Exploring the Effect of Various Maintenance Activities on the Accumulation of TD Principal

Abstract—**One of the most well-known laws of software evolution suggests that code quality deteriorates over time. Following this law, recent empirical studies have brought evidence that Technical Debt (TD) Principal tends to increase (in absolute value) as the system grows, since more technical debt issues are added than resolved over time. To shed light into how technical debt accumulation occurs in practice, in this paper we explore specific maintenance activities (i.e., feature addition, bug fixing, and refactoring) and explore the balance between the technical debt that they introduce or resolve. To achieve this goal, we rely on studying Pull Requests (PR), which are the most established way to contribute code to an open-source project. A Pull Request is usually comprised by more than one commits, corresponding to a specific development / maintenance activity. In our study, we categorized Pull Requests, based on their labels, to find the effect that the different maintenance activities have on the accumulation of technical debt across evolution. In particular, we have analysed more than 13.5K pull requests (mined from 10 OSS projects), by calculating the TD Principal (calculated through SonarQube) before and after the Pull Requests. The results of the study suggested that several labels are used for tagging Pull Requests, out of which the most prevalent ones are new features, bug fixing, and refactoring. The effect of these activities on TD Principal accumulation is statistically different, and: (a) the addition of features tends to increase TD Principal; (b) refactoring is having an almost consistent positive effect (reducing TD Principal); and (c) bug fixing activity has undecisive impact on TD Principal. These results are compared to existing studies, interpreted, and various useful implications for researchers and practitioners have been drawn.**

Keywords—GitHub, Pull Request, Mining Repositories

I. Introduction

Technical Debt (TD) was introduced, in the early 90s, to capture the effort (and in turn the cost) that the developers internally "*borrow*" (from the company time-budget) when they adopt "*quick and dirty*" solutions, downgrading the final product in terms of quality, and in particular maintainability [1]. A key technical debt parameter that needs to be considered when managing technical debt is the notion of Principal, which is the number of code inefficiencies that have to be resolved in order for a given project to be in an optimum state in terms of maintainability [2]. Ensuring that a project retains an acceptable quality level—e.g., through imposing the use of quality gates for controlling the amount of introduced TD Principal [3]—can lead to smoother and more effective development, reducing the vicious cycles of technical debt accumulation [4]. Nevertheless, in practice, and due to tight development schedule, the quality of software tends to deteriorate along evolution [3]; whereas the same applies for TD Principal, in the sense that TD Principal seems to be increasing in subsequent versions, even for systems or ecosystems that are well-known for their levels of quality and software development processes (e.g., the Apache ecosystem—despite a decrease in some cases in the TD Principal density metric) [5].

Given the indisputable relation between system growth and quality erosion, it is important to gain an understanding of what kind of maintenance activities are more prone to accumulate TD Principal. According to van Vliet [6] maintenance activities can be mapped to four main categories: (a) activities that aim at ***fixing bugs***; (b) activities that aim at ***adding new features***; (c) activities that aim at improving the quality of code through ***refactoring***; and (d) activities that aim at identifying faults before the end-users. While developing software in modern collaborative software development environments (most commonly Git) these types of actions are usually performed in groups of commits, organized in the form of Pull Requests (PR) [7]. When a developer wants to submit an incremental contribution to a project, he / she is encouraged to create a Pull Request, grouping all the required changes and calling for a review of his/her contribution by one of the core project members. Each Pull Request may consist of one or more commits, whereas as a whole (by definition) is expected to provide a homogenous and completed contribution (e.g., a new functionality, a bug fix, or a code improvement action). The goal of the Pull Request mechanism is to enable the holistic review of these changes, before their merging to the main code branch. The project maintainers can accept the Pull Request to be merged into the project; otherwise, the maintainer can close the Pull Request or ask for changes. Because of the safety and collaboration of the overall functionality that the Pull Requests offers, they are widely used in large projects [8]. On top of this, it is very common for large projects to disable the incorporation of direct changes to the main branch imposing the use of Pull Requests, through the branch protection functionality [9]. According to previous studies [10], software projects can be changed through three types of development activities, which are: (a) addition of new features, (b) bug fixing, and (c) maintenance—which perfectly map the categories proposed by van Vliet.

In this study, we investigate the different effect that such maintenance actives can have on the accumulation of TD Principal. On the one hand, to ***record the type of maintenance activities***, we have used the labels that the developers have provided to characterize the Pull Request. In this way, we have assured the mapping of changes and maintenance categories, since we do not rely on subjective or AI-based approaches

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

(that might lack in terms of accuracy)—but on the expert belief of the developer. Furthermore, by studying Pull Requests instead of commits, we were able to (as much as possible) control the phenomenon of mixing maintenance activities. In other words, we were able to capture the changes that a developer has performed having in mind a specific activity, avoiding individual commits that mix maintenance activities. For example, concerning refactorings, a reverse engineering approach with refactoring mining tools would have led to substantially more commits characterized as refactorings, but for such commits we cannot be sure that they were not blending refactoring with adding new functionalities or bug fixing [11]. On the other hand, to ***record the effect of the change on TD Principal accumulation***, we analysed the project before and after the merge of a Pull Request through SonarQube. Despite the divergence of tools in calculating the value of TD Principal [12], due to the nature of the study: (a) being in need of a measurement and not an identification approach; and (b) the use of Java, we were not able to rely on the tools provided by Tsoukalas et al. [13] or Zozas et al. [14] that use the TD benchmark. Therefore, we opt to rely on SonarQube, which according to Avgeriou et al. [12] is the most popular tool for measuring TD Principal.

The rest of the paper is organized in the following way: Section II present related work; in Section III we present the design of our study. The results of our study along with their discussion are reported in Sections IV and V, respectively. Section VI reports the threats to the validity of our study; finally, we conclude the paper in Section VII.

II. Related Work

In this section, we present existing studies and background information for this research effort. First, we present studies that are related to pull requests and the quality of the code; second, we present studies that use the labelling system of pull requests; and finally, studies about the effect of specific development activities to the quality of the code or technical debt evolution.

A. Pull Requests and Code Quality

In the literature, it is common to explore pull requests to connect their contribution or their acceptance with the quality of the submitted code. Lenarduzzi et al. [15] analysed over 36 thousand pull requests of 28 Java projects focusing on whether the quality of the code that is being introduced is related to the acceptance probability of the pull request. In that study, the PMD tool was used in order to find code quality defects, and it was evident that quality did not play a key role in the acceptance or rejection of the pull request. However, it seems that certain PMD rules are indeed considered by the reviewers for the acceptance of new code. Similar results were found in other studies [16], where the developers' trustworthiness was more important as well as the code quality and structure. Gousios et al. [17] in a large-scale survey of 749 people, that act as integrators in many different systems, found out the factors that affect the decision of accepting or not a pull request. The code quality was the top factor that influenced the decision of the integrators, along with the testing and the alignment with the project's overall idea. The main takeaway was that both technical and social factors play a significant role in the pull request acceptance. The social aspect was found (and confirmed) to be a very important aspect in other studies as well [18], where the developer was the most important factor that influenced the chances of a pull requests.

