Does Code Quality Affect Pull Request Acceptance? An empirical study

Valentina Lenarduzzi[a], Vili Nikkola[b], Nyyti Saarimäki[b], Davide Taibi[b]

[a]*LUT University, Lahti (Finland)*
[b]*Tampere University, Tampere (Finland)*

Abstract

Background. Pull requests are a common practice for making contributions and reviewing them in both open-source and industrial contexts.
Objective. Our goal is to understand whether quality flaws such as code smells, anti-patterns, security vulnerabilities, and coding style violations in a pull request's code affect the chance of its acceptance when reviewed by a maintainer of the project.
Method. We conducted a case study among 28 Java open-source projects, analyzing the presence of 4.7 M code quality flaws in 36 K pull requests. We analyzed further correlations by applying logistic regression and six machine learning techniques. Moreover, we manually validated 10% of the pull requests to get further qualitative insights on the importance of quality issues in cases of acceptance and rejection.
Results. Unexpectedly, quality flaws measured by PMD turned out not to affect the acceptance of a pull request at all. As suggested by other works, other factors such as the reputation of the maintainer and the importance of the delivered feature might be more important than other qualities in terms of pull request acceptance.
Conclusions. Researchers have already investigated the influence of the developers' reputation and the pull request acceptance. This is the first work investigating code style violations and specifically PMD rules. We recommend that researchers further investigate this topic to understand if different measures or different tools could provide some useful measures.

Keywords: Pull Requests, PMD rules, Machine Learning

Email addresses: `valentina.lenarduzzi@lut.fi` (Valentina Lenarduzzi), `vili.nikkola@tuni.fi` (Vili Nikkola), `nyyti.saarimaki@tuni.fi` (Nyyti Saarimäki), `davide.taibi@tuni.fi` (Davide Taibi)

1. Introduction

Pull requests provide developers a convenient way of contributing to projects, and many popular projects, including both open-source and commercial ones, are using pull requests as a way of reviewing the contributions of different developers.

Researchers have focused their attention on pull request mechanisms, investigating different aspects, including the review process [1, 2, 3], how pull requests are assigned to different reviewers [4], and in which conditions they are accepted [1, 5, 6, 7]. Researchers have also highlighted that the reputation of the developer submitting the pull request and especially the implementation of new features seem to be more important acceptance factors than any other aspects, including quality [2, 8].

To the best of our knowledge, no studies have investigated if quality flaws such as code smells, anti-patterns, or coding style violations affecting a pull request have an impact on the acceptance of the pull request itself.

Therefore, in order to understand whether quality flaws can be one of the drivers for accepting pull requests, we designed and conducted a case study involving 28 well-known Java projects to analyze the presence of quality flaws in more than 36K pull requests. We considered the quality flaws highlighted by PMD rules[1] (code smells, anti-patterns, and coding style violations), investigating if the presence of these PMD quality flaws in pull requests affect the chance of its acceptance when it is reviewed by a project maintainer.

We analyzed the quality flaws in pull requests using PMD, one of the most frequently used static analysis tools [9, 10]. PMD evaluates the code quality against a standard rule set available for the major languages, allowing the detection of different quality aspects generally considered harmful, including code smells [11] such as "long methods", "large class", "duplicated code"; anti-patterns [12] such as "high coupling"; design issues such as "god class" [13]; and various coding style violations[2]. Whenever a rule is violated, PMD raises an issue. In the remainder of this paper, we will refer to any quality flaw raised by PMD as a "*PMD issue*" .

Previous work confirmed that the presence of PMD issues in the code, including the code smells and anti-patterns collected by PMD, significantly increases the risk of faults and maintenance effort [14, 15, 16, 17]. Therefore,

[1]https://pmd.github.io
[2]https://pmd.github.io/latest/pmd_rules_java.html

we expect that developers take care of these issues, in order to increase software maintainability and decrease fault-proneness.

Unexpectedly, the application of statistical techniques shows that the presence of PMD issues of any type does not influence the acceptance or rejection of a pull request at all. Therefore, we decided to apply six machine learning models (Decision Trees, Random Forest, Extremely Randomized Trees, AdaBoost, Gradient Boosting, and XGBoost) in order to confirm or reject the results overcoming to the limitation of the statistical techniques. Moreover, we manually inspected 10% of the rejected and 10% of the accepted pull requests, to get further insight on the impact of the quality on pull request acceptance.

Results are confirmed also after the application of the machine learning models. PMD issues in pull requests, including code smells and anti-patterns, are not considered as a problem by project maintainers when accepting or rejecting pull requests.

Structure of the paper. Section 2 describes the basic concepts underlying this work, while Section 3 presents some related work done by researchers in recent years. In Section 4, we describe the design of our case study, defining the research questions, metrics, and hypotheses, and describing the study context, including the data collection and data analysis protocol. In Section 5, we present the achieved results and discuss them in Section 6. Section 7 identifies the threats to the validity of our study, and in Section 8, we draw conclusions and give an outlook on possible future work.

2. Background

In this Section, we first introduce code quality aspects and PMD, the tool we used to analyze the code quality of the pull requests. Then, we will describe the pull request mechanism and finally provide a brief introduction and motivation for the usage of the applied machine learning techniques.

2.1. Code Quality and PMD

Different tools on the market can be used to evaluate code quality. PMD is one of the most frequently used static code analysis tools for Java on the market, along with Checkstyle[3], Findbugs[4], and SonarQube [9].

[3]Checkstyle: https://checkstyle.sourceforge.io
[4] Findbugs: http://findbugs.sourceforge.net

Among the aforementioned tools, we selected PMD since it analyzes uncompiled code, it in order to avoid incompatibilities with required libraries or missing versions of libraries [18]. This allowed us to analyze all the commits included in our dataset, avoiding the risk to not be able to consider "old" commits that are difficult to compile as suggested by Tufano et al. [18]. In their study, they examined 100 Java ASF projects and found that unfortunately on average only 38% of a project's change history could be successfully compiled.

PMD is an open-source tool that aims to identify issues that can lead to technical debt accumulating during development [19]. The specified source files are analyzed and the code is checked with the help of predefined rule sets. PMD provides a standard rule set for major languages, which the user can customize if needed. The default Java rule set encompasses all available Java rules in the PMD project and is used throughout this study.

Issues found by PMD have one of five possible priority values (P):

- P1 Change absolutely required. Behavior is critically broken/buggy.
- P2 Change highly recommended. Behavior is quite likely to be broken/buggy.
- P3 Change recommended. Behavior is confusing, perhaps buggy, and/or against standards/best practices.
- P4 Change optional. Behavior is not likely to be buggy, but more just flies in the face of standards/style/good taste.
- P5 Change highly optional. Nice to have, such as a consistent naming policy for package/class/fields

These priorities are used in this study to help determine whether more severe issues affect the rate of acceptance in pull requests. Rule priority guidelines for default and custom-made rules can be found in the PMD project documentation[4].

PMD does not require compiling the code to be analyzed. This is why, as the aim of our work was to analyze only the code of pull requests instead of the whole project code, we decided to adopt it. PMD defines more than 300 rules for Java, classified in eight categories (coding style, design, error-prone, documentation, multi-threading, performance, and security). The complete set of rules is available on the PMD official documentation[4]. Several rules have also been confirmed harmful by different empirical studies. In Table I, we highlight a subset of rules and the related empirical studies that confirmed their harmfulness.

Table 1: An example of PMD rules and their related harmfulness

PMD Rule	**Defined By**	**Impacted Characteristic**
Avoid Using Hard-Coded IP	Brown et al [20]	Maintainability [20]
Loose Coupling	Chidamber and Kemerer [21]	Maintainability [22]
Base Class Should be Abstract	Brown et al [20]	Maintainability [14]
Coupling Between Objects	Chidamber and Kemerer [21]	Maintainability [22]
Cyclomatic Complexity	Mc Cabe [23]	Maintainability [22]
Data Class	Fowler [11]	Maintainability [24], Faultiness [25], [26]
Excessive Class Length	Fowler (Large Class) [11]	Change Proneness [27], [28]
Excessive Method Length	Fowler (Large Method) [11]	Change Proneness [29], [28] Fault Proneness [27]
Excessive Parameter List	Fowler (Long Parameter List) [11]	Change Proneness [29]
God Class	Marinescu and Lanza [13]	Change Pronenes [30], [31], [32], Comprehensibility [33], Faultiness [30][32]
Law of Demeter	Fowler (Inappropriate Intimacy) [11]	Change Proneness [27]
Loose Package Coupling	Chidamber and Kemerer [21]	Maintainability [22]
Comment Size	Fowler (Comments) [11]	Faultiness [34], [35]

2.2. Git and Pull Requests

Git[5] is a distributed version control system that enables users to collaborate on a coding project by offering a robust set of features to track changes to the code. Features include committing a change to a local repository, pushing that piece of code to a remote server for others to see and use, pulling other developers changesets onto the user's workstation and merging the changes into their own version of the codebase. Changes can be organized into branches, which are used in conjunction with pull requests. Git provides the user a "diff" between two branches, which compares the branches and provides an easy method to analyze what kind of additions the pull request will bring to the project if accepted and merged into the master branch of the project.

