UNIVERSITÉ DE MONTRÉAL

RECOMMENDING WHEN DESIGN TECHNICAL DEBT SHOULD BE SELF-ADMITTED

CÉDRIC NOISEUX
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AÔUT 2017

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

RECOMMENDING WHEN DESIGN TECHNICAL DEBT SHOULD BE SELF-ADMITTED

présenté par: NOISEUX Cédric
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

M. NOM Prénom, Doct., président
Mme NOM Prénom, Ph. D., membre et directrice de recherche
M. NOM Prénom, Ph. D., membre

DEDICATION

À tous mes amis du labos,
vous me manquerez. . . FACULTATIF

ACKNOWLEDGEMENTS

Texte. FACULTATIF

RÉSUMÉ

Les Technical Debts (TD) sont des solutions temporaires et peu optimales introduites dans le code source d'un logiciel informatique pour corriger un problème rapidement au détriment de la qualité logiciel. Cette pratique est répendu pour diverses raisons: rapidité d'implémentation, conception initiale des composantes, connaissances faibles du projet, inexpérience du développeur ou pression face aux dates limites. Les TD peuvent s'avérer utiles à court terme, mais excessivement dommageables pour un logiciel et accaparantes au niveau du temps perdu. En effet, le temps requis pour corriger des problème et concevoir du code de qualité n'est souvent pas compatible avec le cycle de développement d'un projet. C'est pourquoi le sujet des TD a été analysé dans de nombreuses études déjà, plus spécifiquement dans l'optique de les détecter et les identifier.

Une approche populaire et récente est d'identifier les TD qui sont consciemment admises dans le code. La particularités de ces dettes, en comparaison aux TD, est qu'elles sont explicitement documentées par commentaires et intentionnellement introduites dans le code source. Les Self-Admitted Technical Debts (SATD) ne sont pas rares dans les projets logiciels et ont déjà été largement étudiées concernant leur diffusion, impact sur la qualité logiciel, criticité, évolution et acteurs. Diverses méthodes de détection sont présentement utilisées pour identifier les SATD mais toutes demeurent sujet àmélioration. Par exemple, l'utilisation de mots clés (*e.g.: hack, fixme, todo, ugly, etc.*) dans les commentaires en relation avec les dettes techniques ou l'utilisation du Natural Language Processing (NLP) combiné à l'apprentissage machine. Donc, notre étude analyse dans quelle mesure des dettes techniques ayant déjà été consciemment admises (SATD) peuvent être utilisées pour fournir des recommandations aux développeurs lorsqu'ils écrivent du nouveau code. En d'autres termes, le but est d'être capable de suggérer quand admettre des dettes techniques ou quand améliorer du nouveau code en processus de rédaction.

Pour atteindre ce but, une approche d'apprentissage machine a été élaborée, nommée TEchnical Debt IdentificatiOn System (TEDIOUS), utilisant comme variables indépendantes divers types de métriques d'entrées au niveau des méthodes de manière à pouvoir classifier des dettes techniques de conception avec comme oracle des SATD connus. Le modèle a été entrainé et évaluer sur neuf projets Java *open source* contenant des SATD précédemment étiquetés. En d'autres termes, notre approche vise a prédire précisément les TD dans les projets logiciels.

TEDIOUS fonctionne au niveau de granularité des méthodes, en d'autres termes, il

détecte si une méthode contient une dette de conception ou non. Il a été conçu ainsi car les délopppeur ont d'avantage tendance à admettre des dettes techniques au niveau des méthodes ou des blocs de code. Les TD peuvent être classifiés selon différents types: conception, requis, test, code et documentation. Les dettes de conception seulement ont été considérées car elles forment la majorité et analyser chaque type demanderait une analyse personnalisée.