On the aspect of technical debt, Silva et al. [19] by analysing 1.722 pull requests, found out that 30% of the rejected pull requests are due to the presence of technical debt issues. It is also noteworthy that the most frequently attributed reason was code design, which was also identified in the study by Zou et al. [20], who analysed 50.000 pull requests from 117 projects in order to find if the coding style affects the pull request chances of being eventually merged. The study revealed that there the more consistent the added code is to the already existing code base, the higher the probability of a merging the pull request. Another study on three projects, namely Spark, Kafka, and React, analysing pull requests from the perspective of code quality [21] showed that there the discussion of technical debt in pull requests appears to be different than in other software artifacts (e.g., code comments, commits, issues, or discussion forums).

B. Labeling of Pull Requests

Even though we know that the structural characteristics of the pull request are very important for the acceptance [18] as well as the collaboration of the developers [7], limited research focused on the concept of labels for pull requests. Yu et al. [22] found out that developers are not actively using labelling in their pull requests; however, the authors noted that this trend might be changing over time. To overcome this issue, the authors developed an automatic way to label pull requests offering a precision of 60%. Moreover, in a study with 10 open source software projects Pooput et al. [23] analysed the pull request characteristics in order to find the factors that are related to their rejection. They found out that the pull request labelling is very important and closely related to the rejection of pull request.

C. Development Activities Effect on Quality

The effect of some development activities on the quality and more specifically to the technical debt has been studied by Zabardast et al. [10]. They analysed 2.286 commits from one project in order to see how Refactoring, Bug Fixing, and New Development affect technical debt. They found out that technical debt is added more often when new features are being added to the project, and it seems to be reduced with commits that are related to fixing bugs. Finally, Refactoring can be both good and bad, which is a pattern that we can see in many other studies [24]. We should note that the categorization was done with the commit messages while the refactoring was found using the Refactoring Miner tool. Finally, another similar study but more dedicated to the Refactoring was done by Maldonado et al. [25]. They examined 5.733 removals of self-admitted technical debt (SATD) in order to find relations with bugs, improvements, and refactorings. They found out that the SATD is introduced to track future bugs and areas of the code that will need improvements. But the most important takeaway is that the removal of SATD takes place when they are fixing bugs or adding new features. And the removal of SATD it rarely takes place as part of refactorings.

D. Evolution of Technical Debt

On the concept of evolution of a software system there have been numerous studies, most of them concluding that the quality of the system deteriorates over time [26]. As for the evolution of technical debt, Digkas et al. [27] studied the weekly changes of 66 projects for a period of 5 years. They found out that, in absolute values, TD Principal increased over time. On the other hand, normalized TD (TD divided over lines of code), termed TD density, decreased over time for most of the

projects. On another study by Digkas et al. [28], the authors focused on the relation between technical debt density and the characteristics of new code. In particular, they studied 27 projects and analysed 57K commits, finding out that clean code (i.e., the practice of introducing new code that has better quality than the already existing code base) can be a very effective way to reduce technical debt density. Tan et al. [29] studied the evolution of 44 Python projects with a total of over 60K commits, and almost 43K of fixed issues. They found out that half of the technical debt is short term, in the sense that is being repaid in less than two months from its introduction. Another takeaway message of this work, was that the repayment was concentrated in testing, complexity, and duplication removal, categories in which the majority of issues were swiftly resolved.

Molnar and Motogna [30] studied the evolution of three Java projects over a total of 110 releases. In order to capture the amount of technical debt along with other metrics, they used SonarQube. They found out that there was a correlation of technical debt with the number of lines that was being added in each version. They also found that 20% of the total issue types, generated 80% of the total technical debt (adhering to the well-known Pareto principle), with consistent findings for almost all projects. Peters and Zaidman [31] studied the lifespans of some code smells in 8 open-source projects. For this reason, they created a tool namely SACSEA, and used it in order to identify a variety of well-known code smells: God Class, Feature Envy, Data Class, Message Chain Class, and Long Parameter List. The results suggested that the number of code smells increases over time, and that even though developers are aware of them chose not to act. But this is not always true, as easier to resolve code smells are fixed more often.

A similar study was carried out by Tufano et al. [32], where the evolution of five code smells was studied. They found out that artifacts with more code smells tend to attract more smells during their evolution, resembling a rich-gets-richer phenomenon Also new code smells are introduced when software engineers implement new features or when they extend the functionality of the existing ones. Finally, a large-scale study by Bavota and Russo [33] with 159 projects took place in order to investigate the evolution of self-admitted technical debt. In this study 600K commits and 2 billion comments were used. They found out that self-admitted technical debt is mostly represented by code with 30%, while defect and requirement debt have 20% each. Regarding the evolution of the software, the authors suggested that TD is continuously increasing and that when it is resolved it takes about 100 commits to do so.

III. Case Study Design

In this section, we present the study design, by providing information about the research questions, the datasets that we used, and the employed statistical analysis. The case study was designed and reported based on the guidelines provided by Runeson et al. [34].

A. Research Questions

To study the effect of various maintenance activities, documented in pull requests, on the accumulation of TD Principal, we have formulated and set the following research questions:

RQ1: What kind of labels are being used to characterize pull requests?

RQ2: What is the effect of each maintenance activity to the amount of TD Principal?

In recent years, the use of Pull Requests to apply changes in software systems is currently becoming more and more popular. However, the information provided along with Pull Requests and more specifically the labels that the developers use for their characterization, yield for further improvement and standardization. In **RQ1**, we investigate the current status of Pull Request labelling. In particular, we explore what labels usually stand for (how do they characterize the changes), as well as, how often they do describe maintenance activities. For **RQ2**, we focus on the three most known maintenance activities (see Section I), and we investigate their effect on TD Principal accumulation. This will allow us to confirm or reject the hypothesis that refactoring activities generally improve quality aspects, functionality addition tends to reduce quality (especially if it is performed under a tight schedule), and explore the effect of bug fixing on technical debt for which no specific assumptions can be made. To achieve this goal, we explore: (a) the effect of software evolution on TD Principal accumulation—without differentiating among maintenance activity types; (b) the effect of maintenance types; and (c) the interaction of the two factors: software evolution and maintenance activities (see Section III.D for more details on the analysis process).

