IDENTIFYING SELF-ADMITTED TECHNICAL DEBT

Everton da S. Maldonado

A thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Applied Science in Software

Engineering

Concordia University

Montréal, Québec, Canada

August 2016

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By:	**Everton da S. Maldonado**
Entitled:	**Identifying Self-Admitted Technical Debt**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science in Software Engineering

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

	______________________________	Chair
Dr. Juergen Rilling	______________________________	Examiner
Dr. Peter C. Rigby	______________________________	Examiner
Dr. Emad Shihab	______________________________	Supervisor

Approved ____

Chair of Department or Graduate Program Director

____________ 20 ______ ______________________________

Dean

Faculty of Engineering and Computer Science

Abstract

Identifying Self-Admitted Technical Debt

Everton da S. Maldonado

Technical debt is a metaphor coined to express the trade off between productivity and quality, e.g., when developers take shortcuts or perform quick hacks during the development of software projects. These non optimal solutions are often implemented to allow the project to move faster in the short term, at the cost of increased maintenance in the future. The accumulation of technical debt during the ever changing life-cycle of a project is unavoidable, and if not properly managed can severely hinder the development of the project. To help alleviate the impact of technical debt, a number of studies focused on the detection of technical debt. However, a recent study has shown that one possible source to detect technical debt is using source code comments, also referred to as self-admitted technical debt. Therefore, in this dissertation we use empirical studies and NLP techniques to propose an approach to automatically identify self-admitted technical debt.

First, we examine source code comments to determine the different types of technical debt, and we propose four simple filtering heuristics to eliminate comments that are not likely to contain technical debt. Then, we read through more than 33K comments, and we find that self-admitted technical debt can be classified into five main types - design debt, defect debt, documentation debt, requirement debt and test debt. In addition, two most common types of self-admitted technical debt are design and requirement debt, making up between 42% to 84% and 5% to 45% of the classified comments, respectively.

Second, we leverage the knowledge obtained in our first study to present an approach to automatically identify design and requirement self-admitted technical debt using Natural Language Processing (NLP). We study 10 open source projects: Ant, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, Jmeter, JRuby and SQuirrel SQL and find that 1) we are able to effectively identify self-admitted technical debt, significantly outperforming state-of-the-art techniques; 2) that words related to sloppy or mediocre source code are the best indicators of design debt, whereas for requirement debt, words related to enhancing or completing tasks are the best indicators; and 3) we can achieve 90% of the best classification performance, using as little as 23% of the comments for both design and requirement self-admitted technical debt, and 80% of the best performance, using as little as 9% and 5% of the comments for design and requirement self-admitted technical debt, respectively.

Acknowledgments

At the conclusion of a stage, you must look back and take the time to thank and to be grateful for those who have been by our side, because happiness is meaningless without someone to share. First, I am grateful to God who gives me strength, guidance and the opportunity to pursue my masters degree.

I would like to express my gratitude to my advisor Dr. Emad Shihab for his support and dedication during this last two years. Thank you Emad for believing in my potential, for teaching me how to overcome my limitations and that, through hard work, everything is achievable. I must tell you that it has been quite an enjoyable adventure, and that I learned a lot from you.

Apart from my advisor, I would also like to thank my thesis examiners, Dr. Juergen Rilling and Dr. Peter C. Rigby, for taking the time to read my thesis and for their valuable suggestions. Also, I would like to acknowledge Dr. Nikolaos Tsantalis for his help in my research, as well as all other faculty members of the Department of Computer Science and Software Engineering, for providing the necessary guidance.

In addition, a big thank you to all my lab mates and everyone else that I had the opportunity to work with. A special thanks to Moiz Arif, Davood Mazinanian, Ahmad Hassan, Samuel Donadelli, Rabe Abdalkareem and Shahriar Rostami. All of you made this experience so much more enjoyable.

More than anyone else, I would like to thank my wife Fabiana Maldonado. It was your love, continued support and encouragement that made this degree possible. You are more than I could wish for. Also, during this journey, I was very fortunate to count with so many dear friends. I would like to thank Eduardo Lomonaco, Kênia Balestra, Rodrigo Mayer, Luciana Schurt, Carlos and Antonia Lorenz for being always there for me.

Lastly, thank you mom, dad, brothers and grandparents, although far away you are always feeling my heart with your love. I dedicate this thesis for you.

Related Publications

The following publications are related to this thesis:

1. **Everton da S. Maldonado** and Emad Shihab. Detecting and Quantifying Different Types of Self-Admitted Technical Debt. In *Proceedings of the 7th IEEE International Workshop on Managing Technical Debt*, 7 pages, 2015. [Chapter 3]

2. **Everton da S. Maldonado**, Emad Shihab and Nikolaos Tsantalis. Using Natural Language Processing to Automatically Detected Self-Admitted Technical Debt. In *IEEE Transactions on Software Engineering*, 20 pages, Major revision submitted August 2016. [Chapter 4]

Contents

List of Figures

List of Tables

Chapter 1

Introduction

At the core of any software system is the software team who develop it. Their decisions and expertise is what makes the difference between solid well built systems and brittle implementations that my lead to countless hours of patch work. Ensuring the high quality of a software project is not easy, on the contrary it has been proven to be quite a challenge, almost utupic [KNO12]. Developers often have to deal with conflicting goals while developing a software system. Software needs to delivered quickly, without defects and on budget, all of these must happen in a rapidly changing environment.

In practice what happens is that all these conflicting constrains force a tradeoff to be made by software developers [Cun92]. Often this tradeoff means shortcuts and workarounds that results sub-optimal solutions [SG11, KNOF13]. These workarounds allow the project to move faster at first, helping software developers achieve short-term goals at the expense of increased maintenance effort in the future. This phenomena is called *technical debt* (TD). Acquiring careless technical debt can halt a software system to a full stop.

Unfortunately, prior research on technical debt has shown that technical debt is widespread, unavoidable, and therefore, needs to be properly managed to not have a negative impact on the quality of the software system [LTS12]. The first step to manage technical debt is the identification of technical debt, which is the main focus of this thesis.

A number of studies empirically examined technical debt and proposed techniques to enable its detection and management. Most approaches to identify technical debt are based on static source analysis tools, as described in more detail in Chapter 2. However, there are limitations to these approaches. First, static analysis tools depends on arbitrary metrics and thresholds to detect technical debt, and deriving appropriate threshold values is a challenging open problem that has attracted the attention and effort of several researchers [OVPL14, FFZY15, FMZM15]. In addition, the approaches based on source code analysis suffer from high false positive rates [FDW$^+$16]. Also, static analysis tools requires the construction of Abstract Syntax Trees or other more advanced source code representations. For instance, some code smell detectors that also provide refactoring recommendations to resolve the detected code smells [TC11, TMK15] generate computationally expensive program representation structures, such as program dependence graphs [Gra10], and method call graphs [AL12] in order to match structural code smell patterns and compute metrics.

More recently, another approach used source code comments to identify technical debt. Potdar and Shihab [PS14] devised an approach, which uses 62 comment patterns (i.e., words and phrases) used by software developers to indicate "not quite right code". The detection of technical debt through source code comments does not suffer of the same limitations that static analysis tools. For example, it is a more lightweight process that does not depend of source code representations, and does not depend on thresholds of any kind as the developers themselves are admitting the debt. Due to its nature, technical debt found in source code comments is referred to as *self-admitted technical debt*.

However, the identification of self-admitted technical debt poses many challenges.

Since comments are written in natural language, it is very difficult to automatically analyze them, and therefore, comments were analyzed manually by reading through each of them to identify those that indicated self-admitted technical debt. This poses a serious treat to the scalability of this approach. In addition, the comment patterns approach does not take into consideration the many different types of technical debt and it has no means to be evaluated in terms of precision and recall.

At the same time, automatic techniques that leverage machine learning have been proposed to help automatically classify natural language corpora, referred to as NLP techniques. In this thesis, we apply these NLP techniques to automatically identify technical debt from source code comments. The thesis provides two main contributions. The first part focuses on the definition and understanding of technical debt where we manually analyze source code comments to gain insights on the nature of self-admitted technical debt. The second part proposes an approach to identify technical debt using Natural Language Processing (NLP) techniques.

1.1 Research Hypothesis

Prior research and our industrial experience lead us to the formation of our research hypothesis. We believe that:

The identification of technical debt remains limited, and mostly dependent of static source code analysis tools that require expensive and heavy analysis processes, while yielding too many false positives. We hypothesize that source code comments can improve the identification of technical debt. Thus far, the approaches that uses source code comments heavily depend on manual processes, which often do not scale well. We believe that Natural Language Processing (NLP) techniques, when provided with the appropriated training dataset, can tackle the current challenges in the identification of technical debt.

1.2 Thesis Overview

Chapter 2: Literature Review: In this chapter we discuss the definition of the technical debt metaphor and its expansion over time as more developers kept adopting the term to communicate suboptimal solutions and "not quite right code". The literature review provides summarized and concise information that was extracted from websites, blogs and research papers ranging from the creation of the metaphor to the present date in a chronological order. The chapter concludes with our critical evaluation of the current limitations in the field and the challenges surrounding technical debt.

Chapter 3: Analyzing Source Code Comments and Different Types of Self-Admitted Technical Debt: In this chapter we examine source code comments from 5 open source projects to determine the different types of technical debt. First, we propose four simple filtering heuristics to eliminate comments that are not likely to contain technical debt. Filtering out irrelevant comments is very helpful as it allows us focus our attention to the more insightful comments. Second, we manually classify the remaining comments (i.e., more than 33K comments), and we find that self-admitted technical debt can be classified into five main types - design debt, defect debt, documentation debt, requirement debt and test debt. The two most common types of self-admitted technical debt are design and requirement debt, making up between 42% to 84% and 5% to 45% of the classified comments, respectively.

Chapter 4: Proposing an Approach to Automatically Identify Self-Admitted Technical Debt: In this chapter, we present an approach to automatically identify design and requirement self-admitted technical debt using Natural Language Processing (NLP). We study 10 open source projects. We show that our approach can accurately identify self-admitted technical debt, we also discuss the features (i.e., words) that are the best indicators of design and requirement debt. Lastly, as training data is the most crucial point to apply and expand our approach, we conduct a detailed analysis of the quantity of training data to obtain satisfactory classification performances.

1.3 Thesis Contributions

The major contributions of this thesis are as follows:

- A concise review of the state-of-the-art in the technical debt field. Such a review provide the necessary background to enable interested researchers to focus on the main challenges in the field of technical debt.

- We contribute a rich dataset of self-admitted technical making the data used in this thesis publicly available. To the best of our knowledge, there is no similar data available and we believe that the dataset will help future research in the area of self-admitted technical debt providing the necessary means to evaluate and apply different approaches.

- We propose an automatic, NLP-based, approach that outperforms the current state-of-the-art in the identifying design and requirement self-admitted technical debt. Moreover, we investigate the amount of training data necessary to effectively identify technical debt through an empirical experiment, giving support to future enhancement and expansion of our approach, such as the detection of self-admitted technical debt comments in different programming languages or idioms.

Chapter 2

Literature Review

In this chapter we present related work on technical debt. These studies set the current background of technical debt. More specifically, the studies presented in this chapter are classified into two categories. First, we present studies that discuss the definition and the extensibility of the technical debt metaphor. This first part lays the foundation of what it is technical debt and how it is being used nowadays. Second, we present studies that investigate the identification and the implications of technical debt in the source code, including studies that are more related to our own work which is the identification of *self-admitted* technical debt.

Although we provide a broad review of related work of technical debt in this chapter, related work that is closely related to each of our contributions can be found in the chapters 3 and 4.

2.1 Defining and Expanding the Technical Debt Metaphor

At first, most information about technical debt were available on blogs. These blogs were written by industry specialists and evangelists of Agile Methodologies, such as

Martin Fowler [Mar]. Nowadays, academia and industry alike study the applications of the technical debt metaphor. Thus, a large number of studies were dedicate to this matter, and therefore, the original metaphor has been expanded and refined.

The metaphor, *technical debt*, was introduced by Ward Cunningham [Cun92] more than two decades ago to facilitate the communication between developers and non-technical personnel working on the same software project. Cunningham explains how "not quite right code" will affect the maintainability of a project (i.e., require more effort to maintain the project in the future) as interest does on incurred debt.

In other words, every time that an implementation around the code affected by the non-optimal implementation is needed, an interest in the form of effort will be expend in the task. Although debt may speed up the project development at first, accumulated debt will bring the project to a standstill in the long-run. Thus, the technical debt metaphor, provides insight to managers of why it is beneficial to use resources to enhance a particular portion of the code even if it is not broken.

The term has been refined and expanded since, notably by Steve McConnell [McC] in his taxonomy and by Martin Fowler [Mar] with his four quadrants. As these works were very important to the development of a deeper understanding of technical debt and its applicability on software engineering we dedicated subsections 2.1.1 and 2.1.2 for further explanation on the authors' definition of technical debt.

2.1.1 Unintentionally Incurred Debt vs. Intentionally Incurred debt

According to Steve McConnell, technical debt can be divided into two main types: *unintentionally incurred debt* and *intentionally incurred debt*.

Examples of unintentionally incurred debt range from a design approach that just turns out to be error-prone to a junior programmer who writes bad code. This technical debt is the non-strategic result of doing a poor job. In some cases, this type of debt can be incurred unknowingly, for example, when a company acquires another company that has accumulated technical debt over the years.

The second type of technical debt, incurred intentionally, commonly occurs when an organization makes a conscious decision to optimize for the present rather than for the future. An “If we do not get this release done on time, there will not be a next release” type of situation. This leads to decisions like, “We do not have time to reconcile these two databases, so we will write some glue code that keeps them synchronized for now and reconcile them after we ship.” Or “We have some code written by a contractor that does not follow our coding standards; we will clean that up later.” Or “We did not have time to write all the unit tests for the code we wrote the last 2 months of the project. We’ll right those tests after the release” [McC].

Moreover, technical debt incurred intentionally can be of two types: short-term and long term debt. Like with real debt, short-term debt is expected to be paid off frequently. Short-term debt is taken on tactically and reactively, usually as a late-stage measure to get a specific release out the door, whereas long term debt is taken on strategically and pro-actively. For example, “We do not think we are going to need to support a second platform for at least five years, so this release can be built on the assumption that we are supporting only one platform”.

The implication is that short-term debt should be paid off quickly, perhaps as the first part of the next release cycle, whereas long-term debt can be carried for a few years or longer.

Therefore, McConnell presents the following **taxonomy for technical debt** to summarize his thoughts on technical debt:

1. - Debt incurred **unintentionally** due to low quality work

2. - Debt incurred **intentionally**

   (a) - **Short-term debt**, usually incurred reactively, for tactical reasons

        i. - Focused Short-Term Debt. Individually identifiable shortcuts (like a car loan)

        ii. - Unfocused Short-Term Debt. Numerous tiny shortcuts (like credit card debt)

    (b) - **Long-term debt**, usually incurred pro actively, for strategic reasons

2.1.2 Technical Debt Quadrant

On the other hand, Fowler's definition of technical debt is slightly different, and it is represented by four quadrants namely reckless, prudent, deliberate and inadvertent. According to Fowler, debt can be any combination of these four quadrants.

For example, prudent deliberate debt is the one that the team knows they are taking on, and thus puts some thought as to whether the payoff for an earlier release is greater than the costs of paying it off. However, a team not aware of design practices is taking on its reckless debt without even realizing how much workarounds it is getting into (inadvertent). Although reckless debt may not be inadvertent. A team may know about good design practices, but decide to go "quick and dirty" because they think they can not afford the time required to write clean code. The fourth cell of the quadrant is prudent/inadvertent debt. This case represent the case where a skilled development team is creating a project applying the best design to handle the current requirements, however over time, the chosen design proves to be inadequate to the future need of the project. Fowler points out that the point is that while you are programming, you are also learning. It is often the case that it can take a year of programming on a project before you understand what the best design approach should have been.

Figure 1 presents the actual technical debt quadrant, and illustrates on each cell the possible cases that can happen with a development team while working on a software project.

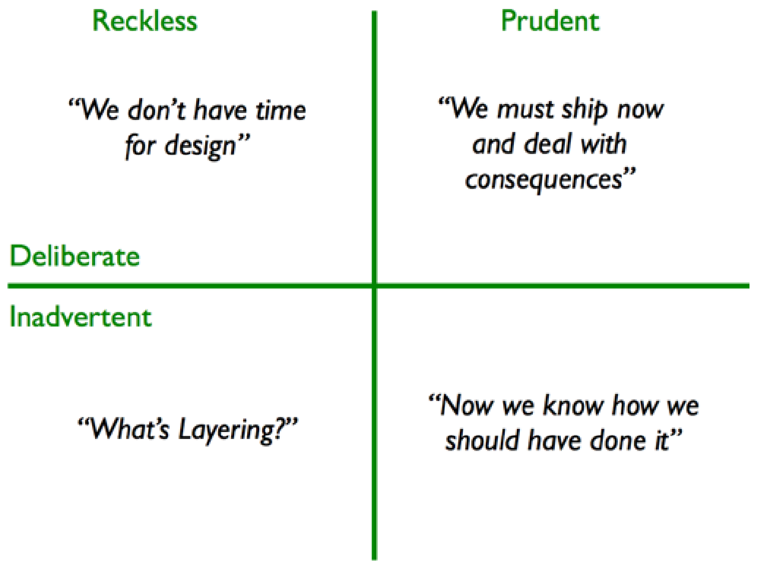

Reckless
Prudent
"We don't have time for design"
"We must ship now and deal with consequences"
Deliberate
Inadvertent
"What's Layering?"
"Now we know how we should have done it"

Figure 1: Technical Debt Quadrant[Mar]

In addition, from the original description "*not quite right code which we postpone making it right*" various people have used the metaphor of technical debt to describe many other *kinds* of debts or faults of software development, including anything that is related to deploying, evolving a software system or anything that is intrinsic to software development such as test debt, people debt, architectural debt, requirement debt, documentation debt, or just an broad generalized software debt [Ste10].

In this matter, Kruchten *et al.* [KNO12] express their concern about how the use (or abuse) of the metaphor could spread it too thin making the metaphor lose its communication power. For example, a not yet implemented requirement, function, or feature does not translate to requirement debt. Similarly, postponing the development of a new function is not a planning debt. Another danger pointed out by the authors relates to the assistance of static code analysis tools on the identification of technical debt. Although these tools are very useful there is a danger of equating whatever the tools can detect with what is technical debt. This approach leads to leaving aside large amounts of potential technical debt that is undetectable by tools, such as structural or architectural debt, technological gaps or self-admitted technical debt as discussed later on this chapter.

