2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering

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

Sultan Wehaibi∗, Emad Shihab∗ and Latifa Guerrouj‡

∗Department of Computer Science and Software Engineering
Concordia University
Montreal, Canada
{s_alweha,eshihab}@encs.concordia.ca

‡Département de Génie Logiciel et des TI
École de Technologie Supérieure
Montréal, Canada
latifa.guerrouj@etsmtl.ca

***Abstract*—Technical debt refers to incomplete or temporary workarounds that allow us to speed software development in the short term at the cost of paying a higher price later on. Recently, studies have shown that technical debt can be detected from source code comments, referred to as self-admitted technical debt. Researchers have examined the detection, classification and removal of self-admitted technical debt. However, to date there is no empirical evidence on the impact of self-admitted technical debt on software quality.**

Therefore, in this paper, we examine the relation between self-admitted technical debt and software quality by investigating whether (i) files with self-admitted technical debt have more defects compared to files without self-admitted technical debt, (ii) whether self-admitted technical debt changes introduce future defects, and (iii) whether self-admitted technical debt-related changes tend to be more difficult. We measured the difficulty of a change using well-known measures proposed in prior work such as the amount of churn, the number of files, the number of modified modules in a change, as well as the entropy of a change. An empirical study using five open source projects, namely Hadoop, Chromium, Cassandra, Spark and Tomcat, showed that: i) there is no clear trend when it comes to defects and self-admitted technical debt, although the defectiveness of the technical debt files increases after the introduction of technical debt, ii) self-admitted technical debt changes induce less future defects than none technical debt changes, however, iii) self-admitted technical debt changes are more difficult to perform, i.e., they are more complex. Our study indicates that although technical debt may have negative effects, its impact is not only related to defects, rather making the system more difficult to change in the future.

I. Introduction

Software companies and organizations have a common goal while developing software projects - to deliver high-quality, useful software in a timely manner. However, in most practical settings developers and development companies are rushed to meet deadlines, rushing them to release. Such situations are all too common and in many cases force developers to take shortcuts [1] [2]. Recently, the term *technical debt* was coined to represent the phenomena of "doing something that is beneficial in the short term but will incur a cost later on" [3]. Prior work showed that there are many different reasons why practitioners take on technical debt. These reasons include: a rush in delivering a software product given a tight schedule, deadlines to incorporate with a partner product before release, time-to-market pressure, as well as meeting customer needs in a timely fashion [4].

More recently, a study by Potdar and Shihab [5] introduced a new way to identify technical debt through source code comments, referred to as self-admitted technical debt (SATD). SATD is technical debt that developers themselves report through source code comments. Prior work [6] showed that SATD is common in software projects and can be used to identify different types of technical debt (e.g., design, defect, and requirement debt).

Intuition and general belief indicate that such rushed development tasks (also known as technical debt) negatively impact software maintenance and overall quality [1], [2], [7]–[9]. However, to the best of our knowledge, there is no empirical study that examines the impact of SATD on software quality. Such a study is critical since (i) it will help us confirm or refute intuition and (ii) help us better understand how to manage SATD.

Therefore, in this paper, we empirically investigate the relation between SATD and software quality in five open-source projects, namely Chromium, Hadoop, Spark, Cassandra, and Tomcat. In particular, we examine whether (i) files with SATD have more defects compared to files without SATD, (ii) whether SATD changes introduce future defects, and (iii) whether SATD-related changes tend to be more difficult. We measured the difficulty of a change in terms of the amount of churn, the number of files, the number of modified modules in a change, as well as, entropy of a change. We perform our study on five open-source projects, namely Chromium, Hadoop, Spark, Cassandra, and Tomcat. Our findings show that: i) there is no clear relationship between defects and SATD. In some of the studied projects however, SATD files have more bug-fixing changes, while in other projects, files without SATD have more defects, ii) SATD changes are associated with less future defects than none technical debt changes, however, iii) SATD changes (i.e., changes touching SATD files) are more difficult to perform. Our study indicates that although technical debt may have negative effects, its impact is not related to defects, rather its impact is in making the system more difficult to change in the future.

The rest of the paper is organized as follows. Section II

978-1-5090-1855-0/16 $31.00 © 2016 IEEE
DOI 10.1109/SANER.2016.72

summaries the related work. In Section III, we describe our research methodology. Section IV presents and discusses the results of our empirical evaluation, while Section V shows some threats to validity related to our study. Finally, Section VI concludes our paper.

II. Related Work

Since our work focuses on SATD, which analyzes comments to detect technical debt, we discuss the work related to three main topics: (i) source code comments, (ii) technical debt, and (iii) software quality.

