THESIS TITLE HERE

Sultan Wehaibi

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

February 2017

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Sultan Wehaibi**

Entitled: **Thesis Title Here**

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:

Dr. TBA _____ Chair

Dr. TBA _____ Examiner

Dr. TBA _____ Examiner

Dr. Emad Shihab _____ Supervisor

Approved _____

Chair of Department or Graduate Program Director

_____ 20 _____ _____

Dean

Faculty of Engineering and Computer Science

Abstract

Thesis Title Here

Sultan Wehaibi

Sometimes developers contributing to a software project settle for a non-optimal solution under pressure to meet deadlines and quotas despite the potential pitfalls that might ensue at later stages in development, on account of which this phenomenon has been called "technical debt." And like its financial analogue, if not carefully monitored and mediated, technical debt can compromise the very project it was intended to expedite. Several approaches have been proposed to aid developers in tracking the technical debt they incur, and one pioneered in a recent study leverages source code comments to detect (self-admitted) technical debt. Therefore, in this thesis we use empirical studies to examine the impact of self-admitted technical debt and code smells (god classes) on software quality.

First, we examine the impact of self-admitted technical debt on software quality for five open-source projects. To measure this, we take into account three criteria commonly associated with quality: (i) on the file level, the relationship between defects and self-admitted technical debt; (ii) on the change level, the potential of self-admitted technical debt to introduce future defects and (iii) the complexity SATD changes impose on the system. In short, the results of our inquiry indicate that (i) self-admitted technical debt and defects exhibit no correlation at the file level, though (ii) SATD changes make the system less susceptible to future defects than non-SATD changes do and (iii) SATD changes are more difficult to execute.

Second, we expand the scope of our investigation to include another aspect of technical debt—code smells, i.e. god classes. Compiling 40 open-source projects, we assess

the impact of two variables—namely, god classes and self-admitted technical debt—on software quality by determining: (i) whether variable-positive files have more defects than variable-negative files, (ii) whether variable-positive changes induce future defects at a higher rate than variable-negative changes, (iii) whether variable-positive changes are more difficult to perform than variable-negative changes and (iv) how much the metric- and comment-based approaches to technical debt file identification overlap.

Our observations are consistent with the following generalizations: (i) neither god nor SATD files are correlated with defects, (ii) introduction of future defects is higher for god and SATD changes, (iii) god and SATD changes are more difficult to effect and (iv) the metric-comment technical debt file overlap ranges from 11% to 34%.

Acknowledgments

I would like

Dedication

always think critically and be skeptical of what you hear.

Related Publications

The following publications are related to this thesis:

1. **Sultan Wehaibi**, Emad Shihab and Latifa Guerrouj. Examining the Impact of Self-admitted Technical Debt on Software Quality. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER16)*, 10 pages, 2015. [Chapter 3]

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software companies and organizations have a common goal when developing software projects—to deliver high-quality, useful software in a timely manner. However, in most practical settings developers and development companies are saddled with deadlines, urging them to release earlier than the ideal date in terms of product quality. Such situations are all too common and in many cases force developers to take shortcuts [21] [40]. Recently, the term *technical debt* was coined to represent the phenomenon of "doing something that is beneficial in the short term but will incur a cost later on" [5]. Prior work has shown that there are many different reasons why practitioners assume technical debt. These reasons include: a rush to deliver a software product given a tight schedule, deadlines to incorporate with a partner product before release, time-to-market pressure, as well as incentives to satisfy customer demands in a time-sensitive industry [24].

More recently, a study by Potdar and Shihab [38] introduced a new way to identify technical debt through source code comments, referred to as self-admitted technical debt (SATD). SATD is technical debt that developers themselves report through source code comments. Prior work [25] has demonstrated that accrual of SATD is commonplace in software projects, where its implementation can identify different

types of technical debt (e.g., design, defect, and requirement debt).

Intuition and general belief concur that such rushed development tasks (also known as technical debt) negatively impact software maintenance and overall quality [50, 43, 14, 40, 21]. However, to the best of our knowledge, there is no empirical study that examines the impact of SATD on software quality. Such a study is critical since it will help us either confirm or refute entrenched preconceptions regarding the technique and better understand how to manage SATD.

Therefore, in chapter 3 of this thesis, we empirically investigate the relationship between SATD and software quality in five open-source projects. In particular, we examine whether (i) files with SATD have more defects compared to files without SATD, (ii) whether SATD changes introduce more future defects and (iii) whether SATD-related changes tend to be more difficult. We measured the difficulty of a change in terms of the amount of churn, the number of files it touches, the number of modified modules and its entropy.

Having patronized the comment-based approach to identify technical debt, we then engage the metric-based approach and thus broaden our inquiry to incorporate the effect of code smells (god classes) on software quality, relative to the comment-based approach. We compare (i) the defects of god and SATD files versus non-god and non-SATD files, (ii) the future defect introduction of god and SATD changes versus non-god and non-SATD changes and (iii) the difficulty of god and SATD changes versus non-god and non-SATD changes, and, in addition, we measure (iv) the overlap between metric- and comment-based technical debt files.

1.1 Research Hypothesis

Prior research has led us to the formation of our research hypothesis. We believe that:

The impact of self-admitted technical debt on software quality [Sultan: ?]

1.2 Thesis Overview

Chapter 2: Literature Review: The technical debt metaphor has come to encompass miscellaneous "quick fixes" and non-optimal solutions since its inception. This chapter synthesizes more detailed discussions of the technical debt metaphor from websites, blogs and research papers to provide a brief chronological survey of the most prevalent of its various applications, including some of the most recent. At the end of this chapter, we offer a critical assessment of the current status of technical debt in the field, as well as its reputation among software developers and the drawbacks it potentially entails.

Chapter 3: Examining the Impact of Self-Admitted Technical Debt on Software Quality: We empirically examine how self-admitted technical debt impacts software quality across five open-source projects (Chromium, Cassandra, Spark, Tomcat and Hadoop) on three accounts: (i) which of SATD and non-SATD files have more existing defects, (ii) which of SATD and non-SATD changes induce more future defects and (iii) which of SATD and non-SATD changes are more difficult to execute. We adhere to precedent in measuring change difficulty using amount of churn, number of files, number of modified modules and change entropy. Our findings demonstrate (i) no clear trend relating self-admitted technical debt and existing defects, (ii) a higher incidence of future defects for non-SATD changes and (iii) greater difficulty in

performing SATD changes. Therefore, self-admitted technical debt adversely affects system maintenance by increasing change complexity but is dissociated from defects.

Chapter 4: Comparing the Impact of Comment- vs. Metric-Based Technical Debt: We pool 40 open-source projects to investigate the ways in which code smells (god classes) and self-admitted technical debt influence software quality and concentrate on three points of view: (i) whether god and SATD files have more defects than non-god and non-SATD files, (ii) to what extent god and SATD changes are correlated with future defects and (iii) whether performing god and SATD changes imposes more difficulty on the system, where difficulty is measured by amount of churn, number of affected files and modified modules and change entropy. We conclude that: (i) god and SATD files are uncorrelated with defects, (ii) god and SATD changes induce a greater number of future defects and (iii) god and SATD changes make the system more complex. Thus, god classes and self-admitted technical debt are detrimental insofar as they increase future defects and change complexity.

1.3 Thesis Contributions

The major contributions of this thesis are as follows:

- A comprehensive review of the state of the art in the technical debt field, noting empirical gaps in the theory which ongoing research should reconcile with current treatments.

Chapter 2

Literature Review

In this chapter we present the key contributions of selected publications pertaining to technical debt, which puts the state of the art in focus and contextualizes the aims of this thesis. The studies we present first are concerned primarily with laying out the technical debt metaphor and establishing which cases such an analogy accurately describes, keeping in mind the utility of the metaphor, particularly in dealing with those less versed in software development jargon. These studies elaborate on the criteria that characterize sub-varieties of technical debt and how it is currently being implemented. Another collection of studies, presented second, discusses the issue of identifying technical debt in the source code and raises several long-term implications.

For the most part, this literature review incorporates prior work that centers on technical debt generally; information specific to the studies under discussion will accompany the following chapters.

2.1 The Popularization of the Technical Debt Metaphor

In the early days of technical debt, blogs curated by industry professionals circulated the most up-to-date information, but this medium largely left those outside the industry in the dark. In the time since, however, a greater emphasis on collaboration and information sharing has spurred extensive research, undertaken by both the industrial and academic fronts, on what exactly is subsumed under the technical debt metaphor. As its usage gains traction, more and more fits neatly into this category.

Ward Cunningham [5] originated the technical debt metaphor over twenty years ago as a means of negotiating a common language for the software developers and non-technical staff assigned to the same project. His original conception likened the additional effort incurred to maintain a project in the long term to the interest accrued on debt, such as a loan. Temporary fixes initially accelerate development and thus confer the short-term advantage of meeting deadlines otherwise unreasonable, yet if sufficient debt accumulates, the project grinds to a halt under the burden of incurred interest. It is the financial familiarity that makes sense of enhancing seemingly functional but unsustainable portions of code in layman's terms.

Steve McConnell [35] popularized the metaphor in his taxonomy, as did Martin Fowler [11] in devising the four quadrants outlined below. Due to the impact these innovations have had in terms of disseminating knowledge of technical debt throughout the software engineering community, the two subsections that follow examine each in turn.

2.1.1 Intentionally vs. Unintentionally Incurred Debt

Steve McConnell recognizes "intentionally incurred" (Type I) and "unintentionally incurred" (Type II) as the two principle classifications of technical debt. The latter comprises error-prone design techniques and poorly written code by an inexperienced programmer, among others. Type II debt results from low-quality work and is sometimes assumed without the recipient's knowledge, as in the case of company acquisitions and mergers.

Type I debt, in contrast, is incurred purposefully and in exchange for an immediate payoff. Software development companies, like all companies, make business decisions, strategically opting to accrue debt from time to time so that a deadline can be met. Justifications for incurring technical debt, such as "If we don't get this release done on time, there won't be a next release," are credible enough that some companies, for instance, use glue code to synchronize multiple databases before proper reconciliation can be conducted or postpone revisions that would ensure consistency in coding standards [35].

McConnell further partitions Type I debt into short- and long-term varieties. In keeping with the technical debt metaphor, short-term debt is assumed reactively and ideally paid off quickly and frequently, whereas organizations take on long-term debt proactively and, depending on the risk, sometimes count on expected income generated by an investment to pay it back.

[Sultan: Do we want an outline of McConnell's taxonomy here?]

2.1.2 The Technical Debt Quadrant

Advocating an alternative interpretation of the metaphor, Fowler [11] conceptualizes a typology of technical debt in which each of his four quadrants is designated either "reckless" or "prudent" and either "deliberate" or "inadvertent," allowing for four possibilities total. Prudent deliberate debt is assumed when a market supplier is fully

aware of what it is taking on and has conducted an in-depth cost-benefit analysis to determine whether the hypothetical additional revenue an earlier release generates exceeds the expense of repaying the debt later. The polar opposite, so-called "reckless inadvertent debt," is among the consequences of "not knowing any better," or being unacquainted with sound design practices [11].

As Fowler's quadrant schema demonstrates, reckless debt need not always coincide with inadvertent debt, nor prudent debt with deliberate debt. Companies cognizant of sound design practices, or even ones that ordinarily adhere to them, might opt for the "quick fix" rather than clean code under pressure. Prudent inadvertent debt arises when all parties are satisfied with the software delivered, which functions smoothly at the time and gives no indication of future issues, but it dawns on a developer afterwards that there was a more optimal solution. Of course, this is to be expected since programming is a learning process, albeit one that does not forgive debt incurred along the way [11].

The figure below displays Fowler's technical debt quadrants. Each of these contains a quote that sums up a prototypical scenario in which developers would resort to its particular combination of prudent/reckless and deliberate/inadvertent debt. **[move figure to be displayed underneath this paragraph]**

2.1.3 Other Insights on the Technical Debt Metaphor

The technical debt metaphor has found favor with software developers who need to convey to project stakeholders uninitiated in programming terminology similar debts and patchwork repairs that "kick the can down the road," temporarily delaying payment of money owed and putting off the effort of isolating a solution viable in the long term. Concepts falling under this umbrella include test and people debt, architectural and requirement debt and documentation and generalized software debt [44].

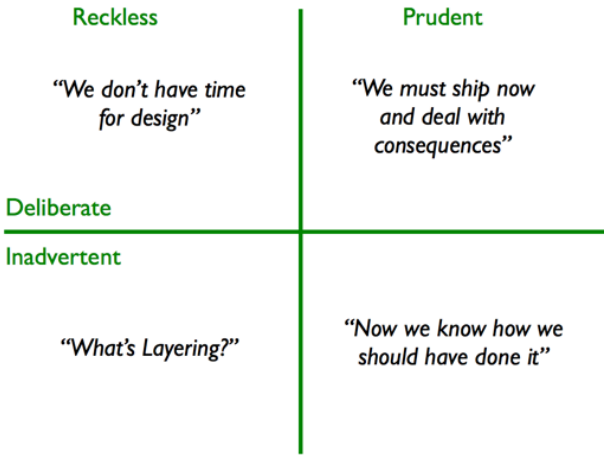

Figure 1: Technical Debt Quadrant

Broadening the metaphor to cover too many varieties of debt, however, might ultimately lessen its effectiveness, as Kruchten *et al.* point out [20]. Unimplemented requirements, functions or features do not qualify as requirement debts, just as putting off developing them does not qualify as a planning debt. Heavy reliance on tools alone to detect technical debt is one pitfall that the study highlights, in many cases leading to non-negligible underestimation of the actual technical debt load, since the majority of technical debt accumulates because of structural choices and technological gaps rather than code quality.

Further corroborating the overextension of the metaphor, Spinola *et al.* [43] compiled statements on technical debt that software developers made both online and in published work and selected 14 of them to use as items in two surveys measuring the level of agreement of 37 participants with software development backgrounds. On the whole, most participants strongly agreed that poorly managed technical debt drives up maintenance costs until they outpace consumer value and disagreed that all technical debt is accrued with a developer's full knowledge.

In the same study, the authors speculate that the technical debt metaphor's comprehensibility is what fuels its generalization to phenomena outside the realm of technical

debt in the truest sense. This in turn blurs the boundaries between technical debt and other costs or coding flaws and leads to persistent conflation among non-technical project contributors and, all too often, industry specialists, who adopt the metaphor as a rote catchall [43].

Alves *et al.* [1] have introduced a specialized vocabulary intended to disambiguate the subtleties that an all-purpose term such as *technical debt* overlooks, by sorting concepts extracted from a systematic literature mapping that combed 100 studies published between 2010 and 2014. Their undertaking identified 15 categories of technical debt but remained flexible enough to account for instantiations of technical debt that belonged in multiple categories: design debt, documentation debt, code debt, requirements debt, people debt, process debt, service debt, versioning debt, usability debt, build debt, test automation debt, infrastructure debt, defect debt, test debt and architecture debt. The work of Alves *et al.* and others who have monitored trends in the application of the technical debt metaphor and devised schemata relaying its latest interpretations has allowed developers and their stakeholders to make sense of the dynamic interplay between holdover solutions and deferred expense.

2.2 Technical Debt Indicators and Ramifications

2.2.1 Leveraging Source Code and Static Analysis Tools

Lately, there has been a lot of incentive to engineer better strategies for detecting and managing technical debt. Technical debt often gets out of hand and reaches unsustainable levels because a developer neglects to take stock as it accumulates. For all their limitations, static analysis tools do efficiently pinpoint violations of object-oriented design principles and source code anomalies outside the pre-specified ranges quantifying code quality. Such outliers constitute "bad smells," which fall under the category of design debt.

In a study probing the effects of god classes (another manifestation of design debt) on project maintainability, Zazworka *et al.* [50] examined two commercial applications released by a development company and concluded that god classes are more liable to be defective, and thus higher-maintenance, than non-god classes. For this reason, it is worthwhile for developers to monitor and, where appropriate, rein in the toll that technical debt takes on product quality, at all stages in the process.

God classes and other bad smells—namely, data class and duplicate code—were extracted from open-source systems and scrutinized by Fontana *et al.* [10] in an effort to prioritize the handling of different types of design debt. Their approach ranks bad smells in descending order with respect to negative impact on software quality and encourages developers to rectify higher-priority design debts first.

Zazworka *et al.* [50] elicited an enumeration of technical debt items stored in project artifacts from multiple developers and compared the results with what three static analysis tools identified as fitting the relevant criteria. As different teams reported different technical debt items, consensus engenders underestimation of the actual technical debt load and aggregation proves to be the better method. Similarly, static analysis tools will yield underestimations—some varieties of technical debt going undetected—unless supplemented with human mediation.

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

While strides have been made in locating sources of technical debt and preventing unsustainable accumulation, such as integrating tool- and developer-flagged code, new improvements are constantly proposed, debated and adopted for use alongside

older, "tried and tested" methodologies. One such improvement, from Potdar and Shihab [38], enlists source code comments in isolating technical debt, the benefit of which is that the program developer *confesses* the debt. At best, analysis tools can only *suppose* debt on the basis of semi-arbitrary cutoffs and thresholds, and stop short of guaranteeing that an implementation is less than optimal, i.e., *self-admitted technical debt*.

In their pioneer study capturing the state of the art in self-admitted technical debt identification, Potdar and Shihab [38] extracted source code comments from five open-source projects and conducted manual inspections. The authors read and analyzed more than 100,000 comments and in the end isolated 62 different comment patterns that serve as reliable indicators of self-admitted technical debt, most consisting of simple phrases along the lines of "fixme," "workaround," and "this can be a mess." It was found that: (i) between 2.4% and 31.0% of the files analyzed contained these keywords, (ii) the bulk of the self-admitted technical debt was introduced by more experienced developers and (iii) there is no correlation between time pressures or code complexity and the amount of self-admitted technical debt.

Building on the groundbreaking work of Potdar and Shihab [38], Bavota and Russo [2] considered the growth and evolution of self-admitted technical debt across 159 projects and the effects this has had on software quality, and extracted upwards of 600,000 commits and two billion source code comments. They found that: (i) self-admitted technical debt is diffused, averaging 51 occurrences per system, (ii) it accumulates over time as new occurrences pile up on top of ones which have not yet been corrected and (iii) the occurrences that are corrected have a mean lifespan of 1,000 commits in the system.

2.2.3 Research Leveraging Source Code Comments

[too similar to the last subsection's title?]

A number of studies examined the usefulness/quality of comments and showed that comments are valuable for program understanding and software maintenance [46, 47, 23]. For example, Storey *et al.* [45] explored how task annotations in source code help developers manage personal and team tasks. Takang *et al.* [46] empirically investigated the role of comments and identifiers on source code understanding. Their main finding showed that commented programs are more understandable than non-commented programs. Khamis *et al.* [18] assessed the quality of source code documentation based on an analysis of the quality of language and consistency between source code and its comments. Tan *et al.* proposed several approaches to identify code-comment inconsistencies. The first, called @iComment, detects lock- and call-related inconsistencies [47]. The second approach, @aComment, detects synchronization inconsistencies related to interrupt context [48]. A third approach, @tComment, automatically infers properties from Javadoc related to null values and exceptions; it performs test case generation by considering violations of the inferred properties [49].

Other studies have examined the co-evolution of comment updates as well as the reasons behind them. Fluri *et al.* [9] studied the co-evolution of source code and associated comments and found that 97% of the comment changes are consistently co-changed. Malik *et al.* [26] performed a large empirical study to understand the rationale for updating comments along three dimensions: characteristics of the modified function, characteristics of the change, as well as the time and code ownership. Their findings showed that the most relevant attributes associated with comment updates are the percentage of changed call dependencies and control statements, the age of the modified function and the number of co-changed functions which depend

on it. De Lucia *et al.* [7] proposed an approach to help developers maintain source code identifiers and consistent comments with high-level artifacts. The main results of their study, based on controlled experiments, confirm the conjecture that providing developers with similarity between source code and high-level software artifacts helps to enhance the quality of comments and identifiers.

