How does Technical Debt Evolve within Pull Requests? An Empirical Study with Apache Projects

Felipe E. de O. Calixto
Federal University of Campina Grande
Brazil
felipecalixto@copin.ufcg.edu.br

Eliane C. Araújo
Federal University of Campina Grande
Brazil
eliane@computacao.ufcg.edu.br

Everton L. G. Alves
Federal University of Campina Grande
Brazil
everton@computacao.ufcg.edu.br

ABSTRACT

Technical Debt (TD) refers to the cost to rectify the quality issues affecting a software system. A pull request (PR) represents a discrete unit of work that must adhere to particular quality standards to be integrated into the main codebase. They can serve as a means to gauge how developers address TD and how codebase quality may decay over time. In this work, we present an empirical study on 12 Apache Java repositories, analyzing 2,035 PRs to examine how TD evolves within them. Using the SonarQube tool, we evaluated (i) how TD changes within PRs and (ii) which types of TD issue are most frequently overlooked and resolved. Our results suggest that TD issues are prevalent in PRs, with a tendency to follow a ratio of 1:2:1 (reduced: unchanged: increased). Furthermore, across all issues, we found that the most frequently overlooked are related to code duplication and cognitive complexity, while the most resolved ones include code duplication and obsolete code. These insights can help practitioners become more aware and might inspire the creation of new tools that make managing TD during PRs easier.

CCS CONCEPTS

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

KEYWORDS

technical debt, pull request, software evolution, mining software repositories, empirical study

1 INTRODUCTION

Technical debt is a term coined by Cunningham [7], used to describe the implicit cost generated by technical problems resulting from pressure for fast deliveries, the development of low-quality solutions, and lack of developer knowledge, among other causes. These issues can affect both the source code and the development process itself [21].

Two fundamental concepts associated with the cost of technical debt are: (i) the *principal*, which refers to the cost required to resolve all technical issues and achieve a desirable level of maintainability, whether in a specific component or throughout the software; and (ii) the *interest*, representing the additional cost to implement changes due to existing technical debt [1, 3]. In this work, we focus on the evolution of the technical debt principal, hereafter referred to simply as TD. The amount of TD may affect the evolution and maintainability of a system, increasing the effort needed to make changes [36], and may affect other aspects, such as security and efficiency [12].

The literature categorizes TD issues into various types such as code (e.g., code complexity and duplication), design (e.g., cohesion and coupling problems), test (e.g., insufficient testing), and infrastructure (e.g., slow build times) [17, 24], where code and design-related are among the most commonly explored [6, 13, 15, 24, 28, 30, 39, 44].

In this context, code smells are known quality issues that may degrade code readability and maintainability, making it more susceptible to bugs [41]. Code smells can be addressed by refactoring [18] and are an indicator of the occurrence of TD in the codebase [14].

Automated Static Analysis Tools (ASATs) inspect the source code of a software for detecting quality [38]. Several studies have used ASATs with different goals: CheckStyle[1] [31, 45], FindBugs[2]/SpotBugs[3] [19, 25], PMD[4] [31, 37], and SonarQube[5] [9, 26, 29]. However, only SonarQube presents a summary metric for TD.

TD can be estimated in two main ways: through (i) structural characteristics, such as structural metrics, and (ii) cost estimation methods [8, 16]. SonarQube applies a cost estimation strategy in which heuristics are used to estimate (in minutes, hours, or days) the effort required to address the detected issues. SonarQube summarizes the TD estimation in a specific metric, *technical debt*[6]. Several studies have used this metric as a valid proxy to measure TD in projects [12, 27, 29].

In this work, we present an empirical study that explored 12 Apache Java repositories with the goal of understanding how TD evolves within PRs — whether it increases or decreases — and which types of issues are most neglected and resolved in this context. We address the following questions through this research:

- **RQ1: How does TD evolve within PRs?** We used the SonarQube tool to analyze if TD issues present before a PR opening are fixed within the PR commits.
- **RQ2: Which TD issues are most commonly resolved and neglected within PRs?** We identified the most resolved issues with a PR and those that are frequently neglected.

The contributions of this work are twofold:

- An empirical study that analyzed 2,035 merged PRs from 12 Apache Java projects. We established conclusions on the presence of TD issues, how they evolve, and which issues are most common to happen in the scope of a PR.

[1] https://checkstyle.sourceforge.io/
[2] https://findbugs.sourceforge.net/
[3] https://spotbugs.github.io/
[4] https://pmd.github.io/
[5] https://www.sonarsource.com/products/sonarqube/
[6] https://docs.sonarsource.com/sonarqube/latest/user-guide/metric-definitions/

- A reproduction kit with all artifacts of our study, including scripts and the analyzed database [10]. This kit can help other research in a similar field.

The rest of the paper is organized as follows. Section 2 provides background on some important concepts related to the SonarQube tool and PR-based development. Section 3 discusses the related work. Section 4 introduces the designed applied in our study. The results and discussions are presented in Section 5. Section 6 addresses potential threats to validity, while Section 7 provides some concluding remarks and possible future works.

2 BACKGROUND

2.1 SonarQube

SonarQube is an open source ASAT capable of detecting approximately 5,000 code quality issues in more than 30 languages, including Java. It is used by more than 400,000 organizations around the world [34] and has been utilized in numerous software engineering empirical studies [3, 22, 26, 32, 42]. In our study, SonarQube serves as the tool for measuring code and design TD issues [29] within PRs.

SonarQube employs a set of best-practice rules, known as *coding rules*[7], which trigger warnings upon violation (issues). An issue can be categorized as: *BUG*, indicative of a programming fault that can potentially lead to errors or unexpected behavior during execution; *CODE SMELL*, indicative of sub-optimal code quality, affecting maintainability; and *VULNERABILITY*, a security-related flaw. The metric used by SonarQube to compute TD remediation time, called *technical debt*, primarily focuses on *CODE SMELL* issues. Furthermore, issues are classified by severity level: *INFO*, *MINOR*, *MAJOR*, *CRITICAL*, or *BLOCKER*, in ascending order of severity.

Regarding issue lifecycle, upon initial detection, an issue is marked as *OPEN* (unfixed issue). In subsequent runs, if an issue previously marked as *OPEN* has been properly fixed, or no longer exists, then SonarQube infers it as fixed and labels it as *CLOSED*[8].

SonarQube employs a heuristic [35] to estimate the effort required to resolve each type of issue, using a scale ranging from *TRIVIAL*, which denotes actions such as removing unused imports that do not affect logic, to *COMPLEX*, which indicates actions that potentially require application redesign, such as eliminating cyclic dependencies between packages. For Java, SonarQube estimated efforts range from 5 minutes (*TRIVIAL* issues) to 1 day (*COMPLEX* ones).

2.2 The SQALE Method

SonarQube uses a quality model called SQALE, which considers that measuring quality is measuring TD [23]. In this model, quality represents compliance with requirements, and remediation costs are used to calculate quality indicators. Additionally, the SQALE method classifies requirements into characteristics and sub-characteristics related to code quality, allowing identification of which characteristics are most impacted by issues. Each requirement has an associated remediation function that calculates the necessary effort, measured in time, to meet the requirement in a given component of the system (module, file, or class).

```
public HashMap<String,String> tableConfig;
```

Make tableConfig a static final constant or non-public and provide accessors if needed.

Figure 1: Example of an violation to SonarQube's rule S1104.

Figure 1 presents an example of a SonarQube violation that refers to rule S1104, which requires the class attributes to not be public. To fix this issue, one can convert the attribute to a constant and make it `static final`, or turn the attribute's visibility to `private` (remediation details). In this case, the associated remediation function calculates 10 minutes per occurrence to fix this issue.

