Need," in *Advances in Neural Information Processing Systems*, vol. 30. Curran Associates, Inc., 2017. [Online]. Available : https://papers.nips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html 61

[96] M. Hoang, O. A. Bihorac, and J. Rouces, "Aspect-Based Sentiment Analysis using BERT," in *Proceedings of the 22nd Nordic Conference on Computational Linguistics*. Turku, Finland : Linköping University Electronic Press, Sep. 2019, pp. 187–196. [Online]. Available : https://aclanthology.org/W19-6120 62

[97] S. Garg and G. Ramakrishnan, "BAE : BERT-based Adversarial Examples for Text Classification," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Online : Association for Computational Linguistics, Nov. 2020, pp. 6174–6181. [Online]. Available : https://aclanthology.org/2020.emnlp-main.498 62

[98] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq : Generating Sequences from Structured Representations of Code," in *International Conference on Learning Representations*, 2019. [Online]. Available : https://openreview.net/forum?id=H1gKYo09tX 62

[99] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec : learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 40 :1–40 :29, Jan. 2019. [Online]. Available : https://doi.org/10.1145/3290353 62

[100] X. Zhou, D. Han, and D. Lo, "Assessing Generalizability of CodeBERT," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2021, pp. 425–436, iSSN : 2576-3148. 62

[101] E. Mashhadi and H. Hemmati, "Applying CodeBERT for Automated Program Repair of Java Simple Bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, May 2021, pp. 505–509, iSSN : 2574-3864. 62

[102] S. Fatima, T. A. Ghaleb, and L. Briand, "Flakify : A Black-Box, Language Model-Based Predictor for Flaky Tests," *IEEE Transactions on Software Engineering*, pp. 1–17, 2022, conference Name : IEEE Transactions on Software Engineering. 62

[103] C. Pan, M. Lu, and B. Xu, "An Empirical Study on Software Defect Prediction Using CodeBERT Model," *Applied Sciences*, vol. 11, no. 11, p. 4793, Jan. 2021, number : 11 Publisher : Multidisciplinary Digital Publishing Institute. [Online]. Available : https://www.mdpi.com/2076-3417/11/11/4793 62

[104] U. Ferreira Campos, G. Smethurst, J. P. Moraes, R. Bonifácio, and G. Pinto, "Mining Rule Violations in JavaScript Code Snippets," in *2019 IEEE/ACM 16th International*

Conference on Mining Software Repositories (MSR), May 2019, pp. 195–199, iSSN : 2574-3864. 63

[105] B. Krawczyk, "Learning from imbalanced data : open challenges and future directions," *Progress in Artificial Intelligence*, vol. 5, no. 4, pp. 221–232, Nov. 2016. [Online]. Available : https://doi.org/10.1007/s13748-016-0094-0 65, 67

[106] B. Efron, "Estimating the Error Rate of a Prediction Rule : Improvement on Cross-Validation," *Journal of the American Statistical Association*, vol. 78, no. 382, pp. 316–331, 1983, publisher : [American Statistical Association, Taylor & Francis, Ltd.]. [Online]. Available : https://www.jstor.org/stable/2288636 67

[107] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An Empirical Comparison of Model Validation Techniques for Defect Prediction Models," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 1–18, Jan. 2017. [Online]. Available : http://ieeexplore.ieee.org/document/7497471/ 67

[108] H. B. Mann and D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, Mar. 1947, publisher : Institute of Mathematical Statistics. [Online]. Available : https://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-18/issue-1/On-a-Test-of-Whether-one-of-Two-Random-Variables/10.1214/aoms/1177730491.full 69

[109] D. S. Kerby, "The Simple Difference Formula : An Approach to Teaching Nonparametric Correlation," *Comprehensive Psychology*, vol. 3, p. 11.IT.3.1, Jan. 2014, publisher : SAGE Publications Inc. [Online]. Available : https://journals.sagepub.com/doi/abs/10.2466/11.IT.3.1 69