Most relevant to our research is the work recently undertaken by Potdar and Shihab [38], which uses source code comments to detect self-admitted technical debt. Using the identified technical debt, they studied how much SATD exists, the rationale for SATD and the likelihood of its removal after introduction. Another relevant contribution to our study is Maldonado and Shihab's [25], as their work has also leveraged source code comments to detect and quantify different types of SATD. They classified SATD into five types: design debt, defect debt, documentation debt, requirement debt and test debt. Ultimately, they concluded that the most common type is design debt, accounting for anywhere between 42% and 84% of a total of 33,000 classified comments.

Our study builds on prior work in [38, 25] since we use the comment patterns they produced to detect SATD. However, in a departure from these studies, we examine the relationship between SATD and software quality.

2.2.4 Technical Debt

Other work has focused on the identification and examination of technical debt. It is important to note that the technical debt discussed here is *not* SATD: rather, it is technical debt that is detected through source code analysis tools. For example, Zazworka *et al.* [51] attempted to identify technical debt automatically and then compared their automated identification with human elicitation. The results of their study outline potential benefits of developing tools and heuristics for the detection of technical debt. Also, Zazworka *et al.* [50] investigated how design debt, in the form

of god classes, affects software maintainability and correctness of software products. Their study involved two industrial applications and showed that god classes are changed more often than non-god classes and, moreover, that they contain more defects. Their findings suggest that technical debt may negatively influence software quality. Guo *et al.* [14] analyzed how and to what extent technical debt affects software projects by tracking a single delayed task in a software project throughout its lifecycle. As discussed earlier, the work by Potdar and Shihab [38] is also related to our work, which differs from precedent primarily in that it focuses on SATD.

Our work differs from foregoing research by Zazworka *et al.* [50, 51] since we focus on the relationship between SATD (and not technical debt related to god files) and software quality. However, we believe that our study complements prior studies since it sheds light on the overall impact of SATD and, in particular, its ramifications for software quality.

2.2.5 Software Quality

A plethora of prior work has proposed techniques to improve software quality, the majority of this work having concerned itself with understanding and predicting software quality issues (e.g. [52]). Several studies have examined the metrics that best indicate software defects, including design and code [16], code churn [37] and process metrics [36, 39].

Other studies have opted to focus on change-level prediction of defects. Sliwerski *et al.* suggested a technique known as SZZ to automatically locate fix-inducing changes by linking a version archive to a bug database [41]. Kim *et al.* [19] used identifiers in added and deleted source code and the words in change logs to identify changes as defect-prone or not. Similarly, Kamei [17] proposed a "Just-In-Time Quality Assurance" approach to identify risky software changes in real time. The findings of their study reveal that process metrics outperform product metrics in terms of identifying

risky changes.

Our study leverages the SZZ algorithm and some of the techniques presented in the aforementioned change-level work to study the defect-proneness of SATD-related commits. Moreover, our study complements existing work by taking up the hypothetical correlation between SATD and software defects.

Chapter 3

Examining the Impact of Self-Admitted Technical Debt on Software Quality

3.1 Introduction

Software companies and organizations have a common goal when developing software projects: both aim to deliver high-quality, useful software in a timely manner. However, in most practical settings, developers and development companies are saddled with deadlines, giving them every incentive to release earlier than the ideal date, were product quality alone taken into account. Such situations are all too common and in many cases force developers to take shortcuts [21] [40]. Recently, the term *technical debt* was coined to denote the phenomenon of "doing something that is beneficial in the short term but will incur a cost later on" [5]. Prior work has shown that practitioners cite numerous reasons for assuming technical debt, among them: rushing to compensate for delays and still deliver on time or to make deadlines for incorporating with a partner product before release, alleviating time-to-market pressure and

meeting customer demands in a time-sensitive industry [24].

More recently, a study by Potdar and Shihab [38] introduced a novel method of identifying technical debt reported by developers. This so-called "self-admitted technical debt," abbreviated SATD, is declared in developer source code comments. Prior work [25] has demonstrated that accrual of SATD is commonplace in software projects, where reviewing source code comments can identify different types of technical debt (e.g. design, defect and requirement debt).

Intuition and general belief concur that inducing a technical debt, which many developers resort to in a time crunch, negatively impacts software maintenance and overall quality [50, 43, 14, 40, 21]. However, to the best of our knowledge, there is no empirical study that examines the impact of SATD on software quality. Such a study is critical since it will help us to confirm or refute entrenched preconceptions regarding the technique and better understand how to manage SATD.

Therefore, in this chapter, we investigate the empirical relation between SATD and software quality in five open-source projects. In particular, we examine whether (i) files with SATD have more defects compared to files without SATD, (ii) whether SATD changes introduce more future defects and (iii) whether SATD-related changes tend to be more difficult. We measured the difficulty of a change in terms of the amount of churn, number of files, number of modified modules, and change entropy. Our findings show that: i) while it is true that SATD files have more bug-fixing changes in a number of the studied projects, in other projects, files without SATD have more defects, thus there is no clear relationship between defects and SATD; ii) SATD changes are associated with less future defects than non-SATD changes and iii) SATD changes (i.e., changes touching SATD files) are more difficult to perform. Our study indicates that although technical debt has negative effects, its impact is not related to defects, but rather to making the system more difficult to change in the future.

3.2 Related Work

[TBC]

3.3 Approach

The objective of our study is to investigate the relationship between SATD and software quality. We measure software quality in two ways. First, we employ the traditional measure of counting the defects in a file and defect-inducing changes, which is in line with most prior studies [17, 19, 42]. In particular, we measure the number of defects in SATD-related files and the percentage of SATD-related changes that introduce future defects. Second, since technical debt is meant to represent the phenomenon of taking a short-term benefit at the cost of paying a higher price later on, we employ as another measure the difficulty of the changes related to SATD. Specifically, we use amount of churn, number of files, number of directories and change entropy to quantify difficulty. We formalize our study with the following three research questions:

- **RQ1:** Do files containing SATD have more defects than files without SATD? Do the SATD files have more defects after the introduction of SATD?

- **RQ2:** Do SATD-related changes introduce future defects?

- **RQ3:** Are SATD-related changes more difficult than non-SATD changes?

To address our research questions, we followed the general procedure enumerated in Figure 2, which consists of the following steps. First, we mined the source code repositories of the studied projects (step 1). Then, we extracted source code files at

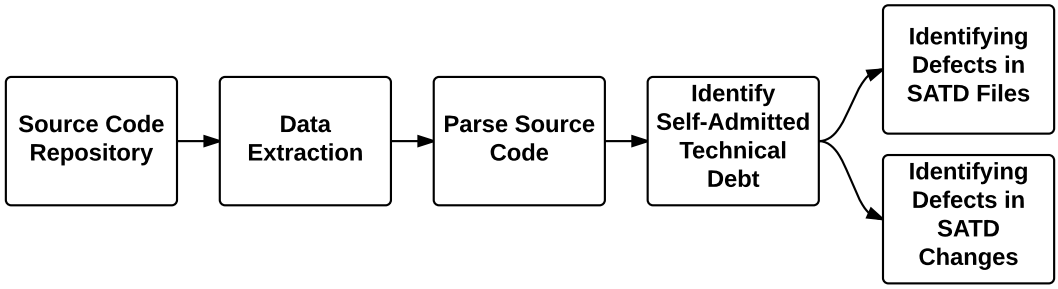

Figure 2: Approach overview.

the level of each analyzed project (step 2). Next, we parse the source code and extract comments from the source code of the analyzed systems (step 3). At this point, we apply the comment patterns proposed by Potdar and Shihab [38] to identify SATD (step 4). Finally, we analyze the changes to quantify defects in files and use the SZZ algorithm to determine defect-inducing changes (step 5).

3.3.1 Data Extraction

Our study analyzes five large open-source software systems—namely Chromium, Hadoop, Spark, Cassandra and Tomcat. We chose these projects because they represent different domains and programming languages (i.e., Java, C, C++, Scala, Python and Javascript) and have a large number of contributors. More importantly, these projects are well-commented (since our approach for the detection of SATD is based on source code comments). Moreover, they are all available to the research community as well as industry practitioners and have considerable development history.

Our analysis requires the source code as input. We downloaded the latest publicly available releases of the systems under consideration, i.e., Chromium, Hadoop, Spark, Cassandra and Tomcat. Then, we filtered the data to extract the source code at the level of each project release. Files not consisting of source code (e.g. CSS, XML, JSON) were excluded from our analysis as they do not contain the comments

Table 1: Characteristics of the studied projects.

Project	Release	Release Date	# Lines of Code	# Comment Lines	# Files	# Committers	# Commits
Chromium	45	Jul 10, 2015	9,388,872	1,760,520	60,476	4,062	283,351
Hadoop	2.7.1	Jul 6, 2015	1,895,873	378,698	7,530	155	11,937
Spark	2.3	Sep 1, 2015	338,741	140,962	2,822	1,056	13,286
Cassandra	2.2.2	Oct 5, 2015	328,022	72,672	1,882	219	18,707
Tomcat	8.0.27	Oct 1, 2015	379,196	165,442	2,747	34	15,914

our analysis relies on.

Table 1 summarizes the main characteristics of these projects. It reports for each: (i) the relevant project release, (ii) the date of the release, (iii) the number of lines of code, (iv) the number of comment lines, (v) the number of source code files, (vi) the number of committers and (vii) the number of commits.

3.3.2 Scanning Code and Extracting Comments

After obtaining the source code of the five software projects, we extracted the comments from their source code files. To this end, we developed a Python-based tool that identifies comments based on the use of regular expressions. This tool also indicates comment type (i.e., single-line or block comments), the name of the file where the comment appears and the line number of the comment. To ensure our tool's accuracy, we employ the Count Lines of Code (CLOC) tool [6]. As long as the total number of lines of comments is the same according to both tools, then the tool we developed can be considered independently reliable.

In total, we found 879,142 comments for Chromium; 71,609 for Hadoop; 31,796 for Spark; 20,310 for Cassandra and 39,024 for Tomcat. Of these, SATD comments numbered 18,435 for Chromium; 2,442 for Hadoop; 1,205 for Spark; 550 for Cassandra and 1,543 for Tomcat. To enable easy processing, we store all of our processed data in a PostgreSQL database, which we query to answer our RQs.

3.3.3 Identifying Self-Admitted Technical Debt

To perform our analysis, we need to identify self-admitted technical debt at two levels: (i) the file level and (ii) the change level.

SATD files: To identify SATD, we followed the methodology outlined in Potdar and Shihab [38], who generated a list of 62 different patterns that indicate SATD. Therefore, in our approach, we determine which comments identify SATD by locating those that match any of the 62 patterns associated with SATD. These patterns are extracted from several projects and some appear more often than others. Examples of these patterns include: "*hack, fixme, is problematic, this isn't very solid, probably a bug, hope everything will work, fix this crap.*" The complete list of the patterns considered in this study is available online[1].

Once we identify the comment patterns, we then abstract up to determine the SATD files. Files containing at least one of the SATD comments are then labeled as *SATD files*, while files that do not contain any of these SATD comments are referred to as *non-SATD files*. We use these SATD files to answer RQ1.

SATD changes: To study the impact of SATD at the change level, we need to identify SATD changes on the basis of the SATD files just identified. We analyze the changes and determine all the files that were touched by each change. If at least one of the files touched by the change is an SATD file, then we label that particular change as an SATD change. If the change does not touch any SATD files, then we label it as a non-SATD change. Table 2 displays the percentage of SATD comments and files for each of the studied systems. From the table, we see that SATD comments exhaust less than 4% of the total comments, and between 10.17% and 20.14% of the files are SATD files.

[1] http://users.encs.concordia.ca/~eshihab/data/ICSME2014/data.zip

Table 2: Percentage of SATD of the analyzed projects.

Project	SATD Comments (%)	SATD files (%)
Chromium	2.09	10.43
Hadoop	3.41	18.59
Spark	3.79	20.14
Cassandra	2.70	16.01
Tomcat	3.95	10.17

3.3.4 Identifying Defects in SATD Files and SATD Changes

To determine whether a change fixes a defect, we search for co-occurrences of defect identifiers in change logs from the Git Version control system using regular expressions like "*fixed issue #ID, bug ID, fix, defect, patch, crash, freeze, breaks, wrong, glitch, properly, proper.*" Sliwersky *et al.* [42] showed that the use of such keywords in the change logs usually indicates the correction of a mistake or failure. A similar approach was applied to identify fault-fixing and fault-inducing changes in prior work [17, 19, 42]. Once this step is performed, we identify, for each defect ID, the corresponding defect report from the corresponding issue tracking system, i.e., Bugzilla[2] or JIRA[3], and extract the relevant information from each report.

After grouping the SATD files and SATD changes, we proceed to identify the defects each contains. To do so, we follow the protocol previous research has adhered to in determining the number of defects in a file and locating defect-inducing changes [17, 19, 42].

Defects in files: Comparing the defectiveness of SATD and non-SATD files hinges on having the number of file defects at our disposal. To ensure this, we extract all the changes that have touched a file throughout the system's entire history. Then, we search for keywords in the change logs that indicate defect-fixing, as demonstrated

[2]https://www.bugzilla.org
[3]https://www.atlassian.com/software/jira

in Figure 3. A subset of the keywords we entered contains: "*fixed issue #ID, bug ID, fix, defect, patch, crash, freeze, breaks, wrong, glitch, proper.*" In cases where a defect identification is specified, we extract the defect report to verify that the defect corresponds to the system (i.e., product).

Second, we establish whether the issue IDs identified in the change logs are true positives. Once we determine the defect-fixing changes, we use these changes as an indication of the defect fixes that occur in a file, i.e., we count the number of defects in a file as the number of defect-fixing changes.

Defect-inducing changes: Similar to the process above, we first determine whether a change fixes a defect. To do so, we use regular expressions and specific keywords referencing a fix to search the change logs (i.e., commit messages) from the source code control versioning system. In particular, we search for the following keywords: "*fixed issue #ID, bug ID, fix, defect, patch, crash, freeze, breaks, wrong, glitch, proper.*" We also search for the existence of defect identification numbers in order to determine which defects, if specified, the changes actually fix.

Once we identify the defect-fixing changes, we map back (using the blame **[special font for commands?]** command) to determine all the changes that altered the fixed code in the past. We take the defect-inducing change to be the change that is closest to but still before the defect report date. In essence, this tells us that this was the last change before a defect showed up in the code. If no defect report is specified in the fixing change, then following the precedent of prior work [17], we assume that the last change before the fixing change was the change that introduced the defect. This approach is often referred to as the SZZ [42] or approximate (ASZZ) algorithm [17] and is to date the state of the art in identifying defect-inducing changes.

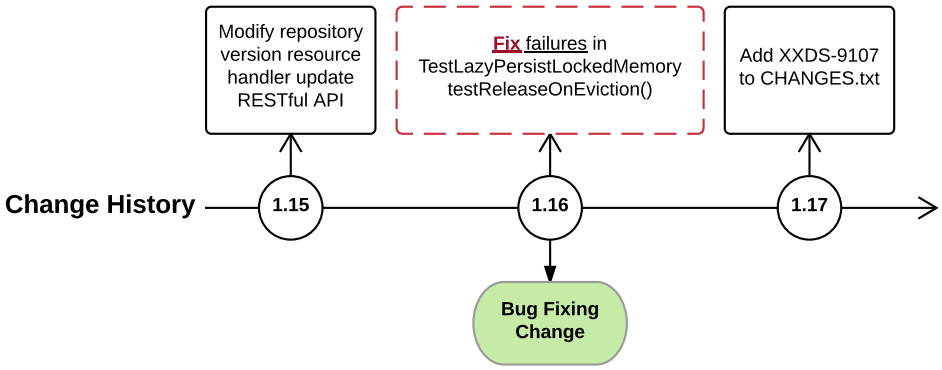

Figure 3: Indicating a bug-fixing change.

3.4 Mann-Whitney-Wilcoxon Rank Sum Test

The Mann-Whitney-Wilcoxon Rank Sum Test is used to analyze the differences between two groups of the same attribute for a data set [27]. This statistical test makes use of median values for its comparison rather than mean values, allowing it to characterize populations that do not follow a normal curve distribution. The main result of the test is the p-value it generates, which quantifies the probability of the null hypothesis being true, with the null hypothesis in this case being that both groups have the same central tendency. In our study we use this test to determine if the distinction between SATD and non-SATD files results in a difference in relevant statistical properties. If it does, then whatever caused a noticeable distinction between the file categories is meaningful to the statistical property. **[was the "iffalse" stuff supposed to show up before section 3.5?]**

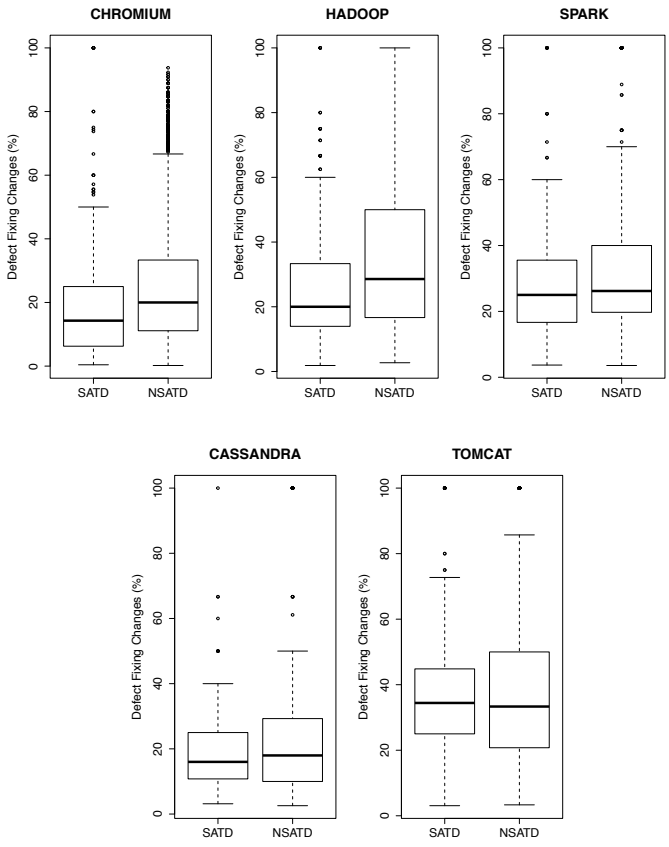

Figure 4: Percentage of defect-fixing changes for SATD and NSATD files.

3.5 Case Study Results

This section reports the results of our empirical study examining the relationship between self-admitted technical debt and software quality.

For each project, we provide the descriptive statistics and statistical results, as well as a comparison with the other projects.

In what follows, we present for each RQ its motivation, the approach we took to address it and our findings.

RQ1: Do files containing SATD have more defects than files without SATD? Do the SATD files have more defects after the introduction of SATD?

Motivation: Intuitively, technical debt has a negative impact on software quality. Researchers have studied technical debt and shown that it negatively impacts software quality [50]. However, this research has neglected SATD, which is prevalent in software projects according to past research [38].

Empirically examining the impact of SATD on software quality provides researchers and practitioners with a better global understanding of the phenomenon, warns them of its future risks and raises awareness of the obstacles or challenges it can pose.

In addition to comparing the defect-proneness of SATD and non-SATD files, we compare the defect-proneness of SATD files before (pre-SATD) and after SATD (post-SATD). This analysis provides us with a different view of the defect-proneness of SATD files and, in essence, tells us whether the introduction of SATD is at all related to the defects we observe.