Pull requests are a code reviewing mechanism that is compatible with Git and are provided by GitHub[6]. The goal is for code changes to be reviewed before they are inserted into the mainline branch. A developer can take these changes and push them to a remote repository on GitHub. Before merging

[5]https://git-scm.com/
[6]https://github.com/

or rebasing a new feature in, project maintainers in GitHub can review, accept, or reject a change based on the diff of the master code branch and the branch of the incoming change. Reviewers can comment and vote on the change in the GitHub web user interface. If the pull request is approved, it can be included in the master branch. A rejected pull request can be abandoned by closing it or the creator can further refine it based on the comments given and submit it again for review.

2.3. Machine Learning Techniques

In this section, we will describe the machine learning classifiers adopted in this work. We used eight different classifiers: a generalized linear model (Logistic Regression), a tree-based classifier (Decision Tree), and six ensemble classifiers (Bagging, Random Forest, ExtraTrees, AdaBoost, GradientBoost, and XGBoost).

In the next sub-sections, we will briefly introduce the eight adopted classifiers and give the rationale for choosing them for this study.

Logistic regression [36] is one of the most frequently used algorithms in machine learning. In logistic regression, a collection of measurements (the counts of a particular issue) and their binary classification (pull request acceptance) can be turned into a function that outputs the probability of an input being classified as 1, or in our case, the probability of it being accepted.

Decision Tree [37] is a model that takes learning data and constructs a tree-like graph of decisions that can be used to classify a new input. The learning data is split into subsets based on how the split from the chosen variable improves the accuracy of the tree at the time. The decisions connecting the subsets of data form a flowchart-like structure that the model can use to tell the user how it would classify the input and how certain the prediction is perceived to be.

We considered two methods for determining how to split the learning data: GINI impurity and information gain. GINI tells the probability of incorrect classification of a random element from the subset that has been assigned a random class within the subset. Information gain tells how much more accuracy a new decision node would add to the tree if chosen. GINI was chosen because of its popularity and its resource efficiency.

Decision tree as a classifier was chosen because it is easy to implement and human-readable; also, decision trees can handle noisy data well because subsets without significance can be ignored by the algorithm that builds the tree. The classifier can be susceptible to overfitting, where the model becomes too specific to the training data and provides poor results when used with new input data. Overfitting can become a problem when trying to apply the model to a mode-generalized dataset.

Random Forest [38] is an ensemble classifier, which tries to reduce the risk of overfitting a decision tree by constructing a collection of decision trees from random subsets in the data. The resulting collection of decision trees is smaller in depth, has a reduced degree of correlation between the subset's attributes, and thus has a lower risk of overfitting.

When given input data to label, the model utilizes all the generated trees, feeds the input data into all of them, and uses the average of the individual labels of the trees as the final label given to the input.

Extremely Randomized Trees [39] builds upon the Random Forest introduced above by taking the same principle of splitting the data into random subsets and building a collection of decision trees from these. In order to further randomize the decision trees, the attributes by which the splitting of the subsets is done are also randomized, resulting in a more computationally efficient model than Random Forest while still alleviating the negative effects of overfitting.

Bagging [40] is an ensemble classification technique that tries to reduce the effects of overfitting a model by creating multiple smaller training sets from the initial set; in our study, it creates multiple decision trees from these sets. The sets are created by sampling the initial set uniformly and with replacements, which means that individual data points can appear in multiple training sets. The resulting trees can be used in labeling new input through a voting process by the trees.

AdaBoost [41] is a classifier based on the concept of boosting. The implementation of the algorithm in this study uses a collection of decision trees, but new trees are created with the intent of correctly labeling instances of data that were misclassified by previous trees. For each round of training, a weight is assigned to each sample in the data. After the round, all misclassified samples are given higher priority in the subsequent rounds. When the number of trees reaches a predetermined limit or the accuracy cannot be improved further, the model is finished. When predicting the label of a new sample with the finished model, the final label is calculated from the weighted decisions of all the constructed trees. As Adaboost is based on decision trees, it can be resistant to overfitting and be more useful with generalized data. However, Adaboost is susceptible to noise data and outliers.

Gradient Boost [42] is similar to the other boosting methods. It uses a collection of weaker classifiers, which are created sequentially according to an algorithm. In the case of Gradient Boost, as used in this study, the determining factor in building the new decision trees is the use of a loss function. The algorithm tries to minimize the loss function and, similarly to Adaboost, stops when the model has been fully optimized or the number of trees reaches the predetermined limit.

XGBoost [43] is a scalable implementation of Gradient Boost. The use of XGBoost can provide performance improvements in constructing a model, which might be an important factor when analyzing a large set of data.

3. Related Work

In this Section, we report on the most relevant works on pull requests.

3.1. Pull Request Process

Pull requests have been studied from different points of view, such as pull-based development [1, 2, 3], usage of real online resources [44], pull requests reviewer assignment [4], and acceptance process [1, 5, 6, 7].

Sometimes pull requests are submitted to edit the same lines of code resulting in a latent collaborative conflict. Therefore, pull request might compete, and overlap changes on the same line, increasing the mergte complexity [45].

Another investigated issue regarding pull requests is latency [46]. It is defined as a complex issue related to many independent variables such as the number of comments and the size of a pull request [47].

Considering the code style as an influencing factor for integrating pull requests, several code style criteria have generally revealed high divergence while several other criteria always indicated consistency. However, code style inconsistency between pull requests and the code would affect the process of merging them into the code [46].

Zampetti et al. [44] investigated how, why, and when developers refer to online resources in their pull requests. They focused on the context and real usage of online resources and how these resources have evolved over time. Moreover, they investigated the browsing purpose of online resources in pull request systems. Instead of investigating commit messages, they evaluated only the pull request descriptions, since generally, the documentation of a change aims at reviewing and possibly accepting the pull request [1].

Yu et al. [4] worked on pull requests' reviewer assignment in order to provide an automatic organization in GitHub that leads to an effort waste They proposed a reviewer recommender, which should predict highly relevant reviewers of incoming pull requests based on the textual semantics of each pull request and the social relations of the developers. They found several factors that influence pull requests latency such as size, project age, and team size. This approach reached a precision rate of 74% for the best recommendation and a recall rate of 71% for top-10 recommendations. However, the authors did not consider the aspect of code quality. The results are confirmed also by [6].

Recent studies investigated the factors that influence the acceptance and rejection of a pull request, such as the developer, the project, and the specific pull request itself [48]. Moreover, the size of the change, its perceived quality, and the context have an important role in the pull requests acceptance [49, 1, 50].

There is no difference in the treatment of pull-requests coming from the core team and from the community. Generally, the merging decision is postponed based on technical factors [51, 52]. Generally, pull requests that passed the build phase are merged more frequently [53].

Also, gender plays a role in the pull requests acceptance: comparing pull requests acceptance rates of contributions coming from men versus women, women's contributions are accepted more often than men's [54]. Pull requests are more likely to be merged if the developer has more social contributions in coding [55]. Moreover, the social connection between the pull request submitter and project manager affects the acceptance when a core team member is evaluating the pull request [49].

Considering geographical location impacts as factors for pull request acceptance, the highest pull request acceptance rates have been discovered in countries with no apparent similarities [48].

Integrators decide to accept a contribution after analyzing source code quality, code style, documentation, granularity, and adherence to project conventions [1]. Pull request's programming language had a significant influence on acceptance [5]. Higher acceptance was mostly found for Scala, C, C#, and R programming languages. Factors regarding developers are related to the acceptance process, such as the number and experience level of developers' [56], and the developers reputation who submitted the pull request [8].

Moreover, testing plays an important role in the acceptance of a pull request [2, 57], positively influencing the majority of contributions to be accepted (85%) [57].

Rejection of pull requests can increase when technical problems are not properly solved and if the number of forks also increases [56]. Other most important rejection factors are inexperience with pull requests, the complexity of contributions, the locality of the modified artifacts, and the project's policy contribution [6]. From the integrators perspective, social challenges needed to be addressed, for example, how to motivate contributors to keep working on the project and how to explain the reasons for rejection without discouraging them. From the contributors perspective, they found that it is important to reduce response time, maintain awareness, and improve communication [1].

3.2. Software Quality of Pull Requests

To the best of our knowledge, only a few studies have focused on the quality aspect of pull request acceptance [1, 2, 7].

Gousios et al. [1] investigated the pull-based development process focusing on the factors that affect the efficiency of the process and contribute to the acceptance of a pull request and the related acceptance time. They analyzed the GHTorrent corpus and other 291 projects. The results showed that the number of pull requests increases over time. However, the proportion of repositories using them is relatively stable. They also identified common driving factors that affect the lifetime of pull requests and the merging process. Based on their study, code reviews did not seem to increase the probability of acceptance, since 84% of the reviewed pull requests were merged. The results showed that only 10% of the pull requests are rejected based on their project process and quality requirements (called as process and tests). If the pull request does not follow the correct project conventions or if the test failed, the pull request is rejected.

In another work, Gousios et al. [2] conducted a survey aimed at characterizing the key factors considered in the decision-making process of pull request acceptance. Quality was revealed as one of the top priorities for developers. The most important acceptance factors they identified are targeted area importance, test cases, and code quality. However, the respondents specified quality differently from their respective perception, as *conformance*, *good available documentation*, and *contributor reputation*.