A. Research Leveraging 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 [10]–[12]. For example, Storey *et al.* [13] explored how task annotations in source code help developers manage personal and team tasks. Takang *et al.* [10] 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.* [14] assessed the quality of source code documentation based on the analysis of the quality of language and consistency between source code and its comments. Tan *et al.* proposed several approaches to identify inconsistencies between code and comments. The first called, @iComment, detects lock- and call-related inconsistencies [11]. The second approach, @aComment, detects synchronization inconsistencies related to interrupt context [15]. A third approach, @tComment, automatically infers properties form Javadoc related to null values and exceptions; it performs test case generation by considering violations of the inferred properties [16].

Other studies examined the co-evolution and reasons for comment updates. Fluri *et al.* [17] studied the co-evolution of source code and their associated comments and found that 97% of the comment changes are consistently co-changed. Malik *et al.* [18] performed a large empirical study to understand the rationale for updating comments along three dimensions: characteristics of a 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.* [19] 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, confirms the conjecture that providing developers with similarity between source code and high-level software artifacts helps to enhance the quality of comments and identifiers.

Most relevant to our work is the recent work by Potdar and Shihab [5] that 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, as well as the likelihood of its removal after introduction. Another relevant contribution to our study is the one by Maldonado and Shihab [6], who have also leveraged source code comments to detect and quantify different types of SATD. They classified SATD into five types, *i.e.*, design debt, defect debt, documentation debt, requirement debt and test debt. They found that the most common type is design debt, making up between 42% to 84% of a total of 33K classified comments.

Our study builds on the prior work in [5], [6] since we use the comment patterns they produced to detect SATD. However, different from their studies, we examine the relationship between SATD and software quality.

B. Technical Debt

Other work focused on the identification and examination of technical debt. It is important to note here 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.* [20] attempted to automatically identify technical debt and then compared their automated identification with human elicitation. The results of their study outline potential benefits of developing tools and techniques for the detection of technical debt. Also, Zazworka *et al.* [7] investigated how design debt, in the form of god classes, affects the software maintainability and correctness of software products. Their study involved two industrial applications and showed that god classes are changed more often and contain more defects than non-god classes. Their findings suggests that technical debt may negatively influence software quality. Guo *et al.* [9] analyzed how and to what extent technical debt affects software projects by tracking a single delayed task in a software project throughout its lifecycle. As discussed earlier, the work by Potdar and Shihab [5] is also related to our work, however, its main difference compared to prior work is that it focused on SATD.

Our work differs from past research by Zazworka *et al.* [7], [20] 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 impact of the SATD and software quality.

C. Software Quality

A plethora of prior work proposed techniques to improve software quality. The majority of this work focused on understanding and predicting software quality issues (e.g. [21]). Several studies examined the metrics that best indicate software defects including design and code metrics [22], code churn metrics [23], and process metrics [24], [25].

Other studies focused on change-level prediction of defects. Sliwerski *et al.* suggested a technique called, SZZ, to automatically locate fix-inducing changes by linking a version archive to a bug database [26]. Kim *et al.* [27] used identifiers in added and deleted source code and the words in change logs to identify changes as defect-prone or not. Similarly, Kamei [28] proposed a "Just-In-Time Quality Assurance" approach

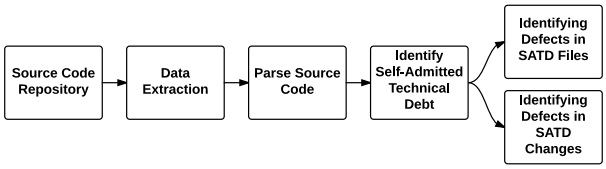

Fig. 1: Approach overview.

to identify, in real-time, risky software changes. The findings of their study reveal that process metrics outperform product metrics for the purpose 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 works since it examines relationship of SATD and software defects.

III. Methodology

The goal of our study is to investigate the relationship between SATD and software quality. We measure software quality in two ways. First, we use the traditional measure, which is used in most prior studies, defects in a file and defect-inducing changes [27]–[29]. 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 phenomena of taking a short term benefit at a cost of paying a higher price later on, we also use the difficulty of the changes related to SATD. In particular, we use the churn, the number of files, the number of directories and the entropy of a change as a measure of 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 the 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 approach shown in Figure 1, 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 comments from the source code of the analyzed systems (step 3). We apply the comment patterns proposed by Potdar and Shihab [5] to identify SATD (step 4). Then, we analyze the changes to quantify defects in files and use the SZZ algorithm to determine defect-inducing changes (step 5).