Approach: To address RQ1, we perform two types of analyses. First, we compare the defect-proneness of files that do and do not contain SATD. Second, for the SATD files only, we compare defect-proneness before and after the introduction of SATD.

Comparing SATD and non-SATD files. To perform this analysis, we follow the procedure for identifying SATD files summarized earlier in section 3.3.3. In a nutshell, we determine which files contain at least one SATD comment and label them as SATD files. Files that do not contain any SATD are labeled as non-SATD files. Once we sort the files, we determine the percentage of defect-fixing changes in each file category (SATD and non-SATD). We opt for percentages over raw numbers so as to normalize our data, since files can have different amounts of changes. To answer the first part of RQ1, we plot the distribution of defects by file category and

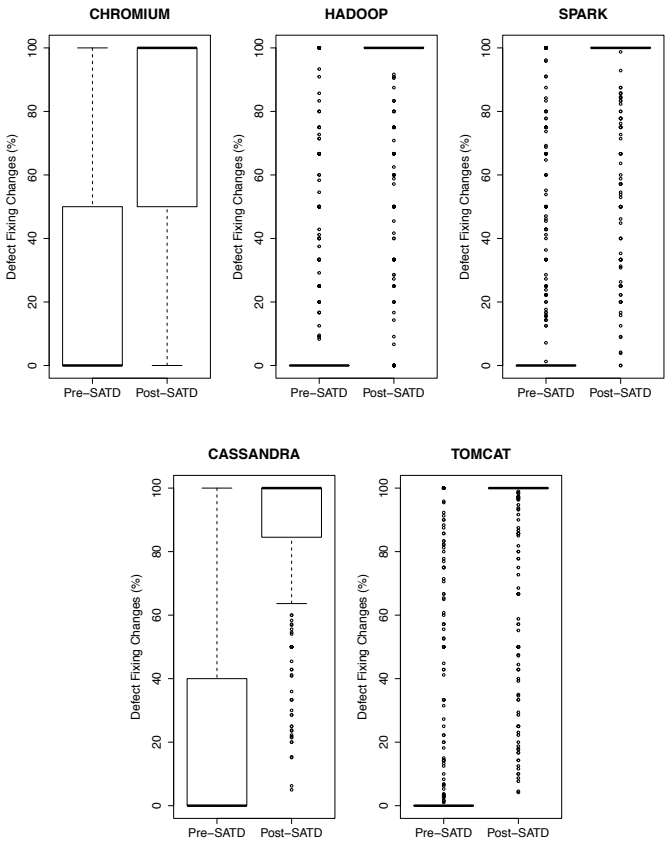

Figure 5: Percentage of defect fixing changes for pre-SATD and post SATD.

perform statistical tests to compare the differences.

We perform the Mann-Whitney [27] test to determine if a statistical difference exists between the categories and Cliff's delta [13] to compute the effect size. We use the Mann-Whitney test instead of other statistical difference tests because it is a non-parametric test that accommodates non-normal distribution (and as we will see later, our data is not normally distributed). We consider the results of the Mann-Whitney test to be statistically significant if the p-value is such that $p <= 0.05$. In addition, we compute the effect size of the difference using the Cliff's delta (d) non-parametric effect size measure, which measures how often values in one distribution are larger than the values in another. Cliff's d ranges in the interval $[-1, 1]$ and is considered small for $0.148 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$.

Comparing files pre- and post-SATD. To compare SATD files pre- and post-SATD, we first determine all the changes that touched a file and then identify the change that introduced the SATD. Next, we measure the percentage of defects (i.e., $\frac{\#\ of\ fixing\ changes}{total\ \#\ changes}$) in the file before and after the introduction of the SATD. We compare percentage of defects instead of raw numbers since SATD could be introduced at different times, i.e., we may not have the same total number of changes before and after the SATD-inducing change. Once we determine the percentage of defects in a file pre- and post-SATD, we perform the same statistical test and effect size measure, i.e., Mann-Whitney and Cliff's delta.

Results - Defects in SATD and non-SATD files: Figure 4 shows the percentage of defect-fixing changes in SATD and non-SATD files for the five projects. We observe that in four out of five cases, the non-SATD (NSATD) files have a slightly higher percentage of defect-fixing changes—in Chromium, Hadoop, Spark and Cassandra. However, in Tomcat, SATD files have a slightly higher percentage of defects. For all projects, the p-values were such that $p < 0.05$, indicating that the difference is statistically significant. However, when we closely examine the Cliff's delta values in Table 3, we see a different trend for Chromium. In Chromium and Tomcat, SATD files often have higher defect percentages than non-SATD files and the effect size is medium for Chromium and small for Tomcat. On the other hand, in Hadoop, Cassandra and Spark, SATD files have lower defect percentages than non-SATD files and this effect is large for Hadoop, medium for Cassandra and small for Spark.

Our findings here underscore that there is no clear trend when it comes to the percentage of defects in SATD versus non-SATD files. In some projects, SATD files have more bug-fixing changes, while in others, it is the non-SATD files that have a higher percentage of defects.

Results - Defects in SATD files, pre- and post-SATD: Figure 5 shows boxplots

Table 3: Cliff's Delta for SATD versus NSATD and POST versus PRE fixing changes.

Project	SATD vs. NSATD	Post- SATD vs. Pre- SATD
Chromium	0.407	0.704
Hadoop	-0.562	0.137
Spark	-0.221	0.463
Cassandra	-0.400	0.283
Tomcat	0.094	0.763

for the percentage of defect-fixing changes in SATD files, pre- and post-SATD. Unsurprisingly, the post-SATD percentage of defect-fixing changes is higher for all projects. In Table 3, the effect size Cliff's delta values corroborate our visual observations in that there is again more defect-fixing in the SATD files post-SATD than pre-SATD. For all projects except Hadoop and Cassandra, where effect size is small, the Cliff's delta is large.

These findings contend that although it is not always clear whether SATD or non-SATD files will have a higher percentage of defects, there is a consistently higher percentage of defect-fixing once SATD has been introduced.

RQ2: Do SATD-related changes introduce future defects?

Motivation: After investigating the relationship between SATD and non-SATD at the file level, we would like to conclude whether SATD changes are more likely to introduce future defects. Whereas the file-level analysis looked at files as a whole, our analysis here is more fine-grained and tailored to assess individual changes.

Studying the propensity of SATD changes to introduce future defects is important, as it informs us as to how SATD and non-SATD changes compare in terms of future introduction of defects and how quickly the impact of self-admitted technical debt on quality can be felt. For example, if SATD changes introduce defects in the very next

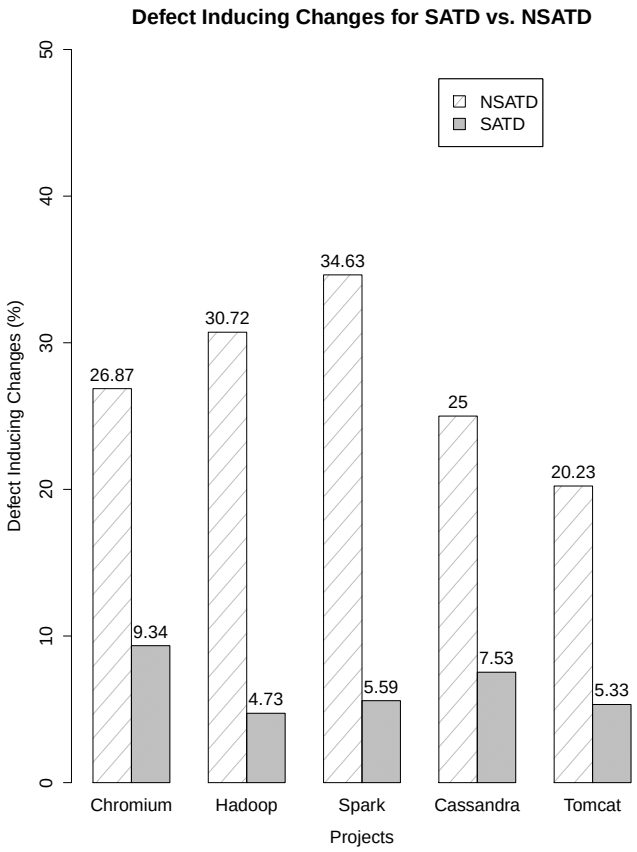

Figure 6: Percentage of defect inducing changes with SATD and NSATD.

change, then this tells us that there is essentially no latent stage and the impact of SATD is felt almost immediately. Our conjecture is that SATD changes tend to be more complex and lead to the introduction of defects.

Approach: To address RQ2, we applied the SZZ algorithm [42] in order to detect defect-inducing changes. Then, we sorted the results into two categories: SATD and non-SATD defect-inducing changes.

Results: Figure 6 demonstrates that non-SATD changes have a higher incidence of defect-inducing changes relative to SATD changes. In Chromium, for example, roughly 10% of the SATD changes induce future defects, compared to about 27% of the non-SATD changes. Our findings here show that contrary to our conjecture, SATD changes actually have a lower chance of inducing future defects.

Table 4: Cliff's Delta for the change difficulty measures across the projects.

Project	# Modified Files	Entropy	Churn	# Modified Directories
Chromium	0.418	0.418	0.386	0.353
Hadoop	0.602	0.501	0.768	0.572
Spark	0.663	0.645	0.825	0.668
Cassandra	0.796	0.764	0.898	0.827
Tomcat	0.456	0.419	0.750	0.390

RQ3: Are SATD-related changes more difficult than non-SATD changes?

Motivation: Thus far, our analysis has confined itself to the relationship between SATD and software defects. However, by definition, technical debt entails some sort of tradeoff where a short-term benefit ends up costing more in the future. Therefore, it remains to be decided to what extent this tradeoff makes effecting changes more difficult after the introduction of technical debt.

Answering this question will help us understand the impact of SATD on future changes and provide us with a different view on how SATD impacts a software project.

Approach: To answer this question, we classify the changes into two groups, i.e., SATD and non-SATD changes. Then, we compare the difficulty of performing the two types of changes. We quantify the difficulty of a change using four different metrics: the total number of modified lines in the change (churn), the number of modified directories, the number of modified files and change entropy. The first three are motivated by earlier work in which Eick *et al.* [8] measure software decay. The change entropy metric is motivated by the work of Hassan [15], in which it measures change complexity.

To measure the change churn, number of files and number of directories, we use

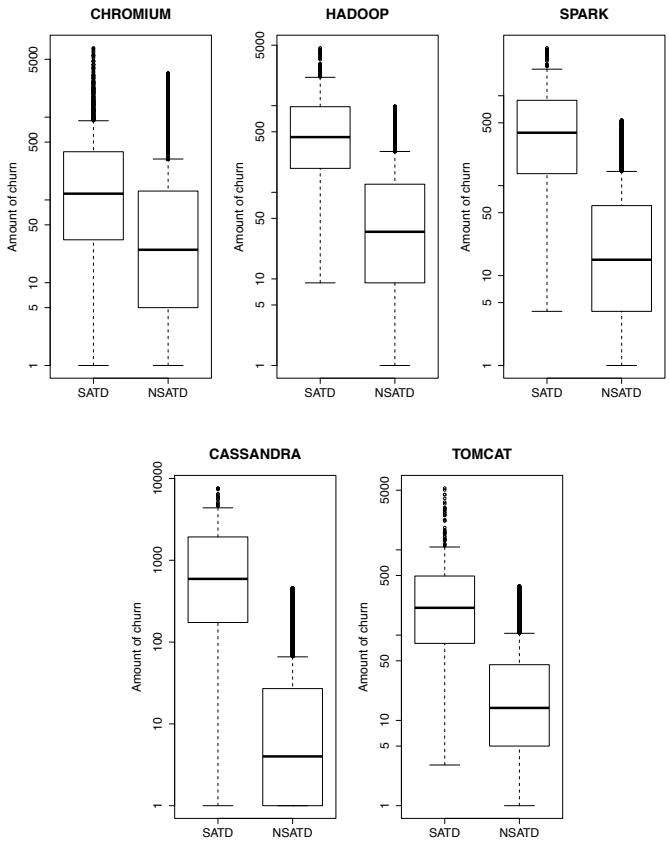

Figure 7: Total number of lines modified per change (SATD vs. NSATD).

data from the change log directly. The churn is given for each file touched by the change, so we simply aggregate the churn of the individual files to determine the overall churn of the change. The list of files is extracted from the change log to determine the number of files and directories touched by the change. When measuring the number of modified directories and files, we refer to a directory as **ND** and a file as **NF**. Hence, if a change involves the modification of a file having the path "net/base/registry_controlled_domains/effective_tld_names.cc," then the directory is *base/registry_controlled_domains* and the file is *effective_tld_names.cc*.

To measure the entropy of a change, we use the change complexity measure proposed by Hassan [15]. Entropy is defined as: $H(P) = -\sum_{k=1}^{n}(p_k * log_2 p_k)$, where k is the proportion file$_k$ is modified in a change and n is the number of files in the change.

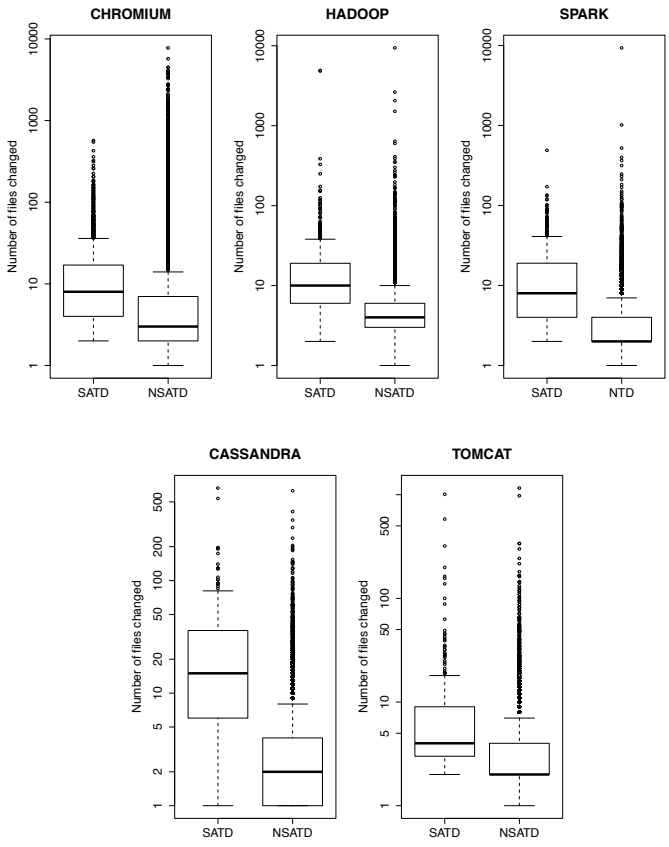

Figure 8: Total number of files modified per change (SATD vs. NSATD).

Entropy measures the distribution of a change across different files. Let us consider a change that involves the modification of three different files named *A, B* and *C* and let us suppose that the number of modified lines in files *A, B* and *C* is 30, 20 and 10 lines, respectively. The entropy is equal to: $(1.46 = -\frac{30}{60}\log_2\frac{30}{60} - \frac{20}{60}\log_2\frac{20}{60} - \frac{10}{60}\log_2\frac{10}{60})$. As in Hassan [15], the above entropy formula has been normalized by the maximum entropy $log_2 n$ to account for differences in the number of files per change. The higher the normalized entropy, the more difficult the change.

Results: Figures 7, 8, 9, 10 reveal that for all difficulty measures, SATD changes have a higher value than non-SATD changes. We also find that the difference between the SATD and non-SATD changes is statistically significant, with a p-value such that $p < 0.05$. Table 4 shows the Cliff's delta effect size values for all projects studied.

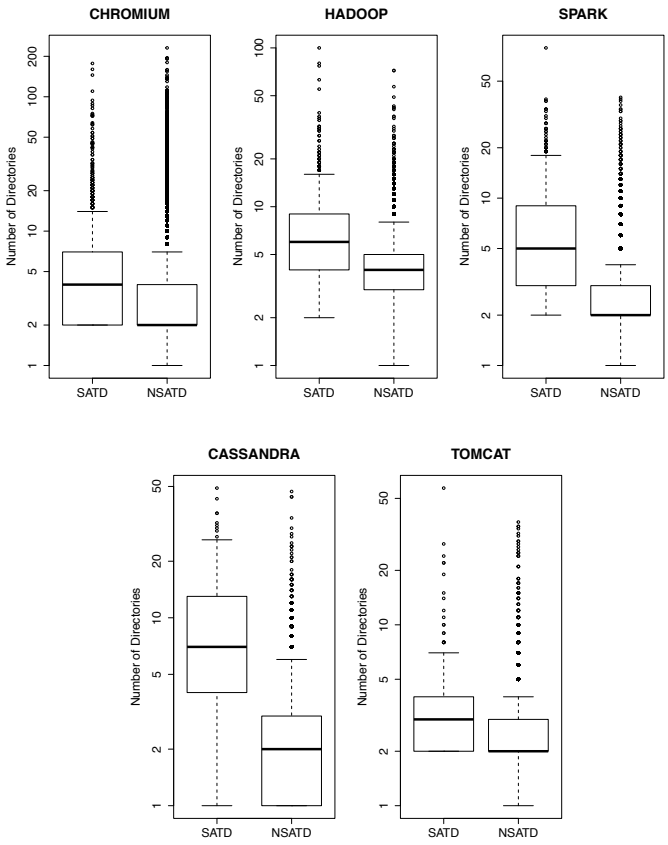

Figure 9: Total number of modified directories per SATD and NSATD change.

We observe that for all projects and all measures of difficulty, the effect size is either medium or large (Cf. Table 4), which indicates that SATD changes are more difficult than non-SATD changes.

In summary, we conclude that SATD changes are more difficult than non-SATD changes, provided that difficulty is measured using churn, the number of modified files, the number of modified directories and change entropy.

3.6 Threats to Validity

Threats to **internal validity** concern any factors that could have confounded our study results. To identify self-admitted technical debt, we use source code comments.

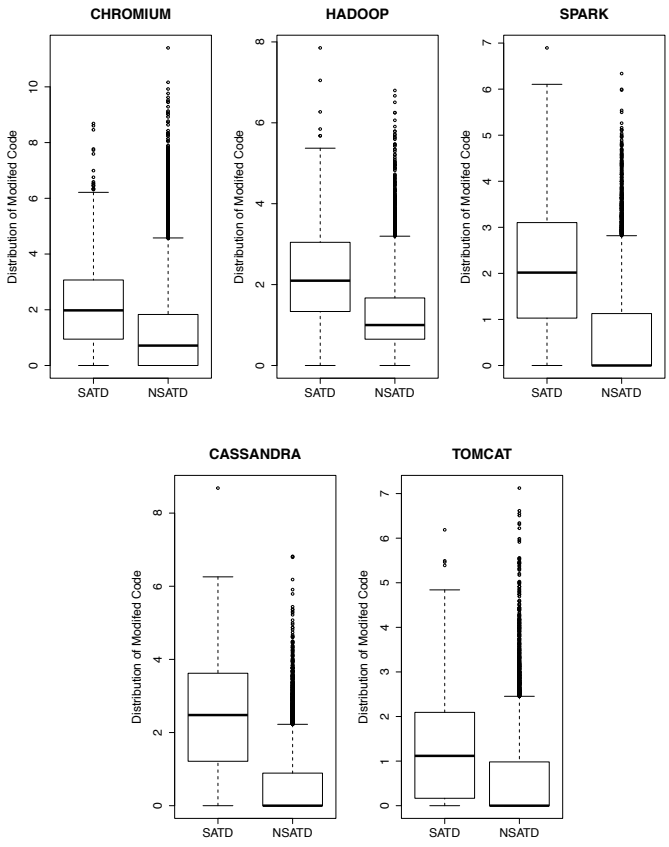

Figure 10: Distribution of the change across the SATD and NSATD files.