Developers consider quality aspect as a big challenge. Pull requests are generally accepted according to the success of the integration testing on all supported platforms. Regarding the factors considered to measure their quality, they are based on the developers' perception. They evaluate the non-functional characteristics of the changes to merge. Developers prefer understandable and elegant code, with good documentation. The code should provide clear added value to the project with a minimal impact. Moreover, the vast majority of the developers manually evaluate the code quality. However, asking which tool they used, developers ranked as first choice (75%) continuous integration such as Travis CI [7] and CloudBees [8] that allow running the test suites against new pull requests. Based on their results, the authors highlighted the need to efficiently automate the pull requests quality evaluation.

Kononenko et al. [7] investigated the pull request acceptance process in a commercial project addressing the quality of pull request reviews from the point of view of developers' perception. They applied data mining techniques on the projects GitHub repository in order to understand the nature of the merge and then conducted a manual inspection of the pull requests. They also investigated the factors that influence the merge time and outcome of pull requests such as pull request size and the number of people involved in the discussion of each pull request. Developers' experience and affiliation were significant factors in both models. Moreover, they report that developers generally associate the quality of a pull request with the quality of its description, complexity, and revertability. However, they did not evaluate the reasons for a pull request being rejected. These studies investigated the software quality of pull requests focusing on the trustworthiness of developers' experience and affiliation [7]. Moreover, these studies did not measure the quality of pull requests against a set of rules but based on their acceptance rate and developers' perception.

Trautsch et al. [58] investigated the usage of automated static analysis tools in open-source projects in the context of software evolution, also applying PMD rules. They measured the impact of PMD rules on software quality considering defect density as a proxy metric for external software quality. Moreover, they compared the data where PMD was and was not used and found a statistically significant difference in defect density. However, they did not analyze the acceptance or rejection of a pull request but only the quality of the code. This means they considered only accepted pull requests in their study.

Our work complements these works by analyzing the code quality of pull requests in popular open-source projects, considering code style violations and specifically PMD rules, as quality factors. Moreover, we investigated how the quality, specifically issues in the source code, affect the chance of a pull request being accepted when it is reviewed by a project maintainer. We measured code quality against a set of rules provided by PMD, one of the most frequently used open-source software tools for analyzing source code.

[7]https://travis-ci.org
[8]https://www.cloudbees.com

4. Case Study Design

We designed our empirical study as a case study based on the guidelines defined by Runeson and Höst [59]. In this Section, we describe the case study design, including the goal and the research questions, the study context, the data collection, and the data analysis procedure.

4.1. Goal and Research Questions

The goal of this work is to investigate the role of PMD issues in pull request acceptance. Accordingly, to meet our expectations, we formulated the goal as follows, using the Goal/Question/Metric (GQM) template [60]:

Purpose	Analyze
Object	the acceptance of pull requests
Quality	with respect to the presence of PMD issues
Viewpoint	from the point of view of developers
Context	in the context of Java projects

Based on the defined goal, we derived the following Research Questions ($\mathbf{RQ}_s$):

$\mathbf{RQ}_1$	What is the distribution of PMD issues violated by the pull requests in the analyzed software systems?
$\mathbf{RQ}_2$	Does the presence of PMD issues affect pull request acceptance?
$\mathbf{RQ}_3$	Do specific PMD issues affect pull request acceptance?

$\mathbf{RQ}_1$ aims at assessing the distribution of the PMD issues violated by pull requests in the analyzed software systems. We also took into account the distribution of PMD issues with respect to their priority level as assigned by PMD (P1-P5). These results will also help us to better understand the context of our study.

$\mathbf{RQ}_2$ aims at finding out whether the project maintainers in open-source Java projects consider quality issues in the pull request source code when they are reviewing it. If code quality issues affect the acceptance of pull requests, the question is what kind of PMD issues generally lead to the rejection of a pull request.

$\mathbf{RQ}_3$ aims at understanding if the presence of any specific PMD issue is more likely to result in the project maintainer rejecting the pull request. As an example, we expect that the pull request affected by well-known code smells or anti-patterns (e.g. god class and inappropriate intimacy) would be rejected because of low maintainability in the code.

4.2. Context

The projects for this study were selected using “criterion sampling” [61]. The criteria for selecting projects were as follows:

- Uses Java as its primary programming language
- Older than two years
- Had active development in last year
- Code is hosted on GitHub
- Uses pull requests as a mean for contributing to the code base
- Has more than 100 closed pull requests

Moreover, we tried to maximize diversity and representativeness considering a comparable number of projects with respect to project age, size, and domain, as recommended by Nagappan et al. [62].

We selected 28 projects according to these criteria. The majority, 22 projects, were selected from the Apache Software Foundation repository[9]. The repository proved to be an excellent source of projects that meet the criteria described above. It includes some of the most widely used software solutions, considered industrial and mature, due to the strict review and inclusion process required by the ASF. Moreover, the included projects have to keep on reviewing their code and follow a strict quality process[10].

The remaining six projects were selected based on the top Trending Java repositories list in GitHub [11]. GitHub provides a valuable source of data for the study of code reviews [63]. In the selection, we manually selected popular Java projects using the criteria mentioned before.

In Table 2, we report the list of the 28 projects that were analyzed along with the number of pull requests (“*#PR*”), the time frame of the analysis, and the size of each project (*“#LOC”).*

4.3. Data Collection

We first extracted all pull requests from each of the selected projects using the GitHub REST API v3 [12].

[9]http://apache.org

[10]https://incubator.apache.org/policy/process.html

[11]https://github.com/trending/java

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

Table 2: Selected projects

Project Owner/Name	**#PR**	**Time Frame**	**#LOC**
apache/any23	129	2013/12 - 2018/11	78,350
apache/dubbo	1,270	2012/02 - 2019/01	133,630
apache/calcite	873	2014/07 - 2018/12	337,430
apache/cassandra	182	2018/10 - 2011/09	411,240
apache/cxf	455	2014/03 - 2018/12	807,510
apache/flume	180	2012/10 - 2018/12	103,700
apache/groovy	833	2015/10 - 2019/01	396,430
apache/guacamole-client	331	2016/03 - 2018/12	65,920
apache/helix	284	2014/08 - 2018/11	191,830
apache/incubator-heron	2,190	2015/12 - 2019/01	207,360
hibernate/hibernate-orm	2,570	2010/10 - 2019/01	797,300
apache/kafka	5,520	2013/01 - 2018/12	376,680
apache/lucene-solr	264	2016/01 - 2018/12	1,416,200
apache/maven	166	2013/03 - 2018/12	10,780
apache/metamodel	198	2014/09 - 2018/12	64,800
mockito/mockito	726	2012/11 - 2019/01	57,400
apache/netbeans	1,020	2017/09 - 2019/01	6,115,770
netty/netty	4,120	2010/12 - 2019/01	275,970
apache/opennlp	330	2016/04 - 2018/12	136,540
apache/phoenix	203	2014/07 - 2018/12	366,580
apache/samza	1,470	2014/10 - 2018/10	129,280
spring-projects/spring-framework	1,850	2011/09 - 2019/01	717,960
spring-projects/spring-boot	3,070	2013/06 - 2019/01	348,090
apache/storm	2,860	2013/12 - 2018/12	359,900
apache/tajo	1,020	2014/03 - 2018/07	264,790
apache/vxquery	169	2015/04 - 2017/08	264,790
apache/zeppelin	3,190	2015/03 - 2018/12	218,950
openzipkin/zipkin	1,470	2012/06 - 2019/01	121,500
Total	**36,340**		**14,776,680**

For each pull request, we fetched the code from the pull request's branch and analyzed the code using PMD. The default Java rule set for PMD was used for the static analysis. All the pull requests contained code written in Java and we did not find pull requests written in any other language. Please note, that we considered only pull requests containing source code and therefore excluded pull requests containing changes not related to the source code like changes to documentation and images). We filtered the PMD issues added in the main branch to only include items introduced in the pull request. The filtering was done with the aid of a diff-file provided by GitHub API and compared the pull request branch against the master branch.

We identified whether a pull request was accepted or not by checking whether the pull request had been marked as merged into the master branch or whether the pull request had been closed by an event that committed the changes to the master branch. Other ways of handling pull requests within a project were not considered.

4.4. Data Analysis

The result of the data collection process was a CSV file reporting the dependent variable (pull request accepted or not) and the independent variables (number of PMD issues introduced in each pull request). Table 3 provides an example of the data structure we adopted in the remainder of this work.

Table 3: Example of data structure used for the analysis

		Dependent Variable	**Independent Variables**		
Project ID	**PR ID**	**Accepted PR**	**Rule1**	...	**Rule n**
Cassandra	ahkji	1	0		3
Cassandra	avfjo	0	0		2

For $\mathbf{RQ}_1$, we first calculated the total number of pull requests and the number of PMD issues present in each project. Moreover, we calculated the number of accepted and rejected pull requests. For each PMD issue, we calculated the number of occurrences, the number of pull requests, and the number of projects where it was found. Moreover, we calculated descriptive statistics (average, maximum, minimum, and standard deviation) for each PMD issue.

In order to understand if PMD issues affect pull request acceptance ($\mathbf{RQ}_2$), we first determined whether there is a significant difference between the expected frequencies and the observed frequencies in one or more categories. First, we computed the contingency matrix and we performed the χ^2 test. Then, we applied logistic regression and the six machine learning techniques, reported in Section 2.3, and compared the accuracy of each regressor.

