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 répandu 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 problèmes 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 particularité 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 amélioration. 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 à prédire précisément les TD dans les projets logiciels.

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

tecte si une méthode contient une dette de conception ou non. Il a été conçu ainsi car les développeurs 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 étiquetées 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 étiquetées: métriques de code source, métriques de lisibilité et alertes générées par des outils d'analyse 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. La 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 prédicteur et sont donc supprimées également. De plus, les métriques sont normalisées pour atteindre des valeurs de performance appréciables au niveau de la prédiction inter-projets. Cette normalisation est nécessaire car le code source des projets varie en termes 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 prédictions sont évaluées 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 précis 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 la performance de TEDIOUS, des métriques standards tels que la précision, le rappel et la mesure F1 sont calculés sur la classe SATD. Ces métriques sont basées 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 étiquetées 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 meilleures performances dans la recommandation 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 complexité ont joué un rôle important dans l'élaboration des prédicteurs.

Pour *RQ3*, Long Method (LM) et Long Parameter List (LP) été évalués 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.

Nous avons déjà conçu et testé une nouvelle approche pour améliorer la performance de TEDIOUS. Elle est similaire à la précédente car elle est basée sur l'apprentissage machine, elle fonctionne au niveau des méthodes et elle utilise des méthodes étiquetées comme des SATD de conception. Toutefois, le modèle de système d'apprentissage favorisé est le Convolutionnal Neural Network (CNN), implémenté spécifiquement pour le contexte de notre étude. Les variables indépendantes ne sont pas des caractéristiques du code source mais plutôt le code source *lui-même*. Comme les caractéristiques de l'approche précédente, le code source a aussi été prétraité, il a été transformé en jetons et un *word embedding* a été réalisé. Le CNN a été testé sur le même ensemble de données, intra-projet et inter-projet, mais aussi selon différentes variables indépendantes, code source avec, sans et partiellement avec commentaires. Une méthode d'analyse similaire a aussi été suivie, utilisant la validation croisée et les métriques de performance standards. **EDIT The performance values were ... and ... They were better than TEDIOUS in the way that ... Donner un appercu de tous les resultats ... EDIT**

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. Pour l'approche utilisant les caractéristiques 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. Pour l'approche utilisant le code source lui-même, **EDIT The performance values were ... and ... They were better than TEDIOUS**

in the way that ... Donner un appercu de tous les resultats ... EDIT

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.

We already designed and tested a new approach in order to improve the performance of TEDIOUS. It is similar to the previous one because it is machine learning based, it works at method-level and it uses design SATD tagged methods. However, the machine learning model favored is a CNN which was implemented for the context of our study. The independent variables are not source code features but rather the source code *itself*. Like features from the previous approach, the source code was also preprocessed, it was tokenized and a word embedding was performed. The CNN was tested using the same dataset, within-project and

across-project, but also using different independent features, source code with comments, without comments or partially with comments. A similar analysis method was followed, using cross validation and standard performance metrics. **EDIT The performance values were ... and ... They were better than TEDIOUS in the way that ... Donner un appercu de tous les resultats ... EDIT**

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. For the approach using source code features, 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. For the approach using the source code itself, **EDIT The performance values were ... and ... They were better than TEDIOUS in the way that ... Donner un appercu de tous les resultats ... EDIT**

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
WMC	Weighted Method Complexity
CBO	Coupling Between Objects
CNN	Convolutionnal Neural Network
QMOOD	Quality Model for Object-Oriented Design
OO	Object-Oriented

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 Definitions

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 conception timeline and under various forms, partly because writing quality code is not always compatible with the standard development life cycle (Brown et al., 2010). 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 phe-

nomenon 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). Technical debts cause a lot of problems: slower conception and execution of the software product (Allman, 2012), worse software maintainability and quality (Wehaibi et al., 2016; Zazworka et al., 2011), and higher production cost (Guo et al., 2011).

