Can Clean New Code reduce Technical Debt Density?

George Digkas, Alexander Chatzigeorgiou, Apostolos Ampatzoglou,
and Paris Avgeriou, *Senior Member, IEEE*

Abstract—While technical debt grows in absolute numbers as software systems evolve over time, the density of technical debt (technical debt divided by lines of code) is reduced in some cases. This can be explained by either the application of refactorings or the development of new artifacts with limited Technical Debt. In this paper we explore the second explanation, by investigating the relation between the amount of Technical Debt in new code and the evolution of Technical Debt in the system. To this end, we compare the Technical Debt Density of new code with existing code, and we investigate which of the three major types of code changes (additions, deletions and modifications) is primarily responsible for changes in the evolution of Technical Debt density. Furthermore, we study whether there is a relation between code quality practices and the 'cleanness' of new code. To obtain the required data, we have performed a large-scale case study on twenty-seven open-source software projects by the Apache Software Foundation, analyzing 66,661 classes and 56,890 commits. The results suggest that writing "clean" (or at least "cleaner") new code can be an efficient strategy for reducing Technical Debt Density, and thus preventing software decay over time. The findings also suggest that projects adopting an explicit policy for quality improvement, e.g. through discussions on code quality in board meetings, are associated with a higher frequency of cleaner new code commits. Therefore, we champion the establishment of processes that monitor the density of Technical Debt of new code to control the accumulation of Technical Debt in a software system.

Index Terms—technical debt, refactoring, clean code, case study

1 Introduction

Technical Debt is a metaphor that captures in monetary terms, the cost of additional maintenance effort caused by technical shortcuts taken usually for expediency [1]. As observed in practice [2], for the majority of software systems, the amount of technical debt increases along evolution, due to growing size and/or reduced quality; this is aligned with software evolution laws [3]. However, the *density* of technical debt, i.e., the normalized amount of Technical Debt per line of code, remains in some cases stable, or is even reduced over time; we have observed this in our previous work on the evolution of the Apache ecosystem [2]. This raises the question how such systems manage to maintain or improve their Technical Debt density.

There are two possible explanations for this phenomenon. The first is that these systems follow a process of systematic perfective maintenance, mostly through the application of refactorings [4]. Code refactoring is the most popular strategy for technical debt repayment [5]. However, refactoring activities are rarely applied systematically in practice [6], [7], [8], [9], [10]. The second explanation is the development of new software artifacts at a level of quality that is above the average, following the clean code paradigm [11]. Introducing new code whose Technical Debt density is kept below the system average, is not trivial and often implies the adoption of an explicit policy for "clean commits". The sheer frequency of new commits in large or ultra-large codebases, which in the case of Google's source code can reach 16,000 on a typical workday [12], renders this strategy even more challenging. Clean code has recently emerged as a promising strategy facilitated by the use of Quality Gates in Continuous Integration systems.

While refactoring has been intensively studied as a technical debt repayment strategy, clean code has not. We argue that if writing clean new code is efficient in managing technical debt in the long term, this has important implications for both practitioners and researchers. The former can scope training activities and software development processes so as to incorporate the best possible practices for writing clean code (e.g., develop quality gates based on the technical debt introduced by new commits). The latter can focus their research efforts on technical debt prevention and repayment within new code chunks.

In this study, we first observe how 'clean' is the new code compared to the $TD_{density}$ of existing code for 27 projects from the Apache Software Foundation. Next, we compare the contribution of new, deleted and modified code to the changes in the system's $TD_{density}$. This allows to understand which activity affects technical debt density the most: writing new code, deleting or modifying code. Finally, we investigate if a decreasing trend in the evolution of $TD_{density}$ is associated with the adoption of relevant

- *G. Digkas is with the Institute of Mathematics and Computer Science, University of Groningen, Netherlands and with the Department of Applied Informatics, University of Macedonia, Greece.*
E-mail: g.digkas@rug.nl, g.digkas@uom.edu.gr
- *A. Chatzigeorgiou is with the Department of Applied Informatics, University of Macedonia, Greece.*
E-mail: achat@uom.edu.gr
- *A. Ampatzoglou is with the Department of Applied Informatics, University of Macedonia, Greece.*
E-mail: a.ampatzoglou@uom.edu.gr
- *P. Avgeriou is with the Institute of Mathematics and Computer Science, University of Groningen, Netherlands.*
E-mail: p.avgeriou@rug.nl

Manuscript received month day, year; revised month day, year.

practices at project management level. To this end we study whether projects that exhibit a high frequency of cleaner new code: a) provide clear guidelines to committers, so as to guarantee the quality of the newly committed code; and b) consider code quality as an important topic in project management.

We rely on the notion of $TD_{density}$ since absolute measures of technical debt (such as the number of identified violations or estimates of the effort to eliminate these violations) increase monotonically with the addition of code (i.e., absolute measures can decrease only by code deletions). TD$_{density}$ is obtained as the ratio of technical debt effort to remediate the issues identified in a piece of code, over the corresponding lines of code. In this way, a lower TD$_{density}$ of newly added code might result in a reduction of the system's total TD$_{density}$, even if the technical debt in absolute terms has increased.

In terms of scope, we focus on code technical debt, which is the most studied type of technical debt in the literature [13], and the most important type of technical debt in industry [14]. In particular, we consider the TD incurred by code smells in the source code. In terms of granularity, we work at the method level: we monitor the introduction of new methods, and the removal or modification of existing methods. This helps to avoid incorrect classification of code changes during code evolution: changes at the instruction level can become cumbersome to track, as modification, removal and introduction of individual instructions can occur simultaneously. This is further justified in Section 3.3. Finally, we emphasize that our scope is open source software, as they provide a long history of commit activity thereby enabling the evolutionary analysis of the study.

The rest of the paper is organized as follows: in Section 2 we present related work, i.e., studies that deal with the evolution of software quality and technical debt in particular, empirical studies that provide evidence on the impact and frequency of refactorings as well as recent work on the development of clean new code through the concept of quality gates. In Section 3, we present the investigated research questions, the case study design and we discuss how we monitor the contribution of new, deleted and modified code to the system's TD$_{density}$. The results of the study on 27 Apache projects are presented in Section 4, while in Section 5, we discuss the findings, by providing interpretations and implications for researchers and practitioners. Finally, in Section 6 we evaluate the validity of the study, whereas Section 7 concludes the paper.

2 Related Work

The current study explores the contribution that new code can have on technical debt density, as a complementary approach to applying refactoring. Therefore, this section discusses previous work on: (a) the evolution of code smells and Technical Debt in particular, (b) evidence on the frequency and impact of refactoring, and (c) the concept of Quality Gates that focus on ensuring a desired level of quality in new commits.

2.1 Evolution of Code Smells

Lehman's seventh law of software evolution states that *the quality of a system will appear to be declining during its evolution, unless proactive measures are taken* [15]. To this end, many studies have explored the evolution of code quality, and if indeed this law stands in practice. Since this paper focuses on code Technical Debt, we scope this sub-section to the evolution of code smells.

One of the first studies that investigate the evolution of code smells was conducted by Olbrich et al. [16]. On their study, they investigate the evolution of two code smells, God Class and Shotgun Surgery, on two projects by the Apache Software Foundation, namely Apache Lucene and Apache Xerces. The results of their study, show that during the software development, there are phases where the number of those code smells can either increase or decrease and those phases are not affected by the size of the systems.

Chatzigeorgiou and Manakos [6] have also investigated the evolution of code smells in open-source object-oriented projects. They used historical data of two open-source software projects, namely: JFlex and JFreeChart and studied the evolution of four code smells namely: Long Method, Feature Envy, State Checking, and God Class smells. The results of their study show that as the projects evolve over time the number of code smells tends to increase, which confirms the Lehman's seventh law. Furthermore, they have also found evidence that developers rarely perform targeted refactoring activities to remove smells. In most of the cases, if code smells disappear over time, this is a side effect of regular maintenance (e.g. removal of code). Another interesting finding was that a significant percentage of smells was not the results of software ageing, but smells were present right from the first version of the code in which they reside.

Tufano et al. [9] also studied the evolution of code smells with the goal of understanding when and why code smells are introduced into the projects and observe their life cycle. The study was based on five code smells: Blob Class, Class Data Should be Private, Complex Class, Functional Decomposition, Spaghetti Code. The results indicate that: (a) in the majority of the cases the code smells are introduced into the projects with the creation of the corresponding classes or files, (b) while projects evolve over time, "smelly" code artifacts tend to become more problematic, (c) new code smells are introduced when software engineers implement new features or when they extend the functionality of the existing ones, (d) the developers who introduce new code smells into the projects, are the ones who work under pressure and not necessarily the newcomers, and (e) the majority of the smells are not removed during the project's evolution and few of them are removed as a direct consequence of refactoring operations.

Peters and Zaidman [7] studied the lifespans of the following code smells: God Class, Feature Envy, Data Class, Message Chain Class, and Long Parameter List. They developed a tool called SACSEA and used it to analyze the history of eight open-source software projects. Their findings show that while projects evolve, the number of code smells increases. Furthermore, they have also found that although developers are aware of the existence of the code smells they do not perform refactorings. Finally, their findings imply that 'simpler code smells (e.g. Feature Envy Methods) are refactored more often, without any evidence on whether this happens intentionally or not.

Digkas et al. [2] analyzed and tracked the evolution of technical debt of sixty-six open-source Java projects by the Apache Software Foundation, over a period of 5 years. In order to track and detect issues that incur technical debt they relied on SonarQube. The results of their study show that on the one hand, there is a significant increasing trend on the size, complexity, number of Technical Debt Issues, and the total Technical Debt over time, which seems to confirm the software aging phenomenon. But on the other hand, when technical debt is normalized over the non-commented lines of code of the project, an evident decreasing trend over time is present for many of the projects. This could possibly be attributed to: (a) developers that perform refactoring activities and fix some of the open Technical Debt Issues; or (b) developers that introduce better quality code in each commit (compared to the project's existing code base).

Prior research provides evidence that the number of code smells increases over time [2], [7] and that smells are often introduced along with the creation of the corresponding classes/files [6], [9]. However, these studies have not investigated the association between overall trends in system quality with the cleanness of new code or the quality practices followed in a project so as to provide insight into the potential of clean new code as a means of reducing Technical Debt.

2.2 Refactoring Frequency and Impact

In this sub-section, we first discuss the frequency at which refactorings are applied, and then we provide evidence on the impact of refactorings on code quality.

Evidence shows that developers rarely apply code refactorings to remove smells. Arcoverde et al. [8] studied the lifespan of code smells within software projects and investigated why developers tend to perform very few refactorings. The results of their explanatory survey show that developers are reluctant to perform refactorings in order to avoid API modifications.

Yamashita and Moonen [17] also tried to shed light on why the developers do not perform refactorings on their projects. They conducted an exploratory survey with 85 developers to investigate how familiar they are with the notion of code smells. The results show that one third of the interviewed developers are not aware of code smells or have limited knowledge about about them. Furthermore, many of them expressed the lack of good supporting tools that would help them identify smelly pieces of code as candidates for refactoring.