Later, Spinola *et al.* [SZV$^+$13] identified and organized a number of statements about technical debt expressed by practitioners in online websites, blogs and published papers. The authors chose 14 statements related to technical debt and conducted two surveys with 37 participants to evaluate the level of agreement on each statement. They found that practitioners strongly agree that if technical debt is not managed effectively, maintenance costs will increase at a rate that will eventually outrun the value it delivers to customers. In addition, they found that practitioners strongly disagree that all technical debt is intentional, the results found by the authors support the expanded technical debt definition proposed by McConnel and Fowler.

Moreover, the authors state that the acceptance and use of the technical debt metaphor is in large part because it is easily understood. However, this can also be a concern to accurately define technical debt. Their reasoning is that because the technical debt metaphor is easy to understand, it is also easy to talk about, expand on, and relate experience to. A quick search of technical debt literature reveals subjective opinions, personal views, and catch phrases on such channels as blogs and online essays. Therefore, more analysis on the use of the metaphor is necessary to organize the technical debt landscape.

Alves *et al.* [ARC$^+$14, AMdM$^+$16] proposes an ontology of terms on technical debt in order to organize a common vocabulary for the area. In their work they extracted and organized concepts derived from the results of a systematic literature mapping. In total, 100 studies, dated from 2010 to 2014, were evaluated. Their work contributed towards the evolution of the technical debt landscape through the organization of the different types of technical debt and their indicators. The authors found the following types of debt in the literature: design debt, architecture debt, documentation debt, test debt, code debt, defect debt, requirements debt, infrastructure debt, people debt, test automation debt, process debt, build debt, service debt, usability debt and versioning debt. Moreover, they state that some instances of technical debt can fit more than one type of technical debt.

These works summarize the definition and expansion of the technical debt metaphor.

2.2 Identification and Implications of Technical Debt

2.2.1 Using Source Code and Static Analysis Tools

A number of studies have focused on the detection and management of technical debt. Much of this work has been driven by the Managing Technical Debt Workshop community. Static analysis tools can help to detect source code anomalies and object oriented violations using metrics and thresholds to evaluate code quality. These violations are commonly referred as bad smells and they follow under the design technical debt type of debt.

Zazworka *et al.* [ZSSS11] conducted a case study to investigate how design debt, in the form of god classes, affects the maintainability of software projects. The authors analyzed two commercial applications of a small-size software development company. They found that god classes suffer more changes and contain more defects than non-god classes showing that technical debt has a negative impact on software quality, and should therefore be identified and managed closely in the development process.

Fontana *et al.* [FFS12] also analyzed design debt in the form of bas smells. The authors focus their attention on three specific code smells (i.e., god class, data class and duplicate code) extracted from open source systems of different domains. They proposed an approach to suggest which bad smell should be addressed first based on the negative impact they have on the quality of the project.

In a follow up study, Zazworka *et al.* [ZSV$^+$13] conducted an experiment where a development team was asked to identify technical debt items in artifacts from a software project that they were familiar with. Then, the authors collected the output of three static analysis tools to automatically identify technical debt and compared it to the results of human elicitation. The authors found that there is little overlap between the technical debt reported by different developers, so aggregation, rather than consensus, is an appropriate way to combine technical debt reported by multiple developers. Moreover, they confirmed that static analysis tools can not detect many different types of technical debt, and therefore, involving humans in the identification process is necessary.

2.2.2 Using Source Code Comments (Self-Admitted Technical Debt)

A lot of effort has been made to identify and manage technical debt. Despite the help provided by static source code analysis tools the identification of technical debt is still an open challenge. More recently, Potdar and Shihab [PS14] found that source code comments can be analyzed to identify technical debt. Differently from most source code analysis tools, that rely on suggested metrics and thresholds to detect a *supposed* debt, technical debt found in the source comment is written by the developer of the program as a *confession*. These developers are explicitly saying that a particular implementation is not ideal, in other words this implementation is *self-admitted technical debt*.

Potdar and Shihab [PS14] conducted the first study to explore source code comments to identify technical debt. They extracted the source comments from 5 open source projects, and manually inspected them. In total, the authors read and analyzed more than 100K comments, and they come up with 62 different comment patterns that indicates the presence of self-admitted technical debt. These comment patterns are words or small phrases such as “retarded”, “stupid” and “remove this ugly code". Besides of the 62 comment patterns they found that self-admitted technical debt comments are present in 2.4% to 31% of the analyzed files, that developers with higher experience tend to introduce most of the self-admitted technical debt and that time pressures and complexity of the code do not correlate with the amount of self-admitted technical debt. This work is the current state-of-the-art in the identification of self-admitted technical debt.

Bavota and Russo [BR16] presented differentiated replication of the work by Potdar and Shihab. The authors analyzed 159 software projects to investigate the diffusion and evolution of self-admitted technical debt and its relationship with software quality. During this study the authors extracted over 600K commits and 2 Billion comments. Their main findings showed that self-admitted technical debt is diffused, with an average of 51 instances per system, increases over time due to the introduction of new instances that are not fixed by developers, and even when fixed, it survives a long time (over 1,000 commits on average) in the system.

Also, Farias *et al.* [FNSS15] developed a Contextualized Vocabulary Model for identifying technical debt on code comments (CVM-TD). CVM-TD uses word classes and code tags to support technical debt identification based on the comment patterns devised in [PS14]. The model created by the authors provided a structure that systematically allows combining terms creating a large vocabulary on technical debt. However, the author point out that CVM-TD needs to be calibrated in order to improve its accuracy.

Wehaibi *et al.* [WSG16] also took advantage of the comment patterns to examine the relation between self-admitted technical debt and software quality. The authors analyzed if files with self-admitted technical debt have more defects compared to files without self-admitted technical debt, if self-admitted technical debt changes are more likely to introduce future defects, and if self-admitted technical debt changes tend to be more difficult. They analyzed 5 open source projects to find that there is no clear relationship between defects and self-admitted technical debt, however, self-admitted technical debt changes are more difficult to perform (i.e., they are more complex).

The work mentioned so far relied on the identification of technical debt through source code comments, specifically using the comment patterns approach. One aspect that is not explored however is how good is the comment patterns approach in identifying technical debt. Another topic that is not explored is that the comment patterns approach does not provide support for the identification of different types of technical debt. The first step to address the afore-mentioned limitations is to create a golden dataset of self-admitted technical debt comments where different approaches can be compared in terms of precision and recall. Also, such dataset, if available could provide insights on the different types of self-admitted technical debt.

Believing that this is a very important step towards the advance of the state-of-the-art in the automatic identification of self-admitted technical debt we focused our efforts to create an golden dataset of manually classified self-admitted technical debt. In the next chapter we explain in detail our approach to create such a dataset of self-admitted technical debt comments. We also analyze the insights achieved during the creation of this dataset, such as what are the different types of self-admitted technical debt comments, how their distribution look likes and the definition of each type of self-admitted technical debt.

Chapter 3

Analyzing Source Code Comments and Different Types of Self-Admitted Technical Debt

3.1 Introduction

The software development process is filled with challenges. There are short deadlines, complex changes that need to be made, high quality expectations and an ever changing environment. Often there is much more that needs to be done than time to accomplish it. These conditions puts developers under increasing pressure to implement their tasks, while achieving many conflicting constraints. In this context, some decisions are made to allow the short term development of the project at the cost of its increased maintenance effort in the future. This phenomena is know as Technical Debt [Cun92].

With the organization of the technical debt community through the managing technical debt workshop [FKNO14], recent work has focused on the detection of technical debt [PS14, ZSV+13], studying the impact of technical debt [ZSSS11] and the appearance of technical debt in the form of code smells [FFS12]. Despite many efforts to detect technical debt, its detection remains a challenge [PS14]. One relatively unexplored aspect of technical debt is self-admitted technical debt, that is technical debt reported in source code comments. Self-admitted technical debt refers to the situation where developers know that the current implementation is not optimal and write comments alerting the inadequacy of the solution.

Recently, Potdar and Shihab [PS14] developed an approach to identify technical debt from code comments, and through manual inspection, were able to mine 62 patterns that effectively identify self-admitted technical debt. However, their approach does not take into consideration the different types of technical debt. Understanding the different types of self-admitted technical debt is important since: 1) it helps the community understand the limitations of understanding technical debt through code comments, 2) it allows us to complement existing technical debt detection approaches and 3) it provides us with a better understanding of the developer's point of view of technical debt.

Therefore, in this chapter we examine and quantify the different types of self-admitted technical debt. To do so, we extract source code comments from 5 well commented open source projects that belongs to different application domains, namely Apache Ant, Apache Jmeter, ArgoUml, Columba and JFreeChart. In total, we examined more than 166K comments. We applied a set of 4 simple filtering heuristics to remove comments that are not likely to contain self-admitted technical debt (e.g., license comments, commented source code, Javadoc comments). Finally, these filtering heuristics resulted in a dataset of 33,093 comments that the first author manually analyzed and classified into different types of self-admitted technical debt.

When classifying the code comments, we found 5 types of self-admitted technical debt which are: design debt, defect debt, documentation debt, requirement debt and test debt. Analyzing the distribution of the comments we found that the most common type of self-admitted technical debt is design debt, making up between 42% - 84% of all the classified comments. In addition to our findings, we contribute a rich dataset of self-admitted technical making the data used in this study publicly available. To the best of our knowledge, there is not similar data available and we believe that the dataset will encourage future research in the area of self-admitted technical providing the necessary foundation for more advanced techniques as Natural Language Processing.

The rest of the chapter is organized as follows. Section 3.2 presents related work. We describe our approach and setup our case study in Section 3.3. Section 3.4 presents the case study results. The threats to validity are presented in Section 3.5 and in Section 3.6 concludes the chapter and discusses future work.

3.2 Related Work

Our work uses code comments to classify self-admitted technical debt. Therefore, we divide the related work into two categories: source code comments and technical debt.

3.2.1 Source Code Comments

A number of studies examined the co-evolution of source code comments and the rationale for changing code comments. For example, Fluri *et al.* [FWG07] analyzed the co-evolution of source code and code comments, and found that 97% of the comment changes are consistent. Tan *et al.* [TMTL12] proposed a novel approach to identify inconsistencies between Javadoc comments and method signatures. Malik *et al.* [MCHM$^+$08] studied the likelihood of a comment to be updated and found that call dependencies, control statements, the age of the function containing the comment, and the number of co-changed dependent functions are the most important factors to predict comment updates.

Other work used code comments to understand developer tasks. For example, Storey *et al.* [SRB$^+$08] analyzed how task annotations (e.g., TODO, FIXME) play a role in improving team articulation and communication. The work closest to ours is the work by Potdar and Shihab [PS14], where code comments were used to identify technical debt.

Our work complements the prior work using code comments. Similar to the prior work, we also leverage source code comments, however, we use the comments to identify self-admitted technical debt. In particular, we focus on the detection and quantification of the *different types* of self-admitted technical debt.

3.2.2 Technical Debt

A number of studies have focused on the study of, detection and management of technical debt. Much of this work has been driven by the Managing Technical Debt Workshop community. For example, Seaman *et al.* [SG11], Kruchten *et al.* [KNOF13], Brown *et al.* [BCG$^+$10] and Spinola *et al.* [SZV$^+$13] make several reflections about the term technical debt and how it has been used to communicate the issues that developers find in the code in a way that managers can understand. Alves *et al.* [ARC$^+$14] proposes an ontology on technical debt terms. In their work they gathered definitions and indicators of technical debt that were scattered across the literature. Their resulting ontology provides several different types of technical debt (e.g., architecture debt, build debt, code debt, design debt, defect debt, etc) grouped by their nature (i.e., the factor that lead to the introduction of the debt at the first place).

Other work focused on the detection of technical debt. Zazworka *et al.* [ZSV$^+$13] conducted an experiment to compare the efficiency of automated tools in comparison with human elicitation regarding the detection of technical debt. They found that there is small overlap between the two approaches, and thus it is better to combine them than replace one with the other. In addition, they concluded that automated tools are more efficient in finding defect debt, whereas developers can realize more abstract categories of technical debt.

In follow on work, Zazworka *et al.* [ZSSS11] conducted a study to measure the impact of technical debt on software quality. They focused on a particular kind of design debt, namely God Classes. They found that God Classes are more likely to change, and therefore, have a higher impact in software quality. Fontana *et al.* [FFS12] investigated design technical debt appearing in the form of code smells. They used metrics to find three different code smells, namely God Classes, Data Classes and Duplicated Code. They proposed an approach to classify which one of the different code smells should be addressed first, based on a risk scale. Moreover, Potdar and Shihab [PS14] used code comments to detect self-admitted technical debt.They extracted the comments of four projects and analyzed more than 101,762 comments to come up with 62 patterns that indicates self-admitted technical debt. Their findings show that 2.4% - 31% of the files in a project contain self-admitted technical debt.

Our work is different from the aforementioned work that uses code smells to detect design technical debt since we use code comments to detect technical debt. Also, our focus is on *self-admitted* technical debt. Our work advances the prior work on self-admitted technical debt by detecting and quantifying the different types of self-admitted technical debt and classifying them accordingly. We also contribute a rich data set of code comments that are classified into the different types of self-admitted technical debt.

3.3 Approach

The main goal of our study is to identify and quantify the different types of self-admitted technical debt found in source code comments. Figure 2 shows an overview

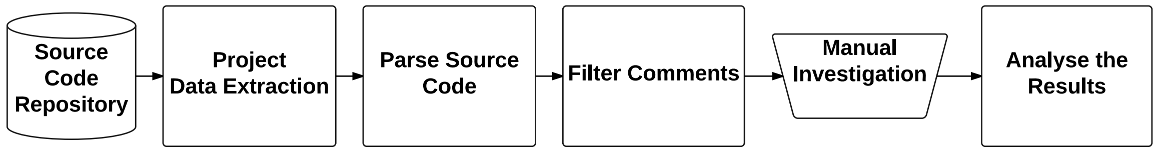


Figure 2: Dataset Creation Approach Overview

Table 1: Details of the Projects Used to Create the Dataset

Project	**Release**	**# of classes**	**SLOC**	**# of comments**	**# of contributors**
Apache Ant	1.7.0	1,475	115,881	21,587	74
Apache Jmeter	2.10	1,181	81,307	20,084	33
ArgoUML	0.34	2,609	176,839	67,716	87
Columba	1.4	1,711	100,200	33,895	9
JFreeChart	1.0.19	1,065	132,296	23,474	19

of our approach, and the following subsections detail each step of it.

3.3.1 Project Data Extraction

To perform our study, we obtain the source code of five open source projects, namely Apache Ant, Apache Jmeter, ArgoUML, Columba and JFreeChart. We chose the aforementioned projects, since they belong to different application domains, and vary in size (e.g., SLOC), and in the number of contributors.

Table 1 provides statistics about each one of the projects used in our study. We provide details about the release used, the number of classes, the total source lines of code (SLOC), the total extracted comments and the number of contributors. A source line of code contain at least one valid character, which is not blank spaces or source code comments. In our study, we only use the Java files to calculate the SLOC, and to do so, we use the tool SLOCCount [Whe04].

The number of contributors was extracted from OpenHub, an on-line community and public directory that offers analytics, search services and tools for open source software [Ope]. It is important to notice that the number of comments shown for each project does not represent the number of commented lines, but rather the number of individual line, block, and Javadoc comments. In total, we obtained more than 166,756 comments, found in 8,041 Java classes.

3.3.2 Parse Source Code

After obtaining the source code of all projects, we extract the comments from their source code. We use JDeodorant [TCC08], an open-source Eclipse plug-in, to parse the source code and extract the code comments. JDeodorant is capable of identify design flaws (i.e., bad smells) in Java projects, and suggest refactoring opportunities to solve them. JDeodrant uses the Eclipse AST framework to create an Abstract Syntax Tree (AST) map of the source code. The AST map contains detailed information about the project such as: the source code comments, its type (i.e., Block, Single-line or Javadoc), the line where each one of these comments begins and finishes. We extract the aforementioned information and store all comments in a relational database to facilitate the processing of the data.

3.3.3 Filter Comments

Source code comments can be used for different purposes in a project like giving context, as part of the documentation, to express thoughts, opinions and authorship, and in some cases, to remove source code from the program. Comments are used freely for developers and with few formalities, if any at all. This informal environment allows developers to bring to light opinions, insights and even confessions (e.g., self-admitted technical debt).

As shown in prior work by Potdar and Shihab [PS14], part of these comments can be identified as self-admitted technical debt, but they are not the majority of cases. With that in mind, we develop and apply 4 filtering heuristics to narrow down the comments eliminating the ones that are less likely to be classified as self-admitted technical debt.

To do so, we developed a Java based tool that reads from the database the data obtained by parsing the source code. Next, it executes the filtering heuristics and stores the result back in the database. The retrieved data contains information like the line number that a class/comment begins/ends and the type, considering the Java syntax, of the comment (i.e., Block, Single-line or Javadoc). With this information we process the filtering heuristics as described next.

We found that license comments are very not likely to contain self-admitted technical debt, and that license comments are commonly added before the declaration of the class. Therefore, we create a heuristic that removes comments that are placed before the class declaration. Since we know the line number that the class was declared we can easily check for comments that are placed before that line and remove them. In order to decrease the chances of removing a self-admitted technical debt comment while executing this filter we calibrated this heuristic to not remove comments containing one of task-reserved words (i.e., "todo", "fixme", or "xxx").

We also notice that some times developers make long comments, using multiple *single-line* comments instead of a Block comment. This characteristic can hinder the understanding of the message. Consider the case that the reader (i.e., human or machine) analyze each one of these comments independently, the message would be incomplete and the meaning lost. To solve that problem, we create a heuristic that searches for consecutive single-line comments and groups them as one. We identify consecutive comments by subtracting the line number of both comments. If the result of the difference is equals a -1 we have a consecutive comment. For example, Single-line comment A is placed in line number 100 and Single-line comment B is placed in line 101. The subtraction of the line numbers will result in -1, therefore the comments are consecutive.

Similarly, is common to find commented source code across the projects, and this can be due to many different reasons. One of the possibilities is that the code is not being used, other is that the code is used for debug purposes only. Based on our analysis, commented source code does not have self-admitted technical debt. Our heuristic remove commented source code using a simple regular expression that captures typical Java code structures.

Lastly, when analyzing Javadoc comments we found that they rarely mention self-admitted technical debt. For the Javadoc comments that does mention self-admitted technical debt we notice that they usually contains one of the task-reserved words (i.e., "todo", "fixme", or "xxx"). Based on this, our heuristic remove all comments of the type Javadoc unless they contain at least one of the task-reserved words. To do so, we create a simple regular expression that search for the task-reserved words before removing the comment.