We used the PMD issues as independent variables (predictors) to determine if a PR is accepted or rejected (dependent variable).

The reason for using multiple machine learning models lies in the fact that each model performs differently on the same data, based on its bias and variance. The bias tells us how much the model is paying attention to the training data - higher bias, less attention to the training data. The variance does the opposite - higher variance, higher attention is given to the training set, model over-fit the data and cannot generalize. As customary done in machine learning studies, in order to find a suitable model, we need to try multiple of them to find the right bias-variance trade-off [64]. The description of the used techniques and the rationale adopted to select each of them is reported in Section 2.3.

χ^2 test could be enough to answer our RQs. However, in order to support possible follow-up of the work, considering other factors such as LOC as an independent variable, machine learning techniques can provide much more accurate results.

In order to evaluate the importance of the different PMD issues in the acceptance of pull requests ($\mathbf{RQ}_3$), we first examined whether issues with a specific priority value affect the accuracy metrics of the prediction models. We used the same techniques as before but grouped all the PMD issues in each project into groups according to their priorities. The analysis was run separately for each project and each priority level (28 projects * 5 priority level groups) and the results were compared to the ones we obtained for RQ2. To further analyze the effect of issue priority, we also combined the PMD issues all the project into one data set and created models based on all available items with the same priority.

Then, we analyzed each PMD issue individually, applying the *drop-column importance* mechanism[13]. After training our baseline model with P number of features, we trained P number of new models and compared each of the new model's tested accuracy against the baseline model. Should a feature affect the accuracy of the model, the model trained with that feature dropped from the dataset would have a lower accuracy score than the

[13]https://explained.ai/rf-importance/

Table 4: Accuracy measures

Accuracy Measure	Formula
Precision	$\frac{TP}{FP+TP}$
Recall	$\frac{TP}{FN+TP}$
MCC	$\frac{TP*TN-FP*FN}{\sqrt{(FP+TP)(FN-TP)(FP+TN)(FN+TN)}}$
F-measure	$2*\frac{precision*recall}{precision+recall}$

TP: True Positive; TN: True Negative; FP: False Positive; FN: False Negative

baseline model. The more the accuracy of the model drops with a feature removed, the more important that feature is to the model when classifying pull requests as accepted or rejected.

For **RQ**$_2$ and **RQ**$_3$, once each model was trained, we confirmed that the predictions about pull request acceptance made by the model were accurate (**Accuracy Comparison**). To determine the accuracy of a model, 5-fold cross-validation was used. The data set was randomly split into five parts. A model was trained five times, each time using four parts for training and the remaining part for testing the model. We calculated accuracy measures (precision, recall, Matthews correlation coefficient, and F-measure) for each model (see Table 4) and then combined the accuracy metrics from each fold to produce an estimate of how well the model would perform.

We started by calculating the commonly used metrics, including F-measure, precision, recall, and the harmonic average of the latter two. Precision and recall are metrics that focus on the true positives produced by the model. Powers [65] argues that these metrics can be biased and suggests that a contingency matrix should be used to calculate additional metrics to help understand how negative predictions affect the accuracy of the constructed model. Using the contingency matrix, we calculated the model's Matthew Correlation Coefficient (MCC), which suggests as the best way to reduce the information provided by the matrix into a single probability describing the model's accuracy [65].

For each classifier to easily gauge the overall accuracy of the machine learning algorithm in a model [66], we calculated the Area Under The Receiver Operating Characteristic (AUC). For the AUC measurement, we calculated Receiver Operating Characteristics (ROC) and used these to find out the AUC ratio of the classifier, which is the probability of the classifier ranking a randomly chosen positive higher than a randomly chosen negative one.

To confirm the results of the automated application of the machine learning models, and gain more qualitative insights on the results, we manually inspected 10% of the accepted PR and 10% of the non accepted PRs of each project, to identify if PRs were rejected or accepted due to quality issues. The validation was performed by two of the authors who individually analyzed all candidate PRs, marking as *"True"* PRs mentioning quality issues in the rejection message and *"False"* otherwise. Moreover, the two authors also reported if the PRs were aimed at fixing a quality issue (e.g. removing issues highlighted by static analysis tools), so as to understand the importance of quality issues from the maintainers' point of view. A total of 37 PRs were classified differently by the two authors (1% of the manually inspected PRs). An open discussion, involving the third author, helped to resolve conflicts and reach a consensus on the classification.

4.5. Replicability

In order to allow our study to be replicated, we have published the complete raw data in the replication package[14].

Please note that the CSV file includes only pull requests containing Java code, while we excluded pull requests including other types of changes (edits to the documentation, HTML, CSS, and others).

5. Results

RQ_1. What is the distribution of PMD issues violated by the pull requests in the analyzed software systems?

For this study, we analyzed 36,344 pull requests violating 253 PMD rules contained more than 4.7 million times (Table 5) in the 28 analyzed projects. We found that 19,293 pull requests (53.08%) were accepted and 17,051 pull requests (46.92%) were rejected. The distribution of the PMD issues differs greatly among the pull requests. For example, the projects *Cassandra* and *Phoenix* contain a relatively large number of PMD issues compared to the number of pull requests, while *Groovy*, *Guacamole*, and *Maven* have a relatively small number of PMD issues.

[14]https://figshare.com/s/d47b6f238b5c92430dd7

Table 7 reports the number of PMD issues ("*#PMD issues*") and their number of occurrences ("*#PMD issues occurrences*") grouped by priority level ("*Priority*").

Taking into account the priority level of the issues, the vast majority of the violated rules (197 out 253) are classified with priority level 3, while the remaining ones (56) are equally distributed among levels 1, 2, and 4. None of the projects we analyzed had any issues rated as priority level 5.

Looking at the PMD rules that could play a role in pull request acceptance or rejection, 241 of the 253 PMD rule types (95%) are present in both cases, while 11 are found only in cases of rejection and 1 only in cases of acceptance (Table 6 and Table 7). However, these 11 rules present only in rejected PRs are violated only a few times. As an example, the rules often violated in the whole dataset ("LocalHomeNaming-Convention" and "LocalInterfaceSession-NamingConvention") are violated only 12 times in the rejected PRs. It is also interesting that all these 11 rules that were violated only in rejected PRs, 7 of them have a medium priority (Change recommended), and 4 a low priority (Change optional).

We discovered that 88 PMD rules have a diffusion rate higher than 60% in the case of acceptance and 127 have a diffusion rate higher than 60% in the case of rejection. The remaining 38 are equally distributed.

Table 8 and Table 9 present descriptive statistic including average ("*Avg.*"), maximum ("*Max*"), minimum ("*Min*"), and standard deviation ("*Std. dev.*") of the twenty most recurrent PMD issues. Moreover, we include the priority of each PMD rule ("*Priority*"), the sum of issue rows of that rule type was found in the issues master table ("*# Total occurrences*"), and the number of projects in which the specific rule has been violated (*"#Project"*). The complete list is available in the replication package (Section 4.5).

Table 5: Distribution of pull requests (PR) and PMD issues in the selected projects - (RQ$_1$)

Project Name	**#PR**	**#PMD issues occurrences**	**% Acc.**	**% Rej.**
apache/any23	129	11,573	90.70	9.30
apache/calcite	873	104,533	79.50	20.50
apache/cassandra	182	153,621	19.78	80.22
apache/cxf	455	62,564	75.82	24.18
apache/dubbo	1,270	169,751	52.28	47.72
apache/flume	180	67,880	60.00	40.00
apache/groovy	833	25,801	81.39	18.61
apache/guacamole-client	331	6,226	92.15	7.85
apache/helix	284	58,586	90.85	9.15
apache/incubator-heron	2,191	138,706	90.32	9.68
apache/kafka	5,522	507,423	73.51	26.49
apache/lucene-solr	264	72,782	28.41	71.59
apache/maven	166	4,445	32.53	67.47
apache/metamodel	198	25,549	78.28	21.72
apache/netbeans	1,026	52,817	83.14	16.86
apache/opennlp	330	21,921	82.73	17.27
apache/phoenix	203	214,997	9.85	90.15
apache/samza	830	96,915	69.52	30.48
apache/storm	2,863	379,583	77.96	22.04
apache/tajo	1,020	232,374	67.94	32.06
apache/vxquery	169	19,033	30.77	69.23
apache/zeppelin	3,194	408,444	56.92	43.08
hibernate/hibernate-orm	2,573	490,905	16.27	83.73
mockito/mockito	726	57,345	77.41	22.59
netty/netty	4,129	597,183	15.84	84.16
openzipkin/zipkin	1,474	78,537	73.00	27.00
spring-projects/spring-boot	3,076	156,455	8.03	91.97
spring-projects/spring-framework	1,850	487,197	15.68	84.32
Total	**36,344**	**4,703,146**	**19,293**	**17,051**

Table 7: Distribution of PMD issues in pull requests - (RQ$_1$)

Priority	**#PMD Issue Types**	**# PMD issues occurrences**	**# PMD issue types (in Accepted PR)**	**#PMD issue types (in Rejected PR)**
4	18	85,688	14	18
3	197	4,488,326	191	197
2	22	37,492	21	21
1	16	91,640	21	16
All	253	4,703,146	243	253