B. Cases and Units of Analysis

Our study is characterized as a multiple, embedded case study [34], where the cases are open-source software projects, and the units of analysis are all the merged Pull Requests of each project. The reason for selecting open-source software projects is the vast amount of open data and the popularity of adopting the mechanism of Pull Requests for contributing to the project. The ten projects that we have investigated in this paper can be found in Table I.

TABLE I. Dataset Demographics

Project	Commits	Pull Requests	Size (Classes)
antlr/antlr4	8,934	1,380	1,453
Netflix/conductor	3,059	1,476	578
DataDog/dd-trace-java	10,329	4,001	3,762
apache/dolphinscheduler	7,608	6,713	2,027
apache/incubator-seatunnel	2,779	2,297	1,700
apache/inlong	3,064	3,323	3,792
provectus/kafka-ui	1,566	1,821	353
apache/rocketmq	7,734	2,691	1,823
apache/skywalking	7,459	4,539	2,089
spring-projects/spring-security	13,086	2,125	5,362

The selection of projects was based on the following criteria:

- The main ***programming language*** is ***Java***, so as to not risk having different results, due to language specific features—we opt for Java since it is the most commonly analysed language in technical debt literature, so that results are comparable to previous work.
- The project is still ***under development*** to ensure that the development practices are not outdated. So, the developers

are actively use pull requests and it is most likely that they are also use labelling.

- The project has ***more than 1,000 closed pull requests***, to have enough data points for each project. The closed pull requests can be either merged or not, and from the merged ones we do not expect all of them to be labelled. So, in this way, we make sure that we have enough data per project for our analysis.
- The project actively ***uses labelling in pull requests***, which is one of the most important criteria, as our maintenance activity categorization is based on them.

C. Data Collection

To gather the appropriate data, i.e., retrieve the pull requests, assess the code quality of the involved code through SonarQube and carry out the appropriate statistical analysis, we developed a software solution to automate the process. In Figure 1, we present the overall functionality of the developed solution. Our complete dataset can be found online[1], as well as the source code of the data collection software[2].

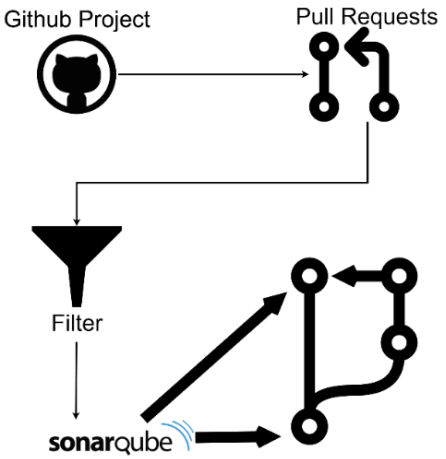

Fig. 1. Data collection process

The data collection process can be split in the following two phases:

Pull Requests Identification. First, we had to extract the pull requests that we were going to analyse. We focused our study on pull requests from GitHub projects that were closed. For this reason, we used the GitHub API to get the pull requests that were closed, merged, and that contained at least one label. Figure 2 represents the pull request that we try to capture from the GitHub API. Moreover, for each pull request that satisfies our constraints we want to get the code before and after the merge of the pull request. This is vital to be able to calculate the TD Principal before and after each pull request.

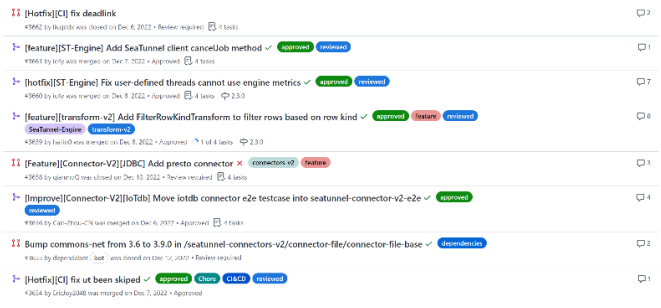

Fig. 2. Pull Request of Interest

Technical Debt Analysis. After having the complete list of the pull requests with the commits before and after their merge, we are able to analyse the code. So, we automated the process by checking out in a specific commit each time and starting the code analysis. For the code analysis, we used the SonarQube which is a very well-known tool for TD Principal calculation [35]. Finally, after the technical debt analysis is finished, we have a dataset with pull request and the TD Principal value before and after the merge.

D. Data Analysis

To answer **RQ1**, we catalogue the text of the labels that are attached to Pull Requests. In some case text processing for labels has been made (e.g., splitting to stopping characters—backslash or colon). The analysis is mostly descriptive, employing simple visualization methods. To understand the purpose of the label as well as their meaning in the context of each project, we consult the descriptions that the project provides. The main goal of the analysis was to explore: (a) how labels are structured and what they do describe; (b) the most frequent labels; and (c) the number of labels that are usually used to characterize a Pull Request.

For answering **RQ2**, by taking into consideration the experimental design of the study (i.e., embedded case study), we used the *Linear Mixed Effects Models* (LMEM) [36]. More specifically, in our setup, Pull Requests were collected from different projects and for this reason, there is a need for taking into account the fact that they may share common characteristics. To this regard, our preference to LMEM rather than other traditional statistical approaches (e.g., ANOVA—Analysis of Variance) is due to the ability of this branch of statistical approaches to handle the nested nature of data (i.e., the Pull Requests are nested within projects).

Mixed Effect Models in a Nutshell

Mixed Effects Models are used for cases that multiple factors might affect a single phenomenon. Drawing an analogy to medicine, consider the case that three different medical treatments are applied for decreasing fever, and we want to check which one is the most efficient one. To achieve this, one usually applies a repeated measures design (e.g., a pre- post-analysis), where the fever is measured before and after intaking the medical treatment. However, apart from the *TREATMENT*, in this scenario, an additional factor is *TIME* (in the sense that fever might decrease even without any medication). To explore both parameters you need to construct a model by considering *TIME and TREATMENT* (as ***fixed main*** effects), and a model considering the main effects along with the ***interaction*** of both *TIME×TREATMENT (**mixed*** effect) on the response.

Given the above, in the context of this research, we explore the effect of *SOFTWARE EVOLUTION* (mapped to TIME) and *MAINTENANCE ACTIVITY* (mapped to TREATMENT) on *TD PRINCIPAL* (mapped to fever). We note that the goal of our experimental setup is to study the effect of several maintenance activities on TD Principal: the factor *Software Evolution*, corresponds to the effect of Pull Requests, without discriminating among *Maintenance Activity* types. In traditional statistics (not considering the nested modelling of this design), this analysis would suggest if there are differences

between types; whereas the second leg of the analysis would unveil which types are different.

To achieve this goal, we created a categorical variable (*Maintenance Activity*) consisting of three categories (namely: ***feature***, ***bug***, and ***refactoring***), which correspond to Pull Request types, based on their labels. We should note that the labels are stated in the same or almost the same way as our categories. More specifically, only labels that we categorize in the ***refactoring*** category can be named differently, but if we dig deeper into their description inside each project, we can see that they are associated with refactoring. In Table II we present the exact label of each project, that we have used so as to categorize it as a development activity.