Murphy-Hill et al. [4] studied broader developers' refactoring habits. Similar to other studies they found that the developers rarely perform refactoring activities and usually, when they do, they combine those refactorings with other code changes. Finally, they observed that even when developers do perform refactoring activities, they do not systematically record them, e.g. as a message on their commits.

A Google initiative in 2009 asked engineers to participate in a companywide "Fixit" week, focusing on resolving warnings issues by a static analysis tool. Only 16% of the total number of warnings were actually fixed, despite the fact that almost half of the reviewed issues resulted in filing a bug report [18]. It is also noteworthy that Google developers deemed 74% of the issues raised early (i.e. at compile time) as 'real problems', compared to 21% of suggested changes for already checked-in code.

A number of studies have empirically investigated the effect of refactoring application on various software qualities. Stroggylos and Spinellis [19] examined the logs in the version control systems of four open-source software projects to extract the commits where refactorings had been performed. Next, they measured the effect of refactorings on selected software metrics. The findings reveal that, despite the expectation that refactorings would improve software quality, measurements on the examined systems indicate the opposite. In particular, it was found that refactoring caused a non trivial increase in metrics related to cohesion and coupling.

To investigate how specific quality factors are affected by refactoring, Bois and Mens [20] proposed a formalism based on abstract syntax tree representation of source code and projected the impact of refactoring on internal quality metric values defined on this representation. The selected refactorings were Extract Method, Encapsulate Field, and Pull Up Method. Although the study is not focused on obtaining extensive empirical results, the application of the examined refactorings can have a mixed effect on different metrics (such as size, coupling and cohesion ones).

Wilking et al. [21] conducted a controlled experiment to investigate how refactorings affect the maintainability and modifiability of the projects. Their approach consisted in randomly inserting 15 syntactical and 10 non-syntactical errors into code and they measured the time that is needed to fix them. Concerning the effect of the refactorings on the modifiability, they evaluated it by adding new implementation requirements and they measured the time and the Lines of Code that are required in order to implement them. The results of their controlled experiment show that there is no direct effect of improved maintainability or modifiability due to refactoring.

In another study, Alshayeb concluded that refactoring application does not necessarily improve external quality attributes such as adaptability, maintainability, understandability, reusability and testability [22]. By applying refactoring techniques as defined by Fowler [23] on three software systems and measuring the impact on selected software metrics, an immense variation of the refactoring effect was found. Thus, the author concluded that he was unable to validate that refactoring as a practice improves quality.

A multi-project study on 23 open-source software projects and more than 29000 refactoring operations to study the effect on internal quality attributes was reported by Chavez et al [24]. The analysis revealed that 65% of the refactoring operations improve the internal quality as measured by a wide set of metrics, while 35% of the refactorings keep the quality attributes unaffected.

Although the above set of research studies is not exhaustive, most of the findings agree on the limited adoption of refactorings in practice and a rather mixed effect on software qualities, at least for quality aspects that can be captured by source code metrics. Such evidence calls for the systematic study of other strategies to sustain or improve quality in software systems over time.

2.3 Quality Gates

The aforementioned law of declining software quality during software evolution entails that it is not sufficient to write good code in the first place; code has to be *kept clean over time*. As Martin vividly states, this practice adheres to the "Boys Scouts of America" rule to *leave the campground cleaner than you found it* [11]. The simple and rational strategy of checking-in code that is cleaner than the average of the existing code-base will eventually yield continuous improvement in software quality. In this sub-section we focus on this strategy for reducing TD$_{density}$, i.e., by ensuring that new code commits do not violate a particular set of rules (i.e. do not introduce new Technical Debt Issues) [11], [15]. This strategy is based on the notion of quality gates [25].

Software engineers can use quality gates in order to set constraints, i.e., reject commits that contain any or particular code or design inefficiencies: In case a 'zero-defect' policy is adopted, the new code will essentially be TD-free. In practice, quality gates can be more flexible i.e., reject commits that contain smells of a given severity, type or priority level. Quality gates can be easily combined with Continuous Integration (CI) practices setting the maximum level of Technical Debt that is acceptable for new commits to the projects repository.

Janus et al. in 2012 [26] have proposed the 3C Approach. It is an extension to the Agile Practice Continuous Integration and it relies on quality gates for agile quality assurance combining software metrics with Continuous Integration. The proposed automated metric-based Quality Gate checks the source code and ensures that it does not exceed any of the defined thresholds before committing it to the version control system. This way the internal Software Quality is assured. In order to deploy and validate their method, they analyzed an agile project that was developed by a German Automotive Industry company and the results show that a significant improvement of its internal quality can be achieved.

Suryanarayana et al. [27] argued that smells are the result of violating some of the best practices and indicate higher-level design problems. They classified the smells based on the primary object-oriented design principle that they violate, namely: abstraction, modularization, and hierarchyduplicate abstraction, insufficient modularization, and multipath hierarchy smells. Based on an experiment/study that they conducted the found that one of the reasons that code smells are inserted into the project is the time pressure, thus the developers prefer to perform a quick (and dirty) fix rather than an appropriate solution. Finally, in order to avoid this symptom, they proposed a design quality gate process that checks if the modified/inserted code violates any of the predefined design-level rules.

Schermann et al. [28] acknowledge that Quality gates, as steps that ensure the reliability of code changes, lead to an inherent trade-off between sustaining a fast pace and risking a lower release quality. To address this issue they proposed a model where software releases are evaluated based on the Confidence (reliability) and Velocity (publishing speed). Their Confidence-Velocity Categorization Model consists of the following four categories: Cautious (low Velocity and high Confidence), Balanced (high Velocity and high Confidence), Problematic (low Velocity and low Confidence), and Madness (high Velocity and low Confidence).

Nevertheless, according to the empirical investigation by Vassallo et al. [29] Continuous Code Quality (CCQ) is not applied in practice. The authors attribute the low use of CCQ to the fact that code quality is not always the top priority for development teams but also the unawareness of how to properly set up quality gates.

3 Case Study Design

Case study is an empirical method that is used for studying phenomena (e.g., projects or activities) in a real-life context [30]. The case study of this paper has been designed and is presented according to the guidelines of Runeson et al. [31].

3.1 Goal and Research Question

The goal of this study is to compare addition, deletion and modification of code regarding their impact on TD$_{density}$. Moreover, to provide further insight to the relevant strategies, we study whether code quality practices are associated with the cleanness of new code. Therefore, we formulate two relevant research questions.

RQ$_1$: *Among the three major types of code changes (insertion, deletion and modification) which is primarily responsible for changes in Technical Debt density?*

RQ$_1$ aims at investigating whether changes in technical debt density from one code revision to the next are primarily associated with addition of new code, deletion or modifications of existing code. Each type of change can incur a negative or positive effect on the system's technical debt density depending on the quality of the code that is added, modified or removed. Code modifications can sometimes be related to the application of refactorings, but in the general case we assume that code changes are the result of maintenance and not necessarily targeting the removal of inefficiencies.

RQ$_2$: *Is the frequency of commits in which new code is cleaner compared to exiting code, associated with the existence of practices targeting code quality?*

RQ$_2$ aims at investigating whether a high percentage of cleaner new code commits is related to the use of practices targeting code quality. In terms of relevant practices we study two project management aspects: (a) the existence of commit guidelines (i.e. what the developer should have in mind before committing his/her code) which are directly or indirectly related to the avoidance of TD rule violations; and (b) the extent to which quality related issues (e.g., code improvement, code quality, refactorings, etc.) are discussed in project board meetings. Assessing the strength of the association can lead to interesting actionable outcomes, which can guide project managers on how to control the quality of their projects.

3.2 Cases and Units of Analysis

This study is characterized as multiple, embedded case study [31], in which the cases are open-source software (OSS) projects and the units of analysis are the revisions across the project history; we analyse changes to the system's TD$_{density}$ in these revisions. The reason for selecting to perform this study on OSS systems is the vast amount of data that is available in OSS repositories, in terms of revisions and classes, as well as quality-related practices. The long history that is available for each OSS project enables researchers to observe overall trends in the evolution of their quality. To retrieve data from only high-quality projects that evolve over a period of time, we have selected to investigate the projects presented in Table 1. We have decided to focus on Apache projects (similarly to the studies by Tan et al. [32] and Tufano et al. [9]) since the Apache Software Foundation, as an OSS development organization, has a reputation for high quality projects, for putting emphasis on process and quality improvement as well as for long-lasting projects.

The project selection process was based on the following criteria:

a. The project should be active (based on the date of its last commit) and therefore still maintained. This criterion aims at ensuring that the analyzed projects are still undergoing development and thus the studied practices are up-to-date. A similar prerequisite has been set by Rausch et al. [33] who studied the build failures in Continuous Integration (CI) workflows of open-source software.
b. The software should be written in Java and use Maven as a build tool. This ensures that the project can be built and allows the retrieval of the project's language version from the corresponding pom.xml file.
c. The software should contain more than 500 classes to ensure the inclusion of systems with a substantial size, functionality and complexity. A minimum number of system classes has also been set as a project selection criterion in the studies by Tan et al. [32] and Olbrich et al. [16].
d. The software should have more than 1000 commits and should be under development for at least 3 years. We have included this criterion for similar reasons to the previous criterion and to be able to observe trends in the evolution of the projects quality. Moreover, this number of revisions provides an adequate set of repeated measures as input to the statistical analysis. A minimum number of commits has also been used as a criterion in other studies on software evolution [33], [32], [16], [7], [34].

The selection of Apache projects enabled us to perform the analysis on RQ_2 which is based on the availability of minutes for Apache Board Meetings. However, we should note that there are also other Apache projects fulfilling the above mentioned criteria beyond those included in our dataset. Due to the complicated data analysis we have excluded projects that are extremely large, either in the number of classes or the number of commits.

TABLE 1: Selected Projects

Project	Classes	NCLOC	Analyzed Revisions
Accumulo	5840	428543	2863
Atlas	932	87637	1454
Beam	3757	176663	2780
Calcite	2606	186633	1448
Cayenne	2615	164170	2116
CXF	4111	353085	5079
DeltaSpike	951	46182	513
Drill	4655	316552	1316
Dubbo	943	61865	728
Flink	5632	341149	5329
Flume	790	51897	789
Giraph	1414	72972	668
Jackrabbit	2883	273574	4260
jclouds	5687	227459	4323
Knox	1083	51429	1033
Kylin	1658	128531	3205
Metron	1433	72579	548
MyFaces	1843	174158	1211
NiFi	4256	371031	1490
oozie	1082	97597	587
OpenWebBeans	561	44299	1583
PDFBox	1279	136916	3758
Pulsar	1837	147182	1503
SIS	1948	181588	828
Storm	3958	243574	738
TinkerPop	1698	95652	5178
Zeppelin	1209	89193	1562

3.3 Tracking the Types of Changes

Considering that software systems evolve through a number of revisions and that in each revision several types of changes may occur simultaneously, we look at the three major types of method changes: the development of new methods, the deletion or the modification of existing ones. These primary types of evolutionary changes have been considered in other studies as well [35], [36], [37] and [38]. As already mentioned, monitoring changes at the instruction level would be more complex and less accurate considering that several types of changes can simultaneously occur in some statements (e.g., modification and introduction of new code). Furthermore, tracking changes at the instruction level is challenging, as one would have to map each instruction (in a particular revision) to the corresponding instruction in the previous revision. This process is complicated by the insertion of new statements, comments, blank lines, etc. Therefore, to be certain about the classification of changes, we monitor changes at the method level.

