2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

A Large-Scale Empirical Study on Self-Admitted Technical Debt

Gabriele Bavota, Barbara Russo
Free University of Bozen-Bolzano, Bolzano, Italy
{gabriele.bavota, barbara.russo}@unibz.it

ABSTRACT

Technical debt is a metaphor introduced by Cunningham to indicate "not quite right code which we postpone making it right". Examples of technical debt are code smells and bug hazards. Several techniques have been proposed to detect different types of technical debt. Among those, Potdar and Shihab defined heuristics to detect instances of self-admitted technical debt in code comments, and used them to perform an empirical study on five software systems to investigate the phenomenon. Still, very little is known about the diffusion and evolution of technical debt in software projects.

This paper presents a *differentiated replication* of the work by Potdar and Shihab. We run a study across 159 software projects to investigate the diffusion and evolution of self-admitted technical debt and its relationship with software quality. The study required the mining of over 600K commits and 2 Billion comments as well as a qualitative analysis performed via open coding.

Our main findings show that self-admitted technical debt (i) is diffused, with an average of 51 instances per system, (ii) is mostly represented by code (30%), defect, and requirement debt (20% each), (iii) increases over time due to the introduction of new instances that are not fixed by developers, and (iv) even when fixed, it survives long time (over 1,000 commits on average) in the system.

CCS Concepts

•Software and its engineering → Software evolution; Maintaining software;

Keywords

Mining Software Repositories, Technical Debt, Empirical Software Engineering

MSR'16, May 14-15, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2901739.2901742

1. INTRODUCTION

Ward Cunningham coined the technical debt metaphor back in 1993 [14] to explain the unavoidable interests (*i.e.,* maintenance and evolution costs) developers pay while working on *not-quite-right code*, possibly written in a rush to meet a deadline or to deliver the software to the market in the shortest time possible [7, 14, 22, 26, 33].

In the last years researchers have studied the technical debt phenomenon from different perspectives. Several authors developed techniques and tools aimed at detecting specific types of technical debt, like code smells (see *e.g.,* [28, 29]) and coding style violations (see *e.g.,* [1]). Also, researchers have studied the impact of different types of technical debt on maintainability attributes of software systems [20, 40] and when and why technical debt instances are introduced in software systems [36].

Recently, Potdar and Shihab [30] pioneered the study of *self-admitted technical debt* (SATD), referring to technical debt instances intentionally introduced by developers (*e.g.,* temporary patches to fix bugs) and explicitly documented in code comments. They showed how it is possible to detect instances of technical debt by simply mining code comments looking for patterns likely indicating (*i.e.,* self-admitting) the presence of technical debt (*e.g., fixme*, *todo*, *to be fixed*, *etc.*). They performed an exploratory study on five software systems aimed at understanding the diffusion of SATD, the factors promoting its introduction, and the amount of technical debt removed by developers after its introduction. Maldonado and Shihab [15] also showed that SATD can be exploited to identify several different types of technical debt (*e.g.,* design and code debt), while Wehaibi *et al.* [37] highlighted that the presence of SATD makes the code more difficult to change in the future. Despite these studies, there is still a noticeable lack of empirical evidence related to the magnitude of the phenomenon, its evolution over time, and its relationship with software quality. This represents an obstacle for an effective and efficient management of technical debt.

In this paper we contribute to enlarge the knowledge about the technical debt phenomenon by presenting a large-scale empirical study on SATD, as a *differentiated replication* [38] of the study by Potdar and Shihab [30][1]. In particular, we mine the complete change history of 159 Java open source systems to detect SATD instances across over 600K commits, for a total of over 2 Billion comments mined. Using these data we analyse the diffusion of SATD (*i.e.,* the number of SATD

[1]Similarities and differences between the two studies are carefully described in Section 5 while discussing the related literature.

instances and the percentage of code comments reporting SATD), its evolution and survivability over time, and who are the developers introducing and fixing the SATD. Then, we perform a manual analysis of a statistically significant sample of 366 comments reporting SATD to identify what are the types of SATD (*e.g.,* design, code, test, requirement debt) more spread in open source systems. In our manual analysis, we adopted an open coding process inspired by the Grounded Theory principles formulated by Corbin and Strauss [13]. Finally, we verified if classes exhibiting low quality as assessed by complexity, coupling, and readability quality metrics are more likely of being affected by SATD.

The achieved results show that:

- Comments reporting SATD are diffused in the mined 159 open source projects, with an average of 51 instances per system (corresponding to 0.3% of the code comments). This result is inline with what observed by Potdar and Shihab [30] on the five software systems subject of their analysis.
- Our manual categorisation highlighted that the most diffused type of SATD is the code debt (30% of instance), followed by defect and requirement debt (20% each) and design debt (13%).
- The number of SATD instances increases during the change history of software systems due to the introduction of new instances that are not fixed (*i.e.,* removed) by developers (on average, ∼57% of SATD instances are fixed). This result confirms previous findings reported in [30]. This 57% of fixed instances shows a very long survivability, by staying in the system for over 1,000 commits, on average.
- In most cases (63%), the developer paying-back the debt is the same that introduced it in the system. In the remaining 37%, developers fixing the technical debt exhibit a higher experience with respect to those who introduced it.
- As also observed by Potdar and Shihab [30], there is no correlation between the code file internal quality and the number of SATD instances they contain.

To ease replication, all the data used in our study are publicly available [5].

This study has multiple implications. On the one hand, it has a descriptive nature, as it provides a characterisation of the investigated phenomenon (*i.e.,* SATD). On the other hand, results of such a study provides useful information to researchers and practitioners interested in better detection and management of technical debt as a way to ease software maintenance and evolution [37].

Structure of the paper. Section 2 defines our empirical study and the research questions, and provides details about the data extraction process and analysis method. Section 3 reports the results of the study, answering our research questions, while Section 4 discusses the threats that could affect the validity of the results achieved. Section 6 concludes the paper and outlines directions for future work, after a discussion of the related literature (Section 5).

2. EMPIRICAL STUDY DESIGN

The *goal* of the study is to analyse SATD in software projects, with the *purpose* of investigating (i) its diffusion, (ii) its evolution over the change history of software projects, and (iii) its relationship with the code internal quality. The *quality focus* is on software quality, which could be negatively affected by the presence of technical debt.