The steps mentioned above significantly reduced the number of comments in our dataset and helped us focus on the most applicable and insightful comments. For example, in the Apache Ant project, applying the above steps helped reduce the number of comments from 21,587 to 4,140 comments meaning that 19.17% of the comments were kept for analysis. Table 2 provides details for each one of the projects.

Table 2: Filtering Heuristics Details

Project	**Total # of comments**	**# of comments after filtering**	**% of TD-related comments**
Apache Ant	21,587	4,140	19.17 %
Apache Jmeter	20,084	8,163	40.64 %
ArgoUML	67,716	9,788	14.45 %
Columba	33,895	6,569	19.38 %
JFreeChart	23,474	4,436	18.89 %

3.3.4 Manual Classification

To classify the comments, we developed a Java based tool that shows one comment at a time and gives a list of possible classifications that can be manually assigned to the comment. The list of possible classifications is based on previous work by Alves *et al.* [ARC$^+$14]. After applying the different filtering steps, we successfully classified 33,093 comments. The more than 33 thousand comments were classified into five different types of self-admitted technical debt, i.e., design debt, defect debt, documentation debt, requirement debt and test debt.

The first author who made the classification has more than 8 years of experience working in the industry as a software engineer, during this time he designed, implemented and maintained several programs using, in particular the Java programming language. He developed solid skills in object orientated programming and design patterns. We consider that these qualifications provide the necessary background to conduct the manual classification of the comments.

3.4 Case Study Results

The goal of our study is to classify and quantify the different types of self-admitted technical debt. To do so, we divide our study in two parts first, we manually read trough all comments identifying self-admitted technical debt among them. Once identified, the self-admitted technical debt, is classified into different types. Second, we quantify these comments identifying the most common types. Our case study is formalized with the following research question:

RQ: What are the types of self-admitted technical debt? How frequent are the different types of self-admitted technical debt in the studied projects ?

Motivation: As shown in previous work [PS14], self-admitted technical can be an indicator of non-optimal solutions. However, technical debt is a general term, and there are many different types of technical debt [ARC$^+$14]. Although we know that self-admitted technical exists, the different types of self-admitted technical debt are still unknown. For example, are we able to detect documentation debt from code comments? Answering this question is important as different types of debt have different approaches to be solved, and therefore each different type may need a tailored solution. It also helps us understand the opportunities and limitations of using code comments to detect technical debt.

Approach: To identify the different types of debt found in the comments we manually read through source code comments as described in Section 3.3. While examining the comments we classify each comment by the nature of the debt, using the descriptions provided by Alves *et al.* as a guideline.

During the classification we notice that some comments can be classified in more than one type of debt (e.g., a comment reporting a design debt can also be causing an unexpected behavior, which is defect debt). Although this is an ambiguous situation, and may have different interpretations depending of who is reading the comments, we defined that each comment would have just one classification type for the sake of clarity.

To mitigate the chance of misclassifying these comments, we take in consideration the more meaningful type for each comment in a given scenario. To do so, whenever a case like this occurred, we did a more detailed investigation (i.e., by examining the source code and any available documentation). In total we read and classified 33,093 comments from five open source projects. The classification took approximately 95 hours and was performed by the first author of the paper.

Results: We found five different types of self-admitted technical debt. Below, we list the different types of technical debt that we were able to detect and provide example comments to help the reader grasp the different types of self-admitted technical debt comments.

- **Self-admitted design debt:** These comments indicate that there is a problem with the design of the code. They can be comments about misplaced code, lack of abstraction, long methods, poor implementation, workarounds or a temporary solution. Lets consider the following comments:

  > *"TODO: - This method is too complex, lets break it up"* - [from ArgoUml]
  >
  > *"/* TODO: really should be a separate class */"* - [from ArgoUml]

  These comments are clear examples of what we consider as self-admitted *design debt*. In the above comments, the developers state what needs to be done in order to improve the current design of the code. Although the above comments are easy to understand, during our study we came across more challenging comments that expressed design problems in an indirect way. For example:

  > *"// I hate this so much even before I start writing it. // Re-initialising a global in a place where no-one will see it just // feels wrong. Oh well, here goes."* - [from ArgoUml]
  >
  > *"//quick & dirty, to make nested mapped p-sets work:"* - [from Apache Ant]

  In the above example comments the authors are certain to be implementing code that does not represent the best solution. Intuitively, we know that kind of implementation will degrade the design of the code and should be avoided.

  > *"// probably not the best choice, but it solves the problem of // relative paths in CLASSPATH"* - [from Apache Ant]

"//I can't get my head around this; is encoding treatment needed here?" - [from Apache Ant]

The above comments expressed doubt and uncertainty when implementing the code and were considered as self-admitted design debt as well.

- **Self-admitted defect debt:** In defect debt comments the author states that a part of the code does not have the expected behavior, meaning that there is a defect in the code.

  "// Bug in above method" - [from Apache Jmeter]

  "// WARNING: the OutputStream version of this doesn't work!" - [from ArgoUml]

  As shown in these examples there are defects that are known by the developers, but for some reason is not fixed yet.

- **Self-admitted documentation debt:** In the documentation debt comments the author express that there is no proper documentation supporting that part of the program.

  *"**FIXME** This function needs documentation"* - [from Columba]

  "// TODO Document the reason for this" - [from Apache Jmeter]

  Here, the developers clearly recognize the need to document their code, however, for some reason they do not document it yet.

- **Self-admitted requirement debt:** Requirement debt comments express incompleteness of the method, class or program as observed in the following comments:

"/TODO no methods yet for getClassname" - [from Apache Ant]

"//TODO no method for newInstance using a reverse-classloader" - [from Apache Ant]

*"TODO: The copy function is not yet * completely implemented - so we will * have some exceptions here and there.*/"* - [from ArgoUml]

The last example shows a comment that could be considered as having more than one type of debt. (i.e., requirement debt and defect debt), but as mentioned in the classification approach, we choose to maintain one type only for each comment. Based on our understanding, the defect debt expressed in the comment would not exist if the requirement debt did not exists. Therefore, the main debt in this comment is a requirement debt (i.e., incomplete implementation of the copy function).

- **Self-admitted test debt:** Test debt comments are the ones that express the need for implementation or improvement of the current tests. As shown in the examples below, test debt comments are very straight forward in their meaning.

  "// TODO - need a lot more tests" - [from Apache Jmeter]

  "//TODO enable some proper tests!!" - [from Apache Jmeter]

After classifying the comments, we notice that not all of the types mentioned in by Alves *et al.* [ARC$^+$14] could be found. We argue that some types like people debt or infrastructure debt are less probable to appear in source code comments. Other types such as build debt could not be found because we are examining comments in Java classes only, not taking in consideration build scripts that are usually written in other languages (e.g., Maven and Ant use XML files as build scripts).

We find five different types of self-admitted technical debt, i.e., design debt, defect debt, documentation debt, requirement debt and test debt.

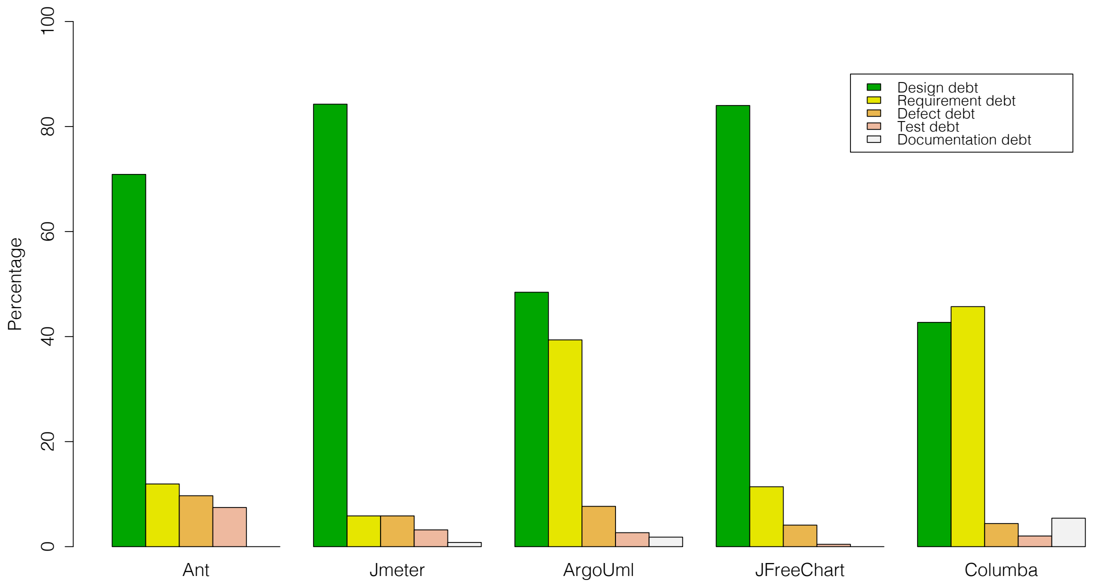


Figure 3: Self-Admitted Technical Debt Types Distribution

In addition to determining the different type of self-admitted technical debt, we would like to quantify the different types. Doing so will help us understand the strengths and weaknesses of using code comments to detect technical debt. After analyzing the more than 33K comments, we found that only 2,457 comments are self-admitted technical debt comments, representing 7.42% (i.e., $\frac{2457}{33093}$) of all the classified comments. The percentage of self-admitted technical debt found for each project is presented in Table 3. ArgoUml is the project with the highest percentage of self-admitted technical debt and Apache Ant has the lowest percentage, amounting to 16.8% and 3.2% respectively.

Figure 3 shows the percentage of each type of self-admitted technical debt across the projects. Since each project has a different number of comments we normalized the data, presenting the percentages of the different types rather than the raw numbers.

Table 3: Self-Admitted Technical Debt per Project

Project	# of analyzed comments	# of self-admitted TD comments	% of self-admitted TD per project
Apache Ant	4,140	134	3.2
Apache Jmeter	8,163	375	4.6
ArgoUML	9,788	1,653	16.8
Columba	6,569	295	4.4
JFreeChart	4,433	219	4.9

For example, if a project has 100 self-admitted technical debt comments and 10 where design debt type, we say that the project has 10% of self-admitted design technical debt.

Analyzing the Figure 3 we find that self-admitted design debt is the most common in 4 out of 5 projects. Self-admitted design technical debt values ranged from 42%, in Columba project with the lowest percentage, to 84% in Jmeter and JFreeChart, projects with the highest percentage. The second most frequent type is self-admitted requirement debt with values between 5% and 45%, followed by self-admitted defect technical debt making up between 4% to 9% of the comments. Self-admitted test technical debt ranged from 0% to 7% whereas self-admitted documentation debt had only 0% to 5% of the comments.

We notice that Columba and ArgoUml have the highest occurrences of self-admitted requirement debt. Columba is a email client application written in Java, which has 9 contributors [Ope], and a considerable number of classes 1,711. It is reasonable to think that developers have limited time to develop features. Therefore, leaving comments of features that need to be implemented in the future (i.e., requirement debt) is more likely.

ArgoUml has a high number of contributors i.e., 87 and yet has a hight number of self-admitted requirement debt. Analyzing the comments we notice that there occurrences about the need of support for internationalization and other comments express the need to implement code to make features compatible with newer versions of the UML language.

Based on that we argue that coupling with external changes that are inherent of the application domain and the adoption of the tool from users all over the world [Ope] had increased the number of self-admitted requirement debt.

We find that the majority of the self-admitted technical debt comments are design debt, which ranged from 42% to 84% across the projects. The second most frequent type was requirement debt that ranged from 5% to 45%. The remaining types have low frequency if considered that they represented less than 10% of the occurrences

3.5 Threats to Validity

Internal validity consider the relationship between theory and observation, in case the measured variables do not measure the actual factors. To classify the source code comments we heavily depended on manual process due the fact that comments are written in natural language and therefore difficult to analyze by a machine. Like any human activity, our manual classification is subject to personal bias and subjectivity. To reduce this bias, in the future, we will ask to other researchers of our lab to classify the dataset as well, verifying and discussing possible divergences of opinion. This is important as changes in this dataset may impact our findings.

When performing our study, we used well-commented Java projects. Since our technique heavily depends on code comments, our results may be impacted by the quantity and quality of comments in a software project. To alleviate the threat, we examined multiple projects. Moreover, there is a risk of removing self-admitted technical debt comments while filtering license comments. To mitigate this risk we do not remove comments that contain one of task-reserved words (i.e., “todo”, “fixme”, or “xxx”).

External validity consider the generalization of our findings. All of our findings were derived from comments in open source projects. To minimize external validity, we chose open source projects from different domains. That said, our results may not generalize to other open source or commercial projects. In particular, our results may not generalize to projects that have a low number or no comments. Other than that, we only analyze projects written in Java, therefore the results obtained may not generalize to projects written in other languages.

3.6 Conclusion and Future Work

The term technical debt is being used for practitioners and researchers in the software engineer community to express shortcuts and workarounds employed in software projects. These shortcuts will most often impact the maintainability of the project hindering the development if not addressed properly. Our work explore specifically self-admitted technical debt, that is the technical debt deliberately introduced by the developers and reported through source code comments.

In our study we analyzed the comments of 5 open source projects which are Apache Ant, Apache Jmeter, ArgoUml , Columba and JFreeChart. These projects are considered well commented and they belong to different application domains. We used them to understand the characteristics of self-admitted technical debt types creating a rich dataset with more than 33,093 classified comments.

We find that self-admitted technical debt can be classified into five types: design debt, defect debt, documentation debt, requirement debt and test debt. We also provide concrete examples of each one of the mentioned types and the rationale to classify them as it was. Moreover, we find that the majority of the self-admitted technical debt comments are design debt. Design debt ranged from 42% to 84% across the projects. The second most frequent type was requirement debt ranging from 5% to 45%. Based on this result, we can say that the self-admitted technical debt types that developers admit to the most are related with the design of the project, potentially indicating that developers feel the need to admit and be forthcoming about such debt. Examining the reasons for these types of debt is an interesting future direction that we plan to pursue.

Moreover, in this chapter we explained our filtering heuristics to remove comments that are not likely to have technical debt. In the next chapter we leveraged all the knowledge obtained by this first study. We will explain in details how we expanded the size of our dataset, doubling the amount of projects being analyzed, and consequently, broadening the application domains involved in the study.

This dataset is then used to train a NLP classifier that is able to automatically identify self-admitted technical debt. To understand how good our approach is in identifying self-admitted technical debt we compare the performance that we obtained with two other baselines, one of these baselines is the current state-of-the-art.

Chapter 4

Proposing an Approach to Automatically Identify Self-Admitted Technical Debt

4.1 Introduction

Developers often have to deal with conflicting goals that require software to be delivered quickly, with high quality, and on budget. In practice, achieving all of these goals at the same time can be challenging, causing a tradeoff to be made. Often, these tradeoffs lead developers to take *shortcuts* or use *workarounds*. Although such shortcuts help developers in meeting their short-term goals, they may have a negative impact in the long-term.

Technical debt is a metaphor coined to express sub-optimal solutions that are taken in a software project in order to achieve some short-term goals [Cun92]. Generally, these decisions allow the project to move faster in the short-term, but introduce an increased cost (i.e., debt) to maintain this software in the long run [SG11, KNOF13]. Prior work has shown that technical debt is widespread in the software domain, is unavoidable, and can have a negative impact on the quality of the software [LTS12].

Technical debt can be deliberately or inadvertently incurred [Mar]. Inadvertent technical debt is technical debt that is taken on unknowingly. One example of inadvertent technical debt is architectural decay or architectural drift. To date, the majority of the technical debt work has focused on inadvertent technical debt [NOKGR12]. On the other hand, deliberate technical debt, is debt that is incurred by the developer with knowledge that it is being taken on. One example of such deliberate technical debt, is self-admitted technical debt, which is the focus of our paper.

Due to the importance of technical debt, a number of studies empirically examined technical debt and proposed techniques to enable its detection and management. Some of the approaches analyze the source code to detect technical debt, whereas other approaches leverage various techniques and artifacts, e.g., documentation and architecture reviews, to detect documentation debt, test debt or architecture debt (i.e., unexpected deviance from the initial architecture) [AMdM$^+$16, XCK$^+$16].

The main findings of prior work are three-fold. First, there are different types of technical debt, e.g., defect debt, design debt, testing debt, and that among them design debt has the highest impact [ARC$^+$14, Mar12]. Second, static source code analysis helps in detecting technical debt, (i.e., code smells) [Mar04, MGV10, ZSV$^+$13]. Third, more recently, our work has shown that it is possible to identify technical debt through source comments, referred to as self-admitted technical debt [PS14], and that design and requirement debt are the most common types of self-admitted technical debt [MS15].

The recovery of technical debt through source code comments has two main advantages over traditional approaches based on source code analysis. First, it is more lightweight compared to source code analysis, since it does not require the construction of Abstract Syntax Trees or other more advanced source code representations.

For instance, some code smell detectors that also provide refactoring recommendations to resolve the detected code smells [TC11, TMK15] generate computationally expensive program representation structures, such as program dependence graphs [Gra10], and method call graphs [AL12] in order to match structural code smell patterns and compute metrics. On the other hand, the source code comments can be easily and efficiently extracted from source code files using regular expressions. Second, it does not depend on arbitrary metric threshold values, which are required in all metric-based code smell detection approaches. Deriving appropriate threshold values is a challenging open problem that has attracted the attention and effort of several researchers [OVPL14, FFZY15, FMZM15]. As a matter of fact, the approaches based on source code analysis suffer from high false positive rates [FDW$^+$16] (i.e., they flag a large number of source code elements as problematic, while they are not perceived as such by the developers), because they rely only on the structure of the source code to detect code smells without taking into account the developers' feedback, the project domain, and the context in which the code smells are detected.

However, relying solely on the developers' comments to recover technical debt is not adequate, because developers might be unaware of the presence of some code smells in their project, or might not be very familiar with good design and coding practices (i.e., inadvertent debt). As a result, the detection of technical debt through source code comments can be only used as a complementary approach to existing code smell detectors based on source code analysis. We believe that self-admitted technical debt can be useful to prioritize the *pay back* of debt (i.e., develop a *pay back* plan), since the technical debt expressed in the comments written by the developers themselves is definitely more relevant to them.