2.3 PR-Based Development

PR-based development has garnered widespread adoption across commercial organizations and open-source projects, enabling developers in distributed teams to autonomously implement changes. Figure 2 gives an overview of this model. Developers can fork a development branch and implement code changes via a series of commits (c_1 to c_2). Subsequently, they submit a PR to be evaluated through code review by other developers. If additional work is necessary, the developer performs changes and introduces new commits (c_3 to c_6), resulting in a potential code merge upon acceptance.

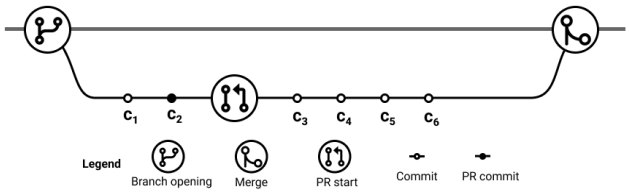

Figure 2: Example of a PR.

Our study focuses on investigating how TD evolves during PRs. Hence, our analysis starts with the commit that initiates the PR, referred as *PR commit*, typically the most recent commit before PR initiation. In Figure 2, the PR commit is c_2. In this paper, our analysis extends to all commits starting from the PR commit until the last commit of the PR branch (c_2 to c_6 in this example). We assume that in these commits, the developer may improve code quality either by their own decision or under the influence of code reviews, as these commits are subject to the code review process.

3 RELATED WORK

Li *et al.* [24] conducted a systematic literature review on TD management. They identified 10 TD types, 8 TD management activities, and 29 TD management tools. Among the selected papers, the authors found that code debt was the most studied TD type, which is aligned to the focus of our study. Avgeriou *et al.* [3] investigated and compared technical debt measurement tools. Among their findings, SonarQube was identified as the most widely used tool for measuring TD.

[7] https://rules.sonarsource.com/
[8] https://docs.sonarsource.com/sonarqube/10.0/user-guide/issues/

Previous studies have addressed how TD evolves in software projects, often using ASATs. Digkas *et al.* [12] analyzed the evolution of TD in 66 Apache Java projects, covering weekly revisions over a five-year period. In this study, SonarQube was the tool for TD measurement. They found that TD, along with other source code metrics, increased in most analyzed projects, while TD normalized by size (lines of code) decreased over time. In addition, they identified that the most common types of issue were related to improper exception handling and code duplication.

Molnar and Motogna [27] investigated how the quantity and composition of TD change throughout software evolution. They evaluated all revisions of three Java projects using SonarQube. Their findings revealed a high correlation between the number of code lines in the files and the TD, with 20% of the types of issues representing 80% of the total TD.

Tan *et al.* [36] studied the evolution of TD in Apache Pyhton projects. They evaluated weekly revisions of each project using SonarQube to identify types of issues and the amount of TD that is resolved. Among their findings, the authors identified that the majority of resolved TD is related to issues concerning testing, documentation, complexity, and code duplication. In addition, most of the TD is short-term, being resolved within two months.

Nikolaidis *et al.* [29] examined how different maintenance activities (e.g., bug fixing) impact TD growth over software evolution. They mined and analyzed 13.5K PRs from 10 open-source Java projects and used SonarQube for TD measurement. As a result, they identified that adding new features tends to increase TD, whereas refactoring tends to reduce it.

Those referenced studies [12, 27, 29, 36] employed SonarQube for TD measurement, the same tool we also employ in ours. In particular, only Nikolaidis et al. [29] centered their investigation on PRs as the primary subject, while the remaining studies opted for analyzing project revisions. Their study investigated how different types of maintenance activities impact TD. Our study, however, distinguishes itself by examining TD evolution within PRs and which types of issues are more frequently addressed and which are addressed less frequently.

Examining project revisions can provide valuable insights into TD's evolution resulting from longer development cycles, which likely involve multiple developers. Conversely, PR analysis, as conducted in our study, enables the identification of individual trends and concerns, as a PR typically represents a small piece of work carried out by an individual developer. Our objective is to ascertain whether developers prioritize TD resolution and to offer insights aimed at enhancing code quality, starting from the perspective of an individual developer.

4 STUDY DESIGN

The objective of this study is to investigate the evolution of TD within the context of a single PR. To accomplish this, we conducted an empirical investigation that involved 2,035 merged PRs from 12 Java projects within the Apache ecosystem. We opted to focus on Apache projects due to their prior examination in existing literature [5, 11, 20, 22, 36, 43] and because Apache offers a robust software development environment characterized by active and high-quality projects.

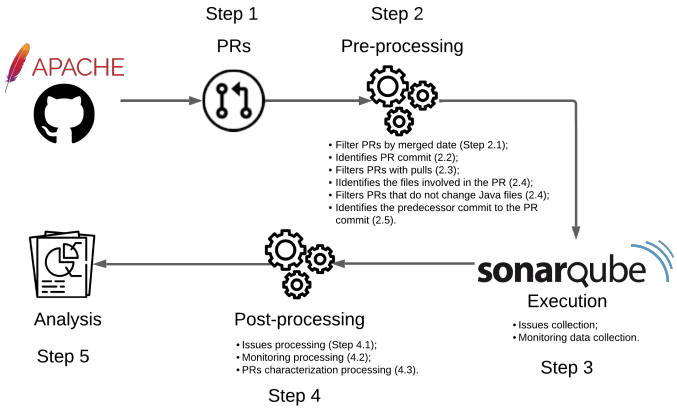

Figure 3: Data collection and processing.

We evaluated PR TD using the SonarQube ASAT, a tool commonly utilized for this purpose [9, 12, 27, 29, 43]. The selection of projects was carried out through convenience sampling, as SonarQube requires compiled code for analysis. Furthermore, we specifically targeted projects primarily written in Java that were not archived as of the execution date of the experiment (March/2024).

To guide our study, we formulated the following research questions:

RQ1: How does TD evolve within PRs? We are interested in assessing how TD varies within PRs and determining whether it decreases, increases, or remains constant. To achieve this, we introduce a TD Variation metric to measure and analyze the changes.

RQ2: Which TD issues are most commonly resolved and neglected within PRs? Considering SonarQube's extensive catalog of coding rules, we hypothesize that developers tend to address issues associated with certain rules more often while neglecting others. Our objective is to pinpoint these variations and gain insights into developers' decisions on issue remediation.

Figure 3 presents an overview of the methodology applied to collect and process data in our study. Initially, we gathered all merged PRs from each project (Step 1), then filtered and processed through various stages (Step 2). After that, for each PR, we ran SonarQube on the commits starting from the PR commit up to the last commit of the PR branch (Step 3) and then conducted post-processing (Step 4) in order to provide data preparation for analysis (Step 5). In the following sections, we detail steps 1-4. Table 1 presents a summary of our dataset of selected projects and PRs.

4.1 Dataset

As Apache projects are hosted on GitHub, we utilized the GitHub REST[9] and GraphQL[10] APIs to extract the PRs from each repository. This extraction was conducted between March 13, 2024, and March 14, 2024, adhering to the following filters:

- Only merged PRs. We selected only merged PRs as they meet a minimum level of code quality and usefulness, ensuring that we are examining code that has been accepted by project reviewers;

[9]https://docs.github.com/en/rest
[10]https://docs.github.com/en/graphql

- Merged date up to February 29, 2024, at 23:59:59. This choice was made to homogenize the data by a cut-off date.

We collected a total of 7,494 merged PRs as shown in Table 1.

4.2 Data Preparation

After collecting the projects and PRs, we conducted a five-step preprocessing on the PRs dataset by (i) filtering of PRs by merged date (Step 2.1), (ii) identifying the PR commits (Step 2.2), (iii) filtering out PRs with pulls (Step 2.3), (iv) identifying modified files and filtering out PRs that do not modify Java files (Step 2.4), and (v) identifying the commits preceding the PR commits (Step 2.5). The PR commit identification was based on the commits creation dates and PR opening date. A PR commit is characterized as the commit with the most recent date before the PR creation. Then, we filtered out the PRs that pulled code from other branches. In our study, we considered only PRs that did not pulled code or that only pulled code in the last commit. We adopted this strategy to prevent including code with issues that were not created within the PR, which could influence our results.