TABLE II. PULL REQUEST CATEGORIZATION BASED ON LABELS

Project	Feature	Bug	Refactoring
antlr/antlr4	type:feature	type:bug	type:cleanup
Netflix/conductor	type: feature	type: bug	type: maintenance
DataDog/dd-trace-java	-	bug	refactoring
apache/dolphinscheduler	Feature	bug	chore
apache/incubator-seatunnel	Feature	bug	Chore
apache/inlong	type/feature	type/bug	-
provectus/kafka-ui	type/feature	type/bug	type/refactoring
apache/rocketmq	type/new feature	type/bug	type/code style
apache/skywalking	feature	bug	chore
spring-projects/spring-security	-	type: bug	-

IV. RESULTS

In this section, we present the results of our analysis, organized by research question. We note that in this section, we just present the raw results, which we interpret and discuss in Section V.

A. Pull Request Labels Analysis

As a first step in answering RQ_1, we present some descriptive statistics on the labels, per project. Each project has its own labels, but the overall usage is uniform. As it can be observed from Table III, the number of labels that each project has, and their categories are at a similar level of magnitude. It is very common for a project to group some labels under the same category. For example, to group the labels for each affected component, development teams might use the terms: "*module*", "*component*", or "*comp*" prefix in the label name. For example, one can observe labels, such as: "*module/controller*", "*module/container*", "*module/admin*", etc. This categorization is very popular and 9 (out of 10) projects that we have analysed, follow this notation.

TABLE III. PROJECT'S PULL REQUEST LABELS CHARACTERISTICS

Project	No. Labels	No. Groups	Groups
antlr/antlr4	46	4	comp, status, target, type
Netflix/conductor	25	1	type
DataDog/dd-trace-java	61	1	dev
apache/dolphinscheduler	80	1	priority
apache/incubator-seatunnel	50	0	-
apache/inlong	22	2	component, type
provectus/kafka-ui	41	4	impact, scope, status, type
apache/rocketmq	44	3	module, progress, type,
apache/skywalking	60	1	complexity
spring-projects/spring-security	52	4	for, in, status, type

Another interesting observation is that it is quite common for a Pull Request to be characterized with more than one labels. The reason behind this is quite straightforward since a developer can label a Pull Request based on: (a) its status—e.g., triage, status, etc.; (b) the component (or components) that it affects—e.g., controller admin, etc.; (c) the type of the change—e.g., bug, refactoring, etc. In Figure 3, we visualize the frequency distribution of the number of labels that are used for the characterization of the analysed Pull Requests. The results suggest that that the vast majority of the identified Pull Requests have up to three labels and the average number of labels is 1.7 per Pull Request.

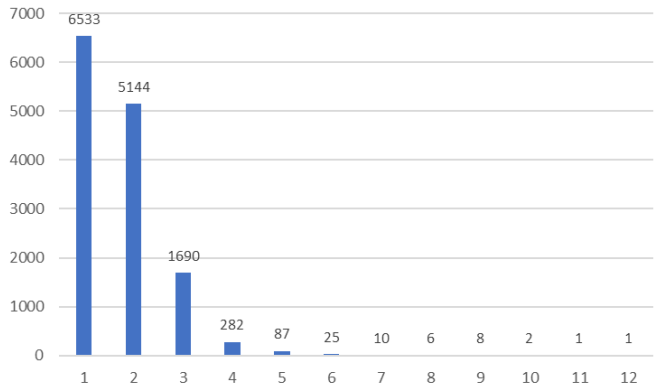

Fig. 3. Number of labels in Pull Requests

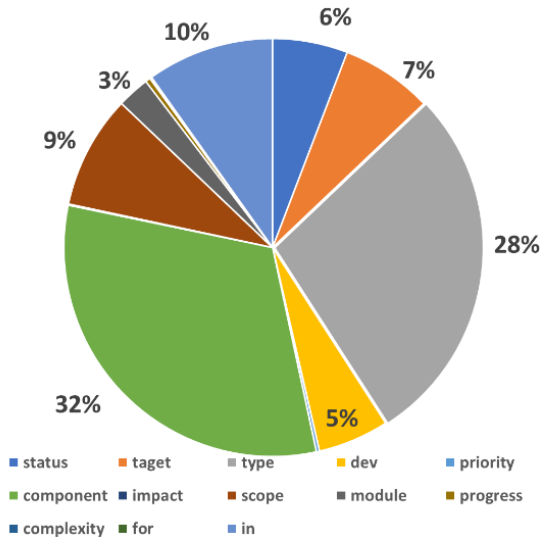

Fig. 4. Popularity of Pull Requests Groups

In Figure 4, we, graphically, depict the popularity of each group. Based on the results, we can observe that the most frequent kind of information is the "WHERE" information,

as captured by the comp, the component, the module, the in, and the target labels (cumulatively summing up to 51%), followed by information on "WHAT" as captured by the type label (28%). Given the fact that information captured by the "*WHERE*" labels are application specific, in Figure 5, we dig further in the "*WHAT*" information, presenting the frequency of the most popular types of maintenance actions. Based on the findings, in order to focus on the targeted maintenance activities, we need to neglect the types: ***enhancement***, ***improve***, and ***documentation***. On one hand, enhancement and improvement could reflect to both quality and/or functionality enhancement: therefore, it would be risky to include them. On the other hand, although documentation most probably refers to quality improvement it cannot be treated as a refactoring. In that sense, for RQ_2 we focus on the top-3 labels, based on popularity that are clearly mapped to maintenance activities.