Table 6: PMD rules that were violated only in accepted or rejected pull requests - (RQ_1)

Rules violated only in rejected PRs			
Rule	**Priority**	**#occur.**	**Rule description**
AvoidCallingFinalize	3	9	Avoid calling finalize() explicitly
DontCallThreadRun	4	4	Dont call Thread.run() explicitly, use Thread.start()
DontUseFloatTypeForLoop-Indices	3	1	Dont use floating point for loop indices. If you must use floating point, use double.
FinalizeOverloaded	3	2	Finalize methods should not be overloaded
FinalizeShouldBeProtected	3	3	If you override finalize(), make it protected
LocalHomeNaming-Convention	4	12	The Local Home interface of a Session EJB should be suffixed by LocalHome
LocalInterfaceSession-NamingConvention	4	12	The Local Interface of a Session EJB should be suffixed by Local
NonCaseLabelInSwitch-Statement	3	5	A non-case label was present in a switch statement
ReplaceEnumerationWith-Iterator	3	10	Consider replacing this Enumeration with the newer java.util.Iterator
StringBufferInstantiation-WithChar	4	1	Do not instantiate a StringBuffer or String-Builder with a char
UselessOperationOn-Immutable	3	1	An operation on an Immutable object (String, BigDecimal or BigInteger) wont change the object itself

Rules violated only in accepted PRs			
Rule	**Priority**	**#occur.**	**Rule description**
MethodWithSameNameAs-EnclosingClass	3	8	Classes should not have non-constructor methods with the same name as the class

Table 8: Descriptive statistics (the 15 most recurrent PMD issues) - Priority, number of occurrences (#occur.), number of Pull Requests (#PR) and number of projects (#prj.) - (RQ$_1$)

PMD issue	**Priority**	**#occur.**	**#PR**	**#prj.**
LawOfDemeter	4	1,089,110	15,809	28
MethodArgumentCouldBeFinal	4	627,688	12,822	28
CommentRequired	4	584,889	15,345	28
LocalVariableCouldBeFinal	4	578,760	14,920	28
CommentSize	4	253,447	11,026	28
JUnitAssertionsShouldIncludeMessage	4	196,619	6,738	26
BeanMembersShouldSerialize	4	139,793	8,865	28
LongVariable	4	122,881	8,805	28
ShortVariable	4	112,333	7,421	28
OnlyOneReturn	4	92,166	7,111	28
CommentDefaultAccessModifier	4	58,684	5,252	28
DefaultPackage	4	42,396	4,201	28
ControlStatementBraces	4	39,910	2,689	27
JUnitTestContainsTooManyAsserts	4	36,022	4,954	26
AtLeastOneConstructor	4	29,516	5,561	28

Table 9: Descriptive statistics (the 15 most recurrent PMD issues) - Average (Avg.), Maximum (Max), Minimum (Min) and Standard Deviation (std. dev.) - (RQ$_1$)

PMD issue	**Avg**	**Max**	**Min**	**Std. dev.**
LawOfDemeter	38,896.785	140,870	767	40,680
MethodArgumentCouldBeFinal	22,417.428	105,544	224	25,936
CommentRequired	20,888.892	66,798	39	21,979
LocalVariableCouldBeFinal	20,670	67394	547	20,461
CommentSize	9,051.678	57,074	313	13,818
JUnitAssertionsShouldIncludeMessage	7,562.269	38,557	58	10822
BeanMembersShouldSerialize	4,992.607	22,738	71	5,597
LongVariable	4,388.607	19,958	204	5,096
ShortVariable	4,011.892	21,900	26	5,240
OnlyOneReturn	3,291.642	14,163	42	3,950
CommentDefaultAccessModifier	2,095.857	12,535	6	2,605
DefaultPackage	1,514.142	9,212	2	1,890
ControlStatementBraces	1,478.148	11,130	1	2,534
JUnitTestContainsTooManyAsserts	1,385.461	7,888	7	1,986
AtLeastOneConstructor	1,054.142	6,514	21	1,423

Summary of RQ$_1$

Among the 36,344 analyzed pull requests, we discovered 253 different PMD rules violated more than 4.7 million times. Nearly half of the pull requests had been accepted and the other half had been rejected. 242 of the 253 PMD rules were violated in both cases. The vast majority of these PMD rules (197) have priority level 3.

Table 10: Contingency matrix - (RQ_2)

	PMD issues	**No PMD issues**
PR accepted	10,563	8,558
PR rejected	11,228	5,528

Table 11: Model reliability - (RQ_2)

	Average between 5-fold validation models							
Accuracy Measure	**L. R.**	**D. T.**	**Bagg.**	**R. F.**	**E. T.**	**A. B.**	**G. B.**	**XG.B.**
AUC	50.91	50.12	49.83	50.75	50.54	51.30	50.64	50.92
Precision	49.53	48.40	48.56	49.33	49.20	48.74	49.30	49.20
Recall	62.46	47.45	47.74	48.07	47.74	51.82	41.80	41.91
MCC	0.02	-0.00	0.00	0.01	0.01	0.00	0.00	-0.00
F-Measure	0.55	0.47	0.47	0.48	0.48	0.49	0.44	0.44

RQ_2. Does the presence of PMD issues affect pull request acceptance?

To answer this question, we first calculated the contingency matrix (Table 10), and χ^2 test. Then, we trained the logistic regressor and the six machine learning models for each project. Once we had all the models trained, we tested them and calculated the accuracy measures described in Table 4 for each model. We then averaged each of the metrics from the classifiers for the different techniques.

As we can see from the contingency matrix (Table 10), there is no significant difference between accepted and rejected pull requests in terms of the presence of PMD issues. Moreover, this is confirmed by the low result of the χ^2 test (0.12).

The overall results for the application of the logistic regression and the machine learning models on all the data are presented in Table 11. The results for each model trained on each of the 28 projects are available in the replication package (Section 4.5). The averaging provided us with an estimate of how accurately we could predict whether maintainers accepted the pull request based on the number of different PMD issues it has.

Results are also confirmed in the analysis of each project independently. None of the 28 projects show different behaviour in terms of the presence of PMD issues in the acceptance or rejection of pull requests.

The results of this analysis are presented in Table 12. For reasons of space, we report only the 30 most frequently violated PMD rules. The table also contains the number of distinct PMD rules that the items of the project contained. The rule count can be interpreted as the number of different types of items found.

As depicted in Figure 1, for every prediction method, the AUC for almost

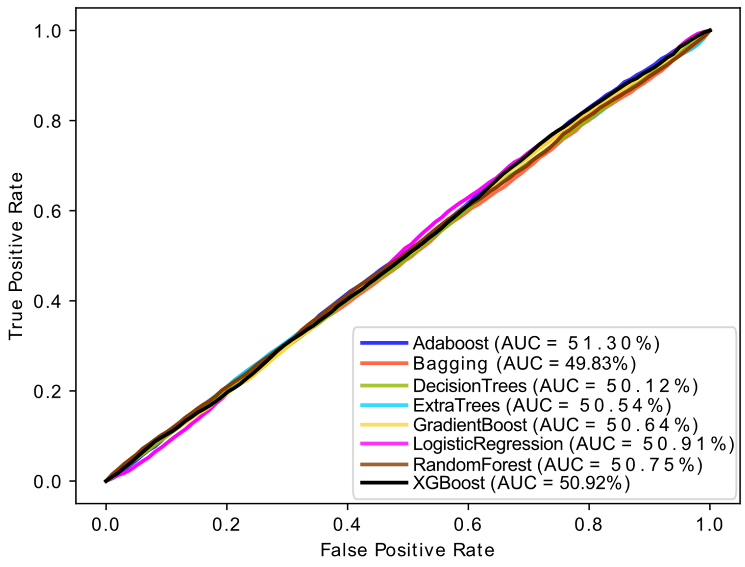

Figure 1: ROC Curves (average between 5-fold validation models) - (RQ$_2$)

Table 12: Summary of the quality rules related to pull request acceptance - (RQ$_2$ and RQ$_3$)[a]

[a]: NOTE FOR REVIEWERS: The current template has smaller margins. This table reflects the actual margins of the standard 2-columns JSS template.