More specifically, the study aims at addressing the following three research questions:

- **RQ$_1$:** *What is the diffusion of self-admitted technical debt in open source systems?* In this research question, we firstly investigate how prevalent the SATD is in open source systems. Then, we categorise the identified technical debt into the different "types" starting from the categories defined by Alves *et al.* [2] and following an open coding procedure.
- **RQ$_2$:** *How does the self-admitted technical debt evolve during the change history of software systems?* This research question analyses the evolution of SATD over the change history of open source systems. In particular, we verify if the technical debt increases/decreases over time, how frequently it is fixed by developers, how long does it survive in the systems, and who the developers introducing and fixing it are.
- **RQ$_3$:** *Are low quality components more prone to self-admitted technical debt?* The third research question aims at empirically investigating whether developers are more prone to introduce SATD in code components exhibiting a low internal quality as assessed by quality metrics (*e.g.,* are developers more prone to introduce technical debt when working on complex classes?).

2.1 Context Selection

The *context* of the study consists of the change history of 159 projects belonging to two software ecosystems, namely Apache, and Eclipse. Table 1 reports for each of them (i) the number of projects analysed, and descriptive statistics of (ii) their size in terms of KLOC, (iii) the number of commits analysed, and (iv) the number of contributors. All the analysed projects are hosted in *Git* repositories.

The 159 considered projects were selected from the list of projects managed by the Apache Software Foundations[2] and from the list of GitHub repositories managed by the Eclipse Foundation[3]. In particular, we only selected:

1. *Java projects.* We limit our analysis to Java projects since, as it will be shown later, the tools we used to extract the data needed to answer our research questions only works on Java code.
2. *Active projects.* We started mining our data on October 15^{th} 2015 and we considered as active projects those having at least one commit performed in the previous month (*i.e.,* after September 15^{th}).

[2]https://projects.apache.org/indexes/quick.html
[3]https://github.com/eclipse

Table 1: Characteristics of ecosystems under analysis.

Ecosystem	#Projects	KLOC				#Commits				#Contributors			
		min	max	mean	overall	min	max	mean	overall	min	max	mean	overall
Apache	120	1	1,550	272	32,692	99	22,433	4,332	519,937	2	463	54	6,426
Eclipse	39	1	1,105	246	9,613	39	26,628	3,540	138,089	5	304	40	1,559
Overall	**159**	**1**	**1,550**	**266**	**42,305**	**39**	**26,628**	**4,138**	**658,026**	**2**	**463**	**50**	**7,985**

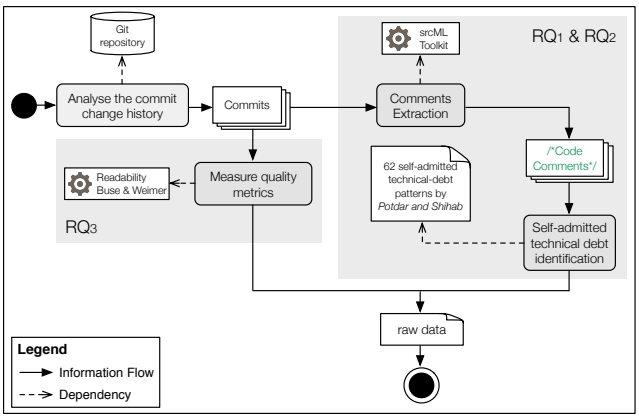

Figure 1: Data extraction process.

Also, the choice of the ecosystems to analyse is not random, but rather driven by the motivation to consider projects having (i) different sizes, going from just one KLOC up to thousands of KLOC, (ii) different architectures, including Apache libraries, and plug-in based architectures in Eclipse projects, and (iii) different development bases, with projects carried out by small development teams as well as by large communities (see Table 1).

2.2 Data Extraction

Figure 1 depicts the main steps behind the data extraction process that we followed to answer our research questions. The two grey parts highlight the data extraction performed for the different research questions.

We started by cloning the *Git* repositories of the 159 subject systems. Then, we built a tool to identify the self-admitted technical-debt in each of the commits performed in the change history of the subject systems. Our tool mines the entire change history of each repository, checks out each commit in chronological order, and exploits `srcML` [11] to extract the comments from each Java code file. The set of retrieved comments is then analysed to identify those reporting SATD. We exploit regular expressions to match inside comments the 62 SATD patterns[4] defined by Potdar and Shihab [30]. This list includes patters likely indicating SATD in code comments (*e.g., this is a hack*, *fixme*, *this is ugly*, *etc.*) and has been defined by manually reading 101,762 comments [30].

The output of this process is the list of SATD instances present in the systems' source code after each of the 658,026 commits object of our study. These data are used to answer all our research questions and are sufficient for RQ_1 and RQ_2 (see Figure 1).

To collect data for RQ_3, we developed a tool that measures complexity, coupling, and readability in the code of each object system[5]. Given the high computational cost of computing such quality metrics, we did not compute them for all snapshots but only for the last snapshot analysed in each subject system. Still, this required the computation of the three quality metrics for 235,602 Java files. Worth noticing here that our analysis has been performed at "file level" for sake of simplicity. Thus, we measured code complexity of a file as the sum of the McCabe's cyclomatic complexity [27] of the methods it contains. Coupling for a file is measured as the sum of the Coupling Between Object [9] of the classes it contains. Note that both the values of complexity and coupling are non-negative and unbounded. Finally, we measured the readability of a file by exploiting the metric proposed by Buse and Weimer [8]. This metric combines a set of low-level code features (*e.g.,* identifiers length, number of loops, *etc.*) and has been shown to be 80% effective in predicting developers' readability judgments. We used the authors' implementation of such a metric[6]. Given a code file, the readability metric takes values between 0 (lowest readability) and 1 (maximum readability).

2.3 Data Analysis

This subsection describes the analyses and statistical procedures that we used to answer the three research questions previously formulated. All analyses are performed per ecosystem (*i.e.,* by showing the results in isolation on the Apache and the Eclipse ecosystems) as well as by considering all 159 systems as a whole.

To answer RQ_1 we analyse the last snapshot of each system and show the boxplots of SATD instances in the subject systems. Since different systems may have a substantially different size (see Table 1), we report both the discrete number of technical debt instances found in each system as well as the percentage of their comments reporting SATD. The "last-commit" analysis for RQ_1 provides a *static* view of the SATD in an exact moment in time, leaving to RQ_2 the investigation of the SATD evolution.