In some cases, though, developers may not add comments when they introduce technical debt. The opposite poses another threat, namely, that developers might introduce technical debt and subsequently remove it without removing the related comment. In both cases the code and comment change inconsistently. However, Potdar and Shihab [38] examined this phenomenon in Eclipse and found that in 97% of cases code and comments change in tandem. As a means of locating SATD, we use the comments compiled by Potdar and Shihab [38], yet there is a possibility that these patterns do not detect all SATD. Additionally, given that comments are written in natural language, Potdar and Shihab had to manually read and analyze them to determine those that would indicate SATD. Manual analysis is prone to subjectivity

and errors and therefore we cannot guarantee that all considered patterns will be perceived as SATD indicators by other developers. To mitigate this threat, we manually examined each comment that we detected and verified that it contained one of the 62 patterns in [38]. We performed this step, independently, for each of the five projects studied. We identify a change as an SATD change if it contains at least one SATD file. Alternatively, we could have defined SATD changes as only those for which all files have SATD. We elected to do it the former way because sometimes SATD in just one file can impact the rest of the change, e.g. it may cause many other files to be changed. When measuring the percentage of file defects after the introduction of SATD, it is difficult to distinguish the differences due to SATD from those attributed to natural evaluation of the files.

Threats to **external validity** concern the possibility that our results may not generalize. To optimize generalizability, we analyzed five large open-source systems and drew our data from the well-established, mature codebase of open-source software projects with well-commented source code. These projects belong to different domains and they are written in different programming languages.

Furthermore, we focused on SATD only, which means that we do not cover all technical debt—there could well be other technical debt that is not self-admitted. Studying all technical debt is beyond the scope of this thesis.

3.7 Conclusion and Future Work

Technical debt is intuitively recognized as bad practice by software companies and organizations. However, there is very little empirical evidence on the extent to which technical debt can impact software quality. Therefore, in this thesis we perform an empirical study, using five large open-source projects, to determine precisely how technical debt relates to software quality. We focus on self-admitted technical debt,

which refers to errors that might be introduced as part of intentional and temporary quick fixes. As in [38], we leverage source code comments to identify such debt on the basis of recurring indicator patterns.

We examined the relationship between self-admitted technical debt and software quality by investigating: (i) whether files with SATD have more defects compared to files without SATD, (ii) whether SATD changes introduce future defects and (iii) whether SATD-related changes tend to be more difficult. We measured the difficulty of a change in terms of amount of churn, number of files and modified modules and change entropy.

Our findings suggest that there is no reliable trend when it comes to defects and self-admitted technical debt. In some of the projects, self-admitted technical debt files had more bug-fixing changes, while in others, files without it had more defects. We also found that SATD changes are less correlated with future defects than non-SATD changes, but more difficult to perform. *[Sultan: Our study demonstrates that although defects are not among the negative effects of technical debt, it can make the system more difficult to change in the future.]* Our study demonstrates that although technical debt may have negative effects, its impact does not extend to defects, but rather to making the system more difficult to change in the future.

We enlist empirical evidence in showcasing some of the drawbacks of technical debt that plague the software development process, in particular increases in system complexity. As a takeaway, practitioners would do well to keep technical debt in check and avoid its unwanted consequences. In the future, we intend to expound upon this research by studying the nature of defective SATD files more closely.

Chapter 4

Comparing the Impact of Comment- Versus Metric-Based Technical Debt

4.1 Introduction

In the development process, developers are constantly striving to maximize efficiency by delivering ever more sophisticated software by ever earlier deadlines. Pressure to meet projected release dates and output quotas raises the stakes and tempts many developers to consider options with long-term consequences which would otherwise be shelved on account of "creating more work in the future" [21, 40]. Code smells such as god classes indicate faulty design principles and other factors that slow development and contribute to underlying technical debt [ref]. Analysts have invoked the *technical debt* metaphor time and again to draw parallels between the financial ramifications of assuming a loan, which will have to be repaid with interest, and the software development equivalent of writing code that temporarily serves the purpose or meets the deadline but incurs the effort of maintenance or repair at a later date [5]. From

the perspective of those who succumb to technical debt in handling time-sensitive deliveries, the looming expiration date of a temporary solution pales in comparison to the prospect of coming up empty-handed on the due date [24]. In the end, technical debt is a last-ditch insurance policy against underperformance stemming from time constraints.

Predictably, as technical debt has become a more popular, if more stigmatized, recourse at developers' disposal, numerous advances have been made in detecting it, some metric-based and others comment-based. The former includes Marinescu's [31] methodology, which detects *god class* code smells according to sets of rules and thresholds defined on various object-oriented metrics. The latter, advocated by Potdar and Shihab's [38] methodology in recent work, flags recurring source code comment patterns that correlate with incidence of *self-admitted technical debt* (SATD). Moreover, the nature of the comments that developers leave has allowed occurrences of SATD to be sub-categorized and analyzed accordingly.

Attitudes towards technical debt are beginning to shift, erring more on the side of caution, or even going so far as to assert that it is outright detrimental in terms of software maintenance and quality control [50, 43, 14, 40, 21]. Empirical evidence, meanwhile, has not kept pace, yet without it developers will be misinformed as to the costs and benefits of technical debt and unequipped to decide responsibly whether it should be assumed in a given scenario—not to mention lacking effective strategies for keeping it in check once assumed.

Our work closes this gap as we study 40 open-source projects that bring into focus the empirical links between both self-admitted technical debt and god classes and software quality. Our inquiry pursues: (i) whether god class and SATD-ridden files contribute more defects than files free of god classes and SATD, (ii) whether emerging defects can be traced to god and SATD changes and (iii) whether god- and SATD-related changes are associated with greater difficulty. As in the previous

chapter, amount of churn, quantity of affected files and modified modules and change entropy all factor into the change difficulty calculation. In the end we observed that: (i) no straightforward correlation exists between incidence of SATD or god files and incidence of defects, (ii) more future defects surfaced after performing god and SATD changes than non-god and non-SATD changes and (iii) god and SATD changes are more difficult to perform than non-god and non-SATD changes. Preliminarily, this concedes that the downsides of god classes and self-admitted technical debt are increases in future defect density and change difficulty.

4.2 Related Work

4.2.1 Identifying and Detecting Code Smells

Fowler and Beck [12] originated the term *code smell* to designate various indicators of object-oriented design flaws which can undermine software maintenance. Code smells respond to the internal and external properties of the system elements they monitor. Though manual code smell detection warns developers of potential vulnerabilities, Marinescu [30] observes that it is time-consuming, non-repeatable and non-scalable. Apart from this, the more familiar the software system is to a developer, the higher the risk of a subjective appraisal of its efficiencies and shortcomings, according to Mntyl [28, 29], and one important corollary of this is that a developer's chances of overlooking design flaws increase. In order to surmount these drawbacks, Marinescu recommends enlisting code metrics to detect system volatilities, and in this spirit, several implementations of this departure from manual detection have been devised [22, 31, 32, 33].

4.3 Approach

As we persevere in studying the interplay between self-admitted technical debt, god classes and software quality, the third must be made amenable to objective quantification [17, 19, 42]. The precedent in accomplishing this task has been to count the defects in SATD files and calculate the rate of future defect introduction among SATD changes, expressed as a percentage. Deferring to the technical debt metaphor and its concept of accruing "interest" to be paid in the long run, we also measure software quality in terms of SATD change difficulty, which we calculate as stipulated earlier. With these metrics standardized, we entertain the research questions that follow:

- **RQ1:** Do god files have more defects than non-god files?

- **RQ2:** Do god-related changes introduce future defects?

- **RQ3:** Are god-related changes more difficult than non-god changes?

We devised a suitable method of operation, visualized in Figure 11, in order to guide our inquiry into these questions. We initiated the process by mining the source code repositories and pulling source code files on a project-by-project basis (steps 1-2). Afterwards the source code files were parsed and comments extracted (step 3). We then implemented Potdar and Shihab's proposed source code comment patterns to identify all instances of self-admitted technical debt, counted defects file-wide and isolated defect-inducing changes by means of the SZZ algorithm (steps 4-5).

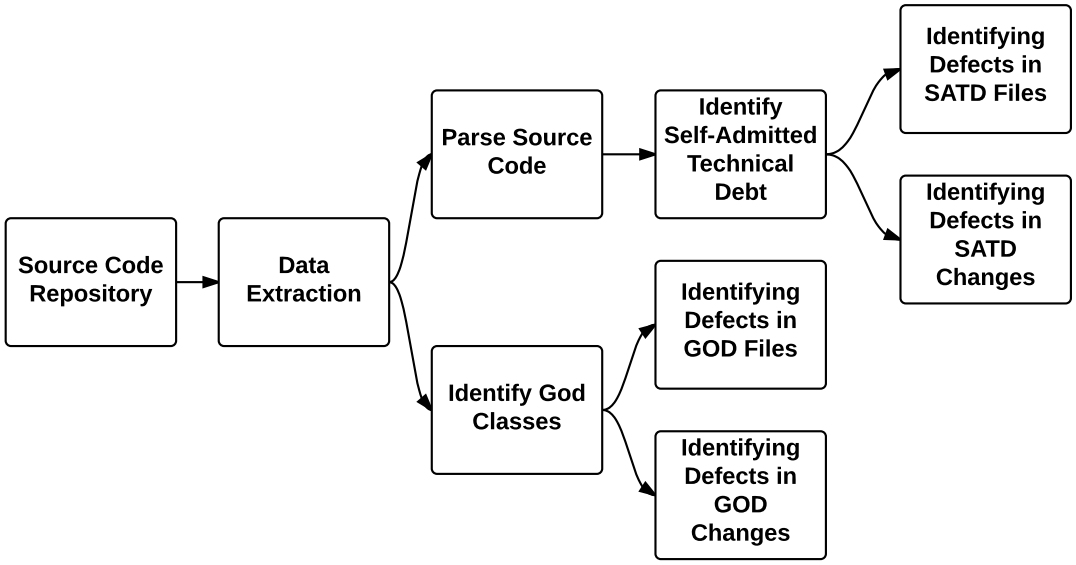

Figure 11: Approach overview.

4.3.1 Data Extraction

Drawing on 40 open-source software projects, we attempted to randomize our sample by domain and size while verifying that each project had amassed substantial source code comments from a wide array of contributors—a prerequisite of Potdar and Shihab's SATD detection technique. Projects "just getting off the ground" in terms of their development history were omitted, and all projects selected were available to software development researchers.

Given the extent to which our approach to locating self-admitted technical debt relies on source code comments, we downloaded the most recent versions of the relevant systems, filtered this input to generate source code and excluded all files lacking source code comments (e.g. CSS, XML, JSON). As for comments suspected of indicating no SATD, e.g. license comments, commented source code, Javadoc comments, etc., four filtering heuristics were deployed to remove them from the results.

Tables 5 and 6 showcase some key identifiers and statistics for each project, including: (i) which release was downloaded, (ii) the number of lines of code it contains,

(iii) the number of comment lines, (iv) a source code file count, (v) the number of committers in the project's development history and (vi) its commit count.

Table 5: Characteristics of the studied projects.

Project	Release	# Lines of Code	# Comment Lines	# Files	# Committers	# Commits
Apache OpenNLP	1.7.1	163,114	30,581	861	11	1,339
Apache Camel	2.18.0	1,626,040	430,818	17,046	333	25,461
Apache Habse	1.2.4	1,310,985	251,409	3,243	166	12,531
Apache Groovy	2.4.2	286,920	85,885	1,517	262	13,422
Apache Oltu	1.0.2	267,88	7,386	300	14	842
Apache Maven	3.3.9	150,724	34,830	1,540	81	10,370
Apache Karaf	5.4.0	155,675	32,906	1,467	86	5,666
Apache Hama	0.6.4	855,64	22,545	499	22	1,592
Apache Tomee	1.7.4	854,611	212,542	6,297	35	10,257
Apache Deltaspike	1.7.2	149,871	45,722	1,842	48	2,044
Apache Curator	2.10.0	124,077	18,458	525	66	1,813
Apache Calcite	1.10.0	448,820	110,410	1,702	102	2,180
Apache Poi	3.15	644,284	182,823	3,298	39	7,963
Apache Zeppelin	0.6.2	124,865	15,314	552	201	2,642
Apache Ant	1.9.7	343,010	109,150	1,927	62	13,425
Apache Stanbol	1.0.0	339,699	107,208	2,044	25	3,398
Apache Kafka	0.10.1	152,685	33,368	1,002	300	2,734
Apache Tika	1.13	142,786	38,984	992	54	3,209
Apache Felix	5.4.0	837,955	211,404	5,039	51	13,240
Apache Phoenix	4.9.0	396,054	69,326	1,620	59	1,748

Table 6: Characteristics of the studied projects.

Project	Release	# Lines of Code	# Comment Lines	# Files	# Committers	# Commits
Apache Wicket	7.5.9	548,923	191,279	4,830	80	19,574
Apache Aurora	0.16.0	317,551	70,774	1,284	105	3,639
Apache Ignite	1.6	1,498,260	443,157	7,361	117	17,933
Apache Helix	0.7.1	141,632	34,430	900	32	2,266
Apache Archiva	2.2.1	191,743	33,641	1,172	41	7,742
Apache Struts	2.5	316,495	83,911	2,307	65	4,634
Apache Derby	10.13.1.1	1,271,629	386,703	3,023	37	8,127
Apache Ambari	2.4.2	2,220,418	421,937	10,223	119	18,025
Apache Nifi	1.0.0	576,512	118,806	3,493	120	2,826
Apache Tiles	3.0.7	51,487	20,173	599	16	1,455
Apache Shiro	1.3.2	81,131	36,854	727	22	1,641
Apache Usergrid	2.1.0	605,286	114,999	2,619	110	10,621
Apache Nutch	2.3	104,214	27,478	843	37	2,217
Apache Zookeeper	3.4.9	196,008	38,867	814	21	1,468
Apache Mina	2.0.16	45,588	14,336	340	29	2,400
Apache Cxf	3.1.8	988,585	196,520	8,806	81	12,302
Apache CloudStack	4.9.0	1,423,346	207,036	6,424	412	29,931
Apache Oozie	4.3.0RC0	256,423	49,510	1,239	22	1,772
Apache Kylin	1.5.4.1	217,645	45,320	1,227	89	5,121
Apache Flink	1.1.2	791,670	195,738	4,154	325	9,513

4.3.2 Scanning Code and Extracting Comments

Having isolated the source code from all of the projects under investigation, we tasked a Python-based tool with extracting the source code comments. Beyond this, the tool reveals each comment's type (i.e., single-line or block comments), the name of its host file and its line number. The Count Lines of Code (CLOC) tool [6] double checks the Python-based tool, and provided that there is no discrepancy between the total number of lines of comments both yield, we have confidence in the accuracy of the tool we developed.

4.3.3 Filter Comments

The content of source code comments is as loosely determined as the formalities that developers observe in making them. Depending on what it is the commenter wishes to call attention to or notify future contributors of, the comments might situate the project within the circumstances of its development, make recommendations for revising the code at a later date, acknowledge who wrote which pieces or who made which fixes or confide that self-admitted technical debt has been assumed. Efforts should be made to decrease the volume of comments, especially when sifting through them in search of self-admitted technical debt confessions. For this reason, we make use of several filtering heuristics to focus our search query.

A Python-based tool reads data retrieved from parsed source code, initiates the filtering heuristics and stores the results in the database. The retrieved data specify each class or comment's starting and ending line numbers as well as Java syntax comment type (i.e., single-line, block or Javadoc). Once this information is acquired, the filtering heuristics are processed.

Self-admitted technical debt is seldom indicated in comments left prior to class declaration, e.g. license comments, among others, so we benefit from any mechanism

that identifies and omits such distractors without also omitting comments that incorporate Java IDE task annotations (i.e., "TODO:", "FIXME:" or "XXX:"). If a comment features any of these keywords, tasks related to the comment will be added to an IDE-generated list for ease of access. As for separating pre- and post-class declaration comments, the number of the line in which class is declared marks the crucial cutoff in that any preceding comments are targeted for removal.

Comment type matters insofar as cumbersome comments stitched together from single-line comment components (and not block comments) impede message interpretation for comments read one by one. A heuristic that pinpoints and collapses sequences of adjacent single-line comments into block comments overcomes the comment type issue.

Commented source code does not indicate self-admitted technical debt in our experience, but rather either code not used at all or code used exclusively for debugging. To eliminate this distractor, we use a regular expression to remove typical Java code structures, i.e., public, private, for, exception, etc.

Most IDEs auto-generate comments when creating a method, constructor, try catch, etc. Due to the nature of the auto-generation of these comments, their SATD content is nonexistent. The majority of Javadoc comments also fail to mention SATD, and those that do are annotated with at least one task (i.e., "TODO:", "FIXME:", "XXX:"). This criterion allows our heuristic to determine which Javadoc comments should be salvaged versus ignored, while no distinction is necessary for auto-generated comments. We designed a regular expression to apply the criterion by checking for task annotations before omitting the comment.

The procedure was conceived with the intention of factoring out the contribution of noise, which ultimately improves the quality of the comment dataset by reducing cases of SATD false positives and prioritizing the most applicable comments.

4.3.4 Identifying Self-Admitted Technical Debt

Our analysis hinges on locating self-admitted technical debt at two levels: (i) the file level and (ii) the change level.

SATD files: We emulated Potdar and Shihab [38] in identifying self-admitted technical debt on the basis of 62 different templates recurring in multiple projects at various frequencies. If a comment is found to match one or more of these templates—among them "*hack, fixme, is problematic, this isn't very solid, probably a bug, hope everything will work, fix this crap*"—it is taken to mean that SATD is present to some degree. The remaining patterns we designated for use in this inquiry are accessible online [1].

We subsequently abstract up from individual comment patterns in order to assign each file its proper label. *SATD files* are those that have SATD comments and are thus the ones we contemplate in addressing RQ1, whereas *non-SATD files* do not have comments matching any of the 62 tell-tale patterns nor do they concern us here.

SATD changes: At the change level, all the files touched by the same change are scrutinized for evidence of self-admitted technical debt. If any one of them is determined to be an SATD file, the whole change is classified as an SATD change. Alternatively, if none of the files touched by a change is an SATD file, the whole change falls into the non-SATD change category. SATD comments are shown to account for less than 6.10% and SATD files for somewhere between 1.37% and 25.03% of the respective totals for all systems in Table 7 and 8, where each system's proportions are listed separately for comparison.

[1] http://users.encs.concordia.ca/~eshihab/data/ICSME2014/data.zip

Table 7: Percentage of SATD and god of the analyzed projects.

Project	SATD Comments (%)	SATD Files (%)	God Files (%)
Apache OpenNLP	2.66	19.02	14.77
Apache Camel	0.95	3.53	12.95
Apache Habse	1.69	18.29	15.05
Apache Groovy	3.41	13.67	14.52
Apache Oltu	1.91	5.67	14.27
Apache Maven	3.76	10.65	13.92
Apache Karaf	2.40	7.44	14.58
Apache Hama	1.44	11.24	14.48
Apache Tomee	1.94	7.16	14.33
Apache Deltaspike	3.95	9.28	9.26
Apache Curator	0.99	5.34	15.32
Apache Calcite	1.67	12.97	14.54
Apache Poi	2.11	16.79	14.43
Apache Zeppelin	1.62	9.37	14.87
Apache Ant	2.23	20.56	14.60
Apache Stanbol	4.03	25.03	14.06
Apache Kafka	1.56	7.73	11.71
Apache Tika	3.29	19.28	14.40
Apache Felix	1.88	9.72	13.59
Apache Phoenix	3.34	13.81	13.47

Table 8: Percentage of SATD and god of the analyzed projects.