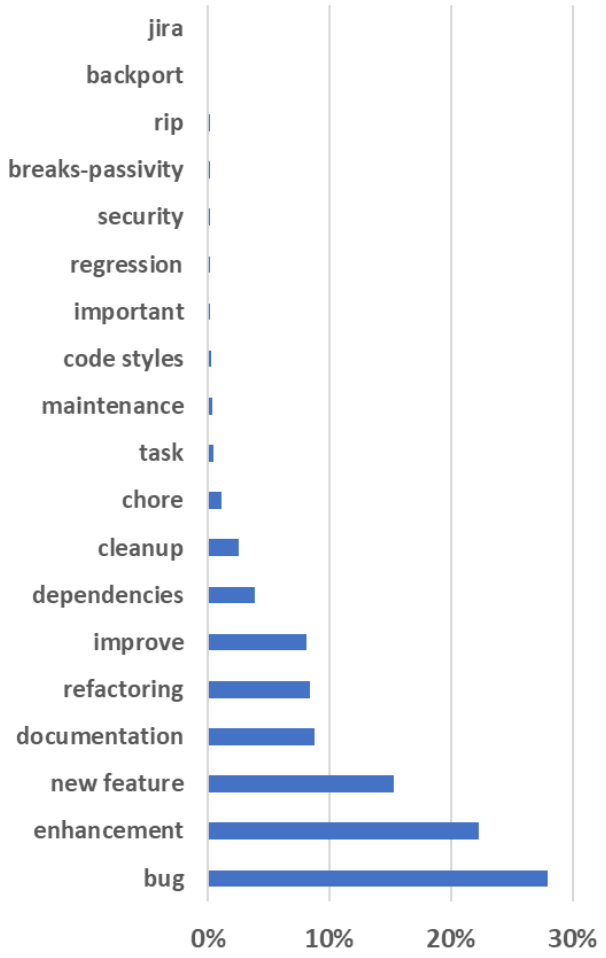

Fig. 5. Popularity of Maintenance Type Labels

B. *Effect of Software Evolution and Maintenance Types on TD Principal Accumulation*

As a first step in answering **RQ_2**, we first explore the frequency of the effect (positive, neutral, negative) of each maintenance activity type on TD Principal. In Figure 6, we present the percentage of cases that a specific maintenance type produced positive or negative effects. From Figure 6, we can observe that around 62% of the total Pull Requests do not impact TD Principal since they either do not introduce or reduce technical debt issues or they introduce / remove an equal amount. By cross-comparing the first row of the stacked barchart with the next ones, we can observe that ***New Features*** are having an almost doubled-up chance of increasing TD Principal, ***Refactorings*** have the highest chances of reducing TD Principal, whereas for ***Bug Fixing***, we can observe the highest percentage of neutral cases.

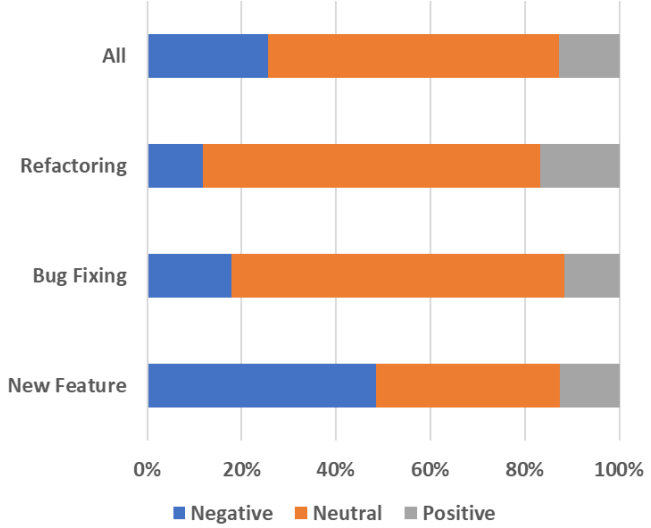

Fig. 6. PR Impact Count for each Maintenance Type

Next, proceeding to LMEM analysis, Table IV provides a summary of the exploratory analysis for the combination of the levels for the two examined factors. As we can observe, on average ***Bug Fixing*** and the addition of ***New Features*** increase the amount of Pull Requests Principal in minutes (even marginally), whereas, ***Refactoring*** reduces it. As explained in Section III.D, we fitted two LMEMs, to investigate if there is a statistically significant interaction effect of the two examined factors on Pull Requests *Principal*. The former model (or the full model) contains the main fixed effects of *Software Evolution* and *Maintenance Activity* and their interaction (*Software Evolution*×*Maintenance Activity*), whereas the second model incorporates only the main effects without their interaction term. At this point, we need to mention that models were fitted on the logarithmic transformation of the response in order to satisfy the assumption of homogeneity of variance means.

These two models were tested against each other through the *Wald* test, whereas the model presenting the lowest *Akaike Information Criterion* (AIC) value was, finally, selected for any further analysis. Indeed, the comparison of the two LMEMs indicated a statistically significant interaction effect ($p = 0.002$) and for this reason, the full model with the interaction term was chosen, since it presented lower AIC[3] value ($\mathrm{AIC} = -13347$) compared to the model without the interaction term ($\mathrm{AIC} = -13339$). As far the final fitted model concerns, Table V summarizes the Analysis of Variance (ANOVA) findings. We have to note that due to the existence of a statistically significant interaction term, emphasis needs to be given on understanding the changes of *TD Principal* caused by *Software Evolution,* based on the type of *Maintenance Activity*.

[3] Akaike Information Criterion: expresses how well the corresponding model fits the data - a lower AIC value indicates better fitting

TABLE IV. DESCRIPTIVE STATISTICS

Maintenance Activity	Software Evolution	*N*	*Mean*	*SD*	*Median*	*Minimum*	*Maximum*
Bug Fixing	After	2380	32,643.51	21569.99	26548.0	52	191261
	Before	2380	32,572.23	21380.83	26564.5	52	191208
New Feature	After	1247	36,495.92	19692.69	30169.0	2230	89138
	Before	1247	36,437.65	19646.78	30136.0	2230	88550
Refactoring	After	740	34,197.39	26259.09	31479.5	2303	89809
	Before	740	34,315.53	26378.01	31479.5	2303	89809

TABLE V. ANOVA RESULTS FROM LMEM

Factor	*DF*	*Den-DF*	F-value	*p-value*
Maintenance Activity	2	4355.8	7.135	<0.001
Software Evolution	1	4364.0	0.237	0.627
Maintenance Activity × Software Evolution	2	4364.0	6.066	0.002

Having concluded that both factors are important, we proceeded to post-hoc contrasts analysis in order to examine the effect of each maintenance activity to the accumulation of *TD Principal*. The results presented in Table VI indicate that there was noted a statistically significant difference on the mean values of *TD Principal* only for the two types of pull requests: Addition of a ***New Feature*** and ***Refactoring***, before and after the merge of a Pull Request. Additionally, a noteworthy finding can be derived from the sign of the estimated mean difference. The positive sign for Addition of ***New Feature*** activity indicates an increase in *TD Principal*, whereas the opposite is the case for the **Refactoring** activity. The findings are also graphically displayed in Figure 7, where the blue bars represent the 95% confidence intervals for the estimated mean differences.

