Validating and Prioritizing Quality Rules for Managing Technical Debt: An Industrial Case Study

Davide Falessi  
California Polytechnic State University  
San Luis Obispo, CA, USA  
dfalessi@calpoly.edu

Alexander Voegele  
Elsevier  
Frankfurt, Germany  
a.voegele@elsevier.com

Abstract — One major problem in using static analyzers to manage, monitor, control, and reason about technical debt is that industrial projects have a huge amount of technical debt which reflects hundreds of quality rule violations (e.g., high complex module or low comment density). Moreover the negative impact of violating quality rules (i.e., technical debt interest) may vary across rules or even across contexts. Thus, without a context-specific validation and prioritization of quality rules, developers cannot effectively manage technical debt. This paper reports on a case study aimed at exploring the interest associated with violating quality rules; i.e., we investigate if and which quality rules are important for software developers. Our empirical method consists of a survey and a quantitative analysis of the historical data of a CMMI Level 5 software company. The main result of the quantitative analysis is that classes violating several quality rules are five times more defect prone than classes not violating any rule. The main result of the survey is that some rules are perceived by developers as more important than others; however, there is no false positive (i.e., incorrect rule or null interest). These results pave the way to a better practical use of quality rules to manage technical debt and describe new research directions for building a scientific foundation to the technical debt metaphor.

Keywords — technical debt, quality rules, maintainability, defect proneness, case study.

I. INTRODUCTION

The development of software-intensive systems is still plagued by cost overruns, schedule slips, and high-profile failures. Moreover, it is well established that more than half of the total development effort [1], and up to 80% of the cost [2], in software projects is spent in the evolution phase. For these reasons, past research effort on software engineering focused on providing and assessing quality rules such as high comment density, low code complexity, and hundreds more [3] [4]. These research advances greatly improved the body of knowledge of how, in theory, the system should be built to facilitate software evolution. In practice, even assuming that this set of quality rules is correct and complete, development resources are limited due to business constraints. Hence, developers need to balance improvements in code quality (i.e., refactoring) with more tangible and short-term objectives, e.g., cost and time-to-market. Balancing these concerns is hard because they are invisible to decision makers, and do not have an immediate impact on functionality and the user. Thus, developers intentionally or unintentionally violate rules, and hence technical debt accumulates over time, making the system harder to maintain. Technical Debt is a new financial metaphor thatpowerfully captures the link between the intangible value of improving the internal quality of the system and its economic consequences in the evolution of the software system [1]. Similarly to value-based software engineering [2], the underlying idea is that refactoring can be seen as an investment decision, in that it incurs a short-term cost (e.g., refactoring effort) for pursuing a long-term benefit (e.g., fewer defects during maintenance).

Several static analyzers exist to manage technical debt [3], including open-source ones like SonarQube (http://www.sonarqube.org/). These tools can analyze the compliance of a software project to hundreds of quality rules including code smells (e.g., god classes) and coding style (e.g., intuitive variable naming). This high number of quality rules allows a good coverage of quality aspects. However, the underlying assumption is that a system should aim to meet all quality rules. This assumption holds in theory, but fails in practice. Specifically, real-life experience with software companies showed that even the companies certified with the highest software process maturity level agree that technical debt should be controlled rather than eliminated [4], [5], [6]. The rationale is that in some situations the effort required to improve the internal quality of the product can be better spent in other more valuable activities, such as reducing time-to-market. We know there is no silver bullet in software engineering [7]. In the same vein, not all rules have the same importance, and the importance can even vary across contexts. Thus, another major practical problem in using tools to manage technical debt is that quality rules are not validated and prioritized according to the application context. As a consequence of the two aforementioned problems, tools for managing technical debt can provide to the developers a huge number of violations [8]. Specifically, our real life experience showed that the effort required for fixing the quality rule violations can be higher than the cost of the entire project, even in the case of a healthy and successful project. In conclusion, without a context-specific validation and prioritization of quality rules, the developers cannot use tools to manage technical debt.