[110] D. Kahneman and A. Tversky, "On the psychology of prediction : Psychological Review," *Psychological Review*, vol. 80, no. 4, pp. 237–251, Jul. 1973, publisher : American Psychological Association. [Online]. Available : https://search.ebscohost.com/login.aspx?direct=true&db=pdh&AN=1974-02325-001&lang=fr&site=ehost-live 74

[111] M. Bar-Hillel, "The base-rate fallacy in probability judgments," *Acta Psychologica*, vol. 44, no. 3, pp. 211–233, May 1980. [Online]. Available : https://linkinghub.elsevier.com/retrieve/pii/0001691880900463 74

[112] V. Markovtsev, W. Long, H. Mougard, K. Slavnov, and E. Bulychev, "STYLE-ANALYZER : fixing code style inconsistencies with interpretable unsupervised algorithms," in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR '19. Montreal, Quebec, Canada : IEEE Press, May 2019, pp. 468–478. [Online]. Available : https://doi.org/10.1109/MSR.2019.00073 79

[113] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix : Learning to Fix Bugs Automatically," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019, place : New York, NY, USA Publisher : Association for Computing Machinery. [Online]. Available : https://doi.org/10.1145/3360585 79

[114] D. Marcilio, C. A. Furia, R. Bonifácio, and G. Pinto, "SpongeBugs : Automatically generating fix suggestions in response to static code analysis warnings," *Journal of Systems and Software*, vol. 168, p. 110671, Oct. 2020. [Online]. Available : https://www.sciencedirect.com/science/article/pii/S016412122030128X 79

[115] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. Le Traon, "Mining Fix Patterns for FindBugs Violations," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 165–188, Jan. 2021, conference Name : IEEE Transactions on Software Engineering. 79

[116] B. Loriot, F. Madeiral, and M. Monperrus, "Styler : learning formatting conventions to repair Checkstyle violations," *Empirical Software Engineering*, vol. 27, no. 6, p. 149, Aug. 2022. [Online]. Available : https://doi.org/10.1007/s10664-021-10107-0 79

[117] J. Andersen and J. L. Lawall, "Generic patch inference," *Automated Software Engineering*, vol. 17, no. 2, pp. 119–148, Jun. 2010. [Online]. Available : https://doi.org/10.1007/s10515-010-0062-z 79

[118] N. Meng, M. Kim, and K. S. McKinley, "Lase : Locating and applying systematic edits by learning from examples," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 502–511, iSSN : 1558-1225. 79

[119] J. Jiang, L. Ren, Y. Xiong, and L. Zhang, "Inferring Program Transformations From Singular Examples via Big Code," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2019, pp. 255–266, iSSN : 2643-1572. 79

[120] R. Bavishi, H. Yoshida, and M. R. Prasad, "Phoenix : automated data-driven synthesis of repairs for static analysis violations," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA : Association for Computing Machinery, Aug. 2019, pp. 613–624. [Online]. Available : https://doi.org/10.1145/3338906.3338952 79

[121] L. Serrano, V.-A. Nguyen, F. Thung, L. Jiang, D. Lo, J. Lawall, and G. Muller, "SPINFER : Inferring Semantic Patches for the Linux Kernel," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 235–248. [Online]. Available : https://www.usenix.org/conference/atc20/presentation/serrano 79

[122] S. A. Haryono, F. Thung, D. Lo, J. Lawall, and L. Jiang, "Characterization and Automatic Updates of Deprecated Machine-Learning API Usages," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2021, pp. 137–147, iSSN : 2576-3148. 79

[123] S. A. Haryono, F. Thung, D. Lo, L. Jiang, J. Lawall, H. J. Kang, L. Serrano, and G. Muller, "AndroEvolve : automated Android API update with data flow analysis and variable denormalization," *Empirical Software Engineering*, vol. 27, no. 3, p. 73, Mar. 2022. [Online]. Available : https://doi.org/10.1007/s10664-021-10096-0 79

[124] J. A. Prenner and R. Robbes, "Out of Context : How important is Local Context in Neural Program Repair?" in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA :

Association for Computing Machinery, Apr. 2024, pp. 1–13. [Online]. Available : https://doi.org/10.1145/3597503.3639086 79

Table des figures

Liste des tableaux