ON THE RELATIONSHIP BETWEEN SELF-ADMITTED TECHNICAL DEBT AND SOFTWARE QUALITY

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

April 2017

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Sultan Wehaibi**

Entitled: **On the Relationship Between Self-Admitted Technical Debt and Software Quality**

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. Tiberiu Popa _____ Chair

Dr. Juergen Rilling _____ Examiner

Dr. Weiyi Shang _____ Examiner

Dr. Emad Shihab _____ Supervisor

Approved _____

Chair of Department or Graduate Program Director

_____ 20 _____ _____

Dean

Faculty of Engineering and Computer Science

Abstract

On the Relationship Between Self-Admitted Technical Debt and Software Quality

Sultan Wehaibi

Developers 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, which has been referred to as "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. Traditionally, developers have relied on metric-based approaches, which use static analysis tools to identify technical debt based on thresholds defined on object-oriented metrics, e.g. code smells. Another technique, 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 how self-admitted technical debt and code smells (God Classes) relate to software quality.

Preliminarily, we examine the relationship between self-admitted technical debt and 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 (SATD); (ii) on the change level, the potential of SATD to introduce future defects and (iii) the complexity SATD changes impose on the system. The results of our study indicate that: (i) SATD files tend to have less defects than non-SATD files and (ii) SATD changes make the system less

susceptible to future defects than non-SATD changes do, though (iii) SATD changes are more difficult to execute.

Until the advent of SATD, god classes were used to detect technical debt, and though others have studied the impact of metric-based approaches on software quality, this work has been limited to a small number of systems. Therefore, we conduct an extensive investigation that compares the relationship between both approaches and software quality on a larger number of projects. We assess how code smells—in particular, god classes (metric-based approach)—and SATD (comment-based approach) are associated with software quality by determining: (i) whether god and SATD files have more defects than non-god and non-SATD files, (ii) whether god and SATD changes induce future defects at a higher rate than non-god and non-SATD changes, (iii) whether god and SATD changes are more difficult to perform than non-god and non-SATD changes and (iv) how much the metric- and comment-based approaches to technical debt file identification overlap. Our results indicate that: (i) neither god nor SATD files are correlated with defects, (ii) introduction of future defects is higher for god- and SATD-related changes, (iii) god- and SATD-related changes are more difficult to perform and (iv) the metric-comment technical debt file overlap ranges from 11% to 34%.

Overall, our study indicates that although technical debt—whether measured by the SATD or god classes—may have negative effects, these do not include file-level defects. Rather, the detriments of technical debt are its tendencies to introduce future defects at an elevated rate and to make the system more difficult to change in the future. In terms of detection methods, our work advocates implementing both the comment- and metric-based approaches to maximize the sources of technical debt identified.

Acknowledgments

The first and last of my thanks go to Allah for His abundant blessings of knowledge, health, patience, endurance, motivation and willingness to change for the better.

I am grateful to my supervisor, Dr. Emad Shihab, for his insightful comments and constructive feedback on the earlier drafts of this thesis. I also appreciate that he encouraged me to attend conferences that ultimately informed much of my research, as I had opportunities to discuss the state of the art with those on the forefront of innovation in the field. Emad, I cannot thank you enough for your tireless support and sharing your expertise with me throughout the last two years.

I am indebted to my thesis examiners, Dr. Weiyi Shang and Dr. Juergen Rilling, for taking the time to read my thesis and offering constructive revisions and valuable suggestions, which were incorporated into the final version. I also thank the other faculty members of the Department of Computer Science and Software Engineering, each of whom had a hand in lightening my journey and providing some moments of welcome levity.

I am fortunate to count myself among my colleagues, who are some of the brightest researchers I have had the pleasure of collaborating with: Rabe Abdalkareem,

Suhaib Mujahid, Davood Mazinanian, Jin Fu, Kundi Yao, Omid Sarbishei and everyone else who broadened my vision by offering their unique perspectives on my work.

Throughout this journey, I have enjoyed the love and support of many of my closest friends: John Meyer, Stéphane Leclerc, Connor Phoennik, Wael El-Dweesh, Haitham Al-Bisaily Osamah Al-Wabel, Ammar Al-Washali, Toshiki Hirao, and Michael Shu. I could never have made it this far in my academic career without a cast of people so invested in my future success.

To my mom, dad and siblings—I am sincerely grateful for the values you instilled in me which have served me well this far and will no doubt carry over into a career as accomplished and satisfying as yours. You have taught me to love and give always, neither expecting immediate returns nor returns at all. As my dad used to tell me, nobody owes you anything—life is what you make it.

"The future belongs to those who prepare for it today."

– Malcom X

Dedication

To my mom, my dad, and my brother Ahmed.

"Always think critically and be skeptical of what you hear."

– Sultan

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

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 [KNOF13] [SNKO15]. 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" [Cun92]. Prior work has shown that there are many different reasons why practitioners assume technical debt. These 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 and incentives to satisfy customer demands in a time-sensitive industry [LTS12] that still expects them to meet its software quality standards.

Most definitions of software quality recognize two subdivisions: external quality and internal quality. In one such definition, Fitzpatrick [Fit96] characterizes software quality as "the extent to which an industry-defined set of desirable features are incorporated into a product so as to enhance its lifetime performance." How the features

that end users enjoy conform to their individual preferences determines external quality. Internal quality, by contrast, is a composite evaluation of the features developers have built into the code on the production side. While industries differ on how they weight the specific features in the development process (quality factors), there is consensus that maintainability goes a long way in determining internal quality.

Studies over the years have proposed different approaches to measure technical debt, which has been found to impact (internal) quality. Zazworka *et al.* [ZSSS11], for instance, recommend combining automated technical debt detection tools with manual detection strategies. For his part, Marinescu [Mar04] has proposed a technique to detect code smells, specifically god classes, based on sets of thresholds defined on various object-oriented metrics.

God classes typically exhibit high complexity and low inner-class cohesion and access foreign class data at a higher rate than the trivial classes whose workloads they consolidate. Moreover, god classes withhold tasks that would otherwise be delegated elsewhere [LDM07]. Their size and many dependencies make the system harder to comprehend and more defect-prone [FB99].

More recently, a study by Potdar and Shihab [PS14] introduced a new way to identify self-admitted technical debt (SATD) through source code comments. SATD is technical debt that developers themselves report through source code comments. Prior work [MS15] has demonstrated that accrual of SATD is commonplace in software projects, where implementing comment-based approaches can identify different types of technical debt (e.g. design, defect, and requirement debt), just as metric-based approaches detect technical debt using static analysis tools. Today, these two approaches are the state of the art in measuring technical debt.

Intuition and general belief concur that technical debt negatively impacts software maintenance and overall quality [ZSSS11, SZV+13, GSG+11, SNKO15, KNOF13]. However, to the best of our knowledge, there is little empirical evidence as to how

SATD and metric-based technical debt are related to 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 and metric-based technical debt.

Since there is no prior work on the relationship between SATD and quality, we conduct a preliminary study in Chapter 3 of this thesis to 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 measure 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 studied the relationship between SATD and software quality, we then expand our study to include another measure of technical debt (metric-based technical debt). Such a study is important for providing researchers and practitioners with different observations of technical debt; comparing the new approach to the traditional approach in terms of how they relate to software quality and advancing the state of the art in understanding and mitigating technical debt.

Therefore, in Chapter 4 of this thesis, we compare the SATD and metric-based approaches across 40 open-source systems in order to validate our preliminary findings with a larger data set. Specifically, 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. In addition, we measure (iv) the overlap between metric- and comment-based technical debt files.

1.1 Thesis Overview

Chapter 2: Literature Review: 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, including its reputation among software developers and the drawbacks it has been found to entail.

Chapter 3: Examining the Relationship Between Self-Admitted Technical Debt and Software Quality: We preliminarily examine how self-admitted technical debt relates to 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, based on our findings, we conclude that self-admitted technical debt adversely affects system maintenance by increasing change complexity but is dissociated from defects.

Chapter 4: Comparing the Relationship Between Comment- Versus Metric-Based Technical Debt and Software Quality: We conduct a wide-ranging study on 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. In the interest of comparing the approaches, we also determine: (iv) to what extent the metric- and comment-based approaches identify the same sources of technical debt. We conclude that: (i) neither god nor SATD files tend to have more defects than non-god and non-SATD files, (ii) god- and SATD-related changes tend to induce a greater number of future defects and (iii) god- and SATD-related changes are more difficult to perform than non-god and non-SATD changes. Thus, god classes and self-admitted technical debt are detrimental insofar as they increase future defects and change complexity. We also found that (iv) the metric- and comment-based approaches complement each other at a rate of 11% to 34%.

1.2 Thesis Contributions

The major contributions of this thesis are as follows:

- Empirically examine the relationship between self-admitted technical debt and software quality.
- Enhance knowledge of the technical debt phenomenon by presenting a large-scale empirical study that compares the SATD (comment-based) and non-SATD (metric-based) approaches.
- Provide evidence that technical debt tends to induce more future defects and increase system complexity.

2

Background & Related Work

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, 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.

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

In section 2.1, we provide background information on technical debt generally; section 2.2 provides a cursory summary of related work divided into six subsections:

Leveraging Source Code and Static Analysis Tools, Source Code Comments, Leveraging Source Code Comments (Self-Admitted Technical Debt), Technical Debt, Software Quality and Identifying and Detecting Code Smells.

2.1 Background

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, which includes more and more as its usage gains traction.

Ward Cunningham [Cun92] 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 metaphor's financial familiarity that makes it effective in explaining how temporarily functional portions of code eventually become unsustainable.

Steve McConnell [McC07] popularized the metaphor in his taxonomy, as did Martin Fowler [Fow07] in devising the four quadrants outlined in Figure 1. Due to the effectiveness of these two methods of explaining technical debt to the software engineering

community, we devote the two subsections that follow to examining each in turn.

2.1.1 Intentionally vs. Unintentionally Incurred Technical Debt

Steve McConnell recognizes "intentionally incurred" (Type I) and "unintentionally incurred" (Type II) as the two principle classifications of technical debt [McC07]. The latter comprises error-prone design techniques and poorly written code by an inexperienced programmer, among others. Unintentionally incurred technical 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 [McC07].

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.

2.1.2 The Technical Debt Quadrant

Advocating an alternative interpretation of the metaphor, Fowler [Fow07] 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 [Fow07].

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 [Fow07].

Figure 1 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.

2.1.3 Additional 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

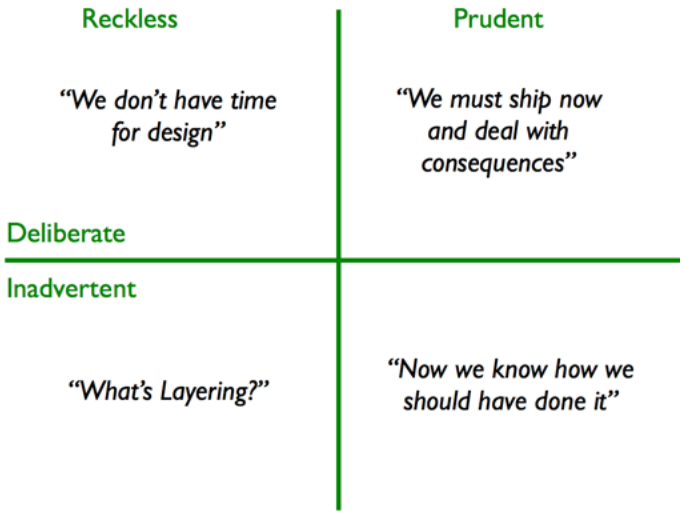

Figure 1: Technical Debt Quadrant

and patchwork repairs that "kick the can down the road" and put off the effort of isolating a solution viable in the long term. Concepts falling under this umbrella include test, requirement, documentation and generalized software debt [Ste10].

Broadening the metaphor to cover too many varieties of debt, however, might ultimately lessen its effectiveness, as Kruchten *et al.* point out [KNO12]. 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. This occurs 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.* [SZV+13] 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 [SZV+13].

Alves *et al.* [ARC+14] 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 Related Work

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 fails to realize how quickly it accumulates. Static analysis tools can 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.* [ZSSS11] 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, mitigate the effect of technical debt 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.* [FZMM13] 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.* [ZSSS11] 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, counting only the items that all teams recognized as technical debt results in an underestimation of the actual technical debt load and for this reason 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 Source Code Comments

A number of studies examined the usefulness/quality of comments and showed that comments are valuable for program understanding and software maintenance [TGM96, TYKZ07, LFB06]. For example, Storey *et al.* [SRB+08] explored how task annotations in source code help developers manage personal and team tasks. Takang *et al.* [TGM96] 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.* [KWR10] assessed the quality of source code documentation based on an analysis of the quality of language and consistency between source code and its comments. Other work, by Tan *et al.*, has proposed several approaches to identify code-comment inconsistencies. The first, called @iComment, detects lock- and call-related inconsistencies [TYKZ07]. The second approach, @aComment, detects synchronization inconsistencies related to interrupt context [TZP11]. 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 [TMTL12].

Other studies have examined the co-evolution of comment updates as well as the reasons behind them. Fluri *et al.* [FWG07] 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.* [MCT+08] 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.* [DLDPO11] 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 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 [PS14], 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 [MS15], 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 [PS14, MS15] since we use the comment patterns they produced to detect SATD. However, we differ from these studies in that we examine the relationship between SATD and software quality.

2.2.3 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 using static source code analysis tools, new improvements are constantly proposed, debated and adopted for use alongside older, "tried and tested" methodologies. One such improvement, from Potdar and Shihab [PS14], enlists source code comments in isolating technical debt, in which the 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 study capturing the state of the art in self-admitted technical debt identification, Potdar and Shihab [PS14] 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, e.g. "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 work of Potdar and Shihab [PS14], Bavota and Russo [BR16] examined 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.

2.2.4 The Relationship Between Technical Debt and Software Quality

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.* [ZSV+13] 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.* [ZSSS11] 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.* [GSG+11] analyzed how and to what extent technical debt affects software projects by tracking a single delayed task in a software project throughout its lifecycle.

Our work differs from foregoing research by Zazworka *et al.* [ZSSS11, ZSV+13] 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 consequences of SATD and, in particular, those pertaining to 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. [ZNZ08]). Several studies have examined the metrics that best indicate software defects, including design and code [JCMB08], code churn [NB05] and process metrics [MPS08, RD13].

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 [SZZ05a]. Kim *et al.* [KWZ08] used identifiers in added and deleted source code and the words in change logs to identify changes as defect-prone or not. Similarly, Kamei [KSA+13] *et al.* 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 hypothesized correlation between SATD and software defects.

2.2.6 Identifying and Detecting Code Smells

Fowler and Beck [FB99] originated the term *code smell* to designate various indicators of object-oriented design flaws that 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 [Mar01] 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 [MVL03, MVL04], 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 alternative to manual detection have been devised [LDM07, Mar04, Mar05, Mar12].

3

Examining the Relationship Between Self-Admitted Technical Debt and 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 date that would be ideal if product quality alone were taken into account. Such situations are all too common and in many cases force developers to take shortcuts [KNOF13] [SNKO15]. 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" [Cun92]. 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 [LTS12].