This paper reports on our experience in validating and prioritizing quality rules for managing technical debt. Our empirical method consists of a survey of six expert developers and the quantitative analysis of two data repositories (i.e., JIRA and SVN) of five .NET-based projects of a CMMI Level 5 software company. In order to support the external applicability
of results we report online the perceived importance of the 106 quality rules and the adopted survey. The contribution of the present paper is twofold:

1. Validating the impact of rule violations to defect proneness.
2. Reporting the importance (i.e. interest) as perceived by developers, of 106 different quality rules.
3. Supporting future replication of the present study by publishing the adopted survey, its results, and lessons learned on how to prioritize quality rules with experts.

The remainder of this paper is structured as follows. Section II describes the technical debt concept and the adopted terminology. Section III presents the design and the results of the study. Section IV discusses the outcomes of the study in the form of six lessons learned. Section V presents the threats to validity and Section VI concludes the paper by summarizing the study results and by presenting future work.

II. TERMINOLOGY

Technical debt is something that is hard to define [9]; therefore, in order to avoid confusion and misunderstandings, here we carefully define the key terms in the technical debt domain. Moreover, technical debt management is usually erroneously associated with applying refactoring activities [5] [9]. Here we define activities different from refactoring that still concern the management of technical debt.

Technical Debt is defined by Steve McConnell [10] as “A design or construction approach that's expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now (including increased cost over time).” We note that in an ideal world, featuring unlimited resources and knowledge of the code should be developed, projects would have the time to apply all quality rules or fix the violated ones. In practice, knowledge, time, and effort is very limited and tradeoffs must be made between short term benefits (e.g., customers satisfaction) and long term ones (e.g., maintainability). A further cause of technical debt is organic decay. Specifically, new solutions, technologies or frameworks became available over time that renders the adopted ones obsolete. For instance, a specific design could be good enough for the first releases of a project but it could make the system too slow once the number of users increases over time. In order to avoid confusion and misunderstandings, we carefully define key terms in the technical debt domain.

Quality rule: An empirically validated software engineering principle dictating how the code should be. Example of quality rules include: high comments frequency, low code complexity, intuitive variable naming, etc.

Quality rule violation: A piece of code not conforming to a quality rule. The same code can violate several rules and the same rule can be violated in several pieces of code. We highlight that quality rule violations relate to the internal quality of the software product and are invisible to the user of the software application. Vice versa, defects relate to the external quality of the code; sometimes they are called quality debts [11]. Thus, a quality rule violation is not a defect but it identifies a piece of code that can lead to a defect during maintenance. In order to better explain the concept of quality rules, violations, and defects we discuss an open source example. Fig. 1 below shows an excerpt of rule violation as reported by SonarQube for line 56 of the “SSLAuthenticator.java” class of the Apache Tomcat system, see online (http://goo.gl/9l4vmP) for the full SonarQube output. The quality rule being violated in the example is titled “Avoid commented-out lines of code”, which dictates that code should not be commented out but should be simply removed. The rationale of the rule is that commenting out code only has disadvantages and no advantages. Specifically, there is no advantage in commenting out code because all the old versions of the code are already available in the version control system (e.g., Git). The disadvantage of commenting out code is that comments may be removed by mistake causing that code to become unintentionally active and eventually produce undesirable system behaviors. In conclusion, the reported example clearly shows that the violation is a system property invisible to the user. As such, it is clearly different from a defect, although the violation can lead to a defect.

Static analyzer: A tool checking the compliance of the (usually) compiled code, with a set of rules. There are two main types of static analyzer tools currently adopted and evaluated in the software engineering research industry: Bug finders and quality analyzers. Bug finders search for defects rather than for quality rule violations. One of the most important example of a bug finder is FindBugs (http://findbugs.sourceforge.net/). Quality analyzers check the compliance of the code to good software engineering principles or programming styles, thus their warnings are quality rule violations rather than defects. Examples of quality analyzers are SonarQube and ReSharper (http://www.jetbrains.com/resharper/).

Refactoring: As introduced by Fowler et al. [12], refactoring is the activity to make the code more understandable and changeable by improving its design. However, there is still a debate about the current meaning of the term [13]. In this work, we refer to refactoring as the activity of fixing a violation, regardless of the specific type of quality rule being violated, e.g., reducing design complexity by re-designing god classes or improving the quality of the coding style by adding comments. Static analyzers support refactoring by suggesting what can be improved in the code. A further type of automated support for refactoring refers to tools suggesting how to refactor or even automatically fix some problems, this type of tool is out of the scope of this study.