Similarly, instead of assessing the entire technical debt of the analyzed systems, i.e., considering violations on every individual line of code, we have opted to consider only TD that can be mapped to class methods. In other words, we consider only SonarQube rule violations which reside in class methods. The reason is that Technical Debt Issues which occur at the class- or file-level (e.g., "The default unnamed package should not be used") are not associated with particular lines of code; as a result it would not be possible to assess what kind of code change caused their introduction or removal.

At each revision a method can be added, deleted, modified or remain unchanged. According to the stated research questions, the goal of this study is two-fold: (a) since multiple types of changes might occur simultaneously, to identify the type of change that has the largest impact on $TD_{density}$ change, i.e. whether the $TD_{density}$ change from one revision to the next is mostly due to the modification, the deletion or the addition of new methods, and (b) to investigate whether the frequency of 'clean' new code commits is related to overall project policies.

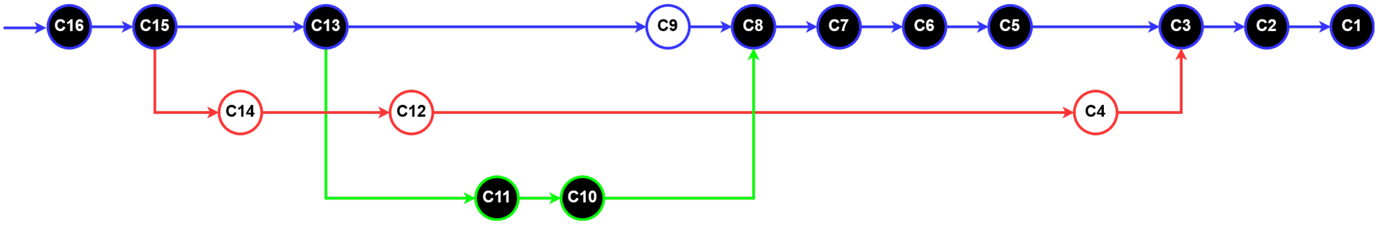

Fig. 1: Seeking the longest path between commit nodes

3.4 Data Collection

To analyze the projects and measure Technical Debt throughout their evolution, we have used SonarQube. SonarQube relies on a set of rules which are checked by static source code analysis; every time a piece of code breaks one of those coding or design rules, a technical debt issue is raised. Thus, SonarQube estimates the effort (in minutes) required to eliminate the identified Technical Debt issues[1]. This effort is obtained by assigning a time estimate for fixing each type of problem and by multiplying all issues of the same type with that estimate. It should be noted that the Apache Foundation ecosystem, has a dedicated SonarQube instance for quality control in its projects. Currently, 336 Apache projects are continuously monitored through SonarCloud, and 90.1% pass the quality criteria set by the development teams[2]. SonarQube reports various types of problems, namely code smells, bugs (issues representing something wrong in the code), vulnerabilities, code duplications and lack of test coverage. We note that in this study, we only consider code smells, since the other two types of problems (i.e., bugs and security vulnerabilities) do not fit the definition of TD [1], in the sense that they do not concern maintainability or evolvability.

For RQ_1 we measure the contribution to the $TD_{density}$ change at each revision that is due to (a) new methods, (b) removed methods, and (c) modified methods. For RQ_2 we consider the revisions in which the $TD_{density}$ of new code is lower than that of existing code.

3.4.1 Data Collection for Answering RQ_1

We have devised a process for analyzing git repositories which can be outlined in the following phases and individual steps:

Phase 1: Retrieval of commits

1) First, the Git history for the project under study is retrieved from its master (default) branch since it reflects the production-ready state of the project.
2) All commits are sorted to form a time series of revisions that have been performed on the source code. This process is non-trivial since even on the master (default) branch a commit can have multiple parents. To treat the data as a single time series, a single parent should be chosen for each commit without forming any branches. For the cases of commits with more than one parent, we have employed an algorithm aiming at identifying the longest path between the commit node under examination and the start node (i.e. the only node with no parent). As an example, Fig. 1 shows that commits 13 and 15 have more than one parents. We select for analysis the path consisting of the black nodes since it forms the longest possible path. Had we selected the series of commits as formed chronologically we would have run into inconsistencies among revisions: for example, if we analyze CM1, CM2, CM3, CM4, CM5 and so forth, any change on Technical Debt at commit CM4 would not be valid for the chronologically subsequent commit CM5, and changes to TD across revisions would yield irrational results. At the same time, the longest path yields the largest number of commits to be analyzed and thus results in a higher granularity for the analysis.
3) To reduce the computation time, a filtering step is applied: we ignore transitions between successive commits that do not involve any changes to Java files. We do this because the analysis of multiple revisions of large projects in SonarQube is extremely computationally-intensive resulting in several hours or even days for analyzing the entire history of the selected open-source projects.
4) From all commits submitted for analysis to SonarQube we retain only the successfully analyzed commits. The reason is that several commits may fail to analyze for various reasons, such as an incorrect pom.xml file that prohibits the build of the project.

Phase 2: Mapping of Technical Debt Issues to methods

To map the identified Technical Debt Issues to the class methods of each revision we perform the following steps:

1) First, for each revision, we retrieve all Technical Debt Issues by performing the corresponding query to the SonarQube database.
2) Next, we map the identified Technical Debt Issues to the methods of the corresponding revision. This is performed by matching the line in which each Technical Debt Issue is reported by SonarQube (in case the Technical Debt Issue concerns multiple lines, SonarQube reports the first one) with the method containing that line.

Phase 3: Tracking method changes

In order to associate variations in the overall $TD_{density}$ of a system with code changes at the method level, we track the type of change (introduction, deletion, modification) occurring to each method as follows:

1) For the new and deleted files of each revision (obtained from git history) we obtain their representation in the form of an Abstract Syntax Tree (AST)[3]. For each

[1] In this study we have considered Technical Debt issues reported as code smells by SonarQube

[2] https://sonarcloud.io/organizations/apache/projects

[3] The AST is obtained through the Eclipse Java Development Tools (JDT)

new/deleted file, we extract all its methods from the AST representation and then tag all these methods as new/deleted, respectively.

2) For the modified files of each revision we track new/deleted/modified/unchanged methods in each transition with the help of the Gumtree Spoon AST Diff tool [39].

Phase 4: Calculating the contribution of new/deleted/modified methods to the change in the system's $TD_{density}$

Finally, we need to calculate, for each revision in the system's history, the contribution of new/deleted/modified methods to the change of the system's $\text{TD}_{density}$. Let us consider a transition from revision *t-1* to revision *t*. To segregate the contribution of each type of change and at the same time ensure that the sum of all contributions is equal to the total change in the system's $\text{TD}_{density}$, we subtract the $\text{TD}_{density}$ of the previous revision from the $\text{TD}_{density}$ that is derived by the addition, removal or modification of code. The calculation is outlined in the following formulas:

Contribution of new methods

$\Delta TD_{density}(new)$ =

$$\frac{\text{TD}_{t-1} + \text{TD}_{new(t)}}{\text{LOC}_{t-1} + \text{LOC}_{new(t)}} - TD_{density}(t-1) \tag{1}$$

Contribution of deleted methods

$\Delta TD_{density}(deleted)$ =

$$\frac{\text{TD}_{t-1} \text{ - } \text{TD}_{deleted(t)}}{\text{LOC}_{t-1} \text{ - } \text{LOC}_{deleted(t)}} - TD_{density}(t-1) \tag{2}$$

Contribution of modified methods[4]

$\Delta TD_{density}(modified)$ =

$$\frac{\text{TD}_{t-1} \pm \Delta\text{TD}_{modified(t)}}{\text{LOC}_{t-1} \pm \Delta\text{LOC}_{modified(t)}} - TD_{density}(t-1) \tag{3}$$

As a result, the change in the system's $\text{TD}_{density}$ is equal to the sum of the individual contributions:

$$\begin{aligned} \Delta TD_{density}(system) = \Delta TD_{density}(new) \\ + \Delta TD_{density}(deleted) \\ + \Delta TD_{density}(modified) \end{aligned} \tag{4}$$

It should be noted that the data collection process has led to an enormous data set of approximately 1.4TB. A replication package with all data required to study the two RQs is available online[5].

3.4.2 Data Collection for Answering RQ$_2$

To explore the association of code quality practices and the quality of new code (RQ$_2$), we use the results of the descriptive statistics (the percentage of commits in which the new code is cleaner compared to existing code [CLEAN_CODE FREQ]), and two other variables. The first variable [COMMIT_GUIDELINES] is binary, and is set to true if: (a) the website of the project has clear and public guidelines for committers (usually termed "How to ..."); and (b) if at least one of the guidelines is not a purely aesthetic/formatting guideline (e.g., indent your code using tabs) and is directly or indirectly related to the rules being checked by SonarQube. A detailed reporting of how each project has been evaluated with respect to Commit Guidelines is presented in the online replication package. The second variable [PROJECT_BOARD_MEETINGS] is related to the emphasis of the project board on quality issues. In particular, for each Apache Software Foundation project, there is a regular meeting (usually every 3 months), in which the managers or key contributors of the project discuss the open issues and strategies for further improvement. To assign a value to [Project Board Meetings] variable, we have parsed the minutes of these meetings, aiming to identify discussions related to:

- *quality control (QC)*, for which we searched for the keywords: "software quality", "code quality", "code improvement", "code review", "guideline", or "sonar"
- *refactorings (REF)*, for which we searched for the occurrence of "refactoring" and "clean up"

and recorded the number of meetings in which each term was identified (variables [QC] and [REF]). Next, we calculated and rounded the MEDIAN value for the two variables ([QC] and [REF]). Every value that was higher than the rounded median was characterized as HIGH, whereas the rest as LOW. Projects characterized as HIGH in both perspectives, have been marked as HIGH in the [Project Board Meetings] variable, whereas all the rest as LOW. In other words, we classify projects in two categories based on the frequency by which project board meetings deal with code quality and refactoring strategies.