We identified 7,584 comments reporting SATD. To understand what are the "types" of SATD more spread in open source systems, we manually analysed a randomly selected set of 366 comments. Such a set represents a 95% statistically significant sample with a 5% confidence interval. In order to analyse and categorise the comments in the sample according to the "type" of technical debt reported in them we followed an open coding process. In particular, all 366 comments were manually analysed by both authors in multiple rounds. In the first *warm-up* round, we inspected and labeled 30 comments randomly chosen. In this round, the labels were the "types" of technical debt defined by Alves *et al.* [2]. After the round, we discussed and reviewed labels and assignments

[4]http://users.encs.concordia.ca/~eshihab/data/ICSME2014/satd.html

[5]Such a tool does only work on Java code.

[6]Available at http://tinyurl.com/kzw43n6

solving conflicts. We repeated this procedure by incrementally refine/extend the set of labels and their assignments until the full set of comments reporting SATD was inspected. The output of the open coding phase is a set of labels derived from the original types of Alves *et al.*, the assignments of such labels to each comment, and the identification of false positive (*i.e.,* comments do not reporting SATD). The new set of labels is illustrated in Figure 3. The bottom labels are the ones we introduced, *e.g.,* in the *defect debt* category, we discriminate between *known defects*, *i.e.,* opened in the issue tracker and yet to be fixed, and defects *partially fixed* with a temporary patch. We qualitatively discuss the findings of this analysis, presenting our classification and examples of comments belonging to the various categories.

To answer RQ_2 we show boxplots reporting:

- The change, in terms of percentage of comments reporting SATD, between the first and the last snapshot analysed in each subject system. This analysis will provide a first overview on the evolution of technical debt over time. Note that the analysis is performed by considering the percentage of comments reporting technical debt (as opposed to the discrete number of SATD instances) to take into account for changes in systems' size between the first and the last mined commits.

- The number of comments reporting SATD introduced and fixed by developers over the change history of the subject systems. This is possible thanks to our commit-level analysis. Indeed, we know the exact commit in which a comment reporting SATD is introduced and possibly fixed. This analysis will provide indication of the amount of technical debt left unfixed in software systems. Also, we investigate if the developer introducing the technical debt is also the one fixing it, and we compare the experience of the developers introducing and fixing the technical debt. Given a file F_i in which the developer D_j introduced/fixed a technical debt instance, we use the number of commits performed in the past by D_j on file F_i as a proxy of her experience. We use the Mann-Whitney test [12] to analyse statistical significance of the differences between the experience of the two distributions of developers. The results are intended as statistically significant at $\alpha = 0.05$. We also estimate the magnitude of the measured differences by using the Cliff's Delta (or d), a non-parametric effect size measure [17]. We follow well-established guidelines to interpret the effect size values: negligible for $|d| < 0.14$, small for $0.14 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [17].

- The survival of SATD in terms of number of commits. This analysis is limited to the SATD instances that have been fixed, sooner or later, by software developers. It aims at verifying how long does it take to fix such issues. Note that we preferred to exploit the number of commits over the number of days as a proxy for survivability since different projects may have different levels of activity (*i.e.,* a different frequency of commits over time). Thus, considering the days would have not taken into account possible periods of inactivity in the analysed projects.

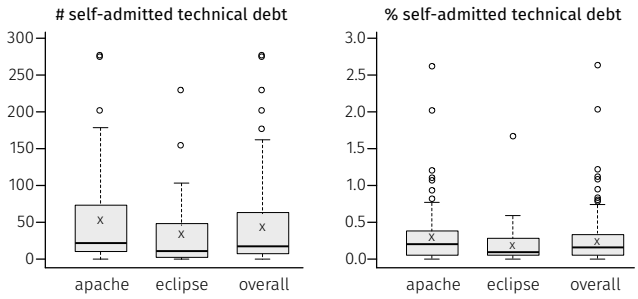

Figure 2: RQ_1: Diffusion of self-admitted technical debt. Absolute number (#) and percentage of comments reporting it (%).

Still in the context of RQ_2, we also qualitatively discuss some interesting cases of "technical debt evolution" we found in our study.

Finally, to answer RQ_3 we compute the Spearman rank correlation [39] between each of the three quality metrics and the number of technical debt instances in the files. As done in [30], since larger files are more likely to contain self admitted technical debt, we compute the *partial* correlation between number of technical debt instances and the three quality metrics while using the LOC as control variable. Partial correlation allows to measure the strength of a linear relationship between two variables whilst controlling for the effect of one or more other variables (*i.e.,* control variables). We interpret the correlation coefficient according to the guidelines by Cohen *et al.* [10]: no correlation when $0 \leq |\rho| < 0.1$, small correlation when $0.1 \leq |\rho| < 0.3$, medium correlation when $0.3 \leq |\rho| < 0.5$, and strong correlation when $0.5 \leq |\rho| \leq 1$.

2.4 Replication Package

All the data used in our study are publicly available [5]. Specifically, we provide the *R* scripts and working data sets used to run the statistical analysis and produce the plots and tables reported in this paper.

3. STUDY RESULTS

This section discusses the results achieved in our study according to the three formulated research questions.

3.1 RQ_1: What is the diffusion of self-admitted technical debt in open source systems?

Figure 2 depicts the diffusion of SATD in the 159 mined software systems when looking separately inside each ecosystem as well as when considering the complete dataset as a whole. The boxplots on the left side report the distribution of the absolute number of SATD instances found in the systems, while those on the right show the percentage of code comments reporting SATD.

On average, projects in the Apache ecosystem exhibit 53 instances of SATD, as opposed to the 33 of projects belonging to the Eclipse ecosystem (51 when considering the whole dataset). The maximum value has been observed in Apache

Lucene-Solr[7], with 314 instances. While such a number might look very high, it is important to consider that this system contains a total of 110,595 comments, thus meaning that 0.3% of its comments report SATD. Still, the achieved results indicate the diffusion of SATD in the analysed systems in terms of discrete number of instances.

When looking at the percentage of comments reporting SATD it is, on average, quite low for both ecosystems (0.4% for Apache, 0.2% for Eclipse, and 0.3% overall). The system exhibiting the highest percentage of SATD comments (2.6%) is Apache Forrest[8], with 45 out of the 1,783 total comments reporting technical debt. The overall observed trend is inline with what reported by Potdar and Shihab [30].