We identified the modified files of each PR branch, to narrow down the scope of the analysis. We considered the TD of each PR as the TD detected in the modified files. During this stage, we excluded the PRs that did not change Java files.

In the last stage, we identified the commit preceding the PR commit. As discussed in Section 4.4, we need to run SonarQube on a commit that occurred before the PR so we can refer to the issues identified in this commit as *pre-existing* issues. We identify the "preceding commit" using two strategies: (i) in cases where there is a commit before the PR commit within the branch, the preceding commit is the last commit before the PR commit; (ii) in cases where there is no commit before the PR commit within the branch, we use the concept of parent commit[11] and consider the parent of the PR commit as the preceding commit. In the example presented in Figure 2, c_1 is the preceding.

4.3 Data Processing

In our study, we used SonarQube (version 10.0.0.68432) combined with the SonarScanner[12] tool (version 4.8.0.2856). All SonarQube analyses were conducted using SonarScanner, which executes them and sends its results to SonarQube. SonarScanner can be natively integrated with build tools (such as SonarScanner for Maven) or used as a standalone command-line interface (CLI), allowing for independent execution regardless of building tool. Since most of the projects were not integrated to SonarQube, they lacked proper configurations to use the native SonarScanner plugin. Therefore, we used the SonarScanner CLI. To run the SonarScanner analysis (Step 3), it is necessary to have the compiled code of the project. Therefore, we downloaded the code project referred to each commit (the commits described in Section 2.3) of each PR and compile it in order to run the SonarQube. In this stage, 1,819 PRs were discarded due to compilation errors (e.g., missing dependencies, dependency version conflicts).

The time required for compiling the code and executing SonarQube analysis was significant due to the projects' size. The average time for SonarQube to execute on a given PR commit was 6 minutes and 18 seconds. Considering that each PR includes multiple commits, the execution time posed a significant challenge in our study. To address this issue, we divided the set of PRs into subsets and ran each subset on a separate virtual machine (VM). For this purpose, we utilized the Oracle VM VirtualBox[13] tool and configured four VMs as follows: 8 GB of RAM; 4 processor cores; 150 GB of storage; and Ubuntu 22.04.1 LTS operating system.

We also developed Python scripts to streamline the process. These scripts facilitate the downloading of commit source code, compilation, execution of SonarQube, and extraction of identified issues via the SonarQube API[14].

4.4 Data Analysis

After running SonarQube, we processed data related to issues (Step 4.1), execution monitoring (Step 4.2), and PRs (Step 4.3). During the issues processing, we classified the issues into two categories: *origin* and *status*. Origin indicates whether an issue existed before the PR (*pre-existing* issue) or it was added during the PR (*new* issue). Status, on the other hand, indicates whether the issue was resolved (*fixed* issue) or not (*unfixed* issue). Pre-existing issues are identified in the commit preceding the PR commit, while new or fixed issues are detected in the commits following the PR commit. Thus, by analyzing the set of commits described in Section 2.3, we can precisely detect in which commit an issue was added or fixed. This allows us, for example, to identify changes in TD measurement, as new issues may be addressed within the PR itself. This approach differs from studies that only compare the code before the PR and after it, as they do not track the code evolution commit by commit [29, 31, 45].

For each issue, we collected the following fields:

- **rule**. Identifier of the violated SonarQube rule;
- **severity**. The severity level, categorized on a scale: *INFO*, *MINOR*, *MAJOR*, *CRITICAL*, and *BLOCKER*, ordered from least to most severe;
- **type**. Classification of the issue: *CODE SMELL*, *BUG*, and *VULNERABILITY*;
- **debt**. Estimated effort in minutes to resolve the issue;
- **origin**. The origin of the issue (*PRE-EXISTING* or *NEW*);
- **status**. The status of the issue: *OPEN* if the issue is open, and *CLOSED* if the issue has been resolved;
- **file**. Affected file;
- **ncloc**: NCLOC (Non-Commented Lines of Code) of the affected file.

Related to monitoring (Step 4.2), we stored information about how long each commit of each PR took to execute. We monitored the SonarQube execution duration to estimate the time required for running the analyses. Other studies employing the same tool may consider this data for more effective experiment planning.

Additionally, we collected PR features to perform a characterization (Step 4.3) of them. The collected features were: added lines, removed lines, code churn (added lines + removed lines), time until

[11] https://git-scm.com/docs/git-commit-tree
[12] https://docs.sonarsource.com/sonarqube/10.0/analyzing-source-code/scanners/sonarscanner/
[13] https://www.virtualbox.org/
[14] https://docs.sonarsource.com/sonarqube/10.0/extension-guide/web-api/

Table 1: Dataset of projects and PRs.

Project	# Classes	NCLOC	# Issues	# PRs			
				Mining	After processing	Post-execution	With issues
accumulo	5,164	440,441	158,730	2,295	1,512	994	952
cayenne	4,716	318,428	1,712	429	336	55	55
commons-collections	839	67,690	2,280	241	68	31	28
commons-io	288	30,501	8,285	374	95	76	73
commons-lang	918	95,929	9,673	496	192	128	122
helix	2,106	189,487	35,978	1,374	669	278	276
httpcomponents-client	879	76,964	5,179	310	159	126	120
maven-surefire	3,036	110,481	1,833	331	71	25	25
opennlp	2,478	155,900	10,731	375	153	96	94
struts	3,419	234,331	9,877	716	407	145	140
wicket	5,254	251,505	1,000	441	157	47	40
zookeeper	1,542	131,712	2,289	112	35	34	34
Total			247,365	7,494	3,854	2,035	1,959

merge, number of modified files, and number of commits. This characterization encompassed all 2,035 PRs analyzed.

Thus, we obtained three datasets: (i) issues, (ii) monitoring and (iii) PRs. We conducted a characterization of the issues and PRs to understand the distributions of their features. The dataset of issues was used to address our research questions, the analysis of which is detailed in the following section.

4.5 Metrics

To help answer RQ1, we defined a normalized metric for TD. Normalization is needed due to the different sizes of the considered projects. For a PR_i that modifies k files, and each file contains n_j issues (where j is the file index), the TD density TDD_i is given by:

$$TDD_i = \frac{\sum_{j=1}^{k} \sum_{a=1}^{n_j} TD_{aj}}{\sum_{j=1}^{k} NCLOC_j} \tag{1}$$

where TD_{aj} represents the TD of issue a in file j, and $NCLOC_j$ is the number of non-comment lines of code [33] in file j.

In this sense, we proposed a metric to express the TD variation in PRs according to its evolution. TD variation can be categorized as: *reduced*, *unchanged*, or *increased*. For a PR_i that modifies k files, the TD variation TDV_i is given by:

$$TDV_i = \sum_{j=1}^{k} \left(\sum_{a=1}^{n_j} TD_{aj}^{New\ AND\ Unfixed} - \sum_{a=1}^{n_j} TD_{aj}^{Pre\text{-}existing\ AND\ Fixed} \right) \tag{2}$$

where $TD_{aj}^{New\ AND\ Unfixed}$ is the TD regarding new unfixed issue a in file j, and $TD_{aj}^{Pre\text{-}existing\ AND\ Fixed}$ is the TD regarding pre-existing fixed issue a of file j. The TD variation is calculated as the difference between the final version (new TD) and the initial version (pre-existing TD). We can simplify this to equation 2, as the difference occurs only in terms of the new TD that has not been addressed and the pre-existing TD that has been addressed. The TD variation can be classified as follows:

- *Reduced*, if $TDV_i < 0$, the TD decreased after the PR;
- *Unchanged*, if $TDV_i = 0$, the TD remained the same;
- *Increased*, if $TDV_i > 0$, the TD increased after the PR.