3.5 Data Analysis

To answer the research questions using the collected data we carried out both descriptive and inferential statistical analysis as follows:

3.5.1 Data Analysis for Answering RQ$_1$

The investigation of the contribution of each type of code change to the variation of the system's $\text{TD}_{density}$ across revisions is quite complicated, as the effect of the three types of changes (additions, deletions and modifications of methods) has to be taken into account. In particular, we are interested in observing whether positive (negative) changes in the system $\text{TD}_{density}$ co-exist with positive (negative) contributions stemming from new/deleted/modified methods.

Independent Variables

Contribution to $\text{TD}_{density}$ of new, deleted and modified code. These are categorical variables.
Categories: Leading to a decrease, increase or no change (stable) in $\text{TD}_{density}$.

Dependent Variable

Direction of $\text{TD}_{density}$ change during a transition from one revision to the next (the cases when the $\text{TD}_{density}$ remained stable are rare and are omitted for clarity). This is a categorical variable.
Categories: Increase/Decrease.

[4] In eqs. (2), (3) the denominator can not obtain the value zero under real circumstances, as this would imply that all lines of code are deleted or modified in a certain revision

[5] Replication package is available at https://drive.google.com/drive/folders/1mxher2vkE68GzKAkz1Y7rjVB0_hTihzt

Analysis

We first obtained contingency tables to describe the relationship between the two categorical variables. It should be noted that to investigate the effect of each type of change, we retained only the transitions when two types of changes occurred simultaneously, that is when two types of changes *compete* for the effect on the overall change in TD$_{density}$. In case only one type of change has occurred during a transition, then it is obvious that the change in TD$_{density}$ will be the result of this single change, and thus including such transitions in the data set would lead to misleading results. The results are displayed in the form of heat maps. For each project (row) three individual heat maps are shown, one for the contribution of new, deleted and modified code, respectively.

To further investigate this relationship, we performed a chi-squared test between the two categorical variables, to determine whether there is a significant relationship between them.

Null Hypothesis H$_0$: assumes that that there is no relationship between the direction of change in the system's TD$_{density}$ (decrease or increase) and the corresponding direction of change caused by new, deleted or modified methods.
Alternative Hypothesis H$_1$: Assumes that there is an association between the two variables.

Finally, to shed light into the effect of new methods vs. the effect of modified methods on code improvement, we illustrate graphically (in a bar chart) the percentage of transitions in which a reduction in the system's TD$_{density}$ co-occurred with positive contributions (i.e. leading to a decrease of TD$_{density}$) by new and modified methods, respectively.

3.5.2 *Data Analysis for Answering RQ$_2$*

To answer RQ$_2$, we explored whether: (a) projects that provide commit guidelines or (b) projects in which Project Board meetings often refer to code quality, are having a statistically significant higher average number of commits of cleaner code, compared to projects that do not provide guidelines and do not discuss code quality often.

Independent Variables

Binary variable [COMMITGUIDELINES] representing whether a project has commit guidelines related to code quality. Values: YES, NO.
Binary variable [PROJECTBOARDMEETINGS] representing the frequency by which quality issues are discussed in project board meetings. Values: LOW, HIGH.

Dependent Variable

The percentage of commits in which the new code is cleaner compared to existing code [CLEAN_CODE FREQ].

Analysis

We explored the discriminative power of the [COMMIT_GUIDELINES] and the [PROJECT_BOARD_MEETINGS] in terms of the [CLEAN_CODE FREQ] variable. To this end we have used boxplots to illustrate any differences in the percentage of cleaner code commits between projects that do not provide commit guidelines vs. those that provide them, and between projects that often refer to code quality issues vs. those that do it less often. Moreover, we have performed independent samples t-test to test any statistically significant differences. being

Null Hypothesis H$_0$: assumes that that there is no difference in the percentage of cleaner code commits, regardless of any adopted code quality practices (means are equal).
Alternative Hypothesis H$_1$: Assumes that the percentage of cleaner code commits differs depending on the adopted code quality practices (means are not equal).

4 Results

In this section we present the results of our study organized by research question, and highlight the major findings. However, prior to answering the research questions, we present a visualisation and descriptive statistics on the TD$_{density}$ of individual commits for the selected projects. This can provide the context upon which we can interpret the results of the research questions.

4.1 Descriptive Statistics

To obtain a first insight into the quality of new code as opposed to the quality of the system in which the new methods are added, we plot the evolution of the system's TD$_{density}$ along with the TD$_{density}$ of individual commits where new methods are added.

Figure 2 illustrates for one project (`Commons IO`)[6] the evolution of the system's TD$_{density}$ (black dots) and the corresponding trend-line, depicting a gradually increasing quality (black line declines over time). On the same plot, blue dots correspond to the revisions, in which the TD$_{density}$ of new methods was lower than that of the system in that revision, while red dots indicate the cases where the TD$_{density}$ of new methods was higher[7]. As it can be observed, for

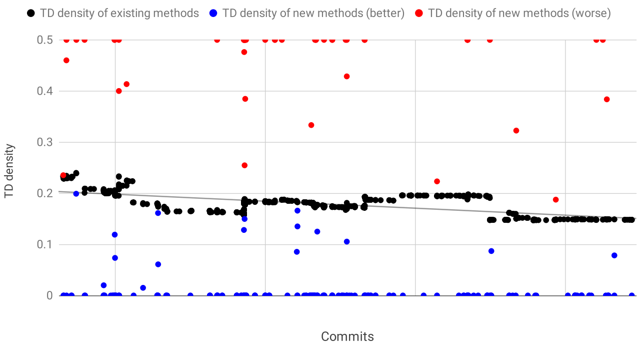

Fig. 2: Contribution of new code on system's TD$_{density}$ (Motivating Example)

[6] Project Commons IO is used as a motivating example and has not been included in our dataset since it is rather small and has a limited number of revisions

[7] For clarity, an upper bound on the displayed TD$_{density}$ values is imposed, that is, data points with an extremely high TD$_{density}$ are not accurately depicted but are simply placed on the upper bound (top of the figure).

TABLE 2: Percentage of revisions in which new methods have lower TD$_{density}$, for all examined projects

Project	%	Project	%	Project	%
Accumulo	65	Flink	77	NiFi	81
Atlas	73	Flume	73	oozie	79
Beam	71	Giraph	74	OpenWebBeans	77
Calcite	86	Jackrabbit	81	PDFBox	77
Cayenne	82	jclouds	84	Pulsar	72
CXF	77	Knox	69	SIS	68
DeltaSpike	72	Kylin	71	Storm	61
Drill	84	Metron	84	TinkerPop	75
Dubbo	81	MyFaces	71	Zeppelin	72

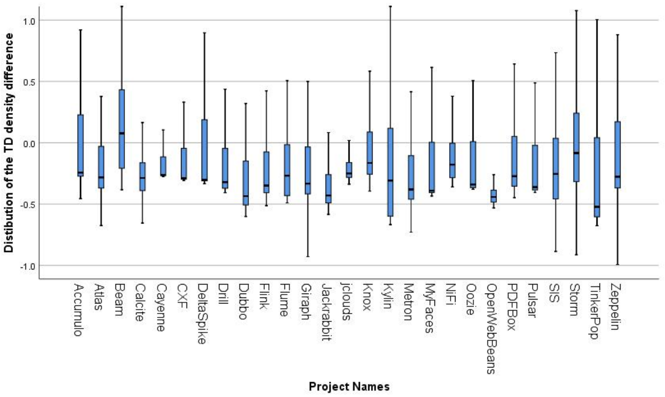

Fig. 3: Distribution of the difference between the TD$_{density}$ of new methods (introduced in new classes) and the TD$_{density}$ of existing system, for all projects

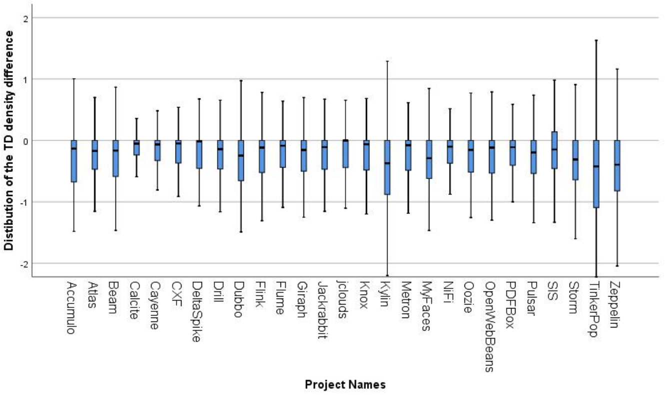

Fig. 4: Distribution of the difference between the TD$_{density}$ of new methods (introduced in existing classes) and the TD$_{density}$ of existing system, for all projects

the vast majority of revisions (77%), the TD$_{density}$ of new methods is lower than the TD$_{density}$ of the host system and in many cases the new code is entirely TD-free (see blue dots on the x-axis).

As it is not possible to show similar plots for all projects, Table 2 shows the percentage of revisions for which the TD$_{density}$ of new code was lower than the TD$_{density}$ of the system in the corresponding revision. The findings confirm the first impression that new code in the examined systems is generally of a higher quality than the existing baseline: in the majority of the commits, new methods have lower TD$_{density}$ than the host system. Considering that many of these systems have a quality that increases over time, it would be reasonable to argue that the cleanness of new code has contributed to the declining trend of the system TD$_{density}$. However, to claim that new code is a prominent factor that leads to the reduction of TD requires further analysis.

Moving on to more detailed descriptive statistics, the boxplots in Fig. 3 and Fig. 4 illustrate the distribution of the difference between the TD$_{density}$ of new methods and the TD$_{density}$ density of the host code. To allow for a fair comparison, we differentiate between the case when one or more new methods are introduced in an existing class and the case when a set of new methods are introduced in the form of a new class. In the former case the TD$_{density}$ of the new methods should be contrasted against that of the class in which they are added, since the class resembles the *neighborhood* of the new code, in terms of functionality and complexity. For the case of completely new classes, the comparison should be made against the entire system in which the new class is added, as the system is the *neighborhood* of the introduced class. The boxplot of Fig. 3 shows the distribution of the difference in TD$_{density}$ between new methods added in *new* classes at revision *i* and the quality of the entire system in the previous revision (*i-1*). The boxplot of Fig. 4 shows the distribution of the difference in TD$_{density}$ between new methods added in *existing* classes at revision *i* and the quality of the class in which they are added, in the previous (*i-1*) revision.

As it can be observed from the boxplots, the median difference (μ) between the TD$_{density}$ of new methods and that of the host code, is negative for all but one projects. For most of the projects, and especially for new methods introduced in existing classes, even the upper quartile is below zero. Thus, it becomes evident that in the transitions in which new code was added (in the form of entirely new methods either in new classes or in existing classes) the TD$_{density}$ of the new code is significantly lower than that of the host code while in many cases it is very close to zero.

