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)
AOÛT 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:

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 déloppeur 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 données d'entrée est appliqué pour retirer les métriques superflus et garder ceux étant les plus pertinents

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 with-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, Matthews Correlation Coefficient (MCC) and Receiving Operating Characteristics (ROC) Area Under the Curve (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, DEtection & CORrection (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**, Long Method (LM) and Long Parameter List (LP) were the specific smells targeted and evaluated by DECOR, similar to Lines Of Code (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.

As for the qualitative analysis of correctly classified or misclassified SATD, several observations were made. When analyzing TP, TEDIOUS was able to correctly identify a wide variety of methods labeled as SATD even though their defining features were significantly different. FP targeted intrinsically complex and long methods that weren't initially labeled as SATD, which isn't necessarily bad because it may lead the developer to review these pieces of code. As an example of FN, some comments mention the presence of a problem in a block of code which isn't trivial when manually analyzed, or when using TEDIOUS.

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

TOTAL = 8 pages

0.5 page

Base sur abstract de l'article VOIR PROPOSITION DE RECHERCHE

1.1 Basic Concepts and Defintions

environ 3 pages

Technical debt Defintion TD (Cunningham 15) Definition classes TD (3 et 22) Nature intentionnelle des TD (24) Awareness des TD est un probleme (17)

Self admitted technical debt Definition SATD (Potdar 16 et 35) Presence des SATD (Potdar 16 et 35) Acteurs des SATD (Potdar 16 et 35) Correlation qualite/SATD (Bavota et Russo 8, 10) Taxonomie des SATD (Bavota et Russo 8)

Features Structural metrics of methods Method readability Warnings of static analysis tools (Checkstyle 1, PMD 2)

Dataset 9 java open source projects from (27), only design debts

Machine Learners 5 machine learners, with fold validation and cross project

Results 50% precision and 52% recall within project with RandomForest 67% precision, 55% recall, 92% accuracy cross project Can be applied on new projects and be good

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

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.