A. Data Extraction

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

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

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

B. Scanning Code and Extracting Comments

After obtaining the source code of the software projects, we extracted the comments from the source code files of each studied project. To this aim, we developed a python-based tool that identifies comments based on the use of regular expressions. This tool also indicates the type of a comment (*i.e.*, single-line or block comments). In addition, the tool shows, for each comment, the name of the file where the comment appears, as well as the line number of the comment. To ensure the accuracy of our tool, we use the Count Lines of Code (CLOC) tool [30]. CLOC counts the total number of lines of comments, which was equal to the number provided by the tool that we developed.

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 comments the number of SATD comments is, 18,435 comments for Chromium, 2,442 for Hadoop, 1,205 for Spark, 550 for Cassandra, and 1,543 for Tomcat. To enable easy processing of our data, we store all of our processed data in a PostgreSQL database and query the database to answer our RQs.

C. Identifying Self-Admitted Technical Debt

To perform our analysis, we need to identify SATD at two levels: (i) file level and (ii) change level.

SATD files: To identify SATD, we followed the methodology applied by Potdar and Shihab [5], which uses patterns indicating the occurrence of SATD. In their work, Potdar and Shihab [5] came up with a list of 62 different patterns that indicate SATD. Therefore, in our approach, we determine the comments that indicate SATD by searching if they contain any of the 62 patterns that indicate SATD. These patterns are extracted from several projects and some patterns appear more

TABLE I: 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

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 made available online[1].

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

SATD changes: To study the impact of SATD at the change level, we need to identify SATD changes. To do so, we use our SATD files to determine the SATD changes. We analyze the changes and determine all the files that were touched by that change. If one or more of the files touched by the change is (are) SATD file(s), then we label that change as an SATD change. If the change does not touch an SATD file, then we label it as a non-SATD change. Table II shows the percentage of SATD comments and files for each of the studied systems. From the table, we see that SATD comments make up less than 4% of the total comments and between 10.17 - 20.14% of the files are SATD files.

TABLE II: 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

D. Identifying Defects in SATD Files and SATD Changes

To determine whether a change fixes a defect, we search, using regular expressions, in change logs from the Git Version control system for co-occurrences of defect identifiers with keywords like "fixed issue #ID", "bug ID", "fix", "defect", "patch", "crash", "freeze", "breaks", "wrong", "glitch", "properly", "proper". Sliwersky *et* al. [29] showed that the use of such key words in the change logs usually refers to the correction of a mistake or failure. A similar approach was applied to identify fault-fixing and fault-inducing changes in prior works [27]–[29]. 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 relevant information from each report.

[1]http://users.encs.concordia.ca/~eshihab/data/ICSME2014/data.zip
[2]https://www.bugzilla.org
[3]https://www.atlassian.com/software/jira

Once we identify the SATD files and SATD changes, our next step is to identify the defects in each. To do so, we follow the approaches used in past research to determine the number of defects in a file and to identify defect-inducing changes [27]–[29].

Defects in files: In order to compare the defectiveness of SATD and non-SATD files, we need to determine the number of defects that exist in a file. To do so, we extract all the changes that touched a file through the entire history of the system. Then, we search for keywords in the change logs that are indicative of defect fixing. A subset of these words that we used involves: "fixed issue #ID", "bug ID", "fix", "defect", "patch", "crash", "freeze", "breaks", "wrong", "glitch", "proper". In the case where a defect identification is specified, we extract the defect report to make sure that the defect corresponds to the system (*i.e.*, product) we are studying, since some communities (*e.g.*, Apache) use the same issue tracking system for multiple products. Second, we verify 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 to search the change logs (i.e., commit messages) from the source code control versioning system specific keywords that indicate a fix. 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 changed the fixed code in the past. Then, we determine the defect-inducing change as the change that is closest and 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 similar to prior work [28], 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 [29] or approximate (ASZZ) algorithm [28] and to-date is the state-of-the-art in identifying defect-inducing changes.