4.2 Relation among the contribution of new / deleted / modified methods and change in system's TD$_{density}$ (RQ$_1$)

We remind that for studying this RQ we created two categorical variables: the first refers to the direction of change (decreasing/increasing[8]) of the system's TD$_{density}$ in each revision. The other refers to the contribution (decreasing, increasing, stable) of new/deleted/modified methods in each revision. The contribution itself is calculated according to equations (1)-(3). We have turned this contribution to a categorical variable depending on whether the contribution is positive, zero, or negative.

The results from the cross-tabulation of frequencies between these two variables are displayed in Table 3 for all the analyzed projects. For each project (composite row) three individual heatmaps are presented, one for the contribution of new, deleted and modified code respectively. Each heatmap is comprised of six cells: the two rows correspond to the

[8] The cases where the system's TD$_{density}$ remained stable are rare and are omitted for the sake of simplicity

TABLE 3: Relation between contribution of new/deleted/modified methods and change in system's TD$_{density}$ (RQ$_1$)

TD Density Change per Project		Contribution to TD Density per new/delete/modified methods								
		New			Deleted			Modified		
		↓	↑	-	↓	↑	-	↓	↑	-
Accumulo	↓	195	47	0	51	52	1	205	39	18
	↑	97	113	0	15	66	0	55	163	5
Atlas	↓	446	56	1	73	156	2	373	126	25
	↑	141	153	2	31	102	2	62	249	7
Beam	↓	550	86	8	137	235	11	542	131	33
	↑	172	198	8	44	184	8	90	311	17
Calcite	↓	607	27	0	44	157	3	446	165	32
	↑	209	108	0	23	102	5	53	277	1
Cayenne	↓	528	46	2	88	202	2	421	135	58
	↑	173	111	1	22	155	4	52	269	12
CXF	↓	119	109	7	90	285	7	917	262	97
	↑	344	310	3	35	178	4	142	548	15
DeltaSpike	↓	102	18	1	21	24	2	95	15	15
	↑	39	36	1	5	23	3	25	55	2
Drill	↓	521	51	1	77	181	6	427	118	35
	↑	212	78	1	17	104	4	23	274	3
Dubbo	↓	144	15	1	28	62	2	123	32	12
	↑	54	27	3	5	35	4	10	80	0
Flink	↓	1560	210	9	307	673	20	1314	465	76
	↑	477	402	5	126	416	13	222	756	18
Flume	↓	224	25	2	21	67	1	178	59	16
	↑	53	69	0	11	33	0	29	99	2
Giraph	↓	172	29	1	31	67	2	143	48	9
	↑	56	47	2	5	33	1	19	91	1
Jackrabbit	↓	915	74	1	111	298	7	711	245	89
	↑	212	170	6	35	149	4	86	345	16
jclouds	↓	1109	66	7	136	433	14	821	294	131
	↑	272	200	6	29	268	6	105	429	24
Knox	↓	271	35	21	44	68	2	241	74	24
	↑	68	88	7	23	44	5	55	124	5
Kylin	↓	714	127	1	177	295	4	688	174	47
	↑	255	254	4	60	211	7	116	460	16
Metron	↓	206	21	1	30	76	1	147	72	9
	↑	71	30	0	15	31	0	13	92	0
MyFaces	↓	205	17	0	37	57	3	176	56	21
	↑	65	96	0	11	44	3	37	138	6
NiFi	↓	473	32	2	50	113	3	337	144	36
	↑	136	97	0	16	67	2	55	188	2
oozie	↓	131	22	1	30	58	0	148	51	13
	↑	60	39	0	5	22	0	21	83	1
OpenWebBeans	↓	330	38	8	71	107	3	308	88	31
	↑	95	74	5	17	76	5	39	167	5
PDFBox	↓	385	38	1	60	85	1	345	66	46
	↑	107	110	0	13	68	1	56	172	11
Pulsar	↓	427	54	1	44	100	4	350	117	33
	↑	117	140	4	24	78	0	63	214	8
SIS	↓	205	22	1	52	98	1	152	74	10
	↑	81	94	1	31	84	2	44	144	2
Storm	↓	133	28	2	37	47	4	136	25	8
	↑	61	86	1	30	47	0	40	116	1
TinkerPop	↓	1170	151	2	200	477	23	949	299	145
	↑	414	393	5	80	350	25	174	696	25
Zeppelin	↓	368	57	0	68	91	3	332	93	27
	↑	155	137	1	22	63	3	61	236	9

increase or decrease in the system's TD$_{density}$, whereas the three columns to the effect (decrease, increase and stable) of the corresponding change type (new, deleted or modified). The intensity of the color (within each 6-cell heatmap) indicates the frequency of occurrence for each combination between the two categorical variables, that is, the direction of change in the system's TD$_{density}$ (decrease or increase), and the direction of change (decrease, increase and stable) for each type of contribution. It should be noted that while absolute numbers (number of transitions in which each combination has been observed) are shown on the heatmaps, the intensity of the colors reflects the corresponding percentage of cases (highest percentage corresponds to the most intense red).

Let us consider as an example, the first project in Table 3 (project `Accumulo`). We focus on the contribution of new code (composite column New) and study separately the two rows, corresponding to transitions where the system's TD$_{density}$ has decreased and increased, respectively:

- *System TD$_{density}$ has decreased (top row)*: The warm (red) color in the upper left cell (labeled with 195) implies that among the cases where new code was added and the system TD$_{density}$ has decreased, the highest frequency (195 out of the total 242 transitions) was observed for new methods that contributed to a decrease in the TD$_{density}$.

- *System TD$_{density}$ has increased (bottom row)*: The red color in the lower center cell (labeled with 113) implies that among the cases where new code was added and the system TD$_{density}$ has increased, the highest frequency (113 out of the total 210 transitions) was observed for new methods that contributed to an increase in the TD$_{density}$.

In other words, in this project, the change in the system TD$_{density}$ co-occured in most of the cases with a contribution of the same direction by new code.

Following this kind of interpreting Table 3, for the contribution of new methods in all projects, it can be observed that when their contribution leads to a reduction of TD$_{density}$, in most of the revisions the same direction of change is observed in the system's TD$_{density}$ (warm red color in the top-left cell in each six-cell heatmap). In other words, in most of the cases, **when new code is *cleaner*, the system's TD$_{density}$ decreases**. However, an impact on the system's TD$_{density}$ cannot be claimed when new code contributes to an increase of the TD$_{density}$. It should be noted that this pattern is consistent among all projects. For deleted methods the most striking observation (most red cells) concerns the cases when the deleted methods contribute to an increase of the TD$_{density}$ (for example, when high quality code is removed from the system). In those cases it seems that **the deletion of high quality code most frequently co-exists with an increase in the system's TD$_{density}$**.

An interesting and repeating pattern is present for modified methods. Intense red colors are observed in alternating rows: this implies that the direction of change in the system's TD$_{density}$ coincides with the contribution of modified code. In other words, **if a method is modified and the TD$_{density}$ of the method decreases, then, for the majority of the cases a decrease in the system's TD$_{density}$ is observed; similarly for an increase in the TD$_{density}$**.

Finally, as expected, the lack of any contribution of new/deleted/modified methods (column –) does not appear to have any association with the overall change in the system's TD$_{density}$ as depicted by the mostly blue cells. It should be emphasized, that these observations, should by no means be interpreted as indications of causality. Investigating whether each type of change (new/deleted/modified code) is responsible for the change in the overall TD$_{density}$ would require a different experimental set up and is beyond the scope of this study.

The chi-square test for independence has been used to discover if there is a relationship between the direction of change in the project's TD$_{density}$ and the contribution of new/deleted/modified code. Table 4 shows for each project the Pearson chi-square value (top row) and whether the results are statistically significant or not depending on the p-value. The bottom row for each project shows the Phi value that tests the strength of the association [40].

As it can be observed, in almost all cases the results are statistically significant (p<0.01) implying that the null hypothesis can be rejected. In other words we can argue that there is a relationship between the direction of change in the systems TD$_{density}$ and the contribution of new/deleted/-modified code. In particular, **the strength of the association appears to be higher for the contribution of modified methods (for most projects), followed by the contribution of new code. The contribution of deleted methods seems to have a significantly lower association to the change in the system's TD$_{density}$**.

TABLE 4: Chi-squared test for New, Deleted, and Modified methods

Project		New	Deleted	Modified
Accumulo	χ^2	61.55**	20.32**	174.75**
	ϕ	0.368	0.324	0.595
Atlas	χ^2	227.08**	75.86**	238.93**
	ϕ	0.532	0.455	0.532
Beam	χ^2	252.51**	32.53**	371.67**
	ϕ	0.495	0.227	0.572
Calcite	χ^2	155.09**	2.49	301.03**
	ϕ	0.403	0.086	0.555
Cayenne	χ^2	154.05**	29.40**	306.58**
	ϕ	0.421	0.247	0.565
CXF	χ^2	499.40**	18.52*	618.96**
	ϕ	0.512	0.174	0.555
DeltaSpike	χ^2	46.84**	9.18	68.50**
	ϕ	0.484	0.337	0.568
Drill	χ^2	50.00**	11.66	405.69**
	ϕ	0.240	0.173	0.677
Dubbo	χ^2	24.30**	9.21	118.01*
	ϕ	0.315	0.258	0.675
Flink	χ^2	425.38**	19.24*	686.37**
	ϕ	0.399	0.111	0.488
Flume	χ^2	95.41**	0.52	100.28**
	ϕ	0.505	0.062	0.511
Giraph	χ^2	74.01**	13.22	106.43**
	ϕ	0.489	0.304	0.582
Jackrabbit	χ^2	289.10**	10.65	380.25**
	ϕ	0.457	0.132	0.502
jclouds	χ^2	486.34**	30.43**	464.59**
	ϕ	0.539	0.184	0.504
Knox	χ^2	154.25**	42.84**	165.54**
	ϕ	0.559	0.475	0.559
Kylin	χ^2	193.14**	45.06**	512.66**
	ϕ	0.377	0.243	0.582
Metron	χ^2	22.99**	0.72	91.61**
	ϕ	0.264	0.069	0.524
MyFaces	χ^2	198.44**	10.18	129.66**
	ϕ	0.715	0.251	0.541
NiFi	χ^2	360.68**	38.11**	189.47**
	ϕ	0.695	0.387	0.496
oozie	χ^2	34.55**	2.81	87.28**
	ϕ	0.337	0.156	0.523
OpenWebBeans	χ^2	97.28**	21.01**	208.81**
	ϕ	0.420	0.271	0.568
PDFBox	χ^2	237.85**	27.78**	231.95**
	ϕ	0.605	0.344	0.572
Pulsar	χ^2	288.13**	16.38*	212.82**
	ϕ	0.622	0.255	0.519
SIS	χ^2	227.86**	35.03**	122.02**
	ϕ	0.750	0.360	0.533
Storm	χ^2	133.34**	4.79	117.20**
	ϕ	0.654	0.170	0.599
TinkerPop	χ^2	480.66**	30.52**	706.49**
	ϕ	0.473	0.161	0.551
Zeppelin	χ^2	100.30**	9.87	240.53**
	ϕ	0.374	0.197	0.561