Despite the advantages of recovering technical debt from source code comments, the research in self-admitted technical debt, thus far, heavily relies on the manual inspection of code comments. The current-state-of-the art approach [PS14] uses 62 comment patterns (i.e., words and phrases) derived after the manual examination of more than 100K comments. The manual inspection of code comments is subject to reader bias, time consuming and, as any other manual task, susceptible to errors. These limitations in the identification of self-admitted technical debt comments makes the current state-of-the-art approach difficult to be applied in practice.

Therefore, in this paper we investigate the efficiency of using Natural Language Processing (NLP) techniques to automatically detect the two most common types of self-admitted technical debt, i.e., design and requirement debt. We analyze ten open source projects from different application domains, namely, Ant, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JMeter, JRuby and SQuirrel SQL. We extract and classify the source comments of these projects. Then, using the classified dataset we train a maximum entropy classifier using the Stanford Classifier tool [MK03] to identify design and requirement self-admitted technical debt. The advantages of the maximum entropy classifier over keyword-based and pattern-based approaches, such as comment patterns, are twofold. First, the maximum entropy classifier automatically extracts the most important features (i.e., words) for each class (i.e., design self-admitted technical debt, requirement self-admitted technical debt, and without technical debt) based on a classified training dataset given as input. Second, the maximum entropy classifier, apart from finding features that contribute positively to the classification of a comment in a given class, also finds features that contribute negatively to the classification of a comment in a given class.

We perform a leave-one-out cross-project validation (i.e., we train on nine projects and test on one project). Our results show that we are able to achieve an average F1-measure of 0.620 when identifying design self-admitted technical debt, and an average F1-measure of 0.403 when identifying requirement self-admitted technical debt. We compare the performance of our approach to a simple (random) baseline and the state-of-the-art approach used to detect self-admitted technical debt [PS14].

Our results show that on average, we outperform the state-of-the-art by 2.3 times, when detecting design debt, and by 6 times when detecting requirement debt.

To better understand how developers express technical debt we analyze the 10 most prevalent words appearing within self-admitted technical debt comments. We find that the top design debt words are related to sloppy or mediocre source code. For example, words such as 'hack', 'workaround' and 'yuck!' are used to express design self-admitted technical debt. On the other hand, for requirement debt, words indicating the need to complete a partially implemented requirement are the best indicators. For example, words such as 'todo', 'needed' and 'implementation' are strong indicators of requirement debt.

Finally, to determine the most efficient way to apply our approach, we analyze the amount of training data necessary to effectively identify self-admitted technical debt. We find that training datasets using 23% of the available data can achieve a performance equivalent to 90% of the maximum F1-measure score for both design and requirement self-admitted technical debt. Similarly, 80% of the maximum F1-measure can be achieved using only 9% of the available data for design self-admitted technical debt, and 5% for requirement self-admitted technical debt.

The main contributions of our work are the following:

- We provide an automatic, NLP-based, approach to identify design and requirement self-admitted technical debt.
- We examine and report the words that best indicate design and requirement self-admitted technical debt.
- We show that using a small training set of comments, we are able to effectively detect design and requirement self-admitted technical debt.
- We make our dataset publicly available[1], so that others can advance work in

[1]https://github.com/maldonado/tse_satd_data

the area of self-admitted technical debt.

The rest of the paper is organized as follows. Section 4.2 describes our approach. We setup our experiment and present our results in Section 4.3. We discuss the implications of our findings in Section 4.4. In Section 4.5 we present the related work. Section 4.6 presents the threats to validity and Section 4.7 presents our conclusions and future work.

4.2 Approach

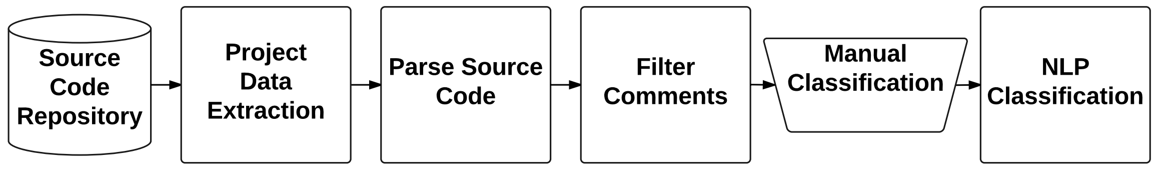


Figure 4: NLP Based Approach Overview

The main goal of our study is to automatically identify self-admitted technical debt through source code comments. To do that, we first extract the comments from ten open source projects. Second, we apply five filtering heuristics to remove comments that are irrelevant for the identification of self-admitted technical debt (e.g., license comments, commented source code and Javadoc comments). After that, we manually classify the remaining comments into the different types of self-admitted technical debt (i.e., design debt, requirement debt, defect debt, documentation debt and test debt). Lastly, we use these comments as training data for the maximum entropy classified and use the trained model to detect self-admitted technical debt from source code comments. Figure 4 shows an overview of our approach, and the following subsections detail each step.

4.2.1 Project Data Extraction

To perform our study, we need to analyze the source code comments of software projects. Therefore, we focused our study on ten open source projects: Ant is a build tool written in Java, ArgoUML is an UML modeling tool that includes support for all standard UML 1.4 diagrams, Columba is an email client that has a graphical interface with wizards and internationalization support, EMF is a modeling framework and code generation facility for building tools and other applications, Hibernate is a component providing Object Relational Mapping (ORM) support to applications and other components, JEdit is a text editor written in Java, JFreeChart is a chart library for the Java platform, JMeter is a Java application designed to load functional test behavior and measure performance, JRuby is a pure-Java implementation of the Ruby programming language and SQuirrel SQL is a graphical SQL client written in Java. We selected these projects since they belong to different application domains, are well commented, vary in size, and in the number of contributors.

Table 4 provides details about each of the projects used in our study. The columns of Table 4 present the release used, followed by the number of classes, the total source lines of code (SLOC), the number of contributors, the number of extracted comments, the number of comments analyzed after applying our filtering heuristics, and the number of comments that were classified as self-admitted technical debt together with the percentage of the total project comments that it represent. The final three columns show the percentage of self-admitted technical debt comments classified as design debt, requirement debt, and all other remaining types of debt (i.e., defect, documentation and test debt), respectively.

Since there are many different definitions for the SLOC metric we clarify that, in our study, a source line of code contains at least one valid character, which is not a blank space or a source code comment. In addition, we only use the Java files to calculate the SLOC, and to do so, we use the SLOCCount tool [Whe04].

The number of contributors was extracted from OpenHub, an on-line community and public directory that offers analytics, search services and tools for open source software [Ope]. It is important to note that the number of comments shown for each project does not represent the number of commented lines, but rather the number of Single-line, Block and Javadoc comments. In total, we obtained 259,229 comments, found in 16,249 Java classes. The size of the selected projects varies between 81,307 and 228,191 SLOC, and the number of contributors of these projects ranges from 9 to 328.

Table 4: Details of All Studied Projects

Project	Project Details				Comments Details			Technical Debt Details		
	Release	**# of Classes**	**SLOC**	**# of Contributors**	**# of Comments**	**# of Comments After Filtering**	**#,(%) of TD Comments**	**% of Design Debt**	**% of Requirement Debt**	**% of Other Debt**
Ant	1.7.0	1,475	115,881	74	21,587	4,137	131 (0.60)	72.51	09.92	17.55
ArgoUML	0.34	2,609	176,839	87	67,716	9,548	1,413 (2.08)	56.68	29.08	14.22
Columba	1.4	1,711	100,200	9	33,895	6,478	204 (0.60)	61.76	21.07	17.15
EMF	2.4.1	1,458	228,191	30	25,229	4,401	104 (0.41)	75.00	15.38	09.61
Hibernate	3.3.2 GA	1,356	173,467	226	11,630	2,968	472 (4.05)	75.21	13.55	11.22
JEdit	4.2	800	88,583	57	16,991	10,322	256 (1.50)	76.56	05.46	17.96
JFreeChart	1.0.19	1,065	132,296	19	23,474	4,423	209 (0.89)	88.03	07.17	04.78
JMeter	2.10	1,181	81,307	33	20,084	8,162	374 (1.86)	84.49	05.61	09.89
JRuby	1.4.0	1,486	150,060	328	11,149	4,897	622 (5.57)	55.14	17.68	27.17
SQuirrel	3.0.3	3,108	215,234	46	27,474	7,230	286 (1.04)	73.07	17.48	09.44
Average		1,625	146,206	91	25,923	6,257	407 (1.86)	71.84	14.24	13.89
Total		16,249	1,462,058	909	259,229	62,566	4,071 (-)	-	-	-

4.2.2 Parse Source Code

After obtaining the source code of all projects, we extract the comments from the source code. We use JDeodorant [TCC08], an open-source Eclipse plug-in, to parse the source code and extract the code comments. JDeodorant provides detailed information about the source code comments such as: their type (i.e., Block, Single-line, or Javadoc), their location (i.e., the lines where they start and end), and their context (i.e., the method/field/type declaration they belong to).

Due to these features, we adapted JDeodorant to extract the aforementioned information about source code comments and store it in a relational database to facilitate the processing of the data.

4.2.3 Filter Comments

Source code comments can be used for different purposes in a project, such as giving context, documenting, expressing thoughts, opinions and authorship, and in some cases, disabling source code from the program. Comments are used freely by developers and with limited formalities, if any at all. This informal environment allows developers to bring to light opinions, insights and even confessions (e.g., self-admitted technical debt).

As shown in prior work [MS15], part of these comments may discuss self-admitted technical debt, but not the majority of them. With that in mind, we develop and apply 5 filtering heuristics to narrow down the comments eliminating the ones that are less likely to be classified as self-admitted technical debt.

To do so, we developed a Java based tool that reads from the database the data obtained by parsing the source code. Next, it executes the filtering heuristics and stores the results back in the database. The retrieved data contains information like the line number that a class/comment starts/ends and the comment type, considering the Java syntax (i.e., Single-line, Block or Javadoc). With this information we process the filtering heuristics as described next.

License comments are not very likely to contain self-admitted technical debt, and are commonly added before the declaration of the class. We create a heuristic that removes comments that are placed before the class declaration. Since we know the line number that the class was declared we can easily check for comments that are placed before that line and remove them. In order to decrease the chances of removing a self-admitted technical debt comment while executing this filter we calibrated this heuristic to avoid removing comments that contain one of the predefined task annotations (i.e., "TODO:", "FIXME:", or "XXX:") [SRB$^+$08]. Task annotations are an extended functionality provided by most of the popular Java *IDEs* including Eclipse, InteliJ and NetBeans. When one of these words is used inside a comment the IDE will automatically keep track of the comment creating a centralized list of tasks that can be conveniently accessed later on.

Long comments that are created using multiple *Single-line* comments instead of a *Block* comment can hinder the understanding of the message considering the case that the reader (i.e., human or machine) analyzes each one of these comments independently. To solve this problem, we create a heuristic that searches for consecutive single-line comments and groups them as one comment.

Commented source code is found in the projects due to many different reasons. One of the possibilities is that the code is not currently being used. Other is that, the code is used for debugging purposes only. Based on our analysis, commented source code does not have self-admitted technical debt. Our heuristic removes commented source code using a simple regular expression that captures typical Java code structures.

Automatically generated comments by the IDE are filtered out as well. These comments are inserted as part of code snippets used to generate constructors, methods and try catch blocks, and have a fixed format (i.e., "Auto-generated constructor stub", "Auto-generated method stub", and "Auto-generated catch block"). Therefore our heuristic searches for these automatically generated comments and removes them.

Javadoc comments rarely mention self-admitted technical debt. For the Javadoc comments that do mention self-admitted technical debt, we notice that they usually contain one of the task annotations (i.e., "TODO:", "FIXME:", or "XXX:"). Therefore, our heuristic removes all comments of the Javadoc type, unless they contain at least one of the task annotations. To do so, we create a simple regular expression that searches for the task annotations before removing the comment.

The steps mentioned above significantly reduced the number of comments in our dataset and helped us focus on the most applicable and insightful comments. For example, in the Ant project, applying the above steps helped to reduce the number of comments from 21,587 to 4,137 resulting in a reduction of 80.83% in the number of comments to be manually analyzed. Using the filtering heuristics we were able to remove from 39.25% to 85.89% of all comments. Table 4 provides the number of comments kept after the filtering heuristics for each project.

4.2.4 Manual Classification

Our goal is to inspect each comment and label it with a suitable technical debt classification. Since there are many comments, we developed a Java based tool that shows one comment at a time and gives a list of possible classifications that can be manually assigned to the comment. The list of possible classifications is based on previous work by Alves *et al.* [ARC+14]. In their work, an ontology on technical debt terms was proposed, and they identified the following types of technical debt across the researched literature: architecture, build, code, defect, design, documentation, infrastructure, people, process, requirement, service, test automation and test debt. During the classification process, we notice that not all types of debt mentioned by Alves *et al.* [ARC+14] could be found in code comments. However, we were able to identify the following types of debt in the source comments: design debt, defect debt, documentation debt, requirement debt and test debt.

In our previous work [MS15], we manually classified 33,093 comments extracted from the following projects: Ant, ArgoUML, Columba, JFreeChart and JMeter. In the current study we manually classified an additional 29,473 comments from EMF, Hibernate, JEdit, JRuby and SQuirrel, which means that we extended our dataset of classified comments by 89.06%. In total, we manually classified 62,566 comments into the five different types of self-admitted technical debt mentioned above. The classification process took approximately 185 hours in total, and was performed by the first author of the paper. It is important to note that this manual classification step does not need to be repeated in order to apply our approach, since our dataset is publicly available[2], and thus it can used as is, or even extended with new classified comments.

Below, we provide definitions for design and requirement self-admitted technical debt, and some indicative comments to help the reader understand the different types of self-admitted technical debt comments.

Self-admitted design debt: These comments indicate that there is a problem with the design of the code. They can be comments about misplaced code, lack of abstraction, long methods, poor implementation, workarounds, or temporary solutions. Usually these kinds of issues are resolved through refactoring (i.e., restructuring of *existing* code), or by re-implementing *existing* code to make it faster, more secure, more stable and so forth. Let us consider the following comments:

"TODO: - This method is too complex, lets break it up" - [from ArgoUml]

"/ TODO: really should be a separate class */"* - [from ArgoUml]

[2]https://github.com/maldonado/tse_satd_data

These comments are clear examples of what we consider as self-admitted *design debt*. In the above comments, the developers state what needs to be done in order to improve the current design of the code, and the payback of this kind of design debt can be achieved through refactoring. Although the above comments are easy to understand, during our study we came across more challenging comments that expressed design problems in an indirect way. For example:

> *"// I hate this so much even before I start writing it. // Re-initialising a global in a place where no-one will see it just // feels wrong. Oh well, here goes."* - [from ArgoUml]

> *"//quick & dirty, to make nested mapped p-sets work:"* - [from Apache Ant]

In the above example comments the authors are certain to be implementing code that does not represent the best solution. We assume that this kind of implementation will degrade the design of the code and should be avoided.

> *"// probably not the best choice, but it solves the problem of // relative paths in CLASSPATH"* - [from Apache Ant]

> *"//I can't get my head around this; is encoding treatment needed here?"* - [from Apache Ant]

The above comments expressed doubt and uncertainty when implementing the code and were considered as self-admitted design debt as well. The payback of the design debt expressed in the last four example comments can be achieved through the re-implementation of the currently existing solution.

Self-admitted requirement debt: These comments convey the opinion of a developer supporting that the implementation of a requirement is not complete. In general, requirement debt comments express that there is still *missing* code that needs to be added in order to complete a *partially* implemented requirement, as it can be observed in the following comments:

> *"/TODO no methods yet for getClassname"* - [from Apache Ant]

> *"//TODO no method for newInstance using a reverse-classloader"* - [from Apache Ant]

> *"TODO: The copy function is not yet * completely implemented - so we will * have some exceptions here and there.*/"* - [from ArgoUml]

> *"TODO: This dialect is not yet complete. Need to provide implementations wherever Not yet implemented appears"* - [from SQuirrel]

To mitigate the risk of creating a dataset that is biased, we extracted a statistically significant sample of our dataset and asked another student to classify it. To prepare the student for the task we gave a 1-hour tutorial about the different kinds of self-admitted technical debt, and walked the student through a couple of examples of each different type of self-admitted technical debt comment. The statistically significant sample was created based on the total number of comments (62,566) with a confidence level of 99% and a confidence interval of 5%, resulting in a stratified sample of 659 comments. We composed the stratified sample according to the percentage of each classification found in the original dataset. Therefore, the stratified sample was composed of: 92% comments without self-admitted technical debt (609 comments), 4% design debt (29 comments), 2% requirement debt (5 comments), 0.75% test debt

(2 comments) and 0.15% documentation debt (1 comment). Lastly, we evaluate the level of agreement between both reviewers of the stratified sample by calculating Cohen's kappa coefficient [Coh60]. The Cohen's Kappa coefficient has been commonly used to evaluate inter-rater agreement level for categorical scales, and provides the proportion of agreement corrected for chance. The resulting coefficient is scaled to range between -1 and +1, where a negative value means poorer than chance agreement, zero indicates exactly chance agreement, and a positive value indicates better than chance agreement [FC73]. The closer the value is to +1, the stronger the agreement. In our work, the level of agreement measured between the reviewers was of +0.81.

We also measured the level of agreement in the classification of design and requirement self-admitted technical debt individually. This is important because the stratified sample contains many more comments without self-admitted technical debt than the other types of debt, and therefore, the coefficient reported above could indicate that the reviewers are agreeing on what is not self-admitted technical debt, instead of agreeing on a particular type of debt. However, we achieved a level of agreement of +0.75 for design self-admitted technical debt, and +0.84 for requirement self-admitted technical debt. According to Fleiss [Fle81] values larger than +0.75 are characterized as excellent agreement.

4.2.5 NLP Classification

Our next step is to use the classified self-admitted technical debt comments as a training dataset for the Stanford Classifier, which is a Java implementation of a maximum entropy classifier [MK03]. A maximum entropy classifier, in general, takes as input a number of data items along with a classification for each data item, and automatically generates *features* (i.e., words) from each *datum*, which are associated with positive or negative numeric *votes* for each class. The weights of the features are learned automatically based on the manually classified training data items (supervised learning). The Stanford Classifier builds a *maximum entropy model*, which is equivalent to a multi-class regression model, and it is trained to maximize the conditional likelihood of the classes taking into account feature dependences when calculating the feature weights.

After the training phase, the maximum entropy classifier can take as input a test dataset that will be classified according to the model built during the training phase. The output for each data item in the test dataset is a classification, along with the features contributing positively or negatively in this classification.