Additionally, we defined the *Pre-existing TDV* and the *New TDV*, which allow us to identify how TD variation occurs given its origin. For a PR_i that modifies k files, its *Pre-existing TDV* and the *New TDV* are calculated as follows:

$$TDV_i^{Pre\text{-}existing} = \sum_{j=1}^{k} \sum_{a=1}^{n_j} TD_{aj}^{Pre\text{-}existing\ AND\ Fixed} \tag{3}$$

$$TDV_i^{New} = \sum_{j=1}^{k} \sum_{a=1}^{n_j} TD_{aj}^{New\ AND\ Fixed} \tag{4}$$

where $TD_{aj}^{Pre\text{-}existing\ AND\ Fixed}$ represents the TD for pre-existing fixed issue a of file j, and $TD_{aj}^{New\ AND\ Fixed}$ represents the TD for new fixed issue a of file j. Both $TDV_i^{Pre\text{-}existing}$ and TDV_i^{New} can be classified as:

- *Reduced*, if the value is greater than zero, the TD decreased after the PR;
- *Unchanged*, if the value is equal to zero, the TD remained the same.

To address RQ2, we ranked the types of issues (rules) considering their position in each project's top 10 frequency. This strategy was necessary due to the difference of issue distribution in our dataset. For instance, the `accumulo` project refers to 64.12% of the found issues, which is expected given its size and number of PRs (48.60% of total), while other projects account for less than 36% of the issues. For this purpose, we propose a metric that aggregates the positions where the rules appear in each project's top 10. For a rule i, its $PositionScore_i$ is calculated as follows:

$$PositionScore_i = \sum_{j=1}^{k} Position_j \quad (5)$$

where $Position_j$ is the position of the rule in project j's top 10. If the rule appears out of the ranking bounds, its position value is 11 (for penalty purposes). The *PositionScore* is then calculated for all k projects (12 in this study). In this case, its score ranges from 12 (position 1 in all 12 projects) to 131 (position 10 in a single project and outside the top 10 in the rest), where lower scores indicate better ranking. We use this method to aggregate the top 10 fixed rules and the top 10 unfixed rules from all projects, and finally rank the 5 rules with the lowest *PositionScore*, i.e., the most frequent across all projects.

5 RESULTS AND DISCUSSIONS

5.1 Data Characterization

Before addressing the research questions, we conducted an exploratory data analysis to better understand the distribution of PR and its issues. Our aim was to examine the typical characteristics of a typical PR and issues considered in our study.

In this analysis, we identified that common PRs modify few files (with a median of 2 to 4) in a few commits (with a median of 1 to 2), although we found some outliers that modify hundreds of files. These results highlight how a typical PR encompasses a small development cycle (few commits) of a single developer focused on very specific parts of the system (few files).

Regarding issue characterization, we found that our dataset consists of 264 distinct rules (out of 622 detectable by SonarQube), of which 202 had at least one fixed instance (76.51%). Additionally, we identified that 94.38% of them are of the *CODE SMELL* type, and 48.43% of the issues have low severity (*INFO* and *MINOR*), although 73.04% of all detected TD are from issues with medium to high severity (*MAJOR*, *CRITICAL*, and *BLOCKER*). As expected, more than 96% of the issues are pre-existing, and just over 4% are new. Only 5.88% of the issues were fixed.

In summary, most of the issues are classified as *CODE SMELL*, meaning they are code quality problems that do not directly affect the system's functionality. Although the number of instances is well distributed among severity levels, the amount of TD is more concentrated in issues with high severity, which can be explained by the fact that more severe problems are more complex to address. The prevalence of pre-existing issues can be attributed to several factors. Even though PRs often modify few files, these files can be extensive and contain a reasonable number of issues, which may go unnoticed or be disregarded by developers, which also explains the large number of unfixed issues. Additionally, atypical PRs that modify up to hundreds of files contribute to these statistics, as it becomes impractical for a developer to address issues across several files.

5.2 RQ1: How does TD evolve within PRs?

To study how TD evolves in PRs, we use the metrics *TDD* (TD Density) and *TDV* (TD Variation; Section 4.5). Figures 4 and 5 show the distributions of both metrics per PR.

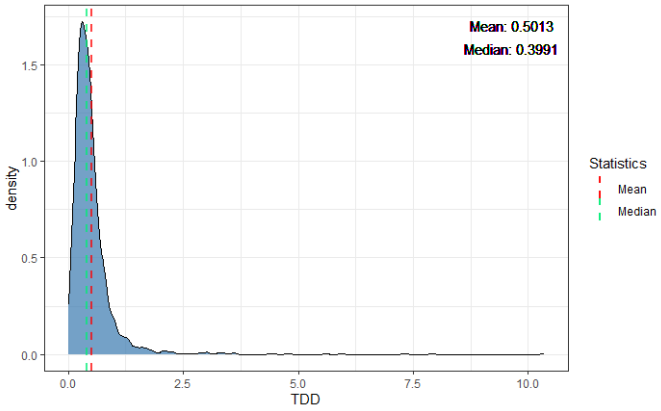

Figure 4: Distribution of TDD per PR.

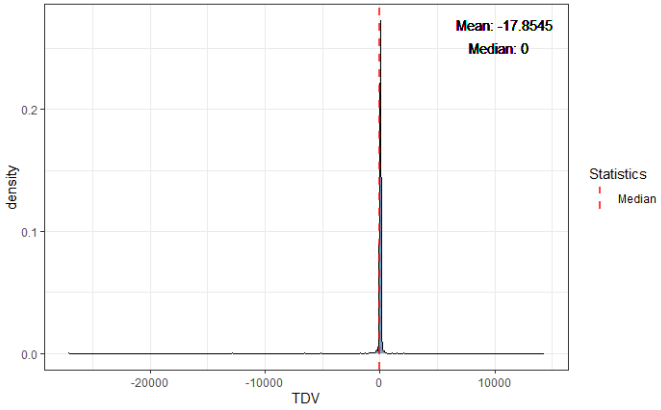

Figure 5: Distribution of TDV per PR.

TDD measures the amount of TD present per NCLOC in a PR. In our dataset, we found a mean of 0.5013 and median of 0.3991, respectively. These values indicate that the majority of PRs have a moderate to high TD density, considering the median of 651 NCLOC and the median density. This would represent a rework of over 4 hours to resolve all TD issues, a time that is somewhat considerable. Thus, most of the time, it would be unfeasible for the developer to address all the TD involved in a PR. It may be better to prioritize fixing the issues with higher severity.

PR number 676[15] from the `wicket` project presented the highest TDD (10.34 – 600 minutes of TD for 58 NCLOC). This PR has a code churn of 27 (12 lines added and 15 lines removed) and modifies 2 files, each containing one issue, both violating rule S110[16] (rule details: inheritance tree of classes should not be too deep) and with 300 minutes of TD each. Both issues are pre-existing and unfixed, meaning they were already in the code before the PR and were not resolved within the PR. Rule S110 imposes a threshold on the inheritance tree, which is by default set to 5, as deep inheritance can lead to high complexity. For each class with a depth greater

[15] https://github.com/apache/wicket/pull/676
[16] https://rules.sonarsource.com/java/RSPEC-110/

than 5, the rule is violated, resulting in a TD of 4 hours plus 30 minutes for each depth level above the threshold. In both classes, the depth is 7. As both are small classes, they concentrate a lot of TD in few NCLOC.