** $p < 0.01$, * $p < 0.05$

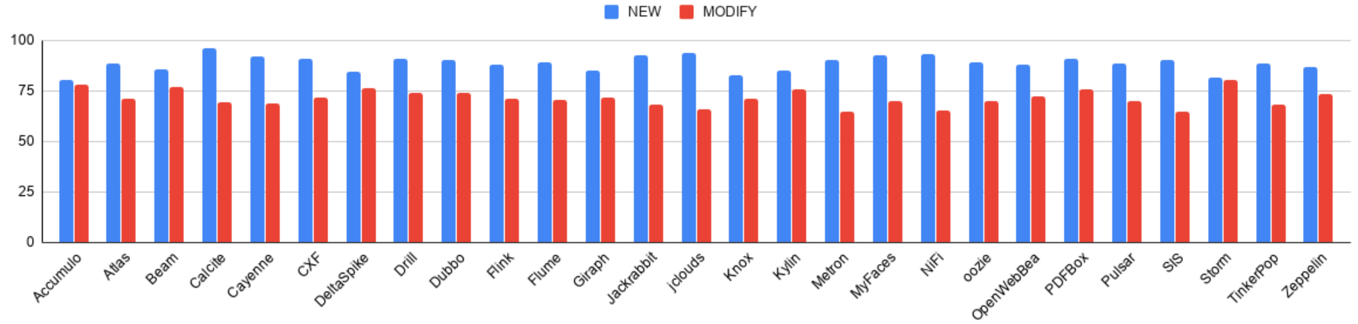

Fig. 5: Percentage of revisions in which a decrease in the system TD$_{density}$ co-occurred with a positive contribution in quality by new and modified methods

TABLE 5: Adoption of Commit Guidelines and frequency of references to code quality issues in board meetings per project

Project	Commit Guidelines	Project Board Meetings
Accumulo	NO	LOW
Atlas	NO	LOW
Beam	NO	HIGH
Calcite	YES	HIGH
Cayenne	NO	LOW
CXF	YES	LOW
DeltaSpike	YES	LOW
Drill	YES	HIGH
Dubbo	YES	LOW
Flink	YES	LOW
Flume	NO	LOW
Giraph	NO	LOW
Jackrabbit	NO	HIGH
jclouds	YES	LOW
Knox	NO	LOW
Kylin	YES	LOW
Metron	YES	HIGH
MyFaces	YES	LOW
NiFi	YES	HIGH
oozie	NO	LOW
OpenWebBeans	NO	HIGH
PDFBox	YES	HIGH
Pulsar	YES	LOW
SIS	NO	HIGH
Storm	YES	LOW
TinkerPop	YES	LOW
Zeppelin	YES	LOW

These results concern both directions of change in the system's TD$_{density}$ and reveal that code modification can contribute positively and negatively to changes in the system quality. If we focus only on the cases where the TD$_{density}$ decreased from one revision to the next, the potential of cleaner new methods becomes more evident: The barchart of Figure 5 displays the percentage of transitions where a decrease of the system's TD$_{density}$ was observed and new/modified methods also contributed to a decrease of TD$_{density}$ (assuming that at least two types of changes were competing in the same transition). The cases where a positive contribution by new methods co-occurred with an improvement in the overall quality, are slightly more frequent.

4.3 Relation between Code Quality Practices and New Code Cleanness (RQ$_2$)

In Table 5, we present the data extraction results for the studied projects, with respect to the employment of code quality practices at project management level.

With respect to the existence of commit guidelines, we can observe that 16 projects provide guidelines related to TD rule violations, and 11 projects do not. By comparing the median values of the percentage of cleaner code commits (see Figure 6) in the two groups, we can observe that projects that provide commit guidelines are having more commits (an increase of the median by 3.88% has been observed) in which the new code is cleaner compared to existing code. However, this difference is not statistically significant, based on the independent samples t-test (sig = 0.09).

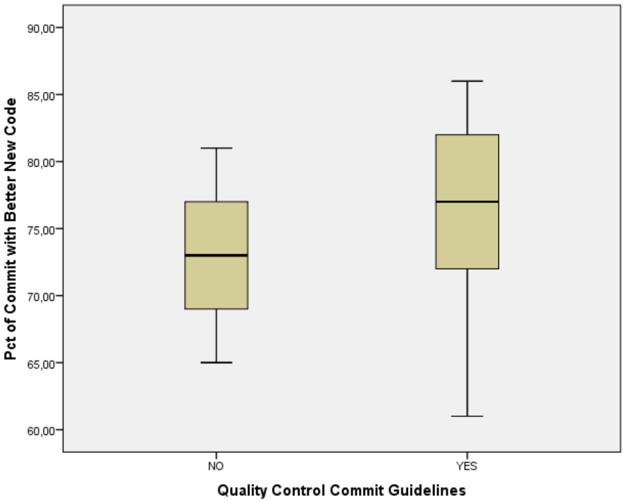

Fig. 6: Distribution of the percentage of cleaner code commits for the two groups of projects, based on the existence of commit guidelines

On the other hand, **the 9 projects in which the project management team more regularly discusses code quality in the board meetings, are having a statistically significant higher percentage of commits in which the code is cleaner** (mean$_{diff}$ = 5%, and sig = 0.05). This difference is visualized in the boxplots of Figure 7, in which we can observe no overlap in the boxes (Q3-Q1) of the two groups.

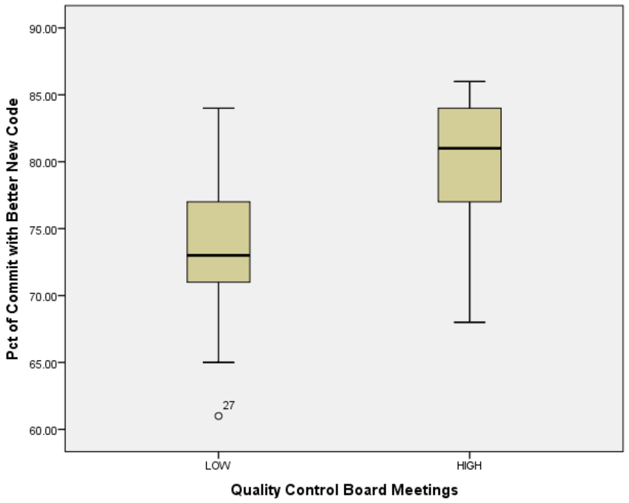

Fig. 7: Distribution of the percentage of cleaner code commits for the two groups of projects, based on the frequency of references to code quality in board meetings

5 Discussion

In this section we first discuss in detail the findings presented in Section 4, attempting our own interpretation. Then, we list potential implications for researchers and practitioners.

5.1 Interpretations of the Results

5.1.1 Quality of new code vs. system quality (Descriptive Statistics)

The descriptive statistics revealed that new code (in the form of new methods), exhibits quality (in terms of TD$_{density}$), that is higher compared to the quality of the systems in which the new methods are introduced. This finding was consistent in all examined projects. For many of the projects in the Apache Software Foundation ecosystem this could be linked to the observed declining trend of the project's TD$_{density}$, although this study cannot establish causality between cleaner new code and improvement of overall quality. It still remains to be studied how developers ensure that new code is cleaner in terms of technical debt. It might be a deliberate choice (e.g., in case the development team applies a Quality Gate to ensure zero or low number of violations for each new commit) or a general trend resulting from improvements in the employed processes and tools or simply the result of higher developer experience.

It is also noteworthy that in a very large percentage of the analyzed commits, new methods had zero or very low TD$_{density}$. In particular, the findings from the selected projects, suggest that the overall system quality can improve (i.e. TD$_{density}$ can decrease) over time if the TD$_{density}$ of new code is systematically kept below the system's average. We caution, that this does not imply a quick fix to a system's quality (in terms of TD); if the existing code base is largely of low quality, the improvement that can be achieved through new commits is limited in the short term. Nevertheless, the improvement of TD$_{density}$ in new code has merit in the sense that most systems evolve for years; thus, in the long run, systematically writing cleaner new code can yield substantial code improvement. This observation, emphasizes the importance of writing clean new code over the practice of refactoring. We also note that several empirical studies have shown that systematic, bad smells refactoring is uncommon in most open-source software projects, rendering the practice of writing clean new code even more valuable. This result can be of particular worth, in the sense that writing clean code can be a best practice for limiting the *software ageing* phenomenon [41] or the 7^{th} of Lehman's laws of evolution which states that the quality of software deteriorates over time [15].

5.1.2 Relation among the contribution of new / deleted / modified methods and change in system's TD$_{density}$ (RQ$_1$)

The in-depth study on the association between the contribution of new / deleted / modified methods and the observed changes in the system's TD$_{density}$ revealed some interesting patterns: (a) among all transitions where the system's TD$_{density}$ has decreased and two or more types of changes were competing, the most frequent case was the addition of new methods that were better in terms of TD$_{density}$, implying that the introduction of clean new code coincided with an improvement in quality; (b) the contribution of method removal is rather mixed, which is reasonable, as the effect of method deletion on the system's TD$_{density}$ depends on the quality of the removed code (the removed methods can be either high or low quality code); and (c) the contribution (positive or negative) of method modification coincides with the direction of change in the system's TD$_{density}$. For the latter case it should be emphasized that modifications refer to any type of changes to a method including adaptive and corrective maintenance.

The chi-square test of independence showed vividly that there is a statistically significant relationship between the direction of change in the system's TD$_{density}$ and the contribution of new, deleted and modified code. Considering that code deletion is not usually performed on the basis of quality improvement but rather dictated by functionality-related reasons, the improvement of code quality is subsequently left up to the addition and modification of code. Method modification has a clear association with the overall technical debt: lowering the TD$_{density}$ of a method during maintenance will reduce the system's TD$_{density}$ and vice-versa. On the other hand, cleaner new methods (which according to the descriptive statistics are very common) have a strong association to a decreasing system TD$_{density}$. These results, further emphasize the importance of writing high-quality new code and monitoring the introduced number of Technical Debt Issues in new code. To provide insight into how TD can be eliminated or avoided during software evolution, we provide two real examples: the first refers to code modification and the second to the introduction of clean new code.

The first example (**code modification**) refers to a pull request in project `Dubbo` (#3474) where the purpose of the change was to properly close resources after use, thereby eliminating an existing TD issue. The intention is also reflected on the title of the pull request: 'Fix Not Properly Closed Resources'. The affected code, prior to the change, was:

```
try {
  UnsafeByteArrayInputStream is = new
  UnsafeByteArrayInputStream((byte[]) args[i]);
 . . .
 } catch (Exception e) {
  . . .
 }
```

and thus was missing a proper close call within a finally block, causing a Blocker issue according to SonarQube. The change in the pull request targeted exactly that problem and fixed it by using the try-with-resources statement, that declares the resources to be closed after the program is finished (which is equivalent to using a finally block prior to Java SE 7). The corresponding code was modified to (note that the resource is declared within parentheses after the try keyword):

```
try(UnsafeByteArrayInputStream is = new
  UnsafeByteArrayInputStream((byte[]) args[i])){
     . . .
   }
   catch (Exception e) {
     . . .
   }
```

thereby, eliminating the abovementioned SonarQube issue.