Rule ID	**Prior.**	**#Prj.**	**#Occur.**	**Importance (%)**							
				A.B.	**Bagg.**	**D.T.**	**E.T.**	**G.B.**	**L.R.**	**R.F.**	**XG.B.**
LawOfDemeter	4	28	1,089,110	0.12	-0.51	0.77	-0.74	-0.29	-0.09	-0.66	0.02
MethodArgumentCouldBeFinal	4	28	627,688	-0.31	0.38	0.14	0.03	-0.71	-0.25	0.24	0.07
CommentRequired	4	28	584,889	-0.25	-0.11	0.07	-0.30	-0.47	-0.17	0.58	-0.31
LocalVariableCouldBeFinal	4	28	578,760	-0.13	-0.20	0.55	0.28	0.08	-0.05	0.61	-0.05
CommentSize	4	28	253,447	-0.24	-0.15	0.49	-0.08	-0.17	-0.05	-0.10	0.05
JUnitAssertionsShouldIncludeMessage	4	26	196,619	-0.41	-0.84	0.22	-0.28	-0.19	-0.10	-0.75	0.14
BeanMembersShouldSerialize	4	28	139,793	-0.33	-0.09	-0.03	-0.38	-0.37	0.17	0.26	0.07
LongVariable	4	28	122,881	0.08	-0.19	-0.02	-0.25	-0.28	0.08	0.24	0.02
ShortVariable	4	28	112,333	-0.51	-0.24	0.09	-0.04	-0.04	0.07	-0.25	-0.54
OnlyOneReturn	4	28	92,166	-0.69	-0.03	0.02	-0.25	-0.08	-0.06	0.06	-0.13
CommentDefaultAccessModifier	4	28	58,684	-0.17	-0.07	0.30	-0.41	-0.25	0.23	0.18	-0.10
DefaultPackage	4	28	42,396	-0.37	-0.05	0.20	-0.23	-0.93	0.10	-0.01	-0.54
ControlStatementBraces	4	27	39,910	-0.89	0.09	0.58	0.29	-0.37	-0.03	0.08	0.25
JUnitTestContainsTooManyAsserts	4	26	36,022	0.40	0.22	-0.25	-0.33	0.01	0.16	0.10	-0.17
AtLeastOneConstructor	4	28	29,516	0.00	-0.29	-0.06	-0.18	-0.19	-0.07	0.15	-0.22
UnnecessaryFullyQualifiedName	4	27	27,402	0.00	0.08	0.25	-0.05	0.00	0.00	0.26	-0.11
AvoidDuplicateLiterals	4	28	27,224	-0.20	0.05	0.33	-0.28	0.12	0.20	0.09	0.07
SignatureDeclareThrowsException	4	27	26,188	-0.18	-0.10	0.04	-0.13	-0.05	0.11	0.33	-0.17
AvoidInstantiatingObjectsInLoops	3	28	25,344	-0.05	0.07	0.43	-0.14	-0.27	-0.13	0.52	-0.07
FieldNamingConventions	3	28	25,062	0.09	0.00	0.16	-0.21	-0.10	-0.01	0.07	0.19

all of the trained models are hovering around 50%. Therefore, overall code quality does not appear to be a factor in determining whether a pull request is accepted or rejected.

There were some projects that showed some moderate success, but these can be dismissed as outliers.

The results might suggest that machine learning could not be the most suitable technique. However, also χ^2 test on the contingency matrix (0.12) (Table 10) confirms the above result that the presence of PMD issues does not affect pull request acceptance (which means that PMD issues and pull request acceptance are mutually independent).

RQ_3. Do specific PMD issues affect pull request acceptance?

Grouping PMD issues by priority level did not provide any improvements over the results of RQ1. The contingency matrix for each priority shows no significant differences between accepted and rejected pull requests in terms of the presence of PMD issues. Also, in this case, the result of χ^2 test is very low, ranging from 0.08 to 0.14. The same results are confirmed by the application of logistic regression and the selected machine learning models. This trend is confirmed for each project analyzed independently and analyzing the projects together.

Also considering the PMD issues individually, results show a very low importance for all the PMD rules (Table 12). No rule has an importance higher than 1%.

In table 12 we show the importance of the 20 most common quality rules when comparing the baseline model accuracy with a model that has the specific quality rule dropped from the feature set.

The manual inspection of the commits provided very interesting results. Out of 1,705 rejected PRs and 1,929 accepted PRs we inspected, none was rejected for quality reasons. However, it is interesting to note that 36 PRs were submitted with the purpose of fixing quality issues detected by static analysis tools and 16 of them were accepted, while the remaining 20 were not. For these 20 rejected PRs the maintainers of the projects often replied that the fixes were not needed, not useful, or only cosmetic changes. In five of the rejected PRs containing quality issue fixes, the maintainers made a cumulative PR fixing all the issues reported by the rejected PRs. [15].

Summary of RQ$_2$ and RQ$_3$
Looking at the results we obtained from the analysis using statistical and machine learning techniques (χ^2 0.12 and AUC 50% on average), PMD issues do not appear to influence pull request acceptance. The manual inspection also confirms that quality issues are never considered as a criteria for the acceptance of PRs.

6. Discussion

In this Section, we will discuss the results obtained according to the RQs and present possible practical implications from our research.

The analysis of the pull requests in 28 well-known Java projects shows that quality flaws detected by PMD, including code smells and anti-patterns, are frequently present in pull requests. Several pull requests contain or create god classes, speculative generality (named "Law of Demeter"), duplicated code, long methods, and many other issues generally considered harmful by several empirical studies [25], [67], [27]. A possible reason for this could be that developers prefer to accept a pull request implementing a new feature or fixing a bug over rejecting it to maintain a certain quality level. Another reason might be the project maintainers' lack of knowledge of code smells, anti-patterns, and other coding rules proposed by PMD.

As a result of this work, PMD issues are not a driver for the acceptance or the rejection of pull requests. All the projects reported the same behaviour, and all the rules had very low importance in the acceptance or rejection of pull requests.

PMD recommends manual customization of the set of rules instead of using the out-of-the-box rule set and selecting the rules that developers should consider, in order to maintain a certain level of quality. However, since we analyzed all the rules detected by PMD, no rule would be helpful and any customization would be useless in terms of being able to predict the software quality in code submitted to a pull request.

[15]
(PRs id 716dfd1c54d1112fd5f6b4339bdd59784e867351,
1c94e5cd9d81ae18861b375056db74f5dfe36597 for Zeppelin and PRs id
b9cbe603539e01111855b03018e79b320ada4228,
856af40924b920f68861090d20049eda72703697,
5353660de45eac91dc1dcee7a24d8dc3d2b4555b for Netty)

The result cannot be generalized to all the open-source and commercial projects, as we expect some projects could enforce quality checks to accept pull requests. Some tools, such as SonarQube (one of the main competitors of PMD), recently launched a new feature allowing developers to check the code quality before submitting a pull request. Even if maintainers are not sensitive to the quality of the code to be integrated into their projects, at least based on the rules detected by PMD, the adoption of pull request quality analysis tools such as SonarQube or the usage of PMD before submitting a pull request will increase the quality of their code; increasing the overall software maintainability and decreasing the fault proneness that could be increased from the injection of some PMD issues (see Table 1).

During the manual inspection of the PRs, we noticed that 11 projects (Any23, Groovy, Hibernate-orm, Incubator-heron, Kafka, Netty, Opennlp, Samza, Spring-boot, Storm, and Zeppelin) adopted Checkstyle to check conformity to coding standards. However, it is interesting to see that only a very limited amount of PRs contains fixes to the issues raised by Checkstyle, also confirming the results obtained in this work.

The results complement those obtained by Soares et al. [6] and Calefato et al. [8], namely, that the reputation of the developer might be more important than the quality of the developed code. The main implication for practitioners, and especially for those maintaining open-source projects, is the realization that they should pay more attention to software quality. Pull requests are a very powerful instrument, which could provide great benefits if they were used for code reviews as well. Researchers should also investigate whether other quality aspects might influence the acceptance of pull requests.

Our findings can have implications for project contributors, core teams and researchers:

Contributors. Code Quality does not seem to be a key driver for the acceptance of PRs. We still recommend contributors to write clean code and to pay attention to coding standards and quality rules, however, prospective contributors should also pay attention to other criteria, such as having a complete test-suite [1] and clear documentation when applicable. Moreover, contributors should also try to submit PRs that can be merged without changes, especially when they are impacting on several files [1][55].

Core Teams are interested to integrate PRs that provide value to their tool. However, core team members should consider the adoption of an automated static analysis tool, integrated into their CD/CI pipeline and require contributors to validate their code against the rules they select.

Researchers should disseminate more the culture of quality in open-source projects, especially related to the development of clean and high-quality code, removing quality issues such as code smells and anti-patterns.

7. Threats to Validity

In this Section, we introduce the threats to validity and the different tactics we adopted to mitigate them.

Construct Validity. This threat concerns the relationship between theory and observation due to possible measurement errors. Above all, we relied on PMD, one of the most used software quality analysis tool for Java. However, even PMD is widely used in industry, we did not find any evidence or an empirical study assessing its detection accuracy. Therefore, we cannot exclude the presence of false positives or false negatives in the detected PMD issues.

We extracted the code submitted in pull requests using the GitHub API[10]. However, we identified whether a pull request was accepted or not by checking whether the pull request had been marked as merged into the master branch or whether the pull request had been closed by an event that committed the changes to the master branch. Other ways of handling pull requests within a project were not considered and, therefore, we are aware that there is a limited possibility that some maintainers could have integrated the pull request code into their projects manually, without marking the pull request as accepted.

Internal Validity. This threat concerns internal factors related to the study that might have affected the results. In order to evaluate the code quality of pull requests, we applied the rules provided by PMD, which is one of the most widely used static code analysis tools for Java on the market. We are aware that the presence or the absence of a PMD issue cannot be the perfect predictor for software quality, and other rules or metrics detected by other tools could have brought to different results. However, the selection of PMD was performed as it allows to detect code smells, anti-patterns, security flaws, and coding style violations without the need of compiling the code. Other tools could be more specific for other purposes. As an example, the usage of DECOR [68] would have allowed capturing more code smells, but not enabled to capture security flaws. We are also aware that code quality is a general term. In this paper, we refer to the quality hindered by the PMD rules. However, code quality goes beyond these observed metrics. One of the major metrics for code quality is the number of defects (this could be observed tracking future commits over the lines changed by the pull request), but other metrics such as having or not test cases in the pull request or the test coverage of the pull request are also indicators of quality and might have yielded to different results.