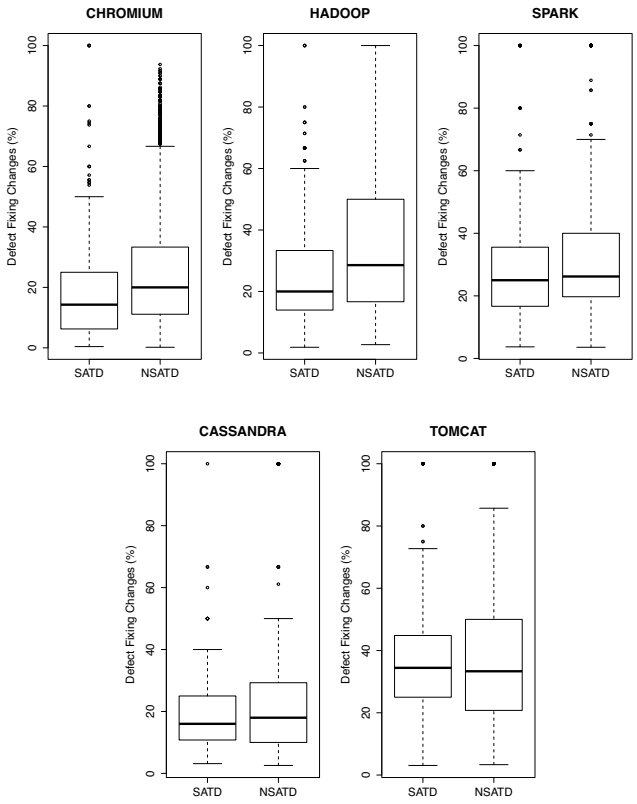

Fig. 2: Percentage of defect fixing changes for SATD and NSATD files.

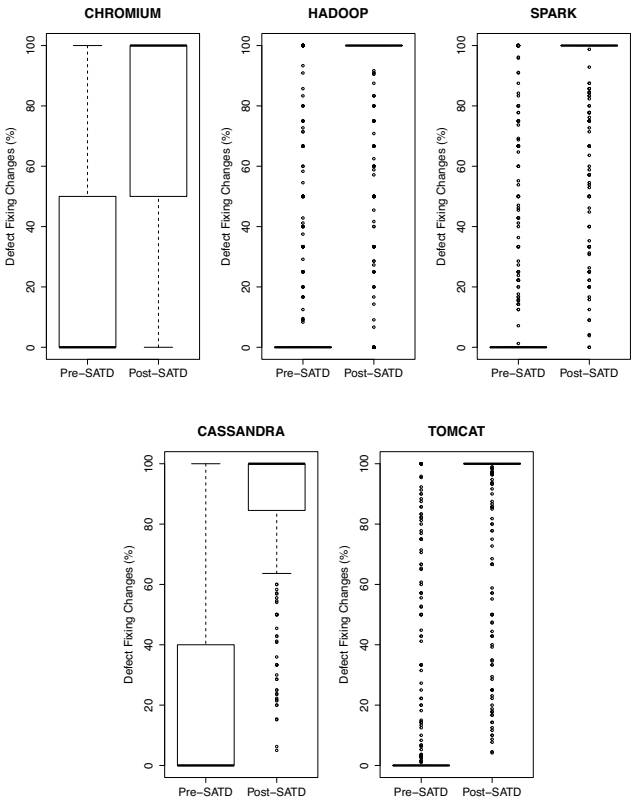

Fig. 3: Percentage of defect fixing changes for pre-SATD and post SATD.

IV. Case Study Results

This section reports the results of our empirical study that examines the relationship between self-admitted technical debt and software quality. For each project, we provide the descriptive statistics and statistical results, as well as a comparison with the other considered projects.

In the following we present for each RQ, its motivation, the approach followed to address it, as well as its findings.

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

Motivation: Intuitively, technical debt has a negative impact on software quality. Researchers examined technical debt and showed that it negatively impacts software quality [7]. However, this study did not focus on SATD, which is prevalent in software projects according to past research [5].

Empirically examining the impact of SATD on software quality provides researchers and practitioners with a better understanding of such SADT, warns them about its future risks, and makes them aware about the obstacles or challenges it can pose.

In addition to comparing the defect-proneness of SATD and non-SATD files, we also 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. In essence, it tells us if the introduction of SATD relates to defects.

Approach: To address RQ1, we perform two types of analyses. First, we compare files in terms of the defect-proneness of files that contain SATD with files that do not contain SATD. Second, for the SATD files, we compare their defect-proneness before and after the SATD is introduced.

Comparing SATD and non-SATD files. To perform this analysis, we follow the procedure outlined earlier in Section III-C to identify SATD files. In a nutshell, we determine files that contain SATD comments and label them as SATD files. Files that do not contain any SATD are labeled as non-SATD files. Once we determine these files, we determine the percentage of defect-fixing changes in each (SATD and non-SATD) file. We use the percenatages instead of raw numbers since files can have a different number of changes, hence using the percentage normalizes our data. To answer the first part of RQ1, we plot the distribution of defects in each of the SATD and non-SATD file sets and perform statistical tests to compare their differences.