Figure 3 shows the results of our categorisation created as described in Section 2.3. Of the 366 comments reporting SATD that we manually analysed, 93 were classified as false positives (25%). This means that ∼25% of the technical debt automatically identified by using the patterns defined by Potdar and Shihab [30] are likely to represent false positives. Clearly, this also affects the results of our study and we will discuss such a threat in Section 4. Representative examples of false positive instances are the following:

1. `//ask the next processor to take care of the message.` The pattern "*to take care*" made our tool classifying the comment as a SATD instance, while it simply describes operations performed in the subsequent line of code.
2. `//we received all the post content send the crap back.` The word "*crap*" suggests the presence of something wrong in the code. Instead, it is just a colorful expression used by the developer to describe the code control flow.
3. `//throws SolrException if there is a problem reloading.` The pattern "*there is a problem*" is interpreted as a SATD, while it just represents an explanation for the exception thrown if specific problems occur.

Future works will be devoted to develop more advanced heuristics aimed at reducing false positives.

The remaining 273 instances were classified as reported in Figure 3. The top nodes (*i.e.,* code debt, design debt, documentation debt, defect debt, test debt, and requirement debt) represent the higher-level classification mapping the 273 instances in the technical debt categories defined by Alves *et al.* [2]. The circled number on the top-right corner of each category represents the number of instances that were mapped in it (*e.g.,* 81 instances were classified as *code debt*). When possible, we defined sub-categories better specialising the types of technical debt identified in our manual analysis. In the following, we describe each category by reporting and commenting representative SATD instances we found.

Code debt. "*Problems found in the source code which can affect negatively the legibility of the code making it more difficult to be maintained*" [2]. We performed a sub-categorisation of this type of technical debt into *low internal quality* (37 instances) and *workaround* (44). In the former we grouped comments clearly reporting unjustified issues with the quality of the source code (*e.g.,* low readability, misuse of programming constructs, unnecessary code complexity, *etc.*).

[7]http://lucene.apache.org/solr/
[8]https://forrest.apache.org

Some interesting examples are:

- `//FIXME isn't this a nice mess of a client? read input, write splits, read splits again.` The developer is referring to the strange logic implemented in the method and introducing unnecessary complexity.
- `//FIXME? YarnRuntimeException is for runtime exceptions only.` In this case, a wrong exception is thrown in the code.
- `//TODO: don't repeat this ugly cast logic (maybe use isCastable in the last else block).` The comment refers to a list of five `else if` branches in a method verifying the type of a parameter.

Technical debt instances classified as *workaround* refer instead to low quality code justified, however, by the need for reaching specific goals. In other words, they represent a compromise between code quality and specific functional/non-functional requirements. Examples are in the following:

- `//The system doesn't have a function that allows retrieval of a sequence of attribute values. [...] The following is a very (VERY) ugly two-part workaround that one could use until something better is available.` The developer is clearly explaining that the "ugly" code is driven by the need for having a specific, unavailable, function.
- `//Hack for backwards compatibility with XalanJ1 stylesheets.` A "hack" needed to guarantee backward compatibility.

Design debt. "*Debt that can be discovered by analysing the source code by identifying the use of practices which violated the principles of good object-oriented design*" [2]. We found 34 instances of design debt, 30 classified as *code smells*, and four as *design patterns*. Code smells clearly refer to violations of good object-oriented design. They include (but are not limited to):

- *Feature Envy*: A method that seems to be more interested in a class other than the one it is implemented in [16] (*e.g.,* `//FIXME: to be moved to ApiResponseHelper`).
- *Code Clones*: Duplicated code (*e.g.,* `//TODO: There is overlap between ValidTypeMap and PrimitiveTypeName in parquet-mr, it might be a good idea to unify these two classes`).
- *Lexical bad smells*: Poor lexicon that can lead to poor comprehensibility and even increase software fault-proneness [3] (*e.g.,* `//really ugly field name that isn't a java Id and can't be`).
- *Long method*: A method containing too many lines of code [16] (*e.g.,* `//FIXME extract some method`).

The four instances classified as design patterns refer instead to the need for introducing a design patter highlighted by developers (*e.g.,* `//FIXME : Façade this object`).

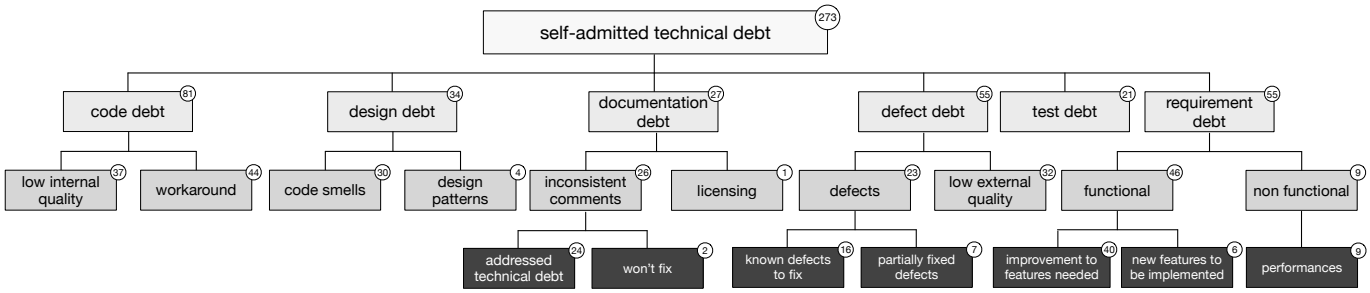

Figure 3: RQ$_1$: Manual categorisation of 366 self-admitted technical debt instances. Excluding 93 false positives.

Documentation debt. "*Problems found in software documentation that can be identified by looking for missing, inadequate, or incomplete documentation of any type*" [2]. We found 27 instances of documentation debt related to code comments[9] reporting misleading information (26) and licensing issues (1). We considered a comment as reporting misleading information if it (i) described an already addressed technical debt (*e.g.,* `//TODO` instances left in the code but describing things already fixed/implemented), or (ii) still pointed to an issue that has been already closed and classified as "won't fix".

Defect debt. "*Known defects [...] that should be fixed but due to competing priorities and limited resources have to be deferred to a later time*" [2]. We expanded the definition of defect debts to not only include known defects, but also problems in the code that *might* result in a defect (indicated as "low external quality" in Figure 3), like those that have been defined by Binder as bug hazards [6], *i.e.,* a circumstance that increases the chance of a bug being present in the software. We found 23 defect debt instances referring to known defects and 32 lowering the system external quality. Those referring to known defects have been further categorised into comments reporting known defects to fix (16) and those highlighting temporary patches aimed at partially fixing the defect while waiting for the official fix (7). Exemplar cases are:

- `//FIXME At sender ack mode this method check only the state transfer and resend is a problem.` A known issue to be solved.