TABLE VI. POST-HOC CONTRASTS

Maintenance Activity	Software Evolution	*Estimate*	*SE*	*z*	*p*
Bug Fixing	After-Before	0.005	0.0006	1.010	0.313
New Feature	After-Before	0.002	0.0008	1.983	0.047
Refactoring	After-Before	-0.003	0.001	-2.762	0.006

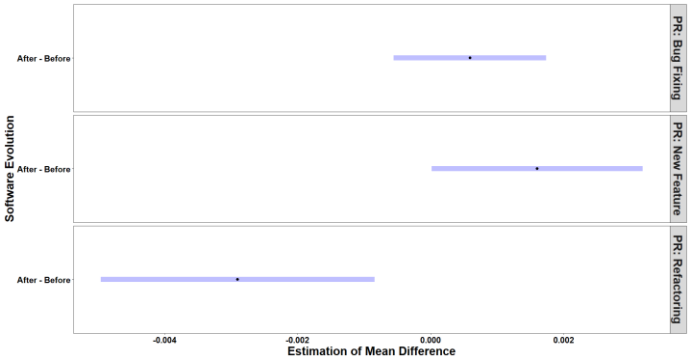

Fig. 7. Difference of Mean Values Accompanied by 95% CIs

V. DISCUSSION

A. Recap and Interpretation of Results

The main findings of this work can be summarized as follows: (a) the labels assigned to Pull Requests can provide useful input for evolution analysis, in the sense that the used labels can characterize both the type of maintenance / development activity, as well as the affected part of the system; (b) adding new features to the software usually increases the amount of TD Principal; (c) refactoring tends to reduce the amount of TD Principal; (d) bug fixing seems to be neutral in terms of changes to the TD Principal amount. Below, each of these findings is discussed in more detail.

Pull Requests Label appear to be a promising way to control development and evolution analysis studies, in the sense that: (a) they characterise accurately the activity that has been performed; and (b) they group commits to larger chunks of evolution that serve a common goal. The fact that the analysis pointed out to various similarities in terms of Pull Requests' groups and labels suggest a homogeneity across different projects; an observation that further strengthens the belief that they can be extremely useful for future studies (see Section V.B). Also, the fact that some Pull Request labels can be mapped in a one-to-one manner to well-established maintenance activities, also seems to enable a more straightforward analysis of the evolution.

TD Principal Accumulation and Adding New Features: The fact that adding a new feature in the code bases has proven to be the costliest (in terms of TD Principal) maintenance activity is well-anticipated. This can be explained by the fact that adding a new feature (in contrast to bug fixing and refactoring) usually ends up to adding additional lines of code in the codebase [10]. These additional lines are more probable to introduce new technical debt issues (violate additional rules of SonarQube) compared to fixing a bug (usually smaller increments/deletions from the code) or dedicated refactoring sessions.

TD Principal Accumulation and Refactoring: This study has provided evidence that performing a refactoring is the maintenance activity type that is most probable (in terms of frequency) and the most efficient (in terms of value) way to reduce TD Principal. This study, opposes several previous ones that were suggesting that the effect of refactorings is undecisive [24]. The most probable reason for this differentiation comes from the study setup that uses Pull Requests instead of commits. Using a Pull Request as a unit of analysis: (a) safeguards that a refactoring is not mixed with other types of maintenance; and (b) might point to refactoring sessions, under which in various commits more than one refactorings take place, increasing the chances of capturing and resolving technical debt issue violations.

TD Principal Accumulation and Bug Fixing: The results of this work were unable to produce solid evidence on the effect of bug fixing on technical debt accumulation. This finding might emerge from the fact that functional bug fixing is usually unrelated to maintainability. In the literature it has been made clear that bugs or defects are not accounted as TD Principal (in the sense that they do not produce technical debt Interest) [37]; although, still relevant to technical debt management (since they are one of the main factors for TD Interest probability—i.e., the reason to change a high-TD Principal class). Our results comply with this view, in the sense that resolving a bug, usually does not affect the value of TD Principal.

Software Evolution and TD Principal Accumulation: One of the key statements that motivated this work was that TD Principal tends to increase along software evolution [27]. Although this statement seems to contradict the main findings of this work (one type of maintenance reduces technical debt, the other is neutral, and only one increase technical debt), through a deeper analysis we can observe that the statement is actually confirmed. In particular, by considering not only the effect, but also the frequency of the labels we can observe that Pull Requests that have a negative effect on TD Principal sum-up to 26%, whereas the Pull Requests that have a positive effect account to 13%. In that sense, we can claim that the cumulative effect of New Feature Pull Requests is higher than the cumulative effect of Refactoring Pull Requests, leading to an overall quality deterioration (increase in the absolute value of TD Principal).

B. Implications to Researchers

Based on the findings of this work, we can provide some useful research implications, as well as, some interesting future work directions. First, the decision to use Pull Requests as a unit of analysis has proven to be a promising alternative to studying software evolution, in the sense that Pull Request in contrast to commits are: (a) providing chunks of revisions that serve a common goal—providing the ability to study specific types of activities or components; and (b) stand at a higher level of granularity—providing (or at least expected) to provide more homogenous results that are not so sensitive as commit level analysis. Based on this, we believe that the following interesting research opportunities can be further explored:

- study software evolution at the level of Pull Request, rather than at commit level, and contrast the results—validate or contradict the current beliefs on software evolution in larger but more cohesive software changes;
- employ Pull Request labels for enabling tracing requirements to source code implementations;
- study technical debt accumulation in specific components (or modules), and contrast it to the centrality of the component in terms of how many Pull Requests they are involved into.

Additionally, the use of LMEM as the employed statistical analysis method, opens new directions in terms of how empirical methods are used in software engineering, in the sense that the vast majority of experimental setups are nested. In terms of analysis, we also suggest a replication of this work to study the effect of software evolution on technical debt accumulation, using multiple timestamps (repeated measures instead of a pre- post- analysis). Moreover, a replication of the study concerning technical debt Interest, on top of the TD Principal analysis would also be beneficial, and would expand the body of knowledge on technical debt evolution.

Finally, the results of this work, in terms of the effect of maintenance types on technical debt accumulation, open up several interesting research directions. The fact that for both refactoring and addition of new features, we have identified cases that the Pull Request has positive or negative effect, renders further investigation highly interesting. In particular, we believe that machine learning, deep learning, or explainable AI models can be used for identifying the characteristics of systems, whose maintenance is beneficial. For example, we plan to isolate the refactoring units of analysis and provide guidance (either as thresholds or rules) on when (based on the structural characteristics of classes to be refactored) a refactoring should be applied, and when it can be neglected. The extracted models will offer a deeper understanding of the phenomenon, but also will result in a very useful and practical solution (tool), since refactoring opportunities (repayment) prioritization is a key-problem in technical debt management. Also, we plan to explore the bug-fixing Pull Requests and investigate the long-term effect that they have on the technical debt interest probability of the involved artifacts (classes and components).