To compare the two sets, we use the Mann-Whitney [31] test to determine if a statistical difference exists and Cliff's

TABLE III: Cliff's Delta for SATD versus NSATD and POST versus PRE fixing changes.

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

delta [32] 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 does not assume a normal distribution (and as we will see later, our data is not normally distributed). We consider the results of the Mann-Whitney test to be statistically significant if the p-value is below $p <= 0.05$. In addition, we computed the effect-size of the difference using the Cliff's delta (d) non-parametric effect size measure, which measures how often values in a distribution are larger than the values in a second distribution. Cliff's d ranges in the interval $[-1, 1]$ and is considered small for $0.148 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$.

Comparing files pre- and post- SATD. To compare SATD files pre- and post- SATD, we determine all the changes that occurred to a file and identify the change that introduced the SATD. Then, 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 the percentage of defects instead of the 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-introducing 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 2 shows boxplots of the percentage of defect fixing changes in SATD and non-SATD files for the five projects. We observe that in all 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 the projects, the p-values were < 0.05, indicating that the difference is statistically significant. However, when we closely examine the Cliff's delta values in Table III, we see a different trend for Chromium. In Chromium and Tomcat, SATD files often have higher defect percentages than non-SATD files and the effect size is medium for Chromium and small for Tomcat. On the other hand in Hadoop, Cassandra and Spark, SATD files have lower defect percentages than non-SATD files and this effect is large for Hadoop, medium for Cassandra and small for Spark.

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

Results - Defects in pre- and post- SATD: Figure 3 shows the boxplots for the percentage of defect-fixing changes in SATD files, pre- and post- SATD. Not surprisingly, the percentage of defect-fixing changes in all projects is higher for post-SATD. Table III shows that the effect size Cliff's delta values also confirm our visual observations that there is more defect fixing post- SATD compared to pre- SATD in the SATD files. For all the projects except Hadoop and Cassandra, the Cliff's delta is large. For Hadoop and Cassandra the Cliff's delta effect size is small

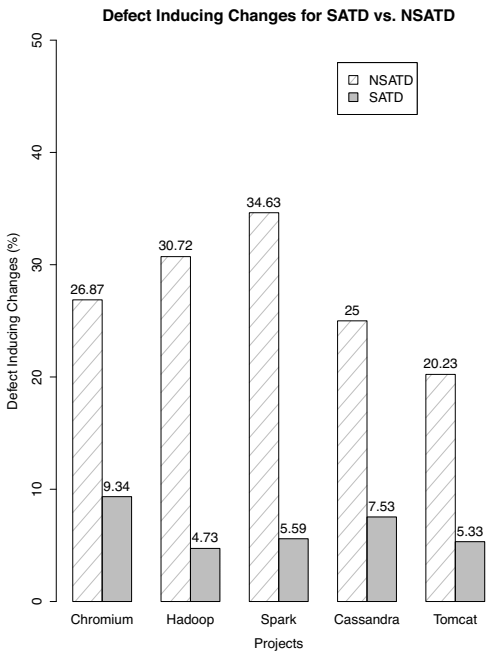

Fig. 4: Percentage of defect inducing changes with SATD and NSATD.

This findings shows that although it is not always clear that SATD files will have a higher percentage of defects compared to non-SATD files, there is a clear trend that shows that once the SATD is introduced, there is a higher percentage of defect-fixing.

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 see if the SATD changes are more likely to introduce future defects. In contrast to the file-level analysis which looks at files as a whole, our analysis here is more fine grained since it looks at the individual changes.

Studying the potential of SATD changes to introduce future defects is important since it allows us to explore (i) how SATD changes compare in terms of future introduction of defects to non-SATD changes and (ii) how quickly the impact of SATD on quality can be felt. For example, if SATD changes introduce defects in the immediate next change, then this tells us that the impact of SATD is felt very quickly. Our conjecture is that SATD changes tend to be more complex and lead to the introduction of defects.

TABLE IV: Cliff's Delta for the change difficulty measures across the projects.

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

Approach: To address RQ2, we applied the SZZ algorithm [29] to find defect-inducing changes. Then, we determined which of the defect-inducing changes are also SATD changes. We also count the number of defect-inducing changes that are non-SATD.

Once the defect-inducing changes are identified, we divided the data into two groups, *i.e.*, defect-inducing changes that are also SATD and defect-inducing changes that are non-SATD.