TEDIOUS est entraîné avec des données étiquetées comme étant des SATD ou non et testé avec des données sans étiquettes. Les données étiquettées contiennent des méthodes marquées comme étant des SATD, obtenues à partir de neuf projets logiciels analysés par un autre groupe de recherche utilisant une approche NLP et validé manuellement. Les projets sont de différentes dimensions (*e.g.:* number of classes, methods, comments, etc.) et contiennent différentes proportions de dettes de conception. Des métriques sont extraits des données étiquettées: métriques de code source, métriques de lisibilité et alertes générées par des outils d'analysis statiques. Neuf métriques de code source ont été retenus pour fournir un portrait de la dimension, du couplage, de la complexité et du nombre de composantes des méthodes. Le métrique de lisibilité prend en considération, entre autres, les retraits, la longueur des lignes et des identifiants. Deux outils d'analyse statique ont été utilisés pour cerner de faibles pratiques de codage.

Le prétraitement des métriques est appliqué pour retirer ceux étant superflus et garder ceux étant les plus pertinents par rapport à la variable dépendante. Certaines caractéristiques sont fortement corrélées entre elles et il serait redondant de toutes les conserver. D'autres subissent aucune ou trop de variations dans le contexte de notre ensemble de données, elles ne seraient pas utiles pour concevoir un predicteur et sont donc supprimées également. De plus, les métriques sont normalisés pour atteindre des valeurs de performance appréciables au niveau de la predictions inter-projets. Cette normalisation est nécessaire car le code source des projets varie en terme de dimensions et complexité. Finalement, l'ensemble de données est déséquilibré, ce qui signifie que le nombre de méthodes étiqueté comme étant un SATD est faible. Un suréchantillonnage a été appliqué sur la classe en minorité pour générer de nouvelles instances artificielles à partir de celles existantes.

Les modèles d'apprentissage machine sont construits à partir de l'ensemble d'entrainement et les predictions sont evaluées à partir de l'ensemble de test. Cinq types de *machine learners* ont été testés: Decision Trees (J48), Bayesian classifiers, Random Forests, Random Trees and Bagging with Decision Trees. Ces modèles ont été retenus pour obtenir une grande variété de résultats, provenant de différents algorithmes considérés comme étant les plus appropriés et pécis dans le contexte de notre étude.

Globalement, le but de notre étude est d'évaluer la performance de prédiction des SATD

selon notre approche. La vision poursuivie est de favoriser une meilleure compréhension et maintenabilité du code source. La perspective est d'être capable de suggérer quand admettre un TD ayant été identifié précédemment. Trois questions de recherche sont abordées:

- **RQ1**: Comment TEDIOUS performe dans la recommandation de SATD intra-projet?
- **RQ2**: Comment TEDIOUS performe dans la recommandation de SATD inter-projet?
- **RQ3**: Comment un *smell detector* au niveau des méthodes se compare avec TEDIOUS?

Pour répondre à **RQ1**, une validation croisée de dix échantillons a été réalisé sur tous les projets, ce qui signifie que chaque modèle est entrainé sur 90% de toutes les méthodes d'un projet et testé sur 10% de ceux-ci. Le processus est répété dix fois pour réduire l'effet du hasard. Une approche similaire est suivie pour **RQ2**, un modèle est entraîné avec 8 projets et testé avec 1.

Pour évaluer le performance de TEDIOUS, des métriques standards tels que la pécision, le rappel et la mesure F1 sont calculés sur la classe SATD. Ces métriques sont basés sur la quantité de vrais positifs, faux positifs et faux négatifs. Pour compléter cette évaluation, la précision, le Matthews Correlation Coefficient (MCC) et le Receiving Operating Characteristics (ROC) Area Under the Curve (AUC) sont calculés, en partie pour tenir compte du nombre de vrais négatifs. Ce qui est visé comme performance des modèles d'apprentissage est un équilibre entre précision et rappel, de suggérer *correctement* le plus grand nombre possible de TD à admettre. MCC et AUC sont des indicateurs utiles pour réduire l'effet du hasard. L'importance des métriques d'entrées est aussi considérée pour évaluer les modèles.