Project	SATD Comments (%)	SATD Files (%)	God Files (%)
Apache Wicket	0.89	5.29	11.52
Apache Aurora	3.97	14.27	14.91
Apache Ignite	0.26	3.12	13.50
Apache Helix	4.02	18.09	15.62
Apache Archiva	6.05	17.71	12.61
Apache Struts	1.85	8.07	12.39
Apache Derby	1.20	22.66	14.37
Apache Ambari	2.29	8.76	13.31
Apache Nifi	0.87	4.62	13.29
Apache Tiles	0.17	1.37	10.90
Apache Shiro	2.67	15.86	12.85
Apache Usergrid	2.20	11.20	12.33
Apache Nutch	2.17	12.25	15.60
Apache Zookeeper	1.71	10.31	13.82
Apache Mina	1.12	5.99	14.71
Apache Cxf	2.78	6.87	14.04
Apache CloudStack	1.98	10.42	14.04
Apache Oozie	1.63	9.73	15.81
Apache Kylin	1.64	8.06	15.38
Apache Flink	0.57	3.72	14.84

4.3.5 God Classes

God classes are classes that consolidate trivial class workloads and avoid task delegation for all but the simplest of operations, and are distinguishable on account of their high complexity, low inner-class cohesion and frequent foreign class data access [22]. Object-oriented design advocates a one-to-one correspondence between classes and responsibilities, which god classes violate by definition [22]. Due to their size and the extent to which they are tied to other classes, god classes can make it more difficult to understand the system [12] and are expected to be more susceptible to defects during system maintenance. The higher the incidence of defects, the more often changes will have to be performed and the bigger those changes will be, compounding maintenance over time [12, 22].

4.3.6 Identifying God Classes

To perform our analysis, we need to identify god classes at the same two levels as self-admitted technical debt: (i) the file level and (ii) the change level.

God files: To identify god classes, we followed the methodology outlined by Marinescu [31], who proposed an approach to specify and detect code smells, specifically *god classes*, based on metric-based heuristics where god classes are identified according to sets of rules and thresholds defined on various object-oriented metrics. The formula provided below in Figure 12 operates on three metrics—namely, weighted method count (WMC), tight class cohesion (TCC) and access to foreign data (ATFD)—and generates one of two outputs. If the output is 1, then the class to which the formula is applied is a god class; if 0, it is a non-god class.

God changes: To study the impact of god classes at the change level, we must first identify which classes are god classes and which are non-god classes. By analogy with the technique used to identify SATD and non-SATD changes, we consider any change containing at least one god file to be a god change and those containing no god files

to be non-god changes.

$$GodClass(C) = \begin{cases} 1 & (AFTD(C), HigherThan(1)) \wedge ((WMC(C), TopValues(25\%)) \vee \\ & (TCC, BottomValues(25\%))) \\ 0 & else \end{cases}$$

Figure 12: God Class Detection Equation

The equation in Figure 12 above describes how we detect a god class, where:

- **weighted method count (WMC)** is the sum of the statistical complexity of all methods in a class [4]. McCabe's cyclomatic complexity [34] is used as a complexity measure for all class methods.
- **tight class cohesion (TCC)** is the number of directly connected public methods in a class [3].
- **access to foreign data (ATFD)** is the number of external classes whose attributes are accessed either directly or indirectly (by accessor methods) [32].

4.3.7 Identifying Defects in SATD Files and SATD Changes

According to Sliwersky *et al.* [42], expressions denoting defect identifiers, e.g. "*fixed issue, bug ID, fix, defect, patch, crash, freeze, breaks, wrong, glitch, properly, proper*," ordinarily certify that an earlier mistake has been corrected when recorded in control system change logs. Other work has proposed comparable methodologies for tracking fault-inducing changes until repaired [17, 19, 42]. Next we pull each defect report from its corresponding issue tracking system, i.e., Bugzilla [2] or JIRA [3], and comb for all pertinent details.

[2]https://www.bugzilla.org
[3]https://www.atlassian.com/software/jira

Once the SATD files and SATD changes have been separated, we go about determining the number of file defects and identifying any defect-inducing changes in the same way foregoing research has [17, 19, 42].

Defects in files: A file defect count is a prerequisite to any defectiveness comparison between SATD and non-SATD files. With this in mind, we first view a file's history and extract all the changes that have touched it. The list this produces is then shortened as change log searches return only results consistent with keywords indicating corrective changes. Among these keywords we have: "*fixed issue #ID, bug ID, fix, defect, patch, crash, freeze, breaks, wrong, glitch, proper.*" We refer to the defect report in the event of a defect identification in order to be certain that the defect is attributable to a system under investigation. This extra step must be undertaken because in communities like Apache, one issue tracking system serves several products. At this point, issue IDs in the change logs that turn out to be false positives are discarded and the true positives point out the defect-fixing changes, i.e., we take the number of defects in a file to be equal to the number of corrective changes.

Defect-inducing changes: Along the same lines, we search the commit messages using regular expressions that convey defect fixes as a means of establishing whether a given change is corrective. Particularly, we search the change logs for keywords such as: "*fixed issue #ID, bug ID, fix, defect, patch, crash, freeze, breaks, wrong, glitch and proper.*" Likewise, defect identification numbers can clarify which defects the changes have in fact repaired.

After the corrective changes have been obtained, we find all prior changes that have touched the code utilizing the `blame` command. Our criterion for identifying the change that induced the defect is that it must be closest to yet still in advance of the defect report date such that it was the last change made before the defect emerged. In the absence of a defect report, we execute the SZZ [42] or approximate

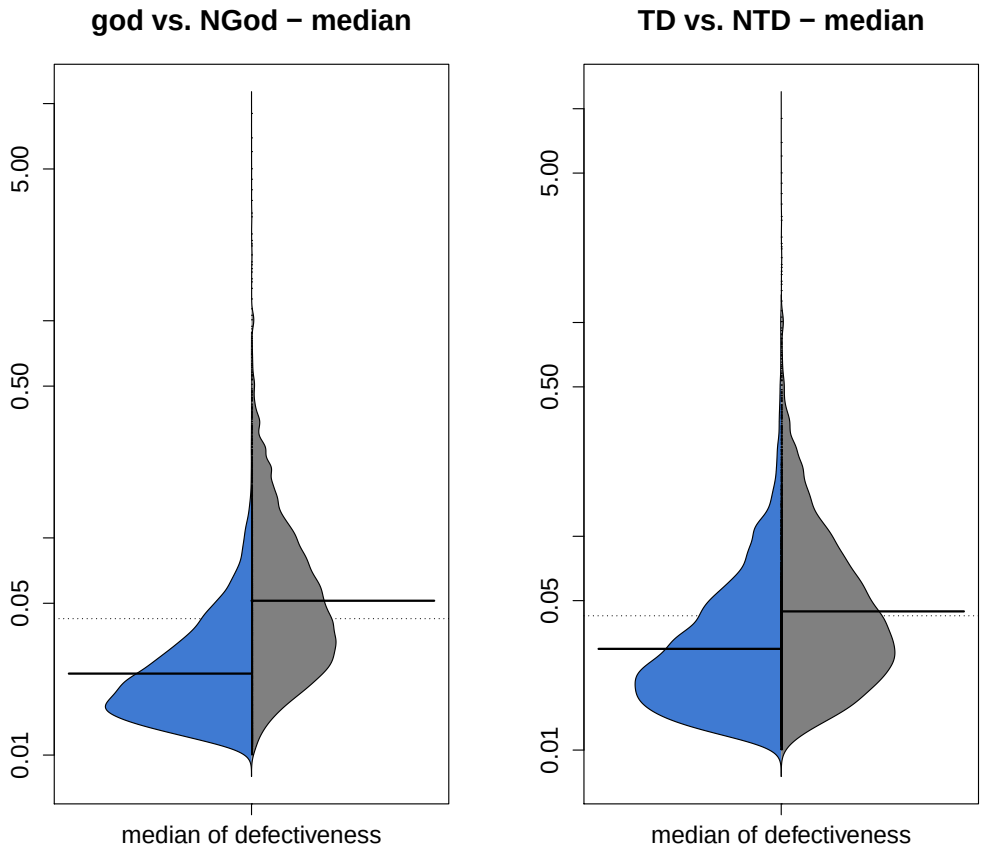

Figure 13: Percentage of defect-fixing changes for SATD and NSATD files.

(ASZZ) algorithm [17] and hold the last change before the corrective change to have induced the defect, as has been done in earlier work [17]. This algorithm currently represents the standard all alternative methods of identifying defect-inducing changes are weighed against.

4.4 Case Study Results

In this section, we present the empirical outcomes of our inquiry into the correlation between both self-admitted technical debt and god classes and software quality.

Statistics and results are listed for all individual projects, accompanied by cross-project comparisons.

Each of the three research questions is restated below, where we summarize its motivation, our approach in treating it and the conclusions we reached as a result.

[Sultan: stopped here]

RQ1: Do god files have more defects than non-god files?

Motivation: Reluctance to resort to code smells and technical debt suggests that most developers believe these adversely affect software quality, and what research has been conducted supports this conviction [50]. *[Sultan: I'm not sure how to incorporate god classes in this case since there has been a lot of work that studied their impact on software quality]* The potential drawbacks of SATD, meanwhile, have remained unexplored despite its research-affirmed ubiquity in software projects [38].

Researchers and developers should be better equipped to negotiate the long-term risks of SATD and have empirical evidence in hand that identifies concrete issues its use can bring about and raises SATD literacy within the community at large.

Approach: We compare god files versus non-god files and SATD files versus non-SATD files in terms of defect-proneness as we take up RQ1.

Comparing god and non-god files: The God Class Detection Equation proposed by [31] provides a way to identify god classes using object-oriented metrics. Files are fed to the equation and depending on the output are labeled either god or non-god files. We calculate the percentage of defect-fixing changes for each file in both categories based on $\frac{\left(\frac{\#\ of\ fixing\ changes}{total\ \#\ changes}\right)}{SLOC}$. We normalize our data since files can have different amounts of changes and apply a test designed to measure whether the differences between the categories are statistically significant.

The Mann-Whitney [27] test accomplishes this non-parametrically and is thus capable of handling non-normal distributions (of which our data's distribution is one

example), unlike parametric alternatives. A statistically significant difference returns a p-value such that $p <= 0.05$.

Comparing SATD and non-SATD files: We identify SATD files in accordance with the procedure detailed in section 3.3.3, labeling files containing any number of SATD comments as SATD files and all others non-SATD files. These two file categories (SATD and non-SATD) undergo calculations yielding the percentage of defect-fixing changes, $\frac{\left(\frac{\#\ of\ fixing\ changes}{total\ \#\ changes}\right)}{SLOC}$, which, unlike pure counts, standardizes the metric across files hosting different numbers of changes. Afterwards the defect distribution is plotted for SATD and non-SATD files and a test is performed to uncover any statistical trends.

We elected to conduct the non-parametric Mann-Whitney [27] test to decide whether a statistical difference exists between the SATD and non-SATD groups rather than a parametric substitute, which could not accommodate non-normal distribution (the distribution of our data turns out to be one such example). A statistically significant difference returns a p-value of at most 0.05 ($p <= 0.05$).

Results - Defects in god files vs. non-god files and SATD files vs. non-SATD files:

The beanplots in Figure 13 display the distribution of *median corrective change rates* for god files versus non-god files and SATD files versus non-SATD files *in each project*. A comparison of *distribution medians* indicates that the defectiveness rates for god and SATD files are lower than the corresponding rates for non-god and non-SATD files.

Figure 18 shows boxplots for the individual projects which compare the distribution of defectiveness rates for god and non-god files. We can see that the rate is higher for non-god files than god files within each project and that this difference is statistically significant such that $p < 0.0001$, which holds when all project medians are consolidated in the distribution in Figure 13.

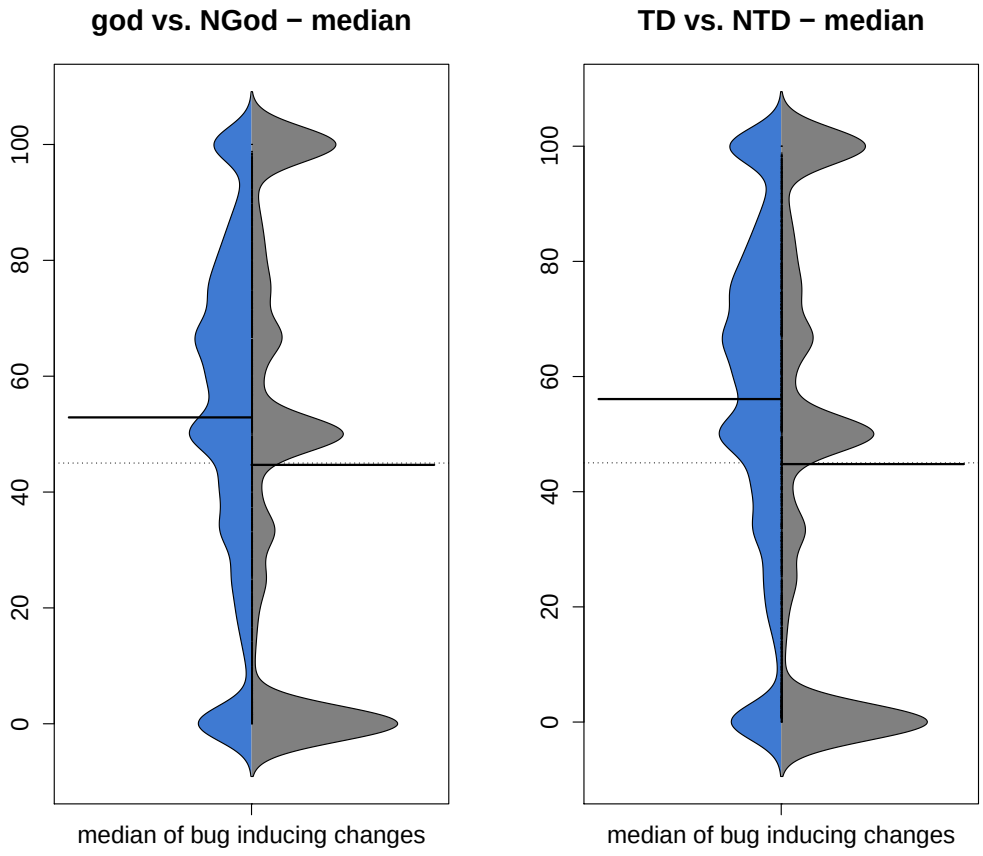

Figure 14: Percentage of defect-inducing changes for GOD vs. NGOD and SATD vs. NSATD.

Although Figure 13 indicates that non-SATD files have a higher median defectiveness rate than SATD files, this trend is not borne out in every individual project. In Figure 19, we observe that projects OpenNLP, Curator and Tiles constitute exceptions where the defectiveness rate is higher among SATD files.

RQ2: Do god-related changes introduce future defects?

Motivation: Having looked at how god and non-god and SATD and non-SATD compare at the file level, we turn our sights to the question of whether god and

SATD changes introduce future defects at a higher rate than non-god and non-SATD changes. Before, entire files were the objects of our analysis; now we require a more fine-grained analysis tailored to assess individual changes.

To investigate how change category relates to introduction of future defects and the duration of the "grace period" before god and SATD changes impact software quality, we should first determine to what extent god classes and SATD are predisposed to introduce future defects. If a god or SATD change causes a defect to be introduced in the change right after, then the delay is minimal and the impact on quality cannot be put off for very long. Our conjecture is that god and SATD changes are more complex overall and introduce defects at a higher rate than non-god and non-SATD changes.

Approach: We identify defect-inducing changes utilizing the SZZ algorithm [42] and subdivide the results it generates into four categories depending on whether the changes contain god classes or not (god versus non-god defect-inducing changes) and whether they contain SATD or not (SATD versus non-SATD defect-inducing changes).

Results:

God and non-god distributions of defect-inducing change rates in each project share a common y-axis in the first plot in Figure 14, as do SATD and non-SATD defect-inducing change rate distributions in the second. For each plot, the distribution median (i.e., the median of the individual project medians) is higher for the variable-positive groups (god changes and SATD changes) than for the variable-negative groups (non-god changes and non-SATD changes). This indicates that god

and SATD changes have more of a tendency to induce future defects than their non-god and non-SATD counterparts.

As a general rule, what is true of the distribution medians is also true of individual project medians, though exceptions exist—among them Apache Hbase, Apache Kafka, Apache Mina, Apache Shiro, Apache Oozie, Apache Flink, Apache Deltaspike and Apache curator in Figures 21 and 22—in which the variable-negative groups appeared to induce more future defects than the variable-positive groups. These isolated counterexamples, while in conflict with the trend observed in Figure 14, are compatible with our findings in chapter 3, where we report that SATD changes have less of a tendency to induce future defects.

RQ3: Are god-related changes more difficult than non-god changes?

Motivation: Up to this point, we have been concentrating on the interplay between both god classes and SATD and defects lowering software quality. As we know from previous research [31], god classes violate object-oriented design principles and have negative long-term implications for system maintainability. Likewise, if we recall the ramifications of the technical debt metaphor, we see that the short-term payoff should come at an increased cost later on in development. The deferred consequences of god classes and technical debt are measured in terms of increasing difficulty, which has yet to be fully examined after detecting god classes and introducing technical debt.

Verifying that god classes and SATD do increase change difficulty will better portray their implications for future changes and software projects and in the end enable developers to see the full picture when deciding whether or not to refactor god classes or introduce self-admitted technical debt.

Approach: We recognize god, non-god, SATD and non-SATD change categories and

compare the difficulty of effecting changes from each category. Four metrics are used to determine change difficulty: churn (i.e., the total number of modified lines), the number of modified directories, the number of modified files and the entropy of the change. For a slightly different purpose, Eick *et al.* [8] chose the first three to measure decay; Hassan [15] utilized the last metric to measure change complexity.

To measure the change churn, number of files and number of directories, we extract these metrics from the change log. To calculate churn for a whole change, we add up the number of lines inserted and deleted in each individual file touched by the change. To count the files and directories touched by a change, we extract the list of files from the change log. We consider a directory to be **ND** and a file to be **NF** when measuring the number of modified directories and files. Thus, if a change involves the modification of a file having the path "src/os/unix/ngx_alloc.h," then the directory is *src/os/unix* and the file is *ngx_alloc.h*.

We adopt Hassan's change complexity measure [15] to calculate change entropy, defined as $H(P) = -\sum_{k=1}^{n}(p_k * log_2 p_k)$, where k is the proportion file$_k$ is modified in a change and n is the number of files in the change. Entropy measures the distribution of a change across files. If we consider a change involving modification of three files *A, B* and *C*, for example, and take the number of lines modified to be 30, 1 and 1, respectively, the entropy equates to $0.40 = -\frac{30}{32}\log_2\frac{30}{32} - \frac{1}{32}\log_2\frac{1}{32} - \frac{1}{32}\log_2\frac{1}{32}$.

The maximum entropy $log_2 n$ has normalized the entropy formula presented above following Hassan [15] so that entropy can be compared for changes touching different numbers of files. The higher the normalized entropy, the more difficult it is to make the change.