Principal: The cost required to fix quality rules violations. It is commonly measured or estimated as the effort for refactoring the code that violates the quality rules [14].

Interest: The consequence of violating a rule, or of not fixing a violated rule. This is commonly measured or estimated...
as decreased productivity or extra defects [9]. In this work we see the level of interest of a quality rule as its level of priority; the rationale is that developers should focus on complying with rules based on the amount of negative effect their violations would have.

### III. STUDY DESIGN AND RESULTS

#### A. Aim, Research Questions, and Hypotheses

This paper reports on a case study about validating and prioritizing quality rules for managing technical debt. Given the terminology presented in Section II, we measure the importance of a rule with its interest, i.e., the consequences of violating, or not fixing a violated, rule. According to the Goal Question Metric [16], in this work we aim to analyze the quality rules, for the purpose of evaluation and characterization, with respect to maintainability (i.e., defect-proneness), from the point of view of the researcher in the context of a CMMILevel 5 organization called Keymind. More specifically, we aim to investigate the following research questions (RQ):

**RQ 1. Do quality rules have an interest? (i.e., Are quality rules important?)**

The available static analyzers make the assumption that having quality rule violations means having technical debt. This assumption clearly holds for the principal of the debt. As a matter of fact, quality rule violations require effort to be fixed, therefore violating quality rules entails having a principal. However, this assumption may not hold true for the interest on the debt principal, because having quality rule violations may not imply having a code that is hard to evolve. Here we conjecture that classes with higher violation density are more defect prone. Our null hypothesis is: \( H_0 = \text{When evolving classes with different violation density, there is no difference in the number of defects introduced per LOC touched.} \) Our alternative hypothesis is: \( H_1 = \text{When evolving classes with different violation densities, there is a higher number of defects introduced per LOC touched in classes with a higher violation frequency.} \)

**RQ 2. Does the interest differ across quality rules? (i.e., Are quality rules equally important?)**

RQ 1 investigates if quality rule violations have an interest, here in RQ 2 we investigate if the amount of interest changes across rules. The rationale is that with the high number of rules available, it is important to understand which of them are really important for avoiding defects and which others we can live with. Here we conjecture that rules have different levels of interest.

#### B. RQ 1. Do quality rules have an interest?

1) **Variables**

Given a specific class evolving in a specific release, our main dependent variable is defect proneness, called defect injection frequency (i.e., DInjFreq), which is defined as the number of defects injected in a class of a release divided by the LOC touched. LOC touched accounts only for commits where the related tickets are of the type “implementation” but not “defect fix”. This variable can be seen as the probability to introduce a defect when modifying a single LOC, regardless of the type of modification (add, delete, or change). We believe that focusing on defects is a strong indicator of the pain felt due to technical debt; higher defect proneness usually implies extra effort (required to fix the extra defects) and decreased reputation of the company or product. We used the defect injection frequency rather than simply number of defects because defects can be injected only if the class is touched, i.e., change, add, or delete LOC. In this work we do not differentiate between types and severity of defects and we do not take into account other negative consequences of violating quality rules other than a higher defect injection frequency. The main independent variable of this study is violations density (VDensity), which is defined, in a specific class of a specific release, as the number of violations divided by size (i.e., LOC). The main advantage of using violation density rather than violation number is to eliminate the effect of size, which is one of the most important influencing factors in maintainability [17] [18]. We also see violations as traps into the code. As such, the probability to have defects, i.e., failing into a trap, depends on the density of the traps as opposed to their number. We note that the use of violation density captures the concept of interaction effect among different quality rules, which resulted to be important during maintenance [19].

2) **Measurement approach**

Fig 2 reports the approach to measure the violation density and the defect injection frequency of a class in a specific release. Specifically, the number of violations and the size is computed at the start of a release whereas the LOC touched and defects are computed at its end.

![Diagram](image-url)

**Fig. 2. Measurement approach, for dependent and independent variables, for a class in a release.**

In order to measure violations of specific classes over different releases we developed a plugin for SonarQube called Technical Debt Analyzer. The plug-in was developed over a nine-month period, costing roughly 5 man months of work. The plug-in was developed using .NET technology and it is about 3000 LOC. The architecture of the plugin is reported in Fig. 3.