Pour répondre à **RQ3**, la performance d'un *smell detector*, DEtection & CORrection (DECOR), a été évalué selon sa capacité à classifier des méthodes étiquettées SATD comme étant des dettes techniques. Des odeurs au niveau des méthodes seulement ont été analysées, tout comme TEDIOUS. Finalement, quelques faux positifs et faux négatifs ont été discuté qualitativement pour exprimer les limites de notre approche.

Pour *RQ1*, les résultats ont démontré que le classificateur Random Forest a atteint les meilleurs performances dans la recommendation de dettes de conception. La précision moyenne ayant été obtenue est 49.97% et le rappel 52.19%. Les valeurs de MCC et AUC pour chaque projet indiquaient la présence de classificateur de qualité. Équilibrer l'ensemble de données a permis d'accroître le rappel au détriment de la précision. La lisibilité, complexité et taille du code source ont joué un rôle significatif dans l'élaboration des prédicteurs.

Pour *RQ2*, la prédiction inter-projet augmente la performance des prédicteurs en comparaison à la validation croisée sur des projets singuliers, grâce à un ensemble d'entraînement

plus large et diversifié. La précision moyenne ayant été obtenue est 67.22% et le rappel 54.89%. Les valeurs de MCC et AUC indiquaient encore une fois la présence de classificateurs de qualité. Encore une fois, la lisibilité, la taille et la compléxité ont joué un rôle important dans l'élaboration des prédicteurs.

Pour *RQ3*, Long Method (LM) et Long Parameter List (LP) été éevalués par DECOR, de manière similaire aux métriques Lines Of Code (LOC) et nombre de paramètres, qui ont joué un rôle important dans l'entraînement des machines d'apprentissage. Toutefois, les performances de DECOR ne se sont pas avérées aussi bonnes que pour TEDIOUS. Le score F_1 pour l'union de LM et LP n'a pu surpasser 22% et la valeur MCC indiquait une faible corrélation de prédiction.

Quelques menaces peuvent affecter la validité de notre étude. Pour les menaces de *validité de la construction*, les erreurs de mesure des SATD étiquettés et des métriques représentent un probléme. Pour les menaces de *validité interne*, les paramètres par défaut seulement ont été appliqués aux machines d'apprentissage. Une optimisation des paramètres pourrait être appliquée pour obtenir de meilleures configurations. L'utilisation intelligente des diagnostiques de performance (AUC, MCC) a permis de réduire les menaces de *validité des conclusions*. Pour les menaces de *validité de la fiabilité*, tous les détails nécessaires sont fournis pour répliquer notre étude. Pour les menaces de *validité externe*, il ne peut être garanti que nos résultats peuvent être généralisés à tous les projets Java, considérant l'échelle réduite de notre étude. Notre approche devrait être étendue à d'avantage de projets, domaines et langages de programmation.

Ce mémoire décrit TEDIOUS, une approche d'apprentissage machine au niveau des méthodes conçu pour recommander quand un développeur devrait admettre un TD de conception, basé sur la taille, la complexité, la lisibilité et l'analyse statique du code source. Les performances intra-projet basées sur 9 projets Java *open source* ont mené à des résultats prometteurs: environ 50% de précision, 52% de rappel et 93% de justesse. Les performances inter-projet se sont avérées encore meilleures: environ 67% de précision, 55% de rappel et 92% de justesse. L'ensemble de données grandement déséquilibré a représenté le plus grand obstacle dans l'obtention de valeurs de performance élevées. Pour les projets les plus volumineux, une précision et un rappel supérieurs à 88% ont été obtenus.