In our case, the training dataset is composed of source code comments and their corresponding manual classification. According to our findings in previous work [MS15], the two most common types of self-admitted technical debt are design and requirement debt (defect, test, and documentation debt together represent less that 10% of all self-admitted technical debt comments). Therefore, we train the maximum entropy classifier on the dataset containing only these two specific types of self-admitted technical debt comments.

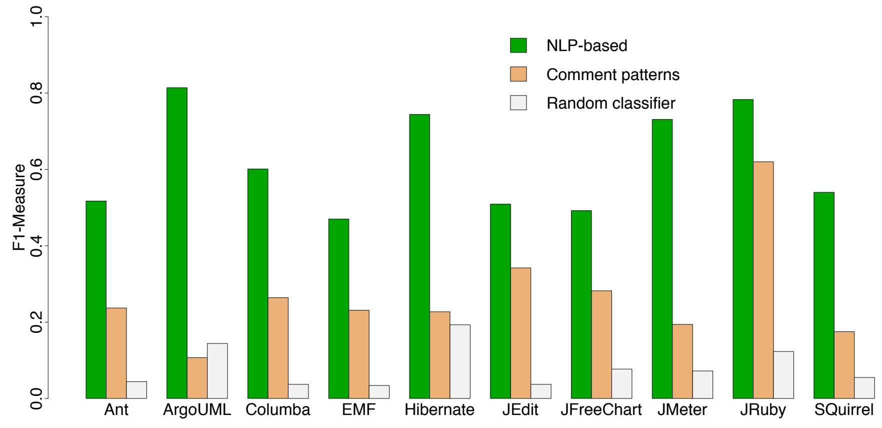
In order to avoid having repeated features differing only in letter case (e.g., "Hack", "hack", "HACK"), or in preceding/succeeding punctuation characters (e.g., "**,**hack", "hack**,**"), we preprocess the training and test datasets to clean up the original comments written by the developers. More specifically, we remove the character structures that are used in the Java language syntax to indicate comments (i.e., '//' or '/*' and '*/'), the punctuation characters, and any excess whitespace characters (e.g., ' ', '\t', '\n'), and finally we convert all comments to lowercase. However, we decided not to remove exclamation and interrogation marks. These specific punctuations were very useful during the identification of self-admitted technical debt comments, and provide insightful information about the meaning of the features.

4.3 Experiment Results

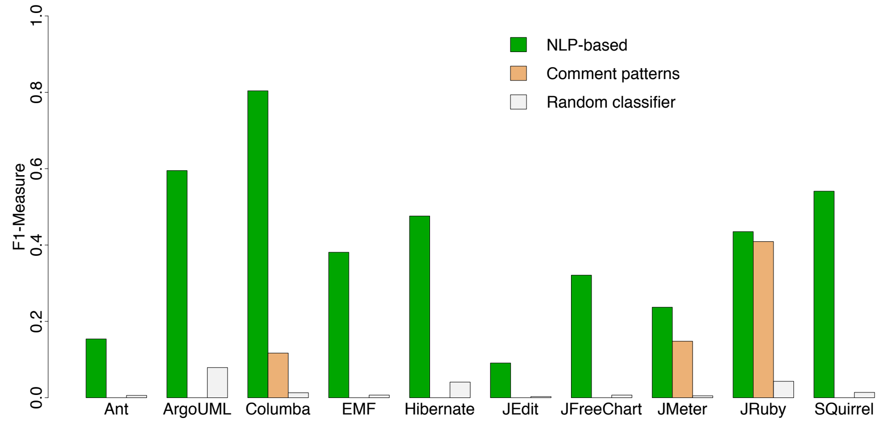
The goal of our research is to develop an automatic way to detect design and requirement self-admitted technical debt comments. To do so, we first manually classify a large number of comments identifying those containing self-admitted technical debt. With the resulting dataset, we train the maximum entropy classifier to identify design and requirement self-admitted technical debt (RQ1). To better understand what words indicate self-admitted technical debt, we inspect the features used by the maximum entropy classifier to identify the detected self-admitted technical debt. These features are words that are frequently found in comments with technical debt. We present the 10 most common words that indicate design and requirement self-admitted technical debt (RQ2). Since the manual classification required to create our training dataset is expensive, ideally we would like to achieve maximum performance with the least amount of training data. Therefore, we investigate how variations in the size of training data affects the performance of our classification (RQ3). We detail the motivation, approach and present the results of each of our research questions in the remainder of this section.

Table 5: Comparison of F1-measure Between the NLP-based, the Comment Patterns and the Random Baseline Approaches for Design and Requirement Debt

Project	Design Debt					Requirement Debt				
	Our Approach	**Comment Patterns**	**Random Classifier**	**Imp. Over Comment Patterns**	**Imp. Over Random Classifier**	**Our Approach**	**Comment Patterns**	**Random Classifier**	**Imp. Over Comment Patterns**	**Imp. Over Random Classifier**
Ant	0.517	0.237	0.044	2.1×	11.7 ×	0.154	0.000	0.006	-	25.6×
ArgoUML	0.814	0.107	0.144	7.6×	5.6 ×	0.595	0.000	0.079	-	7.5 ×
Columba	0.601	0.264	0.037	2.2×	16.2 ×	0.804	0.117	0.013	6.8 ×	61.8×
EMF	0.470	0.231	0.034	2.0×	13.8 ×	0.381	0.000	0.007	-	54.4×
Hibernate	0.744	0.227	0.193	3.2×	3.8 ×	0.476	0.000	0.041	-	11.6×
JEdit	0.509	0.342	0.037	1.4×	13.7 ×	0.091	0.000	0.003	-	30.3×
JFreeChart	0.492	0.282	0.077	1.7×	6.3 ×	0.321	0.000	0.007	-	45.8×
JMeter	0.731	0.194	0.072	3.7×	10.1 ×	0.237	0.148	0.005	1.6 ×	47.4×
JRuby	0.783	0.620	0.123	1.2×	6.3 ×	0.435	0.409	0.043	1.0 ×	10.1×
SQuirrel	0.540	0.175	0.055	3.0×	9.8 ×	0.541	0.000	0.014	-	38.6×
Average	0.620	0.267	0.081	2.3×	7.6 ×	0.403	0.067	0.021	6.0 ×	19.1×



(a) Design Debt



(b) Requirement Debt

Figure 5: Visualization of the F1-measure for Different Approaches

RQ1. Is it possible to more accurately detect self-admitted technical debt using NLP techniques?

Motivation: As shown in previous work [MS15], self-admitted technical debt comments can be found in the source code. However, there is no automatic way to identify these comments. The methods proposed so far heavily rely on the manual inspection of source code, and there is no evidence on how well these approaches perform.

Moreover, most of them do not discriminate between the different types of technical debt (e.g., design, test, requirement).

Therefore, we want to determine if NLP techniques such as, the maximum entropy classifier, can help us surpass these limitations and outperform the accuracy of the current state-of-the-art. The maximum entropy classifier can automatically classify comments based on specific linguistic characteristics of these comments. Answering this question is important, since it helps us understand the opportunities and limitations of using NLP techniques to automatically identify self-admitted technical debt comments.

Approach: For this research question, we would like to examine how effectively we can identify design and requirement self-admitted technical debt. Therefore, the first step is to create a dataset that we can train and test the maximum entropy classifier on. We classified the source code comments into the following types of self-admitted technical debt: design, defect, documentation, requirement, and test debt. However, our previous work showed that the most frequent self-admitted technical debt comments are design and requirement debt. Therefore, in this paper, we focus on the identification of these two types of self-admitted technical debt, because 1) they are the most common types of technical debt, and 2) NLP-based techniques require sufficient data for training (i.e., they cannot build an accurate model with a small number of samples).

We train the maximum entropy classifier using our manually created dataset. The dataset contains comments with and without self-admitted technical debt, and each comment has a classification (i.e., without technical debt, design debt, or requirement debt). Then, we add to the training dataset all comments classified as without technical debt and the comments classified as the specific type of self-admitted technical debt that we want to identify (i.e., design or requirement debt). We use the comments from 9 out of the 10 projects that we analyzed to create the training dataset. The comments from the remaining one project are used to evaluate the classification performed by the maximum entropy classifier. We choose to create the training dataset using comments from 9 out of 10 projects, because we want to train the maximum entropy classifier with the most diverse data possible (i.e., comments from different domains of applications). However, we discuss the implications of using training datasets of different sizes in RQ3. We repeat this process for each one of the ten projects, each time training on the other 9 projects and testing on the remaining 1 project.

Based on the training dataset, the maximum entropy classifier will classify each comment in the test dataset. The resulting classification is compared with the manual classification provided in the test dataset. If a comment in the test dataset has the same manual classification as the classification suggested by the maximum entropy classifier, we will have a true positive (tp) or a true negative (tn). True positives are the cases where the maximum entropy classifier correctly identifies self-admitted technical debt comments, and true negatives are comments without technical debt that are classified as being as such. Similarly, when the classification provided by the tool diverges from the manual classification provided in the test dataset, we have false positives or false negatives. False positives (fp) are comments classified as being self-admitted technical debt when they are not, and false negatives (fn) are comments classified as without technical debt when they really are self-admitted technical debt comments. Using the tp, tn, fp, and fn values, we are able to evaluate the performance of different detection approaches in terms of precision ($P = \frac{tp}{tp+fp}$), recall ($R = \frac{tp}{tp+fn}$) and F1-measure ($F = 2 \times \frac{P \times R}{P+R}$). To determine how effective the NLP classification is, we compare its F1-measure values with the corresponding F1-measure values of the two other approaches. We use the F1-measure to compare the performance between the approaches as it is the harmonic mean of precision and recall. Using the F1-measure allows us to incorporate the trade-off between precision and recall and present one value that evaluates both measures.

The first approach is the current state-of-the-art in detecting self-admitted technical debt comments [PS14]. This approach uses 62 comment patterns (i.e., keywords and phrases) that were found as recurrent in self-admitted technical debt comments during the manual inspection of 101,762 comments. The second approach is a simple (random) baseline, which assumes that the detection of self-admitted technical debt is random (this approach is used as a performance lower bound). The precision of this approach is calculated by taking the total number of self-admitted technical debt over the total number of comments of each project. For example, project Ant has 4,137 comments, of those, only 95 comments are design self-admitted technical debt. The chance of randomly finding a design self-admitted technical debt comment is 0.023 (i.e., $\frac{95}{4{,}137}$). Similarly, to calculate the recall we take into consideration the two possible classifications available: one is the type of self-admitted technical debt (e.g., design) and the other is without technical debt. Therefore, there is a 50% chance that the comment will be classified as self-admitted technical debt. Thus, the F1-measure for the random baseline for project Ant is computed as $2 \times \frac{0.023 \times 0.5}{0.023 + 0.5} = 0.044$.

Results - design debt: Table 5 presents the F1-measure of the three approaches, as well as the improvement achieved by our approach compared to the other two approaches. We see that for all projects, the F1-measure achieved by our approach is higher than the other approaches. The F1-measure values obtained by our NLP-based approach range between 0.470 - 0.814, with an average of 0.620. In comparison, the F1-measure values using the comment patterns range between 0.107 - 0.620, with an average of 0.267, while the simple (random) baseline approach achieves F1-measure values in the range of 0.034 - 0.193, with an average of 0.081. Figure 5(a) visualizes the comparison of the F1-measure values for our NLP-based approach, the comment patterns approach, and the simple (random) baseline approach. We see from both, Table 5 and Figure 5(a) that, on average, our approach outperforms the comment patterns approach by 2.3 times and the simple (random) baseline approach by 7.6 times when identifying design self-admitted technical debt.

It is important to note that the comment patterns approach has a high precision, but low recall, i.e., this approach points correctly to self-admitted technical debt comments, but as it depends on keywords, it identifies a very small subset of all the self-admitted technical debt comments in the project. Although we only show the F1-measure values here, we present the precision and recall values in Table 6.

Results - requirement debt: Similarly, the last five columns of Table 5 show the F1-measure performance of the three approaches, and the improvement achieved by our approach over the two other approaches. The comment patterns approach was able to identify requirement self-admitted technical debt in only 3 of the 10 analyzed projects. A possible reason for the low performance of the comment patterns in detecting requirement debt is that the comment patterns do not differentiate between the different types of self-admitted technical debt. Moreover, since most of the debt is design debt, it is possible that the patterns tend to favor the detection of design debt.

That said, we find that for all projects, the F1-measure values obtained by our approach surpass the F1-measure values of the other approaches. Our approach achieves F1-measure values between 0.091 - 0.804 with an average of 0.403, whereas the comment pattern approach achieves F1-measure values in the range of 0.117 - 0.409 with an average of 0.067, while the simple (random) baseline ranges between 0.003 - 0.079, with an average of 0.021. Figure 5(b) visualizes the performance comparison of the two approaches. We also examine if the differences in the F1-measure values obtained by our approach and the other two baselines are statistically significant. Indeed, we find that the differences are statistically significant ($p<0.001$) for both baselines and both design and requirement self-admitted technical debt.

Generally, requirement self-admitted technical debt is less common than design self-admitted technical debt, which makes it more difficult to detect. Nevertheless, our NLP-based approach provides a significant improvement over the comment patterns approach, outperforming it by 6 times, on average. Table 5 only presents the F1-measure values for the sake of brevity, however, we present the detailed precision and recall values in Table 7.

We find that our NLP-based approach, is more accurate in identifying self-admitted technical debt comments compared to the current state-of-art. We achieved an average F1-measure of 0.620 when identifying design debt (an average improvement of 2.3× over the state-of-the-art approach) and an average F1-measure of 0.403 when identifying requirement debt (an average improvement of 6× over the state-of-the-art approach).

Table 6: Detailed Comparison of F1-measure Between the NLP-based, the Comment Patterns and the Random Baseline Approaches for Design Debt

Project	NLP-based			Comment Patterns			Random Baseline		
	Precision	**Recall**	**F1 measure**	**Precision**	**Recall**	**F1 measure**	**Precision**	**Recall**	**F1 measure**
Ant	0.554	0.484	0.517	0.608	0.147	0.237	0.023	0.5	0.044
ArgoUML	0.788	0.843	0.814	0.793	0.057	0.107	0.084	0.5	0.144
Columba	0.792	0.484	0.601	0.800	0.158	0.264	0.019	0.5	0.037
EMF	0.574	0.397	0.470	0.647	0.141	0.231	0.018	0.5	0.034
Hibernate	0.877	0.645	0.744	0.920	0.129	0.227	0.12	0.5	0.193
JEdit	0.779	0.378	0.509	0.857	0.214	0.342	0.019	0.5	0.037
JFreeChart	0.646	0.397	0.492	0.507	0.195	0.282	0.042	0.5	0.077
JMeter	0.808	0.668	0.731	0.813	0.110	0.194	0.039	0.5	0.072
JRuby	0.798	0.770	0.783	0.864	0.483	0.620	0.07	0.5	0.123
SQuirrel	0.544	0.536	0.540	0.700	0.100	0.175	0.029	0.5	0.055

Table 7: Detailed Comparison of F1-measure Between the NLP-based, the Comment Patterns and the Random Baseline Approaches for Requirement Debt

Project	NLP-based			Comment Patterns			Random Baseline		
	Precision	**Recall**	**F1 measure**	**Precision**	**Recall**	**F1 measure**	**Precision**	**Recall**	**F1 measure**
Ant	0.154	0.154	0.154	0.000	0.000	0.000	0.003	0.5	0.006
ArgoUML	0.663	0.540	0.595	0.000	0.000	0.000	0.043	0.5	0.079
Columba	0.755	0.860	0.804	0.375	0.069	0.117	0.007	0.5	0.013
EMF	0.800	0.250	0.381	0.000	0.000	0.000	0.004	0.5	0.007
Hibernate	0.610	0.391	0.476	0.000	0.000	0.000	0.022	0.5	0.041
JEdit	0.125	0.071	0.091	0.000	0.000	0.000	0.001	0.5	0.003
JFreeChart	0.220	0.600	0.321	0.102	0.266	0.148	0.003	0.5	0.007
JMeter	0.153	0.524	0.237	0.000	0.000	0.000	0.003	0.5	0.005
JRuby	0.686	0.318	0.435	0.573	0.318	0.409	0.022	0.5	0.043
SQuirrel	0.657	0.460	0.541	0.000	0.000	0.000	0.007	0.5	0.014

RQ2. What are the most impactful words in the classification of self-admitted technical debt?

Motivation: After assessing the accuracy of our NLP-based approach in identifying self-admitted technical debt comments, we want to better understand what words developers use when expressing technical debt. Answering this question will provide insightful information that can guide future research directions, broaden our understanding on self-admitted technical debt and also help us to detect it.

Approach: The maximum entropy classifier learns optimal features that can be used to detect self-admitted technical debt. A feature is comment fragment (e.g., word) that is associated with a specific class (i.e., design debt, requirement debt, or without technical debt), and a weight that represents how strongly this feature relates to that class. The maximum entropy classifier uses the classified training data to determine the features and their weights. Then, these features and their corresponding weights are used to determine if a comment belongs to a specific type of self-admitted technical debt or not.

For example, let us assume that after the training, the maximum entropy classifier determines that the features "hack" and "dirty" are related to the *design-debt* class with weights 5.3 and 3.2, respectively, and the feature "something" relates to the *without-technical-debt* class with a weight of 4.1. Then, to classify the comment "this is a dirty hack it's better to do something" from our test data, all features present in the comment will be examined and the following scores would be calculated: $weight_{design-debt} = 8.5$ (i.e., the sum of "hack" and "dirty" feature weights) and $weight_{without-technical-debt} = 4.1$. Since $weight_{design-debt}$ is larger than $weight_{without-technical-debt}$, the comment will be classified as design debt.

For each analyzed project, we collect the features used to identify the self-admitted technical debt comments. These features are provided by the maximum entropy classifier as output and stored in a text file. The features are written in the file according to their weights in descending order (starting from more relevant, ending to less relevant features). Based on these files, we rank the words calculating the average ranking position of the analyzed features across the ten different projects.

Results: Table 8 shows the top-10 textual features used to identify self-admitted technical debt in the ten studied projects, ordered by their average ranking. The first column shows the ranking of each textual feature, the second column lists the features used in the identification of *design* self-admitted technical debt, and the third column lists the textual features used to identify *requirement* self-admitted technical debt.