The Sonar reader component checks the compliance of quality rules over several releases of a project; this was a big improvement over SonarQube functionalities, as SonarQube can analyze only the current version of a project. The data analysis component merges information coming from different sources (e.g., Issue tracker system, version control system, and SonarQube) and uses open-source packages, like R and WEKA, to perform data analysis (e.g., computing the Spearman Rank correlation). Size and LOC touched has been measured by reading the SVN repository. Defects have been measured by analyzing the commit comments in SVN (to understand the specific class related to the defect) and the JIRA
repository (to understand that the change is related to a defect). For instance, a commit in SVN can have a comment like “PROJ1-546”. Our approach is to check the type of the ticket in JIRA having as identifier “PROJ1-546” (i.e., Implementation or Defect) and then update the LOC changed or number of defects counters of the specific class affected by the commit.

TABLE I reports the characteristics of the data analyzed in this case study in terms of percentiles of number of size (LOC), number of rule violations, number of (different) rules violated, LOC touched, number of defects, violation density and defects injection frequency. These details can be useful to understand the extent to which the adopted industrial dataset is comparable to other contexts and will help future studies to compare their results.

According to TABLE I, sizes of classes varies from 14 to 11709 LOC (see Rows 2 and 8, Column 2) and the average number of violations per class is 12 (see Row 9, Column 3). Moreover, the maximum number of different rule violated in a class is 18 and its median is 1 (see Rows 2 and 5, Column 4). We note that the data in column VDensity is different from the data in column Violations divided by the data in column Size. This because a row in TABLE I refers to a percentile of a specific characteristic of a class. In other words, the class with the highest number of violations is likely different from the one with the highest size or the one with the highest violation density. We analyzed the level of completeness of commit messages and we found that around 85% of commits messages are correctly linked to a single ticket in JIRA. This number should be interpreted by taking into consideration that artifacts different from code is managed via SVN, and hence the 85% represents the lower bound of the actual completeness of commit messages for code changes.

2) Main results
Fig. 4 and TABLE II reports the linear regression between the defect injection rate and the violation density in both a qualitative and statically way, respectively. The grey area of Fig. 4 reports the 95% confidence in the linear regression.
In order to investigate the level of interest of quality rules in a specific industrial context we performed a survey involving six developers in rating 106 rules. A default value of “Info” is assigned to all rules (1191) not yet violated. Before running the survey we checked the quality of the rule checker by making sure that the quality rules have a correct rationale, and that portions of the code highlighted as violations really violate the rule. We checked 10 violations for each of the 106 rules. Our results showed that all rules made sense and all violations were actually violating the rule. When designing the survey we wanted each rule to be rated by more than one expert. However, we realized that one expert does not have the time to rate all the rules. Thus, we designed the survey to reach the compromise that each rule is rated by two experts. Therefore, the 106 rules were divided in three different versions of the survey (35+35+36) and each version of the survey was assigned to two respondents. The survey is structured in four main sections:

1. Introduction: This section describes the aim and scope.
2. Background: This section contains one question about the number of years of development experience.
3. Rule ranking: This section describes the rules and asks for a level of interest. Specifically, each of the 35 or 36 rules was described by providing a link to a general description of the rule and a link to a portion of the code, of a Keymind project, violating such rule (Fig. 1). The question asked to developers in order to rate the level of interest of a quality rule is “How much does violating this rule influence the maintainability?” Possible mutually exclusive answers (i.e., levels of interest associated to a quality rule) are: “Very High”, “High”, “Low”, “Very Low”, “I didn’t understand the meaning of the rule”, and “I’m unsure about the level of interest of the rule”.
4. New rule: This section asks the respondents to provide general comments or a new rule.

The survey was implemented via LimeSurvey (https://www.limesurvey.org/) and was piloted twice before going live. The survey invitation was provided via email with a unique link to each respondent. This allowed us to monitor the status of completion of the survey and to send reminders to specific respondents. A pdf of one version of the survey is available online (http://goo.gl/3sM3yl). After the survey, all rules had two ratings, and each expert rated only 35 or 36 rules. Because we wanted to enforce agreement among experts in rule importance, we had two roundtable discussions with all the experts of around three hours each. During this roundtable discussion, all experts discussed together all rules and established their level of interest by reaching a complete agreement. The rating resulting from the survey was used only as a value to initiate the discussion and then it was thoroughly discussed and revised. We opted to use the expert opinion as the rating resulting from this roundtable discussion, rather than from the survey.