TEDIOUS pourrait être utilisé pour diverses applications. Il pourrait être utilisé comme système de recommandation pour savoir quand documenter des TD nouvellement introduits. Deuxièmement, il pour aider à personnaliser les alertes relevées pour les outils d'analyse statique. Troisièmement, il pourrait complémenter des détecteurs d'odeurs préexistants pour améliorer leur performance, comme DECOR. Quant aux travaux futurs, un plus grand

ensemble de données sera étudié pour savoir si ajouter d'avantage d'information est bénéfique pour les performances de notre approche. De plus, nous planifions d'étendre TEDIOUS à la recommandation de plus de types de dettes techniques.

ABSTRACT

TD are temporary solutions, or workarounds, introduced in portions of software systems in order to fix a problem rapidly at the expense of quality. Such practices are widespread for various reasons: rapidity of implementation, initial conception of components, lack of system's knowledge, developer inexperience or deadline pressure. Even though technical debts can be useful on a short term basis, they can be excessively damaging and time consuming in the long run. Indeed, the time required to fix problems and design code is frequently not compatible with the development life cycle of a project. This is why the issue has been tackled in various studies, specifically in the aim of detecting these harmful debts.

One recent and popular approach is to identify technical debts which are self-admitted. The particularity of these debts, in comparison to TD, is that they are explicitly documented with comments and intentionally introduced in the source code. SATD are not uncommon in software projects and have already been extensively studied concerning their diffusion, impact on software quality, criticality, evolution and actors. Various detection methods are currently used to identify SATD but are still subject to improvement. For example, using keywords (*e.g.: hack, fixme, todo, ugly, etc.*) in comments linking to a technical debt or using NLP in addition to machine learners. Therefore, this study investigates to what extent previously self-admitted technical debts can be used to provide recommendations to developers writing new source code. The goal is to be able to suggest when to "self-admit" technical debts or when to improve new code being written.

To achieve this goal, a machine learning approach was conceived, named TEDIOUS, using various types of method-level input features as independent variables to classify design technical debts using self-admitted technical debts as oracle. The model was trained and assessed on nine open source Java projects which contained previously tagged SATD. In other words, our proposed machine learning approach aims to accurately predict technical debts in software projects.

TEDIOUS works at method-level granularity, in other words, it can detect whether a method contains a design debt or not. It was designed this way because developers are more likely to self-admit technical debt for methods or blocks of code. TD can be classified in different types: design, requirement, test, code or documentation. Only design debts were considered because they represent the largest fraction and each type would require its own analysis.

TEDIOUS is trained with *labeled data*, projects with labeled SATD, and tested with

unlabeled data. The labeled data contain methods tagged as SATD which were obtained from nine projects analyzed by another research group using a NLP approach and manually validated. Projects are of various sizes (*e.g.:* number of classes, methods, comments, etc.) and contain different proportions of design debts. From the labeled data are extracted various kinds of metrics: source code metrics, readability metrics and warnings raised by static analysis tools. Nine source code metrics were retained to capture the size, coupling, complexity and number of components in methods. The readability metric takes in consideration indents, lines and identifiers length just to name a few features. Two static analysis tools are used to check for poor coding practices.

Feature preprocessing is applied to remove unnecessary features and keep the ones most relevant to the dependent variable. Some features are strongly correlated between each others and keeping all of them is redundant. Other features undergo important or no variations in our dataset, they would not be useful to build a predictor and thus are removed as well. Additionally, to achieve good cross-project predictions, metrics are normalized because the source code of different projects can differ in terms of size and complexity. Finally, the dataset is unbalanced, which means the amount of methods labeled as SATD is small. Over-sampling was applied on the minority class to generate artificial instances from the existing ones.

Machine learnings models are built based on the training set and predictions are evaluated from the test set. Five kinds of machine learners were tested: Decision Trees (J48), Bayesian classifiers, Random Forests, Random Trees and Bagging with Decision Trees. These models were retained to gather a wide variety of results, from different algorithms which were considered the most appropriate and accurate for the context of this study.