More recently, a study by Potdar and Shihab [PS14] 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 [MS15] 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 technical debt, which many developers resort to in a time crunch, negatively impacts software maintenance and overall quality [ZSSS11, SZV+13, GSG+11, SNKO15, KNOF13]. However, to the best of our knowledge, there is no empirical study that examines the relationship between SATD and 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 fewer 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, defects are not one of them, but making the system more difficult to change in the future is.

3.2 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 [KSA+13, KWZ08, ŚZZ05b]. 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 the level of each analyzed project (step 2). Next, we parse the source code and extract

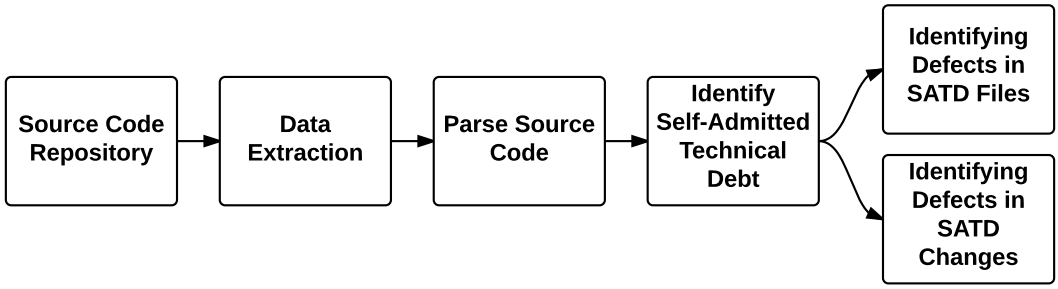

Figure 2: Comment-based (SATD) approach overview.

comments from the source code of the analyzed systems (step 3). At this point, we apply the comment patterns proposed by Potdar and Shihab [PS14] 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).

3.2.1 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 release 45 for Chromium, 2.7.1 for Hadoop, 2.3 for Spark, 2.2.2 for Cassandra and 8.0.27 for Tomcat, as shown in Table 1. 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 our analysis relies on.

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

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.

3.2.2 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 [Dan]. 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.

3.2.3 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 [PS14], 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 the 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

3.2.4 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.* [ŚZZ05b] 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 [KSA+13, KWZ08, ŚZZ05b]. 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 [KSA+13, KWZ08, ŚZZ05b].

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

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

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 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` 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 [KSA+13], 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 [ŚZZ05b] or approximate (ASZZ) algorithm [KSA+13] and is to date the state of the art in identifying defect-inducing changes.

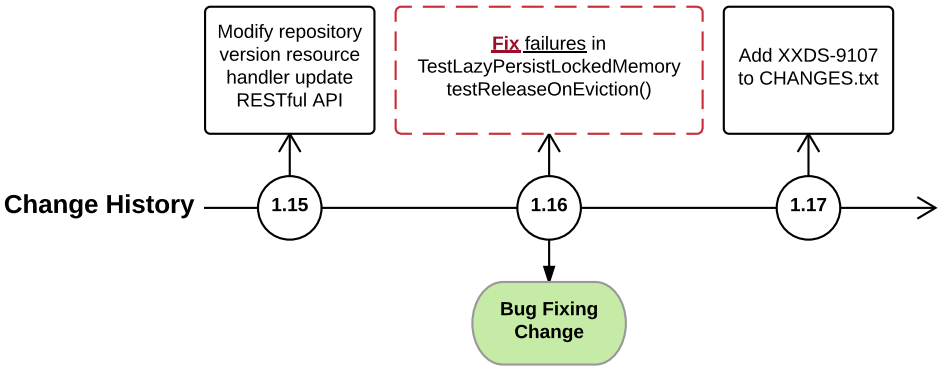

Figure 3: Indicating a bug-fixing change.

3.2.5 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 [MW47]. 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.

3.3 Case Study Results

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

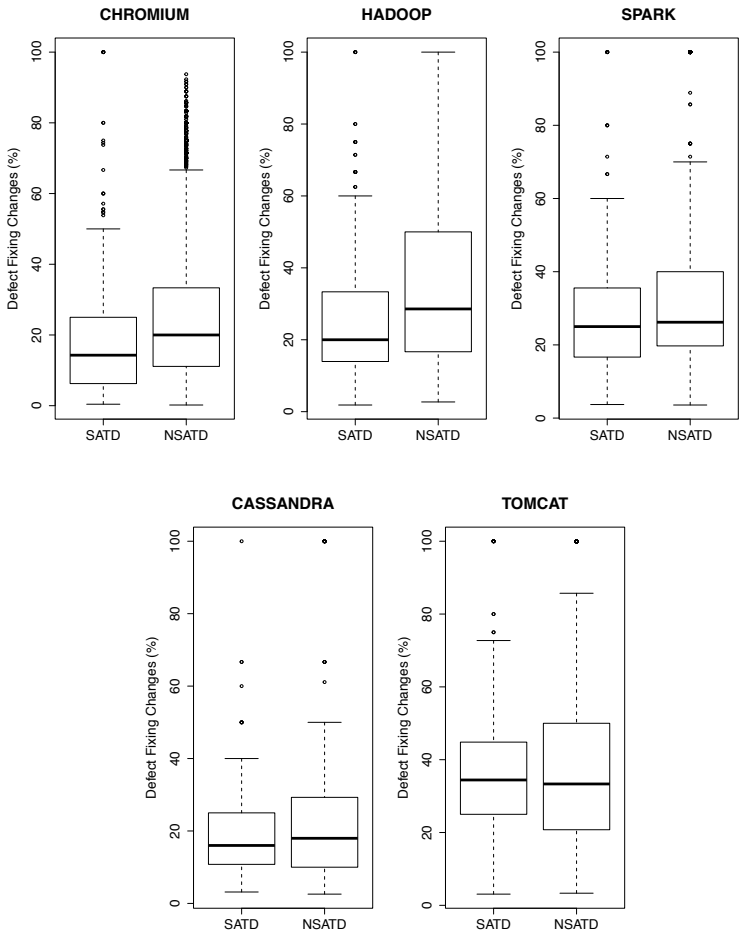

Figure 4: Percentage of defect-fixing changes for SATD and NSATD files.

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: Researchers have studied technical debt and shown that, consistent with long-standing intuition, it negatively impacts software quality [ZSSS11]. However, this research has neglected SATD, which is prevalent in software projects according to past research [PS14]. 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 3.2.3. 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

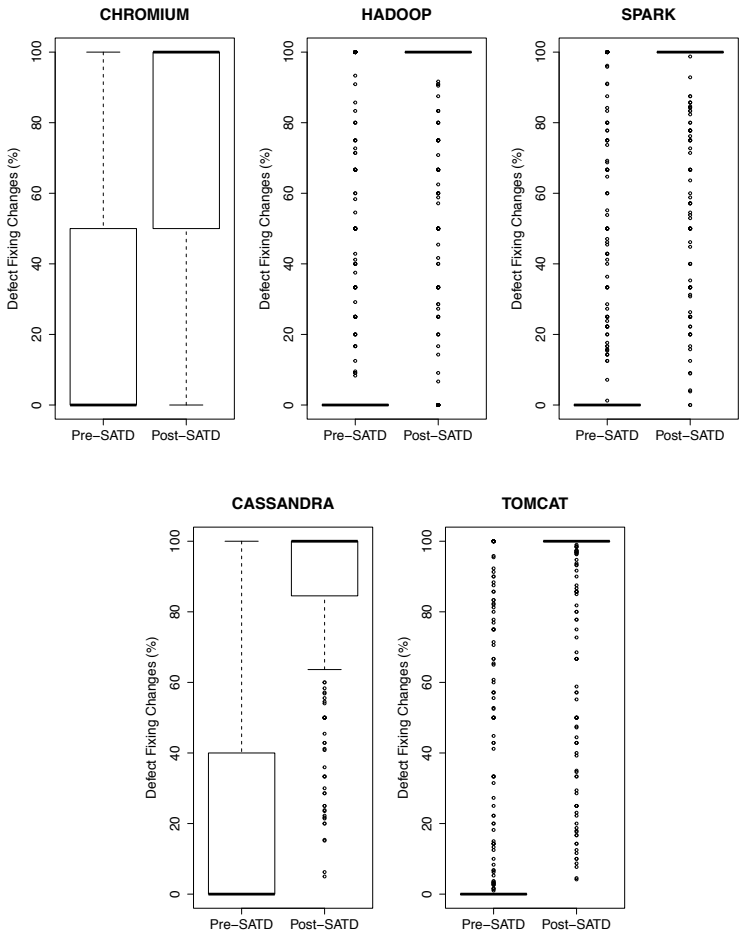

Figure 5: Percentage of defect-fixing changes for pre-SATD and post-SATD.

perform statistical tests to compare the differences.

We perform the Mann-Whitney [MW47] test to determine if a statistical difference exists between the categories and Cliff's delta [GK05a] 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 difference statistically significant if the Mann-Whitney test generates a p-value such that $p \leq 0.05$. 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 $d \geq 0.474$ [GK05b].

RQ1 - Interim Summary

There is no clear relationship between defects and SATD.

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.

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 (M)	0.704 (L)
Hadoop	0.562 (L)	0.137 (N)
Spark	0.221 (S)	0.463 (M)
Cassandra	0.400 (M)	0.283 (S)
Tomcat	0.094 (N)	0.763 (L)

Our findings 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 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 Cliff's delta effect size 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

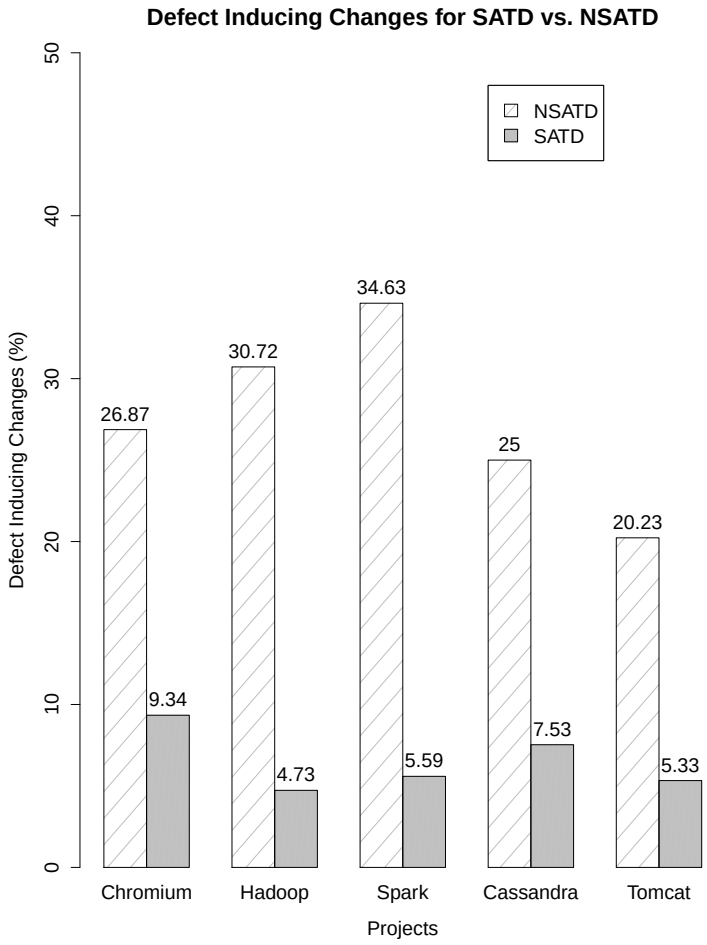

Figure 6: Percentage of defect-inducing changes with SATD and NSATD.

analysis here is more fine-grained and tailored to assess individual changes.

Studying the propensity of SATD changes to introduce future defects is important since it tells us how SATD and non-SATD changes compare in terms of future introduction of defects and how quickly the impact of SATD on quality can be felt. For example, if SATD changes introduce defects in the very next change, this tells us that the impact of SATD is felt almost immediately. Our conjecture is that SATD changes tend to introduce more defects.

Approach: To address RQ2, we applied the SZZ algorithm [ŚZZ05b] in order to detect defect-inducing changes. Then, we sorted the results into two categories: SATD

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

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

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.

RQ2 - Interim Summary

SATD changes are associated with less future defects than non-SATD changes.

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: 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.* [EGK+01] measure software decay. The change entropy metric is motivated by the work of Hassan [Has09], in which entropy is used to measure change complexity.

To measure the change churn, number of files and number of directories, we use 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 files and directories, we refer to a file as **NF** and a directory as **ND**. Hence, if a change involves the modification of a file having the path "net/base/registry_controlled_domains/effective_tld_names.cc," then the file is *effective_tld_names.cc* and the directory is *base/registry_controlled_domains*.

To measure the entropy of a change, we use the change complexity measure proposed by Hassan [Has09]. 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. 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 [Has09], 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

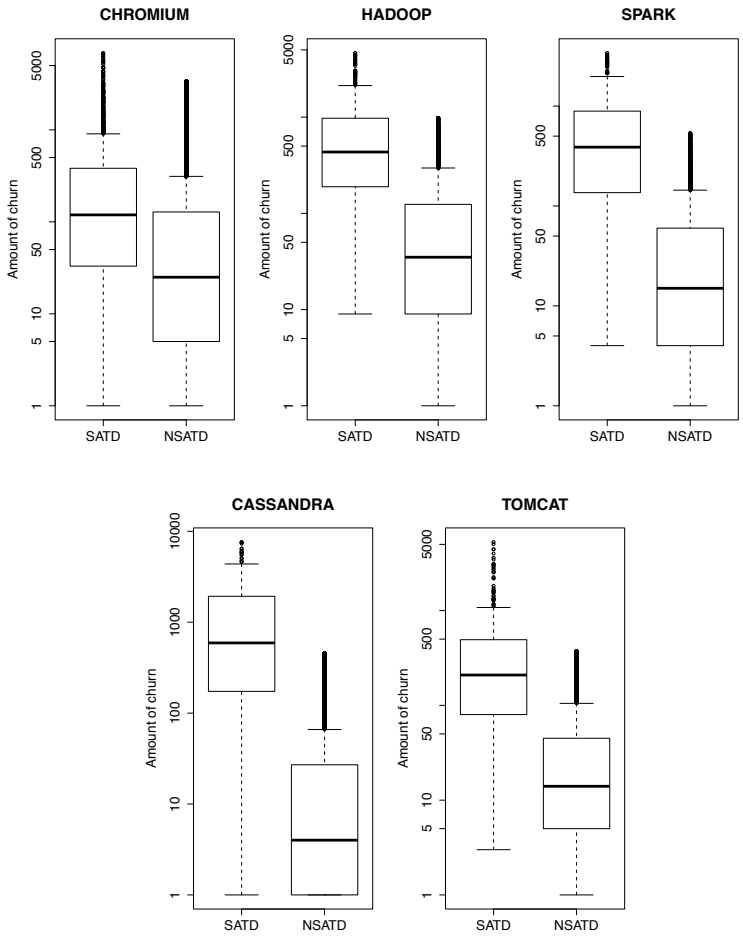

Figure 7: Total number of lines modified per change (SATD vs. NSATD).