3) Raw results

The raw results of the survey are presented in TABLE III. As we can see, the survey acceptance rate was 100% and only 1% of the answers are “I don’t know”, thus suggesting developers

![Fig. 4: Linear regression between the violations density (VDensity) and the number of defects injected per LOC touched (DlnFreq).](image)

<table>
<thead>
<tr>
<th>Term</th>
<th>Estimate</th>
<th>Std Error</th>
<th>t Ratio</th>
<th>P-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDensity</td>
<td>0.383</td>
<td>0.097</td>
<td>3.96</td>
<td>&lt;.0001</td>
</tr>
</tbody>
</table>

TABLE II: LINEAR REGRESSION STATISTICAL DETAILS.

According to Fig. 4, classes with a low value of violation density are clearly less defect prone than classes with a high violation density; specifically the defect injection frequency varies from 3% to 20% with violation density. For example, by considering a class that evolved 100 LOC in a release, this class would in average have 3 defects in case of no violations (i.e., VDensity =0), or 20 defects in case of a violation density of 0.45. Thus, the number of defects introduced during maintenance is in average more than six times higher in classes with a very high violation density. According to TABLE II, defect injection rate and violation density are linearly correlated (i.e., low standard error) and this correlation is strong (i.e., statistically significant because P-value < 0.05). Thus, we can reject H0 and claim with a very high confidence that classes with low violation density have a significant lower number of defects per LOC touched.

C. RQ 2. Does the level of interest differ across quality rules?

1) Variables

The dependent variable in this research question is the level of interest associated to the quality rules and it is measured in a four-point ordinal scale: Very High, High, Low, Very Low. The rationale for using this scale is to allow easy mapping to the five-point scale used in SonarQube: Blocker, Critical, Major, Minor, Info. The last level of the SonarQube scale (i.e., Info) is assigned in our case study to all quality rules that are not yet violated. The independent variable in this research question is the quality rule being violated. Despite the fact that SonarQube supports the checking of several hundreds of quality rules (i.e., 1297 at the time of the survey), only 106 rules are currently violated in Keymind projects and considered in this study.

2) Measurement approach
know what matters in order to facilitate maintenance. It is interesting to note that 17% of the rules had been impossible to understand. By doing some root cause analysis we understood that this happened when respondents were unable to access the private network and hence unable to see the portion of the code violating this rule. Thus we learned that, in order to understand the rule, it is necessary to have not only its definition but also a real example of violation. Additional results include that the survey took around one hour to complete and no respondent proposed new rules. We have also analyzed the agreement among experts’ opinions by measuring the Spearman’s rank correlation, which resulted in a value of 0.0583, suggesting a low agreement level. After the survey, experts discussed the survey results, reaching a complete agreement about the level of interest.

### Main results

Fig. 5 describes the distributions of levels of interest across rules. As we can observe, more than a third of the rules have been rated with a “Very High” level of interest. Moreover, only 6% of the rules have a “Very Low” level of interest, suggesting that quality rules are usually perceived as a useful principle to facilitate maintenance. According to Fig. 5, the median (i.e., “High”) is different from the modal value, i.e., the most frequent value “Very High”. Moreover, given the distribution of levels of interest reported in Fig. 5, in average among rules, there is only a 31% chance (0.36*0.36 + 0.25*0.25 + 0.33*0.33 + 0.06*0.06) of picking two rules with the same level of interest. This led us to conclude that the difference of level of interest among rules is practically significant. Despite the fact that Fig. 5 does not have any practical use per se, it serves to highlight how the reported distribution of levels of interest is new and was unpredictable.

<table>
<thead>
<tr>
<th>Empty</th>
<th>Very High</th>
<th>High</th>
<th>Low</th>
<th>Very Low</th>
<th>don’t know</th>
<th>I didn’t understand the rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>0</td>
<td>14</td>
<td>70</td>
<td>7%</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>%</td>
<td>0%</td>
<td>5%</td>
<td>26%</td>
<td>4%</td>
<td>1%</td>
<td>14%</td>
</tr>
</tbody>
</table>

Fig. 5: Distribution of levels of interest across quality rules as perceived by experts developers.