The TDV metric represents the variation of TD between the beginning and the end of the PR. It indicates how many minutes the TD increased or decreased during the PR, or if it remained unchanged (zero minutes). Figure 5 shows the distribution, which is concentrated around zero (median), indicating that the majority of values have no variation (zero) or low variation. However, there are very large outliers, primarily due to PRs that modify dozens to hundreds of files. Considering the outliers, PR 799[17] of `struts` presented the lowest TDV, at -27,133, which represents a reduction of 27,133 minutes of TD, equal to the entire TD involved in this PR. This PR modified 484 files and 90,610 lines (90,609 deletions and one addition). As its description indicates, this PR removed missed files and obsolete plugins. In other words, it only involves deletions, resulting in the removal of TD. In the top 10 cases with the lowest TD reduction, we identified 6 PRs where files were removed, indicating that at least part of the TD is not resolved as the primary intention.

On the other hand, the PR with the largest increase in TD was PR 2022[18] of `accumulo`, with an increase of 14,300 minutes. This PR modified 151 files and 1,919 lines (985 additions + 934 deletions). It has a total of 14,300 minutes of new TD, and 0 minutes of new TD fixed. For pre-existing TD, it has a total of 67,767 minutes, with zero minutes of pre-existing TD fixed. Among the new TD instances observed in this PR, the most common rules are S117[19] (385 instances; rule details: local variable and method parameter names should comply with a naming convention), S2589 (383 instances, boolean expressions should not be gratuitous)[20], S101[21] (276 instances; rule details: class names should comply with a naming convention), and S1125[22] (272 instances; rule details: boolean literals should not be redundant), representing a total of 1,316 instances out of 2,114 with a TD of 7,340 out of 14,300 minutes (51.33%). Rules S117 and S101 are related to naming conventions, while S1125 and S2589 are related to redundancy in boolean expressions.

The analysis of outliers reveals that cases where TD decreases sharply may not be directly linked to developers' primary intention to fix issues, whereas a significant increase in TD is more related to situations where the developer works on new functionalities.

To classify PRs in terms of TD variation (reduced, increased, or unchanged), we calculated the percentage of PRs with TDV less than zero (reduction), the percentage of PRs with TDV equal to zero (unchanged), and the percentage of PRs with TDV greater than zero (increased). Due to the variety of number of PRs and issues per project, we chose to calculate these percentages for each project and then calculate the overall average percentages.

Table 2 shows the TDV percentages for each repository, as well as the overall average. On average, 24.44% of the PRs decreased the TD, 52.24% remained unchanged, and 23.32% increased the TD. The values are close to the 1:2:1 ratio, meaning that for every four PRs, one decreases the TD, two remain unchanged, and one increases the TD. If we consider that 98.57% of the PRs have at least one pre-existing issue, then the PRs with unchanged TD neglect some amount of TD. Thus, considering both the PRs that keep the TD unchanged and the PRs that increase the TD, we found that 75.56% of the PRs neglect some amount of TD.

Table 2: Percentages of TDV by projects.

Project	TDV Percentages		
	Reduced	Unchanged	Increased
accumulo	26.16%	46.95%	26.89%
cayenne	12.73%	56.36%	30.91%
commons-collections	35.71%	53.57%	10.71%
commons-io	32.88%	50.68%	16.44%
commons-lang	13.93%	78.69%	7.38%
helix	19.20%	36.96%	43.84%
httpcomponents-client	19.17%	55.83%	25.00%
maven-surefire	24.00%	68.00%	8.00%
opennlp	44.68%	32.98%	22.34%
struts	37.14%	36.43%	26.43%
wicket	10.00%	57.50%	32.50%
zookeeper	17.65%	52.94%	29.41%
Mean	**24.44%**	**52.24%**	**23.32%**

Table 3: Percentages of pre-existing TDV by projects.

Project	Pre-existing TDV Percentages	
	Unchanged	Reduced
accumulo	57.64%	42.36%
cayenne	75.93%	24.07%
commons-collections	53.85%	46.15%
commons-io	62.50%	37.50%
commons-lang	81.15%	18.85%
helix	56.51%	43.49%
httpcomponents-client	69.49%	30.51%
maven-surefire	68.00%	32.00%
opennlp	43.96%	56.04%
struts	51.80%	48.20%
wicket	64.10%	35.90%
zookeeper	67.65%	32.35%
Mean	**62.71%**	**37.29%**

We can calculate *TDV* according to the origin of the issues. Tables 3 and 4 show the percentages for the pre-existing *TDV* and the new *TDV*, where a value less than zero indicates a reduction in the specific TD and equal to zero indicates unchanged TD. In the case of new *TDV*, for PRs containing new issues, reduction means solving at least one of the issues. On average, 37.29% of the PRs reduce some amount of pre-existing TD, while 62.71% ignore the pre-existing TD. On the other hand, for PRs with new issues, only 13.96% reduce some amount of new TD, meaning that 86.04% of the PRs in which the author adds new issues do not fix any of the issues

[17] https://github.com/apache/struts/pull/799
[18] https://github.com/apache/accumulo/pull/2022
[19] https://rules.sonarsource.com/java/RSPEC-117/
[20] https://rules.sonarsource.com/java/RSPEC-2589/
[21] https://rules.sonarsource.com/java/RSPEC-101/
[22] https://rules.sonarsource.com/java/RSPEC-1125/

Table 4: Percentages of new TDV by projects.

Project	New TDV Percentages	
	Unchanged	Reduced
accumulo	75.34%	24.66%
cayenne	90.91%	9.09%
commons-collections	66.67%	33.33%
commons-io	84.00%	16.00%
commons-lang	85.00%	15.00%
helix	75.86%	24.14%
httpcomponents-client	94.74%	5.26%
maven-surefire	100%	0.00%
opennlp	95.83%	4.17%
struts	91.43%	8.57%
wicket	94.12%	5.88%
zookeeper	78.57%	21.43%
Mean	**86.04%**	**13.96%**

before the PR is merged. In the specific case of the `maven-surefire` project, which showed 0% of PRs with reduction in new TD and 100% of PRs with unchanged TD, only 7 PRs in this project have new TD.

***RQ1**: Nearly all PRs involve some amount of TD (96.26%). We found that TD varies in a ratio close to 1:2:1, meaning that one in every four PRs reduces TD, two keep it unchanged, and one increases TD. Regarding the origins of TD, 37.29% of PRs with pre-existing TD reduce some amount of it, while only 13.96% of PRs with new TD reduce it, within the PR, to some extent.*

5.3 RQ2: Which TD issues are most commonly resolved and neglected within PRs?

To address RQ2, we aggregate the top 10 most frequently fixed rules per project and all top 10 most frequently unfixed rules per project using the *PositionScore* metric (Section 4.4). Table 5 highlights the five fixed rules and the five unfixed rules with the lowest PositionScore, that is, the most frequent across the projects. As we can see, all frequent rules are of the *CODE SMELL* type, and most have medium to high severity, thus developers more often encounter or add code quality issues whose remediation may be more laborious. Rule S1192 have the lowest PositionScore for both fixed and unfixed rules, appearing in the first position in 8 projects and up to the third position in the remaining 4 projects for unfixed rules. For fixed rules, it appeared in the first position in 2 projects and up to the third position in other 6 projects. This rule concerns the duplication of `String` literals. When the code is affected by this type of violation, maintenance often requires changing the same value in multiple parts of the code. The solution to this problem is to extract the literal value and turn it into a constant, which is then used in all instances of that value. The results for this rule show that depending on the situation, the developer may not see it as an issue or not as a high severity one. One possible situation that may lead the developer to neglect this problem is when it occurs in test classes. Antequil *et al.* [2] showed that approximately 42.9% of

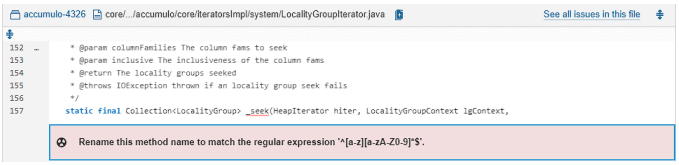

Figure 6: Example of a violation of the rule S100 in PR 4326 of `accumulo`.

regular methods use literals, while more than 82.2% of test methods presented literals. Considering all instances of this type, 68.92% occurred in test classes and 31.08% not.