From Table 8 we observe that the top ranked textual features for design self-admitted technical debt, i.e., *hack*, *workaround*, *yuck!*, *kludge* and *stupidity*, indicate sloppy code, or mediocre source code quality. For example, we have the following comment that was found in JMeter:

*"**Hack** to allow entire URL to be provided in host field"*

Other textual features, such as *needed?*, *unused?* and *wtf?* are questioning the usefulness or utility of a specific source code fragment, as indicated by the following comment also found in JMeter:

*"TODO: - is this **needed?**"*

For requirement self-admitted technical debt, the top ranked features, i.e., *todo*, *needed*, *implementation*, *fixme* and *xxx* indicate the need to complete requirements in the future that are currently partially complete. An indicative example is the following one found in JRuby:

*"**TODO:** implement, won't do this now"*

Some of the remaining lower ranked textual features, such as *convention*, *configurable* and *fudging* also indicate potential incomplete requirements, as shown in the following comments:

> *"Need to calculate this... just **fudging** here for now"* [from JEdit]

> *"could make this **configurable**"* [from JFreeChart]

> *"TODO: This name of the expression language should be **configurable** by the user"* [from ArgoUML]

> *"TODO: find a way to check the manifest-file, that is found by naming **convention**"* [from Apache Ant]

Table 8: Top-10 Textual Features Used to Identify Design and Requirement Self-Admitted Technical Debt

Rank	**Design Debt**	**Requirement Debt**
1	**hack**	**todo**
2	**workaround**	**needed**
3	yuck!	**implementation**
4	kludge	**fixme**
5	stupidity	**xxx**
6	needed?	ends?
7	columns?	convention
8	unused?	configurable
9	wtf?	apparently
10	todo	fudging

It should be noted that the features highlighted in bold in Table 8 appear in all top-10 lists extracted from each one of the ten training datasets, and therefore can be considered as more *universal/stable* features compared to the others.

We also observe that it is possible for a single textual feature to indicate both design and requirement self-admitted technical debt. However, in such cases, the ranking of the feature is different for each kind of debt. For example, the word “todo” is ranked tenth for design debt, whereas it is ranked first for requirement debt. This finding is intuitive, since requirement debt will naturally be related to the implementation of future functionality.

It is important to note here that although we present only the top-10 textual features, the classification of the comments is based on a combination of a large number of textual features. In fact, two different types of textual features are used to classify the comments, namely positive and negative weight features. Positive weight features will increase the total weight of the vote suggesting that the classification should be equal to the class of the feature (i.e., design or requirement debt). On the other hand, negative weight features will decrease the total weight of the vote suggesting a classification different from the class of the feature. On average, the number of positive weight features used to classify design and requirement debt is 5,014 and 2,195, respectively. The exact number of unique textual features used to detect self-admitted technical debt for each project is shown in Table 9. The fact that our NLP-based approach leverages so many features helps to explain the significant improvement we are able to achieve over the state-of-the-art, which only uses 62 patterns. In comparison, our approach leverages 35,828 and 34,056 unique textual features for detecting comments with design and requirement debt, respectively.

We find that design and requirement debt have their own textual features that best indicate such self-admitted technical debt comments. For design debt, the top textual features indicate sloppy code or mediocre code quality, whereas for requirement debt they indicate the need to complete a partially implemented requirement in the future.

Table 9: Number of Unique Textual Features Use to Detect Design and Requirement Debt for Each Project

Project	**Design Debt**			**Requirement Debt**		
	Positive Weight Features	**Negative Weight Features**	**# of Features**	**Positive Weight Features**	**Negative Weight Features**	**# of Features**
Ant	5,299	23,623	28,922	1,812	27,673	29,485
ArgoUML	3,917	26,012	29,929	2,779	27,260	30,039
Columba	5,255	24,182	29,437	2,433	27,561	29,994
EMF	5,346	23,667	29,013	1,889	27,637	29,526
Hibernate	4,914	24,070	28,984	2,748	26,654	29,402
JEdit	5,042	24,644	29,686	1,831	28,267	30,098
JFreeChart	5,361	23,530	28,891	1,902	27,439	29,341
JMeter	5,172	23,916	29,088	1,893	27,716	29,609
JRuby	4,856	24,553	29,409	2,850	27,085	29,935
SQuirrel	4,982	25,146	30,128	1,814	26,914	28,728
Average	5,014	24,334	29,348	2,195	27,420	29,615
Total unique	6,327	31,518	35,828	4,015	32,954	34,056

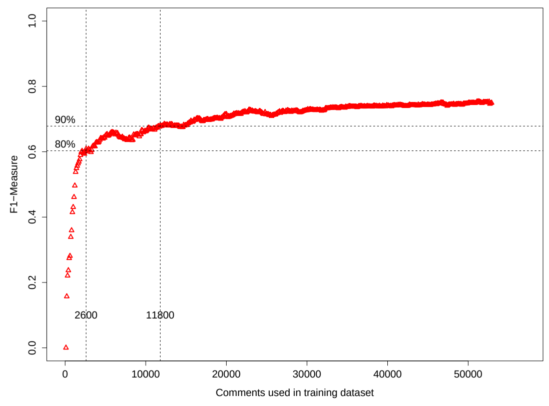
RQ3. How much training data is required to effectively detect self-admitted technical debt?

Motivation: Thus far, we have shown that our NLP-based approach can effectively identify comments expressing self-admitted technical debt. However, we conjecture that the performance of the classification depends on the amount of training data. At the same time, creating the training dataset is a time consuming and labor intensive task. So, the question that arises is: how much training data do we need to effectively classify the source code comments? If we need a very large number of comments to create our training dataset, our approach will be more difficult to extend and apply for other projects. On the other hand, if a small dataset can be used to reliably identify comments with self-admitted technical debt, then this approach can be applied with minimal effort, i.e., less training data. That said, intuitively we expect that the performance of the maximum entropy classifier will improve as more comments are being added to the training dataset.

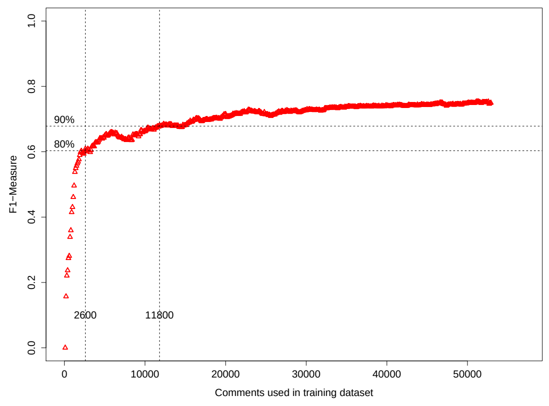
Approach: To answer this research question, we follow a systematic process where we incrementally add training data and evaluate the performance of the classification. More specifically, we combine the comments from all projects into a single large dataset. Then, we split this dataset into ten equally-sized folds, making sure that each partition has the same ratio of comments of self-admitted technical debt and without technical debt as the original dataset. Next, we use one of the ten folds for testing and the remaining nine folds as training data. Since we want to examine the impact of the quantity of training data on performance, we train the classifier with batches of 100 comments at a time and test its performance on the testing data. It is important to note that even within the batches of 100 comments, we maintain the same ratio of self-admitted technical debt and none technical debt comments as in the original dataset. We keep adding comments until all of the training dataset is used. We repeat this process for each one of the ten folds and report the average performance across all folds.

We compute the F1-measure values after each iteration (i.e., the addition of a batch of 100 comments) and record the iteration that achieves the highest F1-measure. Then we find the iterations in which at least 80% and 90% of the maximum F1-measure value is achieved, and report the number of comments added up to those iterations.

Results - design debt: Figure 6(a) shows the average F1-measure values obtained when detecting design self-admitted technical debt, while adding batches of 100 comments. We find that the F1-measure score improves as we increase the number of comments in the training dataset, and the highest value (i.e., 0.824) is achieved with



(a) Design Debt



(b) Requirement Debt

Figure 6: F1-measure Achieved by Incrementally Adding Batches of 100 Comments in the Training Dataset.

42,700 comments. However, the steepest improvement in the F1-measure performance takes place within the first 2K-4K comments. Additionally, 80% and 90% of the maximum F1-measure value is achieved with 3,900 and 9,700 comments in the training dataset, respectively. Since each batch of comments consists of approximately 5% (i.e., $\frac{2{,}703}{58{,}122}$) comments with design self-admitted technical debt, the iteration achieving 80% of the maximum F1-measure value contains 195 comments with design self-admitted technical debt, while the iteration achieving 90% of the maximum F1-measure value contains 485 such comments. In conclusion, to achieve 80% of the maximum F1-measure value, we need only 9.1% (i.e., $\frac{3{,}900}{42{,}700}$) of the training data, while to achieve 90% of the maximum F1-measure value, we need only 22.7% (i.e., $\frac{9{,}700}{42{,}700}$) of the training data.

Results - requirement debt: Figure 6(b) shows the average F1-measure values obtained when detecting requirement self-admitted technical debt, while adding batches of 100 comments. As expected, the F1-measure increases as we add more comments into the training dataset, and again the steepest improvement takes place within the first 2-3K comments.

The highest F1-measure value (i.e., 0.753) is achieved using 51,300 comments of which 675 are requirement self-admitted technical debt. Additionally, 80% of the maximum F1-measure score is achieved with 2,600 comments, while 90% of the maximum F1-measure score with 11,800 comments in the training dataset.

Each batch contains two comments with requirement self-admitted technical debt, since the percentage of such comments is 1.3% (i.e., $\frac{757}{58{,}122}$) in the entire dataset.As a result, the iteration achieving 80% of the maximum F1-measure value contains 52 comments with requirement self-admitted technical debt, while the iteration achieving 90% of the maximum F1-measure value contains 236 such comments.

In conclusion, to achieve 80% of the maximum F1-measure value, we need only 5% (i.e., $\frac{2{,}600}{51{,}300}$) of the training data, while to achieve 90% of the maximum F1-measure value, we need only 23% (i.e., $\frac{11{,}800}{51{,}300}$) of the training data.

We find that to achieve a performance equivalent to 90% of the maximum F1-measure score, only 23% of the comments are required for both design and requirement self-admitted technical debt. For a performance equivalent to 80% of the maximum F1-measure score, only 9% and 5% of the comments are required for design and requirement self-admitted technical debt, respectively.

4.4 Discussion

Thus far, we have seen that our NLP-based approach can perform well in classifying self-admitted technical debt. However, there are some observations that warrant further investigation. For example, when it comes to the different types of self-admitted technical debt, we find that requirement debt tends to require less training data, which is another interesting point that is worth further investigation (Section 4.4.1).

Moreover, we think that is also interesting to know the performance of our approach when trained to distinguish between self-admitted technical debt and non-self-admitted technical debt, i.e., without using fine-grained classes of debt, such as design and requirement debt (Section 4.4.2).

Also, when performing our classification, there are several different classifiers that can be used in the Stanford Classifier toolkit, hence we investigate what is the impact of using different classifiers on the accuracy (Section 4.4.3).

Lastly, we analyze the overlap between the files that contain self-admitted technical debt and the files that contain code smells. This is an interesting point of discussion to provide insights on how technical debt found in comments relates to code smells found by static analysis tools (Section 4.4.4).

4.4.1 Textual Similarity for Design and Requirement Debt

For RQ3, we hypothesize that one of the reasons that the detection of requirement self-admitted technical debt comments needs less training data is because such comments are more similar to each other compared to design self-admitted technical debt comments. Therefore, we compare the intra-similarity of the requirement and design debt comments.

We start by calculating the term frequency-inverse document frequency (*tf-idf*) weight of each design and requirement self-admitted technical debt comment. Term frequency (*tf*) is the simple count of occurrences that a term (i.e., word) has in a document (i.e., comment). Inverse document frequency (*idf*) takes into account the number of documents that the term appears. However, as the name implies, the more one term is repeated across multiple documents the less relevant it is. Therefore, let N be the total number of documents in a collection, the *idf* of a term t is defined as follows: $idf_t = log\frac{N}{df_t}$. The total *tf-idf* weight of a document is equal to the sum of each individual term *tf-idf* weight in the document. Each document is represented by a *document vector* in a *vector space model*.

Once we have the *tf-idf* weights for the comments, we calculate the *cosine similarity* between the comments. The Cosine similarity can be viewed as the *dot product* of the normalized versions of two document vectors (i.e., two comments) [MRS08]. The value of the cosine distance ranges between 0 to 1, where 0 means that the comments are not similar at all and 1 means that the comments are identical.

For example, the requirement self-admitted technical debt dataset contains 757 comments, for which we generate a 757×757 matrix (since we compare each comment to all other comments). Finally, we take the average cosine similarity for design and requirement debt comments, respectively, and plot their distributions. Figure 7 shows that the median and the upper quartile for requirement self-admitted technical debt comments are higher than the median and upper quartile for design self-admitted technical debt. The median for requirement debt comments is 0.018, whereas, the median for design debt comments is 0.011. To ensure that the difference is statistically significant, we perform the Wilcoxon test to calculate the p-value. The calculated p-value is less than 2.2e-16 showing that the result is indeed statistically significant (i.e., $p < 0.001$). Considering our findings, our hypothesis is validated, showing that requirement self-admitted technical debt comments are more similar to each other compared to design self-admitted technical debt comments. This may help explain why requirement debt needs a smaller set of positive weight textual features to be detected.

4.4.2 Distinguishing Self-Admitted Technical Debt from Non-Self-Admitted Technical Debt Comments

So far, we analyzed the performance of our NLP-based approach to identify distinct types of self-admitted technical debt (i.e., design and requirement debt). However, a simpler distinction between self-admitted technical debt and non-debt comments can

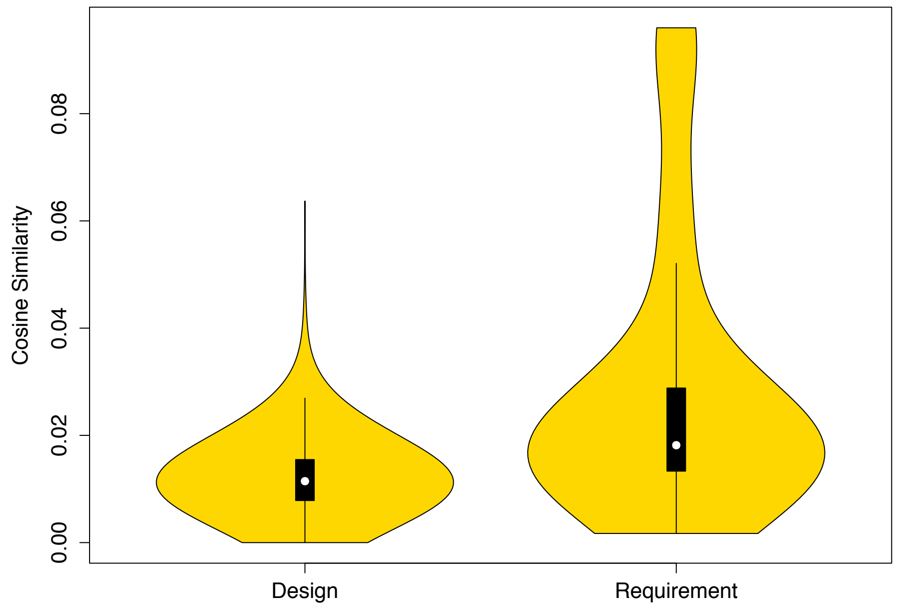


Figure 7: Textual Similarity Between Design and Requirement Debt Comments

also be interesting in the case those fine-grained classes of debt are not considered necessary by a user of the proposed NLP-based detection approach. Another reason justifying such a coarse-grained distinction is that the cost of building a training dataset with fine-grained classes of debt is more expensive, mentally challenging, and subjective than building a training dataset with just two classes (i.e., comments with and without technical debt).

In order to compute the performance of our NLP-based approach using only two classes (i.e., comments with and without technical debt), we repeat RQ1 and RQ2 with modified training and test datasets. First, we take all design and requirement self-admitted technical debt comments and label them with a common class i.e., technical debt, and the remaining comments we kept them labeled as without technical debt. Second, we run the maximum entropy classifier in the same leave-one-out cross-project validation fashion, using the comments of 9 projects to train the classifier and

Table 10: F1-measure Performance Considering Different Types of Self-admitted Technical Debt

Project	**Design Debt**	**Requirement Debt**	**Technical Debt**
Ant	0.517	0.154	0.512
ArgoUML	0.814	0.595	0.819
Columba	0.601	0.804	0.750
EMF	0.470	0.381	0.462
Hibernate	0.744	0.476	0.763
JEdit	0.509	0.091	0.461
JFreeChart	0.492	0.321	0.513
JMeter	0.731	0.237	0.715
JRuby	0.783	0.435	0.773
SQuirrel	0.540	0.541	0.593
Average	0.620	0.403	0.636

the comments from the remaining project to test the classifier. We repeat this process for each of the ten projects and compute the average F1-measure. Lastly, we analyze the textual features used to identify the self-admitted technical debt comments.

Table 10 compares the F1-measure achieved when detecting design debt, requirement debt, separately and when detecting both combined in a single class. As we can see, the performance when detecting technical debt is very similar with the performance of the classifier when detecting design debt. This is expected, as the majority of technical debt comments in the training dataset are labeled with the design debt class. Nevertheless, the performance achieved when detecting design debt was surpassed in the projects where the classifier performed well in detecting requirement debt, for example, in Columba (0.601 vs. 0.750) and SQuirrel SQL (0.540 vs. 0.593).

We find that the average performance when detecting design and requirement self-admitted technical debt combined is better (0.636) than the performance achieved when detecting them individually (0.620 and 0.403 for design and requirement debt, respectively).

Table 11: Top-10 Textual Features Used to Identify Different Types of Self-Admitted Technical Debt

Project	**Design debt**	**Requirement debt**	**Technical debt**
1	hack	todo	**hack**
2	workaround	needed	**workaround**
3	yuck!	implementation	yuck!
4	kludge	fixme	kludge
5	stupidity	xxx	stupidity
6	needed?	ends?	needed?
7	columns?	convention	unused?
8	unused?	configurable	**fixme**
9	wtf?	apparently	**todo**
10	todo	fudging	wtf?

Table 11 shows a comparison of the top-10 textual features used to detect design and requirement debt comments separately, and those used to detect both types of debt combined in a single class. When analyzing the top-10 textual features used to classify self-admitted technical debt, we find once more, a strong overlap with the top-10 textual features used to classify design debt. The weight of the features is attributed in accordance to the frequency that each word is found in the training dataset, and therefore, the top-10 features tend to be similar with the top-10 design debt features, since design debt comments represent the majority of self-admitted technical debt comments in the dataset.