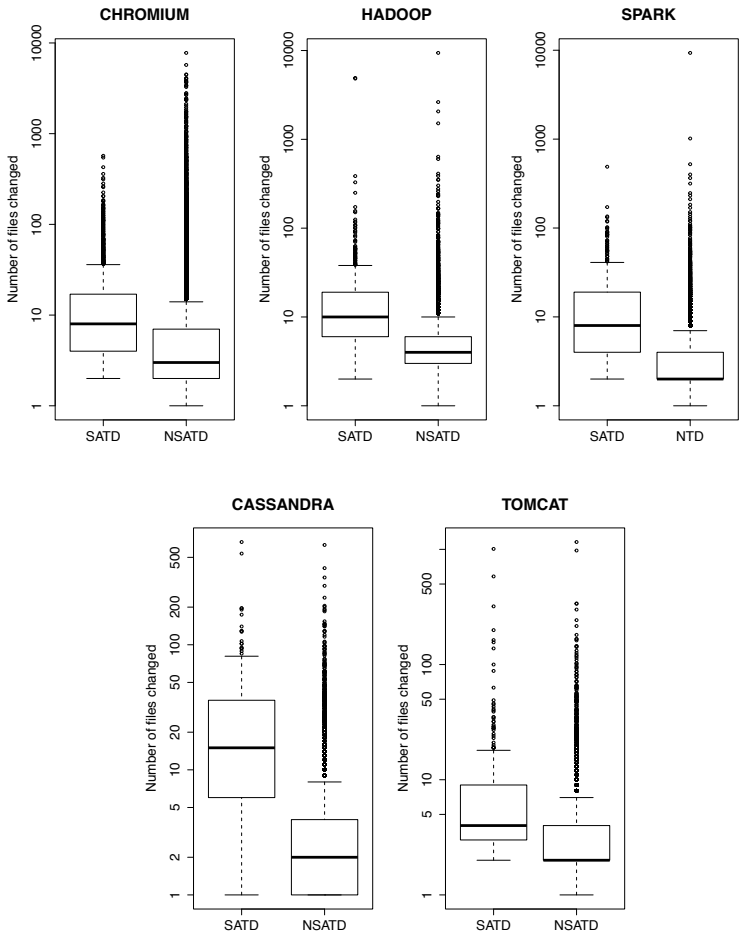

Figure 8: Total number of files modified per change (SATD vs. NSATD).

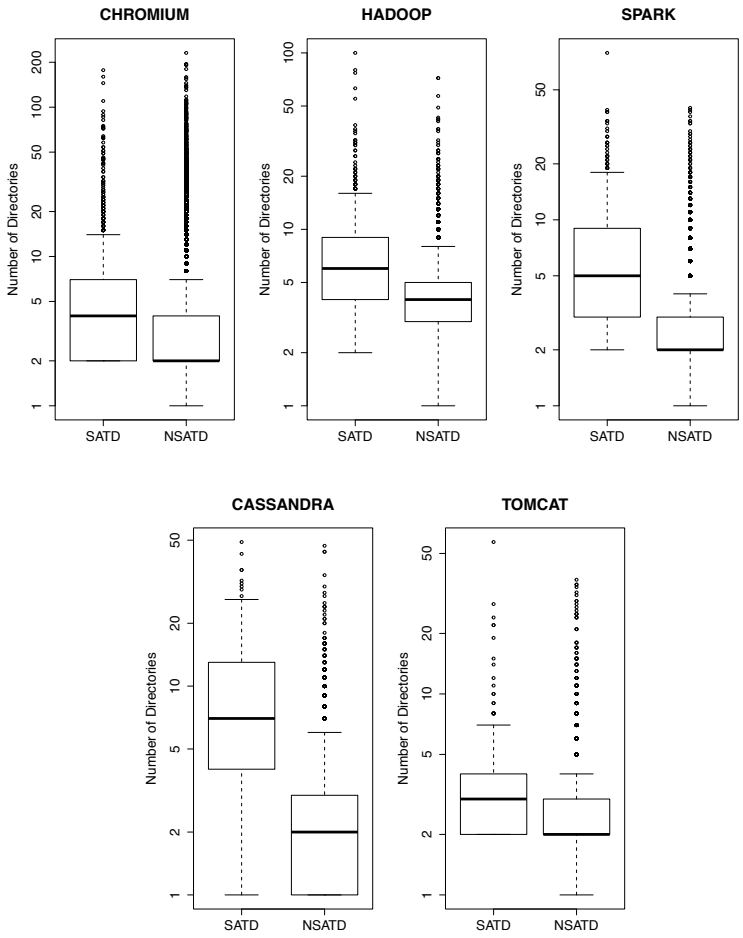

Figure 9: Total number of modified directories per SATD and NSATD change.

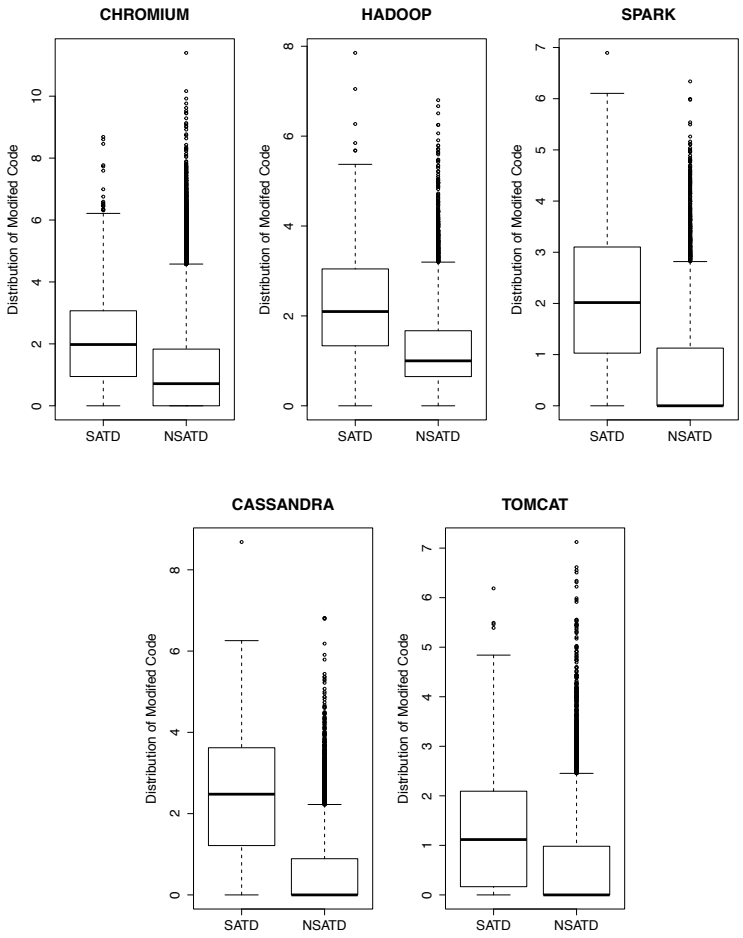

Figure 10: Distribution of the change across the SATD and NSATD files.

higher the normalized entropy, the more difficult the change.

Results: Figures 7, 8, 9 and 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. 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.

> **RQ3 - Interim Summary**
>
> SATD changes are more difficult to perform.

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.

3.4 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. 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 [PS14] examined this phenomenon in Eclipse and found that in between 70% and 90% of cases code and comments change in tandem.

We performed this step, independently, for each of the five projects studied and identified a change as an SATD change if it contained 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. However, 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.

Threats to **construct validity** concern the degree to which indirect metrics fall short of measuring what they were developed to measure. For example, as a means of locating SATD, we use the comments compiled by Potdar and Shihab [PS14], 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 [PS14].

3.5 Conclusion

Technical debt is intuitively recognized as bad practice by software companies and organizations, yet there is very little empirical evidence on the extent to which technical debt can impact software quality. Therefore, in this chapter 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 [PS14], 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. Our findings suggest that there is no reliable trend when it comes to defects and SATD. 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. 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.

4

Comparing the Relationship Between Comment- Versus Metric-Based Technical Debt and Software Quality

4.1 Introduction

In the previous chapter, we studied the relationship between self-admitted technical debt and software quality and found that the presence of SATD complicates future changes. Supplementing these comment-based indicators with the metric-based indicators used in earlier work [ZSSS11] (God Classes), we replicate the study conducted in chapter 3 on a larger scale in order to compare the relationship between both comment- and metric-based technical debt and quality. Of the foregoing studies that have covered how metric-based technical debt affects software quality, few have done so on large datasets. We remedy this on the one hand by integrating both comment- and metric-based approaches, as mentioned, and on the other by introducing a more

granular analysis on the file and change levels. Empirically examining how both approaches relate to software quality and comparing any differences between them will provide researchers and practitioners with a more global understanding of technical debt, warn them of its future risks and raise awareness of the challenges it can pose.

Predictably, as technical debt has become a more popular strategy at developers' disposal, numerous advances have been made in detecting it, some metric-based and others comment-based. The former includes Marinescu's [Mar04] 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 [PS14] 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.

Without access to research that treats technical debt from all angles, 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. If our results are true for a larger number of projects, then they are even more likely to generalize to others.

Our inquiry pursues: (i) whether god class and SATD files have more defects than files free of god classes and SATD, (ii) whether god- and SATD-related changes introduce future defects, (iii) whether god- and SATD-related changes are associated with greater difficulty and (iv) to what extent the comment- and metric-based approaches identify the same instances of technical debt. As in the previous chapter, amount of churn, quantity of affected files and modified modules and change entropy

all factor into the change difficulty calculations. 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, (i)-(iii) concede that the downsides of god classes and self-admitted technical debt are increases in future defect density and change difficulty. Yet the two approaches were found to reinforce each other in that (iv) between 11% and 34% of technical debt sources were identified by both the comment- and metric-based approaches.

4.2 Approach

As we continue to study the interplay between self-admitted technical debt and metric-based debt (God Classes) and software quality, measures must be established in order to quantify software quality [KSA+13, KWZ08, ŚZZ05b]. 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 and SATD files have more defects than non-god and non-SATD files?
- **RQ2:** Do god- and SATD-related changes introduce future defects?
- **RQ3:** Are god- and SATD-related changes more difficult than non-god and non-SATD changes?

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

We devised a suitable methodology, visualized in Figure 11, to guide our inquiry into these questions. We initiate 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 are parsed and comments extracted (step 3). We then identify all instances of self-admitted technical debt, count defects file-wide and isolate defect-inducing changes by means of the SZZ algorithm (steps 4-5).

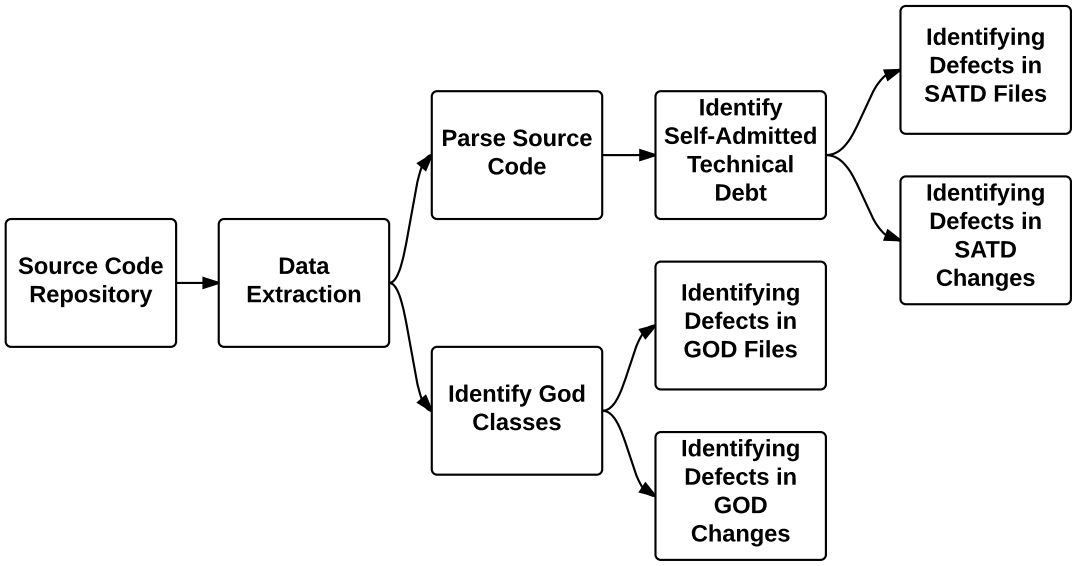

Figure 11: Metric-based (God Classes) approach overview.

4.2.1 Data Extraction

To generalize the scope of this study, we relied on certain criteria in selecting the 40 open-source software projects: (i) well-commented source code, (ii) varying numbers of lines of code (LOC), (iii) different numbers of contributors, (iv) different development domains, (v) mature development history and (vi) issue tracking system capability. The first criterion is a prerequisite for Potdar and Shihab's SATD detection technique; the last is essential to accurately study the introduction of future

defects.

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 (part 1).

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 (part 2).

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

4.2.2 Scanning Code and Extracting Comments

Now source code comments must be extracted from the projects under investigation, for which we implement a Python-based tool. The rest of the extraction process is the same as for subsection 3.2.2.

4.2.3 Filter Comments

Source code comments left by developers might situate the project within the circumstances of its development, communicate their recommendations for revising the code at a later date, acknowledge who wrote which pieces or who made which fixes or confess 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 proposed by Moldonado *et al.* [MST17] 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 the 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, there is no SATD content. 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.

4.2.4 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 [PS14] in identifying self-admitted technical debt on the basis of 62 different patterns recurring in multiple projects at various frequencies. The specifics of these patterns can be found in subsection 3.2.3.

SATD changes: At the change level, all the files touched by the same change are

checked 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. In general, the more SATD files a system contains, the more likely it is to have a higher number of SATD changes. 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 Tables 7 and 8, where each system's percentages are listed separately for comparison.

Table 7: Percentage of SATD and God of the analyzed projects (part 1).

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 (part 2).

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

4.2.5 God Classes

God classes are classes that combine trivial class workloads and generally avoid assigning tasks to other classes. They are distinguishable on account of their high complexity, low inner-class cohesion and frequent foreign class data access [LDM07]. Object-oriented design advocates a one-to-one correspondence between classes and responsibilities, which god classes violate by definition [LDM07]. 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 [FB99] 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 [FB99, LDM07].

4.2.6 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 [Mar04], who proposed an approach to specify and detect code smells, specifically *God Classes*. Their technique leverages metric-based heuristics that identify god classes 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 relationship between god classes and quality 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 [CK94]. McCabe's cyclomatic complexity [McC76] 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 [BK95].
- **Access to Foreign Data (ATFD)** is the number of external classes whose attributes are accessed either directly or indirectly (by accessor methods) [Mar05].

4.2.7 Identifying Defects in God Files and God Changes

According to Sliwersky *et al.* [ŚZZ05b], 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 [KSA+13, KWZ08, ŚZZ05b]. Next we pull each defect report from its corresponding issue tracking system, i.e., Bugzilla [1] or JIRA [2], and comb for all pertinent details. Once the god files and god changes have

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

been separated, we go about determining the number of file defects and identifying any defect-inducing changes in the same way foregoing research has [KSA+13, KWZ08, ŚZZ05b].

Defects in files: A file defect count is a prerequisite to any defectiveness comparison between god and non-god 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. Examples of these keywords can be found in subsection 3.2.4, along with the steps we take to rule out false positives.

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. The keywords used to identify corrective changes are given in subsection 3.2.4, where the procedure for identifying defect-inducing changes is presented.

4.3 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. Each of the three research questions is restated below, where we summarize its motivation, our approach in treating it and the conclusions we reached. Statistics and results are listed for all individual projects, accompanied by cross-project comparisons.