Frequently, TD are introduced consciously and explicitly by developers. In those cases, they are "self-admitted" and explained in comments, describing what is wrong with the related block of code (Potdar and Shihab, 2014). Like TD, self-admitted technical debts are encountered in most software projects. It was found that 31% of files contain SATD, that they remain present for a long period of time and that more experienced developers have the tendency to introduce them (Potdar and Shihab, 2014). This proves that a proper management tool is required to deal with this issue, and that unexperienced developers would greatly benefit from such support in order to decide when code should be reworked and documented as TD. The disparity between the experienced and unexperienced workers may also lie in the fact that the unexperienced ones don't want to admit their faults in order to maintain a positive image towards their superiors.

Another study found that there is no clear correlation between code quality and SATD (Bavota and Russo, 2016). Code quality metrics such as Weighted Method Complexity (WMC), Coupling Between Objects (CBO) and Buse and Weimer Readability (Buse and Weimer, 2010) were computed and analyzed to reach this conclusion. However, the purpose of their work was not mainly to evaluate this relationship but rather to establish a taxonomy of TD. Some threats to the validity of their results could also be made concerning the number of manually analyzed SATD and the level at which the metrics were computed (class-level). A finer analysis would have been required because a single class can contain methods of different length, complexity, cohesion, coupling and readability. This same study found in the analyzed projects on average 51 instances of SATD (0.3% of all comments), that the developer who introduces a TD is generally the same that fixes it and that they aren't all fixed.

1.2 Elements of the Problematic

It is pretty clear that technical debts account for a lot of issues in the development of software applications. They have been extensively analyzed and classified in order to have a better understanding of their impact. However, the identification, as much as the *correct* identification, of SATD remains a struggle for researchers and developers. Current methods

can obtain up to 25% of their total predictions as FP (Bavota and Russo, 2016). This means that a quarter of all automatically identified TD are not really technical debts and that the previous studies could be based on wrong information. It is true that many strategies can be employed to reduce and fix the number of TD in a software project: take your time when implementing a solution, code refactoring, continuous tracking of TD, proactiveness in fixing debts, etc. (Ambler, 2013). However, they are not highly effective and they frequently rely on the willingness of developers to fix the problem and their general knowledge.

Better automatic approaches have been proposed to improve the detection of TD. One of them is to identify comment patterns that relate to self-admitted technical debts (Potdar and Shihab, 2014). Potdar *et al.* manually went through 101 762 comments to determine these patterns, which lead to the identification of 62 SATD patterns. Here are some examples: *hack, fixme, is problematic, this isn't very solid, probably a bug.* The main issue with this approach is the the manual process behind it, which introduces human error and subjectivity. Another approach is to use machine learning techniques, such as NLP, to automatically identify SATD using source code comments (Maldonado et al., 2017). This idea is promising because it does not heavily depend on the manual classification of source code comments. In fact, it outperforms the previous approach. Manual classification is still required to build the training set for the NLP classifier. However, the model built from this dataset can then be used to automatically identify SATD in any project, thus removing any other manual analysis.

It is important to mention that our study does not revolve on proposing a new SATD detection method using information contained in comments, but rather using them as a base for our recommendation tool. Consequently, the proper classification of SATD used by TEDIOUS will directly affect its performance. To properly establish the problematic of our study, several research questions have to be addressed, the main one can be defined like this:

> How can we identify and detect technical debts in a software project using source code features and known self-admitted technical debts in a machine learning approach?

This question can be divided into three others: How does TEDIOUS work for recommending SATD within-project? How does TEDIOUS work for recommending SATD across-project? How would a method-level smell detector compare with TEDIOUS? This approach is based on the hypothesis that current methods to detect technical debts are limited and inefficient and that a new approach could be beneficial to the improvement of detection performance. We also think that manual analysis and human subjectivity is detrimental to the

efficiency of current methods. Consequently, we believe that a well crafted machine learning approach could lead to better results and performance values in identifying technical debts and recommending when they should be self-admitted.

1.3 Research Objectives

> The main objective of this research is to design a machine learning approach that uses as independent variables various kinds of source code features, and as dependent variables the knowledge of previously self-admitted technical debts, to train machine learners in recommending to developers when a technical debt should be admitted.