Globally, the goal of this study is to assess the SATD prediction performance of our approach. The quality focus is understandability and maintainability of the source code, achieved by tracking existing TD. The perspective is to be able to suggest when to admit those TD. Three research questions are aimed to be addressed:

- **RQ1**: How does TEDIOUS work for recommending SATD within-project?
- **RQ2**: How does TEDIOUS work for recommending SATD across-project?
- **RQ3**: How would a method-level smell detector compare with TEDIOUS?

To address **RQ1**, 10-fold cross validation was performed on all projects, which means a machine learner is trained with 90% of a project's methods and tested with 10% of them. The process is repeated 10 times to reduce the effect of randomness. A similar approach is used for **RQ2**, a machine learner is trained with 8 projects and is tested with 1 project.

To assess the performance of TEDIOUS, standard metrics such as precision, recall and F1 score are computed for the SATD category. These metrics are based on the amount of True Positive (TP), False Positive (FP) and False Negative (FN). To complement the evaluation, accuracy, MCC and ROC AUC are computed, partly to take into account the amount of True Negative (TN). What is aimed for in a machine learning model performance is a balance between precision and recall, to suggest as many *correct* TD to admit as possible. MCC and AUC are useful indicators to reduce the effect of chance. The importance of feature metrics is also taken into account to evaluate the models.

To address **RQ3**, the performance of a smell detector, DECOR, was computed and evaluated in classifying as TD methods labeled as SATD. Only method-level smells were analyzed, similarly to TEDIOUS. Finally, some FP and FN were qualitatively discussed in order to explain the limits of our approach.

For **RQ1**, results showed that Random Forest classifiers achieved the best performance recommending design debts. The average precision obtained is 49.97% and the recall 52.19%. The MCC and AUC values of each project generally indicated healthy classifiers. Balancing the dataset increased recall at the expense of precision and code readability, complexity and size played a significant role in building the predictors.

For **RQ2**, cross-project prediction increased the performance of predictors compared to the standard cross-validation on singular projects because of a larger and more diverse training set. The average precision obtained is 67.22% and the recall 54.89%. The MCC and AUC values still indicated healthy classifiers. Similarly to within project predictions, code readability, size and complexity played the most important role in recommending when to self-admit design TD.

For **RQ3**, LM and LP were the specific smells targeted and evaluated by DECOR, similar to LOC and number of parameters metrics, which played an important role in training machine learners in the context of our study. However, the detectors of DECOR were unable to achieve similar performance as TEDIOUS. The F_1 score for the union of LM and LP couldn't surpass 22% and the MCC value leaned towards a low prediction correlation.

Some threats can affect the validity of our study. For *construct validity* threats, the measurement errors of labeled design SATD and metrics represent an issue. For *internal validity* threats, default parameters only were applied for the machine learners. Some kind of optimization could be applied obtain better machine learner configurations. The proper use of performance diagnostics (AUC, MCC) allowed us to reduce the *conclusion validity* threats. For *reliability validity* threats, all necessary details are provided to replicate our study. For *external validity* threats, it cannot be guaranteed that our results can be generalized to all

Java projects considering the small amount of projects analyzed. Our approach would need to be extended to more projects, domains or programming languages.

This paper describes TEDIOUS, a method-level machine learning approach designed to recommend when a developer should self-admit a design technical debt based on size, complexity, readability metrics, and static analysis tools checks. Within-project performance values based on 9 open source Java projects lead to promising results: about 50% precision, 52% recall and 93% accuracy. Cross-project performance was even more promising: about 67% precision, 55% recall and 92% accuracy. Highly unbalanced data represented the biggest issue in obtaining higher performance values. For bigger projects, precision and recall above 88% were obtained.