RQ1: Do god and SATD files have more defects than non-god and non-SATD 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 [ZSSS11]. The potential drawbacks of SATD, meanwhile, have remained unexplored despite its research-affirmed ubiquity in software projects [PS14]. 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, raising 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.

Comparing god and non-god files: The God Class Detection Equation proposed by [Mar04] provides a way to identify god classes using object-oriented metrics. Files are fed to the equation and are labeled either god or non-god files, depending on the output. We calculate the normalized value of defect-fixing changes for each file in both categories based on $\frac{\#\ of\ fixing\ changes}{SLOC}$. We normalize our data by dividing the number of defect-fixing changes by the number of source lines of code, since god files are inherently large, and apply a test designed to measure whether the differences between the categories are statistically significant.

In case this distribution is non-normal, we use the non-parametric Mann-Whitney [MW47] test because, unlike the parametric alternatives, it is capable of handling such distributions. A p-value such that $p \leq 0.05$ indicates that the difference between the samples is statistically significant.

Comparing SATD and non-SATD files: We identify SATD files in accordance with the procedure detailed in section 3.2.3, 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, $\frac{\#\ of\ fixing\ changes}{SLOC}$, which, unlike pure counts, standardizes the metric across files hosting different numbers of source lines of code. Afterwards the defect distribution is plotted for SATD and non-SATD files and a test is performed

to uncover any statistical trends.

Again, we elected to conduct the non-parametric Mann-Whitney [MW47] 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. A statistically significant difference returns a p-value of at most 0.05 ($p \leq 0.05$).

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

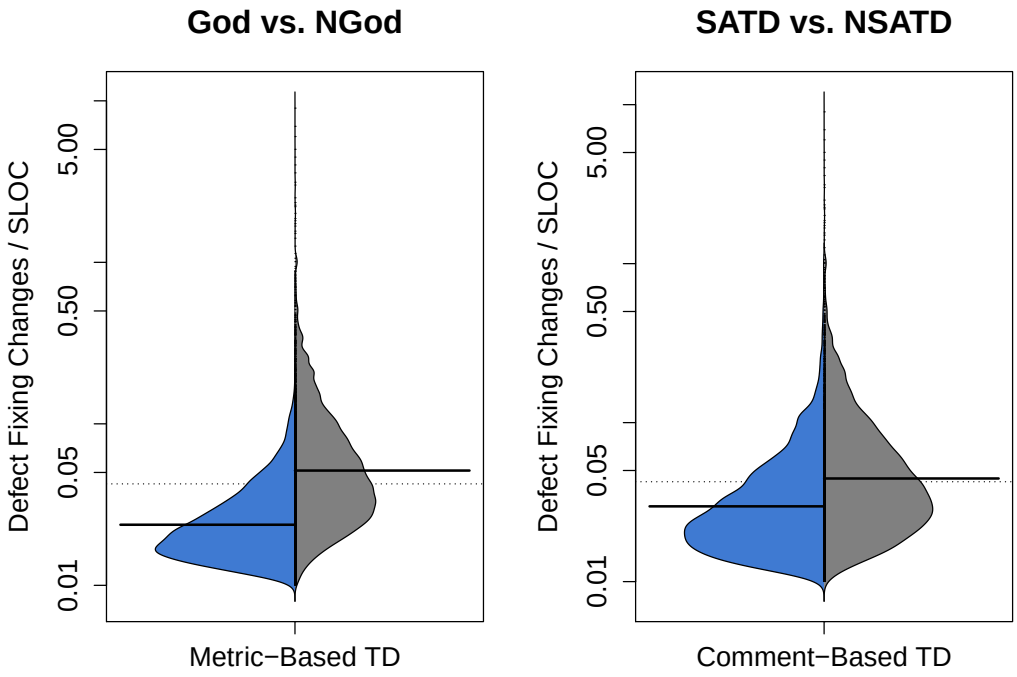

Figure 13: Percentage of defect-fixing changes for (i) God vs. non-God files and (ii) SATD vs. NSATD files.

Beanplots are a convenient way to present and compare univariate data for two groups. Those in Figure 13 display the distribution of *median corrective change rates*, based on $\frac{\#\ of\ fixing\ changes}{SLOC}$, for god files versus non-god files and SATD files versus

non-SATD files *in each project*. The dotted lines represent the overall mean taking data from both groups into account; for this reason, the dotted lines mark the same value on both sides of each beanplot. The solid lines, in contrast, mark the median value for each group and thus differ from one side to the other. 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.

RQ1 - Interim Summary

Neither god files nor self-admitted technical debt files are associated with a higher percentage of defects.

Figure 19 (in the appendix) 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 $p \leq 0.05$, which holds when all project medians are consolidated in the distribution in Figure 13.

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 20, we observe that OpenNLP, Curator and Tiles constitute exceptions where the defectiveness rate is higher among SATD files.

RQ2: Do god- and SATD-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 an 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 software quality is affected, we should first determine to what extent god classes and SATD are predisposed to introduce future defects. Our conjecture is that god and SATD changes introduce future defects at a higher rate than non-god and non-SATD changes. Further, we expect that 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.

Approach: We identify defect-inducing changes utilizing the SZZ algorithm [ŚZZ05b] 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). We then apply the Mann-Whitney test [MW47] to evaluate the statistical significance of the difference between god versus non-god defect-inducing changes and SATD versus non-SATD defect-inducing changes (from the same respective data sets). If the resulting p-value is such that $p \leq 0.05$, then the difference is not attributable to chance but rather statistically significant, and generalizes to other data sets.

Results: God and non-god distributions of defect-inducing change rates in each project share a common vertical axis in the first plot in Figure 14, as do SATD and non-SATD defect-inducing change rate distributions in the second. We observe that the distribution median (i.e., the median of the individual project medians) is higher for the god and SATD changes than for the non-god 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. We also find that the differences between god versus non-god and SATD versus non-SATD defect-inducing changes are both statistically significant with $p \leq 0.05$.

RQ2 - Interim Summary

Both god changes and SATD changes tend to introduce a higher number of future defects.

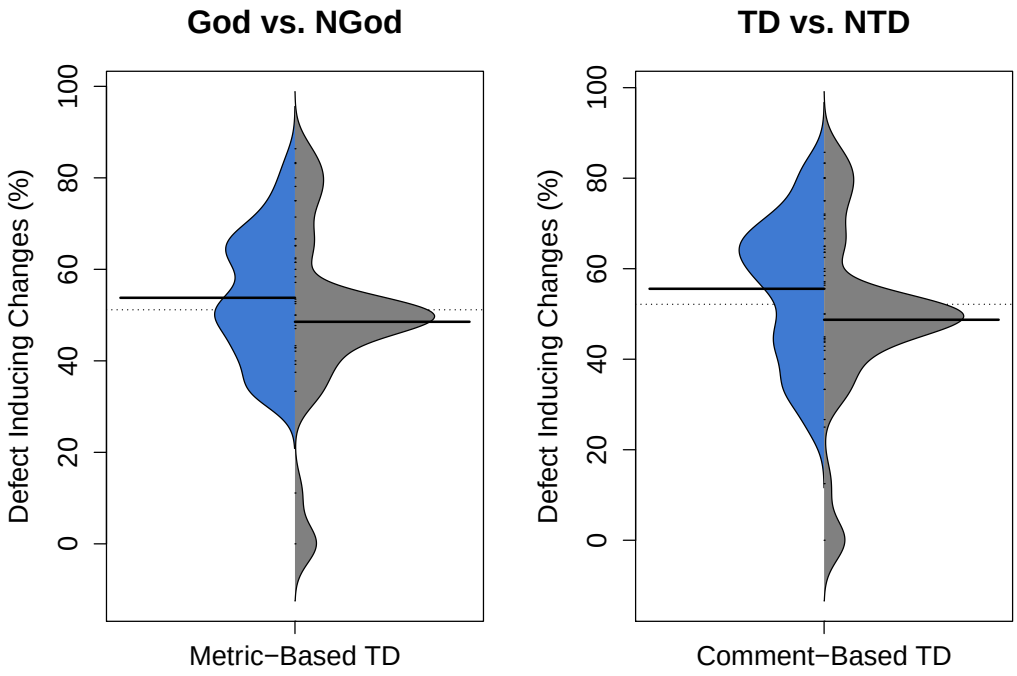

Figure 14: Percentage of defect-inducing changes for (i) God vs. non-God and (ii) SATD vs. NSATD.

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 these projects, the non-god and non-SATD changes appeared to induce more future defects than the god and SATD changes. 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- and SATD-related changes more difficult than non-god and non-SATD 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 [Mar04], 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 executing changes from each category. We reuse the four metrics from chapter 3 to determine change difficulty: churn, the number of modified directories, the number of modified files and the entropy of the change. For a slightly different purpose, Eick *et al.* [EGK+01] chose the first three to measure decay; Hassan [Has09] utilized the last metric to measure change complexity. As in RQ2, we use the Mann-Whitney test [MW47] to determine whether the differences between god and non-god changes and between SATD and non-SATD changes are statistically significant and measure the effect size using Cliff's delta [GK05a], this time with respect to the complexity metric categories.

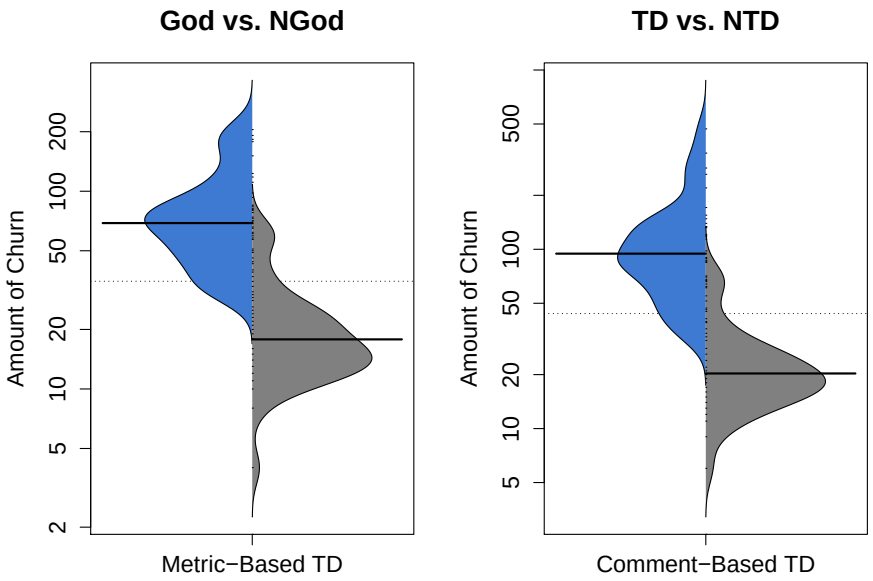

Figure 15: Total number of lines modified per change for (i) God vs. non-God and (ii) SATD vs. NSATD.

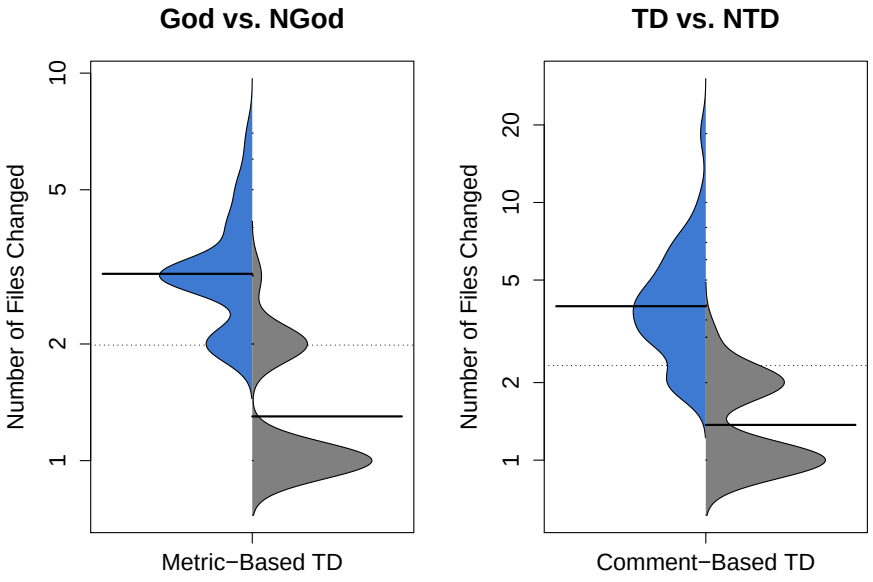

Figure 16: Total number of files modified per change for (i) God vs. non-God and (ii) SATD vs. NSATD.

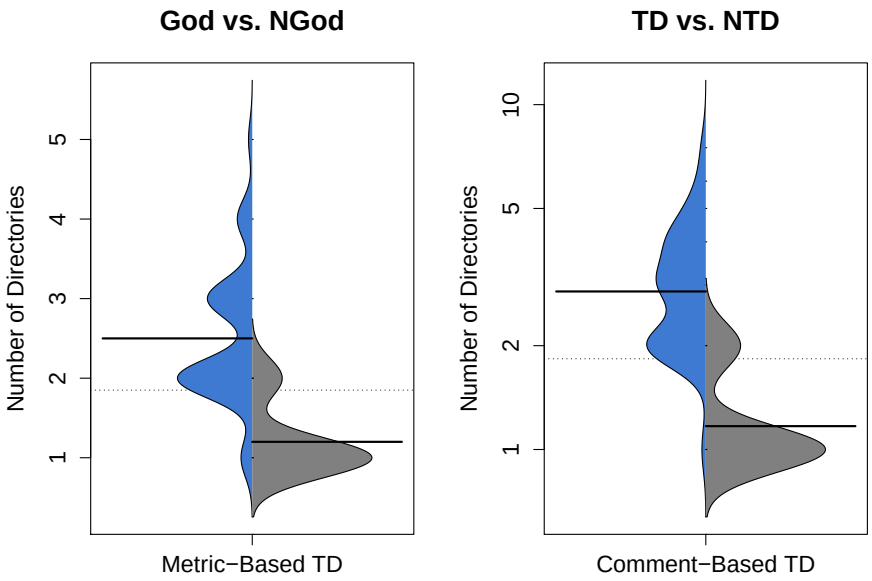

Figure 17: Total number of modified directories per change for (i) God vs. non-God and (ii) SATD vs. NSATD.

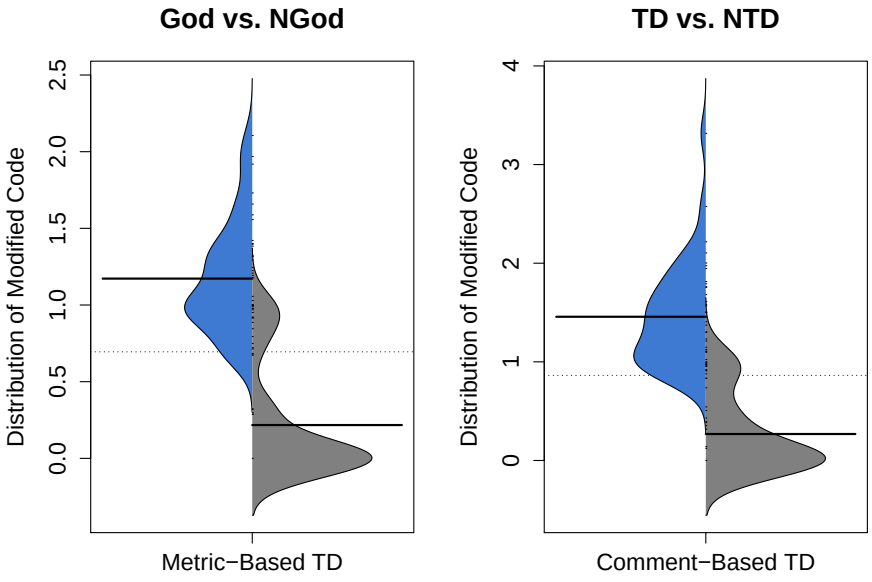

Figure 18: Distribution of the change across files for (i) God vs. non-God and (ii) SATD vs. NSATD.