External Validity. This threat concerns the generalizability of the results. We selected 28 projects; 22 of them were from the Apache Software Foundation, which incubates only certain systems that follow specific and strict quality rules and the remaining six projects were selected from the trending Java repositories list provided by GitHub. In the selection, we preferred projects that are considered ready for production environments and are using pull requests as a way of taking in contributions. Our case study was not based only on one application domain. This was avoided since we aimed to find general mathematical models for the prediction of the number of bugs in a system. Choosing only one domain or a very small number of application domains could have been an indication of the non-generality of our study, as only prediction models from the selected application domain would have been chosen. The selected projects stem from a very large set of application domains, ranging from external libraries, frameworks, and web utilities to large computational infrastructures. The application domain was not an important criterion for the selection of the projects to be analyzed, but at any rate, we tried to balance the selection and pick systems from as many contexts as possible. However, we are aware that other projects could have enforced different quality standards, and could use different quality checks before accepting pull requests. Furthermore, we are considering only open-source projects, and we cannot speculate on industrial projects, as different companies could have different internal practices. Moreover, we also considered only Java projects. The replication of this work on different languages and different projects may bring to different results. Based on our dataset, only eleven projects contained the vast majority of the pull requests (80%)". Moreover, none of the projects we analyzed had any issues rated as priority level 5.

Conclusion Validity. This threat concerns the relationship between the treatment and the outcome. In our case, this threat could be represented by the analysis method applied in our study. We reported the results considering descriptive statistics. Moreover, instead of using only logistic regression, we compared the prediction power of different classifiers to reduce the bias of the low prediction power that one single classifier could have. We do not exclude the possibility that other statistical or machine learning approaches such as deep learning might have yielded similar or even better accuracy than our modeling approach. However, considering the extremely low importance of each PMD issue and its statistical significance, we do not expect to find big differences applying other types of classifiers.

8. Conclusion

Pull requests are one of the most common code review mechanisms. Previous works reported 84% of pull requests to be accepted based on the trustworthiness of the developers [2, 8]. However, we believe that open-source maintainers are also considering the code quality when accepting or rejecting pull requests.

To verify this statement, we analyzed the code quality of pull requests using PMD. It is one of the most widely used static code analysis tools, which can detect different types of quality flaws in the code, including design flaws, code smells, security vulnerability, and potential bugs. We used PMD because it can detect a good number of quality flaws that have been empirically considered harmful by several works. Examples of these quality flaws are God Class, High Cyclomatic Complexity, Large Class, and Inappropriate Intimacy.

We applied basic statistical techniques, but also eight machine learning classifiers to understand if it is possible to predict if a pull request could be accepted or not based on the presence of a set of quality flaws detected by PMD in the pull request code. Moreover, we manually validated the results, analyzing 1,705 rejected and 1,905 accepted PRs. Of the 36,344 pull requests we analyzed in 28 well-known Java projects, nearly half had been accepted and the other half rejected. 243 of the 253 PMD quality flaws were present in each case.

Unexpectedly, the presence of PMD quality flaws of any type in the pull request code does not influence the acceptance or rejection of pull requests at all. Therefore, **the quality flaws in the code submitted in a pull request, including code smells and anti-patterns, do not influence at all the acceptance or PRs**. The same results are verified in all the 28 projects independently. Moreover, also merging all the data as a single large data-set confirmed the result.

Our results complement the conclusions derived by Gousios et al. [2] and Calefato et al. [8], who report that the reputation of the developer submitting the pull request is one of the most important acceptance factors.

We are aware that code quality is a broad and general term and that other aspects such as the test coverage of pull requests or the number of bugs generated by the code in pull requests could bring different results. However, these studies might deserve a future work. We are planning to investigate whether qualities and metrics detected by other tools might affect the acceptance of pull requests, analyzing different projects written in different languages so as to define more accurate quality models [69]. Other tools such as SonarQube might be consider so as to understand if their rules might complement PMD ones. For this purpose, we are planning to follows approaches similar to [70][71][72][73][74] in the context of pull request, also considering other publicly available datasets such as [75][76] and different methodologies such as observational studies [77].

We will also investigate how to raise awareness in the open-source community that code quality should also be considered when accepting pull requests.

Moreover, we will try to understand how harmful developers perceive the different PMD quality flaws, including code smells and anti-patterns, to qualitatively assess over these violations. Another important factor to be considered is the developers' personality as a possible influence on the acceptance of the pull request [78].

References

[1] G. Gousios, M. Pinzger, A. van Deursen, An exploratory study of the pull-based software development model, in: 36th International Conference on Software Engineering, ICSE 2014, pp. 345–355.

[2] G. Gousios, A. Zaidman, M. Storey, A. van Deursen, Work practices and challenges in pull-based development: The integrator's perspective, in: 37th IEEE International Conference on Software Engineering, volume 1, pp. 358–368.

[3] E. v. d. Veen, G. Gousios, A. Zaidman, Automatically prioritizing pull requests, in: 12th Working Conference on Mining Software Repositories, pp. 357–361.

[4] Y. Yu, H. Wang, G. Yin, C. X. Ling, Reviewer recommender of pull-requests in github, in: IEEE International Conference on Software Maintenance and Evolution, pp. 609–612.

[5] M. M. Rahman, C. K. Roy, An insight into the pull requests of github, in: 11th Working Conference on Mining Software Repositories, MSR 2014, pp. 364–367.

[6] D. M. Soares, M. L. d. L. Jnior, L. Murta, A. Plastino, Rejection factors of pull requests filed by core team developers in software projects with

high acceptance rates, in: 14th International Conference on Machine Learning and Applications (ICMLA), pp. 960–965.

[7] O. Kononenko, T. Rose, O. Baysal, M. Godfrey, D. Theisen, B. de Water, Studying pull request merges: A case study of shopify's active merchant, in: 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '18, pp. 124–133.

[8] F. Calefato, F. Lanubile, N. Novielli, A preliminary analysis on the effects of propensity to trust in distributed software development, in: 2017 IEEE 12th International Conference on Global Software Engineering (ICGSE), pp. 56–60.

[9] V. Lenarduzzi, A. Sillitti, D. Taibi, A survey on code analysis tools for software maintenance prediction, in: 6th International Conference in Software Engineering for Defence Applications, Springer International Publishing, 2020, pp. 165–175.

[10] M. Beller, R. Bholanath, S. McIntosh, A. Zaidman, Analyzing the state of static analysis: A large-scale evaluation in open source software, in: 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 1, pp. 470–481.

[11] M. Fowler, K. Beck, Refactoring: Improving the design of existing code, Addison-Wesley Longman Publishing Co., Inc. (1999).

[12] W. J. Brown, R. C. Malveau, H. W. S. McCormick, T. J. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis: Refactoring Software, Architecture and Projects in Crisis, John Wiley and Sons, 1998.

[13] M. Lanza, R. Marinescu, S. Ducasse, Object-Oriented Metrics in Practice, Springer-Verlag, Berlin, Heidelberg, 2005.

[14] F. Khomh, M. Di Penta, Y. Gueheneuc, An exploratory study of the impact of code smells on software change-proneness, in: 2009 16th Working Conference on Reverse Engineering, pp. 75–84.

[15] S. Olbrich, D. S. Cruzes, V. Basili, N. Zazworka, The evolution and impact of code smells: A case study of two open source systems, in: 2009 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 390–400.

[16] M. D'Ambros, A. Bacchelli, M. Lanza, On the impact of design flaws on software defects, in: 2010 10th International Conference on Quality Software, pp. 23–31.

[17] F. Fontana Arcelli, S. Spinelli, Impact of refactoring on quality code evaluation, in: Proceedings of the 4th Workshop on Refactoring Tools, WRT '11, pp. 37–40.

[18] M. Tufano, F. Palomba, G. Bavota, M. DiPenta, R. Oliveto, A. DeLucia, D. Poshyvanyk, There and back again: Can you compile that snapshot?, Journal of Software: Evolution and Process 29 (2017) e1838.

[19] V. Lenarduzzi, T. Besker, D. Taibi, A. Martini, F. A. Fontana, A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools, 1904.12538, arXiv (2020).

[20] W. H. Brown, R. C. Malveau, H. W. S. McCormick, T. J. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, New York, NY, USA, 1st edition, 1998.

[21] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Trans. Softw. Eng. 20 (1994) 476–493.

[22] J. Al Dallal, A. Abdin, Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review, IEEE Transactions on Software Engineering 44 (2018) 44–69.

[23] T. J. McCabe, A complexity measure, IEEE Trans. Softw. Eng. 2 (1976) 308–320.

[24] W. Li, R. Shatnawi, An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution, J. Syst. Softw. 80 (2007) 1120–1128.

[25] D. I. K. Sjberg, A. Yamashita, B. C. D. Anda, A. Mockus, T. Dyb, Quantifying the effect of code smells on maintenance effort, IEEE Transactions on Software Engineering 39 (2013) 1144–1156.

[26] A. Yamashita, Assessing the capability of code smells to explain maintenance problems: An empirical study combining quantitative and qualitative data, Empirical Softw. Engg. 19 (2014) 1111–1143.