C. Implications to Practitioners

The findings of this study also provide some useful implications to practitioners. First, based on the findings and the knowledge that can be extracted from Pull Requests, we encourage developers to use the Pull Requests mechanisms for contributing to projects and use appropriate labels for characterization of the contribution. In addition to this, we encourage the use of the hierarchical (two-level) labels in the form of `group:label` so that information is well-organized. Finally, we encourage organizations to promote the use of a common and standardized scheme for Pull Requests characterization, to avoid a large and diverse number of labels. In terms of maintenance activities, we promote the organization of refactoring sessions, organized under Pull Requests, that lead to substantial improvements in maintainability. Also, since it seems that the addition of new features is the activity that is mostly responsible for the accumulation of technical debt, we encourage the use of quality gates, especially for the cases of adding new features.

VI. Threats to Validity

In this section, we present and discuss construct, reliability, external, and internal threats to the validity of this study [34]. Construct validity reflects to what extent the phenomenon under study really represents what is investigated according to the research questions. The reliability of the case study is related to whether the data is collected and the analysis is conducted in a way that can be replicated with the same results. External validity deals with possible threats while generalizing the findings derived from the sample to a broader population. Finally, internal validity is related to identification of confounding factors, i.e., factors other than the independent variables that might influence the value of the dependent variable.

A. Construct Validity

In our study, construct validity is related to the accuracy of TD Principal quantification. In this work, TD Principal is quantified through SonarQube. SonarQube is the most frequently used tool for measuring TD Principal [12], both in research and practice. Although SonarQube is an established tool, it

focuses on code and partially to design technical debt, neglecting other types of technical debt, like architecture, requirements debt, etc. According to Martini et al. [38], currently in industry static analysers (such as SonarQube) are used to analyse 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. [39] discuss SonarQube as the mostly used tool for technical debt managementin the eight development teams that they have involved in their case study. Despite the identified limitations, especially at the level of Architectural Technical Debt (ATD), SonarQube is considered as extremely useful for code technical debt identification, monitoring, measurement and prioritization. Additionally, although SonarQube could be configured to provide more accurate results (e.g., change remediation times), such a practice is not prominent in the literature, where researchers do not perform any re-configuration of the tool [40].

Finally, when examining the effect of software evolution on technical debt accumulation, we performed a pre- post-analysis on two timestamps. Although this decision was the most fitting one for the current study design, to draw safer conclusions on the effect of evolution on technical debt accumulation a repeated measurement strategy on multiple timepoints would be more informative.

B. Reliability

To mitigate threats to reliability, two different researchers were involved in the data collection and one double-checked the results of the other. Furthermore, one researcher double-checked the results of the data analysis performed by another researcher. All primitive data, as well as the results, can be reproduced by using the tools mentioned in Section III, and followed the specified research protocol and analysis—see Section III, as well.

C. External Validity

Concerning external validity, we have identified two possible threats to the validity of our results. Firstly, all the investigated systems are written in Java and there is a possibility that the results would be different for other programming languages.

Secondly, this study was conducted on 10 open-source projects, which cannot be considered a big sample (considering the size of the population). The pull requests that we analysed are 13,790. As a result, we cannot generalize our findings to all open-source projects, and even more to industrial ones. The main reason was the time-consuming process of the analysis with SonarQube. For example, if each analysis takes 4 minutes, for 27,580 (2 analyses for each pull request) we needed 76 days. We plan to mitigate this threat by conducting a broader study with more open-source projects. As for the execution time, the main limitation lies on our dependency to SonarQube, which falls outside our control. However, we plan to add more metrics and data regarding each analysis, in order to take full advantage of the SonarQube result.

D. Internal Validity

The usual confounding factor (reason for internal validity threats) in this kind of studies is related to a possible lack of control on the categorization of the type of maintenance activities (i.e., mixing types so that before and after values are affected by additional parameters rather than the studied one). In this study, the labels of each Pull Request have been added by the developers, but we cannot be certain that only the development activity expressed in the label took place. To mitigate this threat, we selected large-scale, well-known projects that are expected to use Pull Requests in a consistent manner. Moreover, in the ten selected projects, the reviewers, which check the code before the merge, also assess and change if needed the applied labels—as stated in projects' documentation.

VII. Conclusions

In this study, we focused on exploring the effect that three major types of development activity have on TD Principal. The examined development actions were related to the addition of new features, bug fixing, and refactoring activities. For our dataset we focused on 10 open-source projects from GitHub, using their pull requests as units of analysis. We analysed each pull request before and after its merging in the codebase, and we used their labels in order to classify them to a development activity.

The results of our study shows that the usage of pull request labelling is quite extended, at least for the studied projects. It is clear that labelling pull requests helps in the development and collaboration of developers. It is also evident that the number of labels that exist in the projects is quite large and that labels are also grouped. A key finding was that Pull Requests associated with refactoring activities present higher chances of having a positive effect on TD Principal. On the other hand, the addition of new software features appears to frequently increase the amount of TD Principal. Bug fixing, in general, seems to be neutral in terms of changes to TD Principal.

Acknowledgments

This work has been partially funded by the Horizon Europe Framework Programme of the European Union under Grant agreement no 101058479.

References

[1] W. Cunningham, "The WyCash Portfolio Management System" in OOPSLA'92. Vancouver, 1992

[2] A. Ampatzoglou, N. Mittas, A. A. Tsintzira, A. Ampatzoglou, E. M. Arvanitou, A. Chatzigeorgiou, P. Avgeriou, and L. Angelis, "Exploring the Relation between Technical Debt Principal and Interest: An Empirical Approach", Information and Software Technology, Elsevier, December 2020.

[3] G. Digkas, A. Chatzigeorgiou, A. Ampatzoglou and P. Avgeriou, "Can Clean New Code Reduce Technical Debt Density?," in IEEE Transactions on Software Engineering, vol. 48, no. 5, pp. 1705-1721, 1 May 2022, doi: 10.1109/TSE.2020.3032557.

[4] E. M. Arvanitou, N. Nikolaidis, A. Ampatzoglou, and A. Chatzigeorgiou, "Practitioners' Perspective on Practices for Preventing Technical Debt Accumulation in Scientific Software Development"', 17th International Conference on the Evaluation of Novel Approaches to Software Engineering (ENASE '22), SCITEPRESS, April 2022.

[5] G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou and A. Ampatzoglou, "How do developers fix issues and pay back technical debt in the Apache ecosystem?," 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Campobasso, Italy, 2018, pp. 153-163, doi: 10.1109/SANER.2018.8330205.

[6] H. van Vliet, "Software Engineering: Principles and Practice", John Wiley, 2008

[7] G. Gousios, A. Zaidman, M. -A. Storey and A. v. Deursen, "Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective," 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 2015, pp. 358-368, doi: 10.1109/ICSE.2015.55.

[8] X. Zhang, Y. Yu, G. Georgios and A. Rastogi, "Pull Request Decisions Explained: An Empirical Overview," in IEEE Transactions on Software Engineering, doi: 10.1109/TSE.2022.3165056.