Results: In each one of Figures 15, 16, 17 and 18, a distribution is compiled for god and 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, god and SATD changes are consistently more difficult to perform than non-god and non-SATD changes. Moreover, the Mann-Whitney test [MW47] yields $p \leq 0.05$, indicating that the differences are statistically significant. Tables 9 and 10 show the Cliff's delta [GK05a] effect size values for all projects studied for both god and SATD changes. We observe that for most projects and all measures of difficulty, the effect size is either medium or large, except for *Hbase, Hama, Deltaspike, Calcite and Stanbol* for god changes and *Hbase and Deltaspike* for SATD changes, which have a small effect size.

Project	NF	E	C	ND
Apache openNLP	0.31	0.30	0.35	0.36
Apache Camel	0.42	0.41	0.39	0.37
Apache Hbase	0.23	0.24	0.29	0.22
Apache Groovy	0.36	0.33	0.34	0.40
Apache Oltu	0.49	0.47	0.43	0.48
Apache Maven	0.35	0.34	0.33	0.33
Apache Karaf	0.29	0.26	0.36	0.27
Apache Hama	0.27	0.25	0.32	0.22
Apache Tomee	0.36	0.34	0.36	0.35
Apache Deltaspike	0.30	0.28	0.29	0.29
Apache Curator	0.22	0.33	0.45	0.33
Apache Calcite	0.22	0.24	0.23	0.27
Apache Poi	0.44	0.42	0.36	0.49
Apache Zeppelin	0.44	0.42	0.44	0.46
Apache Ant	0.47	0.45	0.43	0.49
Apache Stanbol	0.27	0.25	0.30	0.24
Apache Kafka	0.35	0.34	0.33	0.35
Apache Tika	0.35	0.33	0.33	0.36
Apache Felix	0.37	0.35	0.34	0.30
Apache Phoenix	0.50	0.45	0.47	0.54

Project	NF	E	C	ND
Apache Wicket	0.58	0.58	0.55	0.53
Apache Aurora	0.50	0.49	0.45	0.57
Apache Ignite	0.52	0.39	0.52	0.53
Apache Helix	0.45	0.42	0.47	0.43
Apache Archiva	0.39	0.39	0.38	0.30
Apache Struts	0.40	0.48	0.43	0.43
Apache Derby	0.54	0.53	0.52	0.52
Apache Ambari	0.34	0.32	0.41	0.33
Apache Nifi	0.40	0.37	0.43	0.44
Apache Tiles	0.30	0.29	0.29	0.36
Apache Shiro	0.36	0.34	0.38	0.39
Apache Usergrid	0.50	0.43	0.46	0.52
Apache Nutch	0.32	0.34	0.31	0.40
Apache Zookeeper	0.31	0.26	0.27	0.45
Apache Mina	0.42	0.39	0.40	0.43
Apache Cxf	0.56	0.54	0.56	0.55
Apache Cloudstack	0.37	0.34	0.36	0.44
Apache Oozie	0.52	0.47	0.47	0.60
Apache Kylin	0.36	0.30	0.35	0.38
Apache Flink	0.53	0.51	0.57	0.59

Table 9: Cliff's delta for the change difficulty measures across the projects for God Changes.

Project	NF	E	C	ND
Apache openNLP	0.34	0.33	0.36	0.39
Apache Camel	0.44	0.42	0.38	0.40
Apache Hbase	0.26	0.27	0.29	0.24
Apache Groovy	0.35	0.32	0.33	0.39
Apache Oltu	0.36	0.33	0.41	0.35
Apache Maven	0.40	0.39	0.37	0.38
Apache Karaf	0.35	0.32	0.40	0.34
Apache Hama	0.34	0.31	0.37	0.28
Apache Tomee	0.35	0.33	0.34	0.34
Apache Deltaspike	0.31	0.29	0.29	0.26
Apache Curator	0.40	0.30	0.51	0.40
Apache Calcite	0.30	0.31	0.29	0.35
Apache Poi	0.48	0.47	0.37	0.51
Apache Zeppelin	0.59	0.57	0.56	0.59
Apache Ant	0.41	0.49	0.47	0.49
Apache Stanbol	0.33	0.30	0.34	0.39
Apache Kafka	0.40	0.39	0.39	0.40
Apache Tika	0.37	0.33	0.30	0.38
Apache Felix	0.30	0.38	0.36	0.33
Apache Phoenix	0.52	0.47	0.44	0.56

Project	NF	E	C	ND
Apache Wicket	0.47	0.47	0.40	0.43
Apache Aurora	0.52	0.51	0.48	0.59
Apache Ignite	0.53	0.60	0.53	0.56
Apache Helix	0.53	0.51	0.50	0.49
Apache Archiva	0.30	0.39	0.39	0.40
Apache Struts	0.37	0.39	0.39	0.34
Apache Derby	0.37	0.36	0.32	0.34
Apache Ambari	0.38	0.36	0.44	0.38
Apache Nifi	0.57	0.55	0.53	0.59
Apache Tiles	0.57	0.56	0.43	0.52
Apache Shiro	0.35	0.39	0.35	0.36
Apache Usergrid	0.51	0.44	0.47	0.52
Apache Nutch	0.49	0.51	0.43	0.54
Apache Zookeeper	0.39	0.35	0.33	0.47
Apache Mina	0.54	0.49	0.47	0.51
Apache Cxf	0.36	0.34	0.31	0.35
Apache Cloudstack	0.40	0.37	0.29	0.46
Apache Oozie	0.53	0.50	0.45	0.57
Apache Kylin	0.57	0.52	0.53	0.59
Apache Flink	0.47	0.44	0.39	0.49

Table 10: Cliff's delta for the change difficulty measures across the projects for SATD Changes.

RQ3 - Interim Summary

God class-related changes and SATD-related changes are more difficult to perform than non-god and non-SATD changes.

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

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

Motivation: Thus far, we have compared how the technical debt identified by comment- and metric-based approaches relates to 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 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: We take the list of self-admitted technical debt 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 11 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. In other work, Moldonado *et al.* [MST17] studied the overlap between self-admitted technical debt and several types of code smells extracted by a static analysis tool for 10 open-source projects. For self-admitted technical debt and god classes specifically, they found an average overlap of 44.2%, which, though higher than the overlap we found, is based on a smaller sample size. Despite considerable overlap, each of the comment- and metric-based approaches identifies some additional sources of technical debt that the other fails to detect.

RQ4 - Interim Summary

The comment-based approach complements the metric-based approach with an overlap ranging from 11% to 34%. Nevertheless, practitioners should integrate both approaches in order to better detect technical debt.

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 11: Percentage of overlap between God and SATD files of the analyzed projects.

4.4 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 [PS14] found that in Eclipse code and comments were updated in tandem between 70% and 90% of the time. Another threat derives from comments intended to indicate SATD that do not correspond to any of the patterns Potdar and Shihab [PS14] 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 [PS14]. 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. Nonetheless, other systems should be analyzed to support the conclusions of this chapter, for one thing, because all the projects studied were written in Java, which limits programming language representation. Additionally, the projects studied were all developed by Apache, so systems developed by other companies could potentially run counter to our findings. Drawing on open-source projects means we have no guarantee that our results hold for industrial systems. 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.

Threats to **construct validity** concern the extent to which indirect metrics do not accurately measure what they are intended to. Code metrics and thresholds were used to detect god classes as Marinescu [Mar04] proposes, and while these have proven effective when applied in other studies, it has not been evaluated whether such strategies are suitable to use in all contexts. If not, findings will depend heavily on the particular metrics and thresholds specified for detection and varying these will alter our findings.

4.5 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 [Mar04] object-oriented metric thresholds and self-admitted technical debt by locating source code comments that match the SATD indicator patterns in [PS14].

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 relationship between self-admitted technical debt and 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. After dealing with god and SATD files and changes separately, we assess (iv) to what extent the metric- and comment-based approaches overlap.

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. As for the overlap between the two approaches, we learned that (iv) the metric- and comment-based approaches identify the same sources of technical debt in 11% to 34% of cases.

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. We advise practitioners to integrate both the comment- and metric-based approach to improve technical debt detection.

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 how comment-based technical debt (self-admitted technical debt) relates to software quality. In this chapter, we conduct a preliminary study to 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 conduct a large-scale study on 40 open-source projects to understand how god classes and SATD influence software quality. We observe that: (i) neither the incidence of god nor SATD files is correlated with defects, (ii) future defects are introduced at a higher rate by god and SATD changes, (iii) the difficulty imposed on the system is greater for god and SATD changes and (iv) the comment- and metric-based approaches agree on 11% to 34% of identified sources of technical debt.

5.2 Contributions

The major contributions of this thesis are as follows:

- Empirically examine the relationship between self-admitted technical debt and software quality.
- Enhance knowledge of the technical debt phenomenon by presenting a large-scale empirical study that compares the SATD (comment-based) and non-SATD (metric-based) approaches.
- Provide evidence that technical debt tends to induce more future defects and increase system complexity.

5.3 Future Work

We believe that this thesis advances the state of the art in understanding the relationship between technical debt and 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 consequences.

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 relationship between technical debt and 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

The following boxplots display the defect-proneness for god versus non-god and SATD versus non-SATD.

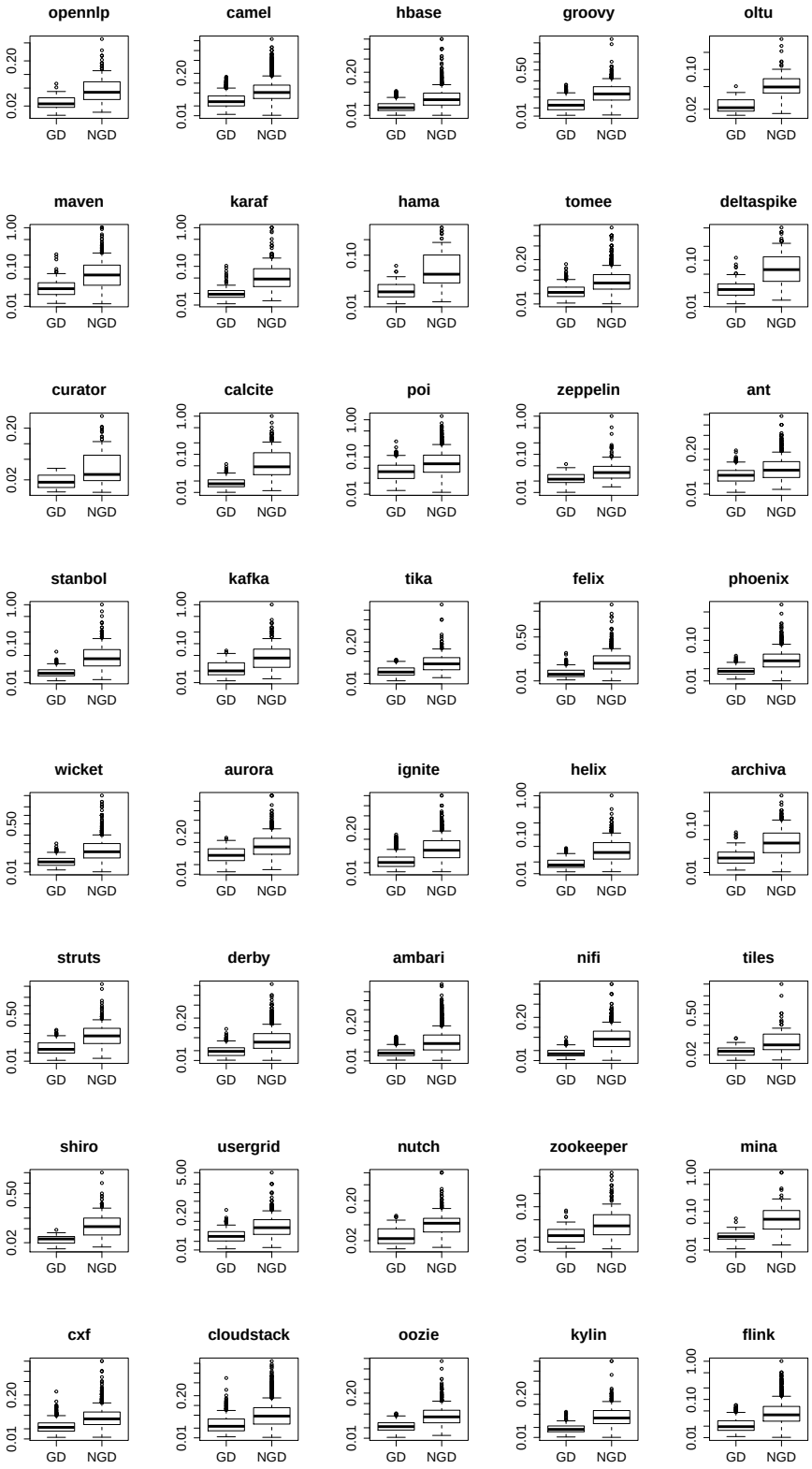

Figure 19: Percentage of defect fixing changes for GOD and NGOD files.