Other rules related to TD issues that developers often overlook are S100 and S106. Rule S106 deals with the use of standard output for logging purposes. Out of its 6,413 occurrences, approximately 97.38% remained unfixed. By analyzing the PRs with the most instances, PR 1433[23] of `accumulo`, some of the issues occurred in a file named `Main.java`, which seems to be a CLI. PR 2180[24] of `accumulo` modifies code snippets associated with the mentioned issue and also appears to be a CLI. Popular and robust CLIs, such as Maven[25] and Gradle[26], use dedicated loggers to display information, which is seen as a good practice. However, in the mentioned project, such practice was not adhered to.

Rule S100 deals with naming convention in methods. This rule expects method names to satisfy the regular expression described in Figure 6, meaning the first character should be a lowercase letter, while the rest can be uppercase or lowercase letters and/or numbers. Figure 6 shows an example of a found issue associated to this rule, PR 4326[27] of `accumulo`, where a method name begins with an underscore symbol "_" instead of a lowercase letter. In several Apache projects, we noticed method names containing the underscore character "_", so we assume that some developers do not consider it as a problem. Since rule S100 also appears among the most fixed, we identified some PRs with most fixed issues of this type. For example, PR 202 of `httpcomponents-client` fixes 16 instances of this issue, all due to method names starting with uppercase letters. The message of one of the commits states `"Use camelCase for Java method names - always"`, indicating that method names should follow camelcase convention.

The cognitive complexity (S3776) appeared among both unfixed and fixed rules, although upon analyzing some PRs, it seems that in several situations, the complexity is mostly resolved when methods or classes are removed. The PR 657[28] of `struts` is an example where a developer explicitly addressed this issue, as evidenced by the comment `"[...] good to see the cognitive complexity reduced afterwards"` and the PR description `"Improve readability of XmlConfigurationProvider class"`. Among the analyzed projects, only `struts` officially uses SonarQube, and this was probably a case where the developer paid attention to the tool's reports to identify and fix the problem. Still, this is an uncommon situation and we

[23] https://github.com/apache/accumulo/pull/1433
[24] https://github.com/apache/accumulo/pull/2180
[25] https://github.com/apache/maven
[26] https://github.com/gradle/gradle
[27] https://github.com/apache/accumulo/pull/4326
[28] https://github.com/apache/struts/pull/657/

Table 5: Top 5 fixed rules and top 5 unfixed rules by PositionScore metric.

	PositionScore	Rule	Description	Severity	Type
Unfixed rules	22	S1192	String literals should not be duplicated	CRITICAL	CODE SMELL
	85	S3776	Cognitive Complexity of methods should not be too high	CRITICAL	CODE SMELL
	100	S100	Method names should comply with a naming convention	MINOR	CODE SMELL
	100	S106	Standard outputs should not be used directly to log anything	MAJOR	CODE SMELL
	101	S117	Local variable and method parameter names should comply with a naming convention	MINOR	CODE SMELL
	PositionScore	**Rule**	**Description**		
Fixed rules	41	S1192	String literals should not be duplicated	CRITICAL	CODE SMELL
	84	S1874	"@Deprecated" code should not be used	MINOR	CODE SMELL
	90	S112	Generic exceptions should never be thrown	MAJOR	CODE SMELL
	94	S2293	The diamond operator ("<>") should be used	MINOR	CODE SMELL
	102	S3776	Cognitive Complexity of methods should not be too high	CRITICAL	CODE SMELL

Noncompliant Code Example

```
List<String> strings = new ArrayList<String>();  // Noncompliant
Map<String,List<Integer>> map = new HashMap<String,List<Integer>>();  // Noncompliant
```

Compliant Solution

```
List<String> strings = new ArrayList<>();
Map<String,List<Integer>> map = new HashMap<>();
```

Figure 7: Example of a compliant and non-compliant code for rule S2293.

believe this rule to be a point where developers should take more attention, as methods with high cognitive complexity are more difficult to test and to maintain.

Among the rules that only appeared as fixed, they were related to obsolete code (S1874), improper exception handling (S112), and redundancy (S2293). Rule S1874 tracks the use of deprecated code annotated with the `@Deprecated` Java annotation; in this case, it is expected that the removal of the deprecated code is planned for sometime.

Rule S2293 states that instead of declaring the object type both in the declaration and in the constructor, one should simplify the constructor by using only the diamond operator without a type <>, and the compiler will infer the type. Figure 7 shows an example of code snippets conforming to the rule and others that do not. The two PRs with the highest number of fixed issues for this rule are PR 2643[29] (107 instances) and 3604[30] (55 instances), both from `accumulo`. In both cases, these rules are fixed by code removals and not by directly addressed. The S112 rule states that generic Java exceptions should never be thrown. This rule recommends catching only the exceptions one intends to handle, and when working with generic exception types, the only way to distinguish between multiple exceptions is by checking their message, which is error-prone and hard to maintain. The PR that stood out with the most issues fixed for this rule was PR 3402[31] of `accumulo`, in which 260 instances were addressed. According to the PR description, its focus was on improving exception handling, as the author clarifies "`[...] try to catch more specific checked exceptions, when appropriate, instead of catching (Exception) [...]`". This case, where the developer identifies and handles a specific problem, does not seem to be common.

***RQ2**: The most common TD issues are related to naming convention, duplication of literals, cognitive complexity, obsolete code, improper exception handling, and redundancy. Naming convention and duplication of literals issues appear to be tolerable to developers. However, they demonstrate concern in removing obsolete code, duplication of literals, and improper exception handling. We found indications that cognitive complexity is not well addressed by developers.*

5.4 Implications

Our findings have implications for practitioners, researchers and tools. For practitioners, our results indicate that the most common TD issues are related to code quality (*CODE SMELL*), often requiring longer remediation times (medium to high severity). Additionally, we observed that TD is neglected in a large portion of PRs (approximately 75%). Projects can consider implementing mechanisms to block PR merges that exceed a certain threshold of new TD. Furthermore, projects opting to use tools with customizable configurations, such as SonarQube, may consider configuring which rules are appropriate for their context and disable those that are not relevant. Lastly, developers are encouraged to be more attentive to frequently overlooked issues, such as cognitive complexity, as complex code tends to be more challenging to test and maintain.

For researchers, further studies could delve into the tolerance of issues, seeking developers' perspectives on this matter. This could contribute the proper configuration of analysis tools and/or the development of tools that could consider the context where the issues occurred and/or adjust severity levels. Moreover, we found that some TD issues are often addressed in code removal situations,

[29]https://github.com/apache/accumulo/pull/2643
[30]https://github.com/apache/accumulo/pull/3604
[31]https://github.com/apache/accumulo/pull/3402

indicating developers did not have proper intention to fix them. Therefore, it is important to better explore such aspect. Surveys and/or code review and commit messages could be used as in this sense.

6 THREATS TO VALIDITY

We used SonarQube to quantify TD, however it can only detect code TD and design TD, it can not detect other types such as requirements debt, social debt, build debt, etc [29]. SonarQube can also present false positives. The first author manually reviewed several samples of the collected data and observed that the occurrence of false positives was minimal, thereby not significantly affecting the drawn conclusions. Additionally, the need to compile the code posed a challenge to include projects in this study, as a significant number of PRs were excluded from the analysis due to compilation errors. Running the analyses is time-consuming, with an average execution time of 6m18s per commit. Therefore, we parallelized the executions on four VMs, allowing us to analyze a reasonable sample of 2,035 PRs in about two weeks. Despite these challenges, SonarQube is the most widely used tool in empirical studies related to TD [3, 9, 12, 27, 29, 43], and, as far as we know, the only tool that estimates the remediation time for TD.