As mentioned previously, the purpose of this study is not to propose a novel method to identify SATD from source code comments, using patterns or NLP (Maldonado et al., 2017; Potdar and Shihab, 2014). It is more about using results of these methods to build our training dataset. Our approach relies more on source code information and metrics to identify possible TD to self-admit.

The main objective can be divided in two purposes. Firstly, tracking and managing technical debts is considered important but lacking in the industry (Ernst et al., 2015). Consequently, TEDIOUS could be used to encourage developers to self-admit TD in order to easily track and fix the issue. This is particularly true for junior developers, who are less prone to doing so than experienced ones (Potdar and Shihab, 2014). Secondly, our tool could be used as an alternative, or a complement, to existing smell detectors in proposing improvements to source code. In other words, TEDIOUS could act as a tracking, managing and improvement tool for software projects.

The general objective can be divided in specific objectives. The first one aims at *defining and extracting relevant features from methods*. These features are the characteristics that describe each method. In contrary to previous studies (Bavota and Russo, 2016), TEDIOUS works at method-level rather than class-level because we found that SATD comments are more frequently related to methods or blocks of code. Features can be: a set of structural metrics extracted from methods, the method's readability or warning raised by static analysis tools.

The second specific objective aims at *identifying self-admitted technical debts*. Only a certain type of technical debt is considered, namely design debts, for various reasons. Firstly, it is the most common type of TD (Maldonado et al., 2017). Secondly, the other types (requirement, code, test and documentation) would require a different analysis and features.

However, adding these types is part of our future work. No real detection method is used to detect those SATD, instead, design related TD from a previously annotated dataset consisting of 9 Java open source projects are used (Maldonado et al., 2017).

The third specific objective aims at *preprocessing the features*. Strongly correlated features are cleaned up to remove redundancy, metrics that don't vary or vary too much are also removed, a normalization is applied to take into account the different nature of projects and the training set is balanced by oversampling the small number of SATD tagged methods.

The fourth specific objective aims at *building and applying machine learning models*. Five machine learners are trained and tested, performing SATD prediction within-project and across-project. The five retained are: Decision Trees (J48), Bayesian classifiers, Random Forests, Random Trees and Bagging with Decision Trees.

We could add another objective which would aim at *improving the first TEDIOUS approach*. To do so, a Convolutionnal Neural Network is designed and implemented, using the source code itself as the independent variable. It is tested using the same dataset, within-project and across-project, and with different independent variable configurations, with, without or partially with comments. The results show improvement compared to the previous approach, **EDIT The performance values were ... and ... They were better than TEDIOUS in the way that ... Donner un appercu de tous les resultats ... EDIT**

1.4 Thesis Overview

Chapter 2: Literature Review The literature review provides a current state-of-the-art overview of the knowledge on technical debts and other related topics. It summarizes relevant information extracted from previous studies concerning four main topics: relationship between technical debt and source code metrics, self-admitted technical debt, code smell detection and automated static analysis tools.

Chapter 3: The Approach and Study Definition The approach followed is thoroughly described in four sections. The types of features are described and the way they are extracted is explained. The provenance and identification of the SATD tagged comments is shared. The preprocessing that is performed on features is demystified and justified. The machine learning models chosen are revealed as well as their configuration. As for the study definition, the dataset characteristics (number of files, classes, comments, etc.) are shared for each project and the analysis method (cross validation, accuracy, precision, recall, F_1 score, MCC, ROC,

AUC) explained.

Chapter 4: Analysis of Study Results and Threats to Validity The study results are analyzed based on each research question: performance for within-project prediction, performance for across-project prediction and comparison with a method-level smell detector. Results indicate that within-project prediction achieves at best 50% precision and 52% recall. Improvement is made for across-project prediction where prediction achieves at best 67% precision and 55% recall. The best machine learner turned out to be Random Forest. It was also found that SATD predictions made by TEDIOUS only weakly relate to method-level code smells. A qualitative discussion on false positives and negatives is also proposed. Following the results analysis, several threats to validity are shared: construct, internal, conclusion and external validity threats.