Specifically, we didn’t know if the experts would rate all rules as “very high” or “very low”. Moreover, in order to enhance the usefulness of the achieved levels of interest, we report online (http://goo.gl/6HyDbK) the expert opinion of all the 106 rules. These values can be used as a starting point and can be refined in contexts that are different from Keymind. The case study results are reported in the form of lessons learned and are structured in three groups: main, unexpected, and methodological.

### IV. Discussion

#### A. Results Related to Research Questions

This section reports the lessons learned (i.e., the answers) related to the research questions we investigated.

**LL 1: Violations matter.** According to historical analysis, classes having a high density of quality rule violations are highly defect prone. Specifically, the number of defects introduced per LOC touched is on average more than five times higher in classes where many quality rules are violated. This clearly motivates practitioners in minimizing the number of violations and somehow validates the use of violations as a proxy to quantify technical debt.

**LL 2: Rules differ in interest.** Some rules are perceived by developers as more important than others. To enhance replicability of results, we report online the interest of each quality rule and the survey itself.

#### B. Further Results

In this section we report results that are not related to our research questions but useful for managing technical debt via quality rules.

**LL 3: No false positives.** Our case study reveals that the violations reported by SonarQube are never wrong. First of all, all portions of the code highlighted as violations actually violate a rule. Second, all quality rules are somehow relevant and a good principle of software engineering, even if some rules are more relevant than others. The absence of false positive is somehow in contrast to several papers in the literature [20] [21]. However, those papers did not investigate if what developers called false positives were due to a problem in the checking mechanism, the rule semantic (i.e., a software engineering principle that is not valid in a context), or the conditions upon which a rule is defined as violated (e.g., a specific complexity threshold may not apply to every contexts). Moreover, we note that our “Very Low” level of interest could be close to what in other studies is referred as false positive [20] [21]. The inconsistency of our results with previous ones clearly paves the way for future research efforts.

**LL 4: Experts’ engagement built trust.** SonarQube already provides a default prioritization of quality rules. However, despite using the default categorization, using a survey to develop a context specific categorization was a good way to build trust of developers on the rules adoption. A direct involvement of developers in the ranking process improved their confidence in tool instrumentation and use.

#### C. Methodological Results

Here we report the lessons we learned related to the survey which could be useful in performing similar surveys.

**LL 5: Examples of violation are required for understanding the rule meaning.** We observed that developers were unable to rate the level of interest of a quality rule by only looking at the description of the rule, without looking at a concrete example of rule violation. This means that the semantics of a quality rule must be described by reporting a violation example.
**LL 6: Rating interest via comparison.** The levels of interest of rules adopted in this case study (i.e., Very high, High, Low, Very Low) are not linked to any objective meaning; i.e., the Very high level of interest does not mean a specific amount of extra defects per LOC touched. Therefore, during the rule prioritization activity, developers had problems in assigning level of interest values. For instance, it was unclear when a quality rule should be rated as “Very High” or “High”. Something that helped us in achieving an agreement about the level of interest of rules was comparing unclassified rules with already classified ones. In other works it was easy to reach an agreement about the level of interest of a rule by finding one rule with the same level of interest among the already classified.

**V. THREATS TO VALIDITY**

**Measurement approach.** There is still controversy and open challenges on how to measure defects in classes of a specific release [22]. Our measurement approach is customized for a high maturity company where we are confident that defects are ticketed in an accurate way. However, we could suggest more sophisticated techniques, like the one proposed by Kim, et al. [22], when analyzing datasets of questionable reliability like open source projects.

**Results generalizability.** The presented results are related to one single company and therefore its applicability to external contexts can be questionable. However, in this study we don’t aim to provide the golden standard of rules interest because we believe there is no silver bullet in software engineering [7]. As a matter of fact, one of the reasons underlying this study is that rules interest can vary across contexts. In order to support the external applicability of results, we report online the achieved levels of interest of the 106 quality rules and the adopted survey. This information, together with the methodological lessons learned reported in Section IV.C, supports replication.