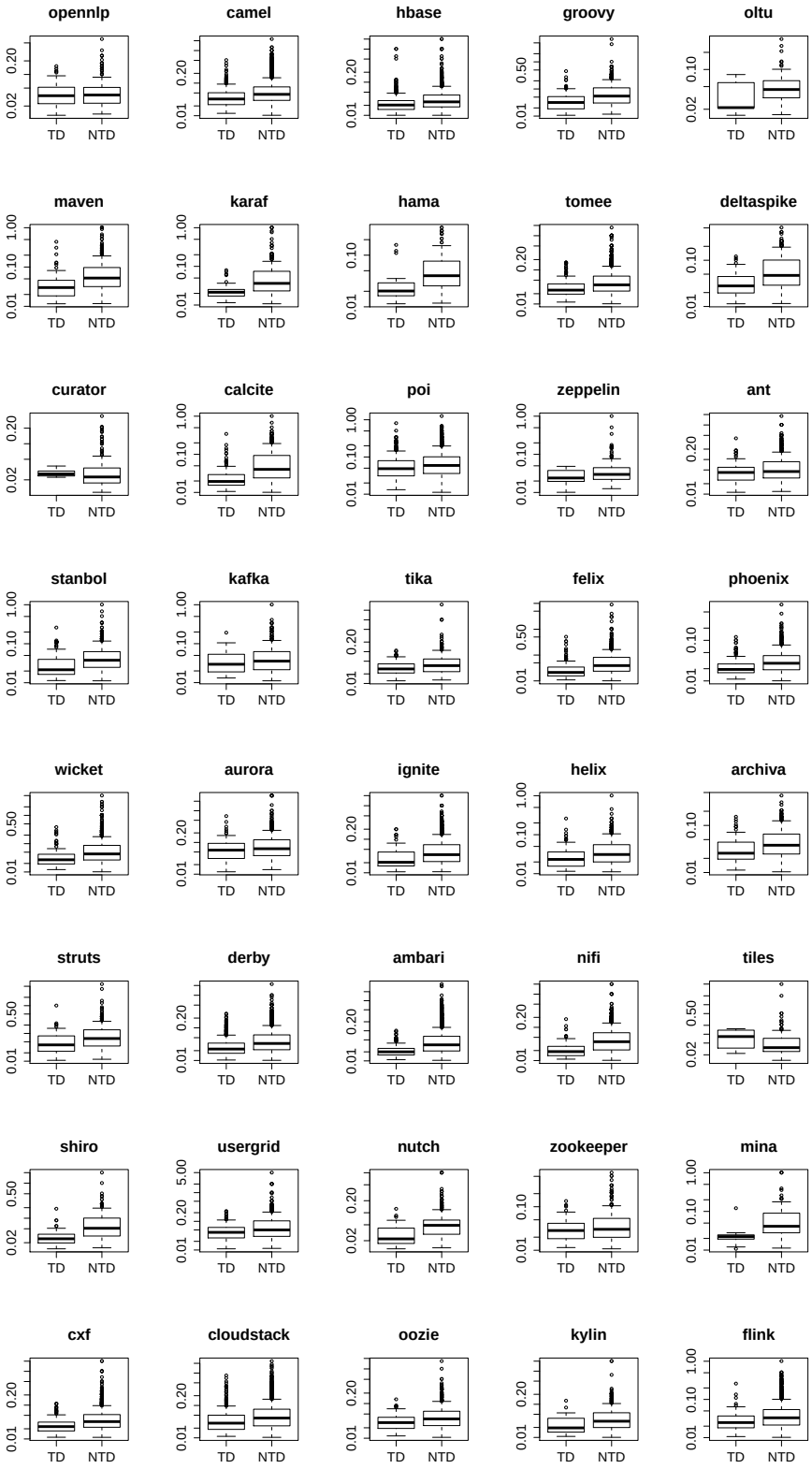

Figure 20: Percentage of defect fixing changes for TD and NTD files.

Appendix B

Defect Inducing Changes

The following boxplots display the propensity of god versus non-god and SATD versus non-SATD changes to introduce future defects.

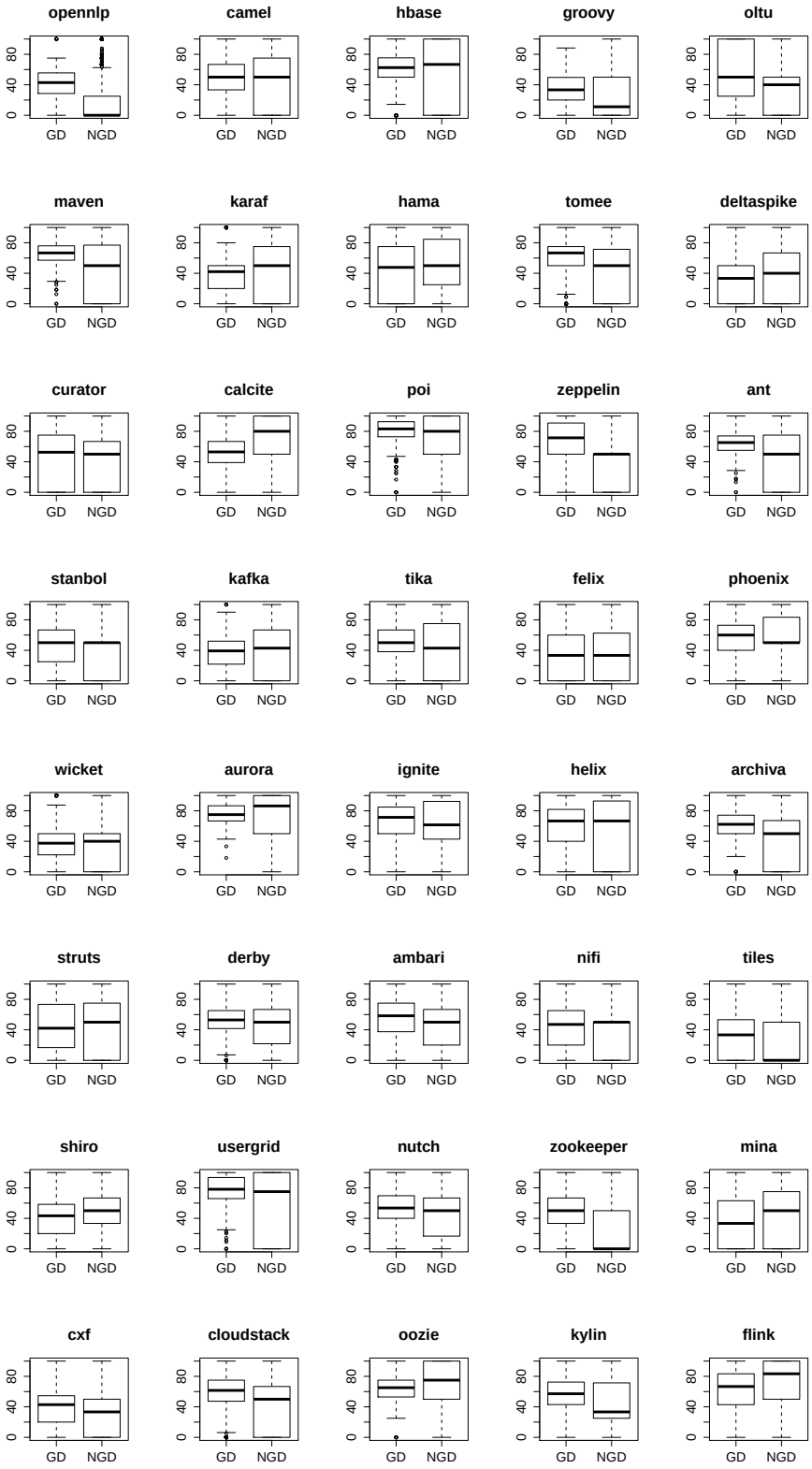

Figure 21: Percentage of defect inducing changes for GOD and NGOD files.

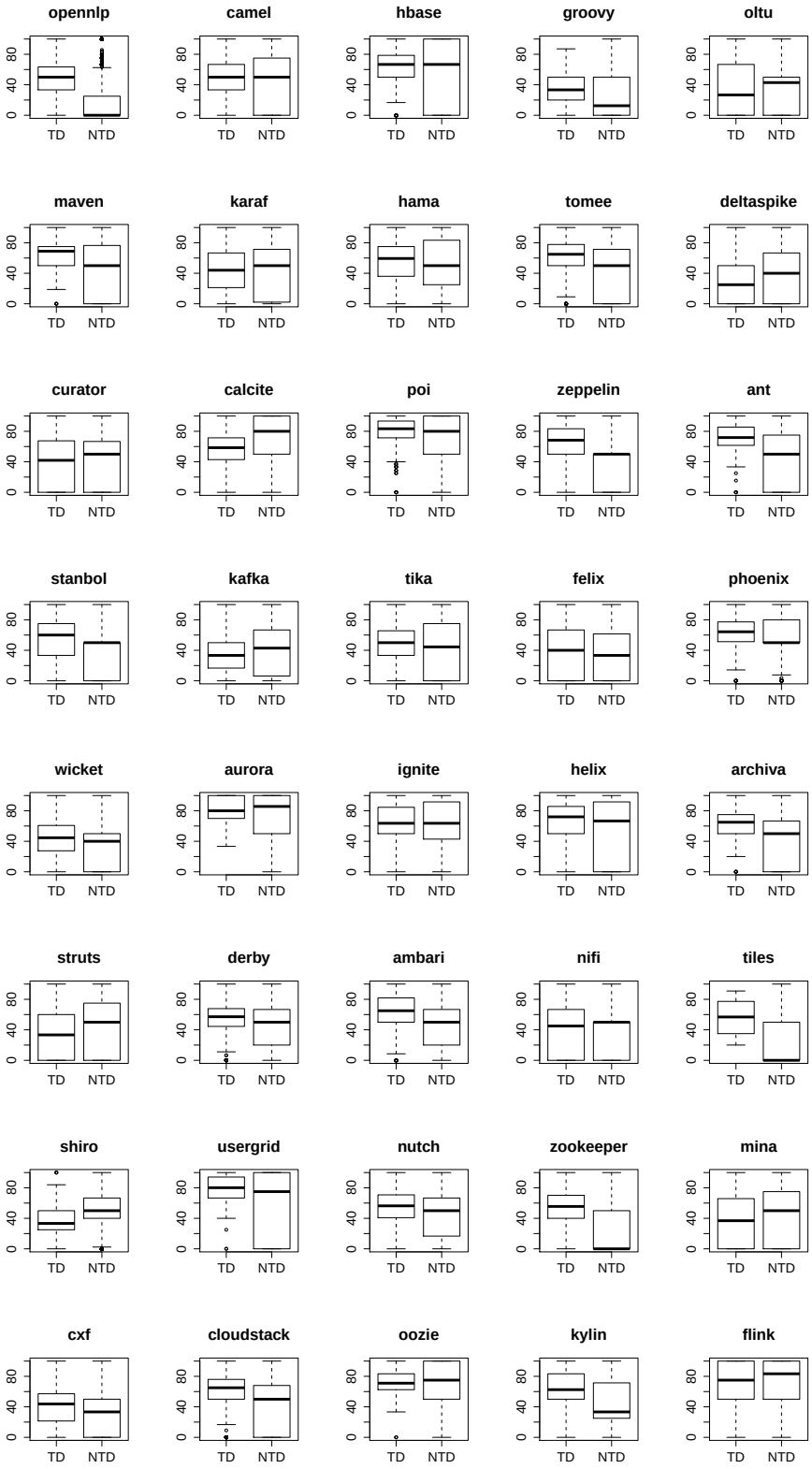

Figure 22: Percentage of defect inducing changes for TD and NTD files.

Appendix C

Complexity on The Change-level

The following boxplots display the change level complexity measured by total number of modified lines in a change, the number of modified directories, the number of modified fies, and change entropy for god versus non-god and SATD versus non-SATD.

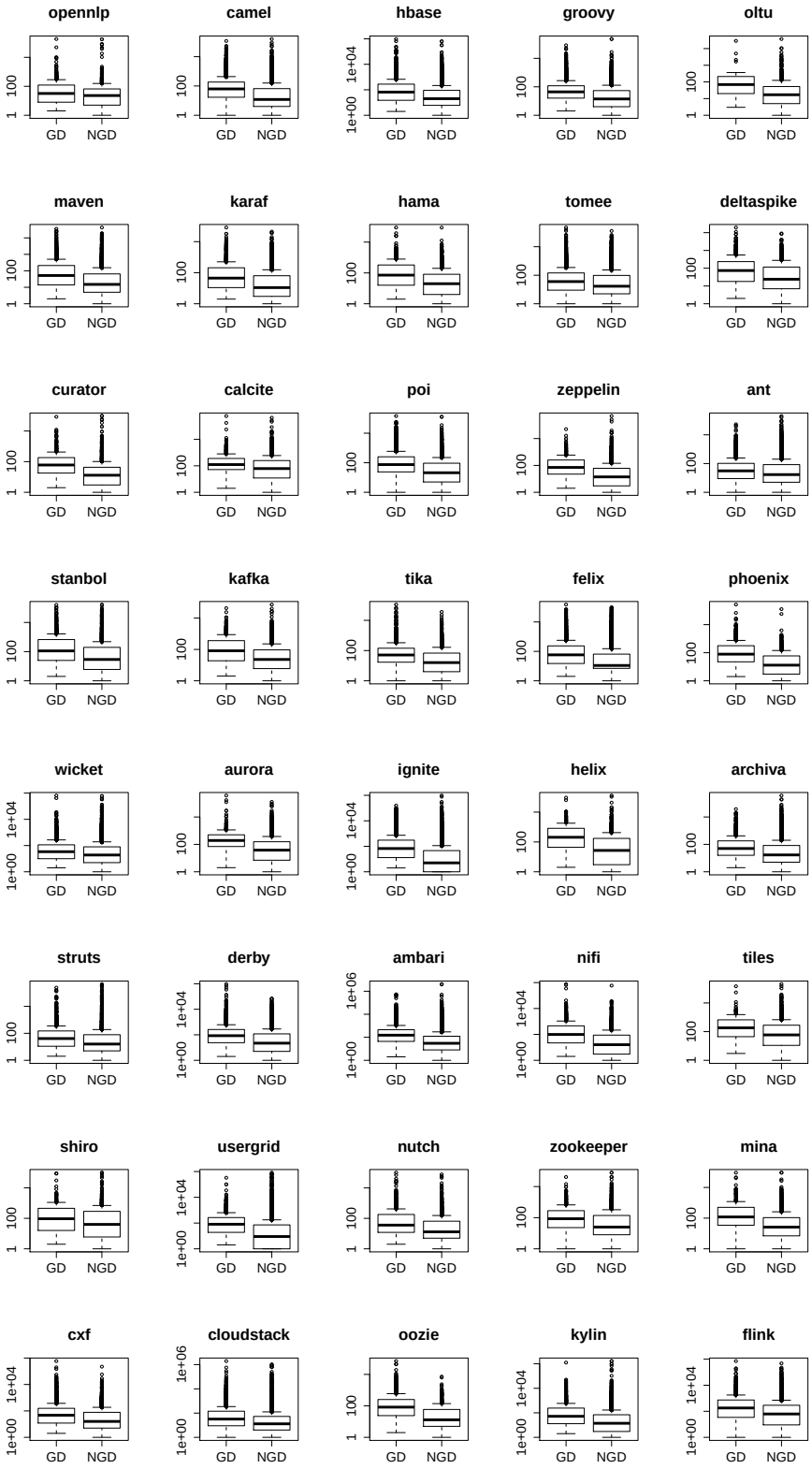

Figure 23: Total number of lines modified per change (GOD vs. NGOD).