Chapter 5: Convolutionnal Neural Network with Comments and Source Code This chapter describes an updated approach to detect TD to self-admit and its preliminary results. First, the CNN characteristics and features are described. Secondly, the approach is explained: the features used, the identification of SATD, the use of word embeddings and the way the CNN is built and applied to the context of our study. Thirdly, the study definition is described: the characteristics of the dataset and the analysis method. Finally, the study results are analyzed based on three prediction contexts: source code with comments, without comments and partially with comments. Various CNN configurations are also analyzed. **EDIT The performance values were ... and ... They were better than TEDIOUS in the way that ... Donner un appercu de tous les resultats ... EDIT**

CHAPTER 2 LITERATURE REVIEW

The subject of this study can be divided in four relevant topics, which will be summarized in this chapter. The literature review introduces each topic based on previous studies and research papers. The first one addresses the relationship between technical debt and source code metrics. The second defines the nature of self-admitted technical debts. The last topic combines the code smell detection approaches and automated static analysis tools.

2.1 Relationship Between Technical Debt and Source Code Metrics

Many researchers have tried to relate technical debts, more specifically design and code, to source code metrics in order to detect them. One of the approach is based on a technique for detecting design flaws, built on top of a set of metric rules capturing coupling, cohesion and encapsulation (Marinescu, 2012). Another study empirically validated the relationship between TD and software quality models. Three TD detection methods were compared with Quality Model for Object-Oriented Design (QMOOD) (Bansiya and Davis, 2002) and only one of them had a strong correlation to quality attributes reusability and understandability (Griffith et al., 2014). Another team studied how five different tools detect technical debts, their principal features, differences and missing aspects (Fontana et al., 2016). They focused on the impact of design smells on code debt to give advices on which design debt should be prioritized for refactoring. These tools all take into account metrics, smells, coding rules and architecture violations but there is only a limited agreement among them and they still ignore some important pieces of information.

2.2 Self-Admitted Technical Debt

Many studies have been conducted in order to describe and classify the nature of self-admitted technical debts. Potdar and Shihab (2014) investigated technical debts in the source code of open source projects and they found out that developers frequently self-admit TD they introduce, explaining why this particular block of code is temporary and needs to be reworked in the form of comments. They are some of the first to acknowledge the existence of SATD and to propose a detection method using pattern matching in source code comments. da S. Maldonado and Shihab (2015) analyzed developers' comments in order to examine and quantify the different types of SATD. A similar approach to Potdar and Shihab (2014) is followed, using pattern matching, to classify the SATD into five types: design, defect,

documentation, requirement and test. It was found that design debts are the most common, making up between 42% and 84% of all comments.

Bavota and Russo (2016) performed a large-scale empirical study on self-admitted technical debt in open source projects. They studied its diffusion and evolution, the actors involved in managing SATD and the relationship between SATD and software quality. They showed that there is on average 51 instances of SATD per system, that code debts are the most frequent, followed by defect and requirement debts, that the number of instances increases over time because they are not fixed by developers, and that they normally survive for a long time. Like Griffith et al. (2014), they found no real correlation between SATD and quality metrics (WMC, CBO, Buse and Weimer readability).

Wehaibi et al. (2016) also studied the relation between self-admitted technical debt and software quality. Their approach is based on investigating if more defects are present in files with more SATD, if SATD changes are more likely to cause the emergence of future defects and whether they are more difficult to perform. They found that no real trend was noticed between SATD and defects, SATD changes did not introduce more future defects compared to none SATD changes but they are indeed more difficult to perform.

A new approach based on NLP techniques was used recently to detect self-admitted technical debts, more specifically design and requirement debts (Maldonado et al., 2017). They extracted comments from ten open source projects, cleaned them to remove the ones considered irrelevant and manually classified them into the different types of SATD. This dataset was then used as the training set for a maximum entropy classifier. It turned out that the model could accurately identify SATD and outperform the pattern matching method of Potdar and Shihab (2014). Comments mentioning sloppy or mediocre source code were the best indicators of design debts and comments related to partially implemented requirement were the best for requirement debts.