Results: Figure 4 shows that non-SATD changes have a higher percentage of defect-inducing changes compared to non-SATD changes. The figure shows that for Chromium for example, approximately 10% of the SATD changes induce future defect. On the other hand, approximately 27% of the non-SATD changes in Chromium induce future defects. Our findings here show that contrary to our conjecture, SATD changes have a lower chance of inducing future defects

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

Motivation: Thus far, our analysis has focused on the relationship between SATD and software defects. However, by definition, technical debt mentions that it provides a tradeoff where a short term benefit ends up costing more in the future. Therefore, we would like to empirically examine this tradeoff by examining whether changes after the introduction of technical debt become more difficult to perform.

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

Approach: To answer this question, we classify the changes into two groups, *i.e.*, SATD and non- SATD changes. Then, we compare the difficulty of each set of changes. To measure the difficulty of a change we use four different measures: the total number of modified lines (*i.e.*, churn) in the change, the number of modified directories, the number of modified files and change entropy. The first three measures are motivated by the earlier work on software decay by Eick *et al.* [33], which uses these three measures to measure decay. The change entropy measure is motivated by the work by Hassan [34], which used change entropy as a measure of 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, we simply aggregate the churn of the individual files to determine the churn of the change. The list of files is extacted from the change log to determine the number of files and directories touched by the change. When measuring

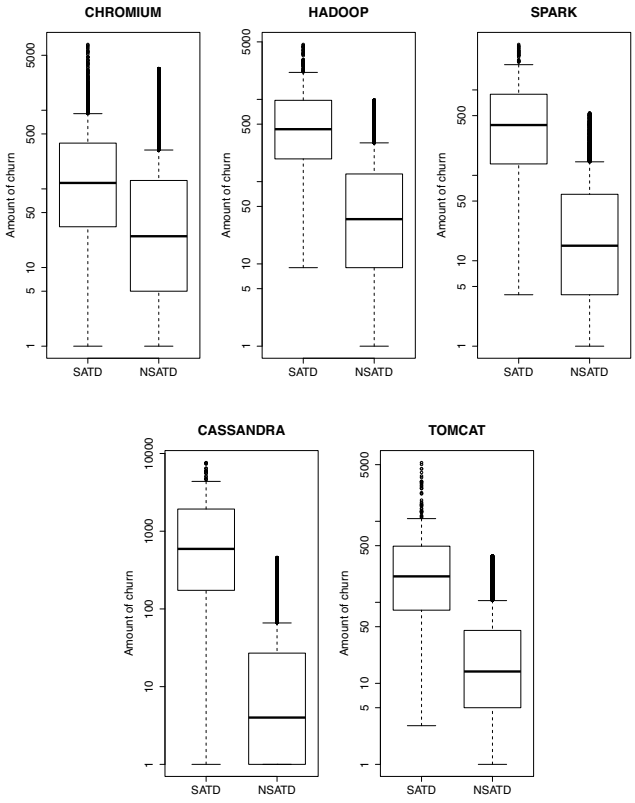

Fig. 5: Total number of lines modified per change (SATD vs. NSATD).

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

To measure the entropy of the change, we use the change complexity measure proposed by Hassan [34]. 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 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 [34], the above Entropy formula has been normalized by the maximum Entropy $log_2 n$ to account for differences in the number of files for different changes. The higher the normalized entropy is, the more difficult the change is.

Results: Figures 5, 6, 7, 8 shows that for all difficulty measures, SATD changes have a higher value than non-SATD changes. We also find that the difference between the

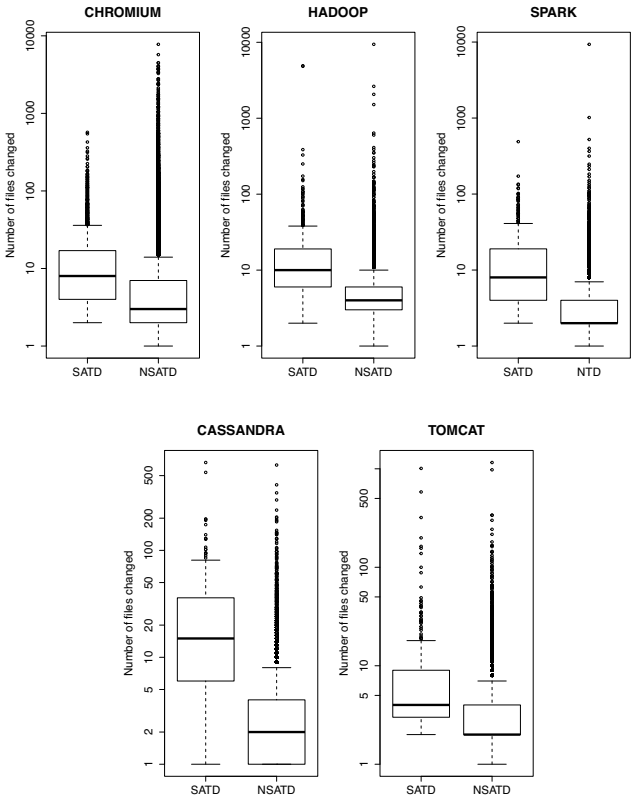

Fig. 6: Total number of files modified per change (SATD vs. NSATD).