Results:

In each one of Figures 15, 16, 17 and 18, a distribution is compiled for god and

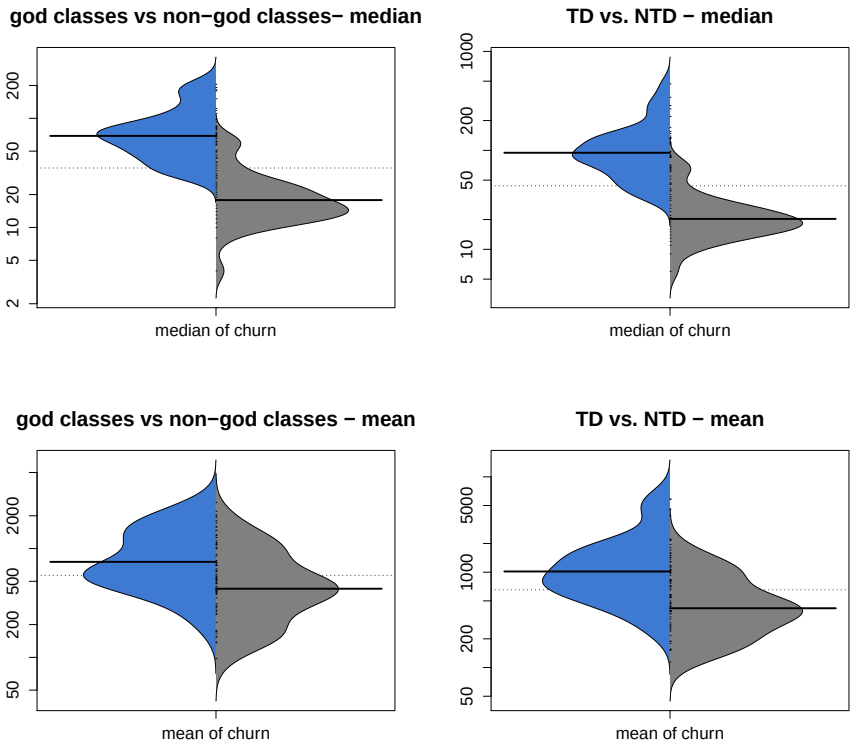

Figure 15: Total number of lines modified per change (SATD vs. NSATD).

non-god and SATD and non-SATD changes by plotting the median values obtained from a different difficulty measure for each project. Juxtaposition of the *distribution medians* demonstrates that regardless of the metric employed to quantify change difficulty (i.e., number of modified files, number of directories, amount of churn or entropy), god and SATD changes are consistently more difficult to perform both on median and on mean than non-god and non-SATD changes.

Upon inspection of the boxplots in Figures 23, 24, 25, 26, 27, 28, 29 and 30, we find that the distribution median and distribution mean inequalities remain unchanged for all projects for all difficulty measures except for OpenNLP for number of directories, where non-god and non-SATD change medians are still not greater than those of god and SATD changes but about equal.

In summary, we surmise that effecting god and SATD changes is more difficult

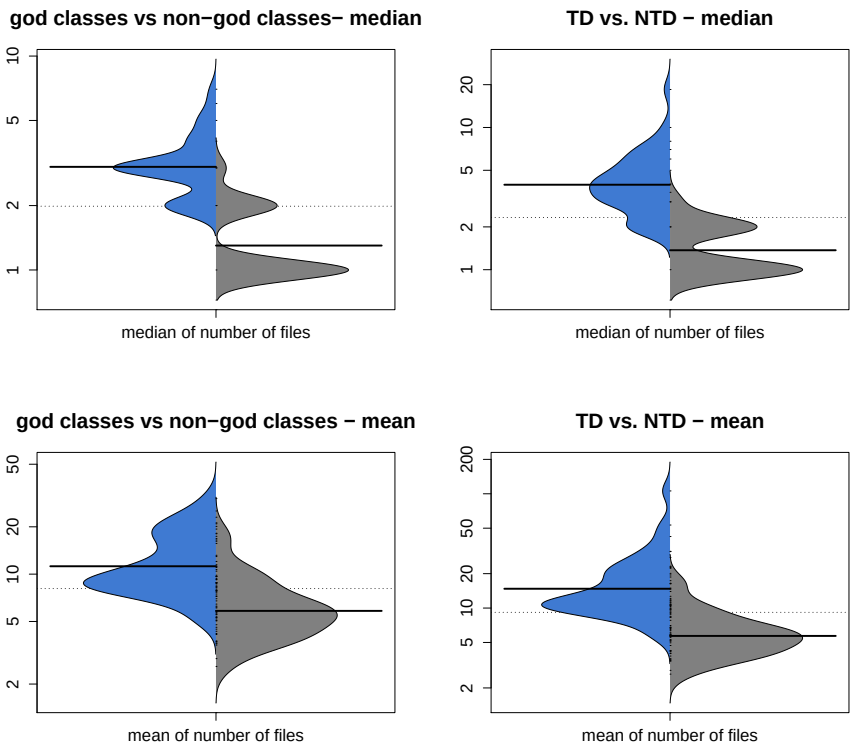

Figure 16: Total number of files modified per change (SATD vs. NSATD).

than effecting non-god and non-SATD changes by all difficulty measures (i.e., number of modified files, number of directories, amount of churn and entropy).

RQ4: Is there an overlap between comment- and metric-based technical debt?

[finish this research question]

Motivation: Thus far, we have compared the impact of technical debt identified by comment- and metric-based approaches on software quality. What remains outstanding now is the amount of overlap between the self-admitted technical debt files that the comment-based approach labels and the god files that the metric-based approach detects. Specifically, our objective is to calculate the percentage of files that contain

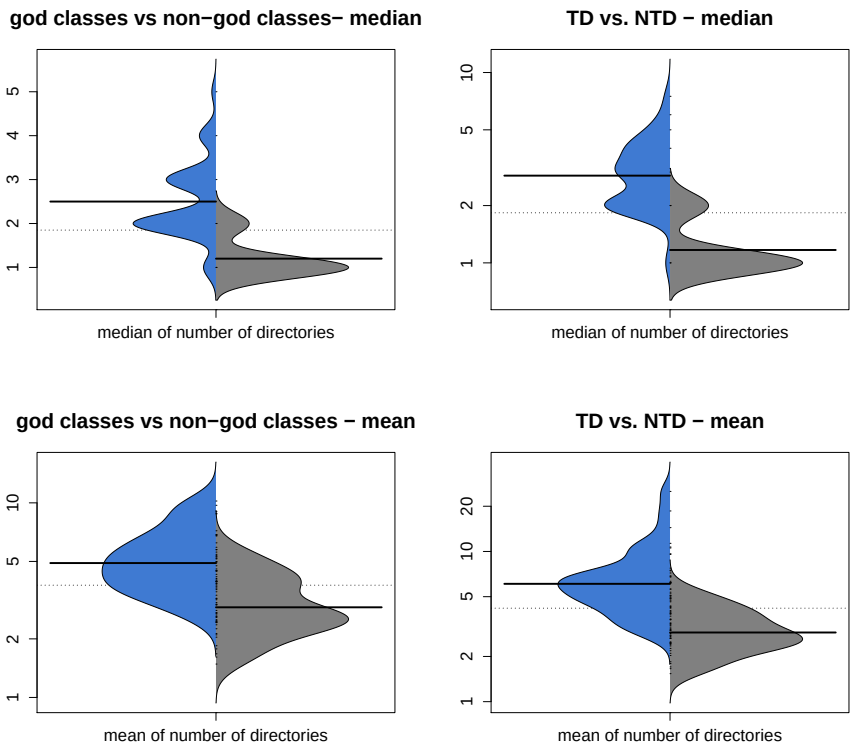

Figure 17: Total number of modified directories per SATD and NSATD change.

technical debt on both counts so that we can garner a better understanding of how comment-based technical debt complements metric-based technical debt.

Approach: To address RQ4, we take the list of SATD files generated by the comment-based approach and the list of god files generated by the metric-based approach and isolate the files that made both lists. We count how many of these files meet the criteria for both approaches and then divide by the total number of files in both lists. The result represents the share of comment-based technical debt that complements metric-based technical debt (overlap), expressed as a percentage.

Results: Table 9 displays the percentage overlap by project. We see that the values range from 11% to 34%. Our findings confirm that the comment-based approach, which uses source code comment patterns to detect technical debt, complements the metric-based approach, which relies on thresholds of object-oriented metrics. Despite

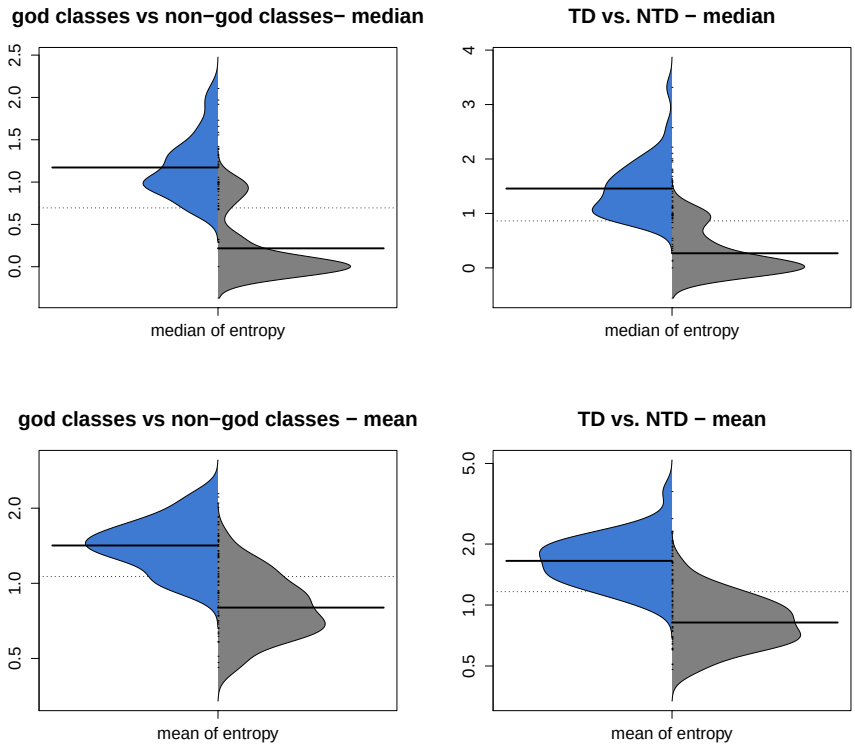

Figure 18: Distribution of the change across the SATD and NSATD files.

the considerable overlap, each approach identifies some additional sources of technical debt that the other fails to detect.

4.5 Threats to Validity

Threats to **internal validity** concern any factors that could have confounded our study results. Since developers might not think to declare the introduction of a technical debt in the first place, or remove the corresponding comment after eliminating a technical debt, one candidate is the use of source code comments. Every time the code and comment do not undergo a change simultaneously, the source code comments become a less and less accurate record. Despite this, Potdar and Shihab [38] found that in Eclipse code and comments were updated in tandem 97% of the time.

Project	Overlap (%)	Project	Overlap (%)
Apache OpenNLP	24.39	**Apache Wicket**	19.33
Apache Camel	16.57	**Apache Aurora**	26.44
Apache Habse	33.88	**Apache Ignite**	14.28
Apache Groovy	30.10	**Apache Helix**	25.45
Apache Oltu	16.68	**Apache Archiva**	30.33
Apache Maven	26.08	**Apache Struts**	24.74
Apache Karaf	23.65	**Apache Derby**	32.56
Apache Hama	29.51	**Apache Ambari**	25.59
Apache Tomee	21.61	**Apache Nifi**	18.92
Apache Deltaspike	20.46	**Apache Tiles**	11.37
Apache Curator	25.23	**Apache Shiro**	31.03
Apache Calcite	31.12	**Apache Usergrid**	26.90
Apache Poi	31.36	**Apache Nutch**	32.11
Apache Zeppelin	23.37	**Apache Zookeeper**	26.70
Apache Ant	28.86	**Apache Mina**	16.78
Apache Stanbol	32.95	**Apache Cxf**	22.25
Apache Kafka	20.64	**Apache CloudStack**	27.91
Apache Tika	33.56	**Apache Oozie**	21.88
Apache Felix	26.00	**Apache Kylin**	22.54
Apache Phoenix	30.99	**Apache Flink**	15.37

Table 9: Percentage of overlap between god and SATD files of the analyzed projects.

Another threat derives from comments intended to indicate SATD that do not correspond to any of the patterns Potdar and Shihab [38] compiled, which, owing to the flexibility of natural language, had to be analyzed manually. This technique is error-prone and somewhat subjective, in that developers could conceivably disagree as to which comments indicate SATD consistently. To mitigate these effects, we conducted manual inspections of all identified comments for each project in turn to certify that each contained one of the 62 patterns in [38]. While we chose to identify any change containing at least one SATD file as an SATD change, we could have reserved this label for changes containing only SATD files. In our view, it is better not to restrict SATD changes in this way because sometimes all it takes is one SATD file to change several other files touched by the same change.

Threats to **external validity** concern the generalizability of our results. In order to optimize this, we analyzed 40 large open-source systems and drew our data from the well-established, mature codebase of open-source software projects with well-commented source code. Programming language and domain varied from project to project. Moreover, we focused on the relationship between SATD and god classes only, which means that other, unadmitted technical debt or code smells might have been overlooked. Nonetheless, studying all technical debt is beyond the scope of this thesis.

4.6 Conclusion

The software development community stigmatizes technical debt, even though it still lacks adequate evidence to formalize its adverse effects on software quality. Accordingly, the empirical study we present in this chapter seeks to identify in what ways god classes and self-admitted technical debt detract from quality. God classes centralize the workload of trivial classes and perform tasks using their data in violation of the

object-oriented design principle stipulating one task per class. Self-admitted technical debt encompasses bugs that develop over time as a result of resorting to quick fixes that "do the job" for the deadline and defer associated costs which could jeopardize the code in the long run. We identify god classes by employing Marinescu's [31] object-oriented metric thresholds and self-admitted technical debt by locating source code comments that match the SATD indicator patterns in [38].

Three correlations allowed us to dissect the relationship between god classes and software quality: (i) whether god files have more defects than non-god files, (ii) whether god changes introduce future defects and (iii) whether god changes are more difficult to perform. Likewise, we examine the impact of self-admitted technical debt on quality by determining (i) whether SATD files have more defects than non-SATD files, (ii) whether SATD changes introduce future defects and (iii) whether SATD-related changes are more difficult to perform. We measured change difficulty for both god classes and self-admitted technical debt in terms of the amount of churn, numbers of files and modified modules in a change and entropy.

In the end, we found that (i) there is no dependable trend between god classes or self-admitted technical debt and defects: three exceptional projects revealed more corrective changes in SATD files than in non-SATD files; (ii) a trend did surface, however, in that both god changes and SATD changes are more correlated with the introduction of future defects and (iii) more difficult to perform than non-god and non-SATD changes.

Our study imparts that although god classes and technical debt may have detrimental effects, these imply nothing with respect to defects *per se*, but increase the number of defect-inducing changes and make the system more difficult to change in the future.

Chapter 5

Summary, Contributions and Future Work

5.1 Summary of Addressed Topics

Chapter 2 offers a synopsis of the latest research on technical debt, which has generated much interest in the software development community in recent years. Consequently, we find it to be an opportune time to pick up where this body of research left off and address some of the inquiries it has not yet delved into, whether not thoroughly enough or not at all. It should do much in the way of informing current debate in the field to determine whether the technical debt metaphor and developers' views of the practice hold up under further scrutiny.

Chapter 3 presents the impact of comment-based technical debt (self-admitted technical debt) on software quality. In this chapter, we analyze the source code comments of five well-commented open-source projects representing various domains and programming languages that have a large number of contributors. We find that: (i) files with SATD have more defects than files without SATD, (ii) SATD changes are associated with less future defects than non-SATD changes and (iii) SATD changes

are more difficult to execute.

Chapter 4 presents the effects of comment- versus metric-based technical debt on software quality. In this chapter, we analyze 40 open-source projects to understand how god classes and self-admitted technical debt influence software quality. We observe that: (i) neither the incidence of god nor self-admitted technical debt files is correlated with defects, (ii) future defects are introduced at a higher rate by god and SATD changes and (iii) the difficulty imposed on the system is greater for god and SATD changes.

5.2 Contributions

The major contributions of this thesis are as follows:

- A diachronic survey of the state of the art in technical debt detection: We supply a comprehensive account of the technical debt metaphor's inception and popularization. Specifically, we study the impact of metric- and comment-based approaches on software quality.

5.3 Future Work

We believe that our thesis advances the state of the art in measuring the impact of technical debt on software quality. Though our research clarifies the dynamics of this complex relationship, there are other dimensions of software quality that should be navigated in order to understand the full force of technical debt's impact.

5.3.1 Automating Technical Debt Management

Our findings lay the groundwork for creating a tool that would assist developers in understanding and mitigating the undesirable long-term consequences of incurring

technical debt. We have every reason to believe that a tool of this kind would facilitate detection and management of different varieties of technical debt while enhancing design practices, which would optimize the overall quality of the system and dovetail formerly discrete stages in the development process.

5.3.2 Diversifying Code Smell Representation

The scope of this thesis was not conducive to studying the overlap between self-admitted technical debt and all instantiations of code smells, yet developers would certainly benefit from further research that demonstrates how code smells besides god classes fit into the picture. The overlap between self-admitted technical debt and lazy class, black sheep, shotgun surgery, etc. could indicate that comment-based approaches to detecting technical debt are more or less reliable than the respective metric-based approaches.

5.3.3 Granularizing Technical Debt Classification

We have focused our attention so far on the impact of technical debt on software quality at the file and change levels. Naturally, a logical progression would be to accommodate the method level, as it would provide more granular insights into the implications of technical debt for quality as a result of increasing confidence in the organization of files into SATD and non-SATD categories.

Appendix A

Defects on The File-level

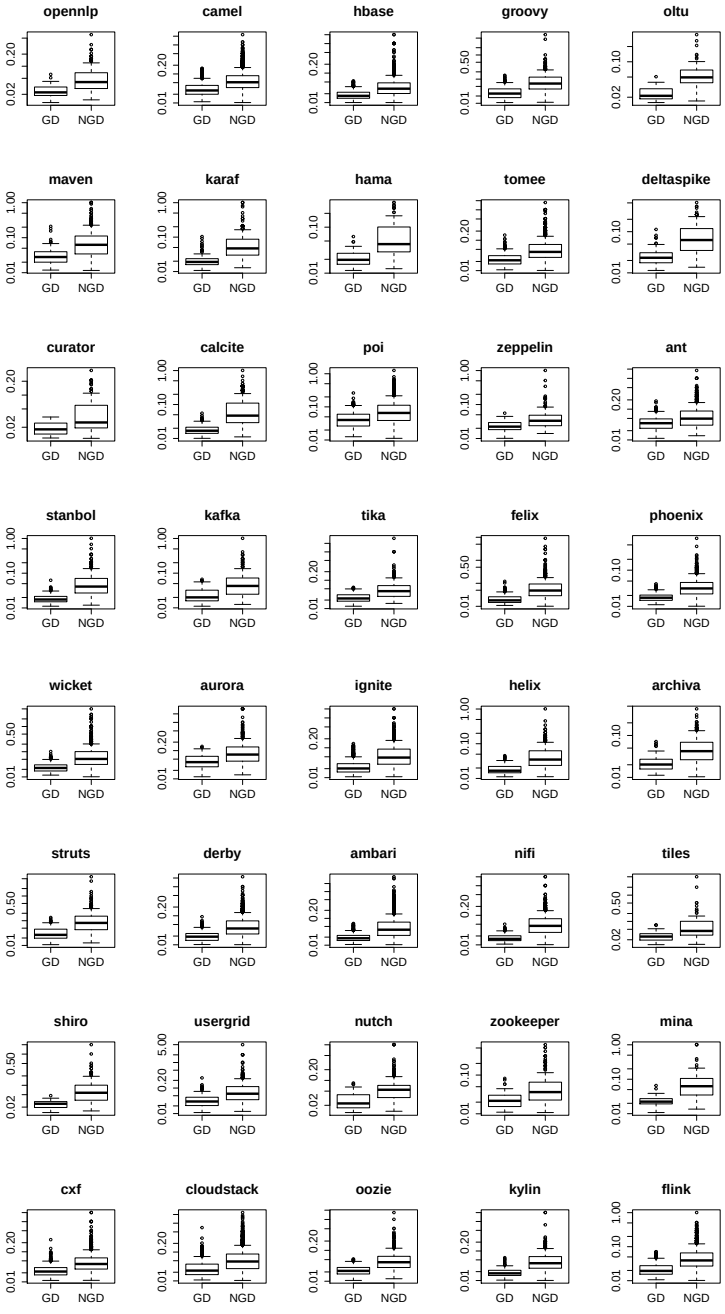

Figure 19: Percentage of defect fixing changes for GOD and NGOD files.