The detection of self-admitted technical debts is a major research approach in the study of SATD, however, this is not the purpose of our work. We do not propose a new approach using source code comments information, instead, we gather information about the structure of the code at method-level in order to recommend to developers when to self-admit technical debts.

2.3 Code Smell Detection and Automated Static Analysis Tools

Several approaches to detect code smells have been proposed in today's literature. Reading techniques have been created to guide developers in identifying Object-Oriented (OO) designs

(Travassos et al., 1999). Some formulate metrics-based rules as a detection strategy that can capture poor design practices (Marinescu, 2004) or use these software metrics to characterize bad smells (Munro, 2005). Moha et al. (2010) propose DECOR, an approach using rules and thresholds on various metrics.

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

E. Allman, "Managing technical debt", *Communications of the ACM*, vol. 55, no. 5, pp. 50–55, 2012.

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.

S. Ambler. (2013) 11 strategies for dealing with technical debt. En ligne: `http://www.disciplinedagiledelivery.com/technical-debt/`

J. Bansiya et C. G. Davis, "A hierarchical model for object-oriented design quality assessment", *IEEE Transactions on software engineering*, vol. 28, no. 1, pp. 4–17, 2002.

G. Bavota et B. Russo, "A large-scale empirical study on self-admitted technical debt", dans *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 315–326.

N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya *et al.*, "Managing technical debt in software-reliant systems", dans *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 2010, pp. 47–52.

R. P. Buse et W. R. Weimer, "Learning a metric for code readability", *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010.

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`

E. da S. Maldonado et E. Shihab, "Detecting and quantifying different types of self-admitted technical debt", dans *7th IEEE International Workshop on Managing Technical Debt, MTD@ICSME 2015, Bremen, Germany, October 2, 2015*, 2015, pp. 9–15.

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

F. A. Fontana, R. Roveda, et M. Zanoni, “Technical debt indexes provided by tools: a preliminary discussion”, dans *Managing Technical Debt (MTD), 2016 IEEE 8th International Workshop on*. IEEE, 2016, pp. 28–31.

I. Griffith, D. Reimanis, C. Izurieta, Z. Codabux, A. Deo, et B. Williams, “The correspondence between software quality models and technical debt estimation approaches”, dans *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*. IEEE, 2014, pp. 19–26.

Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Q. Da Silva, A. L. Santos, et C. Siebra, “Tracking technical debt—an exploratory case study”, dans *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 2011, pp. 528–531.

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.

E. Maldonado, E. Shihab, et N. Tsantalis, “Using natural language processing to automatically detect self-admitted technical debt”, *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017. DOI: 10.1109/TSE.2017.2654244

R. Marinescu, “Detection strategies: Metrics-based rules for detecting design flaws”, dans *Proceedings of the* 20^{th} *International Conference on Software Maintenance*. IEEE CS Press, 2004, pp. 350–359.

——, “Assessing technical debt by identifying design flaws in software systems”, *IBM Journal of Research and Development*, vol. 56, no. 5, pp. 9–1, 2012.

N. Moha, Y.-G. Gueheneuc, L. Duchien, et A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells", *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

M. J. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code", dans *Proceedings of the* 11^{th} *International Software Metrics Symposium*. IEEE Computer Society Press, September 2005.

A. Potdar et E. Shihab, "An exploratory study on self-admitted technical debt", dans *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 91–100.

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

G. Travassos, F. Shull, M. Fredericks, et V. R. Basili, "Detecting defects in object-oriented designs: using reading techniques to increase software quality", dans *Proceedings of the* 14^{th} *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 1999, pp. 47–56.

S. Wehaibi, E. Shihab, et L. Guerrouj, "Examining the impact of self-admitted technical debt on software quality", dans *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, pp. 179–188.

N. Zazworka, M. A. Shaw, F. Shull, et C. Seaman, "Investigating the impact of design debt on software quality", dans *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 17–23.

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.