Different applications could be made of TEDIOUS. It could be used as a recommendation system for developers to know when to document TD they introduced. Secondly, it could help customize warnings raised by static analysis tools, by learning from previously defined SATD. Thirdly, it could compliment existing smell detectors to improve their performance, like DECOR. As for our future work, a larger dataset will be studied to see if adding more information could be beneficial to our approach. Additionally, we plan to extend TEDIOUS to the recommendation of more types of technical debts.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATION

TD	Technical Debt
SATD	Self-Admitted Technical Debt
NLP	Natural Language Processing
TEDIOUS	TEchnical Debt IdentificatiOn System
TP	True Positive
TN	True Negative
FP	False Positive
FN	False Negative
MCC	Matthews Correlation Coefficient
ROC	Receiving Operating Characteristics
AUC	Area Under the Curve
DECOR	DEtection & CORrection
LOC	Lines Of Code
LM	Long Method
LP	Long Parameter List

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

In today's consumer society, products have to be designed and ready to hit the market as fast as possible in order to stand out from other similar products and generate sells. This pressure to produce can affect the quality, maintainability and functionality of the design. In software engineering, the repercussion of this mindset can be measured with the amount of technical debts present in a project. These TD can go unnoticed, which is the danger behind them, or may be admitted by developers. In fact, studies have been conducted on technical debts that are "self-admitted" by developers, commenting why such code represent an issue or a temporary solution. The subject of this paper is to study how previously self-admitted technical debts can be used to recommend when to admit a newly introduced TD.

1.1 Basic Concepts and Defintions

Technical debts are temporary and less than optimal solutions introduced in the code. They are portions of code that still need to be worked on even though they accomplish the purpose they were written for. As Cunningham first described them, TD is "not quite right code which we postpone making it right" (Cunningham, 1992). For example, TD could be workarounds which don't follow good coding practices, poorly structured or hard-to-read code. By definition, technical debts don't typically cause errors, preventing the code from working, but they can in some circumstances. Various reasons can motivate the introduction of technical debts: to rapidly fix an issue, development team is at the early stages of conception, lack of comprehension, skills or experience (Suryanarayana et al., 2015). TD are introduced throughout the whole development life cycle and under various forms. An ontology and landscape have been built to better define the subject. Design, requirement, code, test and documentation debts represent the main branches of the classification tree (Alves et al., 2014; Izurieta et al., 2012). Each branch can be linked to a specific development stage and to specific criteria. For example, design debt "refers to debt that can be discovered by analyzing the source code by identifying the use of practices which violated the principles of good object-oriented design (e.g. very large or tightly coupled classes)" (Alves et al., 2014).

Other work investigated the perception of developers on technical debts. It was found that the most important source of TD is architectural decisions, that recognizing the phenomenon is essential for communication and that there is a lack of tools to manage those debts (Ernst et al., 2015). Additionally, project teams recognize that this issue is unavoidable and necessary (Lim et al., 2012).

1.2 Éléments de la problématique

environ 3 pages

1.3 Objectifs de recherche

0.5 page

TEDIOUS (TEchnical Debt IdentificatiOn System) Supervised Machine Learning approach Method-level Using various features of code source (independant) Knowledge of previous SATD (dependant) To recommend developpers with TD to be admitted

Purpose 1) Encourage self admitting TD (mainly done by experienced, want new to do too 35) 2) Alternative to smell detectors to give opportunities to improve source code

Difference 1) Method-level instead of class-level metrics (8) because SATD comments at method or block-level mainly 2) Only consider certain types of TD (design debt since largest 27) Other types for future since different analysis