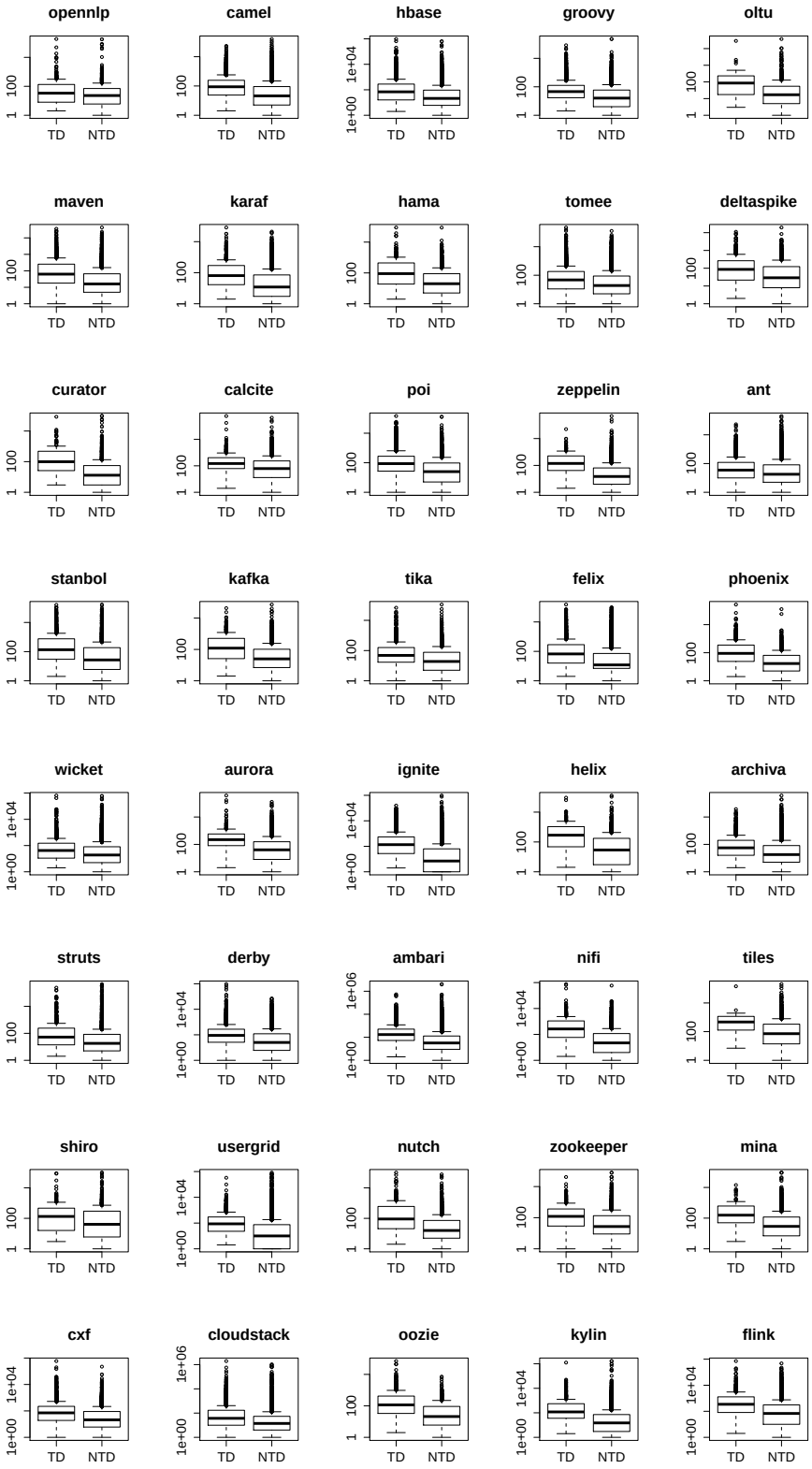

Figure 24: Total number of lines modified per change (TD vs. NTD).

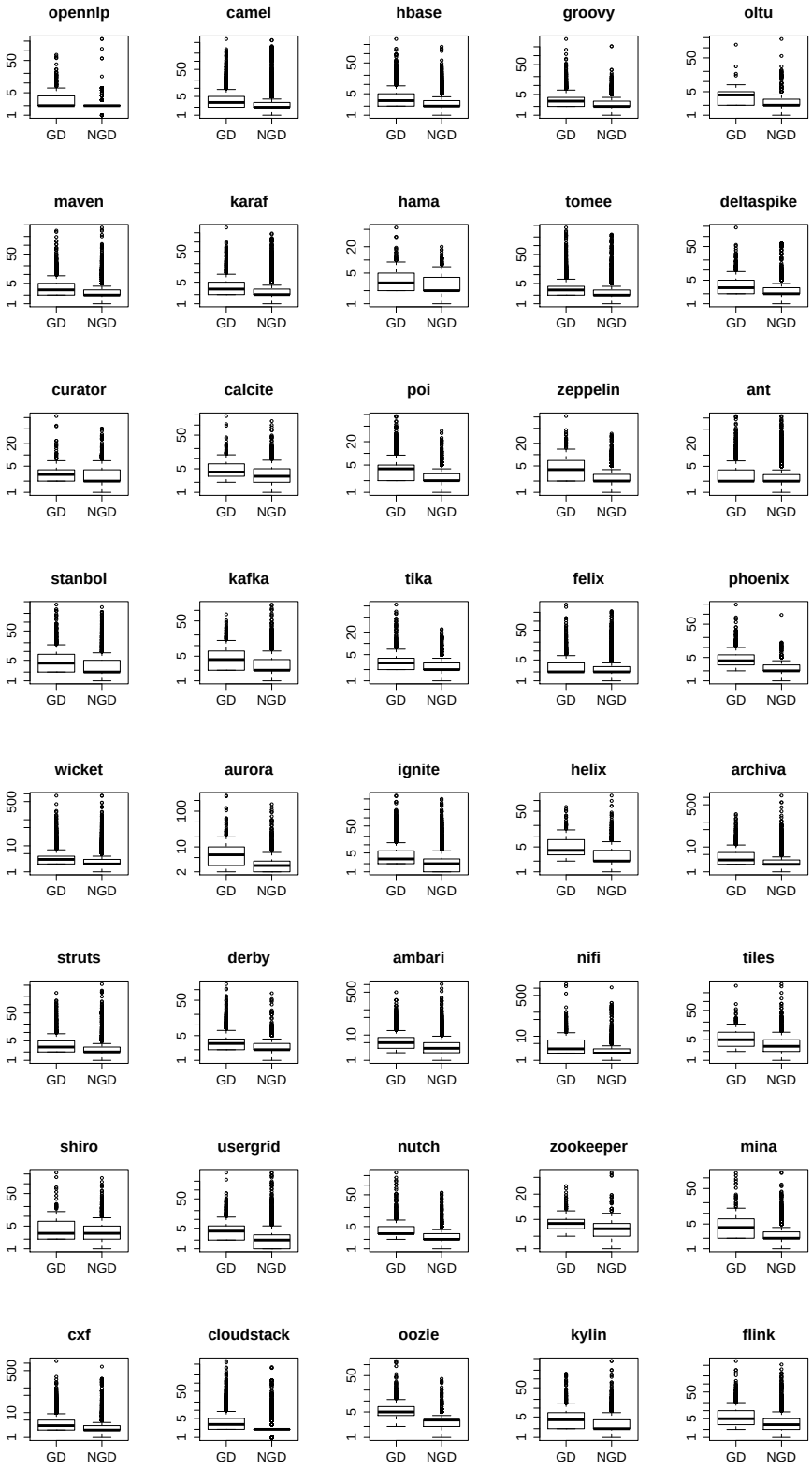

Figure 25: Total number of modified directories per GOD and NGOD change

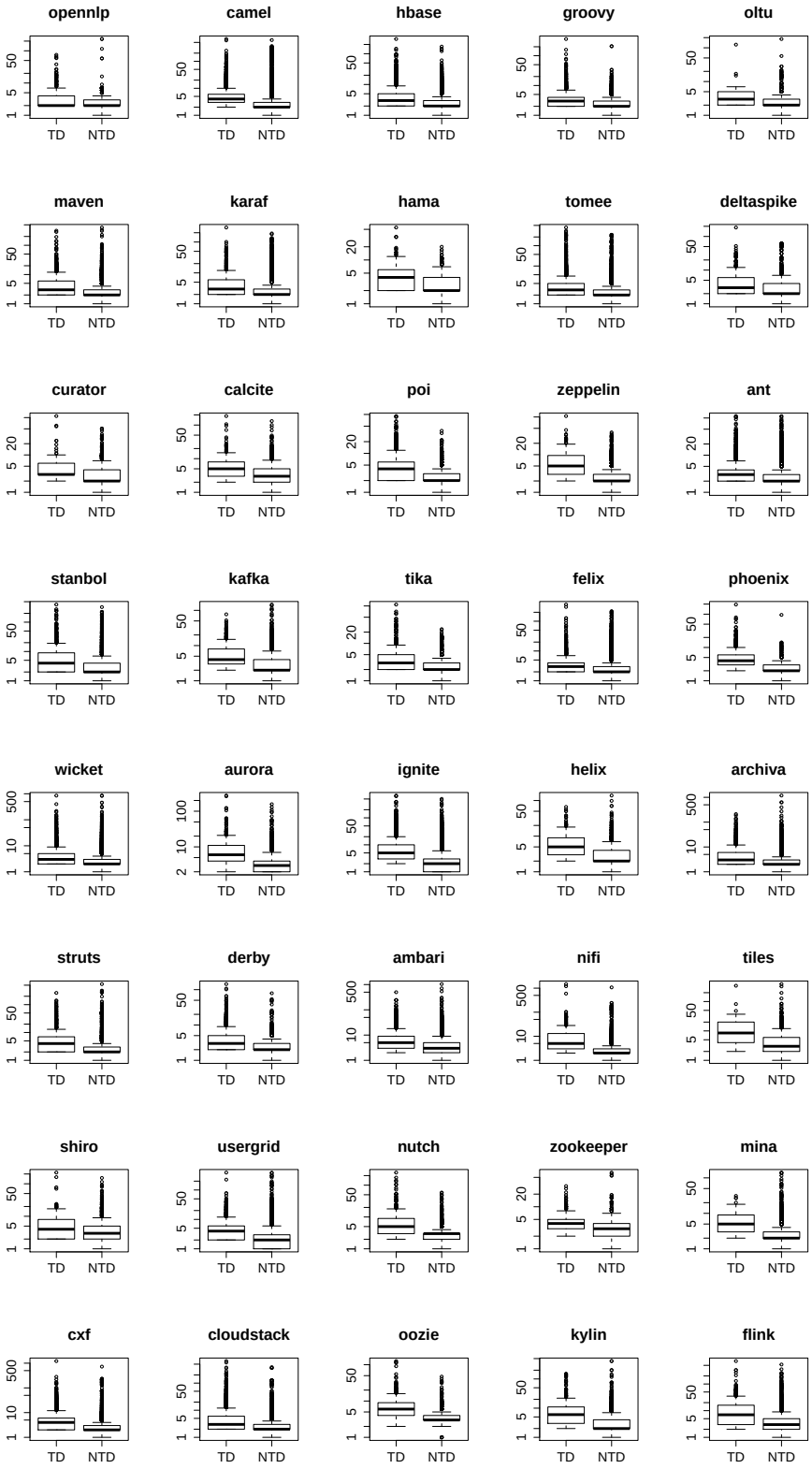

Figure 26: Total number of modified directories per SATD and NSATD change.

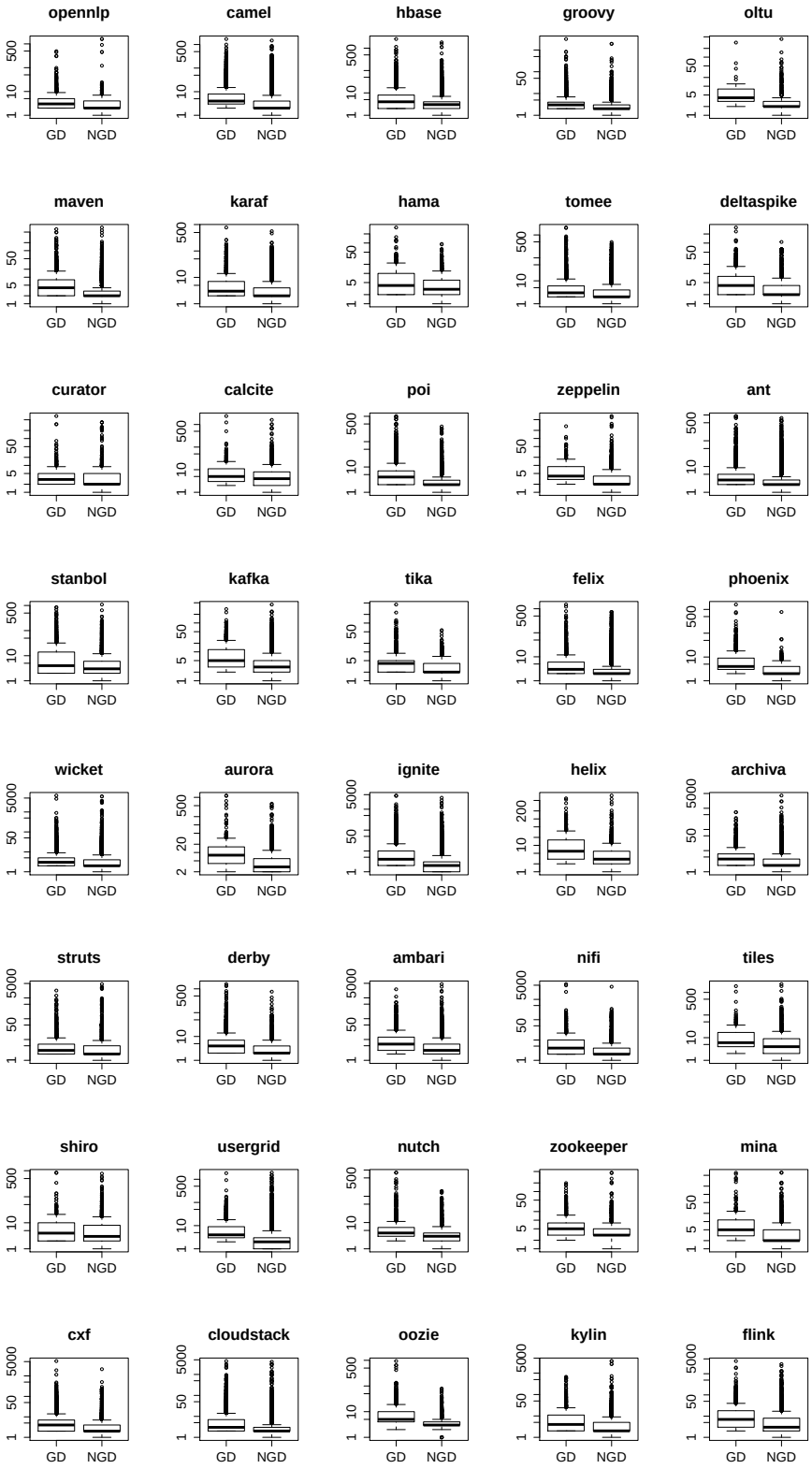

Figure 27: Total number of files modified per change (GOD vs. NGOD).