- `//HACK: these values estimate [...]. To be removed once bug 46112 is fixed.` In this case, a temporary patch, estimating values to be computed, is implemented while waiting the official fix.

As for the "low external quality" category, examples falling in it include missing thrown exception (*e.g.,* `// FIXME: does not throw SQLException`) and issues with missing inputs/parameters control that might increase the chance of observing failures (*e.g.,* `//FIXME Please not wait only for three characters better control that the wait ack message is correct`).

[9]Note that we consider code comments as a type of documentation.

Test debt. "*Issues which can affect the quality of testing activities*" [2]. We found 21 instances of self-admitted test debt. They mainly include comments reporting:

- *The impossibility to reproduce bugs during testing activities*, *e.g.,* `//hard to reproduce - I tried different things but at last I give up.`

- *Failing assert statements to check*, *e.g.,* `//FIXME This assert fails.`

- *Low quality code in test suites*, *e.g.,* `//A dirty hack to modify the env of the current JVM itself.`

Requirement debt. "*Tradeoffs made with respect to what requirements the development team need to implement or how to implement them*" [2]. We identified 55 instances of requirement debt and discriminated between debt referred to functional (46 instances) and non functional (9 instances) requirements. The functional requirement debt instances were further classified into those referring to already implemented features (40) and those requiring the implementation of whole new features (6). Interesting examples are:

- *Comments reporting implementations going against the requirement specification*, *e.g.,* `//TODO: this is actually against the spec but the requirement is rather silly.`

- *Comments referring to missing features*, *e.g.,* `// FIXME: PORT should be retrieved from properties.` The comment refers to a variable declaration `int port = 8888` that assigns a default value to a network port. Such a value should instead be retrieved from the user's preferences.

- *Comments reporting doubts about the implementation of requirements*, *e.g.,* `//FIXME: order desc by update time?`

Finally, concerning the nine requirement debt instances related to non functional requirements, they were all related to performance issues (*e.g.,* `//FIXME this stats update are not cheap!!` , `//FIXME - use cache?`).

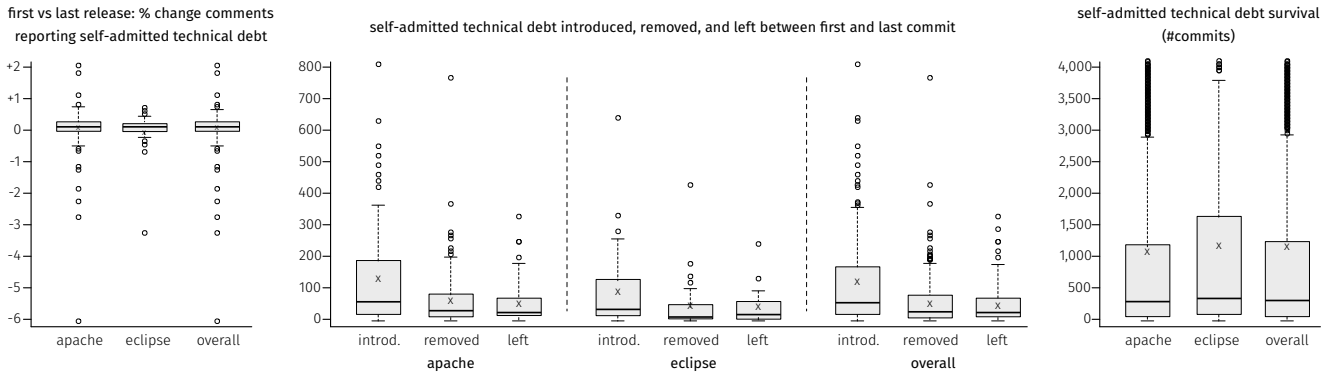

Figure 4: RQ$_2$: Evolution of self-admitted technical debt.

Summary for RQ$_1$. In terms of absolute number, comments reporting SATD are prevalent in the analysed projects, with an average of 51 instances per system (0.3% of comments present in code report technical debt). Our manual categorisation of 366 instances highlighted as the most diffused type of SATD is the code debt (30% of instance), followed by defect and requirement debt (20% each) and design debt (13%).

3.2 RQ$_2$: How does the self-admitted technical debt evolve during the change history of software systems?

Figure 4 provides information about the evolution and survival of SATD in the 159 mined projects. The boxplots on the left depict the distribution of the variation of the percentage of comments reporting SATD between the first and the last snapshot of each subject system. As a general trend it is possible to observe that the percentage of comments reporting technical debt remains quite stable between the first and the last systems' snapshots. Indeed, the average variation is -0.001%. Also, the strongest increase is around 2% while the lowest decrease it close to 6%. Note that this does not mean that the amount (*i.e.,* absolute number of instances) of self admitted technical debt is stable over the system history, but that the proportion of comments reporting technical debt is stable.

Indeed, the boxplots at the center of Figure 4 tell a different story. They depict the absolute number of SATD instances introduced, removed, and left in the two analysed ecosystems as well as in the complete dataset. On average, 115 instances are introduced by developers during the system change history. Of these, 65, on average, are removed in a later commit leading to an average of 51 instances still affecting the last system snapshot we analysed. These data show that the absolute number of SATD instances increases over the change history of software systems, since only ~57% of the introduced instances are then fixed later on (*i.e.,* the debt is payed). This result is inline with what observed by Potdar and Shihab [30] in the analysis of five open source systems: they found that between 26% and 63% of the SATD is payed-back by developers.