4.4.3 Investigating the Impact of Different Classifiers on the Accuracy of the Classification

In our work, the classification performed by the Stanford Classifier used the maximum entropy classifier. However, the Stanford Classifier can use other classifiers too. In order to examine the impact of the underlying classifier on the accuracy of the proposed approach, we investigate two more classifiers, namely the Naive Bayes, and the Binary classifiers.

Figures 8(a) and 8(b) compare the performance between the three different classifiers. We find that the Naive Bayes has the worst average F1-measure of 0.30 and 0.05 for design and requirement technical debt, respectively. Based on our findings, the Naive Bayes algorithm favours recall at the expense of precision. For example, while classifying design debt, the average recall was 0.84 and precision 0.19. The two other algorithms present more balanced results compared to the Naive Bayes, and the difference in their performance is almost negligible. The Logistic Regression classifier achieved F1-measures of 0.62 and 0.40, while the Binary classifier F1-measures were 0.63 and 0.40, for design and requirement self-admitted technical debt, respectively. Tables 12 and **??** provide detailed data for each classifier and all ten examined projects.

Although the Binary classifier has a slightly better performance, for our purpose, the Logistical Regression classifier provides more insightful textual features. These features were analyzed and presented in RQ2.

According to previous work, developers hate to deal with false positives (i.e., low precision) [BBC$^+$10, EBO$^+$15, SvGJ$^+$15]. Due to this fact, we choose to present our results in this study using the maximum entropy classifier, which has an average precision of 0.716 throughout all projects. However, favouring recall over precision by using the Naives Bayes classifier might still be acceptable, if a manual process to filter out false positives is in place, as reported by Berry et al. [BGST12].

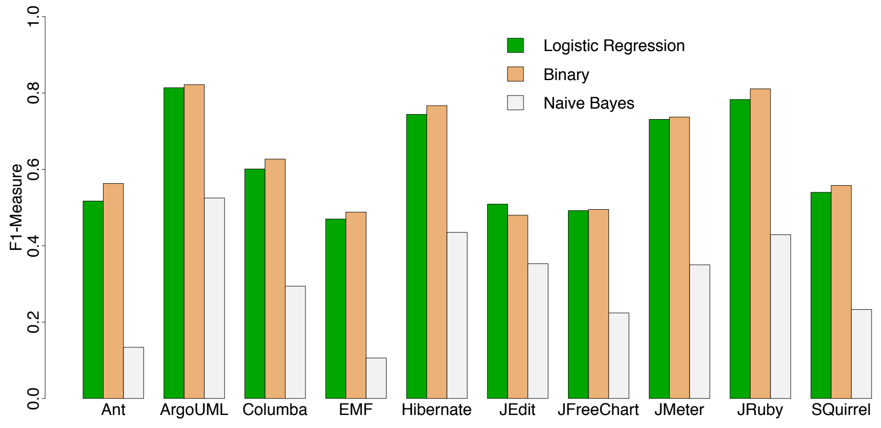
One important question to ask when choosing what kind of classifier to use is how much training data is currently available. In most of the cases, the trickiest part of applying a machine learning classifier in real world applications is creating or obtaining enough training data. If you have fairly little data at your disposal, and you are going to train a supervised classifier, then machine learning theory recommends classifiers with high bias, such as the Naive Bayes [FC04, NJ01]. If there is a reasonable amount of labeled data, then you are in good stand to use most kinds of classifiers [MRS08]. For instance, you may wish to use a Support Vector Machine (SVM), a decision tree or, like in our study, a max entropy classifier. If a large amount of data is available, then the choice of classifier probably has little effect on the results and the best choice may be unclear [BB01]. It may be best to choose a classifier based on the scalability of training, or even runtime efficiency.

Table 12: Comparison Between Different Classifiers for Design Debt

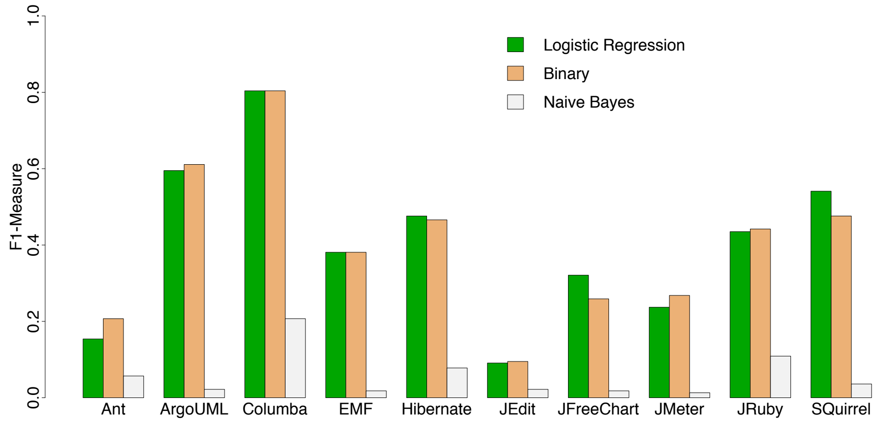
Project	**Maximum Entropy**			**Naive Bayes**			**Binary**		
	Precision	**Recall**	**F1 measure**	**Precision**	**Recall**	**F1 measure**	**Precision**	**Recall**	**F1 measure**
Ant	0.554	0.484	0.517	0.072	0.874	0.134	0.620	0.516	0.563
ArgoUML	0.788	0.843	0.814	0.358	0.985	0.525	0.790	0.858	0.822
Columba	0.792	0.484	0.601	0.181	0.786	0.294	0.840	0.500	0.627
EMF	0.574	0.397	0.470	0.057	0.872	0.106	0.633	0.397	0.488
Hibernate	0.877	0.645	0.744	0.288	0.890	0.435	0.895	0.670	0.767
JEdit	0.779	0.378	0.509	0.227	0.791	0.353	0.807	0.342	0.480
JFreeChart	0.646	0.397	0.492	0.140	0.560	0.224	0.658	0.397	0.495
JMeter	0.808	0.668	0.731	0.224	0.801	0.350	0.819	0.671	0.737
JRuby	0.798	0.770	0.783	0.275	0.971	0.429	0.815	0.808	0.811
SQuirrel	0.544	0.536	0.540	0.133	0.947	0.233	0.567	0.550	0.558
Average	0.716	0.5602	0.6201	0.1955	0.8477	0.3083	0.7444	0.5709	0.6348

Table 13: Comparison Between Different Classifiers for Requirement Debt

Project	Maximum Entropy			Naive Bayes			Binary		
	Precision	Recall	F1 measure	Precision	Recall	F1 measure	Precision	Recall	F1 measure
Ant	0.154	0.154	0.154	0.007	0.769	0.013	0.188	0.231	0.207
ArgoUML	0.663	0.540	0.595	0.119	0.808	0.207	0.659	0.569	0.611
Columba	0.755	0.860	0.804	0.030	0.930	0.057	0.755	0.860	0.804
EMF	0.800	0.250	0.381	0.009	1.000	0.018	0.800	0.250	0.381
Hibernate	0.610	0.391	0.476	0.041	0.781	0.078	0.615	0.375	0.466
JEdit	0.125	0.071	0.091	0.011	0.857	0.022	0.143	0.071	0.095
JFreeChart	0.220	0.600	0.321	0.009	0.800	0.018	0.179	0.467	0.259
JMeter	0.153	0.524	0.237	0.011	0.952	0.022	0.180	0.524	0.268
JRuby	0.686	0.318	0.435	0.058	0.836	0.109	0.679	0.327	0.442
SQuirrel	0.657	0.460	0.541	0.018	0.900	0.036	0.455	0.500	0.476
Average	0.4823	0.4168	0.4035	0.0313	0.8633	0.058	0.4653	0.4174	0.4009



(a) Design Debt



(b) Requirement Debt

Figure 8: Underlying Classifier Algorithms Performance Comparison

4.4.4 Investigating the Overlap Between Technical Debt Found in Comments and Technical Debt Found by Static Analysis Tools

Thus far, we analyzed technical debt that was expressed by developers through source code comments. However, there are other ways to identify technical debt, such as architectural reviews, documentation analysis, and static analysis tools. To date, using static analysis tools is one of the most popular approaches to identify technical debt in the source code [FFS12]. In general, static analysis tools parse the source code of a project to calculate metrics and identify possible object oriented design violations, also known as code smells, anti-patterns, or design technical debt, based on some fixed metric threshold values.

We analyze the overlap between what our NLP-based approach identifies as technical debt and what a static analysis tool identifies as technical debt. We selected JDeodorant as the static analysis tool, since it supports the detection of three popular code smells, namely Long Method, God Class, and Feature Envy. We avoided the use of metric-based code smell detection tools, because they tend to have high false positive rates and flag a large portion of the code base as problematic [FDW$^+$16]. On the other hand, JDeodorant detects only actionable code smells (i.e., code smells for which a behavior-preserving refactoring can be applied to resolve them), and does not rely on any metric thresholds, but rather applies static source code analysis to detect structural anomalies and suggest refactoring opportunities to eliminate them [TCC08].

First, we analyzed our 10 open source projects using JDeodorant. The result of this analysis is a list of Java files that were identified having at least one instance of the Long Method, God Class, and Feature Envy code smells. These code smells have been extensively investigated in the literature, and are considered to occur frequently [OCS10, SYA$^+$13]. Second, we created a similar list containing the files that were identified with self-admitted technical debt comments. Finally, we examined the overlap of the two lists of files. It should be emphasized that we did not examine if the self-admitted technical debt comments actually discuss the detected code smells, but only if there is a co-occurrence at file-level.

Table 14 provides details about each one of the projects used in our study. The columns of Table 14 present the total number of files with self-admitted technical debt, followed by the number of files containing self-admitted technical debt comments and at least one code smell instance, along with the percentage over the total number of files with self-admitted technical debt, for Long Method, Feature Envy, God Class, and all code smells combined, respectively.

JMeter, for example, has 200 files that contain self-admitted technical debt comments, and 143 of these files also contain at least one Long Method code smell (i.e., 71.5%). In addition, we can see that 20.5% of the files that have self-admitted technical debt are involved in Feature Envy code smells, and 48.5% of them are involved in God Class code smells. In summary, we see that 80.5% of the files that contain self-admitted technical debt comments are also involved in at least one of the three examined code smells.

We find that the code smell that overlaps the most with self-admitted technical debt is Long Method. Intuitively, this is expected, since Long Method is a common code smell and may have multiple instances per file, because it is computed at the method level. The overlap between files with self-admitted technical debt and Long Method ranged from 43.6% to 82% of all the files containing self-admitted technical debt comments, and considering all projects, the average overlap is 65%. In addition, 44.2% of the files with self-admitted technical debt comments are also involved in God Class code smells, and 20.7% in Feature Envy code smells. Taking all examined code smells in consideration we find that, on average, 69.7% of files containing self-admitted technical debt are also involved in at least one of the three examined code smells.

Our findings here shows that using code comments to identify technical debt is a complementary approach to using code smells to detect technical debt. Clearly, there is overlap, however, each approach also identifies unique instances of technical debt.

Table 14: Overlap Between the Files Containing self-admitted technical debt and the Files Containing Code Smells as Detected by JDeodorant

Project	# of Files with SATD	# of SATD Files with Long Method	% of SATD Files with Long Method	# of SATD Files with Feature Envy	% of SATD fFles with Feature Envy	# of SATD Files with God Class	% of SATD Files with God Class	# of SATD Files with Any Code Smell	% of SATD Files with Any Code Smell
Ant	73	57	78.0	19	26.0	42	57.5	63	86.3
ArgoUML	419	255	60.8	43	10.2	128	30.5	283	67.5
Columba	117	76	64.9	18	15.3	47	40.1	89	76.0
EMF	53	33	62.2	14	26.4	28	52.8	28	52.8
Hibernate	206	90	43.6	44	21.3	72	34.9	116	56.3
JEdit	108	74	68.5	23	21.2	47	43.5	82	75.9
JFreeChart	106	87	82.0	20	18.8	52	49.0	92	86.7
JMeter	200	143	71.5	41	20.5	97	48.5	161	80.5
JRuby	163	107	65.5	43	26.3	79	48.4	85	52.1
SQuirrel	156	82	52.5	32	20.5	58	37.1	99	63.4
Average			65.0		20.7		44.2		69.7

4.5 Related Work

Our work uses code comments to detect self-admitted technical debt using a Natural Language Processing (NLP) technique. Therefore, we divide the related work into three subsections, namely source code comments, technical debt, and NLP in software engineering.

4.5.1 Source Code Comments

A number of studies examined the co-evolution of source code comments and the rationale for changing code comments. For example, Fluri *et al.* [FWG07] analyzed the co-evolution of source code and code comments, and found that 97% of the comment changes are consistent. Tan *et al.* [TMTL12] proposed a novel approach to identify inconsistencies between Javadoc comments and method signatures. Malik *et al.* [MCHM$^+$08] studied the likelihood of a comment to be updated and found that call dependencies, control statements, the age of the function containing the comment, and the number of co-changed dependent functions are the most important factors to predict comment updates.

Other works used code comments to understand developer tasks. For example. Storey *et al.* [SRB$^+$08] analyzed how task annotations (e.g., TODO, FIXME) play a role in improving team articulation and communication. The work closest to ours is the work by Potdar and Shihab [PS14], where code comments were used to identify technical debt, called self-admitted technical debt.

Similar to some of the prior work, we also use source code comments to identify technical debt. However, our main focus is on the detection of different *types* of self-admitted technical debt. As we have shown, our approach yields different and better results in the detection of self-admitted technical debt.

4.5.2 Technical Debt

A number of studies has focused on the detection and management of technical debt. For example, Seaman *et al.* [SG11], Kruchten *et al.* [KNOF13] and Brown *et al.* [BCG$^+$10] make several reflections about the term technical debt and how it has been used to communicate the issues that developers find in the code in a way that managers can understand.

Other work focused on the detection of technical debt. Zazworka *et al.* [ZSV$^+$13] conducted an experiment to compare the efficiency of automated tools in comparison with human elicitation regarding the detection of technical debt. They found that there is a small overlap between the two approaches, and thus it is better to combine them than replace one with the other. In addition, they concluded that automated tools are more efficient in finding defect debt, whereas developers can realize more abstract categories of technical debt.

In a follow up work, Zazworka *et al.* [ZSSS11] conducted a study to measure the impact of technical debt on software quality. They focused on a particular kind of design debt, namely, God Classes. They found that God Classes are more likely to change, and therefore, have a higher impact on software quality. Fontana *et al.* [FFS12] investigated design technical debt appearing in the form of code smells. They used metrics to find three different code smells, namely God Classes, Data Classes and Duplicated Code. They proposed an approach to classify which one of the different code smells should be addressed first, based on its risk. Ernst *et al.* [EBO$^+$15] conducted a survey with 1,831 participants and found that architectural decisions are the most important source of technical debt.

Our work is different from the work that uses code smells to detect design technical debt, since we use code comments to detect technical debt. Moreover, our approach does not rely on code metrics and thresholds to identify technical debt and can be used to identify bad quality code symptoms other than bad smells.

More recently, Potdar and Shihab [PS14] extracted the comments of four projects and analyzed 101,762 comments to come up with 62 patterns that indicate self-admitted technical debt. Their findings show that 2.4% - 31% of the files in a project contain self-admitted technical debt. Bavota and Russo [BR16] replicated the study of self-admitted technical debt on a large set of Apache projects and confirmed the findings observed by Potdar and Shihab in their earlier work. Wehaibi *et al.* [WSG16] examined the impact of self-admitted technical debt and found that self-admitted technical debt leads to more complex changes in the future. All three of the aforementioned studies used the comment patterns approach to detect self-admitted technical debt. Our earlier work [MS15] examined more than 33 thousands comments to classify the different types of self-admitted technical debt found in source code comments. Farias *et al.* [FNSS15] proposed a contextualized vocabulary model for identifying technical debt in comments using word classes and code tags in the process.

Our work also uses code comments to detect design technical debt. However, we use these code comments to train a maximum entropy classifier to automatically identify technical debt. Also, our focus is on *self-admitted* design and requirement technical debt.

4.5.3 NLP in Software Engineering

A number of studies leveraged NLP in software engineering, mainly for the traceability of requirements, program comprehension and software maintenance. For example, Lormans and van Deursen [LVD06] used latent semantic indexing (LSI) to create traceable links between requirements and test cases and requirements to design implementations. Hayes *et al.* [HDS05, HDS06] created a tool called RETRO that applies information retrieval techniques to trace and map requirements to designs. Yadla *et al.* [YHD05] further enhanced the RETRO tool and linked requirements to issue reports. On the other hand, Runeson *et al.* [RAN07] implemented a NLP-based tool to automatically identify duplicated issue reports, they found that 2/3 of the possible duplicates examined in their study can be found with their tool. Canfora and Cerulo [CC05] linked a change request with the corresponding set of source files using NLP techniques, and then, they evaluated the performance of the approach on four open source projects.

The prior work motivated us to use NLP techniques. However, our work is different from the aforementioned ones, since we apply NLP techniques on code comments to identify self-admitted technical debt, rather than use it for traceability and linking between different software artifacts.

4.6 Threats to Validity

Construct validity and reliability considers the relationship between theory and observation, in case the measured variables do not measure the actual factors. The training dataset used by us heavily relied on a manual analysis and classification of the code comments from the studied projects. Like any human activity, our manual classification is subject to personal bias. To reduce this bias, we took a statistically significant sample of our classified comments and asked a Master's student, who is not an author of the paper, to manually classify them. Then, we calculate the Kappa's level of agreement between the two classifications. The level of agreement obtained was +0.81, which according to Fleiss [Fle81] is characterized as an excellent inter-rater agreement (values larger than +0.75 are considered excellent). Nevertheless, due to the irregular data distribution of our significant sample (which has many more comments without technical debt, than comments with the other classes of debt), we also measured Kappa's level of agreement for design and requirement self-admitted technical debt separately. The level of agreement obtained for design and requirement self-admitted technical debt was +0.75 and +0.84, respectively.

When performing our study, we used well-commented Java projects. Since our approach heavily depends on code comments, our results and performance measures may be impacted by the quantity and quality of comments in a software project.

Considering the intentional misrepresentation of measures, it is possible that even a well commented project does not contain self-admitted technical debt. Given the fact that the developers may opt to not express themselves in source code comments. In our study, we made sure that we choose case studies that are appropriately commented for our analysis.

On the same point, using comments to determine some self-admitted technical debt may not be fully representative, since comments or code may not be updated consistently. However, previous work shows that changes in the source code are consistent to changes in comments [FWG07, PS14]. In addition, it is possible that a variety of technical debt that is not self-admitted is present in the analyzed projects. However, since the focus of this paper is to improve the detection of the most common types of self-admitted technical debt, considering all technical debt is out of the scope of this paper.