Other quality methods could be used to measure TD. While the SQALE method estimates TD as remediation time, models like QMOOD [4] and Quamoco [40] aggregate structural metrics into quality aspects. Moreover, Curtis *et al.* [8] proposed a method for measuring TD based on three variables: the number of issues to be addressed, the amount of time to resolve each issue, and the cost of labor ($ per hour). From the mentioned methods, only Quamoco has an available plugin[32], although it has not been updated in the last 7 years, while QMOOD does not have any dedicated tools. A recent study [46] used other tools such as SonarQube to estimate the metrics described by QMOOD.

Our filtering strategy reduced almost half of the PRs in our initial dataset. However, such filtering was necessary to prevent situations such as dealing with code pulled from other branches that could affect the results. Additionally, we chose to study only Java projects from the Apache Software Foundation. The results may differ for other languages or industrial projects.

The scripts developed for processing and analyzing the PRs were manually validated by the authors and through several tests. For the replication of this study, we detailed all the main steps in the methodology and made our replication kit available [10]. The kit includes the mined and processed PRs at each step, as well as the analysis scripts and the analyzed data.

7 CONCLUSIONS

In this paper, we analyzed how TD evolves within PRs and which types of issues are most resolved and most neglected by developers. For that, we analyzed 2,035 merged PRs from 12 Apache Java projects using the SonarQube tool.

The results indicate that 96.26% of PRs involve some amount of TD. Across the projects, the evolution of TD tends to follow a 1:2:1 ratio (reduction:unchanged:increment), meaning that for every four PRs, one reduces TD, two remain unchanged, and one increases TD. Considering that maintaining TD unchanged can be seen as a form of neglect, approximately 76.63% of PRs neglect TD to some extent. Moreover, 37.29% of PRs with pre-existing TD address some of it, and only 13.96% of PRs introducing new TD address any portion of it.

We identified that the most common types of issues across the studied projects are related to naming conventions, literal duplication, cognitive complexity, obsolete code, improper exception handling, logging bad practices and redundancy. Among these, literal duplication was the most prevalent across all projects. We found indications that some types of issues may be tolerable, such as literal duplication and naming conventions. However, cognitive complexity appears to be a problem that deserves more attention from developers due to its impact on code understanding and testability.

For future work, we plan to study if TD is part of the PR code review. More specifically, whether reviewers are concerned about TD when reviewing a PR to be merged. For that, we plan to relate review comments to code edits that fixed TD issues. Moreover, we plan to conduct a survey with developers and reviewers to understand their tolerance to certain types of TD issues.

[32]https://github.com/MSUSEL/msusel-quamoco-plugin

AVAILABILITY OF ARTIFACTS

All data and scripts used in this study are available for future investigations [10].

ACKNOWLEDGMENTS

The first author was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

REFERENCES

[1] Areti Ampatzoglou, Nikolaos Mittas, Angeliki-Agathi Tsintzira, Apostolos Ampatzoglou, Elvira-Maria Arvanitou, Alexander Chatzigeorgiou, Paris Avgeriou, and Lefteris Angelis. 2020. Exploring the Relation between Technical Debt Principal and Interest: An Empirical Approach. *Information and Software Technology* 128 (2020), 106391. https://doi.org/10.1016/j.infsof.2020.106391

[2] Nicolas Anquetil, Julien Delplanque, Stéphane Ducasse, Oleksandr Zaitsev, Christopher Fuhrman, and Yann-Gael Guéhéneuc. 2022. What do developers consider magic literals? A smalltalk perspective. *Information and Software Technology* 149 (Sept. 2022). https://doi.org/10.1016/j.infsof.2022.106942

[3] Paris C. Avgeriou, Davide Taibi, Apostolos Ampatzoglou, Francesca Arcelli Fontana, Terese Besker, Alexander Chatzigeorgiou, Valentina Lenarduzzi, Antonio Martini, Athanasia Moschou, Ilaria Pigazzini, Nyyti Saarimaki, Darius Daniel Sas, Saulo Soares de Toledo, and Angeliki Agathi Tsintzira. 2021. An Overview and Comparison of Technical Debt Measurement Tools. *IEEE Software* 38, 3 (2021), 61–71. https://doi.org/10.1109/MS.2020.3024958

[4] J. Bansiya and C.G. Davis. 2002. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering* 28, 1 (2002), 4–17. https://doi.org/10.1109/32.979986

[5] Flávia Coelho, Nikolaos Tsantalis, Tiago Massoni, and Everton LG Alves. 2021. An empirical study on refactoring-inducing pull requests. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–12.

[6] Thierry Coq and Jean-Pierre Rosen. 2011. The SQALE Quality and Analysis Models for Assessing the Quality of Ada Source Code. In *Reliable Software Technologies - Ada-Europe 2011*, Alexander Romanovsky and Tullio Vardanega (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 61–74.

[7] Ward Cunningham. 1992. The WyCash Portfolio Management System. In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum)* (Vancouver, British Columbia, Canada) *(OOPSLA '92)*. Association for Computing Machinery, New York, NY, USA, 29–30. https://doi.org/10.1145/157709.157715

[8] Bill Curtis, Jay Sappidi, and Alexandra Szynkarski. 2012. Estimating the Principal of an Application's Technical Debt. *IEEE Software* 29, 6 (2012), 34–42. https:

//doi.org/10.1109/MS.2012.156

[9] Carlos Eduardo C. Dantas, Adriano M. Rocha, and Marcelo A. Maia. 2023. How do Developers Improve Code Readability? An Empirical Study of Pull Requests. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 110–122. https://doi.org/10.1109/ICSME58846.2023.00022

[10] Felipe E. de O. Calixto, Eliane C. Araújo, and Everton L. G. Alves. 2024. [Replication Kit] How does Technical Debt Evolve within Pull Requests? An Empirical Study with Apache Projects. https://doi.org/10.5281/zenodo.12761591

[11] G. Digkas, A. Chatzigeorgiou, A. Ampatzoglou, and P. Avgeriou. 2022. Can Clean New Code Reduce Technical Debt Density? *IEEE Transactions on Software Engineering* 48, 05 (may 2022), 1705–1721. https://doi.org/10.1109/TSE.2020.3032557

[12] Georgios Digkas, Mircea Lungu, Alexander Chatzigeorgiou, and Paris Avgeriou. 2017. The Evolution of Technical Debt in the Apache Ecosystem. In *Software Architecture*, Antónia Lopes and Rogério de Lemos (Eds.). Springer International Publishing, Cham, 51–66.

[13] Robert J. Eisenberg. 2012. A threshold based approach to technical debt. 37, 2 (apr 2012), 1–6. https://doi.org/10.1145/2108144.2108151

[14] Giammaria Giordano, Giusy Annunziata, Andrea De Lucia, and Fabio Palomba. 2023. Understanding Developer Practices and Code Smells Diffusion in AI-Enabled Software: A Preliminary Study. In *Joint Proceedings of the 32nd International Workshop on Software Measurement (IWSM) and the 17th International Conference on Software Process and Product Measurement (MENSURA), Rome, Italy, September 14-15, 2023 (CEUR Workshop Proceedings, Vol. 3543)*, Gabriele De Vito, Filomena Ferrucci, and Carmine Gravino (Eds.). CEUR-WS.org. https://ceur-ws.org/Vol-3543/paper18.pdf

[15] Isaac Griffith and Clemente Izurieta. 2014. Design pattern decay: the case for class grime *(ESEM '14)*. Association for Computing Machinery, New York, NY, USA, Article 39, 4 pages. https://doi.org/10.1145/2652524.2652570

[16] Makrina Viola Kosti, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Georgios Pallas, Ioannis Stamelos, and Lefteris Angelis. 2017. Technical Debt Principal Assessment Through Structural Metrics. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 329–333. https://doi.org/10.1109/SEAA.2017.59

[17] Philippe Kruchten, Robert Nord, and Ipek Ozkaya. 2019. *Managing Technical Debt: Reducing Friction in Software Development* (1st ed.). Addison-Wesley Professional.