Figure 5 shows the evolution of SATD in four of the subject systems, *i.e.,* Apache CloudStack, Apache Ant, Eclipse PDT, and Eclipse JDT UI. All of them show a

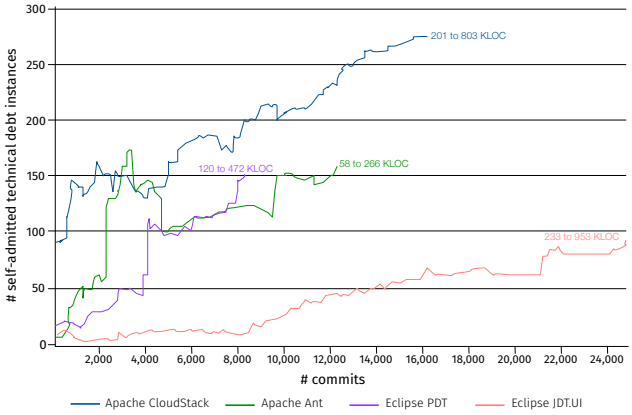

Figure 5: RQ$_2$: Evolution of self-admitted technical debt in four systems.

clear uptrend over time of (i) their size in terms of KLOC, and (ii) the number of instances of SATD. Also, looking at Figure 5 it is clear the presence of some commits resulting in rapid increases and decreases in the number of technical debt instances. A manual inspection of these commits highlighted as they represent huge restructuring of the code, resulting in dozens of files added/deleted (clearly, together with the SATD such files contain). For example, in Apache Ant the 2,352nd commit almost doubled the technical debt instances in the system (from 59 to 115), while the 4,890th commit decreased it from 135 to 97 (see Figure 5). The former (commit #`d1064de`), as described in the commit note, added in the repository *a clone of the main ant source tree so that it can undergo some heavy refactoring.* This explains the doubling of the technical debt instances in a single commit. The latter (commit #`70936fa`), removed the previously created clone, leading to the strong reduction of technical debt instances. Despite these exceptional cases, looking at Figures 4 and 5 it is clear the increasing trend of SATD during the change history of software systems.

The right part of Figure 4 depicts the survival, in terms of number of commits, of the technical debt instances removed by the developers (*i.e.,* the previously mentioned 57%). On

average, technical debt instances survive 1,087 commits (median=266). Thus, it takes long time to pay-back the introduced debt. Interestingly, in 63% of cases the developer paying-back the debt is the same who introduced it in the system. In the remaining 37%, we compared the experience of the developers who introduced and fixed the technical debt (see Section 2.3). The experience of the developers introducing the technical debt is significantly lower than that of developers fixing the technical debt (p-value < 0.001) with a small effect size (d=0.21).

Summary for RQ$_2$. The number of SATD instances increases during the change history of software systems due to the introduction of new instances only in part (∼57%) fixed. This 57% of fixed instances stays in the system for over 1,000 commits on average (median=266). In most cases (63%), the developer paying-back the debt is the same who introduced it in the system. In the remaining 37%, developers fixing the technical debt exhibit a higher experience with respect to those who introduced it.

3.3 RQ$_3$: Are low quality components more prone to self-admitted technical debt?

Table 2 shows the results of the Spearman partial correlation between the three considered quality attributes (*i.e.,* coupling, complexity, and readability) and the instances of SATD in code files when using the lines of code (LOC) as controlling variable.

As it is clear from Table 2, we did not find any correlation. The results for coupling and complexity are inline with what observed by Potdar and Shihab [30], who reported a weak to very weak correlation between these two quality attributes and the number of technical debt instances[10]. The evidence from both studies indicate as the internal quality of the code files plays no role in the introduction of technical debt.

Summary for RQ$_3$. We found no correlation between the code file internal quality and the number of SATD instances they contain.

Table 2: RQ$_3$: Spearman partial correlation between quality attributes and instances of self-admitted technical debt (using LOC as controlling variable).

Coupling (CBO)		
Ecosystem	p-value	ρ
Apache	<0.001	0.02 (No Correlation)
Eclipse	<0.001	0.03 (No Correlation)
Overall	<0.001	0.03 (No Correlation)

Complexity (WMC)		
Ecosystem	p-value	ρ
Apache	<0.001	0.03 (No Correlation)
Eclipse	<0.001	0.03 (No Correlation)
Overall	<0.001	0.03 (No Correlation)

Readability		
Ecosystem	p-value	ρ
Apache	<0.001	0.01 (No Correlation)
Eclipse	<0.001	0.02 (No Correlation)
Overall	<0.001	0.02 (No Correlation)

4. THREATS TO VALIDITY

This section describes the threats that can affect the validity our study.

Threats to *construct validity* concern the relation between the theory and the observation, and in this work are mainly due to the measurements we performed, and in particular to:

- *Imprecisions in the identification of SATD.* We exploited the patterns defined by Potdar and Shihab [30] to detect technical debt instances in code comments. As happen when using automatic detection heuristics, we are aware that our results might be affected by the presence of false positives and false negatives. This has been also confirmed by the manual validation that we performed on 366 code comments, highlighting the presence of 93 false positives. Still, the level of precision resulted from the manual validation (75%) and the vast amount of data object of our study make us confident about the validity of the observed trends and of the distilled findings.

- *Subjectivity in the manual classification.* To mitigate such a threat, both the authors independently classified the type of the 366 SADT instances. Then, a open discussions were performed to solve conflicts.

- *Assessment of the developers' experience.* We used as proxy to assess the developers' experience on a file F_i at date D_j the number of commits she performed on F_i before D_j. We are aware that this is an approximation that might introduce imprecision in the analysis. However, previous work used similar approximations like the total number of commits performed by the developers before D_j [31], or the total number of days she contributed to the project [19].

- *Imprecisions in the measurement of SATD survivability.* We measured the survivability of a SATD instance as the number of commits between its introduction in the system and its removal. As shown in our results, some comments reporting SATD are left in the code even after the technical debt instance they refer to has been fixed (see the discussion about documentation debt in Section 3.1). This results in the introduction of imprecisions in our assessment of SATD survivability. However, our data show that such cases only represent less than 10% of the overall SATD instances. Thus, the impact on our findings should be limited.

Threats to *internal validity* concern external factors we did not consider that could affect the variables and the relations being investigated. When analysing the diffusion and evolution of the SATD (RQ$_1$ and RQ$_2$) as well as when investigating factors promoting the introduction of technical debt (RQ$_3$), we considered the LOC as a confounding factor to be controlled. Also, to reinforce the internal validity and better interpret the statistical results, when possible, we integrated the quantitative analysis with a qualitative one, showing examples we found by manually inspecting code comments.

[10]Potdar and Shihab did not consider the readability as quality attribute in their study.

Threats to *conclusion validity* concern the relation between the treatment and the outcome. Although this is mainly an observational study, wherever possible we used an appropriate support of statistical procedures, integrated with effect size measures that, besides the significance of the differences found, highlight the magnitude of such differences.