[9] T. Hastings and K. R. Walcott, "Continuous Verification of Open Source Components in a World of Weak Links," 2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Charlotte, NC, USA, 2022, pp. 201-207, doi: 10.1109/ISSREW55968.2022.00068.

[10] E. Zabardast, J. Gonzalez-Huerta and D. Šmite, "Refactoring, Bug Fixing, and New Development Effect on Technical Debt: An Industrial Case Study," 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Portoroz, Slovenia, 2020, pp. 376-384, doi: 10.1109/SEAA51224.2020.00068.

[11] 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.

[12] P. C. Avgeriou *et al*., "An Overview and Comparison of Technical Debt Measurement Tools," in *IEEE Software*, vol. 38, no. 3, pp. 61-71, May-June 2021, doi: 10.1109/MS.2020.3024958.

[13] T. Amanatidis, A. Chatzigeorgiou, and A. Ampatzoglou, "The relation between technical debt and corrective maintenance in PHP web applications," Inf. Softw. Technol., vol. 90, pp. 70–74, Oct. 2017, doi: 10.1016/j.infsof.2017.05.004.

[14] I. Zozas, S. Bibi, A. Ampatzoglou and P. Sarigiannidis, "Estimating the Maintenance Effort of JavaScript Applications," 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Kallithea, Greece, 2019, pp. 212-219, doi: 10.1109/SEAA.2019.00042.

[15] V. Lenarduzzi, V. Nikkola, N. Saarimäki, and D. Taibi, "Does code quality affect pull request acceptance? An empirical study," J. Syst. Softw., vol. 171, p. 110806, 2021.

[16] F. Calefato, F. Lanubile and N. Novielli, "A Preliminary Analysis on the Effects of Propensity to Trust in Distributed Software Development," 2017 IEEE 12th International Conference on Global Software Engineering (ICGSE), Buenos Aires, Argentina, 2017, pp. 56-60, doi: 10.1109/ICGSE.2017.1.

[17] G. Gousios, A. Zaidman, M. -A. Storey and A. v. Deursen, "Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective," 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 2015, pp. 358-368, doi: 10.1109/ICSE.2015.55.

[18] D. Moreira Soares, M.L. de Lima Júnior, L. Murta, and A. Plastino, 2021. "What factors influence the lifetime of pull requests". Software: Practice and Experience, 51(6), pp.1173-1193.

[19] M. C. O. Silva, M. T. Valente, and R. Terra, "Does technical debt lead to the rejection of pull requests?," ArXiv Prepr. ArXiv160401450, 2016.

[20] W. Zou, J. Xuan, X. Xie, Z. Chen, and B. Xu, "How does code style inconsistency affect pull request integration? an exploratory study on 117 github projects," Empir. Softw. Eng., vol. 24, no. 6, pp. 3871–3903, 2019.

[21] S. Karmakar, Z. Codabux, and M. Vidoni, "An Experience Report on Technical Debt in Pull Requests: Challenges and Lessons Learned," in Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2022, pp. 295–300.

[22] S. Yu, L. Xu, Y. Zhang, J. Wu, Z. Liao and Y. Li, "NBSL: A Supervised Classification Model of Pull Request in Github," 2018 IEEE International Conference on Communications (ICC), Kansas City, MO, USA, 2018, pp. 1-6, doi: 10.1109/ICC.2018.8422103.

[23] P. Pooput and P. Muenchaisri, "Finding impact factors for rejection of pull requests on github," in Proceedings of the 2018 VII International Conference on Network, Communication and Computing, 2018, pp. 70–76.

[24] N. Nikolaidis, D. Zisis, A. Ampatzoglou, N. Mittas, and A. Chatzigeorgiou, "Using machine learning to guide the application of software refactorings: a preliminary exploration," in Proceedings of the 6th International Workshop on Machine Learning Techniques for Software Quality Evaluation, 2022, pp. 23–28.

[25] E. D. S. Maldonado, R. Abdalkareem, E. Shihab and A. Serebrenik, "An Empirical Study on the Removal of Self-Admitted Technical Debt," 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China, 2017, pp. 238-248, doi: 10.1109/ICSME.2017.8.

[26] 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", Proc. 4th Int. Softw. Metrics Symp., pp. 20-32, 1997.

[27] G. Digkas, M. Lungu, A. Chatzigeorgiou, and P. Avgeriou, "The evolution of technical debt in the apache ecosystem," in European conference on software architecture, 2017, pp. 51 – 66.

[28] G. Digkas, A. Chatzigeorgiou, A. Ampatzoglou and P. Avgeriou, "Can Clean New Code Reduce Technical Debt Density?," in IEEE Transactions on Software Engineering, vol. 48, no. 5, pp. 1705-1721, 1 May 2022, doi: 10.1109/TSE.2020.3032557.

[29] J. Tan, D. Feitosa, P. Avgeriou, and M. Lungu, "Evolution of technical debt remediation in Python: A case study on the Apache Software Ecosystem," J. Softw. Evol. Process, vol. 33, no. 4, p. e2319, 2021, doi: 10.1002/smr.2319

[30] A.-J. Molnar and S. Motogna, "Long-term evaluation of technical debt in open-source software," in Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2020, pp. 1–9.

[31] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining", Proc. 16th Eur. Conf. Softw. Maintenance Reengineering, pp. 411-416, 2012.

[32] M. Tufano et al., "When and why your code starts to smell bad (and whether the smells go away)", IEEE Trans. Softw. Eng., vol. 43, no. 11, pp. 1063-1088, Nov. 2017.

[33] G. Bavota and B. Russo, "A large - scale empirical study on self - admitted technical debt," in Proceedings of the 13th international conference on mining software repositories, 2016, pp. 315 – 326

[34] P. Runeson, M. Host, A. Rainer, and B. Regnell, Case Study Research in Software Engineering: Guidelines and Examples. John Wiley & Sons, 2012.

[35] G.A. Campbell, and P.P. Papapetrou, "SonarQube in action". Manning Publications Co. 2013

[36] J. Jiang and T. Nguyen, Linear and Generalized Linear Mixed Models and Their Applications. New York, NY: Springer New York, 2021. doi: 10.1007/978-1-0716-1282-8.

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

[38] 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, Elsevier, 163, pp. 42-61, 2018.

[39] J. Yli-Huumo, A. Maglyas, and K. Smolander, "How do software development teams manage technical debt?–An empirical study" *Journal of Systems and Software*, Elsevier, 120, pp. 195-218, 2016.

[40] M. Schnappinger, M.H. Osman, A. Pretschner, and A. Fietzke, "Learning a classifier for prediction of maintainability based on static analysis tools" Proceedings of the 27th International Conference on Program Comprehension, IEEE Press, pp. 243-248, 2019.