[18] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software* 167 (2020), 110610. https://doi.org/10.1016/j.jss.2020.110610

[19] Luigi Lavazza, Davide Tosi, and Sandro Morasca. 2020. An Empirical Study on the Persistence of SpotBugs Issues in Open-Source Software Evolution. In *Quality of Information and Communications Technology*, Martin Shepperd, Fernando Brito e Abreu, Alberto Rodrigues da Silva, and Ricardo Pérez-Castillo (Eds.). Springer International Publishing, Cham, 144–151.

[20] Valentina Lenarduzzi, Vili Nikkola, Nyyti Saarimäki, and Davide Taibi. 2021. Does code quality affect pull request acceptance? An empirical study. *Journal of Systems and Software* 171 (2021), 110806. https://doi.org/10.1016/j.jss.2020.110806

[21] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. 2019. The Technical Debt Dataset. *CoRR* abs/1908.00827 (2019). arXiv:1908.00827 http://arxiv.org/abs/1908.00827

[22] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. 2020. Some SonarQube issues have a significant but small effect on faults and changes. A large-scale empirical study. *Journal of Systems and Software* 170 (2020), 110750. https://doi.org/10.1016/j.jss.2020.110750

[23] Jean-Louis Letouzey. 2012. The SQALE method for evaluating Technical Debt. In *2012 Third International Workshop on Managing Technical Debt (MTD)*. 31–36. https://doi.org/10.1109/MTD.2012.6225997

[24] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A systematic mapping study on technical debt and its management. *Journal of Systems and Software* 101 (2015), 193–220. https://doi.org/10.1016/j.jss.2014.12.027

[25] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Shin Yoo, and Yves Le Traon. 2021. Mining Fix Patterns for FindBugs Violations. *IEEE Transactions on Software Engineering* 47, 1 (2021), 165–188. https://doi.org/10.1109/TSE.2018.2884955

[26] Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto. 2019. Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 209–219. https://doi.org/10.1109/ICPC.2019.00040

[27] Arthur-Jozsef Molnar and Simona Motogna. 2020. Long-Term Evaluation of Technical Debt in Open-Source Software *(ESEM '20)*. Association for Computing Machinery, New York, NY, USA, Article 13, 9 pages. https://doi.org/10.1145/3382494.3410673

[28] J. Yates Monteith and John D. McGregor. 2013. Exploring software supply chains from a technical debt perspective. In *2013 4th International Workshop on Managing Technical Debt (MTD)*. 32–38. https://doi.org/10.1109/MTD.2013.6608676

[29] Nikolaos Nikolaidis, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Nikolaos Mittas, Evdokimos Konstantinidis, and Panagiotis Bamidis. 2023. Exploring the Effect of Various Maintenance Activities on the Accumulation of TD Principal. In *2023 ACM/IEEE International Conference on Technical Debt (TechDebt)*. 102–111. https://doi.org/10.1109/TechDebt59074.2023.00018

[30] Ariadi Nugroho, Joost Visser, and Tobias Kuipers. 2011. An empirical model of technical debt and interest *(MTD '11)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/1985362.1985364

[31] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2015. Would static analysis tools help developers with code reviews?. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 161–170. https://doi.org/10.1109/SANER.2015.7081826

[32] Nyyti Saarimaki, Maria Teresa Baldassarre, Valentina Lenarduzzi, and Simone Romano. 2019. On the Accuracy of SonarQube Technical Debt Remediation Time. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 317–324. https://doi.org/10.1109/SEAA.2019.00055

[33] SonarQube. [n. d.]. Metric definitions. https://docs.sonarsource.com/sonarqube/10.0/user-guide/metric-definitions/. Accessed: 2024-03-30.

[34] SonarSouce. 2022. SonarSource Posts Record Growth with its Clean Code Solution. https://www.sonarsource.com/company/press-releases/sonar-record-growth-2022/. Accessed: 2023-05-04.

[35] SonarSource. [n. d.]. Adding Code Rules. https://docs.sonarsource.com/sonarqube/10.0/extension-guide/adding-coding-rules/. Accessed: 2024-03-31.

[36] Jie Tan, Daniel Feitosa, Paris Avgeriou, and Mircea Lungu. 2021. Evolution of technical debt remediation in Python: A case study on the Apache Software Ecosystem. *Journal of Software: Evolution and Process* 33, 4 (2021), e2319. https://doi.org/10.1002/smr.2319 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2319 e2319 smr.2319.

[37] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. 2023. Are automated static analysis tools worth it? An investigation into relative warning density and external software quality on the example of Apache open source projects. *Empirical Software Engineering* 28, 3 (17 Apr 2023), 66. https://doi.org/10.1007/s10664-023-10301-2

[38] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C. Gall. 2018. Context is king: The developer perspective on the usage of static analysis tools. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 38–49. https://doi.org/10.1109/SANER.2018.8330195

[39] Antonio Vetrò. 2012. Using automatic static analysis to identify technical debt. In *2012 34th International Conference on Software Engineering (ICSE)*. 1613–1615. https://doi.org/10.1109/ICSE.2012.6227226

[40] Stefan Wagner, Klaus Lochmann, Lars Heinemann, Michael Kläs, Adam Trendowicz, Reinhold Plösch, Andreas Seidl, Andreas Goeb, and Jonathan Streit. 2012. The quamoco product quality modelling and assessment approach. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 1133–1142.

[41] Aiko Yamashita and Steve Counsell. 2013. Code smells as system-level indicators of maintainability: An empirical study. *Journal of Systems and Software* 86, 10 (2013), 2639–2653. https://doi.org/10.1016/j.jss.2013.05.007

[42] Ping Yu, Yijian Wu, Jiahan Peng, Jian Zhang, and Peicheng Xie. 2023. Towards Understanding Fixes of SonarQube Static Analysis Violations: A Large-Scale Empirical Study. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 569–580. https://doi.org/10.1109/SANER56733.2023.00059

[43] Ehsan Zabardast, Kwabena Ebo Bennin, and Javier Gonzalez-Huerta. 2022. Further investigation of the survivability of code technical debt items. *J. Softw. Evol. Process* 34, 2 (feb 2022), 16 pages. https://doi.org/10.1002/smr.2425

[44] Nico Zazworka, Antonio Vetro', Clemente Izurieta, Sunny Wong, Yuanfang Cai, Carolyn Seaman, and Forrest Shull. 2014. Comparing four approaches for technical debt identification. *Software Quality Journal* 22, 3 (01 Sep 2014), 403–426. https://doi.org/10.1007/s11219-013-9200-8

[45] Weiqin Zou, Jifeng Xuan, Xiaoyuan Xie, Zhenyu Chen, and Baowen Xu. 2019. How does code style inconsistency affect pull request integration? An exploratory study on 117 GitHub projects. *Empirical Software Engineering* 24, 6 (01 Dec 2019), 3871–3903. https://doi.org/10.1007/s10664-019-09720-x

[46] Yusuf Özçevik. 2024. Data-oriented QMOOD model for quality assessment of multi-client software applications. *Engineering Science and Technology, an International Journal* 51 (2024), 101660. https://doi.org/10.1016/j.jestch.2024.101660

Received XX Month XXXX; revised XX Month XXXX; accepted X Month XXXX