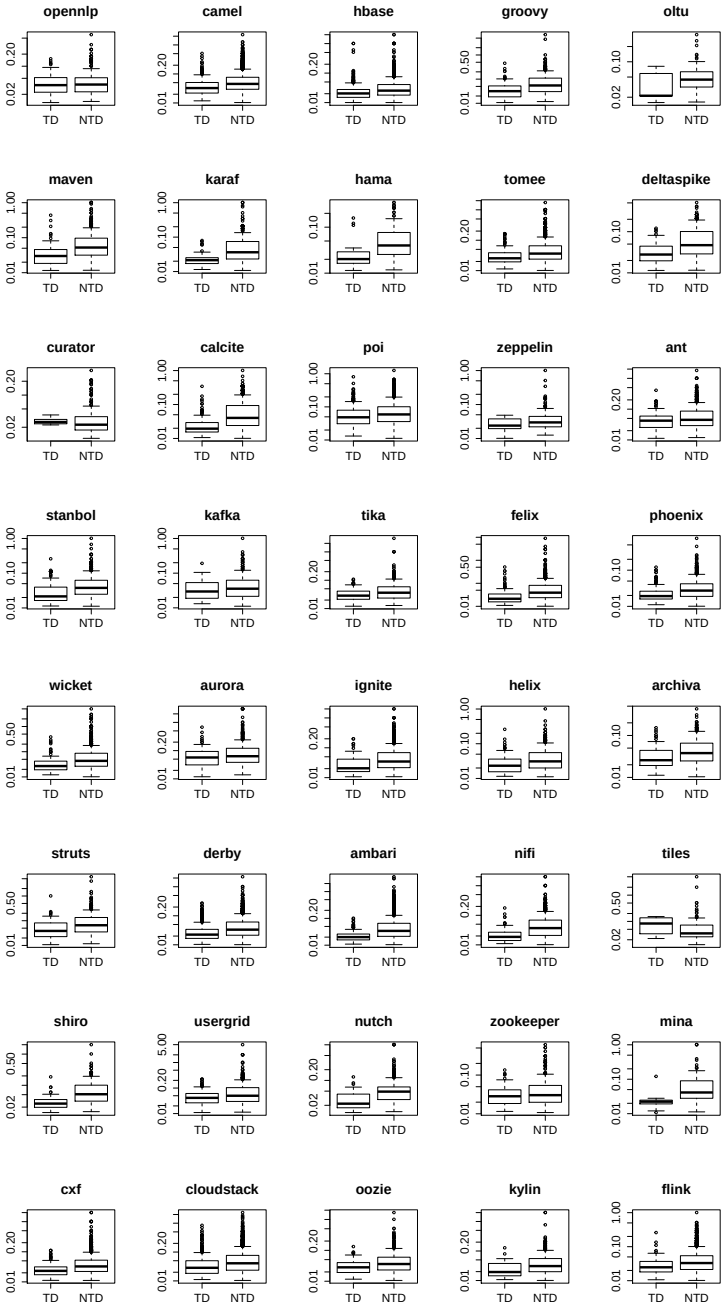

Figure 20: Percentage of defect fixing changes for TD and NTD files.

Appendix B

Defect Inducing Changes

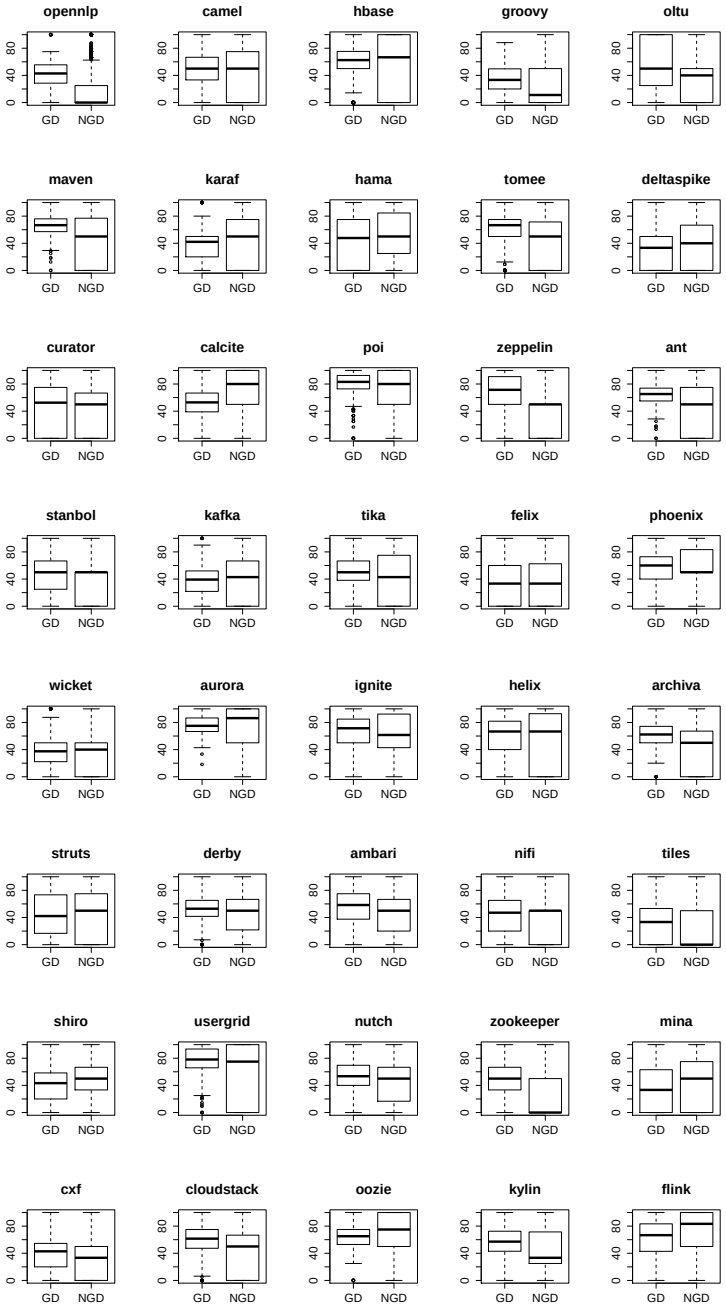

Figure 21: Percentage of defect inducing changes for GOD and NGOD files.

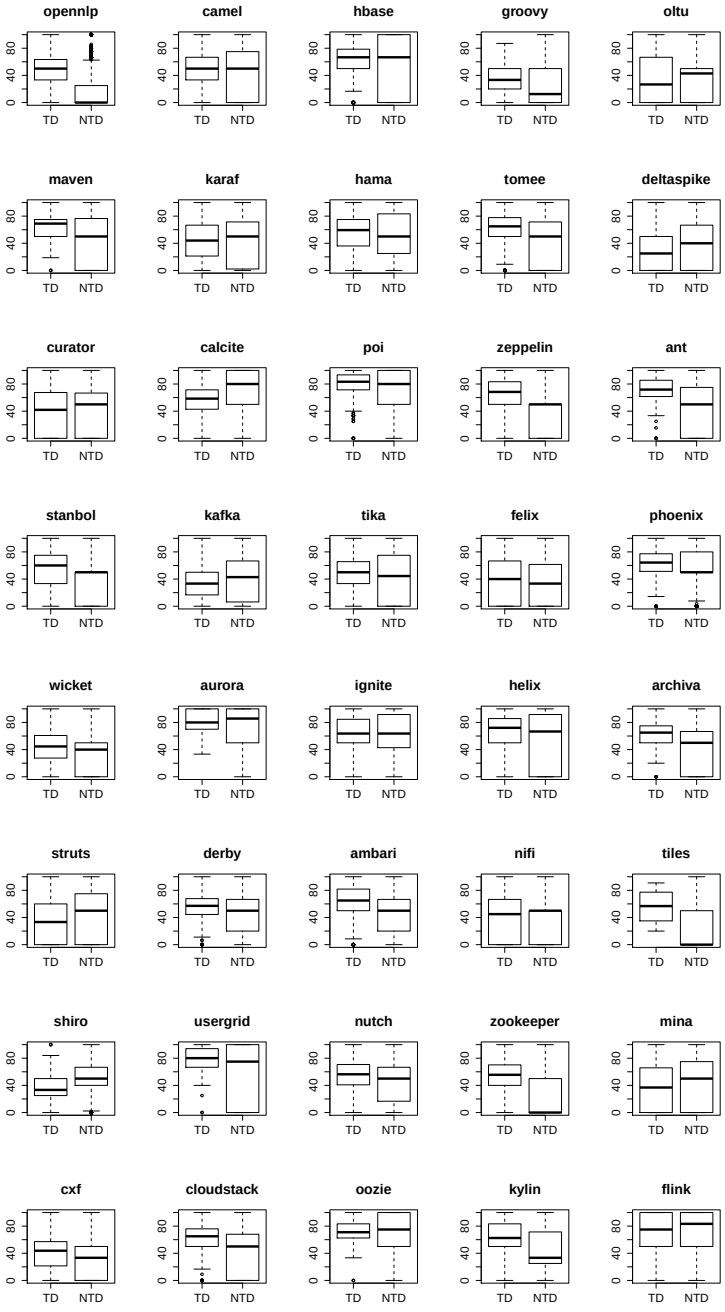

Figure 22: Percentage of defect inducing changes for TD and NTD files.

Appendix C

Complexity on The Change-level

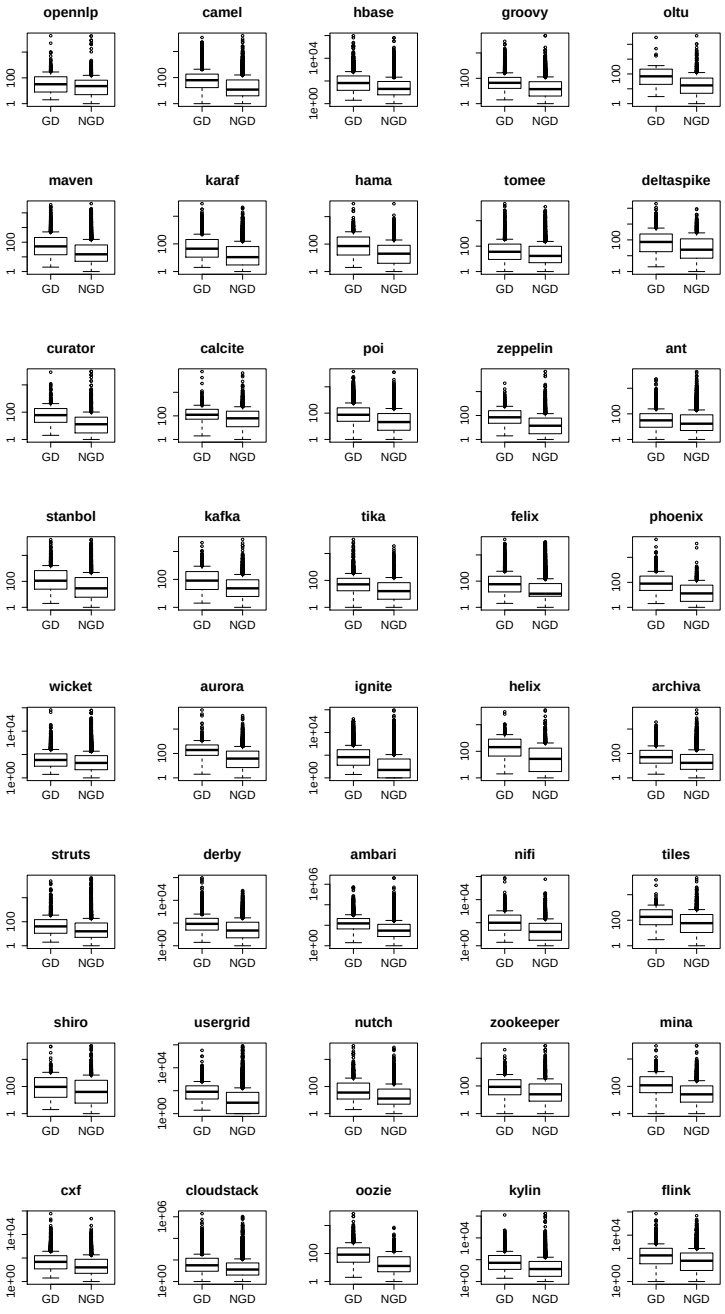

Figure 23: Total number of lines modified per change (GOD vs. NGOD).

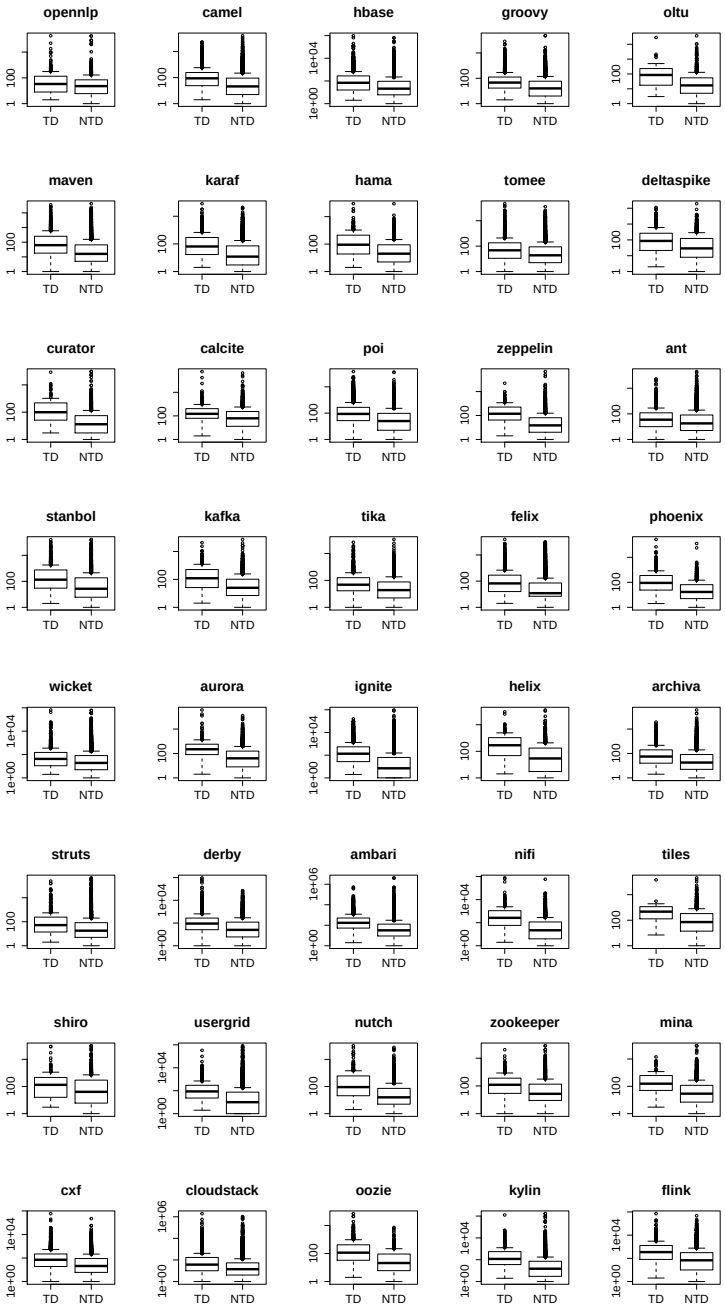

Figure 24: Total number of lines modified per change (TD vs. NTD).

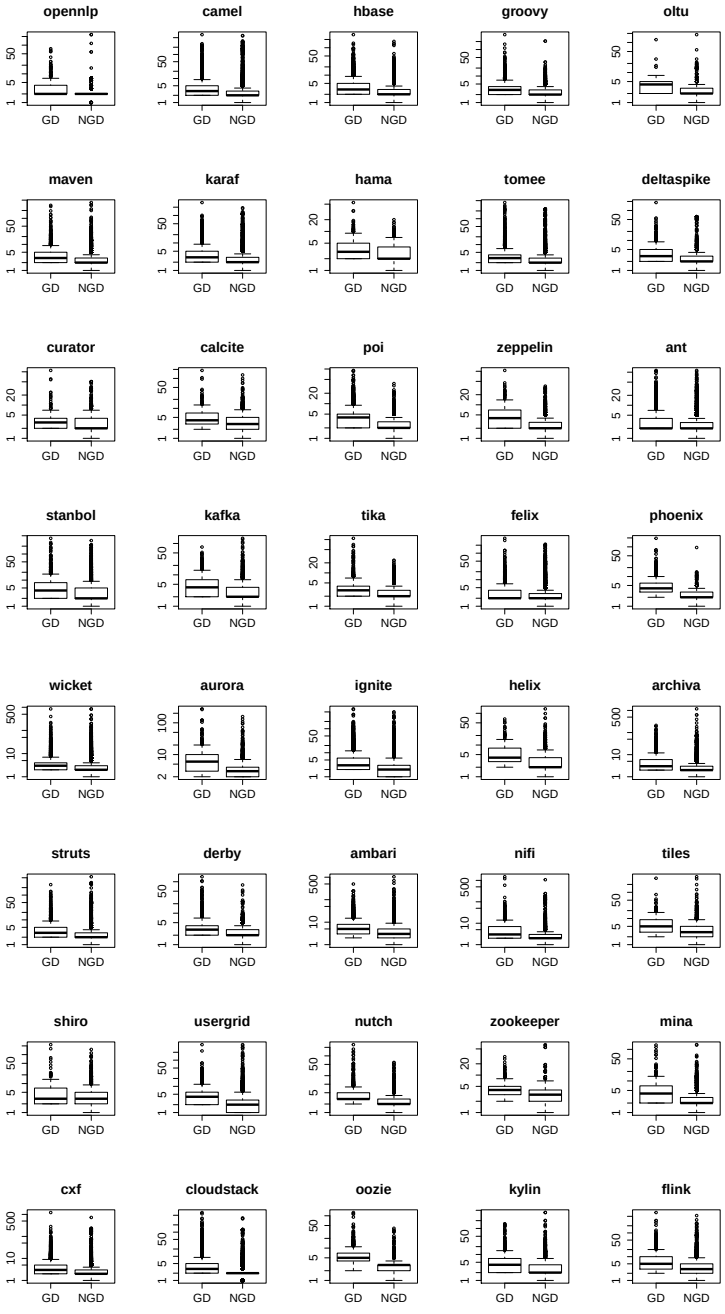

Figure 25: Total number of modified directories per GOD and NGOD change

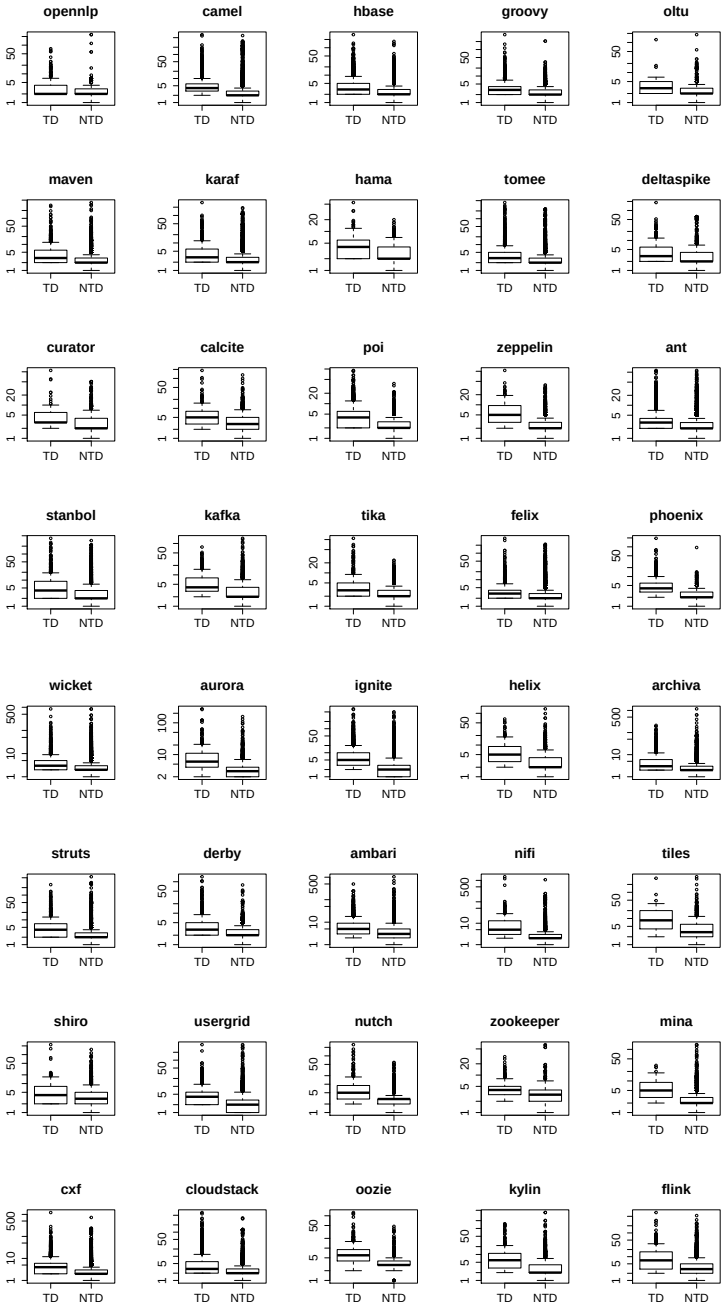

Figure 26: Total number of modified directories per SATD and NSATD change.