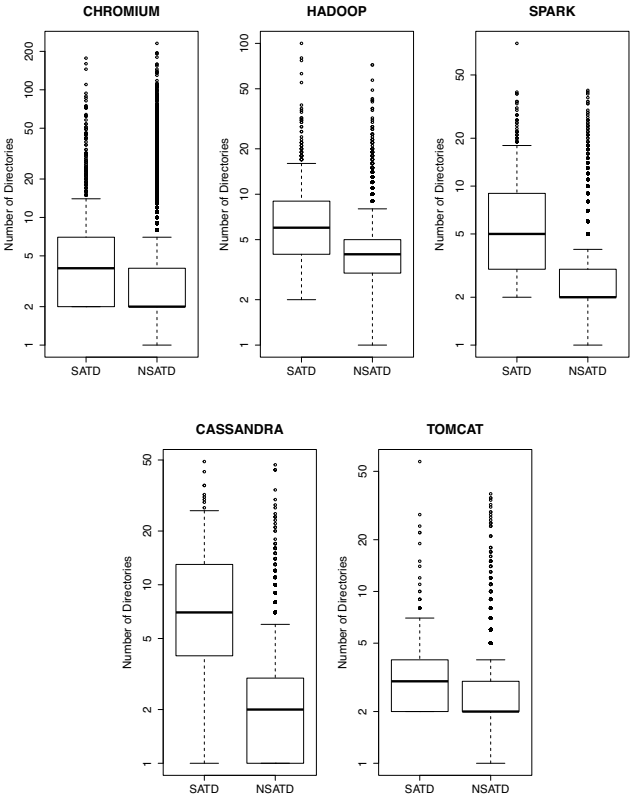

Fig. 7: Total number of modified directories per SATD and NSATD change.

SATD and non-SATD changes is statistically significant, with a $p - value < 0.05$. Table IV shows the Cliff's delta effect size values for all studied projects. We observe that in all projects and for all measures of difficulty the effect size is either medium or large (Cf. Table IV), which indicates that SATD changes are more difficult than non-SATD changes.

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

V. Threats To Validity

Threats to **internal validity** concern any confounding factors that could have influenced our study results. To identify SATD, we use source code comments. In some cases, developers may not add comments when they introduce technical debt. Another threat is that developers may introduce technical debt, remove it and not remove the comment related to that debt, i.e., the code and comment change inconsistently. However, Potdar and Shihab [5] examined this phenomena in Eclipse and found that in the 97% of the cases code and comments consistently change. To identify the SATD, we use the comments provided by Potdar and Shihab [5]. 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 may be perceived as an indicator of SATD by other developers. To mitigate this threat, the first author manually examined each comment that we detected and verified whether it contains patterns from the 62 patterns investigated in [5]. We performed this step, independently, for each of the five studied projects. When identifying a change as SATD change we consider a change to be SATD change when it contains at least one SATD file. Another way is to classify a change as an SATD change only when all files have SATD. The reason we chose to do it this way is because sometimes even SATD in one file can impact the rest of the change, e.g., cause many other files to be changed. When measuring the percentage of defects for files after SATD was introduced, it is difficult to observe if the difference was due to the introduction of SATD or the natural evaluation of the files.

Threats to **external validity** concern the possibility that our results may not generalize. To make our results as generalizable as possible, we analyzed five large open-source systems, *i.e.*, Chromium, Hadoop, Spark, Cassandra, and Tomcat. Our data comes from well-established, mature codebase of open-source software projects with well-commented source code.