Lastly, our approach depends on the correctness of the underlying tools we use. To mitigate this risk, we used tools that are commonly used by practitioners and by the research community, such as JDeodorant for the extraction of source code comments and for investigating the overlap with code smells (Section 4.4.4) and the Stanford Classifier for training and testing the max entropy classifier used in our approach.

External validity considers the generalization of our findings. All of our findings were derived from comments in open source projects. To minimize the threat to external validity, we chose open source projects from different domains. That said, our results may not generalize to other open source or commercial projects, projects written in different languages, projects from different domains and/or technology stacks. In particular, our results may not generalize to projects that have a low number or no comments or comments that are written in a language other than English.

4.7 Conclusion and Future Work

Technical debt is a term being used to express non-optimal solutions, such as hacks and workarounds, that are applied during the software development process. Although these non-optimal solutions can help achieve immediate pressing goals, most often they will have a negative impact on the project maintainability [ZSSS11].

Our work focuses on the identification of self-admitted technical debt through the use of Natural Language Processing. We analyzed the comments of 10 open source projects namely Ant, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JMeter, JRuby and SQuirrel SQL. These projects are considered well commented and they belong to different application domains. The comments of these projects were manually classified into specific types of technical debt such as design, requirement, defect, documentation and test debt. Next, we selected 61,664 comments from this dataset (i.e., those classified as design self-admitted technical debt, requirement self-admitted technical debt and without technical debt) to train the maximum entropy classifier, and then this classifier was used to identify design and requirement self-admitted technical debt automatically.

We first evaluated the performance of our approach by comparing the F1-measure of our approach with the F1-measure of two other baselines, i.e., the comment patterns baseline and the simple (random) baseline. We have shown that our approach outperforms the comment patterns baseline on average 2.3 and 6 times in the identification of design and requirement self-admitted technical debt, respectively. Moreover, our approach can identify requirement self-admitted technical debt, while the comment patterns baseline fails to detect this kind of debt in most of the examined projects. Furthermore, the performance of our approach surpasses the simple (random) baseline on average 7.6 and 19.1 times for design and requirement self-admitted technical debt, respectively.

Then, we explored the characteristics of the features (i.e., words) used to classify self-admitted technical debt. We find that the words used to express design and requirement self-admitted technical debt are different from each other. The three strongest indicators of design self-admitted technical debt are 'hack', 'workaround' and 'yuck!', whereas, 'todo', 'needed' and 'implementation' are the strongest indicators of requirement debt. In addition, we find that using only 5% and 23% of the comments in the training dataset still leads to an accuracy that is equivalent to 80% and 90% of the best performance, respectively. In fact, our results show that developers use a richer vocabulary to express design self-admitted technical debt and a training dataset of at least 3,900 comments (of which 195 comments are design self-admitted technical debt) is necessary to obtain a satisfactory classification. On the other hand, requirement self-admitted technical debt is expressed in a more uniform way, and with a training dataset of 2,600 comments (of which 52 are self-admitted technical debt) it is possible to classify with relatively high accuracy requirement self-admitted technical debt.

In the future, we believe that more analysis is needed to fine tune the use of the current training dataset in order to achieve maximum efficiency in the detection of self-admitted technical debt comments. For example, using subsets of our training dataset can be more suitable for some applications than using the whole dataset due to domain particularities. However, the results thus far are not to be neglected as our approach has the best F1-measure performance on every analyzed project. In addition, we plan to examine the applicability of our approach to more domains (than those we study in this paper) and software projects developed in different programming languages.

Another interesting research direction that we plan to investigate in the future is the use of other machine learning techniques, such as active learning to reduce the number of labeled data necessary to train the classifier. This technique, if proved successful, can further expand the horizon of projects that our approach can be applied to.

Moreover, to enable future research, we make the dataset created in this study publicly available[3]. We believe that it will be a good starting point for researchers interested in identifying technical debt through comments and even experimenting with different Natural Language Processing techniques. Lastly, we plan to use the findings of this study to build a tool that will support software engineers in the task of identifying and managing self-admitted technical debt.

[3]`https://github.com/maldonado/tse_satd_data`

Chapter 5

Summary, Contributions and Future Work

5.1 Summary of Addressed Topics

The main focus of our thesis is to tackle the challenges of self-admitted technical debt identification. First, we conducted a survey of the state-of-the-art in the identification of technical debt research in order to understand the main challenges. Then, we manually analyzed a number of comments from different projects belonging to different application domains. Next, we propose an approach based on NLP techniques that outperforms the current state of the art in the identification of self-admitted technical debt. The remainder of this chapter details the major topics covered in this thesis.

Chapter 2 surveys the state-of-art in technical debt. We believe that such a review is necessary at this time, since a lot of research is aiming to better understand technical debt. Therefore, it is an ideal time to reflect on the definitions and applications of the metaphor as well to evaluate the current challenges in the field.

Chapter 3 presents the result of the quantification of the different types of self-admitted technical debt. In this chapter we analyzed the comments of 5 open source projects. These projects are considered well commented and they belong to different application domains. We used them to understand the characteristics of self-admitted technical debt types creating a rich dataset with more than 33,093 classified comments. We find that self-admitted technical debt can be classified into five types: design debt, defect debt, documentation debt, requirement debt and test debt. However, the two most prevalent types of self-admitted technical debt are design and requirement self-admitted technical debt. Design debt ranged from 42% to 84% across the projects, whereas, requirement debt ranged from 5% to 45%.

Chapter 4 presents an approach that uses classified design and requirement self-admitted technical debt comments to train a maximum entropy classifier to automatically identify self-admitted technical debt. We evaluated the performance of our approach against two other baselines, i.e., the comment patterns baseline and the simple (random) baseline. We show that our approach performance surpassed the comment patterns baseline on average 2.3 and 6 times in the identification of design and requirement self-admitted technical debt, respectively. Analyzing the features to identify self-admitted technical debt we find that the words used to express design and requirement self-admitted technical debt are different from each other. In addition, we find that using only 5% and 23% of the comments in the training dataset still leads to an accuracy that is equivalent to 80% and 90% of the best performance, respectively.

5.2 Contributions

The goal of this thesis is to propose an approach that can effectively identify self-admitted technical debt comments. We make several contributions towards this goal. These contributions were motivated by previous research and our industrial experience. We summarize the main contributions of the thesis in more detail.

The major contributions of this thesis are as follows:

- **A concise review of the state of the art in technical debt:** We provide the readers a concise background evaluation from the the creation of the metaphor until present date. We choose the most relevant sources that define how the technical debt is being used and also the challenges involving the identification of technical debt.

- **A rich dataset of manually labeled technical debt:** To create such dataset we read and analyzed the source code comments of 10 open source projects considered well commented and from different application domains. The comments of these projects were manually classified into specific types of technical debt such as design, requirement, defect, documentation and test debt. In total, our dataset contain 62,566 labeled comments and we made it publicly available to enable future research on the field.

  In addition, to mitigate the risk of creating a biased dataset, we also asked a student that was not involved with our work to classify a stratified sample. Then, we calculate the Kappa's level of agreement between the two classifications. The level of agreement obtained was +0.81, which according to Fleiss [Fle81] is characterized as an excellent inter-rater agreement.

- **An automatic, NLP-based, approach to identify design and requirement self-admitted technical debt:** We have shown that our approach outperforms the current state-of-the-art on average 2.3 and 6 times in the identification of design and requirement self-admitted technical debt, respectively. Moreover, our approach can identify requirement self-admitted technical debt, while the comment patterns baseline fails to detect this kind of debt in the majority of the examined projects. Furthermore, the performance of our approach outperforms the simple (random) baseline on average 7.6 and 19.1 times for design and requirement self-admitted technical debt, respectively.

- **An empirical study to investigate the amount of training data necessary to effectively identify technical debt:** We discuss the implications of the amount training data that is necessary to apply our approach. For example, if we need a very large number of comments to create our training dataset, our approach will be more difficult to extend and apply for other projects. On the other hand, if a small dataset can be used to reliably identify comments with self-admitted technical debt, then this approach can be applied with minimal effort, i.e., less training data. However, we find that using only 5% and 23% of the comments in the training dataset leads to an accuracy that is equivalent to 80% and 90% of the best performance, respectively. Our results also show that developers use a richer vocabulary to express design self-admitted technical debt and a training dataset of at least 3,900 comments (of which 195 comments are design self-admitted technical debt) is necessary to obtain a satisfactory classification. On the other hand, requirement self-admitted technical debt is expressed in a more uniform way, and with a training dataset of 2,600 comments (of which 52 are self-admitted technical debt) it is possible to classify with relatively high accuracy requirement self-admitted technical debt.

5.3 Future Work

We believe that our thesis makes a positive contribution towards the goal of effectively identifying technical debt. However, there are still many open challenges that need to be tackled in order to improve the identification of technical debt. We now highlight some avenues for future work.

5.3.1 Fine tunning the approach to obtain optimal results

Although we conducted a number of diverse experiments with the NLP classifier, we believe that is still a lot of opportunities to be explored that may improve even further our approach. For example, we noticed that using subsets of our training dataset can be more suitable for some applications than using the whole dataset due to domain particularities.

5.3.2 Expanding the scope of our approach

We plan to examine the applicability of our approach to more domains than those we study in this paper and software projects developed in different programming languages. Also, would be interesting to analyze projects that uses comments in different idioms than English. As we showed, we provide filtering heuristics that could be easily adapted to remove irrelevant comments from software projects and that we need a reduced amount of comments to obtain satisfactory results concerning the identification of technical debt. We believe that these factors will help future work to expand considerably.

5.3.3 Tool support for software developers

Lastly, we plan to use the findings of this study to build a tool that will support software engineers in the task of identifying and managing self-admitted technical debt. We envision that such tool would be useful to monitor debt and focus resources during the development of the software project, moreover, when properly managed software developers could take advantage of incurring debt when necessary, without losing track of the overall quality of the system.

Bibliography

[AL12] Karim Ali and Ondřej Lhoták. Application-only call graph construction. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, pages 688–712, 2012.

[AMdM$^+$16] N.S.R. Alves, T.S. Mendes, M. G. de Mendonca, R.O. Spinola, F. Shull, and C. Seaman. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, 70:100–121, 2016.

[ARC$^+$14] N.S.R. Alves, L.F. Ribeiro, V. Caires, T.S. Mendes, and R.O. Spinola. Towards an ontology of terms on technical debt. In *Proceedings of the 6th International Workshop on Managing Technical Debt*, pages 1–7, 2014.

[BB01] Michele Banko and Eric Brill. *Scaling to Very Very Large Corpora for Natural LanguageDisambiguation.* 2001.

[BBC$^+$10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, pages 66–75, 2010.

[BCG$^+$10] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, pages 47–52, 2010.

[BGST12] Daniel Berry, Ricardo Gacitua, Pete Sawyer, and Sri Fatimah Tjong. *The Case for Dumb Requirements Engineering Tools*. 2012.

[BR16] Gabriele Bavota and Barbara Russo. A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 13th International Workshop on Mining Software Repositories*, pages 315–326, 2016.

[CC05] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *Proceedings of the 11th IEEE International Symposium on Software Metrics*, pages 21–29, 2005.

[Coh60] J. Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20:37–46, 1960.

[Cun92] W. Cunningham. The wycash portfolio management system. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications*, pages 29–30, 1992.

[EBO$^+$15] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton. Measure it? manage it? ignore it? software practitioners and technical debt. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, pages 50–60, 2015.

[FC73] J. L. Fleiss and J. Cohen. The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability. *Educational and Psychological Measurement*, 33:613–619, 1973.

[FC04] George Forman and Ira Cohen. Learning from little: Comparison of classifiers given little training. In *Proc. PKDD*, pages 161–172, 2004.

[FDW$^+$16] Francesca Arcelli Fontana, Jens Dietrich, Bartosz Walter, Aiko Yamashita, and Marco Zanoni. Antipattern and code smell false positives: Preliminary conceptualization and classification. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, pages 609–613, 2016.

[FFS12] F.A. Fontana, V. Ferme, and S. Spinelli. Investigating the impact of code smells debt on quality code evaluation. In *Proceedings of the 3rd International Workshop on Managing Technical Debt*, pages 15–22, 2012.

[FFZY15] F. A. Fontana, V. Ferme, M. Zanoni, and A. Yamashita. Automatic metric thresholds derivation for code smell detection. In *Proceedings of the Sixth International Workshop on Emerging Trends in Software Metrics*, pages 44–53, 2015.

[FKNO14] Davide Falessi, Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. Technical debt at the crossroads of research and practice: report on the fifth international workshop on managing technical debt. *ACM SIGSOFT Software Engineering Notes*, pages 31–33, 2014.

[Fle81] J.L. Fleiss. The measurement of interrater agreement. *Statistical methods for rates and proportions.*, pages 212–236, 1981.

[FMZM15] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, pages 1–49, 2015.

[FNSS15] M. A. d. F. Farias, M. G. d M. Neto, A. B. da Silva, and R. O. Spinola. A contextualized vocabulary model for identifying technical debt on code comments. In *Proceedings of the 7th International Workshop on Managing Technical Debt*, pages 25–32, 2015.

[FWG07] B. Fluri, M. Wursch, and H.C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pages 70–79, 2007.

[Gra10] Jurgen Graf. Speeding up context-, object- and field-sensitive sdg generation. In *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, pages 105–114, 2010.

[HDS05] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram. Improving after-the-fact tracing and mapping: supporting software quality predictions. *IEEE Software*, 22:30–37, 2005.

[HDS06] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram. Advancing candidate link generation for requirements tracing: the study of methods. *IEEE Transactions on Software Engineering*, 32:4–19, 2006.

[KNO12] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *Ieee software*, 29, 2012.

[KNOF13] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi. Technical debt: Towards a crisper definition report on the 4th international workshop on managing technical debt. *SIGSOFT Softw. Eng. Notes*, pages 51–54, 2013.

[LTS12] E. Lim, N. Taksande, and C. Seaman. A balancing act: What software practitioners have to say about technical debt. *IEEE Software*, 29:22–27, 2012.

[LVD06] M. Lormans and A. Van Deursen. Can LSI help reconstructing requirements traceability in design and test? In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, pages 47–56, 2006.

[Mar] Martin fowler: Technical debt quadrant. `http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html`. Accessed: 2016-08-01.

[Mar04] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 350–359, 2004.

[Mar12] R. Marinescu. Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development*, 56:1–13, 2012.

[McC] Steve mcconnell: Technical debt. `http://www.construx.com/10x_Software_Development/Technical_Debt/`. Accessed: 2016-08-01.

[MCHM+08] H. Malik, I. Chowdhury, T. Hsiao-Ming, Z. M. Jiang, and A.E. Hassan. Understanding the rationale for updating a function comment. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 167–176, 2008.

[MGV10] R. Marinescu, G. Ganea, and I. Verebi. Incode: Continuous quality assessment and improvement. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, pages 274–275, 2010.

[MK03] Christopher Manning and Dan Klein. Optimization, maxent models, and conditional estimation without magic. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for*

Computational Linguistics on Human Language Technology: Tutorials-Volume 5, pages 8–8, 2003.

[MRS08] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.

[MS15] E. d. S. Maldonado and E. Shihab. Detecting and quantifying different types of self-admitted technical debt. In *Proceedings of the 7th International Workshop on Managing Technical Debt*, pages 9–15, 2015.

[NJ01] Andrew Y. Ng and Michael I. Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. In *Proc. NIPS*, pages 841–848, 2001.

[NOKGR12] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas. In search of a metric for managing architectural technical debt. In *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, pages 91–100, 2012.

[OCS10] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjoberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10, 2010.

[Ope] OpenHub homepage. `https://www.openhub.net/`. Accessed: 2014-12-12.

[OVPL14] P. Oliveira, M.T. Valente, and F. Paim Lima. Extracting relative thresholds for source code metrics. In *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering*, pages 254–263, 2014.

[PS14] A. Potdar and E. Shihab. An exploratory study on self-admitted technical debt. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pages 91–100, 2014.

[RAN07] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th International Conference on Software Engineering*, pages 499–510, 2007.

[SG11] C. Seaman and Y. Guo. Chapter 2 - measuring and monitoring technical debt. volume 82 of *Advances in Computers*, pages 25–46. Elsevier, 2011.

[SRB$^+$08] M. Storey, J. Ryall, R.I. Bull, D. Myers, and J. Singer. Todo or to bug. In *Proceedings of the 30th International Conference on Software Engineering*, pages 251–260, 2008.

[Ste10] Chris Sterling. *Managing Software Debt: Building for Inevitable Change (Adobe Reader)*. 2010.

[SvGJ$^+$15] C. Sadowski, J. v. Gogh, C. Jaspan, E. Soderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 598–608, 2015.

[SYA$^+$13] Dag I.K. Sjoberg, Aiko Yamashita, Bente C.D. Anda, Audris Mockus, and Tore Dyba. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39:1144–1156, 2013.

[SZV$^+$13] R.O. Spinola, N. Zazworka, A. Vetro, C. Seaman, and F. Shull. Investigating technical debt folklore: Shedding some light on technical debt opinion. In *Managing Technical Debt (MTD), 2013 4th International Workshop on*, pages 1–7, 2013.

[TC11] N. Tsantalis and A. Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84:1757–1782, 2011.

[TCC08] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, pages 329–331, 2008.

[TMK15] N. Tsantalis, D. Mazinanian, and G. P. Krishnan. Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering*, pages 1055–1090, 2015.

[TMTL12] S. H. Tan, D. Marinov, L. Tan, and G.T. Leavens. @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *Proceedings of the IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 260–269, 2012.

[Whe04] D. A. Wheeler. *SLOC count users guide*, 2004.

[WSG16] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. Examining the impact of self-admitted technical debt on software quality. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 179–188, 2016.

[XCK$^+$16] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng. Identifying and quantifying architectural debt. In *Proceedings of the 38th International Conference on Software Engineering*, pages 488–498, 2016.

[YHD05] S. Yadla, J. H. Hayes, and A. Dekhtyar. Tracing requirements to defect reports: an application of information retrieval techniques. *Innovations in Systems and Software Engineering*, 1:116–124, 2005.

[ZSSS11] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman. Investigating the impact of design debt on software quality. In *Proceedings of the 2nd International Workshop on Managing Technical Debt*, pages 17–23, 2011.

[ZSV$^+$13] N. Zazworka, R. O. Spinola, A. Vetro, F. Shull, and C. Seaman. A case study on effectively identifying technical debt. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, pages 42–47, 2013.