The second example (**of clean new code introduction**) refers to a pull request in the same project `Dubbo` when new methods are introduced in the new NettyClientHandler class. Up to that point, the code suffered from multiple TD issues related to improper handling of Exceptions, violating the major 'Throwable and Error should not be caught'rule 349 times. Noncompliant code examples are of the following form:

```
try {/* ... */} catch (Throwable t) {/* ... */}
try {/* ... */} catch (Error e) {/* ... */}
```

The rationale for this rule is that Throwable is the superclass of all errors and exceptions in Java, while Error is the superclass of all errors, which are not meant to be caught by applications. Catching either Throwable or Error will also catch OutOfMemoryError and InternalError, from which an application should not attempt to recover[9]. Class NettyClientHandler in pull request #630 is TD free and while it deals with exception handling in all of its methods, none of the methods violates the aforementioned rule.

5.1.3 Code Quality Practices and Cleanness of New Code (RQ$_2$)

We observed that projects in which Code Quality is often being discussed among the management team, exhibit a statistically significant higher percentage of commits with code that is cleaner than the existing codebase. These discussions are sometimes very explicit about the use of tools to measure code quality or the emphasis on cleaning up the code. As an example, in an Apache `PDFBox` board meeting of 2015, under 'Software Quality' it is mentioned that *'There is an ongoing effort to improve `PDFBox` based on the analysis of different tools such as SonarQube, FindBugs and others'*. In a March 2019 meeting of project `Flink` under 'Status' it is noted that *'The release contains some new user-facing features plus a lot of internal cleanup and refactoring, fixing some long term issues ...'*.

Apache Project Management Committees (PMCs) are required to report on their project's health and status quarterly to the Board of Directors. We have observed in our study that projects with a certain level of size and complexity, code quality in general and maintainability in particular becomes a major concern. Guiding the hundreds of volunteers in open-source projects on how to commit high quality code can be facilitated by the use of tools/practices which essentially dictate a minimum threshold of quality that has to be reached before submitting code. We argue that such a 'clean new code' policy is also applicable to industrial projects as a means of sustaining, and even improving TD.

5.2 Implications for Practitioners

Regarding software developers, evidence from the presented case study suggests that new code can have a substantial impact on the quality of an evolving system. The fact that the contribution of new code has a strong association to the changes in the system's TD$_{density}$ implies that writing clean new code can ensure, to a large extent, the gradual improvement of the overall system quality. Of course, code modifications that result in lower TD$_{density}$, either in the form of refactorings or as carefully applied maintenance, has also potential for improving code quality.

In terms of software development strategies, we believe that using Quality Gates to enforce a predefined quality policy can be a simple, yet effective mechanism to manage technical debt in the long term. The findings on the second research question revealed that projects where code quality is a frequent topic in board meetings, are having higher chances to reduce their TD density through the improvement of new code. The existence of explicit commit guidelines was not found to be significantly associated with the frequency of cleaner commits at the project level. However, it is noteworthy that several projects express explicit concerns about code quality in the commit guidelines offered to potential contributors. We list representative guidelines associated with code quality posted in the projects' websites in Table 6. Such guidelines from well-known Apache projects can be considered as best practices and thus reused in other projects.

Ensuring that new code commits are as TD-free as possible, is a promising way of sustaining quality and avoiding quality degradation over time. Putting a Quality Gate in place is a relatively low-cost approach that sets an easy-to-manage, everyday goal for software developers, explicitly emphasizing code quality.

5.3 Implications for Researchers

With respect to researchers working in the area of Technical Debt Management the obtained evidence opens up further opportunities for studies on the impact of new code on quality. Research can focus on establishing whether systems with a degrading quality over time are associated with high TD$_{density}$ of new code and vice-versa. Given that no project exhibits a monotonous trend in its quality over time, it would be reasonable to perform such studies after splitting the timeline of system history into periods with monotonous

[9] https://rules.sonarsource.com/

TABLE 6: Representative Commit Guidelines

Project	Representative Commit Guidelines
Calcite	*Trigger a Coverity scan ... and when it completes, make sure that there are no important issues.*
CXF	*Make use of both PMD and CheckStyle to enforce common coding conventions.*
DeltaSpike	*Follow project's formatting rules and always build and test your changes before you make pull requests*
Drill	*Code should be formatted according to Sun's conventions, contributions should not introduce new Checkstyle violations, contributions should pass existing unit tests, and new unit tests should be provided to demonstrate bugs and fixes*
Dubbo	*Dubbo uses code style that is almost in line with the standard java conventions and suggests the contributors to implement a few unit tests for a new feature or an important bugfix.*
Flink	*Flink i) suggests the developers to comment as much as necessary to support code understanding, ii) provides guidelines for good design and software structure, iii) guidelines for good concurrency and threading and iv) suggestions for dependencies and modules*
jclouds	*Contexts and APIs are thread-safe (or should! Otherwise it is an issue)*
Knox	*Adding new service API support, the committer should give sufficient tests and documentation*
Kylin	*The changes MUST be covered by a unit test or the integration test, otherwise it is not maintainable*
Metron	*Try-finally used as necessary to restore consistent state, Appropriate NullPointerException and IllegalArgumentException argument checks*
MyFaces	*Error and exception handling: If the exception is severe, but there is a chance to continue processing, a message with severity "error" or "warning" should be logged..*
NiFi	*If an unexpected RuntimeException is thrown, it is likely a bug and allowing the framework to rollback the session will ensure no data loss*
PDFBox	*The new code should follow the project's coding conventions where possible.*
Pulsar	*All code should have appropriate unit testing coverage. New code should have new tests in the same contribution. Bug fixes should include a regression test to prevent the issue from reoccurring.*
Storm	*The most important is consistently writing a clear docstring for functions, explaining the return value and arguments. As of this writing, the Storm codebase would benefit from various style improvements..*
TinkerPop	*A rich set of algorithms is an important goal for MLLib, scaling the project requires that maintainability, consistency, and code quality come first.*
Zeppelin	*The project follows Google's Java Code style and suggests the developers to use some formatting plugins to lint their code*

and statistically significant trends in their quality in terms of TD$_{density}$. Declaring that clean new code results in improving quality over time would emit an explicit message to software developers.

Furthermore, the use of refactoring miners can be exploited to investigate which strategy (i.e., writing new code vs. applying regular refactorings) is more efficient for managing technical debt. Evidence on this central question would be highly relevant to software development teams considering both the effort that is associated with each type of quality-improvement approach as well as the attractiveness of each coding activity to developers. Moreover, it would be equally interesting to assess the quality of new code commits pertaining to specific change types, classified from the view point of the goal of change. Such classifications consider for example changes due to fault-fixing, feature addition, enhancements or general maintenance. [42], [43]. Another research direction worth of investigation would be the study of 'ripple' effects of new code to the rest of the system.

Further studies that compare projects that rely on tools such as SonarQube and others that do not, could reveal whether the use of such platforms leads to TD reduction. Moreover, it makes sense to investigate whether there is any relation between the experience of developers and the quality of new code that they introduce. Apart from the analysis of software artifacts we believe that it would be highly interesting to reach out to management boards of open source or industrial projects to obtain their own perception on Technical Debt in new code and analyze their strategies for preventing its accumulation.

6 Threats To Validity

In this section, we present and discuss threats to the validity of the study, including threats to construct, external validity and reliability. The study does not aim at establishing the presence of cause-and-effect relationships, thus it is not concerned with internal validity.

6.1 Construct Validity

Construct validity reflects how far the examined phenomenon is connected to the intended studied objectives. The main involved threat is related to the accuracy by which technical debt can be captured by static analysis tools such as SonarQube. Rule violations reported as Technical Debt Issues are obviously only one manifestation of actual code and design inefficiencies. The lack of any ground truth in technical debt measurement means that the accuracy of SonarQube, or any other technical debt tool, can hardly be validated. According to Martini et al. [44], currently static analyzers (such as SonarQube) are used in industry to analyze the source code in search of technical debt. Only in few cases out of the respondents in their survey (15 companies) practitioners built their own metrics tools for checking (language-specific) rules or patterns that can warn the developers of the presence of technical debt. In a similar discussion, Yli-Huumo et al. [45] found SonarQube to be the most used tool for TDM in the eight development teams that were involved in their case study.

Furthermore, it is known that such tools are not capable of identifying architectural problems or other types of technical debt such as requirements, documentation, test or build debt. This threat however, is partially mitigated by the fact that SonarQube is one of the most widely used tools for technical debt identification and quantification. Moreover, we have used the same tool both for assessing the system's TD$_{density}$ as well as the contribution of new / deleted / modified code on the TD$_{density}$ change. Furthermore, the analysis of TD$_{density}$ evolution is based mostly on a relative assessment of the changes rendering the measurement of absolute technical debt values less important for this type of study.

In addition, we investigate the effect of code insertion, deletion and modification on changes in the system's

TD$_{density}$. Generally, these three types correspond to all possible changes in terms of code. However, we consider only Technical Debt that can be mapped to methods, thus ignoring TD which might occur at the level of classes or files. SonarQube reports violations at the class/file level; however, tracking the types of changes that can introduce or remove such violations requires a different study set-up. More importantly, we consider the co-occurrence of changes (e.g. an increase of TD$_{density}$ by new code and an increase in the system's TD$_{density}$). Thus, we do not capture the underlying causes of variation in the number of Technical Debt issues (such as the introduction of a new library or framework, the implementation of new functionality, etc.).

6.2 Reliability

Reliability reflects whether the study has been conducted and reported in a way so that others can replicate it and reach the same results. To mitigate this threat, the study protocol is extensively described in Section 3 explicitly listing all data collection and analysis steps. It should be emphasized that data has been subject only to automated analysis with no subjective interpretation from the researchers; therefore, researcher bias has been avoided. A replication package is available with all available data to allow for an independent replication of the investigation. Replication to other software ecosystems like the ones by Google, Android and Salesforce is deemed particularly important for validating the findings on the effectiveness of clean new code.

6.3 External Validity

External validity examines the applicability of the findings in other settings, e.g. other software projects, other programming languages and possibly other technical debt identification tools. We have focused only on Java open-source software projects that use maven as a build tool. This limits the ability to generalize the findings to other projects. The fact that the study focuses on twenty-seven projects of the Apache Software Foundation which are highly active and popular among software developers partially mitigates threats to generalization. Nevertheless, the focus on a specific software ecosystem is still a source of threat to the generalizability of the results. Last but not least, the introduction of TD through new code has been analyzed in this study only on open source projects. In industry, projects are characterized by a very tight schedule; this means that our findings cannot be generalized to industrial systems. Thus, replication studies on the effect of new code on the evolution of technical debt are needed to strengthen the validity of the derived conclusions to industrial systems or systems that use different programming languages and environments.