[27] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, A. D. Lucia, On the diffuseness and the impact on maintainability of code

smells: A large scale empirical investigation, Empirical Softw. Engg. 23 (2018) 1188–1221.

[28] F. Khomh, M. Di Penta, Y. Gueheneuc, An exploratory study of the impact of code smells on software change-proneness, in: 2009 16th Working Conference on Reverse Engineering, pp. 75–84.

[29] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, F. Khomh, M. Zulkernine, Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults, Empirical Softw. Engg. 21 (2016) 896–931.

[30] S. M. Olbrich, D. S. Cruzes, D. I. K. Sjberg, Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems, in: 2010 IEEE International Conference on Software Maintenance, pp. 1–10.

[31] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, M. Shaw, Building empirical support for automated code smell detection, in: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '10, pp. 8:1–8:10.

[32] N. Zazworka, M. A. Shaw, F. Shull, C. Seaman, Investigating the impact of design debt on software quality, in: Proceedings of the 2Nd Workshop on Managing Technical Debt, MTD '11, pp. 17–23.

[33] B. Du Bois, S. Demeyer, J. Verelst, T. Mens, M. Temmerman, Does god class decomposition affect comprehensibility?, in: IASTED Conf. on Software Engineering, pp. 346–355.

[34] H. Aman, S. Amasaki, T. Sasaki, M. Kawahara, Empirical analysis of fault-proneness in methods by focusing on their comment lines, in: 2014 21st Asia-Pacific Software Engineering Conference, volume 2, pp. 51–56.

[35] H. Aman, An empirical analysis on fault-proneness of well-commented modules, in: 2012 Fourth International Workshop on Empirical Software Engineering in Practice, pp. 3–9.

[36] D. R. Cox, The regression analysis of binary sequences, Journal of the Royal Statistical Society. Series B (Methodological) 20 (1958) 215–242.

[37] L. Breiman, J. Friedman, C. Stone, R. Olshen, Classification and Regression Trees, The Wadsworth and Brooks-Cole statistics-probability series, Taylor and Francis, 1984.

[38] L. Breiman, Random forests, Machine Learning 45 (2001) 5–32.

[39] P. Geurts, D. Ernst, L. Wehenkel, Extremely randomized trees, Machine Learning 63 (2006) 3–42.

[40] L. Breiman, Bagging predictors, Machine Learning 24 (1996) 123–140.

[41] Y. Freund, R. E. Schapire, A decision-theoretic generalization of on-line learning and an application to boosting, Journal of Computer and System Sciences 55 (1997) 119 – 139.

[42] J. H. Friedman, Greedy function approximation: A gradient boosting machine., Ann. Statist. 29 (2001) 1189–1232.

[43] T. Chen, C. Guestrin, Xgboost: A scalable tree boosting system, in: Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 785–794.

[44] F. Zampetti, L. Ponzanelli, G. Bavota, A. Mocci, M. D. Penta, M. Lanza, How developers document pull requests with external references, in: 25th International Conference on Program Comprehension (ICPC), volume 00, pp. 23–33.

[45] X. Zhang, Y. Chen, Y. Gu, W. Zou, X. Xie, X. Jia, J. Xuan, How do multiple pull requests change the same code: A study of competing pull requests in github, in: IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 228–239.

[46] Y. Yu, H. Wang, V. Filkov, P. Devanbu, B. Vasilescu, Wait for it: Determinants of pull request evaluation latency on github, in: 12th Working Conference on Mining Software Repositories, pp. 367–371.

[47] W. Zou, J. Xuan, X. Xie, Z. Chen, B. Xu, How does code style inconsistency affect pull request integration? an exploratory study on 117 github projects, Empirical Software Engineering (2019).

[48] A. Rastogi, N. Nagappan, G. Gousios, A. van der Hoek, Relationship between geographical location and evaluation of developer contributions in github, in: 12th International Symposium on Empirical Software Engineering and Measurement, ESEM '18, pp. 22:1–22:8.

[49] J. Tsay, L. Dabbish, J. Herbsleb, Influence of social and technical factors for evaluating contribution in github, in: 36th International Conference on Software Engineering, ICSE 2014, pp. 356–366.

[50] D. M. Soares, M. L. de Lima Júnior, L. Murta, A. Plastino, Acceptance factors of pull requests in open-source projects, in: Proceedings of the 30th Annual ACM Symposium on Applied Computing, SAC '15, ACM, New York, NY, USA, 2015, pp. 1541–1546.

[51] V. J. Hellendoorn, P. T. Devanbu, A. Bacchelli, Will they like this? evaluating code contributions with language models, in: 12th Working Conference on Mining Software Repositories, pp. 157–167.

[52] P. C. Rigby, M. Storey, Understanding broadcast based peer review on open source software projects, in: 33rd International Conference on Software Engineering (ICSE), pp. 541–550.

[53] F. Zampetti, G. Bavota, G. Canfora, M. D. Penta, A study on the interplay between pull request review and continuous integration builds, in: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 38–48.

[54] J. Terrell, A. Kofink, J. Middleton, C. Rainear, E. R. Murphy-Hill, C. Parnin, J. Stallings, Gender differences and bias in open source: pull request acceptance of women versus men, PeerJ Computer Science 3 (2017) e111.

[55] L. Dabbish, C. Stuart, J. Tsay, J. Herbsleb, Social coding in github: Transparency and collaboration in an open software repository, in: Conference on Computer Supported Cooperative Work, CSCW 12, p. 12771286.

[56] M. M. Rahman, C. K. Roy, J. A. Collins, Correct: Code reviewer recommendation in github based on cross-project and technology experience, in: 38th International Conference on Software Engineering Companion (ICSE-C), pp. 222–231.

[57] G. Gousios, M.-A. Storey, A. Bacchelli, Work practices and challenges in pull-based development: The contributors perspective, in: 38th International Conference on Software Engineering, ICSE 16, p. 285296.

[58] A. Trautsch, S. Herbold, J. Grabowski, A longitudinal study of static analysis warning evolution and the effects of pmd on software quality in apache open source projects, in: arXiv 1912.02179.

[59] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Softw. Engg. 14 (2009) 131–164.

[60] V. R. Basili, G. Caldiera, H. D. Rombach, The goal question metric approach, Encyclopedia of Software Engineering (1994).

[61] M. Patton, Qualitative Evaluation and Research Methods, Sage, Newbury Park, 2002.

[62] M. Nagappan, T. Zimmermann, C. Bird, Diversity in software engineering research, in: Foundations of Software Engineering, ESEC/FSE 2013, pp. 466–476.

[63] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, D. Damian, An in-depth study of the promises and perils of mining github, Empirical Software Engineering 21 (2016) 2035–2071.

[64] T. M. Mitchell, Machine Learning, McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[65] D. Powers, Evaluation: From precision, recall and f-factor to roc, informedness, markedness & correlation, Mach. Learn. Technol. 2 (2008).

[66] A. P. Bradley, The use of the area under the roc curve in the evaluation of machine learning algorithms, Pattern Recognition 30 (1997) 1145 – 1159.

[67] D. Taibi, A. Janes, V. Lenarduzzi, How developers perceive smells in source code: A replicated study, Information and Software Technology 92 (2017) 223 – 235.

[68] N. Moha, Y. Gueheneuc, L. Duchien, A. Le Meur, Decor: A method for the specification and detection of code and design smells, IEEE Transactions on Software Engineering 36 (2010) 20–36.

[69] V. Lenarduzzi, A. Martini, D. Taibi, D. A. Tamburri, Towards surgically-precise technical debt estimation: Early results and research roadmap, in: International Workshop on Machine Learning Techniques for Software Quality Evaluation, MaLTeSQuE 2019, pp. 37–42.

[70] V. Lenarduzzi, F. Lomio, H. Huttunen, D. Taibi, Are sonarqube rules inducing bugs?, in: International Conference on Software Analysis, Evolution and Reengineering (SANER 2020), pp. 501–511.

[71] N. Saarimaki, M. T. Baldassarre, V. Lenarduzzi, S. Romano, On the accuracy of sonarqube technical debt remediation time, in: 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2019), pp. 317–324.

[72] N. Saarimäki, V. Lenarduzzi, D. Taibi, On the diffuseness of code technical debt in java projects of the apache ecosystem, in: Second International Conference on Technical Debt, TechDebt 19, IEEE Press, 2019, pp. 98–107.

[73] V. Lenarduzzi, N. Saarimaki, D. Taibi, 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.

[74] M. T. Baldassarre, V. Lenarduzzi, S. Romano, N. Saarimaki, On the diffuseness of technical debt items and accuracy of remediation time when using sonarqube, Information and Software Technology 128 (2020) 106377.

[75] G. Gousios, A. Zaidman, A dataset for pull-based development research, in: Working Conference on Mining Software Repositories, MSR 2014, p. 368371.

[76] V. Lenarduzzi, N. Saarimäki, D. Taibi, The technical debt dataset, in: Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE19, p. 211.

[77] N. Saarimäki, Methodological issues in observational studies, SIGSOFT Softw. Eng. Notes 44 (2019) 24.

[78] F. Calefato, F. Lanubile, B. Vasilescu, A large-scale, in-depth analysis of developers personalities in the apache ecosystem, Information and Software Technology 114 (2019) 1 – 20.