Threats to *external validity* concern the generalisation of results. This is, to the best of our knowledge, the largest study—in terms of number of projects (159)—concerning the analysis of SATD and of its evolution. However, other systems should be analysed to support our conclusions. This is especially needed due to the fact that (i) all the projects subject of our study are written in Java, thus calling for the need of analysing software projects written in other programming languages, and (ii) we limited our analysis to open source projects ignoring industrial systems.

5. RELATED WORK

Several works have studied code smells both in production code (see *e.g.,* [20, 36]) as well as in test code (see *e.g.,* [4]). While code smells are a manifestation of technical debt, we omit the discussion of these works due to space constraints. We focus our discussion of the related literature on studies considering the more comprehensive concept of technical debt or directly looking for technical debt instances in code comments.

Storey *et al.* [35] explored the use of task annotations (*e.g.,* TODO, FIXME) in code comments as a mechanism to manage developers tasks. Their findings highlight several activities that are supported by such annotations (*e.g.,* TODOs are sometimes used to communicate by asking questions to other developers). While we use some of these task annotations to identify comments reporting SATD, our work has a totally different goal.

Guo *et al.* [18] tracked the lifecycle of a single delayed maintenance task (*i.e.,* a technical debt instance) to study the effect of this decision on the project outcomes. Their data confirm the harmfulness of technical debt, showing that the decision to delay the task resulted in tripled costs for its implementation.

Klinger *et al.* [21] interviewed four technical architects at IBM to understand how decisions to acquire technical debt are made within an enterprise. They found that technical debt is often acquired due to non-technical stakeholders (*e.g.,* due to imposed requirement to meet a specific deadline sacrificing quality). Also, they highlight the lack of effective communication between technical and non-technical stakeholders involved in the technical debt management.

Kruchten *et al.* [23] built on top of the technical debt definition provided by Cunningham [14] to clearly define what constitute technical debt and provide some theoretical foundations. They presented the "technical debt landscape", classifying the technical debt as visible (*e.g.,* new features to add) or invisible (*e.g.,* high code complexity) and highlighting the debt types mainly causing issues to the evolvability of software (*e.g.,* new features to add) or to its maintainability (*e.g.,* defects). Note that the debt categories defined in [23] are all represented in the types of technical debt later defined by Alves *et al.* [2] and used as starting point in our open coding.

Lim *et al.* [25] interviewed 35 practitioners to investigate their perspective on technical debt. They found that most of participants were familiar with the notion of technical debt—"*We live with it every day*" [25]—and they do not look at technical debt as a poor programming practice, but more as an *intentional decision to trade off competing concerns during development* [25]. Also, practitioners acknowledged the difficulty in measuring the impact of technical debt on software projects. Similarly, Kruchten *et al.* [24] reported their understanding of the technical debt in industry as the result of a four year interaction with practitioners.

Zazworka *et al.* [41] compared how technical debt is detected manually by developers and automatically by detection tools. The achieved results show very little overlap between the technical debt instances manually and automatically detected.

Spinola *et al.* [34] collected a set of 14 statements about technical debt from the literature (*e.g.,* "*The root cause of most technical debt is pressure from the customer*" [32]) and asked 37 practitioners to express their level of agreement for each statement. The statement achieving the highest agreement was "*If technical debt is not managed effectively, maintenance costs will increase at a rate that will eventually outrun the value it delivers to customers*".

Potdar and Shihab [30] pioneered the study of SATD by mining five software systems to investigate (i) the amount of SATD they contain, (ii) the factors promoting the introduction of the SATD, and (iii) how likely is the SATD to be removed after its introduction. Our work represents a *differentiated replication* [38] of the study by Potdar and Shihab [30]. Similarities and differences can be summarised as follows:

- *Size of the studies.* Our study is performed on a total of 159 systems as compared to the five projects analysed in [30]. This clearly helps to improve the generalisability of the achieved results and to corroborate the findings reported in [30].

- *Diffusion of self-admitted technical debt (RQ_1).* Both studies show the number of SATD instances as well as the percentage of comments reporting such debt. We add on top a manual classification of the types of technical debt instances present in the software systems. Such a classification also allowed us to provide indications about the percentage of false positive instances identified by using the SATD patterns defined in [30].

- *Granularity of the performed analysis (RQ_2).* Potdar and Shihab [30] study the evolution of technical debt, and in particular the percentage of SATD that is removed after its introduction, by comparing the technical debt present in first release of each studied system with that present in the subsequent releases (*e.g.,* seven Eclipse releases are considered). Thus, their study is performed at a release-level granularity. Our study exploits a finer granularity level by detecting SATD introduced/removed in each commit. This allowed us to (i) provide a closer view on the evolution of SATD, (ii) compute the survival of technical debt instances, and (iii) comparing the experience of developers introducing and removing SATD.

- *Relationship between code quality and self-admitted technical debt (RQ_3).* Both studies investigate the influence of code quality on the likelihood of introducing technical debt (we add code readability to the code complexity and coupling already investigated in [30]). Potdar and Shihab also investigated the influence of the *time to release* on the introduction of technical debt (*i.e.,* are developers more prone to introduce SATD when they work under pressure?). We excluded such a factor due to the impossibility of automatically (and reliably) recovering the list of releases for the 159 investigated projects. Indeed, we tried to exploit the GitHub APIs[11] to extract the release dates of each project. However, this resulted in 30,112 releases for the 159 projects (190 releases per system, on average). Such a very high number is due to the fact that several projects produces weekly or even daily "releases" freezing small implemented changes. This represents a huge threat when considering a release date as a "deadline" for developers. Thus, we decided to skip such an analysis and postpone it to future work, maybe by manually validating the list of retrieved releases to only consider the major ones.

Alves *et al.* [2] proposed an ontology of terms on technical debt. Part of this ontology is the classification of technical debt types derived by studying the definitions existing in the literature. In the manual analysis of our work (see Section 3.1), we exploit and refine such a classification for the SATD instances found in code elements.

Maldonado and Shihab [15] also used the classification by Alves *et al.* to investigate the types of SATD more diffused in open source projects. They identified 33K comments in five software systems reporting SATD. These comments have been manually read by one of the authors who found as the vast majority of them (∼60%) reported design debt. This work is closely related to our RQ_1, where we manually classified the type of technical debt reported in a statistically significant sample of 366 comments. The main differences between our study and the work by Maldonado and Shihab are the scale of the study and the procedure adopted to classify the technical debt instances. Maldonado and Shihab analysed a much larger set of comments reporting SATD (33K against 366); However, only one of the authors classified the debt types on the basis of his personal opinion. In our study, we adopted an open coding procedure performed by the two authors to classify the analysed instances and reduce the subjectivity bias. Also, we further refined the technical debt categories defined by Alves *et al.* [2] as a result of the open coding procedure.