7 Conclusions

The Technical Debt metaphor is usually associated with degrading quality trends in software systems. However, not all systems *age* over time. In this paper we have made an attempt to shed light into the drivers of technical debt change, by focusing on the contribution of new, deleted and modified code. In particular, we have performed an empirical study on the entire history of twenty-seven open-source projects by analyzing the changes, at method level, for each individual commit. By mapping Technical Debt Issues to new, deleted, and modified methods we have been able to investigate the association between the types of changes and the variation in the system's TD$_{density}$.

The results revealed that the quality of new code in terms of its TD$_{density}$, is, for the majority of the revisions, higher than the quality of the system in which the code is introduced. Moreover, among new, deleted, and modified code, the contribution of code modification exhibits a strong association to the change in the system's TD$_{density}$, followed by the contribution of new code. The contribution of new code is more profound for the transitions in which the quality of the system improved. More specifically, it was found that adding new code that is *cleaner* than the existing codebase coincides very frequently with a reduction in TD$_{density}$ of the system. The same association, has also been observed for the contribution of method modification to the change of the total TD$_{density}$. Finally, we have found indications that projects in which code quality is often discussed in their board meetings, exhibit a higher frequency of cleaner code commits.

The findings of this study suggest that writing code that has fewer Technical Debt Issues than the host code, can prove a very efficient and low-cost approach for managing TD. Applying Quality Gates to ensure that each commit yields fewer violations than the average, essentially leads to an improving quality trend, thereby reversing the software ageing phenomenon. Further studies can reveal whether writing clean new code offers a better cost/benefit ratio than the widely studied strategy of software refactoring.

Acknowledgments

Work reported in this paper has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 780572 (project SDK4ED) and under grant agreement No 801015 (project EXA2PRO).

References

[1] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering (dagstuhl seminar 16162)," in *Dagstuhl Reports*, vol. 6, no. 4. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[2] G. Digkas, M. Lungu, A. Chatzigeorgiou, and P. Avgeriou, "The evolution of technical debt in the apache ecosystem," in *European Conference on Software Architecture*. Springer, 2017, pp. 51–66.

[3] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, "Metrics and laws of software evolution-the nineties view," in *Proceedings Fourth International Software Metrics Symposium*. IEEE, 1997, pp. 20–32.

[4] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2011.

[5] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, vol. 101, pp. 193–220, 2015.

[6] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of code smells in object-oriented systems," *Innovations in Systems and Software Engineering*, vol. 10, no. 1, pp. 3–18, 2014.

[7] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 411–416.

[8] R. Arcoverde, A. Garcia, and E. Figueiredo, “Understanding the longevity of code smells: preliminary results of an explanatory survey,” in *Proceedings of the 4th Workshop on Refactoring Tools*. ACM, 2011, pp. 33–36.

[9] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, “When and why your code starts to smell bad (and whether the smells go away),” *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.

[10] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, “An empirical investigation into the nature of test smells,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 4–15.

[11] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[12] R. Potvin and J. Levenberg, “Why google stores billions of lines of code in a single repository,” *Communications of the ACM*, vol. 59, no. 7, pp. 78–87, 2016.

[13] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, “Identification and management of technical debt: A systematic mapping study,” *Information and Software Technology*, vol. 70, pp. 100–121, 2016.

[14] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, P. Abrahamsson, A. Martini, U. Zdun, and K. Systa, “The perception of technical debt in the embedded systems domain: an industrial case study,” in *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*. IEEE, 2016, pp. 9–16.

[15] M. M. Lehman, “Laws of software evolution revisited,” in *European Workshop on Software Process Technology*. Springer, 1996, pp. 108–124.

[16] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, “The evolution and impact of code smells: A case study of two open source systems,” in *2009 3rd international symposium on empirical software engineering and measurement*. IEEE, 2009, pp. 390–400.

[17] A. Yamashita and L. Moonen, “Do developers care about code smells? an exploratory survey,” in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 242–251.

[18] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, “Lessons from building static analysis tools at google,” *Communications of the ACM*, vol. 61, no. 4, pp. 58–66, 2018.

[19] K. Stroggylos and D. Spinellis, “Refactoring–does it improve software quality?” in *Fifth International Workshop on Software Quality (WoSQ’07: ICSE Workshops 2007)*. IEEE, 2007, pp. 10–10.

[20] B. Du Bois and T. Mens, “Describing the impact of refactoring on internal program quality,” in *International Workshop on Evolution of Large-scale Industrial Software Applications*, 2003, pp. 37–48.

[21] D. Wilking, U. F. Kahn, and S. Kowalewski, “An empirical evaluation of refactoring.” *e-Informatica*, vol. 1, no. 1, pp. 27–42, 2007.

[22] M. Alshayeb, “Empirical investigation of refactoring effect on software quality,” *Information and software technology*, vol. 51, no. 9, pp. 1319–1326, 2009.

[23] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[24] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia, “How does refactoring affect internal quality attributes? a multi-project study,” in *Proceedings of the 31st Brazilian Symposium on Software Engineering*, 2017, pp. 74–83.

[25] D. Falessi, B. Russo, and K. Mullen, “What if i had no smells?” in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2017, pp. 78–84.

[26] A. Janus, R. Dumke, A. Schmietendorf, and J. Jäger, “The 3c approach for agile quality assurance,” in *2012 3rd International Workshop on Emerging Trends in Software Metrics (WETSoM)*. IEEE, 2012, pp. 9–13.

[27] G. Suryanarayana, T. Sharma, and G. Samarthyam, “Software process versus design quality: Tug of war?” *IEEE Software*, no. 4, pp. 7–11, 2015.

[28] G. Schermann, J. Cito, P. Leitner, and H. C. Gall, “Towards quality gates in continuous delivery and deployment,” in *2016 IEEE 24th international conference on program comprehension (ICPC)*. IEEE, 2016, pp. 1–4.

[29] C. Vassallo, F. Palomba, A. Bacchelli, and H. C. Gall, “Continuous code quality: are we (really) doing that?” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 790–795.

[30] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[31] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.

[32] J. Tan, M. Lungu, and P. Avgeriou, “Towards studying the evolution of technical debt in the python projects from the apache software ecosystem.” in *BENEVOL*, 2018, pp. 43–45.

[33] T. Rausch, W. Hummer, P. Leitner, and S. Schulte, “An empirical analysis of build failures in the continuous integration workflows of java-based open-source software,” in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 345–355.

[34] I. Herraiz, G. Robles, J. M. González-Barahona, A. Capiluppi, and J. F. Ramil, “Comparison between slocs and number of files as size metrics for software evolution analysis,” in *Conference on Software Maintenance and Reengineering (CSMR’06)*. IEEE, 2006, pp. 8–pp.

[35] S. Beyer, C. Macho, M. Di Penta, and M. Pinzgen, “Analyzing the relationships between android api classes and their references on stack overflow,” AAU-SERG-2017-002, Tech. Rep., 2017.

[36] F. Jaafar, Y.-G. Guhneuc, S. Hamel, F. Khomh, and M. Zulkernine, “Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults,” *Empirical Softare Engineering*, vol. 21, pp. 896–931, 2016.

[37] Z. Xing and E. Stroulia, “Analyzing the evolutionary history of the logical design of object-oriented software,” *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 850–868, 2005.

[38] B. Dagenais and M. P. Robillard, “Recommending adaptive changes for framework evolution,” *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, Sep. 2011. [Online]. Available: https://doi.org/10.1145/2000799.2000805

[39] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and accurate source code differencing,” in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.

[40] L. Pace, *Beginning R: An introduction to statistical programming*. Apress, 2012.

[41] D. L. Parnas, “Software aging,” in *Proceedings of 16th International Conference on Software Engineering*. IEEE, 1994, pp. 279–287.

[42] A. E. Hassan, “Predicting faults using the complexity of code changes,” in *2009 IEEE 31st international conference on software engineering*. IEEE, 2009, pp. 78–88.

[43] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, “The scent of a smell: An extensive comparison between textual and structural smells,” *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 977–1000, 2017.

[44] A. Martini, T. Besker, and J. Bosch, “Technical debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations,” *Science of Computer Programming*, vol. 163, pp. 42–61, 2018.

[45] J. Yli-Huumo, A. Maglyas, and K. Smolander, “How do software development teams manage technical debt?–an empirical study,” *Journal of Systems and Software*, vol. 120, pp. 195–218, 2016.

George Digkas George Digkas is a double degree PhD student at the University of Groningen, the Netherlands and the University of Macedonia, Thessaloniki, Greece. He received a BSc and MSc in Applied Informatics from the University of Macedonia, Greece in 2014 and 2016, respectively. His research interests include technical debt, software quality and mining of software repositories.

Dr. Alexander Chatzigeorgiou is a Professor of Software Engineering in the Department of Applied Informatics and Dean of the School of Information Sciences at the University of Macedonia, Thessaloniki, Greece. He received the Diploma in Electrical Engineering and the PhD degree in Computer Science from the Aristotle University of Thessaloniki, Greece, in 1996 and 2000, respectively. From 1997 to 1999 he was with Intracom S.A., Greece, as a software designer. His research interests include object-oriented design, software maintenance, technical debt and evolution analysis. He has published more than 150 articles in international journals and conferences and participated in a number of European and national research programs. He is a Senior Associate Editor of the Journal of Systems and Software.

Dr. Apostolos Ampatzoglou is an Assistant Professor in the Department of Applied Informatics in University of Macedonia (Greece), where he carries out research and teaching in the area of software engineering. Before joining University of Macedonia he was an Assistant Professor in the University of Groningen (Netherlands). He holds a BSc on Information Systems (2003), an MSc on Computer Systems (2005) and a PhD in Software Engineering by the Aristotle University of Thessaloniki (2012). He has published more than 70 articles in international journals and conferences, and is/was involved in over 15 R&D ICT projects, with funding from national and international organizations. His current research interests are focused on technical debt management, software maintainability, reverse engineering software quality management, open source software, and software design.

Dr. Paris Avgeriou is Professor of Software Engineering in the Johann Bernoulli Institute for Mathematics and Computer Science, University of Groningen, the Netherlands where he has led the Software Engineering research group since September 2006. Before joining Groningen, he was a post-doctoral Fellow of the ERCIM. He has participated in a number of national and European research projects related to the European industry of Software-intensive systems. He has co-organized several international conferences and workshops (mainly at the International Conference on Software Engineering - ICSE). He sits on the editorial board of Springer Transactions on Pattern Languages of Programming (TPLOP). He has edited special issues in IEEE Software, Journal of Systems and Software and Springer TPLOP. He has published more than 130 peer-reviewed articles in international journals, conference proceedings and books. His research interests lie in the area of software architecture, with strong emphasis on architecture modeling, knowledge, evolution, patterns and link to requirements.