1.4 Plan du mémoire

0.5 page

CHAPTER 2 LITTERATURE REVIEW

TOTAL = 4 pages

2.1 Relationship Between Technical Debt and Source Code Metrics

2.2 Self-Admitted Technical Debt

2.3 Code Smell Detection and Automated Static Analysis Tools

CHAPTER 3 THE APPROACH AND STUDY DEFINITION

TOTAL = 15 pages

3.1 The Approach

2 pages

3.1.1 Features

3 pages

3.1.2 Identification of Self-Admitted Technical Debt

0.5 page

3.1.3 Feature Preprocessing

3 pages

3.1.4 Building and Applying Machine Learning Models

0.5 page

3.2 Study Definition

0.5 page

3.2.1 Dataset

2 pages

3.2.2 Analysis Method

3.5 pages

CHAPTER 4 ANALYSIS OF STUDY RESULTS AND THREATS TO VALIDITY

TOTAL = 18 pages

4.1 Study Results

4.1.1 How does TEDIOUS work for recommending SATD within-project?

4.5 pages

4.1.2 How does TEDIOUS work for recommending SATD across-project?

4 pages

4.1.3 How would a method-level smell detector compare with TEDIOUS?

1.5 pages

4.1.4 Qualitative discussion of false positive and false negatives

4 pages

4.2 Threats to Validity

4.2.1 Construct validity

1 page

4.2.2 Internal validity

1 page

4.2.3 Conclusion validity

1 page

4.2.4 External validity

1 page

CHAPTER 5 CONVOLUTIONAL NEURAL NETWORK WITH COMMENTS AND SOURCE CODE

TOTAL = 15 pages

5.1 Convolutional Neural Network

1 page

5.2 The Approach

1 page

5.2.1 Features

1 page

5.2.2 Identification of Self-Admitted Technical Debt

0.5 page

5.2.3 Word Embeddings

1 page

5.2.4 Building and Applying CNN

2 page

5.3 Study Definition

0.5 page

5.3.1 Dataset

1.5 page

5.3.2 Analysis Method

2 pages

5.4 Study Results

5.4.1 Source Code With Comments

1.5 pages

5.4.2 Source Code Without Comments

1.5 pages

5.4.3 Source Code Partially With Comments

1.5 pages

CHAPTER 6 CONCLUSION

TOTAL = 3 pages

6.1 Summary of Work

1 page

6.2 Limitations of the Proposed Solution

1 page

6.3 Future Work

1 page

BIBLIOGRAPHY

N. S. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, et R. O. Spínola, "Towards an ontology of terms on technical debt", dans *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*. IEEE, 2014, pp. 1–7.

W. Cunningham, "The wycash portfolio management system", dans *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, série OOPSLA '92. New York, NY, USA: ACM, 1992, pp. 29–30. DOI: 10.1145/157709.157715. En ligne: http://doi.acm.org/10.1145/157709.157715

N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, et I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt", dans *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, série ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 50–60. DOI: 10.1145/2786805.2786848. En ligne: http://doi.acm.org/10.1145/2786805.2786848

C. Izurieta, A. Vetrò, N. Zazworka, Y. Cai, C. Seaman, et F. Shull, "Organizing the technical debt landscape", dans *Proceedings of the Third International Workshop on Managing Technical Debt*. IEEE Press, 2012, pp. 23–26.

E. Lim, N. Taksande, et C. Seaman, "A balancing act: what software practitioners have to say about technical debt", *IEEE software*, vol. 29, no. 6, pp. 22–27, 2012.

G. Suryanarayana, G. Samarthyam, et T. Sharma, "Chapter 1 - technical debt", dans *Refactoring for Software Design Smells*, G. Suryanarayana, , G. Samarthyam, et T. Sharma, éds. Boston: Morgan Kaufmann, 2015, pp. 1 – 7. DOI: https://doi.org/10.1016/B978-0-12-801397-7.00001-1. En ligne: http://www.sciencedirect.com/science/article/pii/B9780128013977000011

APPENDIX A DÉMO

Texte de l'annexe A. Remarquez que la phrase précédente se termine par une lettre majuscule suivie d'un point. On indique explicitement cette situation à LaTeX afin que ce dernier ajuste correctement l'espacement entre le point final de la phrase et le début de la phrase suivante.

APPENDIX B ENCORE UNE ANNEXE

Texte de l'annexe B en mode «landscape».

APPENDIX C UNE DERNIÈRE ANNEXE

Texte de l'annexe C.