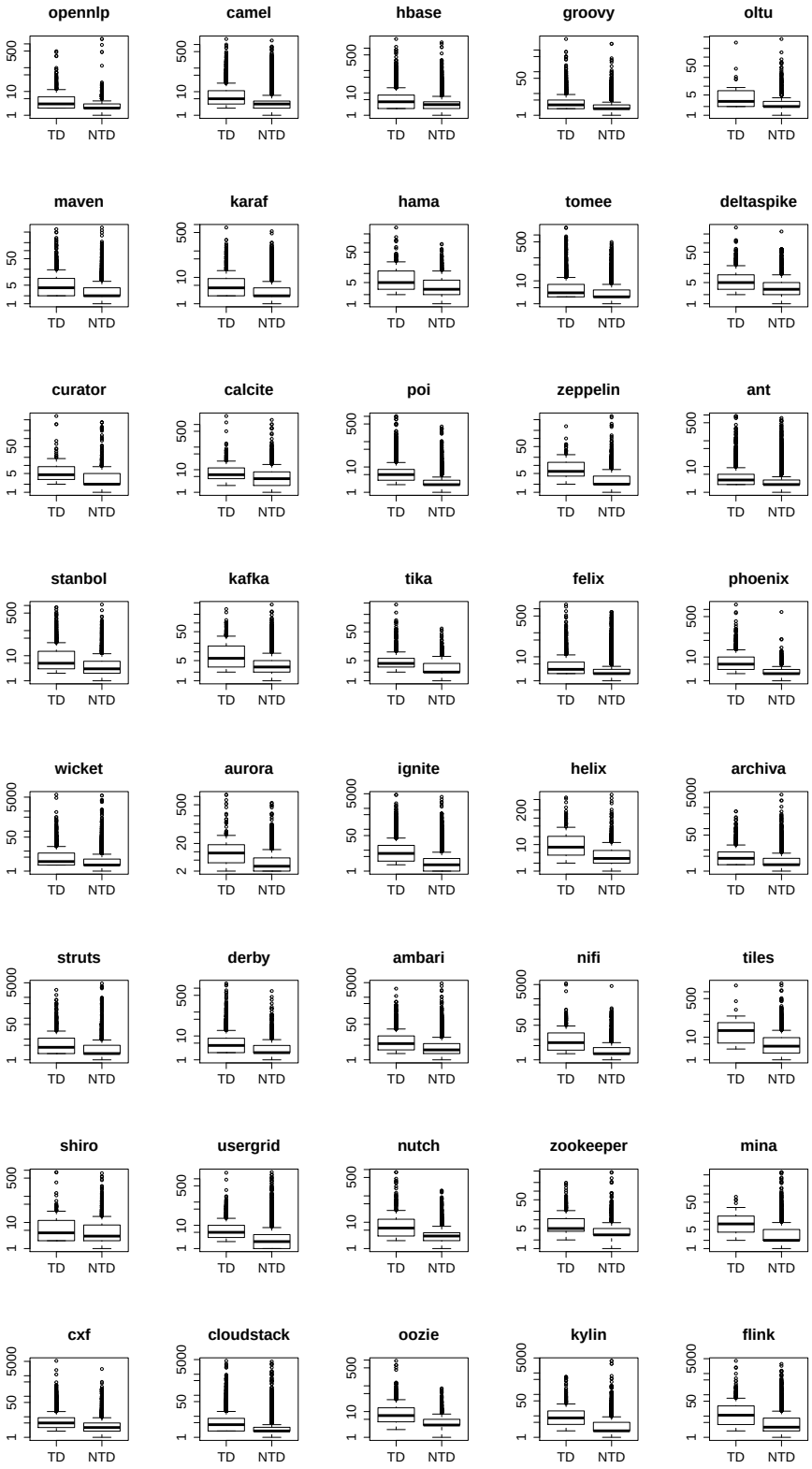

Figure 28: Total number of files modified per change (SATD vs. NSATD).

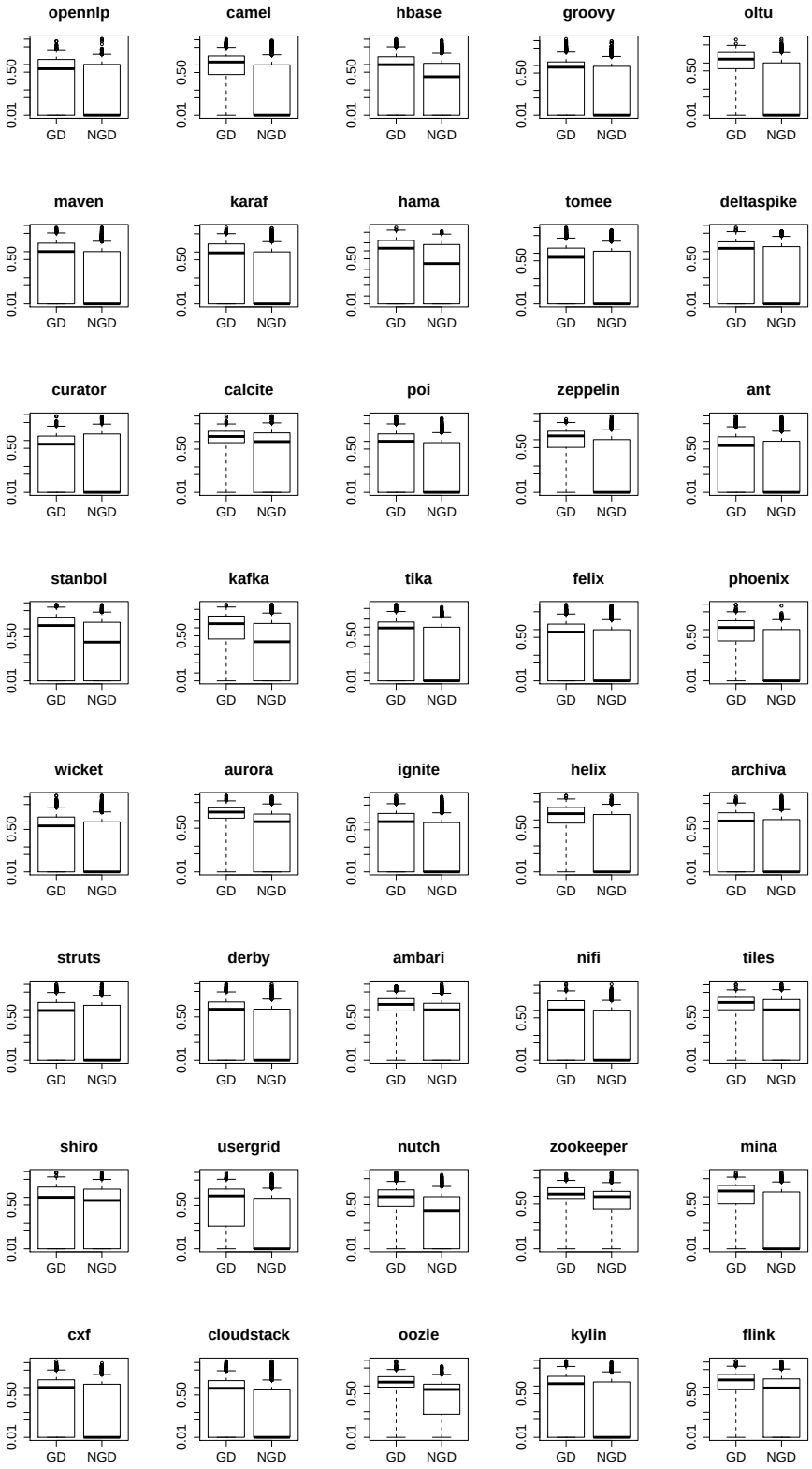

Figure 29: Total number of entropy modified per change (GOD vs. NGOD).

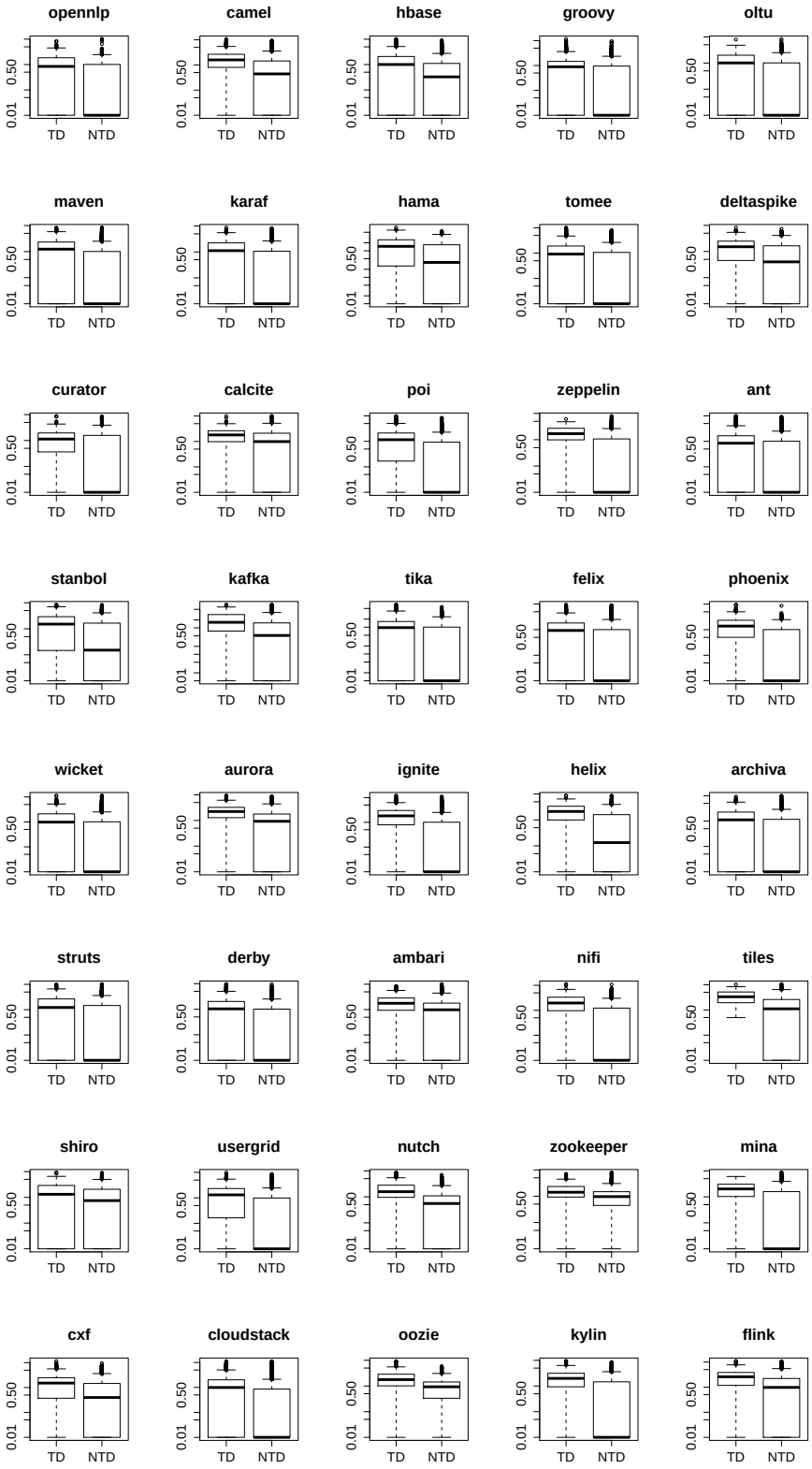

Figure 30: Total number of entropy modified per change (SATD vs. NSATD).

Bibliography

[ARC+14] Nicolli SR Alves, Leilane F Ribeiro, Vivyane Caires, Thiago S Mendes, and Rodrigo 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.

[BK95] James M. Bieman and Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. *SIGSOFT Softw. Eng. Notes*, 20(SI):259–262, August 1995.

[BR16] Gabriele Bavota and Barbara 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.

[CK94] 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.

[Cun92] Ward Cunningham. The wycash portfolio management system. *SIGPLAN OOPS Mess.*, 4(2):29–30, December 1992.

[Dan] Al Danial. Cloc - count lines of code.

[DLDPO11] 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.

[EGK+01] 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. *IEEE Transactions on Software Engineering*, 27(1):1–12, Jan 2001.

[FB99] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code*. Component software series. Addison-Wesley, 1999.

[Fit96] Ronan Fitzpatrick. Software quality: definitions and strategic issues. *Reports*, page 1, 1996.

[Fow07] Martin Fowler. Technical debt quadrant, 2007.

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

[FZMM13] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mntyl. Code smell detection: Towards a machine learning-based approach. In *2013 IEEE International Conference on Software Maintenance*, pages 396–399, Sept 2013.

[GK05a] Robert J. Grissom and John J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition edition, 2005.

[GK05b] Robert J Grissom and John J Kim. *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers, 2005.

[GSG+11] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Q. B. Da Silva, A. L. M. Santos, and C. Siebra. An exploratory case study tracking technical debt. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 528–531, Sept 2011.

[Has09] Ahmed 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.

[JCMB08] Yue Jiang, Bojan Cuki, Tim Menzies, and Nick 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.

[KNO12] P. Kruchten, R. L. Nord, and I. Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, Nov 2012.

[KNOF13] Philippe Kruchten, Robert L Nord, Ipek Ozkaya, and Davide 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.

[KSA+13] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Trans. Software Eng.*, 39(6):757–773, 2013.

[KWR10] Ninus Khamis, René Witte, and Juergen 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.

[KWZ08] Sunghun Kim, E. James Whitehead, Jr., and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, 34(2):181–196, 2008.

[LDM07] 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.

[LFB06] Dawn J Lawrie, Henry Feild, and David Binkley. Leveraged quality assessment using information retrieval techniques. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 149–158. IEEE, 2006.

[LTS12] Erin Lim, Nitin Taksande, and Carolyn Seaman. A balancing act: what software practitioners have to say about technical debt. *IEEE Software*, 29(6):22–27, 2012.

[Mar01] 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.

[Mar04] Radu 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.

[Mar05] R. Marinescu. Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 701–704, Sept 2005.

[Mar12] 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.

[McC76] Thomas 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.

[McC07] Steve McConnell. Technical debt, 2007.

[MCT+08] Haroon Malik, Istehad Chowdhury, Hsiao-Ming Tsou, Zhen Ming Jiang, and Ahmed 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.

[MPS08] Raimund Moser, Witold Pedrycz, and Giancarlo 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.

[MS15] Everton da S Maldonado and Emad Shihab. Detecting and quantifying different types of self-admitted technical debt. In *IEEE 7th International Workshop on Managing Technical Debt (MTD), 2015*, pages 9–15. IEEE, 2015.

[MST17] Everton Maldonado, Emad Shihab, and Nikolaos Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering (TSE)*, page To Appear, 2017.

[MVL03] Mika Mantyla, Jari Vanhanen, and Casper 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.

[MVL04] Mika V Mantyla, Jari Vanhanen, and Casper Lassenius. Bad smells-humans as code critics. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 399–408. IEEE, 2004.

[MW47] Henry B Mann and Donald 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.

[NB05] Nachiappan Nagappan and Thomas 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.

[PS14] Aniket Potdar and Emad 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.

[RD13] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 432–441, 2013.

[SNKO15] Carolyn Seaman, Robert L Nord, Philippe Kruchten, and Ipek 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.

[SRB+08] Margaret-Anne Storey, Jody Ryall, R. Ian Bull, Del Myers, and Janice 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.

[Ste10] C. Sterling. *Managing Software Debt: Building for Inevitable Change*. Pearson Education, 2010.

[SZV+13] Rodrigo O Spínola, Nico Zazworka, Antonio Vetrò, Carolyn Seaman, and Forrest 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.

[SZZ05a] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[ŚZZ05b] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[TGM96] Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4(3):143–167, 1996.

[TMTL12] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary 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.

[TYKZ07] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* iComment: Bugs or bad comments? */. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP07)*, October 2007.

[TZP11] Lin Tan, Yuanyuan Zhou, and Yoann 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.

[ZNZ08] Thomas Zimmermann, Nachiappan Nagappan, and Andreas Zeller. *Predicting Bugs from History*, chapter Predicting Bugs from History, pages 69–88. Springer, February 2008.

[ZSSS11] Nico Zazworka, Michele A Shaw, Forrest Shull, and Carolyn 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.

[ZSV+13] Nico Zazworka, Rodrigo O. Spínola, Antonio Vetro', Forrest Shull, and Carolyn 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.