**VI. RELATED WORK**

Benefits of refactoring activities have been extensively studied. Ratzinger, et al. [25] found that refactoring related features and defects have an inverse correlation. Several studies investigated the developers’ opinions about refactorings and code smells. One of the main recent works by Kim, et al. [13] showed that developers at Microsoft perceive refactoring activities as risky, effort expensive, and in the need of tool support. Sjöberg, et al. [17] performed the first controlled experiment with professionals investigating the impact of code smells to effort in maintenance. Steidl and Eder [8] recently reported an industrial case study about an approach to suggest what to refactor. We share with them the vision that developers lack a structured prioritization mechanism to manage the thousands of quality improvement opportunities provided by static analyzers tool. The main difference between our work and theirs is that they focus on two quality rules (avoiding code clones and long methods) and on the reasons why their violations are not always fixed. Yamashita and Moonen [19] reported an empirical study on the effect of twelve code smells to maintenance problems. Their main result was finding that there is an important interaction effect among code smells. Their focus is on understanding how quality violations impact maintainability whereas we focus on which and how much. A lot of further research effort has been spent investigating and enhancing humans’ performances in refactorings activities [23] [24]. However, this paper relates to understanding of the what to refactor rather than the how.

Decision-making support focuses on identifying the code segment where refactoring will provide the highest benefit. Mens, et al. [26] designed a tool used to detect sections of code that need refactoring and decide which refactoring should be applied. They did so by detecting the existence of “bad smells”. Mens et al. experimentally demonstrated the use of their tool to detect three kinds of “bad smells” including “inappropriate interface”, “unused parameters” and “duplicated code.” Similarly, Zhao and Hayes describe a rank-based decision support for refactoring activities [27]. Zhao and Hayes’ results show an overlap of the classes in need of refactoring identified by the tool and by the experts. The work here shares with the work of both Mens, et al. and Zhao and Hayes the aim of prioritizing refactoring activities due to the high number of possible refactoring activities and the shortage of effort. However one major difference is that they ranked the pieces of code requiring refactoring according to code characteristics rather than the importance of the quality rules being violated in the code. In other words, they focus on prioritizing the “where” (i.e., which classes) whereas we focus on prioritizing the “what” (i.e., which violation). Another important difference is in the way they measured the importance of quality rules. Instead of using experts’ opinions they both used the number of past “refactoring activities”.

Several efforts have been spent in technical debt and quality rules validation and identification. Cai et al. [28] present a decision support system to modularity evaluation. We share with them the application of the technical debt metaphor to support the reasoning of what to refactor. However, a major difference between their work and ours is that they focus on one specific aspect of quality whereas we take into account several hundreds of quality indicators. Moreover, they use extra effort to measure the impact of rule violations whereas we use experts’ opinions. Zazworka, et al. [29] asked developers to spot and rate the level of interest of the violation. Counter to that, here we assume the interest to be a property of the quality rule rather than of the specific violation. Palomba, et al. [30] recently investigated the perceived level of interest of twelve quality rules by surveying developers and students. Similar to the present study, they found that not all quality rules are equally important.

**VII. CONCLUSIONS**

The results of our case study show that classes having a high density of quality rule violations are highly defect-prone. Specifically, the number of defects introduced per LOC touched during maintenance is about five times higher in classes where many quality rules are violated. Moreover, results indicate that quality rules have different interest levels (i.e., some rules are more important than others). We also reported some methodological lessons learned including the high effectiveness of assigning levels of interest to rules via comparison rather than directly.

The presented results relate to one single company. However, one of the main reasons for conducting this study is that rule interests can vary across contexts.
Regarding future work, because quality rule violations resulted in a tangible impact to the error-proneness of a class, then it should be valuable to use the violation information for predicting the number of defects in a class and hence support testing activity. Therefore, we plan to compare the accuracy of defect prediction models in case they are fed with state-of-the-art variables versus fed also with information about violation density. A further research direction is to move the focus from correlation to causation. Specifically, despite this work showing a strong correlation between violation density and defect proneness, this work did not prove that that lowering the violation density implies lowering its defect proneness. A substantial basic research effort is needed to create a body of knowledge about causation rather than correlation.

ACKNOWLEDGEMENT
This work has been fully performed when both co-authors worked for the Fraunhofer Center for Experimental Software Engineering (CESE), USA.

The work of Davide Falessi has been partially supported by Cisco research grant CG# 583194.

We thank the several individuals who contributed to this research effort including Kathleen Mullen and Mark Stein at Keymind for their feedback during survey preparation and revision of this paper. We thank Michele Shaw at Fraunhofer CESE, USA for her feedback during tool ideation. We thank all Keymind developers who responded to the survey and participated in discussions.

REFERENCES