Finally, a very recent study by Wehaibi *et al.* [37] analysed the relation between SATD and software quality in five open source systems. Their main findings show that: (i) the defect-proneness of files containing SATD instances increases after their introduction; and (ii) developers experience difficulties in performing SATD-related changes. This work represented a further motivation to better investigate the SATD phenomenon.

[11]https://developer.github.com/v3/

6. CONCLUSION AND FUTURE WORK

In this paper, we reported an empirical analysis conducted on 159 open source systems and aimed at investigating the SATD phenomenon. In particular, we investigated (i) its diffusion in the mined systems, (ii) its evolution and survivability over time, (iii) who are the developers introducing and fixing SATD, and (iv) the influence of code components internal quality on their likelihood of being affected by SATD.

We mined the complete change history of the subject systems detecting SATD instances after each of the 658,026 analysed commits. This allowed us to study the diffusion and evolution of SATD. Data about the diffusion were also complemented via a manual analysis of a statistically significant sample, with the goal of investigating the types of SATD more spread in open source systems. Finally, we computed code metrics assessing the quality of code components and correlated this information with the number of SATD affecting such files.

Our results show the diffusion of SATD instances in open source projects (on average, 51 instances per system), their long survivability (on average, over than 1,000 commits), and their increasing trend over the projects lifetime. This highlights the need for techniques and tools aimed at providing an effective management of technical debt. Our findings also shed the light on the most diffused types of SATD (code, defect, requirement, and design debt) and the lack of correlation between the code file internal quality and the number of SATD instances they contain.

Our future work agenda includes (i) the definition of better heuristics to detect SATD and reduce the number of false positives, (ii) the investigation of the evolution of different types of technical debt, (iii) the replication of our study on industrial systems and on systems written in different programming languages, in order to corroborate or contradict our findings; and (iv) the investigation of other factors that might promote the introduction of SATD, like those captured by process metrics.

7. REFERENCES

[1] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 281–293, New York, NY, USA, 2014. ACM.

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

[3] V. Arnaoudova, L. Eshkevari, M. Di Penta, R. Oliveto, G. Antoniol, and Y.-G. Gueheneuc. Repent: Analyzing the nature of identifier renamings. *Software Engineering, IEEE Transactions on*, 40(5):502–532, May 2014.

[4] G. Bavota, A. Qusef, R. Oliveto, A. D. Lucia, and D. Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094, 2015.

[5] G. Bavota and B. Russo. Replication package. http://www.inf.unibz.it/˜gbavota/reports/satd.

[6] R. V. Binder. *Testing object-oriented systems: models,*

patterns, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

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

[8] R. P. Buse and W. R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010.

[9] S. Chidamber, D. Darcy, and C. Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Transactions on Software Engineering (TSE)*, 24(8):629–639, 1998.

[10] J. Cohen. *Statistical power analysis for the behavioral sciences*. Lawrence Earlbaum Associates, 1988.

[11] M. Collard, H. Kagdi, and J. Maletic. An xml-based lightweight c++ fact extractor. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 134–143, 2003.

[12] W. J. Conover. *Practical Nonparametric Statistics*. Wiley, 3rd edition edition, 1998.

[13] J. Corbin and A. Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 13(1):3–21, 1990.

[14] W. Cunningham. The WyCash portfolio management system. *OOPS Messenger*, 4(2):29–30, 1993.

[15] E. da S. Maldonado and E. Shihab. Detecting and quantifying different types of self-admitted technical debt. In *7th IEEE International Workshop on Managing Technical Debt, MTD 2015, Bremen, Germany, October 2, 2015*, pages 9–15, 2015.

[16] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

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

[18] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. da Silva, A. Santos, and C. Siebra. Tracking technical debt - an exploratory case study. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 528–531, 2011.

[19] L. T. Jon Eyolfso and P. Lam. Do time of day and developer experience affect commit bugginess? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 153–162, 2011.

[20] F. Khomh, M. D. Penta, Y. Guéhéneuc, and G. Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, 2012.

[21] T. Klinger, P. Tarr, P. Wagstrom, and C. Williams. An enterprise perspective on technical debt. In *Proceedings of the 2Nd Workshop on Managing Technical Debt*, MTD '11, pages 35–38, 2011.

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

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

[24] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi. Technical debt: Towards a crisper definition report on the 4th international workshop on managing technical debt. *SIGSOFT Softw. Eng. Notes*, 38(5):51–54, 2013.

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

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

[27] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[28] N. Moha, Y. Guéhéneuc, L. Duchien, and A. L. Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Trans. Software Eng.*, 36(1):20–36, 2010.

[29] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. D. Lucia. Mining version histories for detecting code smells. *IEEE Trans. Software Eng.*, 41(5):462–489, 2015.

[30] A. Potdar and E. Shihab. An exploratory study on self-admitted technical debt. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 91–100, 2014.

[31] F. Rahman and P. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 491–500, 2011.

[32] K. S. Rubin. *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. Addison-Wesley Professional, 1st edition, 2012.

[33] F. Shull, D. Falessi, C. Seaman, M. Diep, and L. Layman. *Perspectives on the Future of Software Engineering*, chapter Technical Debt: Showing the Way for Better Transfer of Empirical Results, pages 179–190. Springer, 2013.

[34] R. Spinola, N. Zazworka, A. Vetró, C. Seaman, and F. Shull. Investigating technical debt folklore: Shedding some light on technical debt opinion. In *Managing Technical Debt (MTD), 2013 4th International Workshop on*, 2013.

[35] 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*, ICSE '08, pages 251–260, 2008.

[36] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, and D. Poshyvanyk. When and why your code starts to smell bad. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 403–414, 2015.

[37] S. Wehaibi, E. Shihab, and L. 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*, SANER '16, page To appear, 2016.

[38] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson,

B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[39] J. H. Zar. Significance testing of the spearman rank correlation coefficient. *Journal of the American Statistical Association*, 67(339):pp. 578–580, 1972.

[40] 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*, MTD '11, pages 17–23, New York, NY, USA, 2011. ACM.

[41] 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*, EASE '13, pages 42–47, 2013.