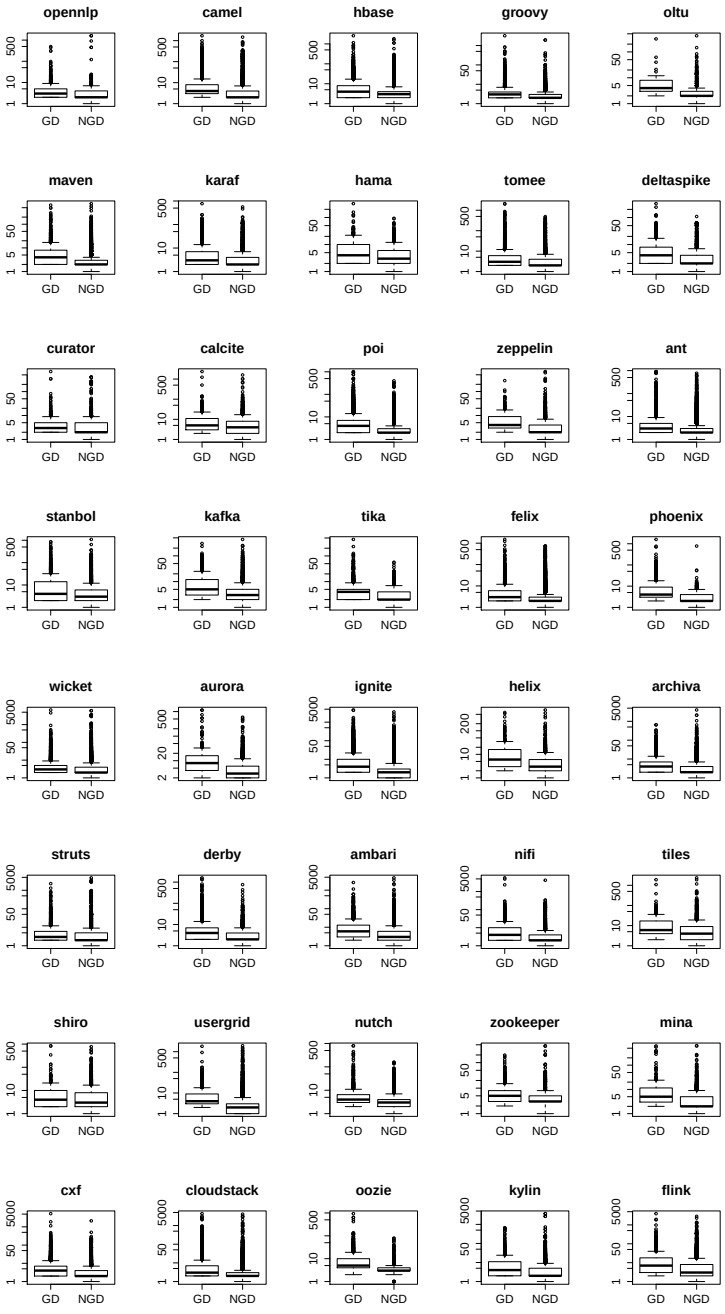

Figure 27: Total number of files modified per change (GOD vs. NGOD).

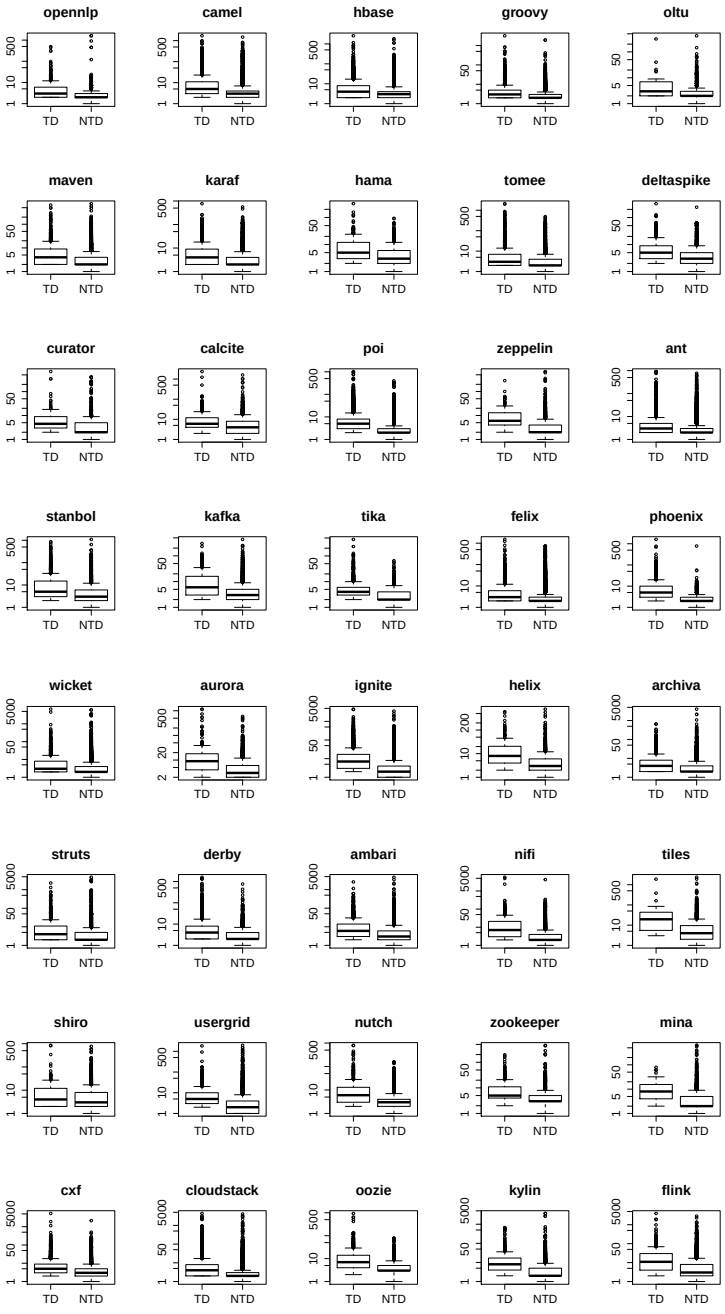

Figure 28: Total number of files modified per change (SATD vs. NSATD).

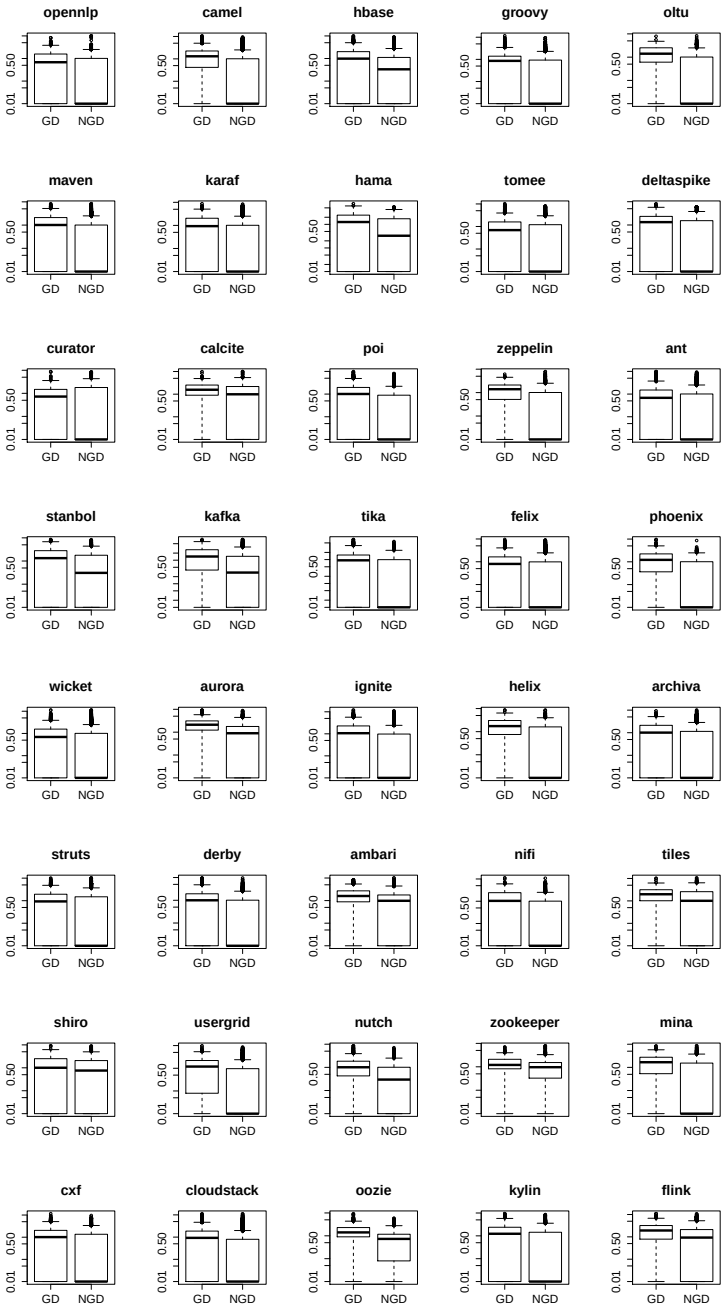

Figure 29: Total number of entropy modified per change (GOD vs. NGOD).

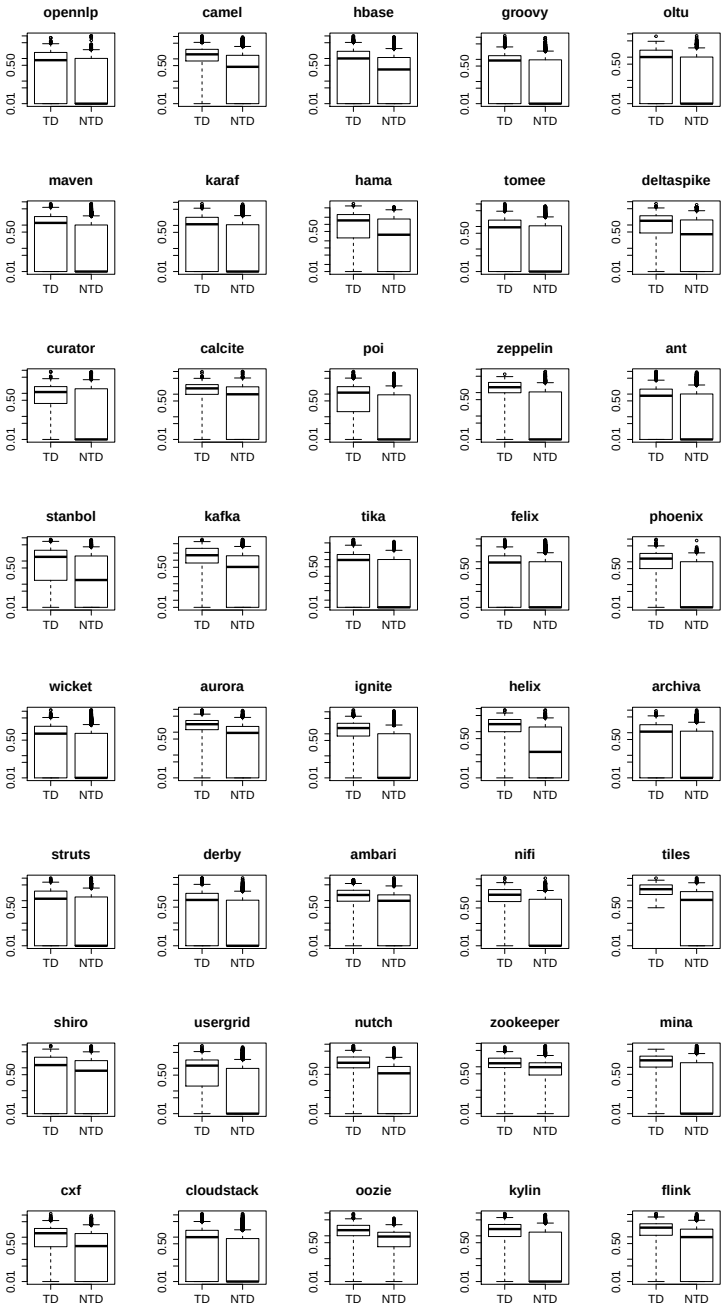

Figure 30: Total number of entropy modified per change (SATD vs. NSATD).

Bibliography

[1] N. S. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spinola. Towards an ontology of terms on technical debt. In *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*, pages 1–7. IEEE, 2014.

[2] G. Bavota and B. Russo. A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 315–326. ACM, 2016.

[3] J. M. Bieman and B.-K. Kang. Cohesion and reuse in an object-oriented system. *SIGSOFT Softw. Eng. Notes*, 20(SI):259–262, Aug. 1995.

[4] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, Jun 1994.

[5] W. Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1993.

[6] A. Danial. Cloc - count lines of code.

[7] A. De Lucia, M. Di Penta, and R. Oliveto. Improving source code lexicon via traceability and information retrieval. *IEEE Transactions on Software Engineering*, 37(2):205–227, 2011.

[8] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *Software Engineering, IEEE Transactions on*, 27(1):1–12, 2001.

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

[10] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mantyla. Code smell detection: Towards a machine learning-based approach. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 396–399. IEEE, 2013.

[11] M. Fowler. Technical debt quadrant, 2007.

[12] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code*. Component software series. Addison-Wesley, 1999.

[13] R. J. Grissom and J. J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition edition, 2005.

[14] Y. Guo, C. B. Seaman, R. Gomes, A. L. O. Cavalcanti, G. Tonin, F. Q. B. da Silva, A. L. M. Santos, and C. de Siebra. Tracking technical debt - an exploratory case study. In *ICSM*, pages 528–531, 2011.

[15] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.

[16] Y. Jiang, B. Cuki, T. Menzies, and N. Bartlow. Comparing design and code metrics for software quality prediction. In *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, PROMISE '08, pages 11–18, 2008.

[17] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Trans. Software Eng.*, 39(6):757–773, 2013.

[18] N. Khamis, R. Witte, and J. Rilling. Automatic quality assessment of source code comments: The javadocminer. In *Proceedings of the 15th International Conference on Applications of Natural Language to Information Systems*, pages 68–79, 2010.

[19] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, 34(2):181–196, 2008.

[20] P. Kruchten, R. L. Nord, and I. Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, Nov 2012.

[21] 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. *ACM SIGSOFT Software Engineering Notes*, 38(5):51–54, 2013.

[22] M. Lanza, S. Ducasse, and R. Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Berlin Heidelberg, 2007.

[23] D. J. Lawrie, H. Feild, and D. Binkley. Leveraged quality assessment using information retrieval techniques. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 149–158. IEEE, 2006.

[24] E. Lim, N. Taksande, and C. Seaman. A balancing act: what software practitioners have to say about technical debt. *Software, IEEE*, 29(6):22–27, 2012.

[25] E. Maldonado and E. Shihab. Detecting and quantifying different types of self-admitted technical debt. In *Proc. MTD*, pages 9–15. IEEE, 2015.

[26] H. Malik, I. Chowdhury, H.-M. Tsou, Z. M. Jiang, and A. E. Hassan. Understanding the rationale for updating a functions comment. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 167–176. IEEE, 2008.

[27] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[28] M. Mantyla, J. Vanhanen, and C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 381–384. IEEE, 2003.

[29] M. V. Mantyla, J. Vanhanen, and C. Lassenius. Bad smells-humans as code critics. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 399–408. IEEE, 2004.

[30] R. Marinescu. Detecting design flaws via metrics in object-oriented systems. In *Proceedings 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems. TOOLS 39*, pages 173–182, 2001.

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

[32] R. Marinescu. Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 701–704, Sept 2005.

[33] R. Marinescu. Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development*, 56(5):9:1–9:13, Sept 2012.

[34] T. J. McCabe. A complexity measure. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[35] S. McConnell. Technical debt, 2007.

[36] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 181–190, 2008.

[37] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering*, pages 284–292, 2005.

[38] A. Potdar and E. Shihab. An exploratory study on self-admitted technical debt. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 91–100. IEEE, 2014.

[39] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 432–441, 2013.

[40] C. Seaman, R. L. Nord, P. Kruchten, and I. Ozkaya. Technical debt: Beyond definition to understanding report on the sixth international workshop on managing technical debt. *ACM SIGSOFT Software Engineering Notes*, 40(2):32–34, 2015.

[41] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[42] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[43] R. O. Spínola, N. Zazworka, A. Vetrò, C. Seaman, and F. Shull. Investigating technical debt folklore: Shedding some light on technical debt opinion. In *Proceedings of the 4th International Workshop on Managing Technical Debt*, pages 1–7. IEEE Press, 2013.

[44] C. Sterling. *Managing Software Debt: Building for Inevitable Change*. Pearson Education, 2010.

[45] M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, and J. Singer. Todo or to bug: Exploring how task annotations play a role in the work practices of software developers. In *Proceedings of the 30th International Conference on Software Engineering*, pages 251–260, 2008.

[46] A. A. Takang, P. A. Grubb, and R. D. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4(3):143–167, 1996.

[47] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or bad comments? */. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP07)*, October 2007.

[48] L. Tan, Y. Zhou, and Y. Padioleau. aComment: Mining annotations from comments and code to detect interrupt-related concurrency bugs. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE11)*, May 2011.

[49] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @tComment: Testing javadoc comments to detect comment-code inconsistencies. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST)*, April 2012.

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

[51] N. Zazworka, R. O. Spínola, 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.

[52] T. Zimmermann, N. Nagappan, and A. Zeller. *Predicting Bugs from History*, chapter Predicting Bugs from History, pages 69–88. Springer, February 2008.