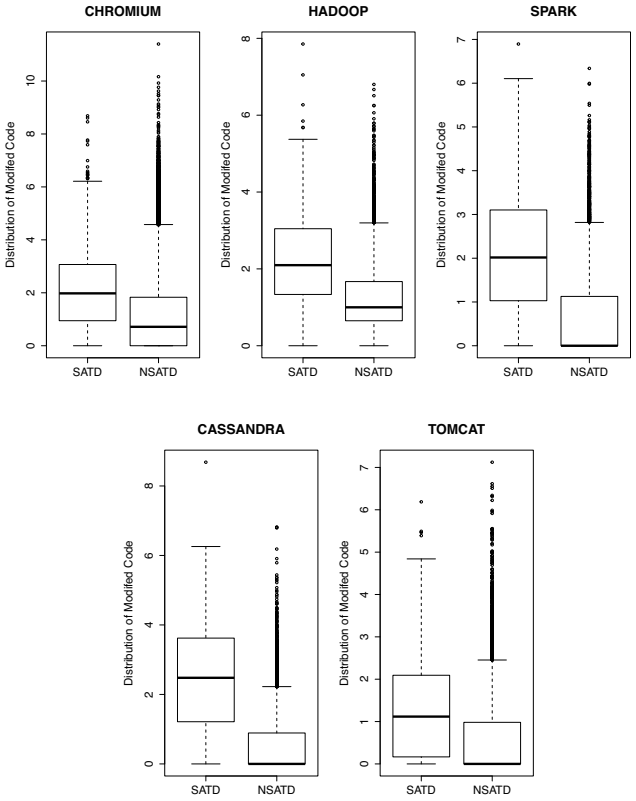

Fig. 8: Distribution of the change across the SATD and NSATD files.

These projects belong to different domains, and they are written in different programming languages.

Furthermore, we focused on SATD only, which means that we do not cover all technical debt and therefore there may be other technical debt that is not self-admitted. Studying all technical debt is out of the scope of this work.

VI. Conclusion And Future Work

Technical debt is intuitively known as a bad practice by software companies and organizations. However, there is very little empirical evidence on the extent to which technical debt can impact software quality. Therefore, in this paper we perform an empirical study, using five large open-source projects, to determine how technical debt relate to software quality. We focus on self-admitted technical debt that refers to errors that might be introduced due to intentional quick or temporary fixes. As in [5], we identify such debt following a methodology that leverages source code comments to distinguish it based on the use of patterns indicating the existence self-admitted technical debt.

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

To perform our study, we analyzed five open-source projects, namely Chromium, Hadoop, Spark, Cassandra, and Tomcat. Our findings show that there is no clear trend when it comes to defects and self-admitted technical debt. In some of the studied projects, self-admitted technical debt files have more bug-fixing changes, while in other projects, files without it had more defects. We also found that self-admitted technical debt changes are less associated with future defects than none technical debt changes, however, we showed that self-admitted technical debt changes are more difficult to perform. Our study indicates that although technical debt may have negative effects, its impact is not related to defects, rather making the system more difficult to change in the future.

We bring empirical evidence on the fact that technical debt may have some negative implications on the software development process in particular by making it more complex. Hence, practitioners need to manage it properly to avoid any consequences. In the future, we plan to further study the nature of the SATD files after they became defective.

References

[1] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi, "Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 5, pp. 51–54, 2013.

[2] C. Seaman, R. L. Nord, P. Kruchten, and I. Ozkaya, "Technical debt: Beyond definition to understanding report on the sixth international workshop on managing technical debt," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 2, pp. 32–34, 2015.

[3] W. Cunningham, "The wycash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.

[4] E. Lim, N. Taksande, and C. Seaman, "A balancing act: what software practitioners have to say about technical debt," *Software, IEEE*, vol. 29, no. 6, pp. 22–27, 2012.

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

[6] E. da_S._Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical debt," in *Proc. MTD*. IEEE, 2015, pp. 9–15.

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

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

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

[10] A. A. Takang, P. A. Grubb, and R. D. Macredie, "The effects of comments and identifier names on program comprehensibility: an experimental investigation." *J. Prog. Lang.*, no. 3, pp. 143–167.

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

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

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

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

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

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

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

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

[19] A. De Lucia, M. Di Penta, and R. Oliveto, "Improving source code lexicon via traceability and information retrieval," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 205–227, 2011.

[20] N. Zazworka, R. O. Spínola, A. Vetro', F. Shull, and C. Seaman, "A case study on effectively identifying technical debt," in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, 2013, pp. 42–47.

[21] T. Zimmermann, N. Nagappan, and A. Zeller, *Predicting Bugs from History*. Springer, February 2008, ch. Predicting Bugs from History, pp. 69–88.

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

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

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

[25] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 432–441.

[26] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, May 2005.

[27] S. Kim, E. J. Whitehead, Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, 2008.

[28] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance." *IEEE Trans. Software Eng.*, vol. 39, no. 6, pp. 757–773, 2013.

[29] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

[30] Cloc - count lines of code. [Online]. Available: http://cloc.sourceforge.net/

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

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

[33] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *Software Engineering, IEEE Transactions on*, vol. 27, no. 1, pp. 1–12, 2001.

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